Java 201 - Architecture logicielle

Loïc Ledoyen

Java 201

Introduction

> Architecture is about the important stuff.
> Whatever that is.
>
> — Ralph Johnson

> It means that the heart of thinking architecturally about software is to decide what is important, (i.e. what is architectural), and then expend energy on keeping those architectural elements in good condition.
> For a developer to become an architect, they need to be able to recognize what elements are important, recognizing what elements are likely to result in serious problems should they not be controlled.
>
> — Martin Fowler

Un logiciel doit répondre à plusieurs objectifs dépendant de son contexte.
Parmi ces objectifs on retrouve :

- Maintenabilité / évolutivité
- Testabilité
- Fiabilité / Robustesse / tolérance à la panne
- Scalabilité
- Sécurité
- Performance

L'architecture logicielle est l'ensemble des moyens techniques utilisés pour répondre à ces objectifs tout en satisfaisant aux contraintes non techniques du projet.

Tous les objectifs ne peuvent pas être atteint d'un coup, et l'architecture logicielle est le compromis qui convient le mieux pour atteindre les objectifs prioritaires dans le contexte du projet.

Les facteurs peuvent être multiples :

- Délais courts (*deadline*)
- Budget limité
- Certification de sécurité nécessaire
- Connexion à un système *historique* utilisant une technologie spécifique
- Etc.

L'ordre de priorité de ces objectifs doit être décidé avec l'ensemble des acteurs du projet et réajusté périodiquement afin de suivre l'évolution du projet dans le temps.
Il s'agit d'un partenariat.

L’équipe de développement met en oeuvre les principes d’architecture logicielle pour répondre au mieux aux objectifs du projet.

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 ré-incorporée 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**.

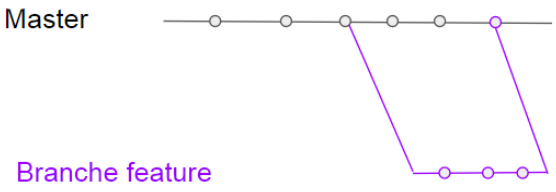
Le **rebase** peut être utilisé pour remettre à jour une branche quand la branche d'origine a changé.

```
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

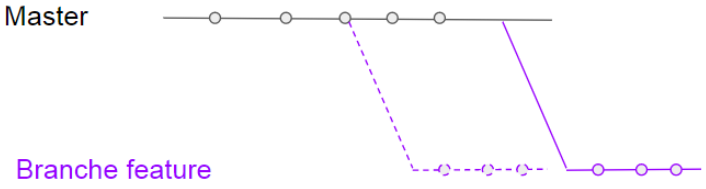
② Affiche les dix derniers commits de la branche courante

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.

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 un SCM
  - Dans **src** on retrouve deux répertoires : **main** et **test** qui contiennent respectivement le code de production, et le code de test, 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

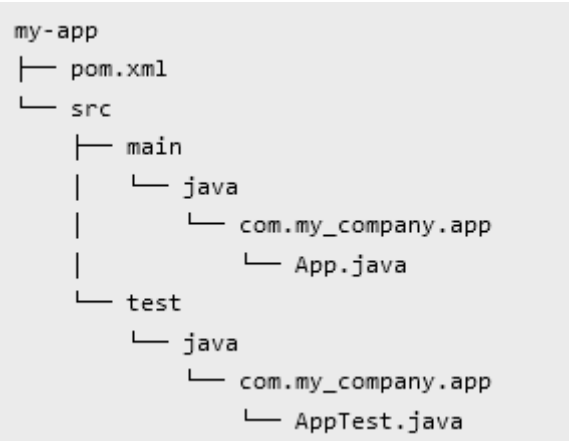


Figure 1. Structure d'un projet Maven

1.3.2. Cycle de vie d'un projet Maven

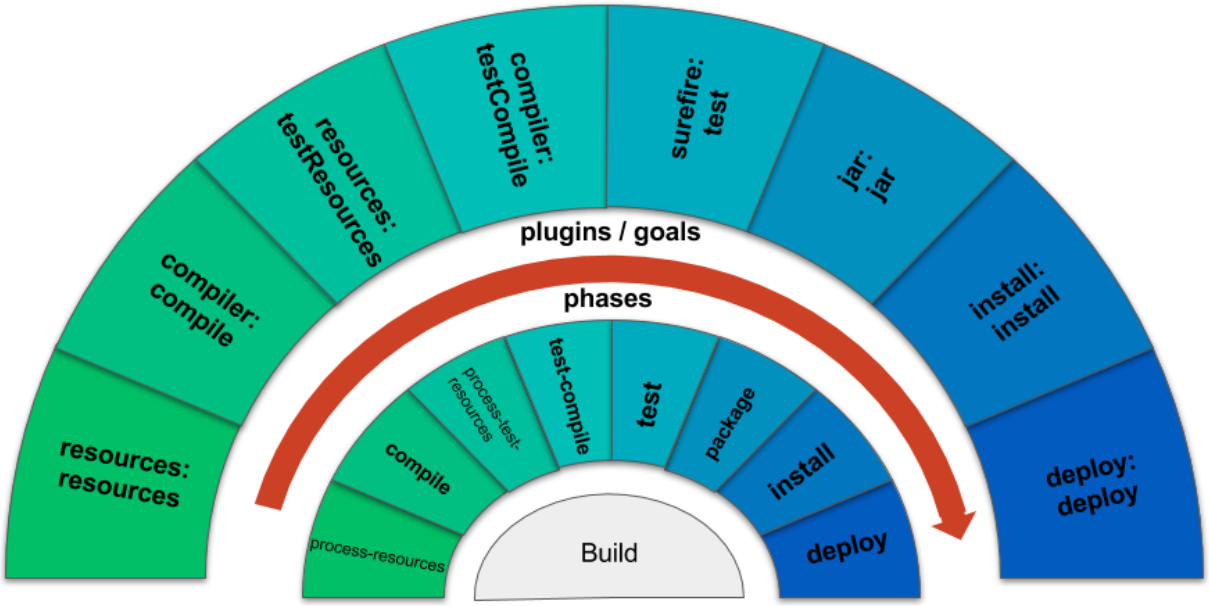
Par défaut Maven utilise un cycle de vie permettant à la grande majorité des projets d'être construit avec peu de configuration.

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 punlie 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

```xml
<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)

```xml
<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 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 le plus répandu aujourd'hui.

1.4.1. Utilisation de l'API de JUnit-Jupiter

```
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 marqué 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 l'appel consécutif de méthode 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 de vérification différent, 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) l'ordre des données utilisé.

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 librairie 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.

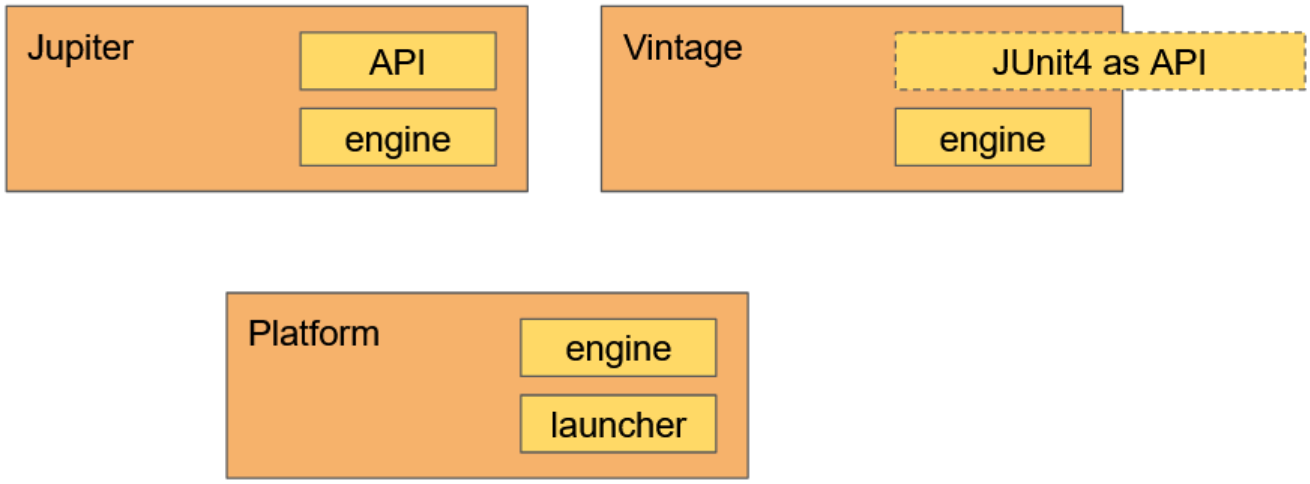
En effet, si le point de départ d'un test est conventionnellement une méthode, des plugins voient le jour pour changer ce paradigme (Cucumber, etc.), où un test peut-être un paragraphe dans un fichier texte.

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. Le Découplage, un concept valide a toutes les échelles

> J'ai répondu à leur problème en leur proposant d'ajouter un niveau d'indirection.
>
> — un architecte qui passait par là

L'objectif qui prévaut sur tous les autres est la **maintenabilité**.
En cas de problème, lié à la performance, à la sécurité ou au fonctionnement d'un logiciel, la solution doit tendre vers un impact minimum.
Plus le changement de code est important, et plus le risque augmente.
Une correction (mais cela vaut pour une évolution également) doit comporter le moins de risque possible.

2.1. A l'échelle d'une classe Java

Ce découplage se traduit, par exemple, par le fait de

2.1.1. Exemple 1 : État interne différent du contrat public

Considérant cette classe :

Listing 2. Fichier TrafficLight.java

```
class TrafficLight {

    private int color;

    public void setColor(int newColor) {
        this.color = newColor;
    }

    public int getColor() { ①
        return color;
    }
}
```

① Changer le type de la représentation interne (`int`) demandera d'adapter l'ensemble du code qui utilise cette classe

La classe pourrait être ré-écrite de façon à découpler :

- d'une part, la représentation interne de l'état de cet objet (un `int`)
- d'autre part, le contrat `public` utilisable par les autres classes

Listing 3. Fichier TrafficLight.java

```
class TrafficLight {

    private int color;

    public Color nextState() {
        color = (color + 1) % 3;
        return Color.values()[color];
    }

    public enum Color {
        GREEN,
        ORANGE,
        RED,
    }
}
```

2.1.2. Exemple 2 : Paramètres de méthode dont les types correspondent au domaine

Considérant cette interface :

Listing 4. Fichier PersonRepository.java

```
interface PersonRepository {

    void savePerson(String firstname, String lastname, int birthYear, int birthMonth);
}
```

A l'usage, il est simple d'inverser un paramètre avec un autre car leurs types sont identiques.
Ainsi la compilation ne va pas aider à détecter un bug, où le mois et l'année sont inversées par exemple.
Il s'agit d'un *couplage de position*.
Un meilleur design pourrait être :

Listing 5. Fichier PersonRepository.java

```
interface PersonRepository {

    Person savePerson(Person person);

    @RecordBuilder ①
    record Person(String firstname, String lastname, YearMonth birthMonth) {}
}
```

① utilisation de la librairie **io.soabase.record-builder:record-builder** pour générer un *builder* correspondant

Il est maintenant difficile de se tromper en écrivant :

```
Person person = PersonBuilder.builder()
    .firstname("Bobby")
    .lastname("Singer")
    .birthMonth(YearMonth.of(1950, Month.MARCH))
    .build();

repository.savePerson(person);
```

2.1.3. Exemple 3 : Contrat public extrait dans une interface

L'intérêt des interfaces est de décorréler le contrat public de l'implémentation concrète afin de pouvoir :

- Substituer un objet par un autre (implémentant la même interface) sans modifier le code appelant
- Cacher l'implémentation (méthodes de l'objet non présentes dans l'interface) dans le code appelant

Le concept de `Logger`, largement utilisé dans l'informatique de gestion est une abstraction pour envoyer un message *quelque part*.
Du point de vue du code métier, peu importe ce '*quelque part*'.
Ainsi cette abstraction est une interface, par exemple :

Listing 6. Fichier Logger.java

```
interface Logger {

    void log(Level level, String message);

    enum Level {
        INFO,
        WARNING,
        ERROR,
        ;
    }
}
```

Et s'utilise de cette manière :

```
record CoffeeShop(CoffeeMaker coffeeMaker, Logger logger) {

    public Cup makeCoffee(String firstname) {
        if(!coffeeMaker.isReady()) {
            logger.log(Level.WARN, "Tried to make some coffee, but the coffee maker is
not ready yet");
            return Cup.EMPTY;
        }
        Cup cup = new Cup(firstname);
        coffeeMaker.pourIn(cup);
        logger.log(Level.INFO, "Made coffee for " + firstname + ", careful it's hot
!");
        return cup;
    }
}
```

Ainsi passer une implémentation de `Logger` qui écrit dans

- la sortie standard
- un fichier
- une base de données
- un broker de message
- un mélange de toutes ces possibilités

ne changera pas le code de la classe `CoffeeShop`.

