Java 202 - Librairies & Frameworks

Loïc Ledoyen

Java 202

Introduction

Le succès de Java dans le monde de l'entreprise est lié à plusieurs caractéristiques :

- Simplicité
- Approche orienté objet
- Forte rétro-compatibilité
- Ouverture
- Rapidité d'exécution (après le temps de chauffe)

Ce qui est en fait un langage de choix aujourd'hui est aussi lié au riche écosystème qui s'est développé sur ces bases.

Dans cet écosystème, on retrouve notamment des outils :

- Construction de projet (Maven, Gradle, etc.)
- IDE (IntelliJ, Eclipse, VSCode, etc.)
- Profiling (JProfiler, Yourkit, etc.)
- Tracing / APM (Jaeger, Datadog, New Relic, AppDynamics, etc.)

Mais également un large choix de frameworks et libraries :

- DI et intégrations (Spring, Quarkus, Micronaut, etc.)
- ORM (Hibernate, jOOQ, etc.)
- Logs (SLF4J, Logback, Log4J, JUL, etc.)
- Tests (JUnit, AssertJ, Mockito, Testcontainers etc.)

Considérant tous ces frameworks et outils à disposition, la qualité logicielle vient de

- Comment sont comprises et agencées ces briques existantes
- Comment elles sont configurées pour permettre une investigation à postériori (troubleshooting)
- Quels indicateurs sont suivis en production

1. Rappels sur les outils

1.1. Git

Git est un SCM (**S**ource **C**ode **M**anagement tool) décentralisé.

On étend par décentralisé le fait qu'il peut y avoir plusieurs instances d'un même dépôt sur des serveurs différents.
Typiquement, dans le monde de l'open-source, quand un individu externe à l'organisation souhaite contribuer au code d'un dépôt, il en fait une copie, travaille sur sa copie, puis propose le code de sa copie pour intégration sur le dépôt *officiel*.

Git est aujourd'hui très répandu, mais fait suite historiquement à d'autres SCMs (CVS, SVN, Mercurial, etc.).
Tous ces outils fonctionnent par différentiel (patch) pour permettre de restaurer une version précédente, ou encore de travailler sur une version parallèle qui pourra plus tard être réincorporée dans la version principale.

Voici le vocabulaire consacré :

- **commit** : une révision / version contenant des modifications de code
- **branch** : un fil de modification, une suite de révisions
- **tag** : alias pour une version spécifique, souvent utilisé pour marquer une version applicative (1.0.25 par exemple)
- **merge** : fusion d'une branche dans une autre
- **checkout** : récupérer le code d'un serveur distant dans une version spécifique

1.1.1. Quelques commandes utiles

- Initialiser un dépôt
  - `git clone <url>` : copie un dépôt distant existant en local
  - Ou `git init` : transforme le dossier courant en dépôt local. Un dépôt distant pourra être

indiqué par la suite avec `git remote add origin <url>`

- Mettre à jour
  - `git fetch --all --prune` : récupère les changements du dépôt distant
  - `git pull` : fusionne les changements distants avec les fichiers locaux
  - `git rebase origin/<current-branch>` : déplace les commits locaux après ceux ayant été poussés sur le dépôt distant (et sur la même branche)
- Changer de branche
  - `git checkout <branch>` : positionne les sources courantes sur la dernière version de `<branch>`
  - `git branch -b <branch>` : crée une branche de nom `<branch>` dont le point de départ est le commit courant
- Observer les changements
  - `git status` : affiche les différences entre les dépôts local et distant
  - `git log --oneline -n 15` : affiche les 15 derniers commits de la branche courante (avec leurs hash)
  - `git diff --stat` : affiche un résumé des changements
  - `git diff --word-diff=color <file>` : affiche les changements effectués sur le fichier `<file>`
- Apporter des changements
  - `git add <file>` : ajoute le fichier `<file>` à l'*index*
  - `git add .` : ajoute tous les fichiers modifiés à l'*index* (traverse les répertoires)
  - `git reset <file>` : enlève le fichier `<file>` de l'*index*
  - `git commit -m "<title>"` : crée un commit avec toutes les modifications dans l'*index* avec le titre `<title>`
  - `git commit --fixup <hash>` : crée un commit de correction d'un commit existant de hash `<hash>` avec toutes les modifications dans l'*index*
  - `git rebase -i --autosquash <hash>` : initie un rebase interactif et déplace et marque les commits de correction pour les fusionner, jusqu'au commit de hash `<hash>` exclu

Source : https://git-scm.com/docs

1.1.2. Pour les utilisateurs de Windows

Git est sensible au bit d'execution des fichiers (`chmod +x`).
Windows ne gérant pas de la même façon les permissions sur les fichiers qu'Unix, il est recommandé de désactiver cette sensibilité avec `git config core.fileMode false`
Pour expliciter le fait qu'un fichier soit executable : `git update-index --chmod=+x <file>`

1.2. Concepts utilisés par Git

On appelle **remotes** les serveurs distants configurés sur une copie locale.
Le remote par défaut est appelé **origin**.

Dans le cas où le dépôt a été cloné (et non initialisé) **origin** pointe sur l'url utilisée lors du *clone*.

Git utilise une base de données (répertoire **.git**) qui contient l'arbre de toutes les modifications de chaque branche.
Celle-ci contient également la version des différents remotes.
Les branches en question sont accessibles avec le nom : <**remote_name**>/<**branch_name**>.
Par exemple **origin/main** est la branche **main** telle que le serveur **origin** la connaissait lors de la dernière synchronisation de la base de donnée.
Il est tout à fait possible d'avoir une (et une seule) version locale et plusieurs versions distantes d'une même branche différente.
Ce sera lors d'un push (envoi de l'historique local vers un remote) que ces versions deviendront les mêmes.

La copie de travail (**working copy**) sont les fichiers contenus dans un dépôt local.
Il est possible de les modifier, d'en ajouter ou d'en supprimer, sans modifier les versions connues par Git.
Il sera, dans tous les cas, possible de revenir à une version connue par Git, grâce à la base de données.

Afin de considérer les modifications opérées sur la copie de travail pour être historisées (embarquées dans un *commit*), il est nécessaire de les indexer.
L'**index** est l'espace accueillant les modifications qui seront comprises dans un commit.
Il est possible d'y ajouter des éléments (ajout, suppression ou modification de fichier) avec la commande `add`.
Y enlever des éléments se fait avec la commande `reset`.
Enfin la commande `status` fait apparaître dans des couleurs différentes les changements qui sont indexés et ceux qui ne le sont pas.
Par défaut les changements indexés sont en vert et les autres en rouge.
Réaliser un *commit* (commande `commit`), embarquera toutes les modifications vertes.

1.2.1. Rebase

Une des fonctionnalités qui démarque Git de ses prédécesseurs est le **rebase**.

