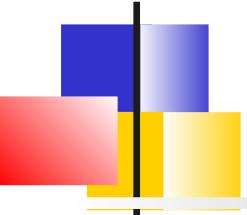


Principes de conception

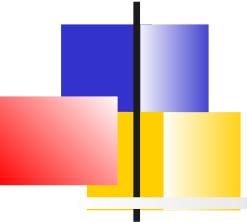
David THIBAU – 2018

david.thibau@gmail.com



Agenda

- Introduction
 - Activités de développement et méthodologies
 - Organisation des équipes et approche DevOps
 - Métriques logiciel
 - Langages, librairies, framework, middleware
- Design patterns
 - Introduction
 - Patterns de construction
 - Patterns de structuration
 - Patterns comportementaux
- Architectures distribuées
 - Problématiques de la distribution
 - Architectures n-tiers
 - Services Web / Services REST
 - Architectures SOA
 - Architectures micro-service
 - Frameworks clients
- Usine à logicielle, CI et DevOps
 - L'intégration continue
 - Provisionning : PaaS, virtualisation/containerisation
 - Workflows de collaboration et SCMs
 - Les outils de build
 - Serveurs d'intégration continue et pipelines



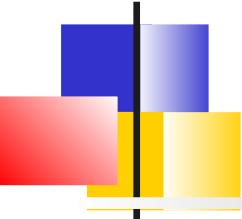
Introduction

Méthodologies de développement

Organisation des équipes et
l'approche DevOps

Métriques logiciel

Langages, librairies, frameworks et
middleware



L'activité de développement

Quelque soit la méthodologie, le développement d'un logiciel nécessite différentes activités :

Tout commence par une **expression de besoin métier** dérivée en

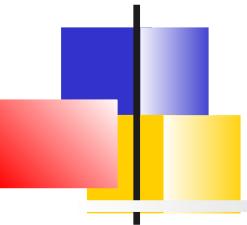
- Du fonctionnel : cahier des charges, user story, use case UML, ...
- Des contraintes techniques : plate-forme cibles, performance, évolutivité, maintenabilité

Des activités d'**analyse** débouchant sur des choix : dimensionnement des équipes, langages, architectures, framework, patterns, ergonomie

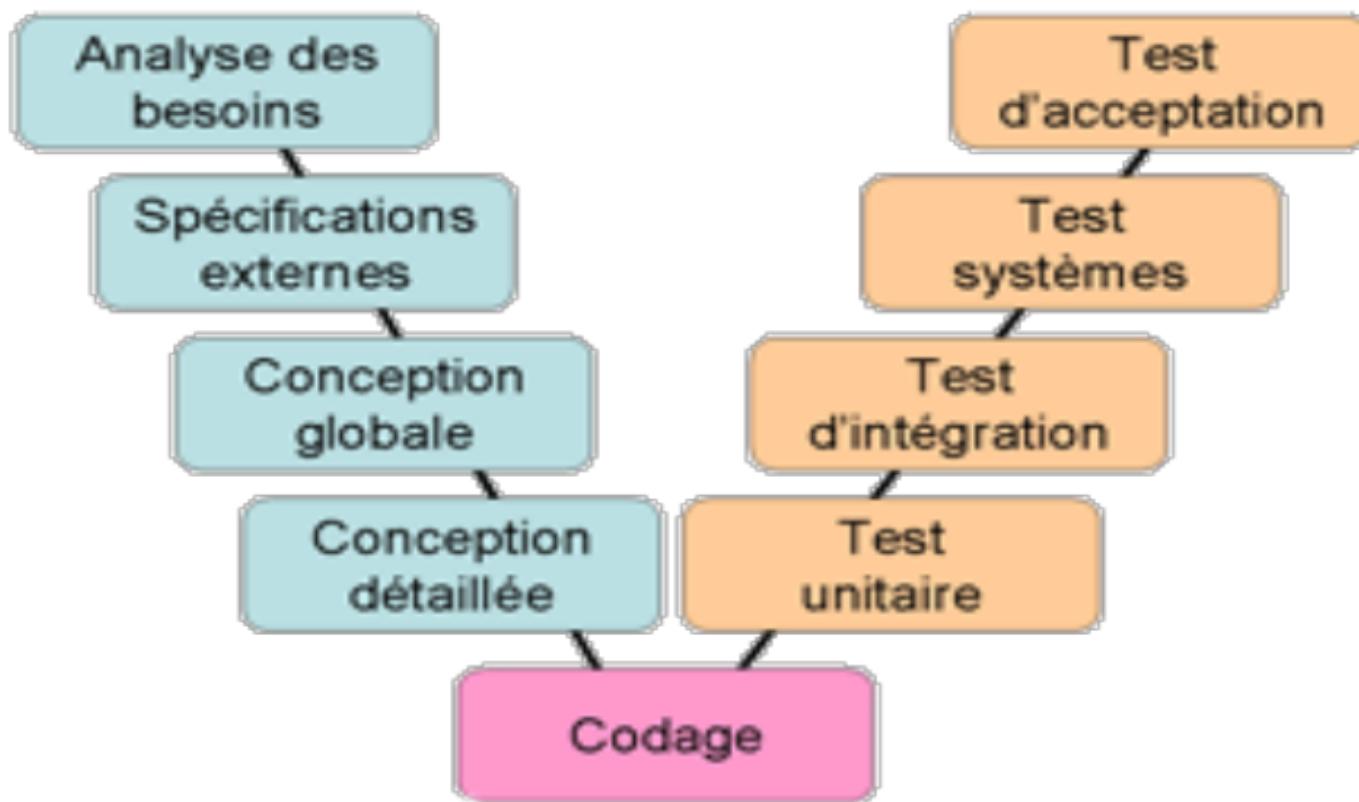
Puis l'implémentation, le **codage** et la vérification de l'implémentation, le **test**

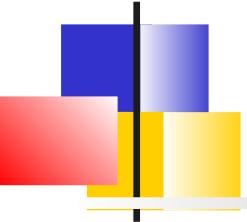
Enfin, la mise à disposition : **déploiement**

La collaboration inter-équipe, la documentation, la passation



Méthode Waterfall (Merise, SADT)





Inconvénients de l'approche

Absence de variables d'**ajustements** (prix, délai, fonctionnel)

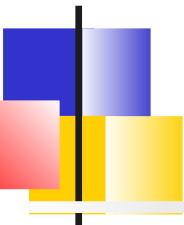
Effet tunnel : La maîtrise d'ouvrage ne voit le résultat qu'après un temps non négligeable

=> Pour réussir un projet :

- Nécessite beaucoup de savoir-faire
- Une vision claire, pertinente et bien exprimée du besoin métier

Approche peu adéquate pour les projets en constante évolution

Méthode d'avant les POO



Itération et dimensionnement

2 facteurs permettent d'améliorer les résultats:

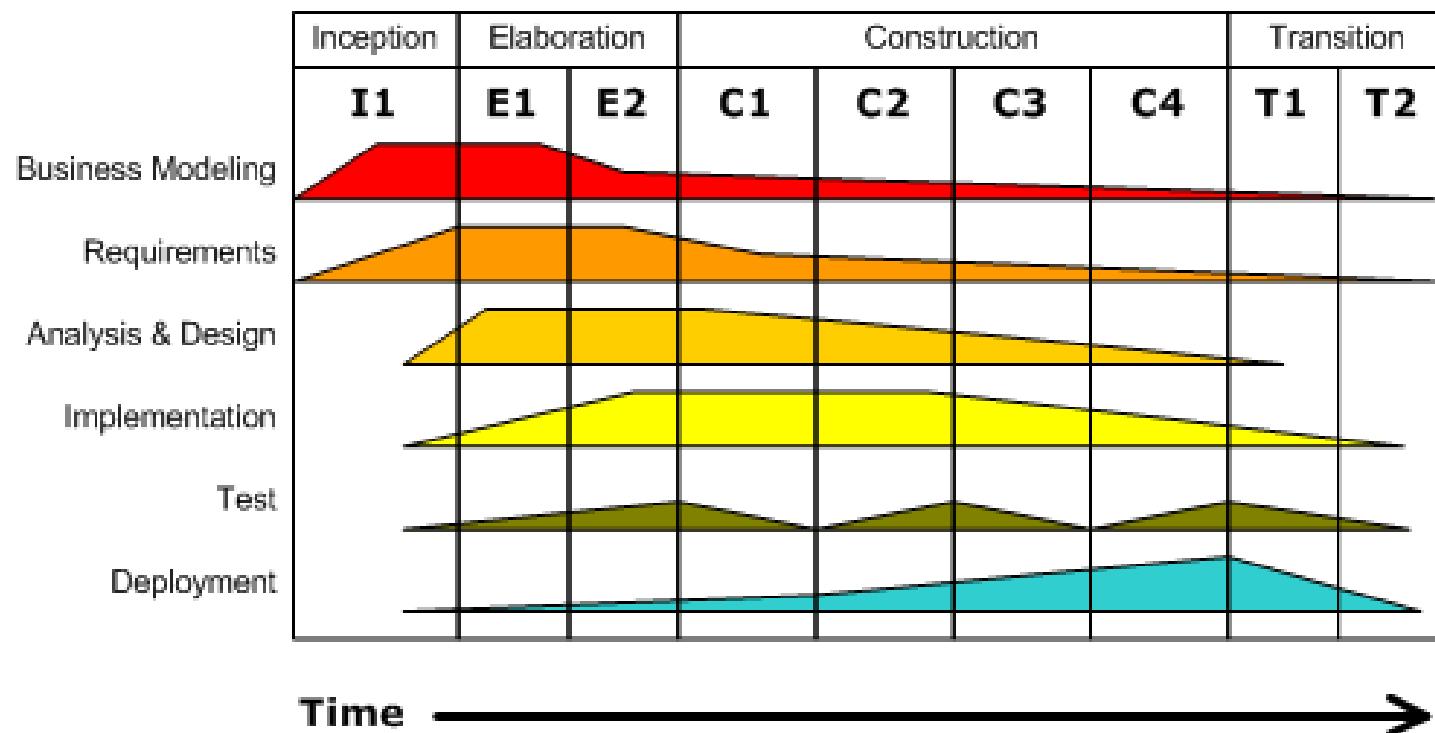
- la mise en place de pratiques **itératives**,
- des projets plus **petits**
=> des équipes de dév. de plus en plus petites

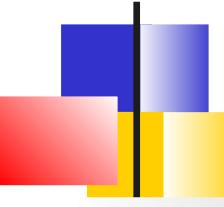
→ Ce sont les piliers de l'agilité que l'on retrouve dans *RUP*, *XP Programming*, *Scrum*

Unified Process

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.

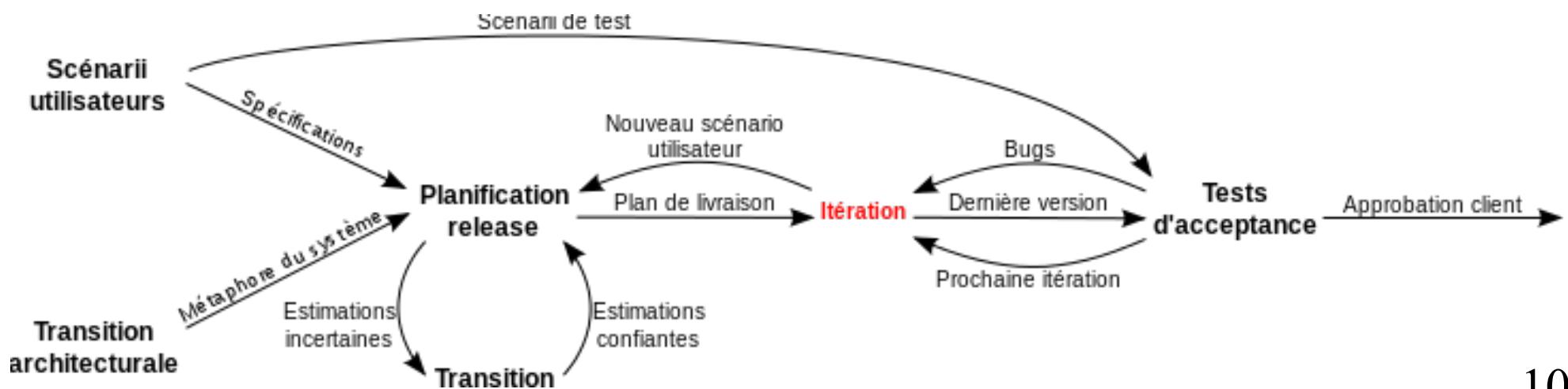


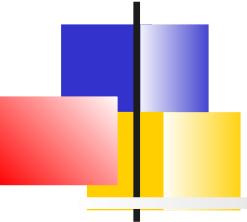


XP: eXtreme Programming

Agilité, Petites équipes, Tout à l'extrême:

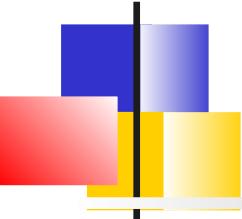
- Revue de code en permanence (par un binôme)
- Tests systématiques
- Refactoring en continu
- Toujours la solution la plus simple ;
- Evolution des métaphores ;
- Intégration continue ;
- Cycles de développement très rapides pour s'adapter au changement.





L'agilité

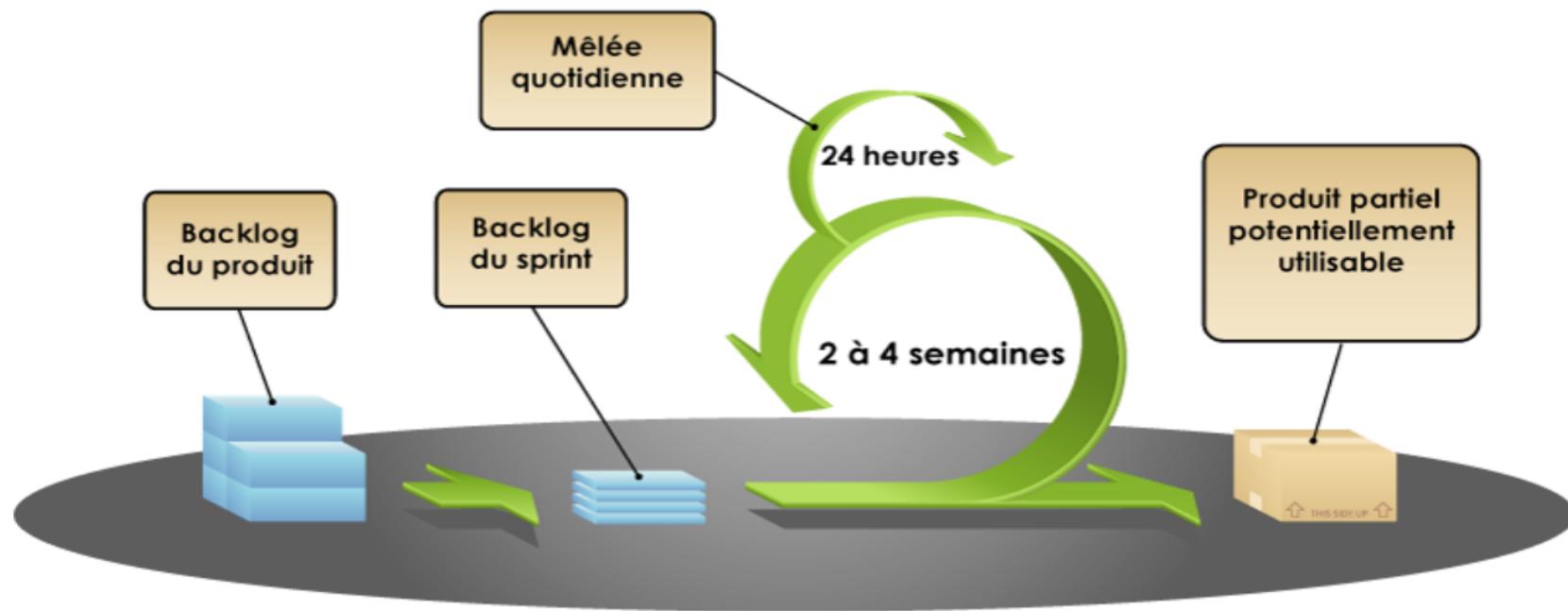
- Le terme agile (regroupant de nombreuses méthodes) est consacré par le manifeste Agile : <http://agilemanifesto.org/> 2001
 - Personnes et interactions plutôt que processus et outils
 - Logiciel fonctionnel plutôt que documentation complète
 - Collaboration avec le client plutôt que négociation de contrat
 - Réagir au changement plutôt que suivre un plan

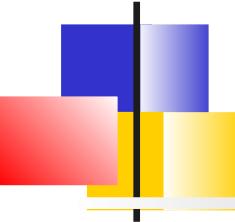


Les 12 principes

1. Satisfaction du client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.
2. Accepter les changements de besoins, même tard dans le projet.
3. Livrez fréquemment un logiciel opérationnel.
4. Utilisateurs et développeurs travaillent ensemble quotidiennement.
5. Les personnes sont motivées et bénéficient d'un environnement et d'un soutien à la hauteur
6. Transmission de l'information via le dialogue en face à face.
7. L'avancement est mesuré via le logiciel opérationnel
8. Rythme de développement soutenable.
9. Excellence technique et bonne conception renforce l'agilité.
10. La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle.
11. Les équipes sont auto-organisées.
12. À intervalles réguliers, l'équipe cherche à s'améliorer.

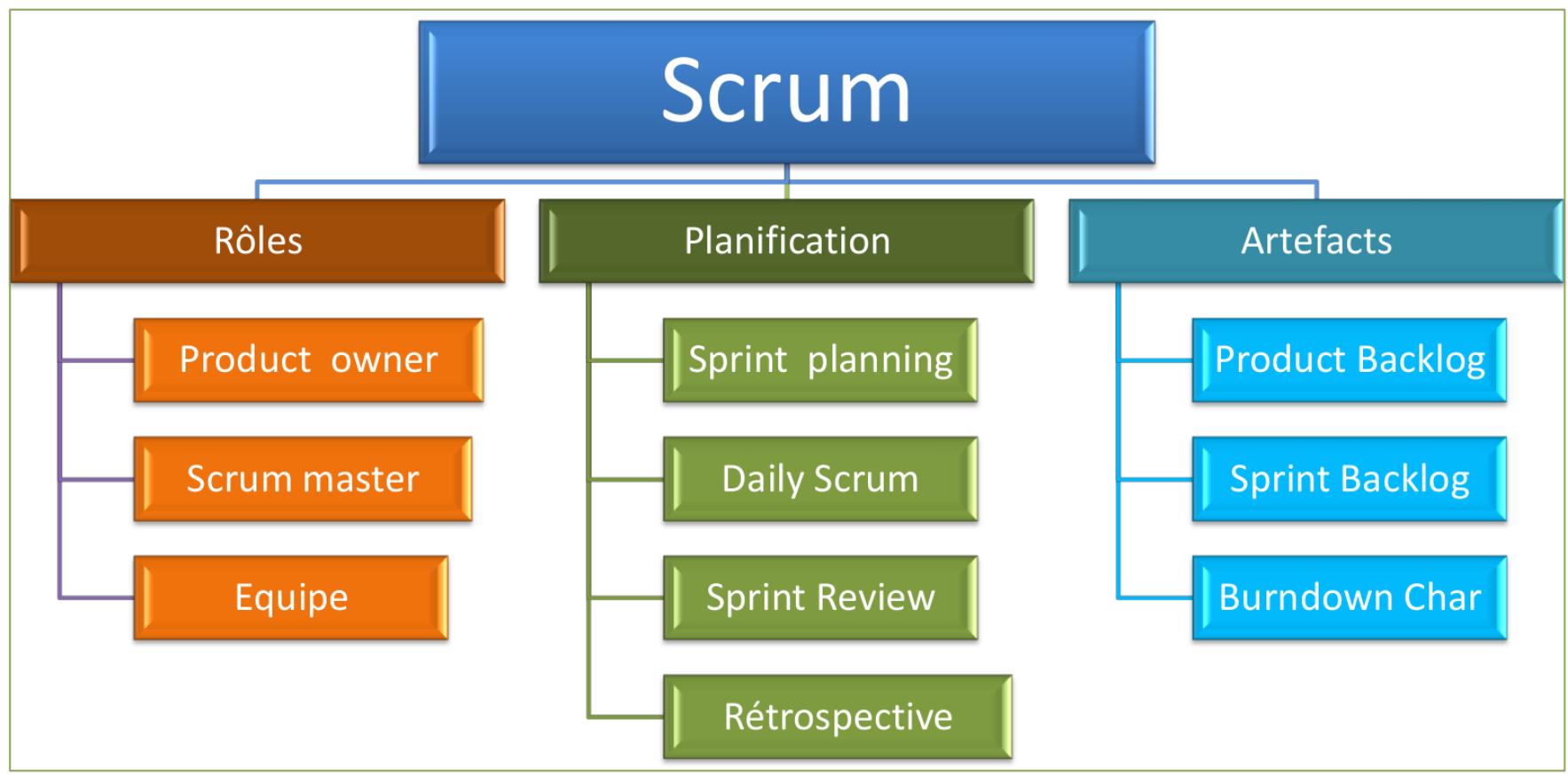
Scrum : rythme

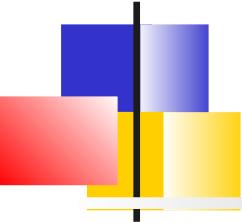




SCRUM

Rôles/Planification/Artefacts





Responsabilité du développeur agile

L'engagement : S'engager à atteindre un but

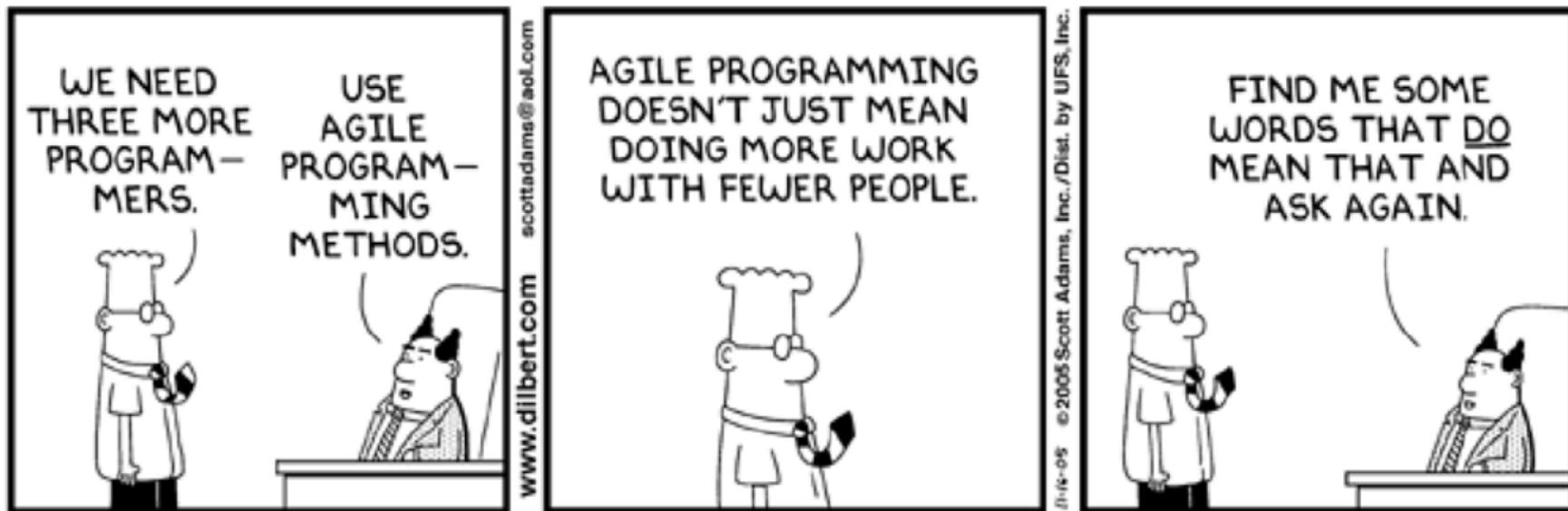
Le focus : Rester concentré sur ses engagements ; il ne faut pas s'inquiéter du reste

L'ouverture et la communication : Rendre disponible et visible toute l'information du projet ; être honnête et communiquer beaucoup

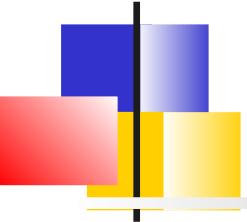
Le respect : Les individus sont moulés par leur expérience. Il est important de respecter les différentes personnes de l'équipe

Le courage : Avoir le courage de s'engager, d'être ouvert et d'exiger le respect

Vision « manager »



© Scott Adams, Inc./Dist. by UFS, Inc.

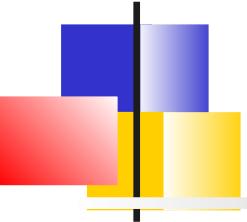


Limites de l'approche

Les développeurs paraissent interchangeables. Chaque jour, ils choisissent une user-story à implémenter et ont des implantations indépendantes à réaliser.

L'absence de vision à long terme, la cadence des sprints peuvent rendre moins gratifiant le travail des développeurs, faire baisser leur motivation et la qualité du code produit

L'approche métier est sacralisée au détriment de l'ingénierie logicielle, les tâches à faire doivent avoir la plus grande « business value », les autres plus techniques sont invisibles



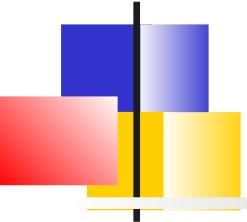
Conclusion sur l'agilité

Adapté :

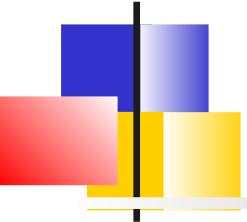
- à de petits projets.
- à des projets innovant : la spécification est affinée par les itérations

Facteur de réussite nécessaires :

- Les experts métiers (product owner) doivent être disponibles à 100 %, ils doivent également être sensible aux problèmes techniques, profil rare en réalité
- L'équipe peut facilement communiquer
- L'équipe bénéficie de développeurs expérimentés capable d'anticiper les évolutions, d'estimer les user story en y incluant les tâches techniques nécessaires au projet, d'argumenter les compromis « business value/ ingénierie logicielle »



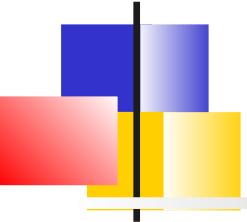
Organisation des équipes, l'approche DevOps



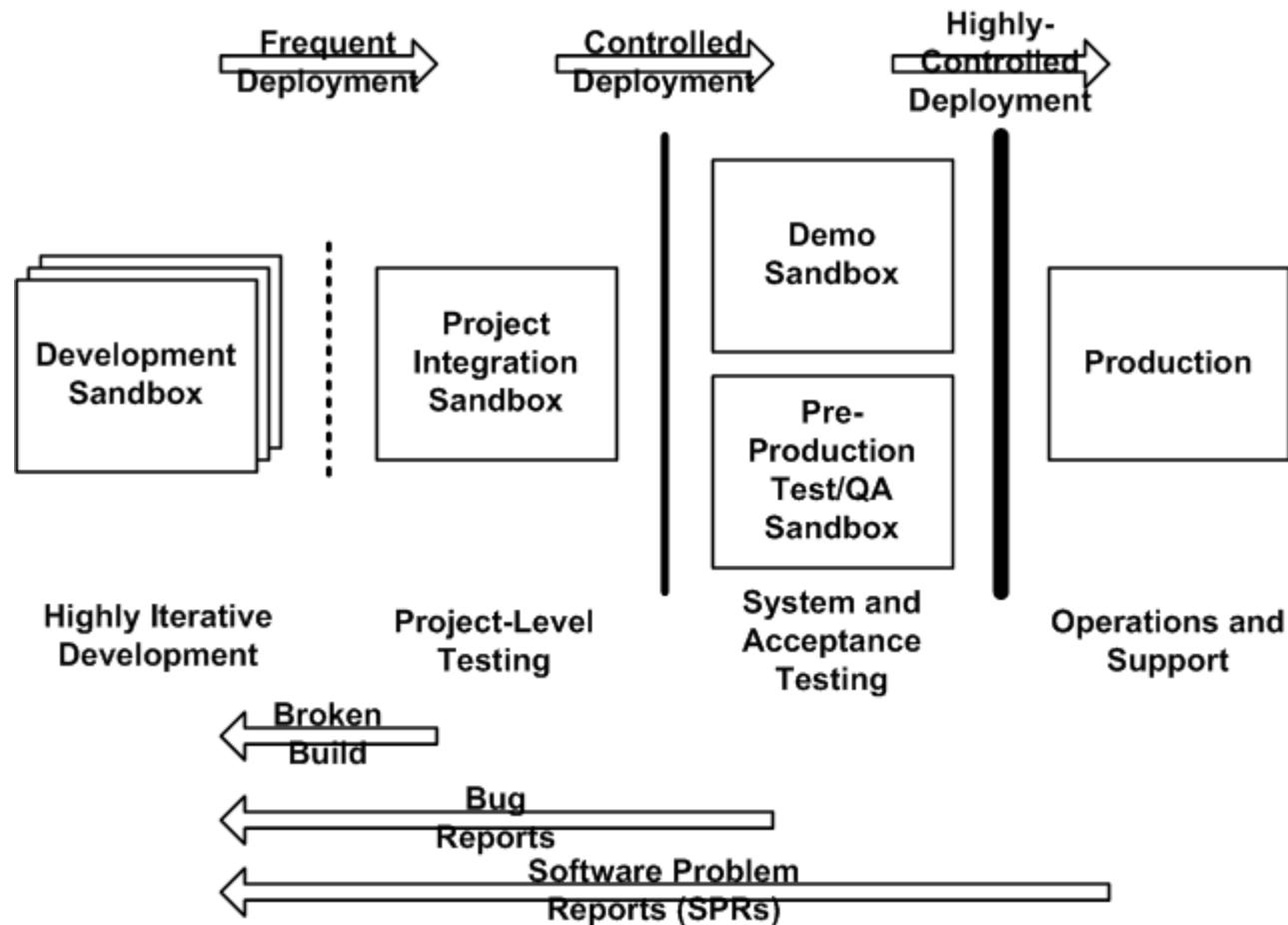
Cycle de vie d'un logiciel

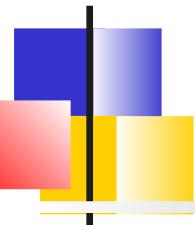
Le cycle de production d'un logiciel passe par plusieurs étapes correspondant à plusieurs environnement :

- **Développement** : Poste du développeur
- **Intégration** : Intégration des modifications de toute l'équipe
- **QA** : Environnement proche de la production permettant la qualification des releases
- **Production** : Exploitation, Support, Maintenance



Environnements



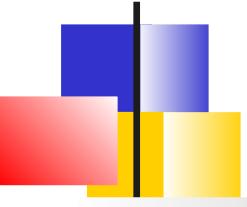


Disparité des environnements

Les environnements en général diffèrent :

- **Développement** : IDE, Code source lisible permettant le debug, configuration serveur pour des déploiements à chaud, base de données simplifiée, ...
- **Intégration** : Configuration pour les tests d'intégration. Sondes, Niveau de trace, Simulation de charge, de données
- **QA** : Configuration pour les tests QA. Presque la production mais pas complètement.
- **Production** : Qualité de service, charge réelle, données de production

Ces environnements sont traditionnellement gérés et utilisés par des équipes distinctes qui ... souvent communiquent peu
Les équipes ont de plus des objectifs différents

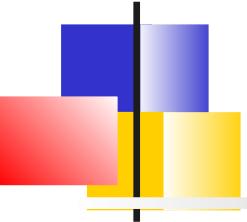


Objectifs des développeurs

L'objectif de cette équipe est d'implémenter les fonctionnalités requises dans le temps imparti.

Les facteurs de réussite sont :

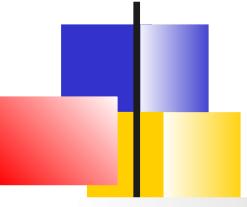
- L'expertise des développeurs
- Leur maîtrise des outils :
 - IDE Raccourci, auto-complétion refactoring,
 - Debug : Point d'arrêt inspection
 - Tracing : Traces configurables
- De la rapidité du cycle Code/Test :
 - Framework de test unitaire léger
 - Compilation requise ou pas
 - Déploiement à chaud, recharge automatique



Quelques profils développeur

Plusieurs classification des développeurs :

- **Développeur junior / senior** : Distinction sur l'expérience
- **Développeur front / back** : Distinction sur le domaine d'expertise
- **Développeur full-stack** : Connaissance de tous les domaines d'expertises requis par le projet
- **Lead developer** : Référent technique dans une équipe, transmets aux autres développeurs, son avis est respecté par les métiers
- **Architecte** : Choisit le cadre technique, les architectures de système



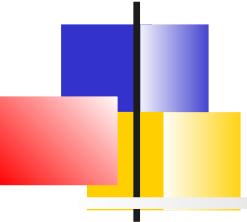
Objectifs Intégration/QA

Les développeurs ont validé leur implémentation dans leur environnement, il est temps de s'assurer que le logiciel fonctionne correctement dans des conditions les plus proches de la production.

L'équipe d'intégration est alors responsable de dimensionner l'architecture et l'infrastructure nécessaire à l'utilisation imaginée du logiciel.

D'autres types de tests sont effectués :

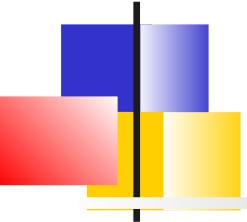
- Fonctionnel
- De conformité
- De charge
- D'acceptance
-



Objectifs des opérations

L'objectif principal d'une équipe opérationnelle est de garantir la stabilité du système et des infra-structures

=> l'équipe se focalise sur la contrainte qualité, au détriment du temps et du coût en contrôlant sévèrement la qualité des changements apportés au système qu'elle maintient.



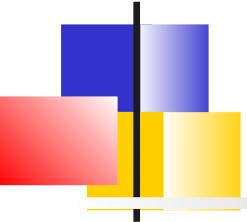
Le constat DevOps

Le constat DevOps :

Les différents objectifs donnés à des équipes qui se parlent peu créent des tensions et des dysfonctionnements dans le processus de mise en production d'un logiciel.

=> Pour l'équipe Ops, l'équipe de développement devient responsable des problèmes de qualité du code et des incidents survenus en production.

=> L'équipe Dev blâme son alter ego Ops pour sa lenteur, les retards et leur méconnaissance des livrables qu'elle manipule

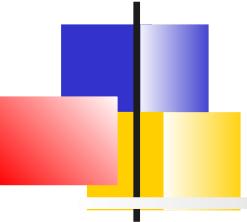


Approche *DevOps*

Devops vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers" chargés d'exploiter les applications existantes au sein d'une même équipe.

Cela impose :

- la réunion des équipes
- la montée en compétence des différents profils.

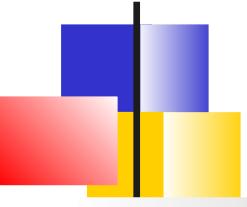


Recommandations *DevOps*

- Un déploiement régulier des applications, la seule répétition contribuant à fiabiliser le processus ;
- Un décalage des tests "vers la gauche", autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- Une intégration continue incluant des "tests continus" ;
- Une boucle d'amélioration courte (i.e. un feed-back rapide des utilisateurs) ;
- Une surveillance étroite de l'exploitation et de la qualité de production factualisée par des métriques et indicateurs "clé".



Métriques logiciel



Métriques

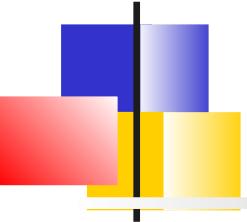
La mesure est fondamentale à toutes les disciplines d'ingénierie ; le software n'est pas une exception

Dans le cadre d'un logiciel, la production de métriques permet :

- Donner un chiffre à des facteurs qualitatifs
- De s'améliorer : améliorer le processus de développement, la qualité du produit
- D'affiner les estimations (prix, délais)

Des métriques sont de 2 types :

- Interne : analyse sur le code source ou compilé (qualité, couverture de test, ...)
- Externe : Sur le livrable (débit, temps de réponse, ...)