2.1.4. Différentes formes de couplage

Différentes formes de couplages peuvent être retrouvées ici : https://connascence.io/

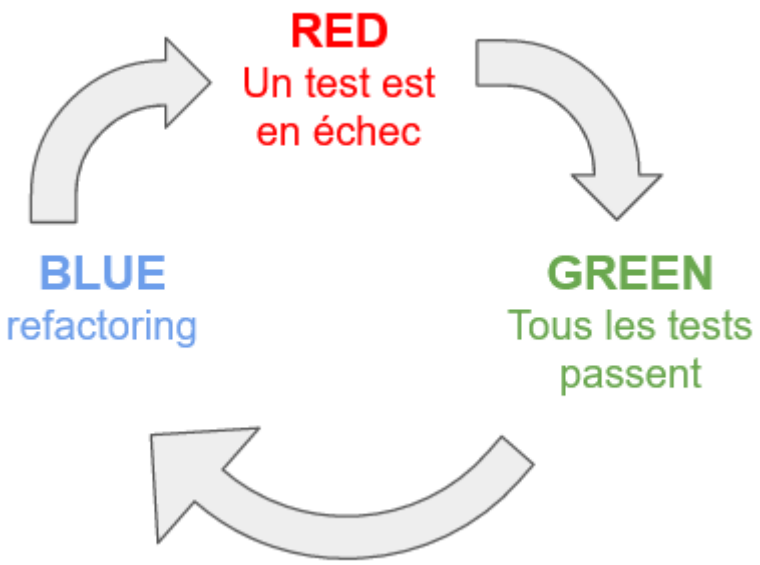
La plupart des couplages peuvent être évités en utilisant le code produit en même temps qu'il est créé.
La technique la plus simple est de suivre les principes du **TDD** (Test Driven Development) ou développement piloté par les tests.

La pratique du TDD consiste à écrire un test *minimal* avant d'écrire le code de production *minimal* qui le fait passer.
Le code produit, répondant strictement aux cas de tests réalisés, est par construction validé par les tests et utilisable facilement (car déjà utilisé dans les tests).

> 1) Write NO production code except to pass a failing test
> 2) Write only enough of a test to demonstrate a failure
> 3) Write only enough production code to pass the test
>
> — Robert "Uncle Bob" Martin, Three laws of TDD

Le fait d'écrire un test minimal est très important car on souhaite, dans la pratique du TDD,

écourter au maximum le temps d'une itération entre les différentes phases :



2.2. A l'échelle d'une application

2.2.1. Découplage métier

Une application touche la plupart du temps à plusieurs concepts métiers distincts.

Ainsi dans une application de e-commerce on peut retrouver, la gestion du catalogue, le descriptif d'un produit, le détail du panier, le paiement, la facturation, etc.

Ces domaines ont des liens entre eux, mais peuvent évoluer indépendamment les uns des autres.
Le code doit retranscrire ces liaisons, mais aussi cette indépendance.
Ainsi le fait de modifier la gestion du panier n'aura (ou ne devrait) pas avoir d'impact sur le paiement.

Mettre en place le découplage entre les composants d'une application permet de diminuer le risque en cas de changement, et d'évaluer plus finement les impacts afin de savoir où mettre l'accent sur les tests par exemple.

Ce genre de découplage nécessite une compréhension profonde des domaines métiers en jeu, pour trouver les points d'interconnexion entre eux et créer un modèle dédié à chaque domaine avec les informations nécessaires à ces échanges.

2.2.2. Découplage technique

De la même façon, il est intéressant de séparer le code dit *métier* (c'est-à-dire qui contient les règles métiers), du code technique.
On parle de code technique quand celui-ci ne porte pas directement de règle métier, par exemple l'interfaçage avec le monde extérieur au travers d'une

- API

- Interface graphique
- Connexion à un broker de message
- Connexion à une base de données

Ainsi il est possible de construire le *coeur* applicatif sans l'aide de framework ou autre librairie afin de simplifier l'écriture des tests et de réduire à sa plus simple forme le code *métier*.

On pourra venir par la suite y brancher des *connecteurs* qui feront le pont entre ce code et les reste du système ou les utilisateurs.

C'est l'architecture hexagonale que nous verrons en détail un peu plus loin.

2.3. A l'échelle d'un ensemble d'application

Les considérations a cette échelle sont valables pour un SI (système d'information).

Dans un système d'information, il est courant que plusieurs applications, opérées par des équipes distinctes doivent échanger des données.

Les choix d'architectures réalisés dans ce cadre doivent aussi bien prendre en compte les objectifs énumérés en introduction (Maintenabilité, Testabilité, Sécurité, etc.) que les frictions humaines entre les équipes.

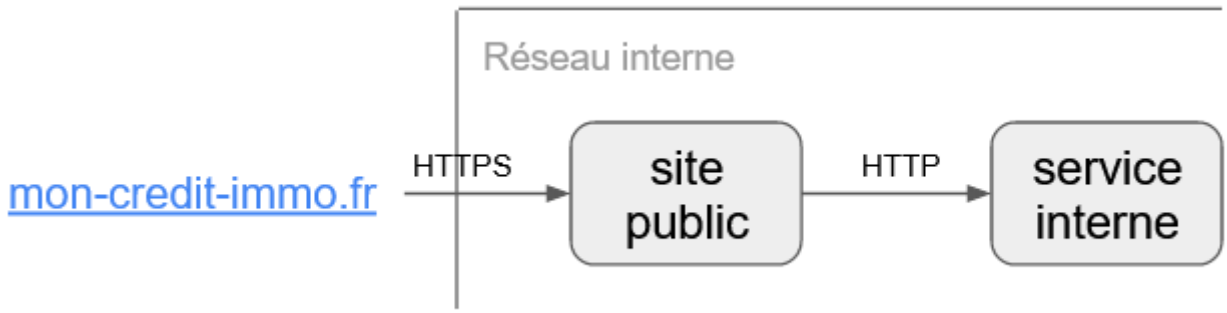
Les équipes qui travaillent sur des applications différentes sont constituées d'hommes et de femmes qui ont des façons de travailler différentes (agile, cycle en V, etc.), des rythmes différents (décalage horaire, temps partiel, etc.), des façons d'opérer la production différentes (avec ou sans coupure de service, livraison continue, ou par lot, etc.).

Ces équipes n'ont pas forcément la même maturité technique non plus.

Il est important de prendre en compte ces éléments pour protéger le service rendu et fluidifier les échanges entre équipes au maximum.

2.3.1. Exemple 1 : communication asynchrone

Considérant une application soumise à de forts traffics de manière irrégulière, comme un site de simulation de crédit immobilier qui sera consulté massivement entre 12h et 14h à la pause déjeuner.



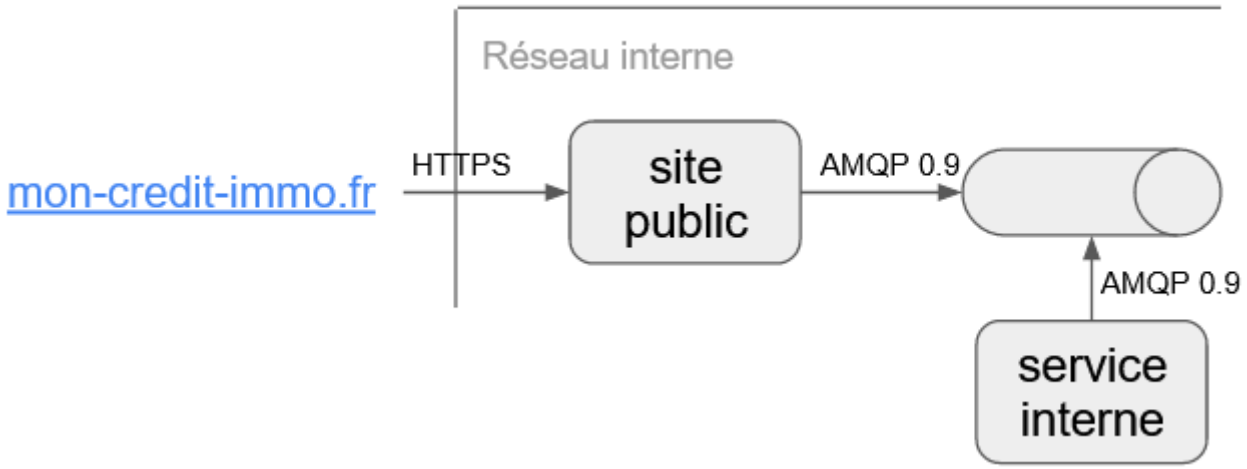
Le site public recueille les demandes de simulation et les envoient à un service interne qui doit :

- Se connecter régulièrement à des APIs externes de différentes banques pour maintenir des bases de calcul précises
- Faire ces calculs (imaginons-les lents pour l'intérêt de l'exercice) sur la base des informations données par les clients
- Renvoyer la simulation au site public

Même s'il y a peu de traffic, une interruption de service de quelques minutes peut être dommageable, et faire perdre des clients ou de la visibilité.

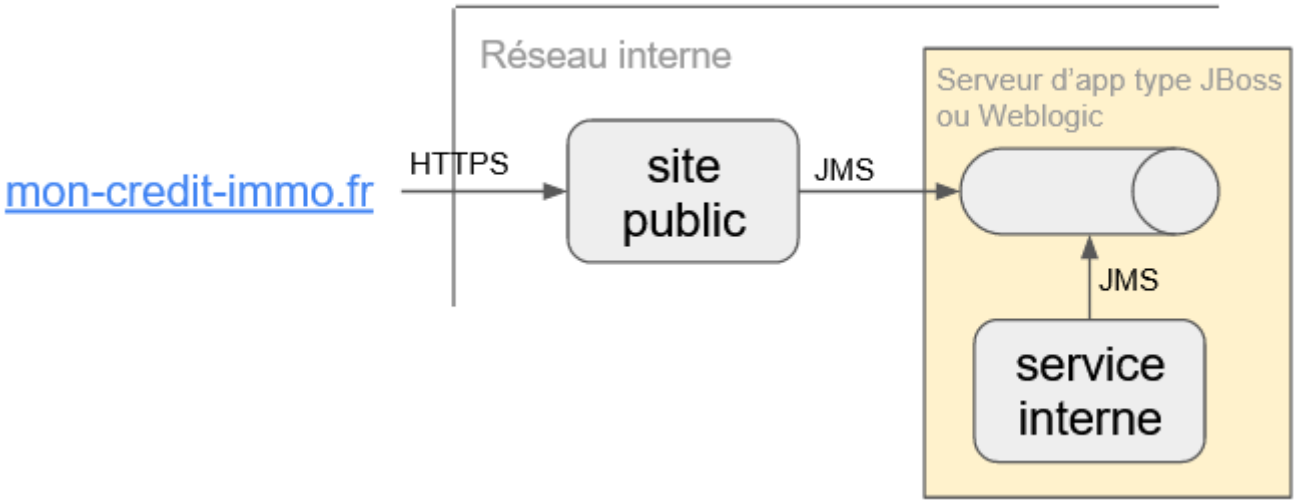
Cependant si la communication entre le site public et le service interne est synchrone, une mise à jour de ce dernier entraînerait *de facto* une coupure de service du site public ou une perte d'information.

Il peut être intéressant d'établir une communication asynchrone au travers d'un broker de message qui fera tampon entre les deux systèmes.
Ainsi il n'y aura ni coupure de service *visible* ni perte d'information.



2.3.2. Exemple 2 : lisser la charge

Dans la continuité du système décrit ci-dessus, considérant que le broker de message est couplé à l'applicatif côté service pour des raisons historiques ou de maturité technologique (server applicatif + fournisseur d'API JMS par exemple).



Envoyer beaucoup de message d'un coup (on parle de *coup de bélier*) à un tel système pourrait diminuer ses performances voir provoquer son arrêt / crash.

Il peut être dans ce cas plus simple que ce soit l'application cliente (celle qui envoie les messages) qui lisse la charge (on parle aussi de *throttling*), pour éviter le couplage de pression entre les deux applications.

