Java 202 - Librairies & Frameworks

Loïc Ledoyen

Java 202

Introduction

vast ecosystem, blabla, lots of libs to choose among, blabla, just tools, blabla, quality come from how we use them.

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**.

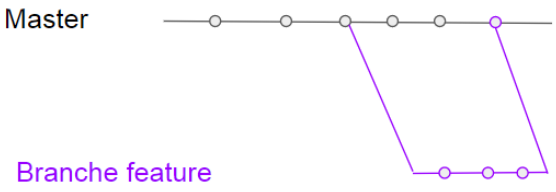
Le **rebase** peut être utilisé pour remettre à jour une branche quand la branche d'origine a changé.

```
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

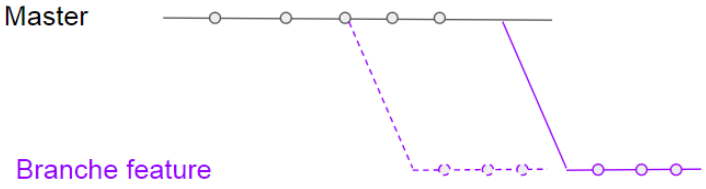
② Affiche les dix derniers commits de la branche courante

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

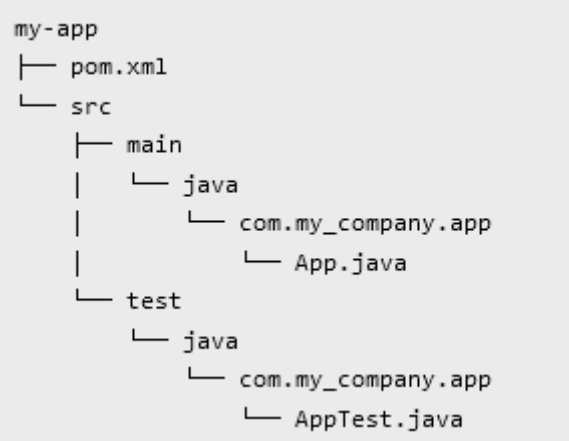


Figure 1. Structure d'un projet Maven

1.3.2. Cycle de vie d'un projet Maven

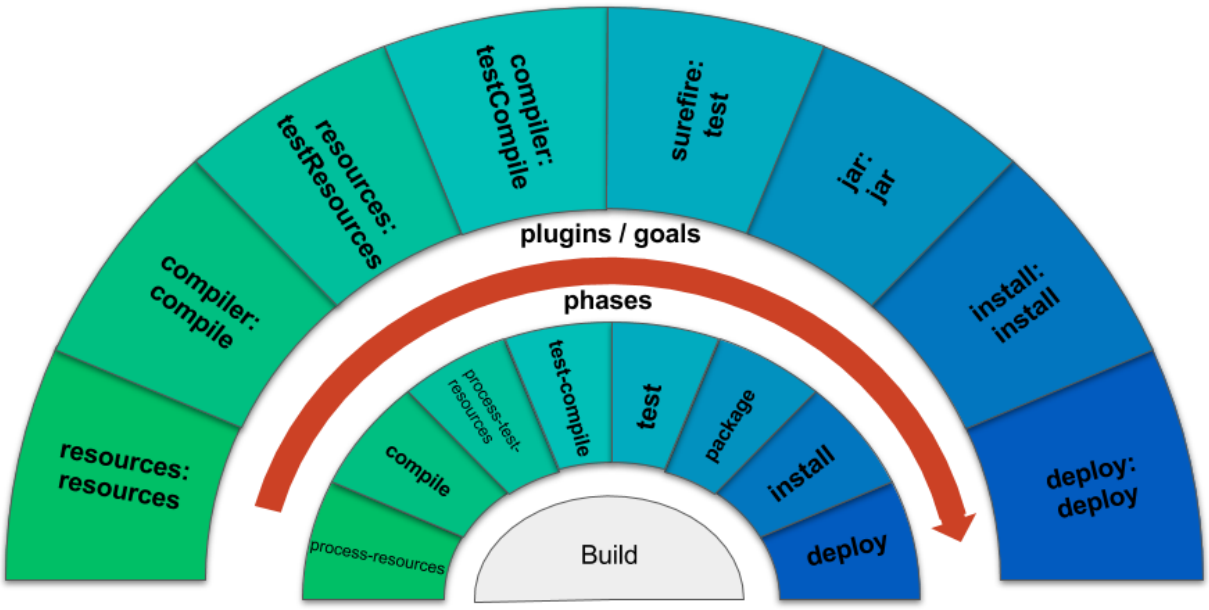
Par défaut Maven utilise un cycle de vie permettant à la grande majorité des projets d'être construit avec peu de configuration.

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.

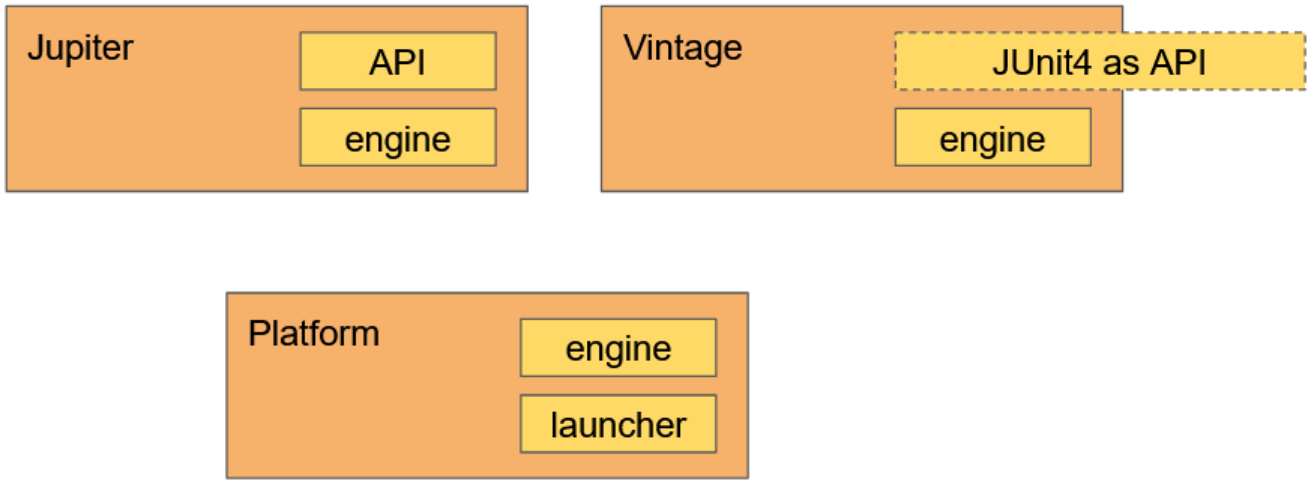
En effet, si le point de départ d'un test est conventionnellement une méthode, des plugins voient le jour pour changer ce paradigme (Cucumber, etc.), où un test peut-être un paragraphe dans un fichier texte.

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.

2. Fonctionnement des frameworks

bla bla

2.1. Compile-time

APT, quarkus

2.2. Runtime

Reflection, sample with Jackson

3. Gestion des logs

blabla sur les nombreuses libs de log

3.1. SLF4J

API, bridges, concrete impl

4. Intégrations

blabla sur la non-exhaustivité

dessin type archi hexa avec un panel des technologies "classiques"

- sync (HTTP/JSON, web-socket, rsocket),
- async (Kafka, Rabbit, etc.)
- SQL (postgresql, mysql, oracle)
- no-sql (couch, cassandra, redis, hbase, neo4j, etc.)
- index (ES)

Code sample ?

⇒ TP amqp
⇒ TODO TP ES ?

5. API design

cf 201

6. Key takeaway points on Production

6.1. 12 factors

6.2. deployment (automation, risk reduction, etc.) (include debate on golden image ?)

6.3. log centralisation

6.4. supervision