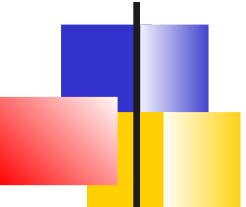


# Usine Logicielle et Livraison Continue : Mise en oeuvre

---

David THIBAU – 2020

david.thibau@gmail.com



# Agenda

---

## Introduction

- Agilité, DevOps, Outils, CI/CD, Micro-services

## Pilotage

- Sprints, Kanbans

## Le SCM

- Workflow de collaboration

## Les outils de build

- Caractéristiques, exemple Maven

## Les dépôts

- SNAPSHOTS, Release

## Les tests et l'analyse statique

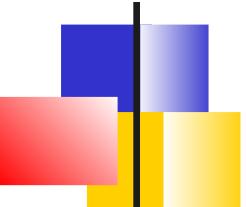
- Typologie des tests
- Analyse statique

## Pipeline CI/CD

- Plateforme d'IC
- Pipelines typiques

## Déploiements

- Automatisation et Alternatives



# Agenda

---

## Introduction

- CI, DevOps, Tests, Build, Outils associés

## Outils de SCM

- Concepts, commandes de base, branches, workflow de collaboration

## Maven

- Pom, Archetype, Dépôts, Dépendance, plugins, Multi-modules

## Release, Nexus

- Phase deploy, plugin Release, Intégration Nexus, Apports de Nexus

## Qualité avec SonarQube

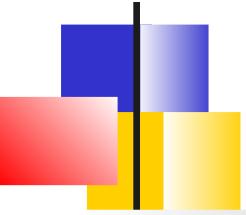
- Architecture, Concepts, Calcul de métriques, Mise en place

## Le serveur Jenkins

- Offre Jenkins, UI, Jobs, Configuration, Plugins, Dimensionnement

## Mise en place de pipeline

- Pipeline Legacy, Plugin pipeline Syntaxe déclarative, Intégration de Sonar/Nexus



# Introduction

---

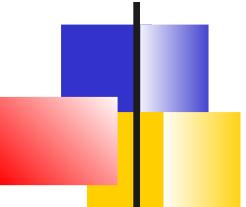
## **Contraintes de l'agilité**

Approche DevOps

Pipeline CI/CD

Cycle de vie du code et outils

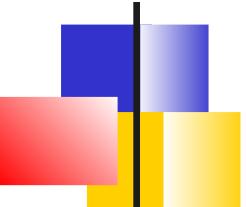
Influence du DevOps sur les  
architectures



# L'agilité

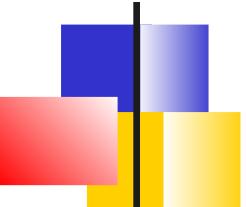
---

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



# Les 12 principes

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



# Méthodes agiles

---

2 facteurs communs à toutes les méthodes agiles :

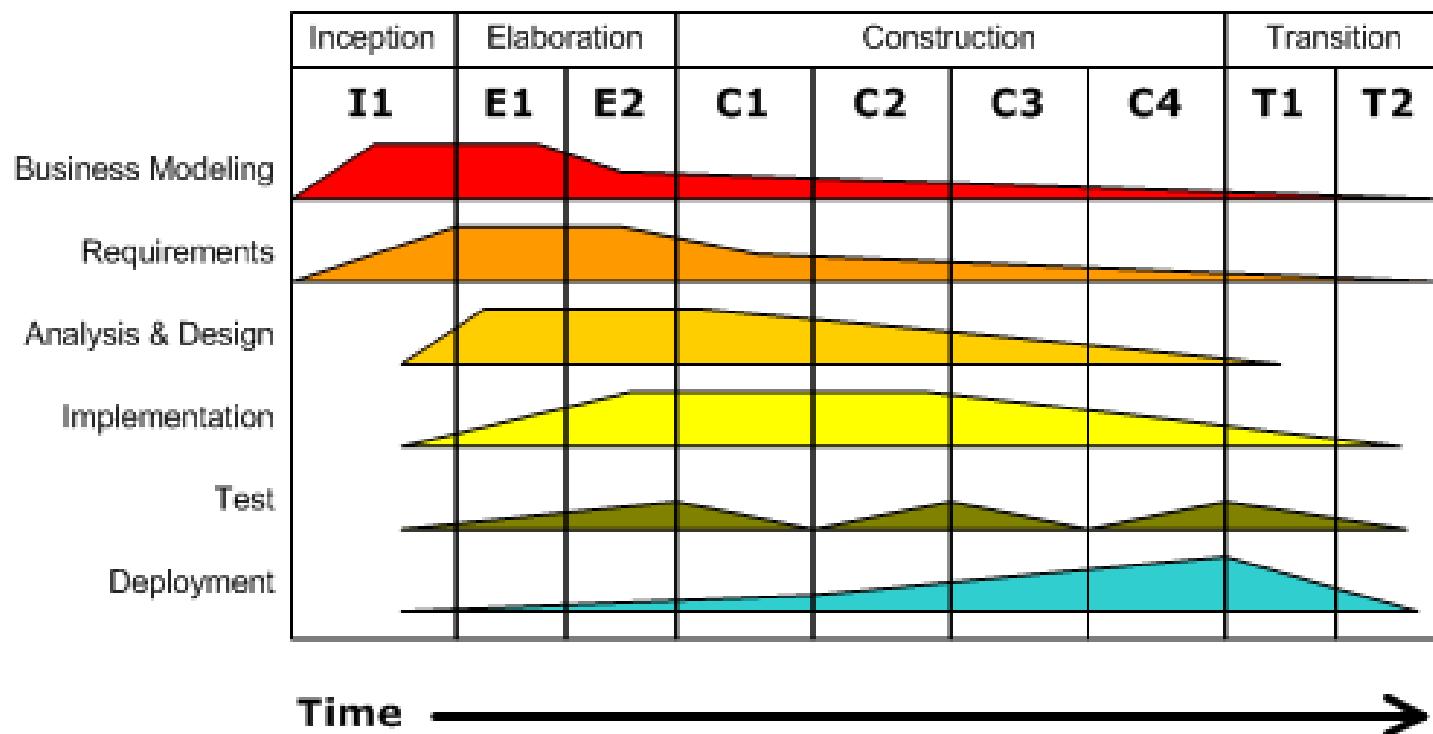
- la mise en place de pratiques **itératives**,
- des projets plus **petits**  
=> des équipes de dév. de plus en plus petites, des périmètres fonctionnels limités

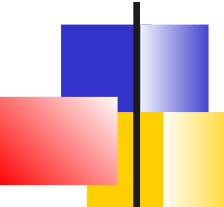
→ C'est ce l'on retrouve dans *RUP, XP Programming, Scrum*

# Unified Process

## Iterative Development

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

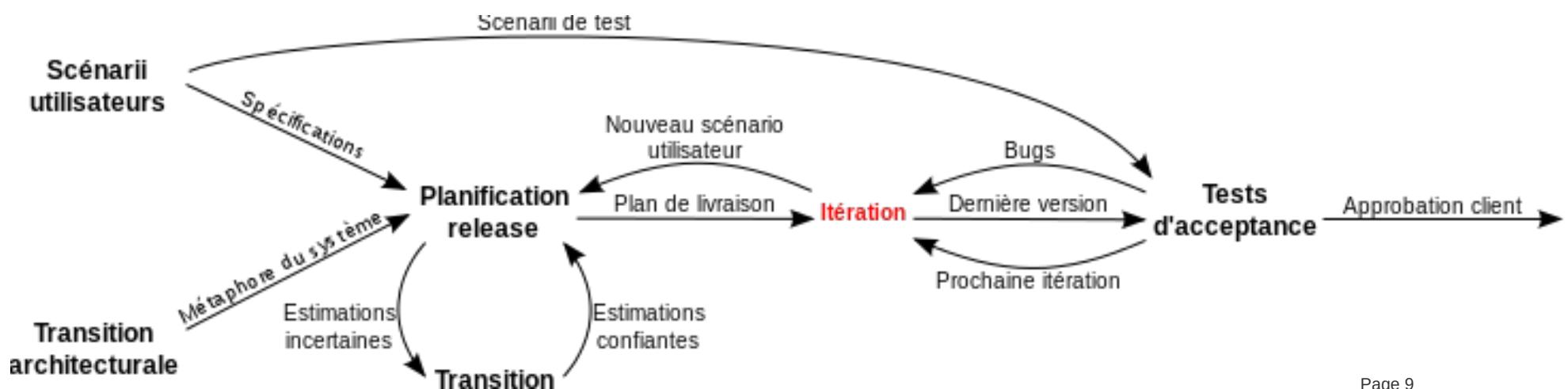


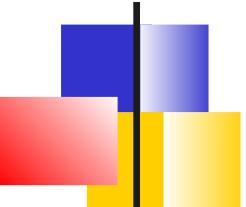


# XP: eXtreme Programming

Agilité, Petites équipes, Tout à l'extrême :

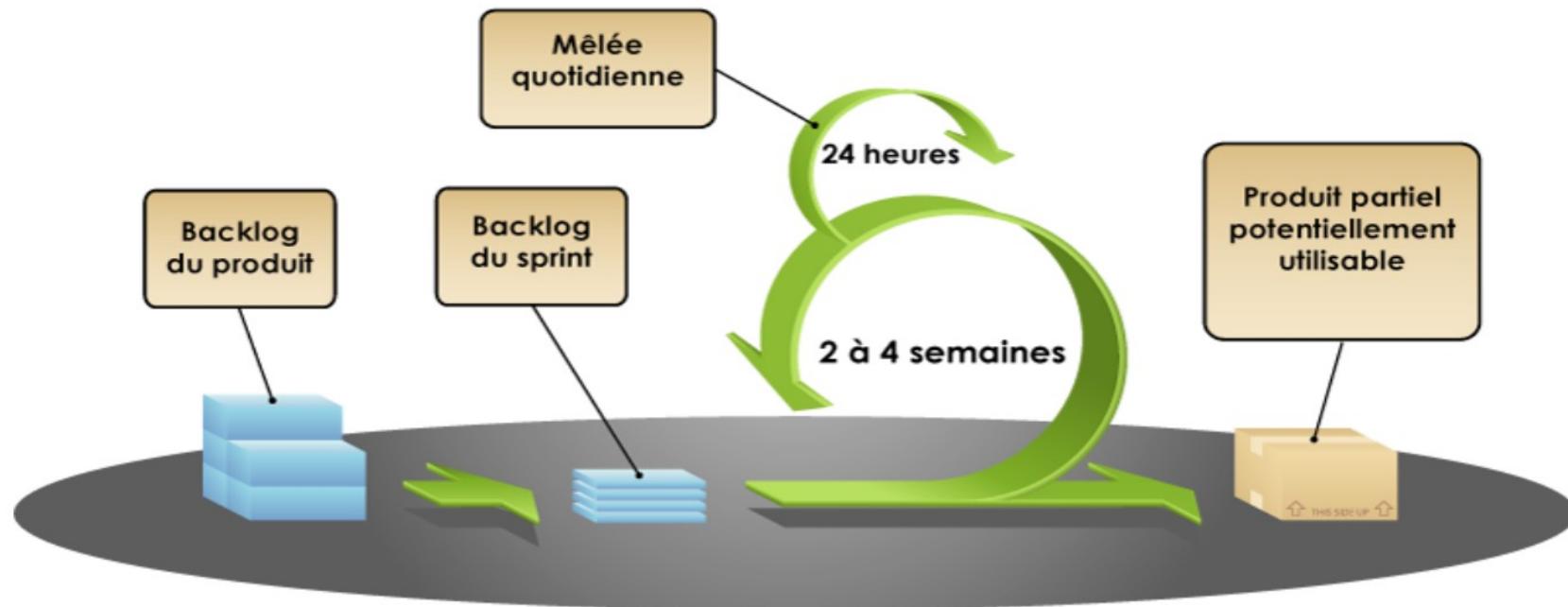
- Revue de code en permanence (par un binôme)
- Tests systématiques
- Refactoring en continu
- Toujours la solution la plus simple ;
- Evolution des métaphores ;
- Intégration continue ;
- Cycles de développement très rapides pour s'adapter au changement.

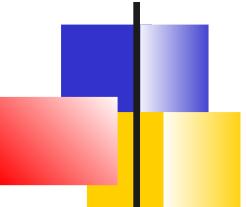




# Scrum : rythme

---





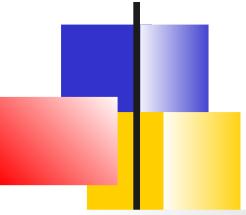
# Contraintes sur les déploiements

---

L'agilité suppose d'augmenter la fréquence des déploiements dans les différents environnements : intégration, recette, production afin :

- De mieux piloter le projet
- De prendre en compte rapidement les retours utilisateurs

Problème : Avant DevOps, l'organisation des services informatiques ne facilitait pas les déploiements



# Introduction

---

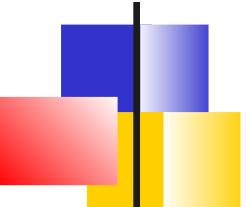
Contraintes de l'agilité

**Approche DevOps**

Pipeline CI/CD

Cycle de vie du code et outils

Influence du DevOps sur les  
architectures

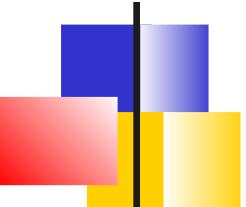


# Cycle de vie d'un logiciel

---

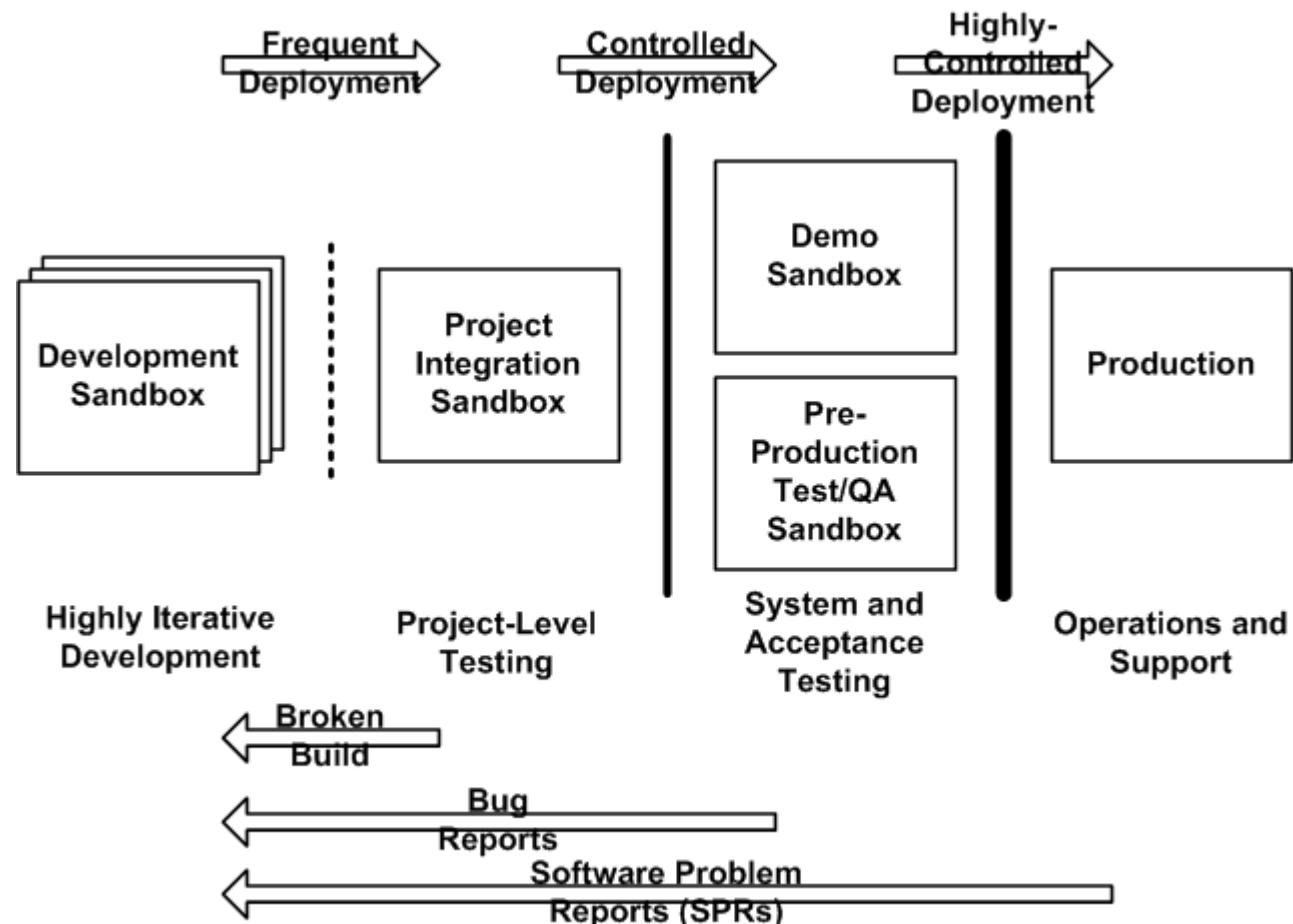
Le cycle de production d'un logiciel passe par plusieurs étapes correspondant à plusieurs environnement :

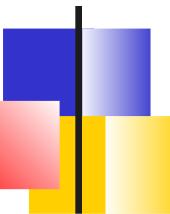
- **Développement** : Poste du développeur
- **Intégration** : Intégration des modifications de toute l'équipe
- **QA** : Environnement proche de la production permettant la qualification des releases
- **Production** : Exploitation, Support, Maintenance



# Avant DevOps

---





# Disparité des environnements

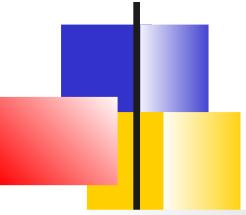
---

Les environnements ne peuvent que différer :

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

Ces environnements sont traditionnellement gérés et utilisés par des équipes distinctes qui ... souvent communiquent peu

Les équipes ont de plus des objectifs différents



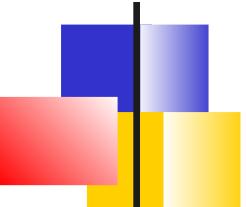
# Objectifs des développeurs

---

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

Les fonctionnalités sont testées en isolation :

- Tests unitaires
- Accès non concurrents
- Base de données légères
- Configuration par défaut des serveurs
- Infrastructure et OS différents que la cible
- Compilation avec mode debug
- ...

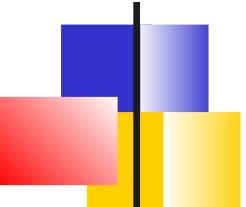


# Objectifs des opérations

---

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

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



# Le constat DevOps

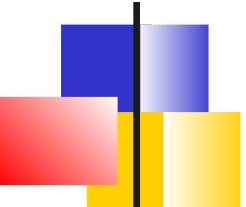
---

## Le constat DevOps :

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

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

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



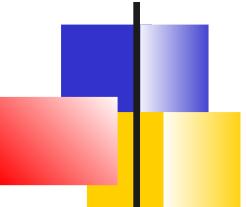
# Approche *DevOps*

---

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

Cela impose :

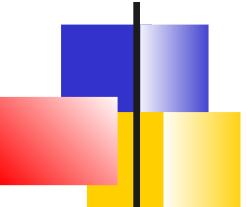
- la réunion des équipes
- la montée en compétence des différents profils.



# Pratiques *DevOps* (1)

---

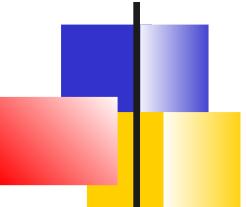
- Un déploiement régulier des applications dans les différents environnements.  
La seule répétition contribuant à fiabiliser le processus ;
- Un décalage des tests "vers la gauche".  
Autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- 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

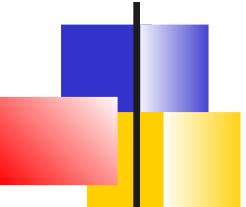


## « As Code »

---

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

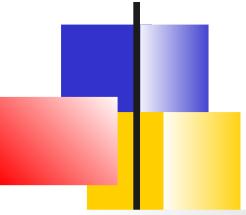
On parle de *Build As Code*, *Infrastructure As Code*, *Pipeline As Code*, *Load Test As Code*, ...



# 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



# Introduction

---

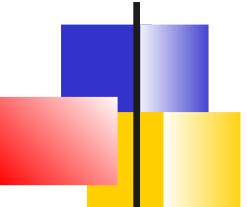
Contraintes de l'agilité

Approche DevOps

**Pipelines CI/CD**

Cycle de vie du code et outils

Influence du DevOps sur les  
architectures



# Avant l'intégration continue

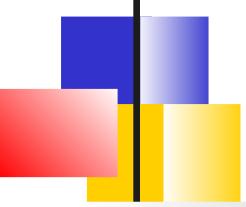
---

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



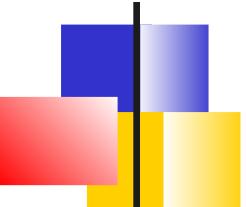
# Plateforme d'intégration continue (PIC)

---

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 **corrigeant le problème ASAP**



# Les tests pour le CD

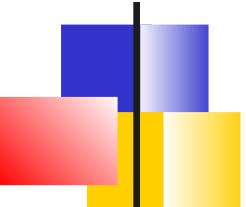
---

Après avoir atteint l'environnement d'intégration, on cherche à automatiser le déploiement dans les différents environnements QA, PROD.

Pour cela, il est nécessaire de renforcer les tests automatisés.

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

Tous les types de tests vont être appliqués : Unitaires, Intégration, Fonctionnel, Performance, Acceptance



# Outil de communication et de motivation

---

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

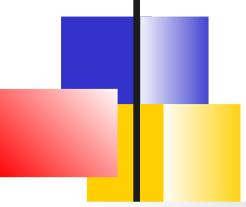
- Rapports de tests et d'analyse qualité
- Couverture fonctionnelle et avancement du projet
- Documentation
- Accès à des plate-formes de recette

=> 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



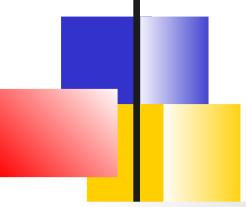
# Livraison continue

## *Continuous delivery*

---

A chaque ajout de code, la PIC met à disposition des releases potentiellement déployable en production

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



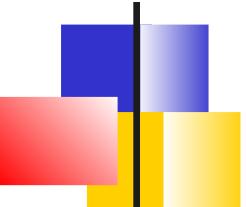
# Déploiement continu

## *Continuous 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 de l'intégration continue appelée **déploiement continu.**

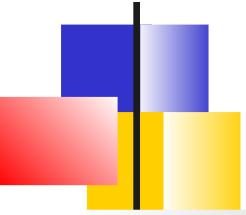


# Phases de la mise en place

---

La mise en place de l'intégration/déploiement continu ne se fait pas en 1 jour. En général, cela passe par plusieurs phases :

1. Pas de serveur de build
2. Serveur de build, tests unitaires/intégration et packaging
3. Déploiement automatisé en environnement de recette et Obtention des métriques
4. Renforcement des tests (fonctionnel, performance), distinction environnement d'intégration et de QA
5. Déploiement de release dans dépôt d'artefacts
6. Automatisation du déploiement en production,
7. Tests d'acceptance et Déploiement continu



# Introduction

---

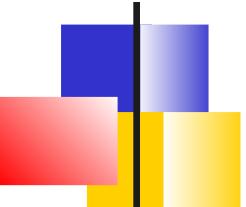
Contraintes de l'agilité

Approche DevOps

Pipelines CI/CD

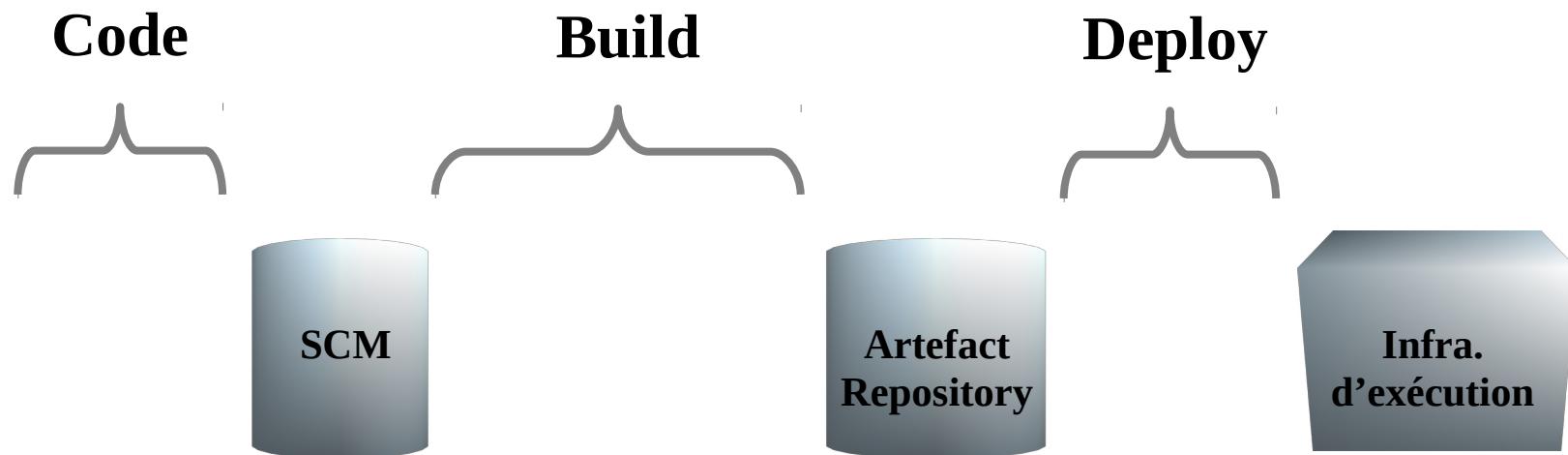
## **Cycle de vie du code et outils**

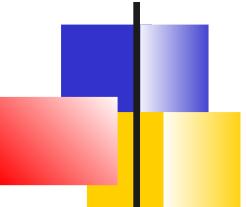
Influence du DevOps sur les  
architectures



# 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





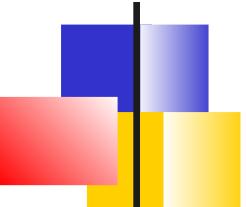
# Types d'outils

---

Plusieurs types d'outils s'intègrent dans le processus de CI/CD :

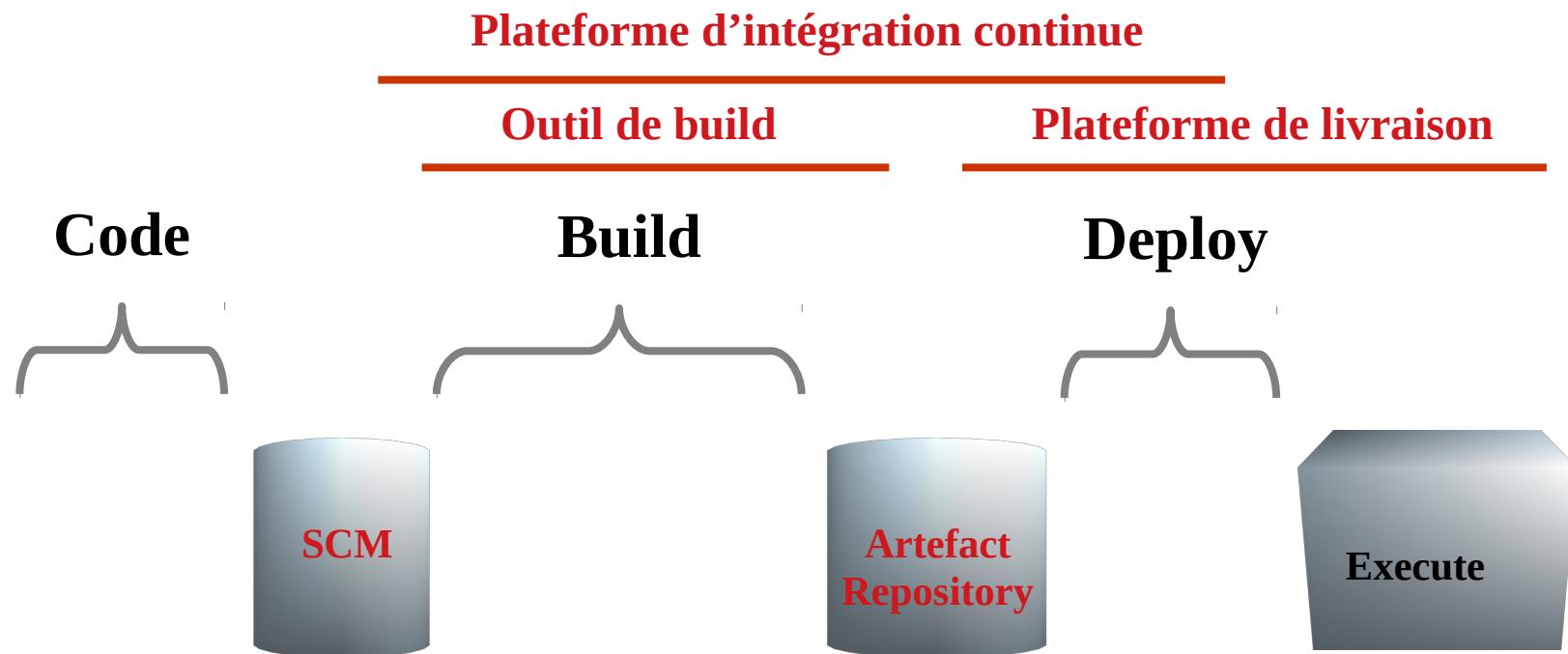
- Le **SCM** : centralise le code source (appliquatif, code de build, des tests, scripts de provisionnement, ...). Il gère plusieurs branches ou versions plus ou moins stables
- **L'outil de build** : Exécute les étapes nécessaires à la production de l'artefact
- La **PIC** qui à partir des branches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

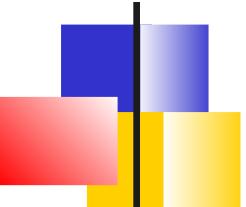
Bien que ces groupes soient clairement identifiés, les outils sont parfois capables de gérer plusieurs aspects du processus



# Outils et Cycle de vie

---





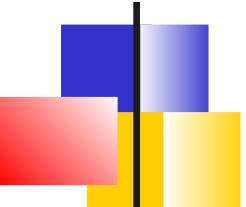
# Dépôts et formats des artefacts

---

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

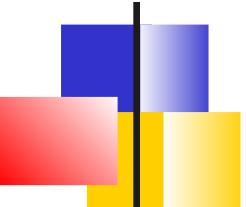
---

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 utiliser
- 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

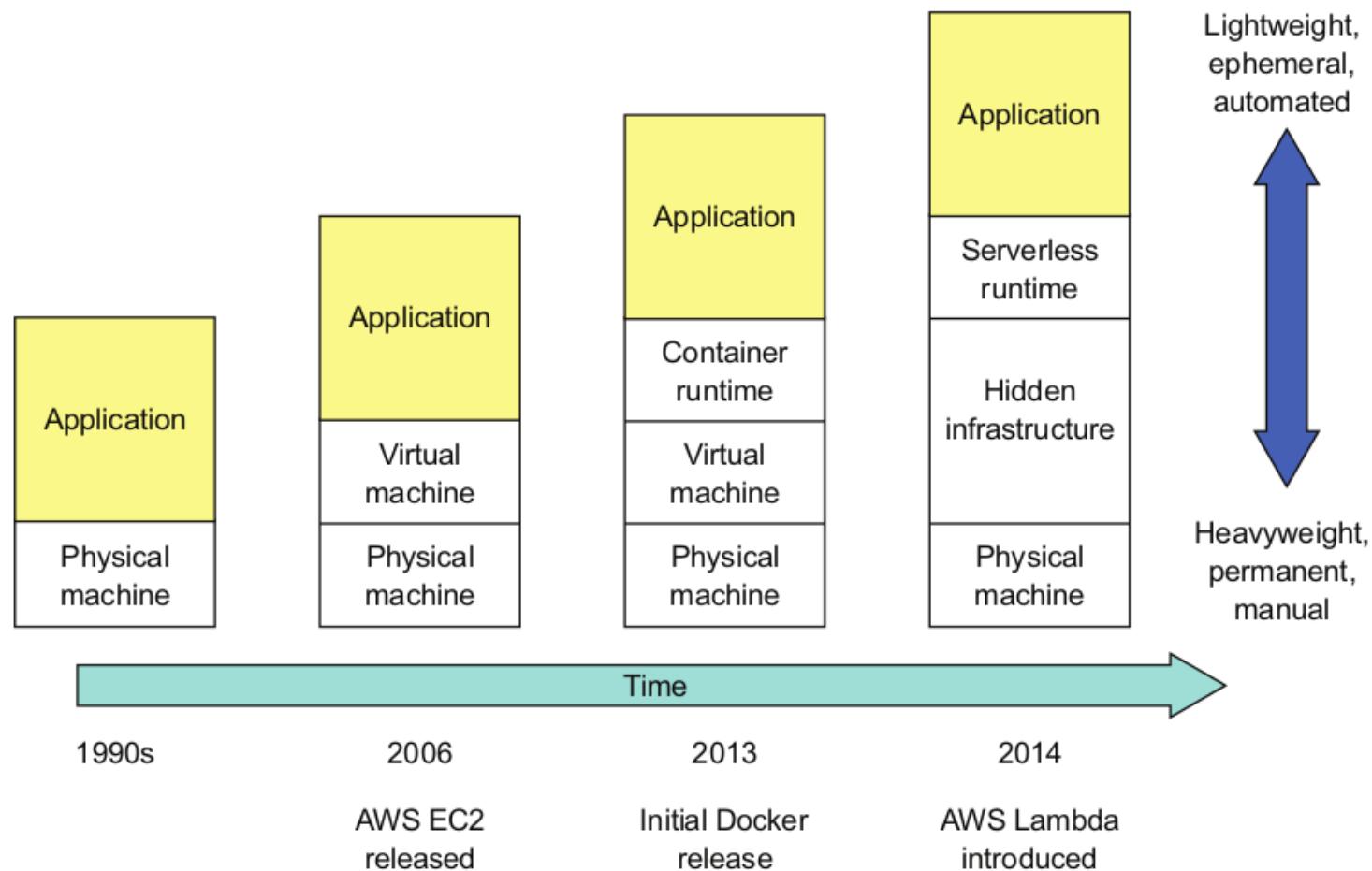
L'application produite nécessite une infrastructure d'exécution constituée de :

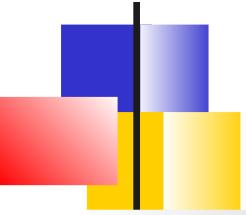
- **Matérielles** : Combien de CPU, RAM, Disque sont nécessaires pour l'application
- **Système d'exploitation** : Quelle est le système d'exploitation Cible
- **Middleware, produits, stack** : Quelles sont le middleware et les produits à installer ? Serveur applicatif, Base de données, ...