Mettre en place ce mécanisme requiert de définir une vitesse maximum (en message/sec par exemple) et de la respecter.
Il existe plusieurs manières de construire un tel système, par exemple avec

- Un batch qui va lire à interval régulier les X plus vieux messages dans une base de donnée (attention cependant, un système de batch *scale* difficilement)
- Une petite application dont l'état n'est pas géré dans sa mémoire (mais plutôt dans une base de donnée, de façon à pouvoir scaler si nécessaire)
- Les fonctionnalités de certains brokers

3. Quelques types d’architecture

> Les organisations qui conçoivent des systèmes (informatiques) tendent inévitablement à produire des designs qui sont des copies de la structure de communication de leur organisation.
>
> — attribuée à Melvin Conway, Loi de Conway

Tout à un coût, et le temps de développement a souvent un coup plus élevé que le reste.

La direction que prend une architecture applicative doit permettre de faire correspondre le coût du développement avec le gain espéré et donc la stratégie de l’entreprise.

Au-delà de l’aspect mercantile de la chose, il est également plus *agréable* d’ajouter de nouvelles fonctionnalités, que de gérer une friction entre équipe ou de multiples bugs.

L’architecture applicative doit donc avoir pour objectif de fluidifier les cycles de développement pour concentrer l’énergie sur la valeur ajoutée.

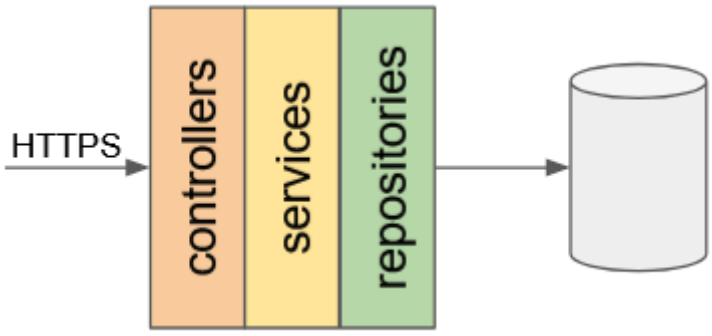
3.1. N-tiers

L’architecture N-tiers est un découpage en couches techniques d’une application.

Pour une application simple (API HTTP d’un côté, base de donnée de l’autre par exemple), on retrouve souvent **3** tiers :

- La couche des contrôleurs dont la responsabilité est de
  - Communiquer en HTTP(S)
  - Sécuriser l’accès (Authentification et Authorisation)
  - Sérialiser ou désérialiser la donnée représentée par des objets *anémiques* (DTO pour Data Transfer Object)
  - Réaliser des contrôles de surface notamment sur la structure des données envoyées et reçues
- La couche des services dont la responsabilité est de
  - Fournir aux contrôleurs des traitements métiers de haut niveau afin qu’aucune logique métier ne soit nécessaire en amont
  - Valider la cohérence des données
  - (Gérer les transactions, si la base de donnée est transactionnelle)
- La couche de persistence (on parle aussi de *repositories*) dont la responsabilité est de
  - Communiquer avec une base de donnée
  - Sérialiser ou désérialiser la donnée représentée par des objets *anémiques* (DTO pour Data Transfer Object)
  - Interpréter les erreurs techniques de la base de données pour être gérées par un traitement

métier dans la couche service



Historiquement, cette architecture est appliquée telle-quelle à l'ensemble de la base de code d'un projet informatique et apporte de la confusion sur les différents domaines métiers.

Il est plus intéressant d'introduire un découpage qui correspond aux différents domaines.

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── controllers
                ├── services
                └── repositories
```

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── blog
                ├── subscriptions
                └── twitter
```

Ainsi le fonctionnement de l'application est plus clair, et cela évite de mélanger par inadvertance des composants entre les domaines.

Par ailleurs rien n'empêche de structurer le code d'un domaine en suivant ce découpage technique.

