

# Intégration continue avec Maven, Nexus, Jenkins, Sonar

---

David THIBAU – 2024

david.thibau@gmail.com



# Agenda

---

## **Introduction**

- DevOps, CI/CD, Tests, Build, Outils associés

## **Outils de SCM**

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

## **Maven**

- Concepts, Dépendance, Adaptation, Profils, Multi-modules

## **Release, Nexus**

- Dépôts d'artefact, Nexus, Processus de release

## **Qualité avec SonarQube**

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

## **Le serveur Jenkins**

- Configuration, Architecture, Jobs Maven, intégration Sonar/Nexus

## **Pipelines CI/CD**

- Pipeline Legacy, Plugin pipeline Syntaxe déclarative,



# Introduction

---

## **Objectifs et pratiques DevOps**

Pipelines CI/CD

Tests et analyse statique

Provisionnement d'infra-structure



# 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



# Contraintes sur la fréquence des 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



# Approche *DevOps*

---

*DevOps* vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers" .

Cela impose :

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

Plus génériquement, *DevOps* signifie le regroupement de toutes les compétences nécessaires à un projet au sein d'une équipe complètement **indépendante**



# Pratiques *DevOps* (1)

---

- Un déploiement régulier, i.e. continu, des applications dans les différents environnements.  
La seule répétition contribuant à fiabiliser le processus
- Un décalage des tests "vers la gauche".  
Tester au plus tôt
- Une pratique des tests dans un environnement similaire à celui de production  
Dev/Prod parity
- Une intégration continue incluant des "tests continus" (à chaque commit);  
Effort sur les tests



# Pratiques *DevOps* (2)

---

- Une boucle d'amélioration courte  
i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation  
Factualisée par des métriques et indicateurs "clé"  
disponible à tout moment
- Une unique source de vérité :  
Les configurations des différents environnements, des  
builds, des tests sont centralisées dans le même SCM que  
le code source
- Automatisation complète.  
Le build, les tests, le provisionnement, le déploiement sont  
automatisés et déclenchés à chaque commit





# Objectif ultime

---

- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime des services
- 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

---

Objectifs et pratiques DevOps

**Pipelines CI/CD**

Tests et analyse statique

Provisionnement d'infra-structure



# En continu

---

**A chaque ajout de valeur** dans le dépôt de source (*push*), l'intégralité des tâches nécessaires à la mise en service d'un logiciel (intégration, tests, déploiement) sont essayées.

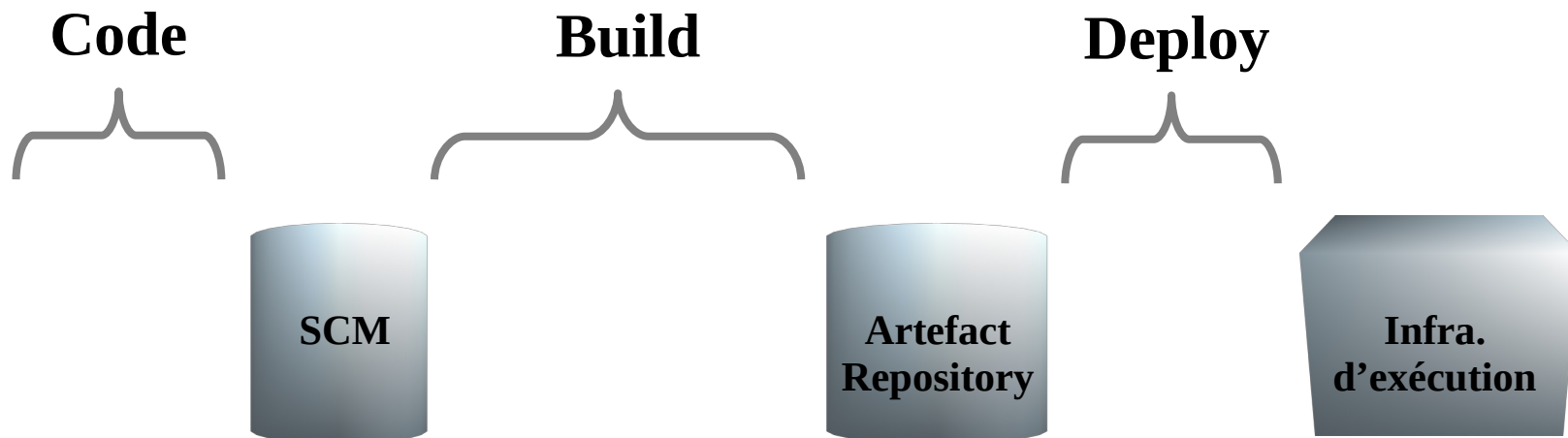
En fonction de leurs succès, l'application est déployée dans les différents environnements (intégration, staging, production)



# Cycle de vie du code Build / Release / Run

La pipeline suit le cycle de vie du code.

- 1) Le code est testé localement puis poussé dans le **dépôt de source**
- 2) Le build construit l'artefact et si le build est concluant stocke une release dans un **dépôt d'artefact**
- 3) L'outil de déploiement accède aux différentes release et les déploie sur l'**infra d'exécution**





# Build is tests !

---

La construction de l'application consiste principalement à :

- Packager le code source dans un format exécutable, déployable
- Effectuer toutes les vérifications automatiques permettant d'avoir confiance dans l'artefact généré et autoriser son déploiement  
(Tests Unitaires/Intégration, Fonctionnel, Performance, Sécurité, Licenses, ...)



# Release et Livraison

---

Si les tests sont concluants, on conserve l'artefact généré dans un dépôts d'artefacts que l'on peut déployer dans les différents environnements.

Les formats peuvent être varié en fonction des stacks technologiques : jar, war, ear, exe, Image docker, etc ..

Si l'artefact a atteint un niveau qualité de production, on crée une **release**

La release consiste généralement à :

- Tagger le dépôt des sources (SCM)
- Stocker l'artefact correspondant dans le dépôt d'artefact et le tagger avec le même tag



# Déploiement

---

Le déploiement s'effectue avec des outils dédiés à l'infrastructure (os, ansible, kubectl, helm, terraform, ...)

Certains outils permettent également de provisionner automatiquement l'infrastructure nécessaire



# Pipelines

Les étapes de construction automatisées sont séquencées dans une **pipeline**.

- Une étape est exécutée seulement si les étapes précédentes ont réussi.
- Les plate-formes d'intégration/déploiement continu ont pour rôle de démarrer et observer l'exécution de ces pipelines





# Distinction CI/CD

---





# Outil de communication

---

La Plateforme a pour vocation de publier les résultats des builds :

- Nombre de tests exécutés, succès/échecs
- Couverture des tests
- Complexité, Vulnérabilités du code source
- Performance : Temps de réponse/débit
- Documentation, Release Notes
- ...

Les métriques sont visibles de tous en toute transparence

- => Confiance dans ce qui a été produit
- => Motivation pour s'améliorer



# Phases de la mise en place

---

La mise en place de pipeline de CI/CD passe généralement par plusieurs phases :

1. Mise à disposition d'une PIC
2. Automatisation tests unitaires et d'intégration
3. Déploiement dans un environnement d'intégration (review app)
4. Mise en place d'analyse de code, de tests fonctionnels, performance, d'acceptation automatisés, Collecte des métriques
5. Renforcement des tests, Release automatique, déploiement en QA
6. Renforcement des tests d'acceptation, Validation de déploiements
7. Totale confiance dans les tests permet le Déploiement continu



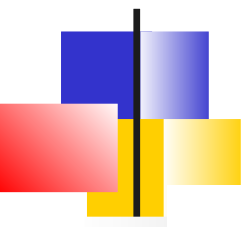
# Introduction

---

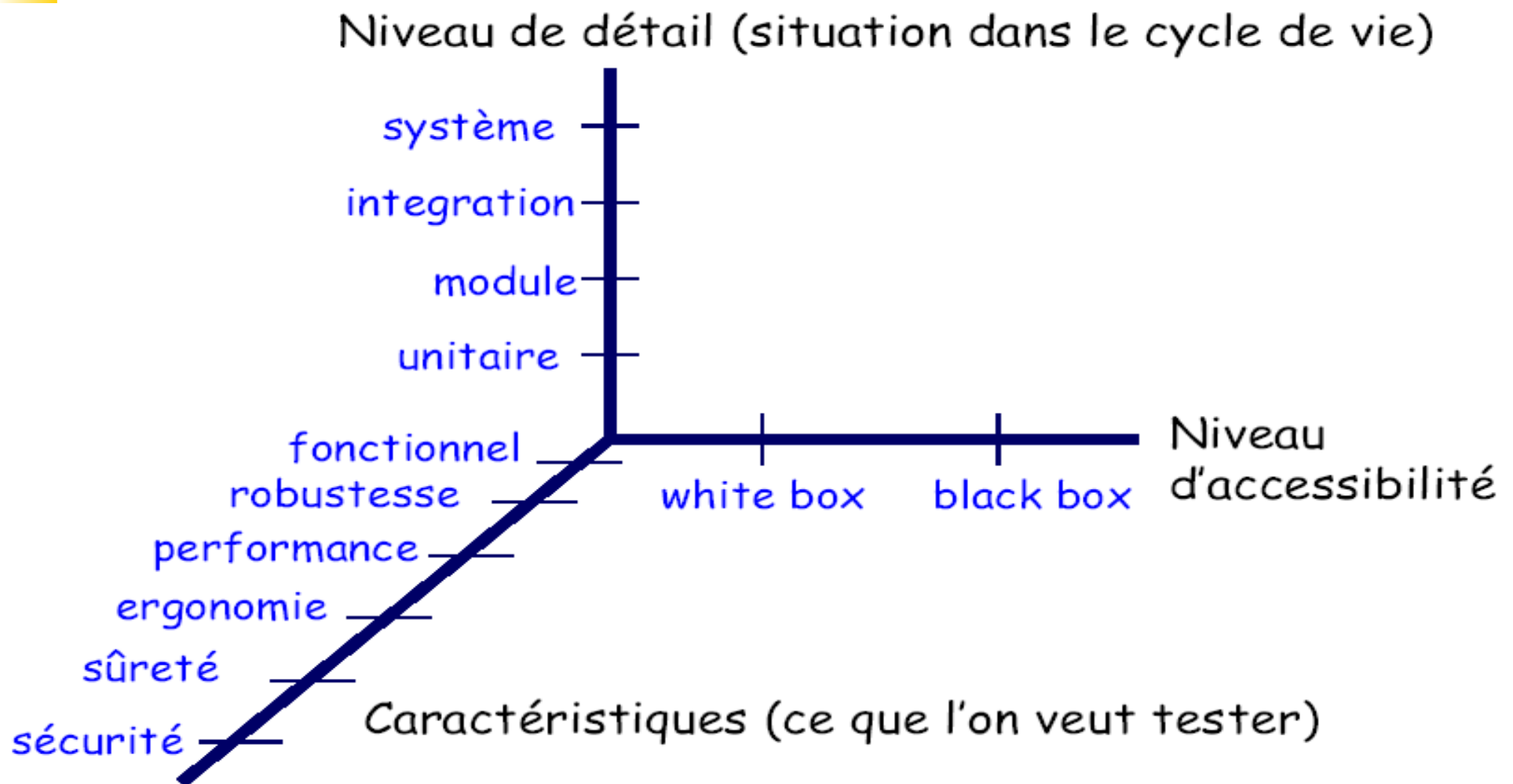
Objectifs et pratiques DevOps  
Pipelines CI/CD

**Tests et analyse statique**

Provisionnement d'infra-structure



# 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 ?*



# Le framework : JUnit

---

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

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

- Au départ, uniquement accès sur les tests unitaires...
- Mais utilisé comme moteur de démarrage de tout type de test : (intégration, fonctionnel, acceptation)



# Isolation et Mock objects

---

Lors du test d'un sous-ensemble du système (unitaire ou intégration), la partie à tester doit être isolée de ses dépendances

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

=> Utilisation de Mock Objects simulant les interfaces





# Exemple *MockBean*

---

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



# Test d'intégration

---

Les **tests d'intégration** font participer plusieurs composants du système.