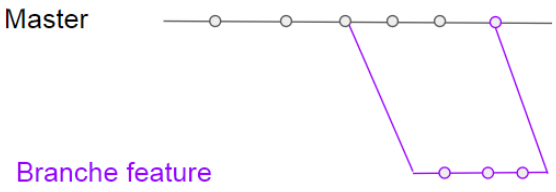
Le **rebase** peut être utilisé pour remettre à jour une branche quand la branche d'origine a changé.

```
git fetch --all --prune ①
git log --one-line -n 10 ②
git rebase origin/main ③
```

① Récupère la base de donnée du remote par défaut (**origin**) pour toutes les branches

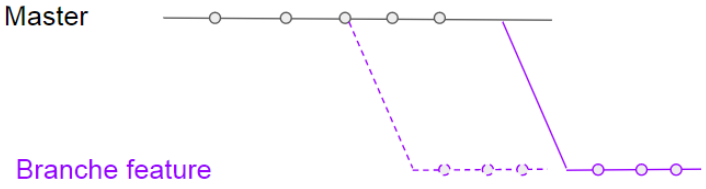
② Affiche les dix derniers commits de la branche courante

③ Modifie l'historique de la branche courante en mettant les commits réalisés après la base à la suite des derniers commits de la branche `main` telle qu'elle est connue par **origin**



Merge :

- résolution des conflits au dernier moment
- commit de merge supplémentaire



Rebase :

- résolution des conflits au fil de l'eau
- merge optionnel (quand la branche est en avance)

Le **rebase** peut également être utilisé en mode ***interactif*** pour modifier son historique local :

- Ajouter des modifications dans un commit
- Changer le nom d'un commit
- Fusionner des commits
- Supprimer des commits
- Ré-ordonner des commits

CONSEIL	Ne pas utiliser le rebase sur une branche partagée par plusieurs développeurs, et encore moins `main`

1.3. Maven

Maven est un outil de construction de projet (Build Automation tool) autour de la JVM.

Il permet entre autre d'orchestrer :

- Gestion des dépendances
- Compilation des sources
- Lancement des tests
- Génération de la documentation
- Assemblage des binaires

Sa grande extensibilité lui permet de s'adapter à différents languages (Java, Scala, Kotlin, etc.) et à différents scénarios (intégration continue, génération de code, déploiement, etc.).

1.3.1. Structure d'un projet

Maven propose de baser l'organisation d'un projet sur des conventions (nommage, structure des

répertoires, etc.) plutôt que sur de la configuration pure comme ses prédécesseurs (Make, Ant, etc.).

Cette structure est composée de

- Un fichier pom.xml qui contient toutes les informations nécessaires à Maven pour construire le projet. Sa structure minimale est la suivante

Listing 1. Fichier pom.xml

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId> ①
    <artifactId>my-app</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <properties> ②
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

① Le triplet `groupId`, `artifactId` et `version` sont les coordonnées qui identifient un projet Maven et permettent des dépendances avec d'autres

② Section optionnelle, permettant de fixer l'encodage et la version de Java utilisée pour éviter des conflits par la suite

- Un répertoire **src** qui contiendra tous les fichiers que l'on souhaite conserver dans le SCM
  - Dans **src** on retrouve deux répertoires : **main** et **test** qui contiennent respectivement le code de production, et le code de test (code qui ne sera pas inclus dans les binaires produits lors de la phase de ***packaging***)
    - Dans ces deux répertoires, on trouve un répertoire du nom du langage utilisé, dans cet exemple, **java**
      - Enfin dans ce**s** répertoire**s java** (ou **groovy**, etc.), on retrouve le code. Ce code est organisé en packages, eux-mêmes étant constitués de répertoires

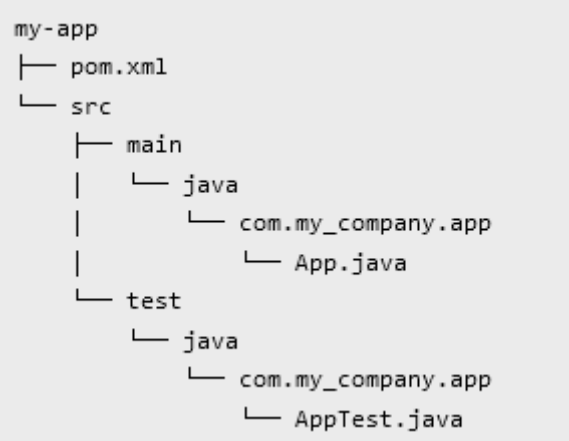


Figure 1. Structure d'un projet Maven

1.3.2. Cycle de vie d'un projet Maven

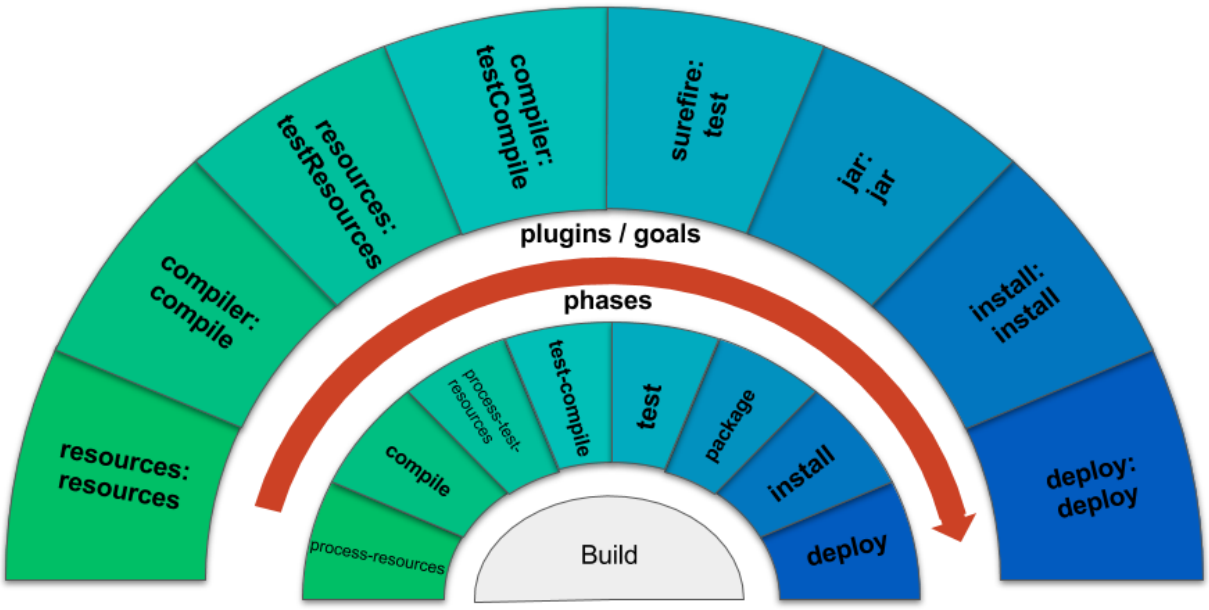
Par défaut Maven utilise un cycle de vie permettant à la grande majorité des projets d'être construit avec peu de configuration.

Les principales **phases** sont :

- **clean** : nettoie les fichiers compilés ou générés
- **compile** : compile les sources *principales (main)*
- **test-compile** : compile les sources de *test*
- **test** : lance les tests
- **package** : construit le binaire (**jar** par défaut)
- **install** : place le binaire dans le dépôt Maven local
- **deploy** : place le binaire dans un dépôt Maven distant
- **site** : génère la documentation

Chaque phase est associable à un ou plusieurs **plugins**, ce qui rend Maven très extensible.

Voici les associations par défaut :



Des plugins sont fournis directement par Maven, comme le **maven-clean-plugin**, qui supprime les fichiers compilés et générés.

D'autres sont créés par la communauté sans voir besoin de modifier l'outil. Par exemple :

- **cukedoctor-maven-plugin** : produit une version HTML du résultat des tests Cucumber
- **sonar-maven-plugin** : analyse le code avec différents outils (PMD, Checkstyle, JaCoCo, etc.) et publie les résultats vers un serveur Sonar

Source : https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

1.3.3. Balises de configuration

Toutes les balises doivent être contenues dans le bloc `<project>`.