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── blog
                │   ├── controllers
                │   ├── services
                │   └── repositories
                ├── subscriptions
                │   ├── controllers
                │   ├── services
                │   └── repositories
                └── twitter
```

3.2. Hexagonale

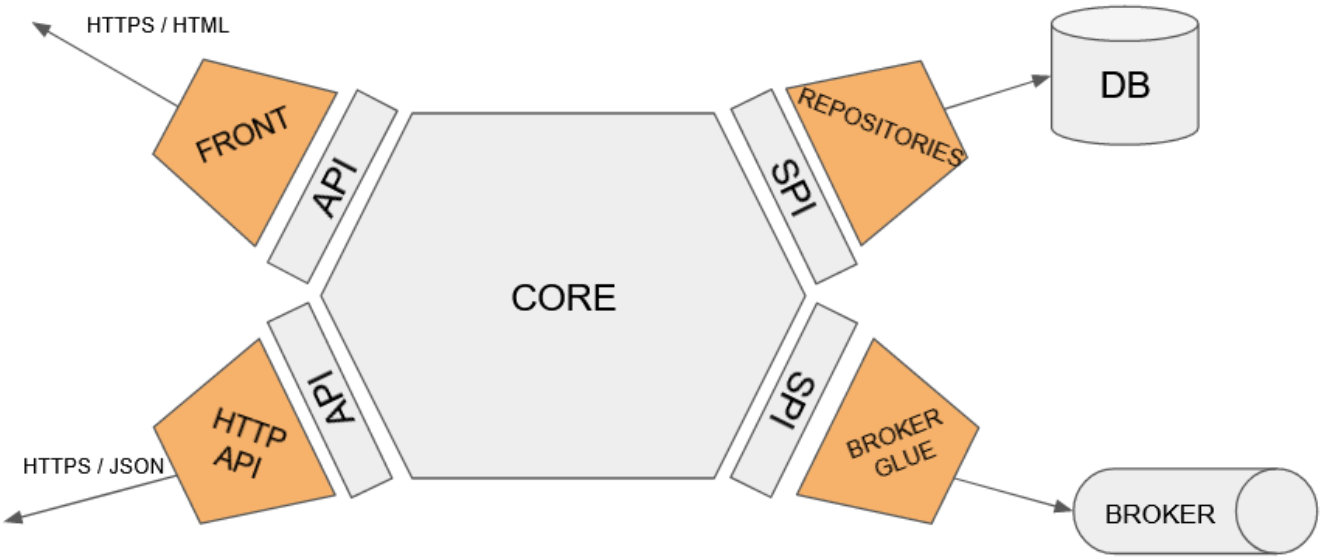
L'**architecture hexagonale** vise à séparer le code *métier* du code *technique*.

Pour celà on distingue le coeur (*core*) de l'application, des *adaptateurs* (*adapters*) qui font le lien entre le code *métier* et le reste du monde ; que ce soit via des communications *machine-to-machine* (HTTPS, MQTT, etc.) ou *human-to-machine* (interface graphique, email, SMS, etc.).

Les objectifs sont les suivants :

- Découpler les problématiques techniques en dehors du code qui contient les règles métiers

- Simplifier les tests unitaires du code métier, du fait de l'absence de frameworks et librairies
- Permettre (théoriquement) de remplacer un *adaptateur* sans modifier le reste de l'application. Dans la pratique, ce cas arrive assez peu et le cas échéant, le changement de paradigme qui y est souvent associé (SQL NOSQL, synchrone asynchrone) nécessite de revoir l'architecture de l'application.



On appelle :

- **API** (**A**pplication **P**rogramming **I**nterface) une interface qui est implémentée par le code métier, et est appelée par un adaptateur
- **SPI** (**S**ervice **P**rovider **I**nterface) une interface qui est appelée par le code métier, et est implémentée par un adaptateur.

On commence une application qui suit les principes de l'architecture hexagonale par le code d'un domaine métier / fonctionnalité.

Ce code ne doit contenir aucune dépendance vers un framework ou une librairie.

À cette fin, il est plus simple de l'isoler dans un module dédié de sorte que ce code n'ait effectivement aucune dépendance.

Par la suite d'autres modules peuvent être ajoutés contenant le code des adaptateurs, ces modules auront une dépendance sur le module coeur.

3.3. Monolithique vs micro-services

> I'll keep saying this… if people can't build monoliths properly, microservices won't help.
>
> — Simon Brown, @simonbrown

L'approche monolithique consiste à former une seule application avec la totalité des fonctionnalités.

Ce qui peut impliquer de *mélanger* plusieurs domaines métiers, en reprenant l'exemple d'un site de e-commerce : gestion du catalogue, détail du panier, paiement, facturation, etc.

Il est donc nécessaire de structurer le code d'une application monolithique en *modules* pour isoler les parties du code qui n'ont pas à interagir entre elles.

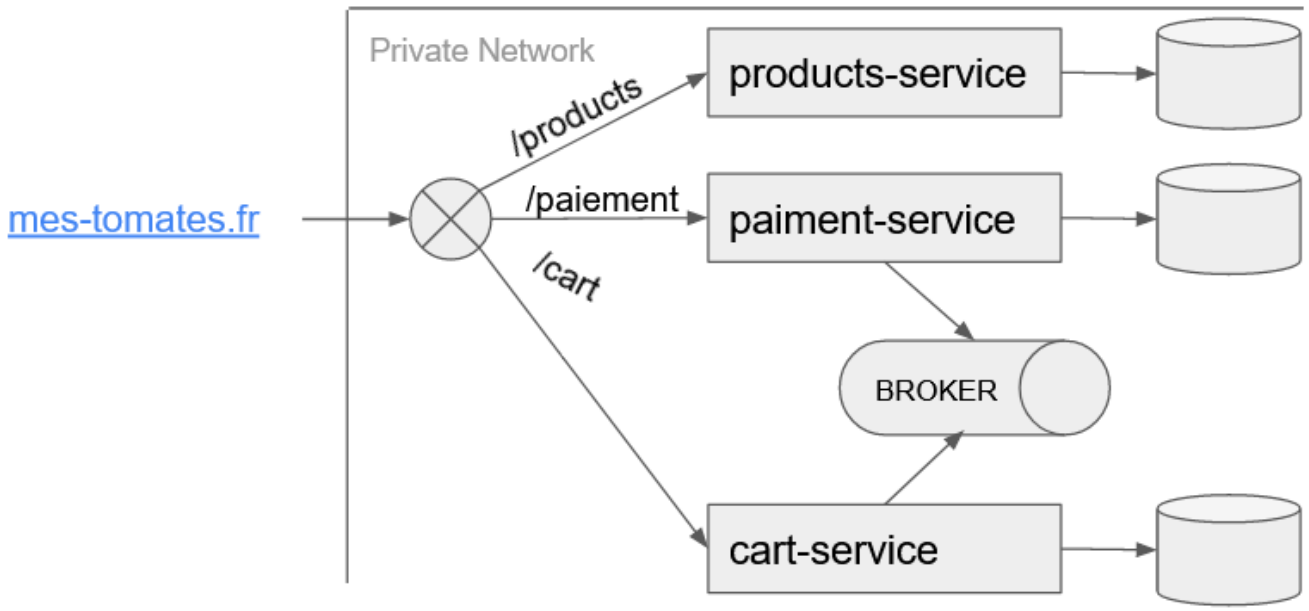
Le risque dans le cas où une telle structure n'est pas mis en place est de voir la complexité de l'application augmenter de manière exponentielle à chaque ajout de fonctionnalité.
On parle dans ce cas de code *spaghetti*.
La métaphore des *spaghettis* vient des morceaux de code entrelacés, mélangés, ce qui les rend difficile à tester en isolation du fait des nombreuses interactions entre les domaines voir les couches (au sens N-tiers).

L'approche micro-service a ceci de différent que les différents domaines sont gérés par des applications différentes.

Même s'il est toujours possible de partager du code au travers de bibliothèques communes, il est nécessaire, comme dans un monolithe de ne pas mélanger les domaines.
Une des règles qui transcrit le mieux ce principe est le fait que les bases de données soient distinctes pour chaque micro-service.
Cette règle permet également d'éviter un couplage au niveau du modèle de donnée.

De même, si des micro-services doivent communiquer entre eux, il est préférable que ce soit de manière asynchrone (à travers un broker de message par ex.).

Ainsi la coupure d'un service n'affecte pas les autres.



Une architecture basée sur les microservices a des inconvénients :

- Plusieurs applications différentes doivent être déployées fréquemment

  → La livraison et le déploiement doivent être robustes et automatisés
- Il est plus difficile d'investiguer un problème, pouvant être lié à une succession d'évènements dans différentes applications

→ Les logs doivent être uniques, informatives et centralisées