Par exemple, la couche contrôleur avec la couche DAO

- Ils sont généralement plus lourds et plus lents que les tests unitaires car ils nécessitent :
  - Des middleware supplémentaires (Serveur applicatifs, Base de donnée, ...)
  - Une préparation des données de test. Initialisation de la BD par exemple

Cela reste des tests white box écrits par les développeurs

Des techniques d'accélération sont en général appliqués :  
Usage de serveurs embarqués, parallélisation des tests, etc..



# Tests fonctionnels

---

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

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



# Exemple : Selenium Web Driver

---

```
public class MonTest {
    @Test
    public void testGoogle() {
        // 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 est 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()); // Verifier le titre de la page
        // Google fait la recherche dynamique avec JavaScript.
        // Attendre le chargement de la page de 10 secondes
        // Verifiez 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 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 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)
- Outils OpenSource : *Apache JMeter, Gatling*



# Tests d'acceptation

---

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

- Ils sont mis au point avec le client, le testeur et les développeurs (*Les 3 Amigos*)
- Dans les méthodes agiles, ils complètent et valident une « User Story »
- Avec l'approche BDD (*Behaviour Driven Development*), l'expression des tests peut être faite en langage naturel.



# Tests d'acceptance

## Syntaxe Gherkin

#language : fr

**Fonctionnalité:** Abandon lors de la saisie de carte de crédit invalides

Dans les tests utilisateurs, nous avons vu beaucoup de gens qui ont fait des erreurs en entrant leur n° de carte de crédit. Nous devons être aussi utiles que possible ici pour éviter de perdre des utilisateurs à ce stade crucial de la transaction.

**Contexte:**

Étant donné que j'ai choisi certains articles à acheter

Et que je suis sur le point d'entrer les détails de ma carte de crédit

**Scénario:** numéro de carte de crédit trop court

Lorsque j'entre un numéro de carte de 15 chiffres seulement

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message m'informant du nombre correct de chiffres

**Scénario:** la date d'expiration ne doit pas être antérieure

Lorsque j'entre une date d'expiration de carte qui est dans le passé

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message me disant que la date d'expiration doit être fausse



# Classification pipeline DevOps

---

Dans le cadre d'une pipeline DevOps, la position des tests dans la pipeline se détermine en fonction de :

- Durée d'exécution :
- Le test nécessite t il un provisionnement d'infrastructure ? (Tests black-box)

Typiquement :

- Tests unitaires en premier
