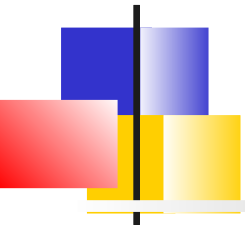
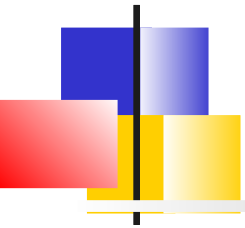

PLB
FORMATION INFORMATIQUE ET MANAGEMENT



DevOps Java

David THIBAU – 2020

david.thibau@gmail.com



Agenda

Introduction

- Le constat DevOps
- CI et CD
- Cycle de vie du code et outils DevOps
- DevOps et Architecture des systèmes

Gestion des sources

- Typologie des SCMs
- Workflows de collaboration et usage des branches

Outils de build

- Caractéristiques demandées, composants et outils
- Maven
- Tests et analyse qualité

Release et dépôts d'artefacts

- Dépôts privés
- Processus de release
- L'outil Nexus

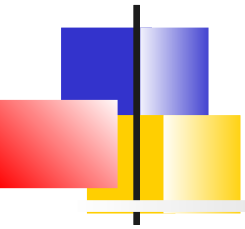
Plateforme d'intégration continue

- Concepts, Architecture
- Pipelines typiques
- Solutions

Déploiements

- Considérations
- Virtualisation
- Gestion de configuration. Le cas Ansible
- Containerisation. Le cas docker
- Orchestrateur de conteneur
- Kubernetes
- Kubernetes dans la pipeline CD

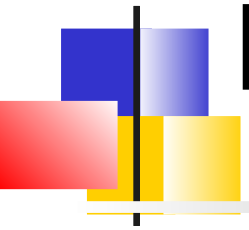
Le constat DevOps



Cycle de vie d'un logiciel

Le cycle de production d'un logiciel nécessite plusieurs environnements :

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

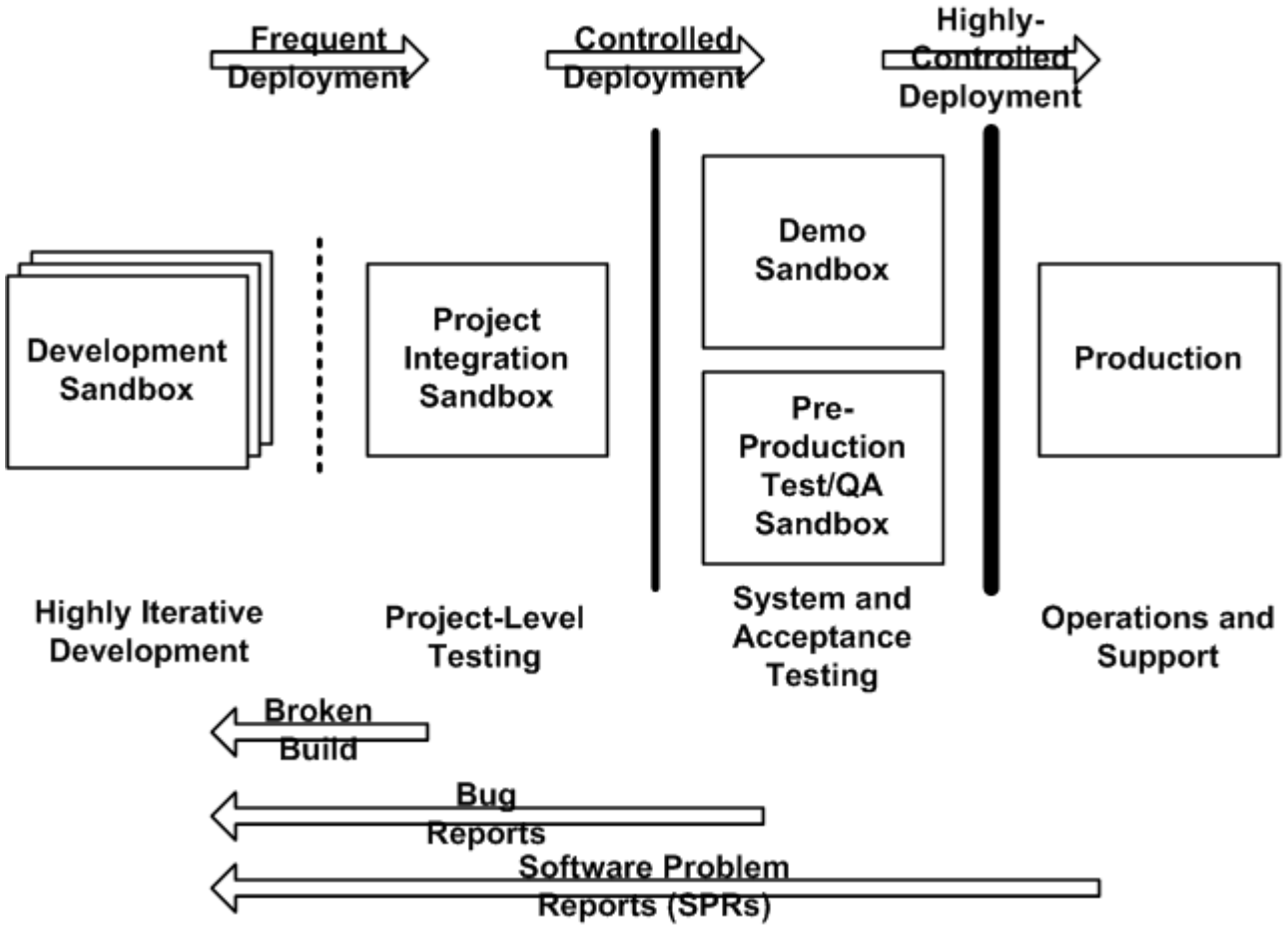


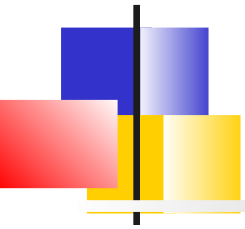
Disparité des environnements

Les environnements ne peuvent que diffèrer :

- **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 des charges, des données
- **QA** : Le plus proche de la production mais pas les mêmes données, pas les mêmes charge.
- **Production** : Qualité de service, charge réelle, données de production, utilisateurs finaux

Fréquence de déploiement



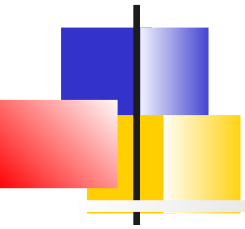


Disparité des objectifs

Ces environnements étaient traditionnellement gérés et utilisés par des équipes distinctes qui … souvent communiquaient peu

Les équipes ont de plus des objectifs différents

- Développeur : Implémenter les fonctionnalités requises dans le temps imparti.
- Intégrateur : Dimensionner l'architecture pour atteindre des SLA
- QA : Valider fonctionnellement le système dans des scénarios pas toujours anticipés par les dev.
- Opérations : Garantir la stabilité du système et des infra-structures

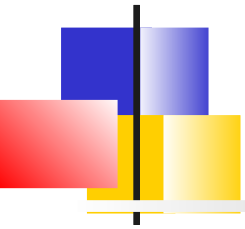


Le constat DevOps

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

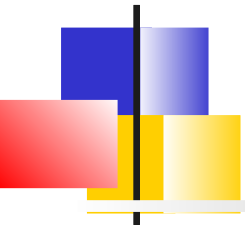


Approche *DevOps*

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.

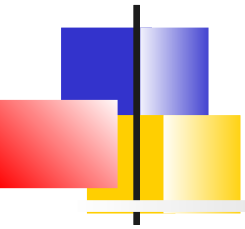
Cela impose :

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



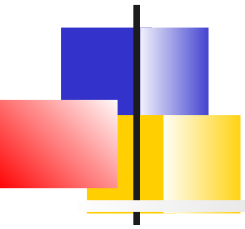
Pratiques *DevOps* (1)

- Un déploiement régulier des applications dans les différents environnements.
  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" ;



Pratiques *DevOps* (2)

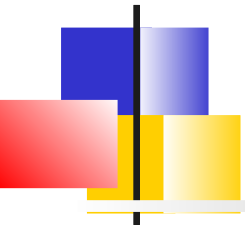
- Une boucle d'amélioration courte
  i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation et de la qualité de production factualisée par des métriques et indicateurs "clé".
- Les configurations des différents environnements, des builds, des tests centralisées dans le même SCM que le code source



« As Code »

Les outils *DevOps* permettent d’automatiser/exécuter la construction/ fourniture des ressources à partir de l’unique point central de vérité

On parle de *Build As Code*, *Infrastructure As Code*, *Pipeline As Code*, *Load Test As Code*, ...



Objectif ultime

- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime
- Possibilité d’effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes auto-correctifs, capable de se reprendre en cas de défaillance ou erreurs

CI/CD

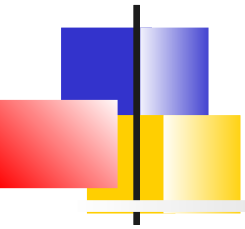
Avant l’intégration continue

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 ressemblante à la production.

Différents types de problèmes pouvaient survenir nécessitant parfois des réécritures de lignes de code et introduire des délais dans la livraison

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

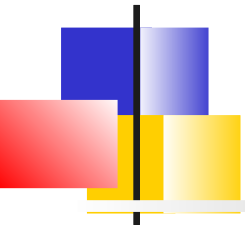


Plateforme d’intégration continue (PIC)

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, teste automatiquement et déploie l'application dans l’environnement d’intégration

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



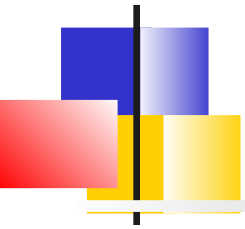
Build is tests !

Chaque modification poussée par un développeur dans le SCM va être automatiquement testée dans le maximum d'environnements.

=> A tout moment l'application est considérée comme étant potentiellement livrable

L'activité de build intègre alors **tous les types de tests** que peut subir un logiciel (unitaires, intégration, fonctionnel, performance, analyse qualité)

Outil de communication et de motivation



La PIC permet également de publier et historiser les résultat des builds:

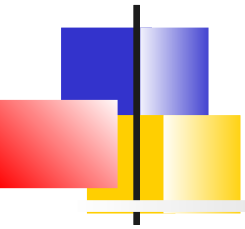
- Rapports de tests et d'analyse qualité
- Couverture fonctionnelle et avancement du projet
- Documentation
- Accès aux environnements de test

=> Confiance dans la robustesse du code développé, motivation

=> Réduction des coûts de maintenance.

=> Transparence : les métriques sont visibles par tous les acteurs (fonctionnels, techniques)

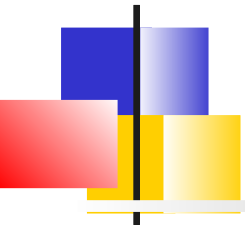
=> Plateforme de recette disponible en permanence permettant d'affiner les spécifications



Livraison continue
Continous delivery

La **livraison continue** consiste à livrer en permanence des versions du logiciel stable et potentiellement déployable en production.

Le déploiement en production étant automatisé, les fonctionnels peuvent décider en toute autonomie le déclenchement d’une mise en production.



Déploiement continu

Continous deployment

Combiné avec des tests d'acceptance automatisé, les builds réussis peuvent être déployer automatiquement en production sans aucune intervention manuelle

C'est le stade ultime du DevOps appelé **déploiement continu.**

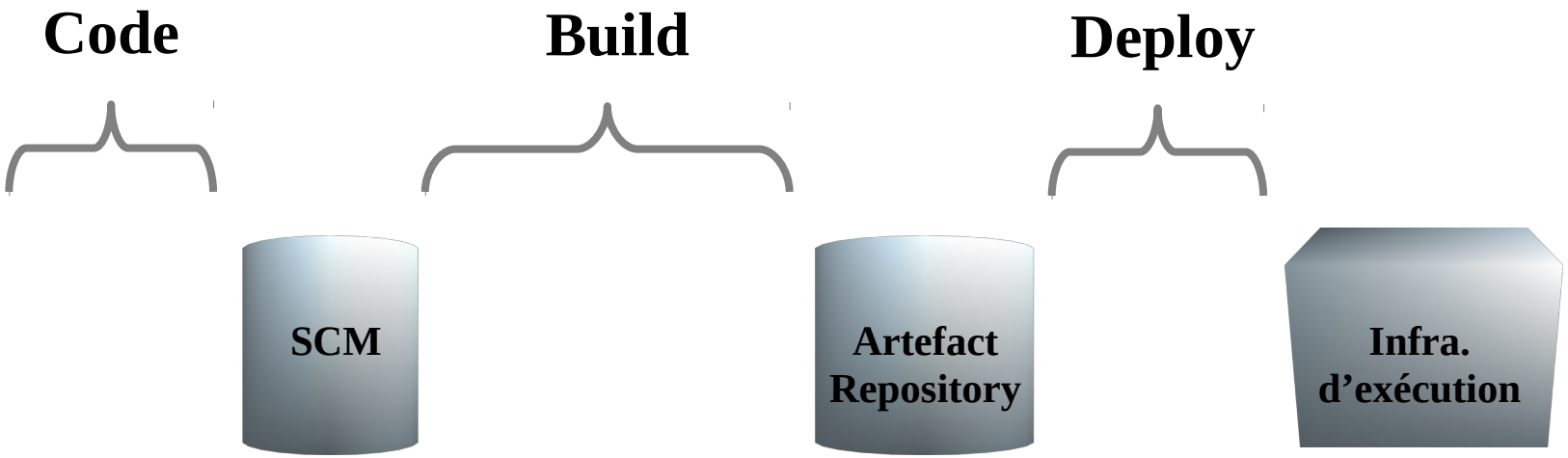
Cycle de vie du code et outils DevOps

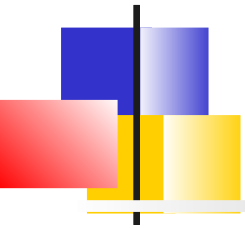
Cycle de vie du code

1) Le code est testée localement puis poussé dans le dépôt

2) Le build construit l'artefact et le stocke dans un dépôt

3) L'outil de déploiement récupère l'artefact et le déploie sur l'infra d'exécution





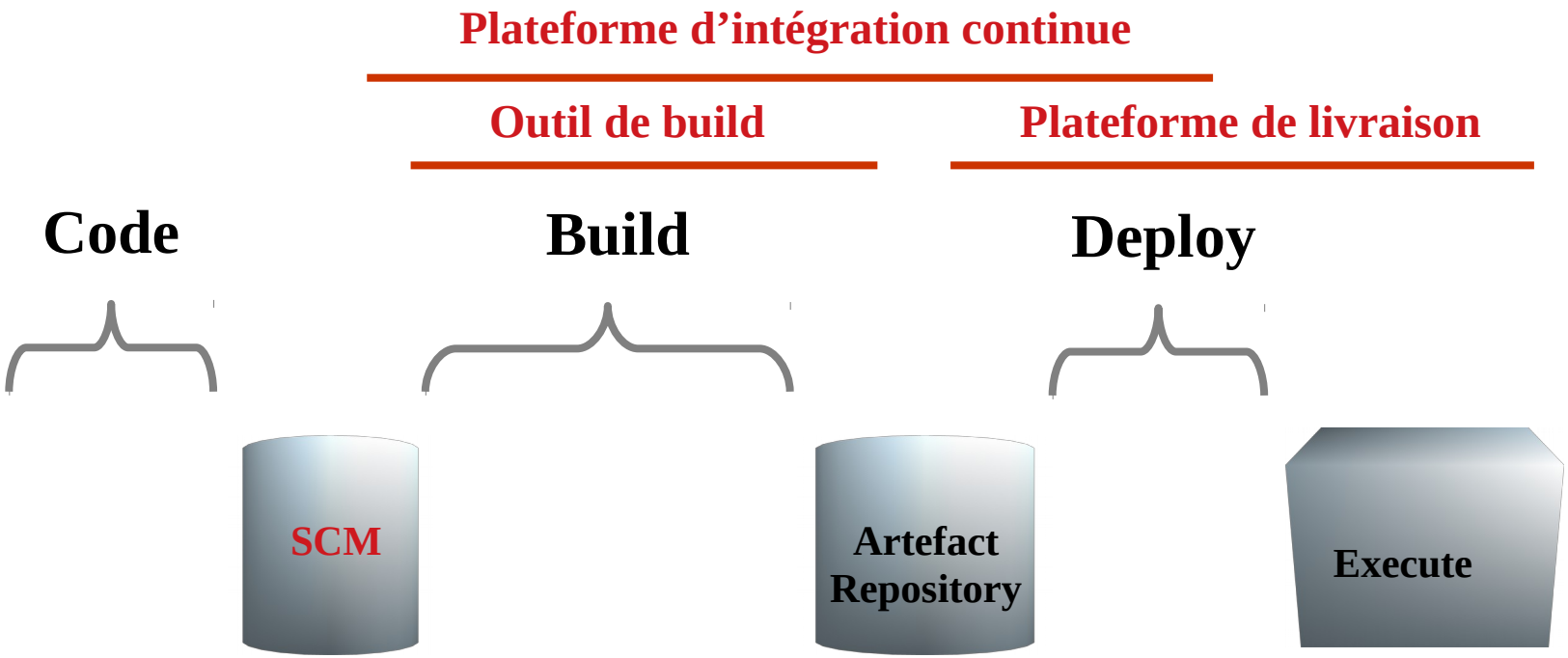
Types d’outils

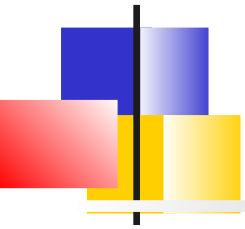
3 types d’outils s’intègrent dans le processus de CI/CD :

- Le **SCM** : centralise le code source (applicatif, code de build, des tests, scripts de provisionnement, …).
  Il gère plusieurs branches plus ou moins stables
- **L’outil de build** : exécute les étapes nécessaires à la production de l’artefact
- La **PIC** qui à partir des branches du SCM exécute des pipeline de construction différentes et décident des déploiement dans les dépôts
- La **plate-forme de livraison** permet de contrôler une version à livrer et provisionner les cibles de production

Certains outils sont parfois capables de gérer plusieurs aspects du processus (Ex : Gitlab CI)

Outils et Cycle de vie



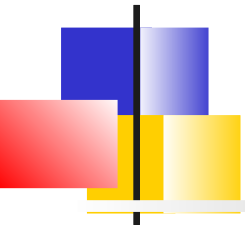


Dépôts et formats des artefacts

En fonction de la plateforme de livraison, différents types d'artefacts peuvent être générés par le build

- Code applicatif à déployer sur un serveur pré-provisionné.
  Ex : *Appli JavaEE déployé sur un serveur applicatif provisionné mutualisant des applications*
- Code applicatif + serveur embarqué
  Ex : *Application standalone déployé sur un serveur provisionné (OS + JVM par exemple)*
- Image d'un conteneur ou plusieurs images collaborant
  Ex : *Architecture Microservices déployé sur orchestrateur de conteneurs ou Cloud*

Certaines solutions ont comme vocation de gérer tous ces formats. D'autres sont spécialisés



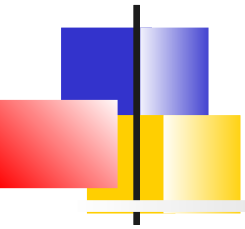
Release et Snapshots

2 types d'artefacts sont stockés :

- Les **SNAPSHOTS** : ce sont des versions non stables du logiciels qui n'ont pas vocation à être déployée en production.
  Par contre, d'autres projets dépendants peuvent les utilisés
- Les **releases** : ce sont des versions stables et bien testées qui sont déployables en production
  Les releases sont taggées Leur n° de version correspond à un tag des sources dans le scm

Généralement le processus de release consiste à :

- Tagger le dépôt des sources : SCM
- Produire un artefact et le stocker (à tout jamais) dans un repository d'artefact avec son n° de version



Infrastructure

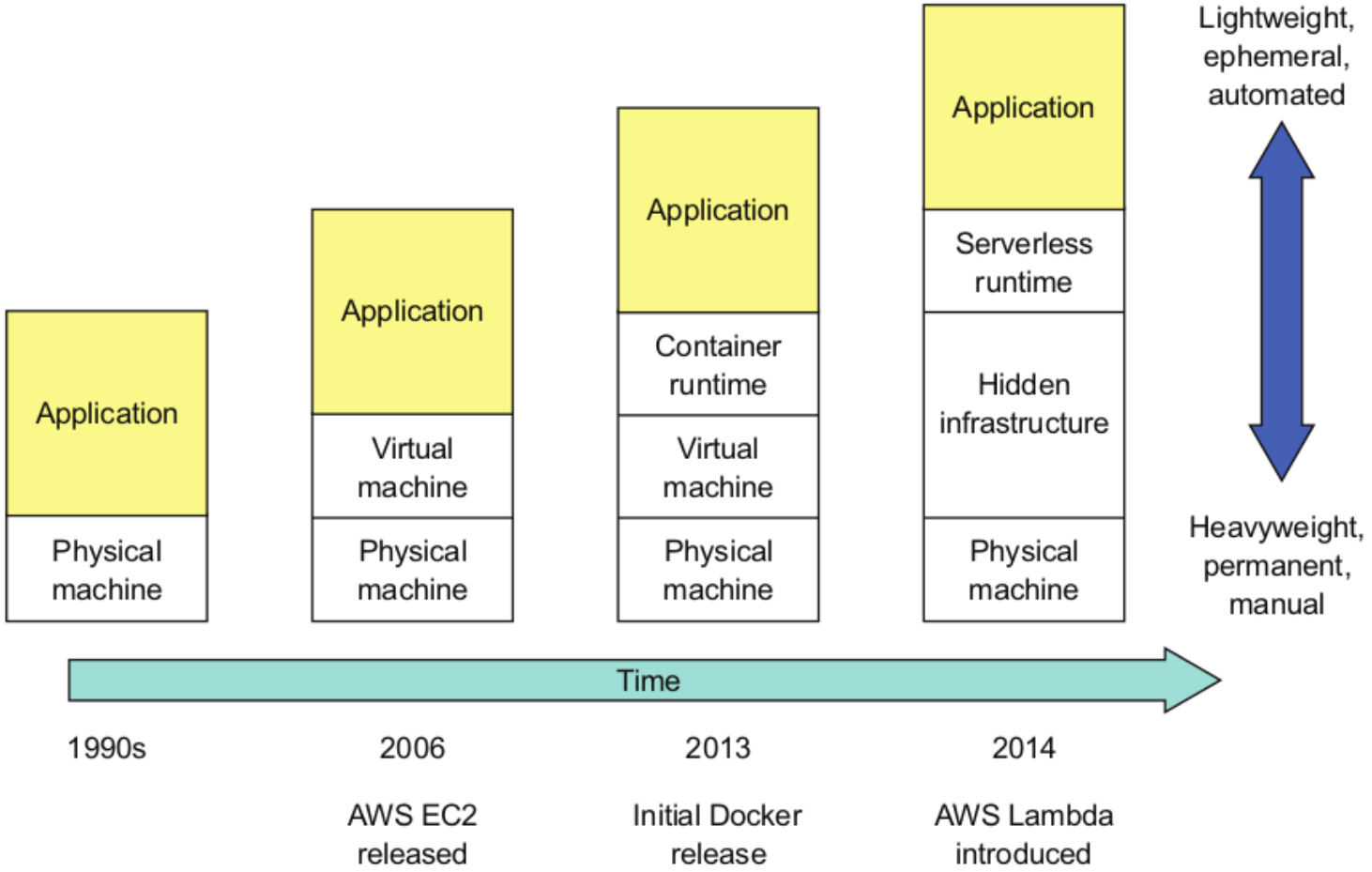
L’application produite nécessite une infrastructure d’exécution constituée de :

- **Matérielles** : Combien de CPU, RAM, Disque sont nécessaires pour l’application
- **Système d’exploitation** : Quelle est le système d’exploitation Cible
- **Middleware**, **produits, stack** : Quelles sont le middleware et les produits à installer ? Serveur applicatif, Base de données, …

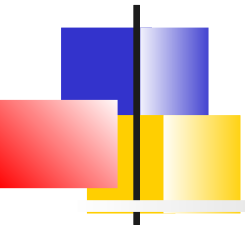
L’infrastructure est déclinée dans les différents environnements requis (intégration, QA, production)

Préparer l’infrastructure et les logiciels nécessaires s’appelle le **provisionnement**. Dans un contexte de CI/CD, il doit être également automatisé.

Evolution des infrastructures



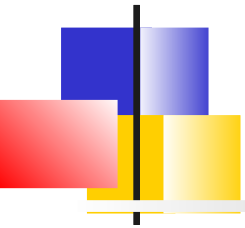
Influence de DevOps sur l’architecture des systèmes



Introduction

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « ***micro-services*** »

Lés méthodes agiles, le CI/CD, les technologies REST et les conteneurs facilitent la construction de grand systèmes orientés services

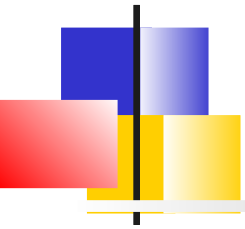


Architecture

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

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

On l'appelle également *SOA 2.0*



Caractéristiques

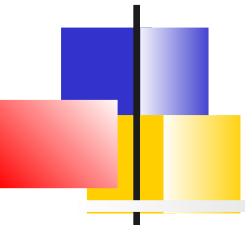
Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

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),

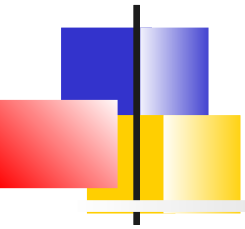
Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes

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 : Les fautes et dysfonctionnement sont plus faciles à identifier

Communication inter-équipe renforcée : Full-stack team



Contraintes

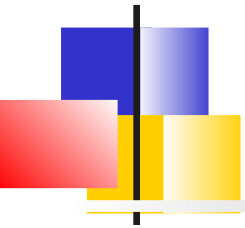
Réplication : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

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.