Norme ISO-9126

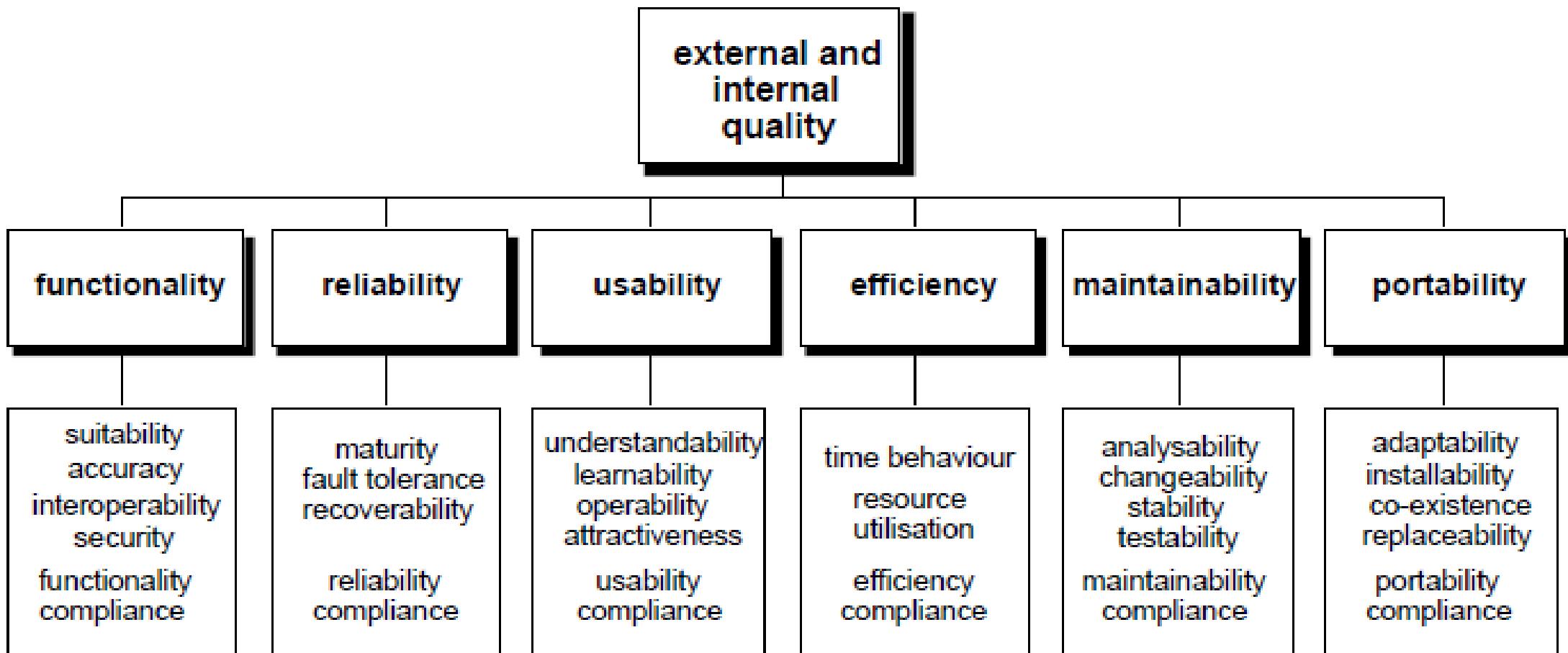
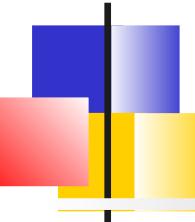
La norme ISO-9126 est la première norme établissant un modèle qualité de référence

- Elle normalise un nombre important de critères qualité
- Et surtout de métriques (façon de mesurer)

Quatre documents

- Part 1 : définit le modèle qualité (les caractéristiques)
- Part 2 : définit les métriques externes (mesurent le comportement du SI qui contient le logiciel)
- Part 3 : définit les métriques internes (mesurent le logiciel lui-même)
- Part 4 : définit les métriques de qualité d'usage (mesurent les effets de l'utilisation du logiciel dans un contexte d'utilisation donné)

Norme ISO-9126 : modèle qualité



Norme ISO-9126 : niveaux de métriques

3 niveaux de métriques :

D'usage (« quality in use »)

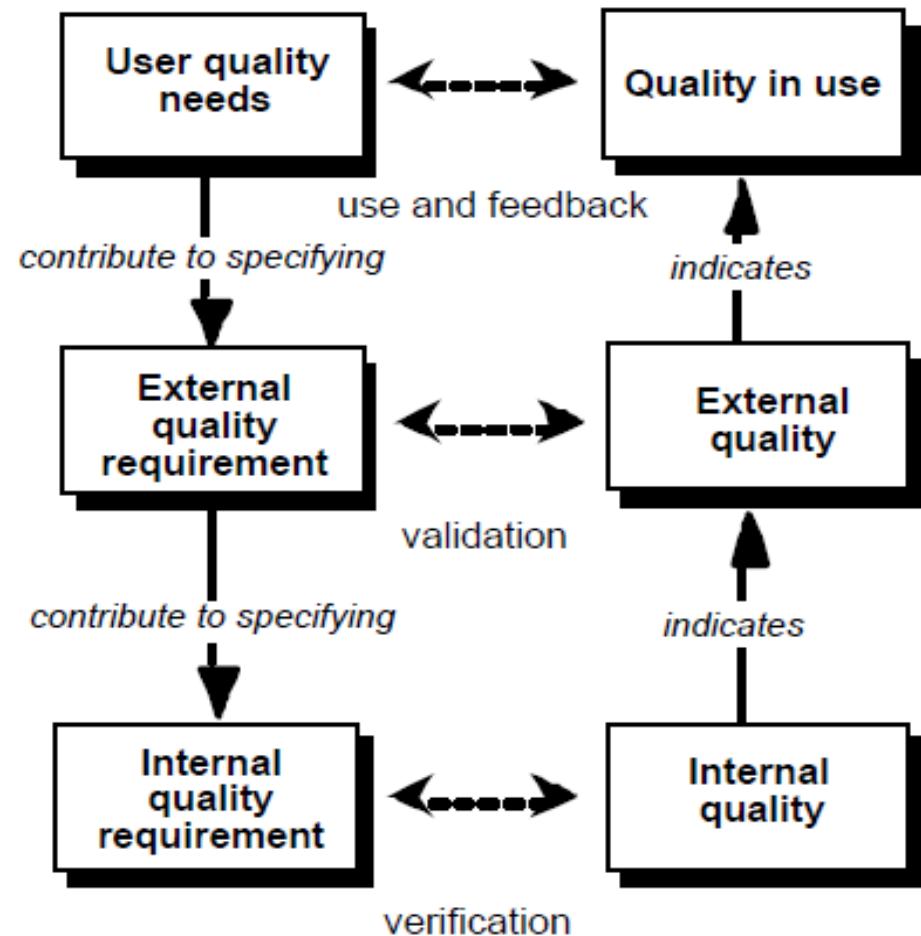
- Satisfaction des exigences du point de vue de l'utilisateur
- Mesure du résultat de l'utilisation du logiciel dans son environnement plutôt que des propriétés du logiciel lui-même

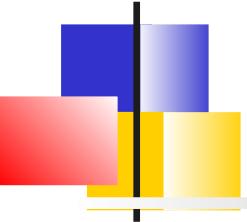
Externes

- Mesure du comportement à l'exécution

Internes

- Mesures statiques de composants





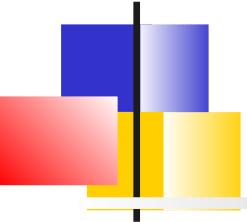
SQuaRE

(Software product Quality Requirements and Evaluation)

SQuaRE est une série de standard pour remplacer ISO/IEC 9126

Le principal apport de SQuaRE est la nouvelle caractéristique **Sécurité** avec les sous-caractéristiques suivantes :

- Confidentialité : Données accessibles par les seules personnes autorisées.
- Intégrité : Protection contre la modification de données
- Non-répudiation : Actions prouvées
- Traçabilité : Action pouvant être tracées
- Authenticité : Identité prouvée



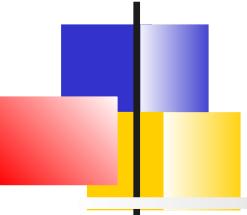
Outils Qualité

SonarQube est devenu l'outil standard de facto qui regroupe tous les outils de calcul de métrique interne d'un logiciel (toute technologie confondue)

Il intègre 2 aspects :

- Définitions des règles de codage du projet, détection des transgressions 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.



Analyse des règles

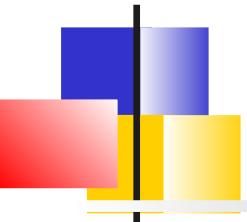
Base de règles

- Les règles sont fournies par SonarSource ou par des plugins
- On peut rajouter ses propres règles
- Elles peuvent être activées/désactivées dans des « profils qualité »

L'Analyse consiste à vérifier les règles du profil qualité du projet.

Lorsqu'une règle est transgressée, elle produit une Issue

- Bugs : Incidence sur la **fiabilité** du logiciel
- Code smells : Incidence sur la **maintenabilité**
- Vulnérabilité : Incidence sur la **sécurité**



Règle

"hashCode" and "toString" should not be called on array instances

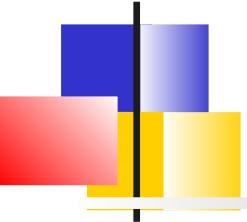
squid:S2116  

 Bug  Major  No tags Available Since December 8, 2017 SonarAnalyzer (Java) Constant/issue: 5min

While `hashCode` and `toString` are available on arrays, they are largely useless. `hashCode` returns the array's "identity hash code", and `toString` returns nearly the same value. Neither method's output actually reflects the array's contents. Instead, you should pass the array to the relevant static `Arrays` method.

Noncompliant Code Example

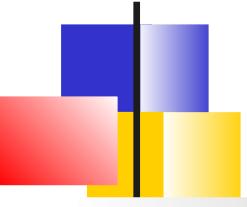
```
public static void main( String[] args )
{
    String argStr = args.toString(); // Noncompliant
    int argHash = args.hashCode(); // Noncompliant
```



Métriques

Sonar définit 9 axes pour les métriques qualité associé à un projet :

- Complexité
- Documentation
- Duplications
- Issues
- Maintenabilité
- Porte qualité
- Fiabilité
- Sécurité
- Tests



Exemple : Complexité

Complexité cyclomatique :

- Calculé à partir du nombre de chemins possibles dans le code.

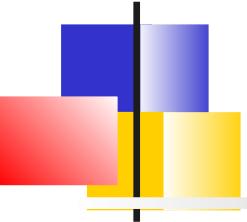
Complexité Cognitive :

- Difficulté à comprendre les chemins dans le code

Voir <https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html>

Mesures dérivées :

- Complexité moyenne par classe.
- Complexité moyenne par fichier
- Complexité moyenne par méthode.



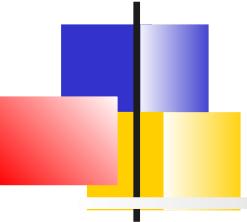
Exemple : Documentation

Commentaires :

- Nombre de lignes comportant une ligne de commentaire ou du code commenté.
- Nombre de lignes de code commentées
- Pourcentage de lignes de commentaires par rapport au total de lignes

Méthodes publiques et API

- Pourcentage de l'API publique documentée :
- Nombre de méthodes publiques nom commentées.



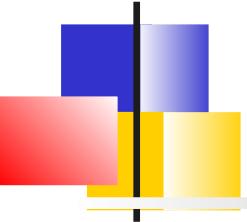
Couverture des tests

La couverture des tests concernent exclusivement les tests unitaires

Il s'agit de vérifier le code utile exécuté durant l'exécution des classes de tests. 2 axes de vérification :

- Les lignes
- Les chemins ou conditions

Sonar s'appuyait vers le projet OpenSource Cobertura; dans les dernières versions il supporte jacoco.



Métriques couverture des tests

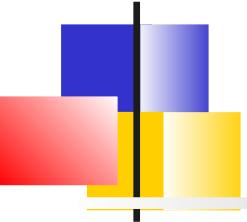
Couverture des conditions : $(CT + CF) / (2*B)$

- CT = Conditions ayant été évaluées à 'true' au moins 1 fois
- CF = Conditions ayant été évaluées à 'false' au moins 1 fois
- B = Nombre Total de conditions

Couverture des lignes : LC / EL

- LC = Lignes à couvrir moins les lignes non-couvertes
- EL = Lignes à couvrir

Couverture : $(CT + CF + LC)/(2*B + EL)$

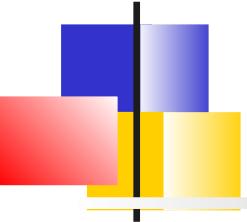


Porte qualité

Les portes qualité définissent des seuils pour un métrique

- Pour un avertissement
- Pour une erreur empêchant une release.

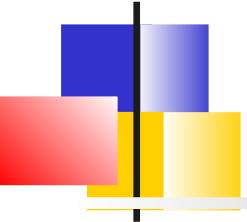
SonarQube fournit des portes par défaut qui sont adaptées en fonction du projet.



Autre aspects

De façons plus pragmatique, les applications webs et agiles ont adopté d'autre critères de qualité qui peuvent venir avec leur indicateur :

- Agilité
- Scalabilité
- Résilience
- Accessibilité
- Adoption/On-boarding, Taux de conversion

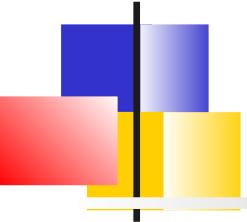


Performance

Concurrence/débit : Le plus grand nombre d'utilisateurs simultanés que le système doit supporter ou la cadence de transaction à atteindre

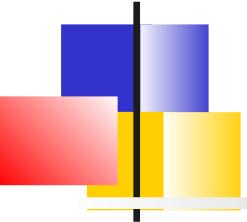
Temps de réponse observés par l'utilisateur final, décomposé en

- Temps accès service
- Temps de réponse serveur
- Temps de réponse du rendu



Langages de programmation

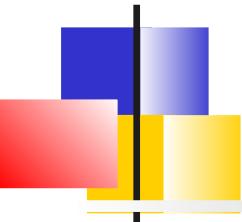
Typologie des langages
Librairies, Framework, Middleware
Langages back-end
Langages front-end
Mobile



Classification des langages

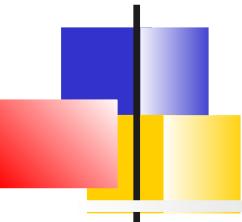
Les langages peuvent être classifiées selon de nombreux axes, retenons les suivants :

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



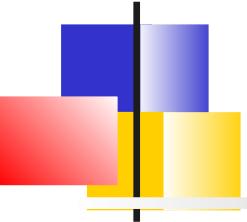
Classification (1)

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



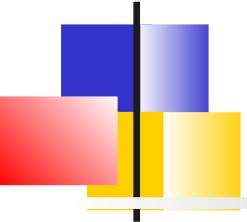
Classification (2)

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

- 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 attention au garbage collector !)
- 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.



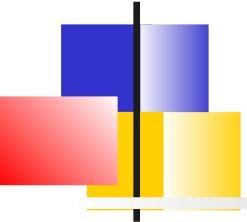
Classification des langages objets

- Héritage simple/multiple
- Surcharge des opérateurs
- Réflexion
- Génériques
- Lambda Expressions
- Collecte mémoire / ramasse miette
- Variables ou méthodes de classes (static)
- Visibilité des attributs ou méthodes
- Design By Contract (Conditions, invariants d'une méthode (Eiffel))
- Multithreading
- Expressions régulières
- Arithmétique de pointeurs
- Intégration avec d'autres langages (exemple JNI)
- Modèles de sécurité

Voir <http://www.jvoegele.com/software/langcomp.html>



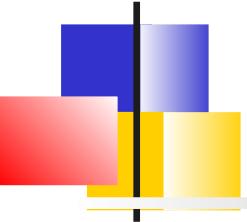
Librairies, framework, middleware



Librairies, framework, middleware

Plutôt que de réinventer la roue, nous utilisons des

- **Librairies** : Code utilitaire
- **Frameworks** : Offrent un modèle de programmation, des utilitaires et des services techniques
- **Middlewares** : Apportent des services techniques aux applications, en particulier permet la distribution



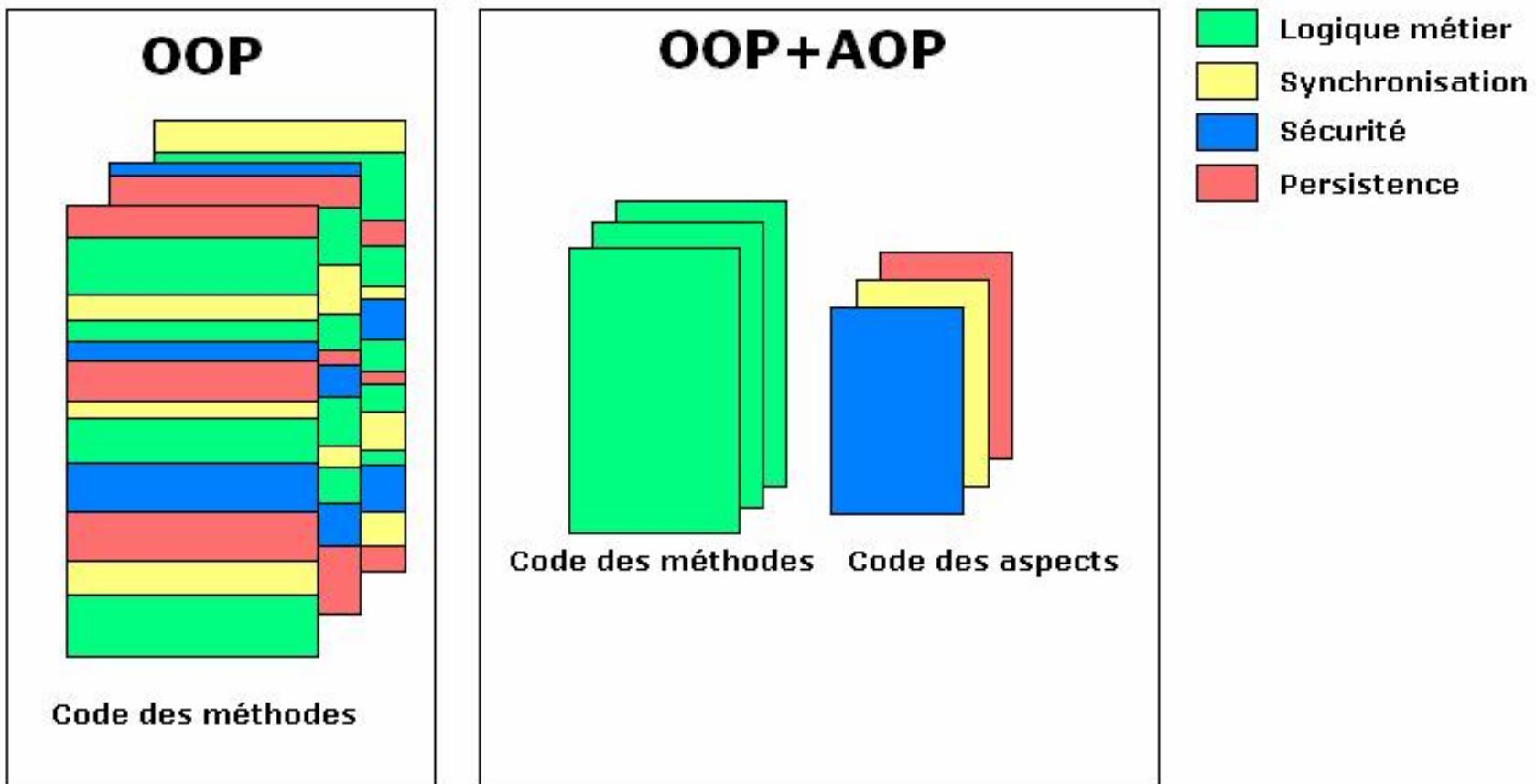
Framework et IoC

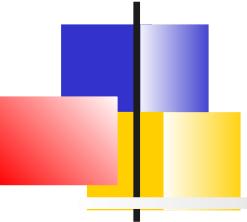
A la différence d'une simple librairie, les frameworks et middleware appliquent généralement le pattern **d'Inversion of Control**.

- Ce n'est plus notre programme qui utilise une librairie mais les frameworks qui pilotent nos composants
- Cette « inversion de contrôle » rend les frameworks comme des squelettes de programmes extensibles que l'utilisateur remplit avec ses propres méthodes.

Ces techniques s'appuient souvent sur l'*Aspect Oriented Programming* et l'utilisation de chaîne d'interception

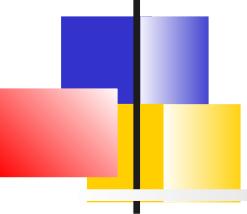
AOP et Cross-cutting concerns





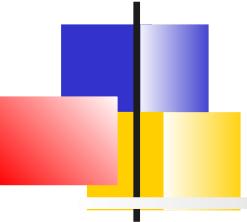
Intercepteur transaction

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



IoC versus Dependency Injection

- ❖ L'injection de dépendance est juste une spécialisation du pattern IoC
- ❖ Le framework appelle les méthodes permettant **d'initialiser** les attributs de vos objets.

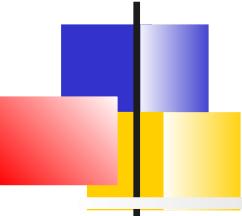


Usages

- Spring et Serveurs Java EE : Injection de ressources, ajout de comportements (transactionnel, sécurité, ...)
- AngularJS : Injection de services (scopes, objets Ajax, ...)
- .NET : Injection de dépendances
- PHP-DI : Container d'injection de dépendances



Langages back-end



Java

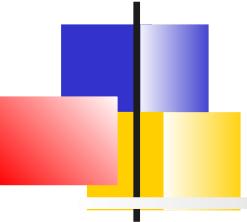


Présenté officiellement en 1995, encore le langage le plus utilisé.

Évolutions qui apportent les modernités (performance, lamda expressions, API de plus en plus riche, parallélisation, optimisation des compilateur, ...)

Syntaxe reprise par d'autres langages : *Scala*, *Groovy*, *Clojure*

Ses atouts : portabilité, évolutivité/maintenabilité, lisibilité, outils et framework



C#

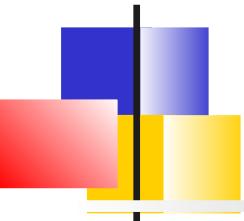
C#

Langage de la plate-forme .NET depuis 2002

Dérivé du C++ et inspiré de Java (+ la surcharge des opérateurs)

Code exécutable en MSIL, interprété à la volée

Objet, fortement typé, héritage simple, métadonnées via les attributs, garbage collector, réflexion, génériques, programmation fonctionnelle



php



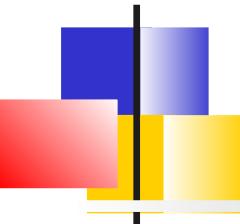
1994 : Langages de scripting, Spécification formelle depuis 2014

Généralement embarqué dans un fichier HTML,
80 % des sites web

Interprété par un module d'un serveur web
(Moteur Zend)

Programmation orienté objet basique depuis PHP3

Multi-plateforme, peu typé, souple, facile à apprendre par un débutant mais ... pas très sûr, pas très maintenable, pas très lisible



Ruby



Standardisé au Japon en 2011, puis ISO 2012

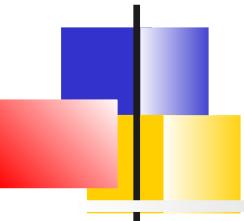
Popularisé via le framework web *Ruby on Rails*

Inspiré de Smalltalk, Syntaxe proche d'Eiffel et Ada

Interprété : l'interpréteur officiel Ruby MRI mais plein d'autre implémentations

Fortement objet, héritage simple, ramasse miettes, exception, réflexion et modification des classes au runtime, threads, typage dynamique, surcharge d'opérateurs

Librairies pour HTTP, OpenSSL, JSON, XML, ...



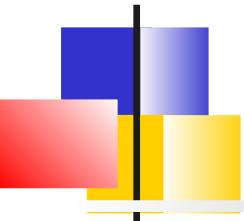
Python python™

Démarré fin des années 80

Langage de script utilisé

- côté serveur via un module web,
- côté client via *Pyjamas* et *IronPython*
- mais également utilisé pour le mapping base de données, l'intelligence artificielle, logiciel de 3D, animation, système linux, installateur linux...

Concis et lisible, orienté objet, fonctionnel, AOP
gestion de la mémoire, typage dynamique,
bibliothèque standard riche



PERL



Créé en 1987 pour faciliter le traitement des chaînes de caractères

Inspiré de C, *sed*, *awk*, *sh* ;

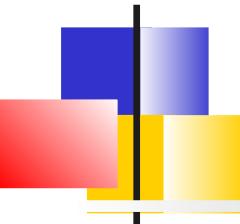
Popularisé par les sites Web et le CGI mais également dans des tâches d'administration

Procédural, prend en charge les expressions régulières ... très puissant mais pas très élégant ni lisible



Langages front-end

i.e., supportés par le navigateur



HTML5

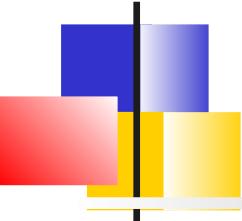


Origine 1989 / 1992.

Actuellement, spécification HTML5 et HTML Living Standard (suivre les innovations des navigateurs)

Langage à balise inspiré de SGML,

- Départ : structuration et mise en forme de documents
- Arrivée de Css : Seulement la structuration
- HTML5 : applications en ligne, extensible via XML, utilisé pour les applications mobiles
nouveautés : canvas, offline, drag and drop, contenu éditable, stockage (futur cookies), messages cross-document



CSS



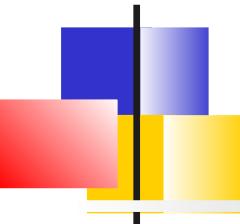
Pris en charge par les navigateurs dans les années 2000

S'occupe de la présentation du contenu web

Feuille de styles « cascadées »

Souffre des disparités entre navigateur,
CSS3 n'est pas encore terminé

Apporte le responsive design : *bootstrap*,
web starter kit, *gumby*, *foundation*, ...



Javascript



Créé en 1995 (en 10 jours !) pour le serveur HTTP de Mosaic.
Standard ECMA Script en 1997

Interprété mais également JIT compiler depuis peu. Bcp d'implémentations du moteur (*Rhino, SpiderMonkey*)

Principalement côté client web, (a souffert de la disparité des navigateurs et du DOM, a apporté Ajax et JSON) mais également document PDF, OpenOffice mobile, MongoDB et serveur avec *Node.js*

Librairies standard minimales (Math, Date, regexp)

Framework : *jquery, Angular JS1, Backbone.js, Ember.js, Spine.js, Dojo, React, Polymer* ...

https://dzone.com/articles/react-angular-bootstrap-and-polymer-the-basics?utm_source=Top%205&utm_medium=email&utm_campaign=top5%202016-09-02

Orienté objet via prototype, non typé, fonctionnel, fonction anonyme, closure

TypeScript

Langage open source de Microsoft par le créateur de c#

A pour but d'améliorer et de sécuriser la production de code JavaScript.

Le typescript est transcompilé en JavaScript

Caractéristiques :

- Typage statique optionnel des variables et des fonctions,
- la création de classes et d'interfaces,
- l'import de modules,
- Conserve l'approche non-constrictrice de JavaScript. Il
- Supporte la spécification ECMAScript 6.



Langages mobile



Plateforme Androïd

Plate-forme pour les objets connectés

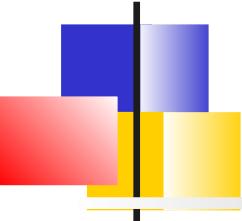
- Noyau linux
- JVM nommée Dalvik
- WebKit, OpenGL, SQLite
- Framework
- Applications standard

80 % de part des marchés dans le mobile

Développement en Java

IDE Android Studio (basé sur IntelliJ IDEA)

Upload d'applications sur Google Play



Apple

(iOS, Mac OS/X, Watch OS, TV OS ...)



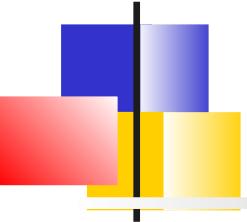
2 langages :

- *Objective C* (1980), (extension du C + Smalltalk)
basé sur la bibliothèque de classes Cocoa
- *Swift* (2014), le futur (iOs + Linux), compatible
avec Cocoa (et donc Objective C), OpenSource
Cible également les plateformes .NET et Androïd

1 IDE : *XCode*

Programmation Objet, librairies graphiques

Licence nécessaire pour uploader sur l'AppStore



Les autres

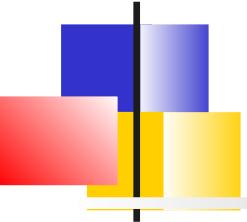
Groovy : Basé sur Java, Script, DSL
(Graddle, Jenkins)

Scala : Basé sur Java, programmation fonctionnelle et OO

Go : Google simplicité

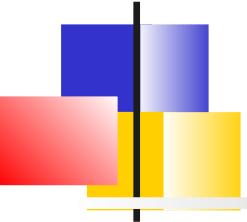
R : Data Science

Voir : <https://www.tiobe.com/tiobe-index/>



Les Design Patterns

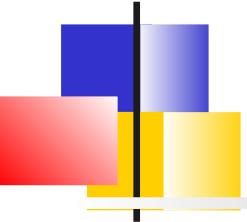
Introduction
Patterns de création
Patterns structuraux
Patterns comportementaux



Design patterns

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

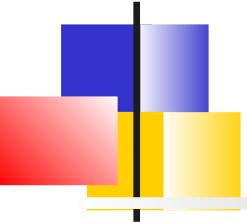
- Il décrit une solution standard, utilisable dans la conception de différents logiciels.
- Il est issu de l'expérience des concepteurs de logiciels.
- Il décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées en fonction des besoins.
- Ils ont une influence sur l'architecture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



Types de patterns

On peut distinguer :

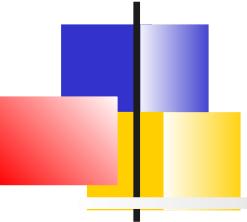
- Le **patron d'architecture** qui apporte des solutions sur la manière de concevoir l'organisation à grande échelle (architecture) d'un logiciel en faisant abstraction des détails.
Il concerne la structure générale d'un logiciel, sa subdivision en unités plus petites, comporte des guides de bonnes pratiques et des règles générales qui ne peuvent pas être traduites directement en code source.
- Le **patron de conception** qui suggère une manière d'organiser des modules ou des classes pour résoudre un problème récurrent.
Le patron de conception parle d'instances, de rôles et de collaboration



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.

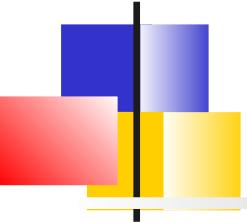
Les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70, dont son livre ***A Pattern Language*** définit un ensemble de patrons d'architecture.



Formalisme

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

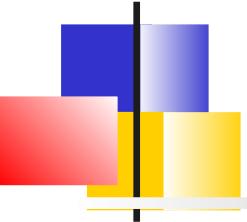
- Nom : Permettant de communiquer entre concepteurs
- Description du problème à résoudre
- Description de la solution : les éléments de la solution, avec leurs relations. i.e. diagramme de classes
- Conséquences : résultats issus de la solution.



Orthogonalité des patrons

Chaque patron doit correspondre à une approche différente, qui ne répète pas les idées ou stratégies présentes dans d'autres patrons : **l'orthogonalité**

Ceci permet au concepteur de résoudre chaque aspect d'un problème d'une façon organisée ; les patrons sont alors combinés pour construire la solution globale.



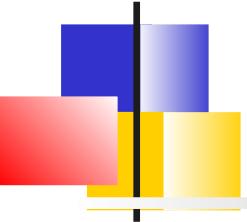
Typologie des patterns

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.



Patterns de construction



Exemple Factory

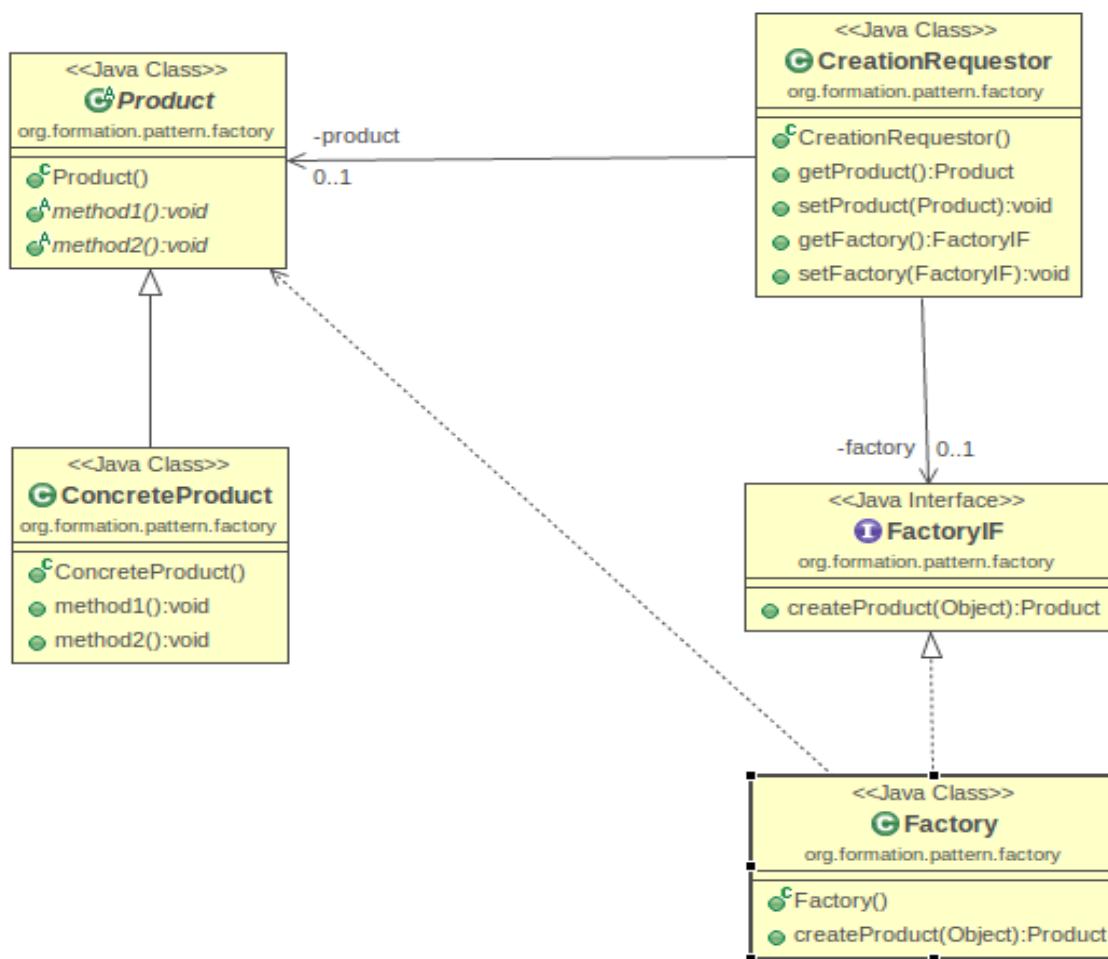
Nom : *Factory Method* [GoF95]

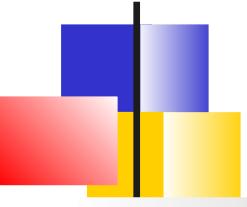
Problème à résoudre : Écriture d'une classe réutilisable avec des types de données arbitraires.