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





# Introduction

---

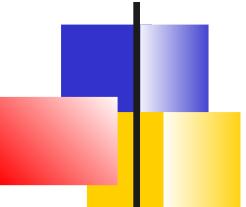
Contraintes de l'agilité

Approche DevOps

Pipelines CI/CD

Cycle de vie du code et outils

**Influence du DevOps sur les  
architectures**

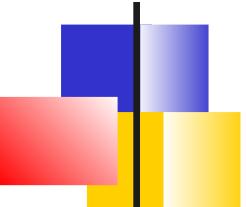


# Introduction

---

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

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



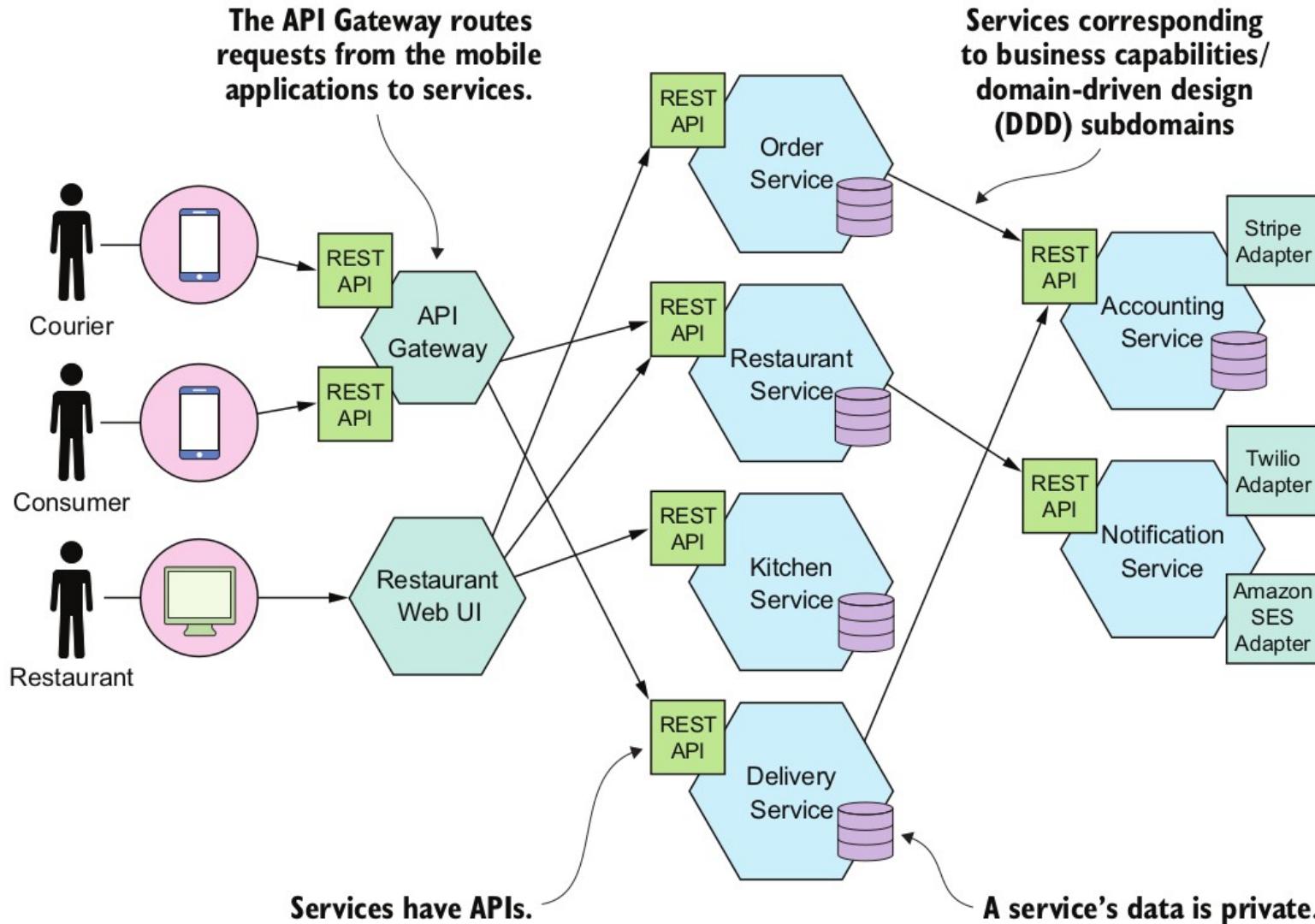
# Architecture

---

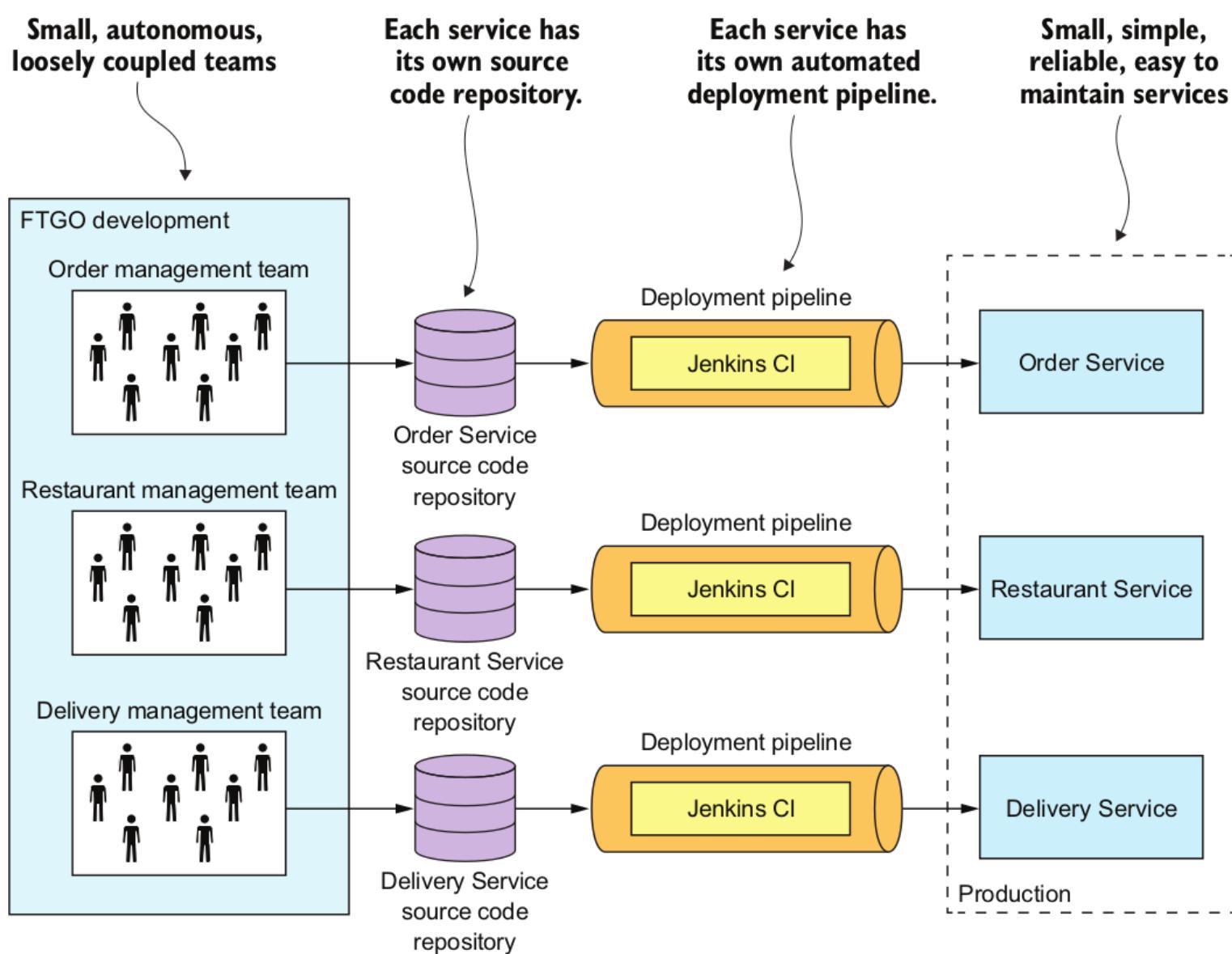
Une architecture micro-services implique la décomposition des applications en très petits services

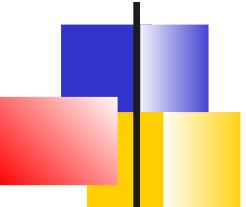
- faiblement couplés
- ayant une seule responsabilité
- Développés par des équipes full-stack indépendantes.

# Une architecture micro-service



# Organisation DevOps

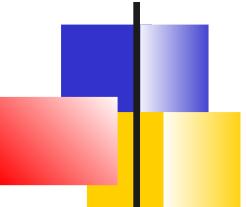




# Outils de pilotage

---

Les exemples de *JIRA* et *Gitlab*



# Introduction

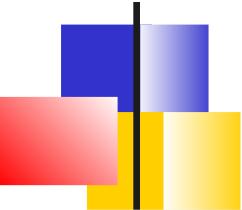
---

Les outils de pilotage sont à destination de toute l'équipe agile :

- Métier, PO, ScrumMaster, ...
- Full-stack développeur

Ils intègrent les méthodes agiles, permettent le suivi de modifications de code, la fourniture de métriques projet, le suivi des pipelines, des déploiement.

Constat : Peu d'outils arrive à agréger toutes ces fonctionnalités, à part peut-être *Github*, *Gitlab CI*



# JIRA

*Atlassian*

---

Se définit comme

- Gestion de projet agile
- Suivi d'issues

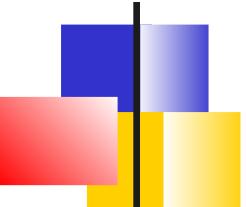
Propose 2 méthodes Agiles

- Kanban (lean)
- Scrum

Au final, *JIRA* gère des **issues** qui peuvent être : des user-story, des épopées, des bugs, des demandes de fonctionnalités, des tâches, ...

Chaque type d'issues a ses propres champs

Disponible en distribution cloud



# Différences entre les 2 méthodes

---

## Kanban :

- Le travail est suivi sur un tableau que tout le monde peut voir
- Montre le travail en cours Pas de notion de sprints, ni de backlog

## Scrum :

- Le travail est suivi sur un tableau que tout le monde peut voir
- Les spécifications sont maintenues dans un backlog
- Le travail est divisé en sprint
- L'équipe se réunit chaque matin

# Tableau de bord : Scrum

JIRA Dashboards Projects Issues Boards Create Search ⌂ 0 days remaining Complete Sprint Board ⌂

All sprints Switch sprint +

QUICK FILTERS: Product UI Server Only My Issues Recently Updated

12 To Do	4 In Progress	1 Code Review	7 Done
<p>▼ TIS Developer Love 3 issues</p> <p>TIS-37 When requesting user details the service should return prior trip SeeSpaceEZ Plus 2</p> <p>TIS-10 Bad JSON data coming back from hotel API SeeSpaceEZ Plus</p>	<p>TIS-8 Requesting available flights is now taking &gt; 5 seconds SeeSpaceEZ Plus</p>		
<p>▼ Everything Else 21 issues</p> <p>TIS-68 Homepage footer uses an inline style - should use a class Large Team Support</p> <p>TIS-20 Engage Saturn Shuttle Lines for group tours Space Travel Partn... 3</p> <p>TIS-12 Create 90 day plans for all departments in the Mars Office Local Mars Office 9</p> <p>TIS-15 Establish a catering vendor to provide meal service Local Mars Office 4</p>	<p>TIS-17 Engage Saturn's Rings Resort as a preferred provider Space Travel Partn... 3</p> <p>TIS-26 Engage the Red Titan Hotel as a preferred provider Space Travel Partn... 3</p> <p>TIS-33 Select key travel partners for the Saturn Summer Sizzle Summer Saturn Sale 1</p>	<p>TIS-67 Developer Toolbox does not display by default Large Team Support</p> <p>TIS-66 Add pointer to main css file to instruct users to create child themes Large Team Support</p>	<p>TIS-45 Email non registered users to sign up with Teams In Space Large Team Support 2</p> <p>TIS-49 Draft network plan for Mars Office Local Mars Office 5</p> <p>TIS-69 Add a String anonymizer to TextUtils Large Team Support</p> <p>TIS-23 Local Mars Office 1</p>

Developer Toolbox does not display by default  
Attach Files

Screen Shot 2015-08-13 at 4.1 326 kB 20/Aug/15 12:08 PM

Sub-Tasks Create Sub-Task

Issue Key	Summary	Status	Actions
TIS-127	Check Java version	OPEN	edit

Development

1 branch	Updated 17/May/14
7 commits	7:32 AM
1 pull request OPEN	Latest 17/May/14
3 builds	7:30 AM
	Updated 17/May/14
	2:31 PM

Deployed to Staging and Production

Project administration

# Backlog

JIRA Dashboards Projects Issues Boards Create Search  Teams in Space Scrum: Teams in Space Board 

**Backlog**

QUICK FILTERS: Product UI Server Only My Issues Recently Updated

**EPICS**

All issues SeeSpaceEZ Plus Large Team Support Space Travel Partners Summer Saturn Sale Afterburner Plus Local Mars Office TIS-2 Build out a local office on Mars Issues (10) Completed (0) Unestimated (0) Estimate (57)

**Sprint 6** 11 issues

30/May/15 10:05 PM - 13/Jun/15 10:05 PM

Linked pages

TIS-8 Requesting available flights is now taking > 5 seconds (2.0) SeeSpaceEZ Plus  
TIS-56 Add pointer to main css file to instruct users to create child themes (2.0) Large Team Support  
TIS-45 Email non registered users to sign up with Teams In Space (3.0) Large Team Support  
TIS-49 Draft network plan for Mars Office (2.1) Local Mars Office  
TIS-68 Homepage footer uses an inline style - should use a class (Large Team Support)  
TIS-17 Engage Saturn's Rings Resort as a preferred provider (2.1) Space Travel Partners  
TIS-69 Add a String anonymizer to TextUtils (Large Team Support)  
TIS-20 Engage Saturn Shuttle Lines for group tours (3.0) Space Travel Partners  
TIS-23 Engage JetShuttle SpaceWays for short distance space travel (3.0) Space Travel Partners  
TIS-67 Developer Toolbox does not display by default (Large Team Support)  
TIS-30 Create Saturn Summer Sizzle Logo (2.2) Summer Saturn Sale

**Sprint 7** 5 issues

Start Sprint Linked pages

TIS-12 Create 90 day plans for all departments in the Mars Office (2.1) Local Mars Office  
TIS-15 Establish a catering vendor to provide meal service (2.1) Local Mars Office  
TIS-16 Establish relationship with local office supplies company (2.1) Local Mars Office  
TIS-11 Register with the Mars Ministry of Labor (2.1) Local Mars Office  
TIS-13 Register with the Mars Ministry of Revenue (2.1) Local Mars Office

+ Create issue

5 issues Estimate (23)

**Backlog** 52 issues Create Sprint

Teams in Space / TIS-67 Developer Toolbox does not display by default Attach Files

Screen Shot 2015-08-13 at 4:11 326 kB 20/Aug/15 12:08 PM

**Sub-Tasks**

Create Sub-Task

Issue Key	Summary	Status	Actions
TIS-127	Check Java version	OPEN	 

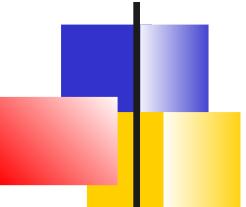
**Development**

1 branch Updated 17/May/14 7:32 AM  
7 commits Latest 17/May/14 7:30 AM  
1 pull request OPEN Updated 17/May/14 7:32 AM  
3 builds Latest 16/May/14 2:31 PM

Deployed to Staging and Production

Create branch

Project administration <<



# *Gitlab*

---

*Gitlab* a la même approche que JIRA, du point de vue métier, il gère des **issues**.

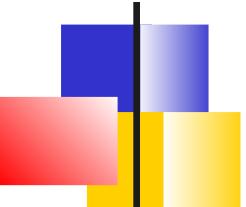
Les *issues* permettent la collaboration avant et pendant leur implémentation

Elles peuvent être utilisées pour :

- Discuter de l'implémentation d'une nouvelle idée
- Suivi de tâches
- Backlog agile, Reporting de bug, Demande de support

Elles sont toujours associées à un projet.

Elles peuvent être visualisées par groupe de projets.



# Données associées à une issue

## Contenu :

- Titre
- Description et tâches
- Commentaires et activité

## Membres

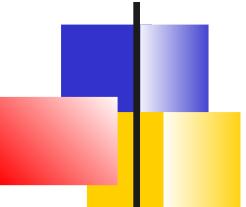
- Auteur
- Responsables

## Etat

- Status (ouvert/fermé)
- Confidentialité
- Tâches (terminée ou en suspens)

## Planning et suivi

- Milestone
- Date de livraison
- Poids
- Suivi du temps
- Tags (Labels)
- Votes
- Reaction emoji
- Issues liées
- Epic (collection d'issues) affectée
- Identifiant et URL



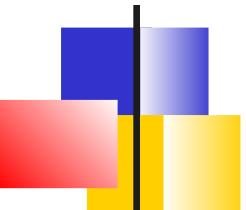
# Visualisation des issues

---

Les issues peuvent être visualisées via :

- Une **liste**. Elle affiche toutes les issues du projet ou de plusieurs projets. On peut les filtrer ou faire des actions par lots (bulk)
- Le **tableau de bord Kanban** qui affiche des colonnes en fonction des labels (par défaut statut de l'issue) ou des responsables. Les workflows sont customisable via les labels
- **Epic** : Vision transversale aux projets des issues partageant un thème, un milestone

# Liste



GitLab

Projects Groups Activity Milestones Snippets

GitLab.org > GitLab Community Ed... > Issues

Open 9,395 Closed 19,748 All 29,143

Search or filter results... Most popular

Relations between Issues  
#4058 · opened a year ago by Job van der Voort Discussion Product work UX feature proposal issues

Add group wiki page support  
#4037 · opened a year ago by Ken Phillis Jr ⏱ Next 3-6 months Accepting Merge Requests Community Contribution Platform UX customer feature proposal shortlist wiki

Merge train/Release train/Merge when master succeeds: run build on merged code before merging  
#4176 · opened a year ago by Frederik Zahle ⏱ Backlog CI/CD EE Premium Product work ci-build customer direction feature proposal frequently duplicated

Custom Roles  
#12736 · opened a year ago by Rolando Torres ⏱ Backlog Platform SP2 customer direction feature proposal frequently duplicated permissions user management

Please bring squash option when merging MRs to CE  
#34591 · opened 3 months ago by Jonas Kello Discussion feature proposal merge requests stewardship

Customize branch name when using create branch in an issue  
#21143 · opened a year ago by Adi Gerber ⏱ 10.1 Accepting Merge Requests Community Contribution Discussion In review UX ready feature proposal frequently duplicated issues repository

Move Fast-Forward Merge and Semi-Linear Merge to CE  
#20076 · opened a year ago by Markus KARG ⏱ 10.1 Deliverable Discussion In review UX ready backend feature proposal frontend merge requests

Provide an option/toggle in settings so that private repo commits show up on public user profile graph  
#14078 · opened a year ago by Cameron Banga ⏱ Next 3-6 months Platform feature proposal frontend graphs settings

Let's encrypt support for GitLab Pages  
#28996 · opened a year ago by Job van der Voort ⏱ Backlog CI/CD feature proposal pages

Shared CI runners for groups  
#10244 · opened a year ago by Lucas Koenig ⏱ Next 3-6 months CI/CD CDD customer feature proposal frequently duplicated

21143 1 1 215 67 updated a week ago

20076 1 1 140 86 updated about 9 hours ago

14078 1 1 137 35 updated a day ago

28996 1 1 117 47 updated 2 months ago

10244 1 1 103 45 updated about 16 hours ago

21143 1 1 215 67 updated a week ago

20076 1 1 140 86 updated about 9 hours ago

14078 1 1 137 35 updated a day ago

28996 1 1 117 47 updated 2 months ago

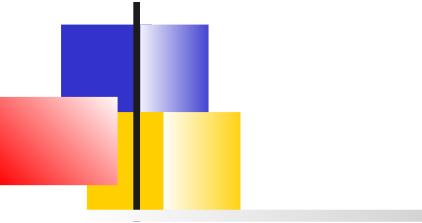
10244 1 1 103 45 updated about 16 hours ago

# Kanban

The image shows a digital Kanban board with three columns: "In dev", "In review", and "Closed". Each column contains several items represented as cards.

- In dev:** Contains one item.
  - Title:** Read git data via gitaly
  - Status:** In dev
  - Labels:** Plan, backend
  - Description:** Work for the Plan team. Covers Issues, Labels, Milestones, Boards, and more. See <https://about.gitlab.com/handbook/product/categories/>. Ping @victorwu for questions and comments.
  - Priority:** P1
  - Milestones:** Product Vision 2019, UX ready
  - Tags:** backend, code review, devops:create, direction, feature proposal, frontend, merge requests
  - Link:** <gitlab-org/gitlab-ce#18008>
  - Comments:** 5
- In review:** Contains two items.
  - Title:** `ExpireBuildArtifactsWorker` is broken
  - Status:** In review
  - Labels:** P3, S3, Verify, database, devops:verify, missed-deliverable, performance
  - Description:** gitlab-org/gitlab-ce#41057
  - Milestones:** Product Vision 2019
  - Tags:** Accepting merge requests, In review, Stretch, Verify, database, devops:verify, meta, missed-deliverable, performance
  - Link:** <gitlab-org/gitlab-ce#41057>
  - Comments:** 95
- Closed:** Contains four items.
  - Title:** include brand ai styles
  - Status:** To Do
  - Link:** <gitlab-org/design.gitlab.com#5>
  - Milestones:** Product Vision 2019
- Closed:** Contains one item.
  - Title:** static page example
  - Status:** To Do
  - Link:** <gitlab-org/design.gitlab.com#4>
  - Milestones:** Product Vision 2019
- Closed:** Contains one item.
  - Title:** welcome page
  - Status:** To Do
  - Link:** <gitlab-org/design.gitlab.com#3>
  - Milestones:** Product Vision 2019
- Closed:** Contains one item.
  - Title:** add some real components
  - Status:** To Do
  - Link:** <gitlab-org/design.gitlab.com#2>
  - Milestones:** Product Vision 2019

# Vue détaillée issue



Open Issue #1395 opened about 14 hours ago by  Marcia Ramos 0 of 2 tasks completed **1** New issue Close issue Edit

**GitLab Issue 12**

Hello World! 🎉

This is my issue's description, written in markdown (GitLab Flavored Markdown).

This is an **<h3>**

Let's quote someone here

Add a task list:

Task 1  
 Task 2

Mention merge requests ([gitlab-org/gitlab-ee!1784 \(merged\)](#)) and issues ([gitlab-org/gitlab-ee#2101 \(closed\)](#)) and hover over them to see their titles.

Invite users to collaborate with `@mentions : @marcia` **13**

Edited just now by Marcia Ramos

**1 Related Merge Request**

**14** !1784 How to Configure LDAP with GitLab EE in GitLab.org / GitLab Enterprise Edition Merged

**15**

0 likes 0 dislikes 15

Marcia Ramos @marcia assigned to @axil and @jivanl about 14 hours ago

Marcia Ramos @marcia changed milestone to **9.2** about 14 hours ago

Marcia Ramos @marcia added **on it** label about 14 hours ago

Marcia Ramos @marcia changed time estimate to 30m about 14 hours ago

**18** Create a merge request

**Create a merge request**  
Creates a branch named after this issue and a merge request. The source branch is 'master' by default.

**Create a branch**  
Creates a branch named after this issue. The source branch is 'master' by default.

**10** Notifications Unsubscribe

**11** Reference: [gitlab.com/www-g...](#)

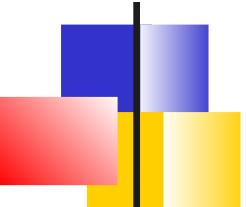
**16**

Write Preview

Write a comment or drag your files here...

Markdown and slash commands are supported Attach a file

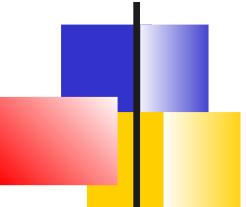
**17** Comment Close issue



# Actions sur une issue (1)

---

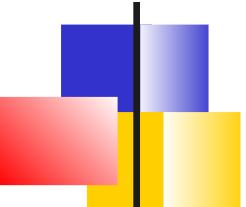
1. Création, Fermeture, Edition des champs de base
2. Ajouter à sa Todo List, la marquer comme terminé
3. Responsable(s) de l'issue, peut être changé à tout moment
4. Affecter une issue à un milestone
5. Temps estimé, temps passé
6. Date de livraison, peut être changée à tout moment
7. **Labels.** Catégorise les issues et permet de mettre en place des workflows personnalisé reflété dans le Kanban
8. Poids. Indicateur sur l'effort nécessaire associé à l'issue



# Actions sur une issue

---

9. Participants. Indiqués dans la description ou qui ont participé à la discussion
10. Notifications. Permet de s'abonner/désabonner
11. Référence. Permet de copier l'URL d'accès
12. Titre et description, mentions (markup)
13. Commentaires et discussions (threads)
14. Merge requests associés
15. emoji



# Milestones

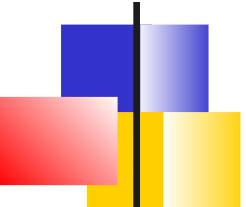
---

Ils permettent d'organiser les issues et MR dans un groupe cohérent, avec une date de début et une date d'échéance (facultatives).

Ils peuvent définir :

- des sprints Agile
- des releases

Ils peuvent être associés à des groupes



# SCM

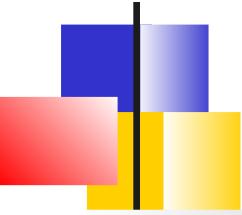
---

## **Caractéristiques de Git**

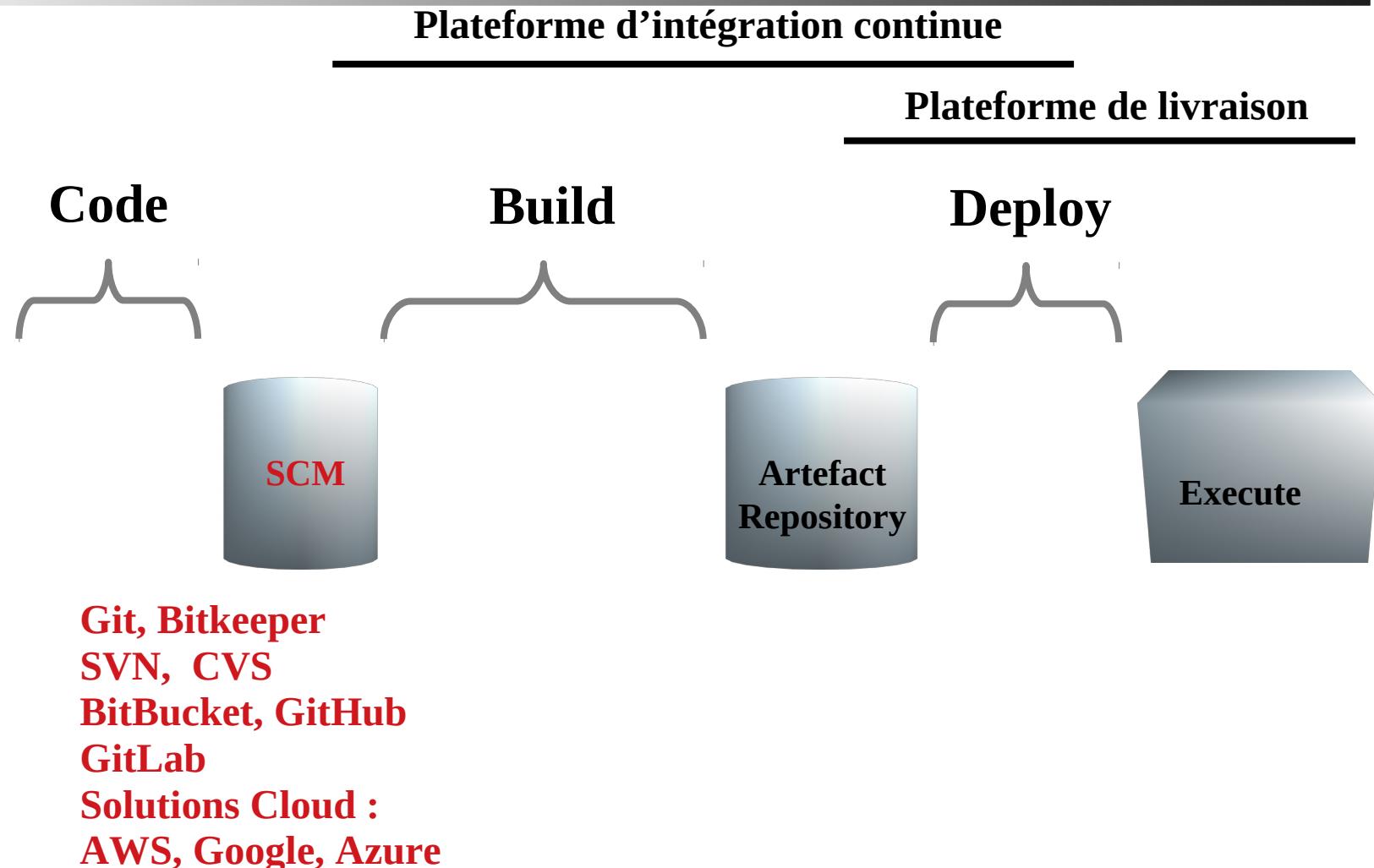
Branches locales et distantes

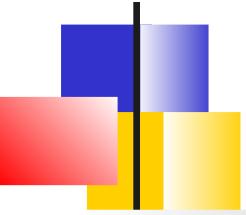
Workflows de collaboration et usage des  
branches

Merge/Pull Request



# SCMS et Cycle de vie





# SCM

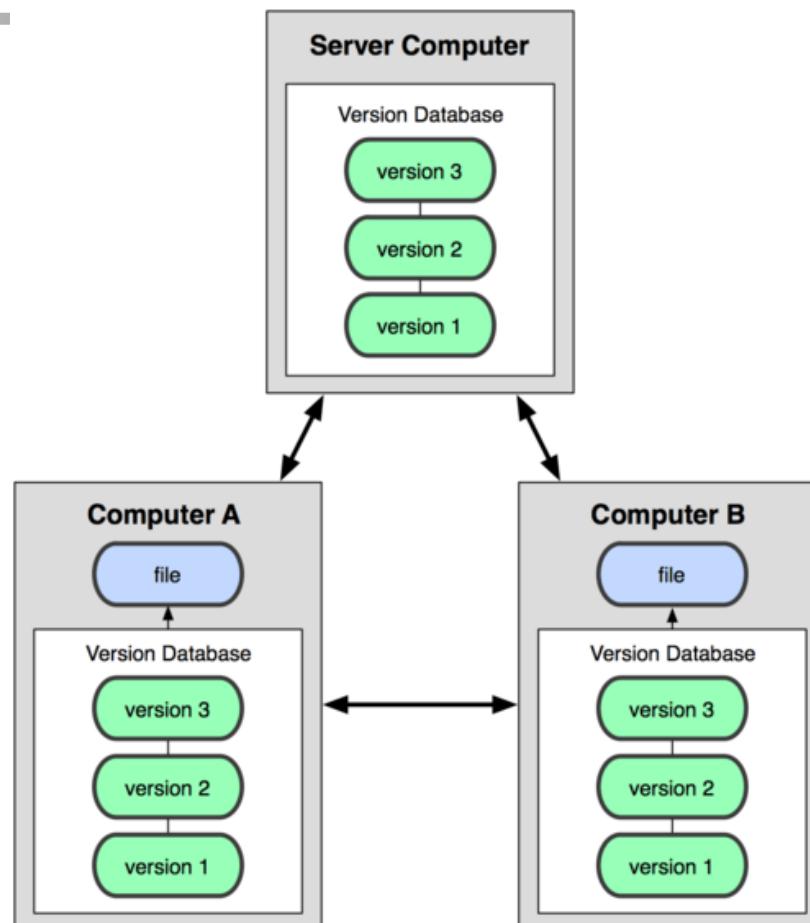
---

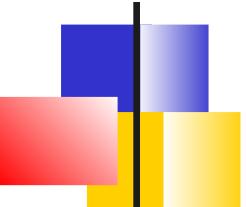
Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

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

---

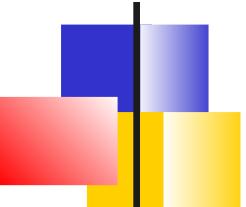
La plupart des opérations sont locales et donc très rapides

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

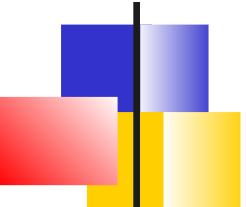


# SCM : Commit, Branches et Tag

---

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

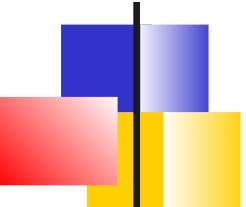
**checkout** : Extraction d'une révision particulière

**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 **patches**

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*

The screenshot shows the gitk graphical interface for Git. The main window displays a commit history for the 'master' branch. A specific commit is selected, showing its details and a diff view of the changes made to the file 'lib/grit.rb'.

**Commit Details:**

- SHA1 ID: 2a8c38b72388fc64780d0e8e6d7b16d45dfb6ebd
- Author: Jos Backus <jos@catnook.com> 2009-01-30 17:49:40
- Committer: Scott Chacon <schacon@gmail.com> 2009-01-30 18:23:08
- Parent: 66933d0e3329c7abe598c2246abebaf90d04b40 (test for current head)
- Branch: master
- Follows: v0.7.0
- Precedes:

**Message:**

Make the number of bytes to be read from git's stdout configurable.

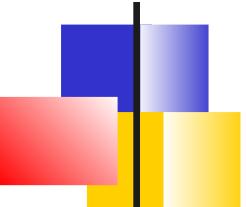
**Signed-off-by:** Scott Chacon <schacon@gmail.com>

**Diff View:**

```
index 7c1785d..fe3d6b5 100644
@@ -22,11 +22,19 @@ module Grit
   include GitRuby

   class << self
-     attr_accessor :git_binary, :git_timeout
+     attr_accessor :git_binary, :git_timeout, :git_max_size
   end

-   self.git_binary = "/usr/bin/env git"
```



# Quelques commandes avancées

---

***git revert <commitID>*** : Faire les modifications inverses d'un commit

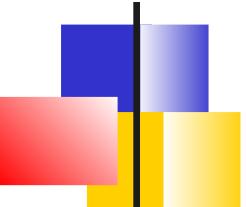
***git cherry-pick <commitID>*** : Appliquer un commit particulier

***git stash*** : Mettre de côté sans committer pour réutilisation ultérieure

***git commit --amend*** : Réécrire le dernier commit

***git rebase -i*** : Réécriture d'historique

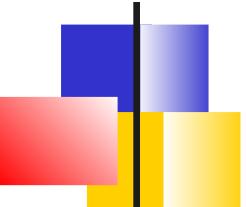
***git remote prune <remote>***: Nettoyer les branches distantes qui n'existent plus



# SCM

---

Caractéristiques de Git  
**Branches locales et distantes**  
Workflows de collaboration et usage des  
branches  
Merge/Pull Request



# Les branches

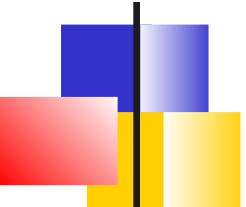
---

Les branchement signifie que le code diverge de la ligne principale de développement et que les deux branches évoluent indépendamment

Les branches Git sont très légères et les opérations de création et de basculement instantanées

=> Git encourage donc des workflows avec des branchements et des fusions de branches nombreuses.

=> En général, on crée une branche pour commencer un travail, quand le travail est terminé, on l'intègre dans la branche d'où l'on vient

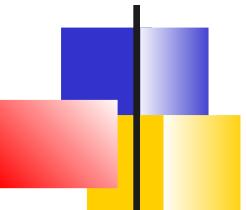


# Branches locales/distantes

---

On distingue :

- les branches **locales** qui ne sont vues que par un développeur et qui lui facilitent son travail de tous les jours
- Les branches **distantes** qui sont des branches partagées par toute l'équipe ou par une partie de l'équipe



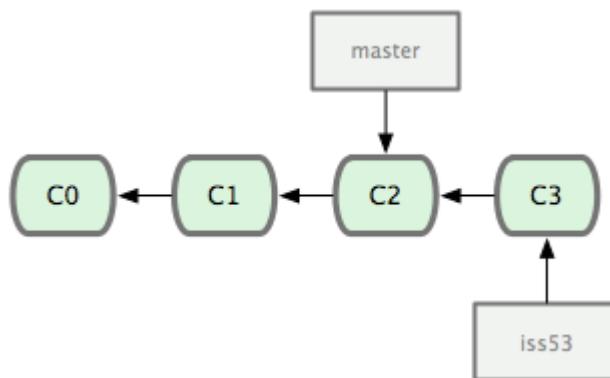
# Création branche locale

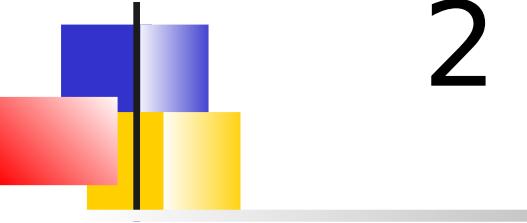
---

La création d'une branche le basculement  
du workspace se fait comme suite

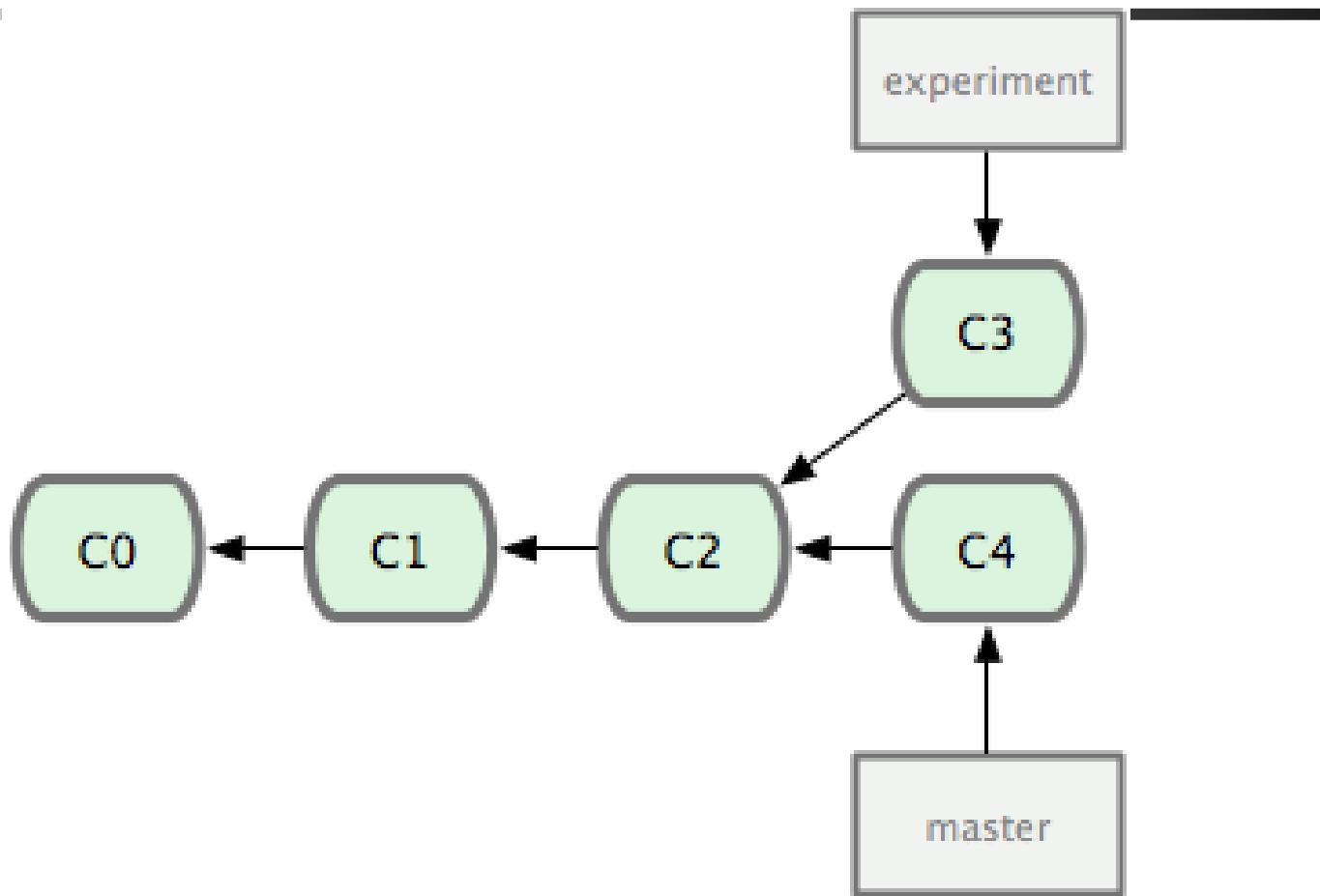
```
$ git checkout -b iss53  
Switched to a new branch 'iss53'
```

Après quelques commits :