- Tests d'intégration,
- Test fonctionnel, d'acceptation de performance sur les infrastructures d'intégration, recette, pré-production et de production





# 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 détection de vulnérabilités
- Les vérifications de l'usage des licenses
- La production de métriques adossés à la norme ISO25010 relative à la qualité



# Métriques internes et Outils Qualité

---

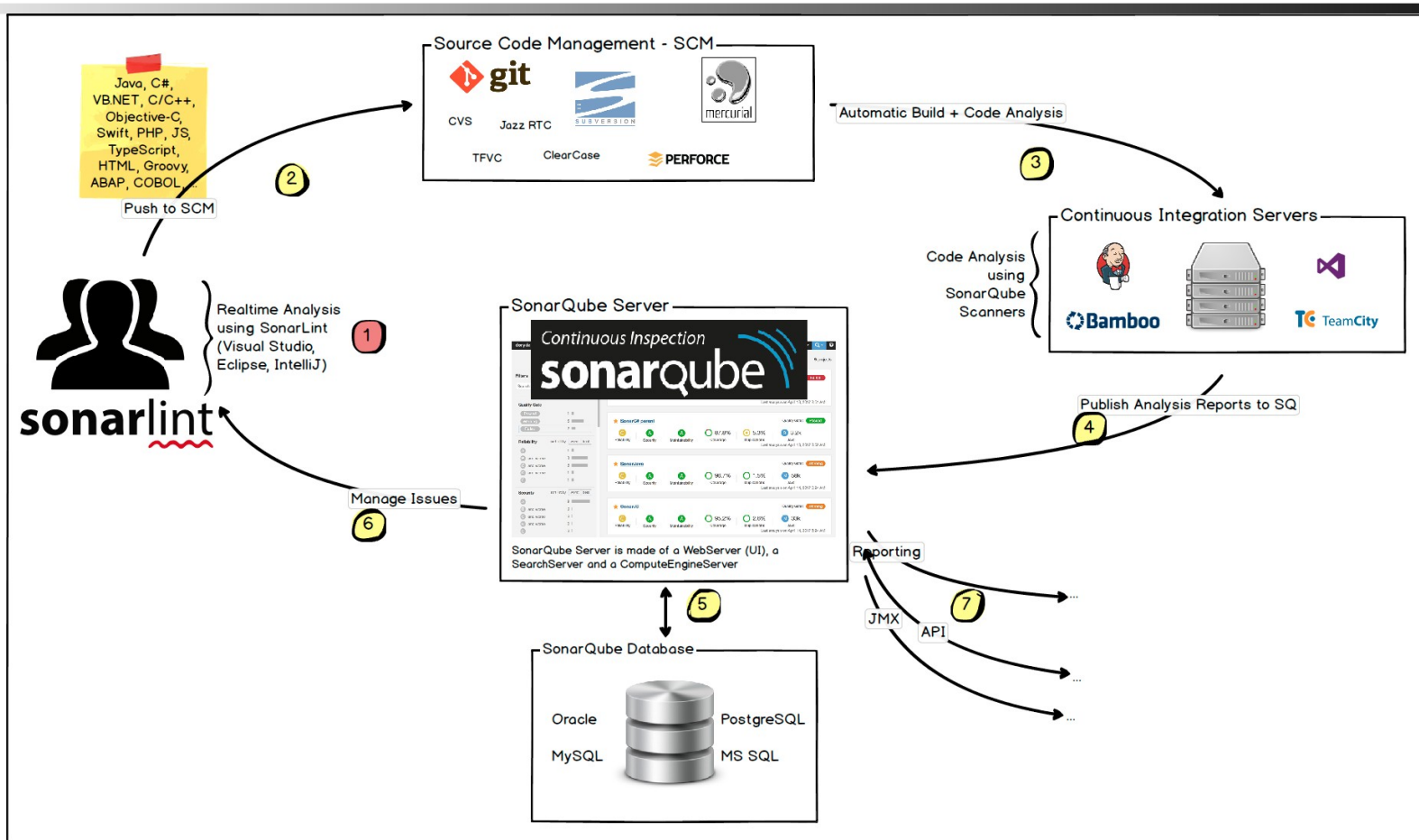
**SonarQube** propose une plate-forme qui regroupe tous les outils de calcul de métriques interne d'un logiciel (toute technologie confondue)

Il intègre 2 aspects :

- Détection des transgressions de règles de codage et estimation de l'influence sur la fiabilité, la sécurité et la maintenabilité de l'application
- Calculs des métriques internes

Pour chaque projet, il est possible de définir des **portes qualités** : seuils minimal des métriques afin que le projet soit livrable

# Analyse continue





# Introduction

---

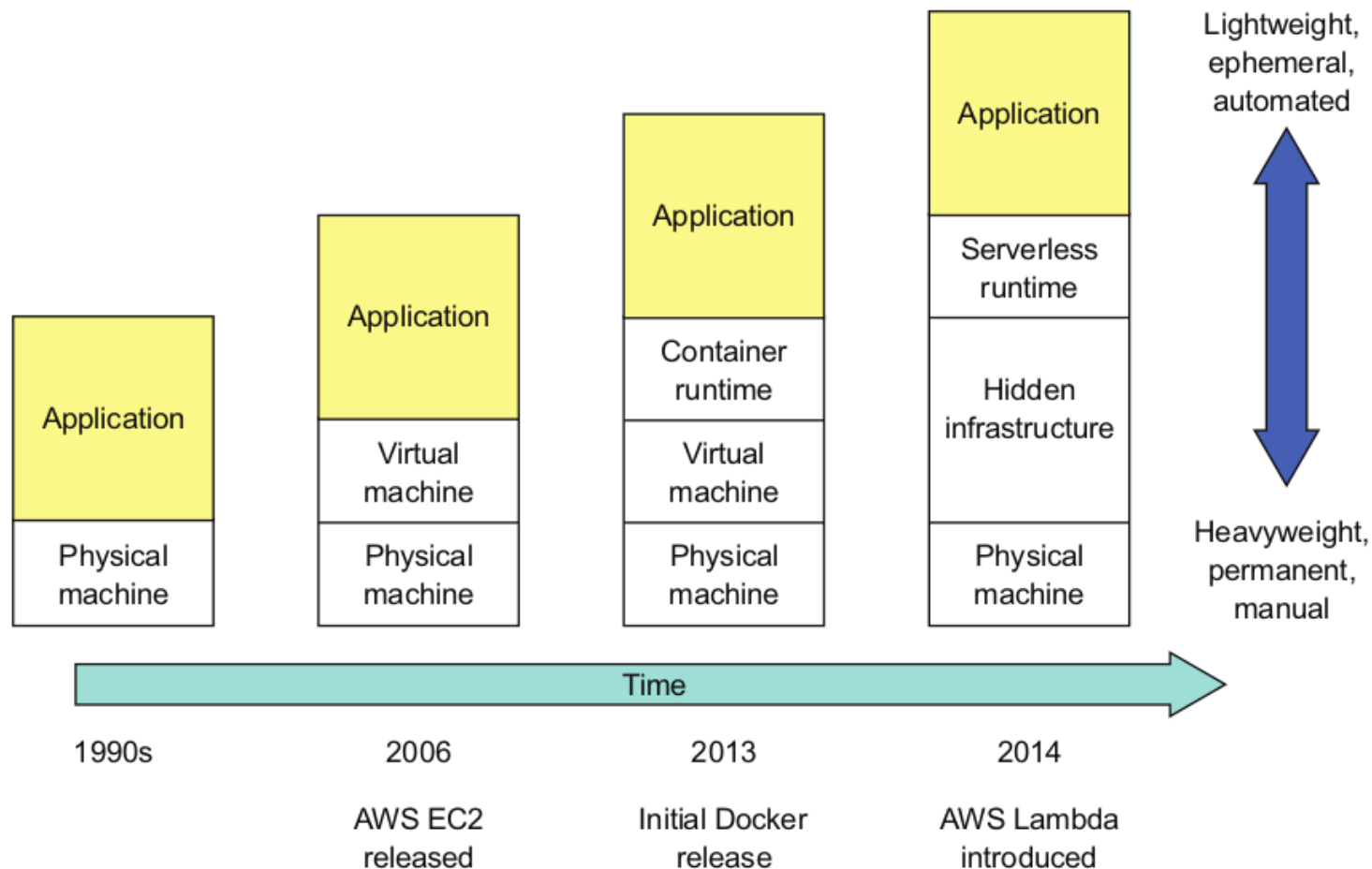
Constat et objectifs DevOps

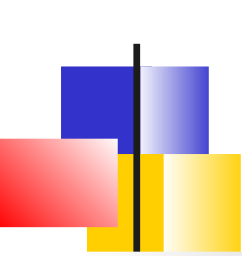
Pipelines CI/CD

Tests et analyse statique

**Provisionnement d'infra-structure**

# Evolution des infrastructures





# Modèle classique de déploiement

## ***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 sûr que les environnements de dev, de QA ou même les instances en production soient identiques

=> Difficulté de revenir à une version précédente



# Modèle DevOps

## ***Déploiement immuable***

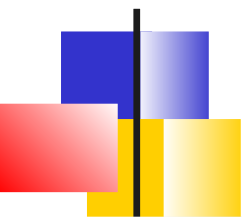
---

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

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

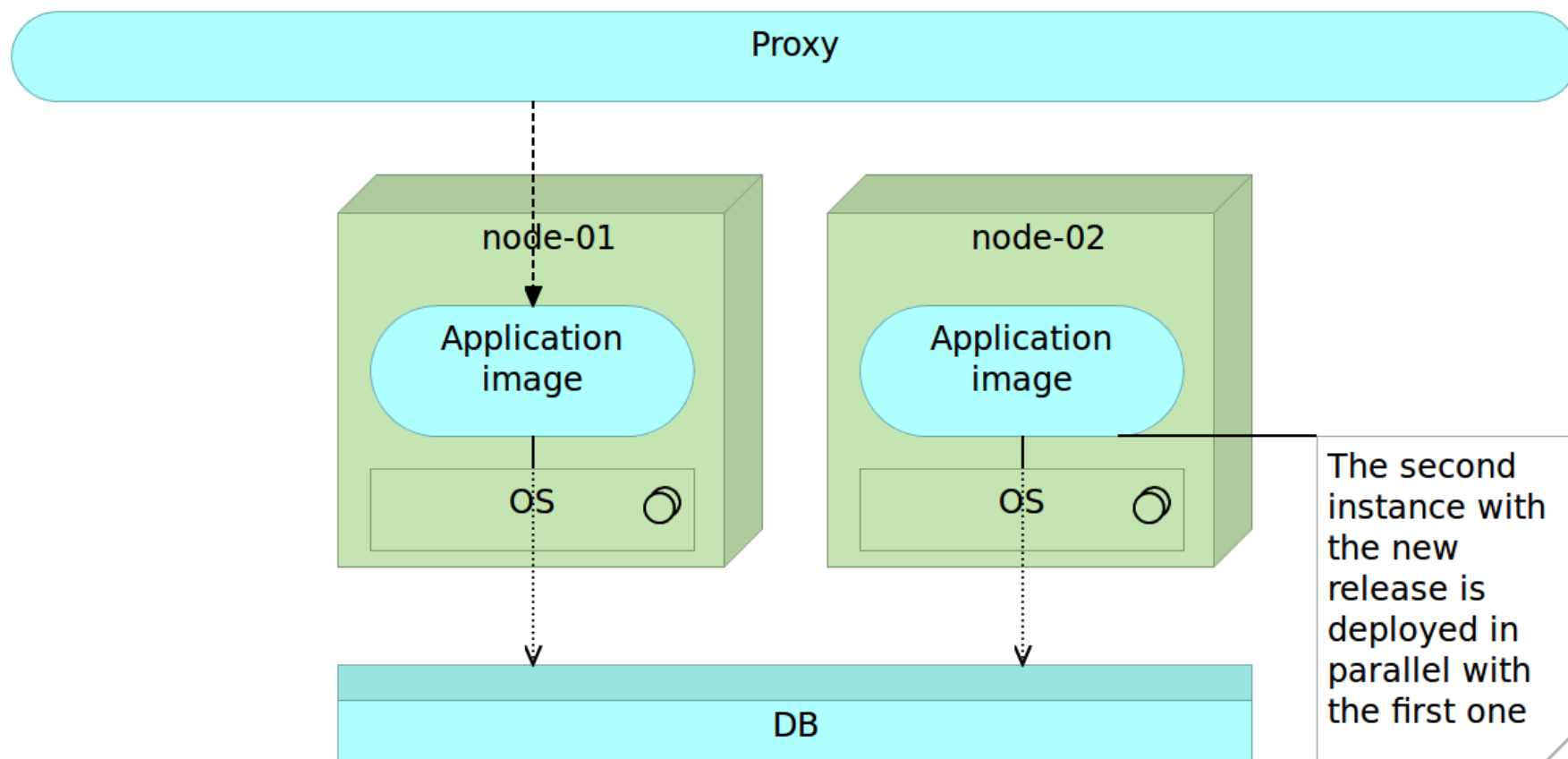
C'est ce tout qui est déployé

Plusieurs versions peuvent être déployées sur la même infrastructure



# Upgrade

## *déploiement immuable et Reverse Proxy*



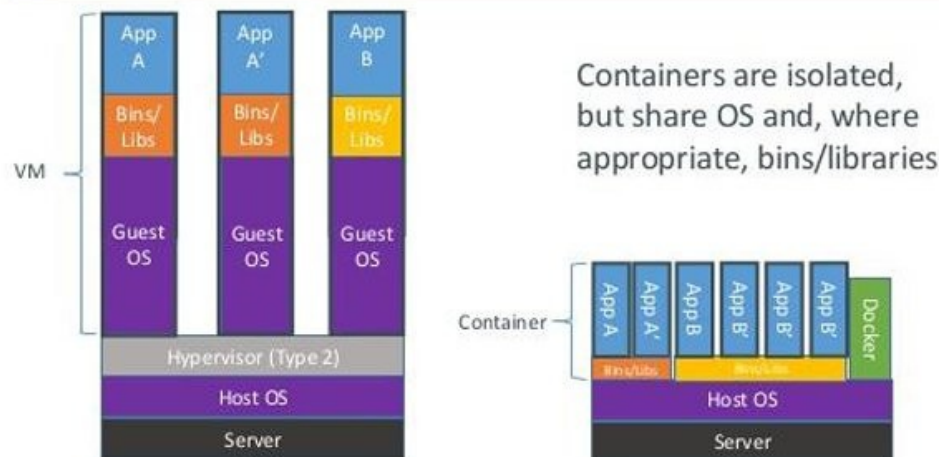


# Virtualisation et Containerisation

Les déploiements immuables sont théoriquement possible avec la virtualisation et la containerisation.

Mais la virtualisation est trop gourmande en ressources

## Containers vs. VMs





# Orchestrateur de conteneurs

## *Fonctionnalités applicatives*

---

En choisissant la containerisation, on peut profiter des orchestrateurs de container qui offrent de nombreuses fonctionnalités

- Surveillance des déploiements, redémarrage automatique en cas de problème
- Historique des déploiements, roll-back
- Déploiements Blue/Green ou même Canary Deployment
- Scaling automatique
- Services pour les architectures distribuées :
  - Configuration centralisée, gestion des secrets
  - Service de discovery, réplication et load balancing
- Sécurité : Pattern gateway, mTLS, compte service, ...



# Exemple : Ressource *deployment*

---

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 2
  spec:
    containers:
      - image: dthibau/annuaire:1.0.1
        name: annuaire
```