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

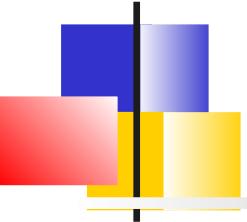
Product : Classe abstraite (ou interface) mère de toutes les implémentations

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

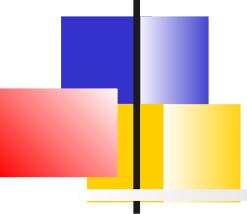
Factory : Classe implémentant l'interface *Factory* qui instancie une implémentation spécifique de *Product*



Conséquences

L'objet demandant la création de classes est indépendant des implémentations concrètes créés

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



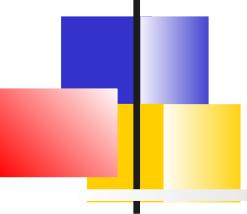
Autres patterns de construction

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



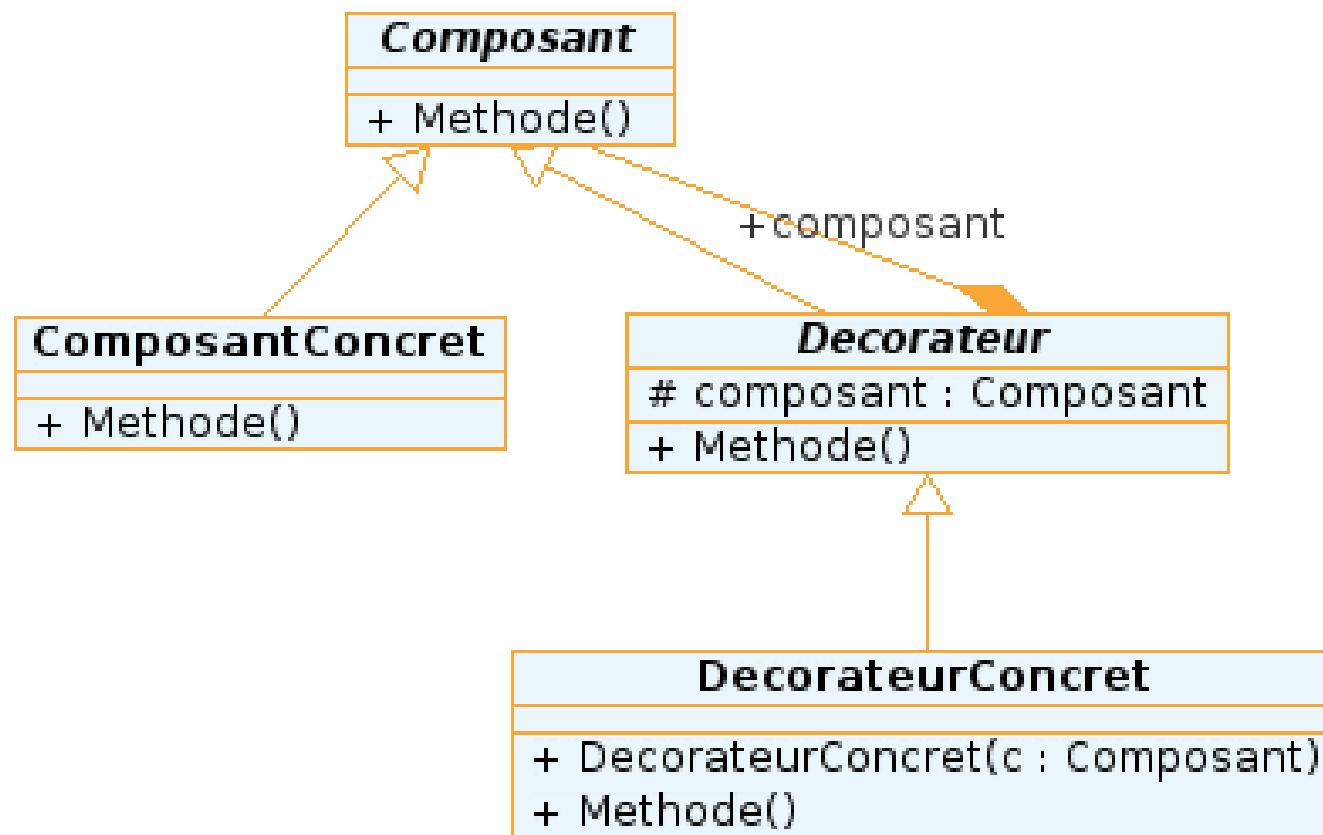
Exemple structuration : Décorateur

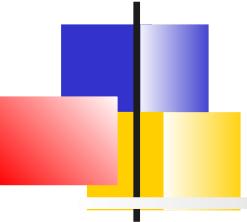
Nom : Decorator [GoF95]

Problème à résoudre :

Étendre les fonctionnalités d'un objet de façon transparente pour un client

Solution





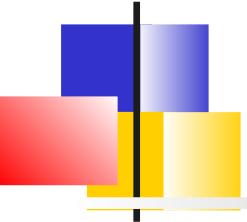
Classes impliquées

Composant : Classe abstraite ou interface mère de toutes les classes concrètes dont les fonctionnalités peuvent être étendues avec le pattern Decorateur

ComposantConcret : Classes concrètes pouvant être étendues

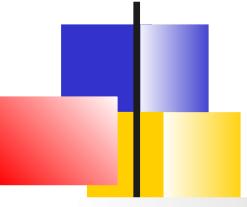
Decorateur : Classe abstraites encapsulant le composant Concret. Elle implémente l'interface Composant en appelant la méthode de sa référence concrète

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";  
    }  
}
```



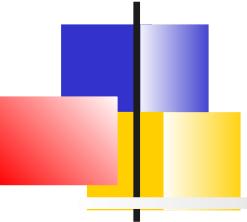
Décorateurs

```
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
    }
}

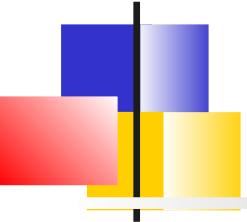
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) { super(windowToBeDecorated); }

    @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());  
  
    }  
  
}
```



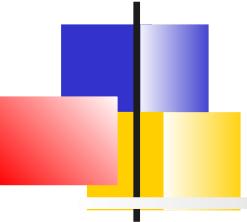
Conséquences

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)



Autres patterns structuraux

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

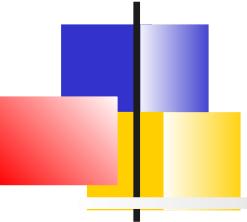
Bridge [GoF95] : Permet à une hiérarchie d'abstractions et d'implémentations d'être implémentée à bases de classes indépendantes pouvant être combinées dynamiquement

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



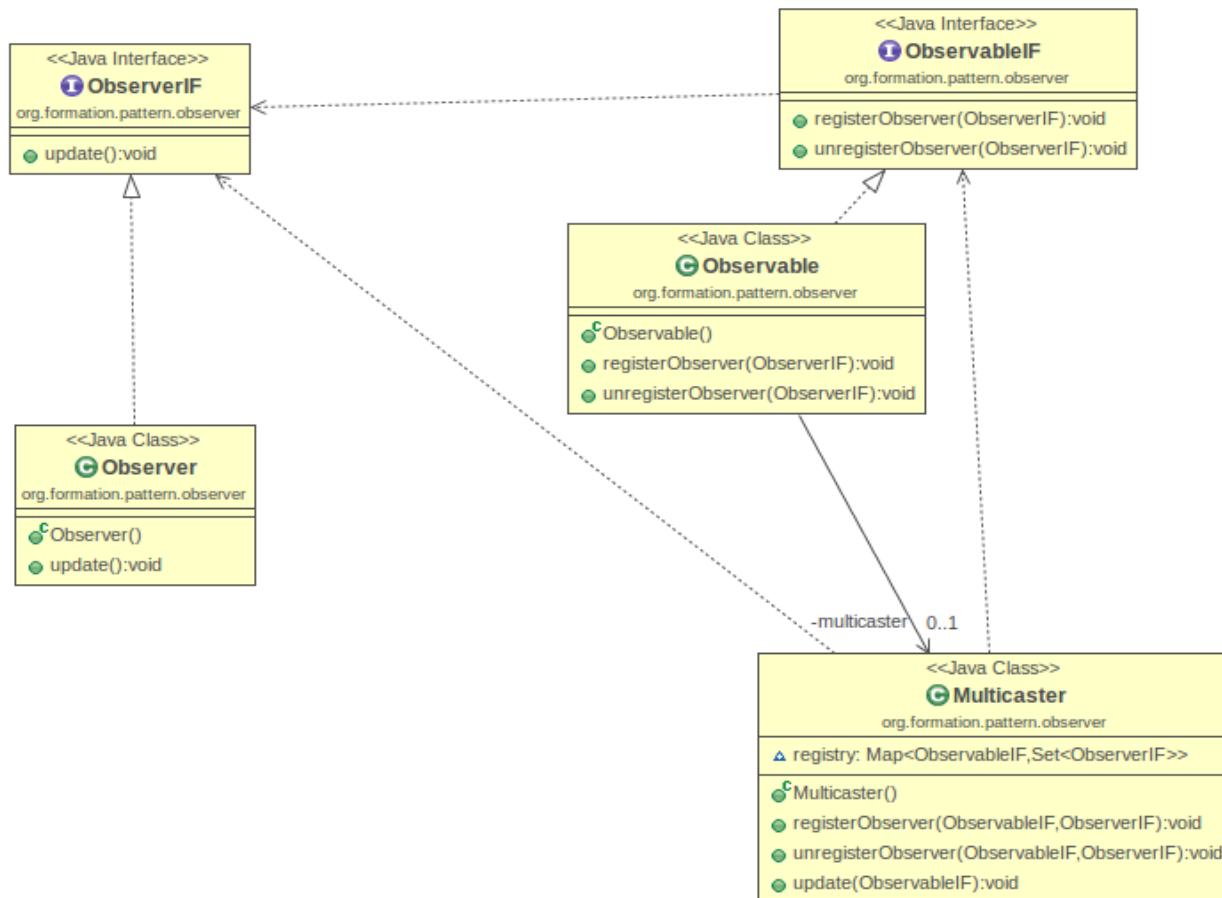
Exemple patterns comportementaux : Observer

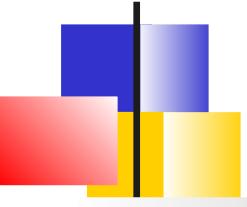
Nom : Observer [GoF95]

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

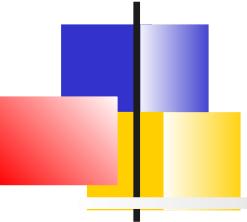
ObserverIF : Une interface définissant une méthode *notify*. Un objet *Observable* utilise cette méthode lorsque son état change

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



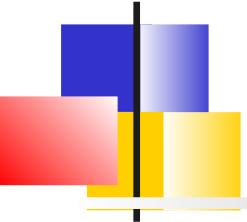
Conséquences

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



Autres patterns comportementaux

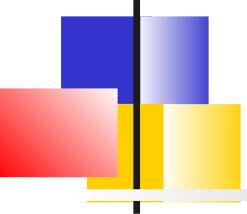
Command [GoF95] : Encapsule des commandes dans des objets afin de les manipuler

Chain Of Responsability [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



Patterns architecturaux de l'informatique distribuée

Problématiques de l'informatique distribuée

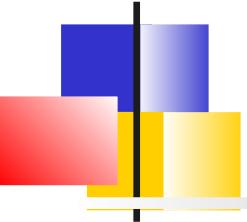
Architecture n-tiers

Services Web

Services REST

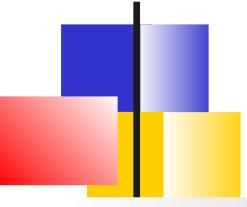
SOA, ESB, MOM

Micro-services



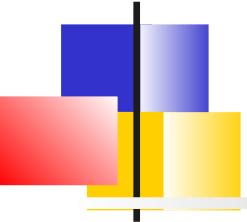
Systèmes distribués

- ❖ Le développement des réseaux a fait envisager des applications dans lesquelles les objets pourraient se trouver répartis sur plusieurs ordinateurs interconnectés.
=> On parle alors d'application **répartie** ou **distribuée**
- ❖ Définition :
Un système distribué est un ensemble d'ordinateurs indépendants vus par ses utilisateurs comme un unique système cohérent



Modèle client/serveur

- ❖ Ces systèmes distribués reposent sur le modèle **client/serveur**
- ❖ Un objet est client d'un autre objet situé sur une machine pouvant être éloignée géographiquement
- ❖ Ce type de modèle est présent à différents niveaux du système
Exemple architecture n-tiers :
 - Client léger – Serveur HTTP
 - Contrôleur MVC – Service Backend
 - Couche DAO – Serveur Base de données



Avantages de la distribution

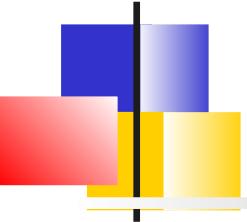
Performance, le système a plus de CPUs, mémoire à sa disposition

Scalability, répartition de charge : 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 plate-formes clientes

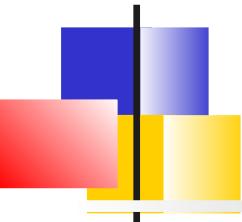


Inconvénients

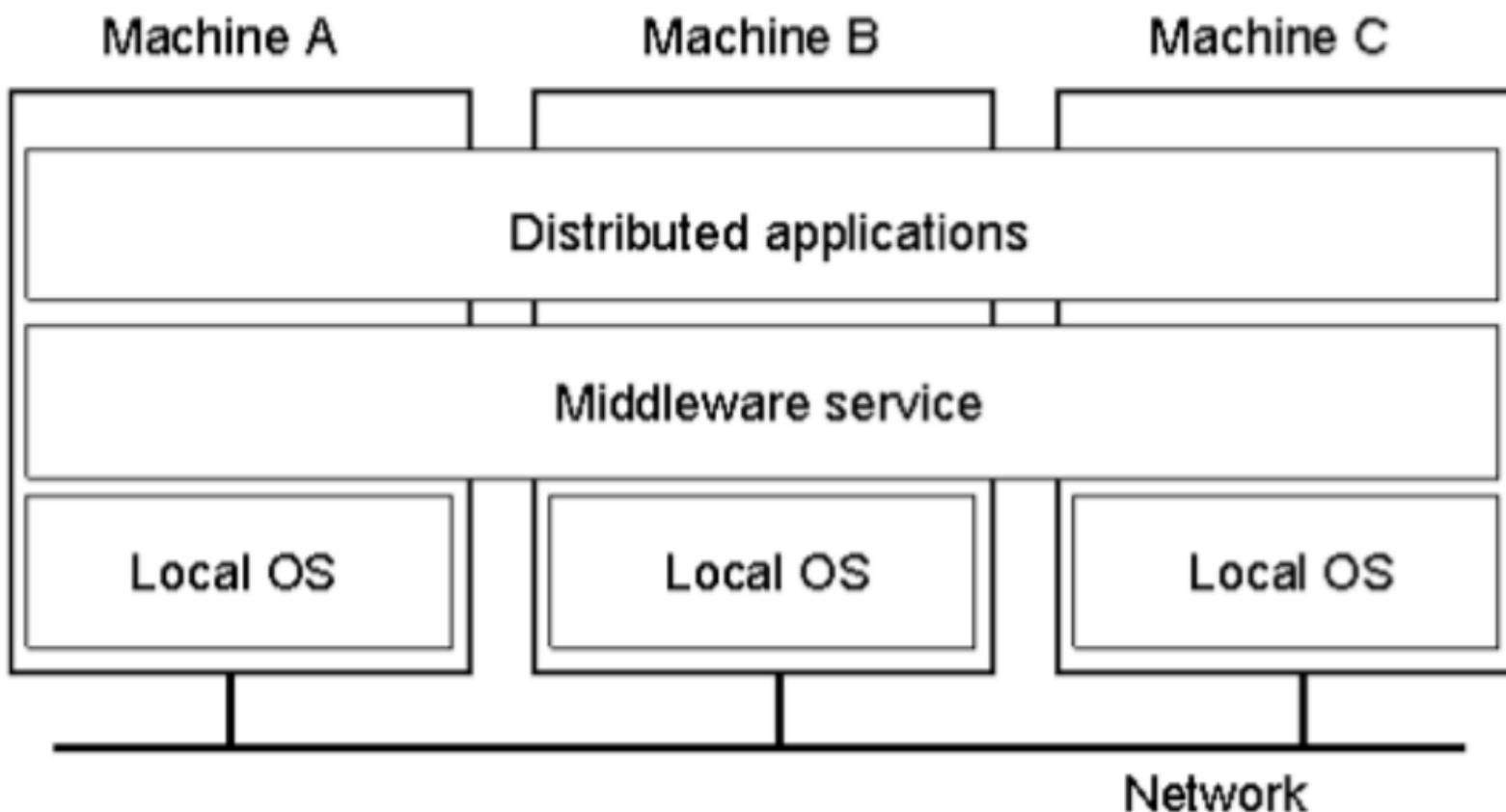
Complexité :

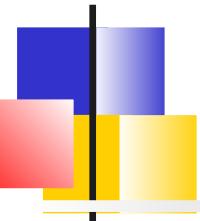
- Transfert entre 2 zones mémoire : mécanisme de sérialisation, adaptation des données, ...
- Latence et asynchronismes dus aux réseau
- Contexte de sécurité à propager
- Transactions distribuées complexes

Fragilité : plus de composants peuvent tomber en erreur



Organisation





Apport du middleware

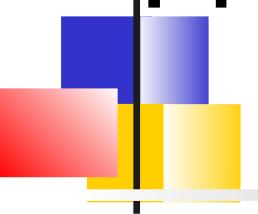
Transparence : Accès, Localisation, Réplication, Concurrence, Défaillance,

Ouverture : Permet des évolutions plus faciles

Fiabilité : Grâce à la redondance (qu'il faut gérer)

Performance : Si pas de gains, cela ne vaut pas la peine

Scalability : Possibilité d'ajouter des ressources sans interruption du service



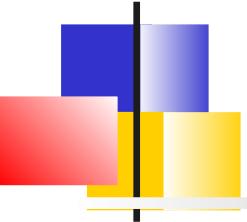
Problématiques de l'informatique distribuée

Les développeurs utilisent les services du middleware :

- Nommage des ressources => dynamicité de la localisation
- Transfert entre mémoires distantes => transparence de l'aspect *remote*

Ils doivent choisir les modèles de programmation appropriés :

- Modèle synchrone vs asynchrone
- Stateless vs stateful
- Modèle multi-thread vs single thread



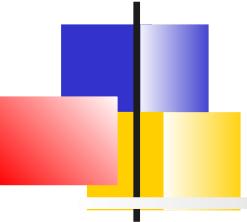
Service de nommage

Un service de nommage a pour rôle d'associer des noms à des objets ou des ressources, permettant ensuite de localiser un objet ou une ressource en utilisant le nom qui lui a été donné

Un service de nommage fournit donc deux services:

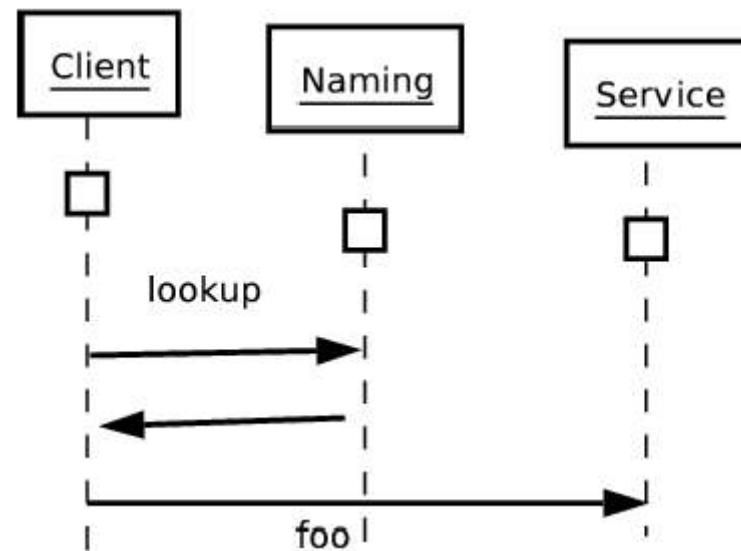
- un service **d'association (binding)** d'une ressource ou d'un objet à un nom
- un service de **consultation (lookup)** permettant de localiser la ressource ou l'objet à partir de son nom

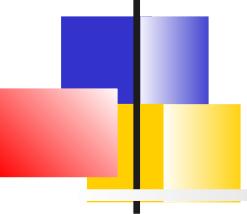
les services d'annuaires (directory services) sont des services de nommage avancés qui permettent d'organiser les ressources de façon arborescente, fournissant également des méta-données sur ces ressources



Localisation d'objet

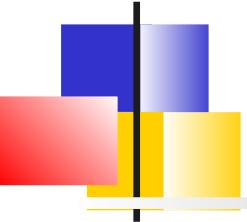
- ❖ Le lookup de la ressource est
 - soit **explicite** : Le développeur utilise une API d'accès au serveur de nom
 - soit la ressource est automatiquement **injectée** par le middleware via de la configuration ou des annotations





Services de nommage et annuaire

- ❖ Il existe de nombreux services d'annuaires
 - Internet : DNS
 - JavaEE : JNDI
 - Webservices : UDDI
 - Bus : Service de nommage CORBA
 - Java : registry RMI
 - Micro-services : Eureka, Consul, Zookeeper



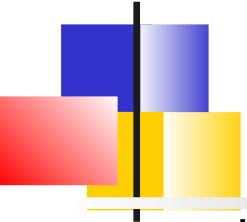
Échange de données par flux

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
- Format texte adapté au programme consommateur (ex : JSON)



Librairies

Java :

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

Javascript :

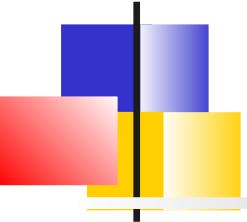
- XML : JKL Parse XML

PHP :

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

.NET :

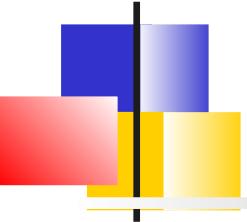
- XML : *System.Xml.Serialization*
- JSON : *JSON.Net, JavaScriptSerializer*



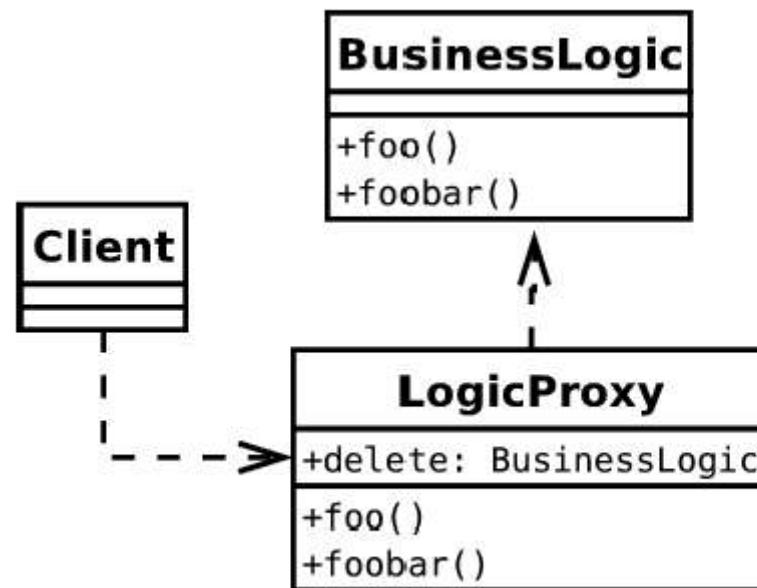
Pattern Proxy

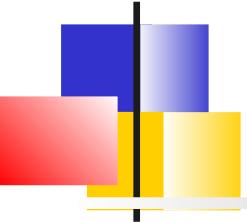
Le pattern **proxy** est un pattern très fréquemment utilisé dans les autres patterns.

- Les appels de méthodes s'effectuent indirectement via un objet intermédiaire le proxy.
- Le client ne sait pas qu'il interagit avec un proxy



Design pattern Proxy





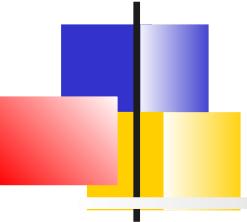
Modèle synchrone / asynchrone

Une fois la ressource distante localisée.

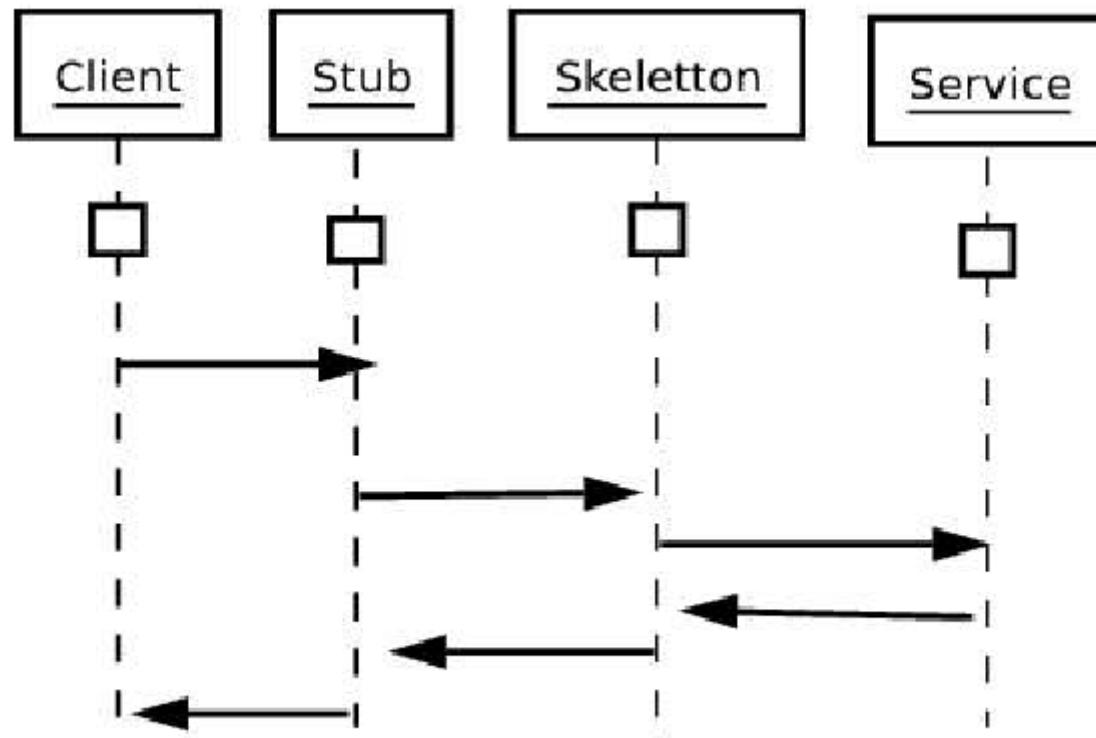
L'interaction peut être

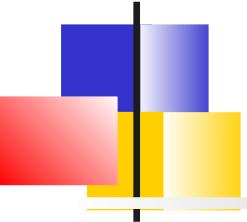
- synchrone/bloquante
- ou asynchrone/non bloquante

Le synchronisme est moins complexe,
l'asynchronisme, si il est bien utilisé
permet généralement d'optimiser les
performances



Exemple appel synchrone EJBs Remote





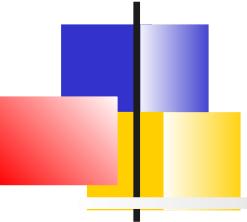
Asynchronisme Javascript

En javascript, il y a une seule thread.

La thread est en attente d'événement qu'elle distribue (dispatch) aux gestionnaires (*handler*)

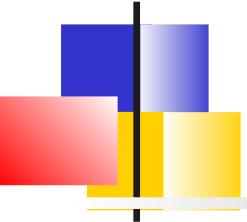
Lors d'un appel à une fonction longue (appel I/O par exemple), il est conseillé d'utiliser un appel asynchrone non bloquant.

- De façon directe, en utilisant une fonction de **callback**
- De manière plus élaborée et plus élégante en utilisant l'objet **Promise**



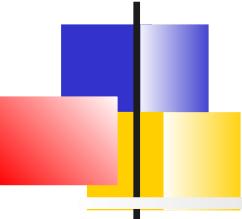
Call-back

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



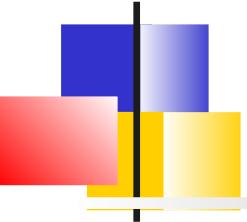
Plus élégant : *Promise*

```
// 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 ) ;
```



Asynchronisme Java et classe *Future*

```
public class CallableFutures {  
    public static void main(String[] args) {  
        ExecutorService executor = Executors.newFixedThreadPool(10);  
        List<Future<Long>> list = new ArrayList<Future<Long>>();  
        for (int i = 0; i < 20000; i++) {  
            Callable<Long> worker = new Somme();  
            Future<Long> submit = executor.submit(worker);  
            list.add(submit);  
        }  
        long sum = 0;  
        for (Future<Long> future : list) {  
            try { sum += future.get(); }  
            catch (InterruptedException e) { e.printStackTrace(); }  
            catch (ExecutionException e) { e.printStackTrace(); }  
        }  
        System.out.println(sum);  
    } }
```



Asynchronisme et MOM

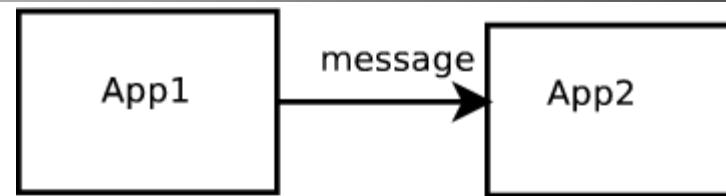
Les **MOM (Message Oriented Middleware)** existent depuis plusieurs dizaines d'années.

C'est un concept de programmation distribuée qui permet de découpler les applications qui interagissent (à l'inverse d'un modèle de style *RPC*)

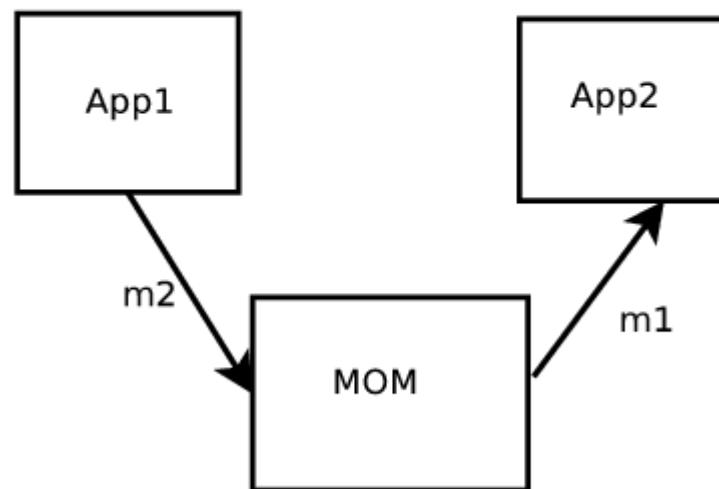
- Un émetteur poste un message (le + souvent données XML)
- Le MOM garantit la livraison du message à ses destinataires
- Les destinataires traitent le message
- Si une réponse est requise, le canal de réponse est indiqué dans le corps du message

=> L'émetteur et le récepteur n'ont pas de dépendances et le traitement du message est généralement asynchrone

Message Oriented Middleware vs RPC

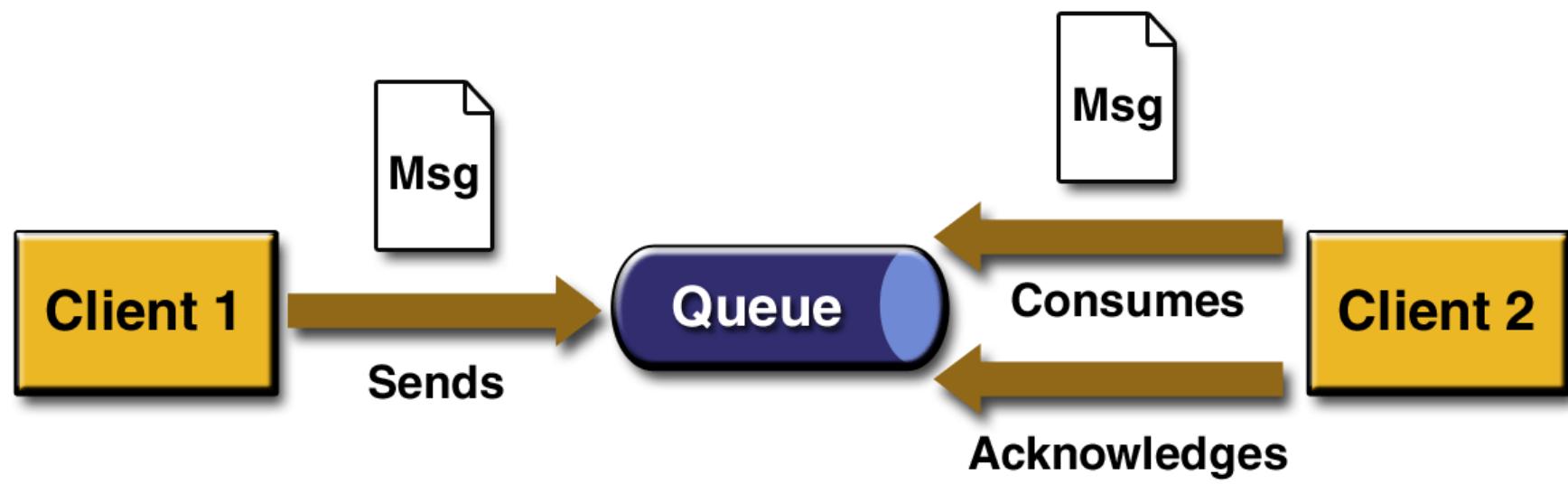


communication synchrone

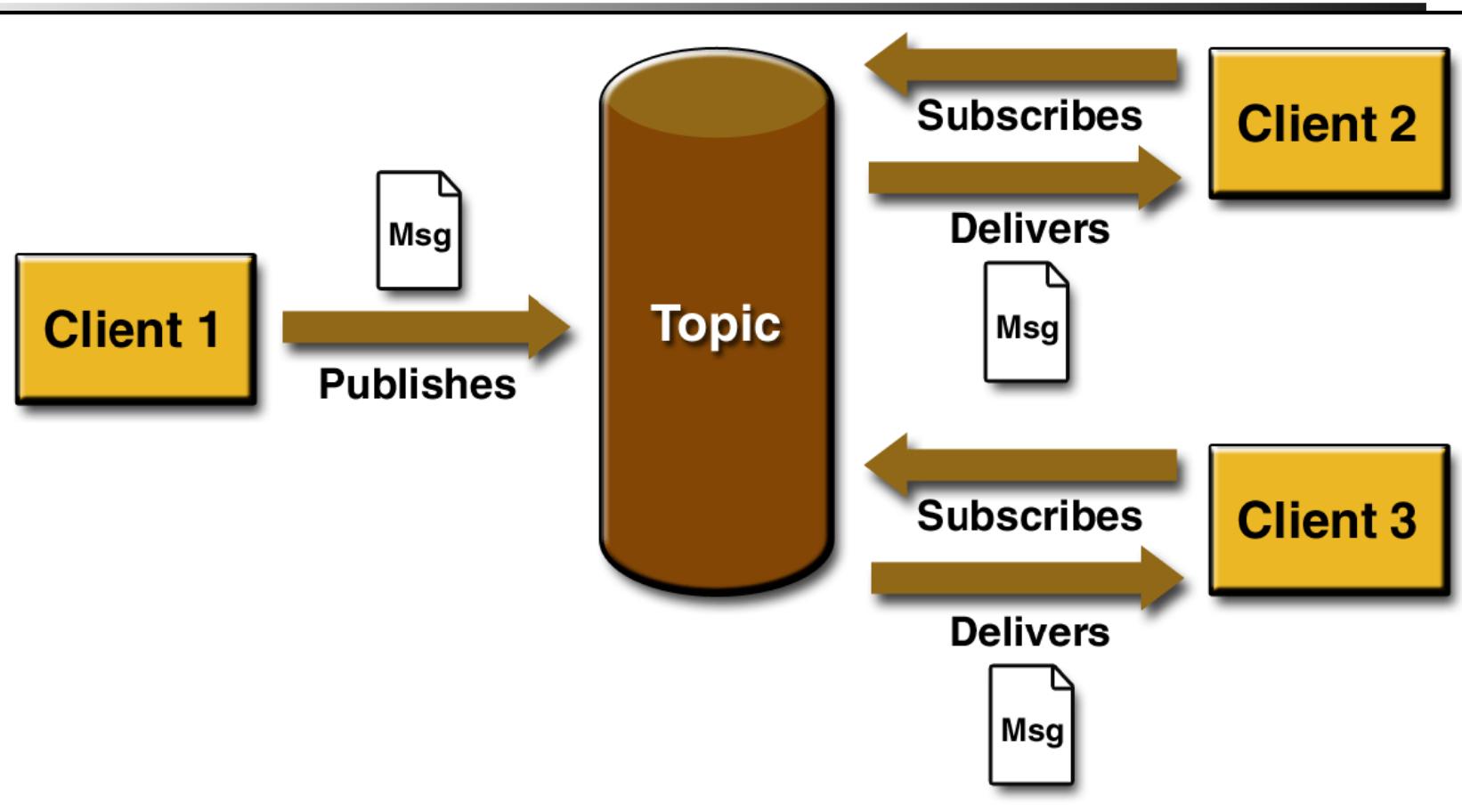


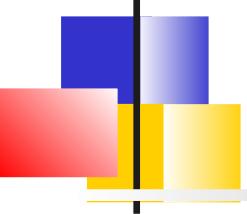
communication asynchrone

Point to Point



Publish/Subscribe





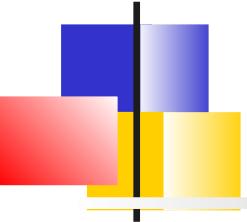
Qualité de service

MOM garantit que les messages seront délivrés même si les applications réceptrices sont «indisponibles» au moment de l'émission

Différentes qualités de service peuvent être associées aux messages :

- *Exactement-un* : Garantit que le message ne sera envoyé qu'une seule fois à chaque destinataire
- *Au moins-un* : Le message peut être envoyé en doublon
- *Au plus-un* : Certains messages peuvent ne pas arriver à destination
- *Ordonné* : Les messages sont délivrés dans leur ordre d'émission

=> Plus la qualité de service est importante moins le débit de transfert est important



Implémentations

Java EE, Spécification JMS

- ActiveMQ, HornetQ, Oracle AQ,
Websphere MQ (MQ Series)

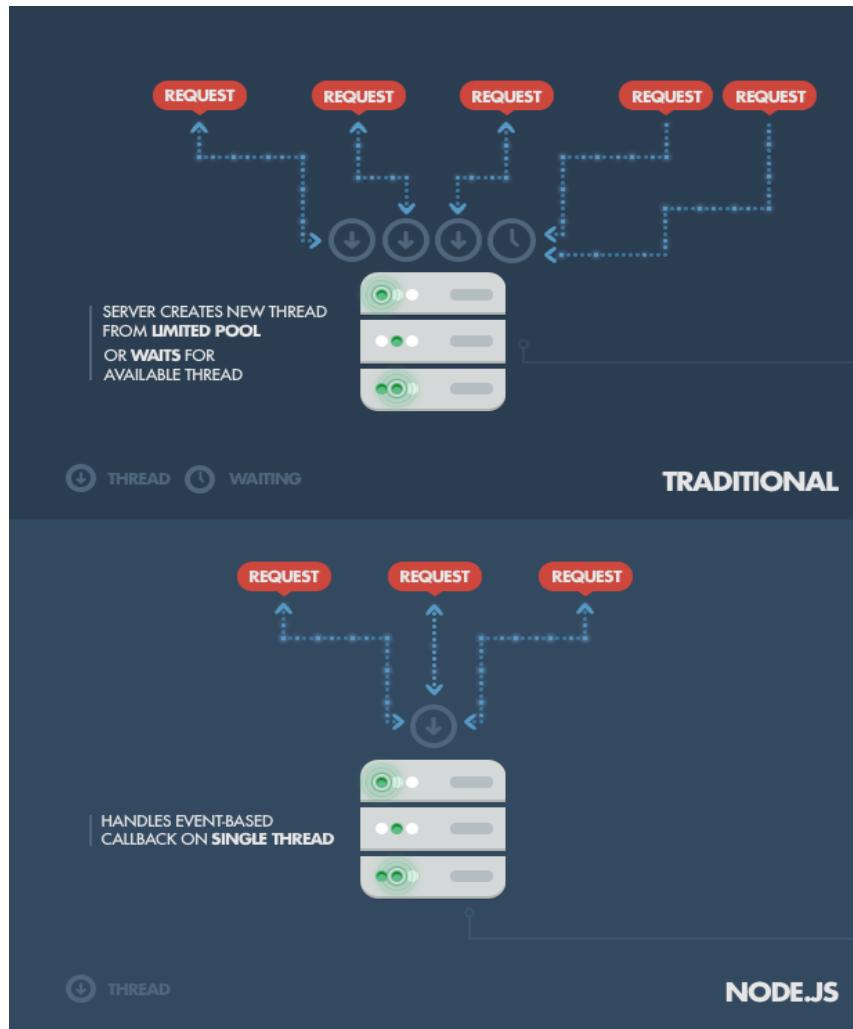
.NET : DotNetMQ

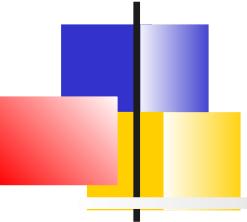
Tibco

MSMQ de Microsoft

...

Serveurs Multi-thread vs Mono-thread





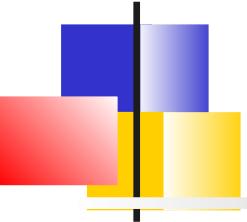
Serveurs Multi-thread

Les techniques des serveurs web traditionnels consistent à utiliser des pool 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



Exemple Configuration Tomcat

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

<Connector port="8080"

    maxThreads="250"

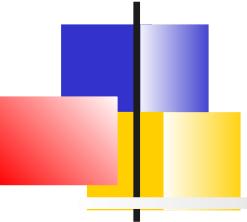
    maxHttpHeaderSize="8192"

    emptySessionPath="true"

    enableLookups="false" redirectPort="8443"
acceptCount="100"

    connectionTimeout="20000"

    disableUploadTimeout="true"/>
```



Serveur mono-thread : *Node.js*

Node.js se distingue en étant mono-threadé (javascript).