Dans les balises notables, on retrouve :

- `properties` : cette balise contient des propriétés (clé et valeur) qui peuvent être utilisées par la suite, soit par convention par les plugins, soit explicitement avec l'écriture `${my-property}`

```
<properties>
    <my-test-lib.version>1.2</my-test-lib.version>
</properties>
```

- `dependencies` : cette balise contient toutes les dépendances d'un projet sur d'autres (internes, externes, frameworks, bibliothèques, etc.)

```
<dependencies>
    <dependency> ①
        <groupId>com.mycompany</groupId>
        <artifactId>my-lib</artifactId>
        <version>1.45.3</version>
    </dependency>
    <dependency>
        <groupId>com.mycompany</groupId>
        <artifactId>my-test-lib</artifactId>
        <version>${my-test-lib.version}</version> ②
        <scope>test</scope> ③
    </dependency>
</dependencies>
```

① Le bloc `dependencies` est constitué de blocs `dependency` (singulier), chacun contenant les coordonnées d'une dépendance

② La valeur de la version fait référence à la propriété `my-test-lib.version`, donc `1.2`

③ Ce second bloc est indiqué avec le ***scope*** `test`, cette dépendance ne sera donc disponible que pour le code de test

- `build/plugins` : cette balise contient tous les plugins utilisés par le projet ainsi que leurs configurations

```
<build>
    <plugins>
        <plugin> ①
            <groupId>org.apache.maven.plugins</groupId> ②
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
            <configuration> ③
                <failIfNoTests>true</failIfNoTests>
            </configuration>
        </plugin>
    </plugins>
</build>
```

① À l'instar de la balise `dependencies`, la balise `plugins` contient des blocs de `plugin` (singulier)

② Ici c'est le plugin `maven-surefire-plugin` qui est utilisé. Il s'agit du plugin par défaut pour lancer les tests. Un plugin est un projet Maven, et à ce titre est référencé grâce à ses coordonnées (`groupId`, `artifactId` et `version`) comme pour les dépendances

③ La balise configuration permet de modifier le comportement du plugin, ici le plugin va faire échouer le build si aucun test n'est trouvé

- `profiles` : cette balise permet d'ajouter des pans de configuration qui sont désactivables. Un profil peut ajouter des `properties`, des `dependencies`, des `plugins` et même des `modules` (utilisés pour les projets multi-modules)

```
<profiles>
    <profile>
        <id>disable-tests</id> ①
        <properties>
            <maven.test.skip>true</maven.test.skip>
        </properties>
    </profile>
</profiles>
```

① Balise obligatoire, un profil doit avoir un `id`, ce qui permet de l'activer en ligne de commande, par exemple : `mvn install -P disable-tests`

1.4. JUnit

Java ne propose pas dans le JDK d'outils pour construire et exécuter des tests.

Maven propose un répertoire de source, des phases (compilation & exécution) et un scope pour gérer ce code qui n'est pas destiné à la production.
Cependant, Maven ne fourni pas directement d'outil pour déclarer ou exécuter ces tests.

C'est là qu'entrent en jeu les frameworks de tests.
Il en existe plusieurs, et JUnit est aujourd'hui le plus répandu.

1.4.1. Utilisation de l'API de JUnit-Jupiter

```java
class CalculatorTest {

    private final Calculator calculator = new Calculator();

    @Test ①
    void simple_division() {
        int result = calculator.divide(8).by(2); ②

        Assertions.assertThat(result) ③
            .as("division of 8 by 2")
            .isEqualTo(4); ④
    }

    @Test
    void division_by_zero_should_throw() {
        Assertions.assertThatExceptionOfType(IllegalArgumentException.class) ⑤
            .isThrownBy(() -> calculator.divide(3).by(0)) ⑥
            .withMessage("Cannot divide by zero"); ⑦
    }

    @ParameterizedTest ⑧
    @CsvSource({
        "0, 3, 3",
        "3, 4, 7"
    }) ⑨
    void addition_cases(int a, int b, int expectedResult) { ⑩
        int result = calculator.add(a).and(b);

        Assertions.assertThat(result) ③
            .as("addition of " + a + " and " + b)
            .isEqualTo(expectedResult);
    }
}
```

① Méthode identifiée comme un test car marquée avec l'annotation `org.junit.jupiter.api.Test`

② Élément déclencheur, du code de production est exécuté

③ On vérifie le résultat du code de production (ici avec la bibliothèque **AssertJ**)

④ Ces trois lignes forme une seule expression, le compilateur ne tenant pas compte des sauts de ligne. Ce genre d'écriture est appelé **fluent interface** et repose sur des appels consécutifs de méthodes de sorte à former des phrases. Ici littéralement : vérifie que la variable `result` en tant que "division of 8 by 2" est égal à 4

⑤ Type différent de vérification, ici on vérifie qu'une erreur est produite, le test sera non passant si aucune erreur n'est produite ou si le type de l'erreur est différent de celui indiqué

⑥ Une fonction est passée à l'API de vérification, elle sera exécutée par la bibliothèque, dans un bloc `try / catch`

⑦ Vérification du message de l'erreur, si le message ne correspond pas, le test sera non passant

⑧ Méthode identifiée comme un test paramétré, elle sera exécutée autant de fois qu'il y a de jeux de données. La méthode dans cet exemple sera exécutée 2 fois.

⑨ Le jeu de donnée, ici passé comme un CSV (Comma Separated Values), d'autres sources de données sont possibles.

⑩ La méthode prend donc des paramètres dont le nombre correspond aux données dans les jeux de données. L'ordre des paramètres doit correspondre à l'ordre des données.

1.4.2. Comment JUnit fonctionne avec Maven

JUnit-Jupiter défini plusieurs choses :

- Une API pour déclarer une méthode comme étant un test (`@Test`, etc.)
- Un moteur d'exécution qui sait détecter les tests et les lancer

JUnit fourni également un lanceur de moteur(s) d'exécution : **junit-platform-launcher**

Enfin, le plugin **maven-surefire-plugin** "sait" se connecter (entre autres) à ce *launcher* (depuis la version 2.22.0).

NOTE

Pour résumer :

- La phase **test** de Maven est associé au plugin **maven-surefire-plugin**
- Ce plugin peut lancer **junit-platform-launcher** (si cette bibliothèque est présente sur le classpath)
- Ce *launcher* peut lancer les moteurs d'exécutions construits avec l'API de moteur d'exécution **junit-platform-engine**, notamment **JUnit-Jupiter**
- **JUnit-Jupiter** sait détecter et lancer les tests déclarés avec son API

1.4.3. Un peu d'histoire

JUnit est un vieux framework (1997) et celui-ci a beaucoup évolué au fil des versions de Java.
La version 4, arrivée en 2006 (après Java 1.5) a longtemps été utilisée, du fait de la simplicité d'écriture apportée par le support des annotations (`@Test`).
En 2015, une campagne de financement participatif est lancée pour créer JUnit**5**, une réécriture totale du framework.

Le constat de l'équipe est que le côté monolithique qui a jusque-là prévalu, a amené des dérives dans l'API du framework, qui est à la fois permissive et très complexe.