→ Une solution de traçage des appels doit être mise en place (correlation-id, header via, APM, etc.)

- Le coût en infrastructure est plus élevé, car il y a plus d'applications actives et d'organes techniques pour les interconnexions

Et a des avantages :

- Le code d'un micro-service est plus petit, plus simple
- Les micro-services peuvent évoluer indépendamment les uns des autres, et être développés par des équipes différentes dans des *repositories* différents
- Les micro-services peuvent être dimensionnés indépendamment

  → Le domaine *produit* de notre site e-commerce sera plus sollicité que le domaine *paiement*, il sera possible d'affecter plus ressources à celui-ci sans en dépenser inutilement pour les services faiblement sollicités

4. Dimensionnement & considérations relatives

Historiquement, la construction d'application monolithique et dont l'état est géré en mémoire (*stateful*) nécessitait avec l'augmentation de la charge des serveurs plus puissants.

Il s'agissait donc d'augmenter

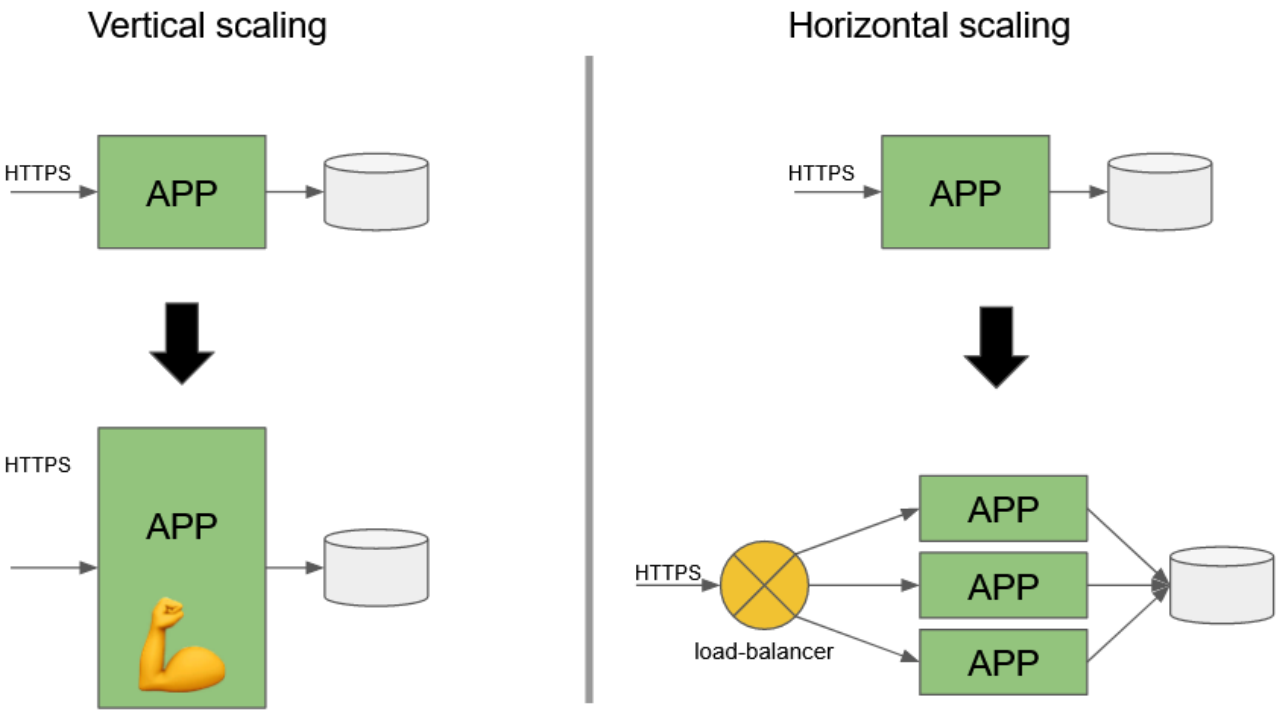
- RAM (mémoire)
- Fréquence CPU
- Nombre de coeur CPU
- Taille du disque dur

Dans ce cas, on parle de **dimensionnement vertical**.
Il se trouve que le coût du matériel n'est pas proportionnel à la puissance, mais plutôt exponentiel.
Ce type de dimensionnement (ou *scaling*) arrive donc rapidement à une limite technique (puissance maximale des CPUs à une époque, etc.) ou financière.

Certains systèmes, comme les bases de données SQL traditionnelles (Oracle, PostgreSQL, MariaDB, etc.) ne peuvent augmenter leurs capacités que de cette manière.

À l'inverse, les applications peuvent être structurées de manière à pouvoir être dimensionnées différemment.

On parle dans ce cas de **dimensionnement horizontal**, qui consiste à augmenter le nombre de conteneurs / vms / serveurs pour augmenter de manière linéaire le nombre d'appels que le système peut traiter.



Un autre aspect négatif de l'approche instance unique est que tout changement nécessitant un redémarrage de l'application (configuration, nouvelle version, etc.) interrompt le service rendu.

Dans le cas où il y a plusieurs instances d'une même application, que ce soit derrière un **répartiteur de charge** (*load-balancer*) ou qu'elles consomment les messages d'un broker, en éteindre une ne coupe pas le service.

Il est tout à fait possible de configurer le **répartiteur de charge** afin de ne plus distribuer de connexion vers une instance et de laisser les connexions existantes se terminer afin de pour voir éteindre l'instance sans impact pour les utilisateurs.

Traiter toutes les instances d'une même application comme celà pour réaliser une mise à jour est appelé **rolling update**.

Attention toutefois à ce que les systèmes connectés à l'application (clients d'une API, base de données, etc.) puissent fonctionner avec les deux versions qui vont cohabiter durant la mise la mise à jour.

4.1. Identifier les SPOFs

Les SPOFs (**S**ingle **P**oint **O**f **F**ailure) sont les parties d'un système qui ne peuvent pas avoir plus d'une instance et qui par conséquent entraine une coupure de service en cas d'arrêt (crash, mise à jour, etc.).

Il est important de les identifier et de monitorer leur état pour pouvoir réagir en cas de problème.

Cette réaction peut être automatique (bascule sur un réplica jusque-là passif) ou manuelle (démarrage de services alternatifs).

Tenir compte des SPOFs peut également influer sur l'architecture du système pour limiter au strict minimum la charge qui y est envoyée, ou metre en place des mécanismes pour alléger cette charge (un cache de données par exemple).

Ces considérations sont applicables à tout système dont on voudrait diminuer l'usage (une API extérieure payante par exemple) et rejoigne le méta-objectif du découplage (diminuer les dépendances entres composants).

Les bases de données (SQL) sont également souvent des SPOFs ; il est cependant possible dans la plupart des cas de mettre en place des réplicas pouvant remplacer la base principale en cas d'indisponibilité.

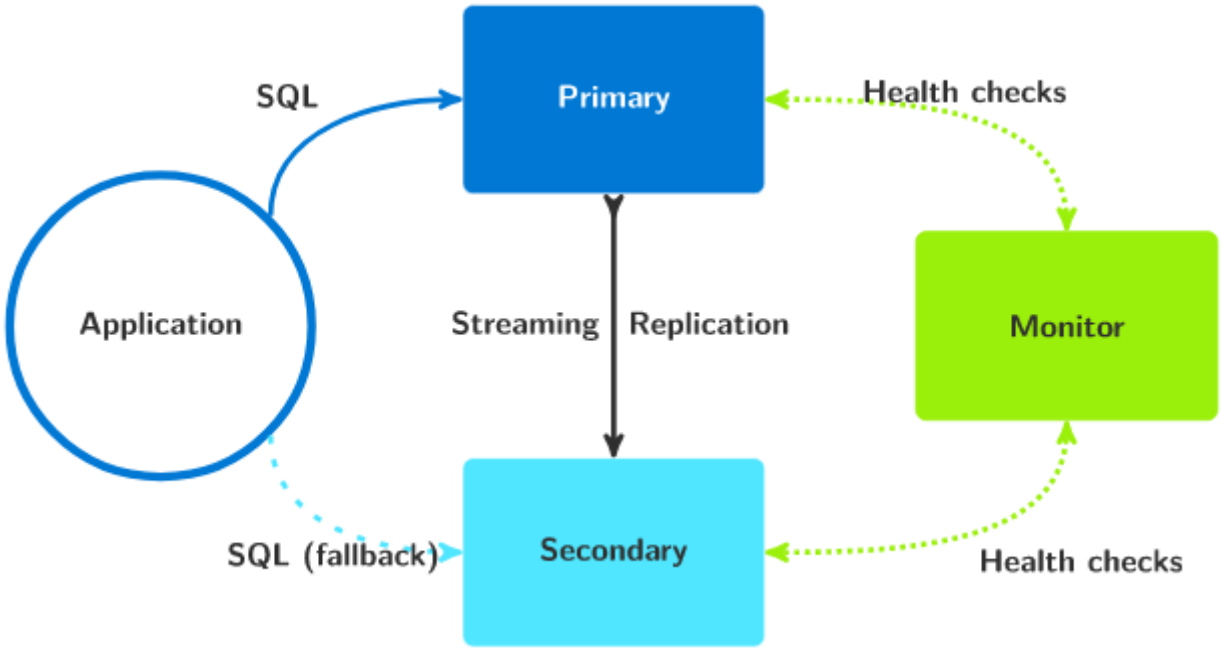


Figure 2. Mécanisme de bascule pour PostgreSQL (`pg_auto_failover`)

4.2. Identifier les points de contention

