Java 201 - Architecture logicielle

Loïc Ledoyen

Java 201

Introduction

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'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. Rebase

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

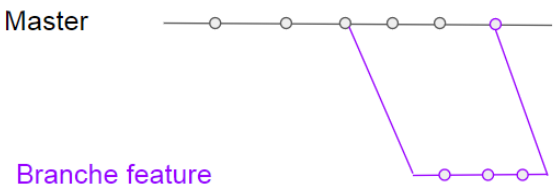
Le **rebase** peut être utilisé pour remettre à jour une branche quand la branche d'origine a changé.

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

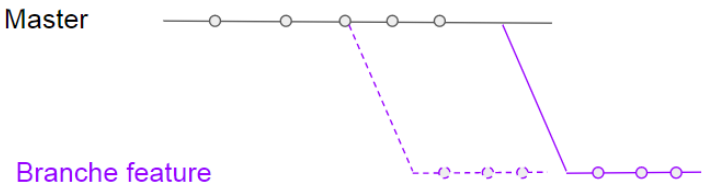
① Récupère la base de donnée du remote par défaut (**origin**) pour toutes les branches

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`

1.1.2. 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

1.1.3. 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. 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.).

1.2.1. 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

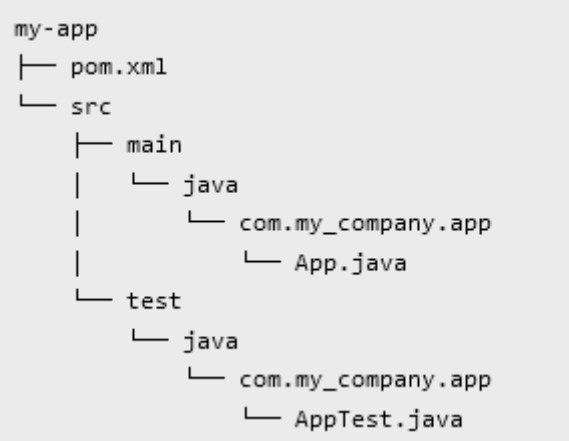


Figure 1. Structure d'un projet Maven

1.2.2. 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 ou de le désactiver en ligne de commande, par exemple : `mvn install -P disable-tests`

1.3. Junit

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/

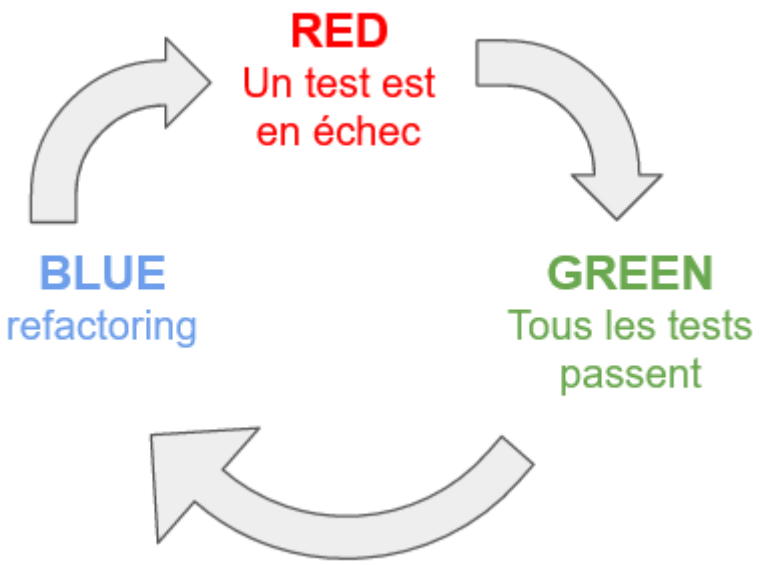
La plupart des couplages peuvent être évités en utilisant le code produit en même temps qu'il est créé.
La technique la plus simple est de suivre les principes du **TDD** (Test Driven Development) ou développement piloté par les tests.

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.

