





Ingénierie logicielle

Principes de conception

David THIBAU – 2020

david.thibau@gmail.com



Agenda

- Introduction

- Software Design ? Engineering ?
- Concepts de la conception
- Principes et recommandations
- Qualité du design, qualité du code

- Fondamentaux du développement

- Typologie des langages
- OOP et Design patterns
- Frameworks, IoC et AOP
- Programmation réactive

- Programmation distribuée

- Apports et contraintes
- Services techniques cœur
- Typologie des interactions Client/Serveur
- Modèles concurrentiels du serveur

- Architectures applicatives

- Services backend
- Message Brokers
- Architectures micro-service
- Patterns pour l'interface utilisateur
- BigData, Machine Learning



Introduction

Software design ? Engineering ?
Concepts de la conception
Principes et recommandations
Qualité du design, qualité du code



Activités de développement

Quelque soit la méthodologie (cycle en V, Agile), le développement d'un logiciel englobe différentes activités :

- Tout commence par l'**expression de besoin métier** dérivée en du fonctionnel et des contraintes techniques :
- Des activités d'**analyse** débouchant sur des choix : dimensionnement des équipes, langages, architectures, framework, patterns, ergonomie
- Puis l'implémentation, le **codage** et la vérification de l'implémentation, le **test**
- Enfin, la mise à disposition : **build** et **déploiement**



Définition

« Software Design »

Le **Software Design ou conception logicielle** est le processus par lequel un agent crée un artefact logiciel, destinée à atteindre des objectifs, à l'aide d'un ensemble de composants primitifs et soumis à des contraintes¹. (*Wikipedia*)

Autrement dit : L'activité qui fait suite aux spécifications fonctionnelles et qui est avant la programmation

1. Ralph, P. and Wand, Y. (2009) : *A proposal for a formal definition of the design concept*



Analyse des spécifications

Un des premières activité de la conception est l'**analyse du cahier des charges**.

Elle débouche sur une reformulation des contraintes selon un formalisme adapté à l'ingénierie logicielle :

- Story-boards, UX Design
- User Story
- Organigramme
- Diagrammes UML
- ...



Conception

La **conception** est la première étape de la transformation du problème en solution.

Elle met en œuvre un ensemble de principes, de concepts et de pratiques permettant à un ingénieur logiciel de modéliser le produit à construire.

Le modèle de conception résultant fournit des détails sur les structures de données logicielles, l'architecture, les interfaces et les composants nécessaires à la mise en œuvre du système.

Il est évalué en termes de qualité et revu avant le codage et les tests.



Ingénierie

L'ingénierie est l'utilisation de principes scientifiques pour concevoir et construire des machines, des structures et d'autres éléments ... (Wikipedia)

- Dans le cadre de la conception logicielle, les principes scientifiques sont loin d'être évidents, de plus les changements perpétuels des contraintes (avancées technologiques) rend instables les formalismes utilisés¹.



Outils liés à la conception



Abstraction

L'abstraction est un outil de conception puissant, qui permet de considérer les composants à un niveau abstrait en négligeant les détails de leur implémentation.

- Seules les informations pertinentes à un but particulier sont exposées.
=> Permet de contrôler la complexité
- Chaque étape du processus de développement est accomplie à travers différents niveaux d'abstraction. Du plus haut vers le plus bas



Types d'abstraction

3 types d'abstractions sont utilisées dans la conception logicielle :

- **Fonctionnelle** : Utilisation de groupes de sous-programmes ayant des routines publiques ou privées.
- **Des données** : Spécifier les données d'un objet à un certain niveau d'abstraction
- **Contrôle** : Spécifier l'effet désiré, sans préciser le mécanisme de contrôle exact.
Ex. Spécification d'un sous-programme et son gestionnaire d'exception



Architecture

L'**architecture logicielle** définit les différents composants du système, leurs propriétés et leurs relations, c'est souvent le premier choix de conception

Elle sert de schéma directeur pour le système et le projet en développement.

- Définit les sous-tâches de conception devant être exécutées.
- Fournit un aperçu du système à tous les intervenants et leur permet de communiquer

Ces choix fondamentaux sont très coûteux à changer



Design patterns

Un **design pattern (ou patron de conception)** fournit une description de la solution à un problème de conception récurrent de telle sorte que la solution puisse être réutilisée.

La description du pattern permet de déterminer:

- Si le motif peut être réutilisé et applicable au projet en cours
- Il décrit précisément le problème de conception à résoudre et les conséquences de l'utilisation du pattern.



Types de design pattern

Les design patterns peuvent être utilisés tout au long du processus de conception.

En fonction du niveau d'abstraction, on distingue différents types de pattern :

- Au niveau le plus haut des **patterns architecturaux**
- Les **design patterns**. Ils concernent en général un composant, un sous-composant ou une relation de l'architecture
- Les **idiomes** : Ce sont des patterns de bas-niveau décrivant la mise en œuvre d'un composant logiciel dans un langage de programmation spécifique.



Modularité

La **modularité** partitionne un système complexe en un ensemble de modules pouvant être développés indépendamment.

Cela permet de

- planifier le développement de manière plus efficace,
- d'effectuer des tests et du débogage de façon séparé et efficace
- d'effectuer des travaux de maintenance et d'évolution plus facilement.

Cela implique une phase d'intégration :

- Plus le nombre de modules est grand, plus l'effort est important.



Encapsulation

Les modules doivent être conçus de manière à ce que les structures de données et les détails de traitement d'un module ne soient pas accessibles aux autres modules.

Cette **encapsulation** apporte :

- Un faible couplage, limitant les effets d'une modification de composant
- Une communication via des interfaces contrôlées, diminuant la probabilité d'effets indésirables
=> Un logiciel de qualité supérieure.



Refactoring

Le **refactoring** est une activité importante de la conception

Elle peut être définie comme un processus de modification d'un composant logiciel pour améliorer sa structure interne sans changer son comportement externe.

- Réorganisation des modules
- Modification du modèle pour correspondre plus finement au réel
- ...



Concurrence

Pour utiliser les ressources efficacement, plusieurs tâches doivent être exécutées simultanément.

La **concurrency** est donc un des concepts majeurs de la conception.

- Introduit de la complexité :
synchronisation, dead-lock, verrous, ...



Modèle de conception

Pour une conception complète, 4 modèles sont nécessaires :

- **Modèle de données** : spécifie les structures de données et leurs relations. En général, la première conception.
- **Modèle architectural** : spécifie la relation entre les éléments structurels, les patrons de conception et architecturaux et les facteurs influant sur l'implémentation.
- **Modèle des composants** : Description détaillée de la manière dont les éléments structurels du logiciel seront réellement mis en œuvre.
- **Modèle de l'interface** : décrit la manière dont le logiciel communique avec l'extérieur (système ou utilisateurs finaux).



Conception et développement ?

Jusqu'où aller dans la conception ? Et en particulier dans la modélisation des composants

L'ambiguïté du terme *design* provient qu'à certains égards, le code source d'un programme est le *design* du programme.

« TEX would have been a complete failure if I had merely specified it and not participated fully in its initial implementation. »

Donald Knuth



Formalismes ?

Quel formalisme utilisé pour la conception ?

- Rien d'universel
- Doit être compris par tous les acteurs
(développeurs mais aussi quelquefois métier)
- Différents en fonction des modèle
 - Données : Modèle relationnel, Schéma XML, JSON, Flux binaire, Diagramme de classe¹
 - Architectural : Diagramme de composant¹ ou de déploiement¹
 - Composant : Diagramme composite¹, de classe¹
 - Interface : Story-board, Maquette, Flow de navigation, Use case¹, WSDL, HAL



Diagrammes UML

UML est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système.

Il propose 13 types de diagrammes différents, classés en 2 familles :

- Structuraux : Classes, Package, Objet, Composant, Structure composite, Déploiement
- Comportementaux : Activité, séquence, use case, états, communication, interaction, chronogramme



Principes et recommandations



Introduction

Les erreurs durant la phase de conception sont souvent les plus coûteuses

Certains principes ou recommandations servent de cadre à la conception permettant d'améliorer la qualité et de limiter les risques.

Cela concerne la conception et/ou le codage.



Principes de conception (1)

Correspondre au modèle d'analyse: on doit savoir comment le modèle de conception satisfait à toutes les exigences représentées par le modèle d'analyse.

Choisir le bon paradigme de programmation: Les paradigmes de programmation (langages orientés procédure, orientés objet, ...) doivent être choisis en tenant compte de contraintes de temps, de disponibilité des ressources et de la nature des besoins de l'utilisateur.

Uniforme et intégrée: La description des interfaces entre les composants sont correctement définies et de façon consistante

=> Règles, le format et les styles sont établis avant le démarrage de la conception .



Principes de conception (2)

Flexible: Doit pouvoir s'adapter facilement aux modifications.

=> Abstraction et modularité doivent être appliquées efficacement

Garantir un minimum d'erreurs conceptuelles : Doit s'assurer que les principales erreurs conceptuelles de conception, telles que l'ambiguïté et l'incohérence, sont traitées à l'avance.

Dégradation en douceur: Capacité à gérer les changements et les circonstances inhabituels sans affecter les fonctionnalités du logiciel.



Principes de conception (3)

Correspondance entre le logiciel et le problème du monde réel: La conception du logiciel doit être structurée de telle sorte qu'elle soit toujours en rapport avec le problème du monde réel.

Réutilisation : Pour augmenter la productivité. Et «ne pas réinventez la roue».

Testabilité: Les tests font partie intégrante du développement, ils déterminent si les spécifications sont satisfaites.

Prototypage: Permet d'affiner les spécifications. De valider un design .



Recommandation pour les développeurs (1)

Measure twice and cut once

Être sûr de soi => Test de ce que l'on a fait

Don't Repeat Yourself (DRY)

Pas de copié/collé

Occam's Razor

Se baser sur les hypothèses les plus simples

Keep It Simple Stupid (KISS)

Lisibilité, maintenance



Recommandation pour les développeurs (2)

You Aren't Gonna Need It (YAGNI)

N'implémentez que ce dont vous avez besoin en premier lieu et ultérieurement, si nécessaire, augmentez les fonctionnalités.

Concevoir avant de coder

Réfléchir avant d'agir

Pas d'optimisation prématurée

Principe du moindre étonnement

Le code ne doit pas surprendre



Recommandation pour les développeurs (3)

S.O.L.I.D. :

- Single responsibility
- Open/Closed : Ouvert pour l'extension, fermé pour la modification
- Lisko substitution : Les sous-classes doivent compléter mais pas remplacer le comportement des classes mères
- Interface Segregation : Aucun client ne doit être obligé de dépendre de méthodes qu'il n'utilise pas;
- Dependency inversion : Les implémentations doivent dépendre des interfaces et non des implémentations.



Recommandation pour les développeurs (4)

Law of Demeter (LoD) ou principe de moindre connaissance :

- Les classes doivent être indépendantes
- Il est nécessaire de réduire le nombre de connections ou couplage entre les classes
- Les classes en relation doivent être dans le même module (package/répertoire)



Qualité du design, qualité du code

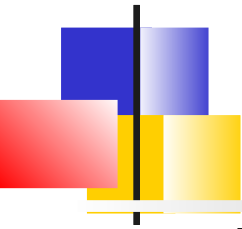


Introduction

L'approche « ingénierie » nécessite que l'on puisse évaluer la qualité du produit final, le logiciel et donc la qualité du code et de la conception.

Des efforts¹ ont été entrepris pour normaliser un modèle qualité comprenant :

- La définition de critères qualité
- Et pour chaque critère, la définition de métriques



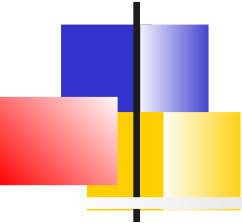
Métriques

Le modèle qualité définit 2 types de métriques :

- **Interne** : Analyse sur le code source ou compilé
Ex : Lisibilité du code, Respect règles de codage, complexité, documentation, copié/collé ...)
- **Externe** : Test du livrable en condition (DevOps) ou sur banc d'essai
Ex : Temps de réponse, Débit, Résistance à la charge, Taux de disponibilité.

Donner des chiffres à des facteurs qualitatifs permet

- D'améliorer le processus de développement
- D'affiner les estimations (prix, délais)



Qualité du code : ISO 25010

SOFTWARE PRODUCT QUALITY

Functional Suitability

- Functional Completeness
- Functional Correctness
- Functional Appropriateness

iso25000.com

Performance Efficiency

- Time Behaviour
- Resource Utilization
- Capacity

Compatibility

- Co-existence
- Interoperability

Usability

- Appropriateness
- Recognizability
- Learnability
- Operability
- User Error Protection
- User Interface Aesthetics
- Accessibility

Reliability

- Maturity
- Availability
- Fault Tolerance
- Recoverability

Security

- Confidentiality
- Integrity
- Non-repudiation
- Authenticity
- Accountability

Maintainability

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

Portability

- Adaptability
- Installability
- Replaceability



Définitions (1)

Functionality (Fonctionnalité) : Capacité à délivrer les fonctionnalités attendues

Efficiency (Efficacité) : Capacité à fournir les performances appropriées selon les ressources utilisées dans les conditions spécifiées

Compatibility (Compatibilité) : Le logiciel peut fonctionner avec d'autres produits. Par exemple, une version antérieure

Usability (Utilisabilité) : Capacité à être compris, appris, utilisé, convivial pour l'utilisateur dans les conditions d'utilisation spécifiées



Définitions (2)

Reliability (Fiabilité) : Capacité à maintenir le niveau de performance attendu dans les conditions d'utilisation spécifiées

Security (Sécurité) : Capacité de protéger les informations afin que les personnes ou systèmes disposent du degré d'accès aux données approprié à leur type et à leur niveau d'autorisation.

Maintainability (Maintenabilité) : Capacité à être modifié (corrigé, amélioré, adapté) selon les changements (environnement/spécification)

Portability (Portabilité) : Capacité à être transféré d'un environnement à un autre



Métriques internes et Outils Qualité

SonarQube est la plate-forme qui regroupe tous les outils de calcul de métriques internes d'un logiciel (toute technologie confondue)

Il intègre 2 aspects :

- Détection des transgressions de règles de codage et estimation de la dette technique
- Calculs des métriques internes et définition de porte qualité

Sa mise en place nécessite une adaptation en fonction du projet.

De nombreux critères qualité ne sont pas couverts



Critères qualité liés à la conception

Les sous-caractéristiques qualité liées à la conception sont plus difficile à évaluer quantitativement. Par exemple :

- Extensibilité : De nouvelles fonctionnalités peuvent être ajoutées au logiciel sans modifications majeures de l'architecture sous-jacente.
- Testabilité : Facilité à implémenter les tests
- Usabilité/Learnability : Courbe d'apprentissage
- ...



Fondamentaux du développement

Typologie des langages
OOP et Design Patterns
Frameworks, IoC et AOP
La programmation réactive



Typologie des langages

<https://medium.com/web-development-zone/a-complete-list-of-computer-programming-languages-1d8bc5a891f>



Classification des langages

Les langages peuvent être classifiés selon de nombreux axes, retenons les suivants :

- Typage dynamique ou statique
- Générique ou dédié à un domaine
- Environnement géré ou non.
- Compilé, interprété
- Paradigme : Objet, procédural, fonctionnel, déclaratif



Approches de la programmation

2 principales approches de la programmation :

- **Impérative** : Se concentre sur l'exécution en séquençant des instructions qui modifie l'état d'un programme
- **Déclarative** : Se concentre sur ce qu'il faut exécuter, définit la logique du programme, mais pas le flux de contrôle détaillé.



Paradigmes les plus populaires

Actuellement, 3 paradigmes sont les plus répandus. Il font tous partie de l'approche impérative :

- **Programmation procédurale ou structurée** : spécifie les étapes qu'un programme doit suivre pour atteindre l'état souhaité.
- **Programmation fonctionnelle** : Traite les programmes comme une évaluation de fonctions mathématiques et évite les données d'état et modifiables
- **Programmation orientée objet (OOP)** : Organise les programmes en tant qu'objets: structures de données constituées de champs de données et de méthodes associées à leurs interactions.



Classification (1)

Language	Type System	Problem Space	Runtime Environment	Paradigm
Java	Static	General	Managed	OO, Imperative
C#	Static	General	Managed	OO, Imperative
VB.NET	Static	General	Managed	OO, Imperative
Ruby	Dynamic	General	Managed	OO, Imperative
C	Static	General	Unmanaged	Procedural, Imperative
C++	Static	General	Unmanaged	OO, Imperative
Groovy	Dynamic	General	Managed	OO, Imperative
JavaScript	Dynamic	General	Managed	OO, Imperative
Python	Dynamic	General	Managed	OO, Imperative
PHP	Dynamic	General	Managed	Procedural, Imperative
ActionScript	Dynamic	General	Managed	OO, Imperative
Fortran	Static	General	Unmanaged	Procedural, Imperative
Perl	Dynamic	General	Managed	Procedural, Imperative
COBOL	Static	General	Managed	Procedural, Imperative



Classification (2)

Language	Type System	Problem Space	Runtime Environment	Paradigm
SQL	Dynamic	DSL	Managed	Declarative
XQuery	Dynamic	DSL	Managed	Declarative
BPEL	Dynamic	DSL	Managed	Declarative
XSLT	Dynamic	DSL	Managed	Declarative
XAML	Dynamic	DSL	Managed	Declarative
Lua	Dynamic	General	Managed	Functional, Imperative
Smalltalk	Dynamic	General	Unmanaged	OO, Imperative
Objective-C	Static	General	Unmanaged	OO, Imperative
ABAP	Static	General	Managed	OO, Imperative
Erlang	Dynamic	General	Managed	Functional
F#	Static	General	Managed	OO, Functional
Scala	Static	General	Managed	OO, Functional, Imperative
M	Dynamic	Purpose	Managed	Declarative
Clojure	Dynamic	General	Managed	OO, Functional, Imperative



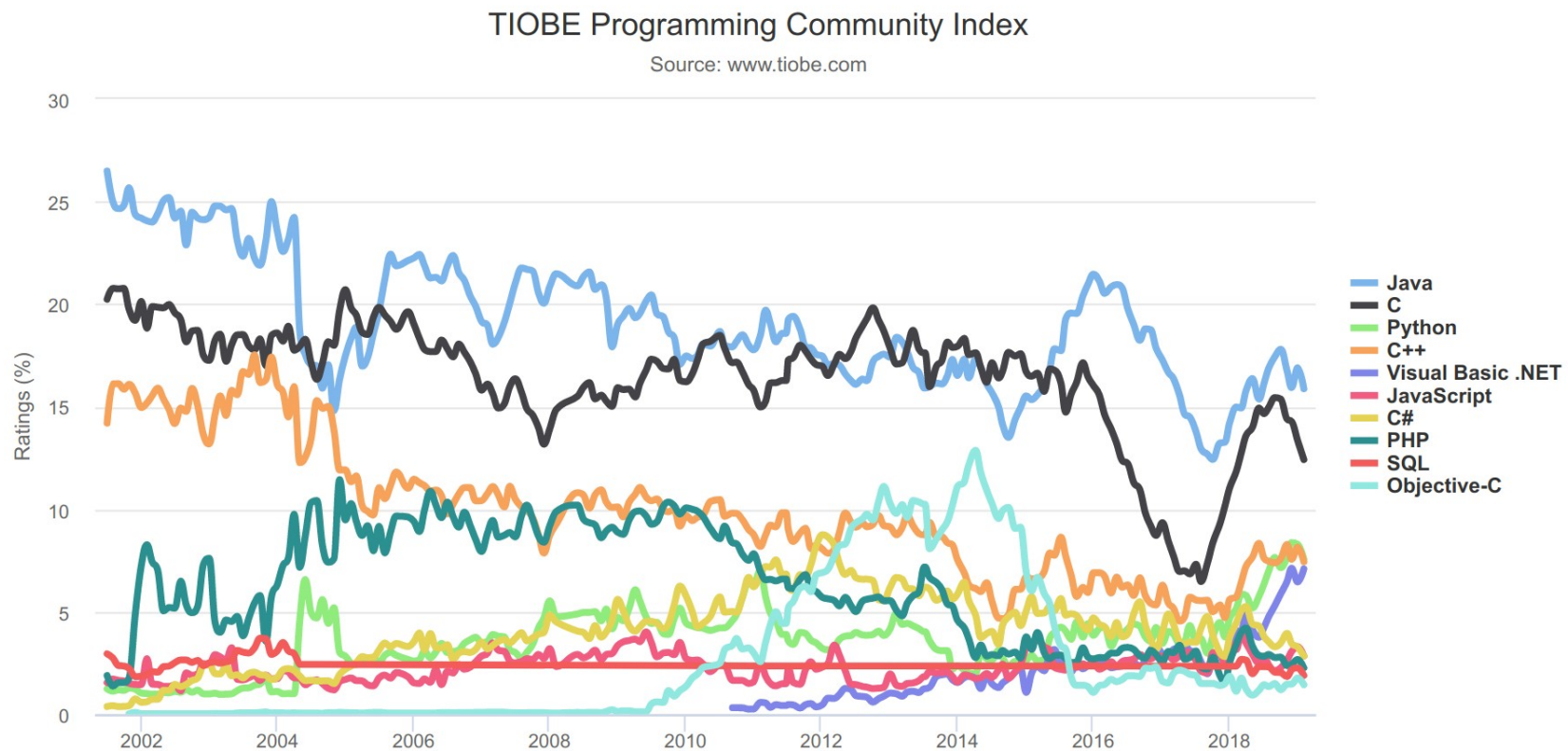
Considérations générales

- Un langage à typage dynamique est plus souple et plus puissant mais peut générer des erreurs à l'exécution
- La compilation ralentit le cycle de développement mais permet de se protéger d'erreurs à l'exécution
- Un interpréteur apporte généralement de la portabilité
- Un environnement géré facilite le travail du développeur (mais peut conduire à un gaspillage de ressources. GC Java)
- La programmation objet produit des projets plus maintenables et plus évolutifs
- La programmation fonctionnelle redevient à la mode avec Javascript et Java 8. Cela peut apporter beaucoup de généricité à un code.

Popularité 2019

Feb 2019	Feb 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.876%	+0.89%
2	2		C	12.424%	+0.57%
3	4	⬆	Python	7.574%	+2.41%
4	3	⬇	C++	7.444%	+1.72%
5	6	⬆	Visual Basic .NET	7.095%	+3.02%
6	8	⬆	JavaScript	2.848%	-0.32%
7	5	⬇	C#	2.846%	-1.61%
8	7	⬇	PHP	2.271%	-1.15%
9	11	⬆	SQL	1.900%	-0.46%
10	20	⬆	Objective-C	1.447%	+0.32%
11	15	⬆	Assembly language	1.377%	-0.46%
12	19	⬆	MATLAB	1.196%	-0.03%
13	17	⬆	Perl	1.102%	-0.66%
14	9	⬇	Delphi/Object Pascal	1.066%	-1.52%
15	13	⬇	R	1.043%	-1.04%
16	10	⬇	Ruby	1.037%	-1.50%
17	12	⬇	Visual Basic	0.991%	-1.19%
18	18		Go	0.960%	-0.46%
19	49	⬆	Groovy	0.936%	+0.75%
20	16	⬇	Swift	0.918%	-0.88%

Evolution





OOP et Design Patterns



Design patterns

Un **patron de conception** (plus souvent appelé **design pattern**) est un arrangement caractéristique de modules ou classes, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

- Il décrit une solution standard, utilisable dans la conception de différents logiciels.
- Il est issu de l'expérience des concepteurs de logiciels.
- Il décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées en fonction des besoins.
- Ils ont une influence sur l'architecture logicielle d'un système informatique.



Historique

Formalisés dans le livre du « *Gang of Four* » (*Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides*) intitulé ***Design Patterns - Elements of Reusable Object-Oriented Software*** en 1995.

Les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70, dont son livre ***A Pattern Language*** définit un ensemble de patrons d'architecture.



Formalisme

La description d'un patron de conception suit un formalisme fixe :

- Nom : Permettant de communiquer entre concepteurs
- Description du problème à résoudre
- Description de la solution : les éléments de la solution, avec leurs relations. i.e. diagramme de classes
- Conséquences : résultats issus de la solution.



Typologie des patterns

Il existe trois familles de patrons de conception selon leur utilisation :

- Les patrons de **construction** : ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- Les patrons **structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation) ;
- Les patrons **comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.