A partir de ce type de fichier *.yaml*, on peut créer la ressource et donc déployer via :

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



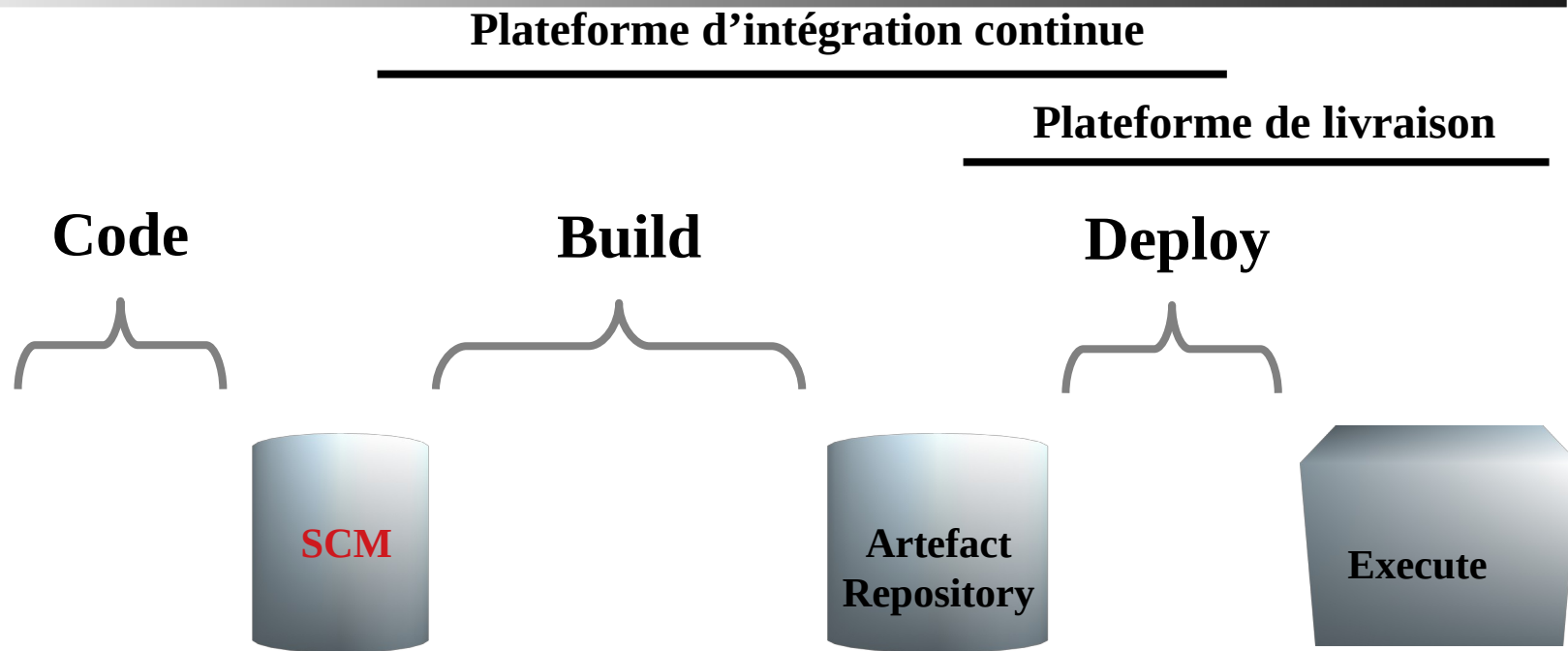
# SCM

---

## **Typologie**

Branches et collaboration  
Principales commandes Git

# Les SCMS dans le Cycle de vie



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



# SCM

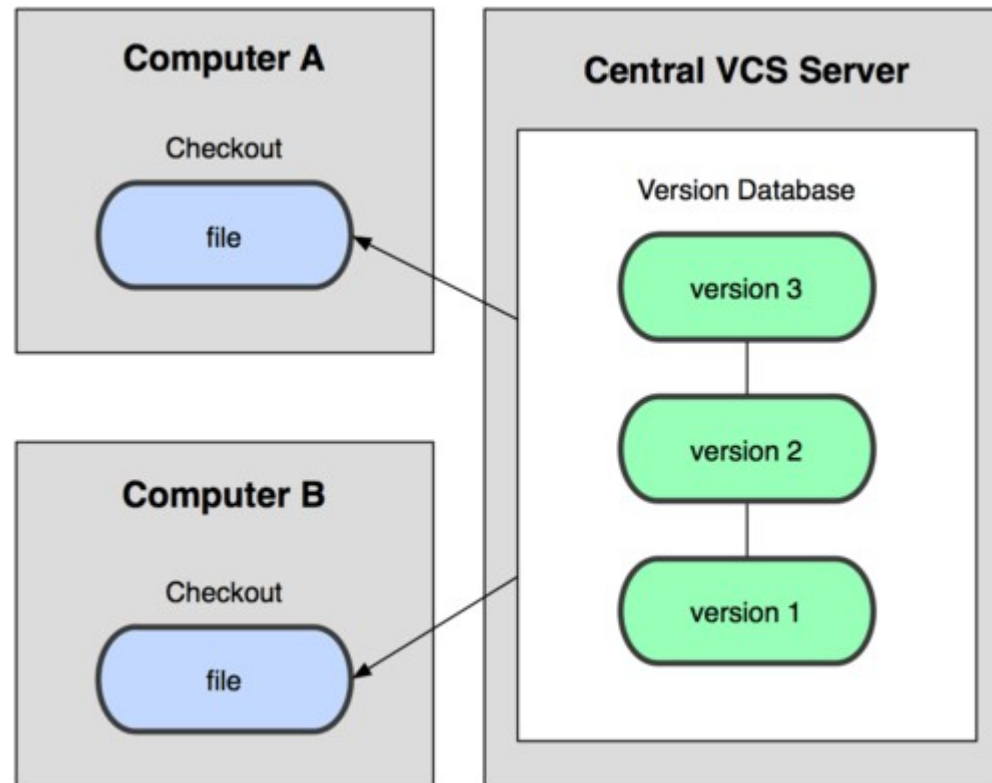
---

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

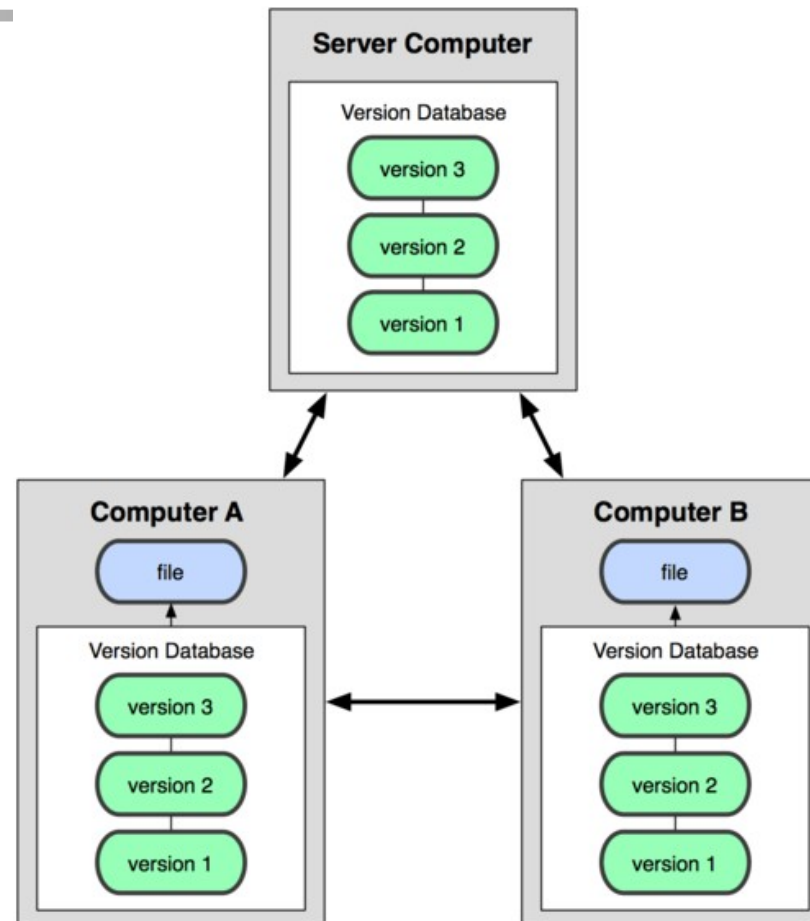
Le système permet :

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

# SCM centralisés : SVN, CVS, Perforce



# SCM distribués : Git, Bitbucket







# Principales opérations

---

***clone, copy*** : 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



# Stockage des données

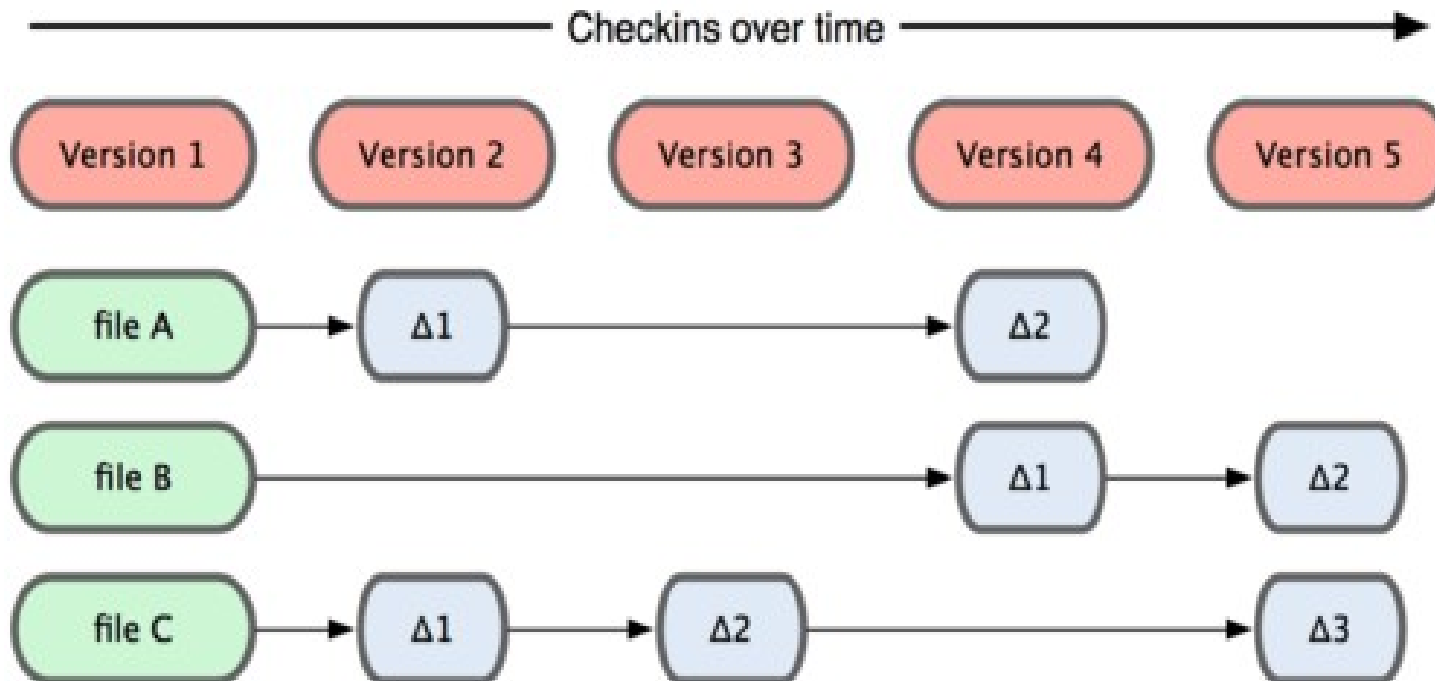
Les systèmes traditionnels stockent en général le fichier original et toutes les modifications qui lui ont été apportés (*patch*)

Les nouveaux systèmes stockent des instantanés complets de l'arborescence projet

- Pour être efficace, si un fichier est inchangé, son contenu n'est pas stocké une nouvelle fois mais plutôt une référence au contenu précédent

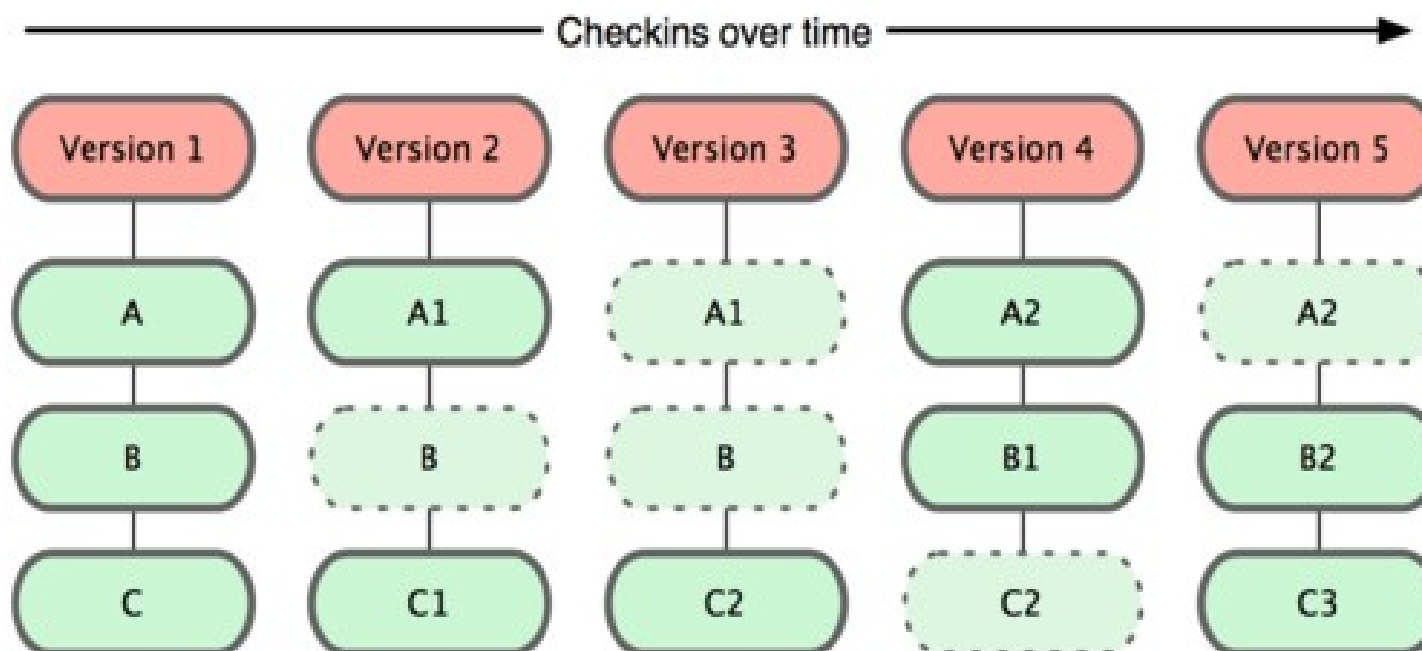


# Approche standard





# Approche Git





# Gestion des branches

Les outils diffèrent également par leur gestion des branches

- Branche par copie. CVS, SVN
  - => La création de branche est une opération lourde qui a des impacts sur la taille du projet
  - => Dans les SCM centralisés, le dév. n'utilise pas de branches locales
- Branche par pointeurs. Git BitBucket
  - La création de branche n'a pas d'impact sur le projet
  - Le projet peut contenir de nombreuses branches
  - En distribué, les dév. Peuvent utiliser des branches locales pour démarrer tout nouveau travail