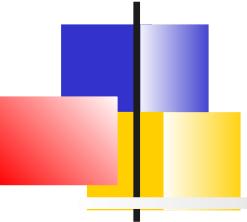
Une boucle sur événements (arrivée de requête) utilise des appels I/O asynchrone non bloquants.

=> très adapté aux systèmes effectuant de nombreuses requêtes réseau,

=> Pas adapté si des traitements lourds bloquants sont nécessaires pour traiter la requête.

Node.js est également très facilement scalable (cluster de nœuds)

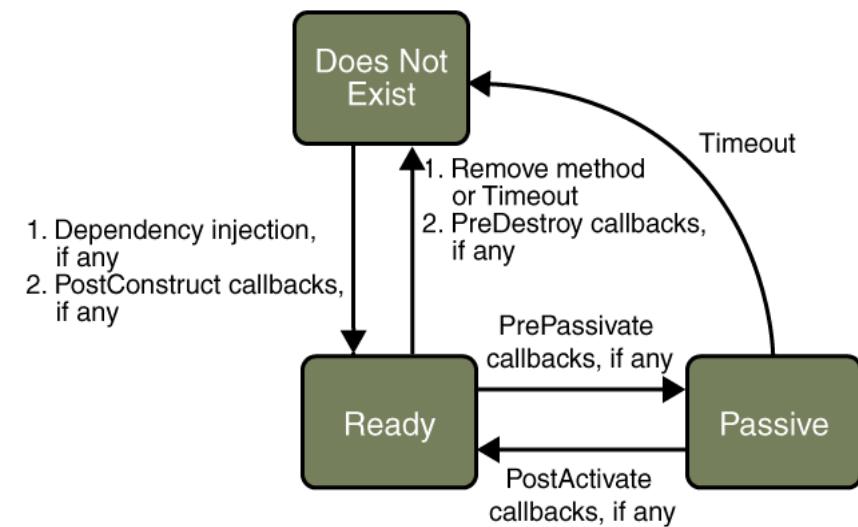
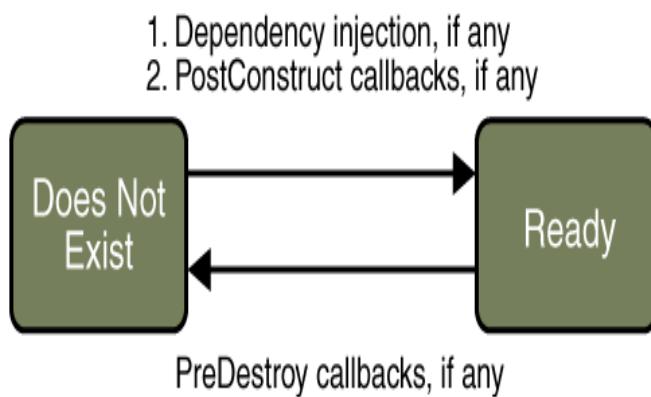
=> En cluster, il peut profiter pleinement de toutes les ressources CPU disponibles comme un serveur multi-thread.



Stateless versus Stateful

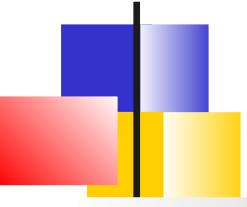
- ❖ **Stateless** : Le serveur est amnésique, il ne se rappelle pas des requêtes précédentes.
 - Tâches du serveur/container facilitées. Gestion des pool par ex.
 - Plutôt performant. (Peu d'instanciation, réutilisation des objets)
 - Pas terrible pour la bande passante.
 - Très scalable (pas de nécessité à répliquer un état entre des nœuds d'un cluster)
 - Exemple : Services REST, Spring, ...
- ❖ **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 : EJB session stateful, JSF

Exemple EJB stateful vs stateless





Architecture n-tiers



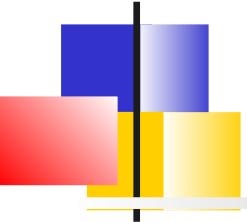
Définition

La logique applicative est divisée en composants selon leurs fonctions (les tiers)

Les composants peuvent être installés sur des machines différentes

L'architecture la plus courante est une architecture 3-tiers : Les machines clientes, le serveur d'application et les bases de données ou système propriétaires

Ce modèle étend les architectures standards client-serveur 2 tiers en plaçant un serveur d'application multi-threadé entre le client et le back-end de persistance.

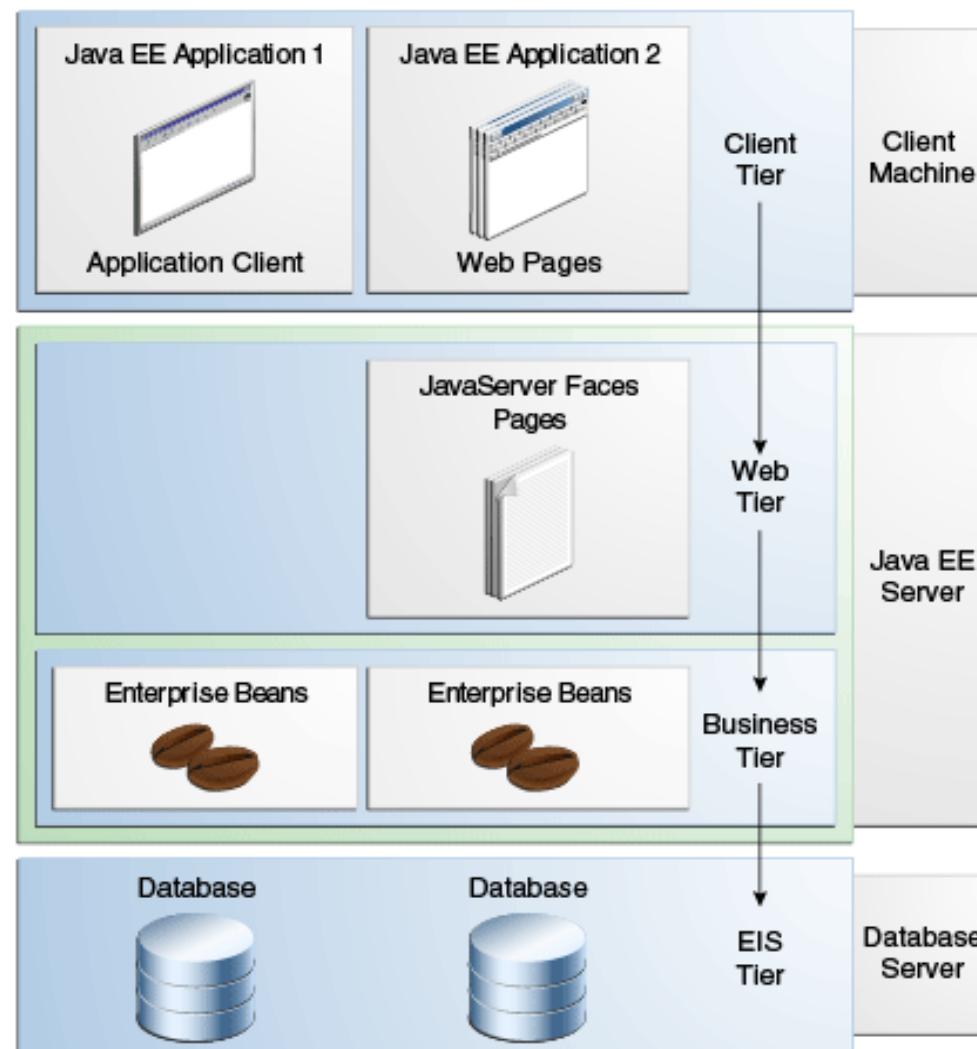


Serveurs applicatifs

Les architectures n-tiers s'appuient sur des serveurs applicatifs multi-threadés qui offrent de nombreux services techniques aux développeurs :

- Transactions distribuées (JTA)
- Sécurité (Authentification, Autorisation)
- Service de nommage (JNDI)
- Pools de threads, de connexions BD, d'EJBs
- Caches distribués
- MOM
- Injection de dépendance, Intercepteurs
- Clustering

Tiers Java EE



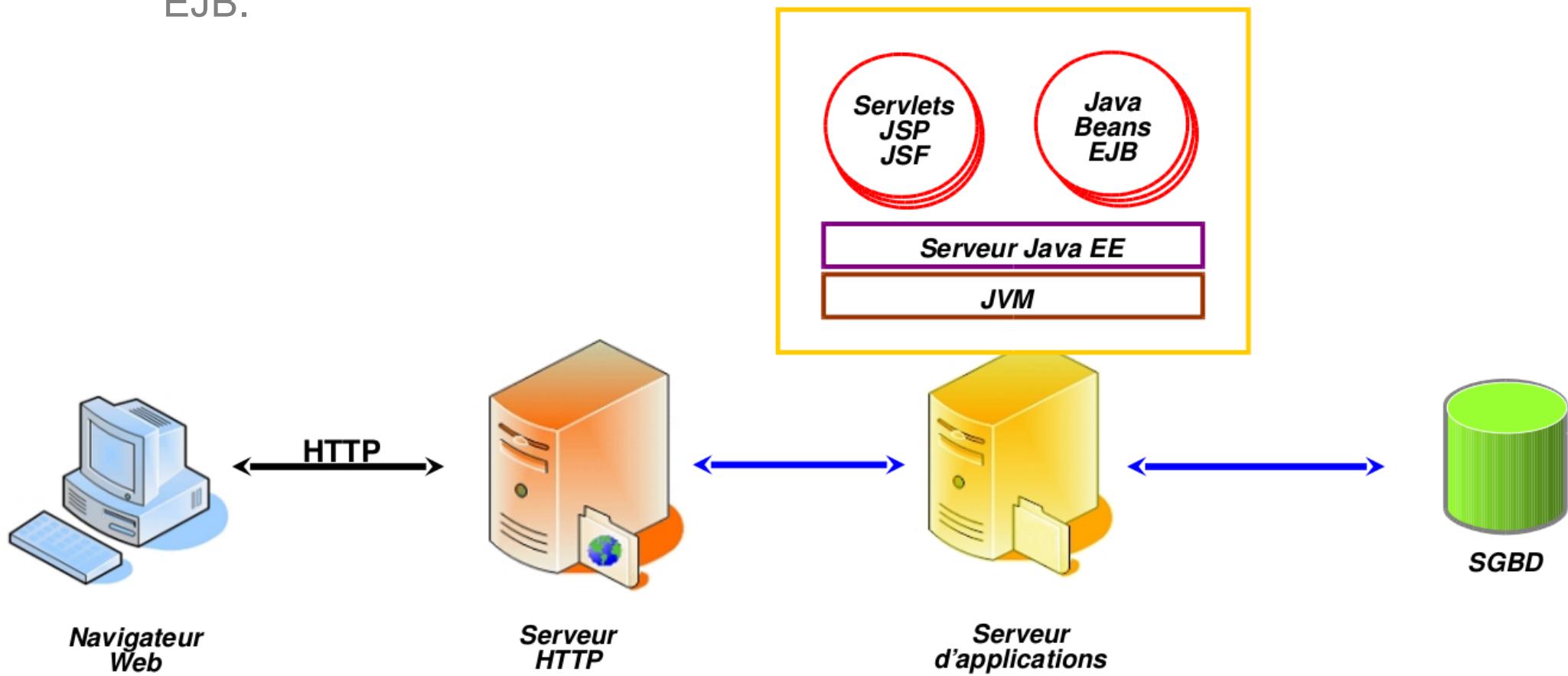
Architectures de déploiement (1)

- Application Web classique :

Partie présentation : Servlet, JSP et

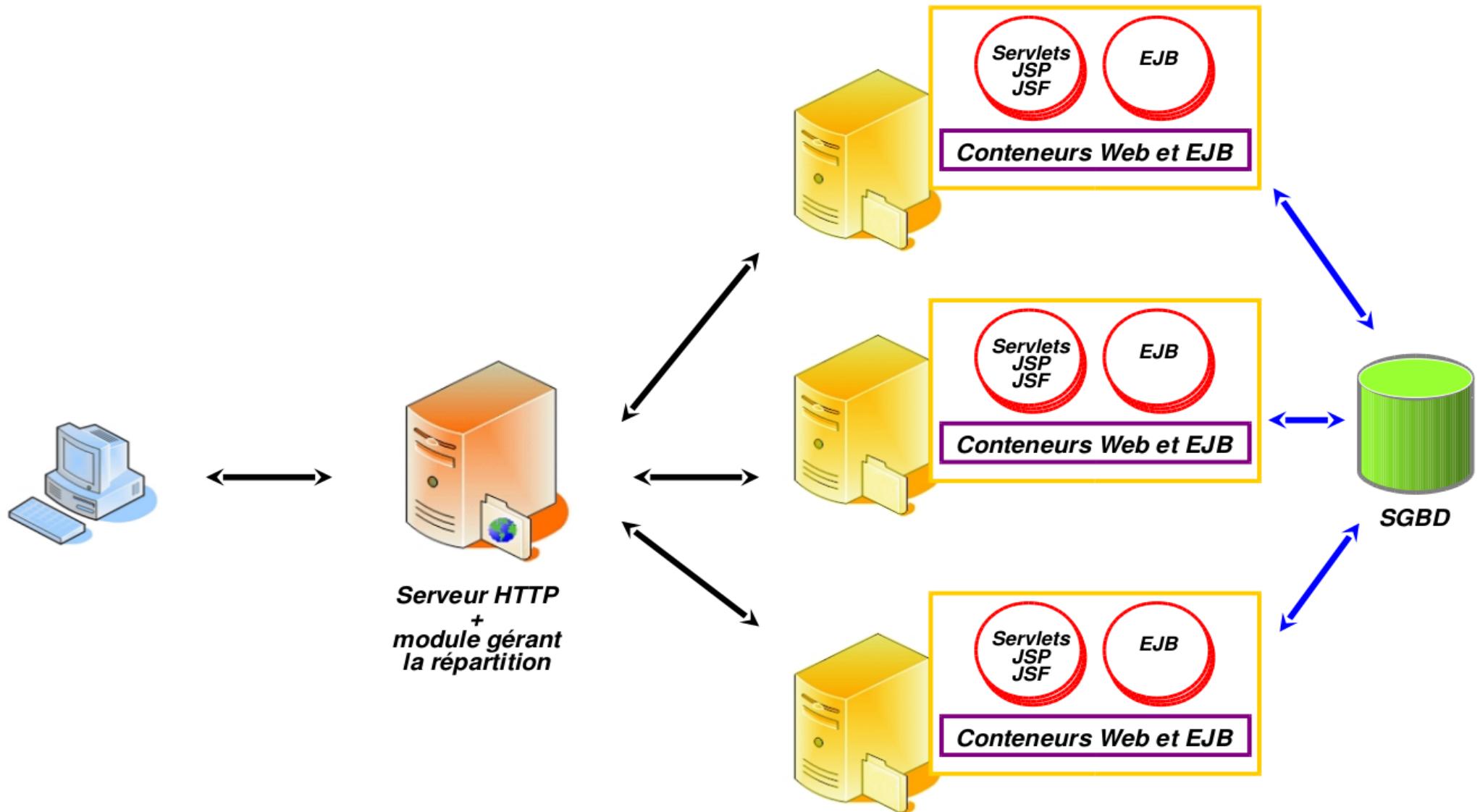
JSF. Partie modèle : JavaBeans ou

EJB.

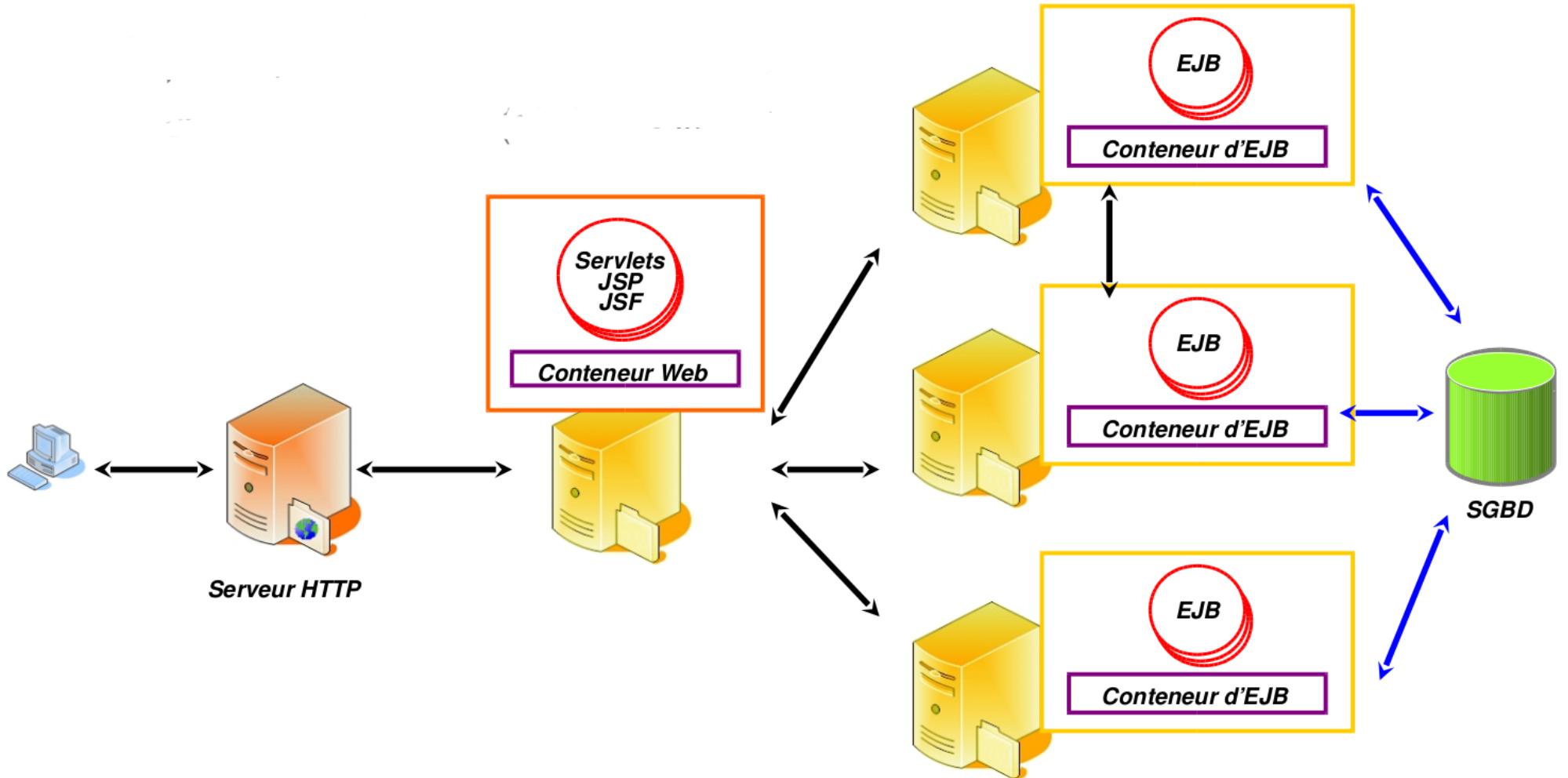


Architectures de déploiement (2)

- Répartition des requêtes HTTP par le serveur HTTP :

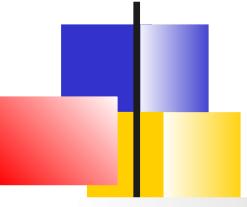


Architectures de déploiement (3)





Services web



Définitions

Un service est un composant logiciel fourni via un réseau et accessible par un point de terminaison (**endpoint**).

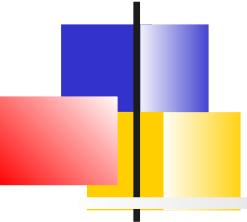
Le fournisseur et le consommateur de service utilisent des **messages** pour échanger les requêtes d'invocation et les réponses

Les messages sont des documents autonomes faisant le minimum d'hypothèses sur les capacités technologiques du récepteur.

Sur le plan technique, les services web basés sur http(s) peuvent être implémentés différemment. Les principaux types étant :

- SOAP Web services
- "RESTful" web services.

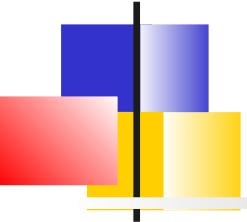
Les services web peuvent être combinés afin de construire des services sophistiqués à forte valeur ajoutée.



XML

Les services web **SOAP** sont caractérisés par leur grande interopérabilité, leur extensibilité et leurs descriptions pouvant être traitées automatiquement.

- Ces caractéristiques sont apportées principalement par **XML**.



Rappels XML

Un document XML est un document à balises :

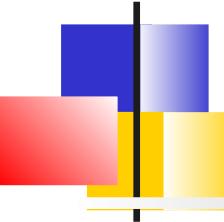
- Bien Formé
- Respectant éventuellement un schéma (.xsd)

L'utilisation des schémas permet :

- la validation d'un document,
- les fonctionnalités de complétion d'un éditeur,
- La documentation

Les schémas sont identifiés par une URI et définissent un espace de noms (**namespace**)

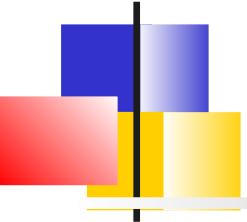
Un document XML peut ainsi agréger différentes balises provenant de schéma différents via les namespaces



Avantages de XML

Adopter XML pour représenter ses données a de nombreux avantages :

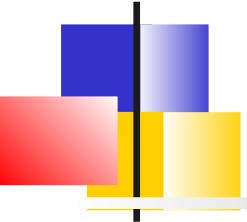
- XML est **universellement adopté**, de nombreuses applications ont déjà des entrées/sorties en XML
- Modèle de représentation de **riche** : Données hiérarchiques, listes, types complexes
- **Self-description** : XML embarque ses méta-données
- XML est **dynamique** : XML est faiblement couplé à son schéma. Les requêtes (exemple Xpath) ne sont pas dépendantes du schéma. (A l'opposé d'une requête SQL)



Avantages de XML (2)

Adopter XML pour représenter ses données a de nombreux avantages :

- Le format est **souple** : Les données d'un document XML n'ont pas de contraintes de taille, l'ordre n'est pas imposé, ...
- XML est **lisible**. En général, la lecture d'un fichier XML permet de comprendre quelles données, il représente
- XML est **extensible**, on peut ajouter ou modifier des portions de XML sans que les applications dépendantes d'autres portions soient affectés. C'est la résilience au changement

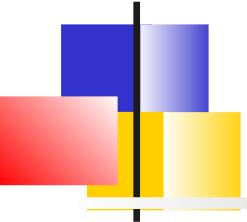


Inconvénients de XML

Rigueur : L'utilisation de schéma crée des exigences supplémentaires

Verbosité : Le volume des données à échanger est fortement augmenté par les balises, les références aux schémas, les espaces de noms

Exigences sur le client : Le client doit intégrer un parseur XML



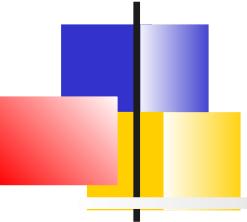
Technologies associés

XPath est un langage d'expression pour accéder à des nœuds dans un document XML.

Il utilise des éléments XML séparés par des barres obliques (/) pour délimiter la hiérarchie

XQuery permet de requêter sur un ensemble de documents et de reconstruire un document XML

XSLT permet d'appliquer de transformer un document XML en un autre document XML en appliquant une feuille de style. Les feuilles de style utilisent Xpath. Permet par exemple de faire des adaptations de format



Exemple XSLT

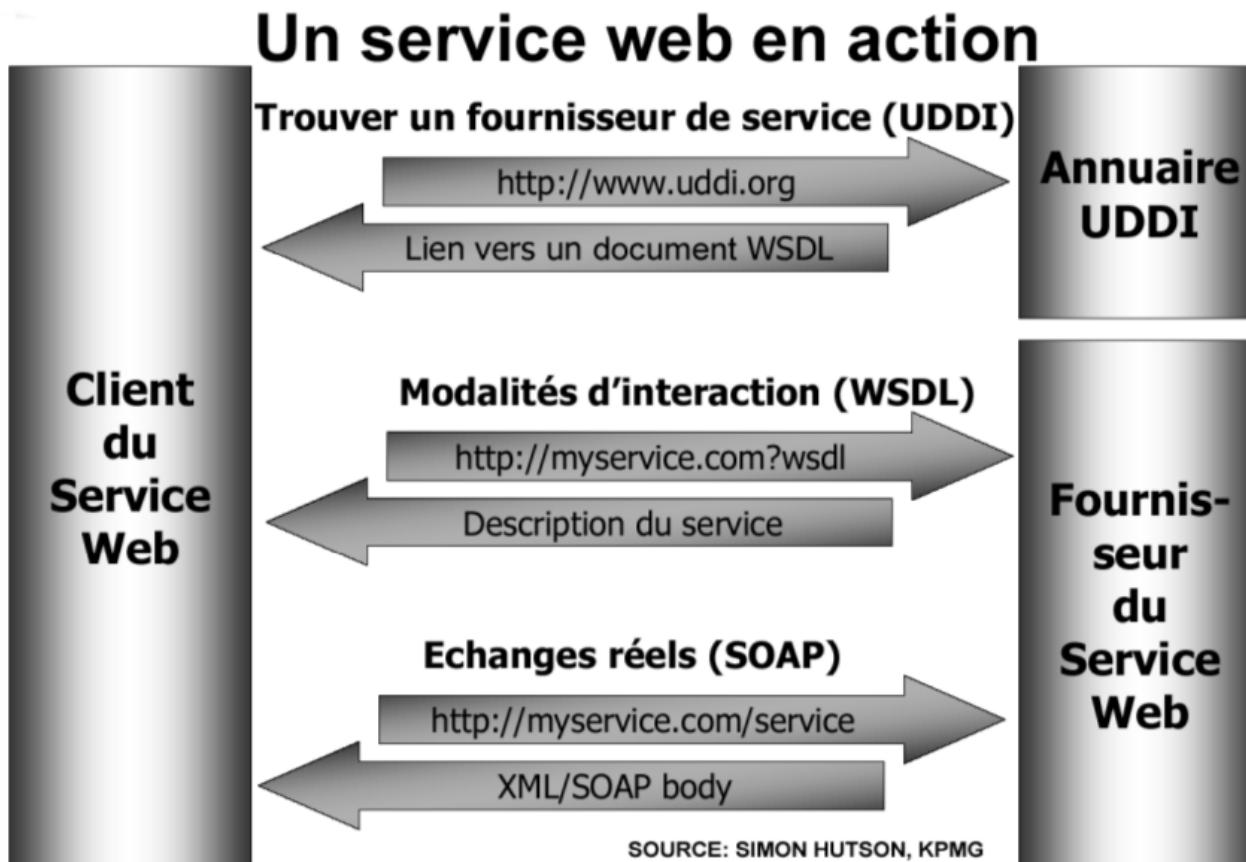
```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<!-- XSLT utilise XPath -->
<xsl:template match="/livre">
  <xsl:element name="book">
    <xsl:attribute name="isbn">
      <xsl:value-of select="@id" />
    </xsl:attribute>
    <!-- XSLT est récursif -->
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

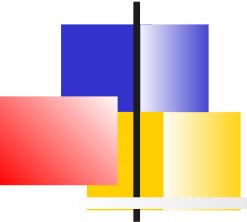
<xsl:template match="sections">
  <chapters>
    <xsl:apply-templates/>
  </chapters>
</xsl:template>

<xsl:template match="section">
  <xsl:value-of select="@titre"/>
</xsl:template>

</xsl:stylesheet>
```

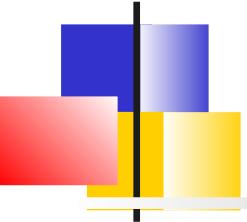
Scénario complet , service Web





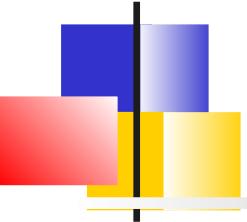
SOAP

- **SOAP** est un protocole RPC. Il effectue des appels à des objets distants via un protocole de transport.
(HTTP, SMTP, FTP)
- Bien que les objets soient sérialisés en XML, les clients SOAP voient des objets.
- L'enveloppe SOAP est constituée de 3 partie :
 - ◆ Une entête
 - ◆ Un corps de message
 - ◆ A l'intérieur du corps, éventuellement une faute



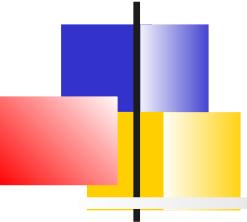
Exemple SOAP

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:doubleAnInteger xmlns:ns1="urn:MySoapServices">
<param1 xsi:type="xsd:int">123</param1>
</ns1:doubleAnInteger>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```



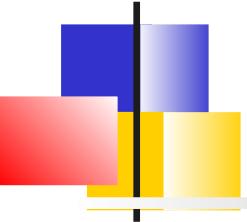
WSDL

- **WSDL** est une normalisation regroupant la description des éléments permettant de mettre en place l'accès à un Web Service
- C'est un document XML qui débute par la balise **<definition>** et qui contient les balises suivantes :
 - ◆ **<message>** : définition de données en entrée et sortie du service
 - ◆ **<portType>** : description des méthodes disponibles sur le WS
 - ◆ **<binding>** : protocole de communication (RPC/DOCUMENT)
 - ◆ **<service>** : URL du service



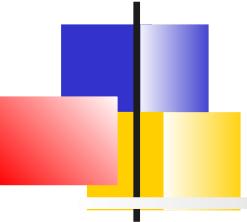
Exemple (1/2)

```
<?xml version="1.0" ?>
<definitions name="WSbibliotheque"
targetNamespace="http://corail1.utt.fr:8080/ws/WSbiblio.wsdl"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/" >
<!-- Données d'entrée/sortie -->
<message name="getPrixRequest">
    <part name="livre" type="xsd:string" />
</message>
<message name="getPrixResponse">
    <part name="return" type="xsd:float" />
</message>
<!-- Méthodes disponibles du service web -->
<portType name="WSbiblioPortType">
    <operation name="getPrix">
        <input message="getPrixRequest" name="getPrix"/>
        <output message="getPrixResponse" name="getPrixServeur"/>
    </operation>
</portType>
```



Exemple (2/2)

```
<!-- Protocole de communication -->
<binding name="WSbiblioBinding" type="WSbiblioPortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getPrix">
        <soap:operation soapAction="" />
        <input><soap:body use="encoded" namespace="urn:cite_biblio"/></input>
        <output><soap:body use="encoded" namespace="urn:cite_biblio"/></output>
    </operation>
</binding>
<!-- URL du service -->
<service name = "WSBibliothèque">
    <documentation>Prix d'un livre</documentation>
    <port name = "WSbiblioPortType" binding="WSbiblioBinding">
        <soap:address location="http://corail1.utt.fr:8080/soap"/>
    </port>
</service>
</definitions>
```

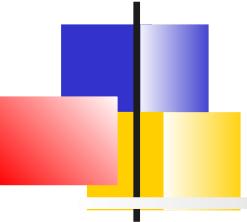


Sécurité et web services

WS-Security fournit les moyens pour sécuriser les services.

Il s'appuie en général sur HTTPS et permet de :

- Passer des jetons d'authentification entre les services
- Crypter les messages ou des parties de messages
- Signer les messages



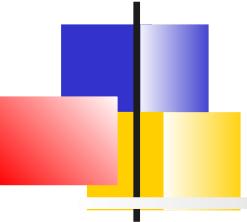
Librairies Framework

Java : JAX-WS, Apache Axis 2, Apache CXF, Spring WS

.NET : ASP.NET Web API, WCF

Php : Zend Soap

Javascript : *node-soap* (node.js), librairies clientes, package soap npm, ...



Exemple JAX-WS

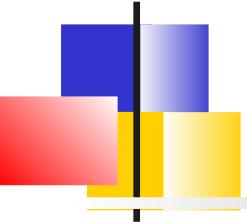
```
package helloservice;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
    private final String message = "Hello, ";

    public Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```



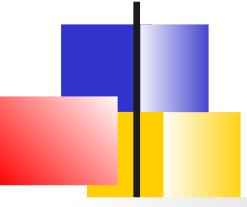
Consommation

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {

    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```



JAX-WS Annotations

@WebService : Classe d'implémentation du service Web

@SOAPBinding : Mapping SOAP

@WebMethod : Méthode exposée par le service

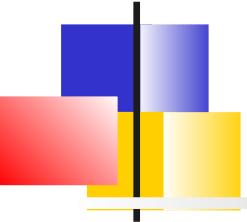
@RequestWrapper et **@ResponseWrapper** : Classe Java encapsulant les paramètres SOAP d'entrée ou de sortie

@WebFault : Mapping Exception Java vers wsdl:fault

@Oneway : Indique le service ne renvoie pas de réponse

@WebParam : Paramètre d'entrée/sortie du service

@WebResult : Spécifie la partie SOAP concernant la valeur de retour

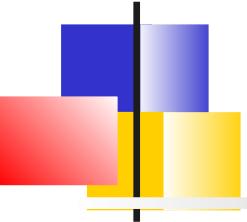


Exemple .NET

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace PetitExemple
{

    public class Exemple : System.Web.Services.WebService {
        [WebMethod]
        public int Additionne(int a, int b) {
            return a + b;
        }
    }
}
```



Consommation WebService .NET

Génération du proxy avec l'adresse WSDL :

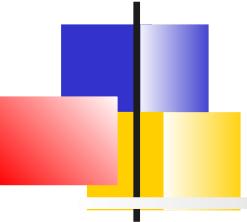
```
wsdl url_du_service_web?WSDL /out:myProxy.cs
```

Utilisation du proxy :

```
namespace UtilisePetitExemple {  
    public class ExempleConsommation : System.Web.UI.Page {  
        protected System.Web.UI.WebControls.Label reponse_webservice;  
  
        private void Page_Load(object sender, System.EventArgs e) {  
            reponse_webservice.Text=new myProxy().Additionne(5,10).ToString();  
        }  
    }  
}
```



Services REST

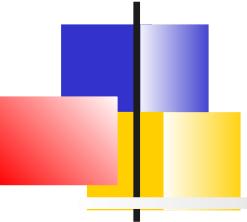


Introduction

REST est adapté pour des scénarios d'intégration basiques. A la différence de SOAP, il ne nécessite pas des messages XML ni de définition de l'API via WSDL.

Les services web **RESTful** utilisent les standards W3C et IETF (HTTP, XML, URI, MIME) et ont une infrastructure légère

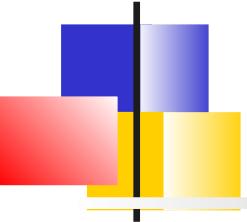
Ils peuvent donc être construits avec un outillage minimal et à faible coût .



Cas d'utilisation

Un design *RESTful* est approprié si :

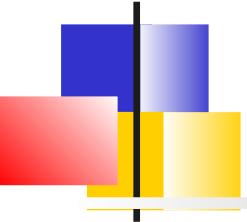
- Les services web sont stateless
- Une infrastructure de cache peut être mise en place pour booster les performances
- Le producteur et le consommateur de service ont une connaissance mutuelle de leur contexte. Chaque partie doit alors s'entendre sur le format de données à échanger.
 - Il faut noter que les applications commerciales qui exposent des services *RESTful* fournissent également des boîtes à outils décrivant les interfaces nécessaires dans différents langages.
- REST est particulièrement utile pour des devices légers (téléphone, ...) qui ne veulent pas de la surcharge dues aux entêtes SOAP et au parsing XML



Representational State Transfer (1/3)

REST (*Representational State Transfer*) a été décrit pour la première fois par Roy Fielding en 2000

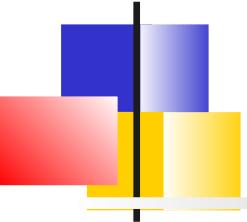
- **Client-serveur** : les responsabilités sont séparées entre le client et le serveur. L'interface utilisateur est séparée de celle du stockage des données. Cela permet aux deux d'évoluer indépendamment.
- **Sans état** : chaque requête d'un client vers un serveur doit contenir toute l'information nécessaire pour permettre au serveur de comprendre la requête, sans avoir à dépendre d'un contexte conservé sur le serveur. Cela libère de nombreuses contraintes sur le serveur.
- **Mise en cache** : le serveur envoie une réponse qui donne l'information sur la possibilité de cette réponse à être mise en cache (date de péremption, ...).



Representational State Transfer (2/3)

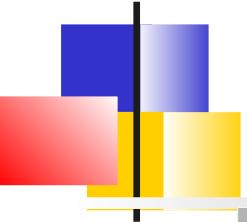
Une interface uniforme ; cette contrainte agit selon quatre règles essentielles :

- **l'identification** des ressources : chaque ressource est identifiée unitairement ;
- **la manipulation** des ressources : chaque manipulation a une représentation définie ;
- un **message auto-descriptif** : les messages expliquent leur nature.
- **Hypermédia** comme moteur d'état de l'application : chaque accès aux états suivants de l'application est décrit dans le message courant.



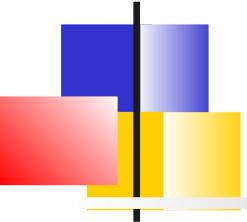
Representational State Transfer (3/3)

- **Un système par couche** : Un client ne peut pas savoir si il est connecté directement au serveur final ou un serveur intermédiaire. Les serveurs intermédiaires (proxy, gateway) peuvent améliorer la scalabilité (répartition de charge, cache partagé) ou renforcer la sécurité.
- **Code-On-Demand** (facultatif) : la possibilité pour les clients d'exécuter des scripts obtenus depuis le serveur.
 - Cela permet d'éviter que le traitement ne se fassent que du côté serveur et permet donc de faire évoluer les fonctionnalités du client au cours du temps.
 - En revanche cela réduit la visibilité de l'organisation des ressources. Un état devient dépendant du client et non plus du serveur ce qui contredit la règle 2. Il faut donc être prudent en utilisant cette contrainte.