Les 12 facteurs de réussite

I. Outil de scm : Unique source de vérité

II. Dépendances: Déclare explicitement et isole les dépendances du code source

III. **Configuration** : Configuration séparée du code

IV. Services d'appui (backend) : Considère les services d'appui comme des ressources attachées, possibilité de switcher sans modification de code

V. Build, **release**, **run** : Permet la coexistence de différentes releases en production

VI. Processes : Exécute l'application comme un ou plusieurs processus stateless. Déploiement immuable

VII. Port binding : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP

VIII. Concurrence : Montée en charge grâce au modèle de processus

IX. Disposability : Renforce la robustesse avec des démarrages et arrêts rapides

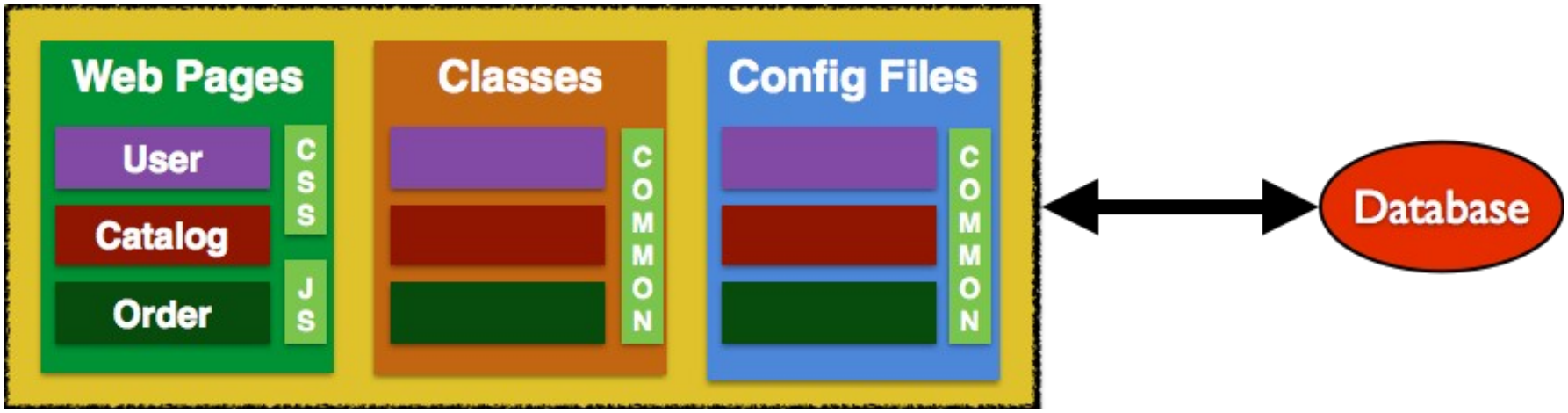
X. Dev/prod parity : Garder les environnements de développement, de pré-production et de production aussi similaires que possible

XI. Logs : Traiter les traces comme un flux d'événements

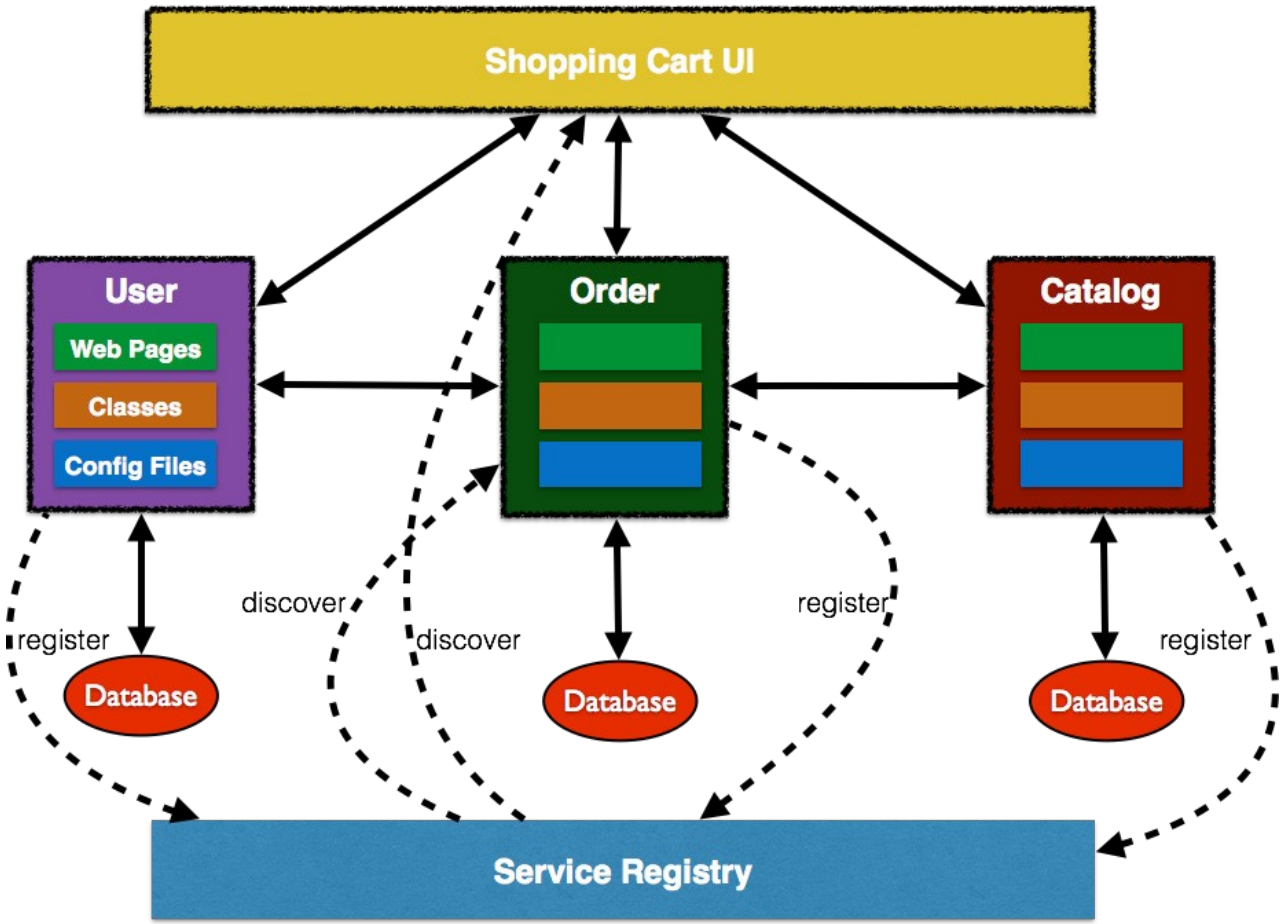
XII. Processus d'Admin : Considérer les tâches d'administration comme un processus parmi d'autres

https://12factor.net/

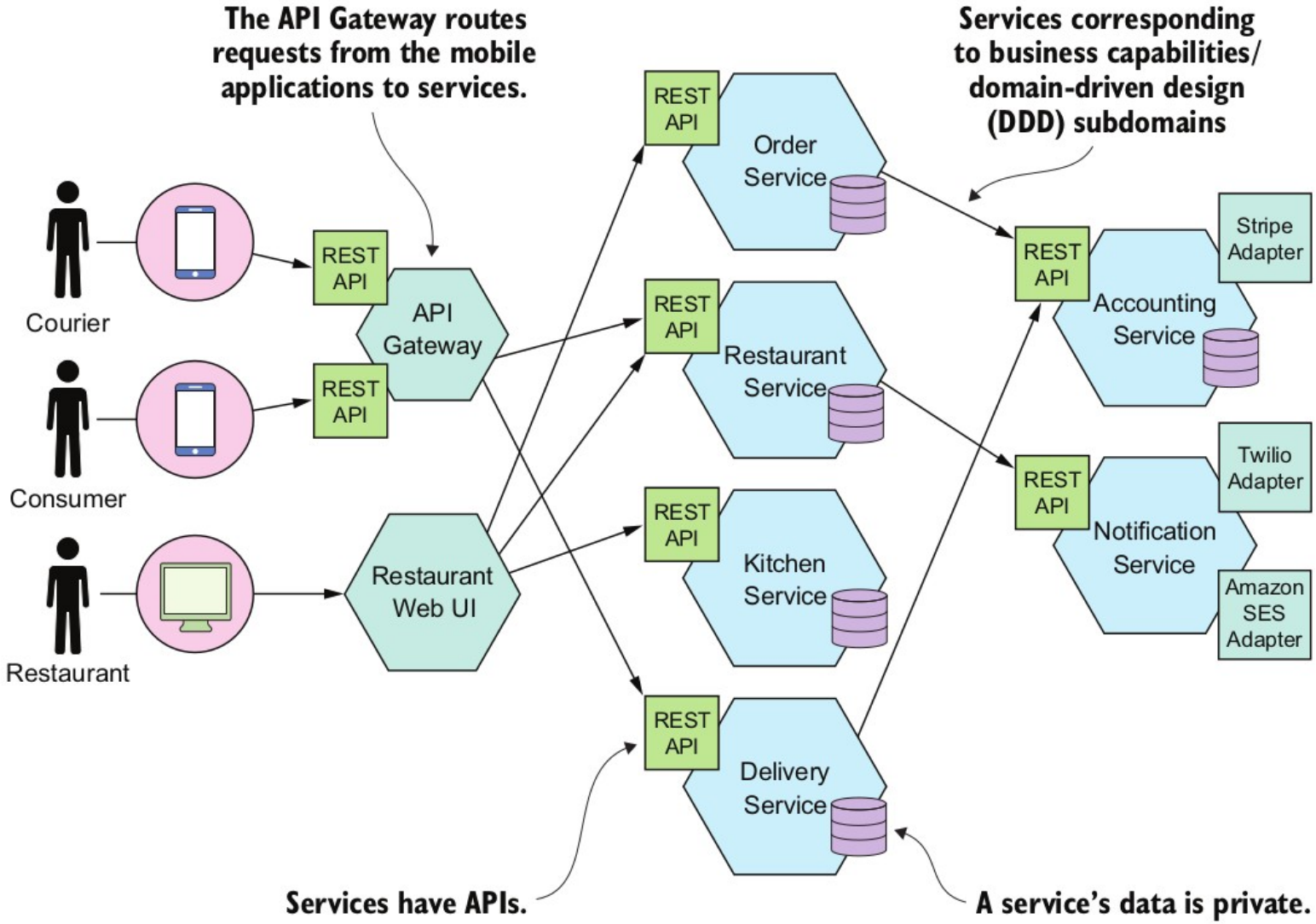
Architecture monolithique



Version micro-service

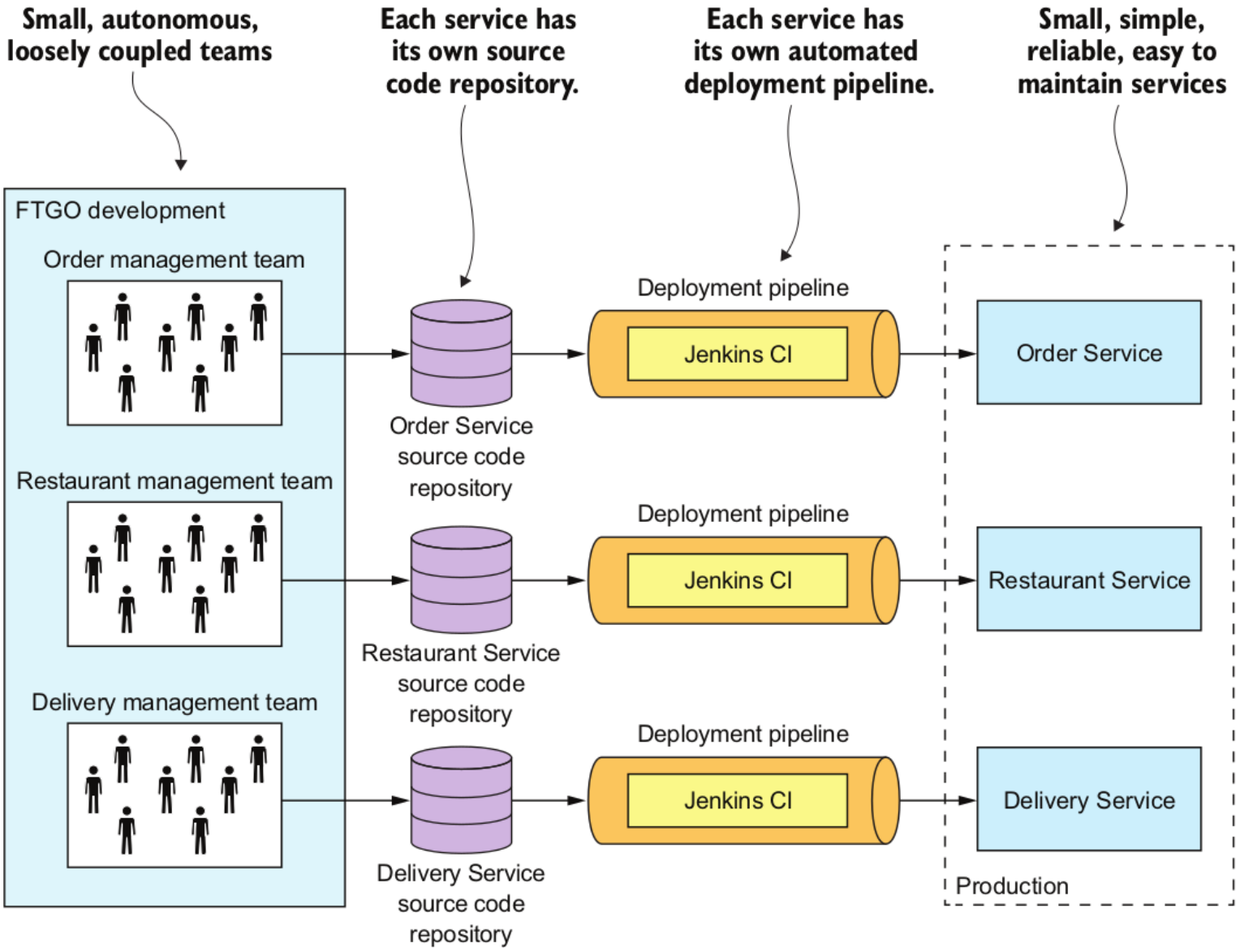


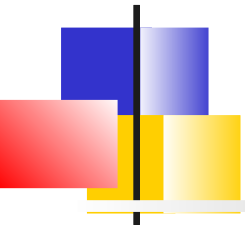
Une architecture micro-service



Microservices Pattern, Chris Richardson, 2017

Organisation DevOps

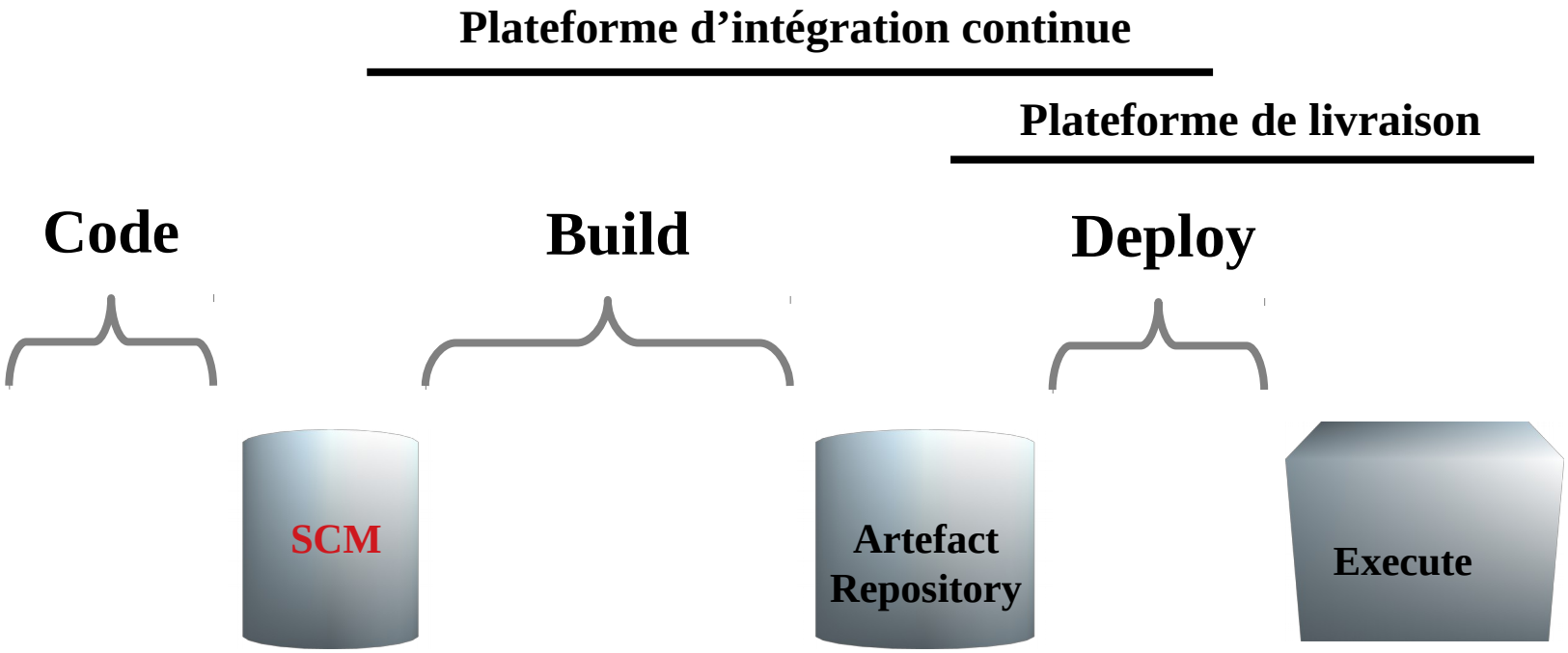




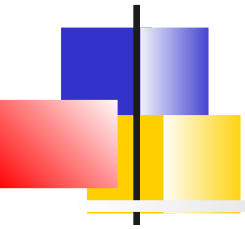
SCM

GIT : Un SCM distribué
Workflows de collaboration

SCMS et Cycle de vie



Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab
Solutions Cloud :
AWS, Google, Azure



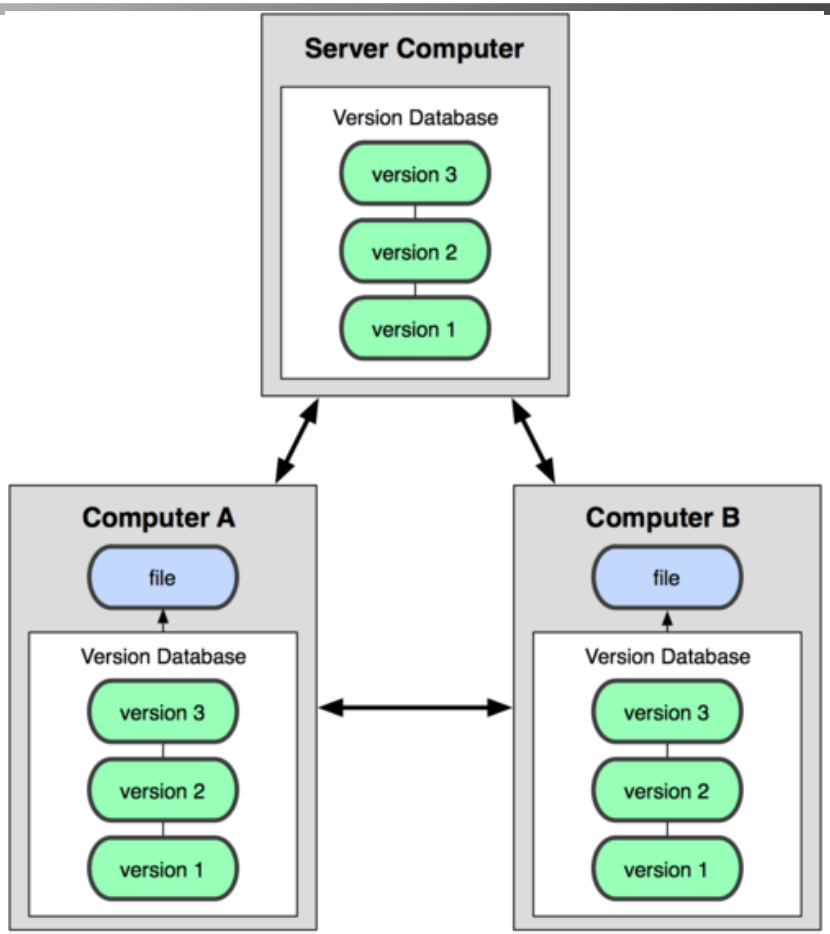
SCM

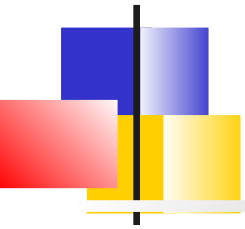
Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

Le système permet :

- De restaurer des fichiers
- Restaurer l'ensemble d'un projet
- Visualiser tous les changements effectués leurs auteurs et leurs commentaires associés
- Le développement concurrent (branche)

SCM distribués : Git, Bitbucket





Atouts de Git

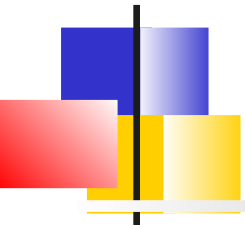
La plupart des opérations sont locales et donc très rapides

Les branches sont des pointeurs.

- La création et la suppression de branches sont des opérations légères.
- On peut avoir de nombreuses branches dans un projet

Extraire une version particulière est très rapide car Git n’a pas besoin de reconstruction particulière (application de patch)

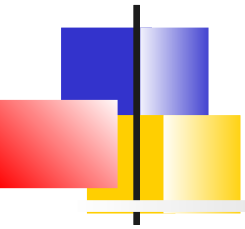
Les modes de collaboration sont variés



SCM : Commit, Branches et Tag

Les SCM contiennent l'historique complet des sources du projet

- Les **commits** (Ids) permettent d'isoler chaque modification apportée par les développeurs, les historiser et les documenter
- Les **branches** permettent à la branche principale de rester stable (et donc disponible au fonctionnel), lorsque les travaux engagés dans une branche de développement sont terminés, ils sont intégrés dans le tronc commun
- Les **tags**/**étiquettes** permettent de fixer les versions des sources. Ils correspondent en général à des release de l'application et servent à identifier les versions en production



Principales opérations

clone : Recopie intégrale du dépôt

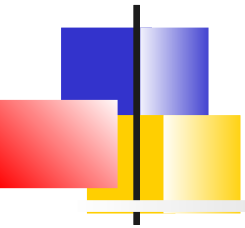
checkout : Extraction d'une révision particulière

commit : Enregistrement de modifications de source

push/***pull*** : Pousser/récupérer des modifications d'un dépôt distant

log : Accès à l'historique

merge, ***rebase*** : Intégration des modifications d'une branche dans une autre



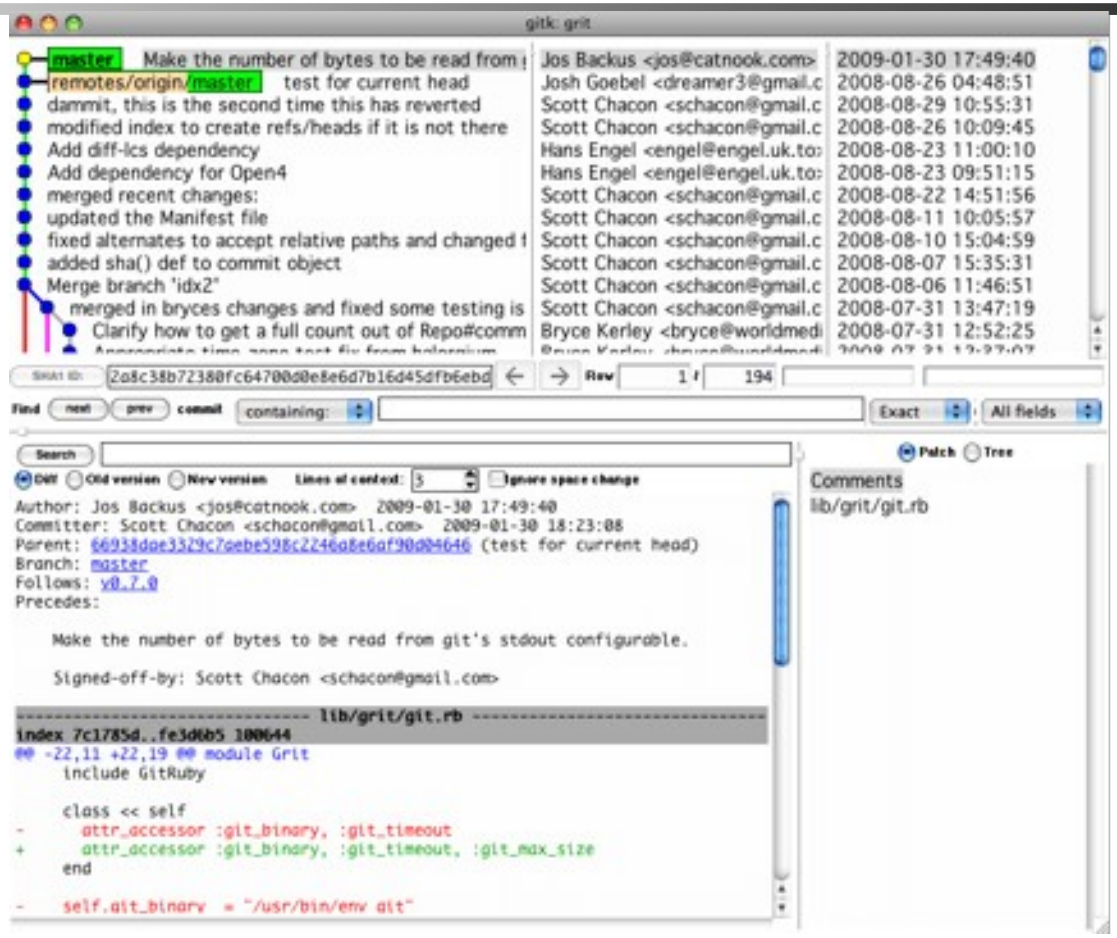
Patch

Chaque modification du code source committé peut être visualisé sous forme de ***patchs***

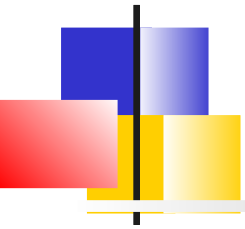
Un patch indique les blocs de lignes d’un fichier ayant été modifiés

```
@ -35,7 +35,7 @@ stage('Parallel Stage') {
 stage('Déploiement artefact') {
   steps {
     echo 'Deploying..'
-      sh './mvnw -Pprod clean deploy'
+      sh './mvnw --settings settings.xml -Pprod clean deploy'
       dir('target/') {
         stash includes: '*.jar', name: 'service'
       }
```

Historique *: gitk --all*



Workflows de collaboration et usage des branches

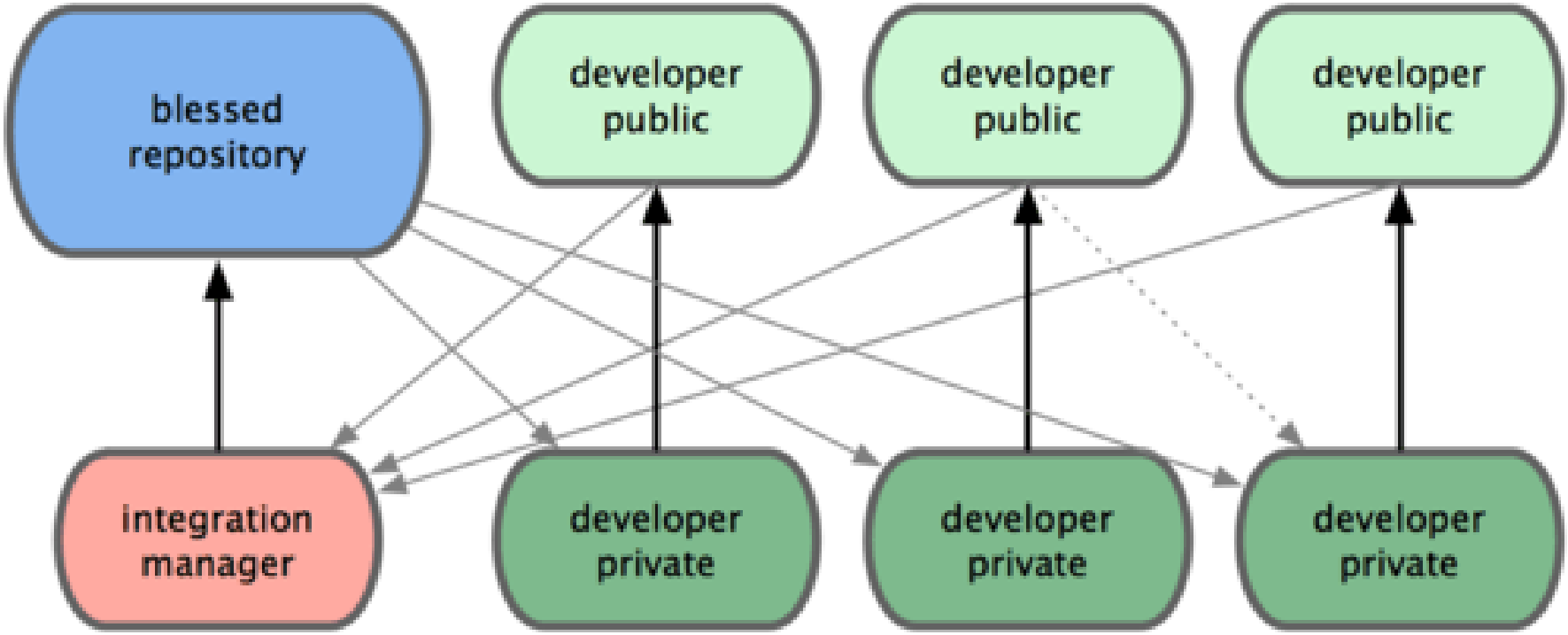


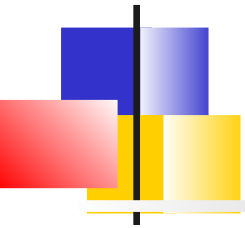
Introduction

Les SCMs distribués ont introduit différents workflows de collaboration entre développeurs :

- Projets OpenSource (Linux, Github, ...) : **Workflow avec intégrateur** basé sur les *pull-request*
- Editeur logiciel avec maintenance concurrente de plusieurs releases : **Gitflow**
- Projet DevOps avec déploiement continu : **GitlabFlow** basé sur les merge-request

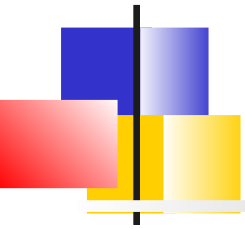
Gestionnaire d'intégration





Étapes

1. L'intégrateur pousse vers son dépôt public.
2. Un contributeur clone ce dépôt et introduit des modifications.
3. Le contributeur pousse son travail sur son dépôt public.
4. Le contributeur envoie à l'intégrateur un e-mail de demande pour tirer depuis son dépôt. (***pull-request***)
5. Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne localement.
6. Si cela lui paraît adéquat, le mainteneur pousse les modifications fusionnées sur le dépôt principal