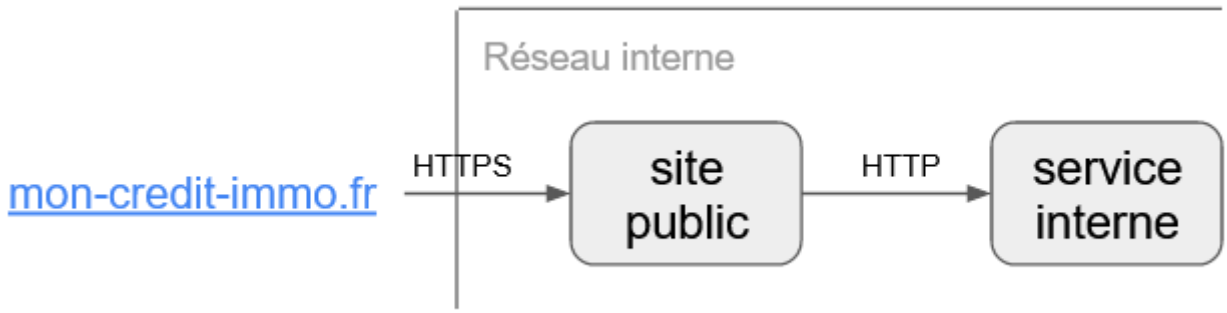
Les équipes qui travaillent sur des applications différentes sont constituées d'hommes et de femmes qui ont des façons de travailler différentes (agile, cycle en V, etc.), des rythmes différents (décalage horaire, temps partiel, etc.), des façons d'opérer la production différentes (avec ou sans coupure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



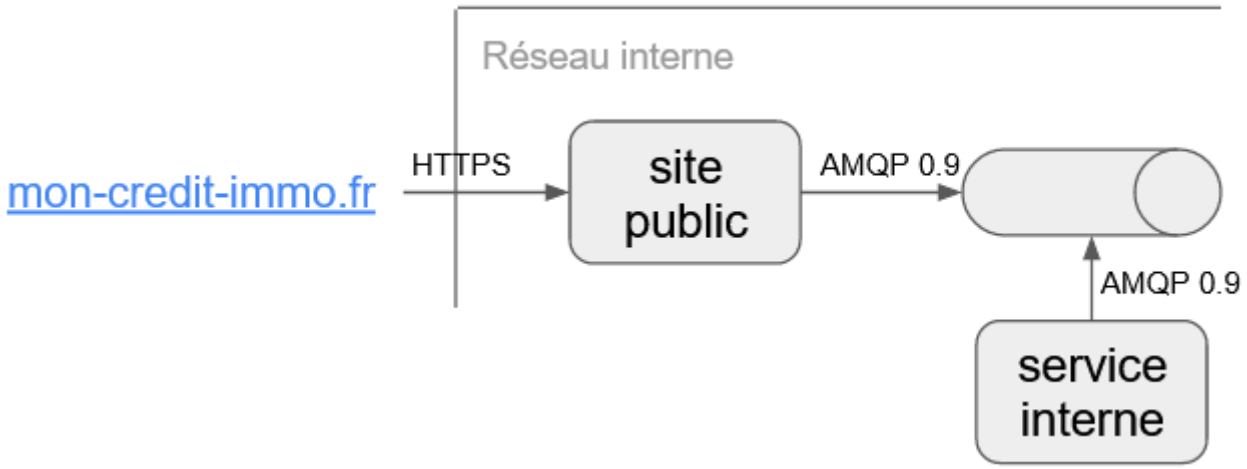
Le site public recueille les demandes de simulation et les envoient à un service interne qui doit :

- 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.