RestFul API

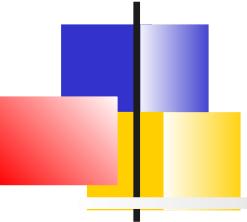
URI	GET	PUT	POST	DELETE
Collection, http://api.example.com/resources/	Liste les URI et autres détails des membres de la collection	Remplace la collection avec une autre collection	Crée une nouvelle entrée dans la collection. La nouvelle URI créé automatiquement et retournée par l'opération.	Supprime 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



Format JSON

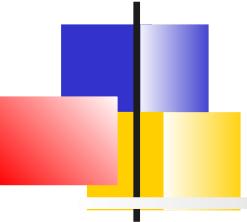
JavaScript Object Notation, est un format de données textuelles dérivé de la notation des objets du langage JavaScript.

- Il permet de représenter de l'information structurée tout comme XML.
- Il est cependant moins verbeux et directement compris par JavaScript.



Exemple

```
{  
  "menu": {  
    "id": "file",  
    "value": "File",  
    "popup": {  
      "menuitem": [  
        { "value": "New", "onclick": "CreateNewDoc()"},  
        { "value": "Open", "onclick": "OpenDoc()"},  
        { "value": "Close", "onclick": "CloseDoc()"}  
      ]  
    }  
  }  
}
```

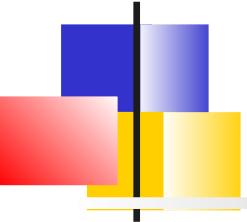


Sérialisation JSON

Un des principales problématiques des interfaces REST et la conversion des objets du domaine Java 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

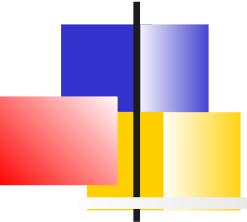


API Jackson

Jackson propose 3 principales API :

- Une API de streaming capable de lire ou écrire du contenu JSON sur un modèle évènementiel (analogue au Stax Parser de XMLà)
- Une API basée sur un modèle d'arbre. Un contenu JSON est transformé ou produit à partir d'une représentation mémoire en arbre (analogue au parser DOM)
- Du Data Binding permettant de convertir des POJO via ses accesseurs ou via des annotations (analogue à JAXB)

Les sérialisations/désérialisations sont effectuées généralement par des **ObjectMapper**



Annotations Jackson

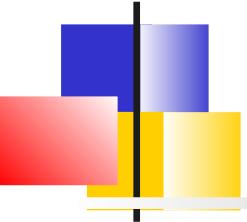
@JsonProperty, @JsonGetter, @JsonSetter,
@JsonAnyGetter, @JsonAnySetter, @JsonIgnore,
@JsonIgnorePoperty, @JsonIgnoreType : Permettant de définir les propriétés JSON

@JsonRootName : Arbre JSON

@JsonSerialize, @JsonDeserialize : Indique des dé/sérialiseurs spécialisés

@JsonManagedReference, @JsonBackReference,
@JsonIdentityInfo : Gestion des relations bidirectionnelles

....



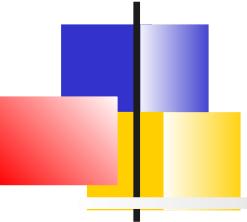
Librairies Framework

Java : JAX-RS (Jersey), Apache CXF,
Spring HATEOAS

.NET : ASP.NET Web API

Php : Httpful, Extension Curl

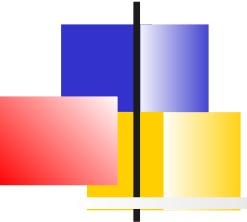
Javascript : XMLHttpRequest, jQuery,
AngularJS, Ember.js, Backbone.js, ...



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();
    }
}
```



Annotations JAX-RS

@Path : Indique l'URI d'accès

@GET, @POST, @PUT, @DELETE, @HEAD : Désigne une méthode traitant une action HTTP

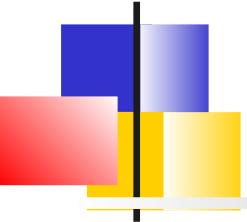
@PathParam, @QueryParam : Permettent d'extraire un paramètre de la requête HTTP

@Consumes : Spécifie le MIME-type utilisé par le client

@Produces : Spécifie le MIME-type fourni par le service

@Provider : Permet de spécifier des classes

MessageBodyReader et *MessageBodyWriter* utilisées pour les opérations de marshalling

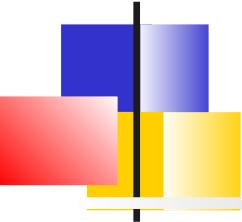


Swagger

Swagger est une solution permettant d'obtenir à partir d'une API RESTful :

- Une documentation interactive
- La génération d'un SDK client
- La découverte automatique des services

Cela remplit un peu les mêmes fonctionnalités qu'un WSDL.



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

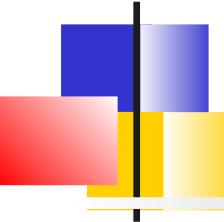
DELETE /user/{username} Delete user

GET /user/{username} Get user by user name

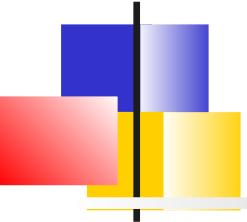
PUT /user/{username} Updated user

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

VALID { }



Architecture SOA, ESB et gestion des flux



Définition d'une architecture SOA

Une architecture orientée services est une architecture logicielle s'appuyant sur un ensemble de services de granularité moyenne.

L'objectif est de décomposer une fonctionnalité en un ensemble de fonctions basiques, appelées services, fournies par des composants et de décrire finement le schéma d'interaction entre ces services.

Le service est le composant de plus bas niveau offert par une architecture SOA. Il expose une **interface à couplage faible** offrant un accès à une fonctionnalité **métier** ou **technique**.

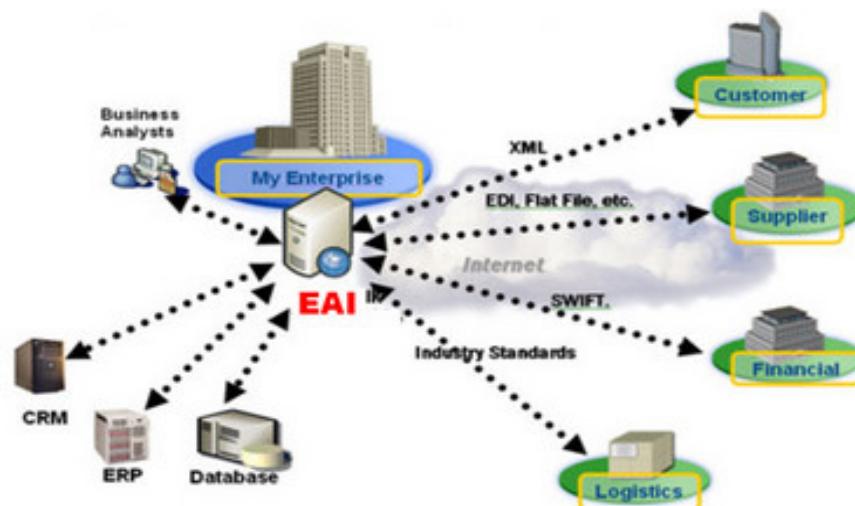
SOA de surface

Le SOA de surface s'appuie sur des applications existantes pour les faire communiquer entre elles.

Le SOA de surface n'influence pas la conception des applications en silo et peut introduire une complexité technique forte (ESB, EAI...)

Le SOA de surface peut être vu comme alternative synchrone à des échanges asynchrone par batchs.

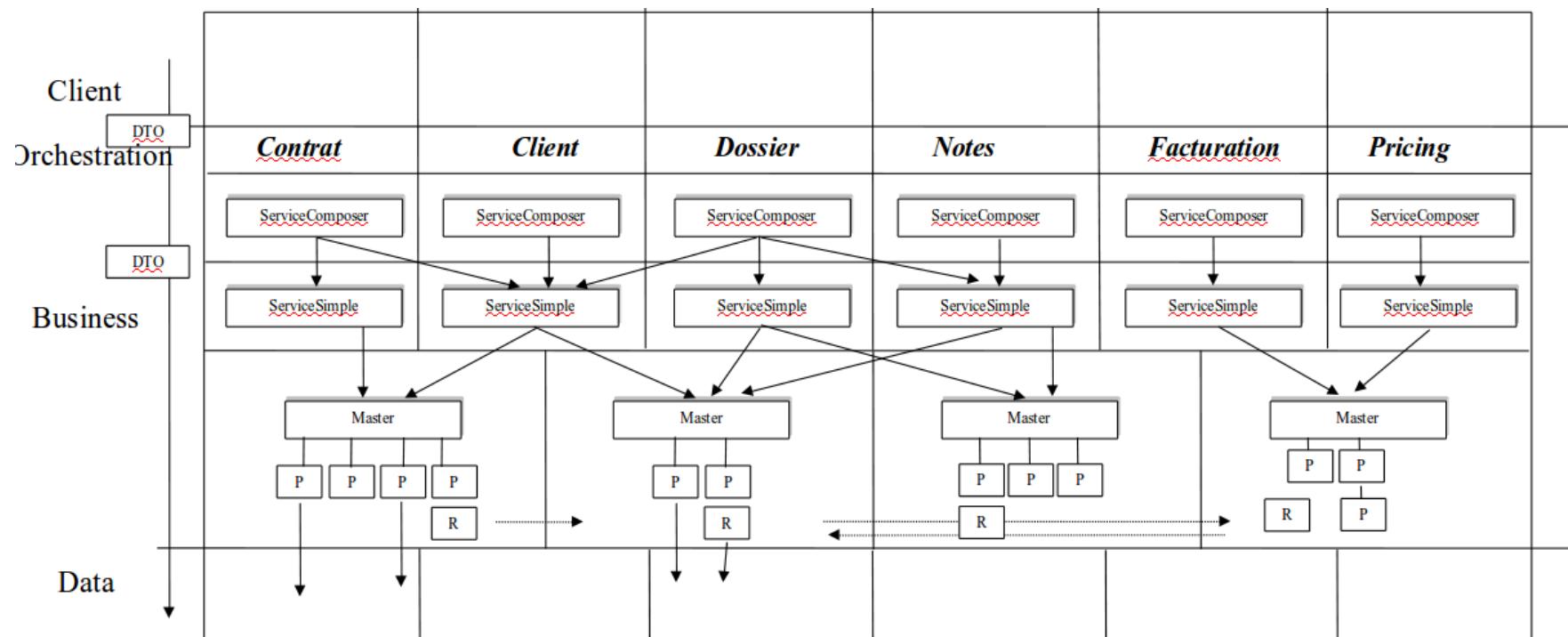
Le SOA de surface se base sur des services réutilisables a posteriori

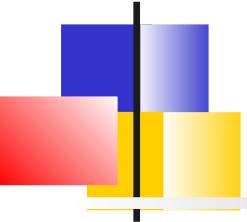


SOA de profondeur

Le SOA de profondeur peut être vue comme une architecture de refonte d'un SI.

Il organise les projets en s'appuyant sur des concepts transverses. Il doit permettre la réutilisation et l'orchestration de services.





Enjeux pour le SI

Répondre plus rapidement aux besoins du métier

Maîtriser la complexité croissante du SI

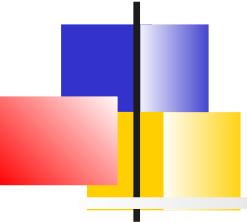
Mutualiser les développements

Pérenniser ses développements

Réduire les coûts de développement

Augmenter ses ROI

Standardiser ses processus



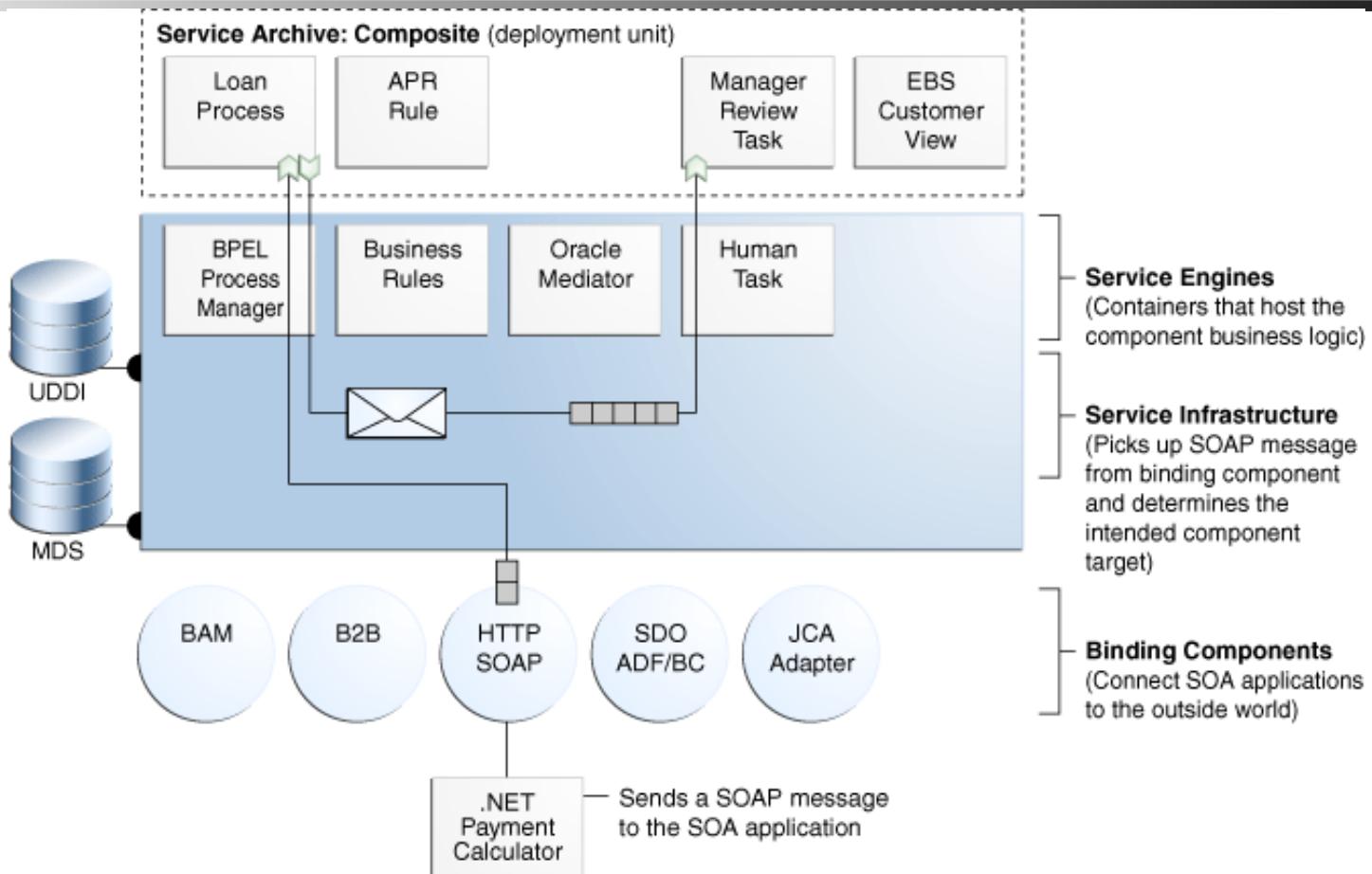
Framework SCA

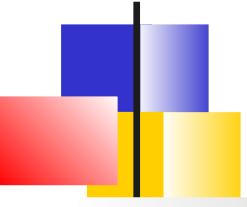
SCA (Service Composition Architecture) permet de :

- Créer des composants services écrits avec différentes technologies (Java, BPEL, C++, XSLT)
- Assembler les composants dans une application composite. Les composants forment les briques de base pour construire une application. Ils sont faiblement couplés et peuvent s'invoquer en synchrone ou asynchrone

Exemple : Oracle SOA Suite (s'appuyant sur les services web). L'application produite est un fichier *composite.xml* qui compose des services déployés individuellement

Exemple Oracle SOA





ESB

L'**ESB (Enterprise Service Bus)** est une nouvelle génération d'EAI construite sur des standards comme XML, Java Message Service (JMS) ou encore les services web.

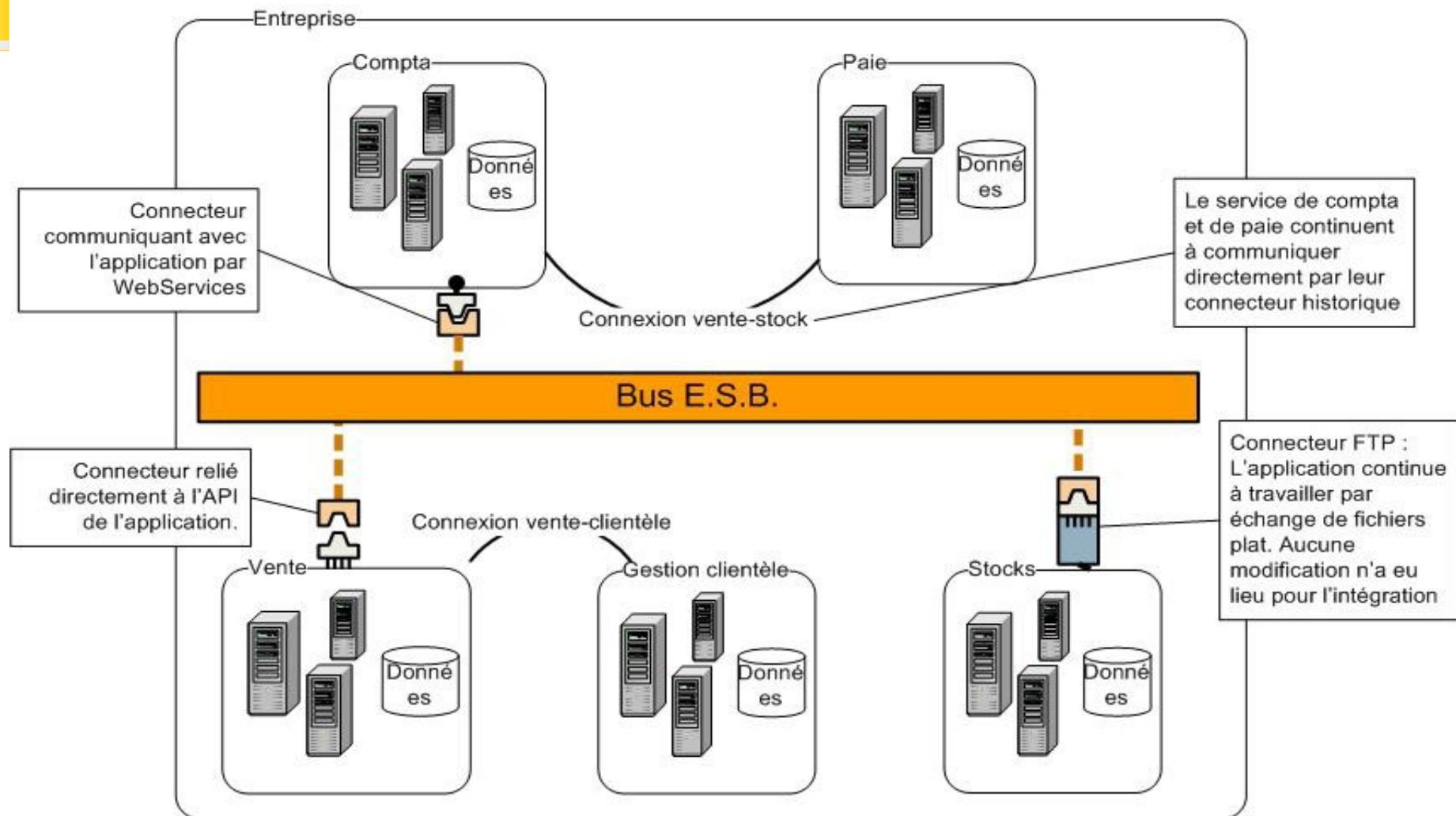
La différence majeure avec l'EAI est l'architecture **distribuée** des ESB grâce à l'utilisation de conteneurs de service qui collaborent.

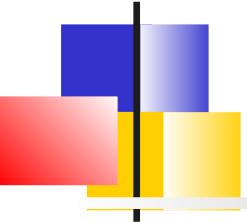
Ces « mini-serveurs » contiennent la logique d'intégration et peuvent être déposés n'importe où sur le réseau.

Ils proposent des interfaces « haut-niveau » permettant de connecter, orchestrer les services entre eux

Exemples : Mule, Jboss Fuse, Talend ESB, WSO2 ESB, IBM, Oracle, BizTalk Microsoft

Le modèle ESB





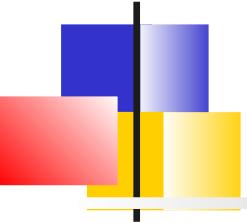
Message brokers

Les architectures ESB permettent s'appuie généralement sur des messages brokers

Cette technologie est souvent utilisée seule pour l'intégration de services à faible couplage.

Beaucoup de solutions sont disponibles actuellement, certaines respectent le standard JMS.

Citons : ActiveMQ, RabbitMQ, Redis, ...

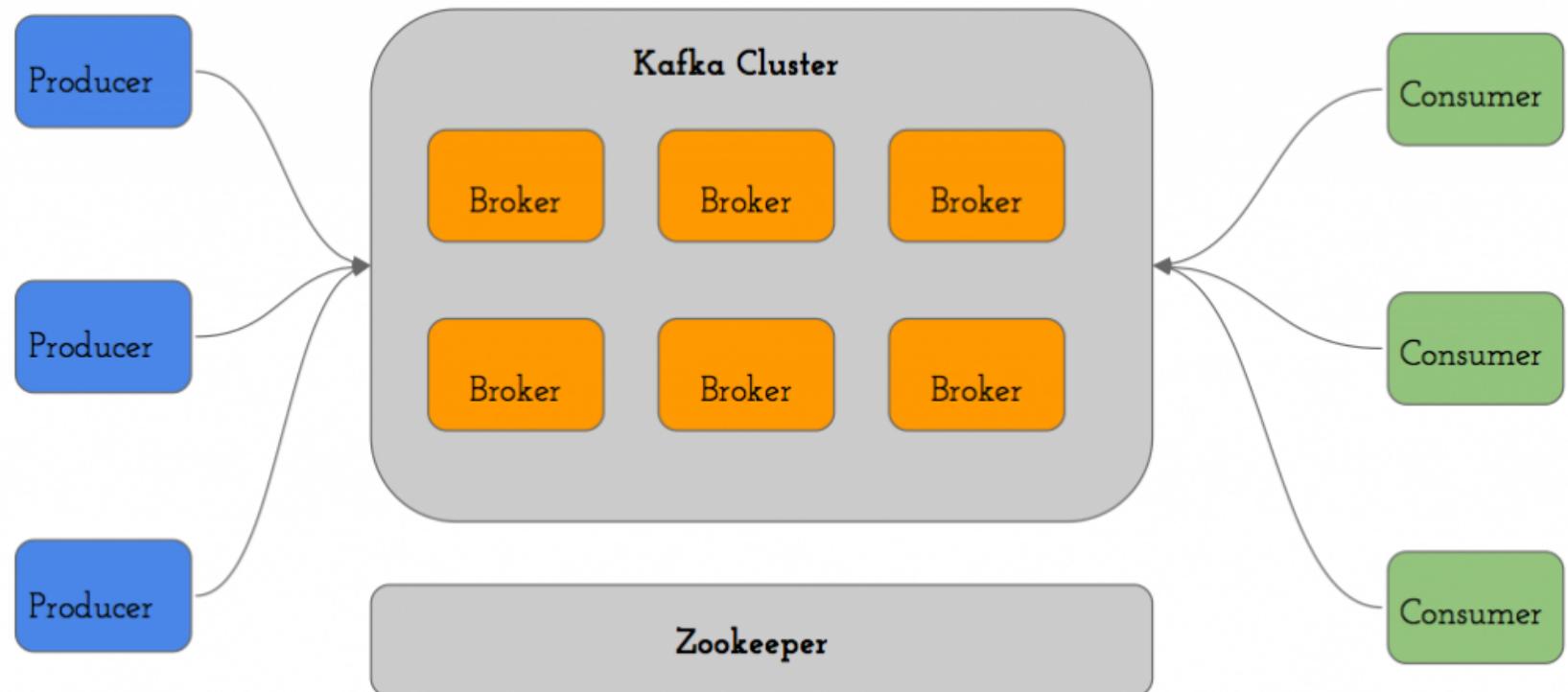


Application au BigData et gestion de flux

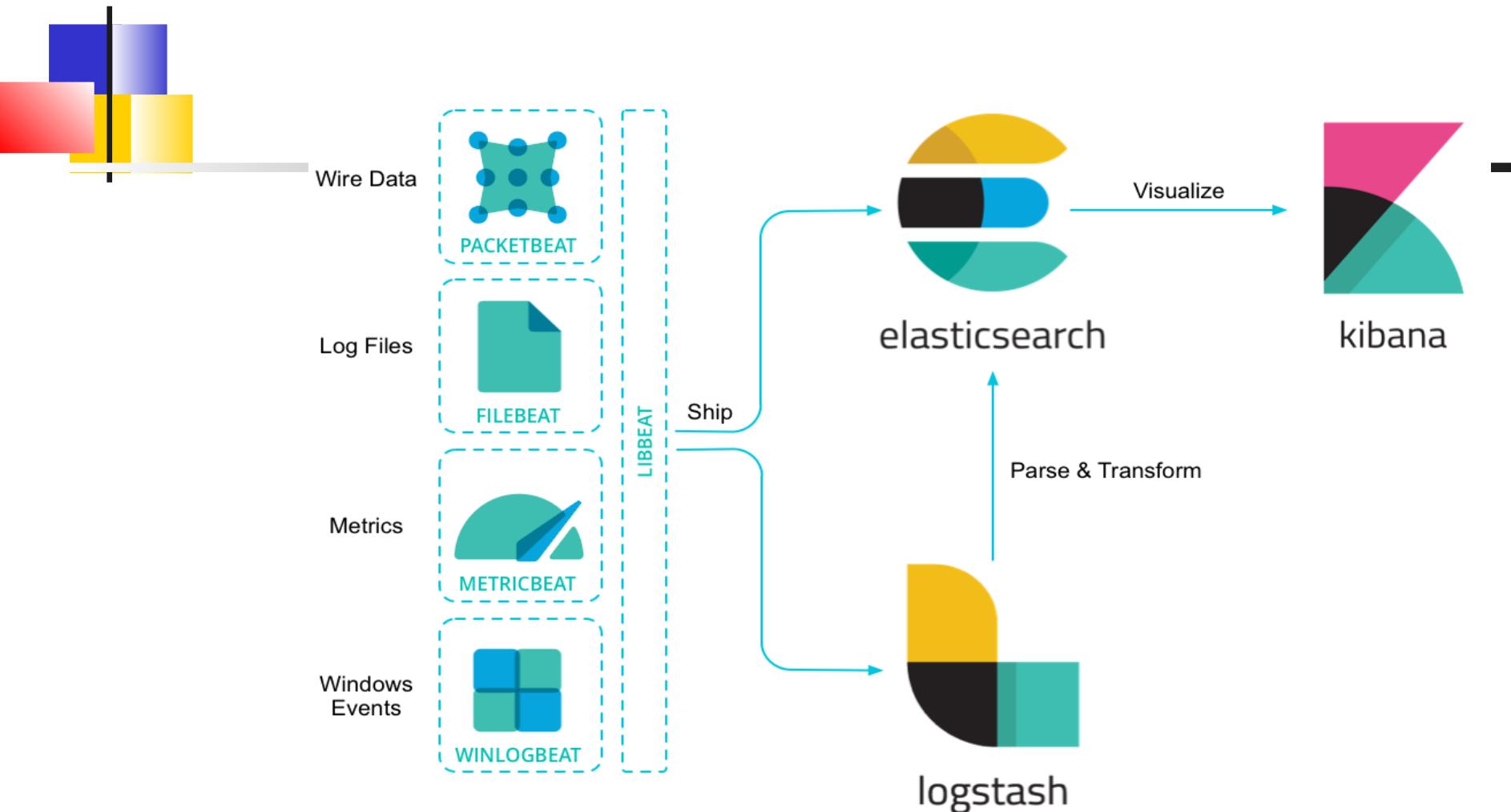
L'apparition du BigData nécessite des solutions de + en + performante visant le temps réel. Ces solutions utilise nativement le clustering

- Le projet **Kafka** vise à devenir la plateforme centralisée de stockage et d'échange de toutes les toutes les données émises par une entreprise en temps réel.
Il est basé sur le modèle Pub&Sub des messages brokers
- La solution **ElasticStack** propose une suite permettant d'indexer d'énormes volumes de données et en particulier toutes les données d'une entreprise

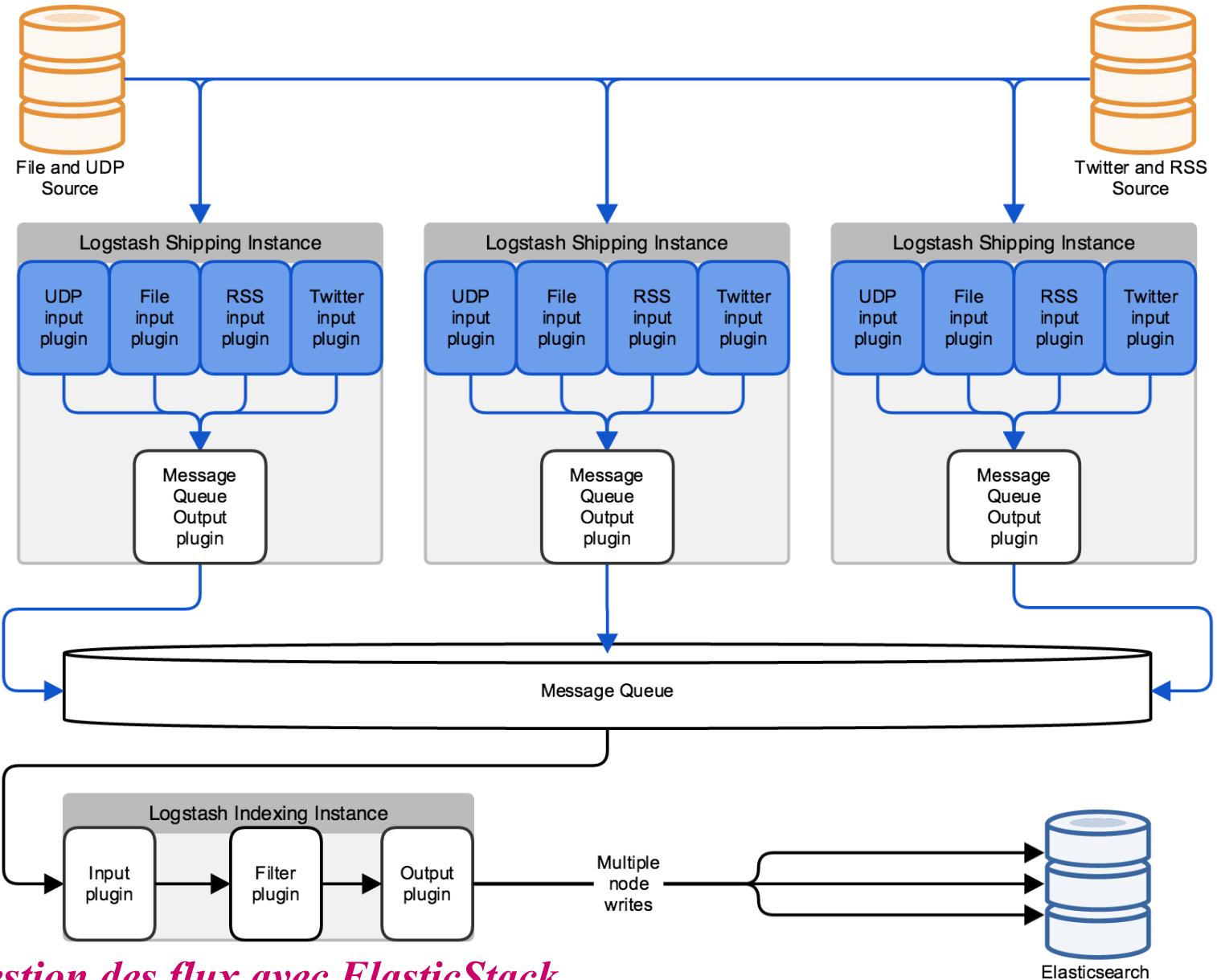
Kafka



Architecture ELK

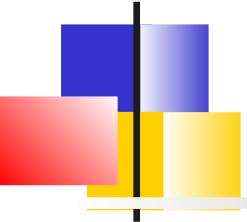


Architecture d'ingestion clusterisée





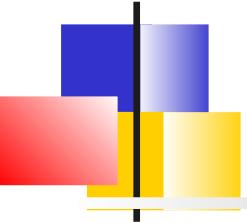
Architecture micro-services



Introduction

Le terme « **micro-services** » décrit un nouveau pattern de développement visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel

Les méthodes agiles, la culture *DevOps*, le *PaaS*, les containers d'application et les environnement d'injection de dépendances permettent d'envisager de construire de grand systèmes orientés services de façon modulaire



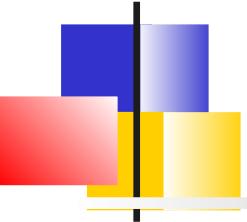
Architecture

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

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

Le but étant de livrer et maintenir des systèmes complexes avec la rapidité et la qualité demandées par le business digital actuel

On l'appelle également *SOA 2.0* ou *SOA For Hipsters*



Caractéristiques

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

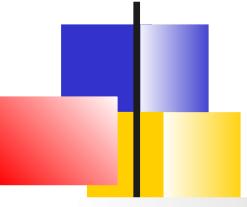
Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

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),

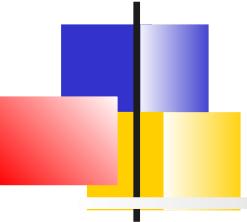
Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes

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...)

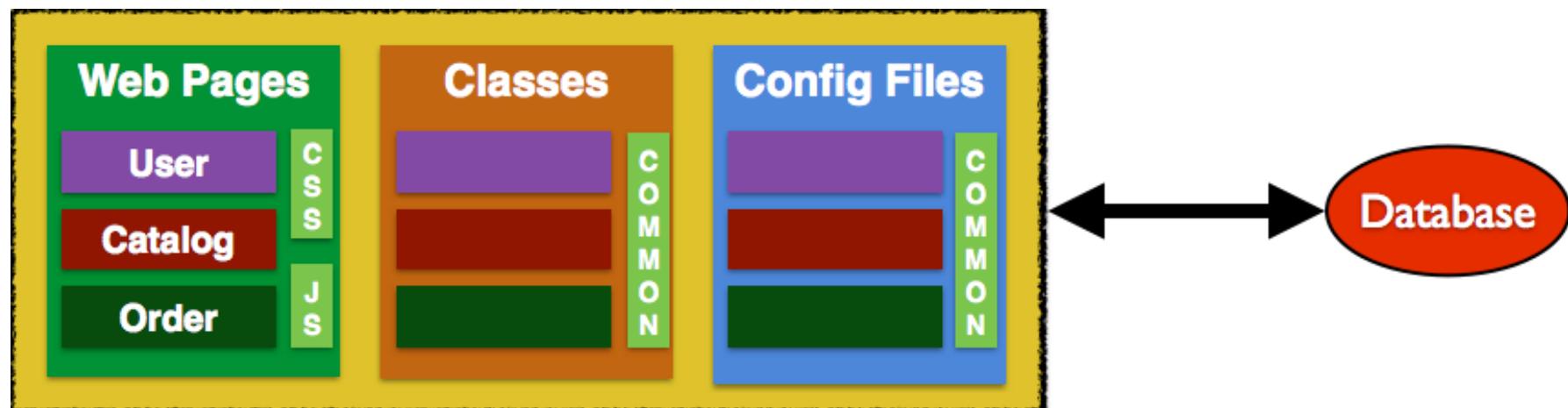
Découverte automatique : Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certains métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

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

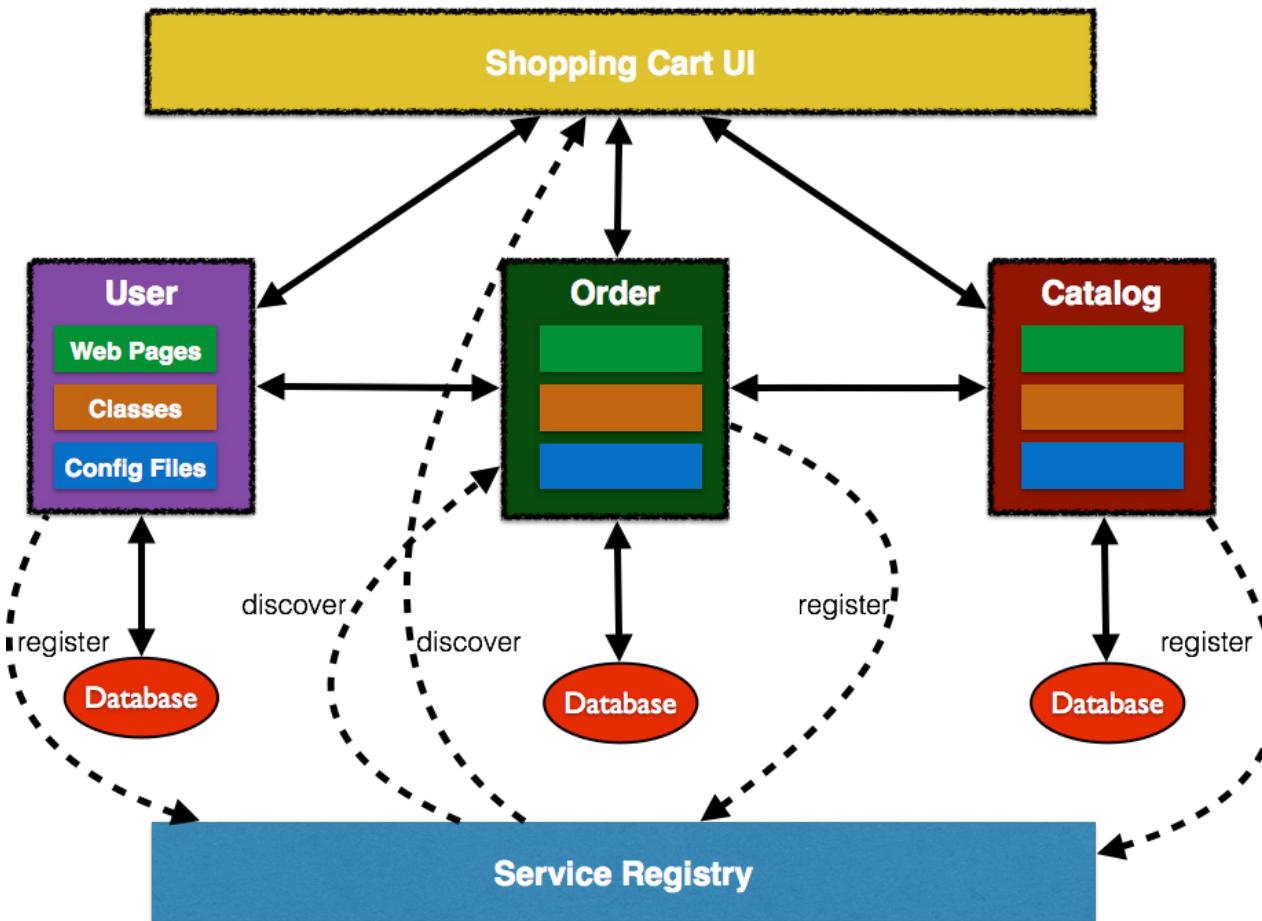
Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

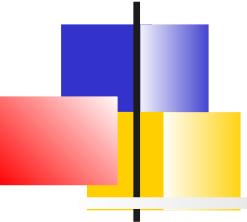
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.

Architecture monolithique



Version micro-service





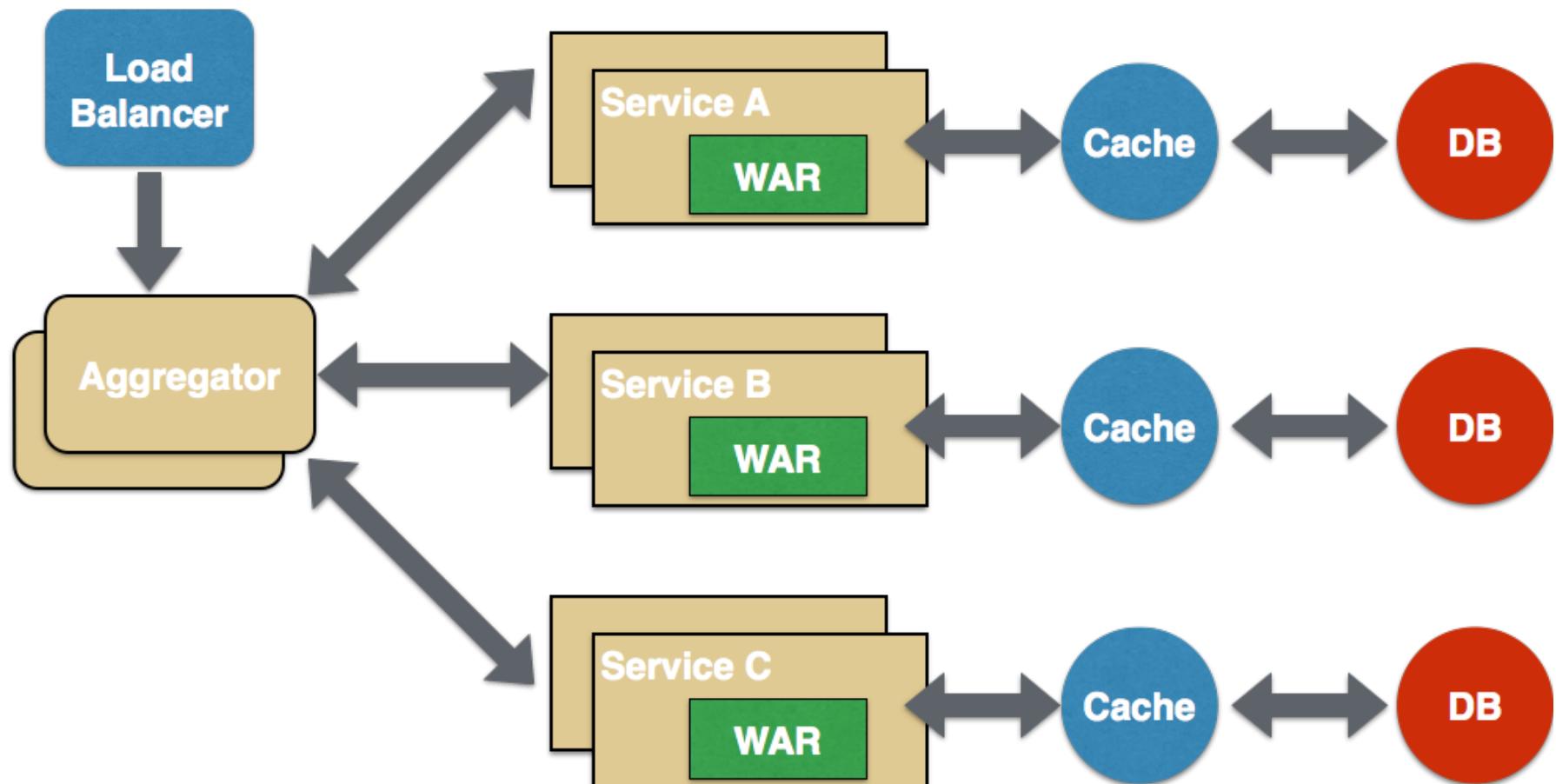
Patterns

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

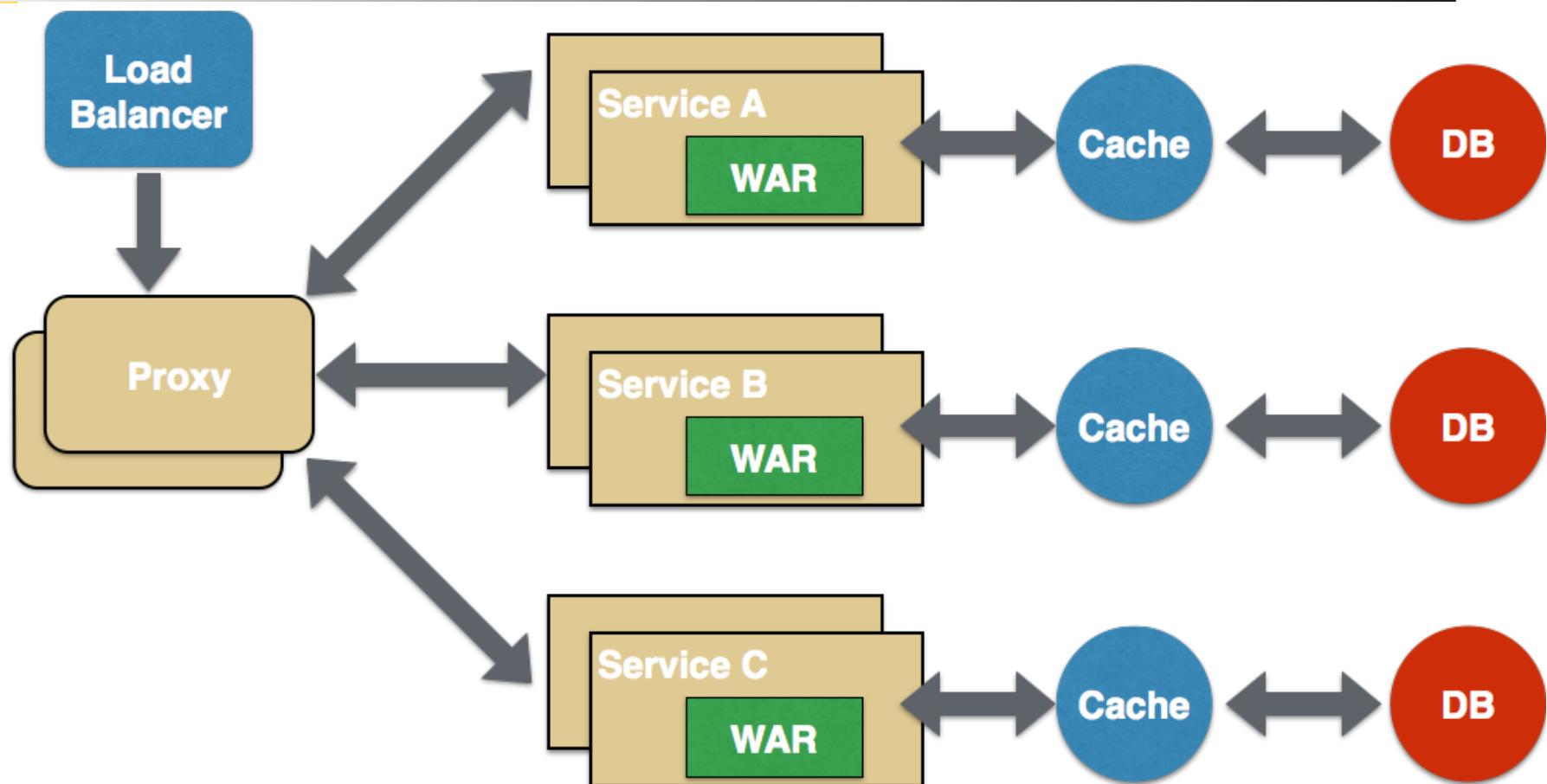
Comme design patterns communs, citons :

- **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

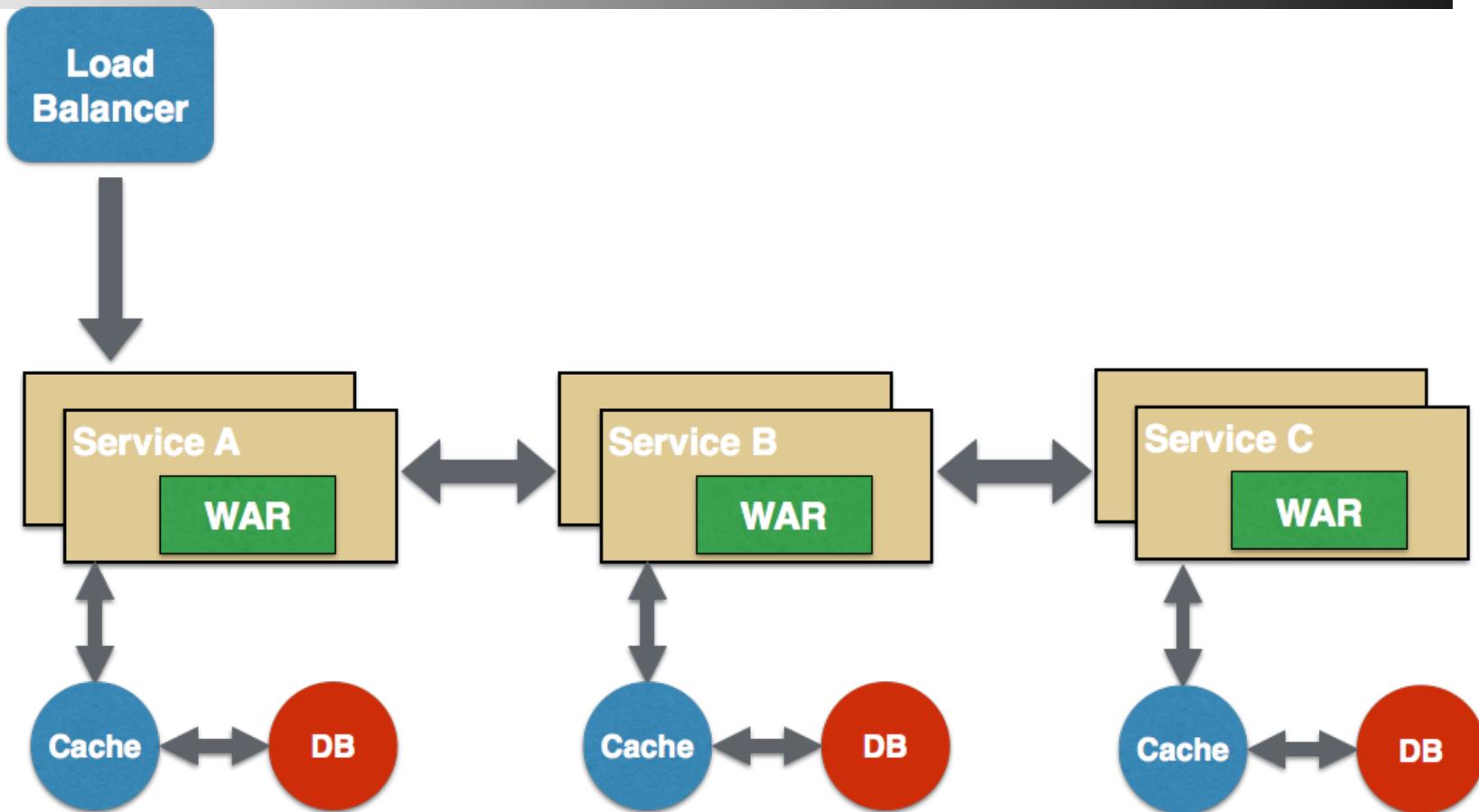
Agrégateur



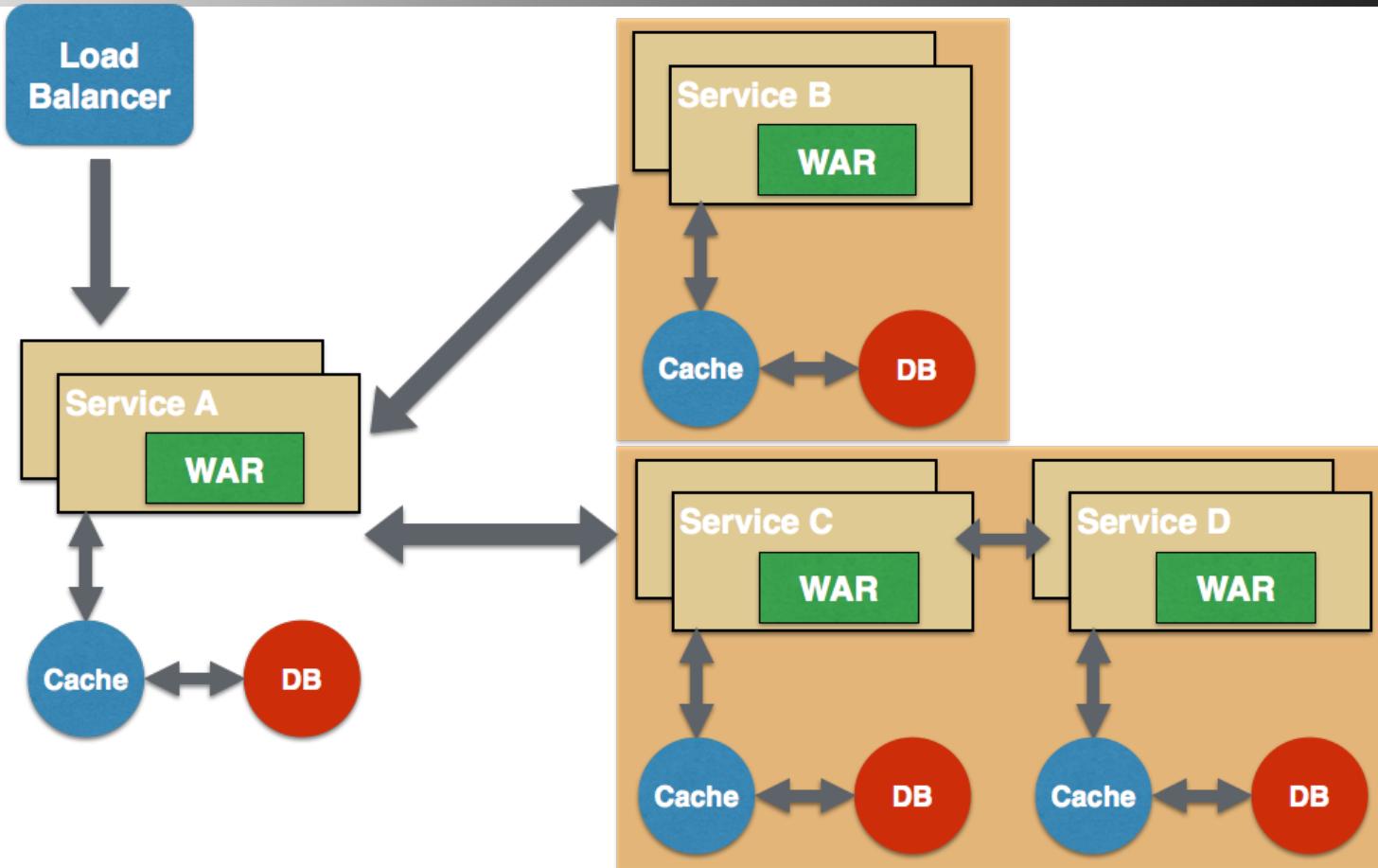
Proxy

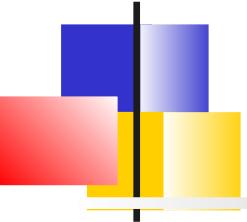


Chaîne



Branche



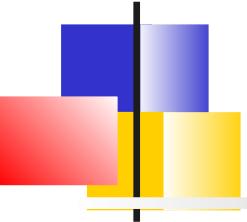


Clients REST

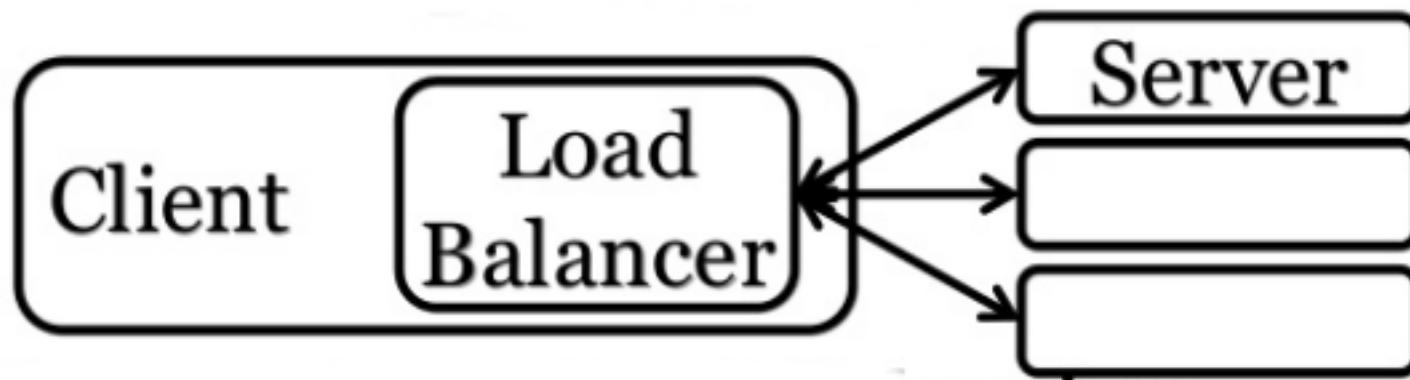
Dans cette architecture un micro-service est souvent le client REST d'un autre micro-service

Les clients doivent avoir des capacités :

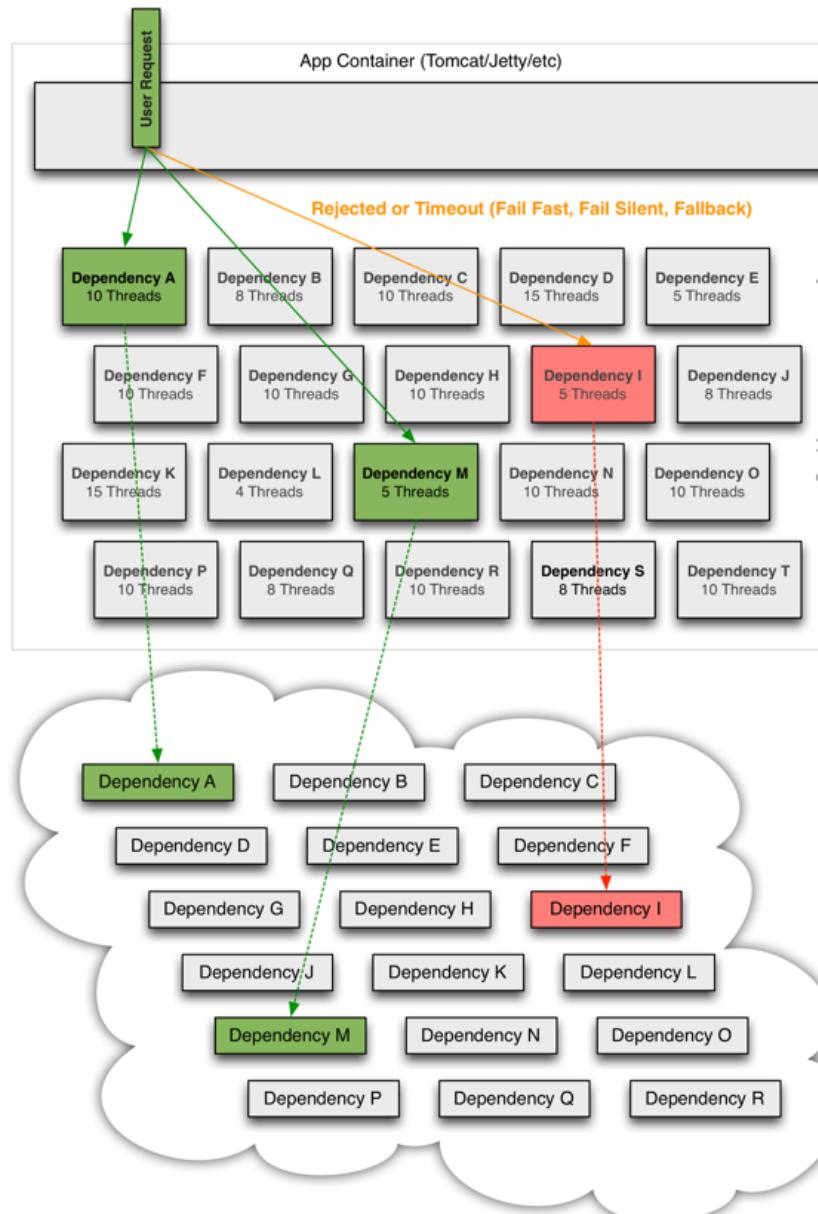
- De répartition dynamique de charge
- De résilience (code alternatif en cas de disfonctionnement du serveur)

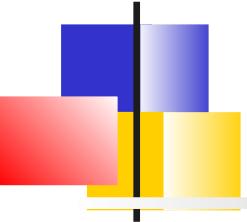


Répartition dynamique de charge



Résilience avec les pools de threads Hystrix

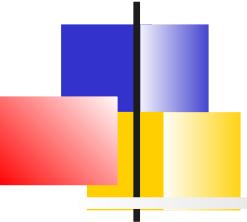




Micro-services techniques

Les architectures micro-services basées sur une distribution massive nécessitent des micro-services d'infrastructure :

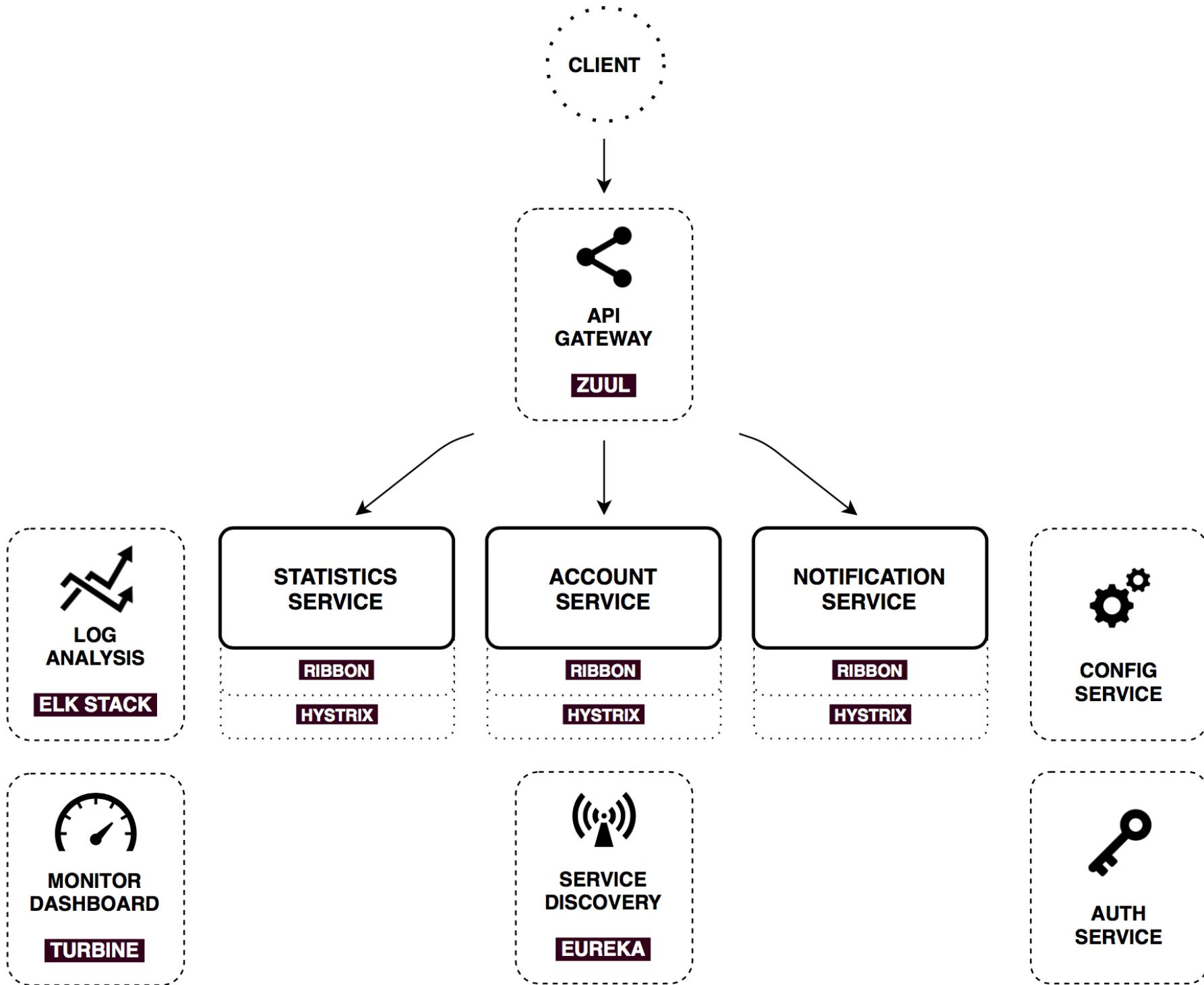
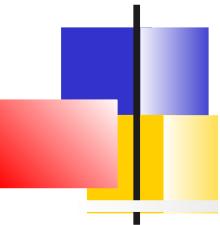
- Service **d'annuaire** permettant de localiser un micro-service disponible
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Services **d'authentification** offrant une fonctionnalité de SSO parmi l'ensemble des micro-services
- Service de **monitoring** surveillant la disponibilité, centralisant les fichiers journaux
- Service de répartition de charge, de gestion d'appartenance au cluster, d'élection de maître, ...
- Service de session applicative, horloge synchronisée,

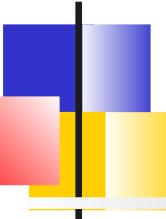


Spring Boot / Spring Cloud

Sprint Cloud basé sur Spring Boot fournit un framework de développement de micro-services qui :

- Apporte tous les micro-services techniques nécessaires
- Bénéficie de l'environnement Spring Boot et de l'écosystème Spring (Testabilité, Spring MVC / REST, Spring Data (SQL ou NoSQL), ...)





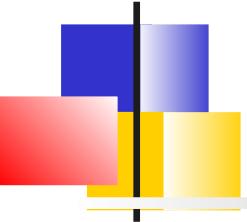
Infrastructure de déploiement

La nécessité que chaque micro-service soit scalable et que l'on puisse facilement créer de nouvelles instances de n'importe quel service détermine l'infrastructure de déploiement :

- Virtualisation + outils de gestion de conf
(Puppet, Chef, Ansible)
- Images d'une solution de containerisation
(Docker)
- Offre Paas (AWS, Google, ...)



Frameworks clients



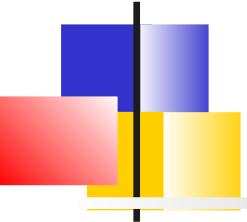
Introduction

De nombreux frameworks sont apparus ces derniers temps sur la couche cliente.

Ils sont pour la plupart basés sur Javascript.

Citons :

- *jQuery*
- *React JS*
- *Angular 1 et 2*
- *Ext JS*



jQuery

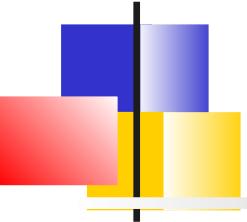
Première version en 2006

A fortement contribué à l'engouement récent pour JavaScript

Est présent pratiquement dans tous les autres framework

La bibliothèque de bas niveau apportant les fonctionnalités suivantes :

- Parcours et modification du DOM (y compris le support des sélecteurs CSS 1 à 3 et un support basique de XPath) ;
- Gestion des événements ;
- Effets visuels et animations ;
- Manipulations des CSS (ajout/suppression des classes, d'attributs...) ;
- Requêtes Ajax ;
- Utilitaires (version du navigateur web...).
- Plugins : Réutilisation de composants jQuery

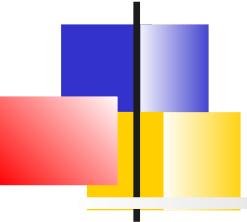


Exemples