Gitflow

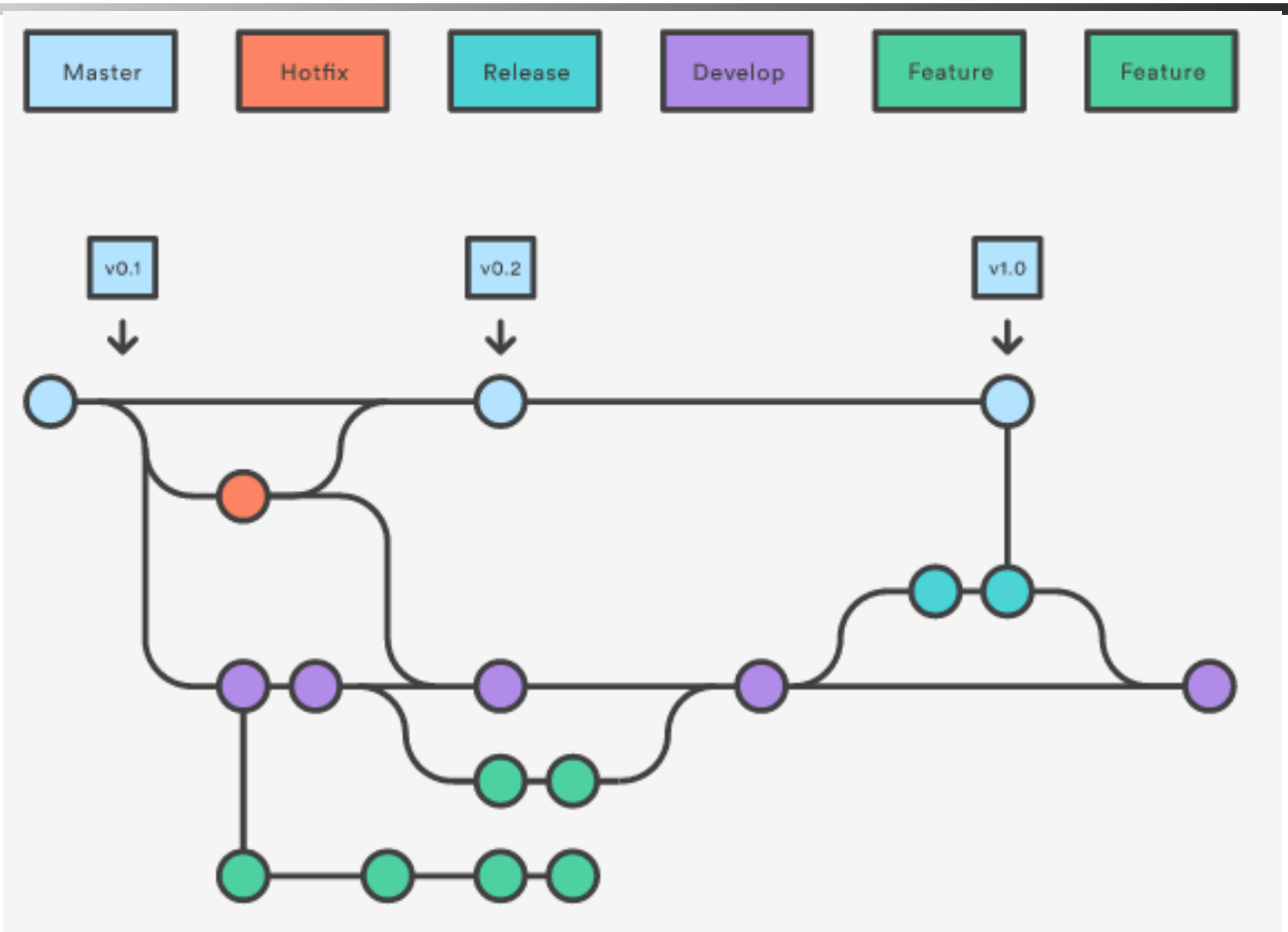
Le workflow **Gitflow** définit un modèle de branches orientées vers la production de releases maintenables

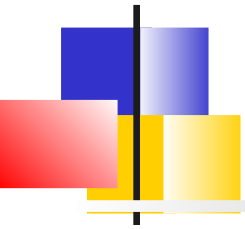
- Adapté pour projets d'édition logiciel
- Des rôles très spécifiques sont assignés aux différentes branches et Gitflow définit quand et comment elles doivent interagir

Il utilise des :

- Des branches **longues** (master, dev) qui existent pendant tout le projet
- Des branches **courtes** qui sont supprimées dés lors qu'elles ont atteint leur but

Branches Gitflow

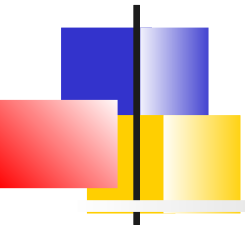




Déclinaisons

On peut décliner *Gitflow* avec d'autres branches annexes comme des branches de revue de code

- Elles permettent de valider des modifications avant de les intégrer dans une branche supérieure. Exemple de *Gerrit*

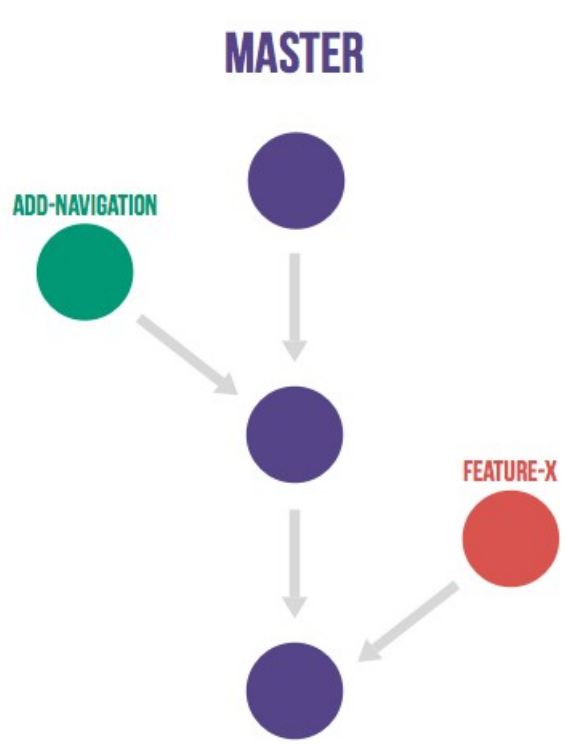


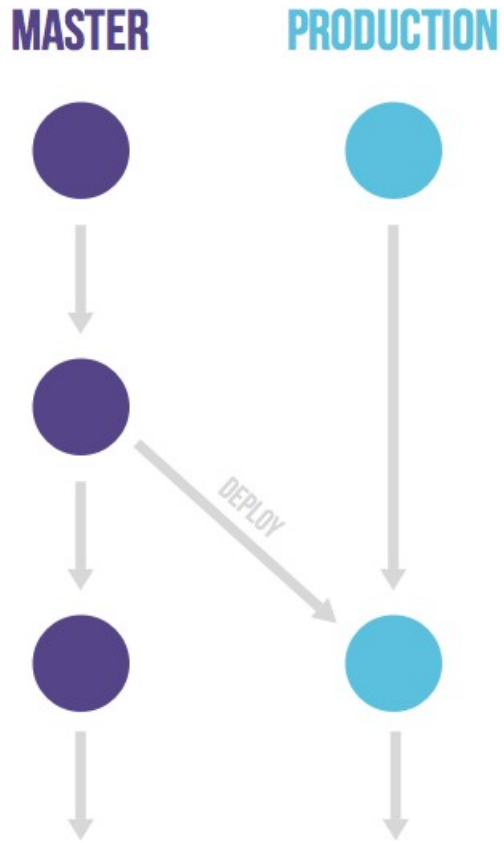
Gitlab Flow

Gitlab Flow est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

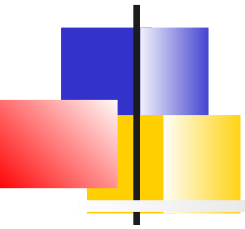
1) Les corrections ou fonctionnalités sont développées dans une *feature branch*
2) Lorsqu'elle sont prêtes, un *merge request* est émis : Demande de fusion dans la branche principale
3) Des tests sont effectués en isolation. Si ils sont probant, les modifications sont fusionnées dans la branche principale
4) Les modifications intégrées dans la branche principale sont potentiellement livrables en production

Features, Master and Production





Démo Merge Request



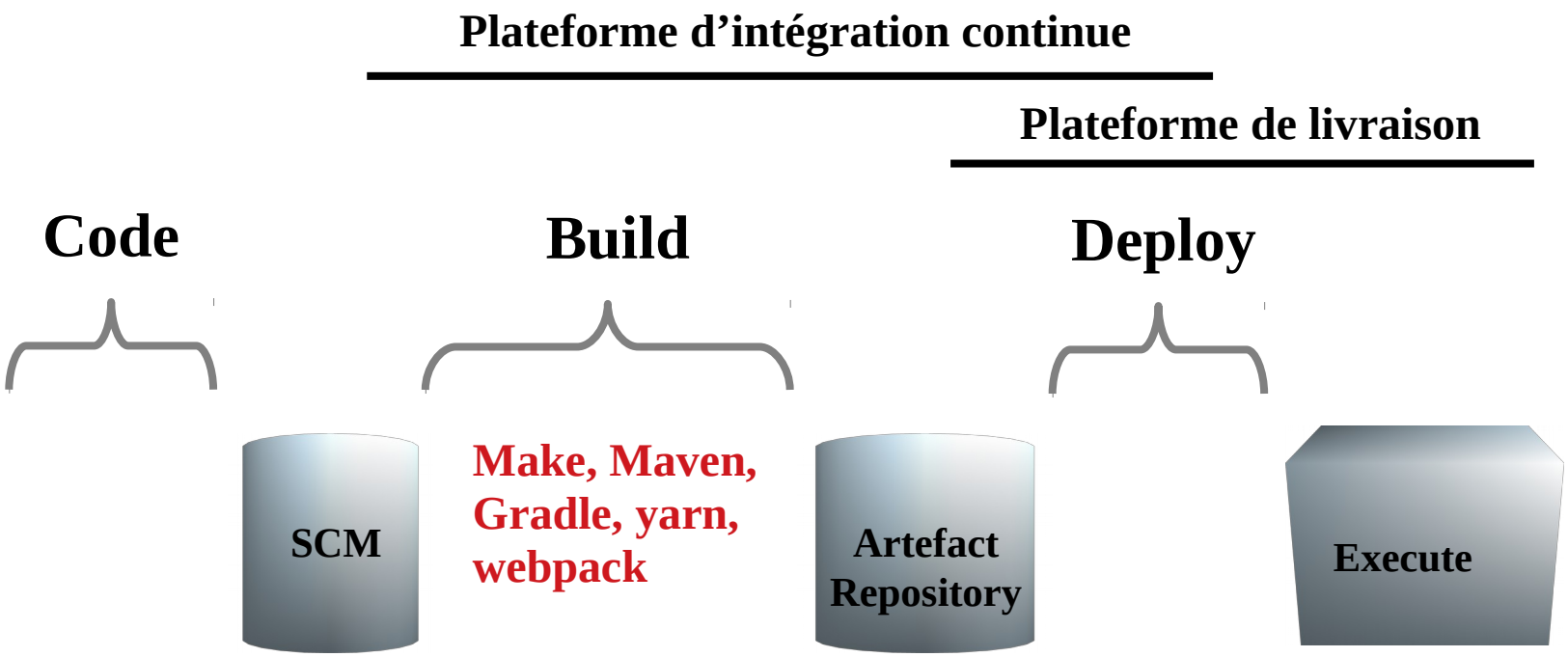
Outils de build

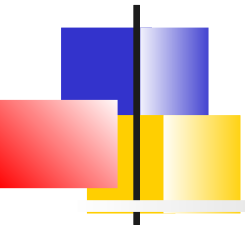
Caractéristiques d’un outil de build

Maven

Les tests et la qualité

Les outils de build dans le cycle de vie





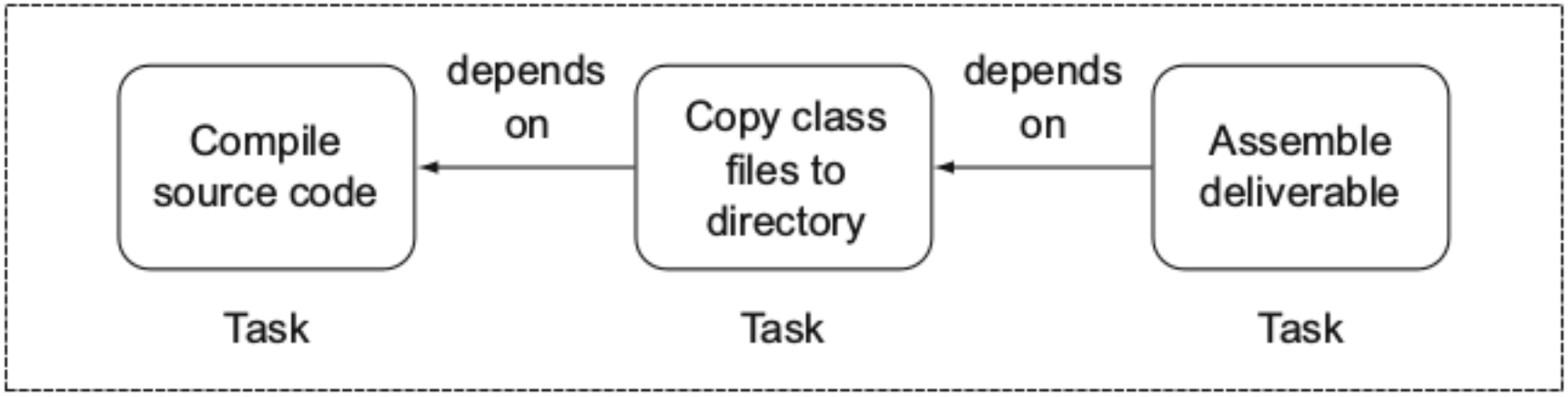
Pré-requis d’un build

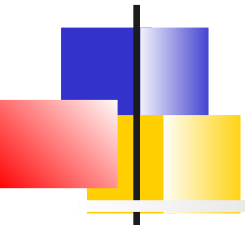
- **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- **Sûr** : Confiance dans son exécution

Graphe de build

Un build est une séquence ordonnée de tâches unitaires.

Les tâches ont des dépendances entre elles qui peuvent être modélisées via un **graphe acyclique dirigé** :





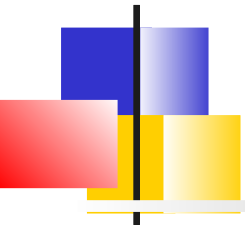
Composants d’un outil de build

Le fichier de build : Contient la configuration requises pour le build, les dépendances externes, les instructions pour exécuter un objectif sous forme de tâches inter-dépendantes

Une tâche prend une entrée effectue des traitements et produit une sortie. Une tâche dépendante prend comme entrée la sortie d’une autre tâche

Moteur de build : Le moteur traite le fichier de build et construit sa représentation interne. Des outils permettent d’accéder à ce modèle via une API

Gestionnaire de dépendances : Traite les déclarations de dépendances et les résout par rapport à un dépôt d’artefact contenant des méta-données permettant de trouver les dépendances transitive

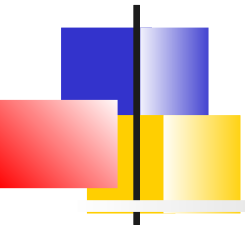


Monde Java

Apache Ant : L'ancêtre. Pas de gestionnaire de dépendances. Plein de tâches prédéfinies. Facilité d'extension. Pas de convention

Maven : L'actuel. Gestionnaire de dépendances. Convention plutôt que configuration. Extension par mécanisme de plugin. Verbeux car XML

Gradle : Futur ? Gestionnaire de dépendances. Allie les qualités de Maven et des capacités de codage du build. Concis

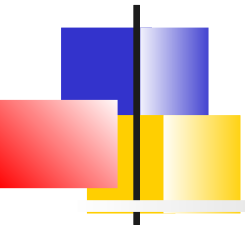


Outils de build

Caractéristiques d’un outil de build

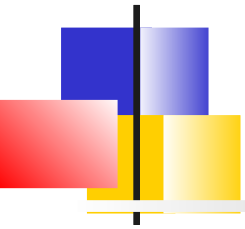
Maven

Les tests et la qualité



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

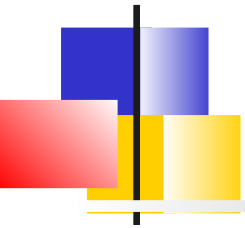


Qu’est ce que Maven

- Définition officielle (Apache) : Outil de gestion de projet qui fournit :
  - un **modèle** de projet
  - un ensemble de **standards**
  - un **cycle de vie (ou de construction)** de projet
  - un système de **gestion de dépendances**
  - de la logique pour exécuter des **objectifs** via des **plugins** à des **phases** bien précises du cycle de vie/construction du projet

- Pour utiliser Maven, on doit décrire son projet en utilisant un modèle bien défini. (*pom.xml*)
- Ensuite, Maven peut appliquer de la logique transverse grâce à un ensemble de plugins partagés par la communauté Java (pouvant être adaptés)

Convention plutôt que Configuration

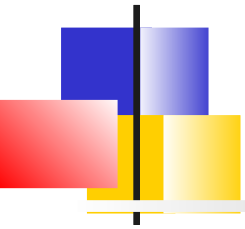


- Concept permettant d'alléger la configuration en respectant des conventions
- Par exemple, Maven présuppose une organisation du projet dont le but est de produire un jar:
  - ***${basedir}/src/main/java*** contient le code source
  - ***${basedir}/src/main/resources*** contient les ressources
  - ***${basedir}/src/test*** contient les tests
  - ***${basedir}/target/classes*** les classes compilées
  - ***${basedir}/target*** le jar à distribuer

- Les plugins cœur de Maven se basent sur cette structure pour compiler, packager, générer des sites webs, etc..

Modèle conceptuel d’un projet

- Maven maintient un modèle de projet. Les développeurs doivent le renseigner :
  - Coordonnées permettant d’identifier le projet
  - Mode de licence
  - Développeurs et contributeurs
  - Dépendances vis-à-vis d’autres projets
  - …
- Ce « Project Object Model » est décrit dans un fichier XML unique :
  pom.xml

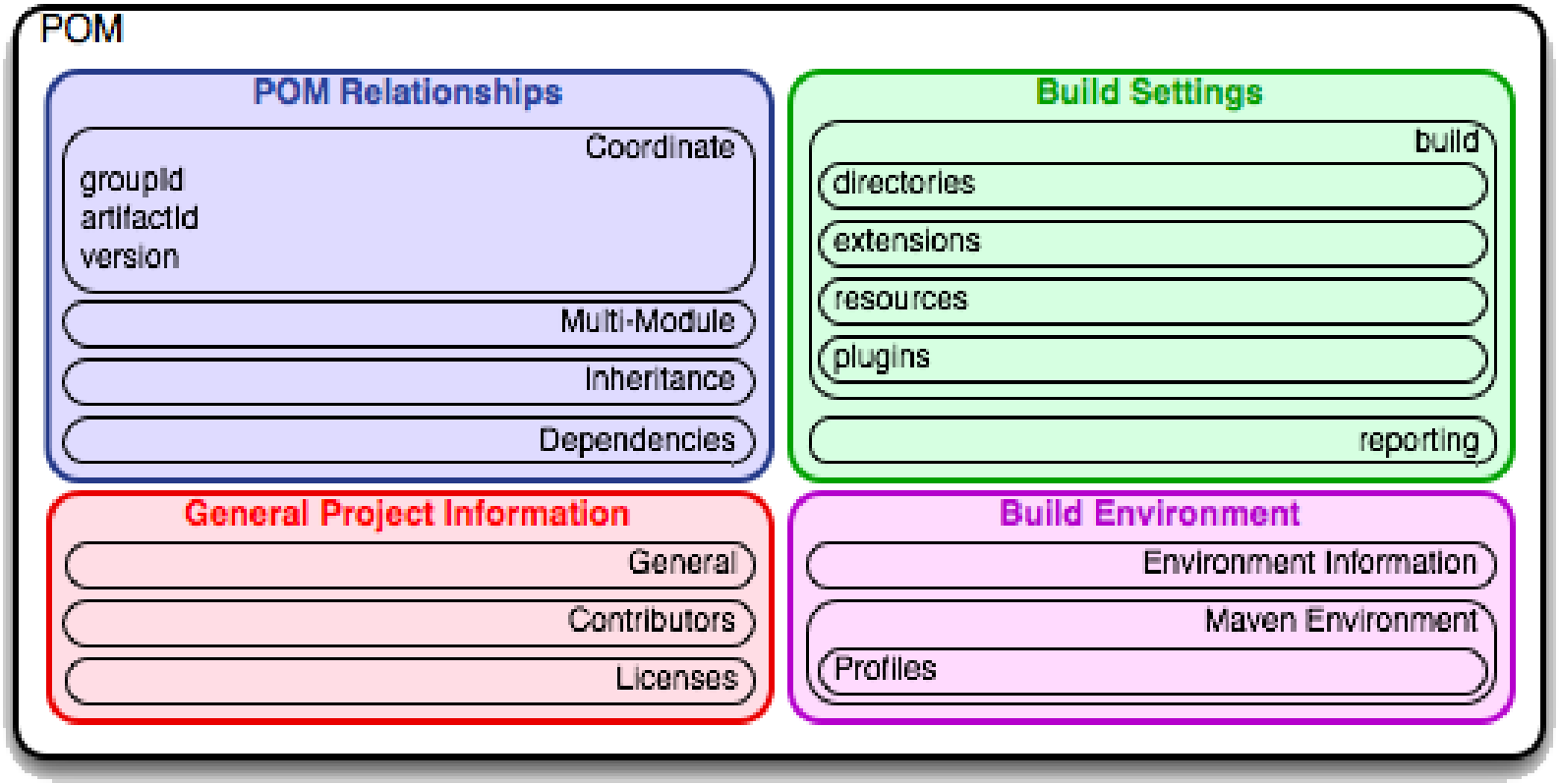


Contenu du POM

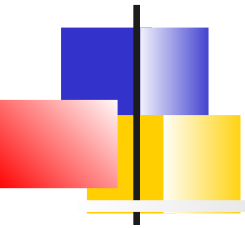
Le POM contient 4 types de description :

- **Relations entre POM** : Identification Maven, Packaging, relation d'héritage, définition de sous module, dépendances vers librairies tierces ou autres projet, dépôts utilisés etc..
- **Information générale du projet** : URL, développeurs, contributeurs, licence, ...
- **Configuration du build** : Personnalisation du build par défaut : configuration des plugins, attachement d'objectifs aux phases
- **Profils** : Surcharge des dépendances et du build lorsque des profils sont activés

Contenu du POM

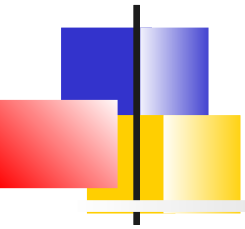


Exemple *pom.xml*



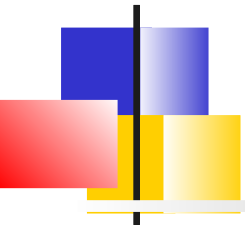
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch03</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Coordonnées Maven



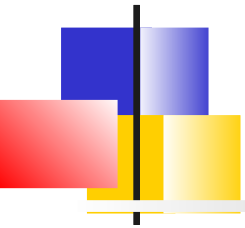
- Le fichier POM nomme le projet, fournit un ensemble d'identifiants unique : les **coordonnées**, et définit les relations entre ce projet et les autres via les dépendances, les parents et les pré-requis.
- Les coordonnées Maven sont composés de :
  - ***groupId***
  - ***artifactId***
  - ***version***
  - ***packaging***
- Ces coordonnées identifient de façon unique un projet dans l'espace Maven. Elles sont utilisées pour exprimer une dépendance, identifier un plugin ou un projet parent
- Le format est alors : ***groupId:artifactId:packaging:version***
- Exemples :
  - *mavenbook:my-app:jar:1.0-SNAPSHOT*.
  - *junit:junit:jar:3.8.1*.

Coordonnées Maven



- Le fichier POM identifie le projet via ses **coordonnées** Maven. Les coordonnées sont également utilisées pour identifier les dépendances, les parents, les plugins

- Les coordonnées sont composées de :
  - ***groupId*** : Le groupe, la société. La convention de nommage stipule qu'il commence par le nom de domaine de l'organisation qui a créé le projet (*com.plb* ou *org.apache*, ..)
  - ***artifactId*** ; Un identifiant unique à l'intérieur de *groupId* qui identifie un projet unique
  - ***version*** : Un n° de version. Les projets en cours de développement utilise un suffixe spécial : ***SNAPSHOT***.



Propriétés

Un POM peut inclure des propriétés qui sont évaluées à l'exécution :

- La syntaxe d'utilisation est ***${myProp}***

Maven fournit des propriétés implicites :

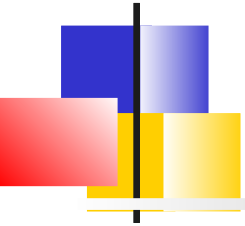
- ***env*** : Variables d'environnement système
- ***project*** : Le projet
- ***settings*** : Accès au configuration de *setting.xml*

Des propriétés peuvent être définis dans le *pom.xml*

```
<properties> <foo>bar</foo> </properties>
```

Elles peuvent être surchargées en ligne de commande en positionnant une propriété JVM : ***-D***

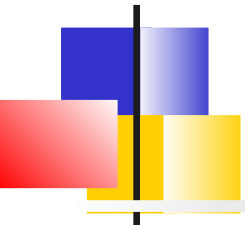
Gestion des dépendances



- La possibilité de localiser un artefact à partir de ces coordonnées permet d’indiquer les dépendances dans le projet POM.

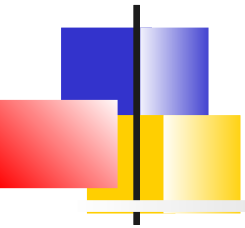
- Le fait que le POM associé à tout artefact soit stocké dans le dépôt permet à Maven de résoudre les dépendances transitives.
  => Si par exemple, votre projet dépend d'une librairie B qui dépend elle-même d'une librairie C, il n'est pas nécessaire d'indiquer la dépendance sur C
- L'ajout de dépendance est réalisé en ajoutant une balise ***<dependency>*** sous la balise ***<dependencies>*** et en indiquant les coordonnées Maven

Exemple



```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
</dependencies>
```

Périmètres

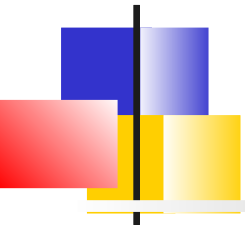


La balise <*dependency*> peut spécifier un périmètre de la dépendance via la balise ***scope***.

5 périmètres sont disponibles :

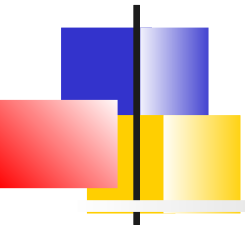
- ***compile*** : Le périmètre par défaut, les dépendances sont tout le temps dans le classpath et sont packagées
- ***provided*** : Dépendance fournie par le container. Présent dans le classpath de compilation mais pas packagée
- ***runtime*** : Présent à l'exécution et lors des tests mais pas à la compilation Ex : Une implémentation JDBC
- ***test*** : Disponible seulement pour les tests
- ***system*** : Fournie par le système (pas recherché dans un repository)

Exemple



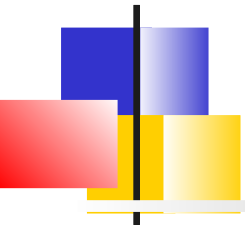
```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-io</artifactId>
  <version>1.3.2</version>
  <scope>test</scope>
</dependency>
```

Exclusion de dépendances transitives

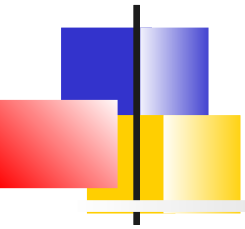


```
<dependencies>
 <dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate</artifactId>
   <version>3.2.5.ga</version>
   <exclusions>
     <exclusion>
       <groupId>javax.transaction</groupId>
       <artifactId>jta</artifactId>
     </exclusion>
   </exclusions>
 </dependency>
 <dependency>
   <groupId>org.apache.geronimo.specs</groupId>
   <artifactId>geronimo-jta_1.1_spec</artifactId>
   <version>1.1</version>
 </dependency>
</dependencies>
```

Ou trouver les dépendances ?



- Le site http://www.mvnrepository.com permet de retrouver les *groupId* et les *artifactId* nécessaires pour indiquer les dépendances
  - Il fournit une interface de recherche vers le dépôt public Maven.
  - Lorsque l'on recherche un artefact, il liste l'*artifactId* et toutes ses versions connues.
  - En cliquant sur une version, on peut récupérer l'élément <*dependency*> à inclure dans son POM



Maven

Concepts généraux et *pom.xml*

Commandes Maven

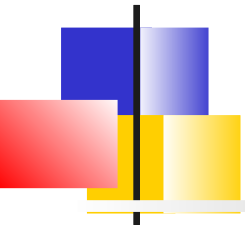
Dépôts Maven

POMs parents

Gestions de versions

Adaptation du build

Profils



Commandes Maven

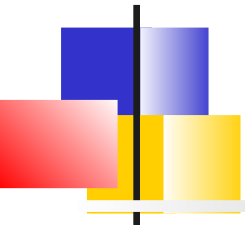
Les commandes Maven sont exécutées via :

mvn <plugins:goals> <phase>

La commande peut donc spécifier :

- L'exécution d'un **objectif** d'un plugin particulier
- L'exécution d'une **phase** … provoquant l'exécution de plusieurs objectifs de différents plugins

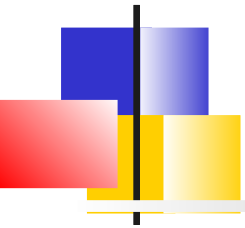
Plugins et objectifs



- Un **plugin** Maven est composé d'un ensemble de tâches atomiques : les **objectifs** (goals)
- Exemples :
  - le plugin ***Jar*** contient des objectifs pour créer des fichiers jars
  - le plugin ***Compiler*** contient des objectifs pour compiler le code source
  - le plugin ***Surefire*** contient des objectifs pour exécuter les tests unitaires et générer des rapports de tests.
  - D'autres plugins plus spécialisés comme le plugin *Hibernate3*, le plugin *Jruby, ...*

Maven fournit également la possibilité de définir ses propres plugins. Un plugin spécifique peut-être écrit en Java, ou Ant, Groovy, beanshell, ...