Quand un système est soumis à de fortes charges, des points de contention (ou goulot d'étranglement) peuvent apparaître.

Il s'agit de partie de système dont la vitesse de traitement est plus lente et qui constitue la vitesse maximale du système dans son ensemble.

NOTE	À l'instar d'un système d'écoulement d'eaux pluviales, si un des tuyaux a un faible débit, c'est tout le système qui est plus lent, et qui risque l'engorgement

Ces points de contention peuvent provoquer ds réactions en chaîne ou en cacher d'autres.

Pour les identifier et si nécessaire les améliorer, il est nécessaire d'avoir une supervision précise des différents composants du système.
La supervision est constituée entre autres, d'un ensemble de sondes qui permettent de relever des données à interval régulier et d'en faire des courbes.
Ces courbes témoignent de l'évolution du système dans le temps et permettent de faire la corrélation entre des évènements exogènes et des conséquences endogènes, notamment l'apparition de points de contention.

4.3. Répartir la charge

4.3.1. Pour les communications synchrones

Pour distribuer la charge sur plusieurs instances d'un même service, on utilise des répartiteurs de charge qu'ils soient matériel (F5, Altéon, etc.) ou logiciel (HAProxy, Nginx, Traefik, etc.).

Il existe différentes configurations possibles, notamment pour affecter des poids aux nœuds cibles

ou établir une affinité de session (*sticky session*).

Les répartiteurs de charge logiciel sont plus simples à configurer et peuvent même être pilotés par une API.
Cela permet notamment d'automatiser des opérations telles que le **rolling update** précédemment vu.

4.3.2. Pour les communications asynchrones

Pour ce qui est des communications asynchrones, on va tendre le plus possible vers l'utilisation d'un *broker de message* qui fera le tampon entre les applications communicantes.

Un certain nombre d'outils peuvent être mis en place pour rendre un broker *hautement disponible* (**H**igh **A**vailability) et capable d'ingérer de grosses volumétries.

Ce faisant, le broker pourra accueillir les éventuels coups de béliers (très grosses charges temporaires) sans redistribuer la pression aux applications clientes.

En effet, quand deux systèmes communiquent, il est préférable de fonctionner en *flux tiré* de sorte que l'application cliente consomme quand elle le veut et au rythme ou elle le peut les messages qui lui sont destinés.

La répartition de charge se fait naturellement avec l'augmentation du nombre d'instances de l'application cliente (pour Kafka, au sein d'un même **client-group**).

Il faut cependant être vigilant à la stratégie de livraison de message :

- **at most once** : un message est délivré au plus une fois. Par conséquent, il peut y avoir des pertes de message (cas marginal).
  - Cas d'usage : notifications, données peu importantes
- **at least once** : un message est délivré au moins une fois. Par conséquent, il peut être délivré plusieurs fois (cas marginal) et l'application cliente doit être capable de dé-dupliquer ces messages
  - Cas d'usage : tout le reste, du moment qu'on ne peut pas se permettre de perdre un message (commande client, transaction bancaire, changement d'une fiche patient, etc.)

4.4. Exemple d'évolution d'architecture

Considérant une application *classique* comme celle-ci :

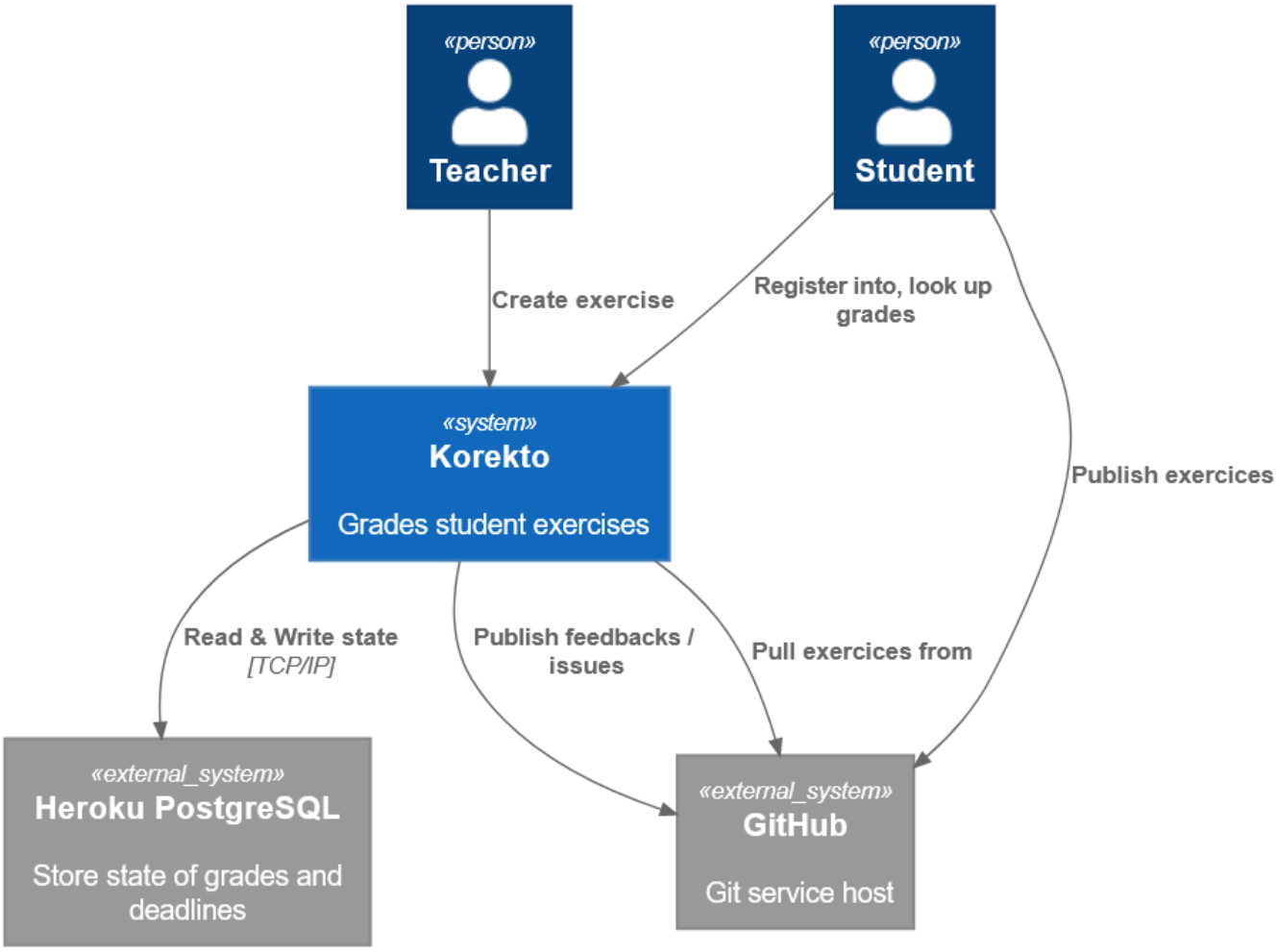


Figure 3. Diagramme de Contexte

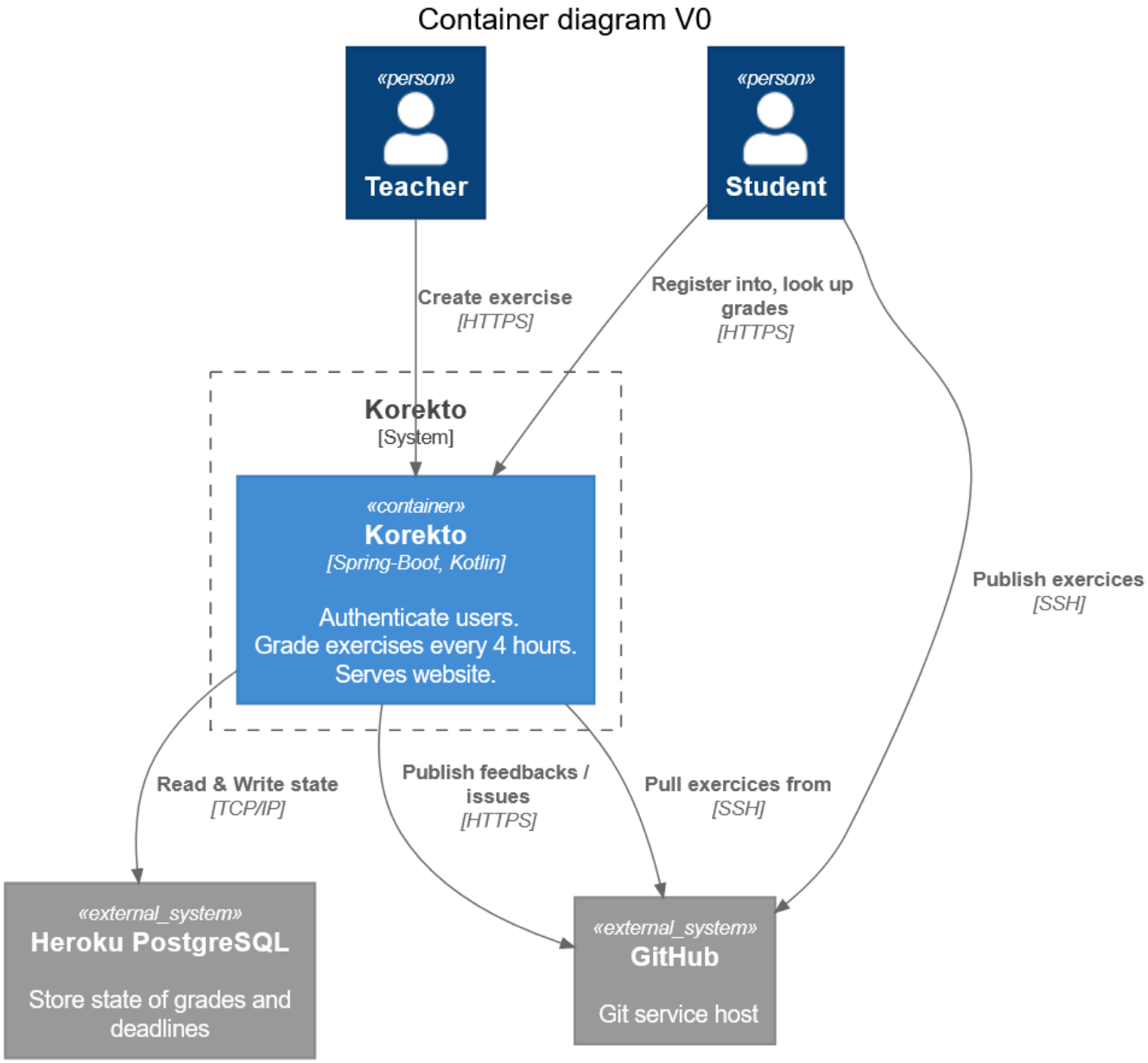


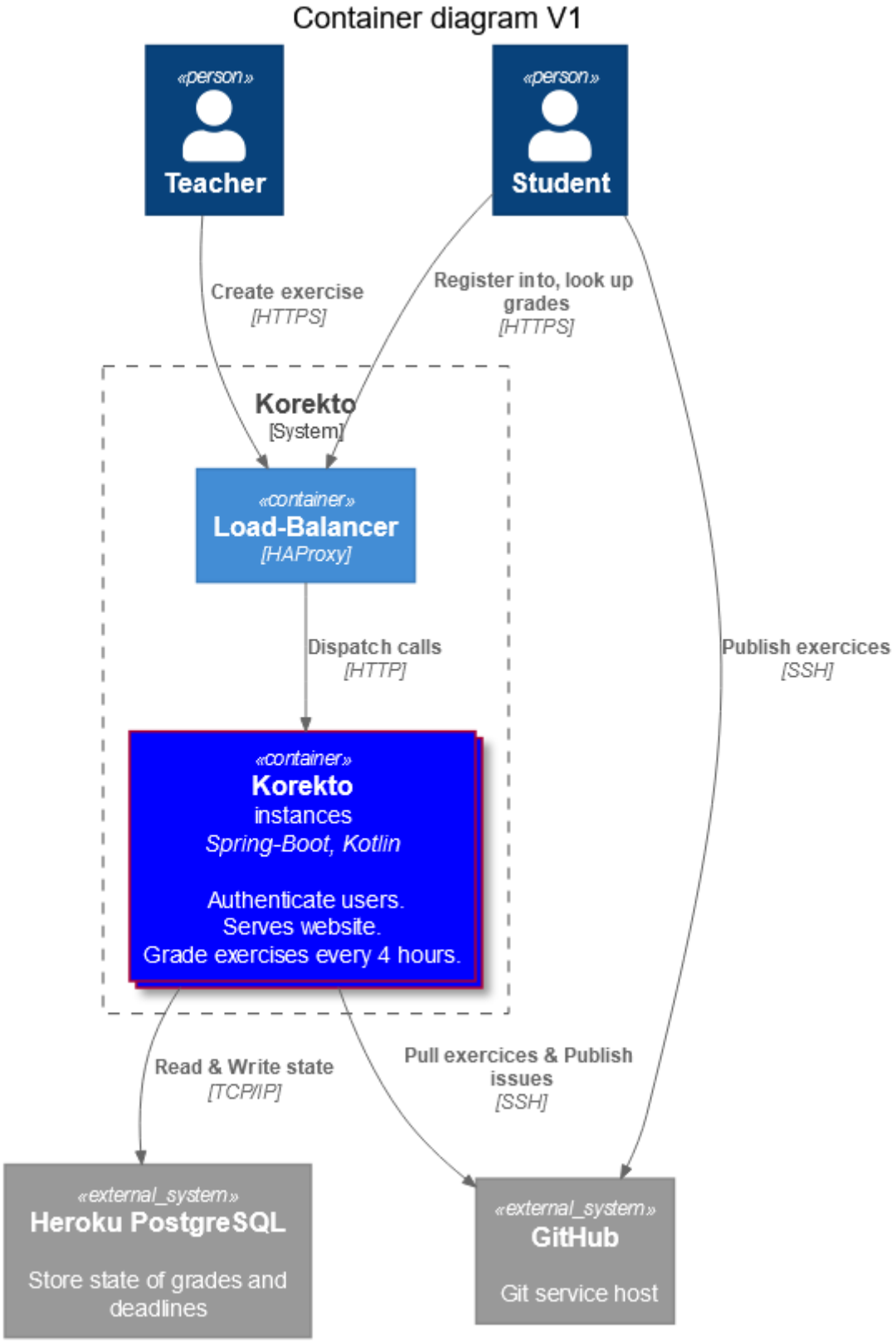
Figure 4. Diagramme de Container

4.4.1. Introduction d'un répartiteur de charge

Le dimensionnement horizontal, c'est la capacité du système à répartir la charge sur plusieurs instances (ou nœuds).
Mais du point de vue des utilisateurs, il n'y a qu'un seul service (https://korekto.io par exemple).
Pour faire cela il est nécessaire d'introduire un répartiteur de charge.

Attention ici, la partie qui exécute une tâche toutes les 4 heures ne doit être active que sur une seule instance, sous peine de faire le même travail plusieurs fois, ce qui peut provoquer :

- Charge CPU inutile
- États incohérents en base



4.4.2. Découpage de l'applicatif

Certaines fonctions de l'application, sont exécutées plus au moins souvent, et utilisent plus ou moins de CPU.

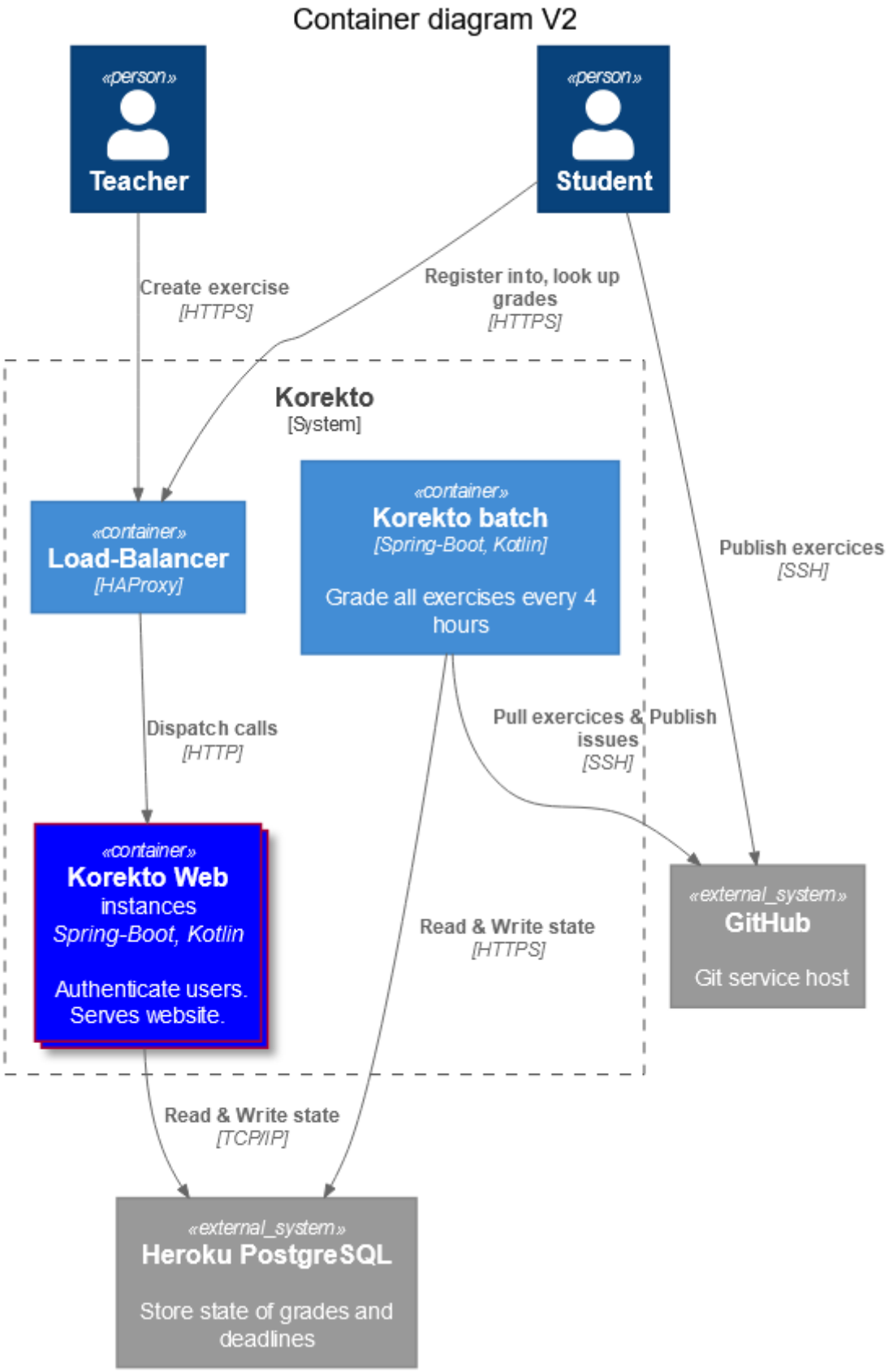