```
# Manipulation du DOM
$( "button.continue" ).html( "Next Step..." )

# Gestion d'évènements
var hiddenBox = $( "#banner-message" );
$( "#button-container button" ).on( "click", function( event ) {
    hiddenBox.show();
});

# Ajax
$.ajax({
    url: "/api/getWeather",
    data: {
        zipcode: 97201
    },
    success: function( result ) {
        $( "#weather-temp" ).html( "<strong>" + result + "</strong> degrees" );
    }
});
```



Critères de choix des frameworks

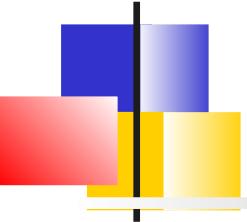
Pour les développeurs C# ou Java , Ext ou Angular permettent de travailler avec JavaScript comme avec les langages Objets.

Quantité de ressources sur le web, qualité des documentation : Ext JS et Angular 2 proposent sûrement le plus de choses

Cadre de développement structurant : Angular 2 est sûrement le framework qui propose un cadre structurant dont les développeurs ne peuvent pas sortir. Cela produit du code plus lisible et maintenable

Respect des standards : En particulier , HTML/CSS et ECMAScript. Ext JS ne repose pas sur HTML/CSS mais sur un système propriétaire de génération de HTML, Ext JS ne repose pas sur les outils de build habituels (npm, ...),

Testabilité : Angular et React ont beaucoup de support pour le test unitaire . Mais React ne s'appuie pas sur Karma l'outil le plus répandu

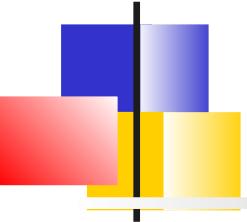


Angular 1

Développement en Javascript

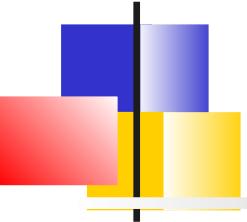
Concepts de base :

- MVC :
 - Contrôleurs en Javascript
 - Modèle : Données JSON
 - Vue : DOM HTML
- Data bindings : Les mises à jour sur le modèle Javascript se répercutent automatiquement sur les balises HTML
- Scopes : Pour traiter un use case, des variables sont positionnées dans des contextes
- Injection de dépendances, les services « backend » nécessaires aux contrôleurs sont injectés
- Testabilité
- Directive : Encapsulation et réutilisation de code



Exemple Vue

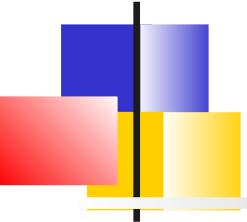
```
<html ng-app="phonecatApp">  
  <head>  
    ...  
    <script src="bower_components/angular/angular.js"></script>  
    <script src="app.js"></script>  
  </head>  
  <body ng-controller="PhoneListController">  
  
    <ul>  
      <li ng-repeat="phone in phones">  
        <span>{{phone.name}}</span>  
        <p>{{phone.snippet}}</p>  
      </li>  
    </ul>  
  
  </body>  
</html>
```



Exemple Service REST

angular.

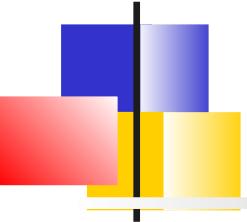
```
module('core.phone').  
factory('Phone', ['$resource',  
function($resource) {  
    return $resource('phones/:phoneId.json', {}, {  
        query: {  
            method: 'GET',  
            params: {phoneId: 'phones'},  
            isArray: true  
        }  
    });  
}  
]);
```



Exemple Contrôleur

```
// Injection du service Phone

controller: ['Phone',
    function PhoneListController(Phone) {
        this.phones = Phone.query();
        this.orderProp = 'age';
    }
]
```



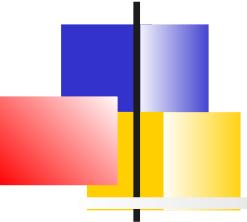
Angular 2

Développement en TypeScript,
JavaScript ou Dart

Cible Web et Mobile

Concepts cœur :

- Les mêmes qu'Angular 1 en mieux
(Binding, Dependency Injection)
- MVC devient Web Components :
Composants réutilisables, annotés

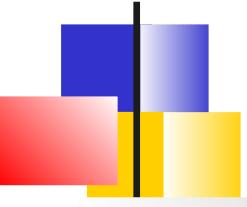


Exemple Component <my-app>

```
<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';
}
```



Service

```
import { Injectable }      from '@angular/core';
import { Headers, Http } from '@angular/http';
import 'rxjs/add/operator/toPromise';
import { Hero } from './hero';

@Injectable()
export class HeroService {

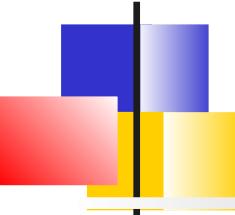
  private headers = new Headers({'Content-Type': 'application/json'});
  private heroesUrl = 'app/heroes'; // URL to web api

  constructor(private http: Http) { }

  getHeroes(): Promise<Hero[]> {
    return this.http.get(this.heroesUrl)
      .toPromise()
      .then(response => response.json().data as Hero[])
      .catch(this.handleError);
  }

  getHero(id: number): Promise<Hero> {
    return this.getHeroes()
      .then(heroes => heroes.find(hero => hero.id === id));
  }

  delete(id: number): Promise<void> {
    ...
  }
}
```



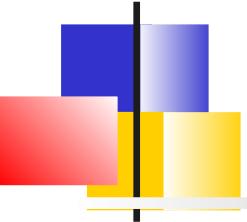
Web Component

```
import { Component, OnInit } from '@angular/core';
import { Router }           from '@angular/router';
import { Hero }             from './hero';
import { HeroService }      from './hero.service';

@Component({
  selector: 'my-heroes',
  templateUrl: 'app/heroes.component.html',
  styleUrls:  ['app/heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  // Injection de service lors du constructeur
  constructor(
    private heroService: HeroService,
    private router: Router) { }