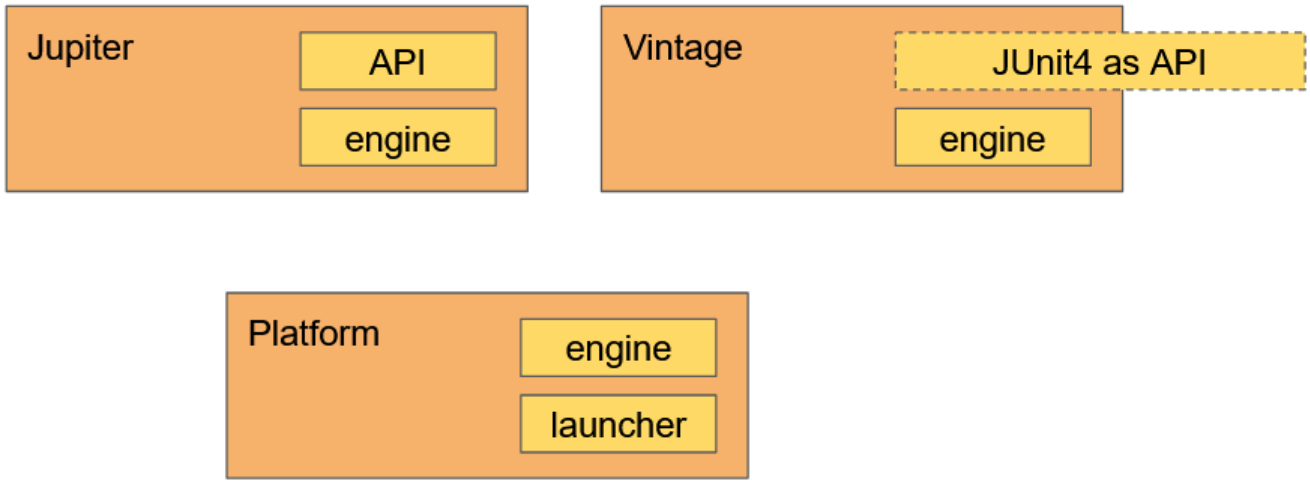
En effet, si le point de départ d'un test est conventionnellement une méthode, des plugins voient le jour pour changer ce paradigme (Cucumber, etc.), où un test peut-être un paragraphe dans un fichier texte.

Par ailleurs, même s'il existe plusieurs points d'extension dans cette version 4, le plus utilisé est le `Runner`, qui a le défaut de ne pas être composable.
Cela demande aux équipes fournissant des plugins de fournir des outils qui fonctionnent depuis plusieurs points d'extensions (`Runner`, `Rule`, initialisation dans une méthode de `setUp`, etc.) pour

contourner ce défaut

L'équipe de JUnit5 estime que chaque paradigme devrait avoir sa propre API et son propre moteur d'exécution, pour que le code soit plus spécifique, et donc plus simple.

L'architecture résultante est la suivante :



JUnit-platform est un framework pour construire (et lancer) des moteurs d'exécutions.

JUnit-Vintage est un moteur d'exécution qui est compatible avec l'API de JUnit4.

JUnit-Jupiter est le moteur d'exécution d'une nouvelle API qui profite (entre autres) de points d'extension multiples et composables.
:imagesdir: ./images

2. Fonctionnement des frameworks

Le principe des frameworks est de proposer des fonctionnalités sur un code arbitraire en l'analysant.

Ils peuvent aussi bien se baser sur

- Des conventions implicites, comme l'existence sur un objet d'un constructeur avec des paramètres non-ambigües
- Une déclaration explicite en utilisant l'API fournie par ceux-ci (annotations, interfaces, fichiers de configuration)

Cette analyse d'un code non connu à l'avance peut se réaliser soit

- Au moment de la compilation, dans ce cas la phase de compilation est plus longue
- Au démarrage de l'application, dans ce cas le démarrage de l'application est plus long

2.1. Compile-time

La brique de base pour générer du code au moment de la compilation est APT (Annotation Processing Tool), une API pour réaliser des plugins de compilation apparue en Java 6.
Celle-ci permet d'afficher des erreurs au moment de la compilation, mais aussi de générer du code.

Par exemple record-builder est un plugin de compilation qui permet de générer des classes "builder" des `record` annotés avec `@RecordBuilder`.

APT a cependant plusieurs contraintes, notamment le fait

- De ne travailler que sur du code source (donc pas sur des dépendances)
- Que le déclenchement des plugins se fait uniquement sur la base d'annotations.

Ainsi, il est impossible avec APT de déclencher un comportement sur un code qui n'est pas annoté ou qui est déjà compilé.

Il existe néanmoins d'autres solutions, comme le framework **Quarkus** qui a choisi d'ajouter une phase de transformation (**Augment**) *après* la construction des binaires par l'intermédiaire d'un plugin Maven ou Gradle.

L'intérêt de cette stratégie *compile-time* est d'avoir un binaire

- Potentiellement plus petit (le code des plugins de compilation n'y est pas présent)
- Potentiellement moins gourmand en mémoire, si le but est de remplacer l'AOP runtime (Aspect Oriented Programming)
- Plus rapide au démarrage

Dans le cas de **Quarkus** et **Micronaut**, l'intérêt est également de produire un binaire pouvant être (re-)compilé vers du code machine beaucoup plus rapide et petit, mais ne profitant pas du JIT (Just-In-Time Compiler) grâce au compilateur `native-image` AOT (Ahead-Of-Time) de GraalVM.

2.1.1. Anatomie d'un plugin de compilation

Un plugin de compilation Java est un service au sens de la **Service Provider Interface**.

Il faut donc déclarer les classes implémentant l'interface `javax.annotation.processing.Processor` dans le fichier `META-INF/services/javax.annotation.processing.Processor`.

Nous allons voir un cas très simple d'un plugin affichant une erreur de compilation si un objet annoté avec `@Sether` contient des méthodes *setters*, au sens de la spécification Java Beans.

Dans notre cas, le fichier `META-INF/services/javax.annotation.processing.Processor` ne contiendra qu'une ligne :

```
fr.lernejo.sether.SetherProcessor
```

Considérant l'annotation suivante :

```
@Target(ElementType.TYPE) ①
@Retention(RetentionPolicy.SOURCE) ②
public @interface Sether {
}
```

① Cette annotation ne peut être placée que sur un type (Class, Interface, etc.)

② Cette annotation ne sera pas gardée à la compilation, elle ne pourra pas être découverte au *runtime*

Voici le code du `Processor` :

```
@SupportedSourceVersion(SourceVersion.RELEASE_17) ①
@SupportedAnnotationTypes({
    "fr.lernejo.sether.Sether" ②
})
public class SetherProcessor extends AbstractProcessor {

    private Messager messager; ③

    @Override
    public synchronized void init(ProcessingEnvironment env) {
        messager = env.getMessager();
    }

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
        for (TypeElement annotation : annotations) { ④
            Set<? extends Element> annotatedElements = roundEnv
.getElementsAnnotatedWith(annotation); ⑤
            for (Element e : annotatedElements) {
                for (Element ee : e.getEnclosedElements()) { ⑥
                    if (ee.getKind() == ElementKind.METHOD && ee.getSimpleName
().toString().startsWith("set")) {
                        messager.printMessage(Diagnostic.Kind.ERROR, "Setters are
evil", ee); ⑦
                    }
                }
            }
        }
        return true;
    }
}
```

① Explicite que ce plugin est compatible jusqu'à la version 17 de Java

② Le plugin sera déclenché quand l'annotation `@Sether` sera présente sur un élément

③ Cet objet dont une référence est récupérée à l'initialisation permet d'envoyer des messages au compilateur. La compilation retournera un code d'erreur si au moins un message de type `ERROR` est envoyé.

④ Boucle sur les annotations pouvant déclencher le plugin, ici une fois.

⑤ Récupération de tous les éléments annotés avec `@Sether`, ici un ensemble de classe

⑥ Boucle sur les éléments contenus dans une classe : champs, méthodes, etc.