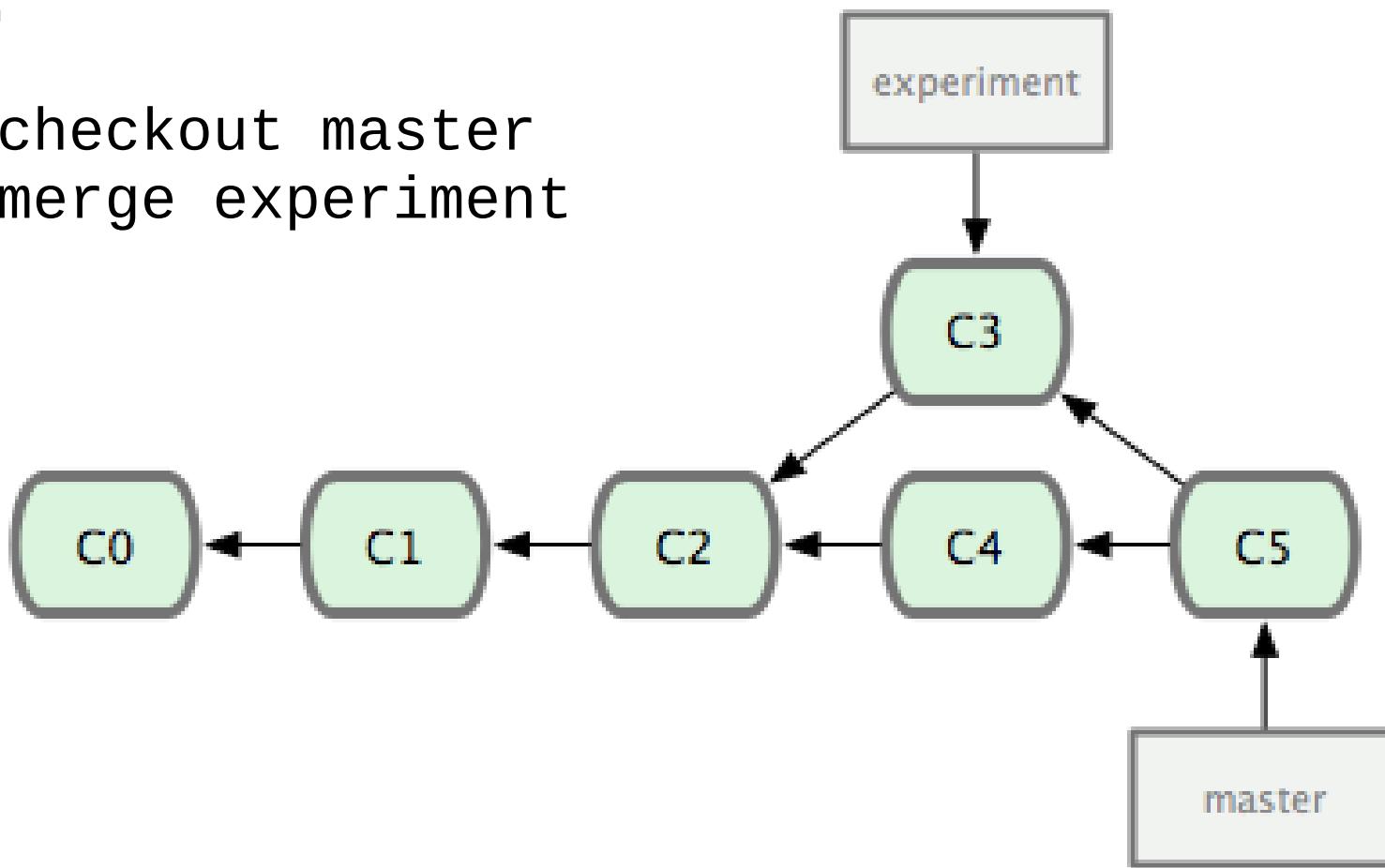


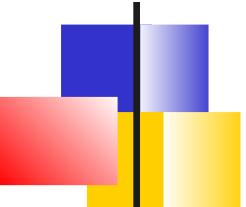
# 2 branches divergentes



# Résultat d'un merge

```
git checkout master  
git merge experiment
```





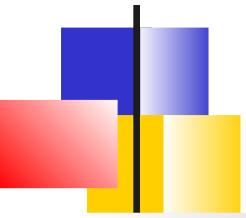
# Merge et conflit

---

Si des conflits apparaissent lors de la fusion, l'opération s'interrompt

Il faut alors :

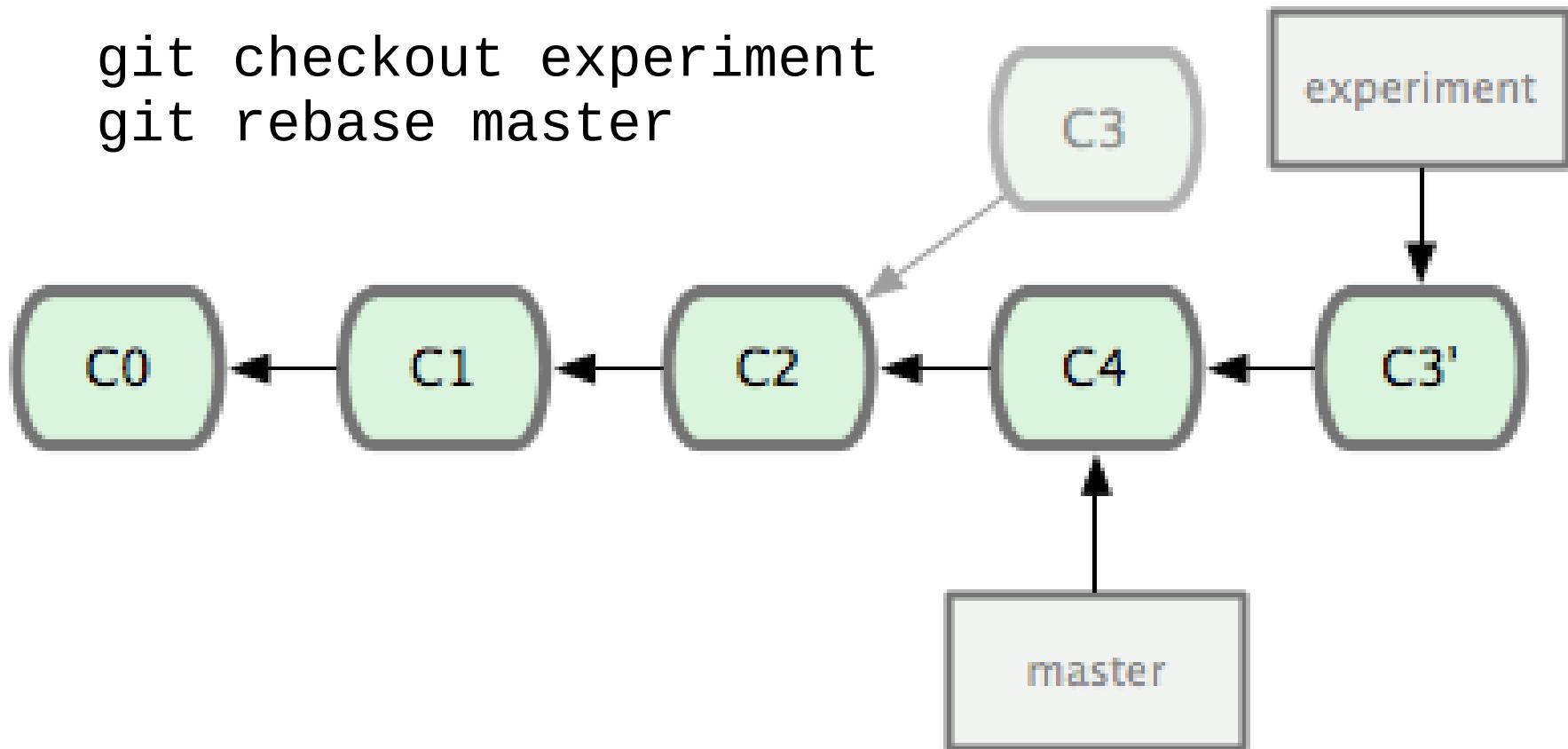
- Résoudre chaque conflit et l'indiquer à *git* avec  
**git add**
- Quand tous les conflits sont réglés  
**git commit**

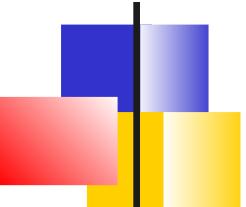


# Résultat d'un rebase

---

```
git checkout experiment  
git rebase master
```





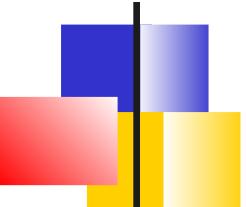
# Rebasing et conflit

---

Si un conflit apparaît lors de l'application d'un patch particulier, l'opération de rebasing s'interrompt

Il faut alors soit :

- Résoudre le conflit et continuer l'opération de rebasing  
`git add` après la résolution du conflit  
`git rebase --continue` pour continuer le rebasing
- Ignorer l'application de ce patch  
`git rebase --skip`
- Arrêter l'opération de rebasing  
`git rebase --abort`



# Branches distantes

Les **branches distantes** sont des références à l'état des branches sur un référentiel distant.

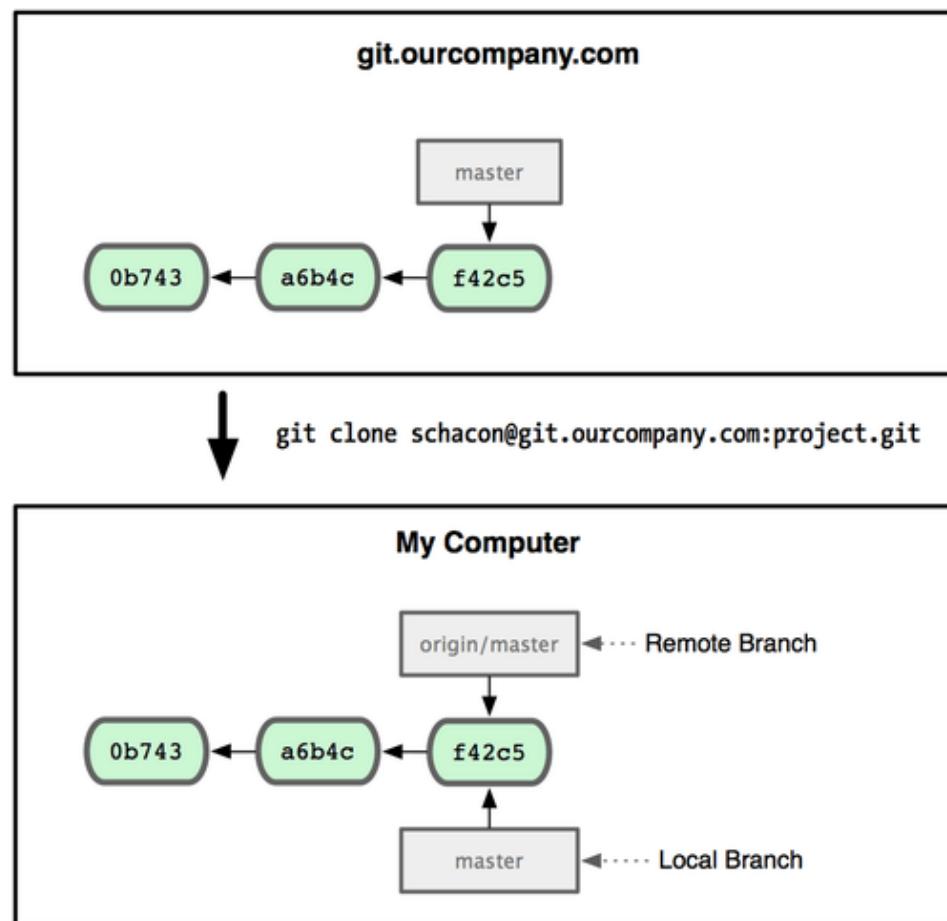
Ce sont des branches locales que l'on ne peut pas modifier.

La référence est mise à jour dès lors qu'il y a une communication réseau

- Les branches distantes sont donc comme des signets qui rappellent l'état de la branche, la dernière fois que l'on s'est connecté au référentiel distant

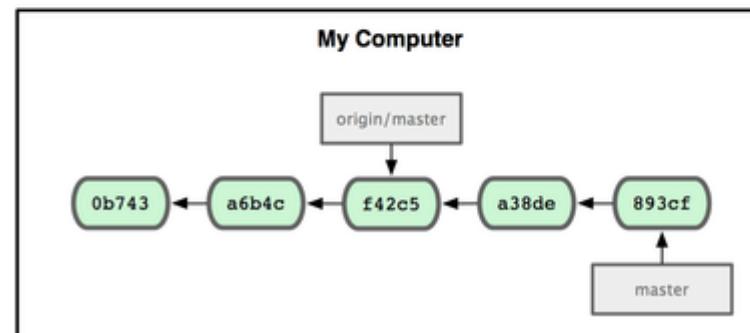
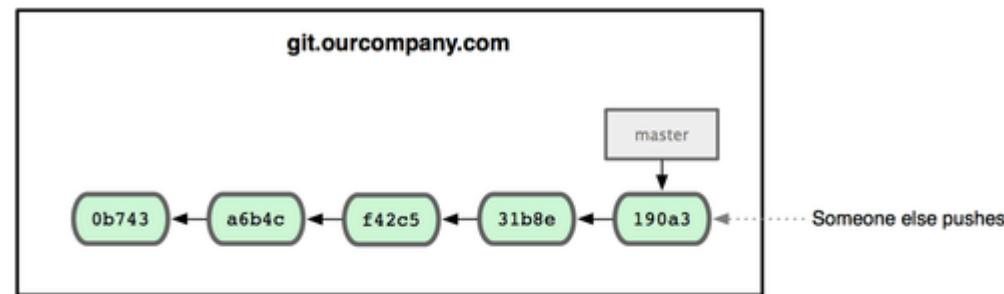
Elles sont référencées dans les commandes *Git* par  
**(remote)/(branch)**

# Exemple après clone



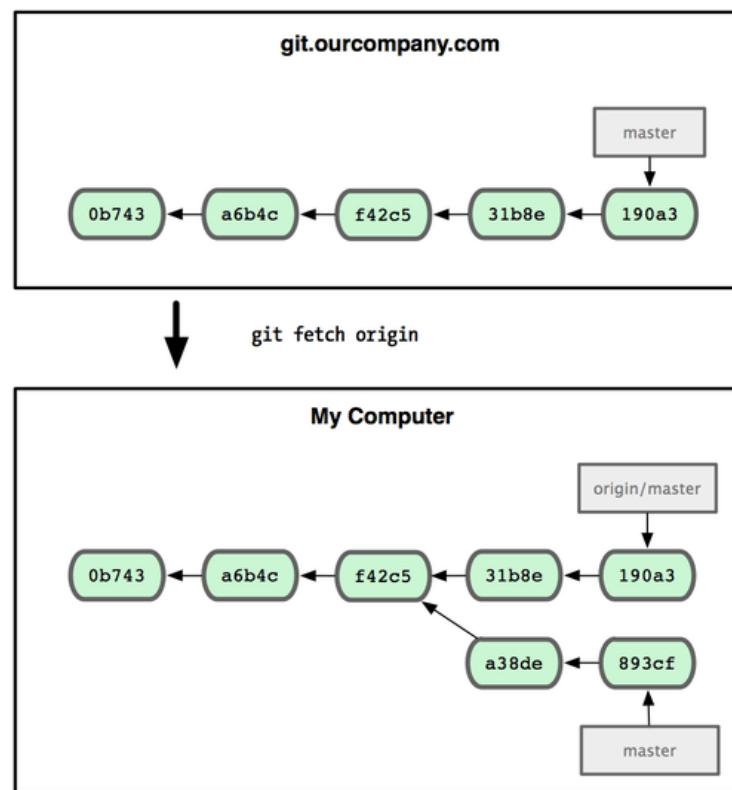
# Déplacement

Sans contact avec le serveur d'origine, le pointeur *origin/master* ne se déplace pas

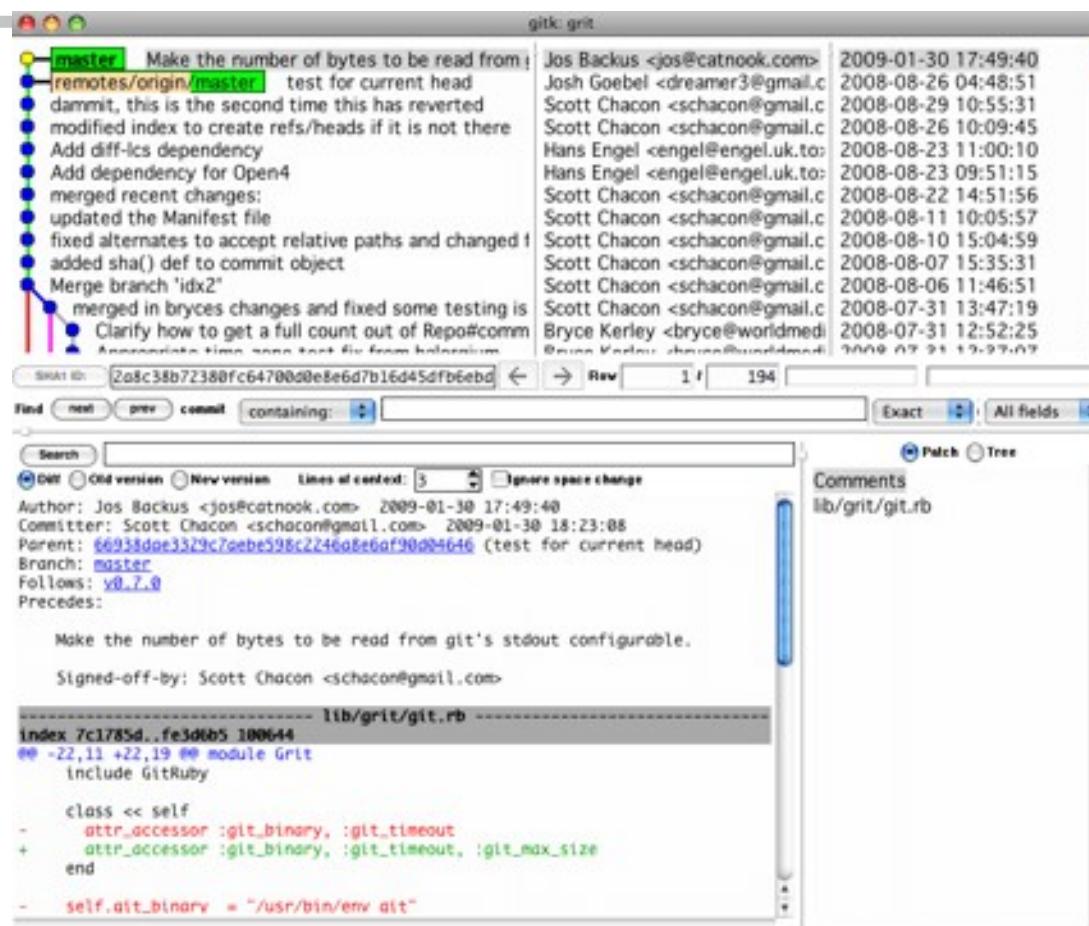


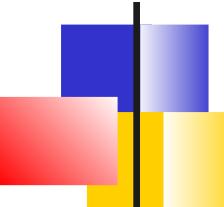
# Synchronisation

Pour synchroniser une branche distante, on exécute la commande **git fetch origin** qui rapatrie les nouvelles données et met à jour la base de données locale en déplaçant le pointeur *origin/master* à sa nouvelle position



# Exemple gitk





# Syntaxe complète push

---

La syntaxe complète de la commande push est :

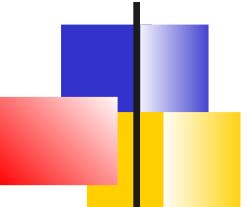
```
$ git push origin serverfix:serverfix
```

Ce qui veut dire

*« Recopier ma branche locale nommée **serverfix** dans la branche distante nommée **serverfix** »*

Si l'on veut donner un autre nom à la branche distante, on peut utiliser :

```
$ git push origin serverfix:autrenom
```



# Branche de suivi

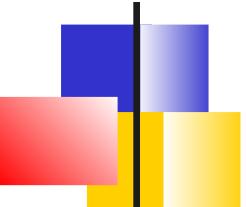
---

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement une **branche de suivi**.

Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante

Dans une branche de suivi *git push*, et *git pull* sélectionne automatiquement le serveur impliqué

C'est le même mécanisme lorsque l'on clone un dépôt



# Branches de suivi

---

Il y a plusieurs façons de créer des branches de suivi :

L'option `--track` :

```
$ git checkout --track origin/serverfix
```

Si la branche n'existe pas localement et que son nom correspond exactement à une branche de suivi, on peut utiliser le raccourci :

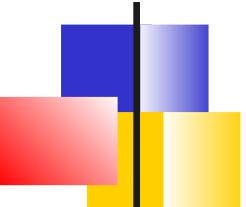
```
$ git checkout serverfix
```

Si l'on veut renommer la branche

```
$ git checkout -b sf origin/serverfix
```

Enfin, si on veut utiliser une branche locale existante :

```
$ git branch --set-upstream-to origin/serverfix
```



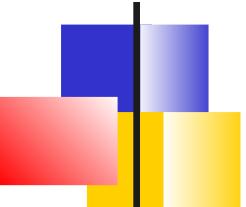
# SCM

---

Caractéristiques de Git  
Branches locales et distantes

## **Workflows de collaboration et usage des branches**

Merge/Pull Request



# Révision et Clé de Hash

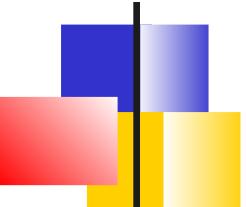
---

Chaque instantané du projet est identifié dans le dépôt par une clé :

- N° de révision dans CVS, SVN
- Clé de hash dans Git

Une branche ou un tag est une façon de donner un nom à un commit particulier

- Une branche avance avec les commits.
- Un tag est fixe et immuable

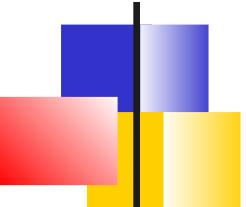


# Utilité des branches

---

Une branche est créée lorsque on veut démarrer des développements sans impacter la branche d'origine (master ou autre)

Éventuellement, lorsque les développements sont terminés ; il sont intégrés à la branche d'origine



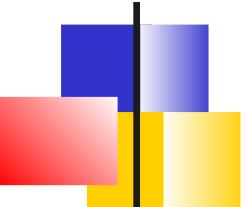
# Usage

---

Localement, un développeur crée des branches dès qu'il démarre un nouveau travail. Il supprime la branche locale lorsque son travail est terminé.

Sur le serveur de référence, les branches créées servent à la collaboration.

Les branches stables du serveur ont souvent des fonctions différentes (production, intégration)

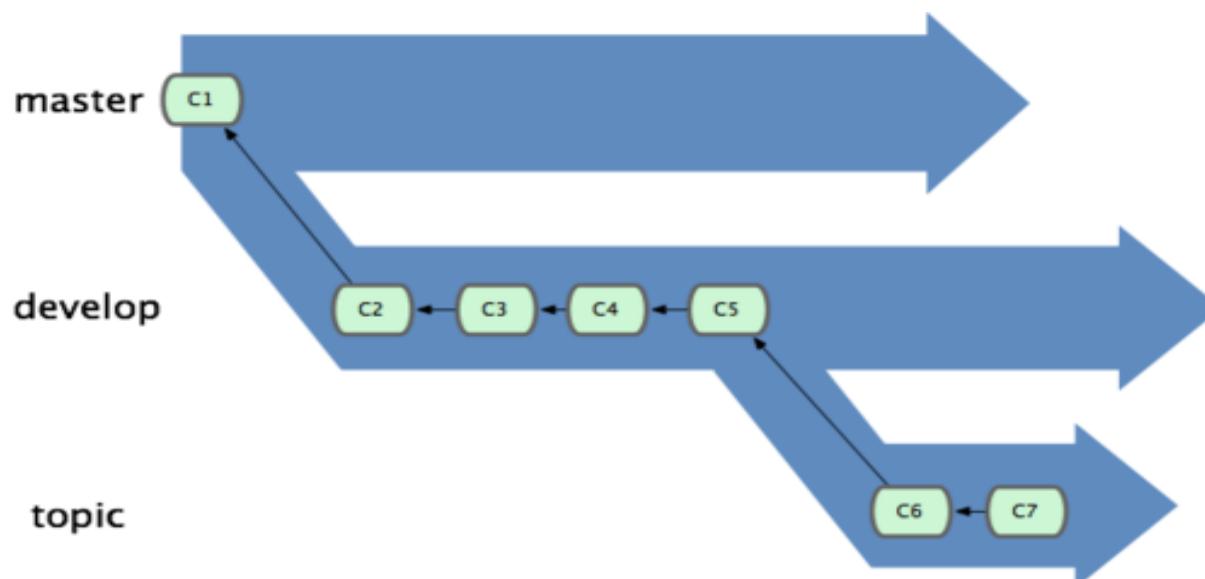


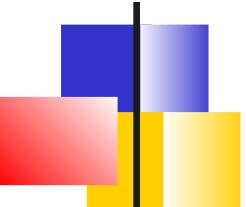
# Branches longues et thématiques

Sur un projet, on a généralement donc plusieurs branches ouvertes correspondantes à des étapes du développement et des niveaux de stabilité

Lorsqu'une branche atteint un niveau plus stable, elle est alors fusionnée avec la branche d'au-dessus.

On distingue les branches longues et les branches de features utilisés que pendant le développement de la fonctionnalité





# Gitflow

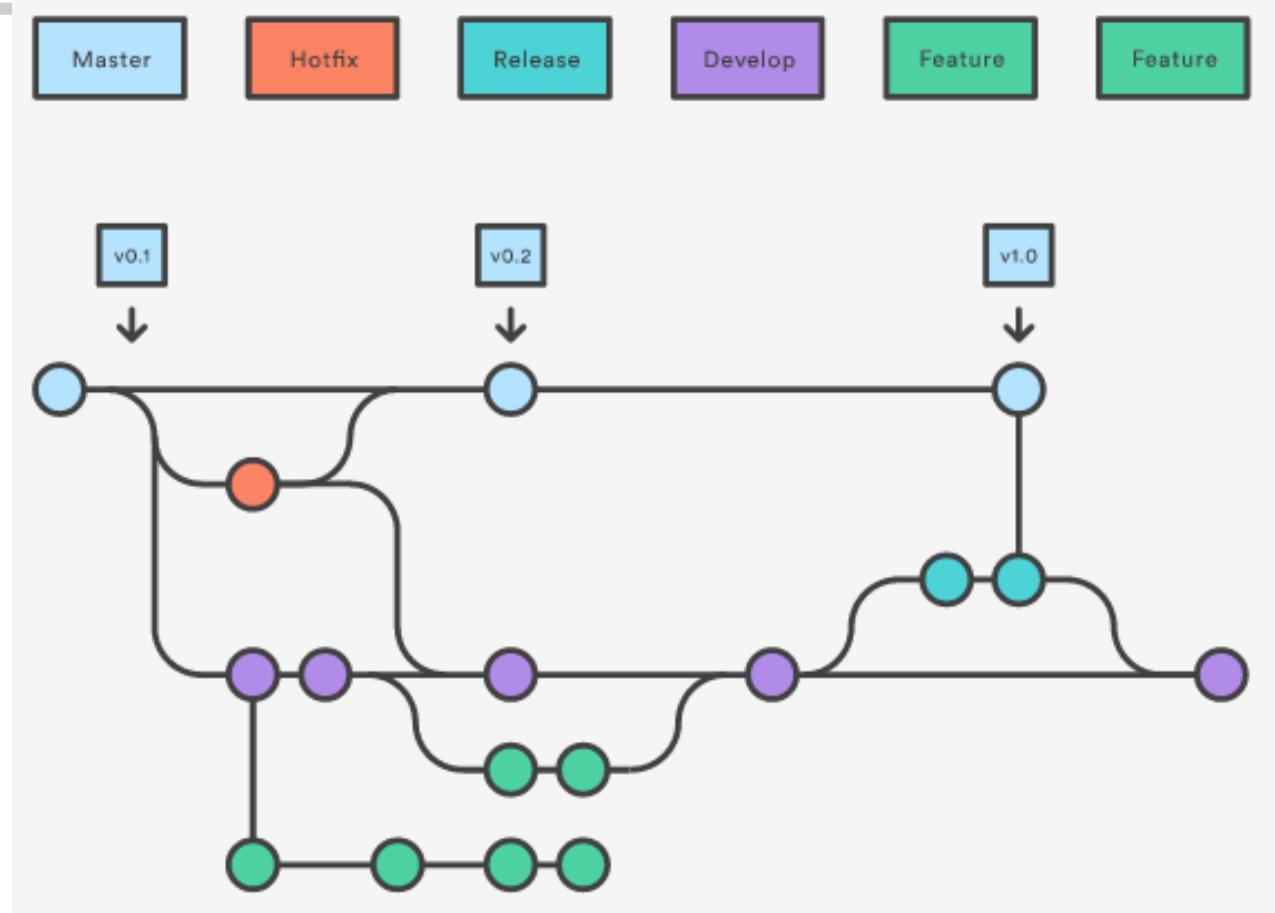
---

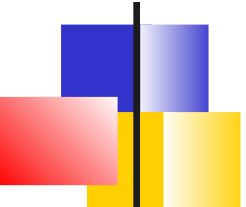
Le workflow **Gitflow** définit un modèle de branches orientées vers la release d'un projet

- Adapté pour la gestion de grands projets
- Il assigne des rôles très spécifiques aux différentes branches et définit quand et comment elles doivent interagir

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

# Branches Gitflow



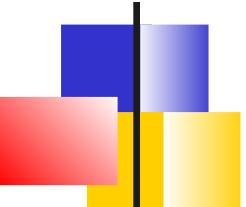


# Revue de code

---

En plus des branches de Gitflow, les gros projets utilisent également des branches de revue de code permettant de valider les modifications avant de les intégrer dans la branche supérieure.

Des outils tels que Gerrit, Gitlab, Github permettent la mise en place transparente de ce type de fonctionnement



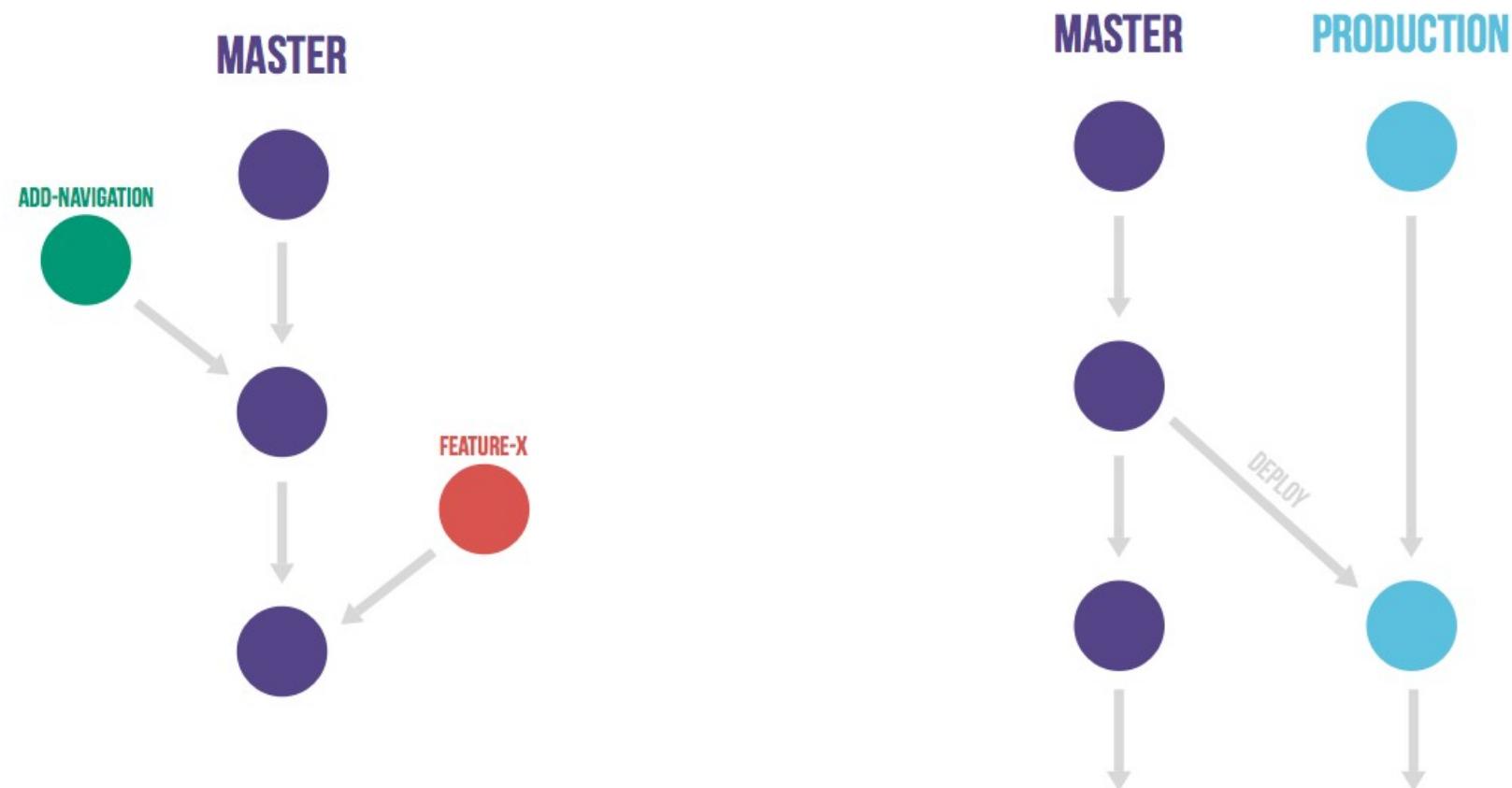
# Gitlab Flow

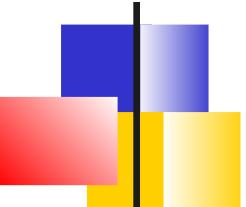
---

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

- 1) Les fix ou fonctionnalités sont développés dans une feature branch
- 2) Via un merge request, elles sont intégrées dans la branche master
- 3) Il est possible d'utiliser d'autres branches :
  - production : Chaque merge est taggée
  - release : Branche de préparation d'une release
- 4) Les Bug fixes/hot fix patches sont repris de master via des cherry-picked

# Features, Master and Production



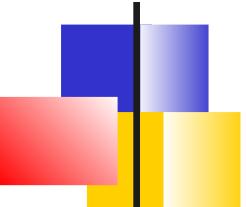


# Branche de production

---

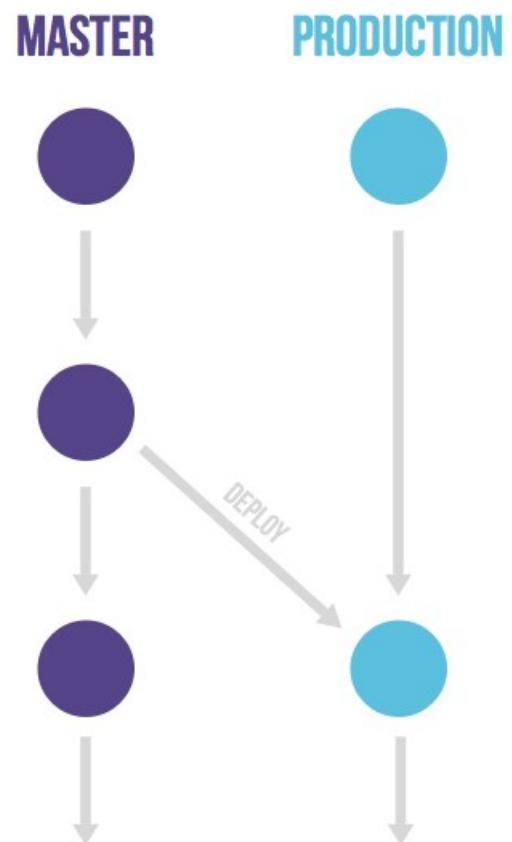
Si l'on veut maîtriser les déploiement vers la production, il est possible d'utiliser une branche ***production*** qui reflète le code déployé

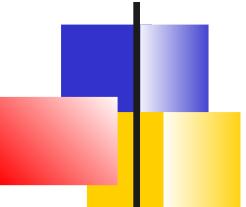
- Lorsque l'on veut déployer, il suffit de fusionner *master* avec la branche de production et déployer à partir de *production*.



# Master et Production

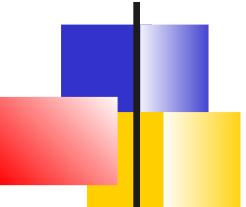
---





# Workflow typique

- 
- 1) Les travaux sont effectués localement dans une branche
  - 2) Ils sont ensuite poussés sur Gitlab
  - 3) Un merge request est créé
  - 4) *Gitlab* permet alors d'effectuer une revue de code ainsi que de collaborer sur les modifications en cours
  - 5) Éventuellement, ces modifications peuvent être déployées sur une « Review Apps »
  - 6) Des approbations peuvent être demandées au *maintainers* avant le merge dans la branche master



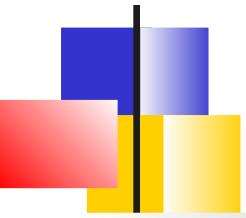
# Branches d'environnement

---

Si il est souhaitable d'avoir des environnements de validation (staging, pré-production), Gitlab associe à chaque **environnement** déclaré une branche.

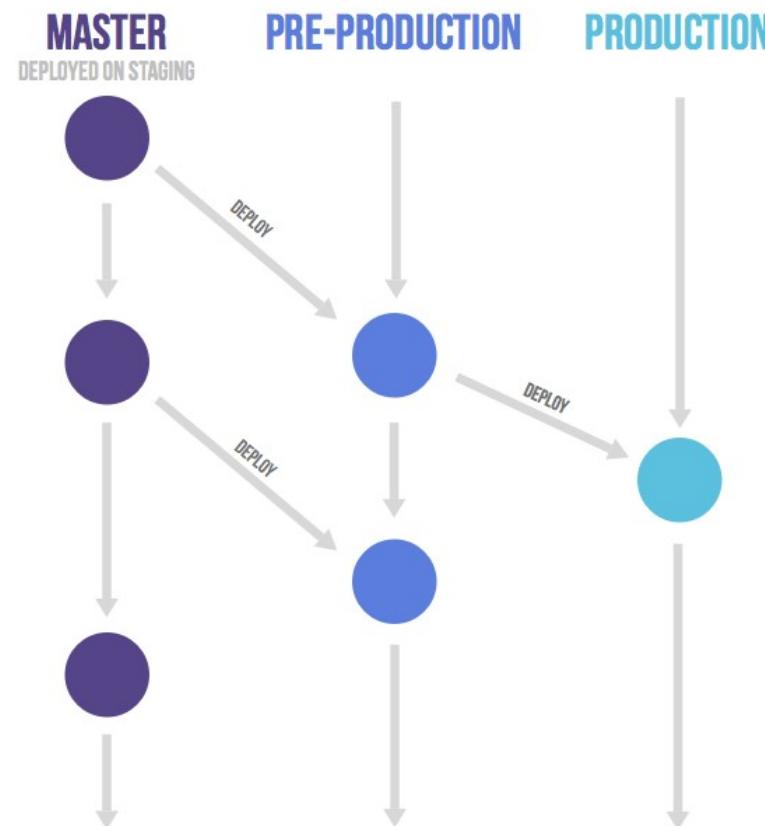
=> L'historique des déploiements sur un environnement particulier est aisément consultable

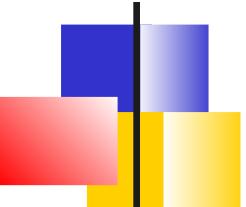
Pour passer à un environnement aval, il suffit de merger avec la branche en amont.