# SCM

---

Typologie  
**Branches et collaboration**  
Principales commandes Git



# 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.

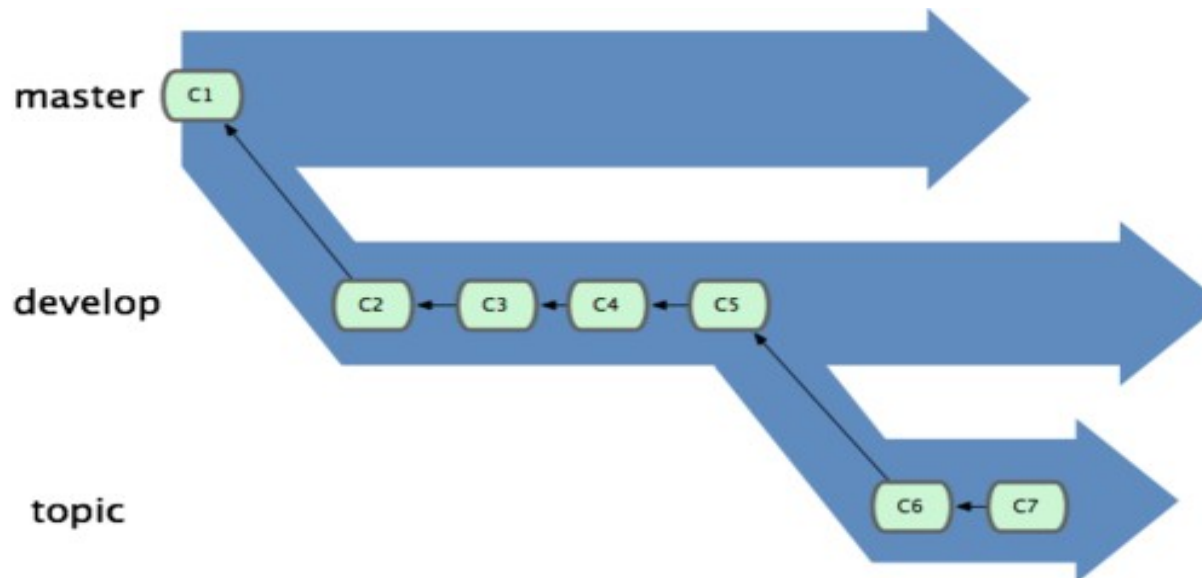
Les branches stables du serveur ont souvent des fonctions différentes (production, intégration)

# 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





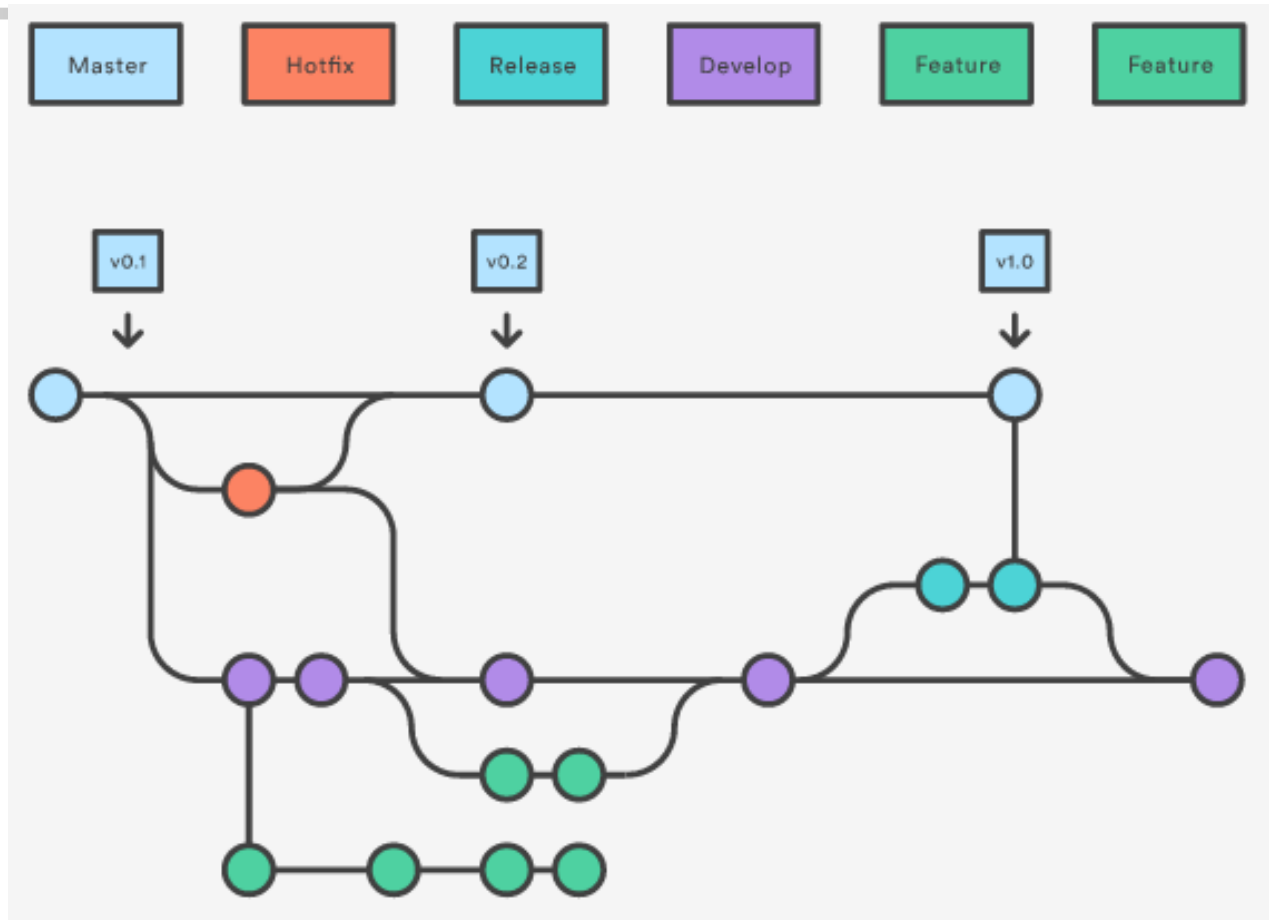
# Gitflow

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



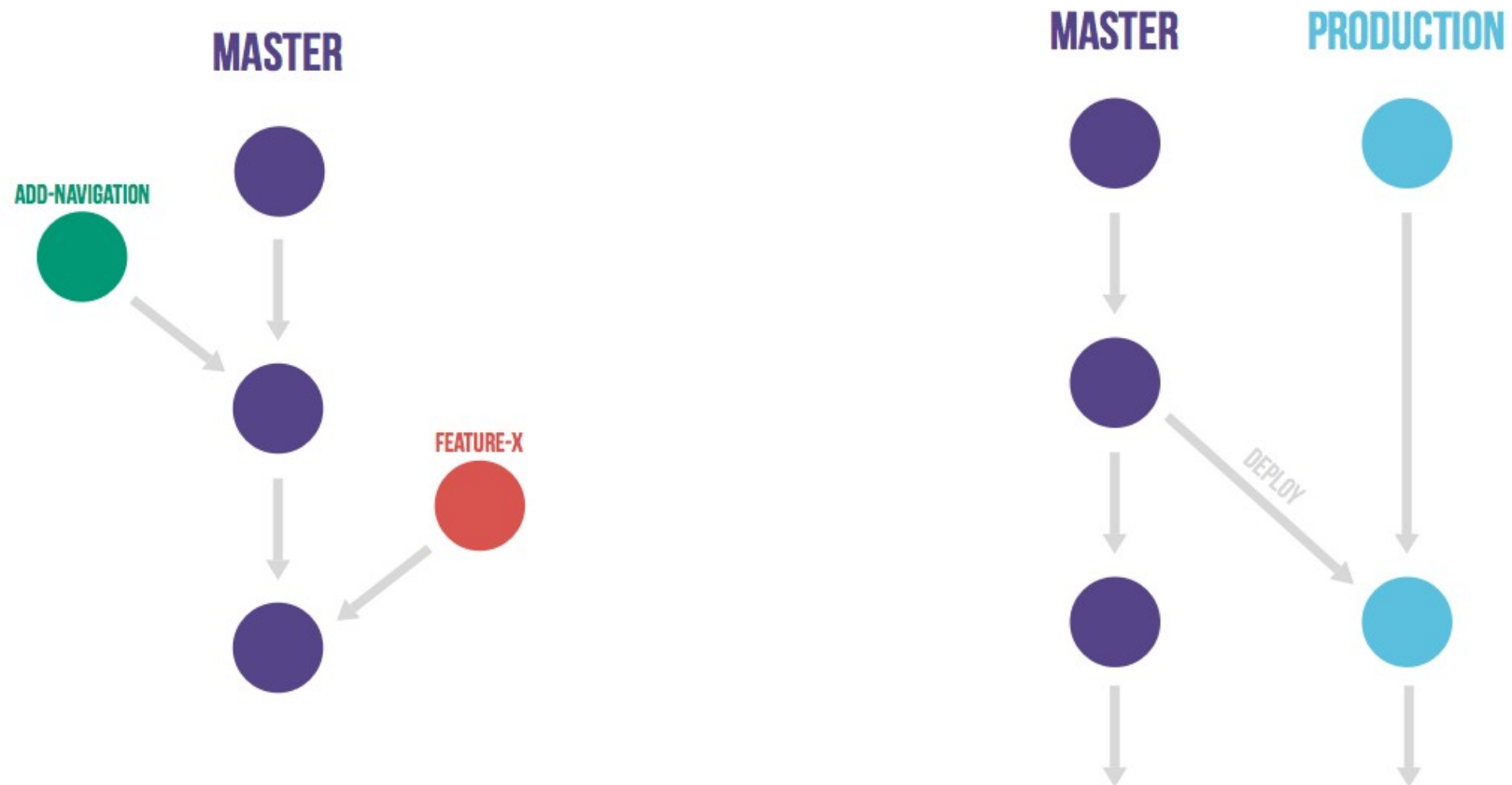
# Gitlab Flow

---

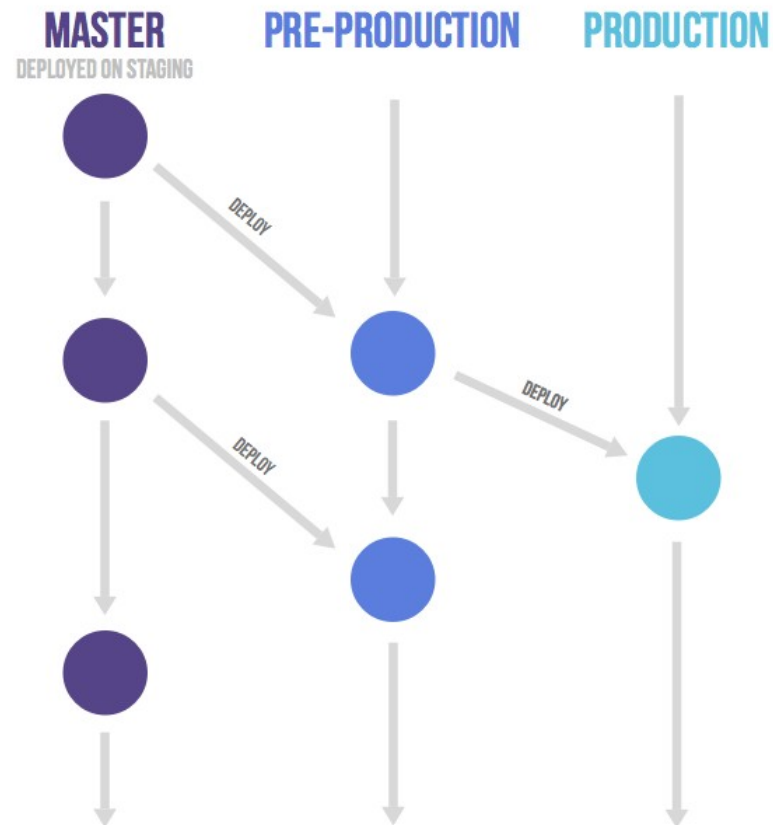
**Gitlab Flow** est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