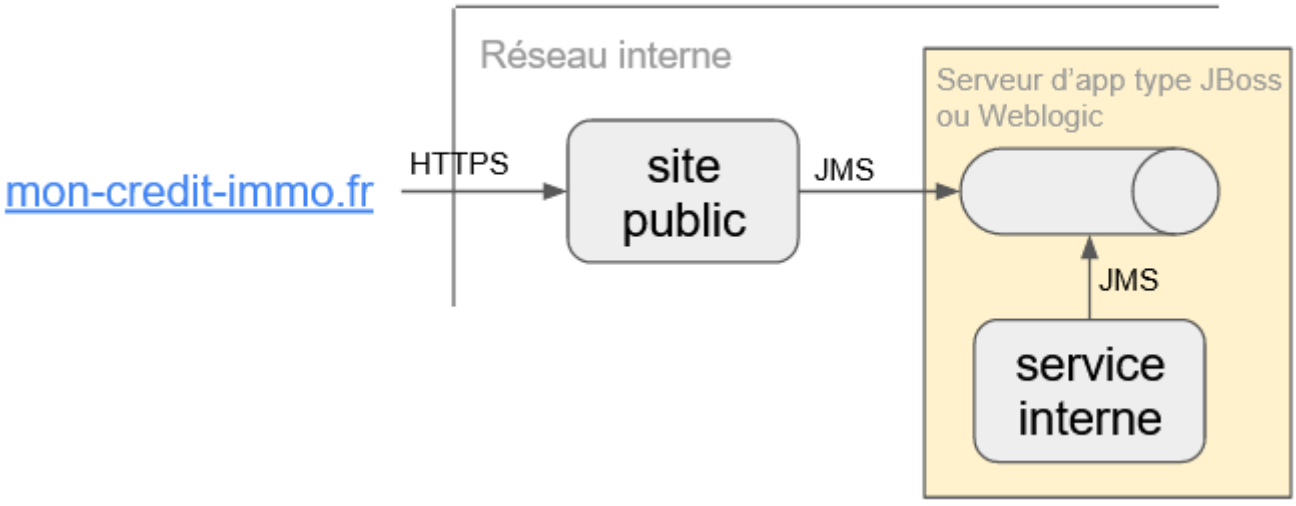
Cependant si la communication entre le site public et le service interne est synchrone, une mise à jour de ce dernier entraînerait *de facto* une coupure de service du site public ou une perte d'information.

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.

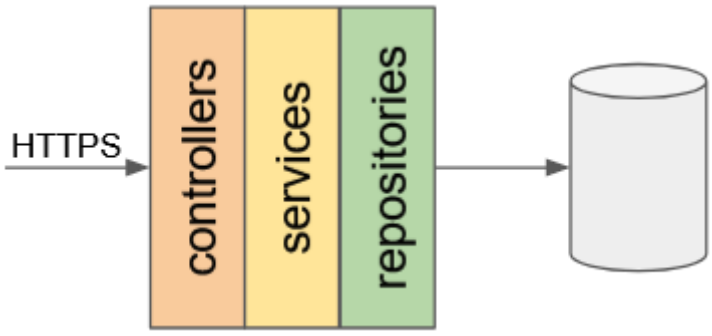
3.1. N-tiers

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

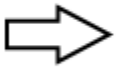
métier dans la couche service



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

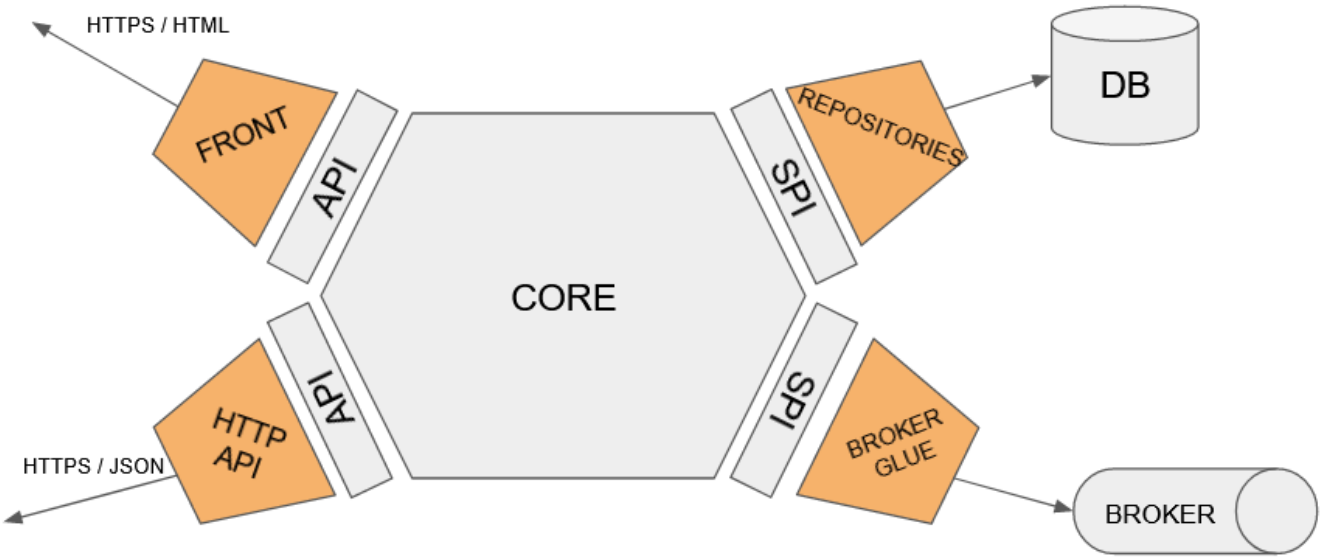
L'**architecture hexagonale** vise à séparer le code *métier* du code *technique*.