Exemple Factory

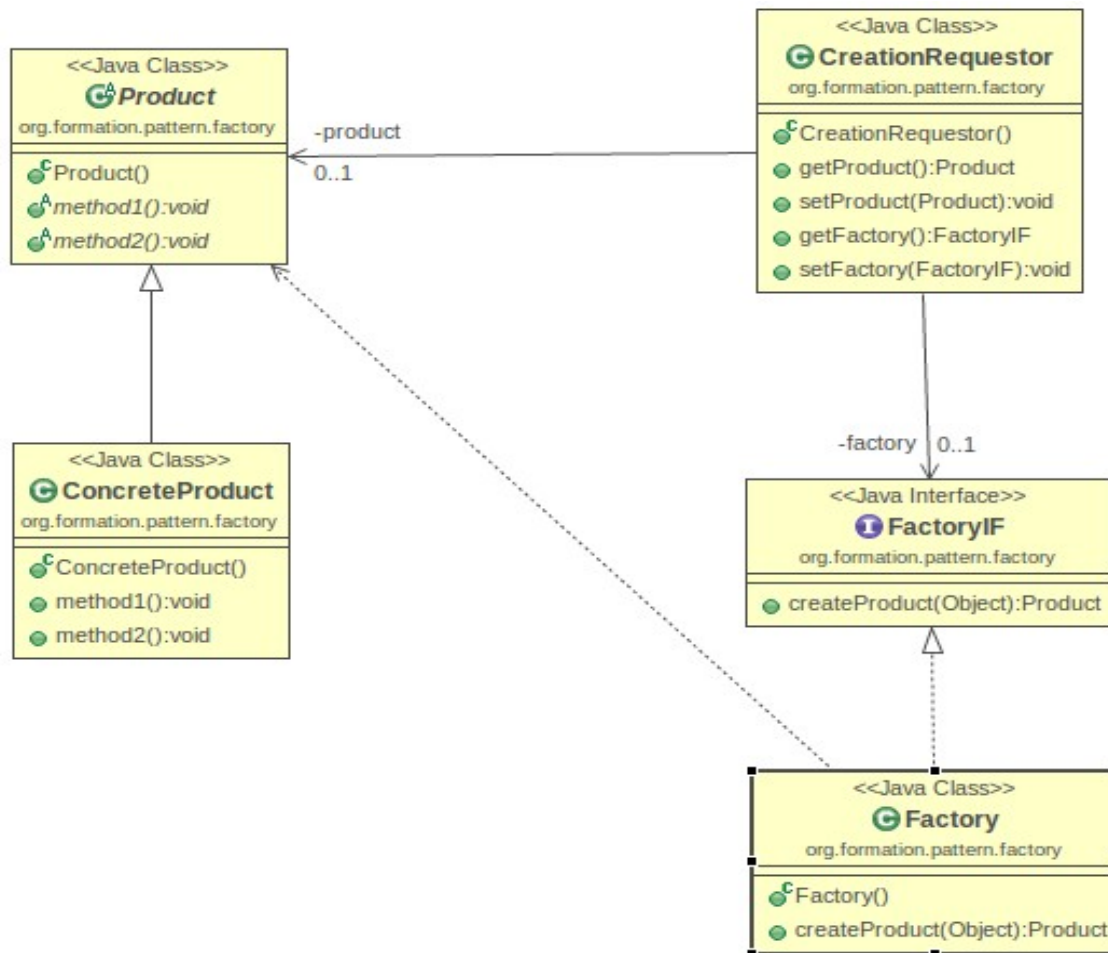
Nom : *Factory Method* [GoF95]

Problème à résoudre : Écriture d'une classe réutilisable avec des types de données arbitraires.

La classe doit pouvoir instancier d'autres classes sans en être dépendantes.

Solution : Elle délègue le choix de la classe à instancier à un autre objet et référence la nouvelle classe créée à travers une interface

Solution





Classes impliquées

Product : Classe abstraite (ou interface) mère de toutes les implémentations

Concrete Product : Classe concrète instanciée

Creation Requestor : Classe indépendante de l'application mais devant créer des classes spécifiques à l'application

Factory Interface : Interface indépendante de l'application déclarant a méthode de création. En général, elle prend un argument *String* discriminant

Factory : Classe implémentant l'interface *Factory* qui instancie une implémentation spécifique de *Product*



Conséquences

CreationRequestor est indépendant des implémentations concrètes créés

L'ensemble des classes concrètes pouvant être instanciées peut changer dynamiquement



Autres patterns de construction

Singleton [GoF95] : Garantit qu'une seule instance d'une classe est créée et utilisée par les autres objets qui en ont besoin

Builder [GoF95] : Permet à un objet client de construire un objet complexe en ne spécifiant que son type et son contenu.

Les détails de la construction sont cachés au client

Prototype [GoF95] : La construction utilise des objets prototypes passés en paramètre et retourne des copies

Object Pool[Grand98] : Gérer la réutilisation d'objets lorsqu'un type d'objet est coûteux à instancier ou lorsque le nombre d'instance est limité



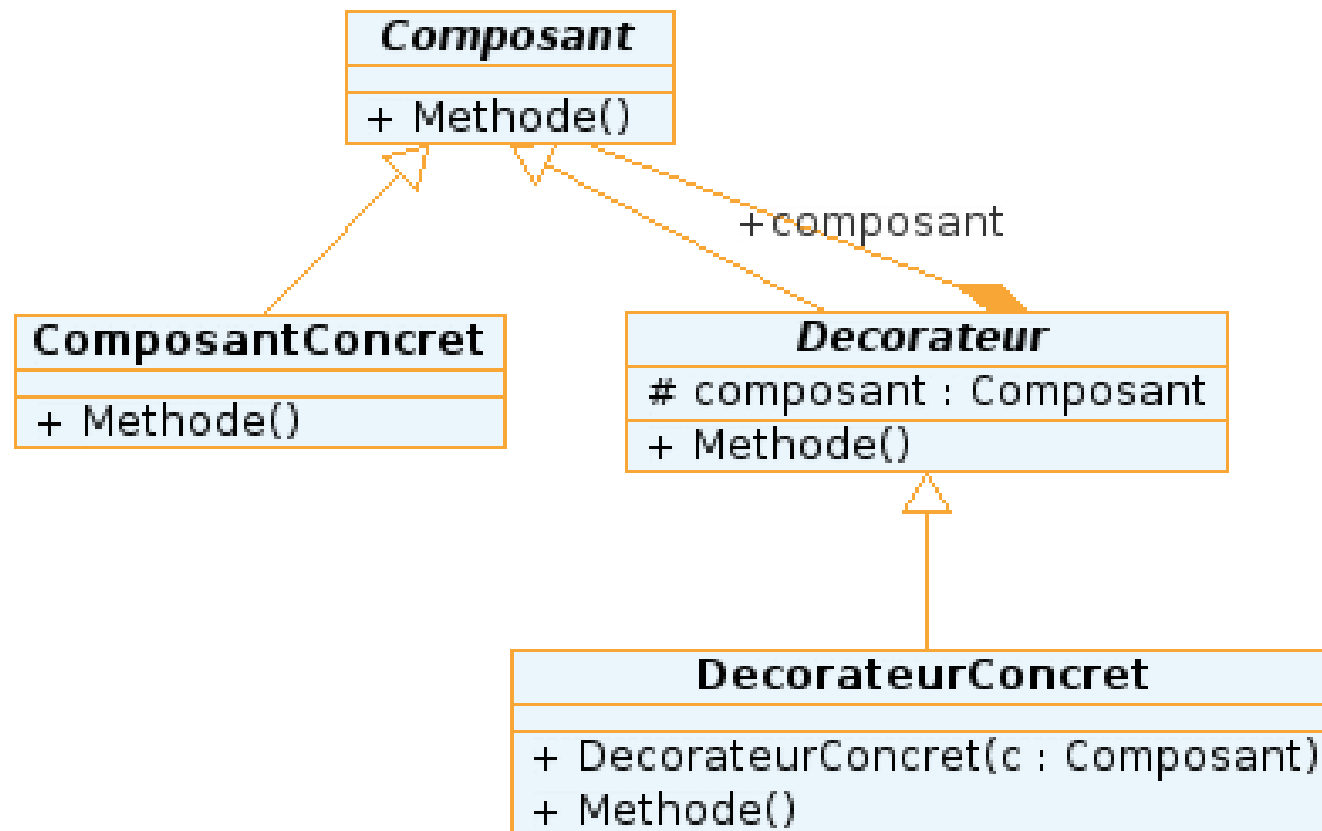
Exemple structuration : Décorateur

Nom : Decorator [GoF95]

Problème à résoudre :

Pouvoir combiner différentes fonctionnalités d'un objet de façon transparente pour un client

Solution





Classes impliquées

Composant : Classe abstraite ou interface mère définissant les fonctionnalités peuvent être étendues avec le pattern Decorateur

ComposantConcret : Classe implémentant la fonctionnalité de base

Decorateur : Classe abstraites ou interface encapsulant *DecorateurConcret*. Elle implémente l'interface *Composant* en appelant la méthode de sa référence concrète

DecorateurConcret : Classe ajoutant une fonctionnalité



Composants

```
public interface Window {  
    public void draw(); // Draws the Window  
    public String getDescription(); // Returns a description of the Window  
}
```

// Extension of a simple Window without any scrollbars

```
class SimpleWindow implements Window {  
    public void draw() {  
        // Draw window  
    }  
  
    public String getDescription() {  
        return "simple window";  
    }  
}
```




Décorateurs

```
abstract class WindowDecorator implements Window {
    protected Window windowToBeDecorated; // the Window being decorated

    public WindowDecorator (Window windowToBeDecorated) { this.windowToBeDecorated = windowToBeDecorated; }
    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }
    public String getDescription() {
        return windowToBeDecorated.getDescription(); //Delegation
    }
}

class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) { super(windowToBeDecorated); }

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }
    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }
    @Override
    public String getDescription() {
        return super.getDescription() + ", including vertical scrollbars";
    }
}
```



Usage

```
public class DecoratedWindowTest {  
    public static void main(String[] args) {  
        // Create a decorated Window with horizontal and vertical scrollbars  
        Window decoratedWindow = new HorizontalScrollBarDecorator (  
            new VerticalScrollBarDecorator (new SimpleWindow()));  
  
        // Print the Window's description  
        System.out.println(decoratedWindow.getDescription());  
    }  
}
```



Conséquences

Avantages

- Plus flexible que l'héritage : Ajout/retrait dynamique de fonctionnalités
- Possibilité de combiner les décorateurs

Inconvénients

- Possibilité de générer des erreurs (incompatibilité de 2 décorateurs)
- Permet en général de réduire le nombre de classes mais plus d'objets à l'exécution. Les objets sont en plus relativement similaires.
 - => Plus difficile à debugger
 - => L'identification d'un composant par l'identité est plus difficile (à cause du wrapper)



Patterns structuraux

Decorator[GoF95] : Étendre les fonctionnalités d'un objet de façon transparente pour un client

Iterator [GoF95] : Interface définissant un accès séquentiel aux éléments d'une collection

Bridge [GoF95] : Permet à une hiérarchie d'abstractions et d'implémentations d'être implémentée à bases de classes indépendantes pouvant être combinées dynamiquement

Adapter[GoF95] : Permettre à une implémentation existante d'être accessible via un interface sans que l'implémentation implémente l'interface

Facade [GoF95] : Simplifie l'accès à un ensemble d'objets en fournissant un point d'entrée unique

Virtual Proxy [Larman98] : Grâce à un proxy, l'objet réel est construit en mode lazy

Cache Management [Grand98] : Permet un accès rapide à des objets qui seraient longs à accéder



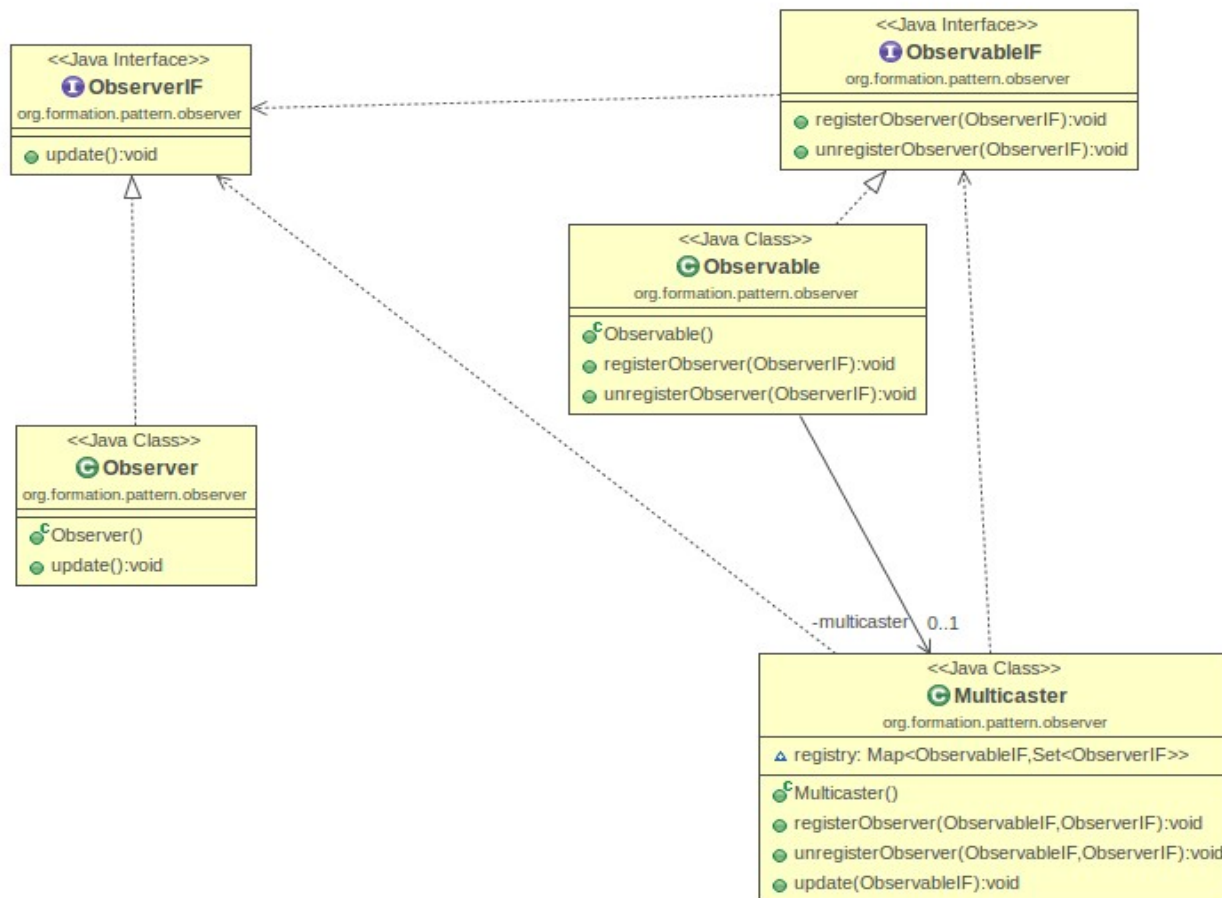
Exemple patterns comportementaux : Observer

Nom : Observer [GoF95]

Problème à résoudre :

Permettre à des objets de dynamiquement enregistrer des dépendances. Les *Observer* sont alors notifiés lorsque l'état de l'objet observé change

Solution





Classes impliquées

ObserverIF : Une interface définissant une méthode *notify*. Un objet *Observable* utilise cette méthode lorsque son état change

Observer : Des instances implémentant *ObserverIF* réagissant aux changements d'états

ObservableIF : Interface définissant des méthodes pour enregistrer ou « désenregistrer » des *ObserverIF*

Observable : Implémente *ObservableIF* et appelle la méthode *notify* de tous les observer enregistrés via la classe de délégation *Multicaster*

Multicaster : Gère les enregistrements d'*Observer* leur notification. Il peut être mutualisé pour tous les *Observable* implémentant la même interface



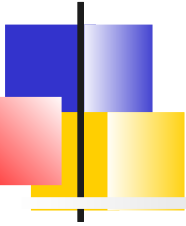
Conséquences

Avantages

- *L'Observer* et *l'Observable* ne se connaissent pas à priori
- Dynamicité et évolutivité

Inconvénients :

- La notification de nombreux objets peut prendre du temps
- On peut introduire des dépendances cycliques



Patterns comportementaux

Observer [GoF95] : Permettre à des objets de dynamiquement enregistrer des dépendances. (Les Observer sont notifiés lorsque l'état de l'objet observé change)

Command [GoF95] : Encapsule des commandes dans des objets afin de les manipuler

Chain Of Responsibility [GoF95] : Permettre à un objet d'envoyer une commande sans connaître quels objets la recevront. Chaque objet dans la chaîne traite la commande et la passe au suivant

Mediator [GoF95] : Utilisation d'un objet pour coordonner les changements entre objets

Snapshot [Grand98] : Capturer un instantané d'un objet afin de pouvoir le restaurer en utilisant une interface

Visitor [GoF95] : Permet d'implémenter de la logique sur tous les objets d'une structure complexe sans les impacter



Frameworks, IoC et AOP



Librairies vs framework

Plutôt que de réinventer la roue, nous utilisons des

- **Librairies** : Code utilitaire
- **Frameworks** : Offrent un modèle de programmation, des utilitaires et des services techniques



Frameworks et IoC

A la différence d'une simple librairie, les frameworks appliquent généralement le pattern d'**Inversion of Control**.

- Ce n'est plus notre programme qui utilise une librairie mais les frameworks qui pilotent nos composants.

Hollywood Principles : Don't call me, I will call you !

- Cette « inversion de contrôle » rend les frameworks comme des squelettes de programmes extensibles que l'utilisateur remplit avec ses propres méthodes/composants.



Injection de dépendances

- ❖ L'injection de dépendance est juste une spécialisation du pattern IoC
- ❖ Le framework appelle les méthodes permettant **d'initialiser** les attributs de vos objets.
- ❖ Le framework ou conteneur est responsable des instanciations
=> Il utilise des patterns de création



Usages, injection de dépendances

- ♦ Spring et Serveurs Java EE : Injection de ressources, ajout de comportements (transactionnel, sécurité, ...)
- ♦ Angular, JSF/JavaEE : Injection de services, Binding
- ♦ .NET : Injection de dépendances
- ♦ PHP-DI : Container d'injection de dépendances



Services techniques

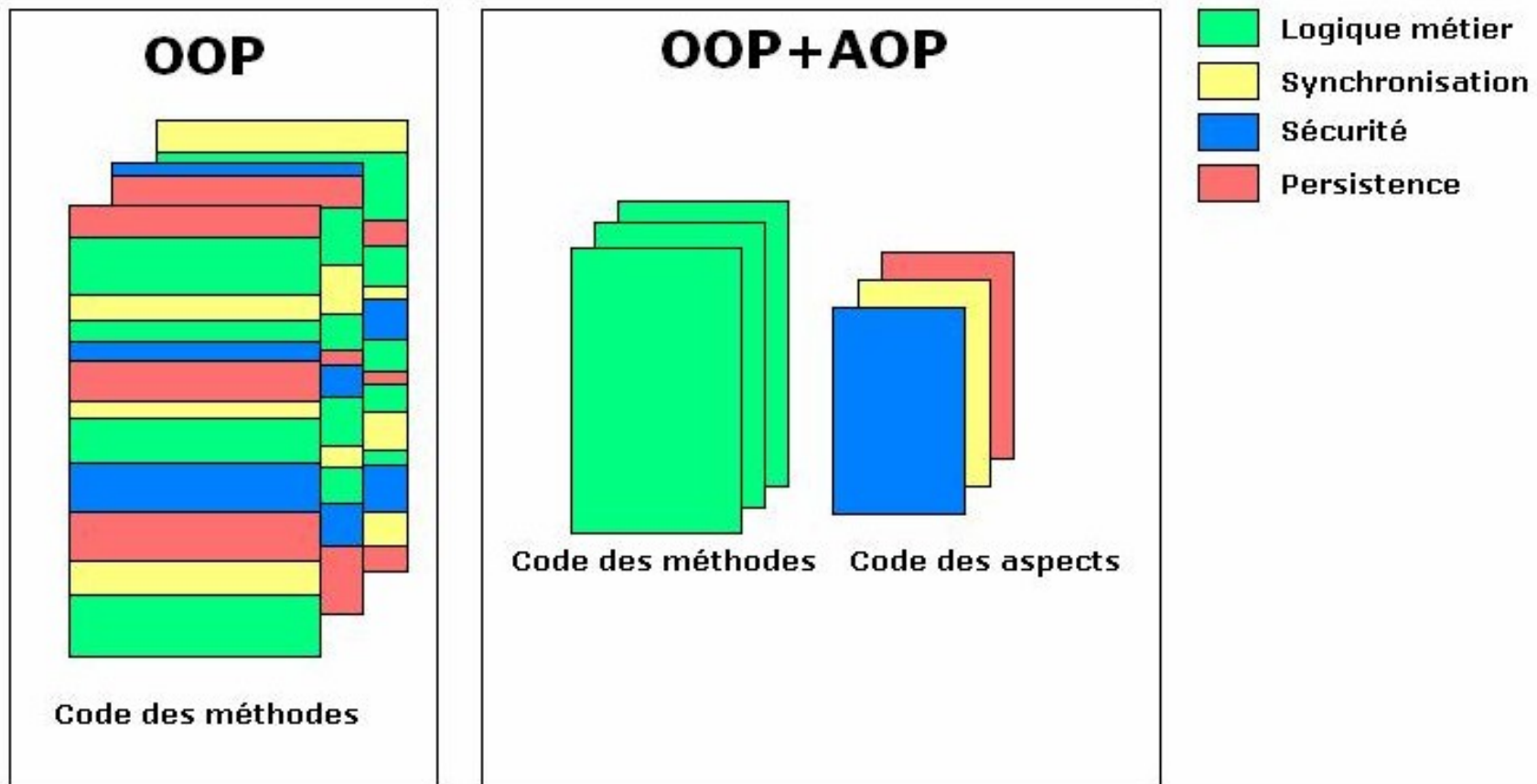
Les framework apporte souvent des services techniques configurables aux composants applicatifs :

- Sécurité
- Transaction
- Accès distant
- Monitoring/Profiling
- Tracing

La configuration s'effectue via des méta-données (XML, Annotation, ...)

Les services sont implémentés généralement à l'exécution via des techniques d'AOP et d'interception

Principes de l'AOP et Cross-cutting concerns





Terminologie AOP

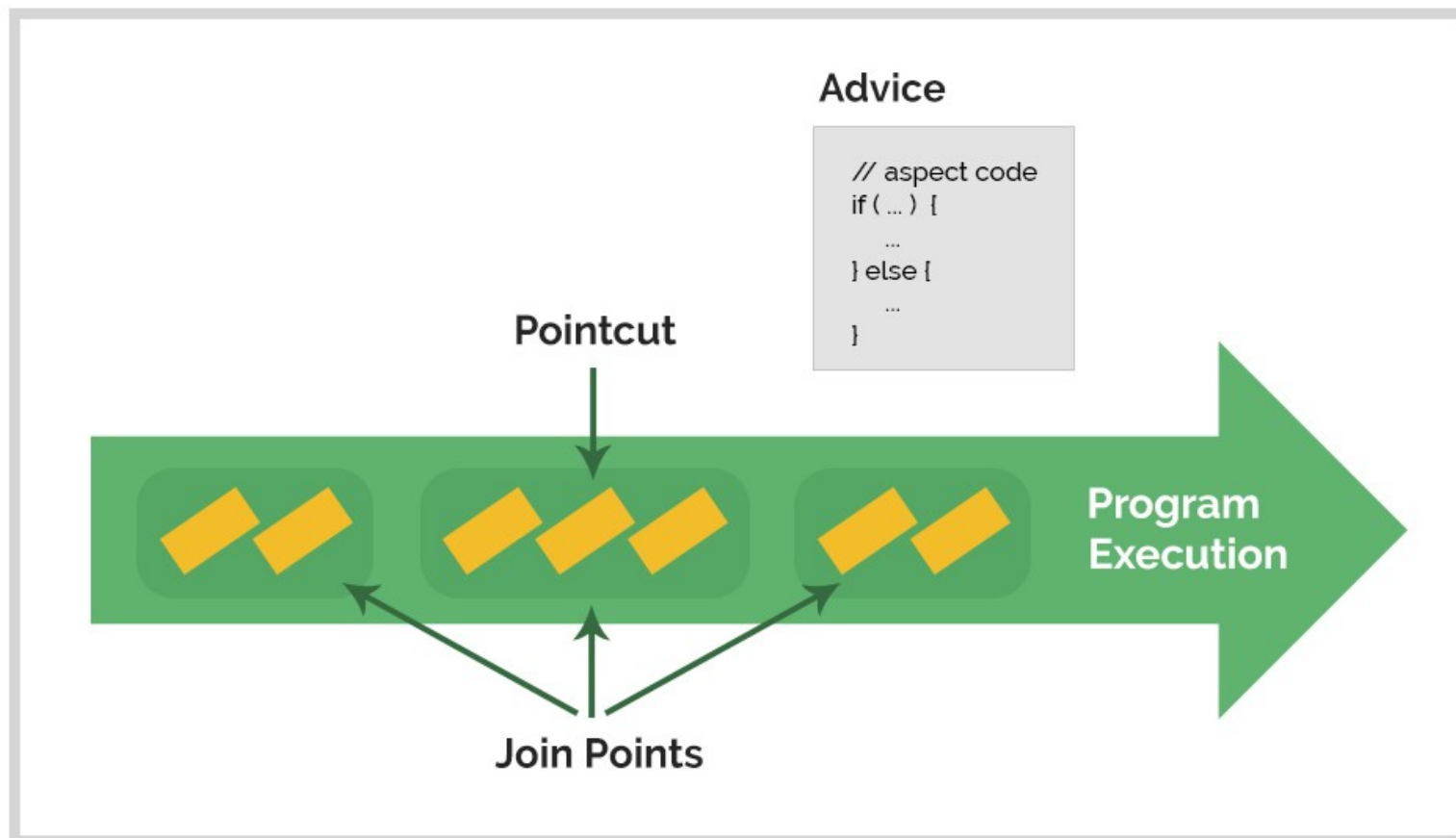
Un **aspect** est la modularisation d'une préoccupation qui concerne plusieurs classes.

Un **point de jonction** est un point lors de l'exécution d'un programme, tel que l'exécution d'une méthode ou le traitement d'une exception.

Un **pointcut** est un prédicat qui aide à faire correspondre un conseil à appliquer par un aspect à un point de jonction particulier.

Un **advice** est une action prise par un aspect à un point de jonction particulier. Les différents types d'advice incluent les advice «autour», «avant» et «après».

Terminologie





Exemple Intercepteur transactionnel

```
public Object invoke(Invocation invocation)
    throws Throwable
{
    if (tm != null)
    {
        Transaction tx = tm.getTransaction();
        if (tx != null) invocation.setTransaction(tx);
    }
    return getNext().invoke(invocation);
}
```



Programmation réactive



Programmation réactive

Paradigme de programmation déclarative qui concerne les flux de données et la propagation du changement.

$a = b + c$

- En programmation impérative, a est affecté au résultat de $b + c$ à l'instant où l'expression est évaluée. Plus tard, les valeurs de b et c peuvent être modifiées sans effet sur la valeur de a .
- En programmation réactive, la valeur de a est automatiquement mise à jour chaque fois que les valeurs de b ou c changent, sans que le programme ait à réexécuter l'instruction.