Maven Plugins et objectifs



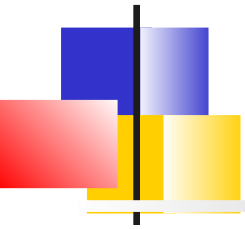
- Un objectif est une tâche spécifique qui peut être exécutée en standalone ou comme faisant partie d'un build plus large.

- La syntaxe d'exécution est la suivante :
  `mvn <plugin>:<objectif>`

- Un **objectif est l'unité de travail** dans Maven
  Exemples : l'objectif de compilation dans le compilateur plugin

- Les objectifs peuvent être configurés via un certain nombre de propriétés. Par exemple : La version du JDK est un paramètre de l'objectif de compilation.

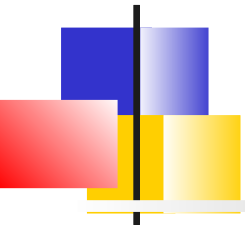
Exemple



Exemple : appel de l'objectif *generate* du plugin *archetype*

```
$ mvn archetype:generate
  -DgroupId=org.formation.simpleProject \
-DartifactId=simpleProject \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.4 \
-DinteractiveMode=false
```

Cycle de vie Maven

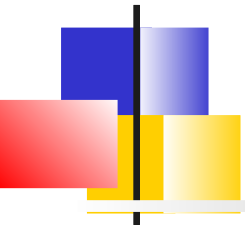


- Le cycle de vie est une **séquence ordonnée de phases** impliquées dans la construction d'un projet

- Maven supporte différents cycles de vie en fonction du type de packaging.
- Le cycle de vie par défaut correspondant à un packaging *jar* est le plus souvent utilisé, il commence avec une phase validant l'intégrité du projet et termine avec une phase déployant le projet dans un dépôt

- Les phases du cycle de vie sont intentionnellement « vagues » : *validation*, *test* ou *déploiement*
  Elles peuvent signifier différentes choses en fonction des projets.

Phase et objectifs

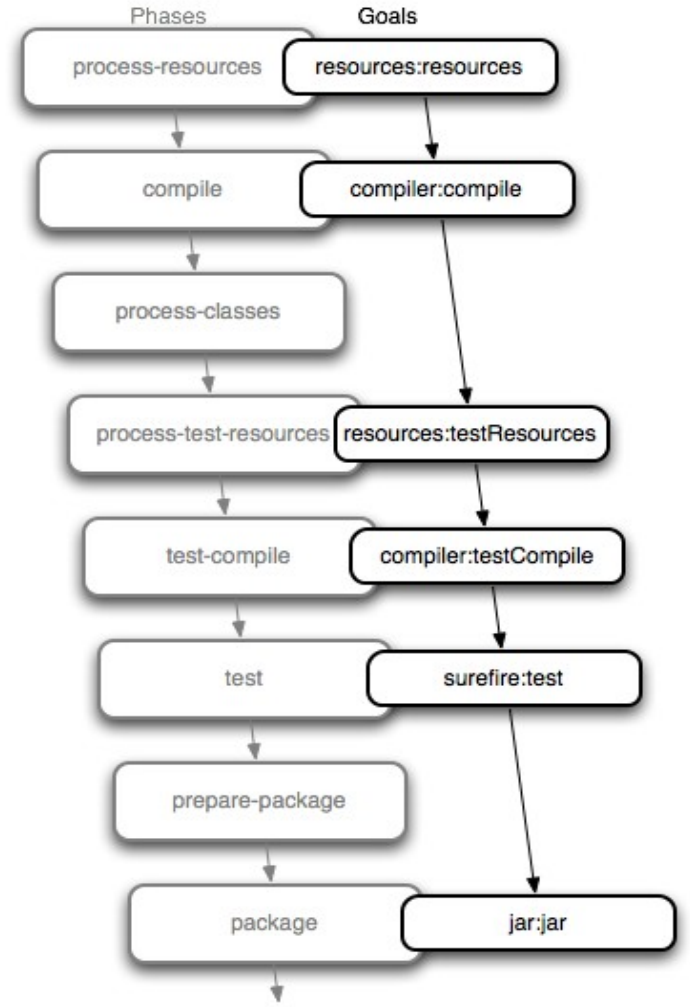


- Les objectifs d’un plugin peuvent être **attachés** à une phase du cycle de vie. Lorsque Maven atteindra une phase dans un cycle de vie, il exécutera les objectifs qui lui sont attachés.

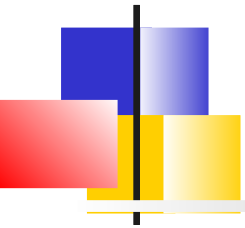
- Par exemple, lors de l’exécution de **mvn install** *install* représente la phase et son exécution provoquera l'exécution de plusieurs objectifs

- L’exécution d’une phase exécute également toutes les phases précédentes dans l’ordre défini par le cycle de vie.

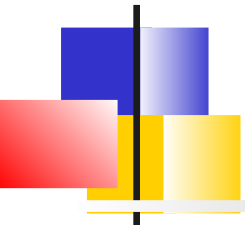
Cycle de vie et objectifs par défaut



Objectifs par défaut pour un packaging jar



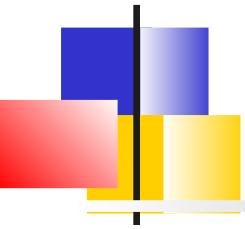
- ***process-resources / resources:resources*** : Copie tous les fichiers ressources de *src/main/resources* dans le répertoire de production
- ***compile / compiler:compile*** : compile tout le code source présent dans *src/main/java* vers le répertoire cible.
- ***process-test-resources / resources:testResources*** : copie toutes les ressources du répertoire *src/test/resources* vers le répertoire de sortie pour les tests
- ***test-compile / compiler:testCompile*** : compile les cas de test présent dans *src/test/java* vers le répertoire de sortie des tests
- ***test / surefire:test*** : exécute tous les tests et créé les fichiers résultats, termine le build en cas d'échec
- ***package / jar:jar*** : crée un fichier jar à partir du répertoire de sortie



Autres types de packaging

D’autres types de packaging existent, elle donne lieu alors à des cycles de vie différents

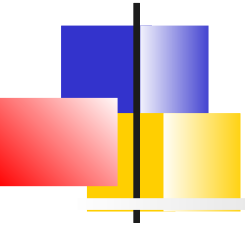
- ***pom*** : Le but est de construire un pom.xml
- ***war*** : Le but est de construire un war
- …



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

Repositories/Dépôts Maven



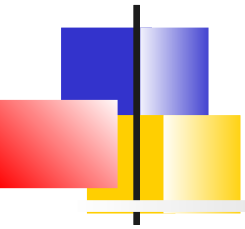
- Un **repository ou dépôt** désigne l'espace de stockage des artefacts construits via Maven (plugins, librairies, jar projet)

- Il peut être :
  - Distant
  - Local

- Maven est livré avec le minimum et récupère à partir d'un dépôt distant ce dont il a besoin. Il utilise par défaut :
  http://repo.maven.apache.org/maven2/

- Des dépôts spécifiques peuvent être configurés via la balise ***<repositories/>***

Structure des dépôts



- La structure arborescente des dépôts correspond aux coordonnées Maven

- Les artefacts sont donc stockés dans des répertoires :

/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>

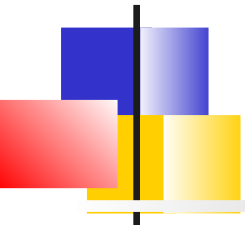
- A côté de l'artefact, on retrouve le pom l'ayant généré, et d'autres méta-données

- L'emplacement standard du dépôt local est :

  ~/.m2/repository/

- Il peut être redéfini dans le fichier central de configuration Maven :

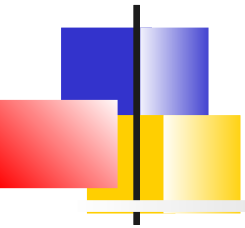
  ~/.m2/settings.xml



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

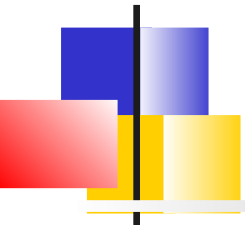
POM effectif



- Il peut exister des relations d'héritages entre les POMs :
  - POM parent utilisé pour mutualiser des configurations communes
  - POM d'un projet multi-modules
- Tous les POM héritent du POM racine dénommé Super POM
- Le fichier *pom.xml* effectif utilisé pour l'exécution est donc en fait une combinaison du fichier *pom* du projet et de tous les fichiers POMs des projets parents dont le POM racine de Maven
- Afin de consulter le fichier réellement utilisé par Maven, il suffit d'utiliser le plugin d'aide :

```
$ mvn help:effective-pom
```

POM Parent



La référence vers le POM parent est indiqué via la balise ***<parent>***

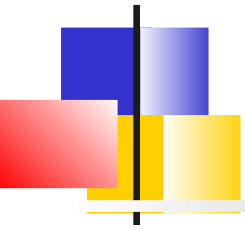
```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
</parent>
```

Maven recherche alors le *pom.xml* correspondant dans le dépôt local.

On peut également indiquer l'élément *<relativePath/>* si le pom parent n'a pas été installé dans le dépôt

```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
  <relativePath>../parent-project/pom.xml</relativePath>
</parent>
```

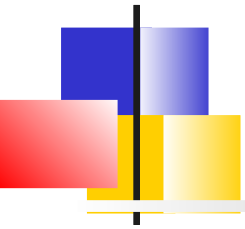
Super POM



Le fichier Super POM fait partie intégrante de l'installation de Maven et est inclus dans le jar *maven-model-builder-x.x.jar*

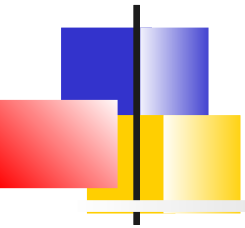
- Il définit entre autres :
  - Le dépôt par défaut Maven.
  - Les plugins de ce dépôt
  - La structure de répertoire par défaut
  - Les versions par défaut des plugins cœur

Projets multi-modules



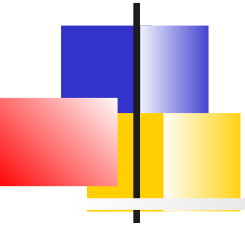
- Un projet multi-modules est défini par un POM qui référence plusieurs sous-modules :
- La définition s'effectue via les balises ***<modules>*** et **<module>**

POM multi-modules



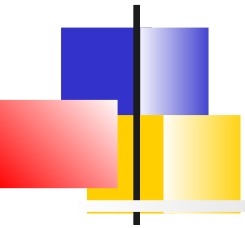
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>
  <modules>
    <module>simple-model</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>
  </project>
```

POM d'un projet multi-modules



- Le POM parent définit les coordonnées Maven
- Il ne créée pas de JAR ni de WAR, il consiste seulement à référencer les autres projets Maven, le packaging a alors la valeur ***pom***.
- Dans l'élément **<modules>**, les sous-modules correspondant à des sous-répertoires sont listées
- Dans chaque sous-répertoire, Maven pourra trouver les fichiers *pom.xml* et inclura ces sous-modules dans le *build*
- Enfin, le POM parent peut contenir des configurations qui seront héritées par les sous-modules

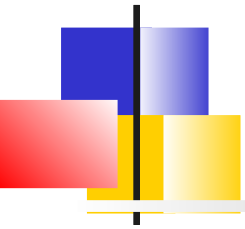
POM des sous-modules



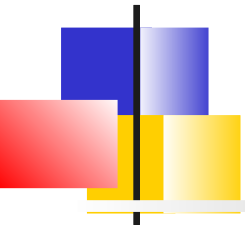
- Le POM des sous-modules référence le parent via ses coordonnées
- Les sous-modules ont souvent des dépendances vers d'autres sous-modules du projet

```
<parent>
  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <version>1.0</version>
</parent>
...
<dependencies>
  <dependency>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-model</artifactId>
    <version>1.0</version>
  </dependency>
...
```

Exécution de Maven



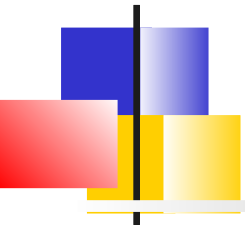
- Lors de l'exécution du build dans le répertoire du projet parent, Maven commence par charger le POM parent et localise tous les POMs des sous-modules
- Ensuite, les dépendances des sous-modules sont analysées et déterminent l'ordre de compilation et d'installation



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

Version du projet



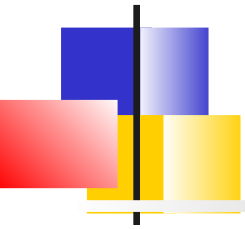
- La version d'un projet Maven permet de grouper et ordonner les différentes releases.
- Elle est constituée de 4 parties :

```
<major version>.<minor version>.<incremental version>-<qualifier>
```

Exemple : *1.3.5-beta-01*

- En respectant ce format, on permet à Maven de déterminer quelle version est plus récente
- SNAPSHOT indique qu'une version est en cours de développement et nécessite de récupérer le code souvent

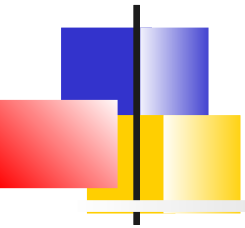
Gestion des versions des dépendances



Pour éviter de disperser dans tous les projets de l'entreprise les mêmes dépendances classiques ... et d'avoir des soucis de mis à jour lors d'une montée de version

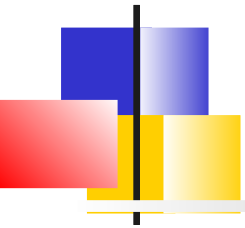
Il est possible de définir dans un POM parent via la balise ***<dependencyManagement>*** les versions des dépendances

<*dependencyManagement*>



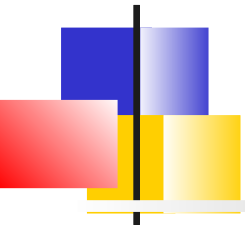
- L'élément <*dependencyManagement*> permet de définir dans un POM parent une dépendance qui peut être utilisée par les POMs enfants sans indiquer la version
- Seuls les enfants faisant référence à l'artefact et n'indiquant pas de version héritent de la dépendance définie dans le parent.

POM parent

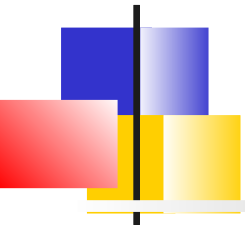


```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.2</version>
    </dependency>
    ...
  <dependencies>
</dependencyManagement>
```

POM enfant



```
<parent>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
</parent>
<artifactId>project-a</artifactId>
...
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
```

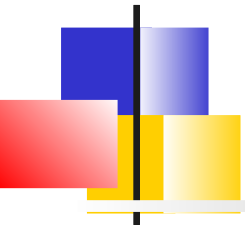


Gestion des versions des plugins

Le même mécanisme est utilisé pour fixer la version des plugins utilisés

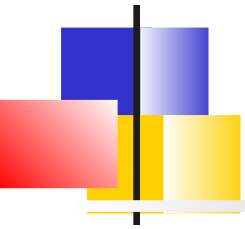
Dans un POM parent, la balise ***<pluginManagement>*** permet de fixer les versions des plugins utilisables ainsi que des propriétés de configuration

=> Les POMs enfants utilisent alors les mêmes versions et configurations des plugins



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

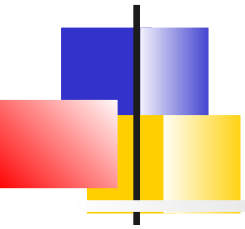


Adaptation

2 mécanismes sont possibles pour adapter le cycle de construction Maven aux spécificités d'un projet

- **Configurer** les plugins utilisés en surchargeant la configuration par défaut.
  Voir la doc du plugin pour trouver les options de configuration
- **Attacher** des objectifs de nouveaux plugins à des phases du build

Exemple *compiler:compile*

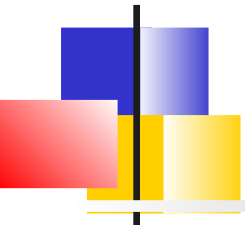


Par défaut, cet objectif compile tous les sources de *src/main/java* vers *target/classes*

Le plugin *Compile* utilise alors *javac* et suppose que les sources sont en Java 1.3 et vise une JVM 1.1 !

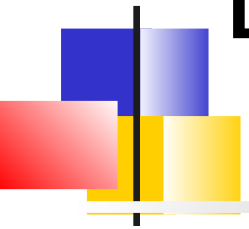
Pour changer ce comportement, il faut spécifier les versions visées dans le *pom.xml*

Exemple



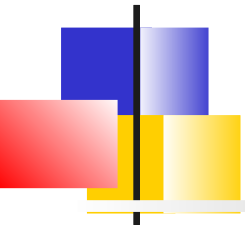
```
<project> ...
<build> ...
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins> ...
</build> ...
</project>
```

Association objectif/phase. L’exemple du plugin Assembly



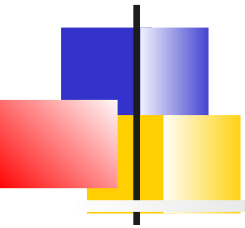
- Le plugin ***Assembly*** permet de créer des distributions du projets sous d'autres formats que le types d'archives standard (*.jar*, *.war*, *.ear*)
  - Il est ainsi possible de générer un fat jar (exécutable du projet)
- Par défaut, ce plugin n’est pas utilisé par le cycle de construction

Attacher assembly avec package

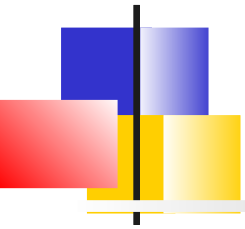


On peut configurer Maven afin qu'il exécute l'assemblage lors du cycle de vie par défaut en attachant l'objectif d'assemblage à la phase de packaging

Exemple

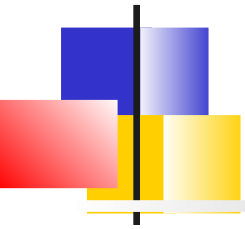


```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
         <descriptorRefs>
           <descriptorRef>jar-with-dependencies</descriptorRef>
         </descriptorRefs>
      </configuration>
     <executions>
      <execution>
         <id>make-assembly</id>
         <phase>package</phase>
         <goals>
           <goal>single</goal>
         </goals>
      </execution>
    </executions>
    </plugin>
  </plugins>
</build>
```



Maven

Concepts généraux et *pom.xml*
Commandes Maven
Dépôts Maven
POMs parents
Gestions de versions
Adaptation du build
Profils

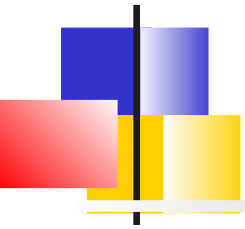


Profils

- Les ***profils*** permettent de personnaliser un build en fonction d'un environnement (test, production, ...)
  - Maven permet de définir autant de profils que désirés qui **surchargent** la configuration par défaut
  - Les profiles sont **activés** manuellement, en fonction de l'environnement ou du système d'exploitation
  - Les profiles, configurés dans le *pom.xml,* sont associés à des **identifiants**

Exemple

```
<profiles>
  <profile>
  <id>production</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
         <debug>false</debug>
         <optimize>true</optimize>
        </configuration>
      </plugin>
    </plugins>
  </build>
  </profile>
</profiles>
```



<profiles>

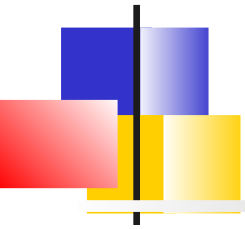
La balise ***<profiles>*** liste les profils du projet, elle se trouve en fin du fichier *pom.xml*

Chaque profile a un élément ***<id>***, l'activation du profile en ligne de commande s'effectue via ***-P<id>***

Un élément ***<profile>*** peut contenir les balises qui se trouve habituellement sous *<project>*

Exemple

```
<profiles>
  <profile>
    <id>prod</id>
    <dependencies>
      <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
      </dependency>
   </dependencies>
   <build>
     <plugins><plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
             <executable>true</executable>
        </configuration>
        <executions><execution>
          <goals>  <goal>build-info</goal>  </goals>
        </execution> </executions>
     </plugin> </plugins>
   </build>
  </profile>
</profiles>
```



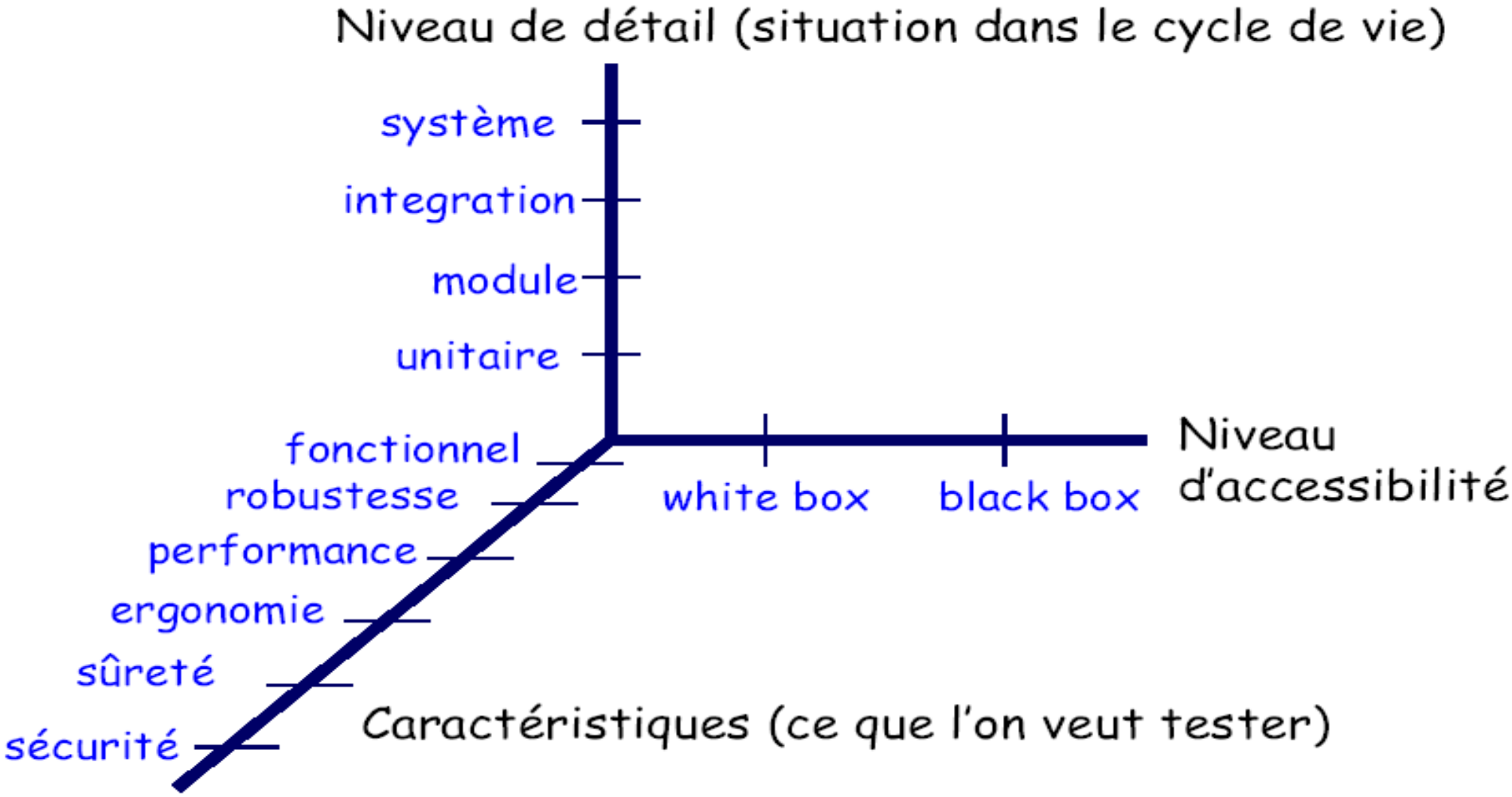
Outils de build

Caractéristiques d’un outil de build

Maven

Les tests et la qualité

Types de test



Types de test

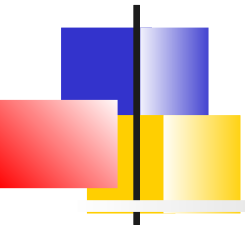
Test Unitaire :
Est-ce qu'une simple classe/méthode fonctionne correctement ?
JUnit, PHPUnit, UnitTest, Karma, Mockito

Test d'intégration :
Est-ce que plusieurs classes/couches fonctionnent ensemble ?
BD/Serveurs embarqués

Test fonctionnel :
Est-ce que mon application fonctionne ?
Selenium, HttpUnit, Protractor

Test de performance :
Est-ce que mon application fonctionne bien ?
JMeter, Gatling

Test d'acceptance :
Est-ce que mon client aime mon application ?
Cucumber



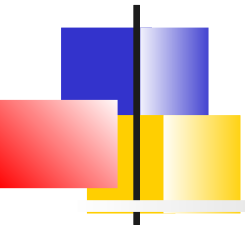
Classification DevOps

Dans l’optique d’une pipeline DevOps, on classifie les tests en fonction de :

- Nécessite t il le provisionnement d’infrastructure ?
- Durée d’exécution

=> Conditionne la position des tests dans la pipeline

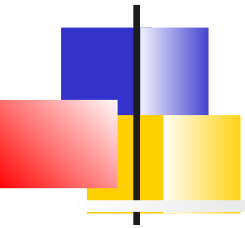
Typiquement : Tests unitaires en premier, test fonctionnel, d’acceptance et performance en dernier sur les infrastructures de pré-production et de production



Tests d'acceptance

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

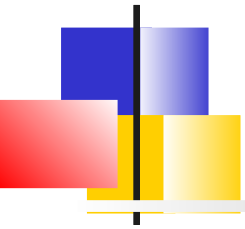
- Ils sont mis au point avec le client, le testeur et les développeurs
- Dans les méthodes agiles, ils complètent et valident une « User Story »
- Avec l'approche BDD (*Behaviour Driven Development*), l'expression des tests peut être faite en langage naturel.



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
```

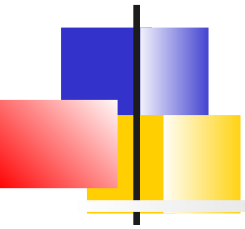


Analyse statique

L'analyse statique est l'analyse du code sans l’exécuter.