Il peut être intéressant de découper une application monolithique en plusieurs petites applications pour pouvoir dimensionner finement les fonctions les plus sollicitées.

Ce découpage doit respecter une unique règle pour être profitable :

IMPORTANT

Les applications découpées à partir d'un monolithe doivent être indépendantes d'un point de vue métier et technique.

C'est-à-dire que si une de ces applications est amenée à évoluer, elle doit pouvoir être mise à jour sans impact dans les autres applications



4.4.3. Suppression des SPOFs

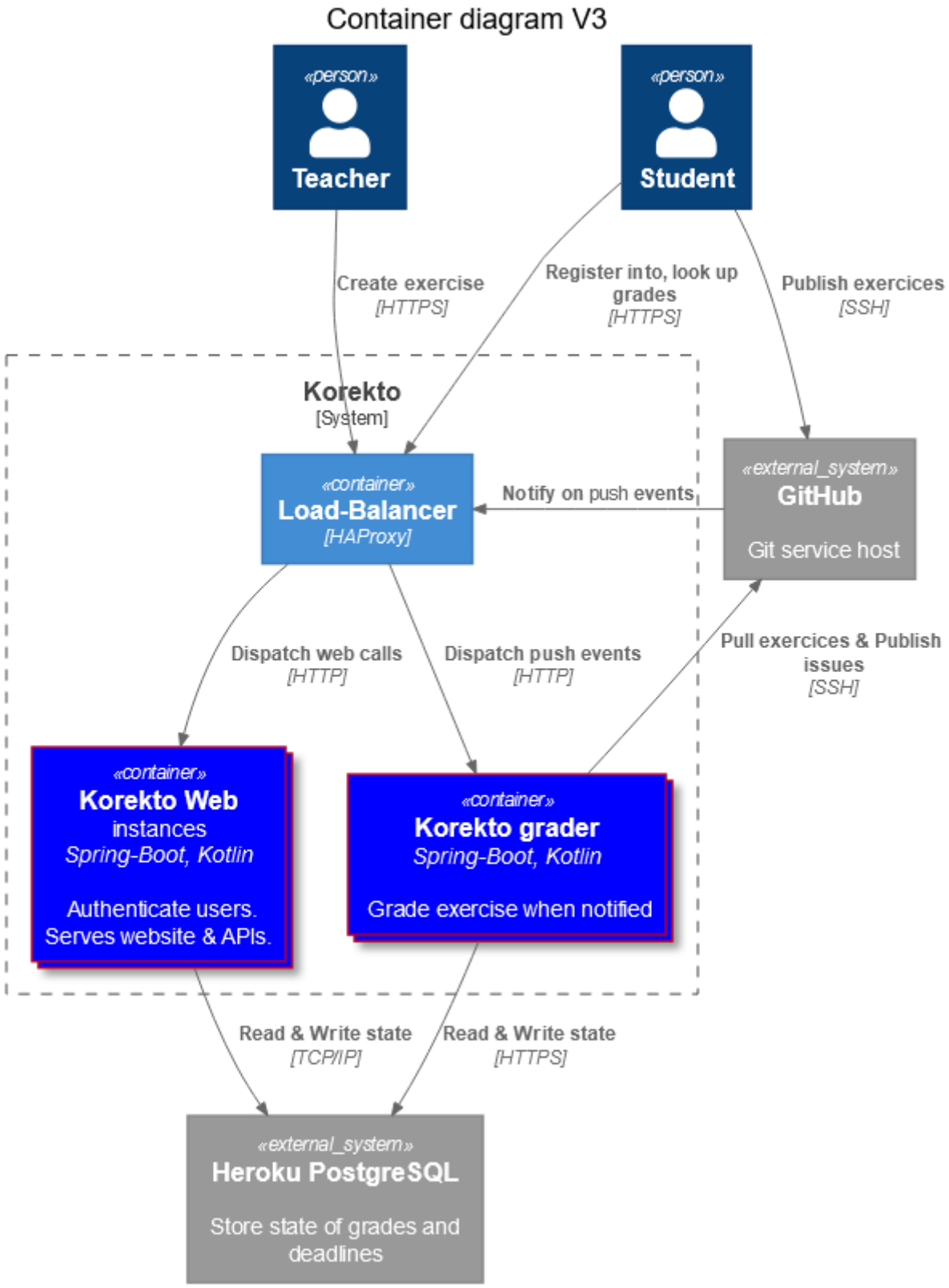
Les points de défaillance unique (**S**ingle **P**oint **O**f **F**ailure), sont des conteneurs du système qui ne peuvent pas être répliqués.
Cela pose 2 problèmes :

- Si le conteneur ne fonctionne plus, il y a une coupure de service
- Si le trafic augmente (ici le nombre d'exercices à corriger), la seule solution est par définition le dimensionnement vertical.

La cause des SPOFs est souvent un état en mémoire qui ne peut être facilement répliqué.
C'est le cas de notre conteneur **[batch]**.

En effet, si plusieurs instances devaient s'exécuter, comment pourraient-elles simplement se synchroniser pour se répartir le travail et reprendre celui des autres instances dans le cas de pannes ?

Dans notre cas, il est possible de faire plus simple en utilisant les hooks de GitHub pour être notifié en cas de changement, plutôt que de lancer une tache quoi qu'il arrive.



4.4.4. Ne pas communiquer par le stockage !

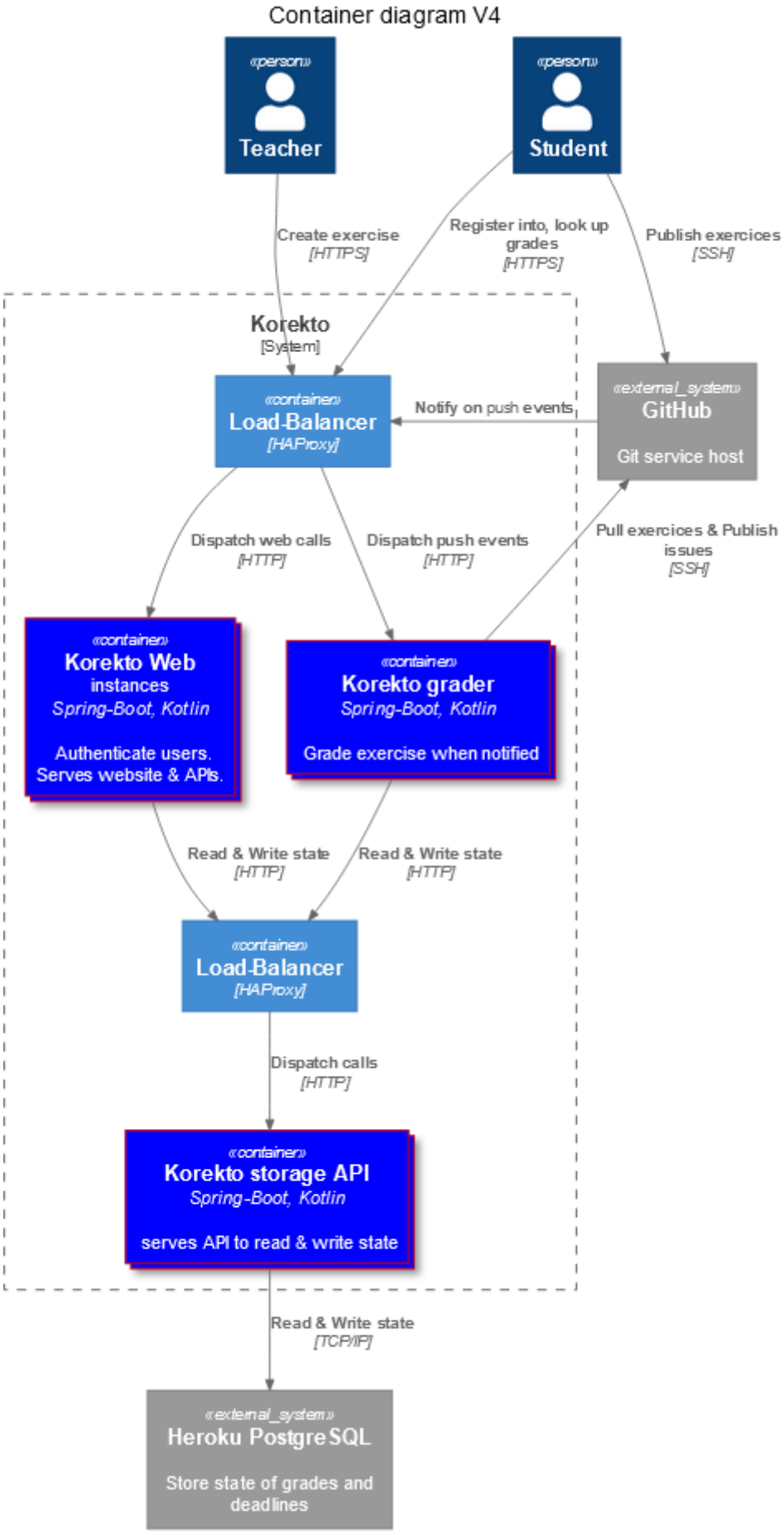
Quand nous avons découpé notre applicatif, nous avons commis une grave erreur.
En effet, le conteneur **[web]** et le conteneur **[grader]** communiquent par la base de données.

C'est un **anti-pattern** d'architecture, car :

- Comment changer la base par une autre sans impacter les 2 conteneurs ?
- Comment changer la façon de stocker la donnée sans impacter les 2 conteneurs ?

Bref, ces 2 conteneurs sont couplés.

Le but ici va être d'introduire un nouveau *container* qui va abstraire le stockage.



Ainsi, dans le cas où

- Une partie de la donnée doit être stockée de manière différente (dans une base de donnée relationnelle par ex)

  → la seule modification à apporter est dans le code du conteneur **[storage API]**.

- Le format de spécification d'exercice évolue pour inclure plus de données -→ [storage API] on propose une nouvelle API (GET / POST / PUT / DELETE) /api/admin/specification/exercise/${name}/v2

  → **[web]** on utilise cette nouvelle API pour que les professeurs puissent écrire des spécifications plus détaillées

  → **[grader]** on utilise cette nouvelle API pour corriger les exercices plus finement

  → **[storage API]** on supprime l'ancienne API (/api/admin/specification/exercise/${name}/v1) maintenant que plus aucun conteneur ne l'utilise

Ce genre de changement peut être apporté sans coupure de service.

5. Documenter l’architecture

Historiquement, la documentation d’un projet informatique était très verbeuse (Cahier des charges techniques, **D**ossier d’**A**rchitecture **T**ransverse, etc.).

Or, un projet informatique évolue vite, plus vite que ces documents.
Cette "documentation" devient alors obsolète et amène des incompréhensions ou des bugs car certaines informations deviennent fausses avec le temps.

Documenter un projet est cependant nécessaire pour transmettre la connaissance entre les différents individus qui vont s’y succéder.

L’objectif est donc d’écrire le minimum vital de documentation en éliminant le superflu et de raccrocher le plus possible cette documentation au code.

On parle alors de documentation vivante, si celle-ci évolue comme le code, et notamment si des parties sont générées à partir du code.

5.1. README

Le fichier README, écrit en Markdown ou en AsciiDoc (comme ce cours) se trouve à la racine du projet / dépôt.
Il s’agit de la porte d’entrée pour un projet inconnu.

Il renseigne sur les questions suivantes :

- À quoi sert le projet ?
- Comment utiliser le projet (outil de construction, etc.)
- Comment contacter et travailler avec les autres personnes sur ce projet

Très populaire dans le monde de l’open-source cette pratique et également importante en entreprise dès lors que de nombreux projets cohabitent dans un même système d’information et que différentes équipes doivent travailler ensembles.