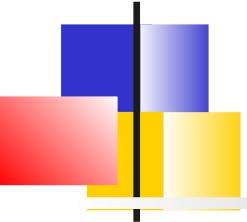
  getHeroes(): void {
    this.heroService
      .getHeroes()
      .then(heroes => this.heroes = heroes);
  }

  add(name: string): void {
    name = name.trim();
    ...
  }
}
```



Usines à logiciel, Intégration continue, DevOps

- L'intégration continue
- Gestion de l'infrastructure
- Les SCMs et workflows de collaboration
- Les outils de build
- Les tests
- Le serveur d'intégration continue



Avant l'intégration continue

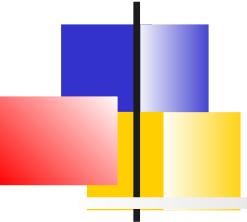
Le cycle de développement classique intégrait une **phase d'intégration** avant de produire une release :

- intégrer les développements des différentes équipes sur une plate forme identique à la production.

Différents types de problèmes pouvaient alors survenir nécessitant quelquefois des réécritures de lignes de code produites il y a longtemps

La phase d'intégration pouvait alors introduire de long délais de livraison

=> L'intégration continue a pour but de lisser l'intégration **pendant** tout le cycle de développement

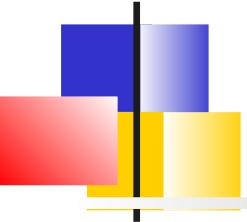


Intégration continue

L'intégration continue dans sa forme la plus simple consiste en un outil surveillant les changements dans le **Source Control Management (SCM)**

Lorsqu'un changement est détecté, l'outil construit et teste automatiquement l'application

Si ce traitement échoue, l'outil notifie immédiatement les développeurs afin qu'ils **corrigeant le problème ASAP**



Outil de communication et de motivation

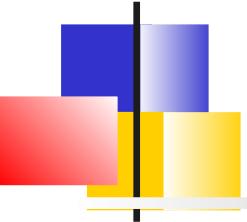
L'intégration continue permet également de donner **en temps réel** des métriques sur la santé du projet :

- Résultats des différents tests
- Qualité du code
- Couverture fonctionnelle et avancement du projet
- Documentation

L'accès constant à ces données permet de donner de la confiance dans la robustesse du code développé et de réduire les coûts de maintenance.

Les métriques sur la qualité du code sont alors visibles aussi bien par les fonctionnels que par les développeurs

=> Cette transparence motive les équipes pour produire un code de qualité



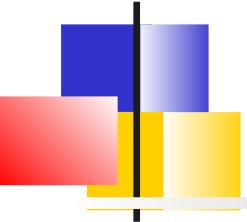
Collaboration avec les utilisateurs

Dans son essence, l'intégration continue a plusieurs objectifs :

- **réduire les risques** en fournissant plus rapidement un retour sur les derniers développements
- **faciliter la communication** entre développeurs et fonctionnels
- **encourager la résolution collaborative** des problèmes et l'amélioration des processus

Avec le déploiement automatique ou 1 click, l'intégration continue permet de **mettre à disposition des testeurs et utilisateurs finaux** les derniers développements plus rapidement et plus facilement

Combiné avec des tests d'acceptance automatiques, l'intégration continue devient alors un outil de communication donnant une vision claire de l'état courant des développements.



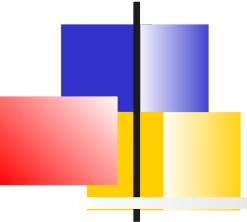
Déploiement et livraison continus

Le déploiement de tous les builds *réussis* directement en production est appelé **déploiement continu**. C'est le stade ultime de l'intégration continue.

- Cependant, les utilisateurs préfèrent un cycle de release géré plutôt que d'avoir leur application en continual changement.
- Les équipes commerciales ou marketing peuvent également jouer un rôle dans la livraison d'une nouvelle release

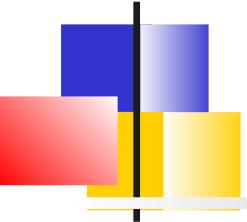
La **livraison continue** est une légère variante du déploiement continu qui prend en compte ces considérations

- Chaque build réussi est soumis aux experts métier qui décident si les changement sont livrés en production



Mentalité « CI »

- Les projets doivent avoir un process de build sûr, automatisé, reproductible sans intervention humaine
- Les problèmes concernant le build doivent être résolus dès qu'ils apparaissent.
- Le processus de déploiement doit être entièrement automatique
- Les tests ont une importance primordiale, l'équipe de développement doit se concentrer sur l'élaboration de tests pouvant donner une confiance dans l'application produite

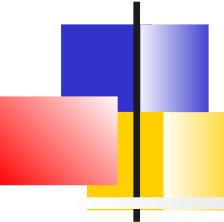


Phases de la mise en place

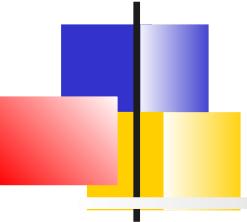
La mise en place de l'intégration continue ne se fait pas en 1 jour.

En général, cela passe par plusieurs phases qui chacune améliore l'infrastructure et modifie les pratiques des équipes de développement.

1. Pas de serveur de build
2. Serveur de build et Nightly Builds
3. Nightly Builds et test basiques automatisés
4. Obtention des métriques
5. Renforcement des tests
6. Tests d'acceptance et déploiement automatisé
7. Déploiement continu



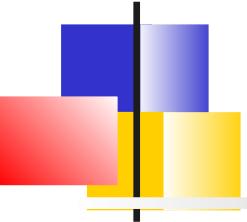
Virtualisation, Paas, containerisation L'exemple de Docker



Introduction

L'intégration continue nécessite de provisionner l'infrastructure pour les différents environnements, dès le début du projet

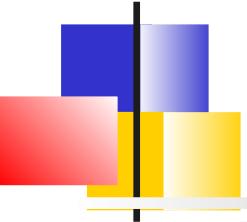
Cette difficulté est résolue par les avancées récentes dans les techniques de virtualisation et de containerisation.



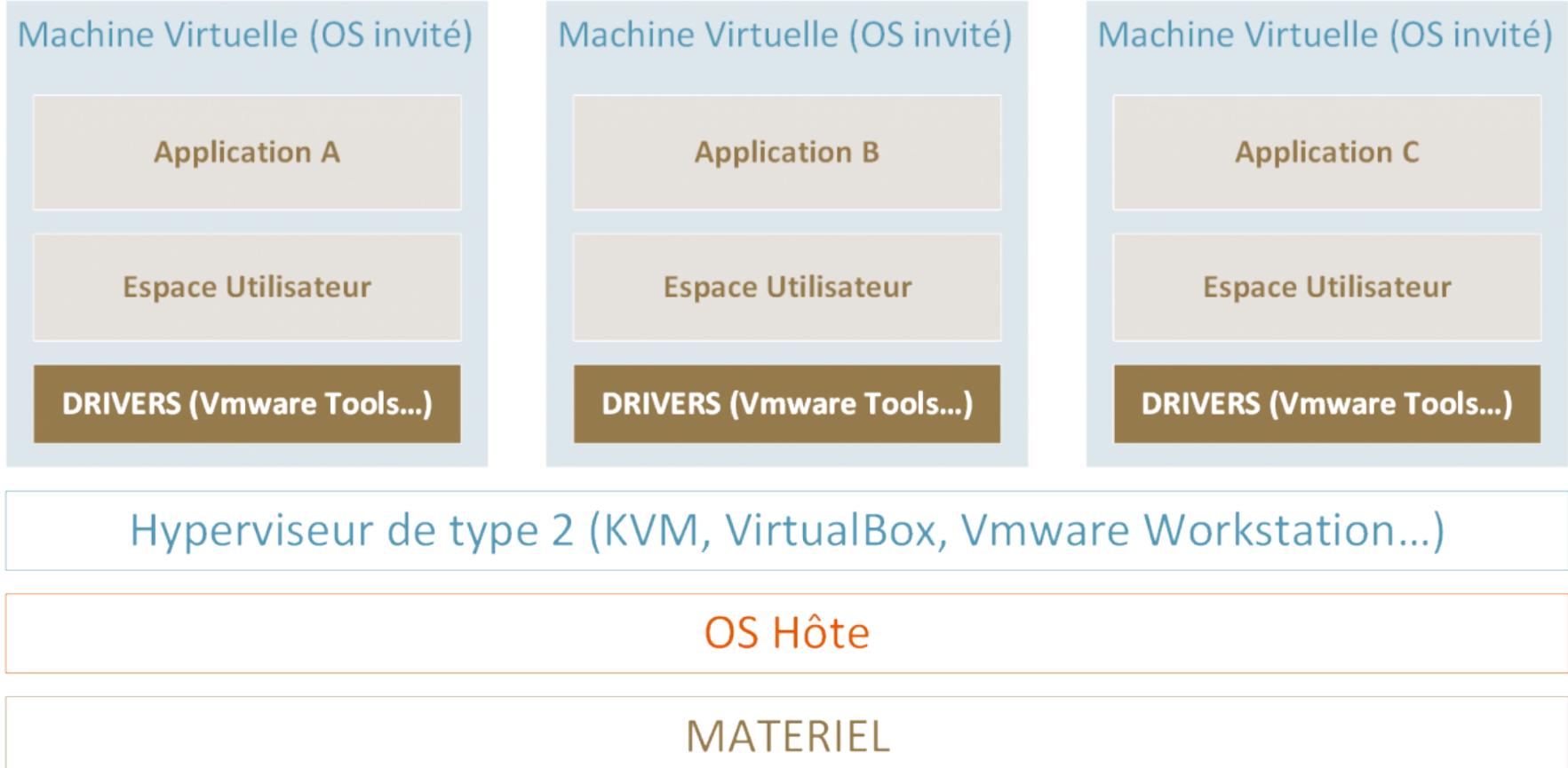
Offres Cloud

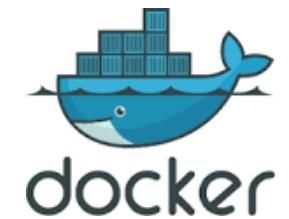
PaaS (*Platform As A service*) : Offre permettant de développer, exécuter et gérer les applications sans la complexité de mise en place de l'infrastructure

- Google App Engine
- OpenShift de RedHat
- Microsoft Azure
- Cloudbeees
- SalesForce.com

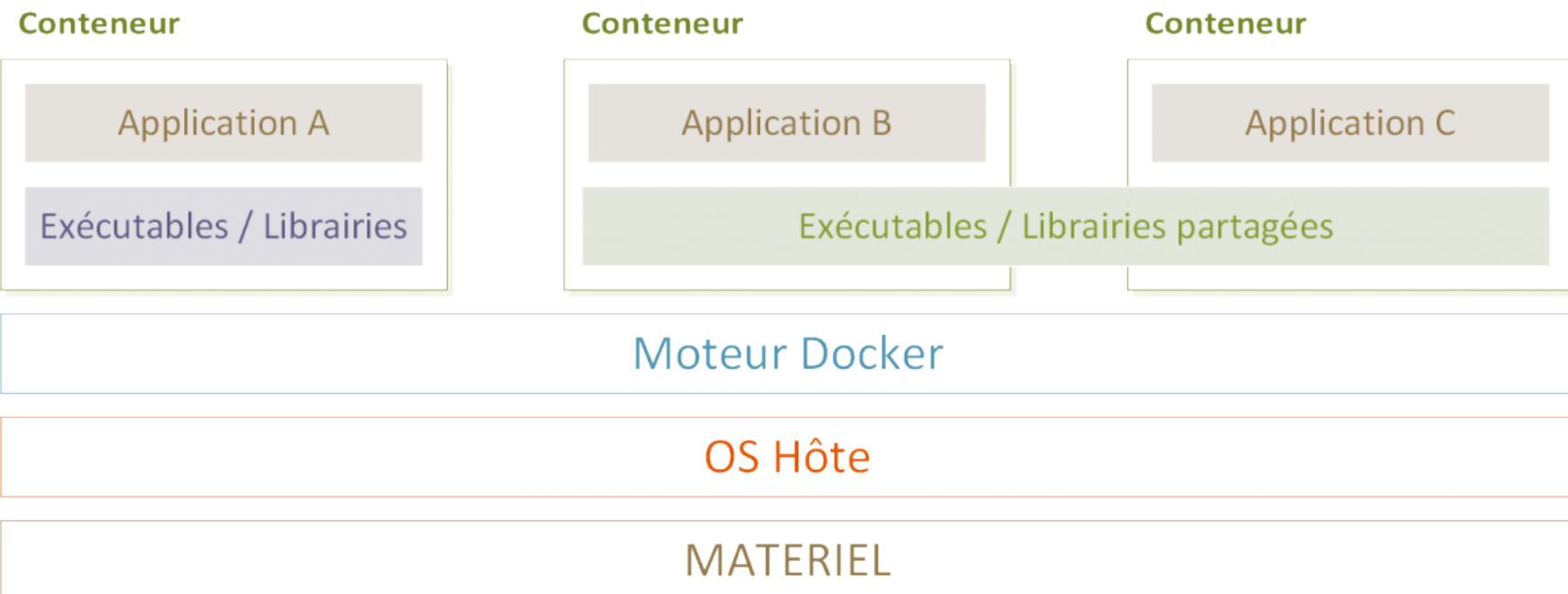


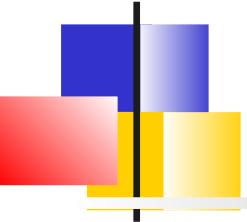
Virtualisation





Containerisation





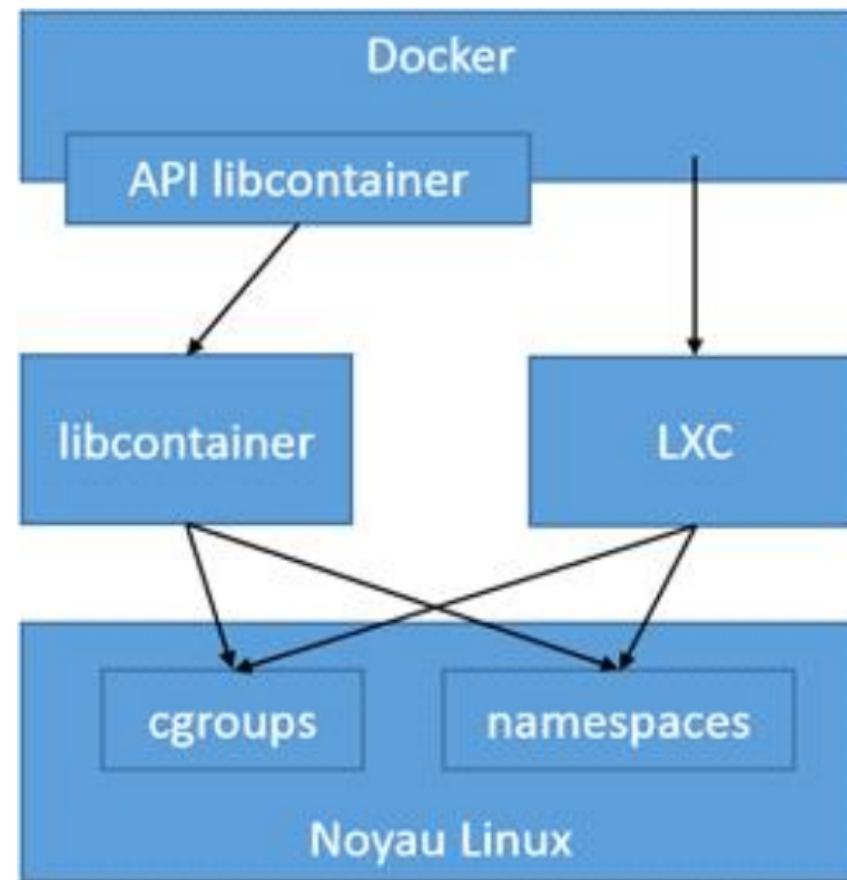
Technologies sous-jacentes

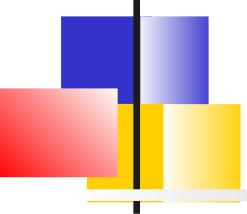
Les technologies sous-jacentes à la containerisation sont présents dans le noyau Linux :

- **Espace de noms** : Cela permet l'étanchéité entre containers. (Processus, Interfaces réseau, Point de montage fichier)
- **Control Groups (Cgroups)** : Permet de cloisonner l'accès aux ressources (Mémoire, CPU, Disque). Exemple : Allouer uniquement 15% de CPU à un processus

Anciennement, encapsulées dans *LXC (Linux Container)*, ces technologies sont dorénavant présentes dans une librairie offrant une API standard : ***libcontainer***

Implémentation Docker





Avantages de la containerisation

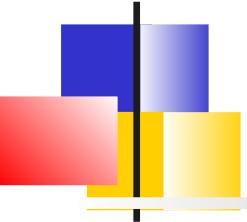
Rationalisation des ressources, à la différence de la virtualisation, seuls ce dont on se sert est chargé !

Chargement du container 50 fois plus rapide que le démarrage d'une VM

A ressources identiques, nb d'applications multipliées par 5 à 80.

Permet l'avènement des architecture micro-services (application composée de nombreux services/container devient envisageable)

Nécessite une approche DevOps



Le succès de docker

Facilité : un conteneur vide peut être récupéré et instancié en une ligne de commande, et le lancement d'un processus s'effectue de la même manière que sur une machine locale.

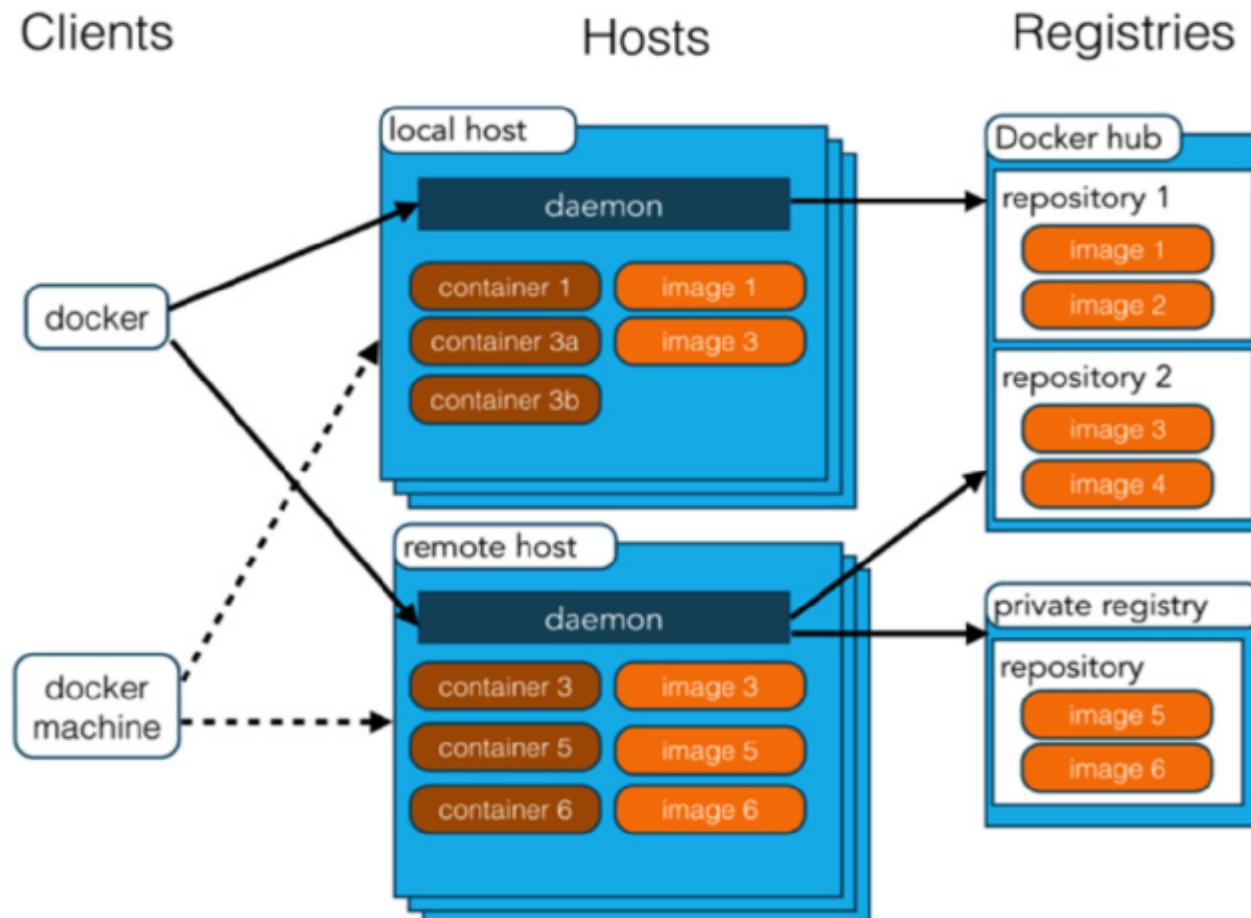
Automatisation : Ce chargement peut être automatisé par la réalisation d'un fichier contenant un texte écrit dans un format décrivant les étapes de chargement du processus dans le conteneur. *Dockerfile*

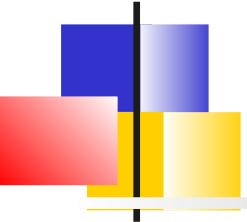
Diffusion : Toute personne ayant créé un conteneur peut ensuite diffuser le tout par Internet, en ayant la garantie que n'importe quelle personne pourra exécuter le conteneur.

Rapidité d'exécution : le lancement du conteneur se fait quasi instantanément, et sans nécessiter de temps d'installation.

Portabilité : toute personne exécutant le conteneur obtiendra exactement le même résultat que l'émetteur du moment qu'il soit pourvu du programme Docker.

Docker architecture





Commandes Docker

```
#Récupération d'une image
docker pull ubuntu

#Récupération et instantiation
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

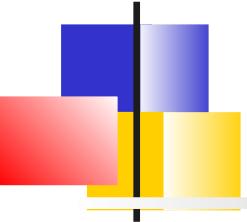
#Lister les images
docker images

#Identifier les différences entre l'image et le conteneur
docker diff <container_id>

#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]
```

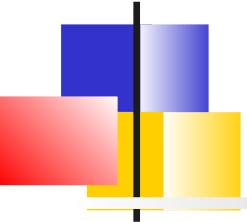


Exemple DockerFile

```
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" ]
```

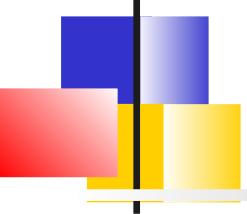


Communication entre machines

Chaque conteneur s'exécutant a sa propre interface réseau (gérée par Docker)

Par défaut, il est isolé

- De la machine hôtes
- Des autres containers



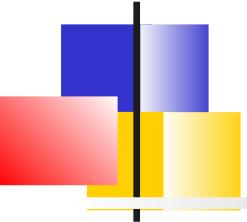
Communication avec la machine hôte

Pour communiquer à la machine hôte, en général l'image déclare les ports utilisés dans son fichier *Dockerfile* (en fait ce n'est qu'informatif)

EXPOSE 8080

Ensuite au démarrage d'un conteneur on associe le port exposé à un port disponible de la machine hôte.

```
docker run -p 80:8080 myImage
```



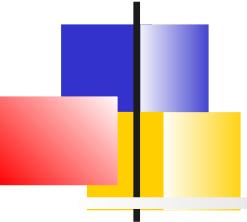
docker-compose

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

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

Docker s'installe séparément sur Linux et est inclus dans Docker pour les distributions Mac ou Windows

Orienté au départ pour faciliter le développement et l'intégration ; il intègre de plus en plus des fonctionnalités pour la production



Exemple configuration

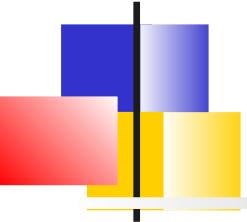
```
# Le fichier de configuration définit des services, des networks et des volumes.

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

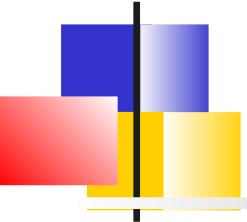
pull / push : Pull/push les images des services

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



Docker Swarm

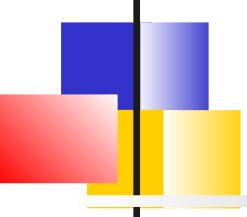
Docker Swarm offre une solution de clustering des docker machines

Un ensemble de machines est alors vu comme une seule et l'architecture de cluster augmente la sûreté de fonctionnement.

C'est donc un produit orienté production



Outils de SCM et workflow de collaboration

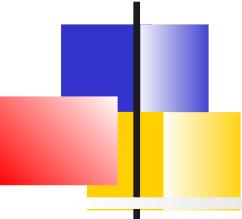


SCM

Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

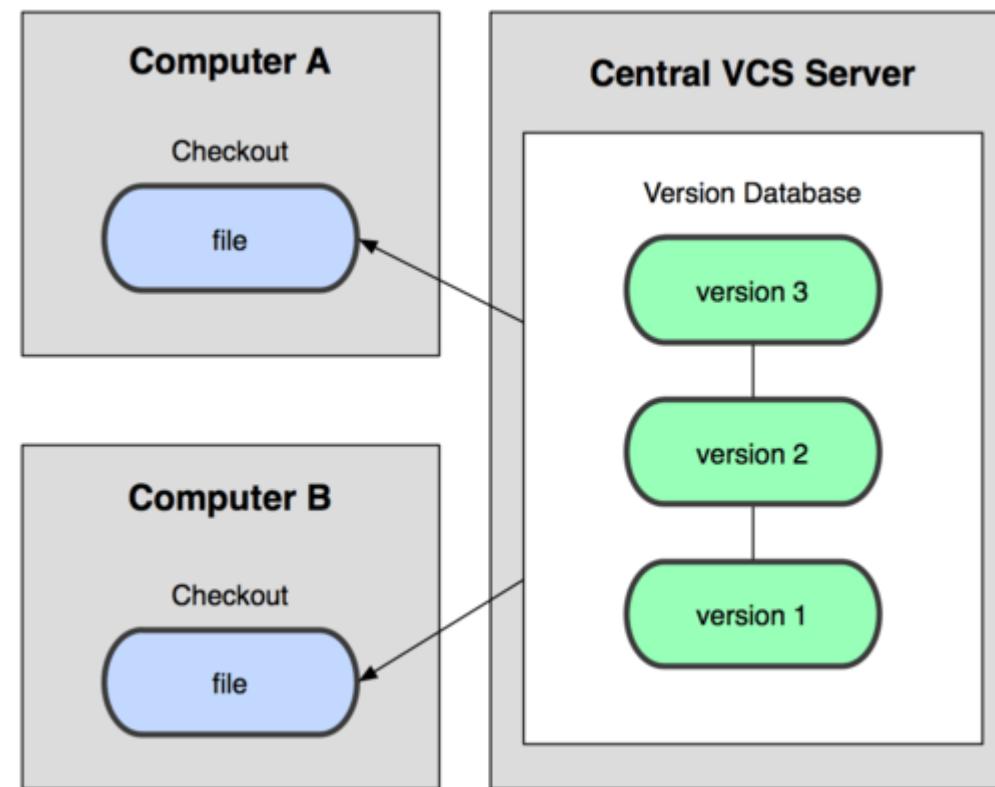
Le système permet :

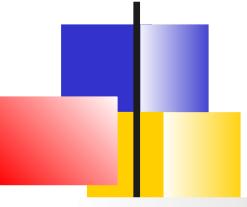
- De restaurer des fichiers
- Restaurer l'ensemble d'un projet
- Visualiser tous les changements effectués et leurs auteurs



SCM centralisés

CVS, Subversion et Perforce



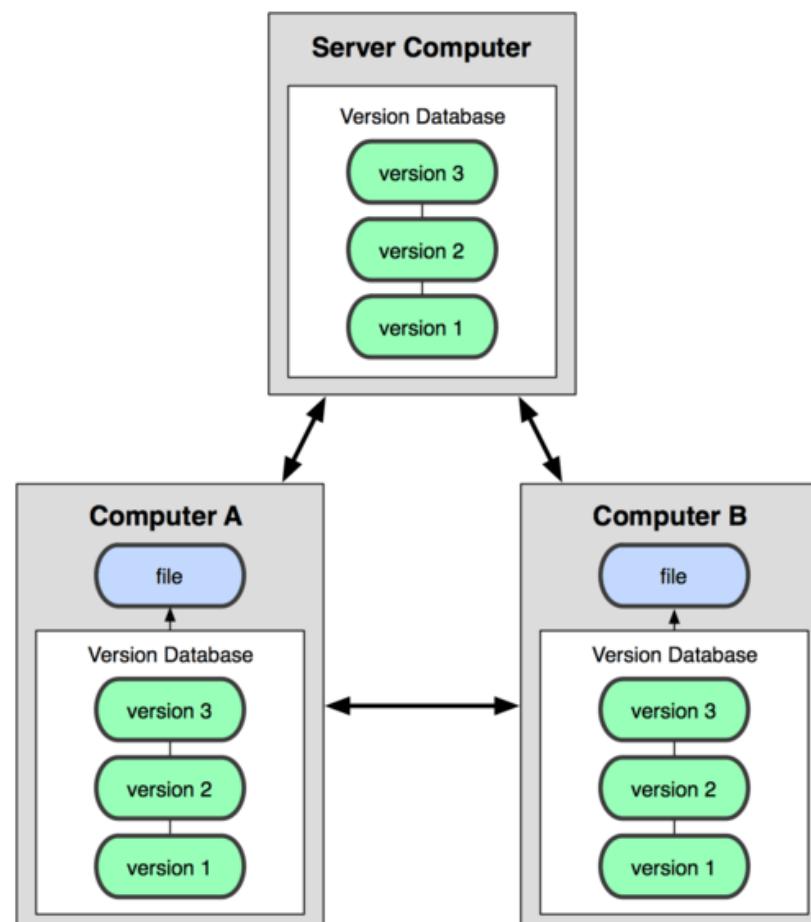


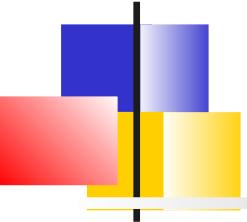
SCM distribué

Avec les SCMs **distribués**, les clients ne récupèrent pas le dernier instantané des fichiers mais l'intégralité du dépôt ou référentiel

- Ainsi si un serveur défaillait, les clients peuvent recréer le référentiel à partir de leur copie locale et continuer à collaborer
- Le fait de disposer de plusieurs référentiels distants permet de collaborer avec différents groupes de personnes de façon différente et de mettre en place différents workflows
- Exemple : *Git*

SCM distribué

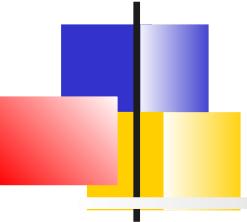




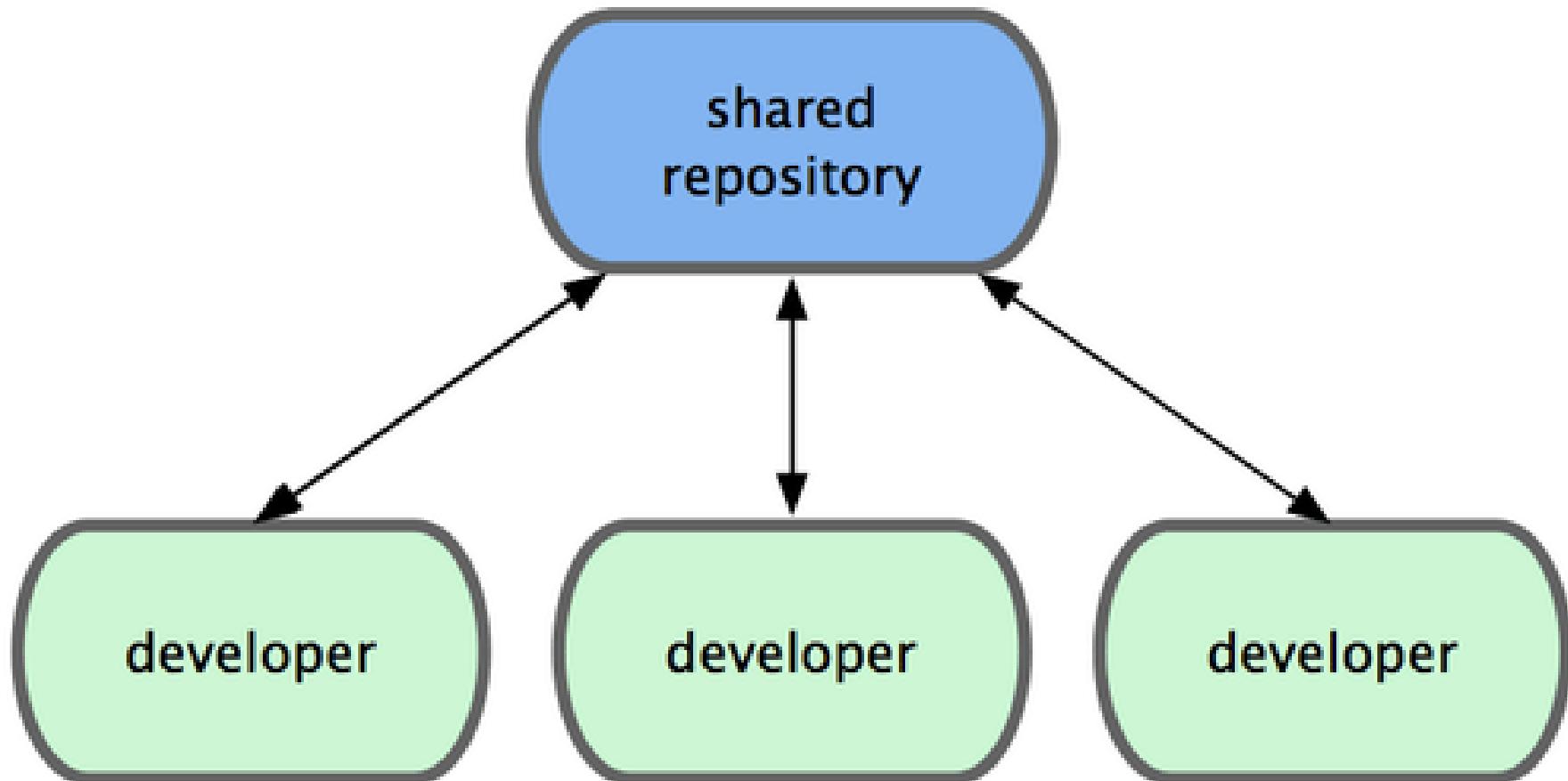
Workflows de collaboration

L'utilisation de dépôt distribué a fait apparaître des nouveaux workflows de collaboration entre les développeurs.

- Workflow centralisé : Workflow identique à ceux utiliser avec les SCM centralisé. Convient au tout petit projet ~ 2 développeurs
- Gitflow : Workflow d'entreprise gérant un processus de release et travaillant sur plusieurs versions en parallèle
- Avec intégrateur : Projet OpenSource avec collaborations diverses

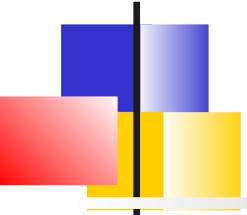


Workflow centralisé



Scénario





Gitflow

Le workflow **Gitflow** définit un modèle de branches orientées vers la release d'un projet

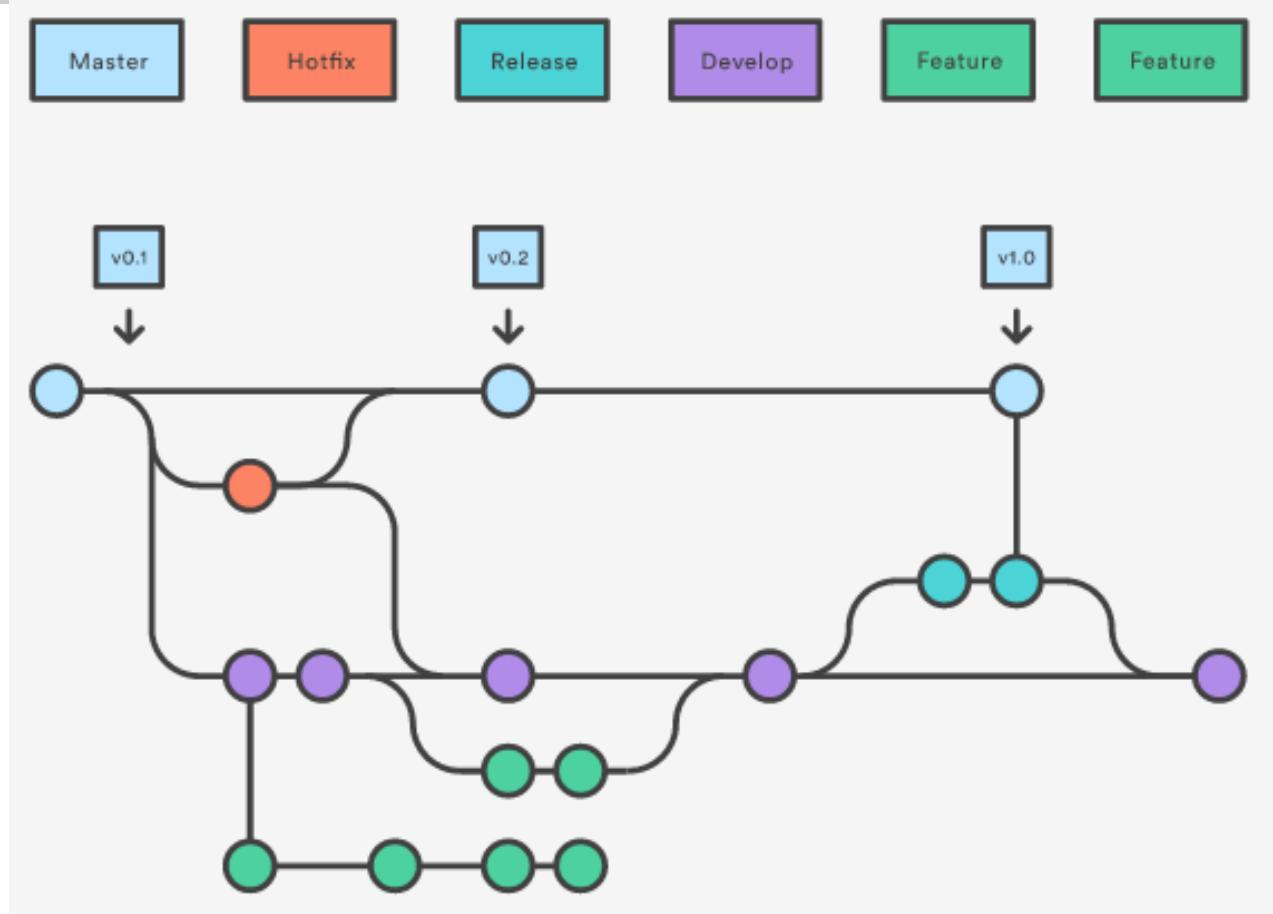
Il est adapté pour la gestion de grands projets

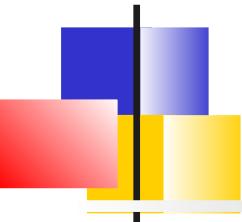
Le workflow Gitflow est une extension des « branches thématiques »

Il assigne des rôles très spécifiques aux différentes branches et définit quand et comment elles doivent interagir

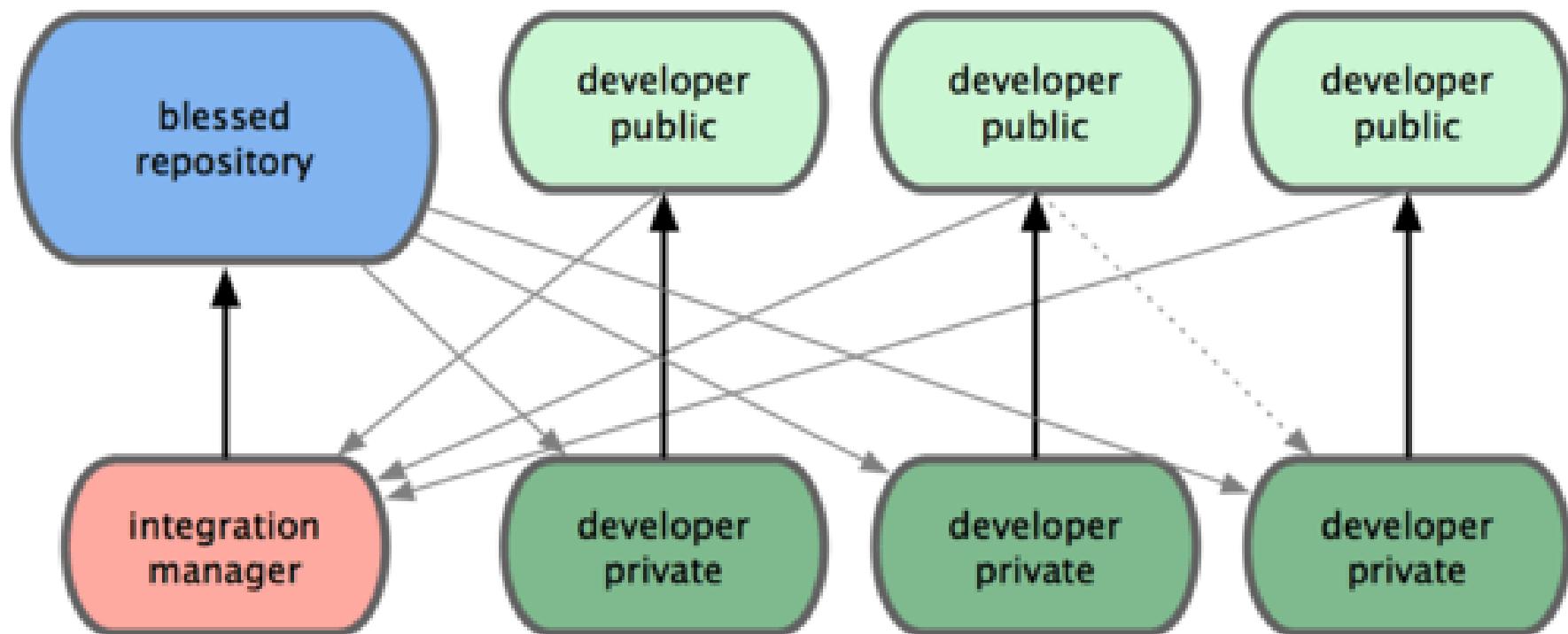
En plus de la branche longue, il utilise différentes branches pour la préparation, la maintenance et l'enregistrement de releases

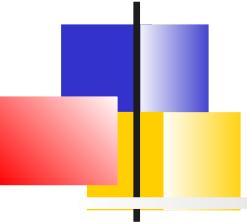
Branches Gitflow





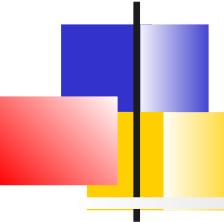
Workflow avec intégrateur



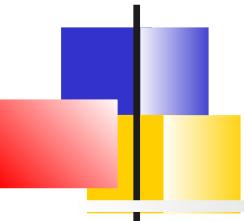


TP

Mise en place d'un dépôt Git



Outils de build



Apache Ant



Librairie Java « préhistorique » permettant d'automatiser les tâches de build.