Les objectifs sont :

- La mise en évidence d'éventuelles erreurs de codage
- La vérification du respect du formatage convenu.
- La production de métriques adossés à la norme ISO25010 relatif à la qualité



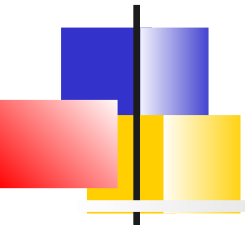
Métriques internes et Outils Qualité

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

Il intègre 2 aspects :

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

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



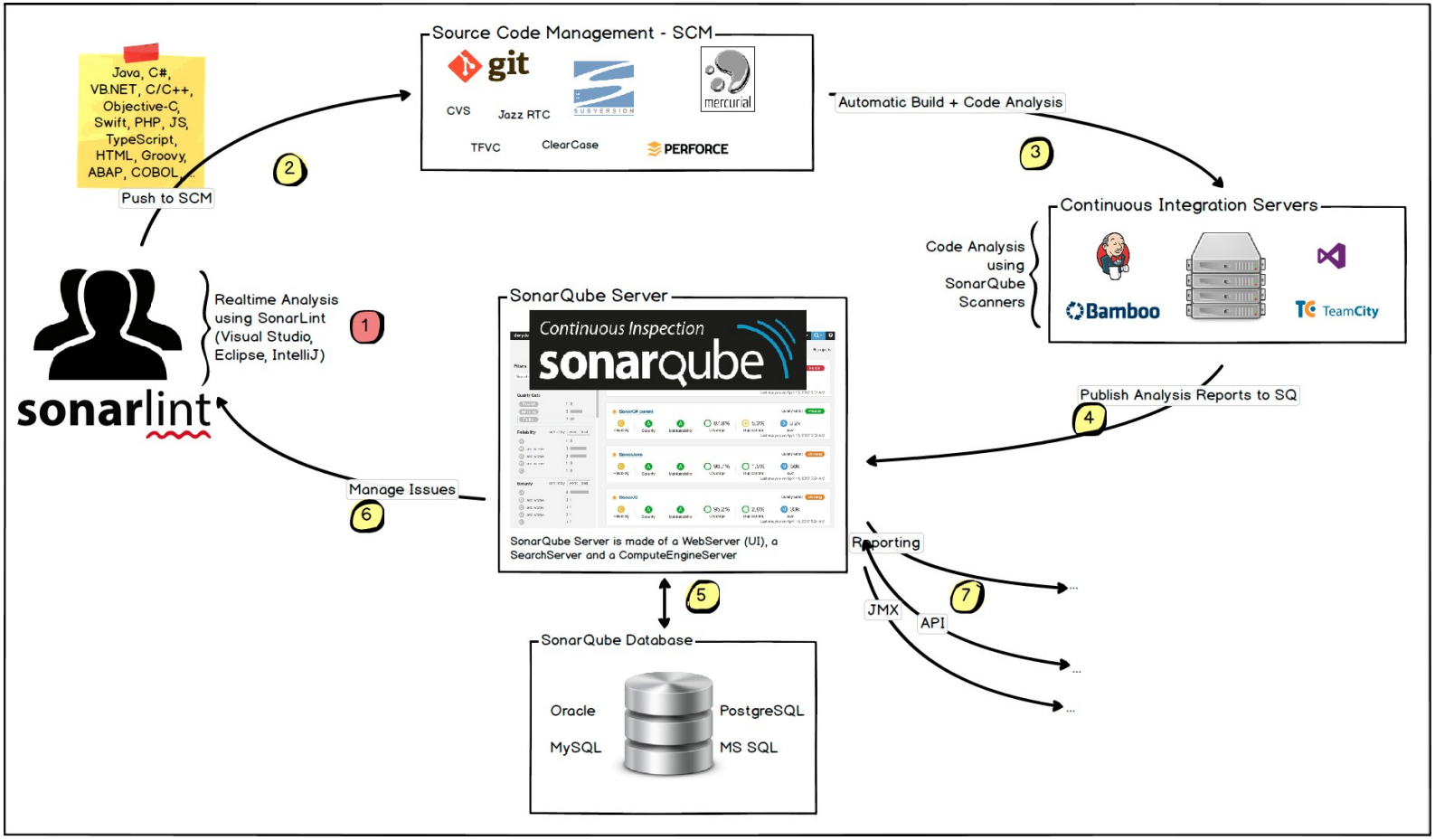
CI et Porte qualité

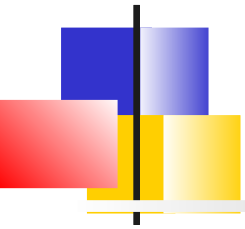
Les **portes qualité** définissent un ensemble de seuils pour les différents métriques. Le dépassement d'un seuil :

- Déclenche un avertissement
- Empêche la production d'une release.

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

Analyse continue



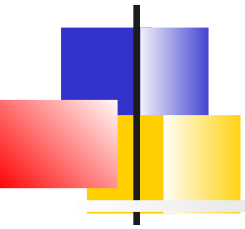


Gestion de releases

Dépôts privés

Processus de release

Nexus

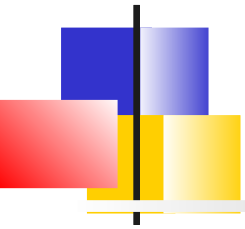


Dépôts

Avec Maven, il existe donc différents types de dépôts utilisés pour récupérer les dépendances et pour y installer des artefacts.

- Le dépôt **local**
- Les dépots **publics** fournis par Maven, JBoss, Sun, ...
- Les dépôts **miroirs** généralement configurés dans le fichier *settings.xml*
- Les dépôts **privés**, propres à une organisation ou entreprise

Dépôt privé



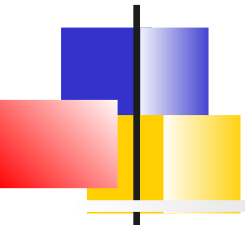
L'utilisation de Maven dans le contexte d'une entreprise nécessite souvent la mise en place d'un dépôt interne à l'entreprise :

- sécurité,
- bande passante
- et surtout possibilité d'y publier les artefacts produits par la société.

Les artefacts produits peuvent être récupérés via *http*, *scp*, *ftp* ou *cp*

Les outils spécialisés sont *Nexus*, *Artifactory*, *Archiva*

Utilisation d'un dépôt interne



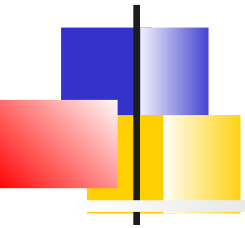
Il suffit d'indiquer :

- une balise ***<repository>*** dans le fichier *pom.xml* qui référence un id de serveur défini *dans* une balise ***<server>*** dans du fichier *settings.xml*
- Dans *settings.xml*, on trouve les crédentiels d’accès au dépôt interne

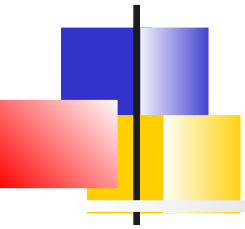
Balise repository dans *pom.xml*

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://myserver/repo</url>
    </repository>
  </repositories>
  ...
</project>
```

Balise <*server*>



```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>my-internal-site</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
  ...
</settings>
```



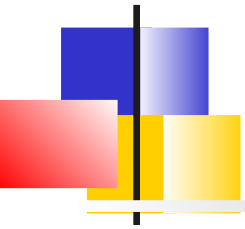
Deploy plugin

Le plugin ***deploy*** permet de déployer un artefact dans un dépôt distant.

En général, un dépôt d'entreprise.

Le plugin offre 2 objectifs principaux :

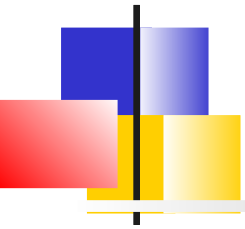
- ***deploy*** : attaché à la phase *deploy*, il publie l'artefact dans un dépôt distant
- ***deploy-file*** : Permet de déployer un fichier qui n'a pas été construit par Maven



<distributionManagement>

Afin que l'objectif soit effectif, il faut spécifier dans le POM une balise ***<distributionManagement>*** qui indique le dépôt à utiliser

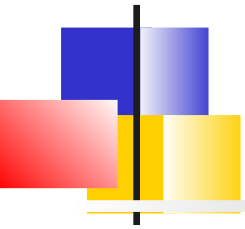
```
<distributionManagement>
    <repository>
      <id>internal.repo</id>
      <name>MyCo Internal Repository</name>
      <url>Host to Company Repository</url>
    </repository>
 </distributionManagement>
```



SNAPSHOTS et RELEASES

En général, 2 types d'artefacts sont stockés dans les **dépôts**

- Les **SNAPSHOTS**, ils fournissent le dernier état (plutôt stable) de la version en cours de développement
  => Les projets dépendants peuvent alors anticiper les futures versions
- Les **releases**, ils sont immuables.
  Ils sont taggés avec le même tag que les sources les ayant produits.



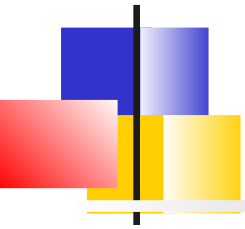
Gestion de releases

Dépôts privés

Processus de release

Nexus

Production d’une release

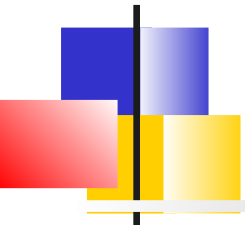


La processus de production d’une release consiste généralement en :

- Générer l’artefact et s’assurer que les tous les tests sont OK
- Modifier le n° de version et intégrer la branche de travail dans la branche principale
- Tagger le commit avec un n° de version
- Publier l’artefact dans le dépôt d’artefact
- Incrémenter le n° version (et lui apposer le suffixe SNAPSHOT).

Ce processus doit être automatisé soit par la PIC, soit par des plugins Maven.

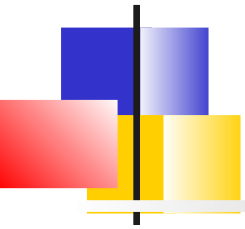
Maven et SCM



Maven supporte les interactions avec un système de contrôle de version.

L'emplacement du SCM doit être configuré dans le *pom.xml* via la balise ***<scm>***

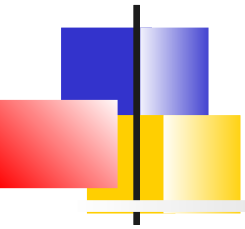
Ainsi, certains plugins (*SCM*, *Release*) peuvent utiliser ces informations pour automatiser des opérations avec le SCM



Exemple GIT

```
<scm>
<url>https://github.com/kevinsawicki/github-maven-example</url>
<connection>scm:git:git://github.com/kevinsawicki/github-maven-example.git</connection>
<developerConnection>scm:git:git@github.com:kevinsawicki/github-maven-example.git</developerConnection>
</scm>
```

Plugin *Release*



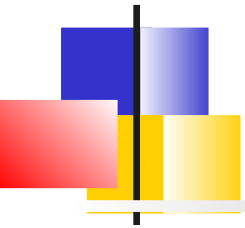
Le plugin ***Release*** permet d'effectuer des releases en automatisant les modifications manuelles des balises <*versions*> et les opérations avec le SCM.

Une release est effectuée en 2 étapes principales :

prepare : Préparation

perform : Réalisation de la release

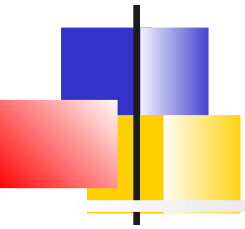
Préparation



La préparation effectue les étapes suivantes :

- Vérification que toutes les sources ont été commitées
- Vérification qu'il n'y a pas de dépendance vers des versions SNAPSHOT
- Change la versions dans les POM de *x-SNAPSHOT* vers une nouvelle version saisie par l'utilisateur
- Transforme l'information SCM dans le POM pour inclure le tag de destination de la release
- Exécute les tests avec les POMs modifiés
- Commit les POMs modifiés
- Tag le code dans le SCM avec le nom de version
- Change les versions dans les POMs avec une nouvelle valeur : y-SNAPSHOT
- Commit les POMs modifiés
- Génère un fichier *release.properties* utilisé par l'objectif *perform*

Exemples de commande



Reprend la commande ou elle s'est arrêtée

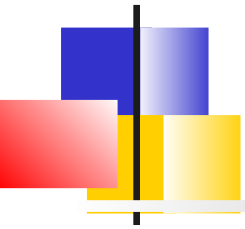
```
mvn release:prepare
```

Reprend la commande depuis le début

```
mvn release:prepare -Dresume=false
```

Ou

```
mvn release:clean release:prepare
```

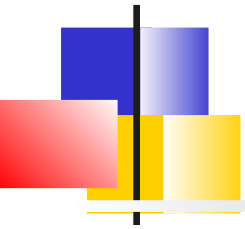


Perform

L'objectif ***perform*** effectue les traitements suivants :

- Effectue un check-out à partir d'une URL lue dans le fichier *release.properties*
- Appel de l'objectif prédéfini *deploy*

Process de release



Le process de *release* consiste à appeler successivement les objectifs :

release:prepare

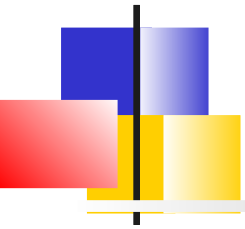
release:perform

Pour réussir, on doit s'assurer que :

- Le *pom* effectif n'a pas de dépendances sur des versions SNAPSHOTS
- Tous les changements locaux ont été commités
- Tous les tests passent

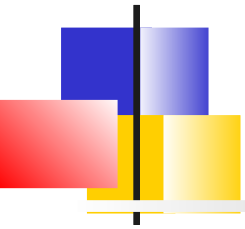
Tout cela peut se faire en une seule commande maven :

mvn release:prepare release:perform -B



Gestion de releases

Dépôts privés
Processus de release
Nexus



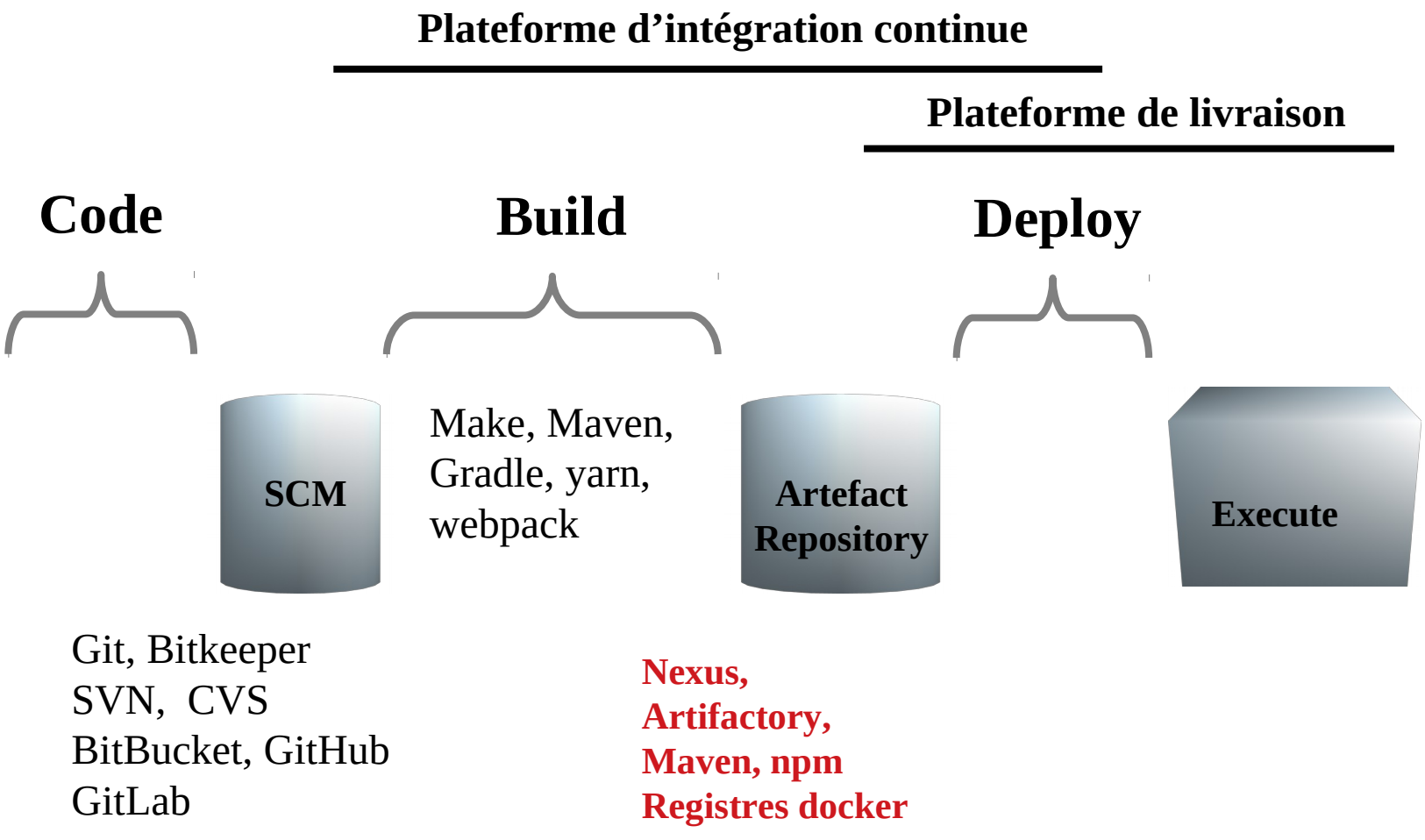
Nexus

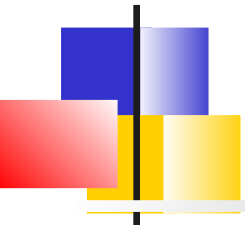
Nexus est un gestionnaire de dépôt qui offre principalement 2 fonctionnalités :

- Proxy d'un dépôt distant (Maven central) et cache des artefacts téléchargés
- Hébergement de dépôt interne (privé à une organisation)

Nexus support de nombreux types de dépôts ... et bien sût les dépôts Maven

Les gestionnaires de dépôts dans le cycle de vie

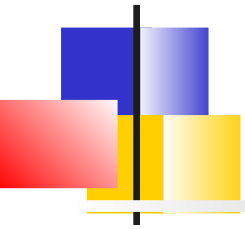




Concepts

Nexus manipule des **dépôts** et des **composants**

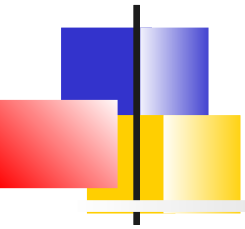
- Les composants sont des artfacts identifiés par leurs coordonnées. Nexus permet d’y attacher des méta-données Pour être générique envers tous les types de composants. Nexus utilise le terme asset
- Les dépôts peuvent être des dépôts Maven, npm, RubyGems, ...



Types de dépôt

Nexus permet de définir différents types de dépôts :

- **Proxy** : Nexus se comporte alors proxy d'un dépôt distant avec des fonctionnalités de cache
- **Hosted** : Nexus stocke des artefacts produits par l'entreprise
- **Group** : Permet de grouper sous une URL unique plusieurs dépôts

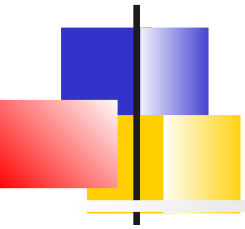


Interface utilisateur

Nexus fournit un accès anonyme pour la recherche de composants

Si on est loggé, on a accès aux fonctions d’administration (gestion des dépôts, sécurité, traces, API Rest, …)

La fonctionnalité de recherche propose de nombreuses options



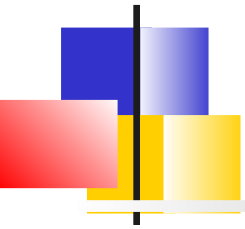
Installation par défaut

La création d'un dépôt dans Nexus s'effectue avec un login Administrateur

Administration → repositories → Add

L'installation par défaut à déjà créé le dépôt groupe nommé ***maven-public*** qui regroupe :

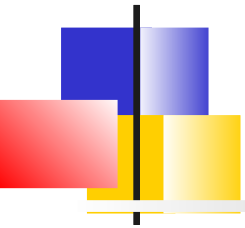
- *maven-releases* : Pour stocker les releases internes
- *maven-snapshots* : Pour stocker les snapshots internes
- *maven-central* : Proxy de Maven central



Intégration Maven

La configuration Maven consiste à modifier le fichier *settings.xml* pour utiliser Nexus comme :

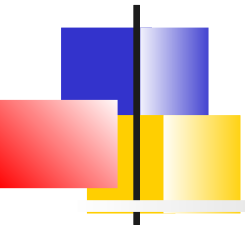
- Proxy de Maven Central
- Dépôt des artefacts snapshots
- Dépôt des artefacts de releases
- Définir un dépôt groupe permettant une URL unique pour les dépôts précédents



Exemple

```
<servers>
    <server>
      <id>nexus-snapshots</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
    <server>
      <id>nexus-releases</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
  </servers>

  <mirrors>
    <mirror>
      <id>central</id>
      <name>central</name>
      <url>http://localhost:8081/repository/maven-public/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
```



Configuration projet

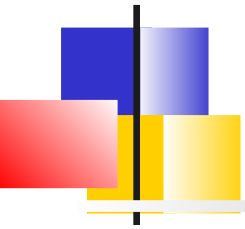
Au niveau du projet Maven, il faut indiquer

- La balise repository pour télécharger les dépendances à partir de Nexus
- la balise <*distributionManagement*> pour déployer vers nexus

Configuration projet

```
<project>
...
<repositories>
    <repository>
      <id>maven-group</id>
      <url>http://your-host:8081/repository/maven-group/</url>
    </repository>
  </repositories>