⑦ Envoi du message d'erreur au compilateur, associé à l'élément `ee` posant problème, ainsi le compilateur pourra afficher des éléments de contexte comme le fichier incriminé et la ligne de l'erreur

Un tel plugin ajouté comme dépendance `provided`

```
<dependency>
    <groupId>fr.lernejo</groupId>
    <artifactId>sether</artifactId>
    <version>${sether.version}</version>
    <scope>provided</scope>
</dependency>
```

Provoquera des erreurs de compilation si les conditions sont réunies :

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-
plugin:3.1:compile (default-compile) on project sample: Compilation failure:
Compilation failure:
[ERROR] ~/workspace/sample/src/main/java/fr/lernejo/sample/Machin.java:[12,17] Setters
are evil
[ERROR] ~/workspace/sample/src/main/java/fr/lernejo/sample/Machin.java:[16,17] Setters
are evil
```

De manière similaire à `Messager`, `Filer` permet de créer des fichiers qui seront par la suite compilés.

Écrire du code source *à la main* peut être verbeux et compliqué, c'est pourquoi il est préférable d'utiliser des bibliothèques pour manipuler ces concepts à plus haut niveau, comme ce que propose JavaPoet.

2.2. Runtime

Au *runtime*, c'est-à-dire après que l'application ait démarrée, l'analyse du code se fait par :

- *Class scanning*, en regardant chaque élément du *classpath* (archive jar, répertoire) et en allant *lire* le *bytecode* qu'ils contiennent
- Les APIs `java.lang.reflect` ou `java.lang.invoke`, permettant sur un objet `Class` arbitraire de récupérer et utiliser ses champs, méthodes, constructeurs, etc.

Les frameworks fortement modulaires comme **Spring** ont historiquement choisi cette approche.

Cela permet au développeur de configurer dynamiquement son application, et d'y ajouter des fonctionnalités par l'intermédiaire des nombreux points d'extensions fournis.

Cela permet également aux frameworks comme **Spring-Boot** de déclencher des comportements en fonction des éléments présents dans le *classpath*.

Par exemple, si **Spring-Boot** détecte la bibliothèque **spring-rabbit**, le framework va configurer un certain nombre de *beans* permettant notamment d'envoyer des messages au broker RabbitMQ.
Ces beans seront ensuite disponibles pour injection dans le code applicatif.

Par exemple le code ci-dessous permet l'envoi d'un tel message :

```
@SpringBootApplication
public class Launcher {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(Launcher.class); ①
        RabbitTemplate template = context.getBean(RabbitTemplate.class); ②

        template.convertAndSend("", "hello_queue", "Hello Rabbit !");
    }
}
```

① Démarre le contexte de Spring, entrainant le *class scanning* et la création des objets déclarés comme beans grâce à l'API du framework

② L'objet de type `RabbitTemplate` récupéré ici a été initialisé par le framework et communique par défaut avec l'instance locale sur le port par défaut `localhost:5672`. Pour pointer sur une instance différente, il suffit de re-définir la propriété `spring.rabbitmq.addresses`.

3. IoC, DI & AOP

3.1. IoC - Inversion de Contrôle

L'inversion de contrôle est le fait de déléguer l'ordonnancement d'une application à un *framework*.

L'idée générale est de créer de petites briques indépendantes et de laisser le *framework* les assembler et les appeler.

D'une certaine manière, ce n'est plus le code du développeur de l'application qui dicte la façon dont sont instanciées les classes et dont sont déclenchées les fonctionnalités.

En pratique, rien n'est magique, et les frameworks qui réalisent cette inversion de contrôle fonctionnent sur la base de conventions et de configurations, pour que le comportement de l'application soit déterministe et configurable.

3.2. DI - Injection de dépendance

L'injection de dépendance est un mécanisme qui permet à un framework de construire et d'initialiser des objets dépendants d'autres objets.

Il s'agit de décrire suffisament d'informations à propos des paramètres qui sont nécessaires au bon fonctionnement d'un objet et le framework pourra créer l'instance en question.

La manière la plus élégante est au travers d'un constructeur assignant des champs `private` et `final`.

Par exemple en utilisant Quarkus-Arc :

```
@QuarkusMain
public class Main {

    public static void main(String... args) {
        Quarkus.run(ExtensionApp.class, args); ①
    }
}
```

① À partir d'ici, on délègue au framework l'instanciation et l'exécution du code de l'application. C'est ici que le contrôle sur l'exécution s'inverse.

```
public class ExtensionApp implements QuarkusApplication {

    private final ExtensionService extensionService;

    public ExtensionApp(@RestClient ExtensionService extensionService) { ①
        this.extensionService = extensionService;
    }

    @Override
    public int run(String... args) {
        extensionService.getExtensionsById(args[0]).forEach(System.out::println); ②
        Quarkus.waitForExit();
        return 0;
    }
}
```

① Quarkus-Arc va créer une instance de cette classe et y *injecter* (comprendre, passer en paramètre), une instance de type `ExtensionService` avec le *qualifier* `@RestClient`.
Les *qualifiers* sont un moyen de lever l'ambiguité quand il existe plusieurs implémentations d'un même type, ou quand on veut expliciter le rôle d'un composant.
Ici on déclare explicitement que le rôle de l'objet injecté est d'être un client HTTP.

② Cette méthode est appelée par le *framework*.
Si une méthode d'un objet peut être appelée, c'est que cet objet a été instancié (et donc injecté).
Cette méthode ne fait qu'afficher dans la sortie standard le retour de la méthode `getExtensionsById`.

```
@RegisterRestClient(baseUri = "https://stage.code.quarkus.io/api") ①
public interface ExtensionService {

    @GET
    @Path("/extensions")
    Set<Extension> getExtensionsById(@QueryParam("id") String id);

    record Extension (
        String id,
        String name,
        String shortName,
        List<String> keywords
    ){}
}
```

① Annotation qui va déclencher la création d'un *proxy dynamique*, c'est-à-dire une implémentation de cette interface.
La création de ce proxy est gérée par la librairie **quarkus-rest-client**.
Il va intercepter les appels aux méthodes annotées (ou méta-annotées) avec `@HttpMethod` et se servir des autres annotations pour récupérer les informations nécessaires à créer les requêtes HTTP correspondantes.
Ici la requête sera `GET https://stage.code.quarkus.io/api/extensions?id={id}`

Le corps de la réponse sera ensuite *désérialisé* grâce à la librairie **quarkus-rest-client-jackson**.

3.3. AOP

La Programmation Orientée Aspect (AOP en anglais), est le fait de rajouter du comportement autour de méthodes existantes sans les modifier.

Le fait de rajouter ces comportements que ce soit à la compilation ou au démarrage de l'application est appelé tissage (weaving).

L'intérêt de l'AOP est de factoriser du code générique, comme :

- La collecte de statistiques
- La génération de log
- L'établissement d'une *transaction*
- Le re-essai en cas d'erreur
- Etc.

Exemple de proxy dynamique simple :

```
public class TimeLoggerProxy {
    private static final Logger logger = LoggerFactory.getLogger(TimeLoggerProxy.
class);

    public static <T> T create(T object, Class<T> objectInterface) {
        String interfaceName = objectInterface.getSimpleName();
        InvocationHandler invocationHandler = (proxy, method, args) -> {
            long startTime = System.currentTimeMillis();
            try {
                return method.invoke(object, args);
            } finally {
                long endTime = System.currentTimeMillis();
                logger.debug(interfaceName + "#" + method.getName() + " took " +
(endTime - startTime) + " ms");
            }
        };
        return (T) Proxy.newProxyInstance(TimeLoggerProxy.class.getClassLoader(), new
Class<?>[] { objectInterface }, invocationHandler);
    }
}
```

Ici une interface est nécessaire car la technologie de *proxy dynamique* fournie par la JDK ne fonctionne pas sur les classes.

Il existe cependant plusieurs librairies qui permettent de dépasser cette limitation (Byte Buddy, CGlib, Javassist, etc.).

Voici comment peut être décrit ce comportement avec Spring-AOP :

```
@Aspect
public class TimeLoggerAspect {

    private static final Logger logger = LoggerFactory.getLogger(TimeLoggerProxy.
class);

    @Around("@annotation(fr.lernejo.LogTime)") ①
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        long startTime = System.currentTimeMillis();
        Signature signature = pjp.getSignature();
        String className = signature.getDeclaringType().getSimpleName();
        String methodName = signature.getName();
        try {
            return pjp.proceed();
        } finally {
            long endTime = System.currentTimeMillis();
            logger.debug(className + "#" + methodName + " took " + (endTime -
startTime) + " ms");
        }
    }
}
```

① Cet aspect sera appliqué sur toutes les méthodes annotées avec `@LogTime`

4. Gestion des logs

Listing 2. Exemple de log peu utile