Un ingénieur de build écrit des scripts ant (xml) permettant de construire, déployer, archiver, ...

Le projet ne respecte pas de structure particulière



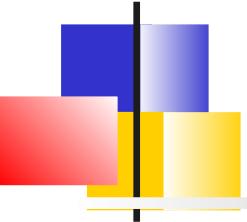
Apache Maven

Outil de gestion de projet permettant de gérer le build, les tests, les déploiement, la génération de documentation d'un projet Java

Basé sur le *Project Object Model (pom.xml)*

Gestion des dépendances via la notion de dépôts d'artefacts

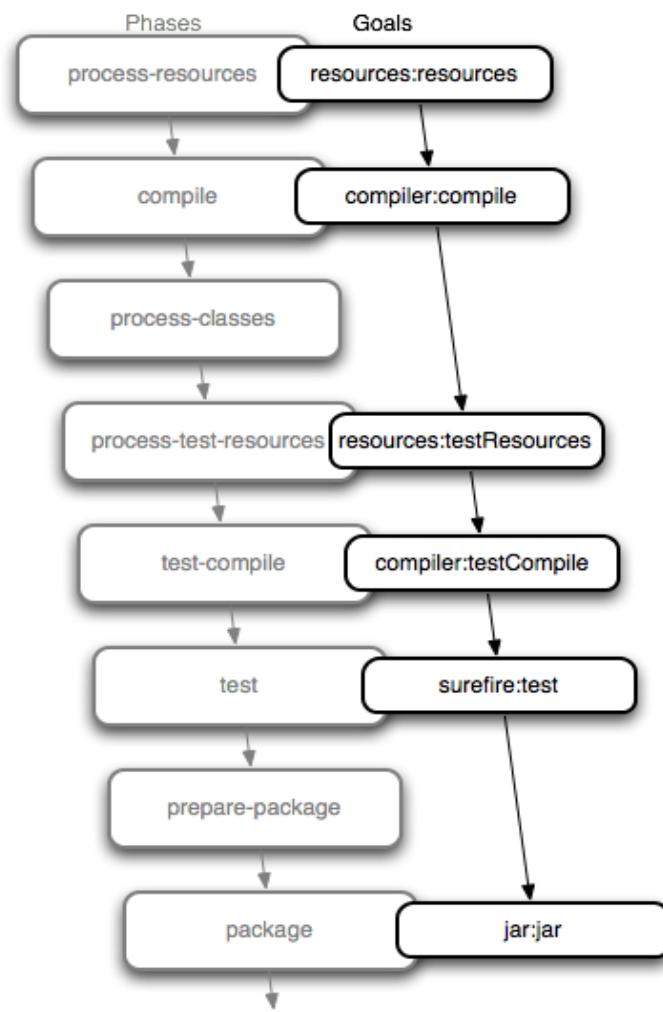
Impose une structure de projet via les archetypes



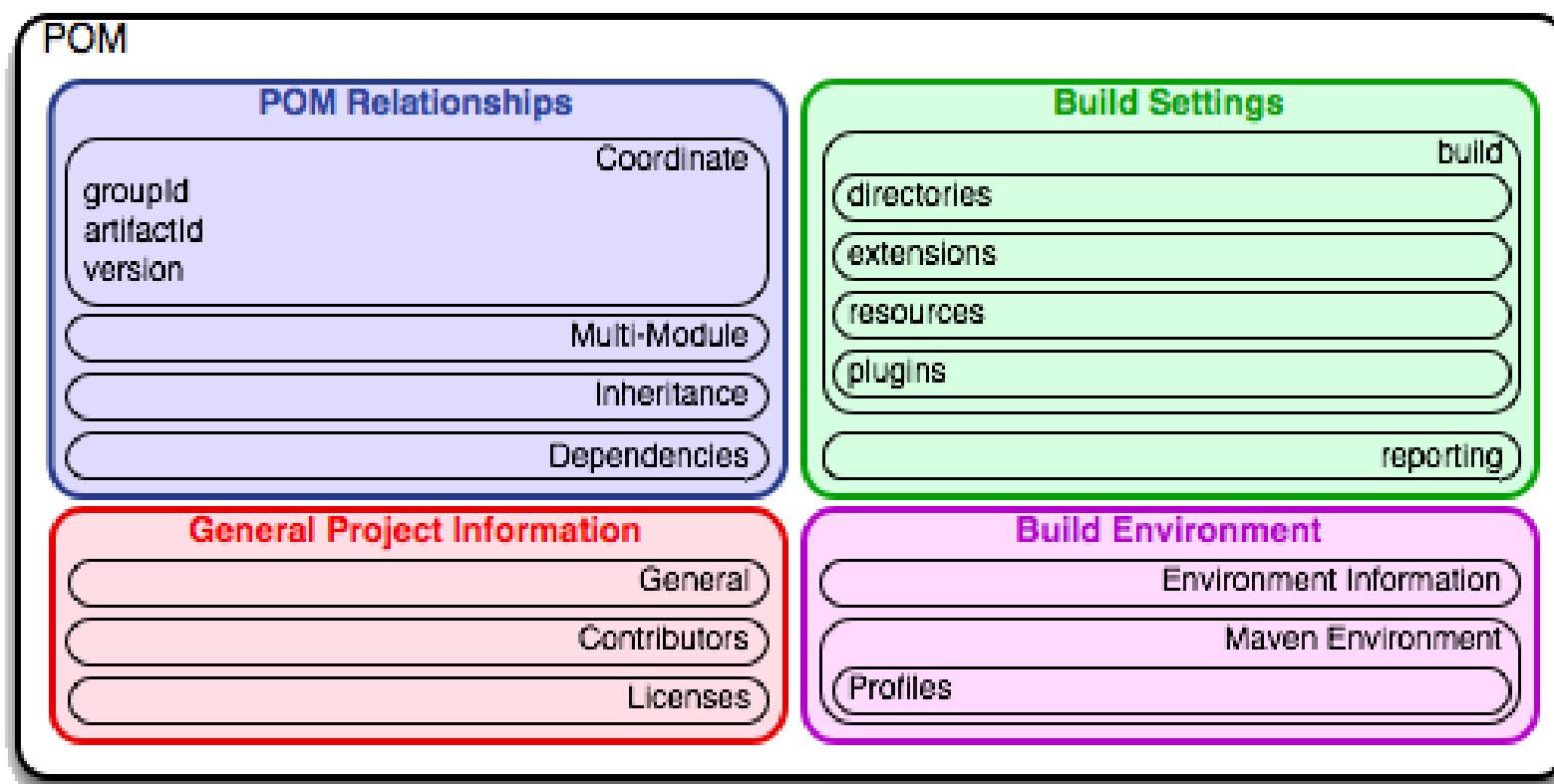
Cycle de vie Maven

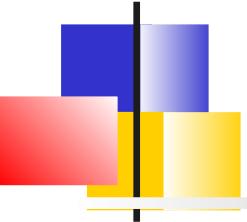
- Le cycle de vie est une **séquence ordonnée de phases** impliquées dans la construction d'un projet
- Maven supporte différents cycles de vie. Le cycle de vie par défaut est le plus souvent utilisé, il commence avec une phase validant l'intégrité du projet et termine avec une phase déployant le projet en production
- Les phases du cycle de vie sont intentionnellement « vagues » : *validation, test ou déploiement*
Elles peuvent customiser en fonction des projets.

Cycle de vie par défaut



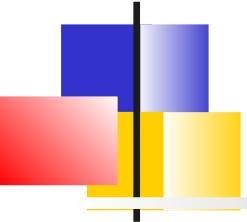
Contenu du pom.xml





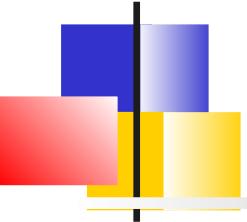
Dépôts Maven

- Un repository ou dépôt désigne l'espace de stockage des fichiers nécessaires à la construction du projet (plugins, librairies dépendantes, jar construit)
- Chaque artefact est identifié de façon unique grâce à ses coordonnées Maven (*groupe, artifactId, version*)
- Il peut être :
 - **Public** : Exemple Maven Central où se trouve la plupart des projets OpenSource Java
 - **Privé** : Le dépôt d'une entreprise
 - **Local** : (Les fichiers ont été rapatriés localement)



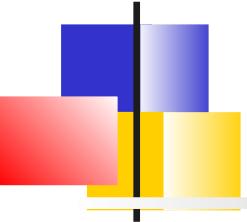
Gestion des dépendances

- La possibilité de localiser un artefact à partir de ces coordonnées permet d'indiquer les dépendances dans le projet POM.
- Le fait que le POM associé à tout artefact soit stocké dans le dépôt permet à Maven de résoudre les dépendances transitives. Le support pour les dépendances transitives est une de fonctionnalités les plus puissantes de Maven.
- Il est possible de définir différents périmètres de dépendances. Par exemple, la dépendance *junit:junit:jar:3.8.1* n'existe que dans le périmètre de test.
- Le périmètre fourni indique à Maven si il doit empaqueter la dépendance ou non.



Dépendances transitives

- Maven permet de s'affranchir des dépendances transitives. Si par exemple, votre projet dépend d'une librairie B qui dépend elle-même d'une librairie C, il n'est pas nécessaire d'indiquer la dépendance sur C
- Lorsqu'une dépendance est téléchargée vers le dépôt local, Maven récupère également le *pom* de la dépendance et est donc capable d'aller chercher les autres artefacts dépendants. Ces dépendances sont alors ajoutées aux dépendances du projet courant.
- Si ce comportement n'est pas désirable, il est possible d'exclure certaines dépendances transitives



Automatiser le versionning et les release avec Maven

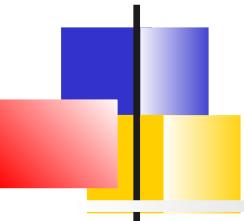
L'objectif est d'automatiser et synchroniser :

- La création d'un tag dans le SCM

- La publication d'une release dans un repository interne

Le scénario : pouvoir périodiquement publier une copie versionnée et incrémenter la version du projet

Intégration continue, nightly build,



Graddle



OpenSource reprenant les idées de Ant et Maven mais basé sur une description Groovy

Représentation graphique du build

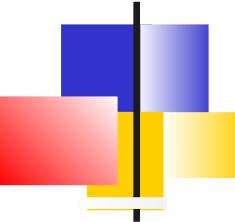
- Multi-langages : Au départ pour Java, Groovy et Scala puis Javascript, Python, C++
- Intégration avec Containerisation et CI
- Personnalisation par programmation
- Outils de reporting associés



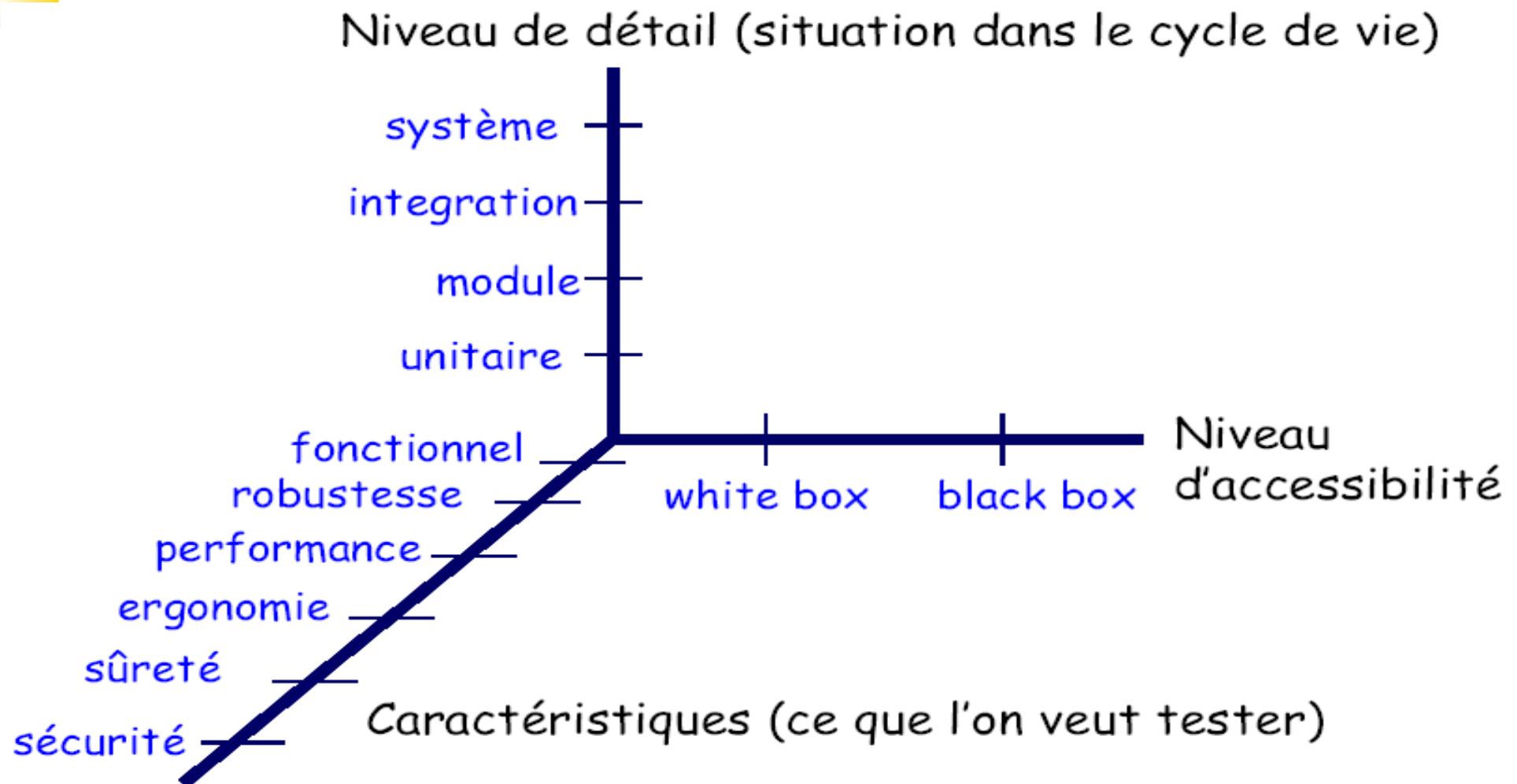
Le test

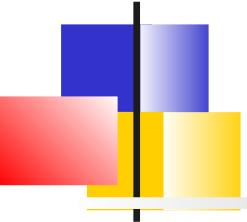


Les tests



Types de test



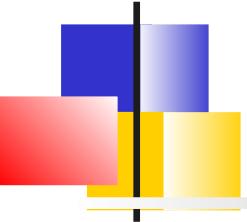


Isolation et Mock objects

Les parties à tester doivent être isolées

- Tests peuvent être réalisés même si les parties dont dépend le code ne sont pas encore développées
- Permet d'éviter les effets de bord
- Permet de tester le code lorsque les parties dont il dépend ont des erreurs

=> Utilisation de Mock Objects simulant les interfaces

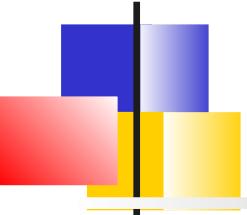


JUnit: le framework

Framework fourni par les gourous à l'origine de XP...

Originellement destiné à la plate-forme Java mais depuis porté dans d'autres langages (C/C++/.NET etc..)

Uniquement accès sur les tests unitaires...



Test d'intégration

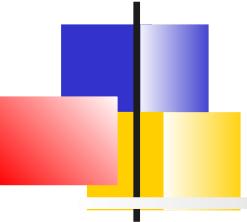
Les tests d'intégration consiste à exécuter des tests sans les mock objects
Ces tests sont plus lourd que les tests unitaires

- l'infrastructure complète de l'application est nécessaire (Serveur applicatifs, format de déploiement)
- Les données de test doivent être préparées (création de bd de test, réinitialisation afin que les différents tests soient indépendants)

=> Ils sont plus lents et sont intégrés généralement au processus de build après que tous les tests unitaires soient passés

Il existe des solutions permettant de faciliter leur mise en place. Exemple Arquillian :

- Tests d'intégration peuvent s'effectuer de l'IDE ou du build
- Arquillian s'occupe de tout, création d'archive, déploiement, injection de dépendances



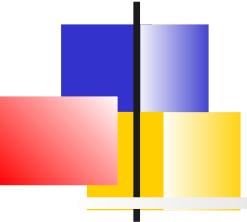
Exemple Arquillian

```
@RunWith(Arquillian.class)

public class MySimpleBeanTest {
    @Inject private MySimpleBean mysimplebean;

    @Deployment public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class, "test.jar")
            .addClass(MySimpleBean.class)
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Test
    public void testIsDeployed() {
        Assert.assertNotNull(mysimplebean);
    }
}
```

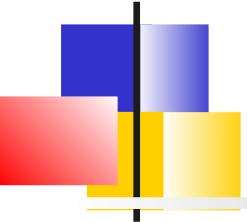


Tests fonctionnels

Les tests fonctionnels sont des tests en boîte noire qui exécutent des scénarios d'usage de l'application et vérifient leur conformité

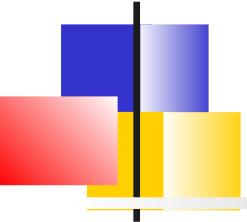
Ils sont en général fortement dépendant de l'interface utilisateur et de ce fait sont difficiles à automatiser et maintenir

Dans le cas d'une application web, ils simulent ou pilotent un navigateur et vérifient les réponses fournies par le serveur



Exemple : Selenium Web Driver

```
public class MonTest {  
    public static void main(String[] args) {  
        // Créer une nouvelle instance de Firefox driver  
        WebDriver driver = new FirefoxDriver(); // Utiliser ca pour visiter Google  
        driver.get("http://www.google.com");  
  
        // Déterminer le champ dont le name et q  
        WebElement element = driver.findElement(By.name("q"));  
        element.sendKeys("Selenium"); // Taper le mot à chercher  
        element.submit(); // Envoyer la formulaire  
        System.out.println("Page title is: " + driver.getTitle()); // Vérifier le titre de la page  
        // Google fait la recherche dynamique avec JavaScript.  
        // Attendre le chargement de la page de 10 secondes  
        // Vérifiez le titre "Selenium - Recherche Google"  
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {  
            public Boolean apply(WebDriver d)  
            {return d.getTitle().toLowerCase().startsWith("selenium");}  
        });  
        System.out.println("Page title is: " + driver.getTitle());  
        //Fermer le navigateur  
        driver.quit();  
    }  
}
```



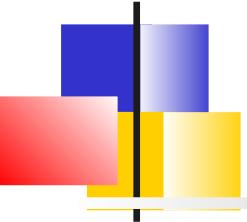
Tests d'acceptance

Les tests d'acceptance ont pour but de valider que la spécification initiale est bien respectée

Les tests sont mis au point avec le client, le testeur et les développeurs

Dans les méthodes agiles, ils complètent et valident une « User Story »

En utilisant l'approche BDD (*Behaviour Driven Development*), l'expression des tests peut être faits en langage naturel.



Exemple Gherkin

Feature: Refund item

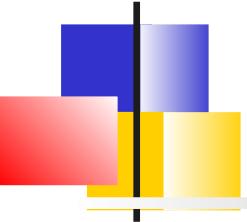
Scenario: Jeff returns a faulty microwave

Given Jeff has bought a microwave for \$100

And he has a receipt

When he returns the microwave

Then Jeff should be refunded \$100

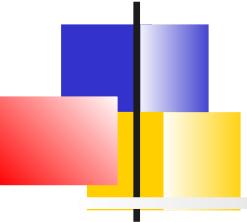


Tests de performance

Un test de performance est un test dont l'objectif est de déterminer la performance d'un système informatique. [Wikipedia]

Plus concrètement, mesurer les temps de réponse d'un système en fonction de sa sollicitation.

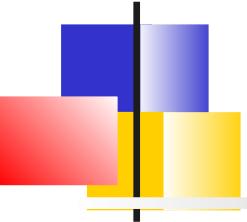
=> Définition très proche de celle de test de charge



Plan de test

Le **plan de tests** est l'expression du besoin de la campagne de tests. C'est le premier livrable qui contient généralement :

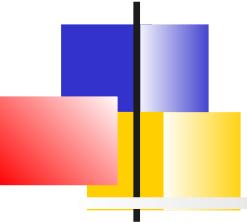
- la présentation du projet
- les objectifs,
- le modèle de charge,
- le type de tests à réaliser,
- les scénarios fonctionnels (ou cas d'utilisation) à tester accompagnés des jeux de données nécessaires
- un planning d'exécution de ces tests



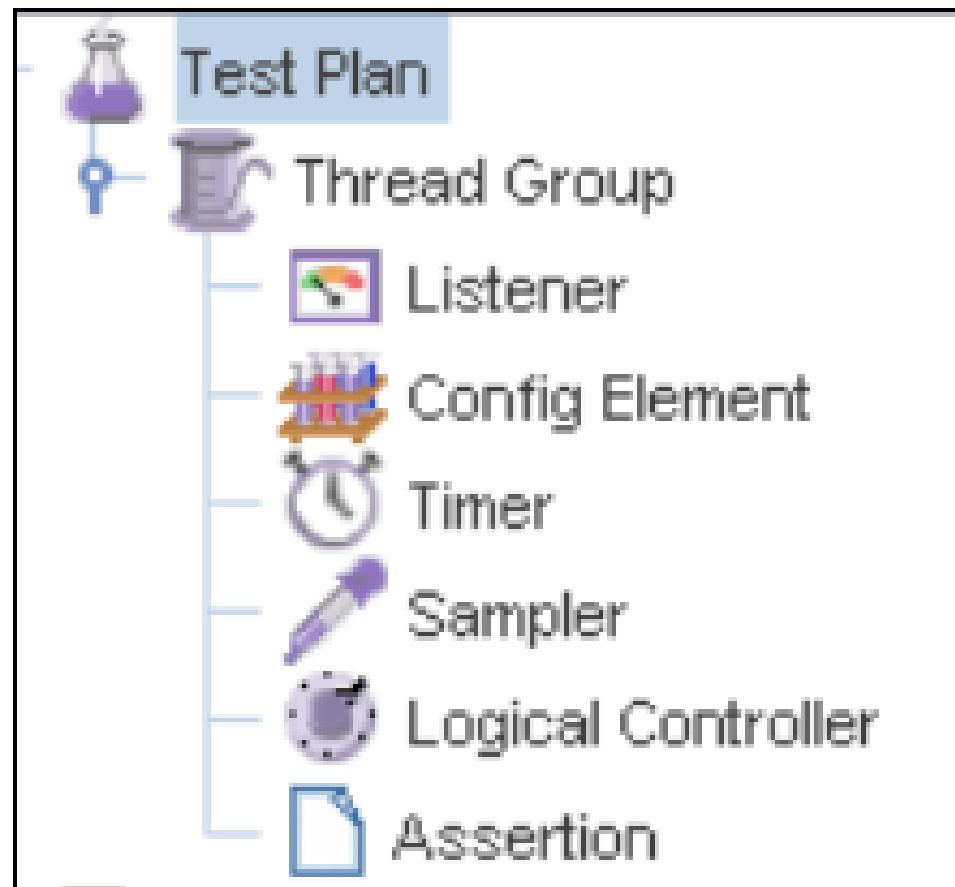
Modèle de charge

La modélisation de la charge consiste à estimer la **charge représentative de l'activité réelle ou attendue** de l'application

- Cette modélisation s'estime à partir d'un modèle d'usage : nombre d'utilisateurs simultanés, nombre de processus métier réalisés, périodes d'utilisation, heures de pointe...
- Elle contient un nombre d'utilisateurs à simuler, leur répartition sur les différents scripts (scénarios fonctionnels), leurs rythmes d'exécution respectifs.
- Accessoirement, le modèle peut tenir compte des profils de montée ou descente de charge des groupes d'utilisateurs

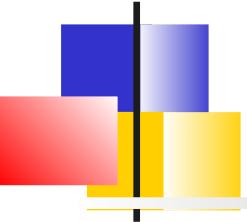


Elements d'un plan de test JMeter





Serveur d'intégration continue



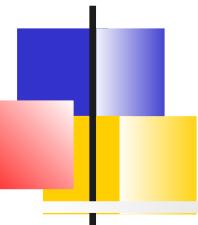
Serveur d'intégration continue

Un serveur d'intégration continue a pour objectif :

- Automatiser les builds et les déploiements en intégration ou en production
- Fournir une information complète sur l'état du projet (état d'avancement, qualité du code, couverture des tests, métriques performances, documentation, etc.)

Il est multi-projets, multi-branches, multi-configuration

Il nécessite beaucoup de ressources

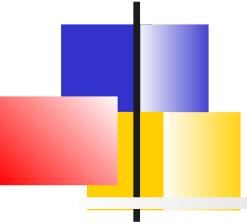


Architecture Maître / esclaves

Le serveur central Jenkins distribue les jobs de build sur différentes ressources appelés les esclaves.

Les esclaves sont :

- Des machines physiques ou virtuelles sont préinstallés les outils nécessaires au build
- Des images docker qui sont alors exécutés lors du build

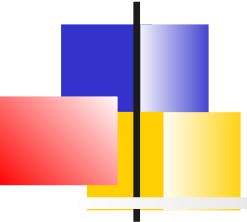


Jobs et pipeline

Un job est une façon spécifique de compiler, tester, packager, déployer un projet.

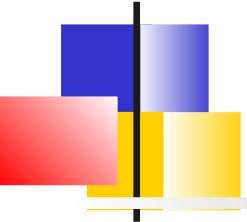
Les build jobs peuvent être très différents : compilation, test unitaires, rapport qualité, génération de documentation, création d'une release

=> Un projet contient en général plusieurs jobs organisés en **pipeline**

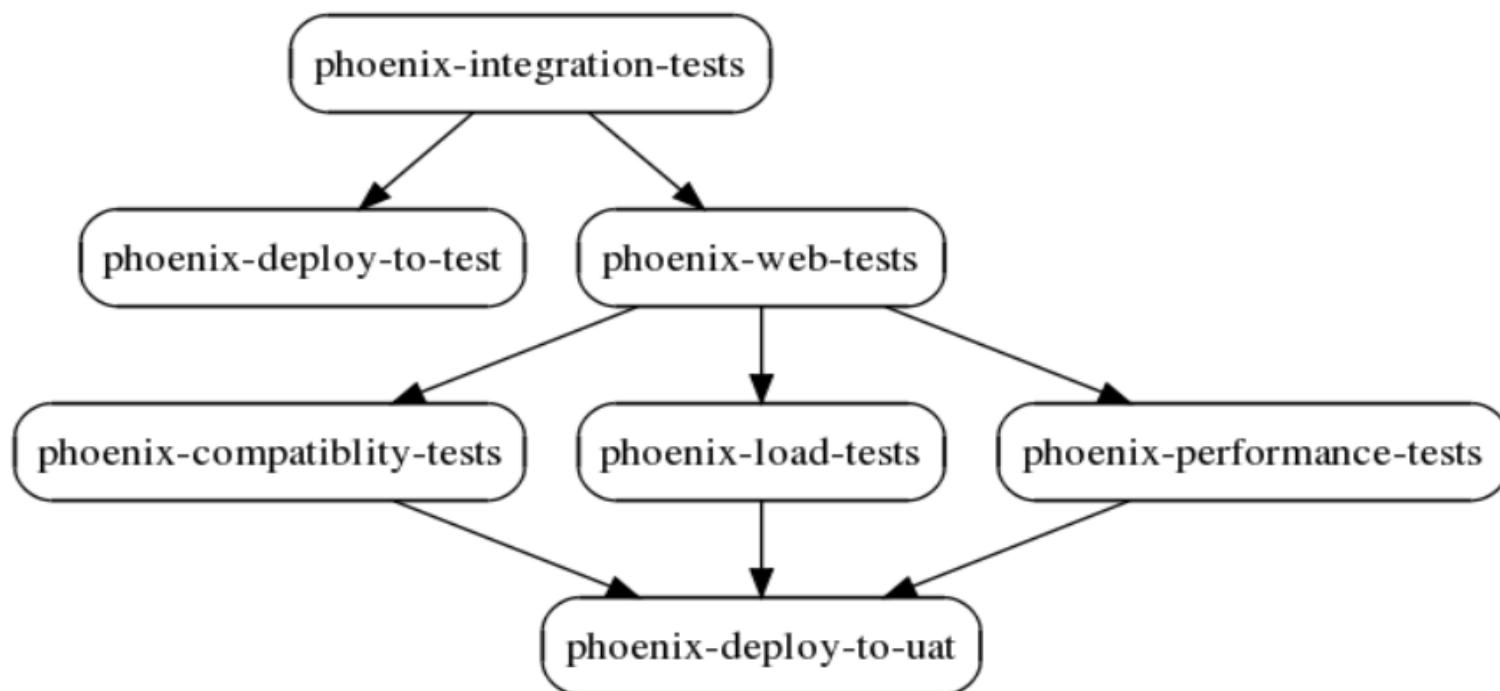


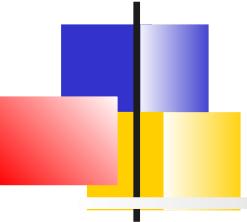
Scénario typique d'un job

- 1.Un développeur pousse une modification dans un dépôt
- 2.Un job est déclenché, il récupère les dernières versions des sources du SCM dans l'espace de travail
- 3.Il effectue le build (compilation, tests, packaging ou autre)
- 4.Des résultats sont collectés et publiés sur le serveur, des artefacts sont archivés
- 5.L'équipe est éventuellement notifiée
- 6.Le job déclenche éventuellement un ou plusieurs autres jobs



Exemple séquencement de jobs





Pipeline et promotion

Une **pipeline** représente le cycle de vie d'un build et donc d'une révision particulière d'un projet.

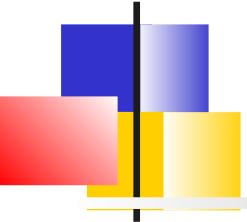
Une pipeline est constituée de phases. Par exemple :

1. Développement
2. Test qualité
3. Test d'acceptance
4. Production

La **promotion** d'un build représente le passage d'une phase à une autre

Legacy Pipeline avec déploiement manuel



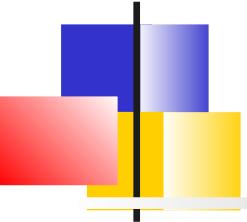


Approche DevOps

Dans la dernière version de Jenkins,
l'approche DevOps est permise.

Un fichier **Jenkinsfile** faisant partie
intégrante des sources du projet décrit la
pipeline de déploiement continu

- Le fichier est commité et versionné dans
un dépôt Git
- La description est effectuée via un langage
spécifique DSL construit avec Groovy



Illustration

```
#!groovy
stage('Init') {
    node {
        checkout scm
        gitCommit = sh(returnStdout: true, script: 'git rev-parse HEAD').trim()
    }
}
stage('Build') {
    parallel frontend : {
        node {
            checkout([$class: 'GitSCM', branches: [[name: gitCommit ]], dir("plbsi-front") {
                sh 'npm install' sh './node_modules/.bin/ng build -e prod'
                stash includes: 'dist/**', name: 'front'
            } }],
            backend : {
                node {
                    checkout([$class: 'GitSCM', branches: [[name: gitCommit ]], sh 'mvn clean test'
                } })
            }
        }
    }
}
stage('Integration') {
    node {
        dir("plbsi-rest/src/main/resources/static") { unstash 'front' }
        sh 'sudo cp plbsi-rest/target/plbsi-rest-4.0.0-SNAPSHOT.jar /home/plbsi/bin/plbsi-rest.jar'
        sh 'sudo /etc/init.d/plbsi-rest.sh start'
    }
}
```

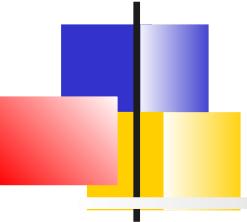


Tableau de bord (multi-branches)

Jenkins

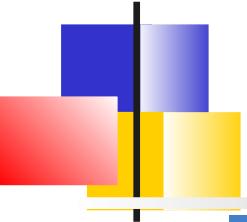
Pipelines Administration Déconnexion

Tableau de bord Nouveau Pipeline

Favorites

<input checked="" type="checkbox"/> plbsi-angular	 master	 #5c3e3ac	 2 months ago	  
<input checked="" type="checkbox"/> plbsi-angular	 develop	 #d07b3cb	 4 months ago	  

Nom	Santé	Branches	Pull requests
AutomaticDeployDev		-	- 
DeployPrevious		-	- 
DeployProduction		-	- 
MinimalCheck		-	- 
plbsi-angular		1 en échec	
TestManual		-	



Détail pipeline (*Git comments*)

Jenkins

Pipelines Administration Déconnexion

plbsi-angular ★ ⚙️

Activité Branches Pull requests

État	Run	Commit	Branche	Message	Durée	Terminé	
✓	59	5c3e3ac	master	Reverting changes made to test the app	2m 43s	2 months ago	↻
✓	58	f02c8c3	master	Testing	1m 55s	2 months ago	↻
✓	57	4817901	master	Truncate to 1000 elements	2m 47s	2 months ago	↻
✓	56	2a1bd27	master	Displaying number of results	3m 43s	2 months ago	↻
✓	55	078c37d	master	Minor fixes	2m 33s	3 months ago	↻
✓	54	bd30380	master	Minor fixes	1m 42s	3 months ago	↻
✓	53	771198a	master	Triggering a build	1m 43s	3 months ago	↻
✓	52	0502224	master	Fixing minor bug	1m 53s	3 months ago	↻
✓	51	15847a9	master	Adding the CV upload feature (Work in progress)	1m 47s	3 months ago	↻
✓	50	58e1858	master	- Removing mandatory constraints on intervenant details fields, -	2m 5s	3 months ago	↻
✗	6	d380d47	prodEnv	Essai JenkinsFile	1m 46s	3 months ago	↻

Détail d'un build

Jenkins

Pipelines Administration Déconnexion

plbsi-angular ★ ⚙️

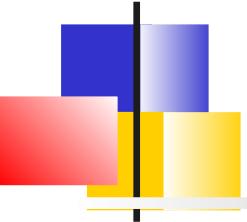
✓ plbsi-angular #59 Pipeline Modifications Tests Artefacts ⚡ ⚙️ 🗑️ ✖️

Branche: master 2m 43s Modifications par pro.rsausage
Commit: 5c3e3ac 2 months ago

```
graph LR; Init((Init)) --> Build((Build)); Build --> Integration((Integration)); subgraph Backend [backend]; Build --> Backend; end; subgraph Frontend [frontend]; Backend --> Frontend; Frontend --> Integration;
```

Étapes - Integration

✓	> Restore files previously stashed	<1s
✓	> Shell Script	27s
✓	> Shell Script	<1s
✓	> Shell Script	<1s
✓	> Shell Script	<1s
✓	> Shell Script	2s



TP7 – Intégration continue

Let's do a job avec Jenkins ...