Applications

Ce paradigme se répand actuellement :

- Front-end : Manipuler des événements d'interface utilisateur et des réponses d'API ou des flux (web sockets)
- Back-end : Traitement asynchrone indépendant des modèles concurrentiels sous-jacent.
- Quelque-soit les langages et frameworks : RxJs, RxJava, RxNet, RxPy, Reactive Streams, JDK9, Spring Reactor, ...



Pattern et *ReactiveX*

La programmation réactive se base sur le pattern **Observable** qui est une combinaison des patterns *Observer* et *Iterator*¹

Elle utilise en plus la programmation fonctionnelle permettant de définir facilement des opérateurs

Elle est formalisée par l'API **ReactiveX** (<http://reactivex.io>) et de nombreuses implémentations existent pour différent langages (*RxJS*, *RxJava*, *Rx.NET*)

1. On peut boucler (*Iterator*) mais les éléments arrivent au fur et à mesure (*Observer*)



API

Un **observable** représente un flux d'évènements ou source de données.

On peut créer un observable à partir d'un tableau, d'une ressource REST, d'évènements utilisateur, d'une requête NoSQL

Les **subscriptions** ou **abonnements** sont les objets qui déclenchent le flux. Sans abonné pas d'évènements !

Les abonnés peuvent gérer le débit des évènements (back-pressure)

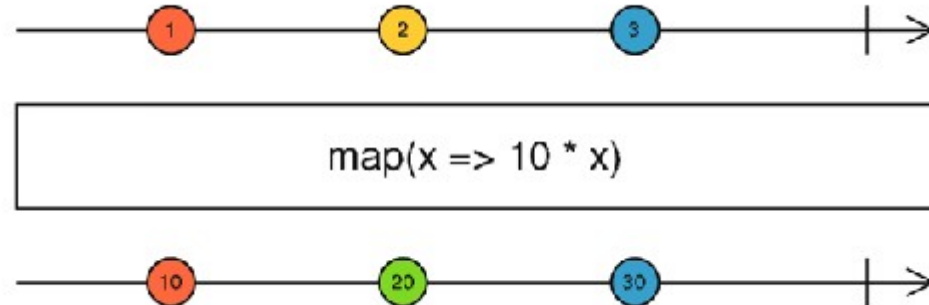
Les **opérateurs** offre un moyen de manipuler les valeurs d'une source, en renvoyant une observable des valeurs transformées. Ils peuvent être combinés.



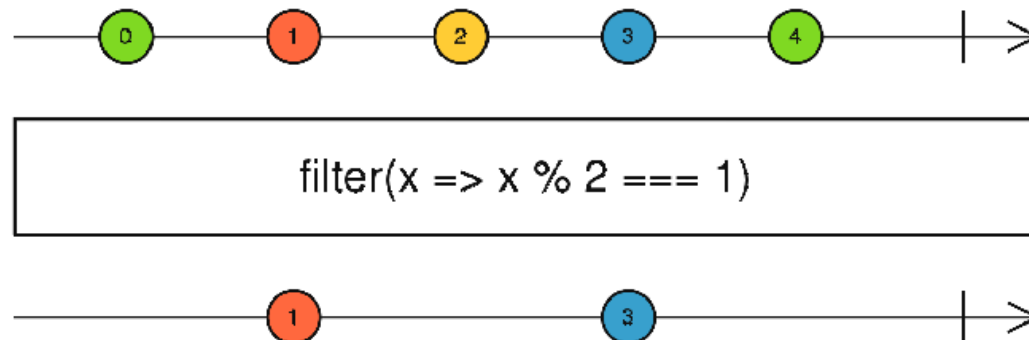
Marble diagrams

Pour expliquer les opérateurs, la doc utilise des ***marble diagrams***

Exemple *map* :



Exemple *filter*

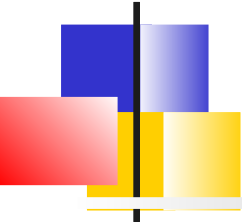




Exemple RxJS

```
// L'appel de cette méthode est non-bloquant
// Opérateur map utilisé
find(id: number): Observable<Tweet> {
    return this.http.get<Tweet>(`${this.resourceUrl}/${id}`,
        { observe: 'response' })
        .map((res: Tweet) => this.convertResponse(res));
}

-----
load(id) {
    this.tweetService.find(id)
        .subscribe(
            // Fonction exécutée lors de la réception de l'évt
            (tweetResponse: HttpResponse<Tweet>) => {
                this.tweet = tweetResponse.body;
            }
        );
}
```



Chaînage d'opérateur avec RxJS

```
// observable, pipe chaine les opérateurs
inputValue
  .pipe(
    // Attente de 200ms
    debounceTime(200),
    // Si la valeur est identique, ignore
    distinctUntilChanged(),
    // si une nouvelle valeur arrive pendant que le traitement est actif
    // annule la demande précédente et «passe» à un nouvel observable
    switchMap(searchTerm => typeaheadApi.search(term))
  )
// Crée l'abonnement et déclenche le flux
.subscribe(results => {
  // Mise à jour du DOM
});
```



Programmation distribuée

Apports et contraintes

Services techniques

Interactions Client/Serveur

Modèles concurrentiels du serveur



Systèmes distribués

- ❖ Le développement des réseaux a fait apparaître les applications **réparties ou distribuées** :

Différents composants logiciels qui coopèrent pour fournir un service cohérent s'exécutent sur des ressources distinctes

- ❖ Ces systèmes reposent sur le modèle **client/serveur** :

Un objet est client d'un autre objet situé sur une machine distincte : le serveur/service



Avantages de la distribution

Performance, le système a plus de CPUs, mémoire, espace disque à sa disposition

Scalabilité : Les ressources peuvent être ajustées dynamiquement à la charge utilisateur

Fiabilité, disponibilité, Tolérance aux pannes : des éléments du système peuvent être arrêtés sans arrêter le service

Mutualisation : Certains services ou fonctionnalités peuvent être mutualisés dans plusieurs systèmes

Déploiement : Les déploiements ne nécessitent généralement pas de déplacement physique sur les plateformes clientes



Inconvénients

Complexité :

- Transfert entre 2 zones mémoire : mécanisme de sérialisation, adaptation des données, ...
- Latence et asynchronisme dus aux réseau
- Contexte de sécurité à propager
- Transactions distribuées complexes
- Surveillance dispersée

Fragilité :

- plus de composants peuvent tomber en erreur
- Dépendance vis à vis du réseau



Programmation distribuée

Apports et contraintes

Services techniques

Interactions Client/Serveur

Modèles concurrentiels du serveur



Services techniques

Les services techniques requis à la programmation distribuée sont apportés par le middleware ou un framework :

- Localisation du service via un service **d'annuaire**
=> Le code client est indépendant de l'emplacement du service
- Fourniture d'un **proxy**
=> Le code client ne se soucie pas des aspects réseau
- Bibliothèques de **sérialisation/désérialisation**
=> Le code client ne travaille qu'avec des objets



Service de nommage

Un service de nommage permet de localiser un objet ou une ressource en utilisant le nom

Un service de nommage fournit deux services:

- un service **d'association (binding)** d'une ressource à un nom ou **d'enregistrement**
- un service de **consultation (lookup)** ou de **découverte** permettant de localiser la ressource à partir de son nom
Ce service retourne généralement un proxy



Exemples

Internet : DNS

JavaEE : JNDI

Webservices : UDDI

Bus : Service de nommage CORBA

Micro-services : Eureka, Consul,
Zookeeper, Services Kubernetes



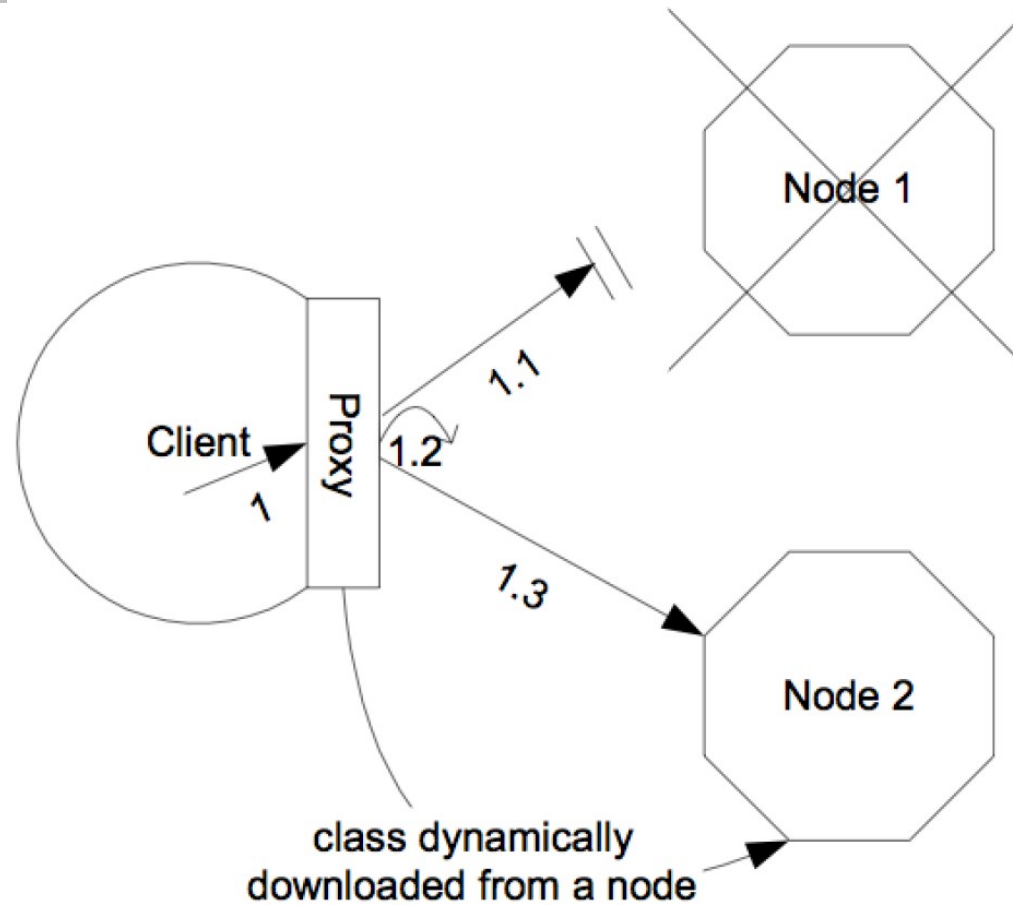
Proxy client

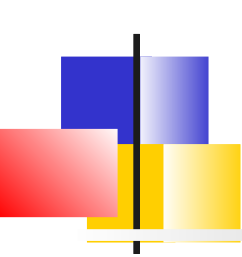
L'objet retournée lors d'une découverte ou lookup est un proxy fourni par le framework.

Le proxy peut intégrer d'autre services techniques :

- Répartition de charge si le service est répliqué
- Fail-over si un des répliques est en erreur
- Fall-back si le service n'est pas utilisable

Intercepteur / proxy





Échange de données par flux

Les données sont échangées via des flux (**stream**).

Les objets mémoire sont **sérialisés** puis recréés sur le système cible (désérialisation).

Les formats d'échanges sont variés

- Binaires
- XML
- JSON

Il faut souvent adapté le format au client : DTO, Annotations, Médiateur, XSL, ...



Librairies

Java :

- Binaire : *readObject, writeObject*
- XML : *JAXB, xstream*
- JSON : *MOXy, Jackson, Toplink*

Javascript :

- XML : JKL Parse XML

PHP :

- XML : Natif, SimpleXML, ...
- JSON : `json_encode($myObject);`

.NET :

- XML : *System.Xml.Serialization*
- JSON : *JSON.Net, JavaScriptSerializer*



Programmation distribuée

Apports et contraintes

Services techniques

Interactions Client/Serveur

Modèles concurrentiels du serveur



Stateless versus Stateful

- ❖ **Stateless** : Le serveur est amnésique, il ne se rappelle pas des requêtes précédentes.
 - Tâches du serveur/container facilitées. Gestion des pool par ex.
 - Plutôt performant. (Peu d'instanciation, réutilisation des objets)
 - Pas terrible pour la bande passante.
 - Très scalable (pas de nécessité à répliquer un d'états entre des nœuds d'un cluster)
 - Exemple : Services REST, , ...
- ❖ **Stateful** : Un état est conservé côté serveur : session utilisateur, conversation, ...
 - Tâches serveur plus complexe
 - Autant d'instanciation que de clients simultanés. Quand libérer les objets ? Timeout
 - Limiter l'espace mémoire via des mécanismes de passivation/activation
 - Moins bonne scalabilité, nécessite de la réplication d'état
 - Échanges avec le client limités aux Ids
 - Exemple : Frameworks basés sur la session HTTP



Exemples stateless , stateful

Modèle stateful :

- Application web utilisant la session HTTP

Modèle stateless :

- Appel Service Web (Rest ou SOAP)



Modèle synchrone / asynchrone

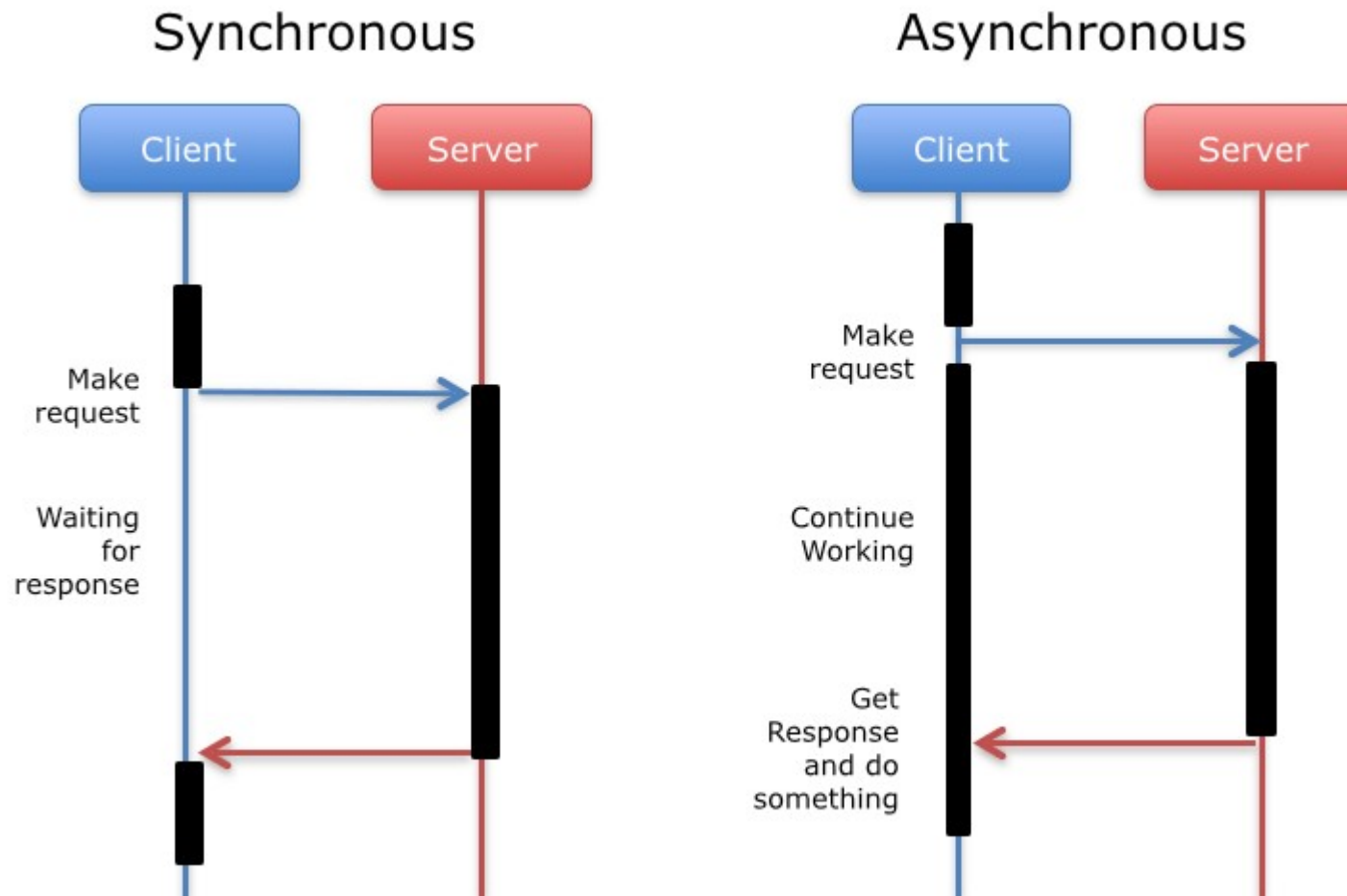
L'interaction avec le serveur peut être

- synchrone/bloquante
- ou asynchrone/non bloquante

Le synchronisme est moins complexe,

L'asynchronisme, si il est bien utilisé,
permet généralement d'optimiser les
ressources

Synchrone vs Asynchrone





Asynchronisme Javascript

En javascript, il y a une seule thread.

Lors d'un appel à une fonction longue (appel I/O par exemple), il est conseillé d'utiliser un appel asynchrone non bloquant.

- De façon directe, en utilisant une fonction de ***callback***
- De manière plus élaborée et plus élégante en utilisant l'objet ***Promise***
- En utilisant le modèle réactif : ***Observable***



Exemple Call-back

```
console.log("About to get the website ...") ;  
$.ajax("http://somedown.example.com", {  
    success : function (result) {  
        console.log(result) ;  
    },  
    error : function() {  
        throw new Error("Error getting the website") ;  
    }  
}) ;  
console.log("Continuing about my business ...") ;
```



Plus élégant : *Promise*

// Callback

```
$("#button").click(function() {  
    promptUserForTwitterHandle(function (handle) {  
        twitter.getTweetsFor(handle, function (tweets) {  
            ui.show(tweets) ;  
        }) ;  
    }) ;  
}) ;
```

// Promises

```
$("#button").clickPromise()  
    .then(promptUserForTwitterHandle)  
    .then(twitter.getTweetsFor)  
    .then(ui.show) ;
```



Serveur : Modèle concurrentiel



Modèle concurrentiel

Le modèle concurrentiel concerne le nombre de threads/processus présents sur le serveur.

Théoriquement, le nombre dépend :

- de l'infrastructure : Nombre de CPUs disponibles
- Des accès I/O :
 - Bloquant : La thread doit attendre
 - Non bloquant : La thread peut exécuter du travail lors de l'attente



Serveurs Multi-thread

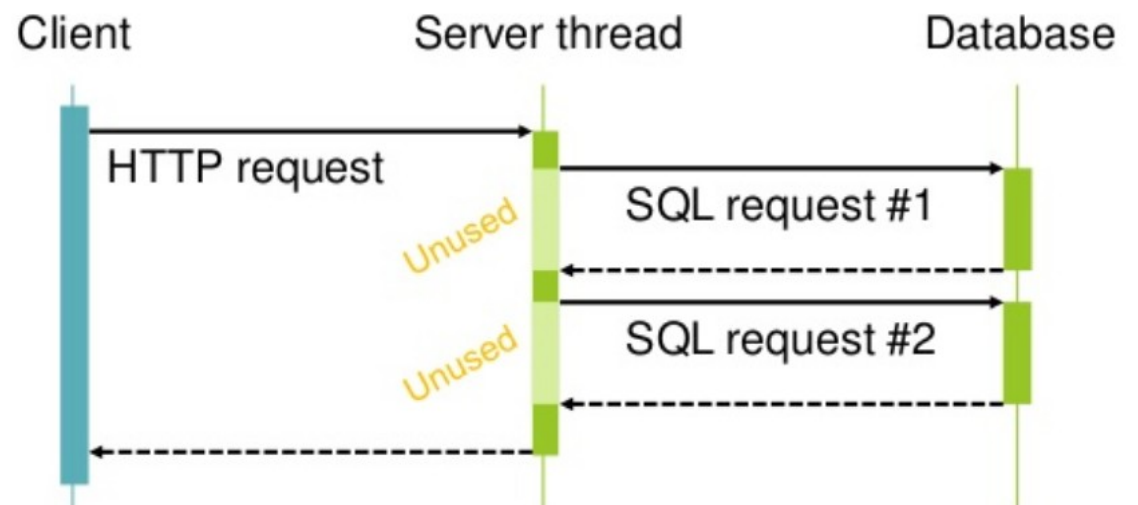
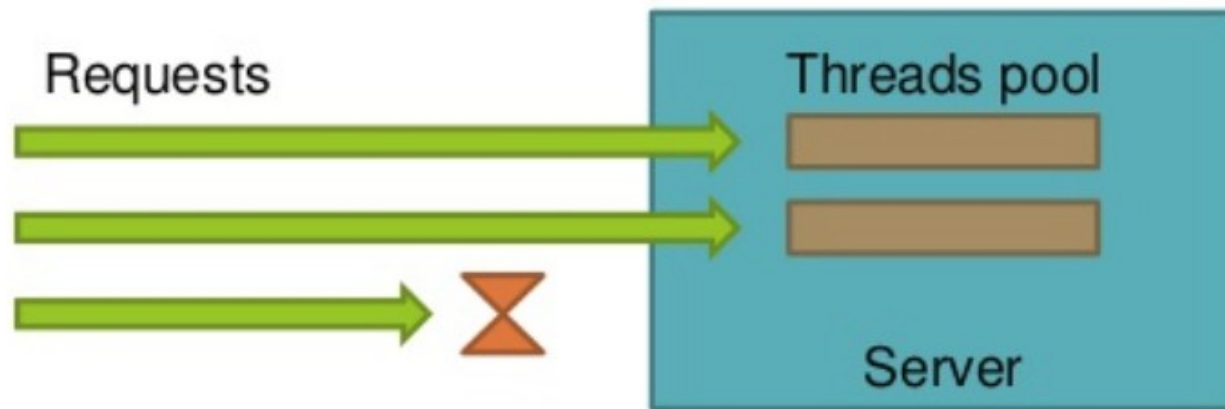
Les serveurs web traditionnels utilisent des pools de threads (ou de processus).

Chaque connexion utilise une thread du pool ... ou attend qu'une thread soit disponible

- Les threads se partagent les CPUs disponibles et l'OS doit effectuer des changements de contexte pour partager le CPU entre les threads
- Les threads partagent le même espace mémoire et les développeurs doivent faire attention aux problèmes de concurrence

Si une thread fait des appels I/O (Accès BD par exemple), elle est bloquée

Modèle bloquant





Configuration pool Tomcat

```
<!-- A HTTP/1.1 Connector on port 8080 -->
```

```
<Connector port="8080"
```

```
    maxThreads="250"
```

```
    maxHttpHeaderSize="8192"
```

```
    emptySessionPath="true"
```

```
    enableLookups="false" redirectPort="8443"
```

```
    acceptCount="100"
```

```
    connectionTimeout="20000"
```

```
    disableUploadTimeout="true"/>
```



Serveur mono-thread

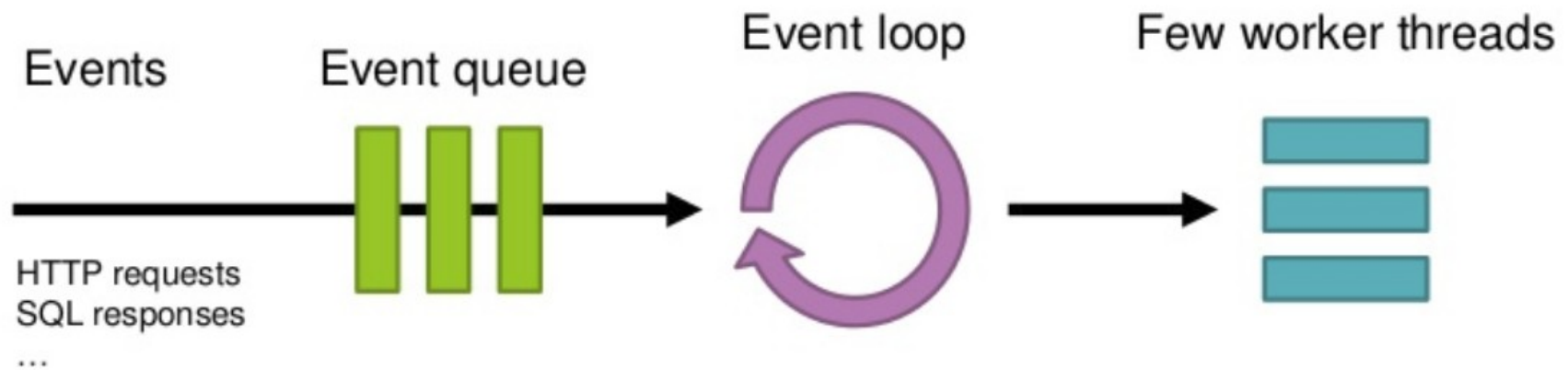
Dans un modèle mono-thread (*Node.js*, *Spring WebFlux*), une unique thread d'écoute exécute une boucle d'attente d'événements (arrivée de requête).

Elle délègue le traitement à un petit pool de worker threads qui utilisent des appels I/O asynchrones non bloquants. Pendant les appels I/O, les worker peuvent traiter d'autres requêtes

- => Très adapté aux systèmes effectuant de nombreuses requêtes réseau non bloquant (Architecture microservices)

- => Très scalable : Peu de threads supporte beaucoup de charge

Modèle non bloquant





ServerLess

La migration de nombreuses fonctionnalités côté FrontEnd, les modèles de programmation pilotés par des événements ont donné naissance à des système « server less ».

=> Il n'y a plus de serveurs toujours en exécution mais le code applicatif est exécuté dans des *conteneurs éphémères*.

