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

1.2. Maven

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 1. 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 2. 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 3. 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 4. 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 5. 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/

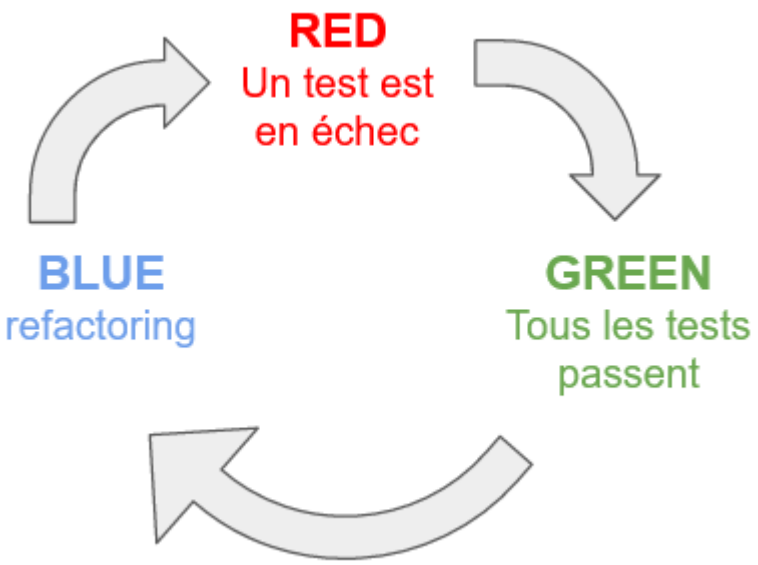
La plupart des couplages peuvent être évités en utilisant le code produit en même temps qu'il est créé.
La technique la plus simple est de suivre les principes du **TDD** (Test Driven Development) ou développement piloté par les tests.

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.

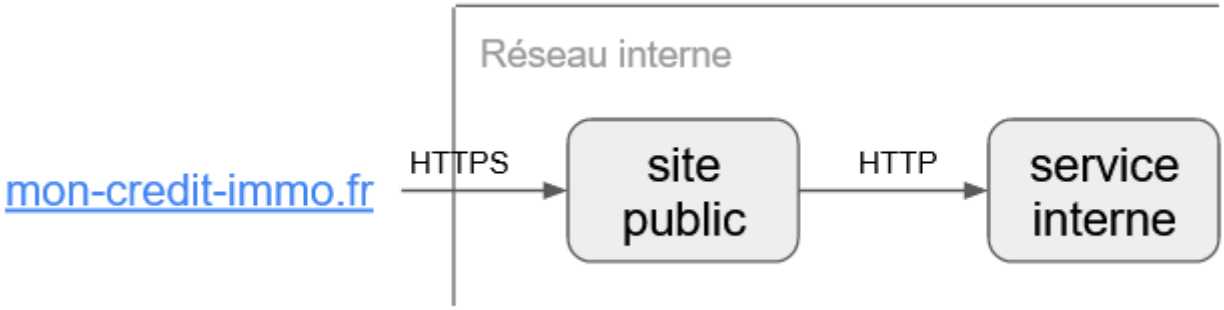
Les équipes qui travaillent sur des applications différentes sont constituées d'hommes et de femmes qui ont des façons de travailler différentes (agile, cycle en V, etc.), des rythmes différents (décalage horaire, temps partiel, etc.), des façons d'opérer la production différentes (avec ou sans coupure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