# Pré-production et production

---





# Releases

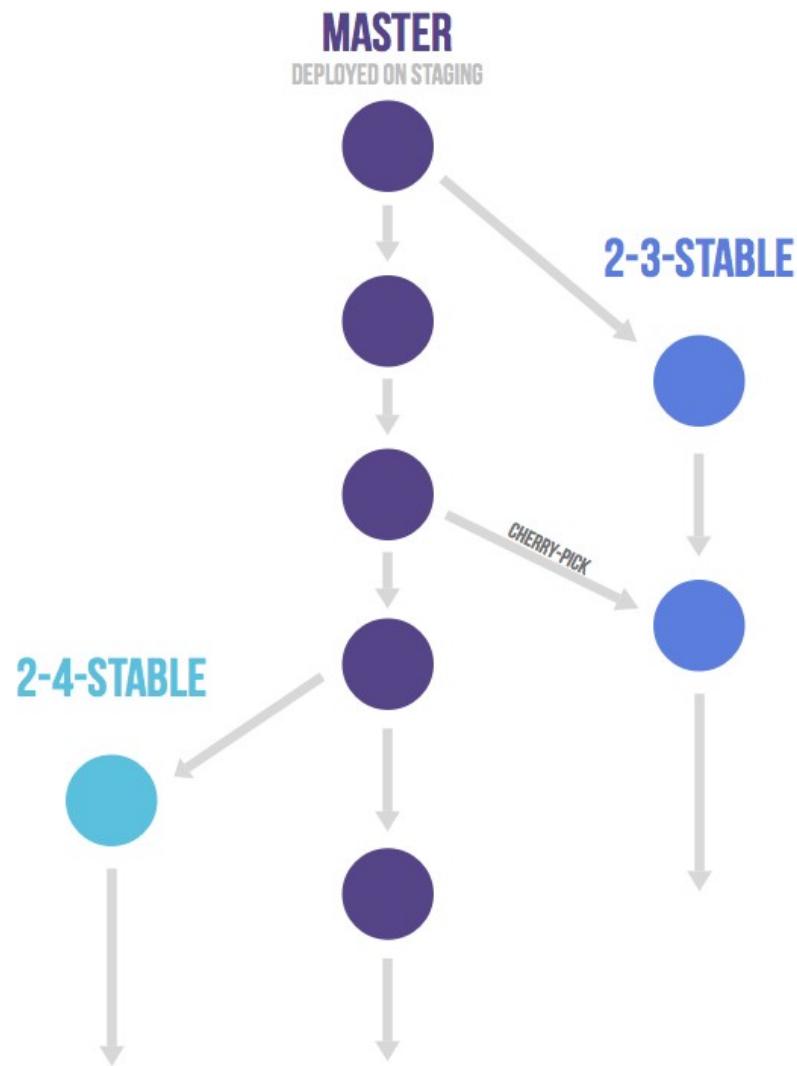
---

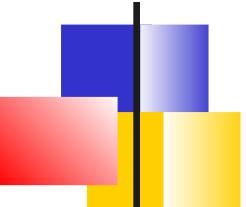
Pour la distribution de software, il est possible de mettre en place des **branches de release**

- Lors de la préparation d'une release, une branche stable est créé à partir du *master*
- Un tag est créé pour chaque version
- Les bugs critiques trouvés à posteriori sont mergés dans master puis appliqués dans la branche de release via des Cherry-pick

Gitlab permet de visualiser les *Releases* d'un projet via l'UI et de fournir les artefacts construits via téléchargement

# Branches de releases

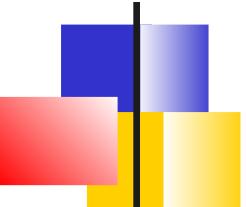




# SCM

---

Caractéristiques de Git  
Principales commandes Git  
Workflows de collaboration et usage des branches  
**Merge/Pull Request**



# Introduction

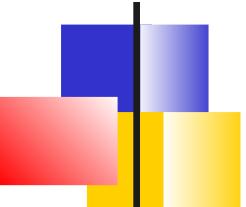
---

Le **Merge/Pull Request** sont la base de la collaboration sur Gitlab/GitHub

C'est un emplacement de collaboration qui permet

- Comparer les changements entre 2 branches, 2 dépôts
- Revoir et discuter des modifications de code

- 1) Un développeur soumet ses modifications à un (ou plusieurs) mainteneur de projet.
- 2) Ceux-ci peuvent alors facilement voir les modifications, tester la nouvelle version, y apporter ses suggestions d'amélioration.
- 3) Une fois approuvée la MR/PR disparaît et le nouveau code est intégré dans la branche/dépôt stable

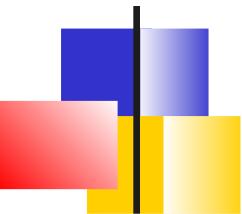


# Cycle de vie d'une MR dans Gitlab

---

- 1) Lors du traitement d'une issue, le développeur crée un *merge request* qui a pour effet de créer une branche. Le travail n'est pas prêt à être fusionné mais la collaboration et la revue de code peuvent commencer dans une feature branch. Le *Merge Request* est préfixé par **Draft**
- 2) Lorsque la fonctionnalité est prête, le développeur enlève le statut *Draft* et le mainteneur du projet est prévenu.
- 3) Le mainteneur a alors la possibilité de faire une revue des modifications, demander au développeur des améliorations, abandonner la MR
- 4) Si il approuve la MR, la feature branch est fusionnée dans une branche stable (master). Il peut choisir le mode de fusion et en général la branche de features est détruite .

# Vue projet



GitLab / 🚧 GitLab.org / 🚧 GitLab Community Edition ▾

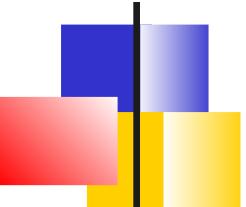
Merge Requests

Open 472 Merged 11,188 Closed 1,969 All 13,629

Last created ▾

Search or filter results...

MR Title	Created By	Updated
test MR	!13679 · opened 17 minutes ago by Mike Greiling	updated 14 minutes ago
Add docs for group issues page and group merge requests page	!13678 · opened 20 minutes ago by Victor Wu	updated 2 minutes ago
Docs update links guideline to inline links	!13677 · opened 36 minutes ago by Marcia Ramos	updated 34 minutes ago
WIP: Clean up new dropdown styles	!13676 · opened 50 minutes ago by Winnie Hellmann	updated 15 minutes ago
Greatly reduce test duration for git_access_spec	!13675 · opened 58 minutes ago by Robert Speicher	updated 20 minutes ago
Implement new system note icons	!13673 · opened about 3 hours ago by Bryce Johnson	updated less than a minute ago
WIP: Prepare 9.5 RC6	!13672 · opened about 3 hours ago by Jose Ivan Vargas Lopez	updated about an hour ago
Use Gitaly 0.33.0	!13671 · opened about 4 hours ago by Jacob Vosmaer (GitLab)	updated about 4 hours ago
[WIP] Make the import take subgroups into account	!13670 · opened about 5 hours ago by Bob Van Landuyt	updated about 5 hours ago



# Revue des MRs

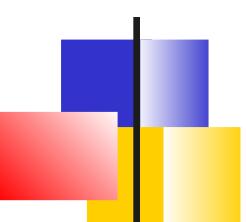
---

Lors de l'examen des différences liées à une MR, il est possible de démarrer une **revue**.

Cela permet de créer des commentaires sur les lignes de code modifiées.

Dans une MR, accéder à l'onglet *Changes*

- Lors de la saisie d'un commentaire, activer le bouton *Start review*
- Puis les boutons *Add to review*
- Et finalement *Finish Review*



# Commentaires et discussions

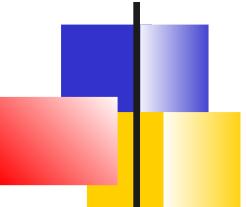
---

Un commentaire sur une MR, une ligne de code ou autre peut être transformé en discussion (thread)

- Une discussion contient un fil de messages et a un statut : « A résoudre » ou « Résolu »

Il est possible de

- voir toutes les discussions non résolues
- De déplacer les discussions non résolues vers une issue
- d ‘empêcher une fusion si des discussions sont non-résolues



# Squash

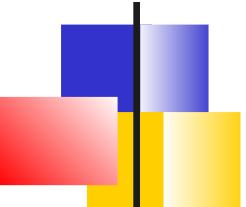
---

Lors d'un merge, il est possible de convertir tous les commits de la feature branch en un seul et donc d'avoir un historique plus concis : **squash**

Le message de commit est alors :

- Repris du premier message de commit multi-lignes
- Le titre du merge request si il n'y a pas de messages multi-lignes

Il peut être personnalisé au moment du merge

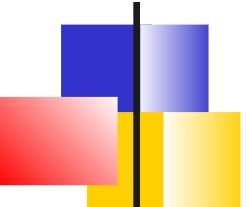


# Méthodes de merge

---

Différentes méthodes de merge peuvent être configurés au niveau du projet. Elles ont une influence sur l'historique du projet :

- **Merge commit (défaut)** : Chaque fusion crée un commit de merge
- **Merge commit avec historique semi-linéaire** : Chaque fusion crée un commit de merge mais la fusion n'est possible que si c'est une fast-forward  
Si un conflit arrive, l'utilisateur a la possibilité de rebaser
- **Fast-forward merge** : Pas de commit de merge, seules les fast-forward sont possibles  
Si un conflit arrive, l'utilisateur a la possibilité de rebaser

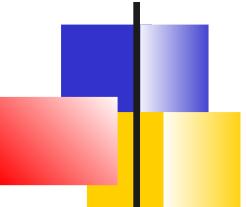


# Vérifications avant merge

---

Il est possible de configurer plusieurs vérifications avant un merge :

- Vérifier que la pipeline réussisse
- Vérifier que les discussions sont closes
- Imposer un approbateur particulier
- ...



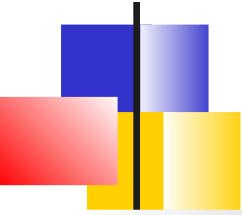
# Les outils de build

---

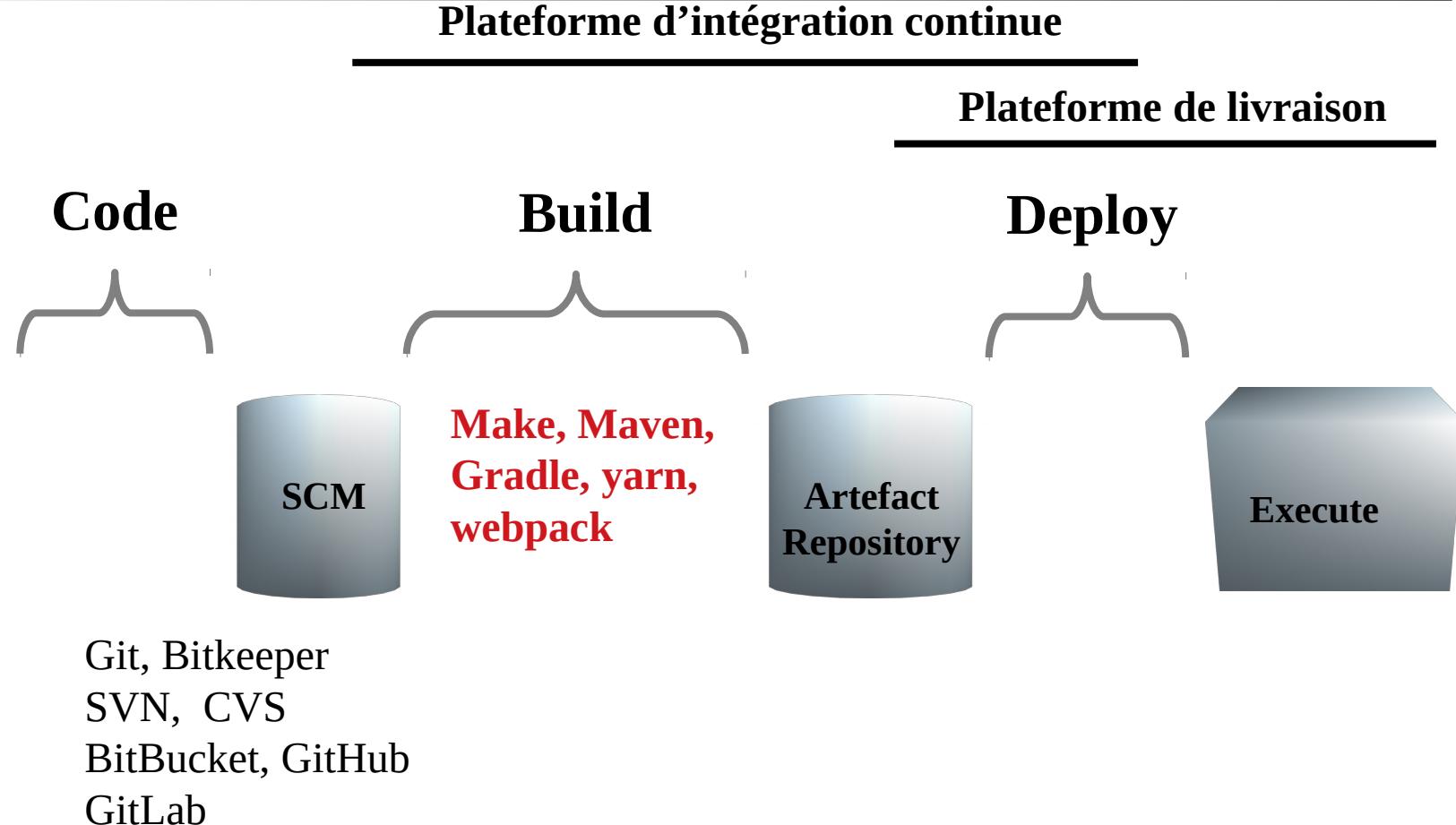
## **Caractéristiques des outils de build**

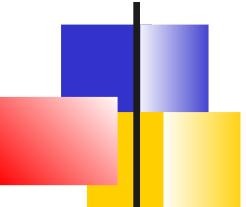
L'exemple de Maven

Release et dépôts d'artefact



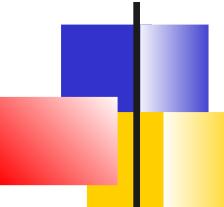
# Les outils de build dans le cycle de vie





# Pré-requis d'un build

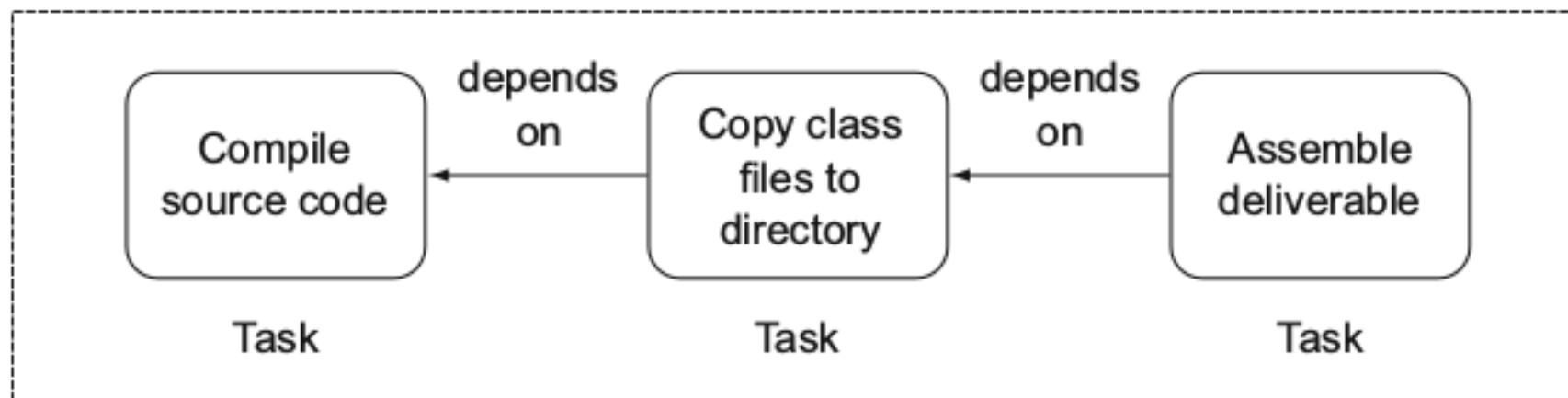
- **Proscrire les interventions manuelles**  
sujettes à erreur et chronophage
- Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- **Sûr** : Confiance dans son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

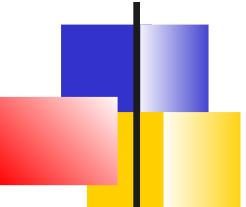


# Graphe de build

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** :





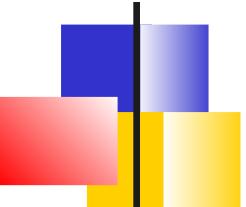
# Composants d'un outil de build

**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



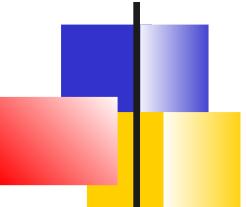
# Monde Java

---

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

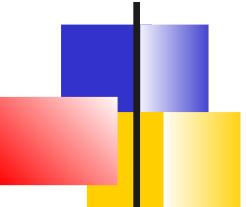
**Maven** : L'actuel. Gestionnaire de dépendances. Convention plutôt que configuration. Extension par mécanisme de plugin. Verbeux car XML

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



# Exemple pom.xml

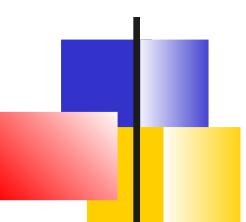
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <!-- Coordonnées du projet -->
  <groupId>org.dthibau</groupId>
  <artifactId>forum</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>Forum</name>
  <!-- Dépendances du projet -->
  <dependencies>
    <dependency>
      <groupId>io.github.jhipster</groupId>
      <artifactId>jhipster</artifactId>
      <version>2.0.4</version>
    </dependency>
    <dependency>
      <groupId>com.jayway.jsonpath</groupId>
      <artifactId>json-path</artifactId>
      <version>2.0.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```



# Exemple pom.xml (2)

---

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-resources-plugin</artifactId>
      <version>${maven-resources-plugin.version}</version>
      <!-- Adaptation du cycle de build -->
      <executions>
        <execution>
          <id>docker-resources</id>
          <phase>validate</phase>
          <goals>
            <goal>copy-resources</goal>
          </goals>
          <configuration>
            <outputDirectory>target/</outputDirectory>
            <resources>
              <resource>
                <directory>src/main/docker/</directory>
              </resource>
            </resources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  <plugins>
</build>
</project>
```



# Monde JavaScript/TypeScript

---

***npm, yarn*** : Gestion de dépendance

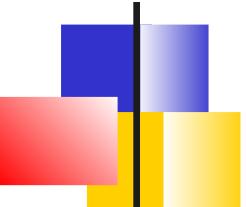
***webpack*** : Création de bundle pour la production

***grunt*** : Automatisation de tâches

***gulp*** : Optimisation, Minification de code

***yeoman*** : Générateur de code

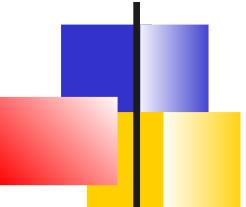
***tslint*** : Analyse de code source, règles de codage



# Exemple *package.json*

---

```
{ // Identification projet
  "name": "forum",
  "version": "0.0.0",
  "description": "Description for forum",
  "private": true,
  "license": "UNLICENSED",
  "cacheDirectories": [
    "node_modules"
  ], // Dépendances
  "dependencies": {
    "@angular/common": "4.3.2",
    "@angular/forms": "4.3.2",
    "@angular/http": "4.3.2",
    "@angular/platform-browser": "4.3.2",
    "@ng-bootstrap/ng-bootstrap": "1.0.0-beta.5",
    "bootstrap": "4.0.0-beta",
  }, // Dépendances pour le développement
  "devDependencies": {
    "@angular/cli": "1.4.2",
    "@angular/compiler-cli": "4.3.2",
    "@types/jasmine": "2.5.53",
    "webpack": "3.6.0",
    "webpack-dev-server": "2.8.2",
  }, // Moteur
  "engines": {
    "node": ">=6.9.0"
  }, // Raccourci
  "scripts": {
    "start": "yarn run webpack:dev",
    "serve": "yarn run start",
    "build": "yarn run webpack:prod",
    "test": "karma start src/test/javascript/karma.conf.js",
  }
}
```



# Autres mondes

---

Python :

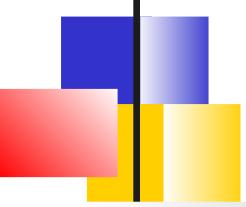
- *pip install* et *requirements.txt*  
<https://realpython.com/python-continuous-integration/>

PHP :

- Composer, *composer.json*

Ruby :

- RubyGems : *Rakefile* et *\*.gemspec*



# Les outils de build

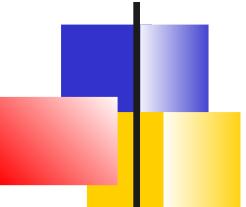
## L'exemple Maven

---

**Concepts généraux**

Personnalisation du build

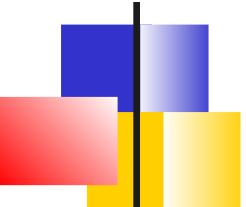
Utilisation des profils de build



# Qu'est ce que Maven

---

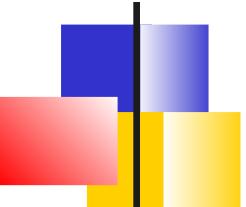
- Définition officielle (Apache) : Outil de gestion de projet qui fournit :
  - un **modèle** de projet
  - un ensemble de **standards**
  - un **cycle de vie (ou de construction)** de projet
  - un système de **gestion de dépendances**
  - de la logique pour exécuter des **objectifs** via des **plugins**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

---

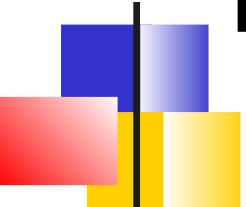
- Concept permettant d'alléger la configuration en respectant des conventions
- Par exemple, Maven presuppose une organisation du projet dont le but est de produire un jar:
  - **`${basedir}/src/main/java`** contient le code source
  - **`${basedir}/src/main/resources`** contient les ressources
  - **`${basedir}/src/test`**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



# Avant/Après

---

- Avant Maven, une personne était dédiée à l'élaboration des scripts de build
- Avec Maven, il suffit de :
  - Check-out du source
  - Exécuter ***mvn install***

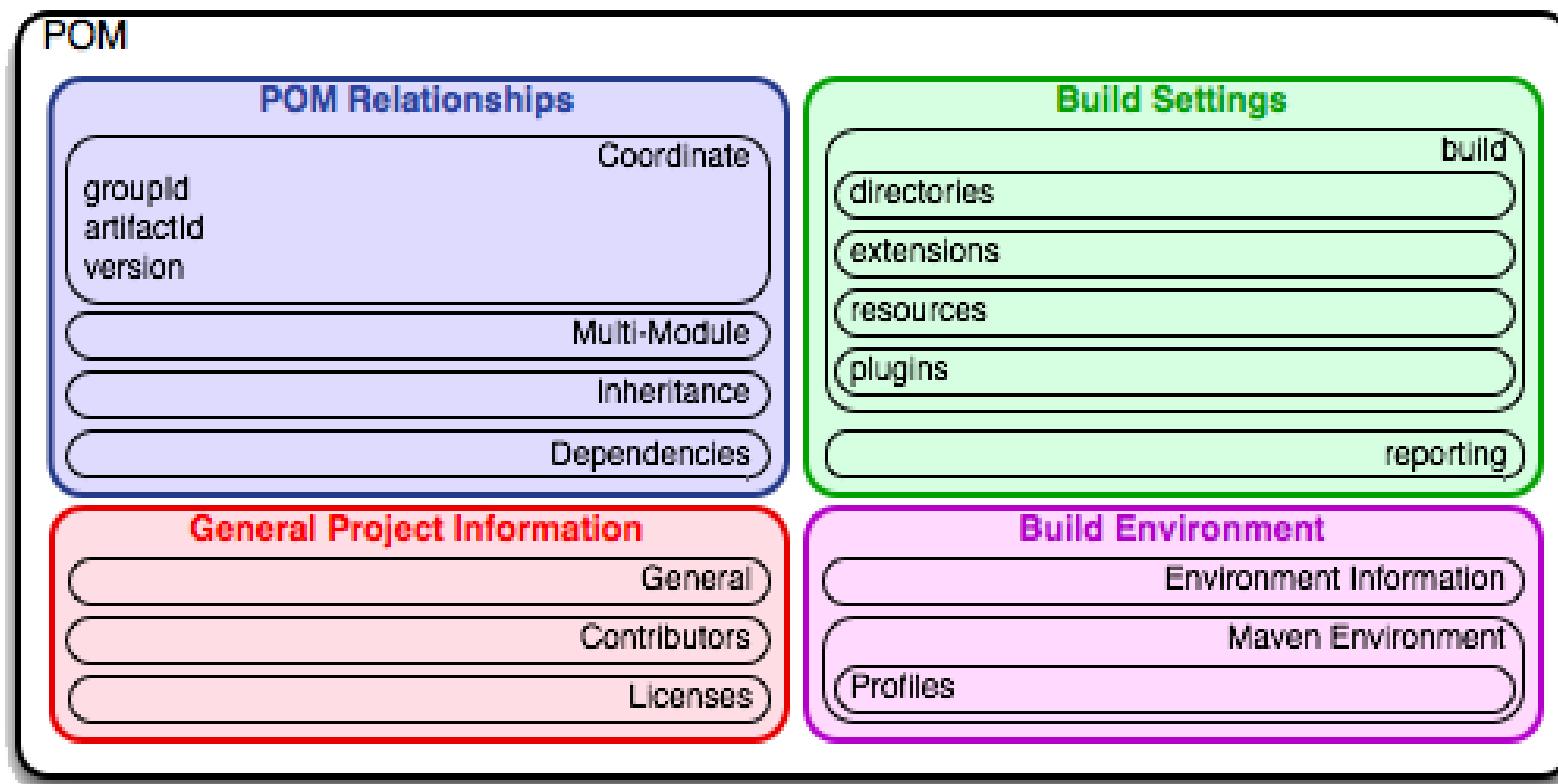


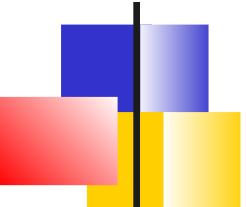
# Modèle conceptuel d'un projet

---

- 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
  - Personnalisation du build par défaut
  - ...
- Ce « Project Object Model » est décrit dans un fichier XML unique : **pom.xml**

# Contenu du POM

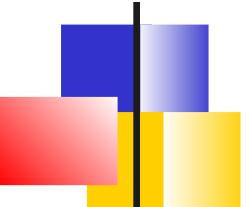




# Exemple pom.xml

---

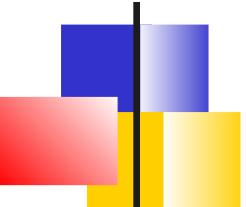
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook.ch03</groupId>
<artifactId>simple</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

---

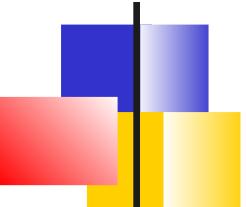
- **groupId** : Le groupe, la société. La convention de nommage stipule qu'il commence par le nom de domaine de l'organisation qui a créé le projet (*com.plb* ou *org.apache*, ...)
- **artifactId** ; Un identifiant unique à l'intérieur de *groupId* qui identifie un projet unique
- **version** : Une release spécifique d'un projet. Les projets en cours de développement utilise un identifiant spécial : SNAPSHOT.
- Le format de **packaging** est également important dans l'espace Maven, cependant il ne fait pas partie de l'identifiant projet. Les coordonnées *groupId:artifactId:version* rendent le projet unique.



# Propriétés

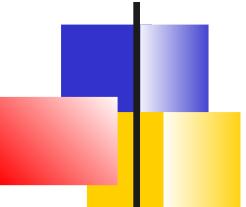
---

- Un POM peut inclure des propriétés :  
 `${project.groupId}-${project.artifactId}`
- Ces propriétés sont évaluées à l'exécution
- Maven fournit des propriétés implicites :
  - **env** : Variables d'environnement système
  - **project** : Le projet
  - **settings** : Accès au configuration de *setting.xml*
  - Propriétés **Java**
- On peut en définir via :  
`<properties> <foo>bar</foo> /properties>`



# Gestion des dépendances

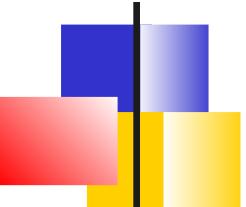
- La possibilité de localiser un artefact à partir de ces coordonnées permet d'indiquer les dépendances dans le projet POM.
- Le fait que le POM associé à tout artefact soit stocké dans le dépôt permet à Maven de résoudre les dépendances transitives. Le support pour les dépendances transitives est une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



# Dépendances transitives

---

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

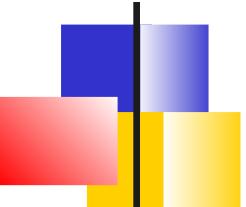


# Ajout de dépendances

---

L'ajout de dépendance est réalisé en ajoutant une balise **<dependency>** sous la balise **<dependencies>** et en indiquant les coordonnées Maven

```
<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

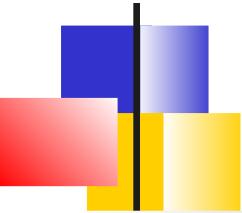
```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
</dependency>
```

# Exclusion de dépendances

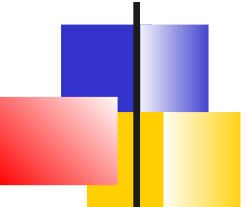


```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>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

# Où 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



# Commandes Maven

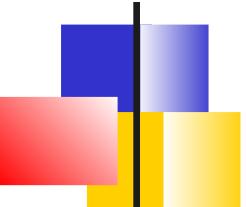
---

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

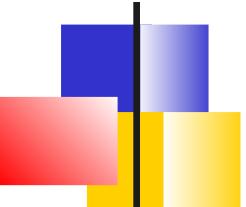


# Plugins et objectifs

---

- 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*, ...

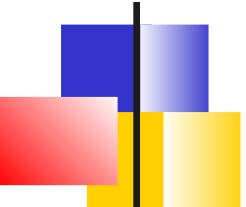
Maven fournit également la possibilité de définir ses propres plugins. Un plugin spécifique peut-être écrit en Java, ou Ant, Groovy, beanshell, ...



# Maven Plugins et objectifs

---

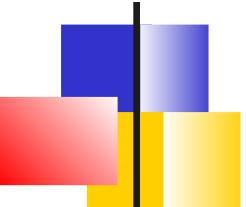
- Un objectif est une tâche spécifique qui peut être exécutée en standalone ou comme faisant partie d'un build plus large.
- La syntaxe d'exécution est la suivante :  
**mvn <plugin>:<objectif>**
- Un **objectif est l'unité de travail** dans Maven  
Exemples : l'objectif de compilation dans le compilateur 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.



# Cycle de vie Maven

---

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

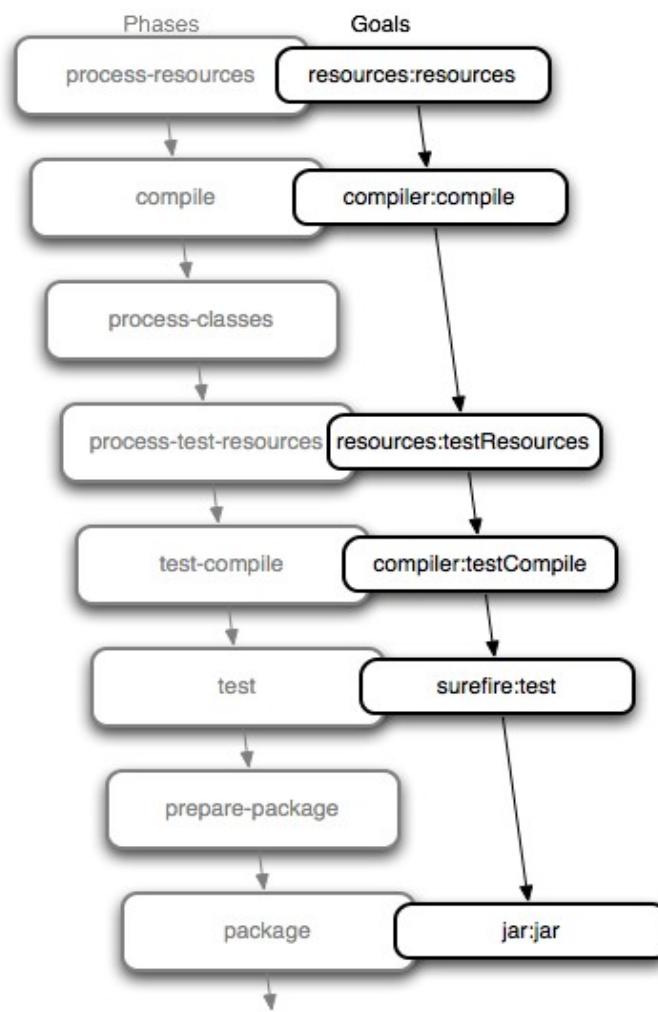


# Phase et objectifs

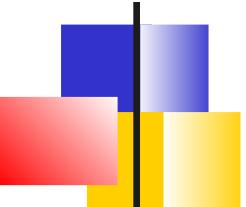
---

- 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 par défaut



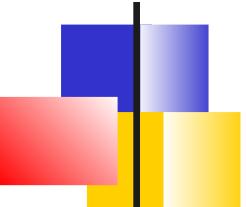
Note: There are more phases than shown above, this is a partial list



# 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 crée les fichiers résultats, termine le build en cas d'échec
- ***package / jar:jar*** : crée un fichier jar à partir du répertoire de sortie
- ***install/ install:install*** : Installe l'artefact dans le dépôt local. Par défaut *~/.m2*
- ***deploy / deploy:deploy*** : Déploie l'artefact vers le dépôt distant. Par défaut MavenCentral



# Exemple

---

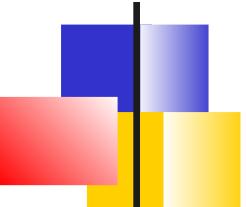
Exemple : Lancement des objectifs liés à la phase test suivi de l'appel de l'objectif *sonar* du plugin *sonar*

```
$ mvn test sonar:sonar
```

# POM parents et 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, de tous les fichiers POMs des projets parents, du fichier POM racine de Maven et des configurations spécifiques de l'utilisateur
- Afin de consulter le fichier réellement utilisé par Maven, il suffit d'utiliser le plugin d'aide :  
**\$ mvn help:effective-pom**



# POM Parent

---

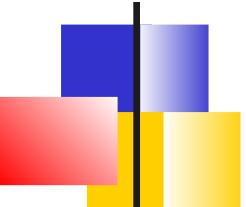
La référence vers le POM parent est indiqué via la balise **<parent>**

```
<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>
```

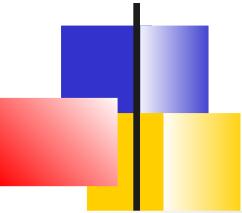


# Projets multi-modules

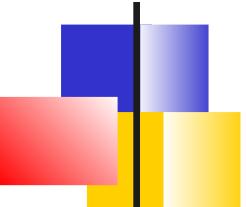
---

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

# POM parent



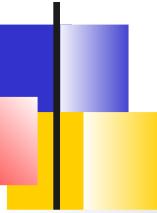
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook.ch07</groupId>
<artifactId>simple-parent</artifactId>
<packagingpackagingmodulesmodules
```



# POM d'un projet multi-modules

- Le POM parent définit les coordonnées Maven
- Il ne crée 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

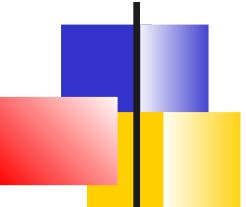
# POM des sous-modules



- 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>
...

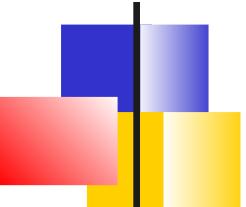
```



# Exécution de Maven

---

- Lors de l'exécution du build, 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
- La commande est naturellement exécutée sur le projet parent



# Version du projet

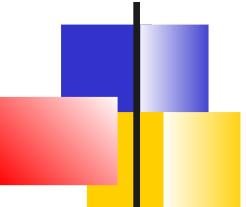
---

- 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

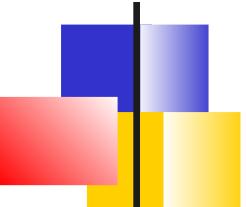


# Gestion des versions des dépendances

---

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

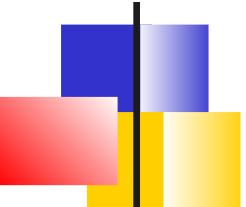


# `<dependencyManagement>`

---

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

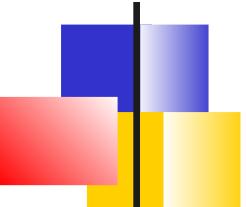
# POM parent



```
<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>
```



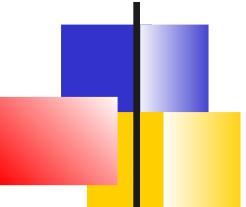
# Importation de dépendances BOM

---

Un BOM (Bills Of Material) est un POM spécifique utilisé uniquement pour contrôler les versions des dépendances d'un project

Il fournit ainsi un point central de configuration.

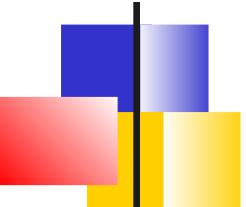
Si un projet enfant importe le BOM, il n'a plus besoin de spécifier les versions de ces dépendances.



# Exemple

---

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>baeldung</groupId>
  <artifactId>Baeldung-BOM</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <description>parent pom</description>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>b</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>c</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

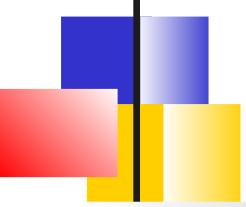


# 2 utilisations

---

## Héritage simple du BOM ou mieux importation

```
<project ...>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>baeldung</groupId>
                <artifactId>Baeldung-BOM</artifactId>
                <version>0.0.1-SNAPSHOT</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```



# Gestion des versions des plugins

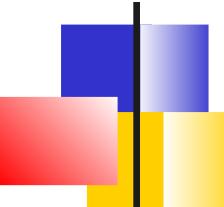
---

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

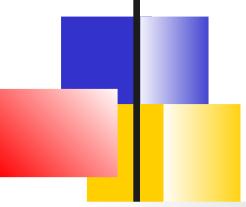
Dans un POM parent, la balise pluginManagement permet de fixer les versions des plugins utilisables ainsi que des propriétés de configuration

Les POMs enfants utilisent alors les mêmes plugins

# Exemple *pluginManagement*



```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
```

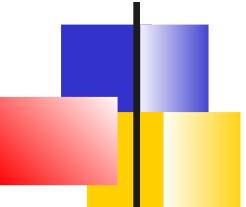


# Les outils de build

## L'exemple Maven

---

Concepts généraux  
**Personnalisation du build**  
Utilisation des profils de build

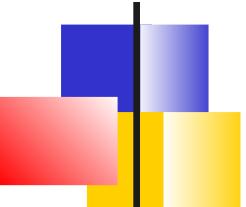


# Adaptation

---

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



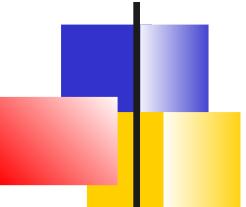
# Exemple *compiler:compile*

---

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 !