On parle alors de «**FaaS**» (Function as a Service) comme AWS Lambda



Architectures applicatives

Services back-end

Message brokers

Architecture Micro-services

Frameworks clients

BigData, Machine Learning

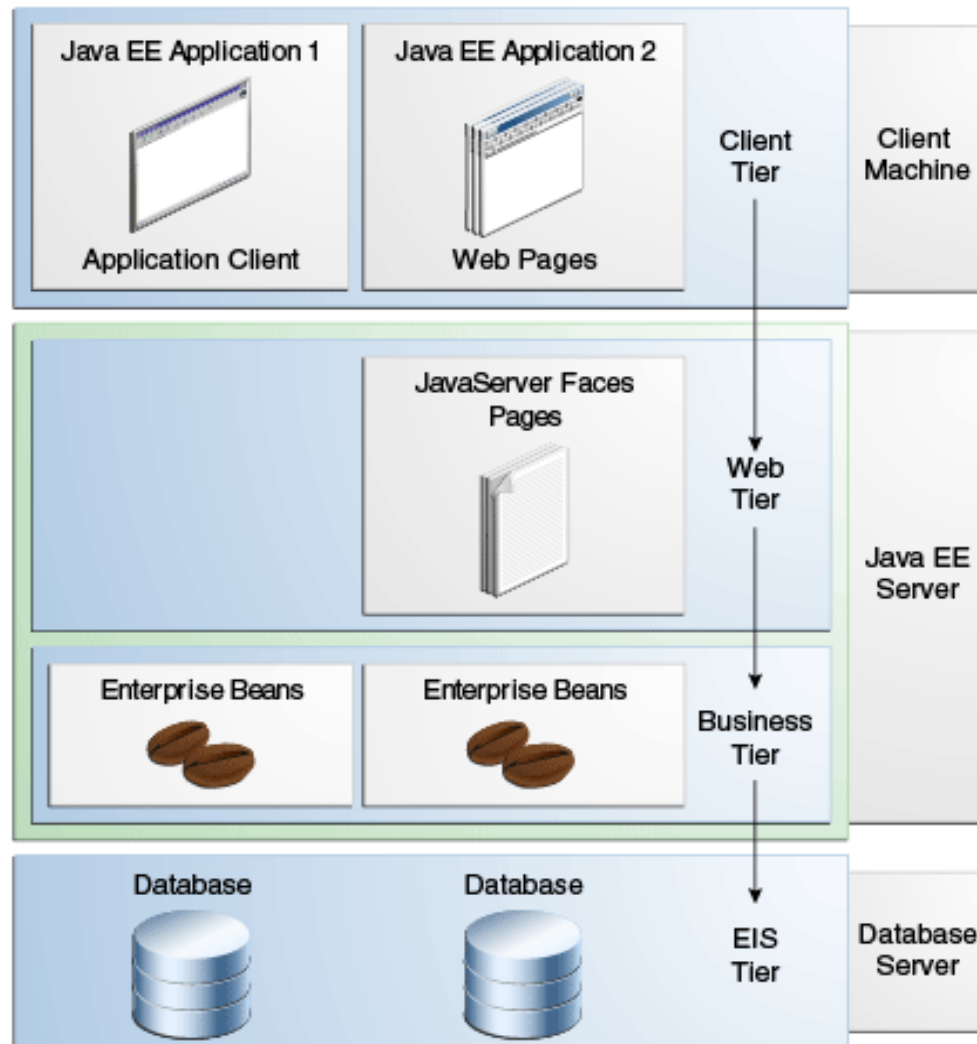


Evolution des architectures

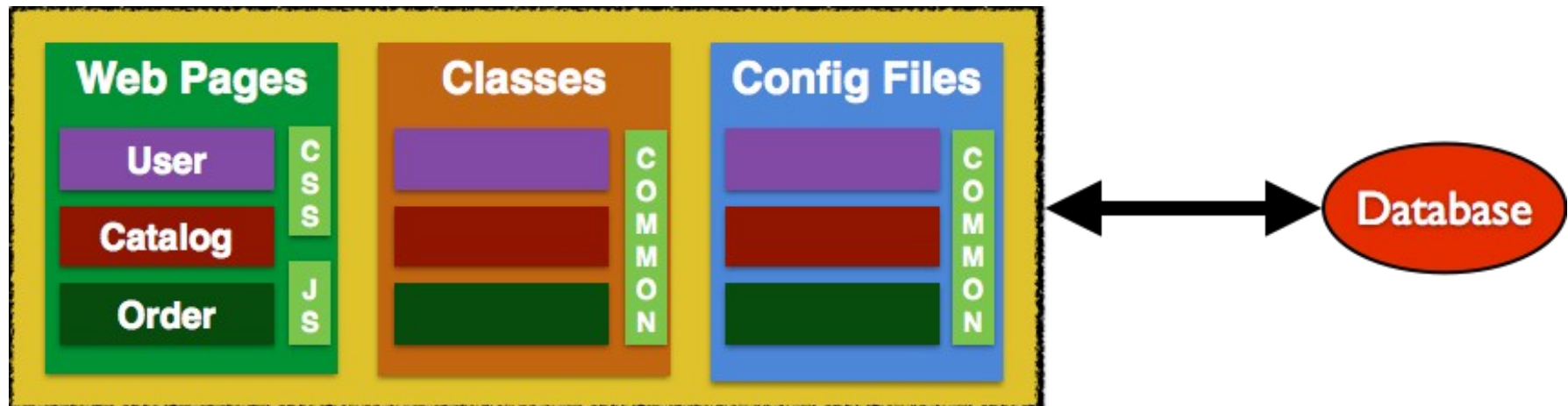
Les architectures des applications back-end adossés à des supports de persistance ont évolués ces dernières années :

- Application 3-tiers stateful déployée sur un serveur applicatif mutualisé
- Application monolithique stateless indépendante
- Architectures micro-services

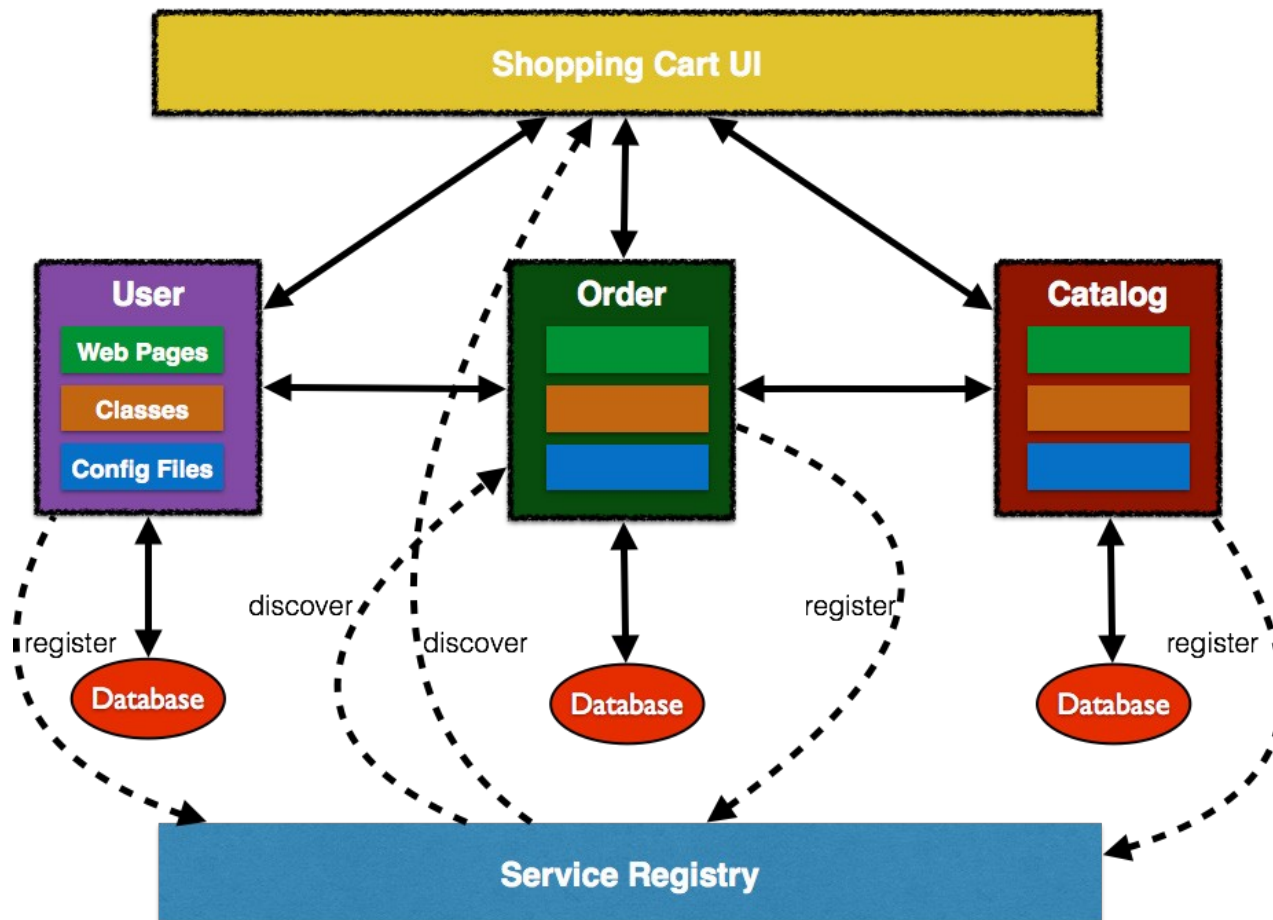
Application 3-tiers Java EE



Architecture monolithique

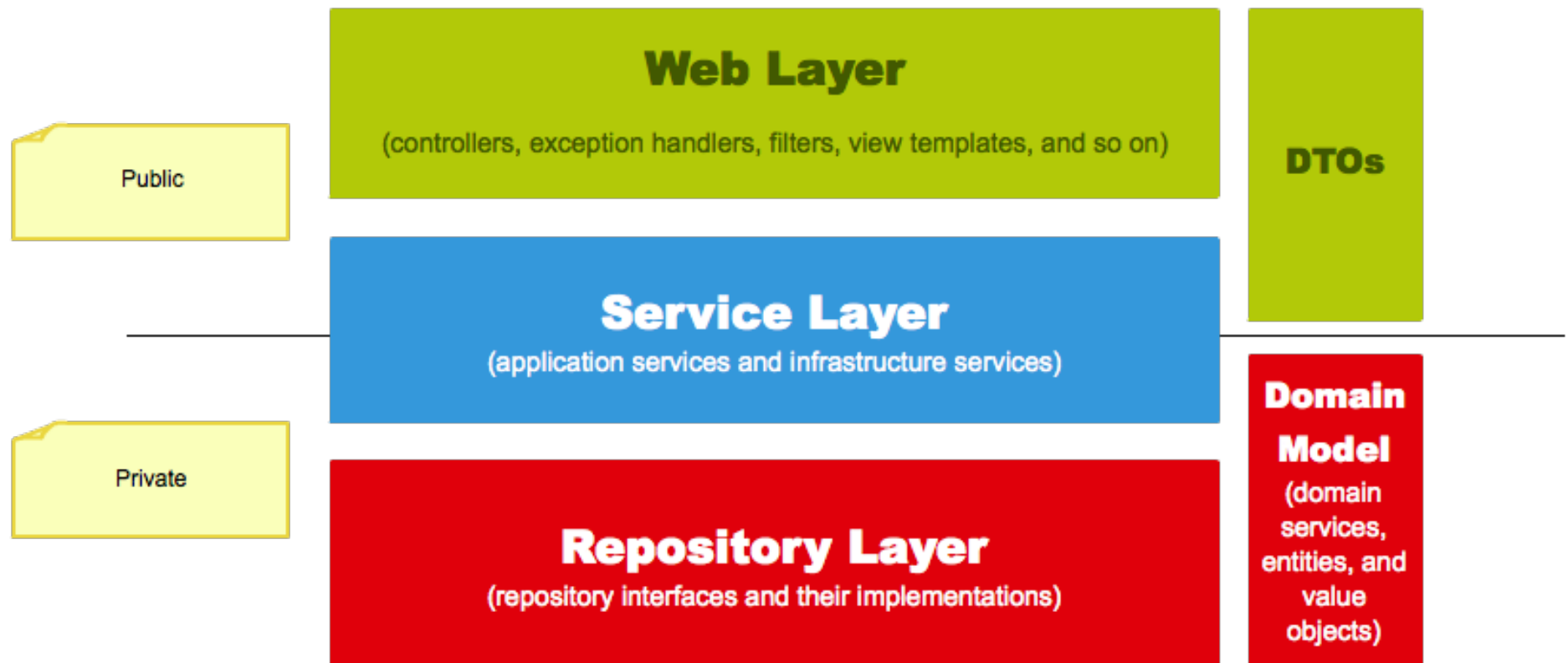


Version micro-service





Packages classiques





Persistance

La couche de persistance comprend :

- Les classes du modèle : modélisant les données du métier
- Les classes utilitaires permettant les opérations de persistance simples et le requêtage

Des frameworks (ORM, ...) permettent de passer du modèle objet au modèle de stockage sous-jacent (SQL, NoSQL, ...)

Considérations :

- Besoin de contraintes, de relations
- Modèle transactionnel, gestion de la concurrence, Transactions distribuées ou locales
- Modèle réactif ou Non
- Big Data ?
- Optimisation index, requêtes, dimensionnement du pool



Services métier

Exposition d'une API métier

Une méthode correspond à l'agrégation de plusieurs opérations de persistance en un opération atomique métier

Considérations :

- Délimitation de transaction
- Propagation de transaction
- Implémentation des règles/processus métier (Moteur de règles / processus)
- Sécurité



Couche Web

Relai des requêtes HTTP vers les méthodes métier appropriés

Framework de mapping, de conversion/validation HTTP/Objet

Considération :

- Format de sérialisation : HTML / XML / Json
- Modèle stateless ou stateful
- Sécurité
- Gestion des codes retours, des erreurs
- Modèle concurrentiel du serveur



Services Web

Les services Web destinés à être consommés par d'autres systèmes ont démarré avec SOAP :

- Basé sur XML et les schémas
- Format auto-descriptif : WSDL
=> Rigueur, verbosité

Actuellement, les services RestFul sont privilégiés :

- Basé sur JSON
- Utilise uniquement HTTP
- OpenAPI
=> Plus léger, plus facile à développer



Définition

Un système REST est défini par 6 contraintes architecturales :

- Client/Serveur
- Stateless
- Réponses indiquant si elles peuvent être cachées
- Architecture en couche : serveurs intermédiaires
- Code à la demande. La réponse peut contenir du script
- Interface uniforme
 - Identification des ressources dans la requête
 - Manipulation des ressources à travers les représentations
 - Message self-descriptif
 - Hypermedia comme moteur des actions possibles (HATEOAS)



RestFul API

URI	GET	PUT	POST	DELETE
Collection, http://api.example.com/resources/	Liste les URI et autres détails des membres de la collection	Remplace la collection avec une autre collection	Crée une nouvelle entrée dans la collection. La nouvelle URI créé automatiquement et retournée par l'opération.	Supprime la collection.
Element, http://api.example.com/resources/item17	Récupère une vue détaillé de l'élément dans le format approprié	Remplace ou créé l'élément	Traite l'élément comme une collection et y ajoute une entrée	Supprime l'élément



Sérialisation JSON

Un des principales problématiques des interfaces REST et la conversion des objets du domaine au format JSON.

Des frameworks spécialisés sont utilisés (Jackson, Gson) mais en général le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bi-directionnelles entre entités
- Adaptation aux besoins de l'interface de front-end
- Optimisation du volume de données échangées

Jackson, par exemple , fournit 2 moyens pour adapter la sérialisation/désérialisation :

- L'enregistrement d'*ObjectMapper*
- L'utilisation d'annotations



Exemple JAX-RS

@Path("/student/data")

```
public class RestServer {
```

```
    @GET
```

```
    @Produces(MediaType.APPLICATION_JSON)
```

```
    public Student getStudentRecord(){
```

```
        Student student = new Student();
```

```
        student.setLastName("Hayden");
```

```
        student.setSchool("Little Flower");
```

```
        return student;
```

```
    }
```

```
    @POST
```

```
    @Consumes(MediaType.APPLICATION_JSON)
```

```
    public Response postStudentRecord(Student student){
```

```
        return Response.status(201).entity("Record entered: " + student).build();
```

```
    }
```

```
}
```



OpenAPI

La spécification ***OpenAPI (OAS)*** définit une interface standard indépendante du langage pour les API RESTful

Cela permet de comprendre les capacités du service sans accès au code source ou à la documentation.

Une définition *OpenAPI* peut ensuite être utilisée par des outils de génération de documentation, de génération de code (client et serveur), de tests, etc.



Exemple Swagger

Swagger Petstore

pet : Everything about your Pets

[Show/Hide](#) | [List Operations](#) | [Expand Operations](#)

store : Access to Petstore orders

[Show/Hide](#) | [List Operations](#) | [Expand Operations](#)

user : Operations about user

[Show/Hide](#) | [List Operations](#) | [Expand Operations](#)

POST	/user	Create user
POST	/user/createWithArray	Creates list of users with given input array
POST	/user/createWithList	Creates list of users with given input array
GET	/user/login	Logs user into the system
GET	/user/logout	Logs out current logged in user session
DELETE	/user/{username}	Delete user
GET	/user/{username}	Get user by user name
PUT	/user/{username}	Updated user

[BASE URL: /v2 , API VERSION: 1.0.0]

VALID {...}



Architectures applicatives

Services back-end

Message brokers

Architecture Micro-services

Frameworks clients

BigData, Machine Learning

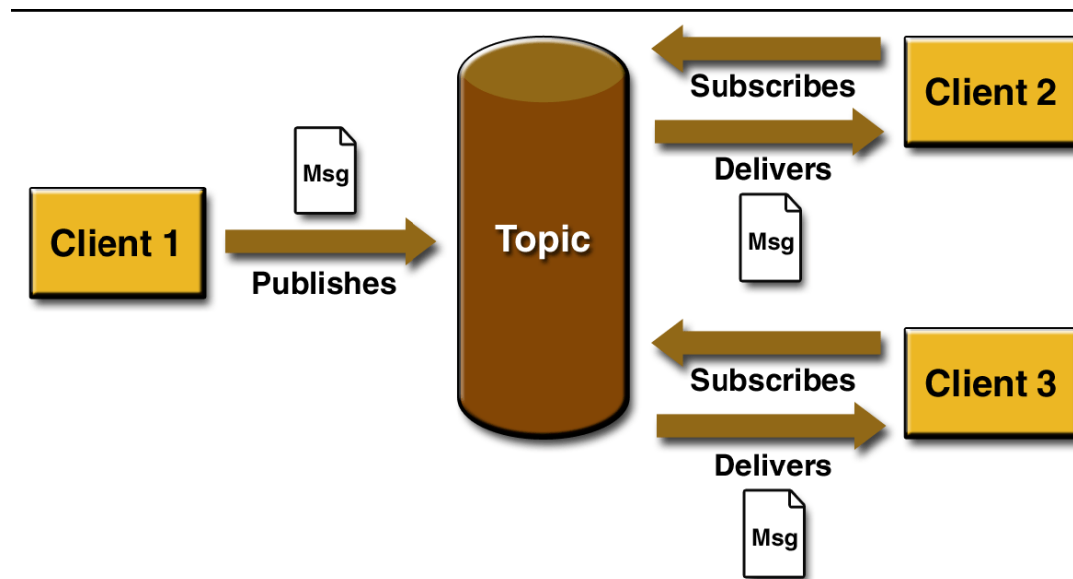
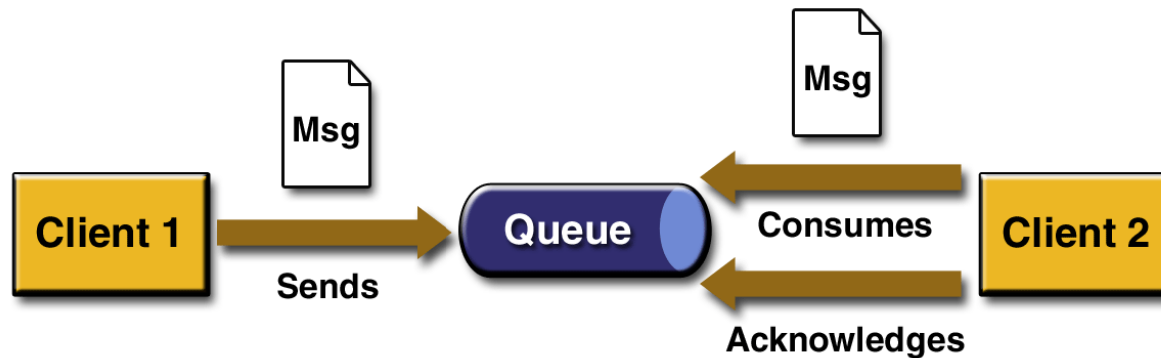


Introduction

Un **message broker** est un agent intermédiaire qui transmet des messages entre un producteur et consommateur.

Les messages brokers sont souvent utilisés pour l'intégration entre systèmes et fournissent le service coeur des MOM, des EAI ou des ESBs

Modèles de communication





Avantages attendus

Un message broker permet de découpler le client du serveur (producteur/consommateur).

=> L'émetteur et le récepteur n'ont pas de dépendances (si ce n'est le format du message) et le traitement du message est asynchrone

Ce faible couplage favorise la montée en charge et l'évolutivité de l'architecture.

- Modification du traitement du message
- Scalabilité des récepteurs
- Ajout de nouveaux destinataires
- Ajout d'information dans le message



MOM

Les **MOM (*Message Oriented Middleware*)** permettent de définir les différents canaux de distribution (file ou topic) et garantir certaines propriétés :

- Garantie de délivrance
- Transactions
- Scalabilité
- Routage

De plus, différentes Qualité de Service peuvent être configurées sur les destinations

- Dimensionnement (~capacité de bufferisation)
- Dead-letter
- Ordre des messages, priorité
- Durée de vie



Standards

JavaEE a bien proposé le standard *JMS* mais cela ne permettait que d'intégrer du Java entre eux.

Les éditeurs se sont alors tournés vers d'autres spécifications comme **AMQP** (*Advanced Message Queuing Protocol*) qui n'ont pas d'exigence sur le langage

=> La flexibilité multilingue est devenue réelle pour les courtiers de messages open source.

D'autres protocoles existent : STOMP, MQTT



Produits

AWS Simple Queue Service (SQS),
Azure Service Bus

Apache ActiveMQ, Fuse (Pro)

HornetQ (RedHat), IBM MQ, Oracle
Message Broker, TIBCO, WSO2

Redis

Apache Kafka

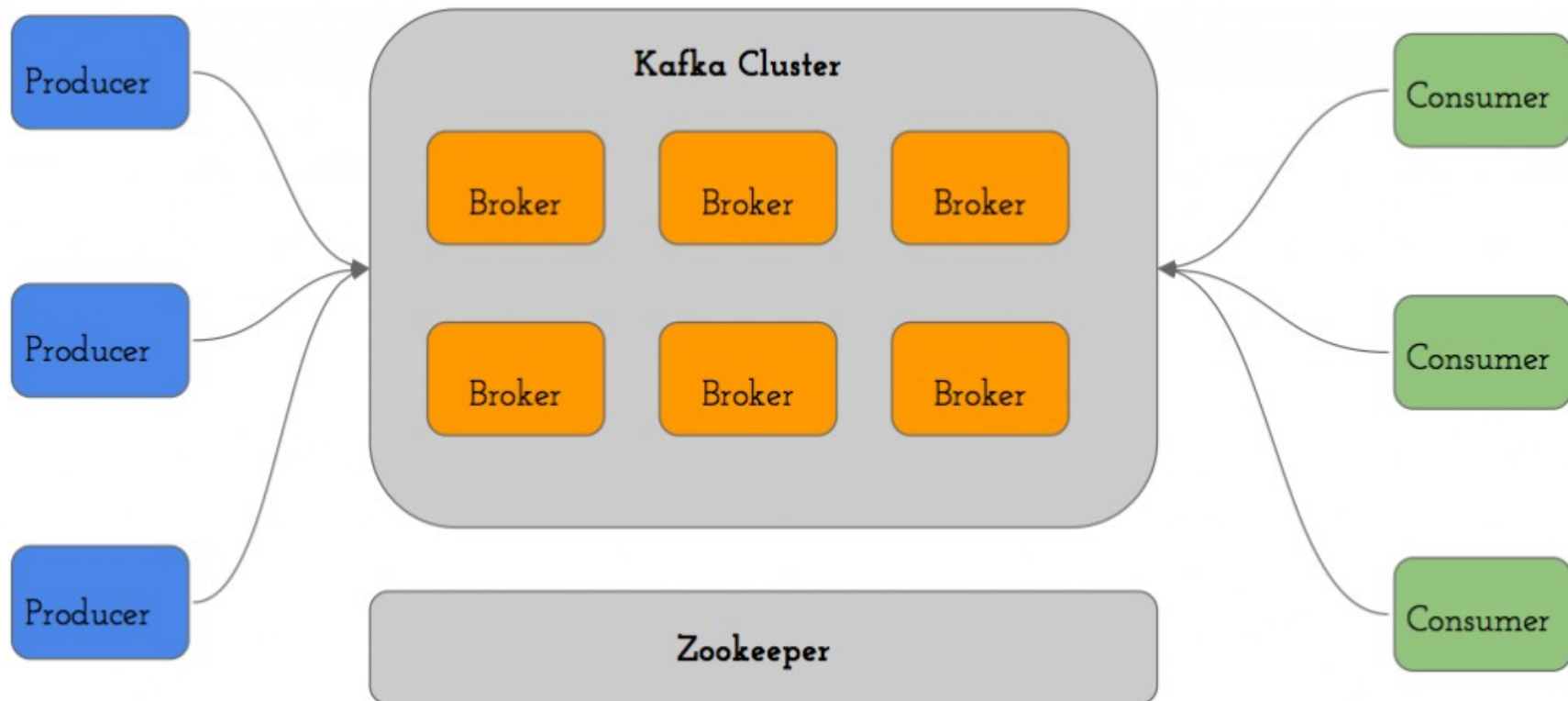


Spécificités de Kafka

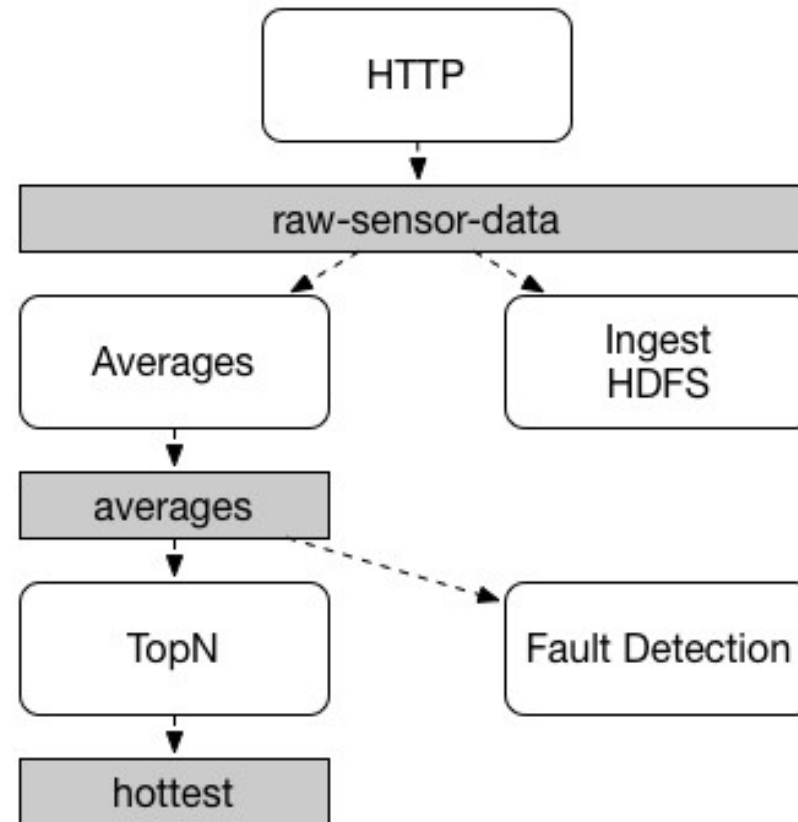
Kafka réinvente le sujet en l'adaptant au BigData :

- Ne supporte que le modèle Pub&Sub
- A la différence des autres brokers, les messages ne sont pas supprimés lorsqu'ils ont été reçus par leur destinataires mais après une date de péremption
=> Les messages peuvent donc être traités à posteriori et on peut donc rejouer un historique
- Taillé pour le BigData, il fonctionne en cluster et est capable de conserver et traiter un volume très important de données

Kafka



Exemple d'architecture micro-service



TopN et *FaultDetection* ont été implémentés dans un 2ème temps



Architectures applicatives

Services back-end

Message brokers

Architecture Micro-services

Frameworks clients

BigData, Machine Learning



Architecture orientée services

Les architectures orientées services préconisent la construction d'application métier en composant des services à moyenne ou petite granularité, faiblement couplés

Les avantages attendus :

- Choix des technologies pour chaque service
- Évolutivité facilitée des services



Architectures SOA

Les **architectures SOA** s'appuient sur un ensemble de services web de granularité moyenne.