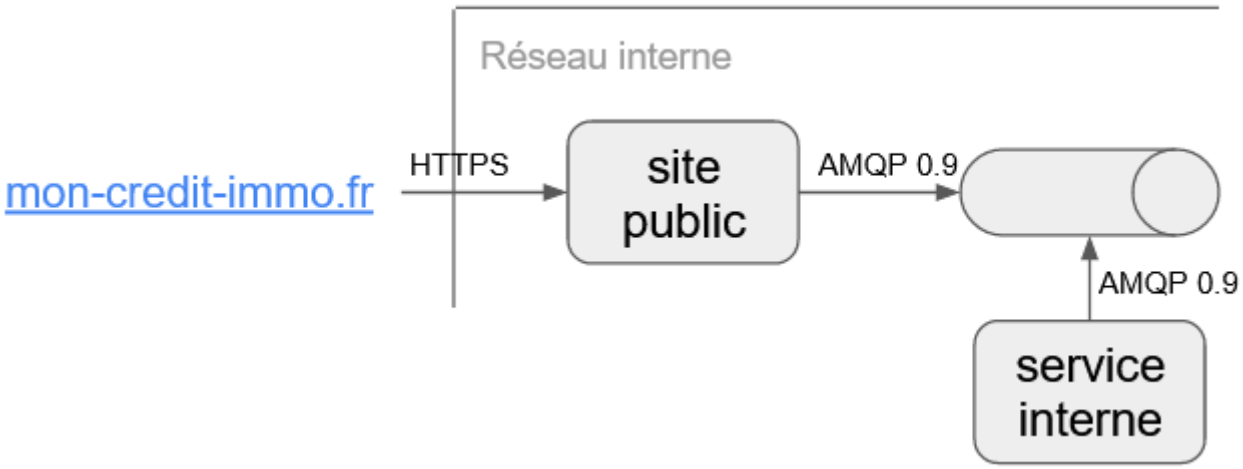
Le site public recueille les demandes de simulation et les envoient à un service interne qui doit :

- 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.

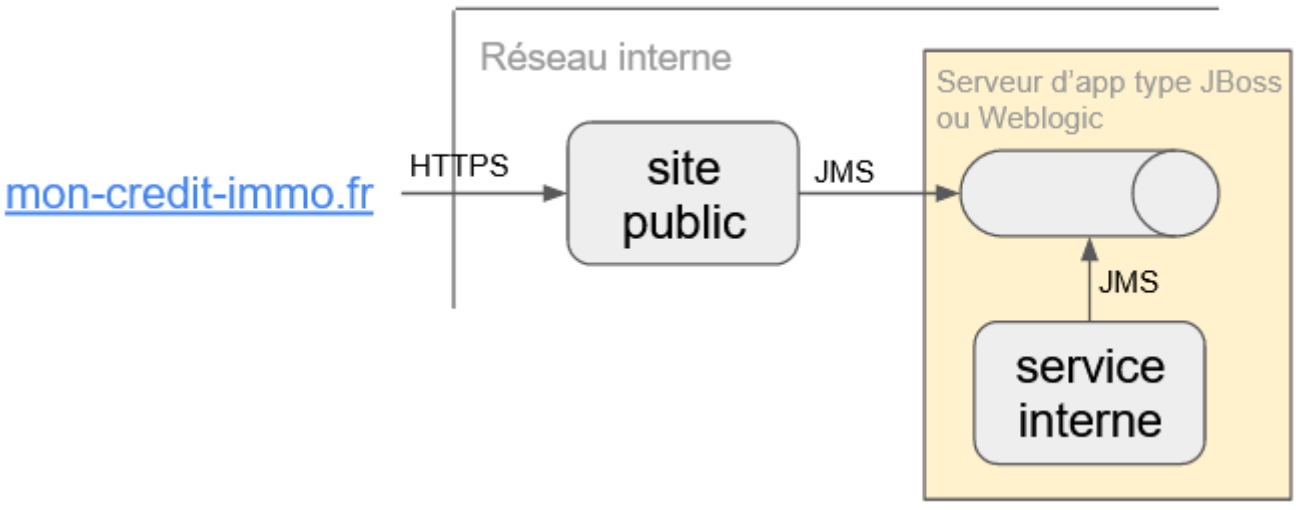
Cependant si la communication entre le site public et le service interne est synchrone, une mise à jour de ce dernier entraînerait *de facto* une coupure de service du site public ou une perte d'information.

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

Objectifs : compatibilité avec le fonctionnement de l’entreprise + diminution du risque

3.1. N-tiers

3.2. Hexagonale

3.3. Monolithique vs micro-services

4. Scaling & 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