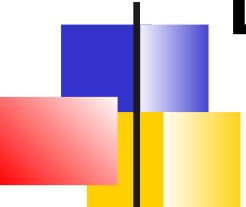
Pour changer ce comportement, il faut spécifier les versions visées dans le *pom.xml*



# Exemple

---

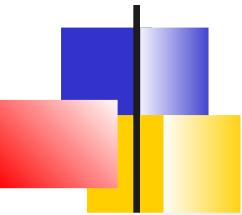
```
<project> ...
<build> ...
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins> ...
</build> ...
</project>
```



# 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

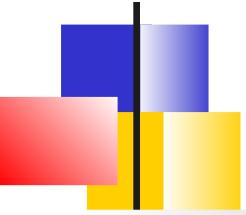


---

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>
```

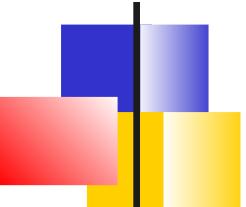


# Les outils de build

## L'exemple Maven

---

Concepts généraux  
Personnalisation du build  
**Utilisation des profils de build**



# Profils

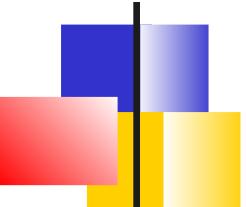
---

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

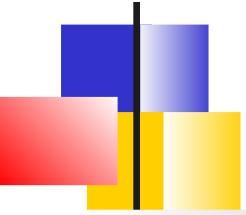
Les profiles, configurés dans le *pom.xml*, sont associés à des **identifiants**



# 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>
```



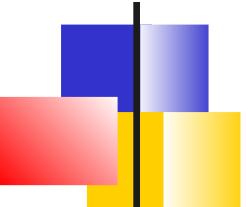
# `<profiles>`

---

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>`

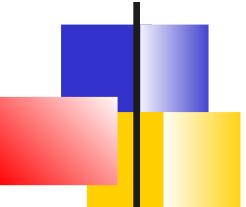


# Activation automatique des profils

---

Il est possible d'activer automatiquement un profil en fonction de :

- ✓ La version du JDK
- ✓ Les propriétés de l'OS
- ✓ Les propriétés Maven (ou l'absence de propriété)
- ✓ La présence d'un fichier

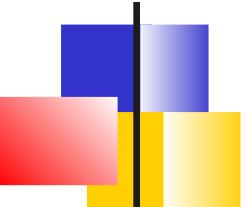


# Dépôts et Releasing

---

## **Les dépôts d'artefacts**

Processus de Release



# Introduction

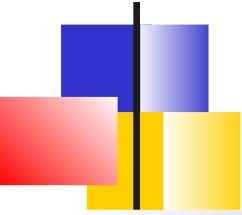
---

Le but principal d'un outil de build est d'une construire un **artefact** : le format d'exécution du logiciel

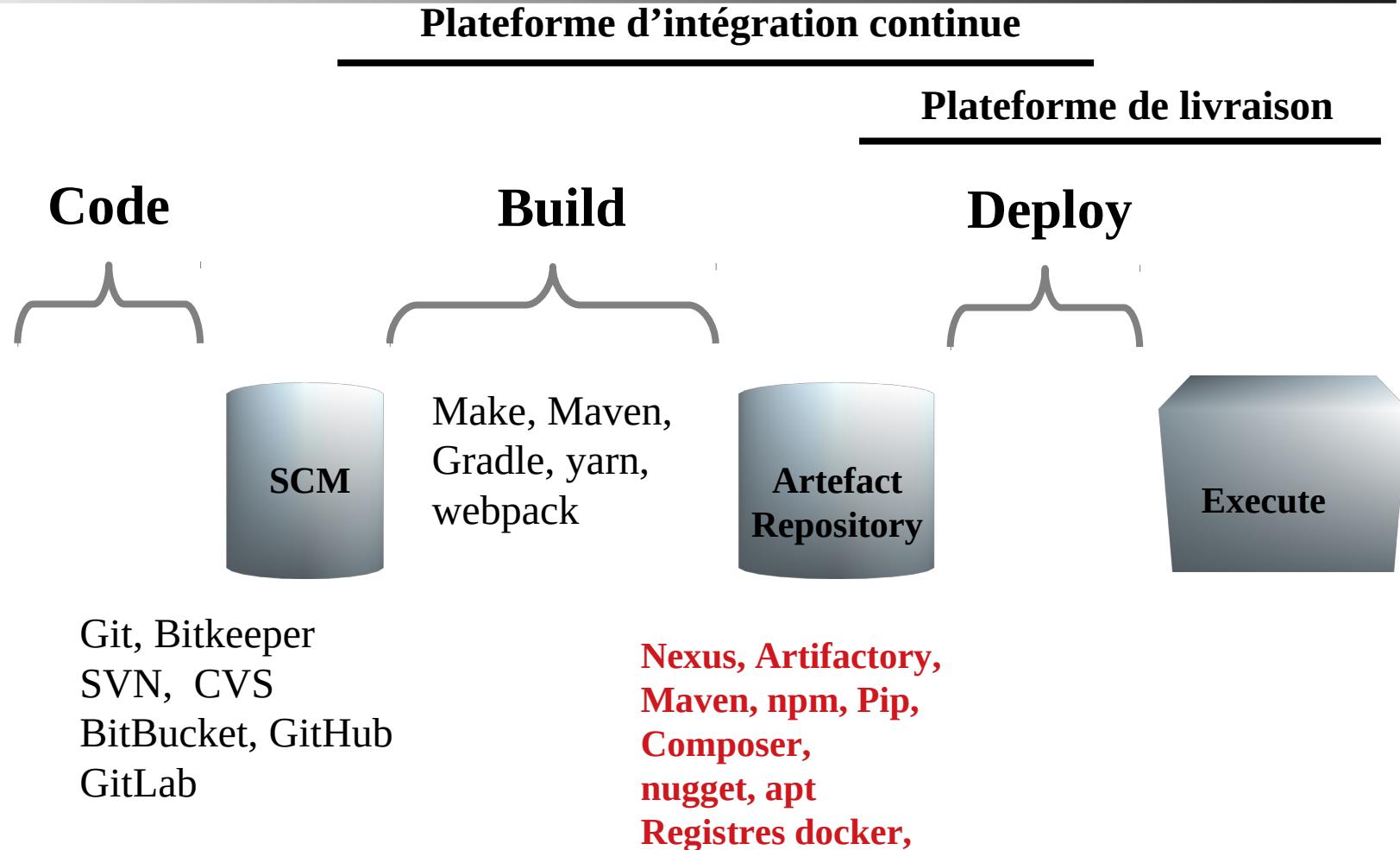
Les artefacts sont stockés et taggés dans des **dépôts**

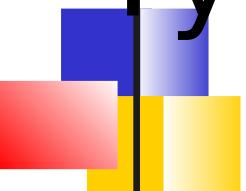
2 types d'artefacts :

- Les **SNAPSHOTS**, ils fournissent le dernier état (à peu près stable) de la version en cours de développement
- Les **releases**, ils sont immuables, taggés et correspondent à un tag des sources les ayant produits.



# Les gestionnaires de dépôts dans le cycle de vie

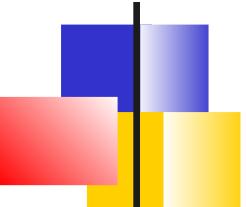




# Types de dépôts et méta-données

---

- Chaque type dépôt (Maven, Docker, npm) adopte une structure de stockage particulière et gère ses propres méta-données.
  - Par exemple pour Maven, structure correspondant au packaging Java et les méta-données : *pom.xml* et chiffrement du jar
- Certains outils, par exemple Nexus, sont capables de gérer plusieurs types de dépôts



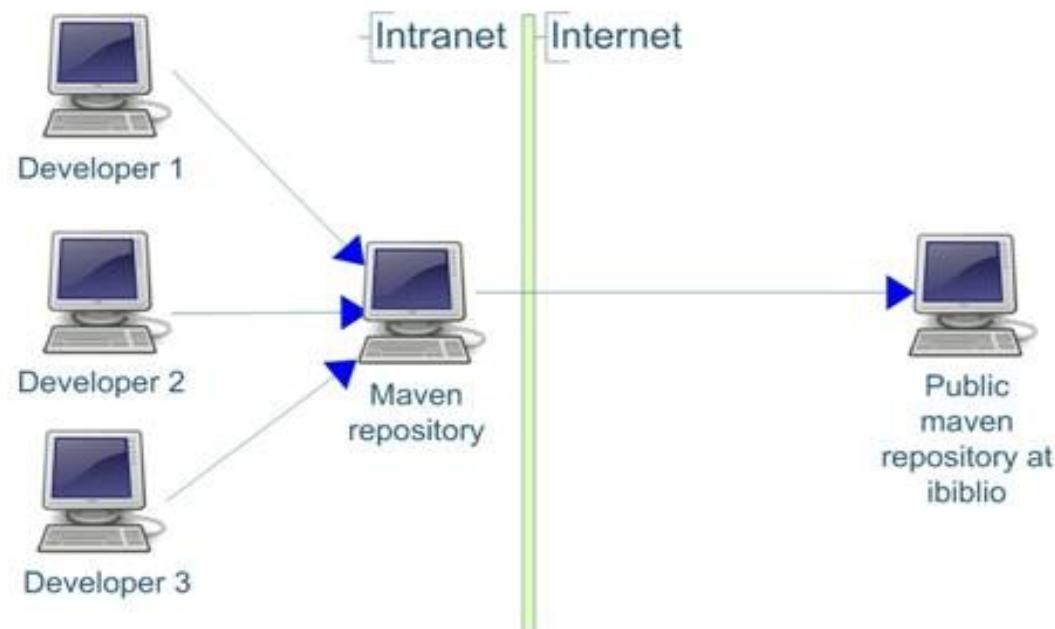
# Local/public/privé

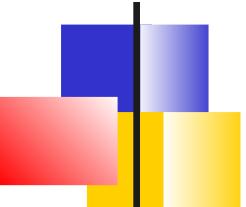
---

Pour un même projet, plusieurs types de dépôts sont utilisés

- Le dépôt **local** présent sur chaque poste de développeur
- Les dépôts **publics** fournis par Maven, RedHat, ...
- Les dépôts **miroirs** généralement configurés dans le fichier *settings.xml*
- Les dépôts **privés**, propres à une organisation ou entreprise

# Dépôt privé



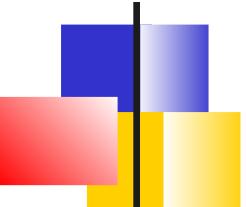


# Types de dépôt

---

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



# Publication vers le dépôt

---

La publication vers le dépôt s'effectue par des commandes en lignes (scp, http, ...) ou directement par l'outil de build

Exemple Maven :

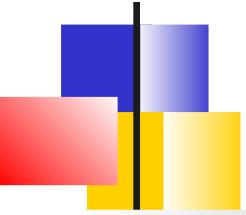
- Configuration du dépôt privé
- *mvn deploy*

Exemple Docker

- Configuration du registre privé
- *docker push*

Exemple Composer

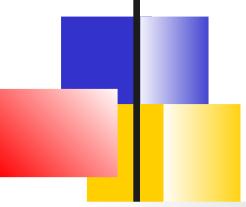
- *Pré-inscription sur Packagist*
- *Soumission de repository*



# Dépôts et Releasing

---

Les dépôts d'artefacts  
**Processus de Release**



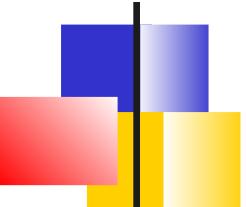
# Production d'une release

---

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

- Affecter/Modifier le n° de version
- Générer l'artefact et s'assurer que tous les tests sont OK
- Commiter et Tagger le SCM avec un n° de version
- Publier l'artefact taggé dans le dépôt d'artefact
- Incrémenter le n° version (et lui apposer le suffixe SNAPSHOT). Committer

Ce processus doit également être automatisé par la PIC.

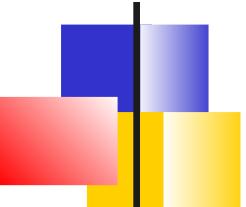


# Automatisation

---

L'automatisation de ce processus peut se faire

- Via des scripts utilisant, les commandes git, l'outil de build et éventuellement l'outil de déploiement vers le dépôt
- Via un support comme le plugin Release de Maven



# Tests et analyse statique

---

## **Typologie des tests**

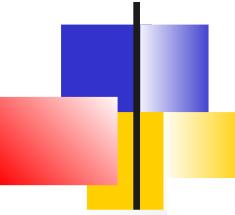
Tests unitaires / intégration

Tests fonctionnels

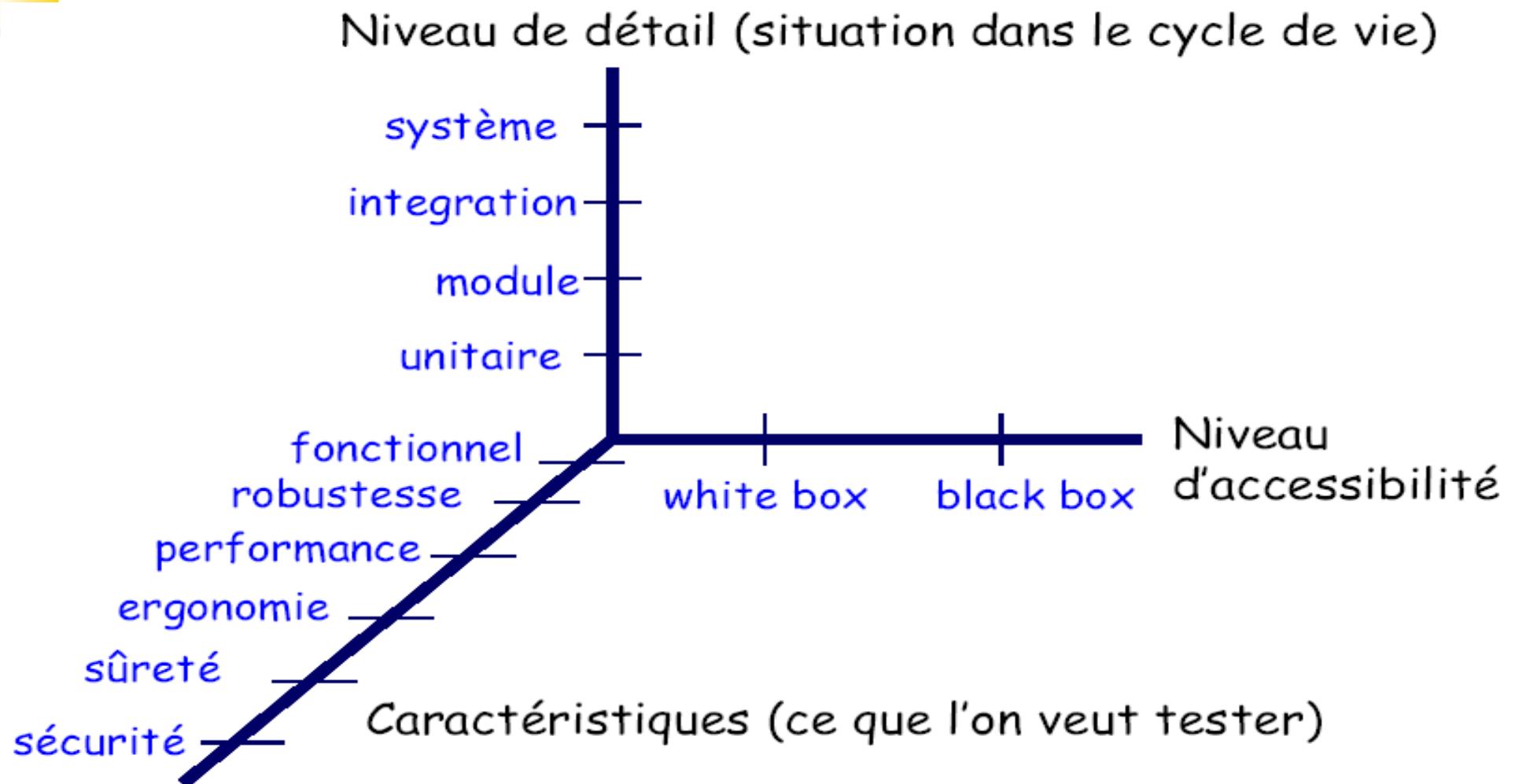
Tests de performance

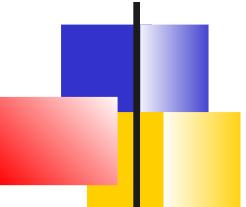
Tests d'acceptance

Analyse statique



# Types de test





# Types de test

---

Test Unitaire :

*Est-ce qu'une simple classe/méthode fonctionne correctement ?*

Test d'intégration :

*Est-ce que plusieurs classes/couches fonctionnent ensemble ?*

Test fonctionnel :

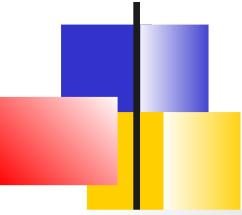
*Est-ce que mon application fonctionne ?*

Test de performance :

*Est-ce que mon application fonctionne bien ?*

Test d'acceptance :

*Est-ce que mon client aime mon application ?*



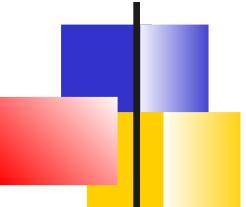
# Classification DevOps

---

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 et analyse statique

---

Typologie des tests

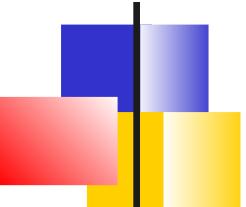
## **Tests unitaires / intégration**

Tests fonctionnels

Tests de performance

Tests d'acceptance

Analyse statique

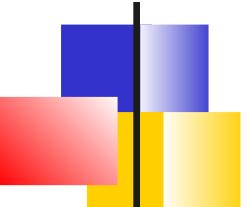


# Introduction

---

Les tests unitaires/d'intégration sont des tests de type white box mise au point et géré par les développeurs uniquement.

- Ils ont pour but de valider des spécifications fines (non compréhensibles du métier)
- Ils permettent d'augmenter la confiance dans l'implémentation produite
- Ils sont en général associés à une méthodologie TDD. Les tests à posteriori étant rarement efficaces



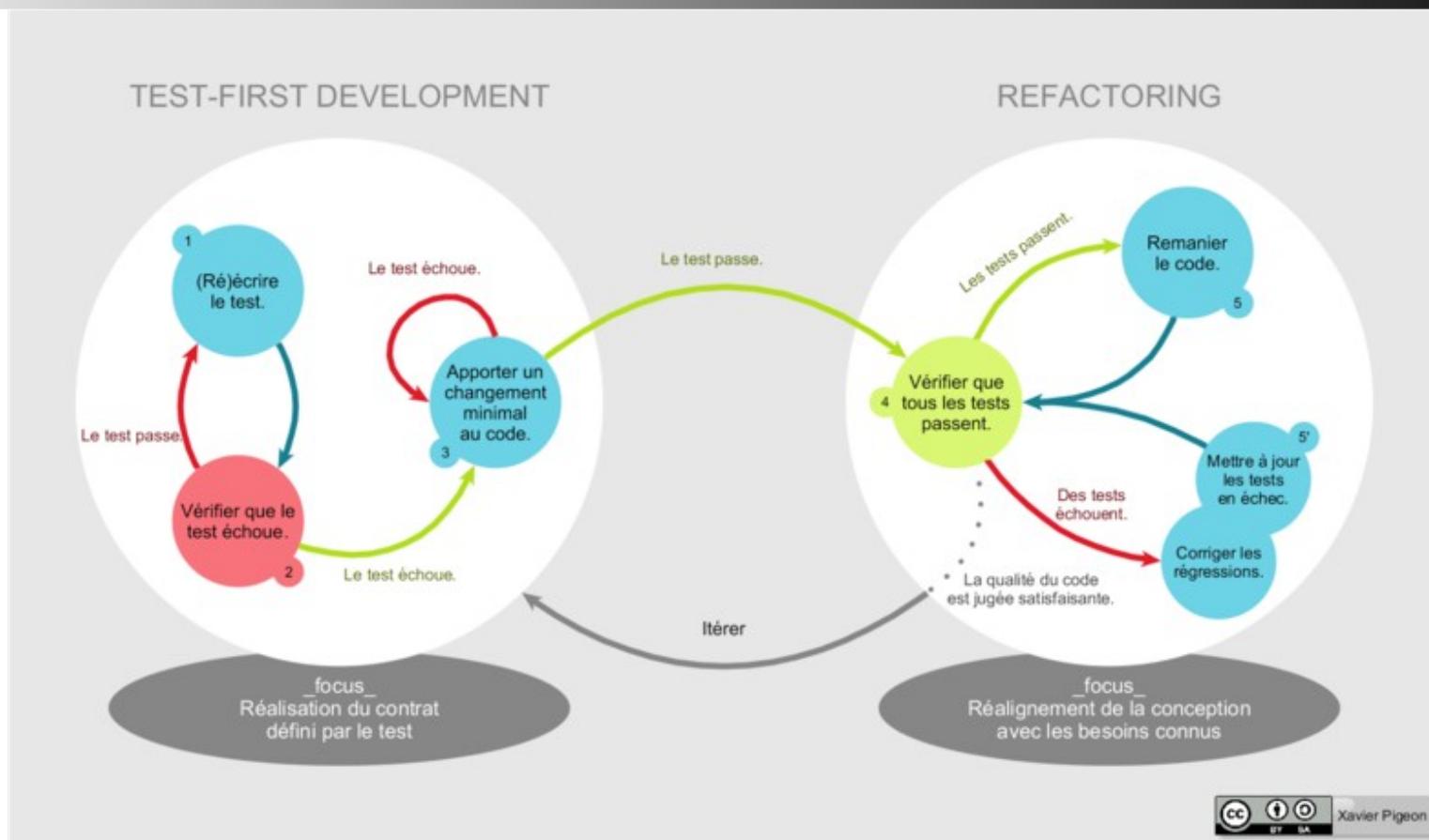
# Test Driven Development

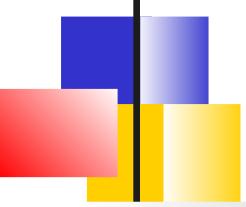
---

Le **TDD** est une méthodologie dont le développement est guidé par le test :

- Le développeur écrit le résultat attendu de la fonction à tester au sein de la classe de test
- Il écrit la classe fonctionnelle en provoquant une erreur pour vérifier que la classe de test détecte les erreurs
- Il finalise l'algorithme de la classe à tester
- Il vérifie que le test ne détecte plus d'erreur. Si nécessaire, il ajoute les tests aux cas limites.

# Cycle TDD

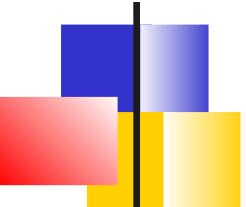




# Pratiques et recommandations :

## Au niveau individuel

- 
- Exécuter vos tests fréquemment
  - Ne pas écrire trop de tests en une seule fois
  - Ne pas écrire de trop grands tests ou des tests qui testent plusieurs choses à la fois
  - Ne pas oublier les assertions
  - Ne pas écrire des tests pour les codes triviaux, les accesseurs par exemple



# Niveaux

---

Débutant :

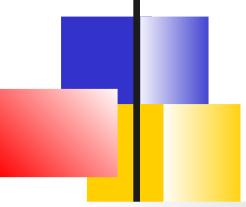
- capable d'écrire un test unitaire avant le vrai code
- capable d'écrire le code suffisant afin qu'un test passe

Intermédiaire :

- Lorsqu'un bug arrive, capable d'écrire le test avant la correction
- Capable de décomposer une fonctionnalité d'un programme en une séquence de tests unitaires à écrire
- Capable de factoriser des éléments réutilisables à partir de tests unitaires existants
- Identifie des patterns de tests pour les cas récurrents. (Par ex. programme récursif)

Senior

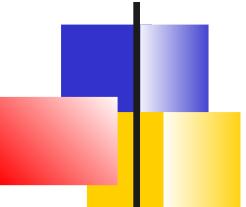
- Capable de formuler une road map de tests unitaires pour une partie d'un logiciel
- Capable de d'appliquer la TDD pour différentes classes de langage : Orienté objet, fonctionnel, événementiel
- Capable d'appliquer la TDD pour différents domaines techniques : Calcul, Interface utilisateur, Accès à la persistance, ....



# Pratiques et recommandations : Au niveau de l'équipe

---

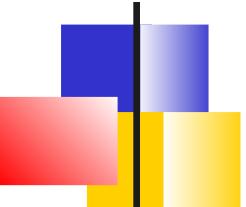
- Toute l'équipe doit adopter la TDD pas seulement quelques développeurs à l'intérieur de l'équipe
- Ne pas abandonner des tests, les maintenir
- S'assurer que l'exécution de tous les tests unitaires restent rapides pour le projet
- Effectuer du refactoring afin que le design soit testable



# Indices de réussite

---

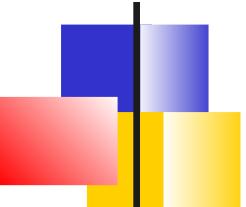
La **couverture de code** permet de donner une indication sur la réussite de l'approche. Un taux de couverture de 70 % est minimal ... mais pas suffisants, les tests doivent être pertinents



# Bénéfices

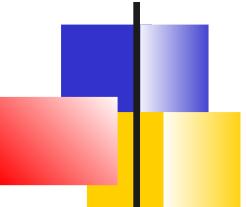
---

- Réduction très significative du nombre de bugs, en contrepartie d'un surcoût modéré du développement initial
- Précision de la spécification, documentation
- Effort de mise au point en fin de projet minimisé
- En testant avant de coder, pas besoin d'un testeur autre que le développeur
- Amélioration de la conception objet, meilleure cohésion et plus faible couplage



# Outils de Tests unitaires

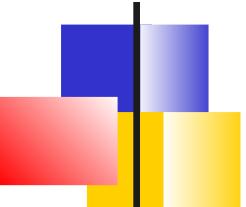
- Java : Junit (Base de la TDD) et TestNG, Mockito, Hamcrest
- Php : PHPUnit
- Javascript, TypeScript : Mocha, Jest (ReactJS), Jasmine, Karma (Angular)
- C# : Nunit, Mock
- Python : PyTest, nose2



# JUnit5 : Exemple

---

```
public class MathTest {  
    protected double fValue1, fValue2;  
  
    @BeforeEach  
    protected void setUp() {  
        fValue1= 2.0; fValue2= 3.0;  
    }  
  
    @Test  
    public void testAdd() {  
        double result= add(fValue1,fValue2);  
        assertTrue(result == 5.0);  
    }  
    @Test  
    public void testSubtract() {  
        double result= sub(fValue2,fValue1);  
        assertTrue(result == 1.0);  
    }  
}
```



# Assertions

---

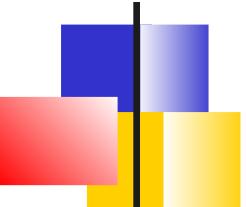
L'implémentation d'un test unitaire consiste généralement à :

- Préparer des données de tests
- Invoquer la méthode à tester
- Vérifier son résultat via une ou plusieurs assertions

Par exemple, JUnit fournit des méthodes d'assertions pour tous les types primitifs, les objets et les tableaux

L'ordre des paramètres est :

- Optionnellement, un message d'erreur si l'assertion échoue
- La valeur attendue
- La valeur réelle

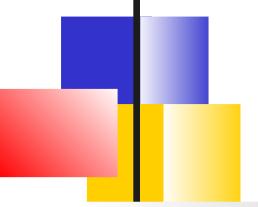


# Matcher

---

Certains frameworks (ex Hamcrest) introduisent la notion de **matchers** qui encapsulent toutes les conditions à tester sur un objet.

Les matchers sont expressifs et peuvent être combinés.



# Principaux matchers Hamcrest

---

**not** : Applique la négation au matcher encapsulé

**equalTo** : Test avec la méthode *equals*

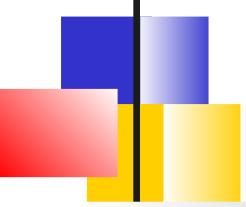
**is** : décorateur pour la lisibilité

**hasToString** : Test le retour de la méthode  
*toString*

**instanceOf, isCompatibleType** : test sur le type

**notNullValue, nullValue** : Test pour *null*

**sameInstance** : Test l'identité



# Principaux matchers Hamcrest (2)

***hasEntry, hasKey, hasValue*** : Test si une map contient une entrée, une clé une valeur

***hasItem, hasItems*** : Test si une collection contient des éléments

***hasItemInArray*** : Test si un tableau contient un élément

***hasProperty*** : Test la propriété d'un JavaBeans

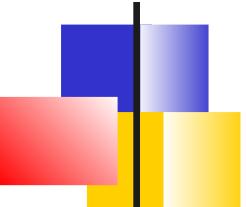
***closeTo*** : Test si une valeur flottante est proche d'une valeur donnée

***greaterThan, greaterThanOrEqualTo, lessThan,***  
***lessThanOrEqualTo***

***equalIgnoringCase*** : Test l'égalité de String sans prendre en compte la casse

***equalIgnoringWhiteSpace*** : Sans prendre en compte les espaces

***containsString, endsWith, startsWith*** : Test sur les String



# Combiner les matchers

---

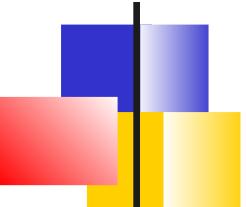
Les matchers peuvent être combinés.

Les méthodes ***both*** et ***either*** retournent des *CombinerMatcher* supportant la méthode *and()* ou *or()* :

```
assertThat(list, both(hasSize(1)).and(contains(42)));
assertThat(list, either(hasSize(1)).or(hasSize(2)));
```

Les méthodes ***allOf*** et ***anyOf*** offrent des raccourcis :

```
assertThat(list, allOf(hasSize(1), contains(42)));
assertThat("test", anyOf(hasSize(1), hasSize(2)));
```



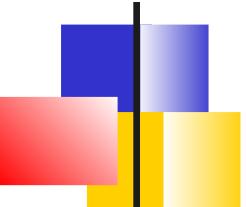
# Isolation

---

Les parties à tester doivent être isolées.

L'isolation apporte plusieurs bénéfices :

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



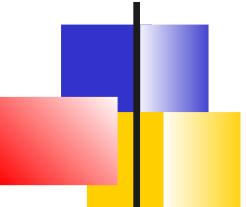
# Test unitaire et Mock objects

---

Quand la classe à tester interagit avec d'autres classes, il est quelquefois nécessaires d'isoler la classe à tester via des "MockObjects".

Simulation des accès à

- SGBD
- Réseau
- Fichiers



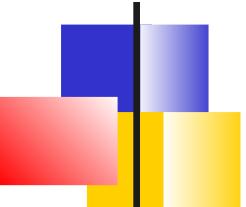
# Mockito

---

**Mockito** permet de créer et configurer les mock objects.

L'exécution d'un test consiste alors à :

- Spécifier les résultats des objets mockés en fonction de paramètres d'entrée. Les injecter dans le code à tester
- Exécuter le code à tester
- Optionnellement, vérifier que l'objet Mock a bien été appelé
- Vérifier le résultat attendu



# Configuration des mocks

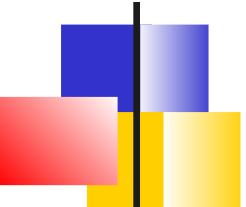
---

Mockito permet de configurer les valeurs de retour via son API

La chaîne de méthodes ***when(...).thenReturn(...)*** permet de spécifier une valeur de retour pour des paramètres prédéfinis

Exemple :

```
// Définir la valeur de retour pour getUniqueId()
MyClass test = mock(MyClass.class);
when(test.getUniqueId()).thenReturn(43);
```



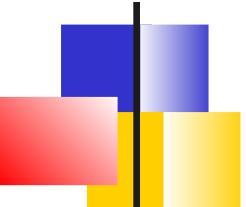
# Vérification des appels

---

Mockito garde une trace de tous les appels et de leurs paramètres ;

La méthode **verify()** permet de vérifier que les appels ont eu lieu

Dans ce cas, on ne vérifie pas le résultat mais plutôt qu'une méthode ait bien été appelée avec les bons paramètres.



# Exemple

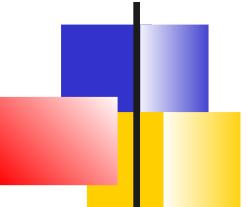
---

```
@RunWith(MockitoJUnitRunner.class)
public class MyControllerTest {

    @Mock
    MyService myServiceMock;

    @Test
    public void testMyMethod() {
        // Conditions and injections
        when(myServiceMock.callService(member)).thenReturn(result);
        MyController myController = new MyController();
        myController.setMyService(myServiceMock);

        // Méthode à tester
        String ret = loginController.login();
        // Assertion
        assertEquals(ret.equals(expected));
        // Vérification
        verify(myServiceMock).callService(member);
    }
}
```



# Distinction unitaire et intégration

---

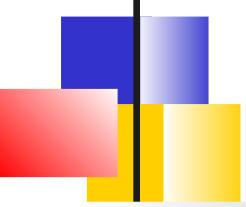
Les tests d'intégration ont pour objectifs de tester la collaboration entre plusieurs composants logiciels

La différence est finalement subtile :

- Un sous-ensemble du système est isolé

Ils nécessitent souvent des services back-end :  
Base de données, Serveurs webs. On utilise des en général des produits embarqués

Ils sont généralement plus longs à s'exécuter



# Bases de données embarquées

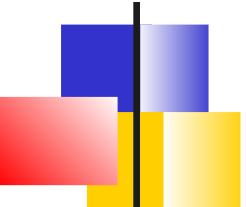
---

Une base de données embarquées est une simple librairie Java exposant JDBC qui se comporte comme un système de base de données indépendant

Elles sont rapides, légères et facilement configurables

Les 3 plus répandues sont :

- HSQLDB
- H2
- Derby



# Conteneurs embarqués

---

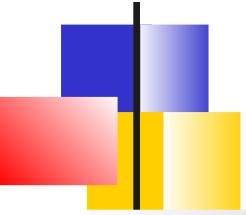
Dans la même idée, un conteneur embarqué est un serveur Java disponible sous forme de jar qui s'exécute à l'intérieur du runner des tests d'intégrations

La librairie offre les mêmes services qu'un serveur indépendant

Les conteneurs embarqués sont même utilisés en dehors des tests d'intégration (Exemple Spring Boot)

Les plus répandus sont :

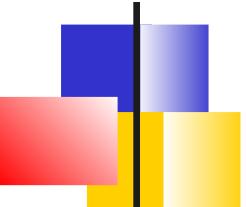
- Jetty
- Tomat
- TomEE (Tomcat EE)
- Undertow



# Tests et analyse statique

---

Typologie des tests  
Tests unitaires / intégration  
**Tests fonctionnels**  
Tests de performance  
Tests d'acceptance  
Analyse statique

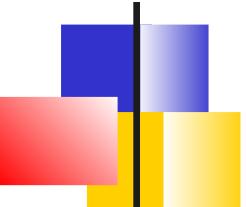


# Tests fonctionnels

---

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

- Dans le cas d'une application web, ils simulent ou pilotent un navigateur et vérifient les réponses fournies par le serveur. Ils sont fortement dépendants de l'interface utilisateur et peuvent être difficiles à automatiser et maintenir



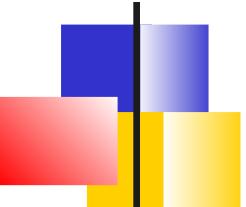
# Outils

---

Selenium, JMeter, HttpUnit : Capable d'envoyer des requêtes vers un serveur et de positionner des assertions sur les réponses obtenues

Protractor (Angular), Cypress (ReactJS)

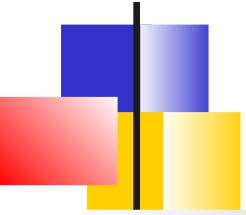
Symphony et WebTestCase



# Exemple : Selenium Web Driver

---

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



# Tests et analyse statique

---

Typologie des tests

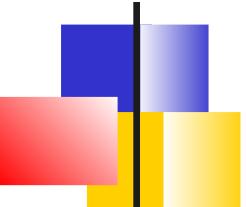
Tests unitaires / intégration

Tests fonctionnels

**Tests de performance**

Tests d'acceptance

Analyse statique

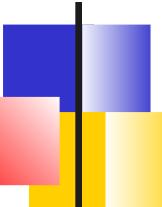


# Tests de performance

---

Les **tests de performance ou de charge** mesurent les temps de réponse ou débit d'un système en fonction de sa sollicitation.

- Ils sont itératifs et permettent d'optimiser l'usage de l'application.
- Ils nécessitent la mise en place :
  - de bancs de test (Isolation, sondes, instrumentation)
  - de benchmark (Pour comparer les optimisations)
  - la définition d'un modèle de charge (Anticiper les charges en production)



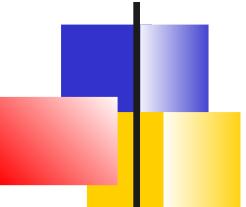
# Variantes des tests de charge

---

Test de dégradations des transactions : Simuler l'activité transactionnelle d'**un seul** scénario fonctionnel

Test de stress : Simuler l'activité maximale attendue tous scénarios fonctionnels confondus en **heures de pointe** de l'application

Test de robustesse, d'endurance, de fiabilité : Simuler une charge importante d'utilisateurs sur une **durée relativement longue**



# Variantes des tests de charge (2)

---

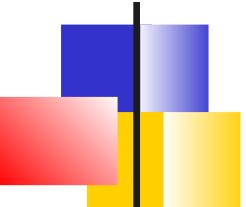
Test de capacité, test de montée en charge :

Simuler un nombre d'utilisateurs **sans cesse croissant** de manière à déterminer quelle charge limite le système est capable de supporter

Test aux limites : Simuler en général une activité **bien supérieure** à l'activité normale

Test de Benchmark : **Comparaisons** de logiciel, matériels, architectures,...

Tests de non-régression des performances, Tests de Composants, Tests de Volumétrie des données



# Panorama des outils

---

HP LoadRunner/Performance Center, ex produit Mercury Interactive, le leader du marché.

Oracle Load Testing, ex produit Empirix, racheté par Oracle Corporation en 2008

IBM Rational Performance Tester.