Chaque service expose une **interface** à **couplage faible** offrant un accès à une fonctionnalité **métier** ou **technique**.

On distingue :

- le SOA de surface : Ajout de services web sur des applications legacy à fins d'intégration
- Le SOA de profondeur : Conception d'une système via une décomposition en service



Historique

Ces types d'architecture ont vu le jour avec **SOA (Service Oriented Architecture)** basé principalement sur les services SOAP

La lourdeur de SOAP et des des serveurs applicatifs ont freiné l'adoption de SOA

Avec DevOps, les technologies REST et les conteneurs les architectures « **micro-services** » visant à améliorer la rapidité des déploiements et des retours utilisateur se répand.



Architecture

Une architecture micro-services implique la décomposition des applications en très petits services

- faiblement couplés
- ayant une seule responsabilité
- Développés par des équipes full-stack indépendantes.



Caractéristiques

Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

DURS (Deploy, Update, Replace, Scale) indépendants : Chaque service peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket,



Bénéfices

Scaling indépendant : les services les plus sollicités (cadence de requête, mutualisation d'application) peuvent être scalés indépendamment (CPU/mémoire ou sharding),

Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes

Maintenance facilitée : Le code d'un micro-service est limité à une seule fonctionnalité

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Un service dysfonctionnant ne pénalise pas obligatoirement le système complet.

Communication inter-équipe renforcée : Full-stack team



Contraintes

Réplication : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte automatique : Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certaines métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

Monitoring : Les services sont surveillés en permanence. Des traces sont générées et éventuellement agrégées

Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



Les 12 facteurs

- I. **Codebase** : Une base de code suivie dans le contrôle de révision, de nombreux déploiements
- II. **Dependencies** : Déclarer et isoler explicitement les dépendances
- III. **Config** : Stocker la config dans l'environnement
- IV. **Backing services** : Traiter les services de support comme des ressources attachées
- V. **Build, release, run** : Séparer les étapes de construction et d'exécution
- VI. **Processes** : Exécuter l'application en tant qu'un ou plusieurs processus stateless
- VII. **Port binding** : Exposer les services via des associations de port
- VIII. **Concurrency** : Scaler via les processus
- IX. **Disposability** : Maximiser la robustesse avec un démarrage rapide et un arrêt clean
- X. **Dev/prod parity** : Gardez le développement, la pré-prod et la production aussi semblables que possible
- XI. **Logs** : Traiter les journaux comme des flux d'événements
- XII. **Admin processes** : Exécuter les tâches d'administration comme un unique processus



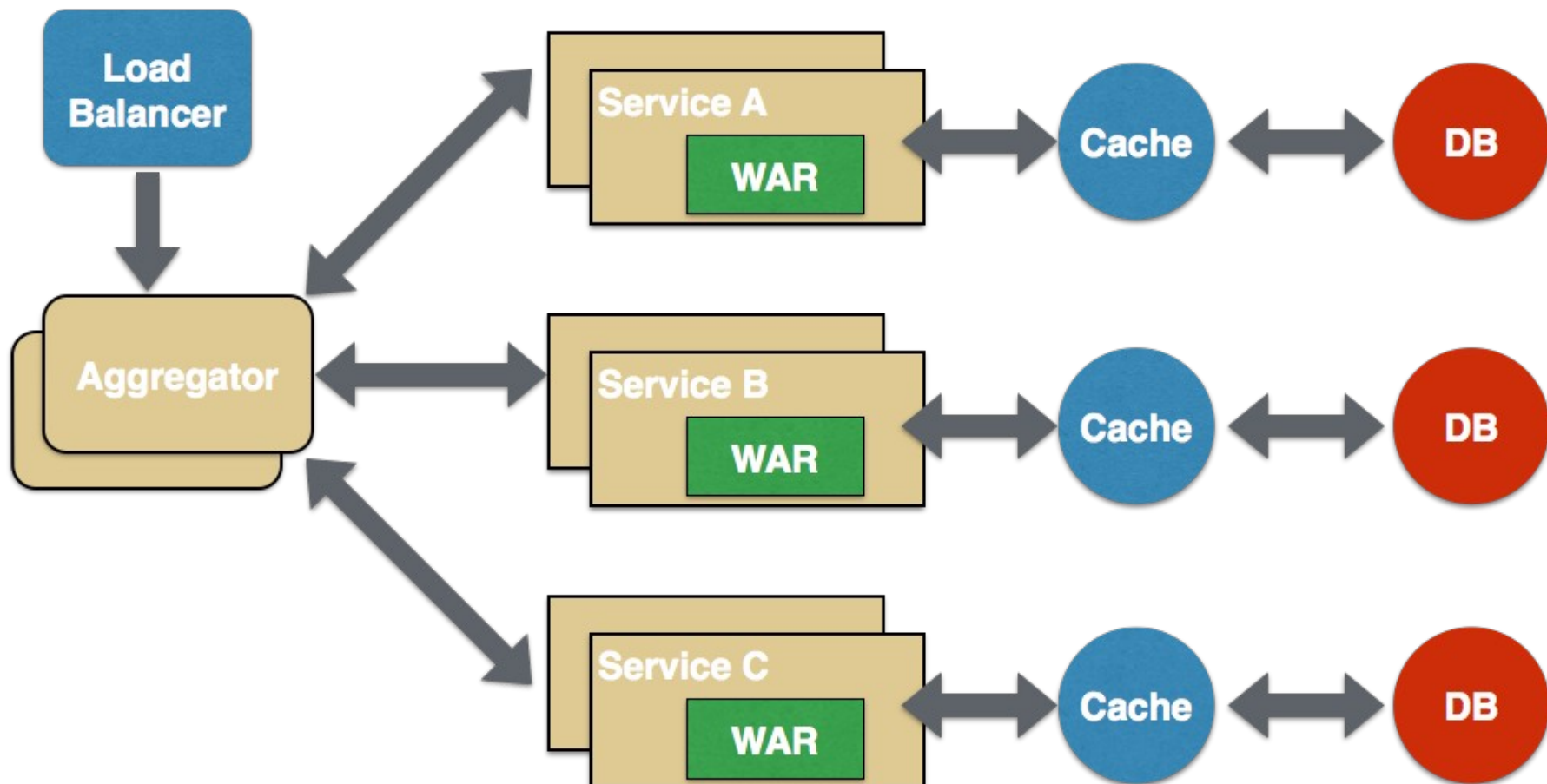
Patterns de composition

Les micro-services peuvent être combinés afin de fournir un micro-service composite.

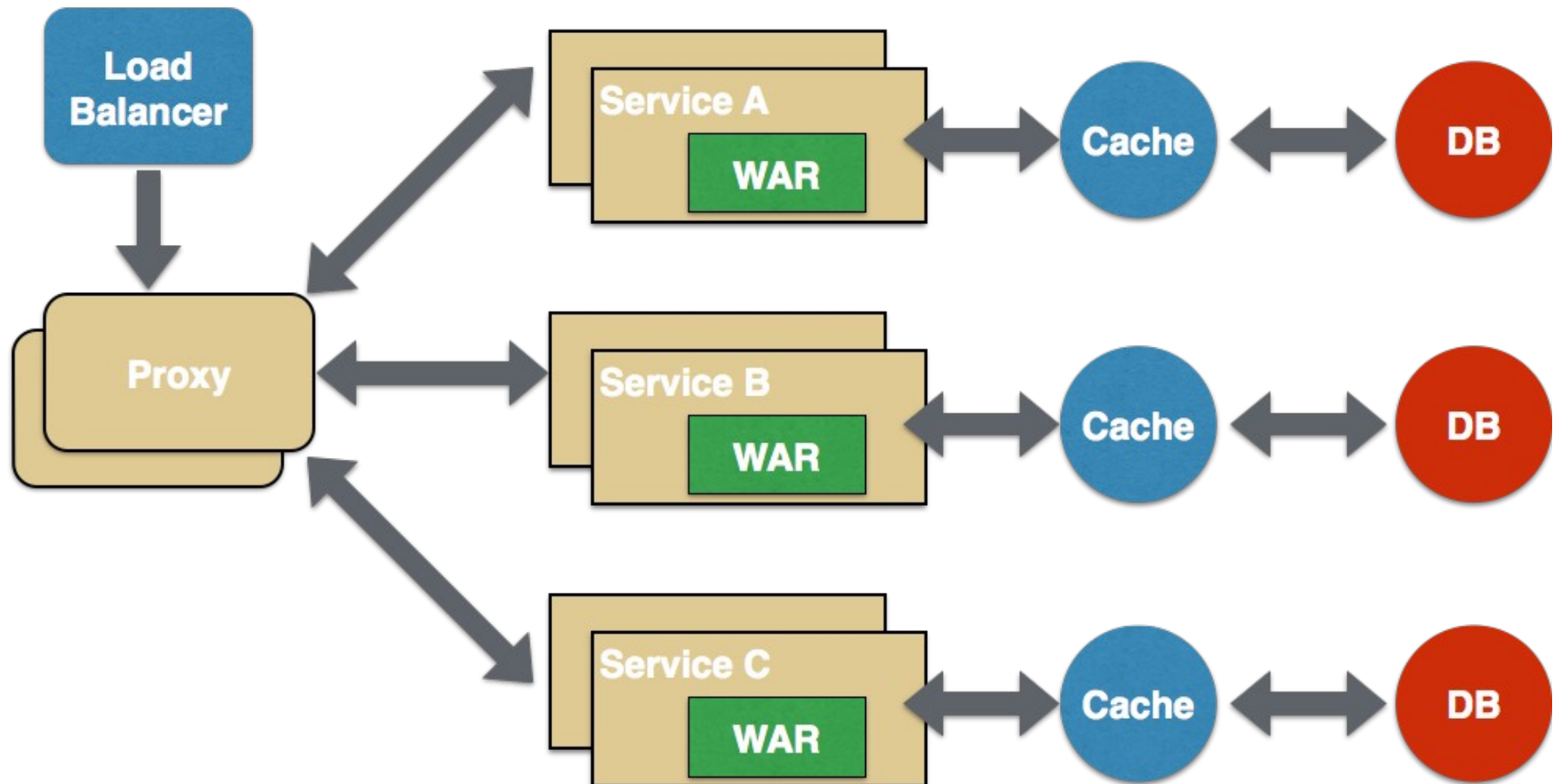
Comme design patterns communs, citons :

- **L'agrégateur** : Agrégation de plusieurs micro-service et fourniture d'une autre API REST
- **Proxy** : Délégation à un service caché avec éventuellement une transformation
- **Chaîne** : Réponse consolidée à partir de plusieurs sous-services
- **Branche** : Idem agrégateur avec le parallélisme

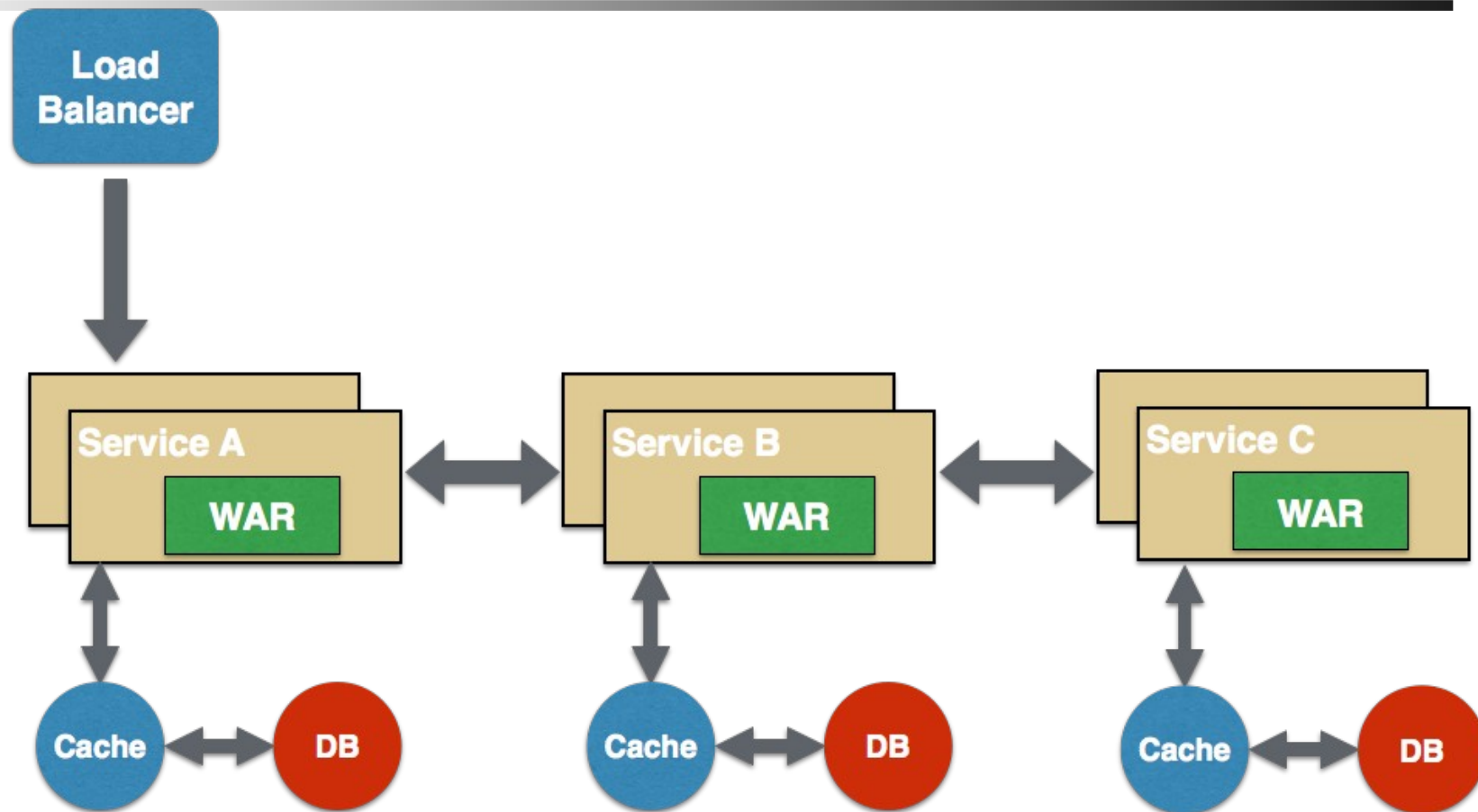
Agrégateur



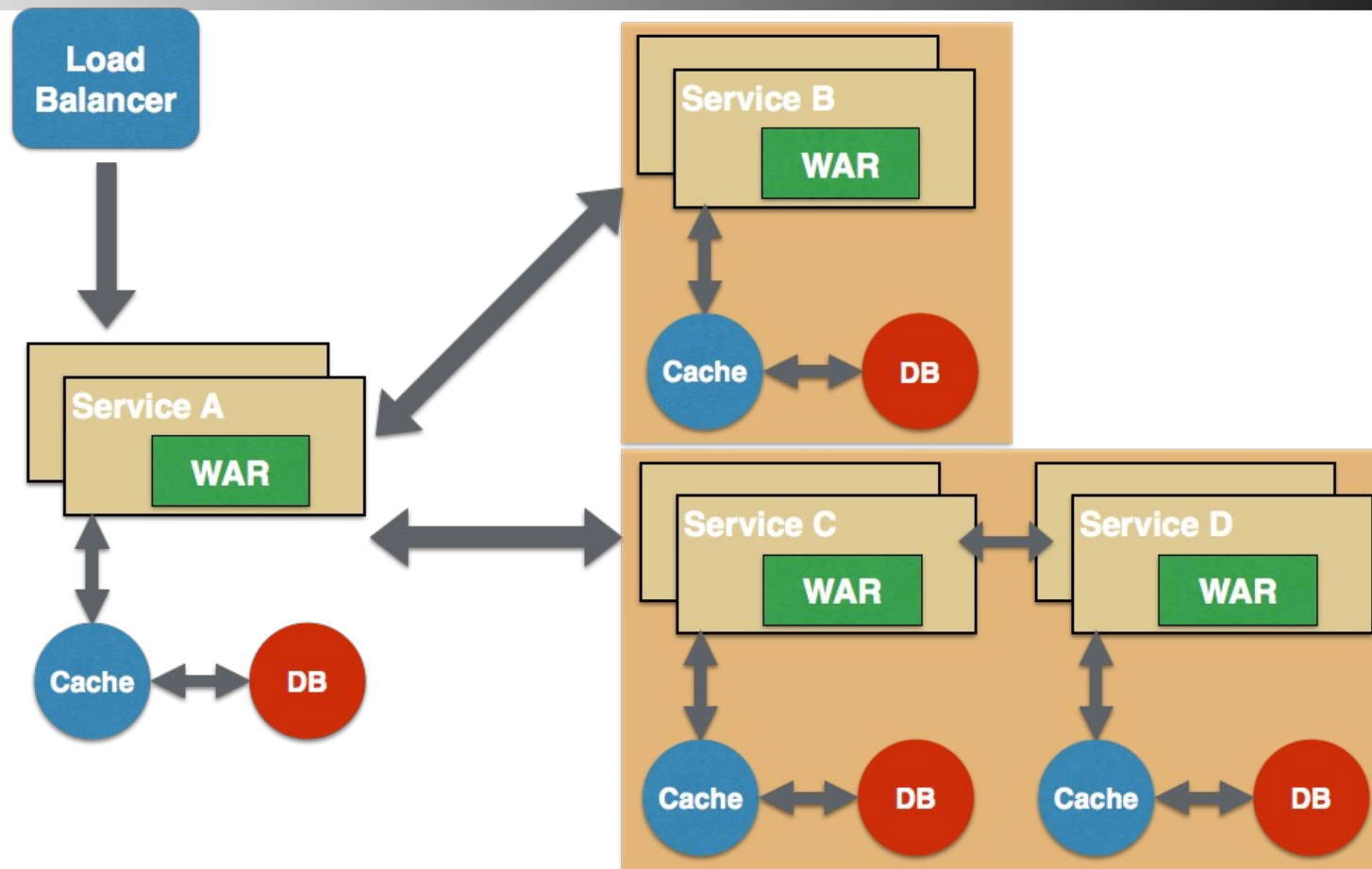
Proxy



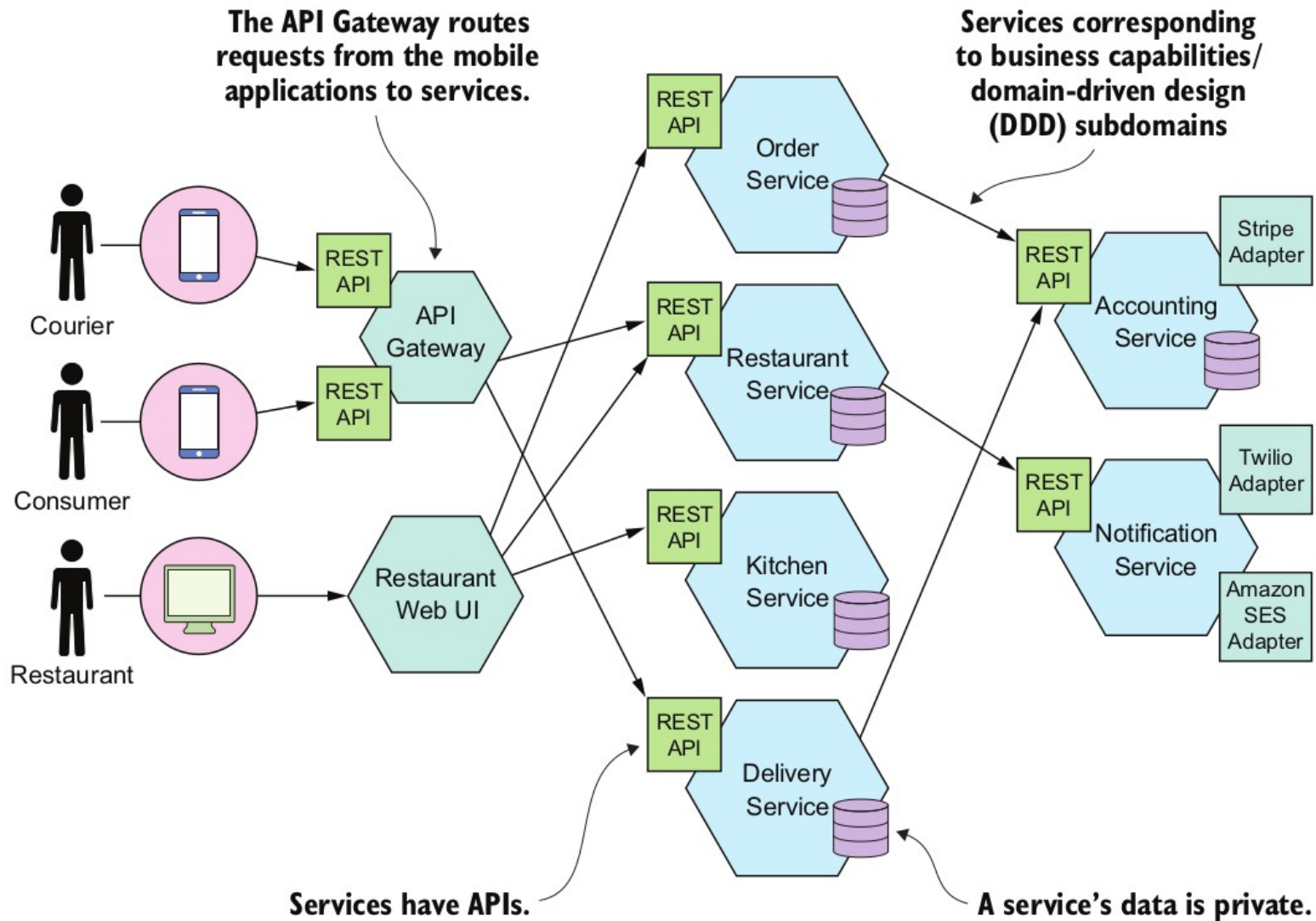
Chaîne



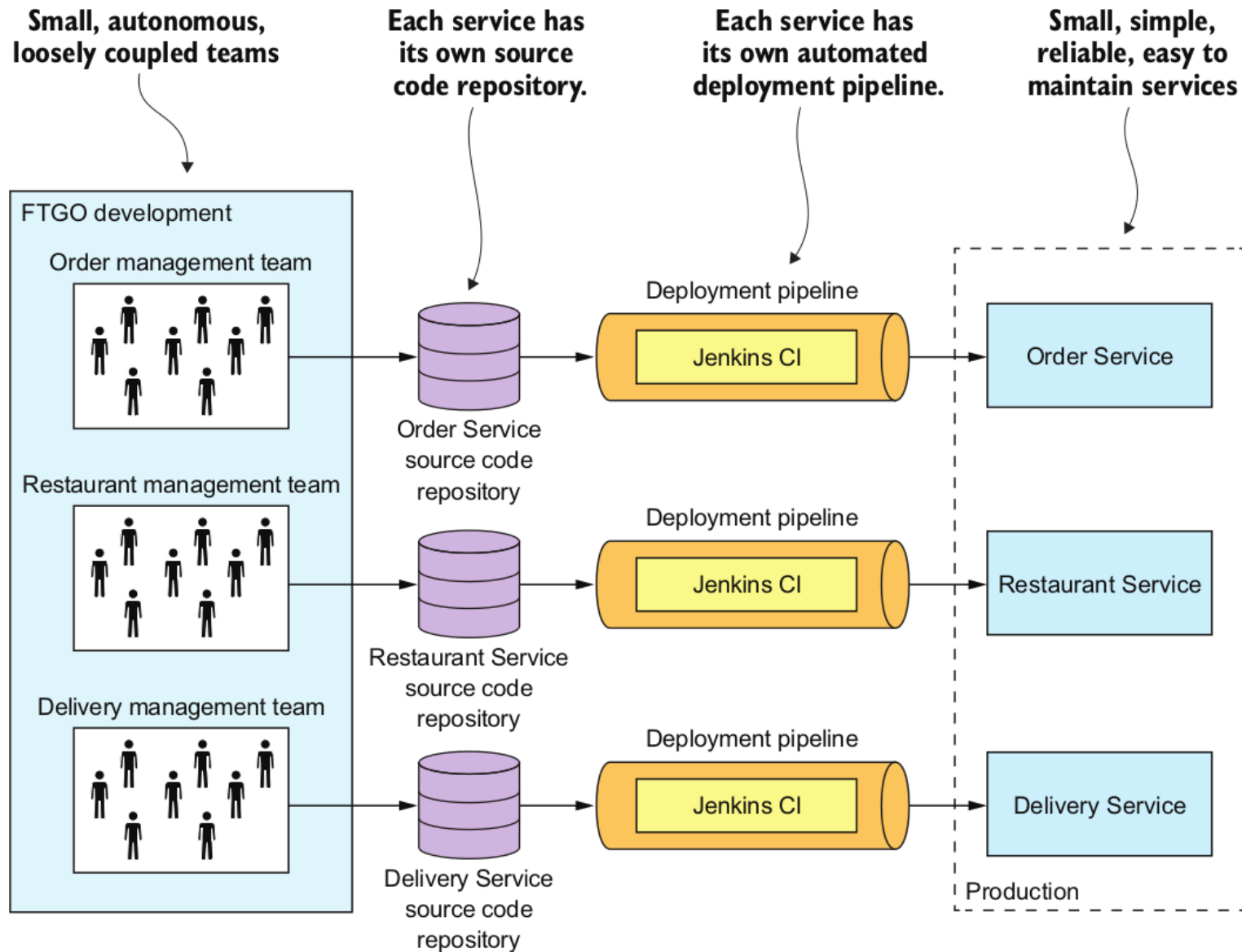
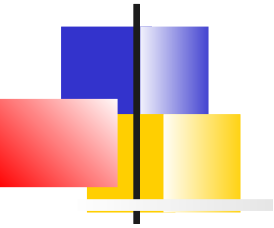
Branche



Une architecture micro-service



Organisation DevOps





Design Patterns

Des efforts de formalisation de design patterns pour les micro-services ont été entrepris.