- 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

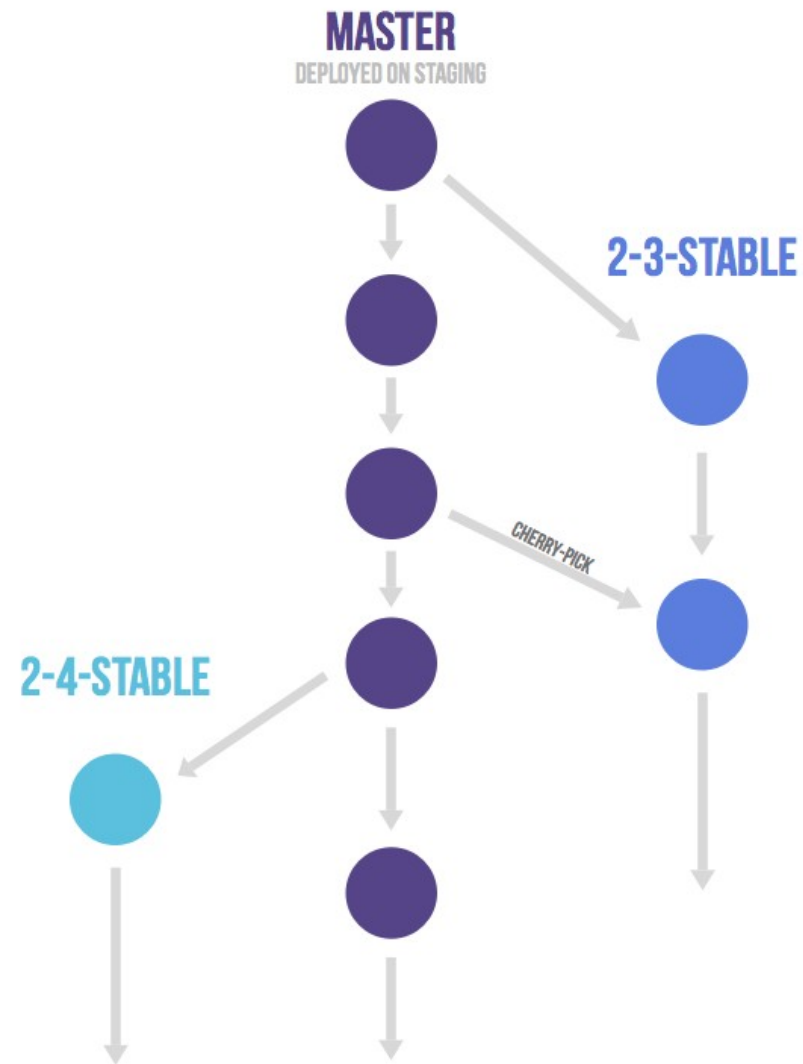


# Déclinaison avec staging





# Branches de releases





# SCM

---

Typologie  
Branches et collaboration  
**Principales commandes Git**



# Créer un dépôt

---

Il y a 2 façons de créer un dépôt :

- **Importer** un projet existant dans Git

```
$ git init
```

- **Cloner** un dépôt d'un autre serveur

```
$ git clone git://github.com/schacon/grit.git
```



# Fichiers du répertoire de travail

---

Chaque fichier du répertoire de travail peut être ***suivi*** ou ***non-suivi***.

Les fichiers non suivis sont indiqués dans ***.gitignore***

Les fichiers suivis peuvent être dans les 3 statuts : ***non-modifié***, ***modifié*** ou ***indexé***

- Lors de l'édition d'un fichier, Git le détecte comme modifié
- Il faut alors les passer dans la zone de staging ou index pour les committer



# *git status*

L'outil principal pour vérifier le statut des fichiers est ***git status***

Par exemple, le résultat de cette commande après un clone :

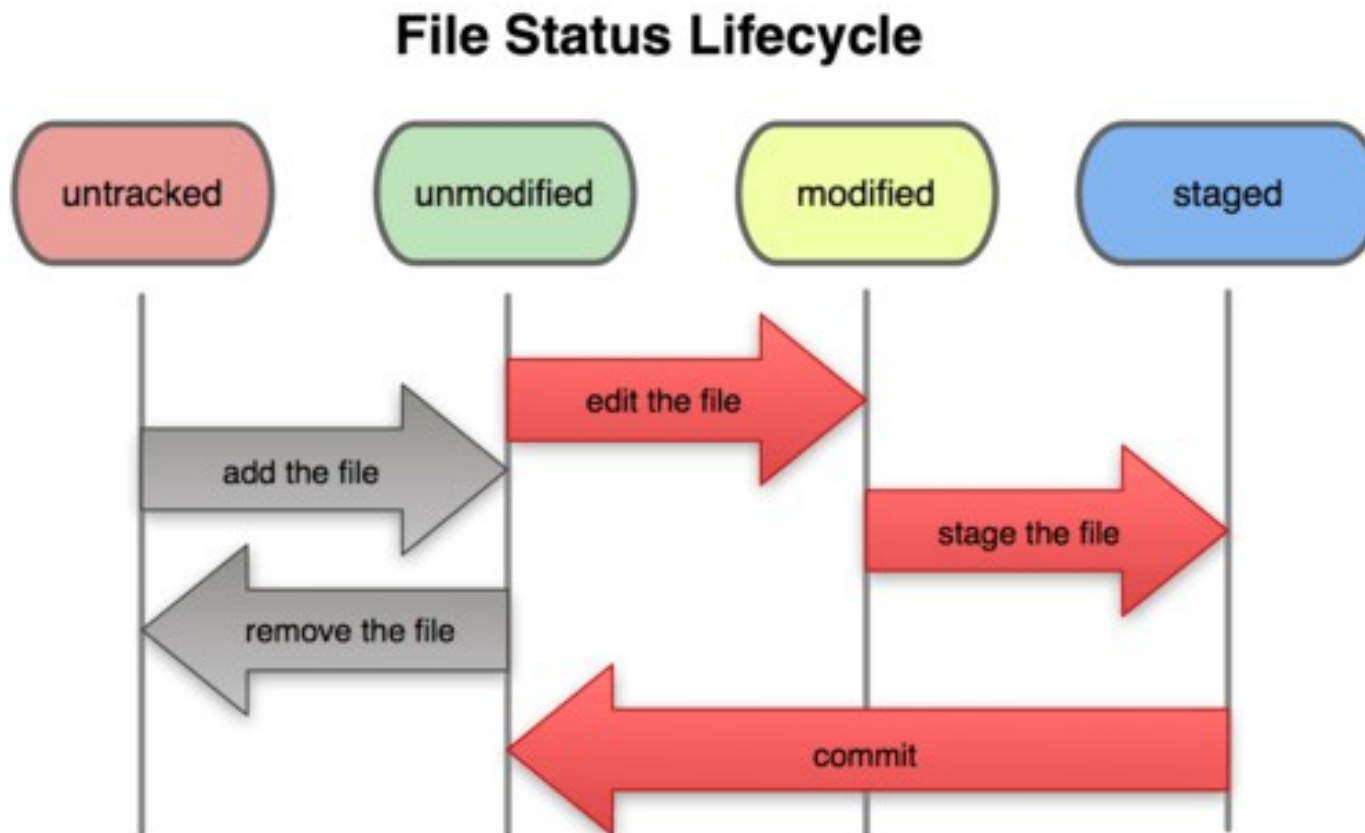
```
$ git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

=> Aucun fichier suivi n'a été modifié sur la branche par défaut *master*

# Statuts des fichiers





# Ajouter des fichiers

Pour commencer à suivre un fichier ou l'indexer, il faut donc utiliser la commande ***git add***.

Par exemple :

```
$ git add README
```

```
$ git status
```

On branch master

**Changes to be committed:**

(use "git reset HEAD <file>..." to unstage)

```
new file:   README
```

La commande *git add* prend en argument un chemin de fichier ou de répertoire (récursif).



# Commit

---

La commande *commit* n'a d'effet que sur l'index.

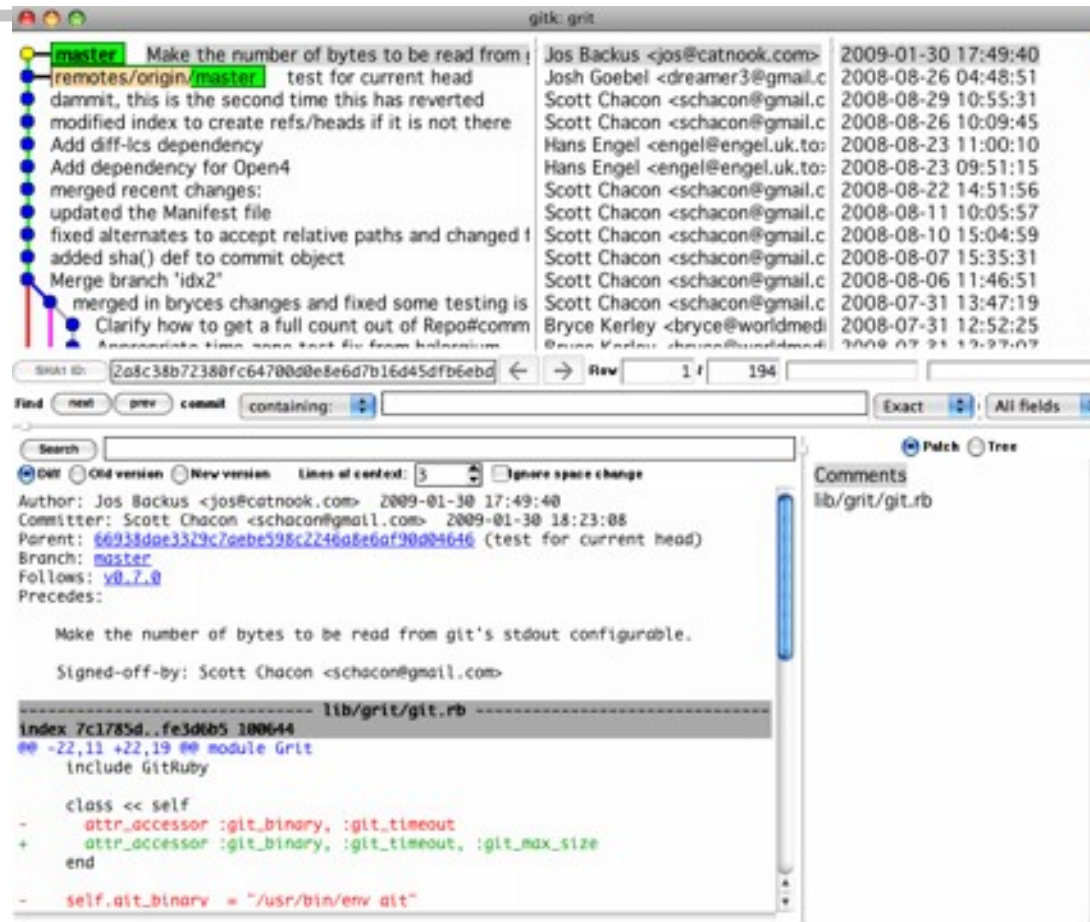
Tous les fichiers créés ou modifiés qui n'ont pas été ajoutés ne participent pas au commit

***git commit*** démarre l'éditeur spécifié par la variable d'environnement *\$EDITOR*

Le développeur doit fournir un commentaire afin que le commit soit effectif.



# Historique : *gitk --all*



The screenshot shows the gitk graphical interface. The top window displays a commit history with a list of commits on the left and a table of commit details on the right. The commit details table includes the commit hash, author, committer, parent, branch, and date. The bottom window shows the details of the selected commit (2a8c38b72380fc64700d0e8e6d7b16d45dfb6bd), including the author, committer, parent, branch, and the commit message. The commit message is "Make the number of bytes to be read from git's stdout configurable." and it is signed-off by Scott Chacon. The bottom window also shows the diff of the commit, with changes in the file lib/grit/git.rb.

Commit Hash	Author	Committer	Parent	Branch	Date
2a8c38b72380fc64700d0e8e6d7b16d45dfb6bd	Jos Backus <jos@catnook.com>	Scott Chacon <schacon@gmail.com>	66933d0e3329c70e8e598c2246a8e6af90004646	master	2009-01-30 17:49:40
66933d0e3329c70e8e598c2246a8e6af90004646	Josh Goebel <dreamer3@gmail.com>	Scott Chacon <schacon@gmail.com>			2008-08-26 04:48:51
...	...	...	...	...	...

Commit Details: 2a8c38b72380fc64700d0e8e6d7b16d45dfb6bd

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: 66933d0e3329c70e8e598c2246a8e6af90004646 (test for current head)  
Branch: master  
Follows: v0.7.0  
Precedes:

Make the number of bytes to be read from git's stdout configurable.  
Signed-off-by: Scott Chacon <schacon@gmail.com>

lib/grit/git.rb

index 7c1785d..fe3d0b5 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"



# Création de branche

Une branche Git est simplement un **pointeur** pouvant se déplacer sur les commits du référentiel.

- La branche par défaut est nommé *master*.

La création de branche se fait par :