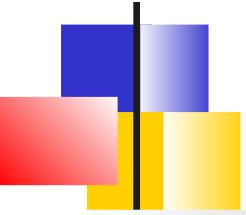
Apache JMeter (produit Open Source). Support *BlazeMeter*

Gatling : Open Source avec version payante

NeoLoad : Version SaaS

Ces outils proposent 2 modes d'utilisation :

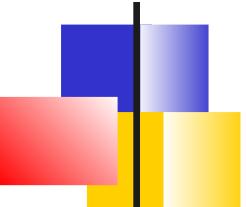
- Un mode graphique permettant l'élaboration des scénarios de test
- Un mode batch qui permet d'effectuer les tirs réels et de les automatiser



# Tests et analyse statique

---

Typologie des tests  
Tests unitaires / intégration  
Tests fonctionnels  
Tests de performance  
**Tests d'acceptance**  
Analyse statique



# Tests d'acceptance

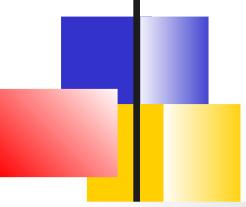
---

A l'origine : *eXtreme Programming (XP)* et *Test Driven Development (TDD)*

Le développeur et l'expert métier collaborent pour écrire des tests automatisés qui expriment le résultat souhaité par le métier

Les tests expriment ce que le logiciel doit faire pour que la partie prenante le trouve *acceptable*

Les tests commencent par échouer au moment de leur rédaction mais permettent de capter le besoin



# Behaviour Driven Development

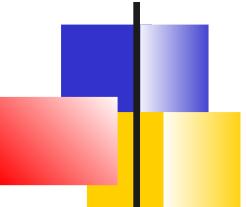
---

Le **BDD** consiste à développer un langage partagé que toute l'équipe agile s'approprie

=> L'utilisation du langage améliore la communication et minimise les fausses interprétations

Le BDD s'appuie sur le TDD et les tests d'acceptance écrits avant l'implémentation sont lisibles par toute l'équipe

Les tests interagissent directement avec le code des développeurs, mais sont écrits dans un langage que les experts métier peuvent comprendre



# Exemple

---

**Fonctionnalité:** Inscription

L'inscription doit être rapide et facile

**Scénario:** Inscription réussie

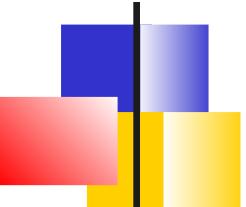
Les nouveaux utilisateurs doivent recevoir un e-mail de confirmation et être accueillis personnellement par le site une fois connecté

**Etant donné que** J'ai choisi de m'inscrire

**Lorsque** Je m'inscris avec des détails valides

**Alors** Je devrais recevoir un e-mail de confirmation

**Et** Je devrais voir un message d'accueil personnalisé



# Spécifications par l'exemple, exécutable

---

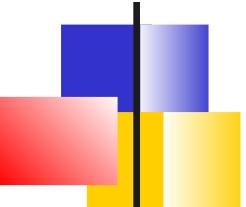
Les tests d'acceptance sont donc des spécifications exécutables

Comme les documents de spécifications traditionnels, ils peuvent être rédigés par l'équipe métier

La spécification se décrit par des exemples concrets qui illustre le besoin

Ils évoluent avec l'avancée du projet et sont versionnés et committés dans le dépôt de sources (SCM)

Ils sont intégrés dans les pipelines de CI/CD



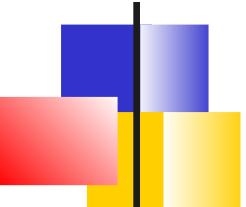
# BDD et agilité

La BDD assume l'utilisation d'une méthodologie agile.

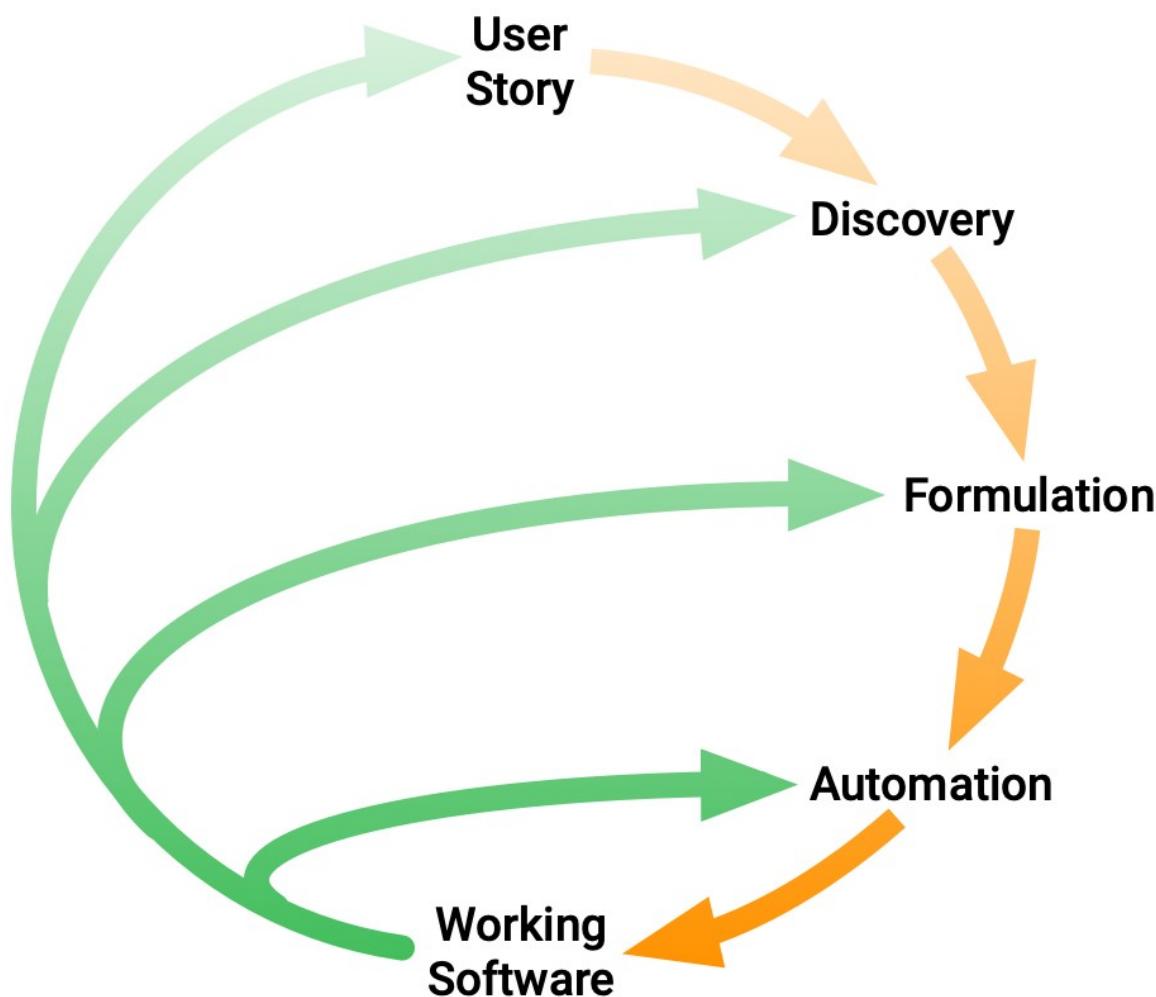
Le travail est planifié par des petits incrément de fonctionnalité, les User Stories

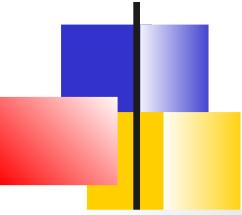
L'activité de BDD consiste à :

1. A partir d'une User Story, parlez d'exemples concrets de la nouvelle fonctionnalité pour explorer, découvrir et convenir des détails de ce qui devrait être fait.
2. Documentez ces exemples de manière automatisable dans la syntaxe Gherkin
3. Implémentez le comportement décrit par chaque exemple



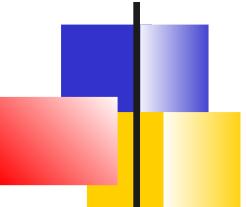
# BDD et Agilité





# Implémentations Cucumber

 Cucumber-JVM Java <small>official</small>	 Cucumber.js Node.js and browsers <small>official</small>	 Cucumber.rb Ruby, Ruby on Rails <small>official</small>
 Cucumber.ml OCaml <small>official</small>	 Cucumber.cpp C++ <small>official</small>	 Cucumber-Lua Lua <small>official</small>
 Android™ Java <small>official</small>	 Kotlin Cucumber-JVM with Kotlin <small>official</small>	 Cucumber-Tcl Tcl <small>official</small>
 Godog Go <small>official</small>	 Behat PHP <small>semi-official</small>	 Behave Python <small>semi-official</small>
 SpecFlow .NET C#, F#, VB.NET <small>semi-official</small>	 Cucumberish iOS, Swift, ObjC <small>semi-official</small>	 Test::BDD-Cucumber Perl <small>semi-official</small>
 Cucumber-Rust Rust <small>unofficial</small>	 unencumbered D <small>unofficial</small>	 Cucumber-Clojure Clojure <small>unmaintained</small>
 Cucumber-Gosu .Gosu Gosu <small>unmaintained</small>	 Cucumber-Groovy Groovy <small>unmaintained</small>	 Cucumber-JRuby JRuby <small>unmaintained</small>
 Cucumber-Jython .Jython Jython <small>unmaintained</small>	 Cucumber-Rhino Rhino <small>unmaintained</small>	 Cucumber-Scala Scala <small>unmaintained</small>



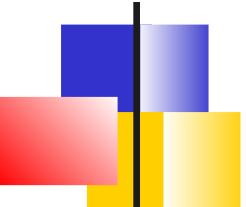
# Illustration du travail technique

---

Scenario: Some cukes

Given I have 48 cukes in my belly

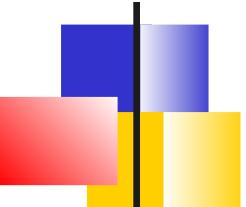
```
public class StepDefinitions {  
    @Given("I have {int} cukes in my belly")  
    public void i_have_n_cukes_in_my_belly(int cukes) {  
        System.out.format("Cukes: %n\n", cukes);  
    }  
}
```



# Tests et analyse statique

---

Typologie des tests  
Tests unitaires / intégration  
Tests fonctionnels  
Tests de performance  
Tests d'acceptance  
**Analyse statique**



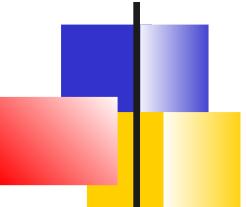
# Analyse statique

---

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

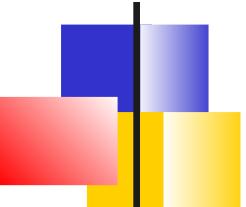
---

**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

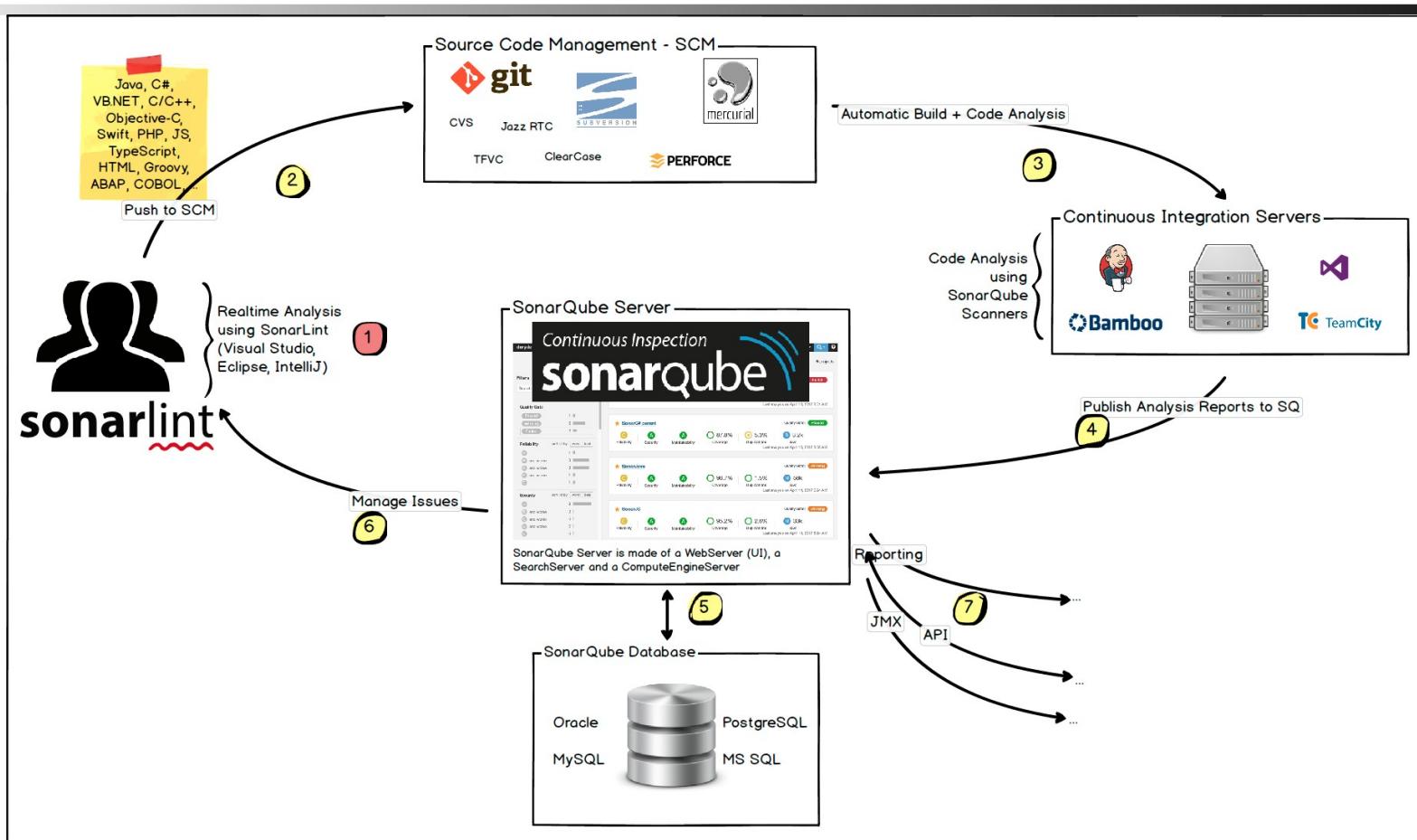
---

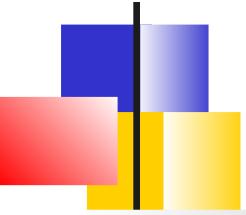
Les **portes qualité** définissent un ensemble de seuils pour les différents métriques. Le dépassement d'un seuil :

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

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

# Analyse continue





# Plateforme d'intégration continue

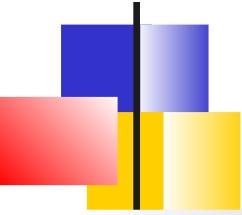
---

## **Concepts communs**

Pipelines typiques

Solutions

L'exemple Jenkins



# PICs dans le Cycle de vie

---

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

---

Plateforme de livraison

Code

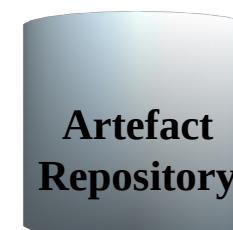


Git, Bitkeeper  
SVN, CVS  
BitBucket, GitHub  
GitLab

Build



Make, Maven,  
Gradle, yarn,  
webpack

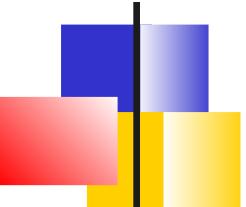


Nexus,  
Artifactory,  
Archiva

Deploy



Execute



# Plateforme d'intégration continue

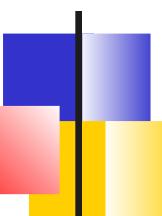
---

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

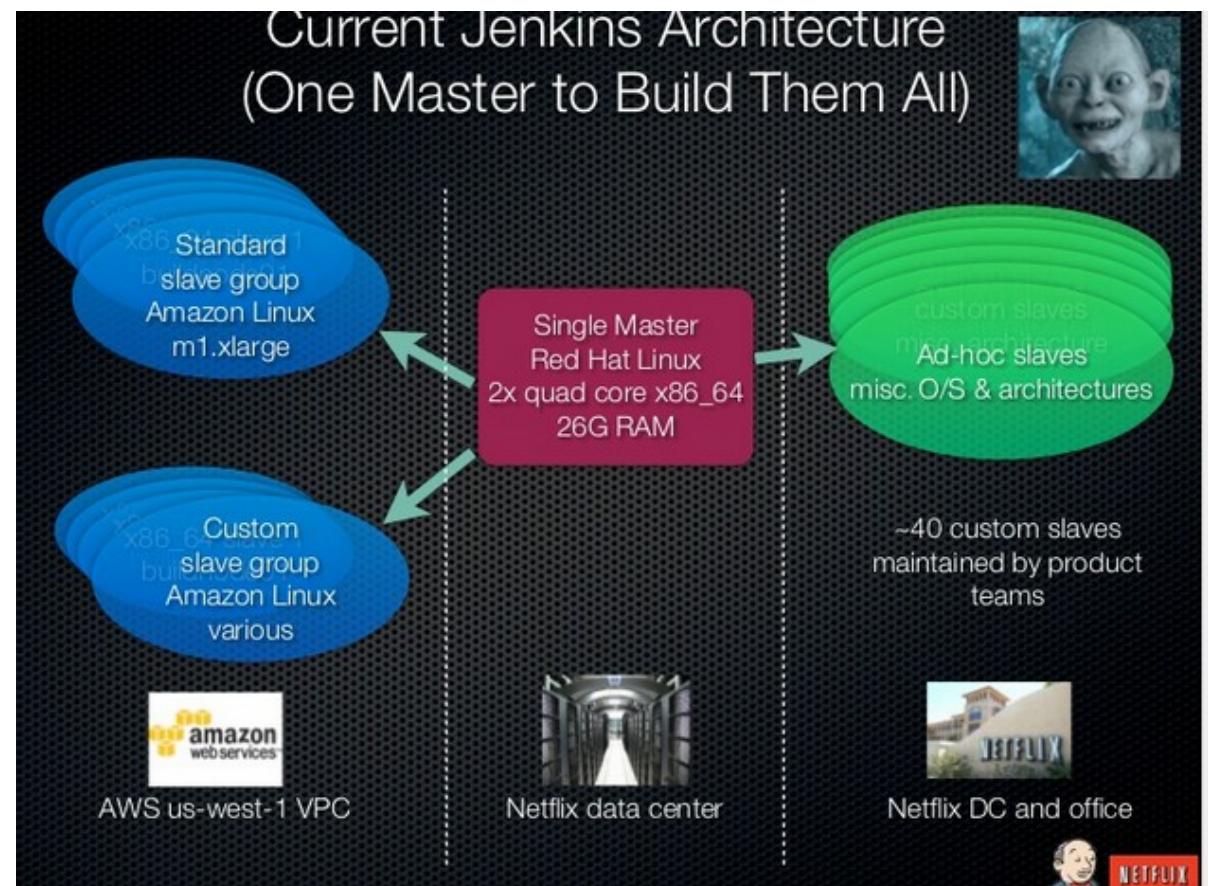
1,600 jobs :

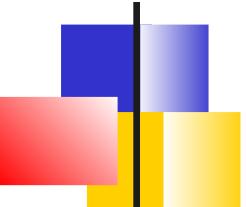
- 2,000 builds/jour
- 2 TB
- 15% build failures

=> 1 maître avec 26Go de RAM

=> Agent Linux sur amazon

=> Esclaves Mac. Dans le réseau interne





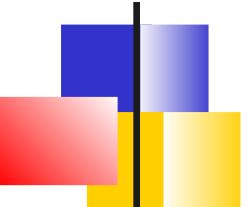
# Outils annexes, plugins

---

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



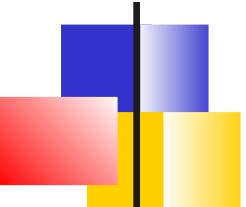
# Déploiement

---

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



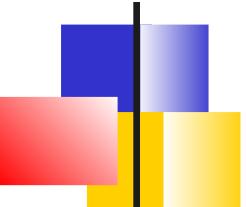
# Approche standard

---

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

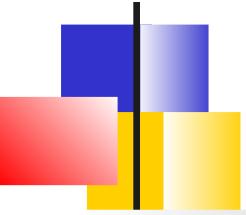
---

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

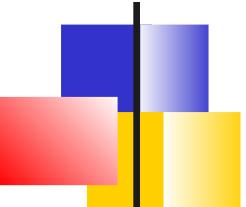
Ou dans certains cas, l'infrastructure de PIC nécessaire au projets (plugins et autre) est elle même conteneurisé et stocké dans le SCM



# Plateforme d'intégration continue

---

Concepts communs  
**Pipelines typiques**  
Solutions  
L'exemple Jenkins



# Principes

---

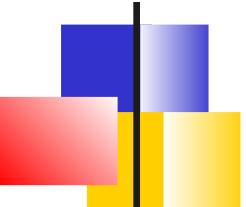
Les pipelines sont généralement découpées en **phases** représentant les grandes étapes de la construction.

Par exemple : Build, Test, QA, Production

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



# Phases typiques

---

**Build** : Construction de l'artefact et tests unitaires

**Test** : Tests d'intégration, Analyse qualité

**CI** : Déploiement en intégration

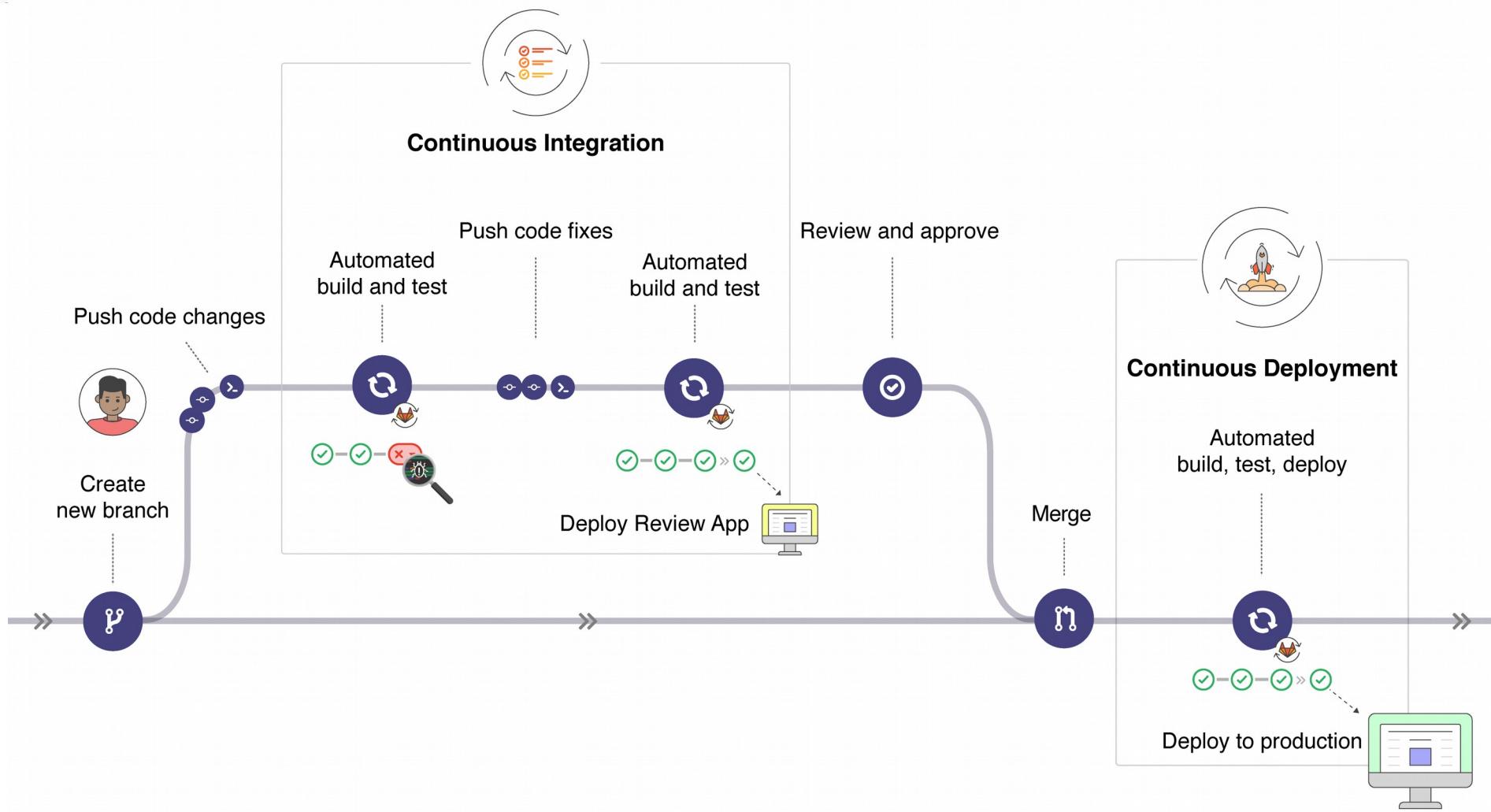
**Test post-déploiement** : Tests fonctionnels de performance sur l'environnement d'intégration ou de benchmark

**Cdelivery** : Déploiement d'une release vers un dépôt d'artefact

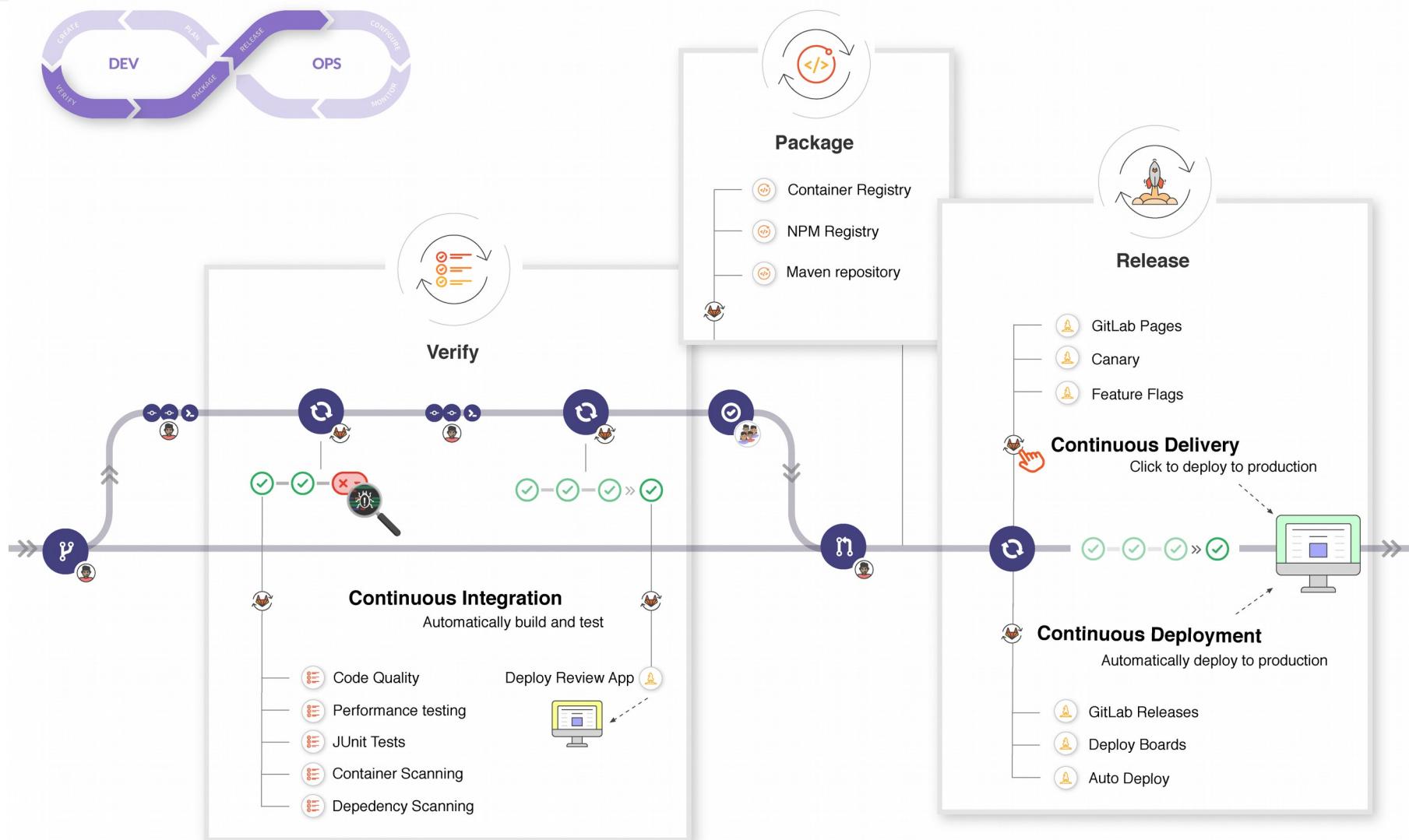
**Déploiement Staging** : Déploiement dans un environnement de pré-production. Tests automatisés ou fonctionnels

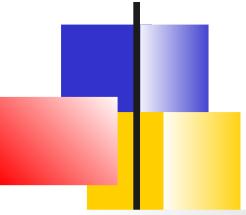
**Déploiement en production** : Tests de post déploiement (probes, etc..)

# AutoDevOps Gitlab CI



# En plus détaillé

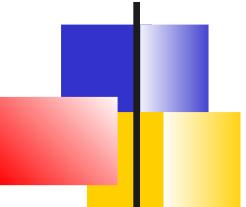




# Plateforme d'intégration continue

---

Concepts communs  
Pipelines typiques  
**Solutions**  
L'exemple Jenkins

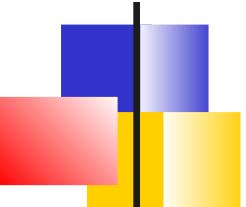


# Gitlab CI

---

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*

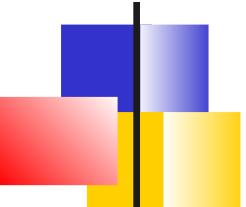
---

```
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
```



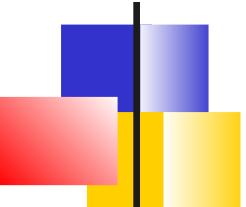
# Gitlab CI

---

Running 0    Finished 327    All 327

List of finished builds from this project

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



# Jenkins

---

Anciennement Hudson, (fork depuis le rachat par Oracle)

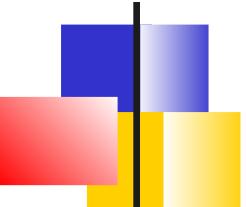
Encore, le plus répandu

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



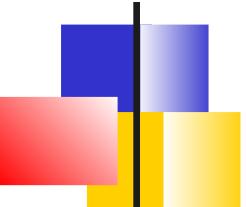
# Approche DevOps

---

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

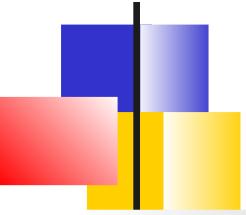
- Le fichier est commité et versionné dans  
un dépôt Git
- La description est effectuée via un langage  
spécifique DSL construit avec Groovy



# Illustration

---

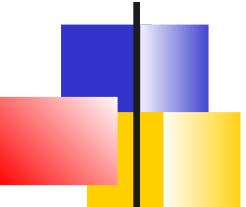
```
pipeline {  
    agent any  
  
    stages {  
        stage('Build') {  
            steps {  
                sh './mvnw -Dmaven.test.failure.ignore=true clean test'  
            } post {  
                always { junit '**/target/surefire-reports/*.xml' }  
            }  
        }  
        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'  
                    }  
                }  
            }  
        }  
    }  
}
```



# Plateforme d'intégration continue

---

Concepts communs  
Pipelines typiques  
Solutions  
**L'exemple Jenkins**



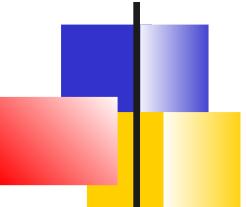
# Point d'entrée

---

Le point d'entrée est la page web « **Administre Jenkins** »

Les liens présents sont dépendants des plugins utilisés mais les plus importants sont :

- Configurer le système : Fonctionnement global, configuration du nœud, mail de l'administrateur, ...
- Configuration des outils : JDK, Maven, ...
- Gestion des plugins : Disponibilité, installation de plugin
- Gérer les nœuds : Ajouter ou supprimer des nœuds esclaves. Distribuer les builds sur les nœuds
- Gestion des crédentiels: Stocker des crédentiels afin que Jenkins interagisse avec d'autres plate-formes (Dépôts, Infrastructure de déploiement, ...)



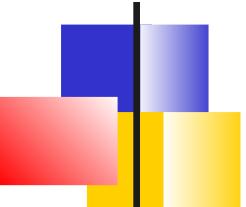
# Jenkins Pipeline

---

Jenkins Pipeline est une **suite de plugins** qui permettent d'implémenter et d'intégrer des pipelines CI/CD

- Chaque changement committé dans le SCM provoque le déclenchement de la pipeline .

La pipeline est modélisée **via du code et un langage spécifique (DSL)**



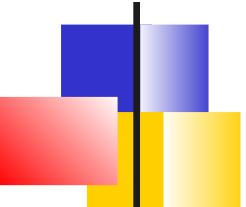
# *JenkinsFile*

---

Typiquement, la description de la pipeline est codée dans un fichier ***Jenkinsfile*** qui fait alors partie du projet

L'utilisation d'un *JenkinsFile* apporte plusieurs avantages :

- Création automatique de pipelines pour toutes les branches du SCM ou les Pull Request
- Revue de code et itération sur les Pipeline
- Historique des révisions de la Pipeline
- Unique source de vérité qui peut être vue et éditée par tous les membres de l'équipe DevOps.



# Syntaxe : Script ou déclaratif

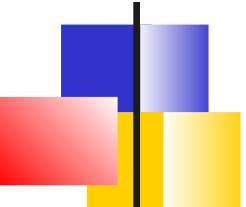
---

2 syntaxes coexistent pour l'instant :

- La syntaxe **déclarative** est plus simple.  
Elle est reconnaissable via un block pipeline :

```
pipeline {  
    /* insert Declarative Pipeline here */  
}
```

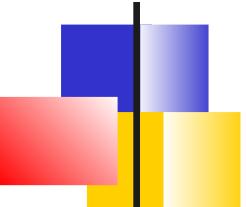
- La syntaxe **script** est un DSL basé sur Groovy. Il permet d'utiliser directement Groovy et donc est très flexible



# Illustration

---

```
// Declarative //
pipeline {
    agent any
    stages {
        stage('Build') {
            steps { echo 'Building..' }
        }
        stage('Test') {
            steps { echo 'Testing..' }
        }
        stage('Deploy') {
            steps {echo 'Deploying...' }
        }
    }
}
// Script //
node {
    stage('Build') { echo 'Building....' }
    stage('Test') { echo 'Building....' }
    stage('Deploy') { echo 'Deploying....' }
}
```

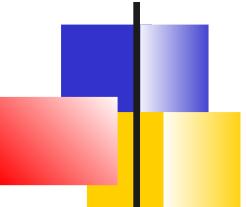


# Termes du DSL

---

Le DSL introduit plusieurs termes et concepts :

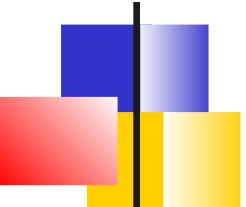
- **Stage (phase)** : Une phase définissant un sous ensemble de la pipeline.  
Par exemple : "Build", "Test", et "Deploy".  
Cette information est utilisée par de nombreux plugins pour améliorer la visualisation de l'avancement de la pipeline
- **Node (agent)** : Les travaux d'une pipeline sont exécutés dans le contexte d'un nœud. Plusieurs nœuds peuvent être déclarés dans une pipeline.
  - Les étapes contenues dans un bloc nœud sont démarrés par un job Jenkins
  - Un espace de travail est créé pour chaque nœud
- **Step (étape)** : Un simple tâche Jenkins.  
Par exemple, exécuter un shell.  
Les plugins liés à pipeline permettent principalement de définir de nouvelles tâches



# Exemple Jenkinsfile

---

```
pipeline {  
    agent any  
  
    tools {  
        // Install the Maven version configured as "M3" and add it to the path.  
        maven "M3"  
    }  
  
    stages {  
        stage('Build') {  
            steps {  
                // Get some code from a GitHub repository  
                git 'https://github.com/jglick/simple-maven-project-with-tests.git'  
  
                // Run Maven on a Unix agent.  
                sh "mvn -Dmaven.test.failure.ignore=true clean package"  
  
                // To run Maven on a Windows agent, use  
                // bat "mvn -Dmaven.test.failure.ignore=true clean package"  
            }  
  
            post {  
                // If Maven was able to run the tests, even if some of the test  
                // failed, record the test results and archive the jar file.  
                success {  
                    junit '**/target/surefire-reports/TEST-*.xml'  
                    archiveArtifacts 'target/*.jar'  
                }  
            }  
        }  
    }  
}
```



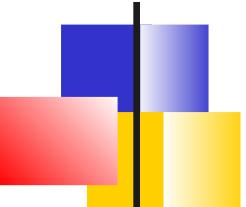
# Steps / tâches

---

En fonction des plugins installé, les tâches/steps disponibles sont :

- Invoquer un shell
- Invoquer les outils de build (Maven, Gradle, ...)
- Enregistrer et publier des tests
- Archiver des artefacts dans un dépôt
- Publier un artefact dans un environnement d'intégration ou de production
- ...

Les aides proposées par Jenkins sont dépendantes des plugins installés



# Documentation

---

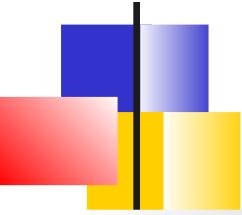
La documentation est incluse dans Jenkins.

Elle est accessible à

*localhost:8080/pipeline-syntax/*

Les utilitaires « ***Snippet Generator*** » et  
« ***Declarative Directive Generator*** » sont  
des assistants permettant de générer des  
fragments de code selon les 2 syntaxes

Les choix disponibles sont dépendants des  
plugins installés



# Snippet Generator

---

## Steps

Sample Step

stage: Stage



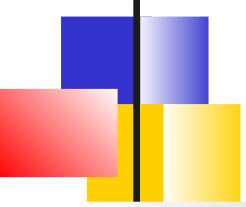
Stage Name

Deploy



**Generate Pipeline Script**

```
stage('Deploy') {  
    // some block  
}
```

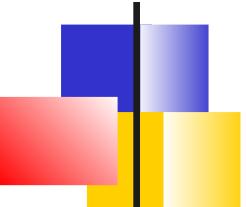


# Références Variables globales

---

En plus des générateurs, Jenkins fournit un lien vers le « **Global Variable Reference** » qui est également mis à jour en fonction des plugins installés.