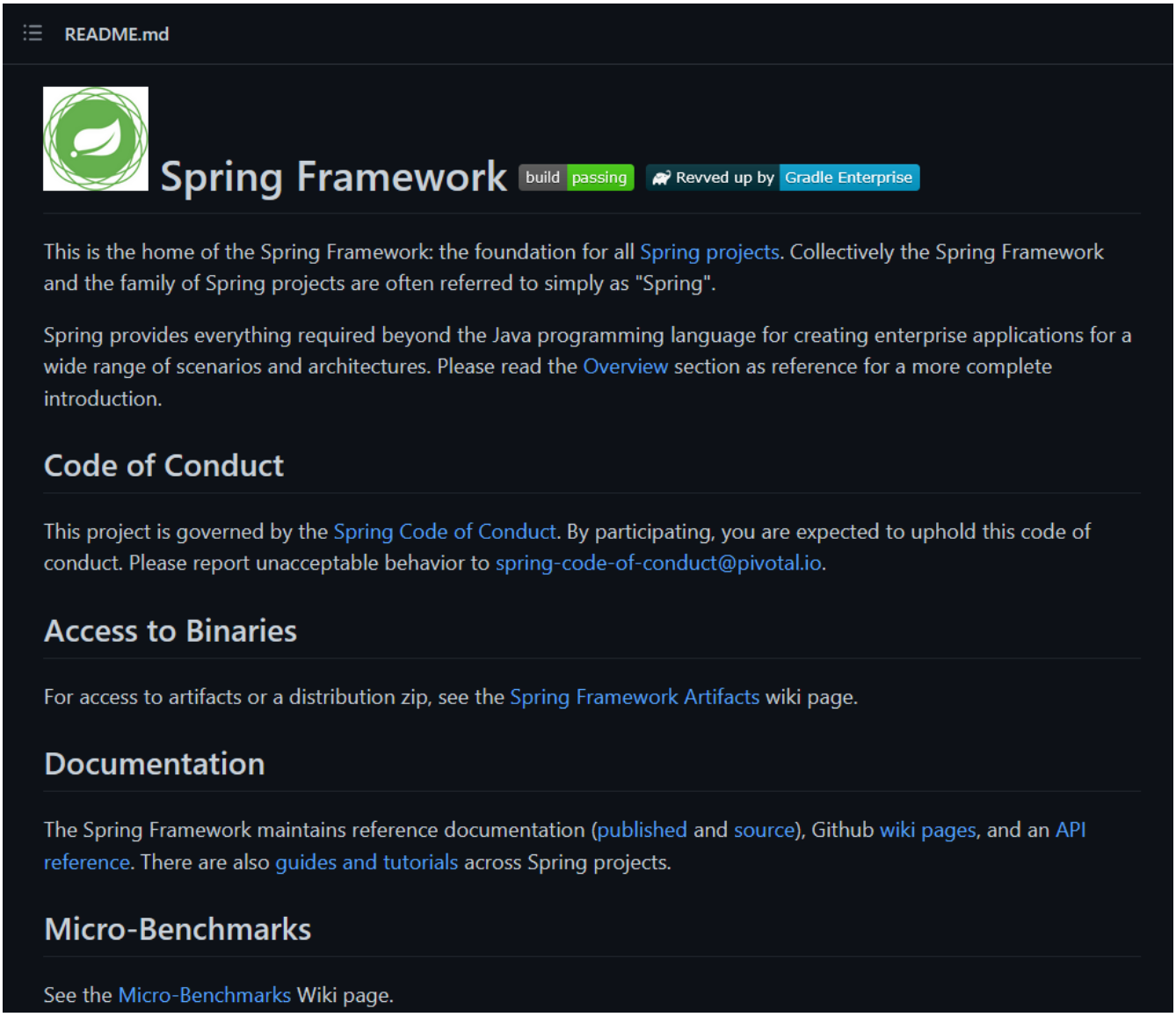


Figure 5. Fichier README du projet Spring

5.2. C4

C4 est une approche pour concevoir un système informatique sans en compliquer inutilement l'architecture.

Pour ce faire, il est nécessaire d'établir un ensemble de termes et d'abstractions avec lesquels tous les intervenants du projet pourront se ***comprendre***.

C'est une étape cruciale, qui permet d'éviter par la suite d'avoir des incompréhensions entre les différents corps de métier (fonctionnels, développeurs, intégrateurs, etc.).

Ce sont ces éléments qui permettront de construire les diagrammes qui seront à la base du projet.

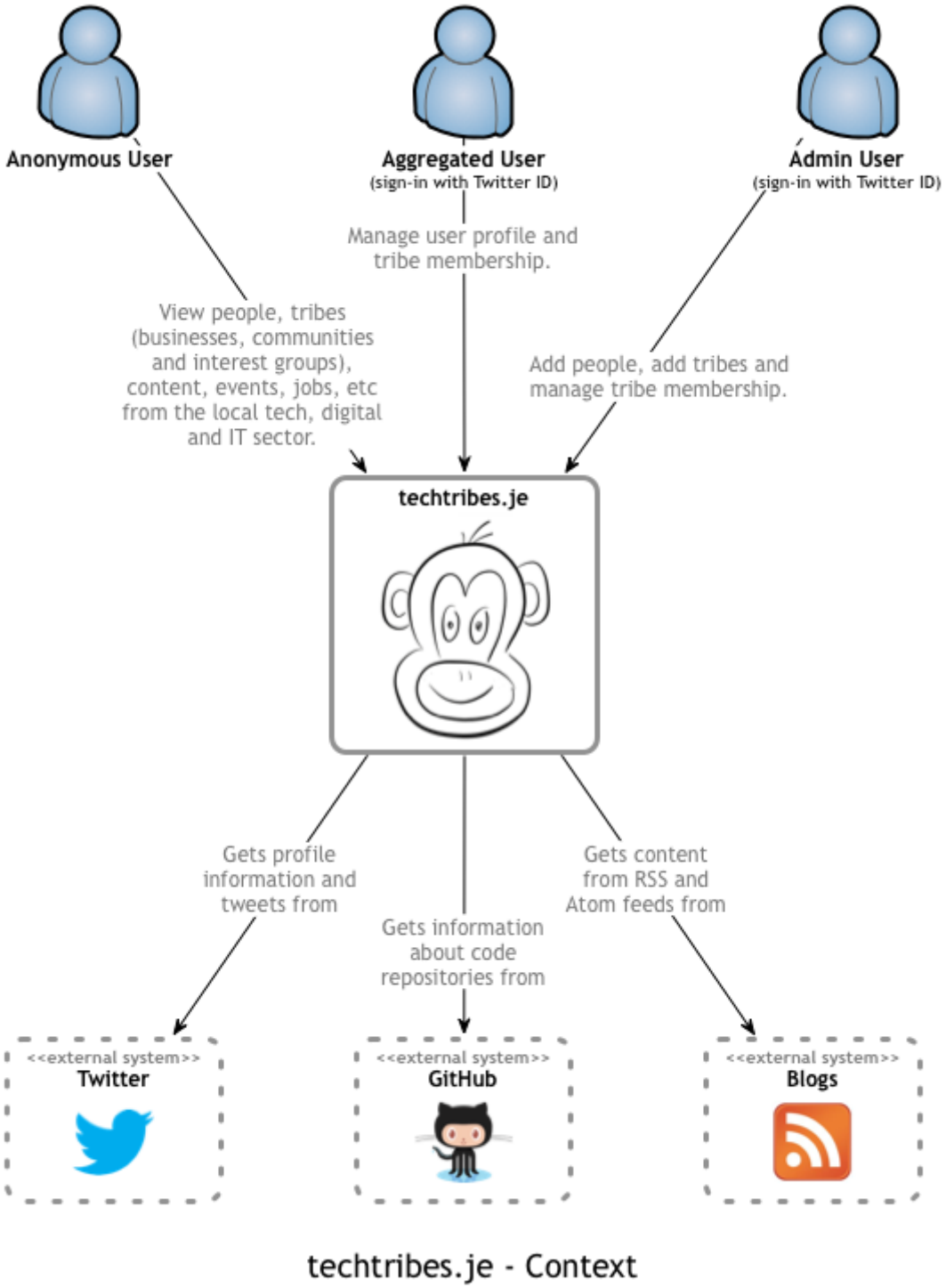
5.2.1. Contexte

Avec cette base, les intervenants peuvent construire le **diagramme de *contexte***.

Ce diagramme permet de comprendre les principales fonctionnalités du système sans afficher **comment** le système est organisé.

Le système est représenté par une unique forme (boite noire) qui sera en interaction avec des personnes ou d'autres systèmes.

Les détails (technologies, protocoles, etc.) n'ont pas d'importance dans ce diagramme, qui doit pouvoir être compris par des intervenants non-techniques.



techtribes.je - Context

5.2.2. Container

Une fois les fonctionnalités du systèmes acquises, on peut *zoome*} d'un cran avec le **diagramme de *conteneurs***.

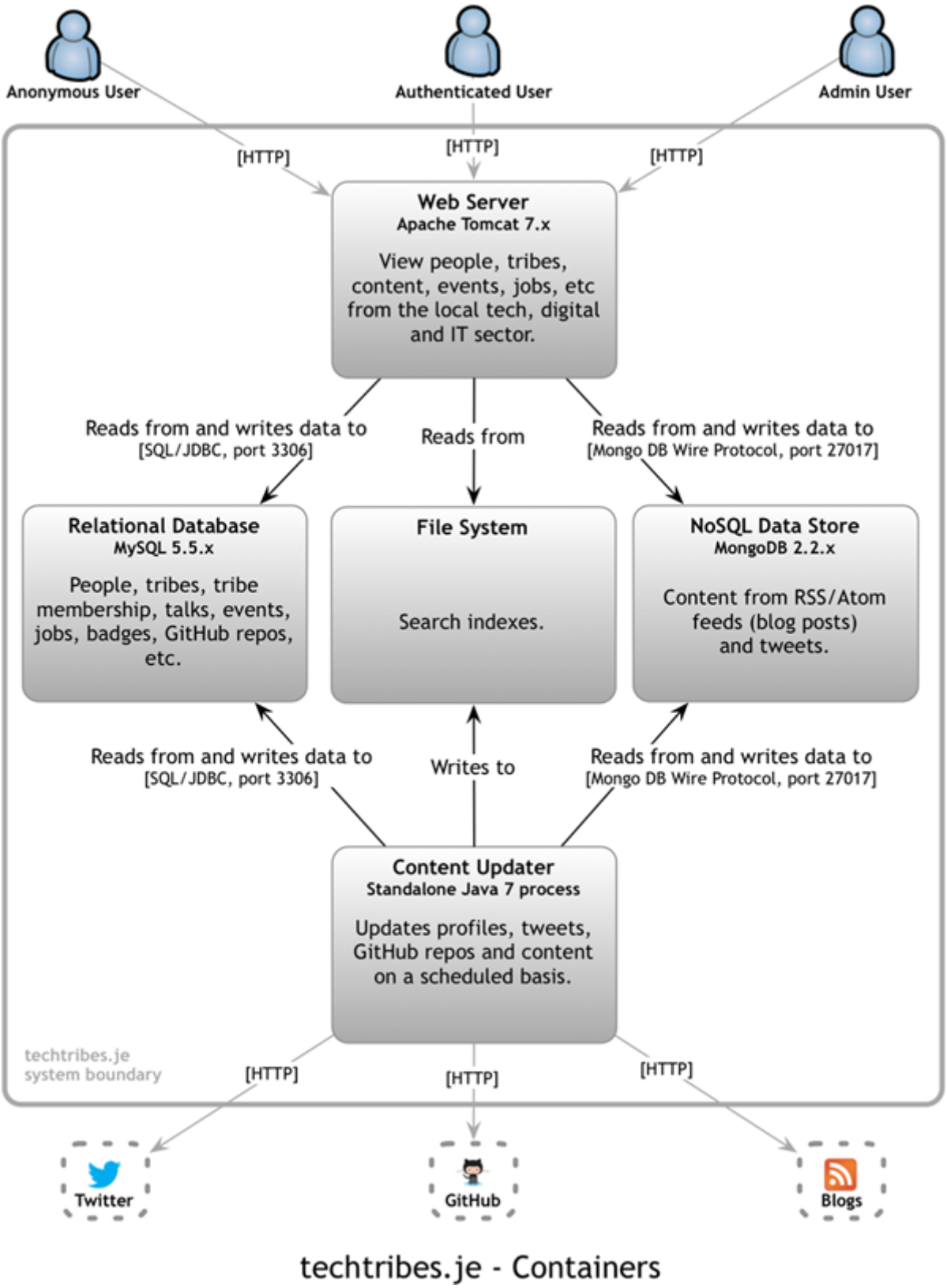
On entend par *conteneur* tout système qui peut héberger du code ou des données (application, base

de données, broker, etc.).

Ce diagramme va refléter l'architecture du système et permettre de visualiser comment les responsabilités sont réparties entre les différentes briques logicielles.

Ici peuvent être représentés

- **Sécurité** : zones réseau, flux chiffrés, traçabilité, etc.
- **Scalabilité** : load-balancers, (SPOF), etc.
- **Fiabilité** : sauvegarde, réplication, etc
- **Supervision** : sondes, bases, IHM, etc.



techtribes.je - Containers

5.2.3. Composant

On peut à nouveau *zoomer* sur chacun des conteneurs grâce au **diagramme de *composants***.

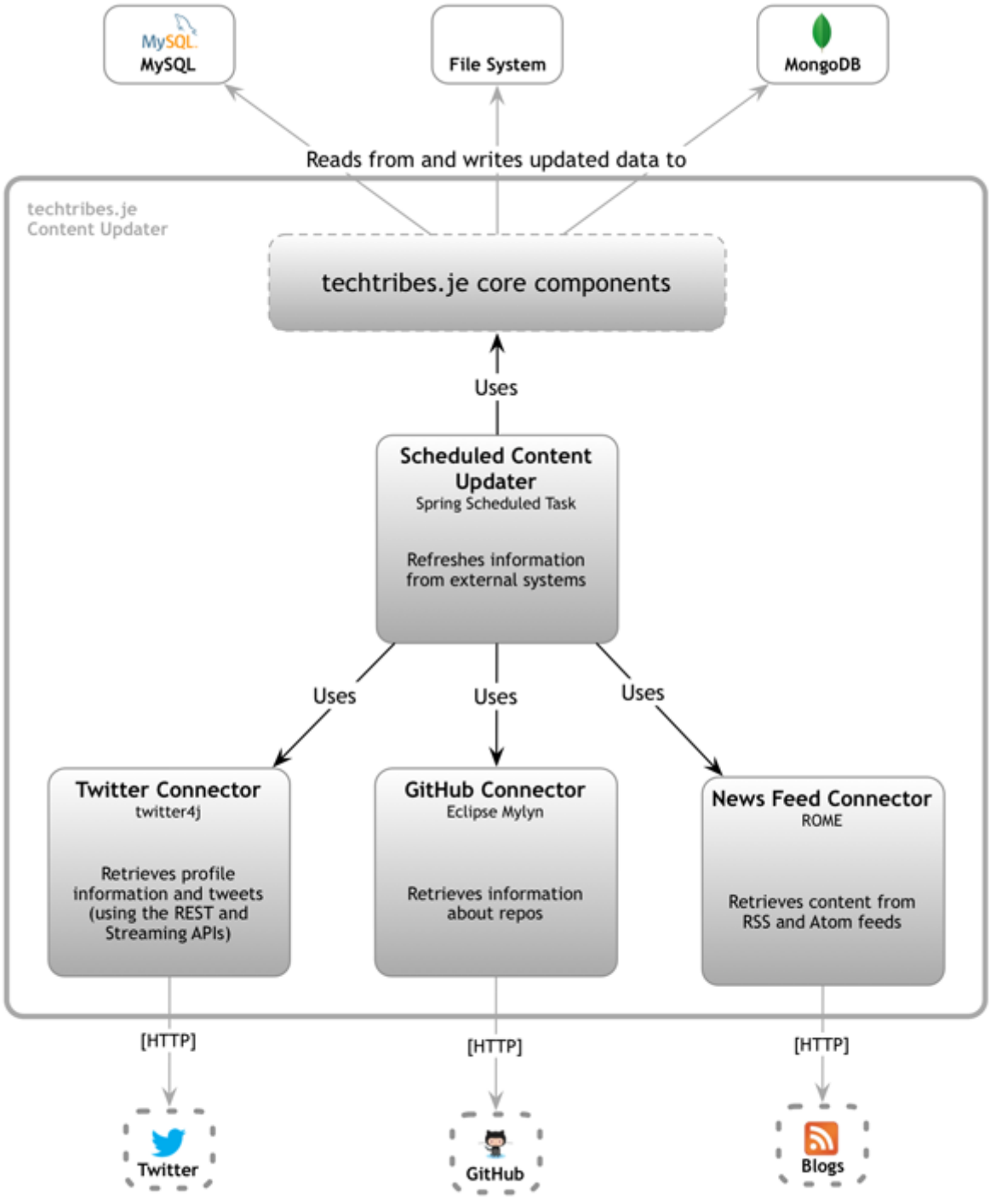
Ce diagramme détaille ce dont est composé un conteneur, avec un découpage possible par

- Fonctionnalité transverse
- Service métier

- Workflow

Un composant a donc une identité logique, une responsabilité et interagit avec d'autres composants.

Sont habituellement consignés les détails d'implémentation et choix technologiques car ce schéma peut être la documentation la plus détaillée d'un projet.



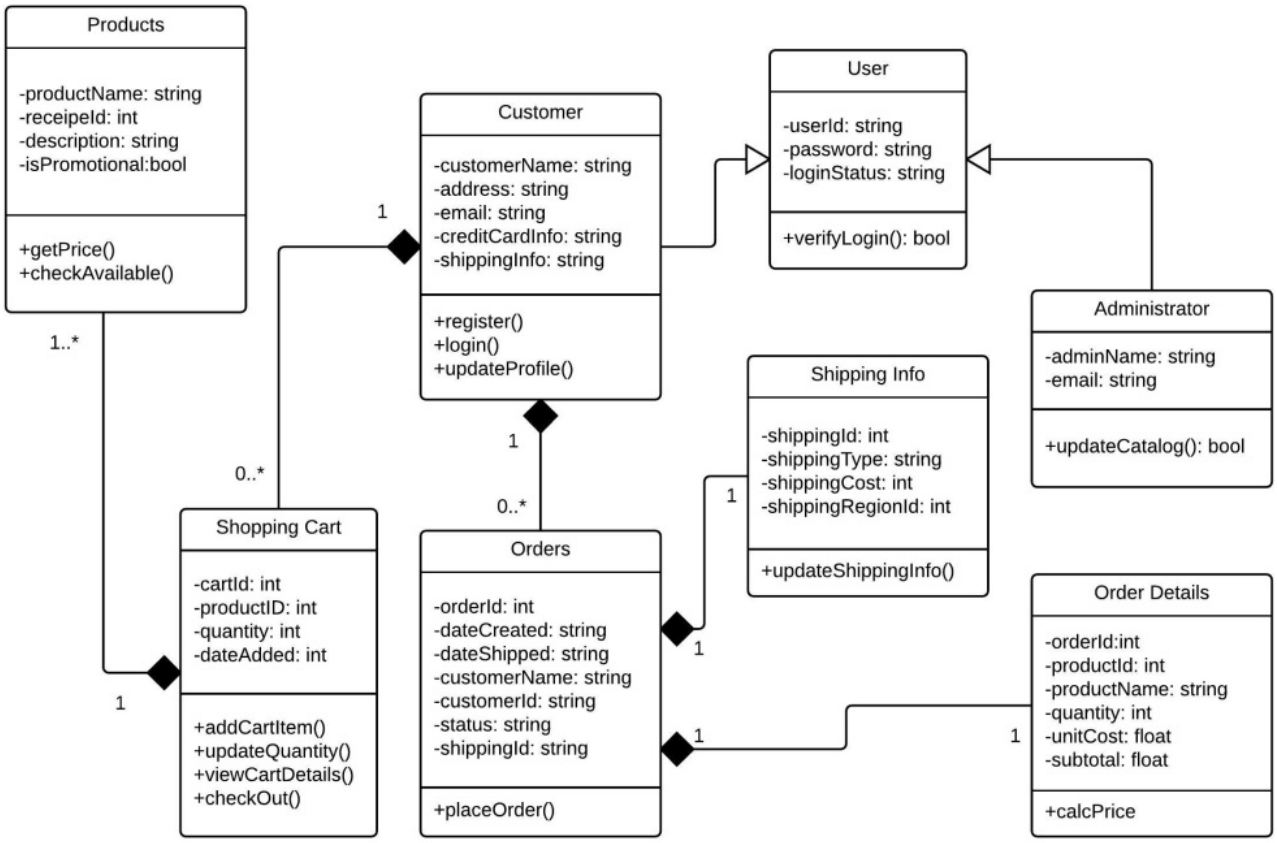
techtribes.je - Components - Content Updater
Standalone Java Process

5.2.4. Classe

Si nécessaire, il est possible de descendre au classique **diagramme UML de *classes***.

Ce diagramme est rarement utile si les besoins sont clairs, et les responsabilités bien réparties.

Cependant, pour modéliser un comportement complexe, ce schéma peut mettre en lumière une interaction complexe entre plusieurs objets.



5.3. ADR (Architectural Decision Records)

L'**ADR** est un journal dont chaque entrée concerne un choix structurant dans l'architecture d'une application.

Il est écrit par les développeurs et stocké le plus souvent à côté du code, dans le même dépôt.

La lecture de ce journal permet de comprendre pourquoi et comment une application est construite.

En effet dans une entrée du journal, on va retrouver

- La date
- Les individus impliqués
- Le contexte : ce qui a amené à faire ce choix (problème de sécurité, d'organisation, de performance, etc.)

- Les différentes solutions envisagées
- La décision
- Les conséquences prévues
- Éventuellement un statut, qui témoigne que la question a été posée, réfléchie, mais que finalement aucune solution n'a été retenue

5.4. Documentation des APIs

5.5. BDD

6. Techniques & bonnes pratiques

6.1. Pragmatisme : KISS, YAGNI & DRY

6.2. BDD

6.3. Logs

6.4. Supervision

Conclusion

Pour aller plus loin

- Out of the tar pit, traité sur la complexité
  https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- Blogs de
  - Martin Fowler
    https://martinfowler.com/
  - Simon Brown
    https://www.codingthearchitecture.com/
  - Jessie Frazelle
    https://blog.jessfraz.com/