```
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.74
#################
2017-02-04 22:33:12 [thread-2] com.github.some.project.DatabaseService l.75 START
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.93 STOP in
17ms
2017-02-04 22:33:12 [thread-2] com.github.some.project.DatabaseService l.94
#################
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.124
java.sql.SQLIntegrityConstraintViolationException
at org.h2.message.DbException.getJdbcSQLException(DbException.java:345)
at org.h2.message.DbException.get(DbException.java:179)
at org.h2.message.DbException.get(DbException.java:155)
at org.h2.command.CommandContainer.update(CommandContainer.java:98)
at org.h2.command.Command.executeUpdate(Command.java:258)
at org.h2.jdbc.JdbcPreparedStatement.execute(JdbcPreparedStatement.java:201)
... 50 more
```

Listing 3. C'est déjà mieux

```
2017-02-04 22:33:12.332 INFO  5jhgd45ui74h c.g.s.p.UserService.create [Joshua] [Bloch]
Save successful
2017-02-04 22:33:12.758 INFO  6jyg45hgduyg c.g.s.p.UserService.create [Doug] [Lea]
Save successful
2017-02-04 22:33:12.964 ERROR hg457gehe4rt c.g.s.p.UserService.create [James]
[Gosling] Save KO: already exists
```

Les logs ou fichiers d'évènements sont ces *fichiers* qui contiennent les informations nécessaires au diagnostique en cas de problème.
Elles peuvent également être utilisées comme source de données pour créer des métriques de supervision.
Les logs sont souvent la seule trace de la chronologie des évènements et sont d'autant plus importantes dans les applications à fort traffic, où résoudre le problème d'un utilisateur peut s'apparenter à chercher une aiguille dans une botte de foin.

4.1. Bibliothèques de log

Côté applicatif, il existe de nombreuses bibliothèques pour écrire dans des fichiers de logs.

Rien que dans l'univers Java, on peut trouver

- JUL (**J**ava **U**til **L**ogging) minimaliste, fourni par le JDK
- Apache JULI (**J**ava **U**til **L**ogging **I**nterface)
- Apache Commons Logging

- Apache Log4J
- JBoss Logging
- Logback
- etc.

On retrouve dans ces différents outils des concepts communs :

- Découplage entre le composant qui collecte les logs (`Logger`) et celui qui écrit à l'extérieur du système (`Appender`)
- Configuration extérieure au code (fichier, propriétés système)
- Niveaux de sévérités
  - **DEBUG** : utilise pour le développement, invisible une fois l'application déployée en dehors d'un poste de développement
  - **INFO** : information sur un évènement dans le système (changement d'état ou réaction à une sollicitation exogène)
  - **WARN** : une erreur est survenue, le système ne s'est pas comporté comme prévu, mais aucune intervention humaine n'est requise immédiatement
  - **ERROR** : une erreur est survenue, une intervention humaine est requise pour corriger le problème
  - (**FATAL** : une erreur irrémédiable est survenue, suite à quoi le système s'est arrêté)

NOTE	Une erreur métier n'est pas forcément une erreur technique. Par exemple : un utilisateur qui entre un mauvais mot de passe est un cas métier normal, géré par le code, et l'évènement correspondant (s'il est tracé) est de sévérité **INFO**.

4.1.1. Architecture de SLF4J

Le problème avec cette multiplicité de choix d'outil est que l'écosystème s'est construit de manière hétérogène.

- Tomcat utilise **JUL** à travers l'abstraction **JULI**.
- Spring utilise **Apache Commons Logging**.
- Le client **HBase** officiel utilise **Log4j**.
- Etc.

Une application qui utilise différents frameworks et bibliothèques devrait alors configurer chacune de ces technologies de *logging* et ce de manière cohérente (même format de ligne, même fichier, même sévérité minimum, etc.).

Ce serait pénible, et le risque de problèmes serait élevé (concurrence d'accès sur un même fichier, etc.).

Une technologie existe cependant, pour unifier tous ces outils : **SLF4J**.

Cet outil est composé

- D'une ***API abstraite*** représentant de manière unifiée un grand nombre des fonctionnalités des bibliothèques de log existantes
- D'***adaptateurs*** entre cette API et les bibliothèques de logs existantes qui ne l'implémentent pas
- De ***bridges***, bibliothèques ayant la même compatibilité binaire (même noms qualifiés des classes, même signatures des méthodes) que les bibliothèques de logs existantes, mais redirigeant les messages vers l'***API abstraite***

[slf4j bridges] | *slf4j_bridges.png*

Dans une application mettant cette technologie à profit, toutes les logs sont redirigées vers cette ***API abstraite*** et envoyées vers une seule et unique bibliothèque de log.

La configuration est alors faite une fois, et il n'y a plus de risque de concurrence d'accès.

4.1.2. MDC (Mapped Diagnostic Context)

Le **MDC** est un outil fourni par la plupart des bibliothèques de log qui permet de transporter de l'information dans un même thread.

L'idée est d'enrichir les informations transverses utiles pour une ligne de log au fur et à mesure de leur disponibilité, sans avoir à les passer en paramètres de toutes les méthodes précédant l'écriture du message.

Considérant ce code :

Listing 4. Fichier MyController.java

```
class MyController {

    private final MyService service;

    public User newUser(User user, @Header("correlationId") String correlationId) {
        return service.newUser(user, correlationId);
    }
}
```

Listing 5. Fichier MyService.java

```
class MyService {

    private final Logger logger = LoggerFactory.getLogger(MyService.class);
    private final MyRepository repository;

    public User newUser(User user, String correlationId) {
        if(isValid(user, correlationId)) {
            return repository.save(user, correlationId);
        } else {
            throw new InvalidUserException();
        }
    }

    private boolean isValid(User user, String correlationId) {
        if(user.age > 110) {
            logger.info("[" + correlationId + "] Invalid User: too old");
            return false;
        } else if(user.age < 1) {
            logger.info("[" + correlationId + "] Invalid User: too young");
            return false;
        }
        return true;
    }
}
```

La variable `correlationId` est passé systématiquement car il est nécessaire de la logger à chaque fois pour réconcilier plusieurs lignes de logs qui concernent le même appel d'un utilisateur.

C'est une bonne candidate pour le **MDC** :

Listing 6. Fichier MyController.java

```
class MyController {

    private final MyService service;

    public User newUser(User user, @Header("correlationId") String correlationId) {
        MDC.put("correlationId", correlationId);
        return service.newUser(user);
    }
}
```

Listing 7. Fichier MyService.java

```
class MyService {

    private final Logger logger = LoggerFactory.getLogger(MyService.class);
    private final MyRepository repository;

    public User newUser(User user) {
        if(isValid(user)) {
            return repository.save(user);
        } else {
            throw new InvalidUserException();
        }
    }

    private boolean isValid(User user) {
        if(user.age > 110) {
            logger.info("Invalid User: too old");
            return false;
        } else if(user.age < 1) {
            logger.info("Invalid User: too young");
            return false;
        }
        return true;
    }
}
```

Il n'y a plus qu'à configurer le bon format de sortie des lignes de log pour que le `correlationId` soit présent.

Listing 8. Fichier logback.xml

```
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - [%X{correlationId}]
%msg%n</pattern> ①
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

① Ici une ligne de log sera structurée comme suit : <heure> <nom du thread> <sévérité> <nom de la classe> - <correlationId> <message>

4.2. Logs GC

Les logs applicatives sont une chose, mais quand un problème arrive en production, son origine ou ses symptômes peuvent être techniques.