<distributionManagement>
    <snapshotRepository>
      <id>nexus-snapshots</id>
      <url>http://your-host:8081/repository/maven-snapshots/</url>
    </snapshotRepository>
    <repository>
      <id>nexus-releases</id>
      <url>http://your-host:8081/repository/maven-releases/</url>
    </repository>
  </distributionManagement>
```



Plateforme d’intégration continue

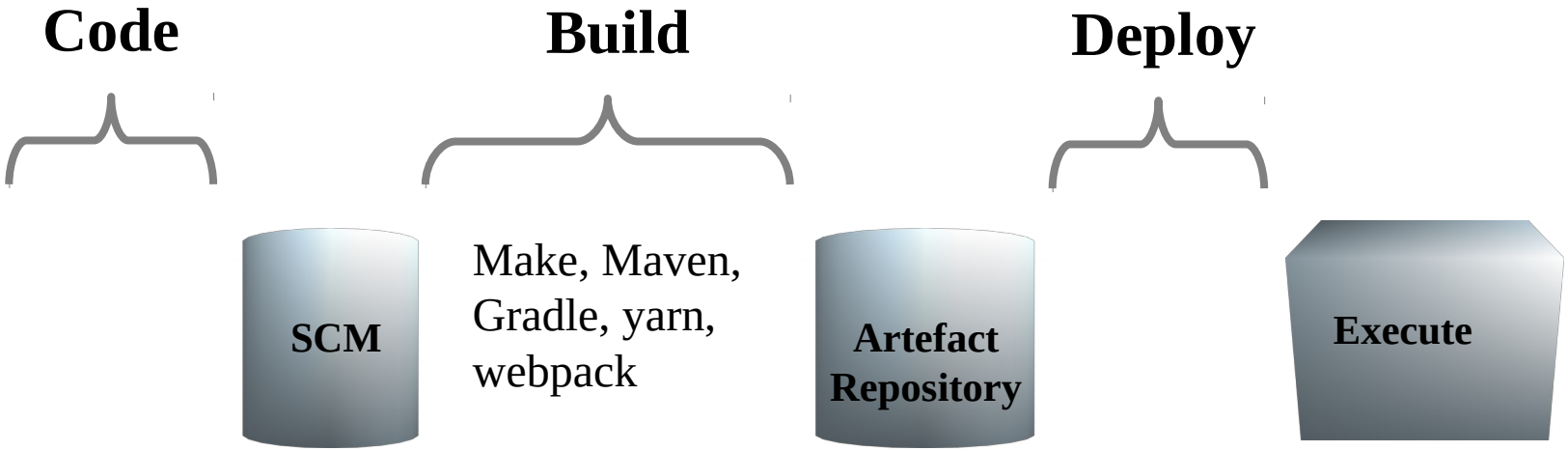
Concepts communs
Pipelines typiques
Solutions

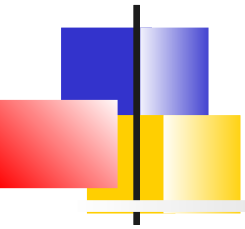
PICs dans le Cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison





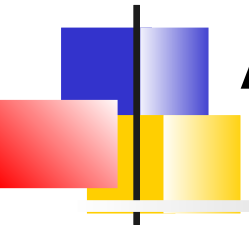
Plateforme d’intégration continue

Une PIC a pour objectifs :

- 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 en général multi-projets, multi-branches, multi-configuration

=> Il nécessite beaucoup de ressources



Architecture Maître / esclaves

Le serveur central distribue les jobs de build sur différentes ressources appelés les esclaves/runners/workers.

Les esclaves sont :

- Des **machines physiques, ou virtuelles** où sont préinstallés les outils nécessaires au build
- Des **machines virtuelles ou conteneurs** qui sont alors provisionnés/exécutés lors du build et qui disparaissent ensuite

Netflix 2012

1 master avec 700 utilisateurs

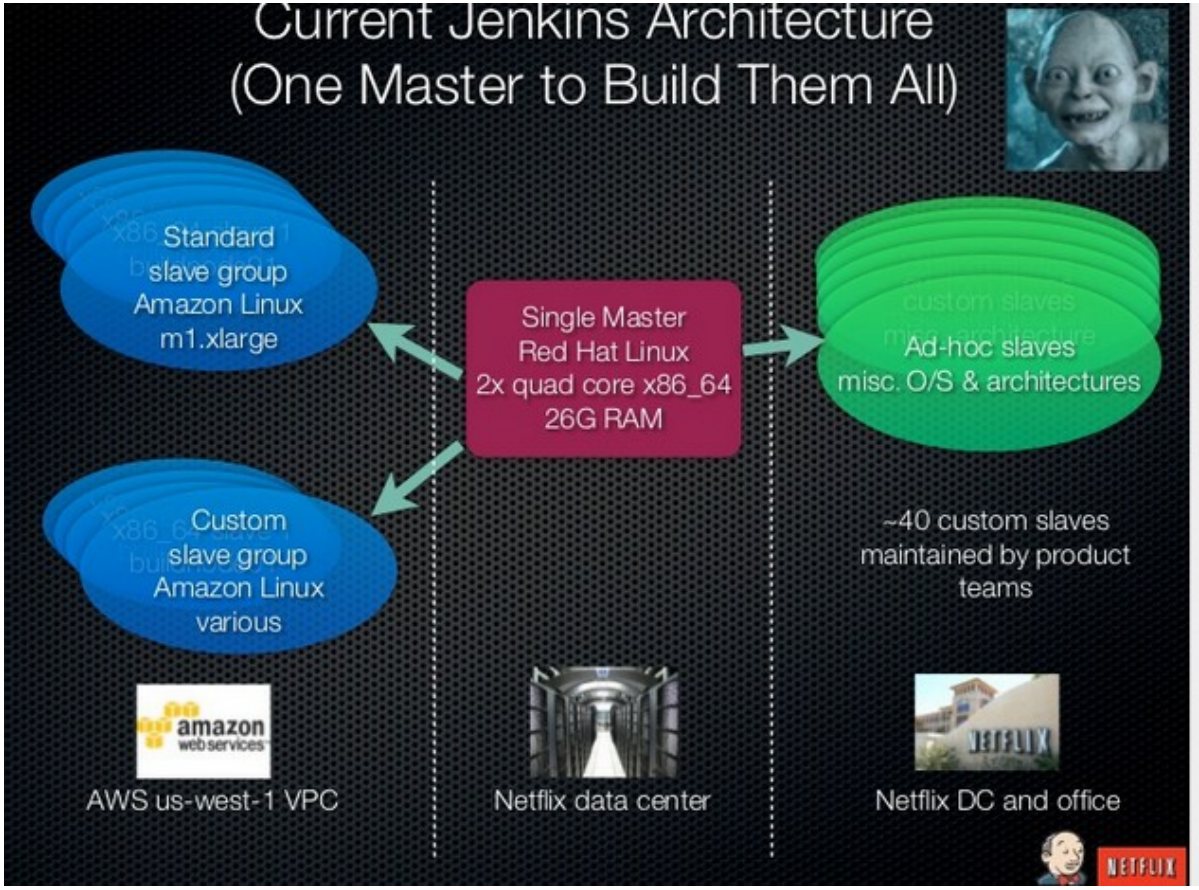
1,600 jobs :

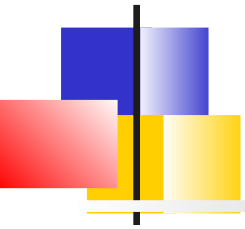
- 2,000 builds/jour
- 2 TB
- 15% build failures

=> 1 maître avec 26Go de RAM

=> Agent Linux sur amazon

=> Esclaves Mac. Dans le réseau interne



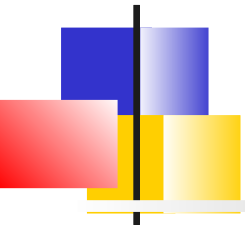


Outils annexes, plugins

La plateforme doit interagir avec de nombreux outils annexes : SCM, Outils de build, de test, plateforme de déploiement

Les outils peuvent être intégrés

- Directement par la solution
- Via des plugins
- Via docker

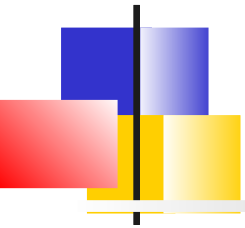


Déploiement

Le PIC a vocation à automatiser des déploiements dans différents environnements

Il doit donc pouvoir :

- Accéder aux environnements statiques.
  Ex : dépôt de l'artefact via ftp, redémarrage d'un service
- Provisionner dynamiquement l'infrastructure. Ex : Création dynamique d'une machine virtuelle
- Déclencher la plateforme de déploiement.
  Ex : Dépôt de l'image d'un conteneur dans un repository

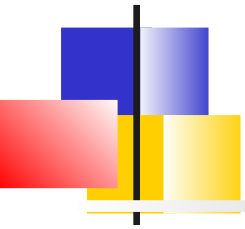


Approche standard

L’approche standard consiste à disposer d’une PIC centralisé.

Des administrateurs :

- installent les plugins nécessaires
- créent des jobs via l’UI et spécifient leur séquencement
- Exploitent et surveillent l’exécution des pipelines



Approche DevOps

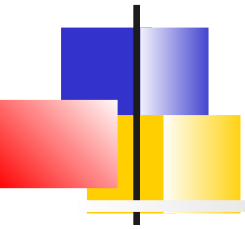
Dans l’approche DevOps, la pipeline est décrite par un DSL

Le fichier de description est stocké dans le SCM, il est géré par l’équipe projet.

- La pipeline peut s’exécuter sans l’administrateur de la PIC

Ou dans certains cas, l’infrastructure de PIC nécessaire au projets (plugins et autre) est elle même conteneurisé et stocké dans le SCM

Pipelines typiques



Principes

Les pipelines sont généralement découpées en **phases** représentant les grandes étapes de la construction.

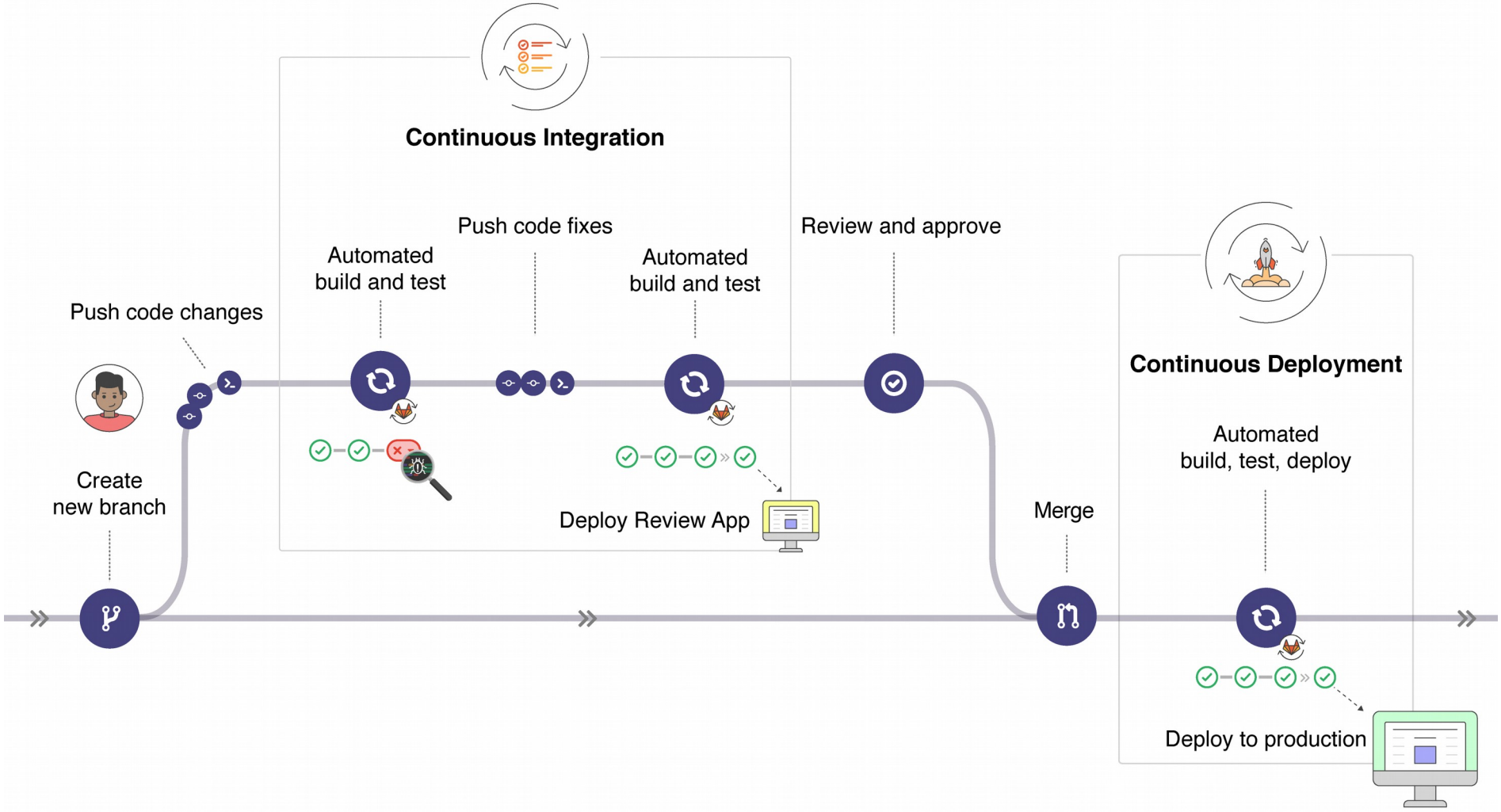
Par exemple : Build, Test, QA, Prodcution

Chaque phase est constituée de tâches ou **jobs** exécutés séquentiellement ou en parallèle

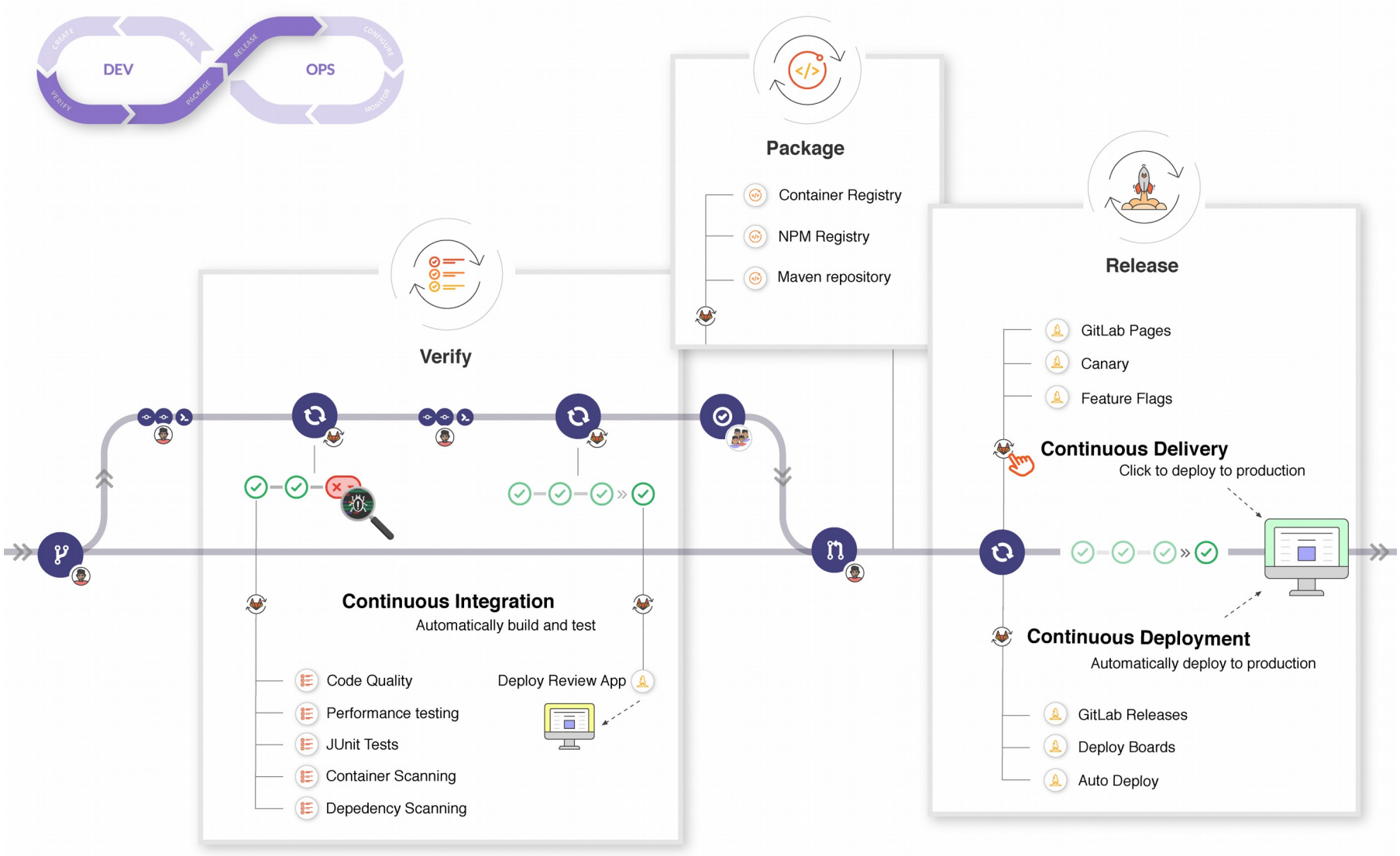
Les jobs exécutent des actions à partir du dépôt ou réutilisent des artefacts précédemment construits

Les jobs publient des rapports résultats

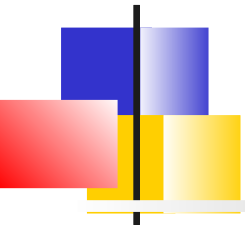
Exemple Gitlab CI



En plus détaillé



Solutions



Gitlab CI

GitLab propose désormais la gestion de constructions et/ou de tests via ***Gitlab-CI***

- Développé en Go, il propose un mode ’server-runner’
- Les *runners* exécutent les pipelines. Les outils nécessaires pour le build sont pré-installés ou des images Docker sont utilisés.
- Les pipelines sont codés via un fichier au format YAML
- Elles s’exécutent sur toutes les branches de GitlabFlow.
- GitlabCI gère les déploiements sur les différents environnements et propose une intégration avec les clusters Kubernetes pour le déploiement

Exemple *.gitlab-ci.yml*

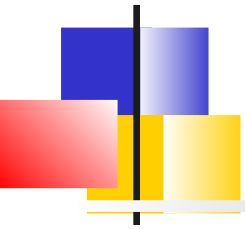
```
image: "ruby:2.5"

before_script:
  - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
  - ruby -v
  - which ruby
  - gem install bundler --no-document
  - bundle install --jobs $(nproc)  "${FLAGS[@]}"

rspec:
  script:
    - bundle exec rspec

rubocop:
  script:
      - bundle exec rubocop
```

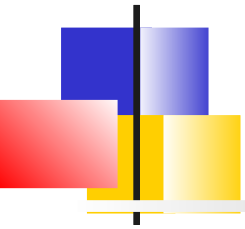
Gitlab CI



Running 0 | Finished 327 | All 327

List of finished builds from this project

Status	Build ID	Commit	Ref	Runner	Name	Duration	Finished at
✔ success	**Build #351965**	23b89d99	artifacts	golang-cross#1059	Bleeding Edge	6 minutes 4 seconds	about 19 hours ago
✔ success	**Build #351548**	634b6f5e	artifacts	golang-cross#1059	Bleeding Edge	5 minutes 43 seconds	about 22 hours ago
✔ success	**Build #349948**	56329a8e	artifacts	golang-cross#1059	Bleeding Edge	6 minutes 2 seconds	1 day ago
✔ success	**Build #349883**	c01876c1	master	golang-cross#1059	Bleeding Edge	5 minutes 39 seconds	1 day ago
✖ failed	**Build #349807**	623f3f5a	master	golang-cross#1059	Bleeding Edge	1 minute 50 seconds	1 day ago
✖ failed	**Build #349804**	338d0a8b	artifacts	golang-cross#1059	Bleeding Edge	1 minute 35 seconds	1 day ago



Jenkins

Anciennement Hudson, (fork depuis le rachat par Oracle)

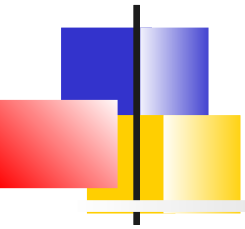
Encore, le plus répandu

Soutenu par la société CloudBees

De nombreux plugins disponibles :

- Plugins Legacy
- Plugin pipeline (DevOps) et visualisation des pipeline via Blue Ocean

Forte intégration avec Docker



Approche DevOps

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

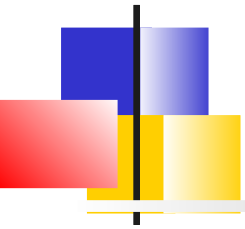
Un fichier ***Jenkinsfile*** faisant partie intégrante des sources du projet décrit la pipeline de déploiement continu

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

Illustration

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        sh './mvnw -Dmaven.test.failure.ignore=true clean test'
       } post {
        always { junit '**/target/surefire-reports/*.xml'     }
       }
    }
    stage('Parallel Stage') {
      parallel {
        stage('Intégration test') {
          agent any
          steps {
            sh './mvnw clean integration-test'
          }
        }
        stage('Quality analysis') {
          agent any
          steps {
           sh './mvnw clean verify sonar:sonar'
          }
        }
      }
   }
```



Déploiement

Considérations

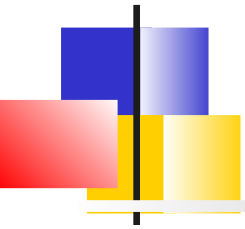
Virtualisation

Outils de gestion de configuration

Containerisation. Le cas docker

Orchestrateurs

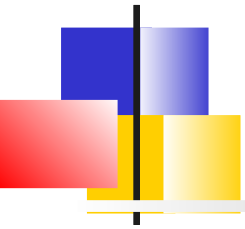
Kubernetes



Introduction

2 décisions importantes concernant l’infra :

- L’architecture applicative : Applications monolithiques ou micro-services
- Format des artefacts : Déploiements mutables ou immuables



Modèle classique

Le serveur monstre mutable

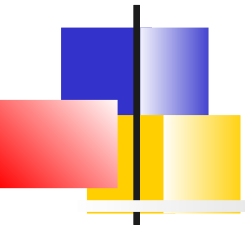
Un serveur Web qui contient toute l'application et mis à jour à chaque déploiement.

- Fichiers de configuration, Artefacts, schémas base de données.

=> il mute au fur et à mesure des déploiements.

=> On n’est plus sur que les environnements de dév, de QA ou même les instances en production soient identiques

=> Difficulté de revenir à une version précédente



Modèle DevOps

Serveur immuable et Reverse Proxy

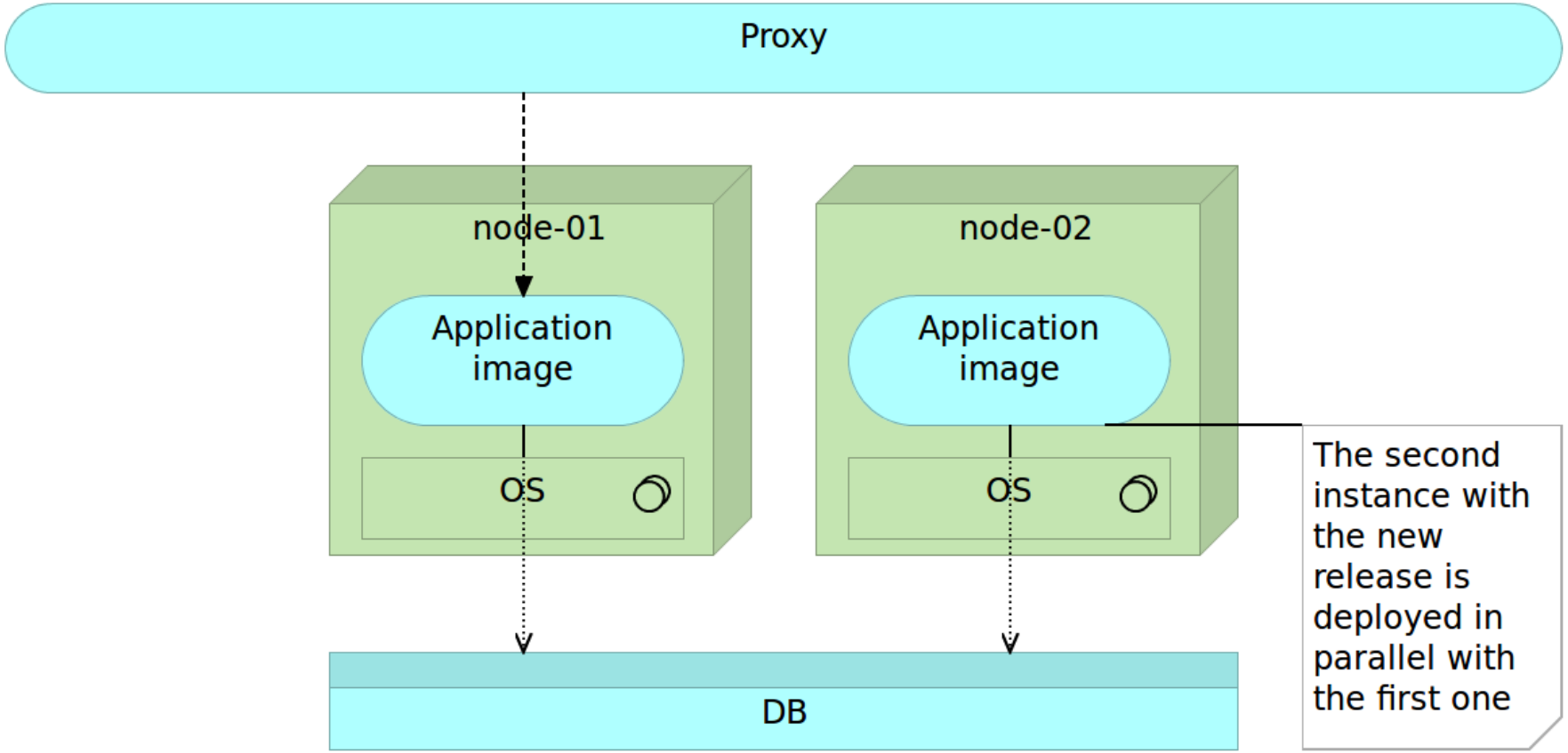
L’approche DevOps s’appuie sur les déploiements immuables qui garantissent que chaque instance déployée est exactement identique.

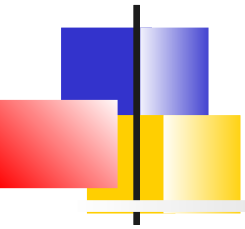
Un package immuable contient tout : serveur d'applications, configurations et artefacts

C’est ce tout qui est déployé

Modèle DevOps

Serveur immuable et Reverse Proxy





Architecture micro-services

La durée du déploiement dépend principalement de :

- La durée de l'instanciation du déploiement immuable
- Des tests de post-déploiement

Une architecture micro-services basée sur des containers optimise énormément ces temps, permettant d'augmenter la fréquence de déploiement et de repli
=> scalabilté dynamique, serverless

Virtualisation

La virtualisation dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

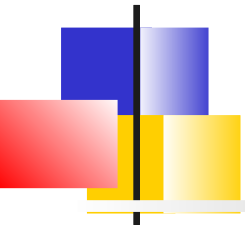
Plateforme de livraison

Code **Build** **Deploy**

Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab

Nexus,
Artifactory,
Archiva

Virtualisation :
VMWare,
Citrix, KVM
VirtualBox

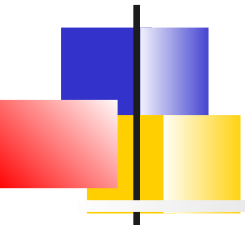


Hyperviseur

Les **hyperviseurs** permettent à une machine nue de superviser plusieurs machines virtuelles.

Les déploiements peuvent ainsi être isolés sur des environnements dédiés.

Bien que le serveur soit virtualisé, le principe reste équivalent à celui d’un serveur dédié.



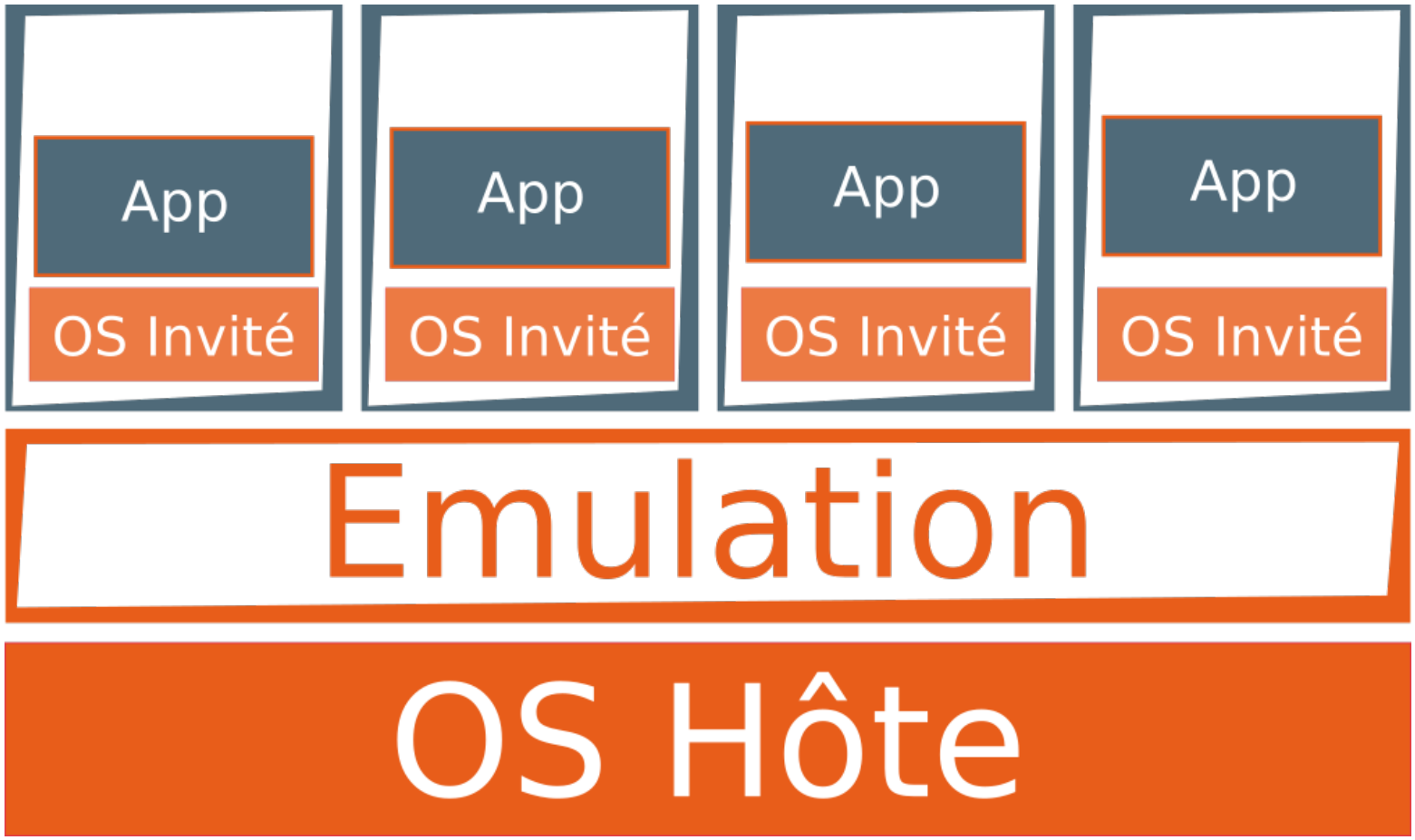
Limitations

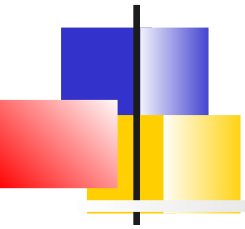
Pour virtualiser un environnement complet le serveur doit être capable

- de supporter la charge de son OS hôte ainsi que les OS invités
- + l'émulation du matériel (CPU, mémoire, carte vidéo...) et la couche réseau associée.

=> Les ressources nécessaires pour exécuter une Machine Virtuelle sont importantes.

Hôte / invité


App
OS Invité
App
OS Invité
App
OS Invité
App
OS Invité
Emulation
OS Hôte

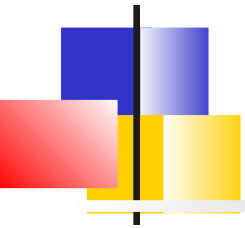


Solutions

Les principales solutions commerciales sont :

- **Microsoft Hyper-V**,
- **VMware vSphere** : Virtualisation + de nombreux autres services
- **Citrix XenServer** : Version OpenSource
- **Red Hat KVM** : Intégré dans le noyau Linux depuis 2.6.20

A un plus petit niveau, Oracle VirtualBox

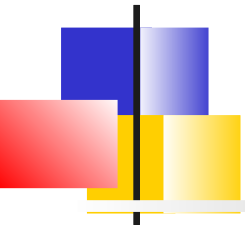


Vagrant

Vagrant est un outil permettant de gérer les machines virtuelles.

Il permet de configurer des machines virtuelles avec de simple fichier (*Vagrantfile*) et ainsi d'automatiser la configuration et le provisionnement

Il est compatible avec plusieurs de virtualisation : VirtualBox, VMware, AWS, ou autre

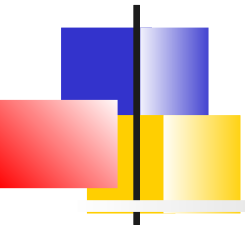


Projet Vagrant

Un projet consiste en la mise au point d'un fichier ***Vagrantfile*** (committé dans le SCM) qui :

- Marque la racine du projet.
- Décrit la machine, les ressources, les logiciel et les modes d'accès

On peut alors s'appuyer sur des ***vagrant box*** qui définissent des machines de base téléchargeables automatiquement

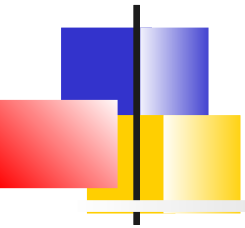


Configuration de la machine virtuelle

Par défaut, le répertoire contenant le *Vagrantfile* est monté sur le répertoire */vagrant* de la machine virtuelle.

=> Il est possible d'y positionner des scripts permettant de provisionner la box

Les autres directives de configuration permettent de faire du mapping de port , d'affecter une IP fixe, ou la raccorder à une réseau existant, de dimensionner la mémoire, etc..

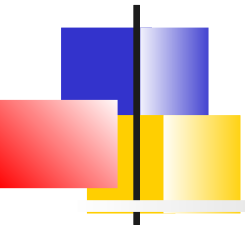


Exemple

```
Vagrant.configure("2") do |config|

  config.vm.define "db" do |db|
    db.vm.provider "virtualbox" do |v|
      v.memory = 2048
      v.cpus = 1
      v.name = "db"
    end
    db.vm.box = "ubuntu/bionic64"
    db.vm.provision :shell, path: "postgres.sh"
    db.vm.network "private_network", ip: "192.168.10.3"
    db.vm.network :forwarded_port, guest: 5432, host: 5444
  end

  config.vm.define "spring" do |spring|
    spring.vm.box = "ubuntu/bionic64"
    spring.vm.provision :shell, path: "spring.sh"
    spring.vm.network "private_network", ip: "192.168.10.2"
    spring.vm.network :forwarded_port, guest: 8080, host: 8000
  end
end
```



Principales commandes

Le client vagrant permet alors de nombreuses commandes :

- ***up***, ***halt***, ***suspend***, ***resume***, ***destroy*** : Démarrage, arrêt, pause, reprise, suppression de toutes les traces
- ***provision*** : Met à jour les logiciels sur une machine s'exécutant
- ***ssh***, ***powershell***, ***rdp*** : Accès distant sur la machine
- ***box*** (add,list, remove) : Gestion des vagrants box
- ***snapshot*** : Sauvegarde d'une machine s'exécutant
- ***push*** : Pousser la configuration sur un serveur FTP ou autre

Outils de gestion de configuration

Provisionnement dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

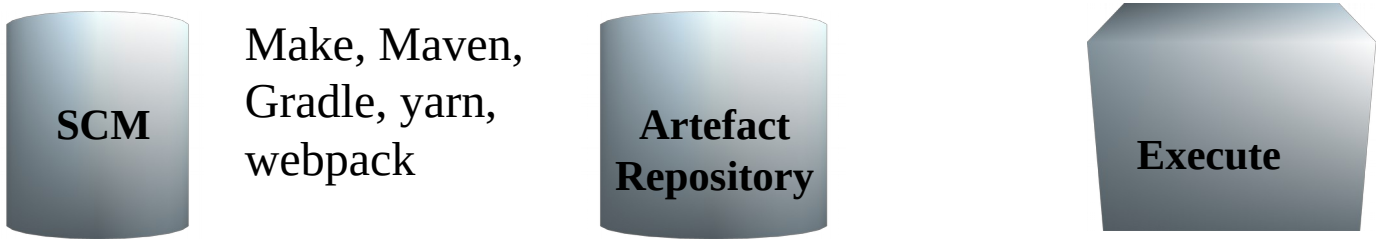
Plateforme de livraison

Code

Build

Deploy

Provisionnement :
Vagrant
Puppet, Chef,
Ansible

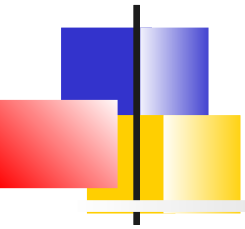


Make, Maven,
Gradle, yarn,
webpack

Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab

Nexus,
Artifactory,
Archiva

Virtualisation :
VMWare,
Citrix, KVM
VirtualBox

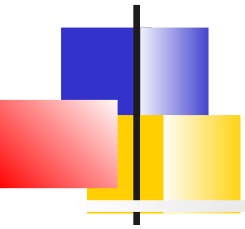


Infrastructure As Code

Pour gérer les configurations des machines virtuels (configuration réseau, sécurité, utilisateur ayant accès, etc.), des solutions sont apparues.