```
$ git branch testing
```

Le basculement vers une branche existante :

```
$ git checkout testing
```

Les 2 à la fois

```
$ git checkout -b testing
```



# Fusion de la branche de développement

Fusion d'une branche de développement dans *master*

```
$ git checkout master
```

```
$ git merge iss53
```

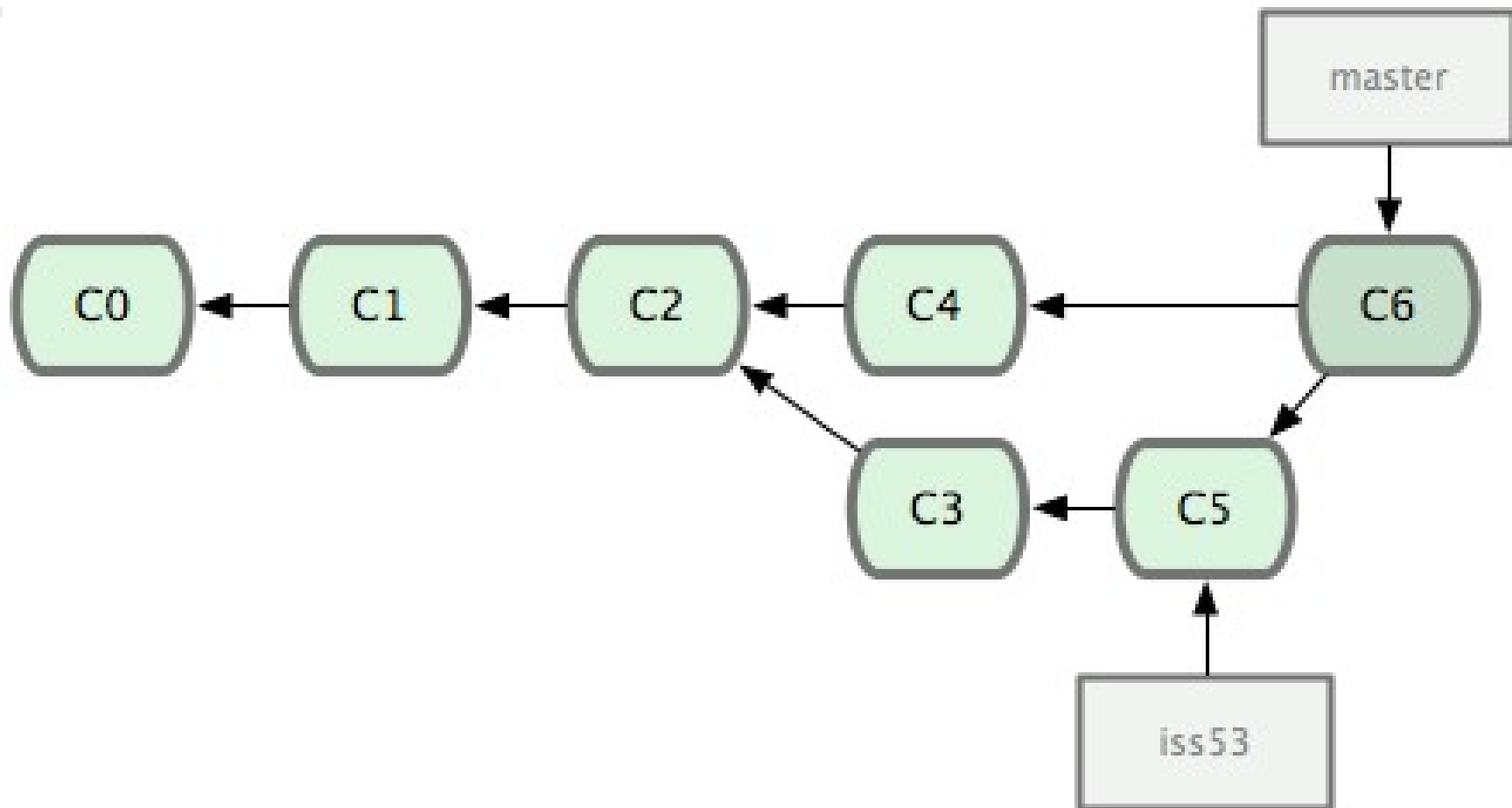
```
Auto-merging README
```

```
Merge made by the 'recursive' strategy.
```

```
README | 1 +
```

```
1 file changed, 1 insertion(+)
```

# Résultat de la fusion





# Rebase

Avec git, l'intégration de modification peut se faire par l'opération ***rebase***

L'opération consiste à rejouer les patches d'une autre branche sur le patch commun d'une autre branche

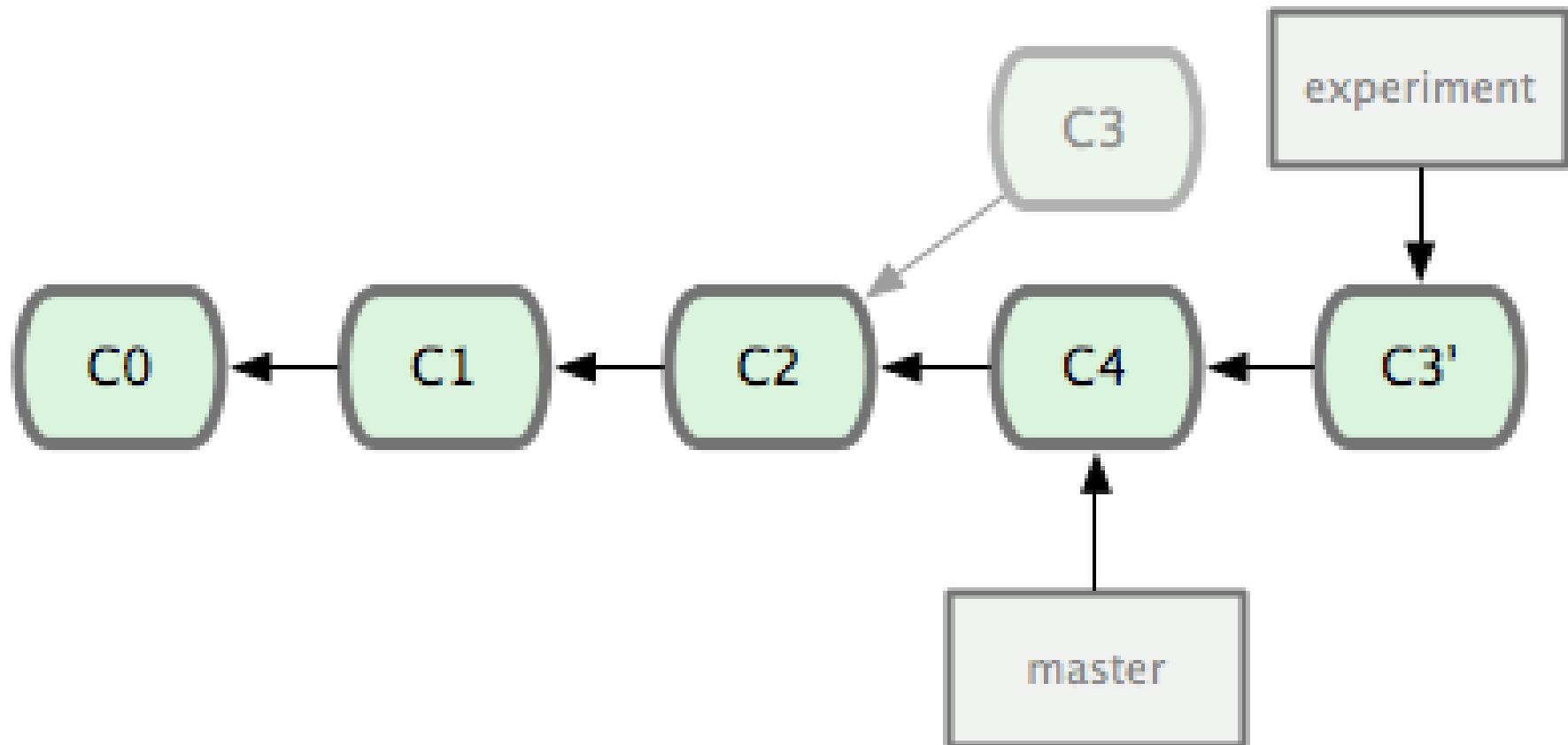
```
$ git checkout experience
```

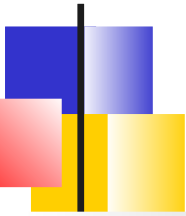
```
$ git rebase master
```

```
First, rewinding head to replay your work on top  
of it...
```

```
Applying: added staged command
```

# Rebase





# Commandes de collaboration

Il y a 4 opérations de synchronisation avec un dépôt distant :

- **clone** : A l'initialisation, récupère l'ensemble du dépôt et extrait la branche master dans le répertoire de travail
- **fetch** : Se synchronise avec le dépôt (récupération des nouvelles infos) sans modifier le répertoire de travail
- **pull** : Se synchronise avec le dépôt et fusionne les modifications avec le répertoire de travail
- **push** : Pousse ses modifications locales vers le dépôt distant. Opération possible seulement si le dépôt local est à jour



# Récupérer un dépôt distant

La commande ***git fetch*** permet de récupérer un dépôt distant  
La commande se connecte au projet distant et récupère toutes les données que l'on ne possède pas déjà

Cette commande ne modifie pas l'espace de travail courant

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master      -> pb/master
* [new branch]      ticgit      -> pb/ticgit
```

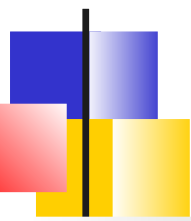
=> *La branche master de Paul est accessible localement par pb/master. Il est possible de la fusionner avec une de ses branches ou d'effectuer un check out complet.*





# *git pull*

A la différence de *git fetch*, la commande ***git pull [remote] [branch]*** récupère et fusionne les données automatiquement dans la branche courante. (comme *git clone* qui permet d'initialiser le dépôt et le répertoire de travail)



# Pousser vers un dépôt distant

---

Lorsque votre projet local a atteint un point de développement à partager, il faut utiliser la commande

***git push [remote-name] [branch-name]***

\$ git push origin master

Cette commande est possible seulement si on a les droits d'écriture sur le dépôt distant et si personne n'a poussé de données entre temps

Si une opération *push* a eu lieu auparavant, il faut d'abord récupérer les données et les fusionner avant de pouvoir les pousser vers le dépôt



# Maven

---

## **Concepts généraux**

Gestion des dépendances  
Adaptation du cycle de build  
Profils de build  
Projets multi-modules



# 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

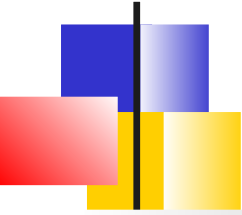


# Qu'est ce que Maven

---

- Définition officielle (Apache) : Outil de gestion de projet qui fournit :
  - un **modèle** de projet
  - un ensemble de **standards**
  - un **cycle de vie (ou de construction)** de projet
  - un système de **gestion de dépendances**
  - de la logique pour exécuter des **objectifs** via des **plugins** à des **phases** bien précises du cycle de vie/construction du projet
- Pour utiliser Maven, on doit décrire son projet en utilisant un modèle bien défini. (*pom.xml*)
- Ensuite, Maven peut appliquer de la logique transverse grâce à un ensemble de plugins partagés par la communauté Java (pouvant être adaptés)

# Convention plutôt que Configuration

- 
- Concept permettant d'alléger la configuration en respectant des conventions
  - Par exemple, Maven présuppose une organisation du projet dont le but est de produire un jar:
    - **`${basedir}/src/main/java`** contient le code source
    - **`${basedir}/src/main/resources`** contient les ressources
    - **`${basedir}/src/test`** contient les tests
    - **`${basedir}/target/classes`** les classes compilées
    - **`${basedir}/target`** le jar à distribuer
  - Les plugins cœur de Maven se basent sur cette structure pour compiler, packager, générer des sites webs, etc..