Le lien documente les variables directement utilisable dans les pipelines

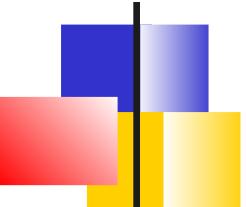


# Variables globales par défaut

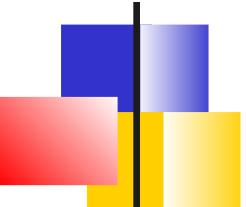
---

Par défaut, *Pipeline* fournit les variables suivantes :

- **env** : Variables d'environnement.  
Par exemple : `env.PATH` ou `env.BUILD_ID`.
- **params** : Tous les paramètres de la pipeline dans une Map.  
Par exemple : `params.MY_PARAM_NAME`.
- **currentBuild** : Encapsule les données du build courant.  
Par exemple : `currentBuild.result`,  
`currentBuild.displayName`



# Syntaxe déclarative



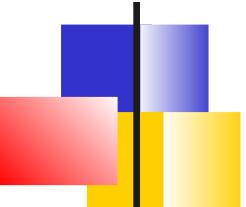
# Généralités

---

Toutes les pipelines déclaratives doivent être dans un bloc ***pipeline***.

Les instructions et expressions suivent la syntaxe Groovy avec les exceptions suivantes :

- Pas de point-virgule comme séparateur d'instructions.  
Chaque instruction est sur sa propre ligne
- Les blocs ne peuvent qu'être des *sections*, *directives*, *steps* ou des *assignments* .
- Une référence de propriété est traitée comme une invocation de méthode sans argument. Par exemple, *input* est traitée comme *input()*

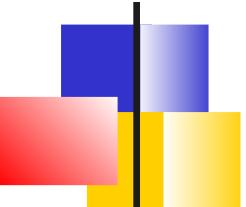


# Sections

---

Les sections contiennent 1 ou plusieurs directives ou steps

- **agent** : Placer globalement ou sur un stage, spécifie l'exécuteur à utiliser
- **stages** : Une séquence d'une ou plusieurs directives
- **steps** : Une ou plusieurs étapes à exécuter à l'intérieur d'une directive *stage*
- **post** : Directives et steps à exécuter à la suite du build selon son statut



# Stages et steps

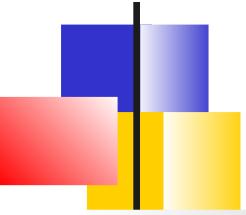
---

## ***stages*** :

- pas de paramètres spécifiques.
- Il est recommandé que la section *stages* contienne au minimum une directive *stage*

## ***steps*** :

- Pas de paramètre
- A l'intérieur de chaque *stage*

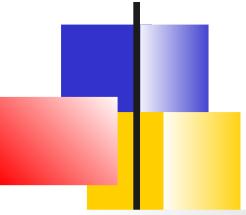


# *agent*

---

La section ***agent*** supporte les paramètres suivants :

- ***any*** : N'importe quel agent.
- ***none*** : Aucun.
  - Permet de s'assurer qu'aucun agent ne sera alloué inutilement.
  - Placer au niveau global, force à définir un agent au niveau de stage
- ***label*** : Agent ayant été labellisé par l'administrateur
- ***node*** : Idem que label mais avec plus d'options
- ***docker*** : Image docker



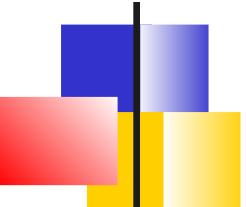
# tools

---

**tools** permet d'indiquer les outils à installer sur l'exécuteur ou agent.

La section est ignorée si *agent none*

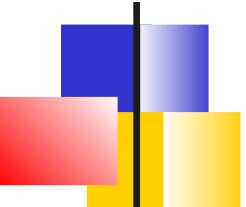
Les outils supportés sont : *maven, jdk, gradle*



# Exécuteur

---

```
// Directive,  
// agent construit à partir d'un Dockerfile  
agent {  
    dockerfile {  
        filename 'Dockerfile'  
    }  
}  
// Installation automatique d'un outil sur l'agent  
  
agent any  
tools {  
    maven 'Maven 3.5'  
}
```

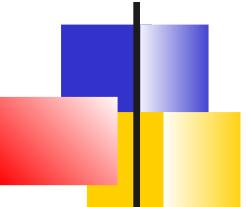


# *post*

---

La section ***post*** définit une ou plusieurs *steps* qui sont exécutées en fonction du statut du build

- *always* : Étapes toujours exécutées
- *changed* : Seulement si le statut est différent du run précédent
- *failure* : Seulement si le statut est échoué
- *success* : Seulement si statut est succès
- *unstable* : Seulement si statut *instable* (Tests en échec, Violations qualité, porte perf., ...)
- *aborted* : Build avorté



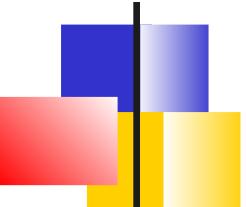
# Environnement

---

La directive ***environment*** spécifie une séquence de paires clé-valeur qui seront définies comme variables d'environnement pour le stage .

- La directive supporte la méthode ***credentials()*** utilisée pour accéder aux crédentiels définis dans Jenkins.

```
pipeline {  
    agent any  
  
    environment {  
        NEXUS_CREDENTIALS = credentials('jenkins_nexus')  
        NEXUS_USER = "${env.NEXUS_CREDENTIALS_USR}"  
        NEXUS_PASS = "${env.NEXUS_CREDENTIALS_PSW}"  
    }  
}
```



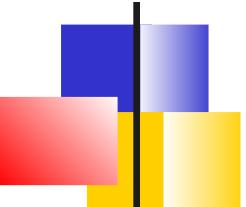
# Option

---

La directive ***options*** permet de configurer des options du plugin pipeline ou d'autres plugins. Par exemple, *timeout*, *retry*, *buildDiscarder*, ..

Exemple :

```
pipeline {  
    agent any  
    options { timeout(time: 1, unit: 'HOURS') }  
    stages {  
        stage('Example') {  
            steps { echo 'Hello World' }  
        }  
    }  
}
```



# *Input et parameters*

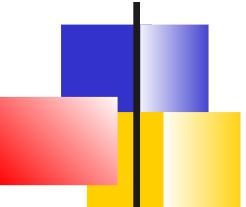
---

La directive ***input*** permet de stopper l'exécution d'une pipeline et d'attendre une approbation manuelle d'un utilisateur

Via la directive ***parameters***, elle peut définir une liste de paramètres à saisir.

Chaque paramètre est défini par :

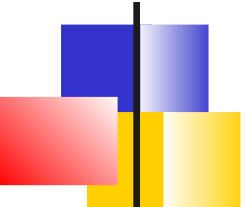
- Un type\_: String ou booléen, liste, ...
- Une valeur par défaut
- Un nom (Le nom de la variable disponible dans le script)
- Une description



# Exemple : input

---

```
// Input
input {
    message "Should we continue?"
    ok "Yes, we should."
    submitter "alice,bob"
    parameters {
        string(name: 'PERSON', defaultValue: 'Mr Jenkins', description:
'Who should I say hello to?')
    }
}
steps {
    echo "Hello, ${PERSON}, nice to meet you."
}
```

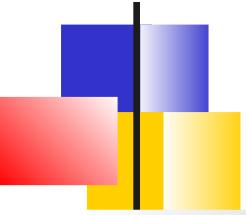


# *triggers*

---

La directive ***triggers*** définit les moyens automatique par lesquels la pipeline sera redéclenchée.

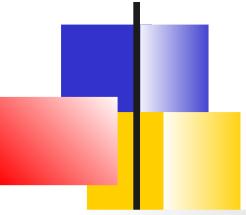
Les valeurs possibles sont *cron*, *pollSCM* et *upstream*



# *stage*

---

La directive ***stage*** se place dans la section stages et doit contenir une section steps, éventuellement une section optionnell ou d'autres directives spécifique à stage.



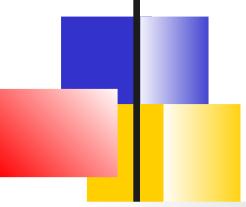
# *when*

---

La directive ***when*** permet à Pipeline de déterminer si le stage doit être exécutée

La directive doit contenir au moins une condition.

Si elle contient plusieurs conditions, toutes les conditions doivent être vraies.  
(Équivalent à une condition *allOf* imbriquée)



# Conditions imbriquées disponibles

---

**branch** : Exécution si la branche correspond au pattern fourni

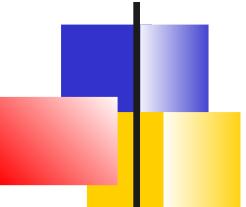
**environnement** : Si la variable d'environnement spécifié à la valeur voulue

**expression** : Si l'expression Groovy est vraie

**not** : Si l'expression est fausse

**allOf** : Toutes les conditions imbriquées sont vraies

**anyOf** : Si une des conditions imbriquées est vraie

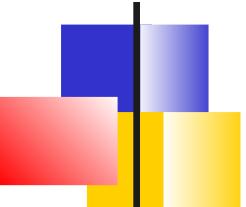


# Parallélisme

---

Les Stages peuvent déclarer des stages imbriqués qui seront alors exécutés en parallèle

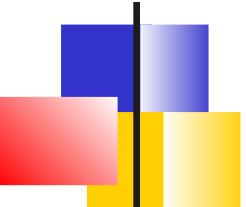
- Les stages imbriquées ne peuvent pas contenir à leur tour de stages imbriqués
- Le stage englobant ne peut pas définir *d'agent* ou de *tools*



# Exemple

---

```
// Declarative //
pipeline {
agent any
stage('Parallel Stage') {
when {branch 'master' }
parallel {
stage('Branch A') {
agent { label "for-branch-a" }
steps { echo "On Branch A" }
}
stage('Branch B') {
agent { label "for-branch-b" }
steps { echo "On Branch B" }
}
}
}
}
```



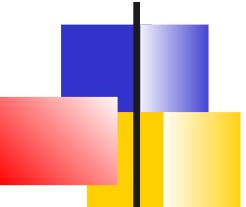
# Steps

---

Les steps disponibles sont extensibles en fonction des plugins installés.

Voir la documentation de référence à :  
<https://jenkins.io/doc/pipeline/steps/>

A noter que la version déclarative à une steps script qui peut inclure un bloc dans la syntaxe script



# Steps

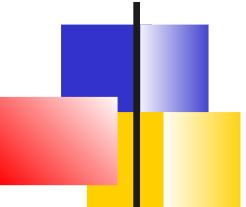
---

```
// Copier des artefacts
def built = build('downstream'); // https://plugins.jenkins.io/pipeline-
    build-step
copyArtifacts(projectName: 'downstream', selector: specific("$
    {built.number}"));

// Archive the build output artifacts.
archiveArtifacts artifacts: 'output/*.txt', excludes: 'output/*.md'

// Step basiques
stash, unstash
deleteDir, writeFile, readFile, pwd
fileExists
mail
```

Voir : <https://jenkins.io/doc/pipeline/steps/>



# Déploiement

---

Considérations

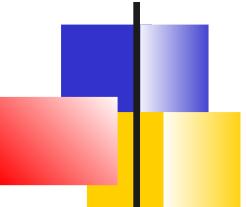
Virtualisation

Outils de gestion de configuration

Containerisation. Le cas docker

Orchestrateurs

Kubernetes

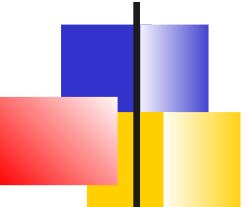


# Introduction

---

2 décisions importantes concernant l'infra :

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



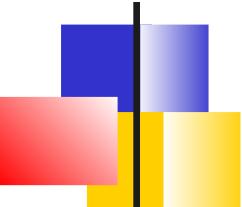
# Modèle classique

*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

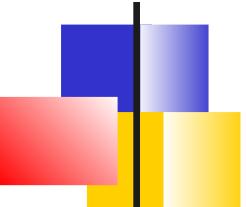
*Serveur immuable et Reverse Proxy*

---

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

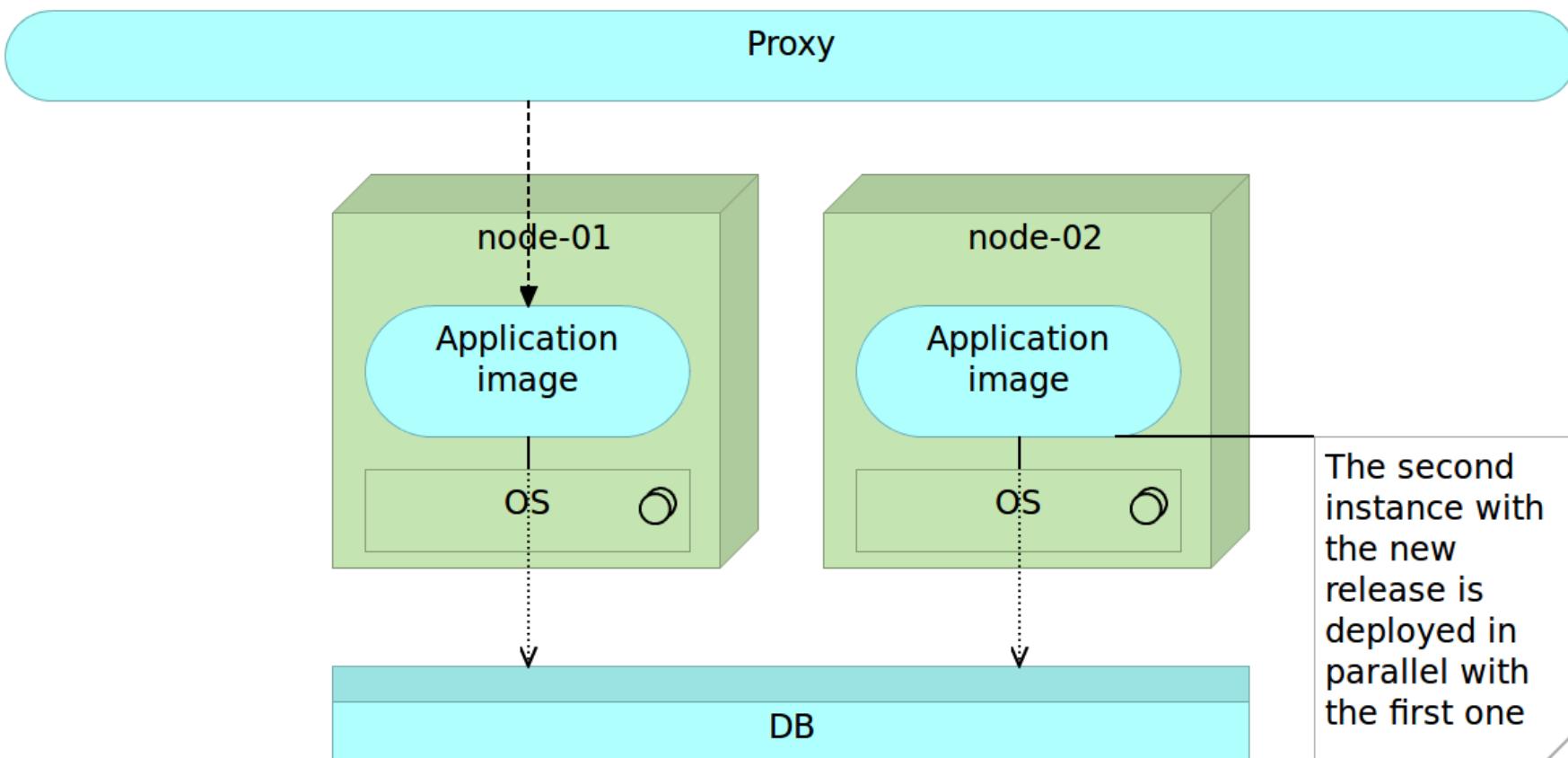
C'est ce tout qui est déployé

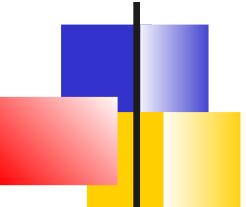


# Modèle DevOps

## *Serveur immuable et Reverse Proxy*

---





# Architecture micro-services

---

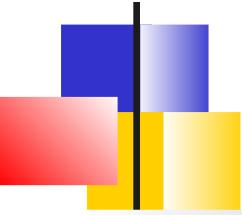
La durée du déploiement dépend principalement de :

- 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  
=> scalabilité dynamique, serverless



# Virtualisation



# La virtualisation dans le cycle de vie

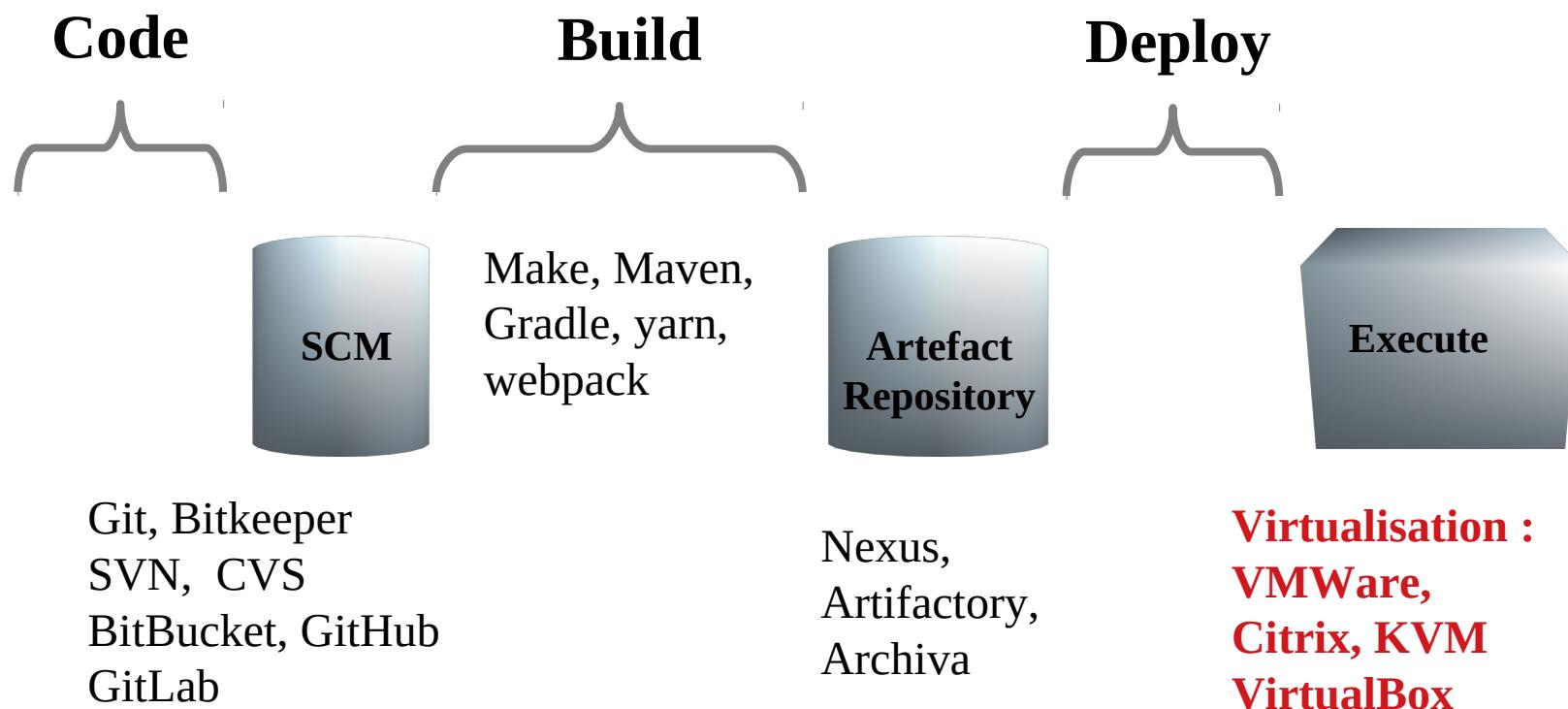
---

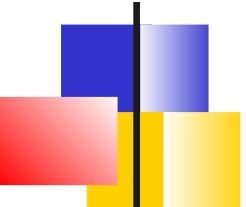
## Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

---

## Plateforme de livraison





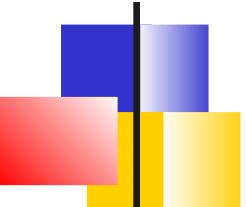
# Hyperviseur

---

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.



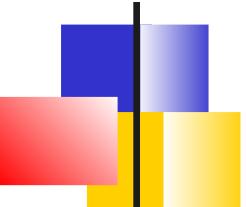
# Limitations

---

Pour virtualiser un environnement complet le serveur doit être capable

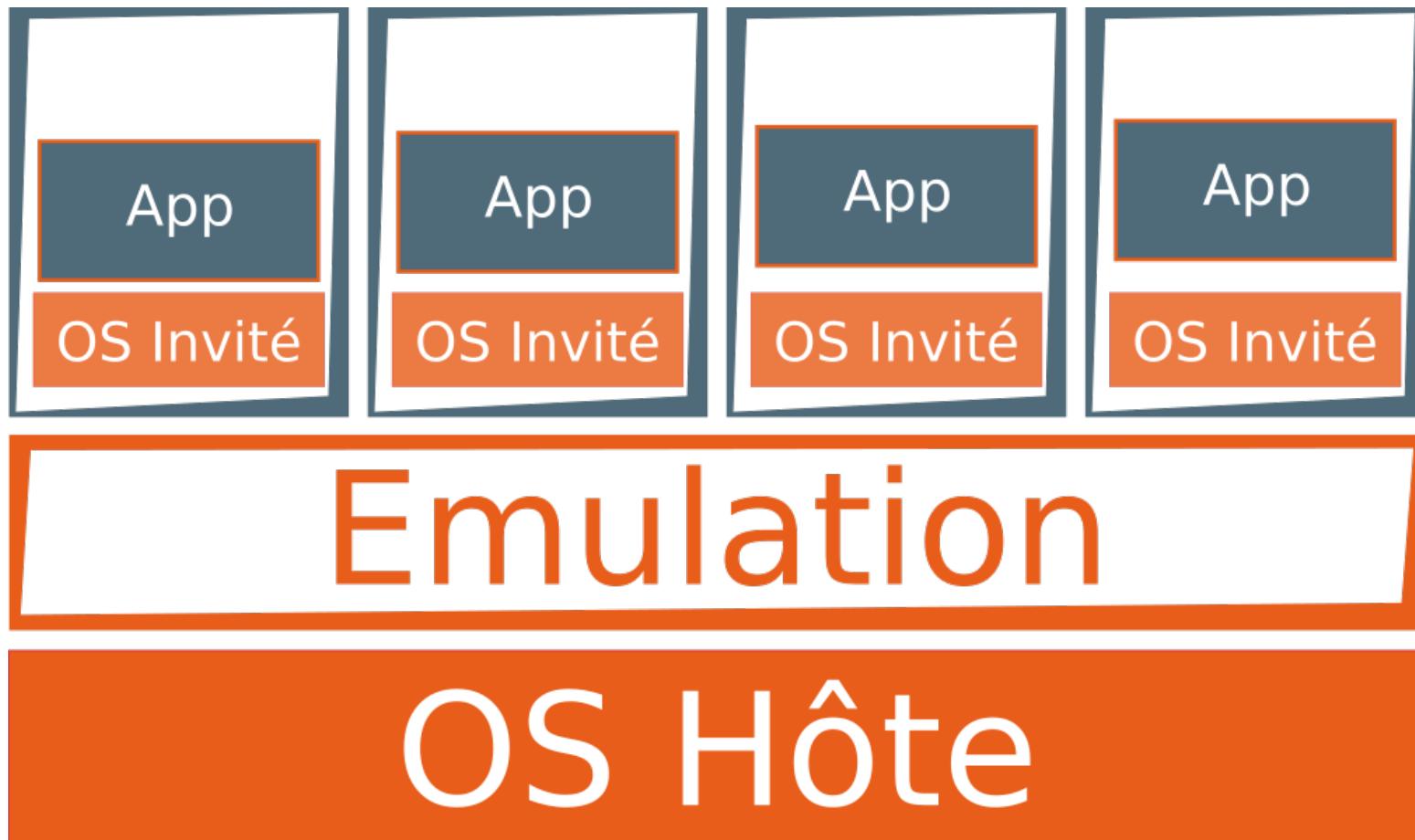
- de supporter la charge de son OS hôte ainsi que les OS invités
- + l'émulation du matériel (CPU, mémoire, carte vidéo...) et la couche réseau associée.

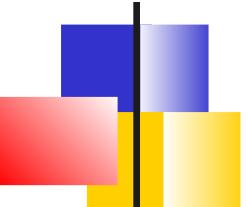
=> Les ressources nécessaires pour exécuter une Machine Virtuelle sont importantes.



# Hôte / invité

---





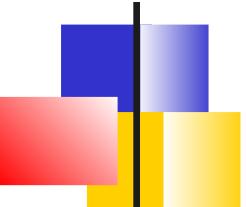
# Solutions

---

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



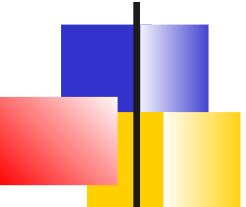
# Vagrant

---

**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



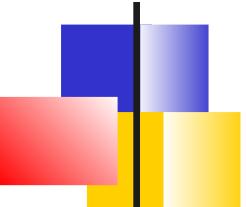
# Projet Vagrant

---

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

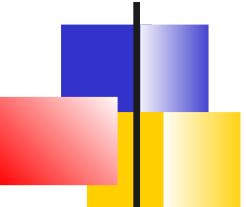


# Configuration de la machine virtuelle

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

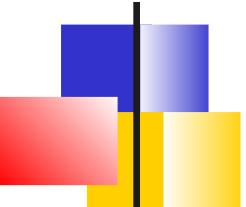
=> Il est possible d'y positionner des scripts permettant de provisionner la box

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
```

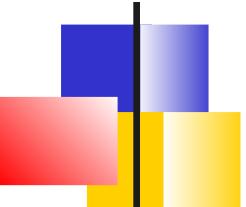


# Principales commandes

---

Le client vagrant permet alors de nombreuses commandes :

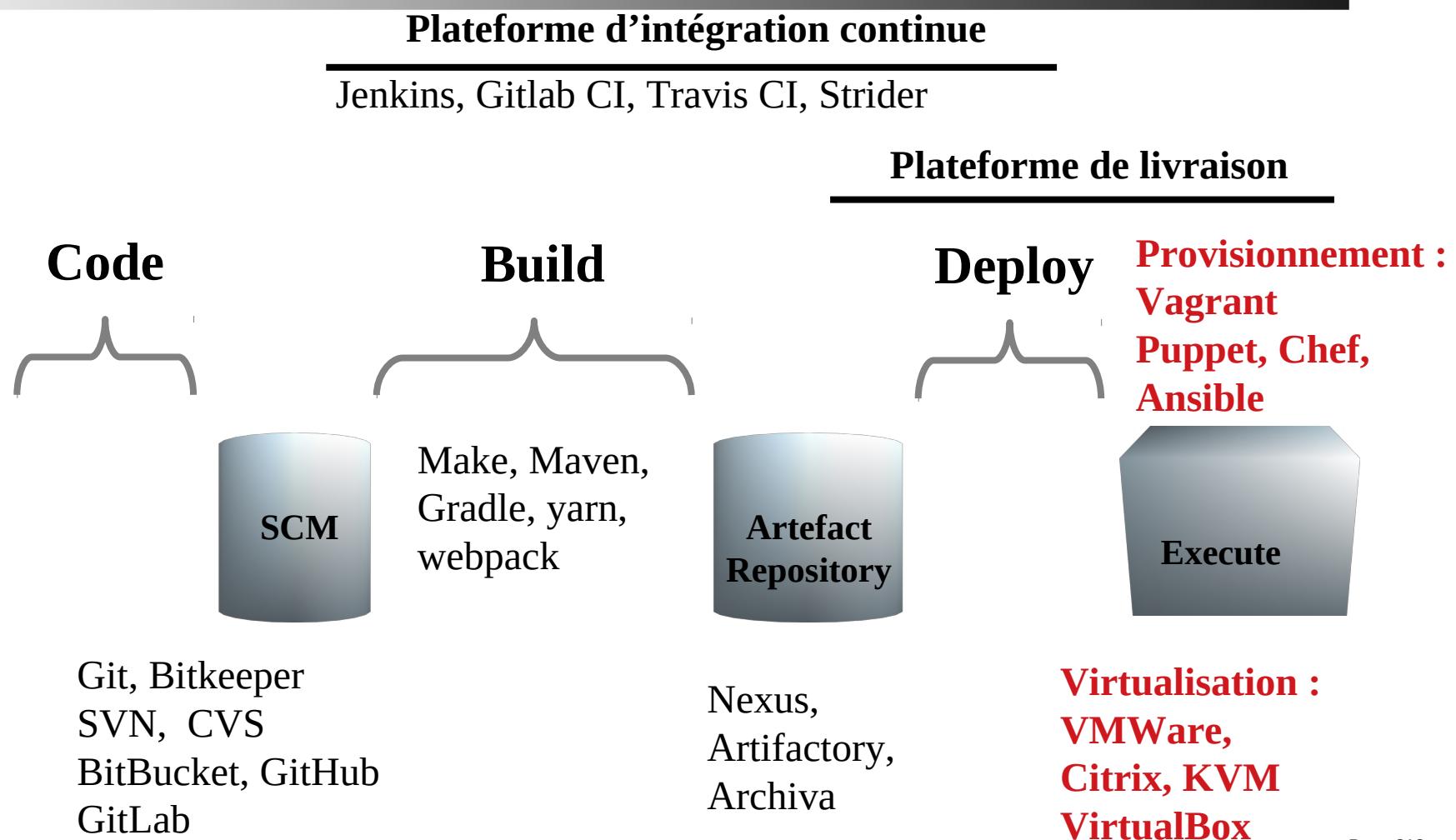
- **up, halt, suspend, resume, destroy** : Démarrage, arrêt, pause, reprise, suppression de toutes les traces
- **provision** : Met à jour les logiciels sur une machine s'exécutant
- **ssh, powershell, rdp** : Accès distant sur la machine
- **box** (add,list, remove) : Gestion des vagrants box
- **snapshot**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

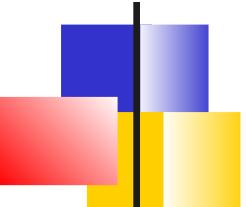


---

# Outils de gestion de configuration

# Provisionnement dans le cycle de vie





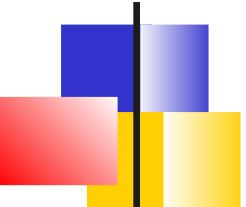
# *Infrastructure As Code*

---

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



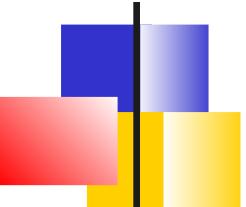
# Gestion de déploiements

---

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

2 approches existent :

- L'approche centralisé nécessite 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*



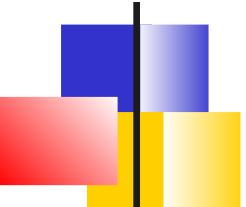
# *Ansible*

---

**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

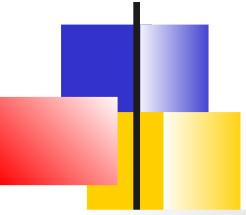
---

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.

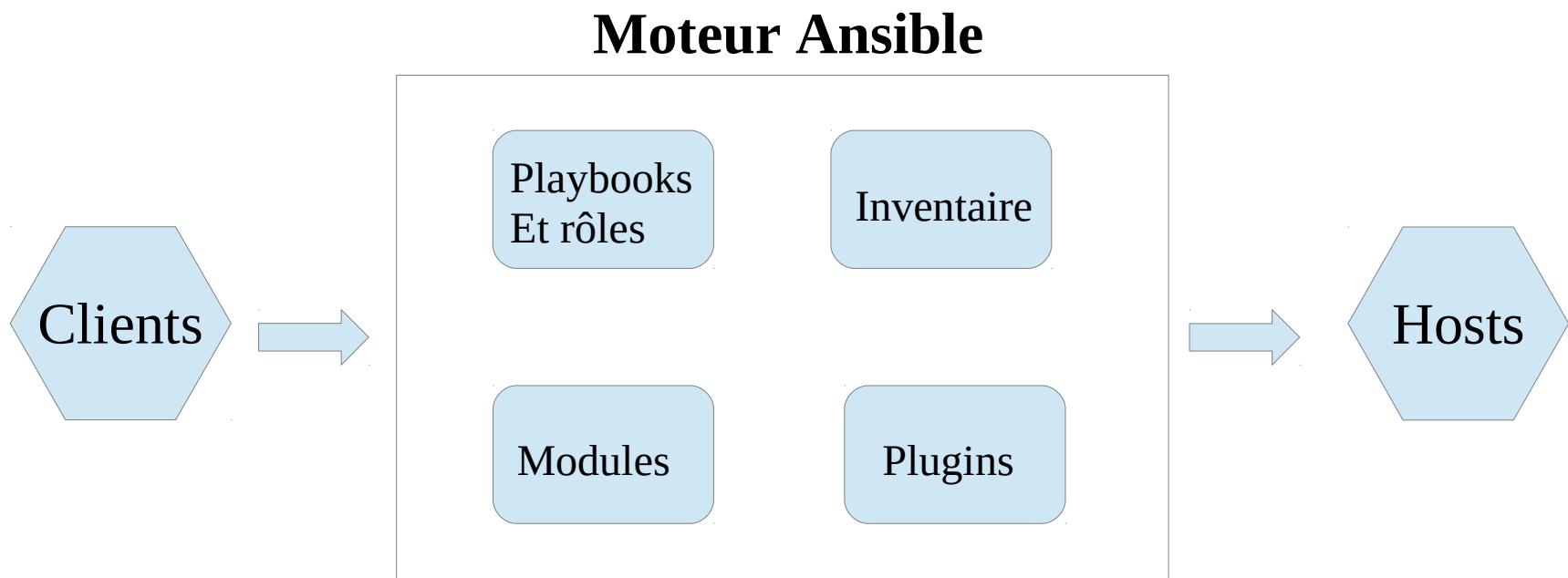
Ex :

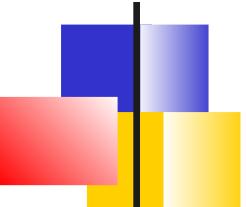
```
ansible all -s -m apt -a 'pkg=nginx  
state=installed update_cache=true'
```



# Architecture

---





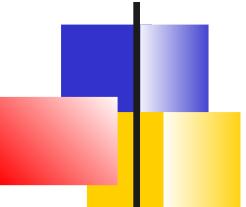
# Inventaire

---

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

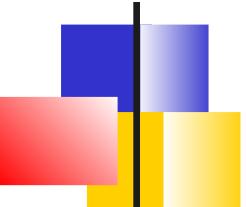
bar.example.com

[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

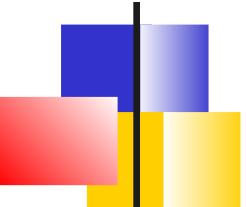
**ansible-doc** : Affiche la documentation

**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

---

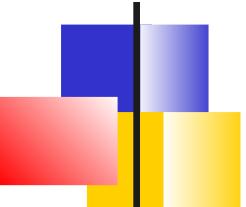
**ansible** : Exécute une tâche sur un ensemble de host

```
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 disponibles :  
*ansible-config*, *ansible-console*, *ansible-pull*, ...