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.

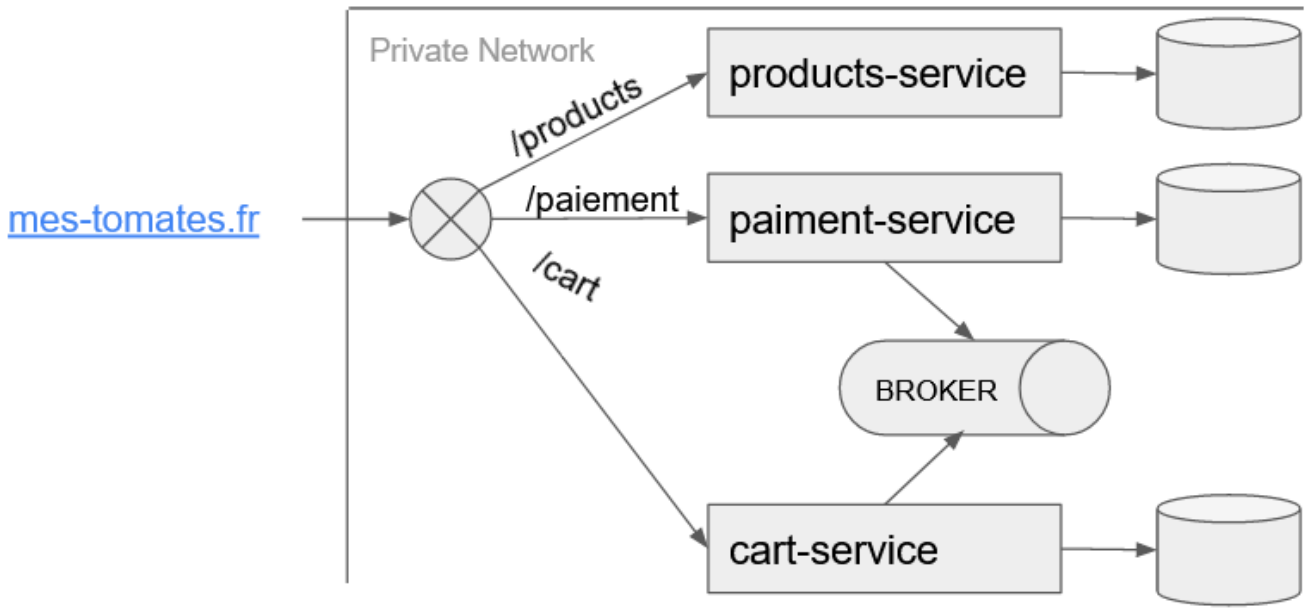
Le risque dans le cas où une telle structure n'est pas mis en place est de voir la complexité de l'application augmenter de manière exponentielle à chaque ajout de fonctionnalité.
On parle dans ce cas de code *spaghetti*.
La métaphore des *spaghettis* vient des morceaux de code entrelacés, mélangés, 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 & SPOF

5. Documenter l’architecture

5.1. README

5.2. C4

5.3. ADR (Architectural Decision Records)

5.4. (OpenAPI, RAML, etc.)

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/