Ils concernent :

- L'infrastructure : Problème concernant exclusivement l'infrastructure.
- Infrastructure applicative : Problèmes d'infrastructure ayant des impacts sur le développement
- Application : Problèmes de développeurs.



Pattern d'infrastructure

Problèmes d'infrastructure

- Le déploiement
- La découverte des services
- L'accès aux APIs des services

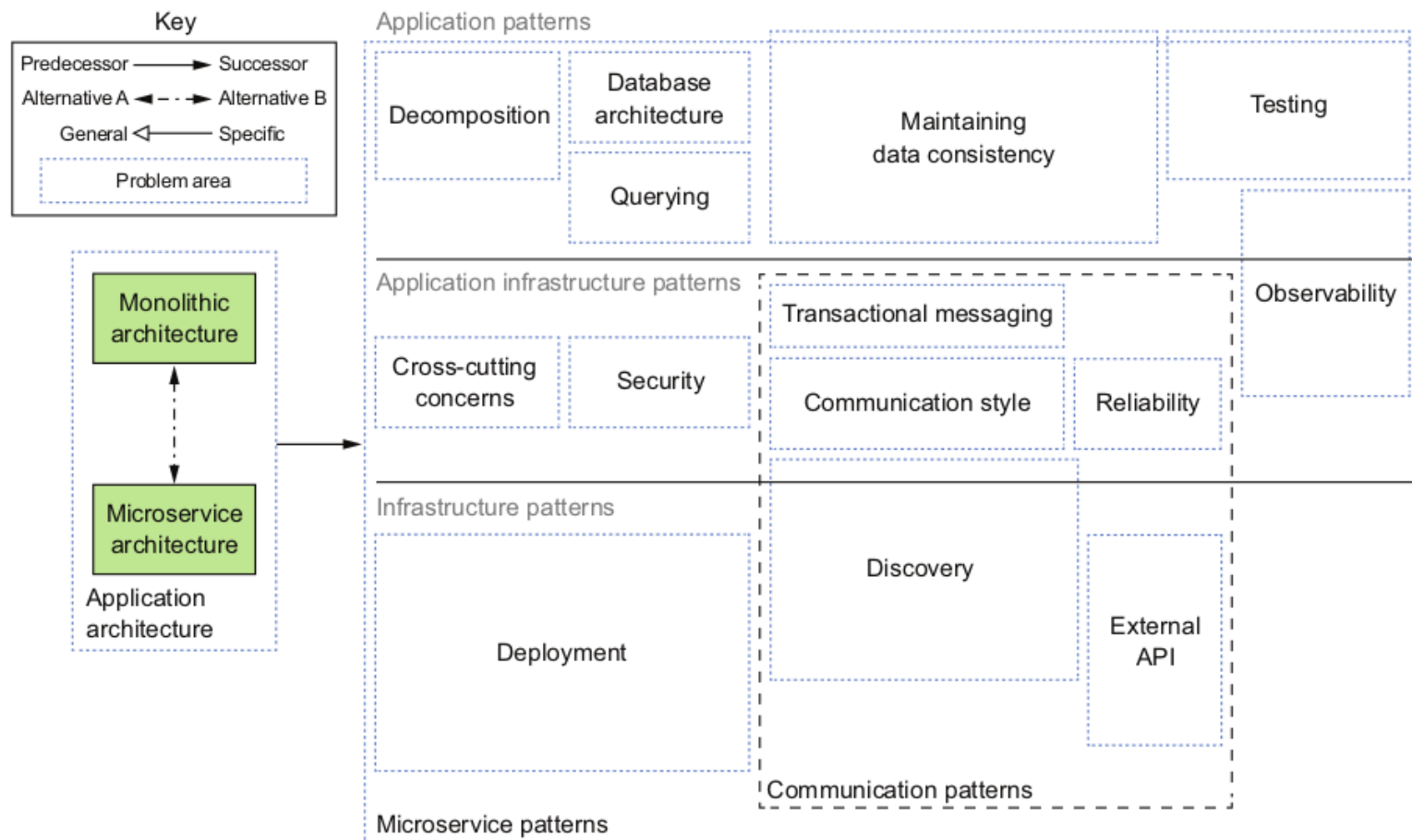
Infrastructure applicative :

- Sécurité
- Services transverses (Tracing, profiling, etc..)
- Messagerie transactionnelle
- Styles de communication (RPC, ...)
- Fiabilité
- Observabilité

Problèmes applicatifs :

- La décomposition
- Architecture des bases de données
- La recherche de données (Querying)
- Maintien de la cohérence des données
- Testabilité

Classification des patterns





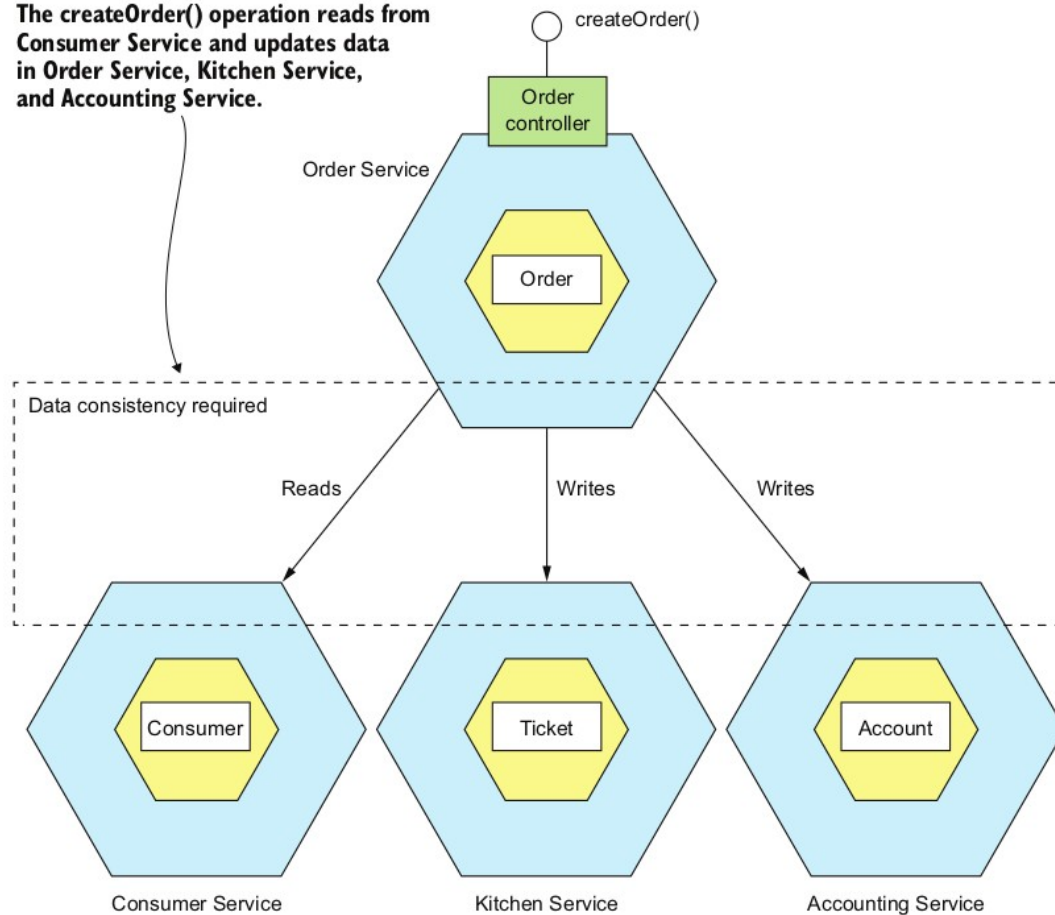
Exemple : Gestion des transactions avec Saga

Problème : Comment implémenter des transactions qui englobent plusieurs services, i.e des opérations qui mettent à jour des données dispersées dans plusieurs services

Solution : Une **saga**, séquence de transactions locales basée sur des messages. Cette séquence présente les propriétés ACD (Atomicité, Cohérence, Durabilité) mais la propriété d'isolation n'est pas respectée. En conséquence, l'application doit utiliser des contre-mesures (opération de compensation) pour empêcher ou réduire l'impact des anomalies de concurrence.

Example

The createOrder() operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.





Structure des transactions

Une saga consiste en une séquences de 3 types de transactions :

- **Transaction compensable** : peuvent potentiellement être annulées à l'aide d'une transaction compensatoire
- **Transaction pivot** : Si elle est validée, la saga s'exécutera jusqu'à la fin.
Une transaction pivot peut être une transaction qui n'est ni compensable, ni réessayable. Cela peut s'agir de la dernière transaction compensable ou de la première transaction réessayable
- **Transaction réessayable** : Elles suivent la transaction pivot transaction et sont assurés de réussir



Exemple

Etapes	Service	Transaction	Tr. de compensation
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetail()	-
3	KitchenService	createTicket()	rejectTicket()
4	AccountingService	authorizeCreditCard()	
5	RestaurantService	ackRestaurantOrder()	
6	OrderService	approveOrder()	

- 1,2,3 : Sont des transactions compensables
- 2 : Une opération de lecture n'a pas besoin de compensation
- 4 : Transaction pivot. Si elle réussit, *createOrder* doit aller jusqu'à la fin
- 5,6 : Transaction réessayable, jusqu'à ce qu'elles réussissent

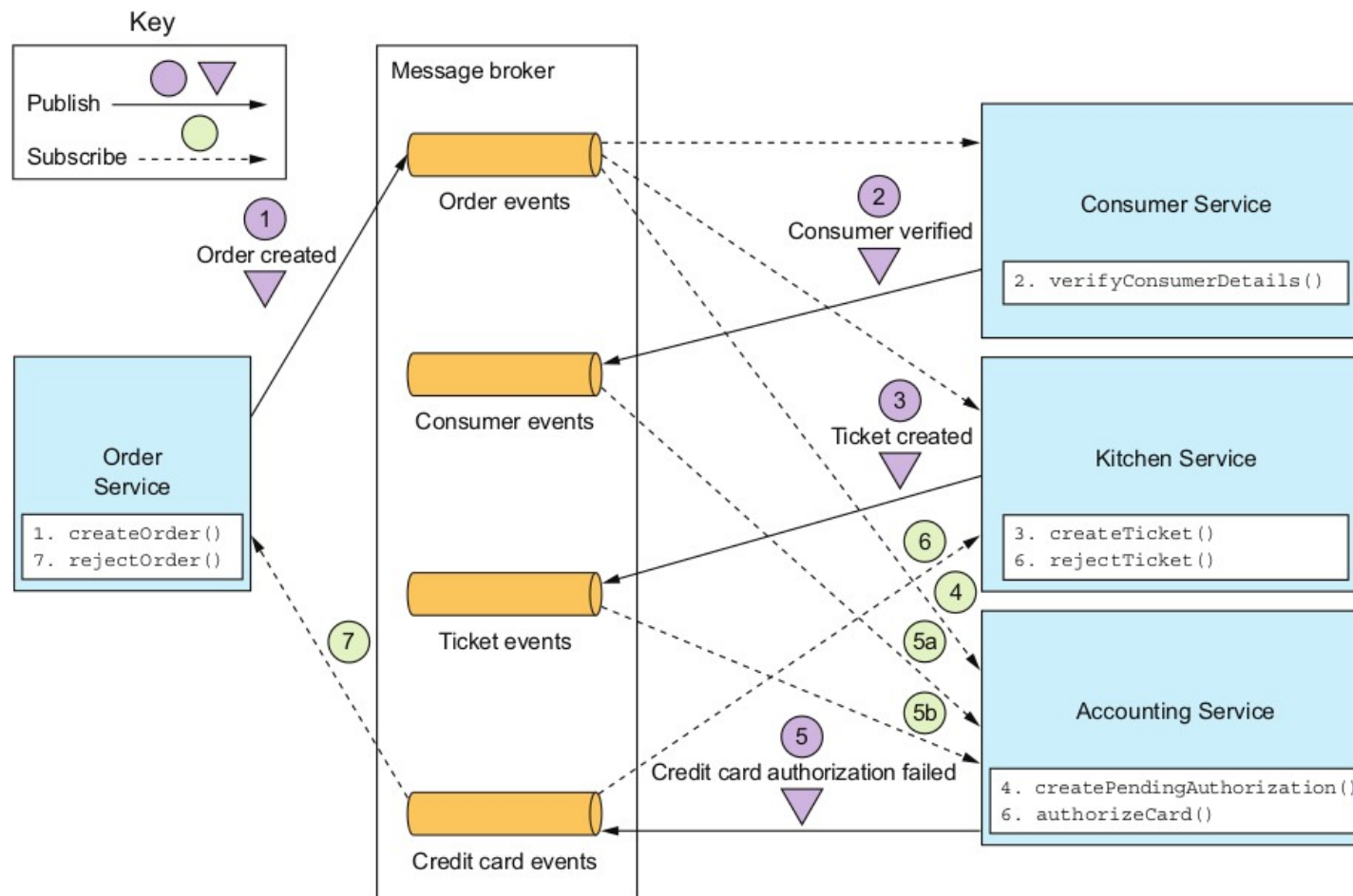


Implémentation

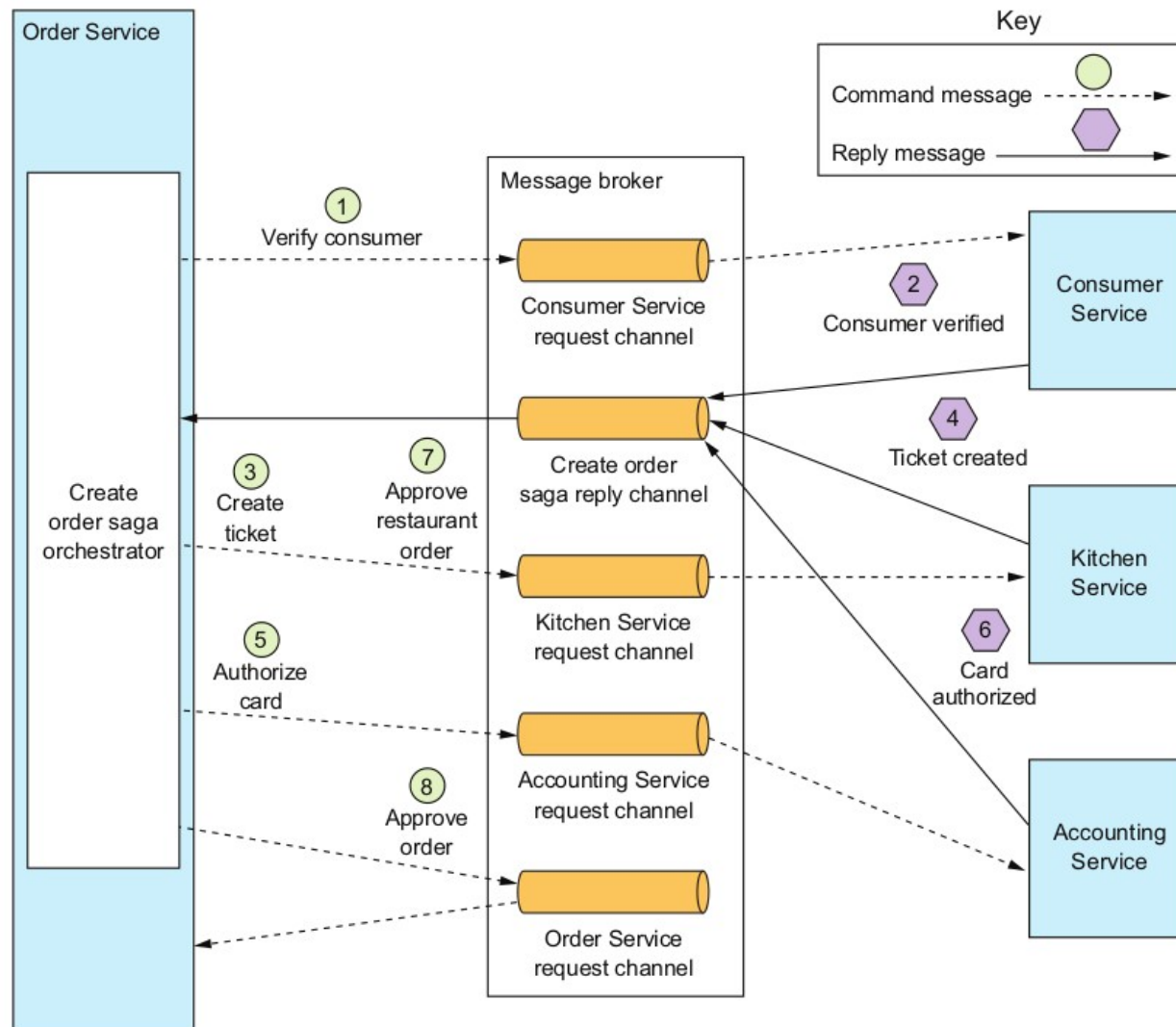
Il existe 2 façons de coordonner une saga:

- **chorégraphie** - les participants à la saga échangent des événements
- **orchestration** - un orchestrateur centralisé utilise une messagerie asynchrone pour piloter les participants

Chorégraphie



Orchestration





Autres patterns

Organisation de la logique métier :

- Transaction script, Domain model, DDD aggregate
- Event sourcing

Requêtage :

- Command query responsibility segregation :
Implémenter une requête qui a besoin de données de plusieurs services en utilisant des événements pour maintenir une vue en lecture seule qui réplique les données des services

Fiabilité :

- Circuit breaker
- Replication pattern : Active/Passive, Multiple master et consensus

API Externe :

- API Gateway



Interface utilisateur



Architecture Client/Serveur

Les environnements comme Delphi, PowerBuilder, Forms s'appuient sur des environnements graphiques permettant de designer des formulaires et leurs contrôles.

Assessment Record

MK76Y	Station ID	NV141
NV140	Date	5/26/2006
NV141	Target	42
NV142	Actual	33
NV143	Variance	-9
RLD8		
RLD9		
RLD14		
RLD15		
RN341		
RN342		
RN451		
RN452		



Data Binding

Les données manipulées sont présentes à 3 endroits et doivent être synchronisées :

- La base de données
- En mémoire, ou session
- Dans les contrôles

Les frameworks permet en général de faire du *DataBinding* uni ou bidirectionnel avec un *ResultSet*



Pattern Observer

Généralement, le formulaire déclare les événements qu'il veut écouter et le développeur peut associer l'exécution d'une fonction au déclenchement de l'événement.

Le formulaire a des références directes sur les contrôles qu'il contient



En résumé

Les développeurs écrivent des formulaires spécifiques aux applications qui utilisent des contrôles génériques.

Le formulaire décrit la disposition des contrôles.

Le formulaire observe les contrôles et dispose de méthodes de gestion permettant de réagir aux événements intéressants déclenchés par les contrôles.

Les éditions simples de données sont gérées via le data binding.

Les modifications complexes sont effectuées dans les méthodes de gestion des événements du formulaire.



MVC *Smalltalk 80*

Motivation : Les objets du domaine doivent être complètement autonomes et fonctionner sans référence à la présentation.

Le rôle du contrôleur est de prendre les entrées de l'utilisateur et de déterminer quoi en faire.

Il y a une paire vue-contrôleur pour chaque élément de l'écran.

Dans un contexte de réutilisation de composants d'interface, la paire vue/contrôleur forme le composant graphique.

Toutes les vues et les contrôleurs observent le modèle. Lorsque le modèle change, les vues réagissent.

A la différence du client/serveur, le contrôleur ignore parfaitement ce que les autres widgets doivent changer lorsque l'utilisateur manipule un widget particulier



En résumé

Fait une séparation forte entre la présentation (vue & contrôleur) et le domaine (modèle)

Partitionne les widgets dans un contrôleur et une vue. Le contrôleur et la vue ne doivent pas communiquer directement mais par le biais du modèle.

Demandez aux vues d'observer le modèle pour permettre à plusieurs widgets de se mettre à jour



Variante MVVM (Model/View/ViewModel)

Déclinaison de MVC. Garde la présentation séparée et la synchronisation d'observateur.

Les vues ne se synchronisent pas directement à partir des objets du modèle mais à partir d'un modèle intermédiaire ; le **modèle de vue**.

Il n'y a plus de contrôleur, le DataBinding prend en charge la synchronisation entre la vue et les propriétés/méthodes du modèle de vue.



Model-View-Presenter (MVP)

Déclinaison du MVC

La vue est une arborescence de composants qui n'ont pas de contrôleurs associés

Tous les événements sont routés au présentateur

Le présentateur agit sur le modèle **et** la vue (à la différence du contrôleur). Il extrait les données des référentiels (le modèle) et les formate pour les afficher dans la vue.



Angular

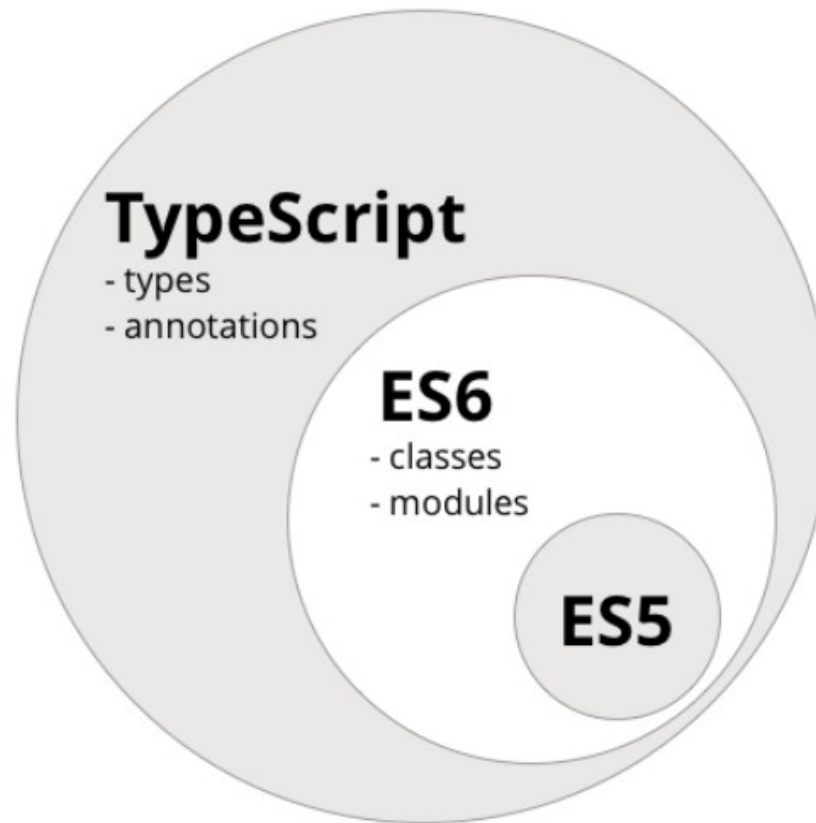
Développement principalement en
TypeScript,

Cible Web et Mobile

Concepts cœur :

- Approche composant (réutilisable, encapsulation)
- Binding, Dependency Injection, appels asynchrones
- Pattern : MV* ? : MVW (Model-View-Whatever Works For You)

TypeScript



ES5, ES6, and TypeScript



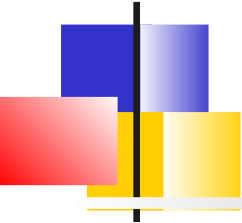
Modèle de programmation

L'interface est constitué de **composants**.