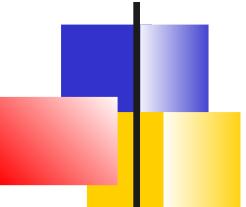
# Playbooks

---

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

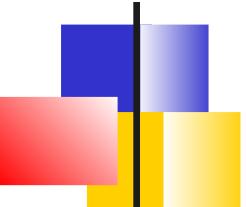


# Configuration d'un déploiement

---

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

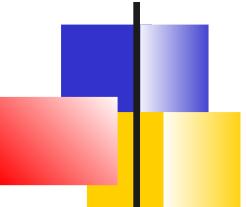
---

```
---
- 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
```



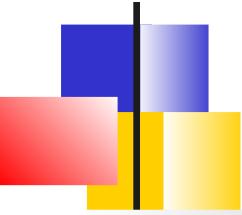
# Rôles

---

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*

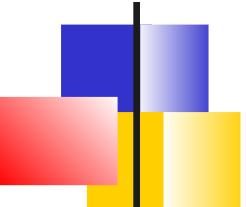


# Arborescence classique

---

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
```



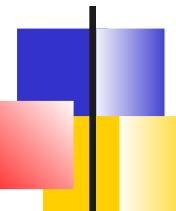
# Terraform

---

Infrastructure as code pour provisionner et gérer :

- Le cloud
- L'infrastructure
- Les services

Les fichiers terraform sont committés, versionnés dans le dépôt



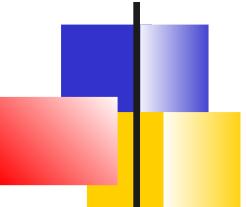
# Plan d'exécution incrémentiel

---

Les fichiers de configuration décrivent les composants nécessaires pour exécuter une seule application ou l'ensemble d'un data center.

Terraform génère un plan d'exécution décrivant ce qu'il va faire pour atteindre l'état souhaité, puis l'exécute pour créer l'infrastructure décrite.

Au fur et à mesure que la configuration change, Terraform est capable de déterminer ce qui a changé et de créer des plans d'exécution incrémentiels qui peuvent être appliqués.

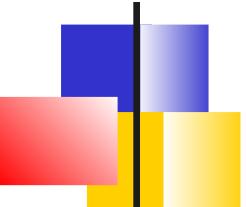


# Périmètre

---

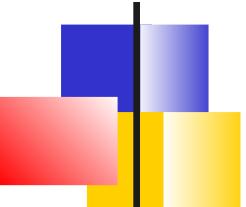
L'infrastructure que Terraform peut gérer comprend des composants de bas niveau tels que les instances de VM, le stockage, le réseau, ainsi que des composants de haut niveau tels que les entrées DNS, les services.

<https://learn.hashicorp.com/tutorials/terraform/kubernetes-provider?in=terraform/kubernetes>



---

# Containerisation : Le cas Docker



# Introduction

---

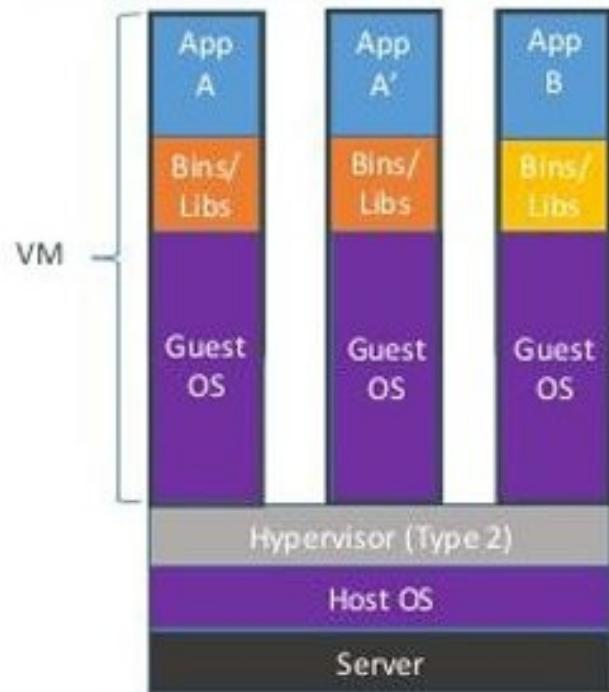
Plutôt que de virtualiser une machine complète, juste créer l'environnement d'exécution minimal pour fournir l'application, le service.

- 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.

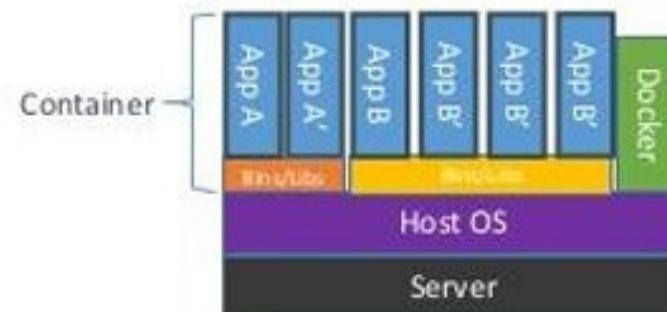
L'apport d'une solution de containerisation est l'isolation d'un processus dans un système de fichiers à part entière

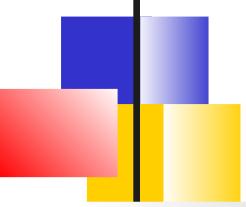
# Containers vs VMs

## Containers vs. VMs



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





# Avantages de la containerisation

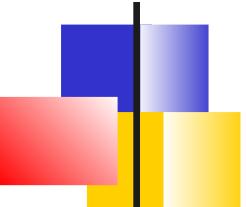
---

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

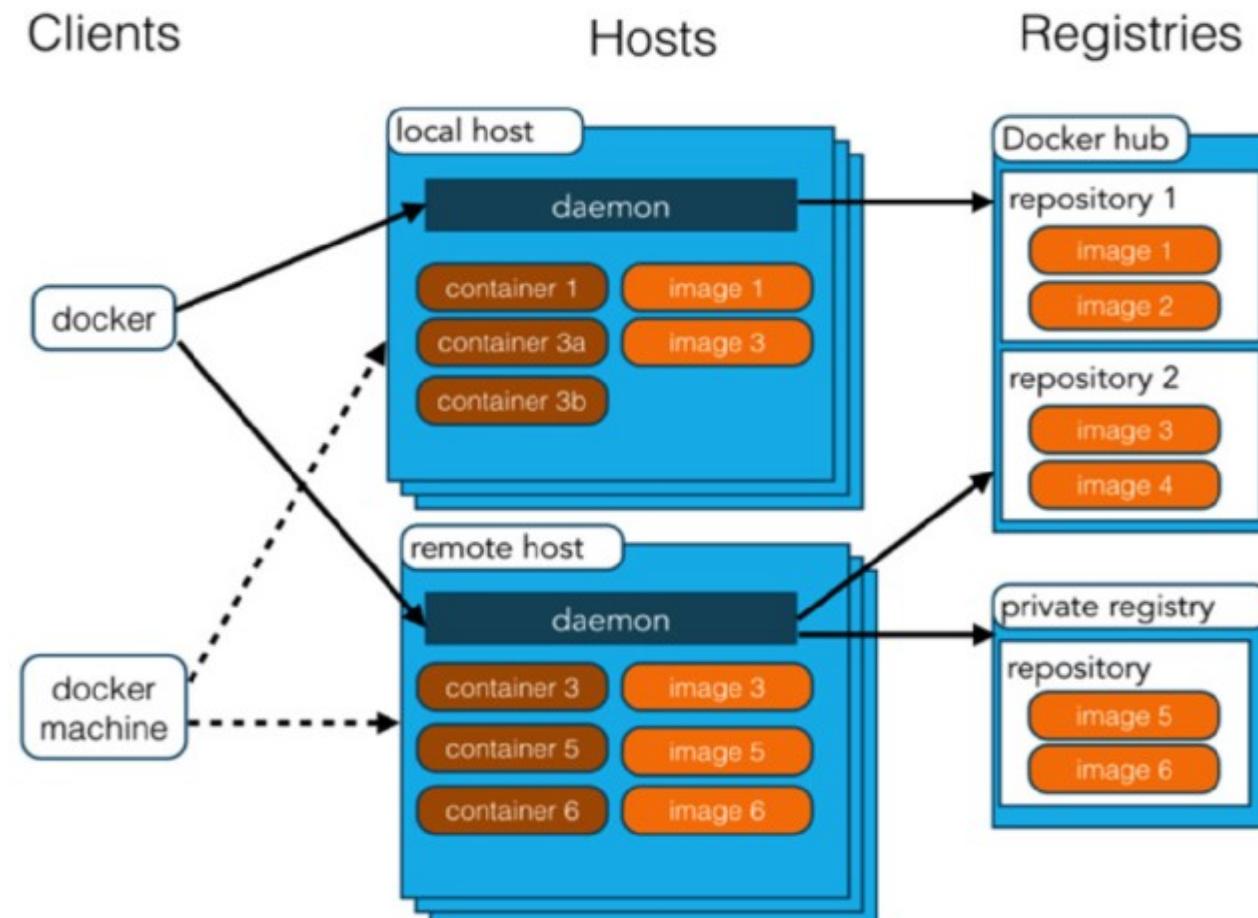
---

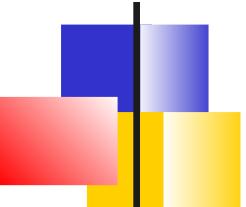
Les développeurs et la PIC travaillent alors avec la même image de conteneur que celle utilisée en production.

- => 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

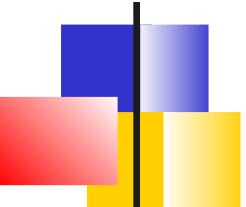




# Commandes Docker

---

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



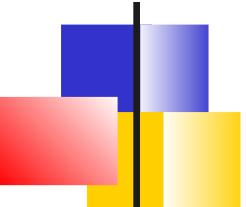
# Exemple DockerFile

---

**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"]
```



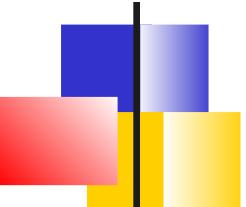
# Isolation du conteneur

---

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

Par défaut, il est isolé

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



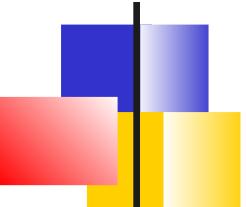
# Communication avec la machine hôte

---

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
```

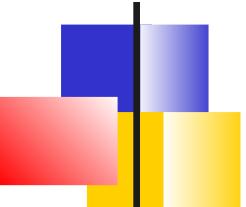


# *docker-compose*

---

***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.



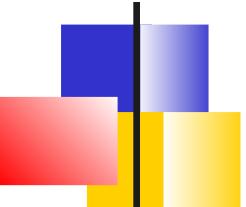
# Exemple configuration

```
# 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

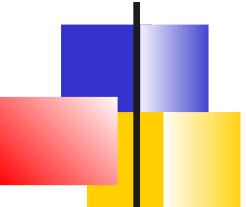
**pull / push** : Pull/push les images des services

**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

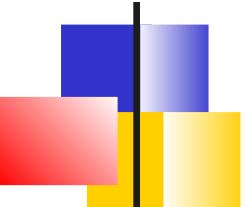


# PIC et Docker

---

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.

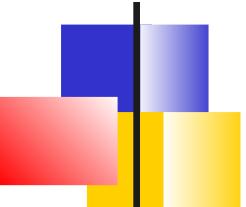


# Techniques d'intégration

---

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

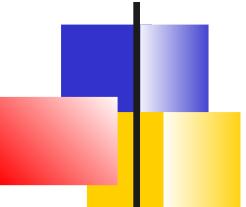
---

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 :

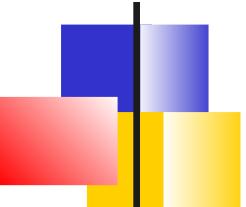
```
docker build -t my-image dockerfiles/  
docker run my-image /script/to/run/tests  
docker tag my-image my-registry:5000/my-image  
docker push my-registry:5000/my-image
```



# Exemple Jenkins : Agent docker

---

```
// Declarative //
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v $HOME/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            Steps { sh 'mvn -B' }
        }
    }
}
// Script //
node {
    docker.image('maven:3-alpine').inside('-v $HOME/.m2:/root/.m2') {
        stage('Build') { sh 'mvn -B' }
    }
}
```



# Exemple Jenkins : Construction d'image

---

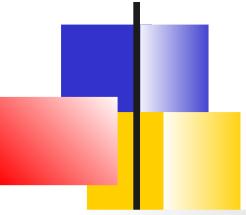
```
#!/usr/bin/env groovy

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'
        }
    }
}
```



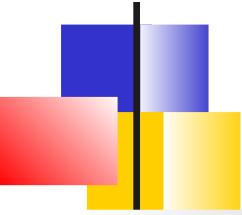
# Orchestrateur de conteneurs

---

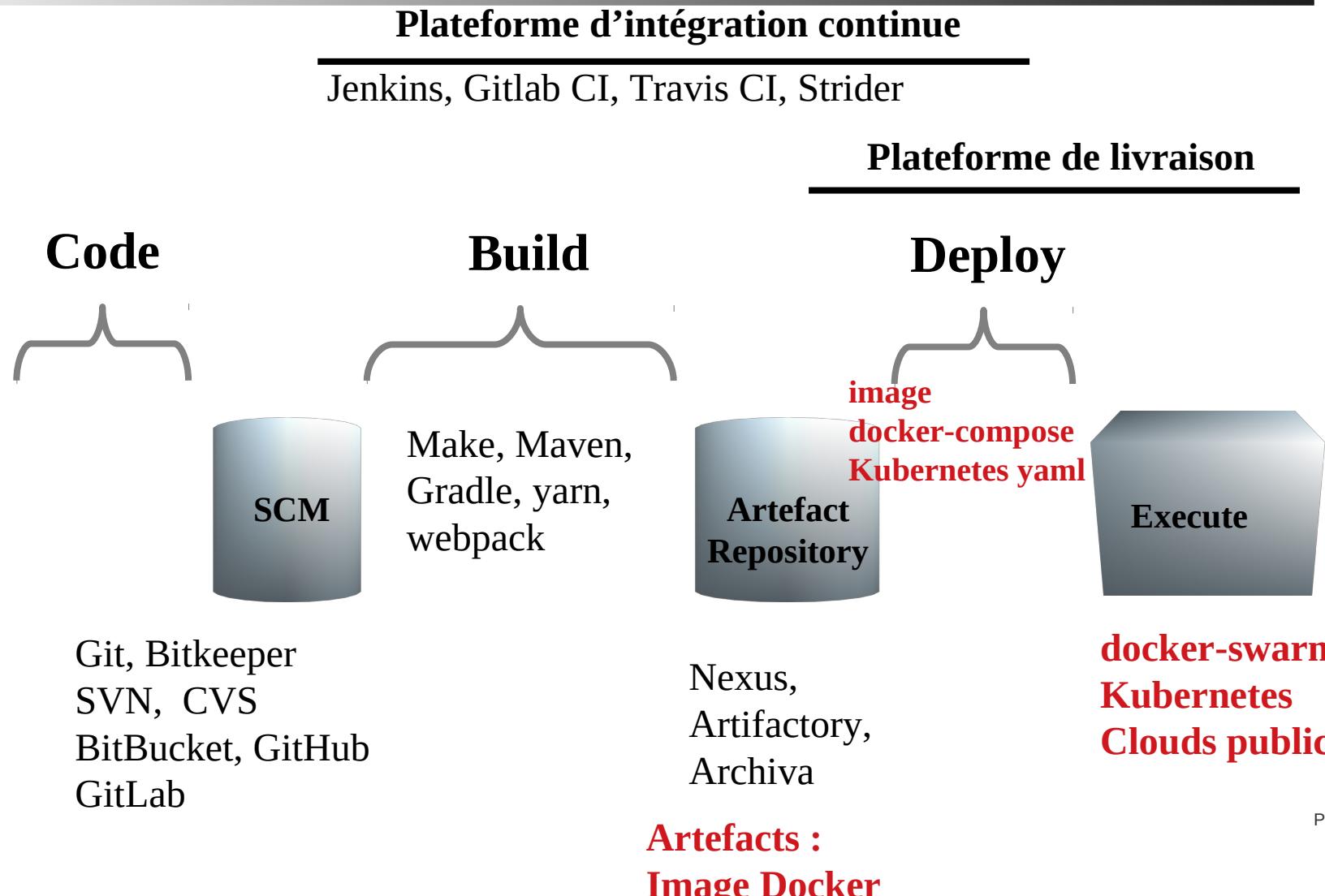
Orchestrateur, scalabilité et  
déploiement

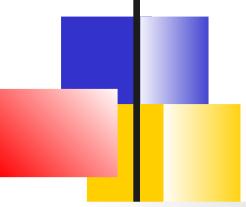
Kubernetes

Intégration dans une pipeline CI/CD



# Cloud dans le cycle de vie





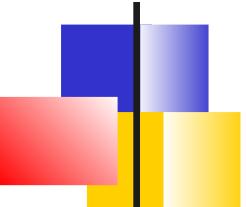
# Principes de l'orchestration de conteneurs

---

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 :

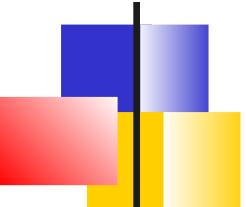
- Redémarrage si plantage
- Scaling si charge importante



# 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

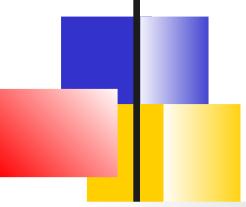


# Apport de l'orchestrateur : Déploiement blue-green

---

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 considérée 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

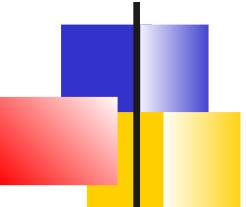
---

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

---

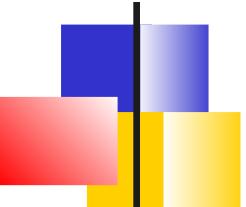
OpenSource : Proviens des projets Google's *Borg* et *Omega*

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)



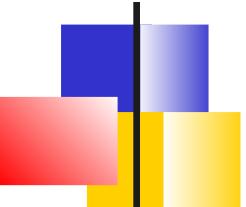
Kubernetes (Mai 2020) :  
91000 commits  
+2000 contributeurs  
+60k\* sur GitHub  
Géré par la Cloud Native Computing Foundation (Groupe neutre)



# 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

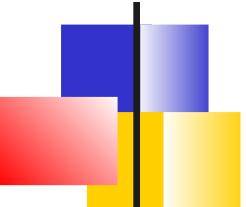


# Auto-correctif

---

*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. »



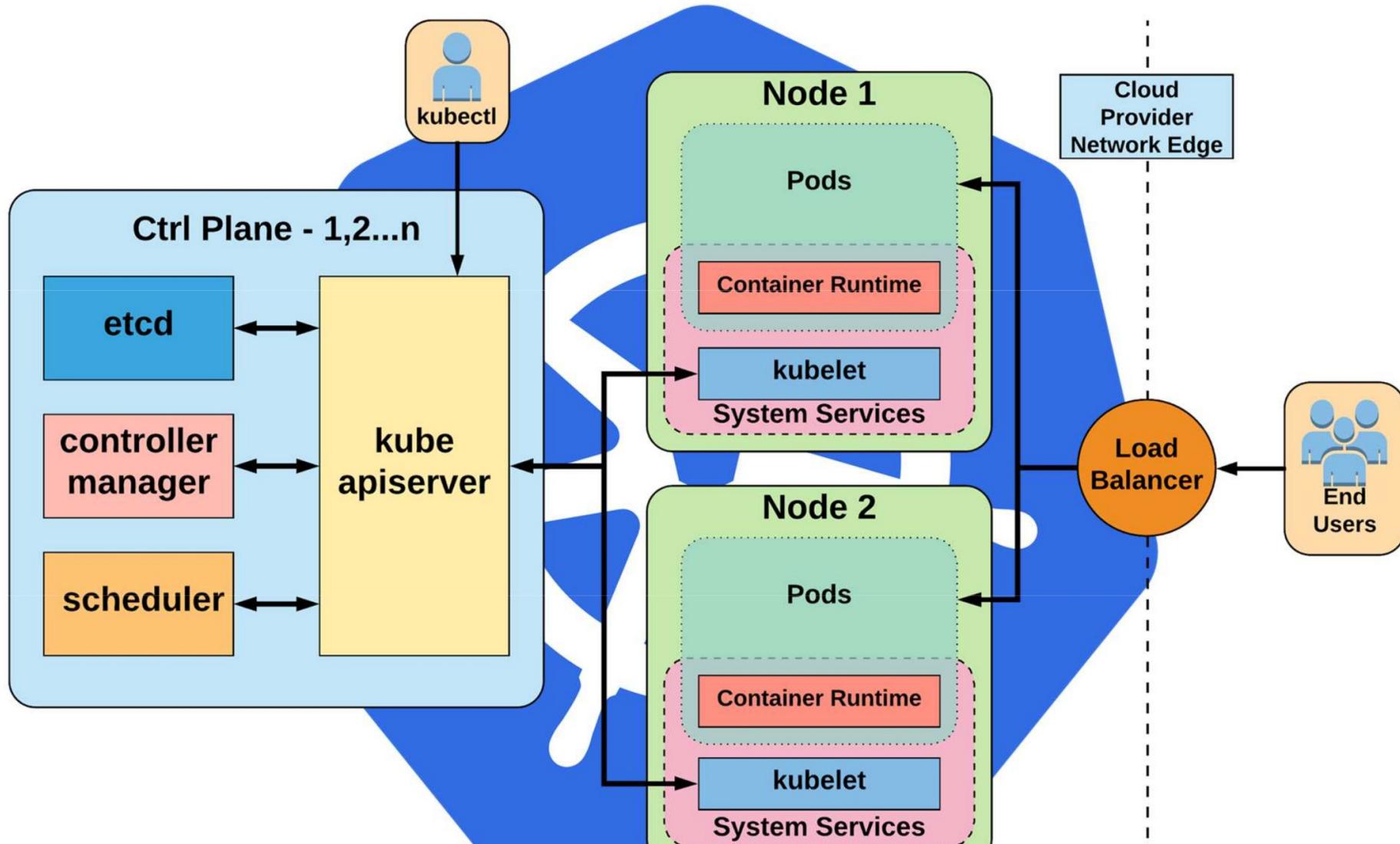
# Fonctionnalités applicatives

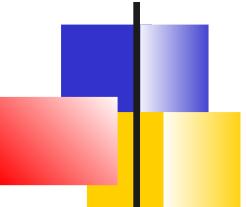
---

- 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*)

*pod = 1 ou plusieurs conteneurs co-localisés*

# Architecture cluster

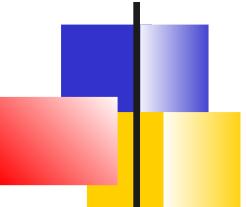




# Règles réseau à l'intérieur du cluster

---

- 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:**

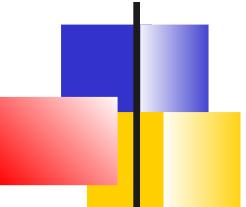
`/apis/<group>/<version>/<resource>`

**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



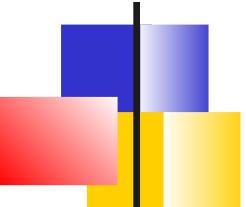
# Principes

---

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

En particulier, 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 points d'accès stables à un service applicatif

# pod

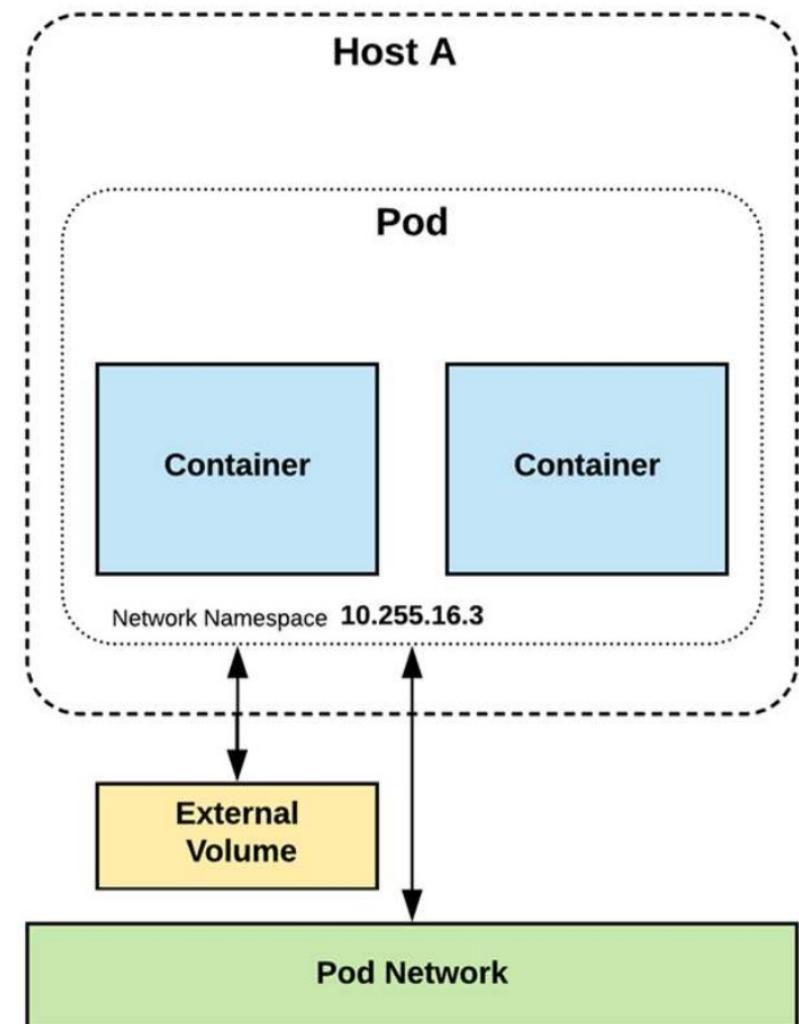
Un **pod** est la plus petite unité de travail

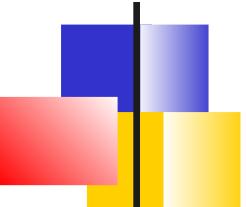
Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- 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

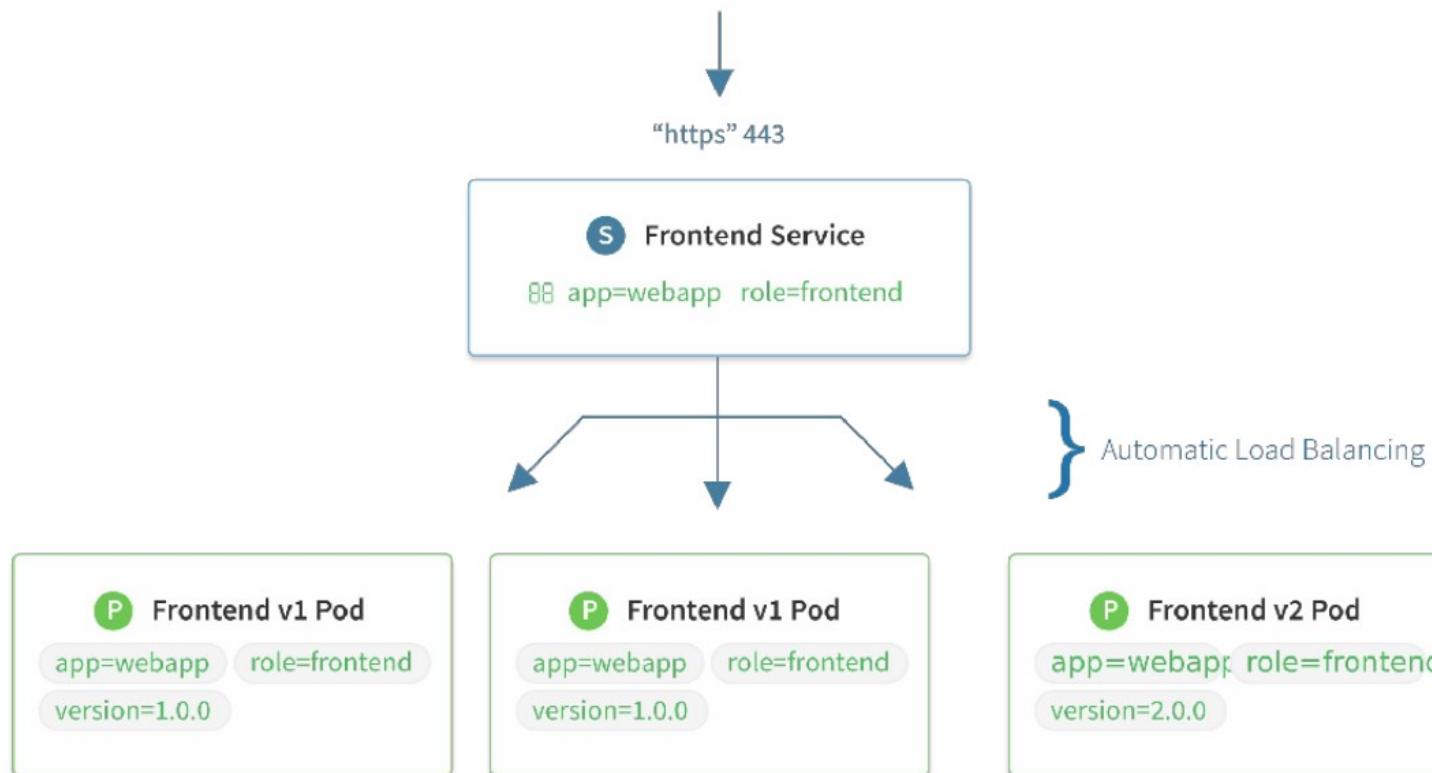
---

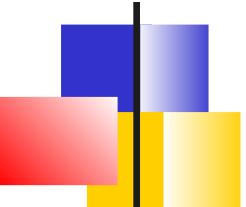
Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *pods*.

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





# Ressource *deployment*

---

Exemple description d'un déploiement:

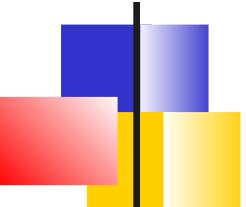
```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  spec:
    containers:
      - image: dthibau/annuaire
        name: annuaire
```

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

***kubectl create -f ./my-manifest.yaml***

**Ou**

***kubectl apply -f ./my-manifest.yaml***

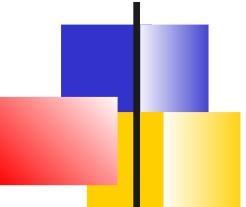


# Exemple service

---

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

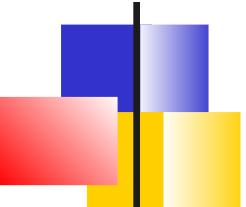
**exec** : Exécuter une commande dans un container

**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

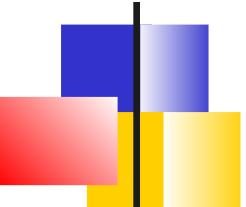
# Supprime les pods et services ayant le noms "baz"
kubectl delete pod,service baz

# 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

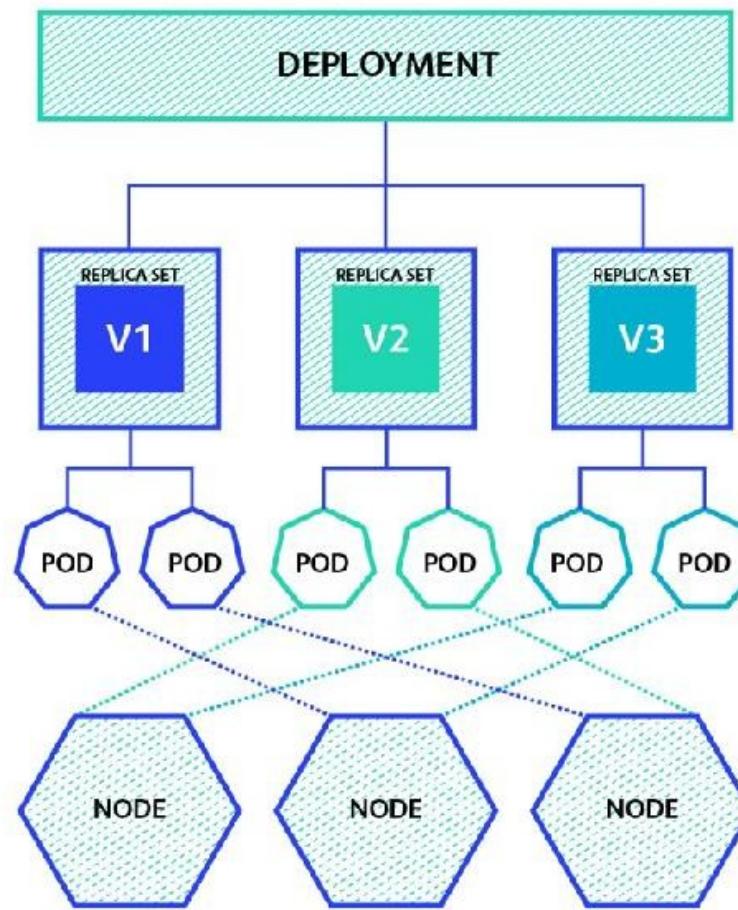
---

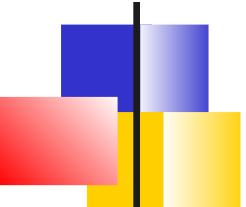
La ressource **deployment** permet de manipuler un ensemble de *Replicaset* (*ensemble de conteneurs répliqués*)

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 vieilles 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

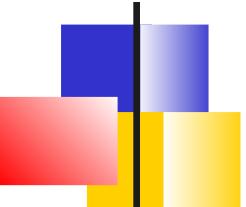
```
kubectl rollout history deployment/nginx-deployment
```

# Roll-back sur la version précédente

```
kubectl rollout undo deployment/nginx-deployment
```

# Scaling

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



# Scheduler et Workload

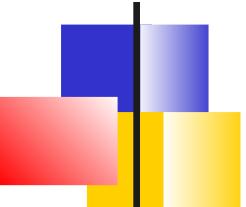
---

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

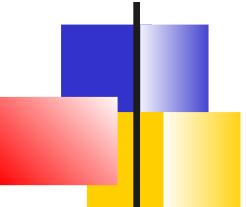
---

## Example Object

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

## Example Status Snippet

```
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

---

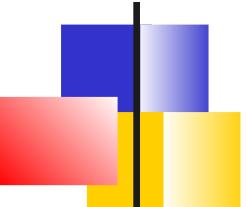
**ClusterRole** : Rôle avec permissions sur l'API

**VolumePersistant** : Système de stockage

**PersistentVolumeClaims** : Demande d'usage d'un volume persistant

**ConfigMaps** : Stockage clé-valeur pour la configuration

**Secrets** : Stockage de crédentiels

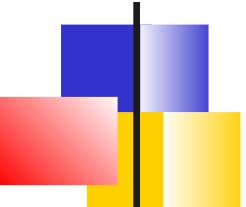


# Écosystème *Kubernetes*

---

De nombreux outils peuvent être ajoutés à une installation cœur 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)
- **Prometheus** : 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

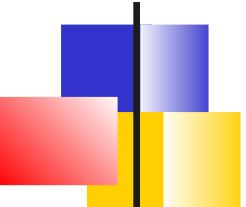
- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Services
- Google Kubernetes Engine
- Digital Ocean
- ...

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

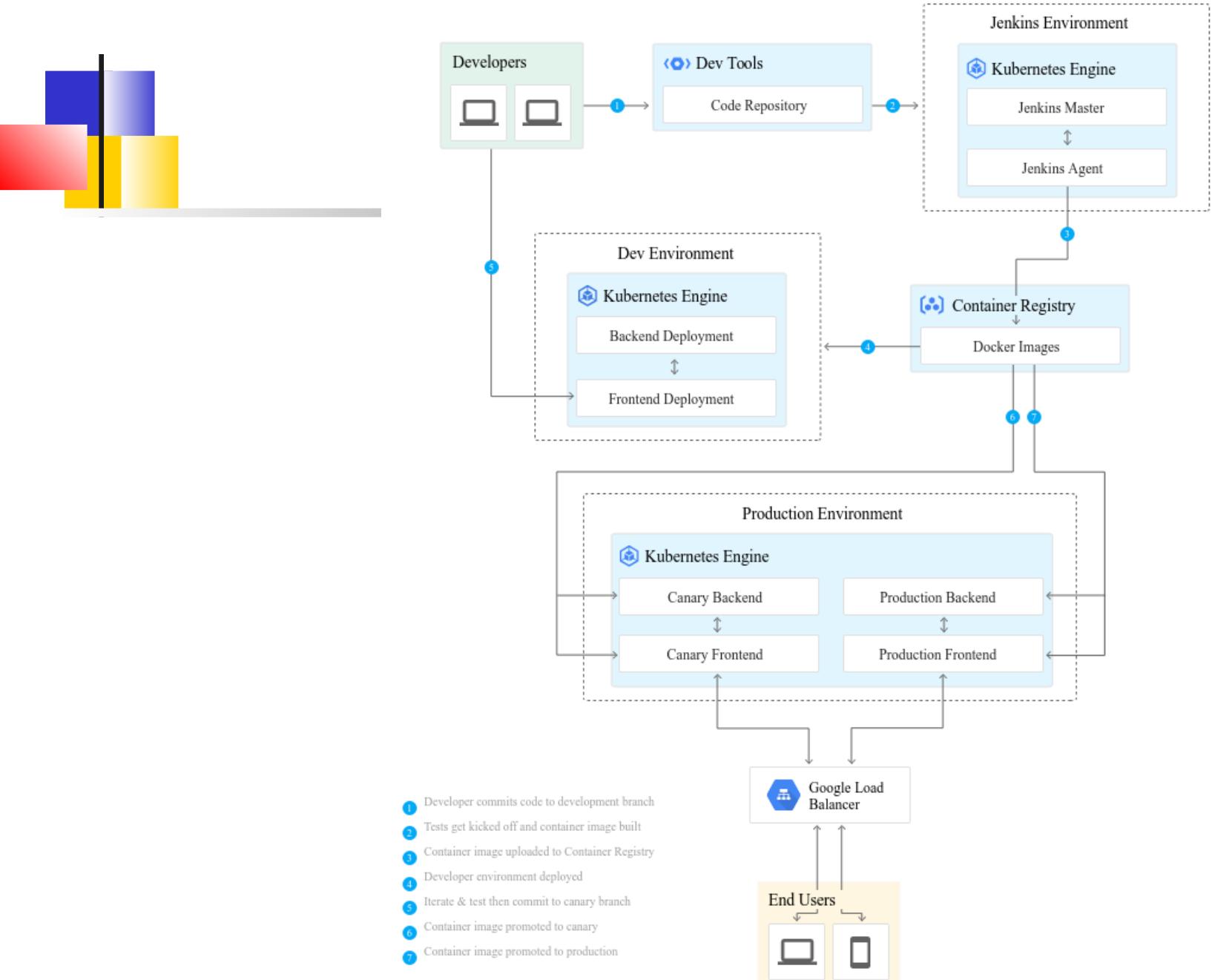
Des versions en ligne comme : <https://labs.play-with-k8s.com/>

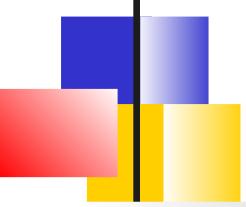
L'outil *Rancher* permet de gérer graphiquement plusieurs installation

*Terraform* permet de provisionner des cluster (et services) as Code



# Kubernetes dans la pipeline CI/CD





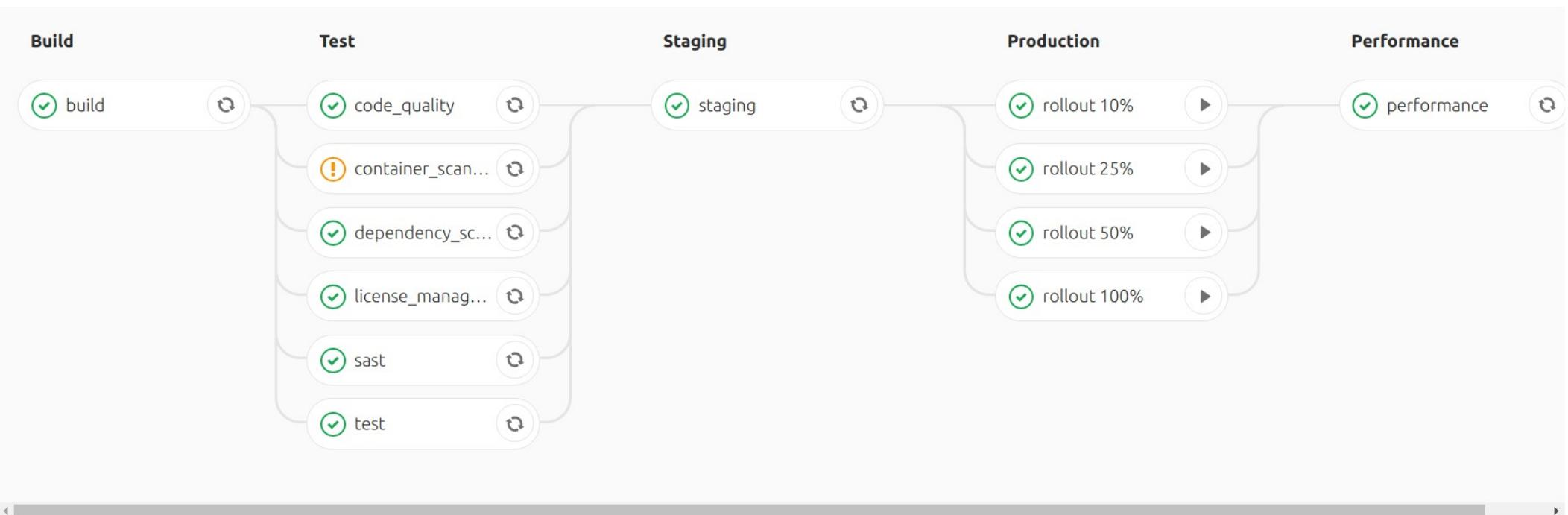
# Exemple AutoDevOps GitlabCI

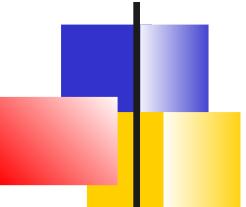
---

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