Il est important de ne pas oublier d'activer les logs du **G**arbarge **C**ollector.
Cela se fait avec un flag sur la ligne de commande du démarrage de la JVM :

```
-Xlog:gc=debug:file=gc.log:pid,time,uptimemillis:filecount=5,filesize=1M
```

Ici la JVM va écrire les évènements à partir de la sévérité **debug** dans un fichier **gc.log** (en gérant la rotation sur 5 fichiers max de 1M chacun) avec les informations PID (processus ID), heure et durée d'exécution de l'application.

Plus d'options ici : https://openjdk.java.net/jeps/158

Alternativement des outils de collecte de métriques tels que **Micrometer** peuvent exporter ces informations vers des systèmes ingérant des *séries temporelles* (Time Series) tels que **Graphite**, **Warp10** ou **Prometheus** (pour du on-premise).

4.3. Heap Dump

Quand la JVM s'arrête de manière non prévue, cela peut être à cause d'un problème de mémoire (**O**ut **O**f **M**emory error) et dans ce cas l'investigation peut se faire sur la base d'un **Heap Dump**.

Un **Heap Dump** est une projection sur fichier de l'état de la mémoire de la JVM.

Voici donc le flag à ne pas oublier sur la ligne de commande du démarrage de la JVM :

```
-XX:+HeapDumpOnOutOfMemoryError
```

Par la suite un tel fichier peut être analysé avec des outils comme **Eclipse MAT** (gratuit) ou **JProfiler** (payant).
:imagesdir: ./images

5. Intégrations

L'écosystème Java s'est grandement développé au fil des années et il est aujourd'hui d'une grande richesse.

Pour chaque technologie ou protocole, il existe presque certainement une librairie, ou plusieurs

- Sql : jooq, hibernate, spring-data, etc.
- No Sql : clients pour couch, cassandra, redis, hbase, neo4j, etc.
- Brokers : clients pour Kafka, Rabbit, etc.
- Serveurs HTTP : Tomcat, Jetty, Undertow, etc.
- Clients HTTP : JDK, Retrofit, Apache http-client, Spring `RestTemplate`, etc.
- Serialization : Jackson, SnakeYAML, JaxB, etc.
- Moteurs d'indexation : clients pour ElasticSearch, OpenSearch, SolR, etc.
- EIP : Spring-Integration, Apache Camel, etc.
- Etc.

Faire un choix sur une technologie d'intégration ou une autre peut être complexe et engageant.

Par exemple, décider d'utiliser Apache Camel pour profiter des nombreux connecteurs de cet écosystème enfermera dans un certain périmètre.

A contrario, rechercher la simplicité en intégrant des bibliothèques minimalistes nécessitera peut-être plus de code, mais laissera le champ libre pour les fonctionnalités à venir.

5.1. Un mot sur JDBC

Java **D**ata**B**ase **C**onnectivity est l'API standard synchrone pour s'interfacer avec une base de donnée relationnelle (SQL).

En tant qu'API, elle fournit l'abstraction `java.sql.Driver` qui est implémentée par les différents clients existants (Oracle, PostgreSQL, MariaDB, etc.).

Cela permet d'utiliser le même code Java, quelle que soit la base (SQL) utilisée.

> Attention cependant, chaque base a son lot de spécificités en plus de la syntaxe SQL-92 et si elles sont utilisées, le code (SQL) n'est plus portable.

Exemple de code utilisant l'API JDBC :

```
class JdbcDemo {

    public static void main(String[] args) throws ClassNotFoundException, SQLException
{
        Class.forName("postgresql.Driver"); ①
        String dbUrl = "jdbc:postgresql://localhost:5432";

        try (Connection connection = DriverManager.getConnection(dbUrl, "postgres",
"example"); ②
             PreparedStatement stmt = connection.prepareStatement("SELECT name, age
FROM person WHERE age > ?")) { ③

            stmt.setInt(1, 28); ④

            ResultSet rs = stmt.executeQuery(); ⑤

            while (rs.next()) { ⑥
                System.out.print(rs.getString(1) + " :  " + rs.getInt(2));
            }
        }
    }
}
```

① Charge le Driver qui va s'enregistrer auprès du `DriverManager`, si la classe `postgresql.Driver` n'est pas présente dans le *classpath*, une exception sera levée

② Ouverture d'une connection vers la base de donnée (en utilisant *login* et *password*). Avec cette syntaxe, appelée *try-with-resources*, la connection sera automatiquement fermée, demande pour le `PreparedStatement` à la ligne d'après

③ Création d'un `PreparedStatement` qui permet de variabiliser la requête SQL avec des paramètres qui seront échappés par l'API (et évitera la fameuse vulnérabilité d'injection SQL). Par ailleurs cette syntaxe va jusqu'à la base de données qui pourra réaliser des optimisations qui profiteront aux executions ultérieures, même avec des paramètres différents.

④ Définition de la valeur du premier (et seul) paramètre

⑤ Execution de la requête et récupération du résultat

⑥ Le résultat étant conceptuellement un `Iterator`, on pourra lire la prochaine ligne tant que la méthode `next()` renvoie `true`

L'API JDBC est pratique car elle abstrait du type de base utilisé et permet d'avoir un code portable (à la syntaxe SQL près).

Cependant ce code est nécessaire à chaque requête, et il peut y avoir beaucoup de duplication si on utilise uniquement cette API.

Spring (entre autres) fournit des utilitaires au-dessus de l'API JDBC qui permettent d'avoir un code d'appel plus concis et de réaliser un mapping sommaire vers des objets Java.

```
class JdbcTemplateDemo {

    public static void main(String[] args) {
        NamedParameterJdbcTemplate jdbcTemplate = new NamedParameterJdbcTemplate
(buildDataSource()); ①

        List<Person> persons = jdbcTemplate.query(
            "SELECT name, age FROM person WHERE age > :age", ②
            Map.of("age", 28),
            (rs, rowNum) -> new Person(rs.getString("name"), rs.getInt("age")) ③
        );

        persons.forEach(System.out::println);
    }

    record Person(String name, int age) {
    }

    private static DataSource buildDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("postgresql.Driver");
        dataSource.setUrl("jdbc:postgresql://localhost:5432");
        dataSource.setUsername("postgres");
        dataSource.setPassword("example");

        return dataSource;
    }
}
```

① Dans une application plus complexe, cette même instance serait *injectée* dans le code des *Repositories* la nécessitant

② Ici les paramètres sont nommés pour éviter le couplage de position

③ Il est possible de passer une fonction qui transforme une ligne en objet et celle-ci sera appelée en boucle pour construire la liste résultante

JDBC est la brique de base sur laquelle sont construits les **O**bject **R**elational **M**apper tels qu'Hibernate ou jOOQ.

6. API design

Les APIs exposées à l'extérieur d'un système peuvent être documentées de différentes manières :

- **Contract First** : on écrit le contrat dans un format donné (RAML, WSDL, etc.), et ensuite on écrit ou on *génère* le code correspondant.
  À noter que si le code est généré, il y a toujours une seconde étape de "raccordement" au code métier.
- **Code First** : on écrit le code technique permettant d'exposer l'API.
  Ce code peut être enrichi de métadonnées sur l'utilisation de l'API (description, signification des différents codes d'erreur, etc.).
  La documentation est ensuite générée sur la base de ce code, notamment avec l'initiative **OpenAPI** (anciennement Swagger) et les nombreux outils de son écosystème.
  Dans le cas où ce mécanisme est inclus dans la construction du projet, la documentation évolue à chaque changement de code.

L'approche **code first** est la plus pérenne, car elle évite qu'un décalage entre la documentation et le code s'introduise au fil du temps.

Il faut cependant être vigilant sur les modifications d'une API et à ce que cela va impliquer pour les clients de cette API.