Les composants sont packagés en **modules** qui déclare les composants, les services

Chaque composant est une classe TypeScript qui

- Est associé à un sélecteur HTML (~balise), un gabarit HTML, un style
- Contient le modèle de données associé au composant
- Contient les méthodes gérant les interactions qui peuvent s'appuyer sur des services back-end injectés



Exemple Component *<my-app>*

```
<body>
  <my-app>Loading...</my-app>
</body>
```

```
-----

import { Component }      from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <nav>
      <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['app/app.component.css']
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```



Exemple composant

```
import { Component, OnInit } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';

import { Critere } from './critere.model';
import { CritereService } from './critere.service';

@Component({
  selector: 'my-critere',
  templateUrl: './critere.component.html'
})
export class CritereComponent implements OnInit {
  criteres: Critere[];
  currentAccount: any;

  constructor(private critereService: CritereService) { }

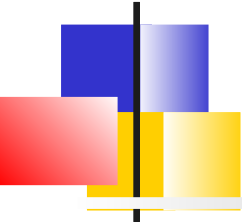
  loadAll() {
    this.critereService.query().subscribe(
      (res: ResponseWrapper) => {
        this.criteres = res.json;
      },
      (res: ResponseWrapper) => this.onError(res.json)
    );
  }

  ngOnInit() {
    this.loadAll();
    this.principal.identity().then((account) => {
      this.currentAccount = account;
    });
    this.registerChangeInCriteres();
  }
}
```



Gabarit

```
<div class="table-responsive" *ngIf="criteres">
  <table class="table table-striped">
    <tbody>
      <tr *ngFor="let critere of criteres ;">
        <td>
          <a [routerLink]="['../critere',
critere.id ]">{{critere.id}}</a></td>
          <td>{{critere.nom}}</td>
        </tr>
      </tbody>
    </table>
  </div>
```



Service (CRUD)

@Injectable()

```
export class CritereService {

    private resourceUrl = SERVER_API_URL + 'api/criteres';

    constructor(private http: Http) { }
    create(critere: Critere): Observable<Critere> {
        const copy = this.convert(critere);
        return this.http.post(this.resourceUrl, copy).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    update(critere: Critere): Observable<Critere> {
        const copy = this.convert(critere);
        return this.http.put(this.resourceUrl, copy).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    find(id: number): Observable<Critere> {
        return this.http.get(`${this.resourceUrl}/${id}`).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    delete(id: number): Observable<Response> {
        return this.http.delete(`${this.resourceUrl}/${id}`);
    }
}
```



Service (Sérialisation)

```
private convertResponse(res: Response): ResponseWrapper {
    const jsonResponse = res.json();
    const result = [];
    for (let i = 0; i < jsonResponse.length; i++) {
        result.push(this.convertItemFromServer(jsonResponse[i]));
    }
    return new ResponseWrapper(res.headers, result, res.status);
}

/**
 * Convert a returned JSON object to Critere.
 */
private convertItemFromServer(json: any): Critere {
    const entity: Critere = Object.assign(new Critere(), json);
    return entity;
}

/**
 * Convert a Critere to a JSON which can be sent to the server.
 */
private convert(critere: Critere): Critere {
    const copy: Critere = Object.assign({}, critere);
    return copy;
}
```



Angular cli

Le développement est facilité par l'outil *Angular CLI* qui permet

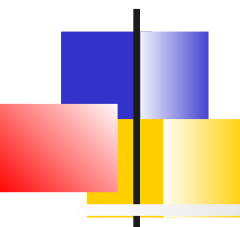
- de générer la structure de l'application, d'un composant
- De gérer les dépendances et le build du projet
- De gérer un profil de développement et un profil de production
- De déployer



Annexes



Docker



Introduction

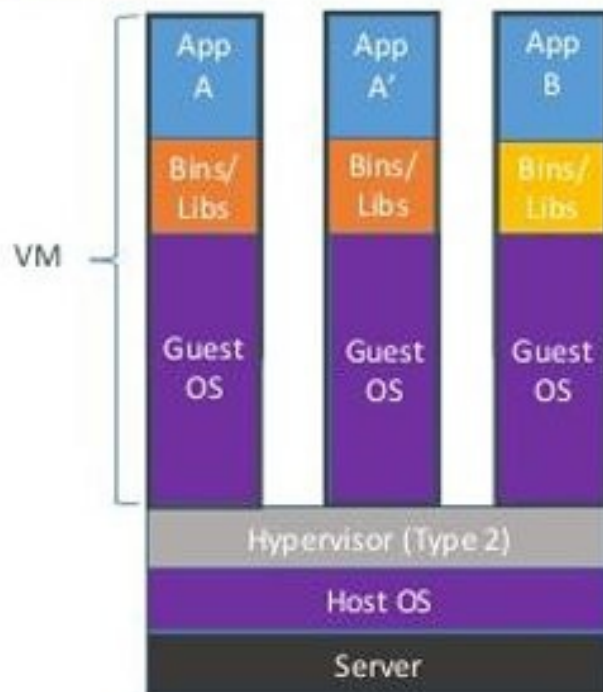
Plutôt que de virtualiser une machine complète, il n'est nécessaire que de créer un environnement d'exécution dans lequel un service est démarré pour fournir l'application.

Le but des conteneurs applicatifs est donc d'isoler un processus dans un système de fichiers à part entière - il n'est plus question de simuler le matériel et les services d'initialisation du système d'exploitation sont ignorés.

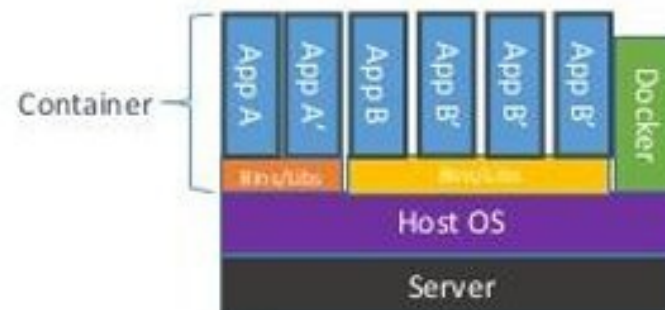
Seul le strict nécessaire réside dans le conteneur, à savoir l'application cible et quelques dépendances.

Containers vs VMs

Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries





Impact sur le déploiement

Les développeurs et la PIC travaillent alors avec la même image de conteneur que celle utilisée en production.

- => Réduction considérable du risque de dysfonctionnements dû à une différence de configuration logicielle.

Il n'y a plus à proprement parler un déploiement brut sur un serveur mais plutôt l'utilisation d'orchestrateur de conteneurs.



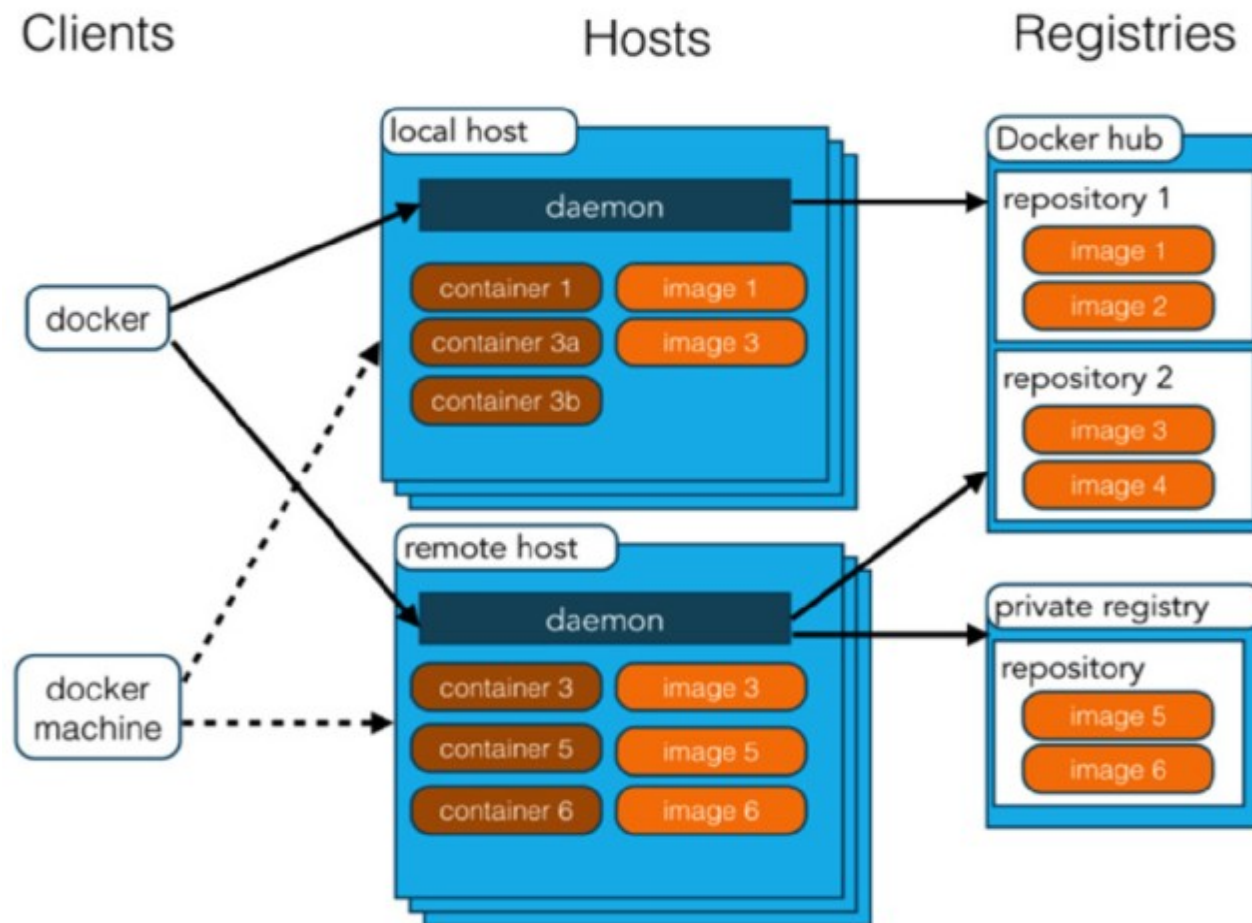
Orchestration de conteneurs

Un simple fichier "manifest" définit comment démarrer un conteneur et la configuration nécessaire.

L'orchestrateur de service va alors trouver une machine disponible, démarrer l'application et faire le nécessaire pour qu'elle soit accessible

Il va permettre également la "mise à l'échelle" (ou scaling) en fonction de la charge courante.

Docker architecture





Commandes Docker

#Récupération d'une image

```
docker pull ubuntu
```

#Récupération et instanciation

```
docker run hello-world
```

#Mode interactif

```
docker run -i -t
```

#Visualiser les sortie standard d'un conteneur

```
docker logs <container_id>
```

#Conteneurs en cours

```
docker ps
```

#Toutes les exécutions de conteneurs (même arrêt)

```
docker ps -a
```

#Lister les images

```
docker images
```

#Construire une image à partir d'un fichier Dockerfile

```
docker build . -t monImage
```

#Committer les différences

```
docker commit <container_id> <image_name>
```

#Tagger une image d'un repository

```
docker tag <image_name>[:tag] <name>[:tag]
```

#Pousser vers un dépôt distant

```
docker push <image_name>[:tag]
```

#Statistiques d'usage des ressources

```
docker stats
```



Exemple DockerFile

FROM ubuntu

MAINTAINER Kimbro Staken

RUN apt-get install -y software-properties-common python

RUN add-apt-repository ppa:chris-lea/node.js

RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >> /etc/apt/sources.list

RUN apt-get update

RUN apt-get install -y nodejs

RUN mkdir /var/www

ADD app.js /var/www/app.js

EXPOSE 8080

CMD ["/usr/bin/node", "/var/www/app.js"]



Isolation du conteneur

Chaque conteneur s'exécute à sa propre interface réseau, son propre système de fichiers (gérés par Docker)

Par défaut, il est isolé

- De la machine hôte
- Des autres containers



Communication avec la machine hôte

Au démarrage d'un conteneur on peut :

- Associer un port exposé par le conteneur à un port local

Option **-p**

- Monté un répertoire du conteneur sur le système de fichier local.

Option **-v**