Elles permettent de décrire dans des fichiers scripts, les opérations de configuration .

- Outils : Chef, Puppet, Ansible, SaltStack
- Langages : Ruby, Python, DSL

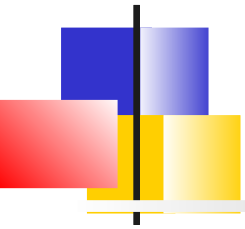


Gestion de déploiements

Les services de déploiements permettent d'intégrer une version sortie de la PIC sur un environnement de production

2 approches existent :

- L'approche centraliseé nécessitent l'installation d'agents sur les serveurs cibles services. Un serveur centralise et orchestre les déploiements
  Ex : *Chef, Puppet*
- Les services "agentless" ne nécessite pas de pré-installation et se base généralement sur *ssh*.
  Pendant, l'opération de déploiement des modules sont installés temporairement sur la machine cible.
  Ex : *Ansible*

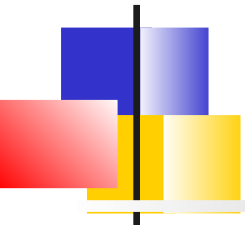


Ansible

Ansible est un moteur d'automatisation de déploiement et de provisionnement.

L'un des intérêts majeurs de Ansible est la "non utilisation d'agent", en d'autres termes les machines cibles n'ont pas besoin d'avoir de services toujours up

- Le seul pré-requis est l'installation de Python et il n'y a pas de support pour Windows



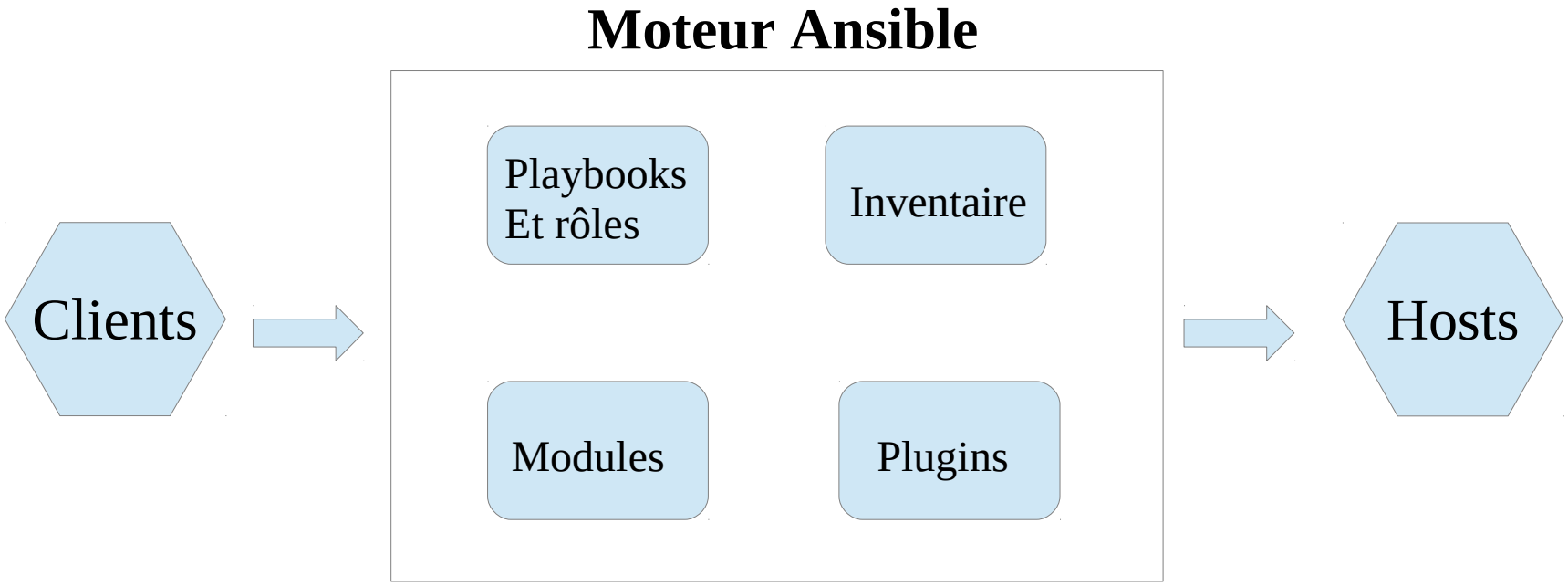
Fonctionnement

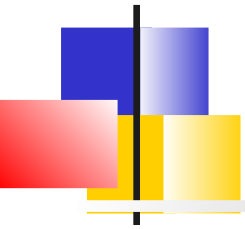
Ansible se connecte (via SSH par défaut) aux différents nœuds et y poussent de petits programmes, nommés "**modules Ansible**"

- Ansible exécute les modules et les enlève une fois l'exécution terminée
- Les modules Ansible peuvent être écrits dans n'importe quel langage du moment qu'ils retournent du JSON. Ils sont idempotents.
  Ex :

```
ansible all -s -m apt -a 'pkg=nginx
state=installed update_cache=true'
```

Architecture



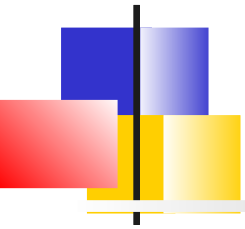


Inventaire

Ansible représente les machines qu'il gère via **l'inventaire**

- Par défaut, un simple fichier texte de type INI mais peut être une autre source de données.

L'inventaire permet également de définir des **groupes de machine**, de leur assigner des variables

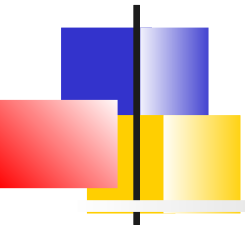


Exemple Inventaire

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```



Principales commandes

ansible : Exécute une tâche sur un ou plusieurs hosts

ansible-playbook : Exécute un playbook

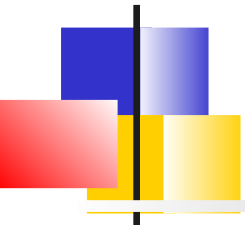
ansible-doc : Affiche la documentation

ansible-vault : Gère les fichiers cryptés

ansible-galaxy : Gère les rôles avec *galaxy.ansible.com*

ansible-pull : Exécute un playbook à partir d'un SCM

...



Commande en ligne

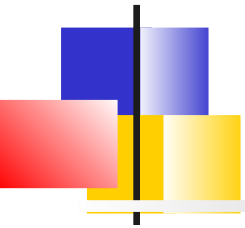
ansible : Exécute une tâche sur un ensemble de host

```
ansible all -m ping
```

```
ansible <HOST_GROUP> -m authorized_key -a "user=root
key='ssh-rsa AAAA...XXX == root@hostname'"
```

```
ansible -m raw -s -a "yum install libselinux-python -
y" new-atmo-images
```

D’autres commandes en ligne sont disponible : *ansible-config, ansible-console, ansible-pull*, ...

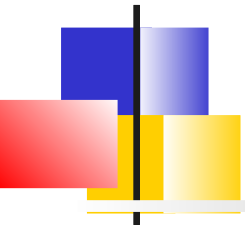


Playbooks

Les ***playbooks*** définissent ce qui doit être appliqué sur les serveurs cibles.

Ils déclarent des configurations mais peuvent également orchestrer les étapes d’un déploiement faisant intervenir différentes machines.

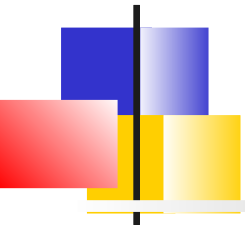
Ils sont commités dans le SCM



Configuration d'un déploiement

La configuration présente dans un playbook est constituée :

- Des **tâches**. Une tâche peut effectuer plusieurs opérations en utilisant un module Ansible.
- **Handler** s'exécute à la fin d'une série de tâches si un certain événement a été émis
- **Variables** : Valeurs dynamiques provenant de différentes sources
- **Gabarits** (Jinja2) : Fichiers variabilisés (Test et boucles disponibles).
- **Roles** sont des abstractions réutilisables qui permettent de grouper des tâches, variables, handlers, etc.



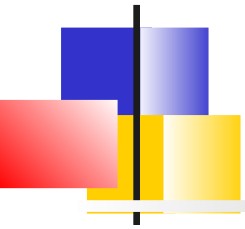
Exemple

```
---
- hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest
  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

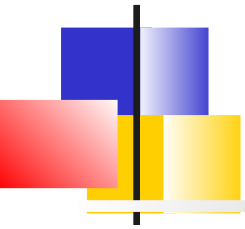


Rôles

Les **rôles** permettent d'organiser des tâches dépendantes et de permettre la réutilisation

Un rôle est un « *sous-playbook* », il peut contenir :

- ***tasks :*** Appels de modules *Ansible*
- ***variables :*** Des valeurs dynamiques positionnées lors de l'utilisation du rôle
- ***template :*** Des gabarits de fichiers (de configuration)
- ***files*** : Les fichiers à copier sur la cible
- ***Handlers : Modules réagissant à un état***

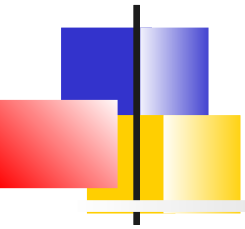


Arborescence classique

Au final, un projet *Ansible* contient principalement des rôles qui sont réutilisés dans des playbooks

```
ansible.cfg
playbook1.yml
playbook2.yml
roles
  role1
    files
      file1
      file2
    tasks
      Main.yml
    templates
      Template1.j2
       Template2.j2
```

Containerisation : Le cas Docker



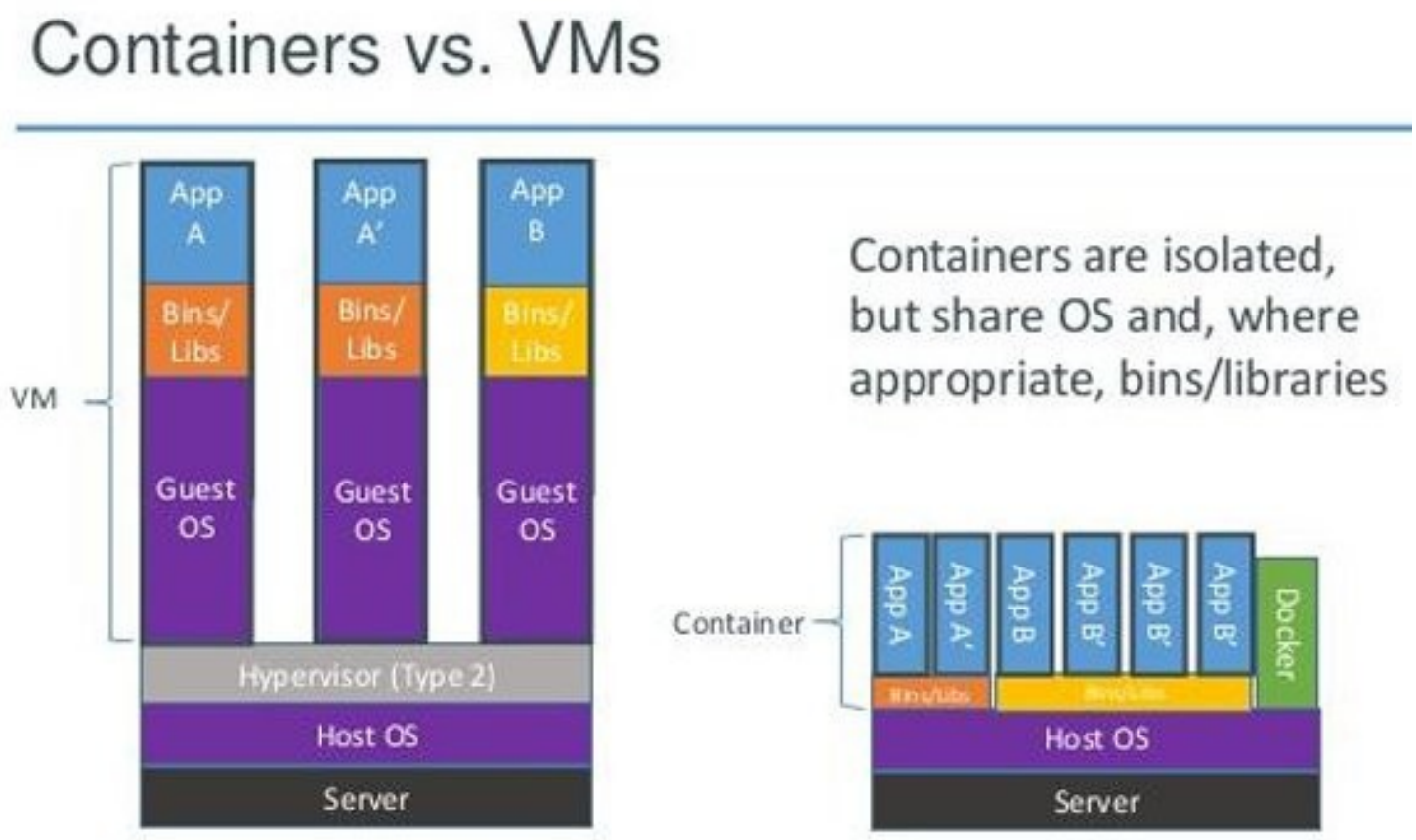
Introduction

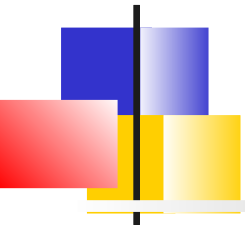
Plutôt que de virtualiser une machine complète, juste créer l'environnement d'exécution minimal pour fournir l'application, le service.

- il n'est plus question de simuler le matériel et les services d'initialisation du système d'exploitation sont ignorés.
- Seul le strict nécessaire réside dans le conteneur : l'application cible et quelques dépendances.

L'apport d'une solution de containerisation est l' isolation d'un processus dans un système de fichiers à part entière

Containers vs VMs





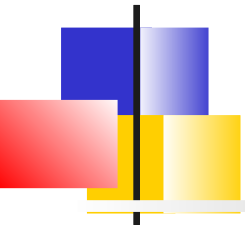
Avantages de la containerisation

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)



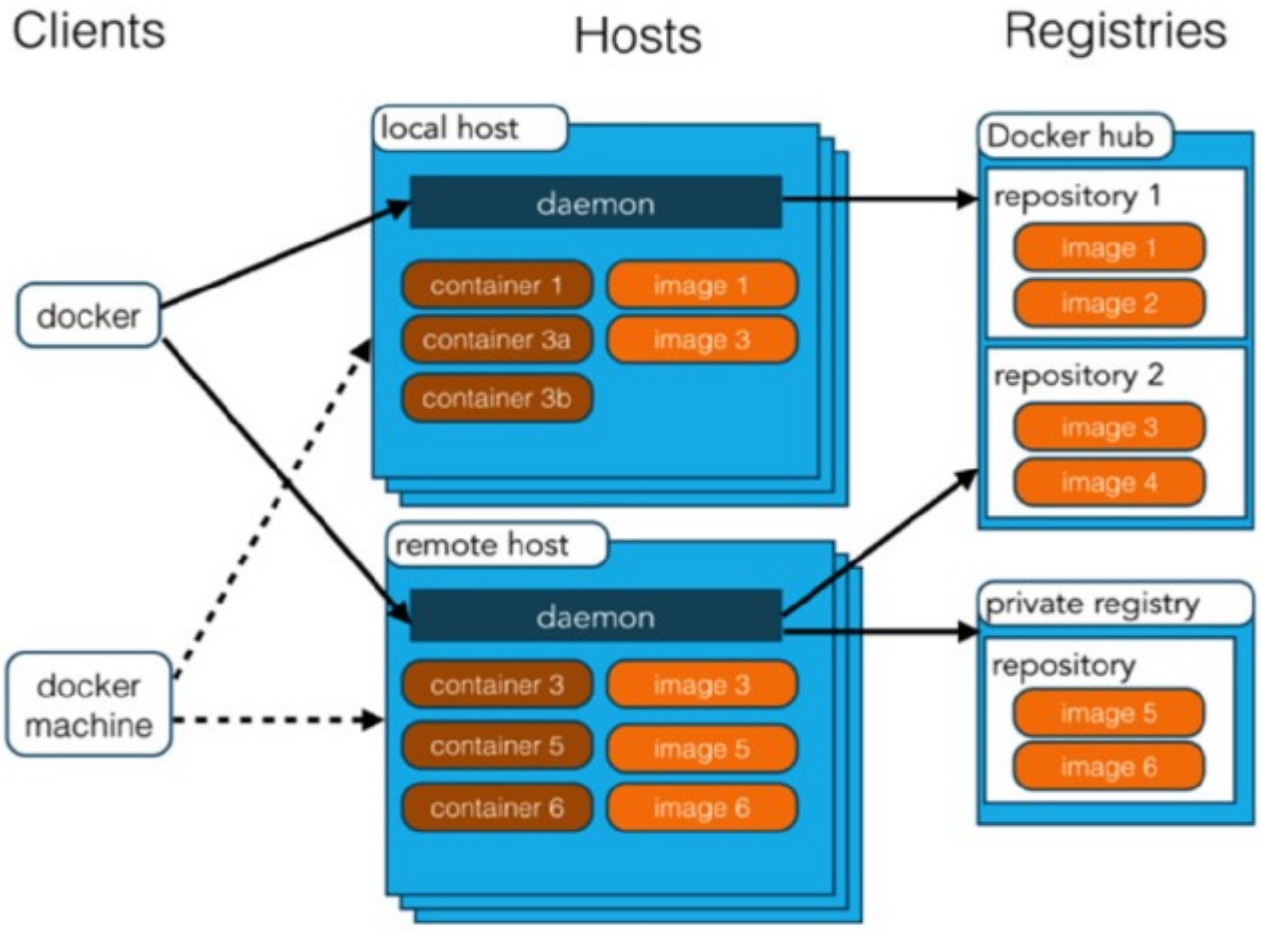
Impact sur le déploiement

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

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

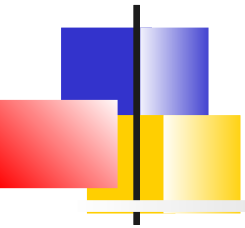
Il n’y a plus à proprement parler un déploiement brut sur un serveur mais plutôt l’utilisation d’orchestrateur de conteneurs.

Docker architecture



Commandes Docker

```
#Récupération d'une image
docker pull ubuntu
#Récupération et instanciation d’un conteneur
docker run hello-world
#Mode interactif
docker run -i -t ubuntu
#Visualiser les sortie standard d'un conteneur
docker  logs  <container_id>
#Conteneurs en cours
docker ps
#Toutes les exécutions de conteneurs (même arrêt)
docker ps -a
#Lister les images
docker images
#Construire une image à partir d’un fichier Dockerfile
docker build . -t monImage
#Committer les différences
docker commit <container_id> <image_name>
#Tagger une image d'un repository
docker tag <image_name>[:tag] <name>[:tag]
#Pousser vers un dépôt distant
docker  push <image_name>[:tag]
#Statistiques d’usage des ressources
docker stats
```

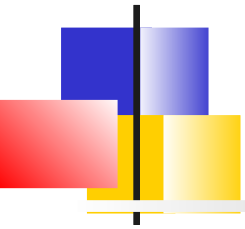


Exemple DockerFile

```
FROM ubuntu
MAINTAINER Kimbro Staken

RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise
universe" >> /etc/apt/sources.list
RUN apt-get update
RUN apt-get install -y nodejs
RUN mkdir /var/www

ADD app.js /var/www/app.js
EXPOSE 8080
CMD ["/usr/bin/node", "/var/www/app.js"]
```



Isolation du conteneur

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

Par défaut, ll est isolé

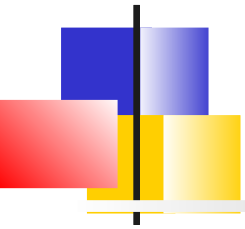
- De la machine hôtes
  => Montage de répertoires, association de ports TCP
- Des autres containers
  => docker-compose et définition de réseau

Communication avec la machine hôte

A l'instanciation d'un conteneur on peut :

- Associer un port exposé par le conteneur à un port local
  Option ***-p***
- Monté un répertoire du conteneur sur le système de fichier local.
  Option ***-v***

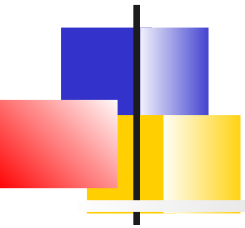
```
docker run -p 80:8080
           -v /home/jenkins:/var/lib/jenkins
            myImage
```



docker-compose

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

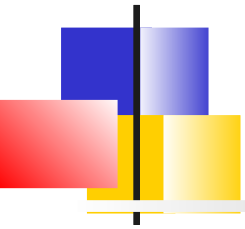
- Avec un simple fichier, on spécifie les différents conteneurs, les ports exposés, les liens entre conteneurs.
- Ensuite avec une commande unique, on peut démarrer, arrêter, redémarrer l'ensemble des services.



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

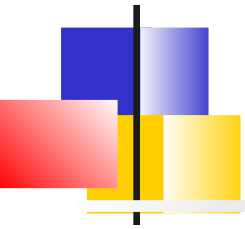
pull / ***push*** : Pull/push les images des services

restart : Redémarrage des services

scale : Fixe le nombre de container pour un service

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

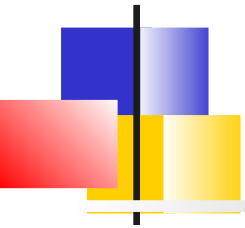
up : Création et démarrage de conteneurs



PIC et Docker

Une PIC utilise *docker* de plusieurs façons :

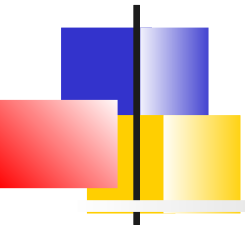
- Les workers utilisent des images pour exécuter des tâches du build ou pour exécuter des services utilisés par le build
- La pipeline a pour but de construire une image et de la pousser dans un dépôt.



Techniques d’intégration

L’intégration peut nécessiter :

- Préinstaller *docker* sur les nœuds esclaves
- Déclarer des registres d’images et les crédentiels pour y accéder
- Permettre du docker in docker. (Une container de build démarre un autre container).
  Utilisation de l’image *dind*



Application dockérisée

Un scénario désormais classique du CI/CD est:

1) Créer une image applicative
2) Exécuter des tests sur cette image
3) Pousser l'image vers un registre distant
4) Déployer l'image vers un serveur

En commande docker :

```
docker build -t my-image dockerfiles/
docker run my-image /script/to/run/tests
docker tag my-image my-registry:5000/my-image
docker push my-registry:5000/my-image
```

Exemple Jenkins

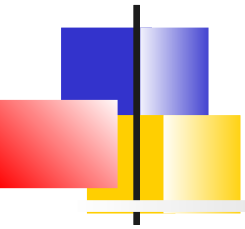
```
#!/usr/bin/env groovy

node {
    stage('checkout') {
        checkout scm
    }

    ..

    def dockerImage
      stage('build docker') {
        dockerImage = docker.build("dthibau/catalog",".")
        }

        stage('publish docker') {
            docker.withRegistry('https://registry.hub.docker.com', 'docker-login') {
                dockerImage.push 'latest'
            }
        }
    }
}
```



Orchestrateur de conteneurs

Orchestrateur, scalabilité et déploiement

Kubernetes

Intégration dans une pipeline CI/CD

Cloud dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison

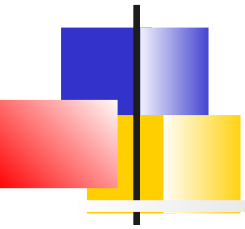
Code **Build** **Deploy**

Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab

Nexus,
Artifactory,
Archiva

Artefacts :
Image Docker

docker-swarm
Kubernetes
Clouds publics

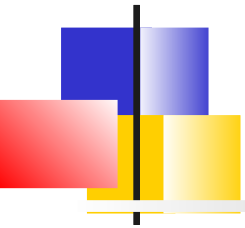


Principes de l’orchestration de conteneurs

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

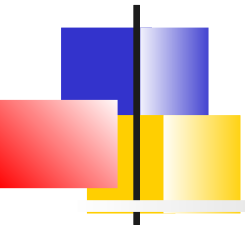
L’orchestrateur va alors trouver une machine disponible, démarrer l’application et faire le nécessaire pour qu’elle soit tout le temps accessible :

- Redémarrage si plantage
- Scaling si charge importante



Orchestrateur

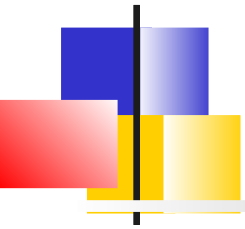
- Gère un pool de ressources :
  les nœuds, les volumes, les adresses Ips
- Il connaît la topologie du cluster
  - Ressources disponibles
  - Applications déployées
- Il connaît l'état de santé ...
  - ...des services
  - ...de la plateform



Apport de l'orchestrateur : Déploiement blue-green

Le déploiement **blue-green,** le service est toujours disponible même pendant une montée de version :

- La version n est déployée : Toutes les requêtes accédant au proxy sont routées vers les services exécutant la version n
- La version n+1 est déployée et s'exécute en même temps que la version n. Certaines requêtes sont dirigées vers la n+1 :
  - Tests automatisés d'une pipeline CI
  - Tests manuels via machine interne au réseau
  - Canary testing : Beta-testeurs sont dirigés vers la n+1
- La version n+1 est considéré comme complètement opérationnelle.
  Lorsque, la version n a répondu à toutes les requêtes en cours, elle est supprimée de l'environnement de production



Apport de l’orchestrateur pour les micro-services

Les architectures micro-services nécessitent des services techniques. Ceux-ci peuvent être apportés par l’orchestrateur :

- Un service de **discovery** permettant de localiser les instances des services déployés sur le réseau interne de l’orchestrateur
- Un service de **proxy** et de **répartition de charge** permettant d’offrir des points unique d’accès aux services déployés
- Des services de **monitoring**
- Un service de gestion centralisée des **configuration et des clés**
- Des services de **résilience** : Redémarrage, scaling

Kubernetes

Kubernetes

OpenSource : Provient des projets Google's *Borg* et *Omega*

Construit depuis le départ comme un ensemble de composants faiblement couplés orientés vers **le déploiement, la surveillance et le scaling de charges de travail** (workload)

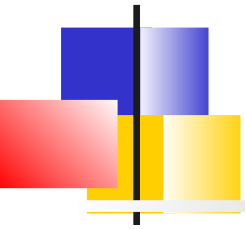
Kubernetes (Mai 2020) :

91000 commits

+2000 contributeurs

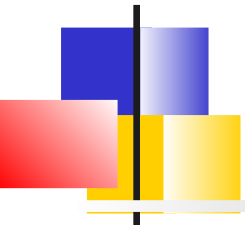
+60k* sur GitHub

Géré par la Cloud Native Computing Foundation (Groupe neutre)



Caractéristiques

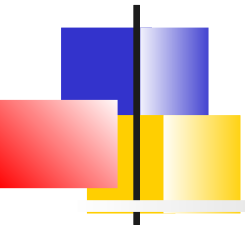
- Peut être vu comme le *noyau linux des systèmes distribués*
- Fournit une abstraction du matériel sous-jacent via une API Rest afin que des charges de travail soit consommées par un pool de ressources partagées
- Agit comme un moteur qui fait converger l’état courant d’un système vers un système désiré
- Gère des applications/déploiements pas des machines
- Tous les services gérés sont clusterisés et load-balancés par nature



Auto-correctif

Kubernetes va TOUJOURS essayer de diriger le cluster vers son état désiré.

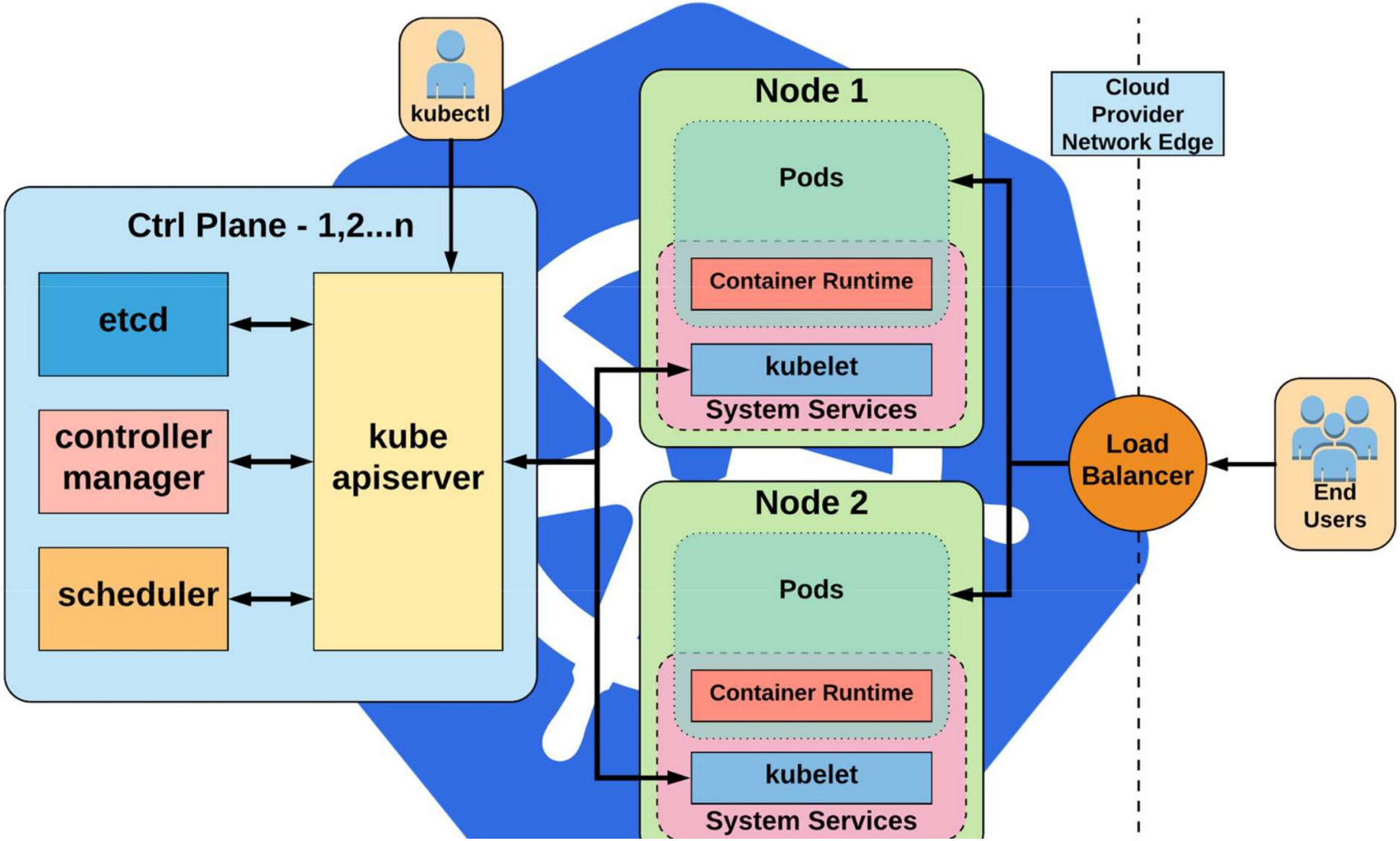
- **Moi**: «Je veux que 3 instances de Redis toujours en fonctionnement."
- **Kubernetes**: «OK, je vais m'assurer qu'il y a toujours 3 instances en cours d'exécution. "
- **Kubernetes**: «Oh regarde, il y en a un qui est mort. Je vais essayer d'en créer un nouveau. "



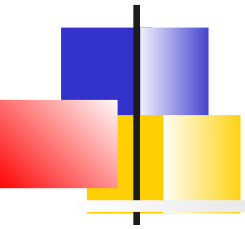
Fonctionnalités applicatives

- Scaling automatique
- Déploiements Blue/Green
- Démarrage de jobs planifiés
- Gestion d’application Stateless et Stateful
- Méthodes natives pour la découverte de services
- Intégration et support d’applications fournies par des tiers (*Helm*)

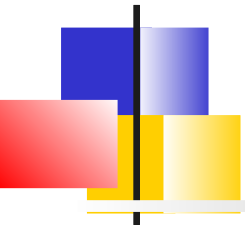
Architecture cluster



Règles réseau à l’intérieur du cluster



- Tous les conteneurs d'un *pod* peuvent communiquer entre eux sans entrave via *localhost*
- Tous les *pods* peuvent communiquer avec tous les autres *pods* sans NAT.
- Tous les nœuds peuvent communiquer avec tous les pods (et inversement) sans NAT.
- L’adresse IP avec laquelle se voit un *pod* est la même adresse que les autres voient de lui.
- Il est possible de partitionner un cluster en plusieurs clusters avec des espaces de noms



API

L’interaction se fait par une API Rest très riche.

L’API est très cohérente et tous les appels suivent le même format

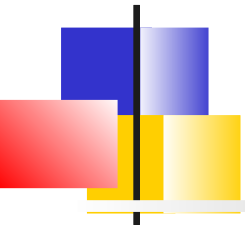
Format:

```
/apis/<group>/<version>/<resource>
```

Examples:

```
/apis/apps/v1/deployments
/apis/batch/v1beta1/cronjobs
```

L’outil ***kubectl*** et le format ***yaml*** sont les plus appropriés pour effectuer les requêtes REST

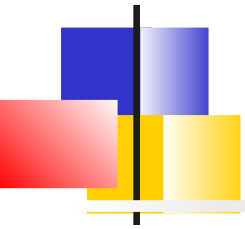


Principes

L'API est une API Rest, elle permet principalement des opérations CRUD sur des ***ressources***

En partculier, le client *kubectl* propose les commandes :

- ***create*** : Créer une ressource
- ***get*** : Récupérer une ressource
- ***edit***/***set*** : Mise à jour d'une ressources
- ***delete*** : Suppression d'une ressource



Ressources applicatives

Les principales ressources d'une application sont :

- ***deployment*** : Un déploiement, les déploiements font référence à des *ReplicaSet*, ils peuvent être historisés
- ***replicaSet*** : Ils définissent le nombre d'instances maximales pour une image de conteneur applicative
- ***pod*** : Ce sont des conteneurs qui s'exécutent, ils sont distribués sur les nœuds par le scheduler de *Kubernetes*
- ***service*** : Ce sont des point d'accès stable à un service applicatif

pod

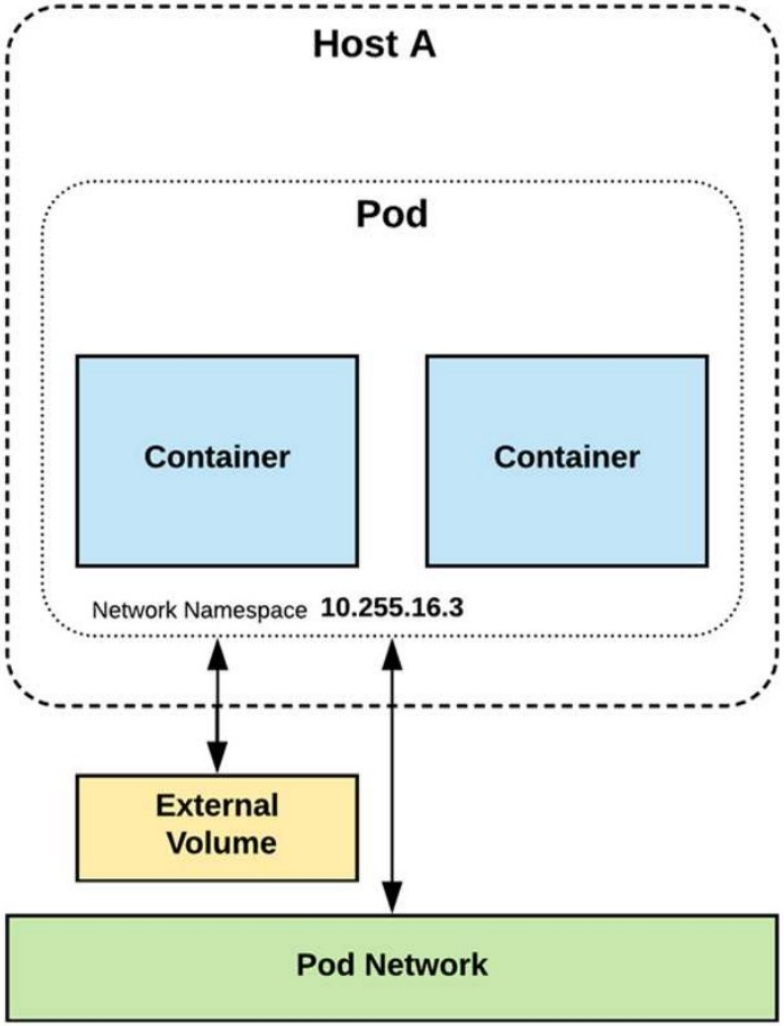
Un ***pod*** est la plus petite unité de travail

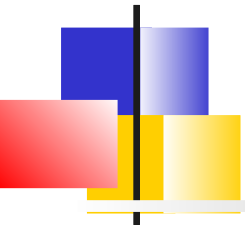
Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

- Sont terminés
- Ont échoués
- Sont expulsés par manque de ressources





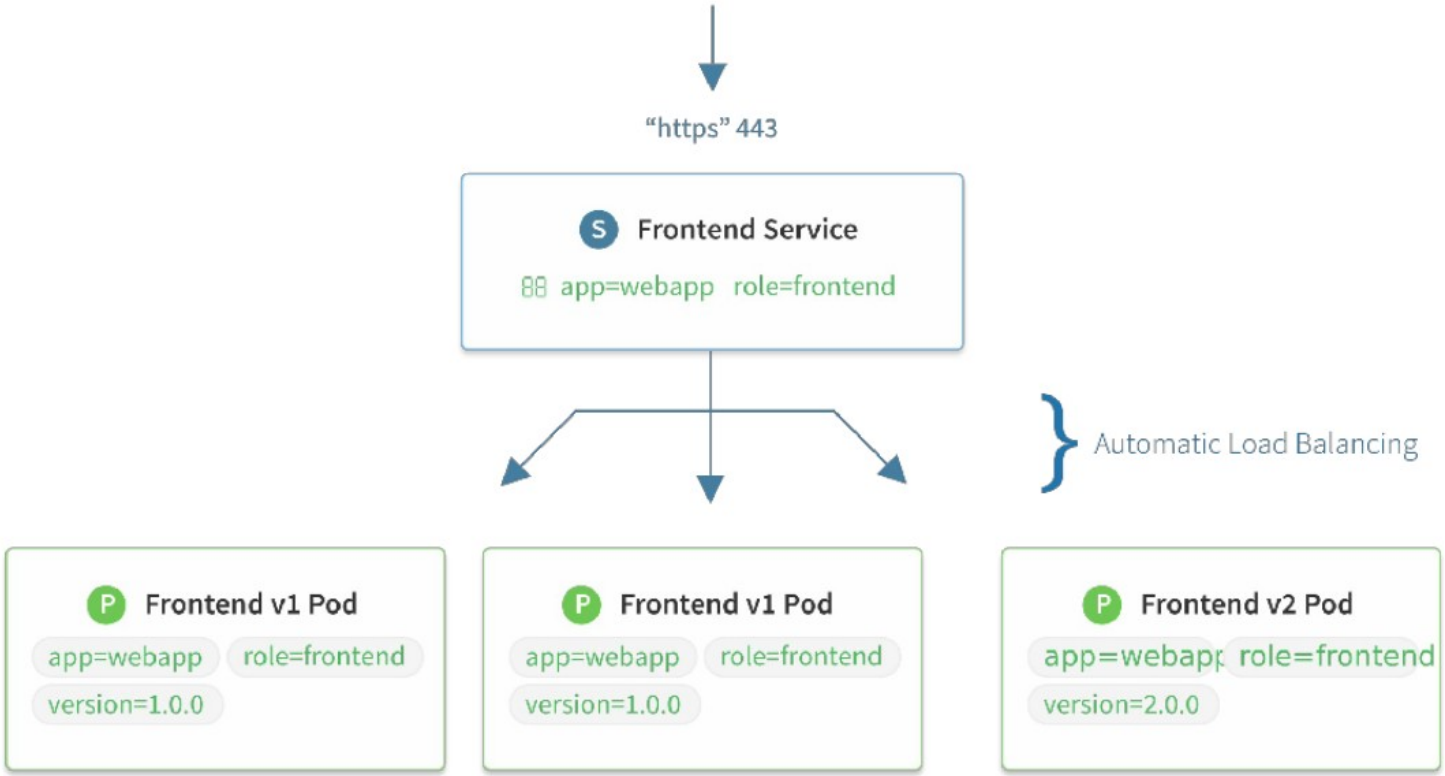
Services

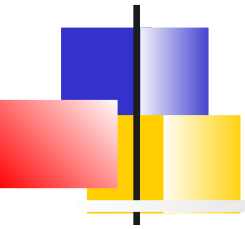
Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *pods*.

Ressource durable. Les services ne sont pas éphémères :

- IP statique du cluster
- Nom DNS statique (unique à l’intérieur d’un espace de nom)

Service


"https" 443
Frontend Service
app=webapp role=frontend
Automatic Load Balancing
Frontend v1 Pod
app=webapp
role=frontend
version=1.0.0
Frontend v1 Pod
app=webapp
role=frontend
version=1.0.0
Frontend v2 Pod
app=webapp
role=frontend
version=2.0.0



Ressource *deployment*

Exemple description d'un déploiement:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
    spec:
      containers:
      - image: dthibau/annuaire
        name: annuaire
```

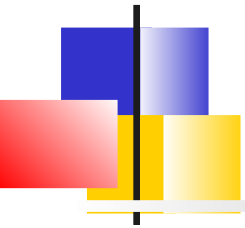
A partir de ce type de fichier *.yml*, on peut créer la ressource via :

kubectl create -f ./my-manifest.yaml

Exemple service

Un service nommé *my-service* qui représente tous les pods ayant le **label *app=MyApp*** et qui mappe son port 80 vers le port 80 des pods

```
kind: Service
  apiVersion: v1
  metadata:
    name: my-service
  spec:
    selector:
      app: MyApp
      ports:
        - protocol: TCP
          port: 80
          targetPort: 80
```



Commandes *kubectl*

create : Crée une ressource à partir d'un fichier ou de stdin.

expose : Exposer un nouveau service

execute : Exécuter une image particulière sur le cluster

set : Mettre à jour des attributs sur une ressource

get : Afficher 1 ou plusieurs ressources

edit : Éditer une ressource

delete : Supprimer des ressources

describe : Afficher les détails sur une ou plusieurs ressources

logs : Afficher les logs d'un container

attach : S'attacher à un container qui s'exécute

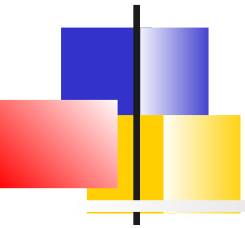
exec : Exécuter une commande dans un container

port-forward : Forward un ou plusieurs ports d'un pod

cp : Copier des fichiers entre conteneurs

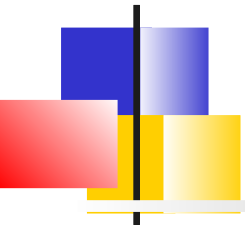
auth : Inspecter les autorisations

...



Exemples

```
# Affiche les paramètres fusionnés de kubeconfig
kubectl config view
# Liste tous les services d'un namespace
kubectl get services
# Liste tous les pods de tous les namespaces
kubectl get pods --all-namespaces
# Description complète d’un pod
kubectl describe pods my-pod
# Supprime les pods et services ayant le noms "baz"
kubectl delete pod,service baz
# Affiche les logs du pod (stdout)
kubectl logs my-pod
# S’attacher à un conteneur en cours d'exécution
kubectl attach my-pod -i
# Exécute une commande dans un pod existant (un seul conteneur)
kubectl exec my-pod -- ls /
# Écoute le port 5000 de la machine locale et forwarde vers le port 6000 de my-pod
kubectl port-forward my-pod 5000:6000
```



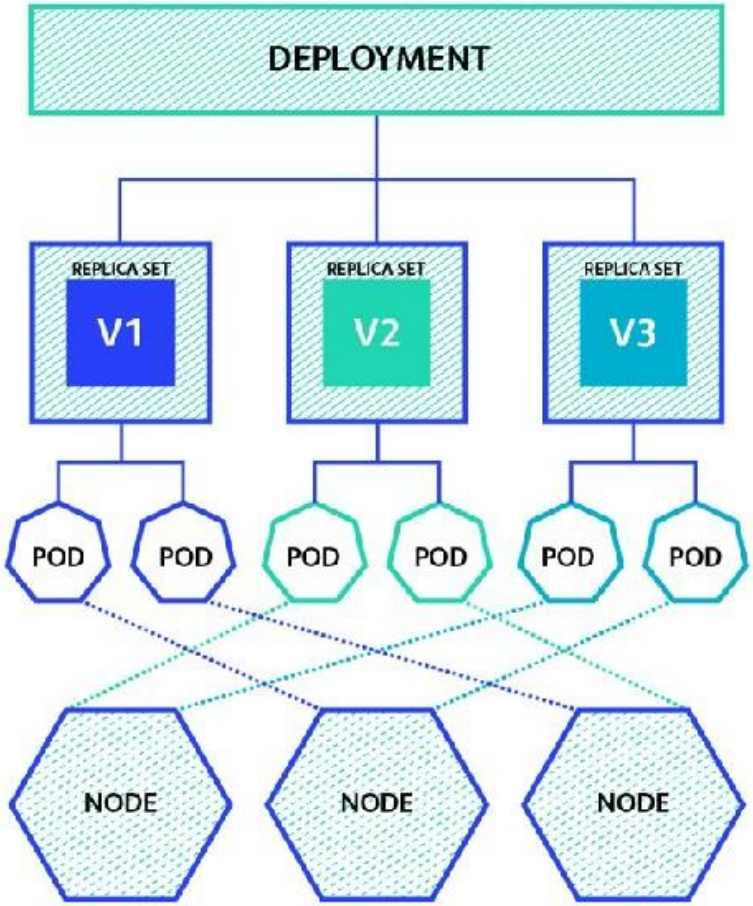
Déploiement

La ressource ***deployment*** permet de manipuler un ensemble de *Replicaset (ensemble de conteneurs répliqués)*

Les principales actions que l'on peut faire sur un déploiement sont :

- Le **rollout**: Création/Mise à jour entraînant la création des pods en arrière-plan
- Le **rollback**: Permet de revenir à une ancienne version des *ReplicaSets*
- La **scalabilité** horizontale : Permet de mettre en échelle l'application horizontalement
- La mise en pause
- La suppression de vielles versions

Versions de ReplicaSet



Commandes de déploiement *kubectl*

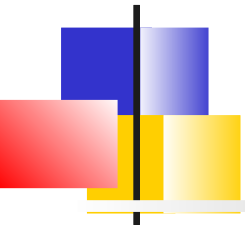
```
# Mettre à jour une image dans un déploiement existant
# Enregister la mise à jour
kubectl set image deployment/nginx-deployment
nginx=nginx:1.9.1 –record

# Regarder le statut d’un rollout
kubectl rollout status deployment/nginx-deployment

# Obtenir l’historique des révisions
kubectl rollout history deployment/nginx-deployment

# Roll-back sur la version précédente
kubectl rollout undo deployment/nginx-deployment

# Scaling
kubectl scale deployment/nginx-deployment --replicas=10
```



Scheduler et Workload

Les actions de l'API sont souvent asynchrones

Pour *Kubernetes*, ces ordres sont considérés comme des **workloads** à exécuter via le scheduler.

Les *workload* sont visibles via l'API, elles comportent 2 blocs de données :

- ***spec*** : La spécification de la ressource
- ***status*** : Est géré par *Kubernetes* et décrit l'état actuel de l'objet et son historique.

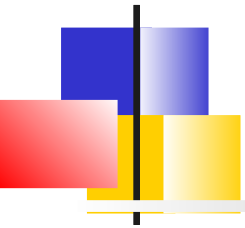
Exemple

Example Object

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

Example Status Snippet

```yaml
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:52Z
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: PodScheduled
```



Autres ressources du cluster

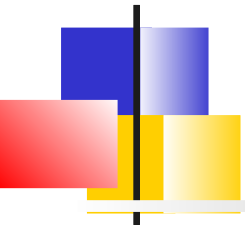
ClusterRole : Rôle avec permissions sur l'API

VolumePersistant : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

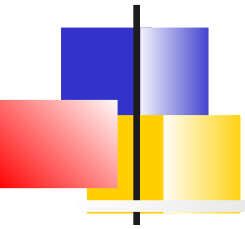
Secrets : Stockage de crédentiels



Écosystème *Kubernetes*

De nombreux outils peuvent être ajoutés à une installation coeur de Kubernetes :

- ***CoreDNS*** : Permet de déclarer dans un DNS interne les services (qui deviennent accessibles via leur nom)
- ***Helm*** : Système de gestion de package permettant d'automatiser l'installation d'autres outils (ressources Kubernetes)
- ***Promotheus*** : Monitoring du cluster, généralement associé à Grafana
- ***Ingress*** : Permettant d'exposer les services à l'extérieur du cluster
- ***Istio*** : Maillage de service (services mesh), ajoute un proxy sur chaque pod qui sécurise, monitore, gère les communications inter-pods



Distribution Kubernetes

Kubernetes est disponible en OpenSource mais une installation nécessite encore beaucoup d'expertise … et beaucoup de ressources

Kubernetes est donc proposé par les acteurs du cloud

- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Services
- Google Kubernetes Engine
- Digital Ocean
- …

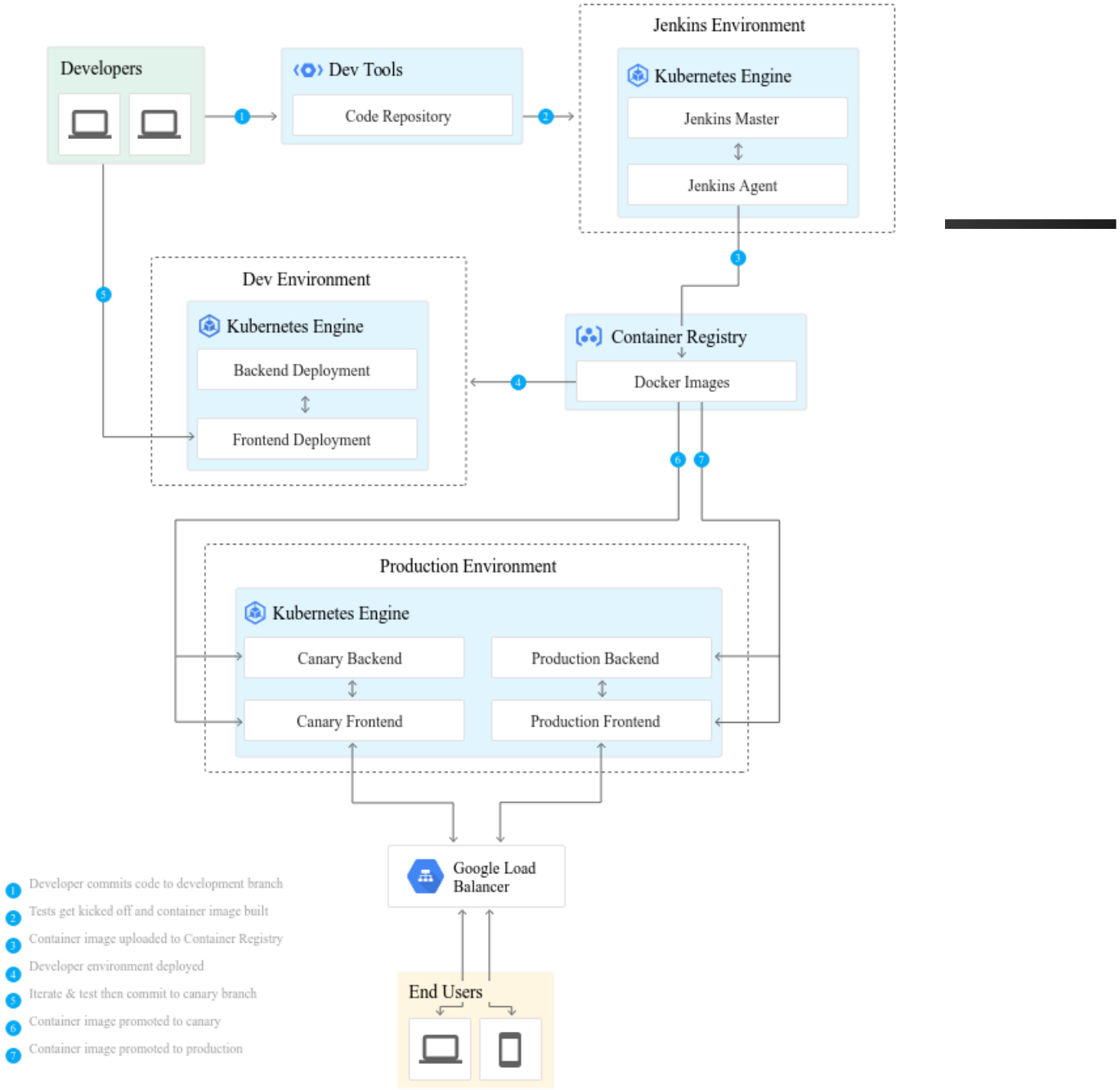
Il est également disponible en version « dev » mono-nœud : *microk8s, minikube*

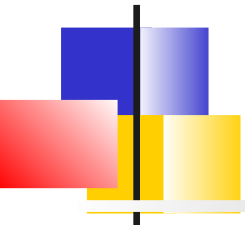
L'outil *Rancher* permet de gérer graphiquement plusieurs installation

TP : Déploiement image sur Google Kubernetes Engine

Kubernetes dans la pipeline CI/CD

https://cloud.google.com/solutions/continuous-delivery-jenkins-kubernetes-engine?hl=fr



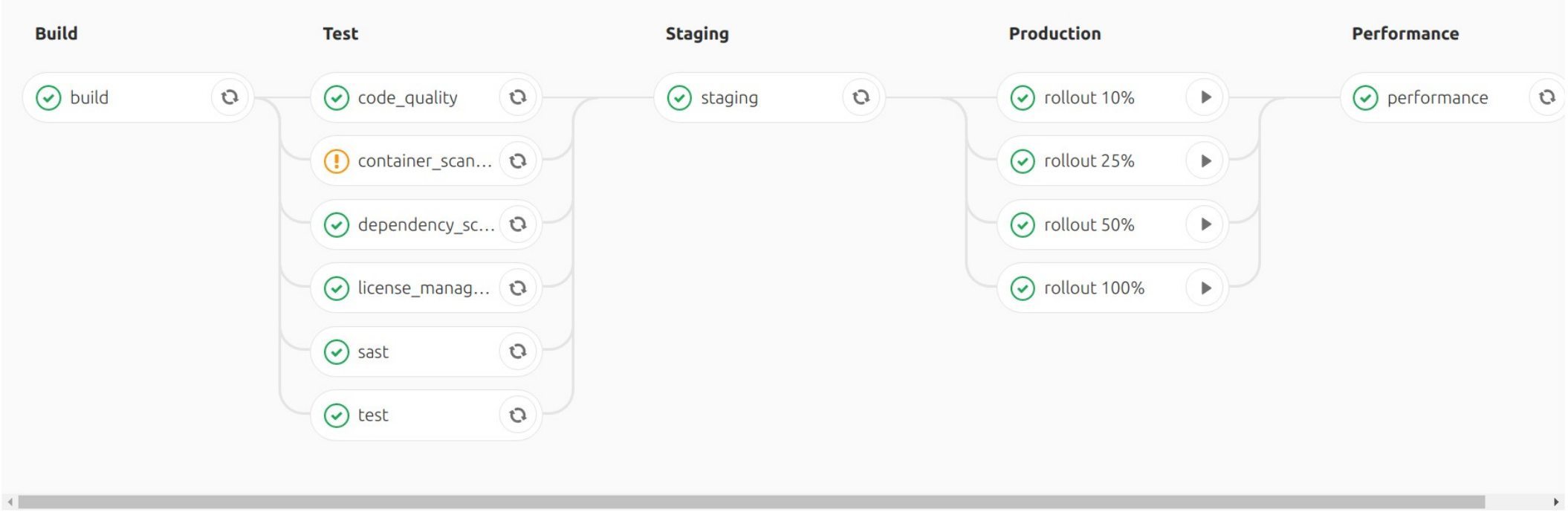


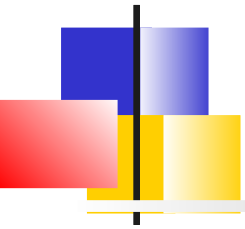
Exemple AutoDevOps GitlabCI

AutoDevOps est la pipeline par défaut qu'essaye d'appliquer GitLabCI, il utilise

- Docker pour builder, tester construire le conteneur
- Base Domain (Pour les review apps) : Un domaine configuré avec un DNS * utilisé par tous les projets
- Kubernetes (GKE ou Existant) : Pour les déploiements
- Prometheus : Pour obtenir les métriques
- Helm : pour installer les outils nécessaires sur le cluster Kubernetes

Pipeline AutoDevops de Gitlab CI





Jenkins X

Nouveau projet de Jenkins/Cloudbees basé sur *Kubernetes*

- CI/CD automatisé : Jenkins applique des pipelines tout seul !
- Chaque équipe a des environnements dédiés aux modifications en cours
- Notion de *pull-request* pour promouvoir un environnement

TP : Démonstration Auto-DevOps GitLab-CI
Déploiement Snapshot sur un microk8s