Certains changements sont sans conséquences

- Ajouter un champ dans une réponse
- Supprimer un champ optionnel dans une requête

→ *Qui peut le plus, peut le moins*.

D'autres changements, comme

- Supprimer un champ dans une réponse
- Ajouter un champ obligatoire dans une requête

Vont entrainer des malfonctions dans la communication, et nécessitent d'être abordés différemment.

Dans le cas où ce second genre de modifications est nécessaire, il est possible de proposer une nouvelle version de l'API tout en continuant de servir la première version.

Ainsi les clients ont le temps de faire le changement de code nécessaire de leur côté pour utiliser la nouvelle version.
Quand tous les clients ont opéré ces changements, le code de la première version peut êre supprimé.

Ce faisant, on introduit un ***découplage*** entre les systèmes en permettant que chacun fasse les changements à son rythme.

7. Notions sur la Production

7.1. 12 factors App

Il s'agit de 12 principes réalisés comme retour d'expérience par les équipes de **Heroku**, une plateforme de déploiement "dans le cloud".

La référence complète peut être trouvée ici : https://12factor.net/

Ces douze principes sont des bonnes pratiques, applicables du domaine du développeur à celui de l'administrateur système.

Leur mise en place permet de simplifier la gestion de l'infrastructure.

7.2. A propos du déploiement (en Production)

En 2019, l'équipe DORA (**D**ev**O**ps **R**esearch and **A**ssessment) de **Google** produit un rapport d'analyse de 6 ans sur le fonctionnement des équipes informatiques dans de nombreuses entreprises.

Ce rapport établit 4 métriques qui permettent de mesurer la performance des équipes de développement :

- La fréquence de déploiement en *production*
- Le délai entre un commit et le déploiement en *production* de celui-ci
- Le pourcentage des déploiements entrainant un problème en *production*
- Temps nécessaire pour restaurer le service après un problème en *production*

Source : https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance

Ces conclusions sont assez logiques quand on regarde ça sous l'angle du risque.

Changer quelque chose dans un système informatique inclus toujours un risque de problème.

Plus le changement est gros, plus le risque qu'il y ai un problème est élevé car le volume de code changé est plus important.

Ainsi, faire des versions (et des déploiements) régulièrement diminue ce risque, car chaque changement est plus petit.

Par ailleurs, la maturité et la technologie qui permettent de déployer régulièrement sont aussi au service de la correction rapide d'une anomalie en *production*.

Concrètement atteindre ces objectifs, c'est mettre en place une industrialisation des processus, par l'intermédiaire des outils tels que :

- SCM : gestionnaire de code source (Git, SVN, CVS, etc.)

- CI/CD : intégration et déploiement continu (Jenkins, GitLab CI, GitHub Actions, Circle CI, etc.)
- Supervision : collecte et suivi des logs et métriques applicatives (ELK, Micrometer, Prometheus, Warp10, Grafana, Datadog, etc.)
- Etc.

7.3. Centralisation des logs

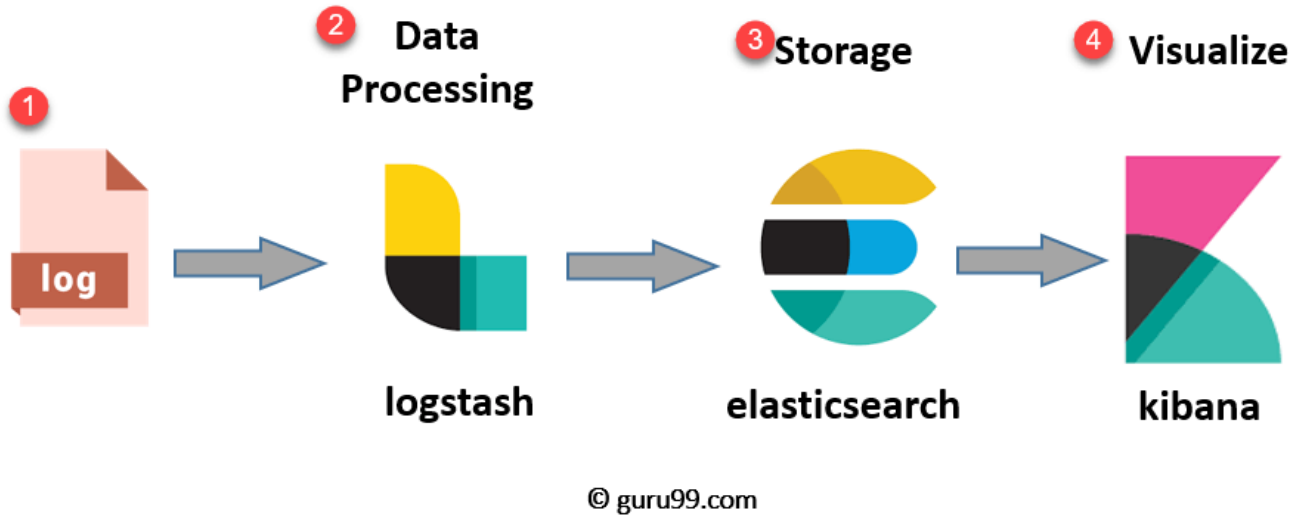
Afin de pouvoir exploiter au mieux les logs, il est important de les centraliser.
C'est-à-dire les envoyer dans un même endroit, que ce soit dans un :

- Système de fichier, sur lequel on pourra utiliser la commande `grep` par exemple
- *ElasticSearch*, *Loki* ou équivalent, sur lequel on pourra faire des requêtes
- SAAS dont c'est le coeur de métier (*Datadog*, *Logz.io*, etc.)

Les logs doivent être claires et apporter le maximum d'information possible.
Pour cela on évite les logs visuelles ou de *debug*.

7.3.1. La stack Elastic

Il existe plusieurs solutions de centralisation *on premise*, la plus connue étant la suite proposée par *Elastic* : **ELK** (**E**lasticSearch, **L**ogstash, **K**ibana).
Aujourd'hui un peu datée, des composants peuvent être remplacés, la chaîne peut être plus complexe pour mieux gérer le dimensionnement, mais la structure reste néanmoins identique.



La brique **Logstash** permet de découper les lignes de logs en documents structurés (JSON) pour ensuite les envoyer à **ElasticSearch**.
ElasticSearch indexe les documents reçus et propose une API de recherche utilisée par **Kibana**.
Kibana fourni une interface Web qui satisfait aussi bien à la recherche exploratoire qu'à la création de tableaux de bord pour suivre l'état du système.

7.4. Supervision

La collecte de logs et de métriques forme la base de la *supervision* d'un système.

Sur cette base, il est nécessaire d'ajouter des composants pour transformer ces données brutes en :

- Alertes, dans le cas où un comportement anormal est détecté
- Tableaux de bord, pour investiguer un comportement anormal, trouver des correlations et converger vers l'origine du problème.

La supervision, c'est l'outillage qui permet de suivre l'état d'un système, de comprendre son fonctionnement, et même de prévoir son utilisation.

CONSEIL

Il faut néanmoins faire preuve de retenue et ne garder que les informations qui ont de la valeur, au risque de se perdre dans trop d'information et de devoir stocker d'énormes quantités d'informations inutiles.

Le résultat doit être construit à l'aide de ces trois questions :

- Quels sont les comportements du système que l'on souhaite identifier comme problématiques (service indisponible, utilisation anormale, etc.) ?
- Quels comportements du système souhaite-t-on comprendre (coup de bélier, interruption de services externes, etc.) ?
- Où placer les sondes qui remonterons les informations nécessaires ?

Que ce soit des outils d'indexation de document, de stockage de séries temporelles ou des bases de données, tous sont des sources d'information pour la constitution de tableaux de bord et d'alerte.

L'outil open-source le plus complet aujourd'hui pour aggréger ces sources est Grafana