```
docker run -p 80:8080  
           -v /home/jenkins:/var/lib/jenkins  
           myImage
```



docker-compose

docker-compose est un outil pour définir et exécuter des applications Docker utilisant plusieurs conteneurs

- Avec un simple fichier texte, on spécifie les différents conteneurs, les ports exposés, les liens entre conteneurs, les volumes montés.
- Ensuite avec une commande unique, on peut démarrer, arrêter, redémarrer l'ensemble des services.



Exemple configuration

Le fichier de configuration définit des services, des networks et des volumes.

version: '2'

services:

annuaire:

build: ./annuaire/ **# context de build, présence d'un Dockerfile**

networks:

- back
- front

ports:

- "1111:1111" **# Exposition de port**

documentservice:

build: ./documentService/

networks:

- back

proxy:

build: ./proxy/

networks:

- front

ports:

- 8080:8080

Analogue à 'docker network create'

networks:

back:

front:



Commandes

build : Construire ou reconstruire les images

config : Valide le fichier de configuration

down : Stoppe et supprime les conteneurs

exec : Exécute une commande dans un container up

logs : Visualise la sortie standard

port : Affiche le port public d'une association de port

pull / **push** : Pull/push les images des services

restart : Redémarrage des services

scale : Fixe le nombre de container pour un service

start / **stop** : Démarrage/arrêt des services

up : Création et démarrage de conteneurs



Sécurité

Techniques associées



Définitions

ISO 25000 définit 5 sous-caractéristiques pour la sécurité :

- **Confidentialité** : Un système garantit que les données ne sont accessibles qu'aux personnes autorisées à y accéder
- **Intégrité** : Un système empêche la modification des programmes ou des données.
- **Non-repudiation** : Mesure par laquelle il peut être prouvé que des actions ou des événements ont eu lieu, de sorte qu'ils ne puissent pas être répudiés ultérieurement.
- **Authentification** : L'identité d'un sujet ou d'une ressource peut être prouvée être celle qui est revendiquée.
- **Responsabilité** : Mesure par laquelle les actions d'une entité peuvent être attribuées uniquement à l'entité.



Approche en couche

Pour rendre l'ensemble du système plus sécurisé, il est recommandé de mettre en œuvre la sécurité indépendamment à chaque niveau ou couche du système.

Par exemple, pour une application web :

- BD: Utilisateur avec mot de passe fort et permissions minimales
- Couche métier : ACLs sur les méthodes en fonction des rôles métier de l'entreprise
- Couche Web : Protection contre attaques, ACLs sur les URLs en fonction des rôles des utilisateurs finaux
- Réseau : Cryptage des données, Certificats, Firewall

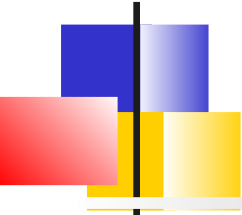


Message Digest

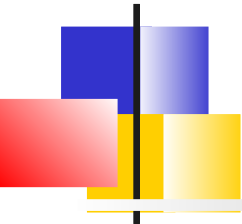
Une empreinte de message (*message digest*) condense en une suite d'octets (~40) un message de n'importe quelle longueur.

=> Cela permet de s'assurer de l'intégrité des données. Ex : SHA1

Cryptographie asymétrique

- 
- Utilisation d'une paire de clés :
 - une **publique** pouvant donc transiter sur le réseau
 - l'autre **privée** jalousement gardée dans un *keystore*
 - Pour décrypter les données cryptées avec une clé publique, il faut la clé privée et vice-versa

Cryptographie asymétrique (2)



- Cette technique permet essentiellement 2 choses :
 - Confidentialité/Intégrité du message à envoyer :
l'expéditeur utilise la clef publique du destinataire pour coder son message.
=> Seul le propriétaire de la clé privée peut décoder le message
 - Authentification :
L'expéditeur utilise sa clef privée pour coder un message
=> le destinataire peut décoder avec la clef publique de l'expéditeur et s'assurer de l'identité de l'émetteur
- Les algorithmes les plus connus actuellement sont DSA et RSA



Certificat numérique et autorité de certification

Un certificat est utilisé principalement pour identifier et authentifier une personne physique.

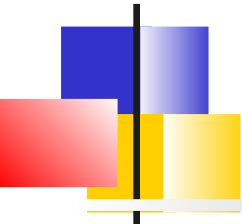
Il est signé par un tiers de confiance (ou chaîne de tiers de confiance) **qui atteste la relation entre la personne physique et l'identité numérique.**

Un certificat contient :

- au moins une clé publique ;
- des informations d'identification, par exemple : nom, localisation, adresse électronique ;
- au moins une signature (construite à partir d'une clé privée) : La sienne ou celle d'un tiers de confiance

Les clés publiques des autorités de certification racine sont pré-installés dans les navigateurs

Keystore, truststore



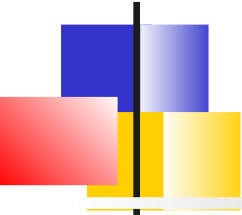
- Les **keystore et truststore** sont des bases de données protégées par un mot de passe stockant des certificats et/ou des clés.
 - Les *keystore* stockent généralement les certificats utilisés par ses serveurs web
 - Les *truststore* stockent généralement les certificats issus par des autorités en qui on a confiance
- Chaque entrée de la base est accessible via un alias



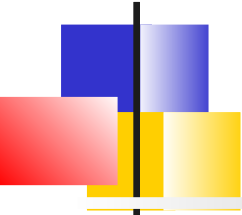
Chiffrement symétrique

- Les techniques de chiffrement **symétrique** (i.e, la même clé est utilisé pour le cryptage et le décryptage) sont plus performantes que les techniques asymétriques
mais sont confrontées au problème de la transmission sûre de la clé
=> Des techniques asymétriques sont utilisées pour crypter la clé symétrique à transmettre
- Les principaux algorithmes sont : *AES, RC4, DES, 3DES*

Scénario

- 
- Alice génère une clé symétrique aléatoire et l'utilise pour crypter son message
 - Alice crypte sa clé symétrique avec la clé publique de Bob
 - Alice envoie la clé symétrique cryptée et le message cryptée
 - Bob utilise sa clé privé pour décrypter la clé symétrique d'Alice
 - Bob peut ensuite décrypter le message

Applications

- 
- Ces techniques sont appliquées dans de nombreux protocoles :
 - **SSH** : Secure Shell
 - **SASL** : Support for the Simple Authentication and Security Layer utilisé par LDAP et IMAP
 - **TLS/SSL** : Secure Socket Layer pour le protocole HTTPS



TLS/SSL

TLS (or SSL) permet 3 ou 4 sous-caractéristiques de la sécurité

- Authentification du serveur
- Confidentialité des données échangées
- Intégrité
- Optionnellement, l'authentification du client (rarement utilisé)

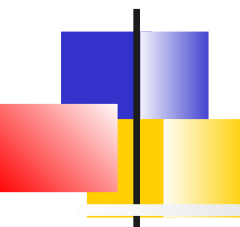


Attaques et OWASP

Open Web Application Security Project (OWASP) :
Communauté OpenSource centrée sur la sécurité du Web.

En particulier :

- Etablit le top 10 des vulnérabilité
- *WebGoat* : Application volontairement vulnérable pour l'éducation
- *WebScarab* : Outil de test permettant de visualiser et modifier les trames http/https
- *Dependency-Check* : S'assure que l'on utilise pas des composants avec des vulnérabilités connues
- *Zap* : Vulnérabilités dans les applications Web
- OWASP testing et review guides : Document d'aide au test de sécurité et à la revue de code



Top 10

Injection : (SQL, NoSQL, OS, et LDAP) données malicieuses envoyées à un interpréteur comme une partie d'une commande ou requête

Broken Authentication and Session Management : Comprend notamment le vol de session ou la récupération de mots de passe.

Sensitive Data Exposure : Exposition des données sensibles (mots de passe, CB,...) => nécessité de chiffrement.

XML External Entities (XXE) : Traitement par un parseur XML d'un contenu ayant une référence à une entité externe

Broken Access Control : Accès ouvert à une console d'administration par exemple

Security Misconfiguration : Configuration par défaut par exemple

Cross-Site Scripting : XSS : injection de contenu dans une page web provoquant des actions non désirées.

Insecure Deserialization : L'attaquant arrive à désérialiser/sérialiser des données échangées entre un client et un backend.

Librairies tierces : Utiliser des composants avec des vulnérabilités connus

Insufficient Logging&Monitoring : Journalisation et surveillance insuffisante rendant difficile la détection de compromission



Entêtes Http

Les en-têtes HTTP de sécurité fournissent une couche de protection supplémentaire qui permet de limiter les vulnérabilités et les attaques.

Content-Security-Policy, X-XSS-Protection, X-Frame-Options, X-Content-Type-Options, Access-Control-Allow-Origin, Strict-Transport-Security, public-key-pins



Big Data



Définition BigData

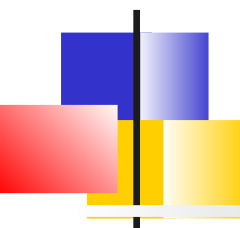
Ensembles de données devenus si volumineux qu'ils dépassent l'intuition et les capacités humaines d'analyse ainsi que celles des outils informatiques classiques

Ce changement d'échelle en volume a des impacts techniques sur toutes les étapes du traitement de données :

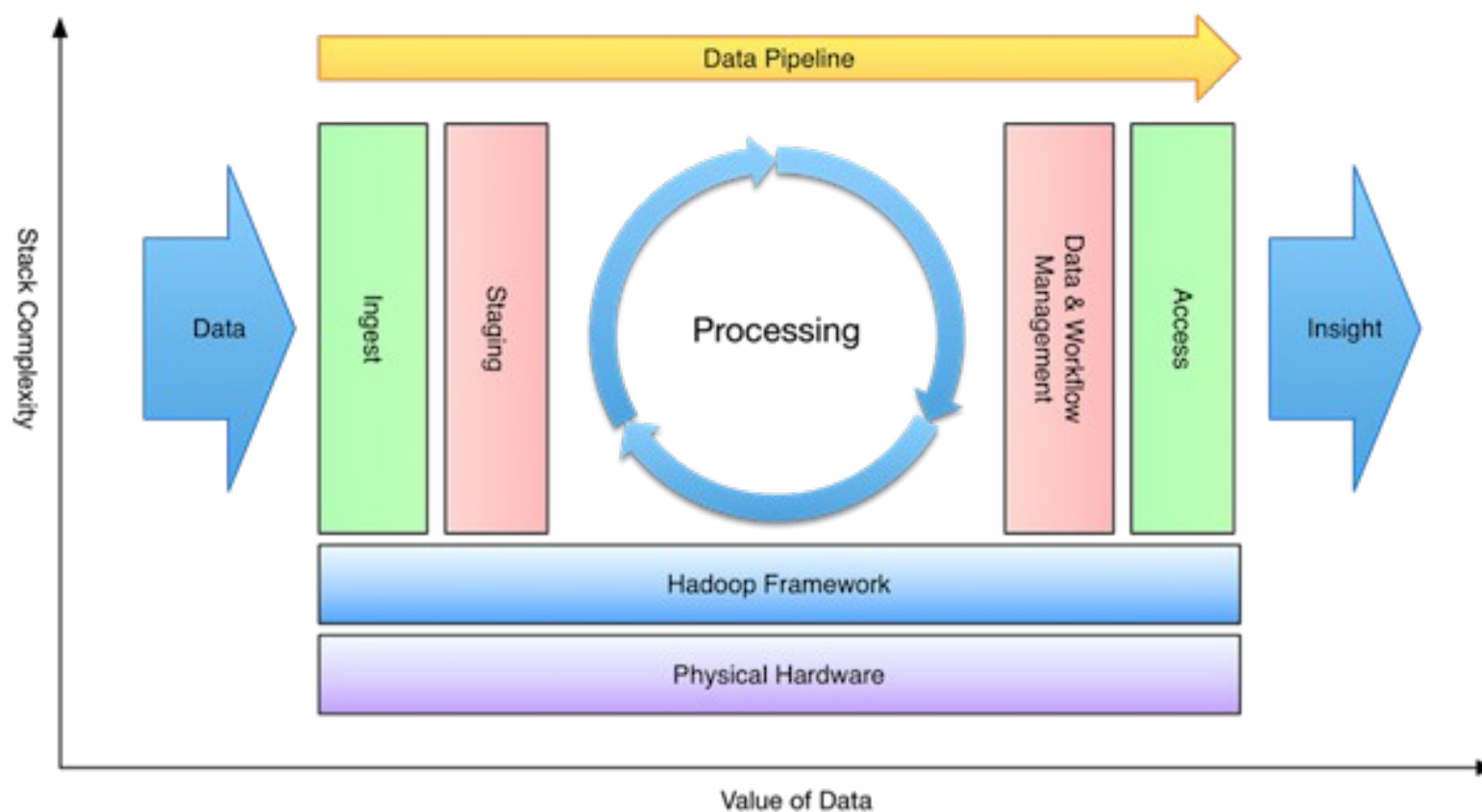
- Alimentation ou Ingestion
- Traitement
- Stockage
- Exploitation et Analyse

Objectif :

Prendre des données brutes et les convertir en valeur.



Architecture : Data Pipeline





Data Ingestion : Problématiques

Techniques d'alimentation :

- Chargement par lots : Périodiquement, des lots de données sont ingérés
- Flux de données : Les données arrivent en temps réel
- Détection de changement dans les sources de données



Stockage Problématiques

Format :

- Fichier - HDFS
- Structuré - NoSQL

Volume :

- Distribution
- Scalabilité
=> Clustering

Archivage :

- Archiver des données qui ne sont pas utilisées fréquemment, tout en assurant un accès occasionnel facile?



Traitement : parallélisation

Exemple : *MapReduce*

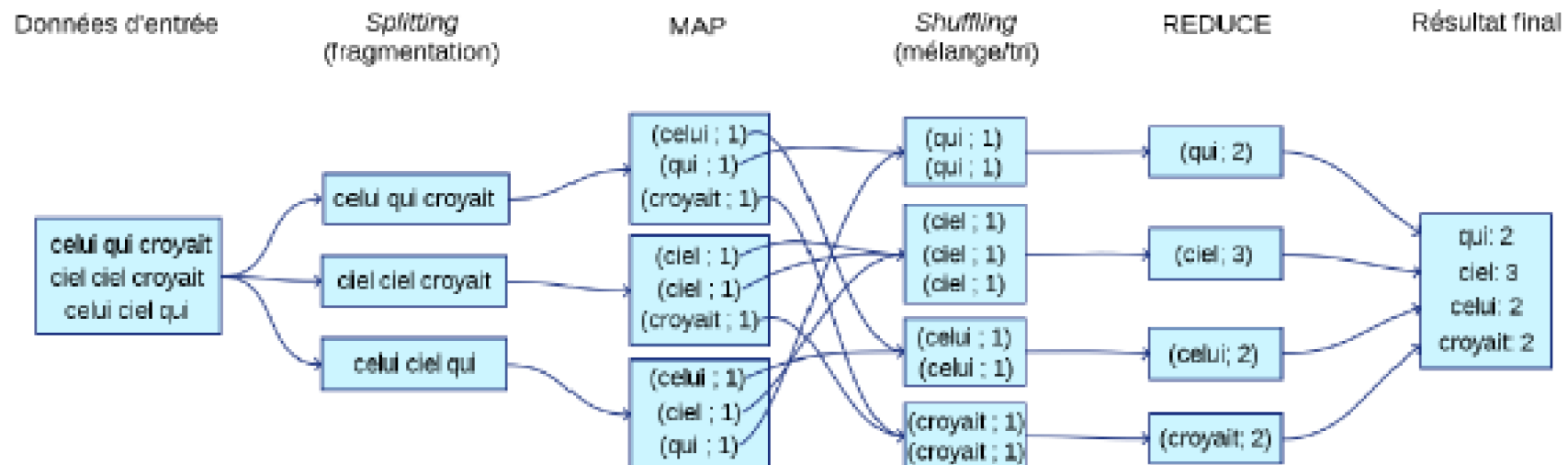
Objectifs :

- Traiter de gros volumes en //
- Équilibrer la charge du réseau
- Tolérance aux pannes
- Fonctionner sur des milliers de serveurs

Rôles :

- **Map** : Extraire des informations sous-forme clé-valeur
- **Reduce** : Agréger des informations ayant la même clé

Exemple : « Compter des mots »





Analyse de données

Dans les outils qui transforment les données brutes en *insights*, patterns, prédictions et données calculées, on peut distinguer

- ***Data Visualisation (DataViz)*** : Outil visuel d'exploration de données
- ***Advanced Analytics***: Agrégations et algorithmes analytiques pour effectuer des calculs complexes
- ***Machine Learning***: Pour identifier des patterns et faire des prédictions ?



Apache Hadoop

Cette solution offre un espace de stockage massif pour tous les types de données, une immense puissance de traitement et la possibilité de prendre en charge une quantité de tâches virtuellement illimitée. Fail-over

Basé sur Java

Principe :

- Diviser les données
- Les sauvegarder sur une collection de machines
- Traiter les données directement là où elles sont stockées plutôt que les copier à partir d'un serveur distribué

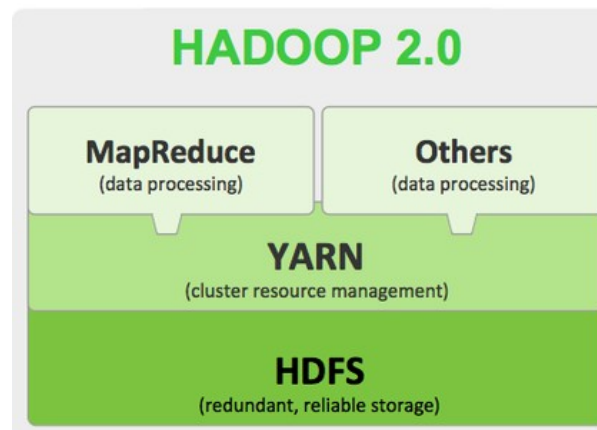
=> Il est possible d'ajouter des nœuds au fur et à mesure que les données augmentent



Modules

Le projet est décomposé en modules :

- **Hadoop Common**: Utilitaires communs.
- **Hadoop Distributed File System (HDFS™)**: Système de fichier distribué avec accès haut débit
- **Hadoop YARN**: Framework pour planification de jobs et gestion des ressources cluster.
- **Hadoop MapReduce**: Basé sur YARN. Traitement parallèle d'énormes jeux de données
- **Hadoop Ozone**: Stockage objet.





Outils Connexes

Apache Sqoop : Transférer des données en base vers Hadoop

Apache Flume : Permet de définir un réseau d'agents redondants qui transportent les données vers HDFS

Avro[™] : Sérialisation de données.

Cassandra[™] : Base de données NoSQL scalable sans SPOF.

HBase[™] : Base de données distribuées, scalable supportant de gros volumes de données structurées .

Hive[™] : Entrepôt de donnée supportant SQL

ZooKeeper[™] :



Outils Connexes (2)

Pig™ : Langage de haut-niveau permettant des traitements // sans *MapReduce*.

Tez™ : Framework permettant de combiner plusieurs jobs MapReduce. Basé sur YARN et intégré dans Hive et Pig

Ambari™ : Interface web pour provisionner, gérer et surveiller le cluster.

Spark™ : Solution plus large proposant un modèle de programmation simple permettant l'ETL, le machine learning, le traitement de flux et le calcul de graphe.

Distributions commerciales : **Cloudera**, **Hortonwork**



Elastic Stack

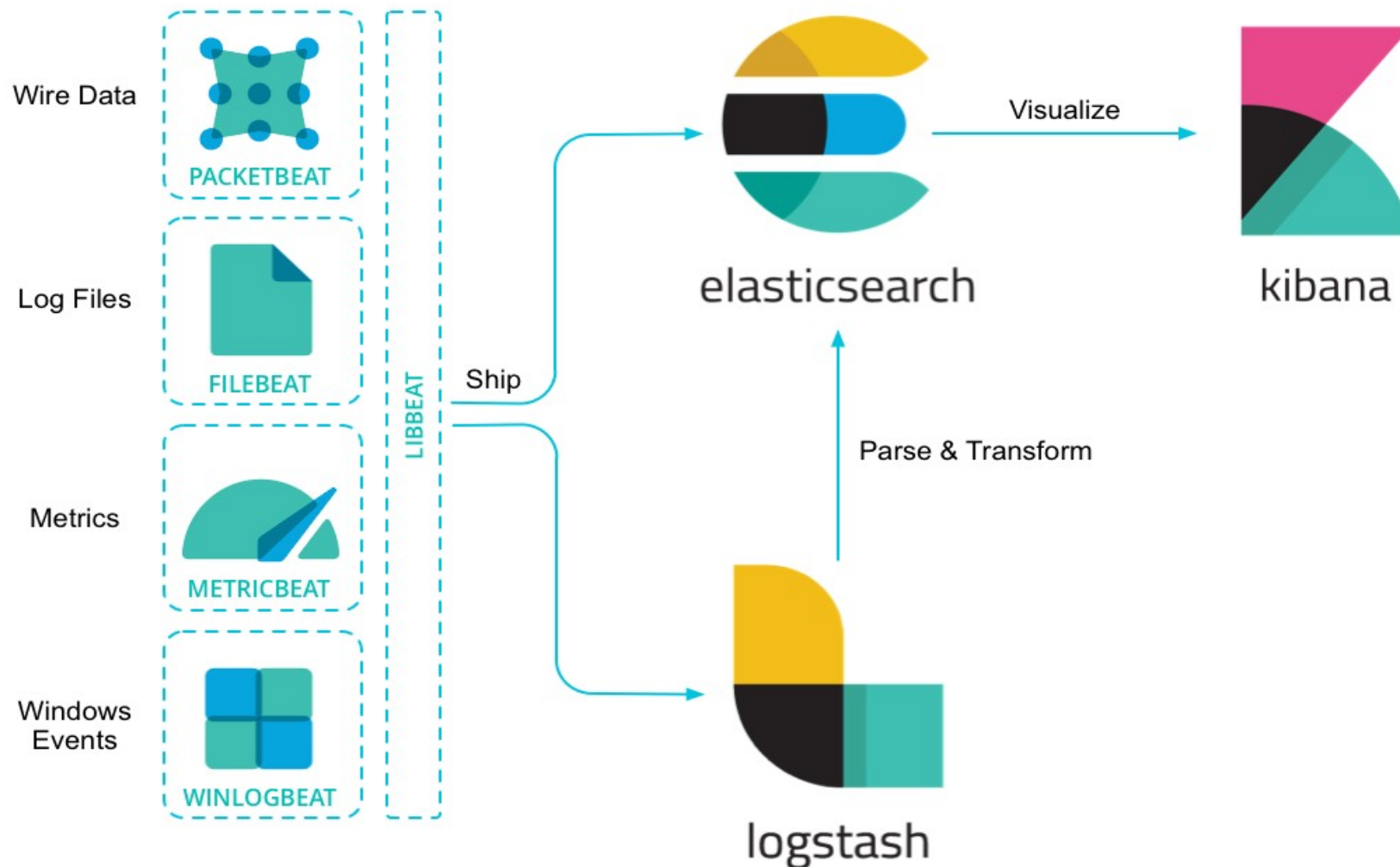
La solution **ElasticStack** propose une suite permettant d'indexer d'énormes volumes de données et d'afficher des tableaux de bord en temps réel (Near-Real-Time)

Principalement, utilisé comme outil de surveillance des SI (Métriques machines, Paquets réseau, Fichiers journaux), il peut indexer d'autres types d'événements

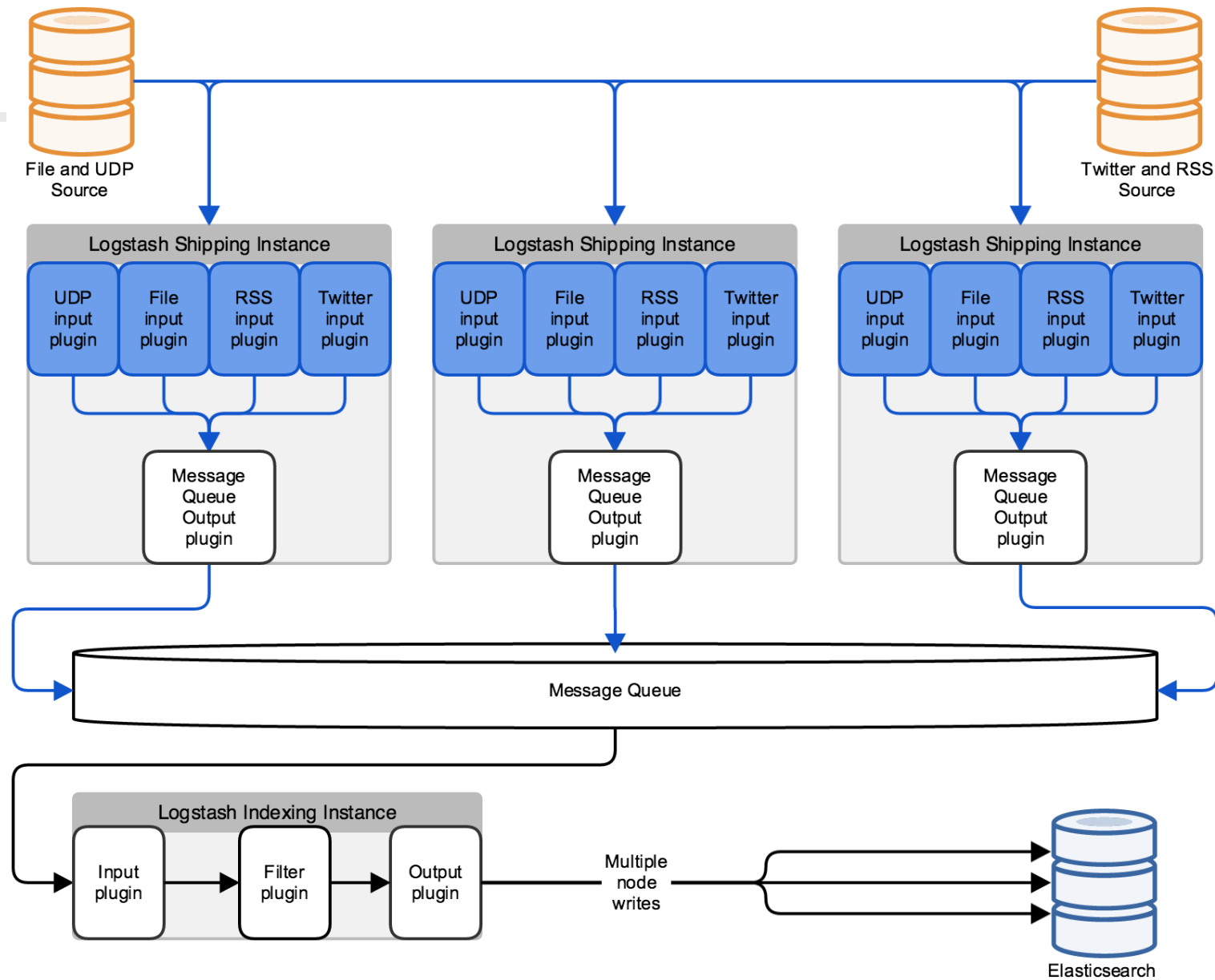
Son succès tient surtout :

- A sa simplicité de mise en place
- A une interface REST très riche
- A son outil de visualisation Kibana

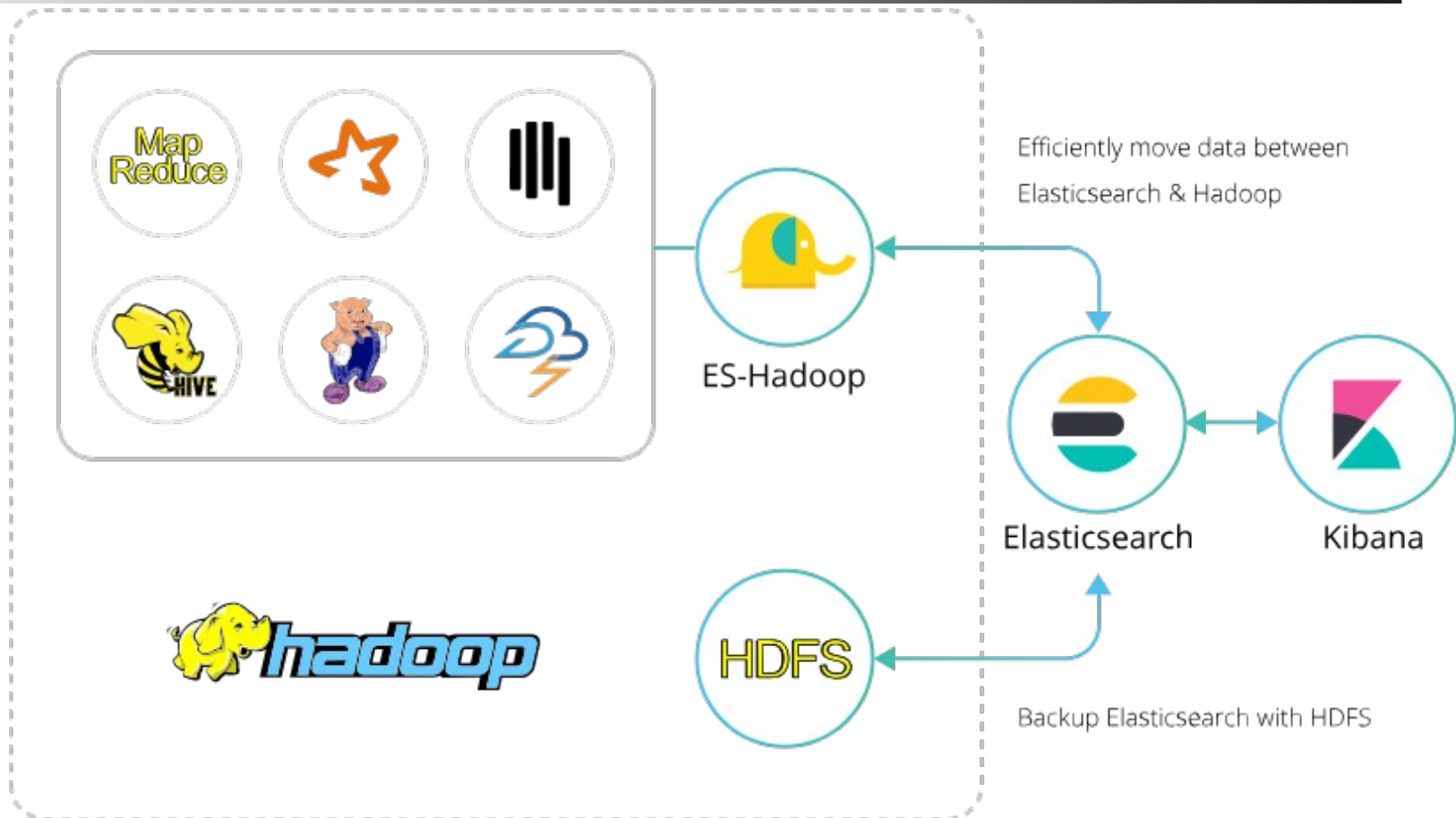
Architecture ELK



Architecture d'ingestion clusterisée



Elastic-Hadoop





Big Data et Machine Learning

Le **Machine Learning** est la technologie qui permet d'exploiter pleinement le potentiel du Big Data.

Oui dit autrement : Le Big Data est l'essence du ML

C'est une sous-branche de l'IA permettant aux ordinateurs d'apprendre à partir des données.

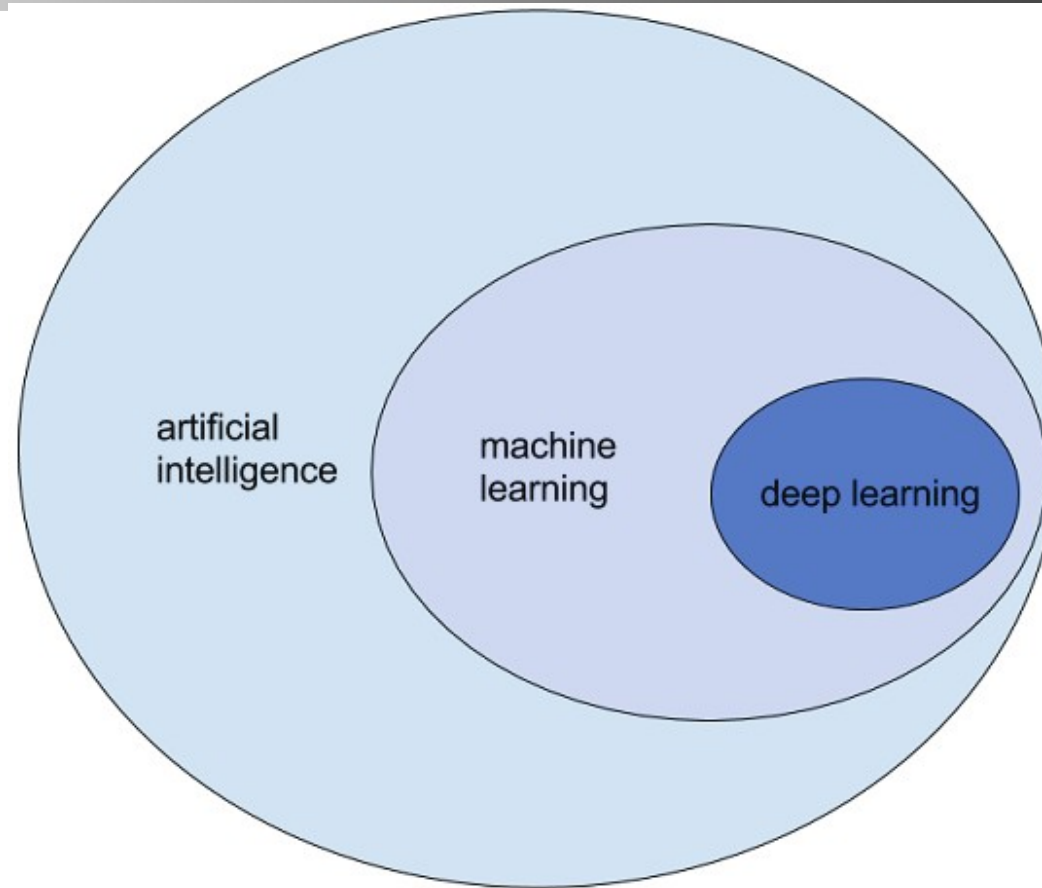
Elle est utilisée pour pour des applications génériques :

- Le traitement du langage naturel (Traduction, conversation)
- Reconnaissance d'image ou video (Classification, Voiture autonome, Robotique)

Mais également pour des applications métier :

- Management : aide à la décision, détection de KPI
- Marketing et commercial : segmentation client, détection de churn
- Services techniques : maintenance prédictive, surveillance d'infrastructure, optimisation chaîne de fabrication, de la consommation

IA et Machine Learning





Algorithmes d'apprentissage

Différents types d'algorithmes d'apprentissage sont disponibles :

- **Supervisé** : Un expert classe au préalable des échantillons de données
- **Non supervisé** : Pas d'expert. Le système identifie des classes d'échantillons similaires
- **Par renforcement** : L'action du système produit une valeur de retour qui guide l'apprentissage



Terminologie du ML

Définition : Les systèmes de ML apprennent comment combiner des entrées pour formuler des prédictions efficaces sur des données qui n'ont encore jamais été observées.

- Une **étiquette** est le résultat de la prédiction
- Une **caractéristique** est une variable d'entrée
- Un **exemple** est un ensemble complet de caractéristiques. Il est exprimé sous formes de vecteur
- Pour effectuer ces prédictions, le système s'appuie un **modèle** (~ fonction des caractéristiques)
- L'élaboration du modèle s'appelle **l'apprentissage**.



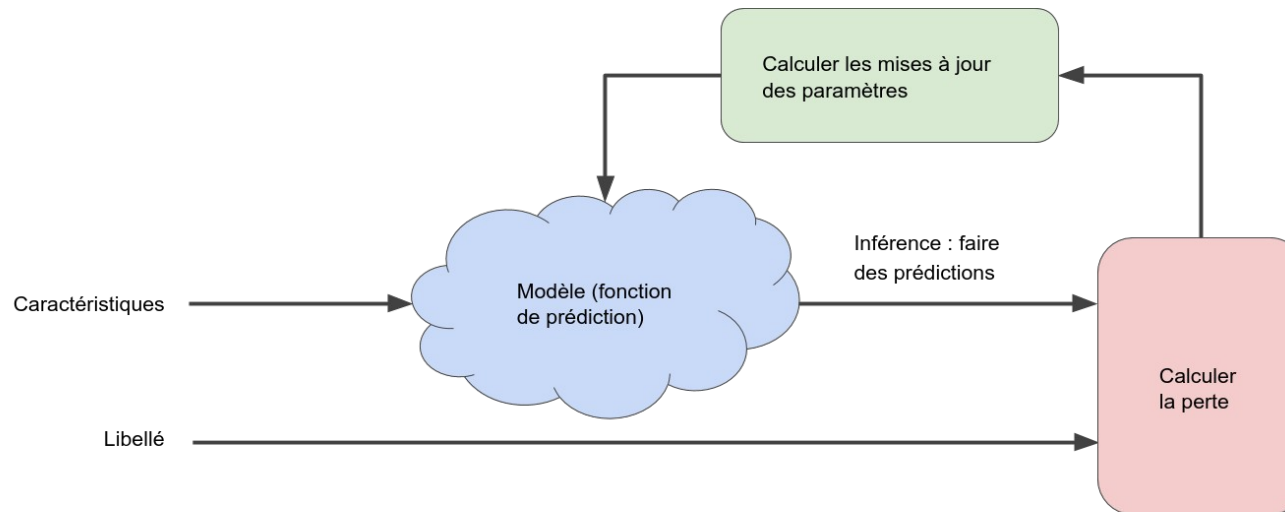
Apprentissage supervisé

L'apprentissage supervisé nécessite de disposer d'un ensemble d'exemples étiquetés.

L'ensemble est généralement partitionné en 2 sous-ensembles :

- Un **ensemble d'apprentissage** permettant de déduire les paramètres du modèle
- Un **ensemble de validation** permettant de valider le modèle trouvé

Apprentissage itératif



L'apprentissage consiste à minimiser la perte (différence entre valeurs réelles et valeurs produites). C'est un processus itératif qui utilise des algorithmes d'optimisation (comme la descente de gradient par exemple).

Réseaux de neurones et Deep Learning

Le **deep learning** est une méthode dont le modèle est constitué d'un réseau de neurones

- Le réseau de neurone apporte de la non-linéarité dans le modèle.
- Les données traversent plusieurs couches qui effectuent des transformations non linéaires sur les données.
- Les couches les plus hautes finissent par proposer un niveau d'abstraction proche de l'humain.