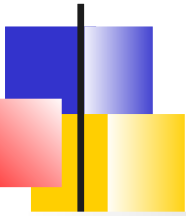


# Avant/Après

---

- 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***

# Réutilisation et adaptation via les plugins



- Le cœur de Maven n'est capable que de parser quelques documents XMLs. La plupart des tâches sont déléguées à des **plugins** qui peuvent affecter le cycle de vie Maven et permettent d'atteindre des **buts**.
- Ces plugins sont récupérés automatiquement à partir du dépôt central Maven lorsqu'une exécution le nécessite





# Modèle conceptuel d'un projet

---

- Maven s'appuie sur modèle de projet :  
« ***Project Object Model*** ».
- Les développeurs doivent le renseigner dans le ***pom.xml*** à la racine du projet:
  - Coordonnées permettant d'identifier le projet
  - Mode de licence
  - Développeurs et contributeurs
  - Dépendances vis-à-vis d'autres projets
  - ...



# Contenu du POM

---

Le POM contient 5 types d'information :

**Relations entre POM** : relation d'héritage, définition de sous module,

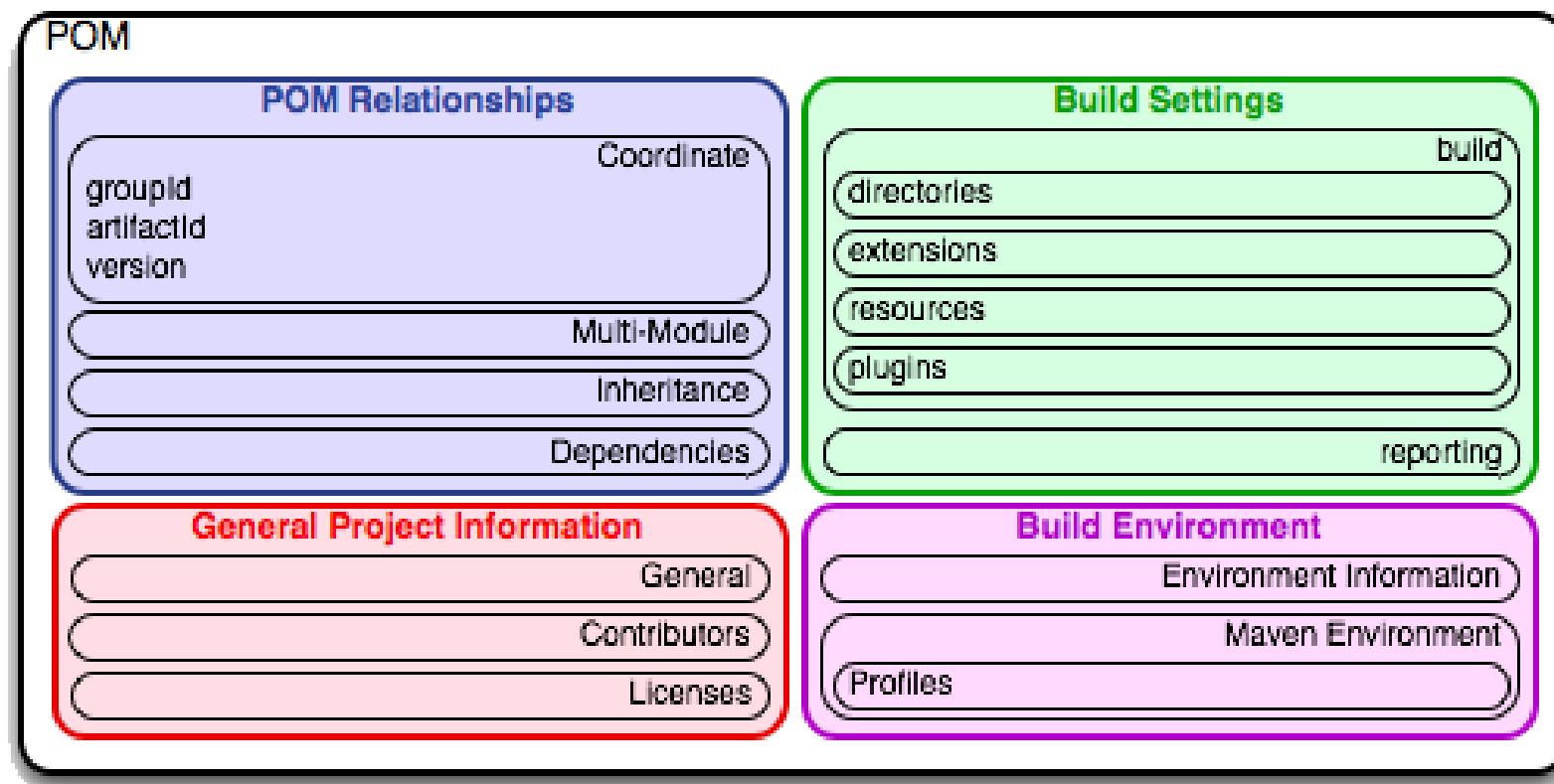
Dépendances du projets : Libraires utilisées pour compiler, exécuter, tester, etc..

**Information générale du projet** : Coordonnées unique, nom, URL, organisation, développeurs, contributeurs, licence

**Configuration du build** : Personnalisation du build par défaut : changer la localisation des sources, ajout de plugin,

**Profils de Build** : Différents profils, surchargeant la configuration par défaut peuvent être définis

# 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>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.10.1</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** conditionne le cycle de construction (Exemples : jar, war, pom) 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 à la configuration de *setting.xml*
  - Propriétés **Java**
- On peut en définir via :  
`<properties> <foo>bar</foo> /properties>`



# 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



# Options Maven

---

**-h** : Permet d'afficher toutes les options disponibles

Les plus importantes étant :

**-D** : Permet de fixer une propriété de la JVM

`mvn -Dsonar.token=$TOKEN`

**-f** : Indique un autre fichier que *pom.xml*

**-l** : Permet d'indiquer un fichier de log

**-o** : Offline, permet de tester si on peut builder sans réseau

**-P** : Profils de build

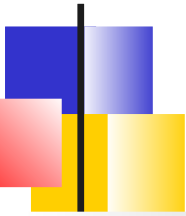
**-q** : Quiet, n'affiche que les erreurs

**-U** : Force la vérification des releases manquantes et réactualise les SNAPSHOTS

**-s** : Permet d'indiquer un autre fichier *settings.xml*

**-X** : Debug





# Variables d'environnement

---

Maven est également sensible à certaines variables d'environnement.

**JAVA\_HOME** est utilisé pour localiser le compilateur

**MAVEN\_OPTS** permet de positionner des options de démarrage de la JVM

Ex : -Xms256m -Xmx512m

Les options de la JVM peuvent également être positionnées dans le fichier *.mvn/maven.config*

**MAVEN\_ARGS** les options de Maven

Les options peuvent également être positionnées dans *.mvn/jvm.config*



# 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 *Sonar*, le plugin *Jruby*, ...

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

Les plugins sont bien documentés :

Ex: <https://maven.apache.org/surefire/maven-surefire-plugin/>



# Maven Plugins et objectifs

---

- Un objectif est une tâche spécifique qui peut être exécutée en *standalone* ou comme faisant partie d'un build plus large.
- La syntaxe d'exécution est la suivante :  
**mvn <plugin>:<objectif>**
- Un **objectif est l'unité de travail** dans Maven  
Exemples : l'objectif de compilation dans le compilateur plugin
- Les objectifs peuvent être configurés via un certain nombre de propriétés.  
Par exemple : La version du JDK est un paramètre de l'objectif de compilation.



# Exemple

---

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

```
$ mvn archetype:create  
  -DgroupId=org.formation.maven.simpleProject \  
-DartifactId=simpleProject \  
-DpackageName=org.formation.maven
```



# Cycle de vie Maven

---

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

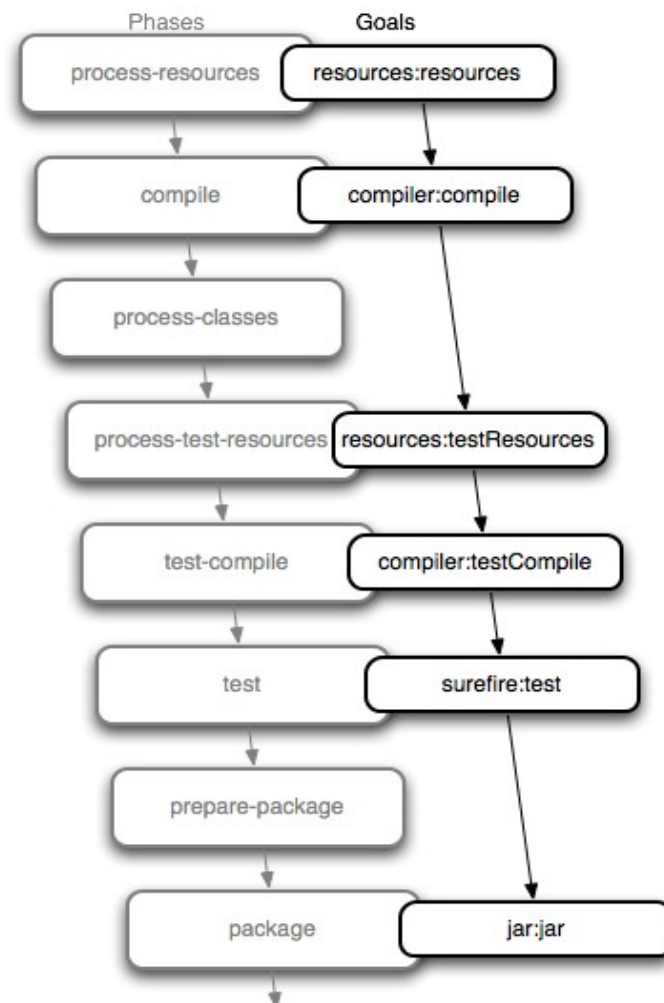


# Phase et objectifs

---

- Les objectifs d'un plugin peuvent être **attachés** à une phase du cycle de construction. Lorsque Maven atteint une phase, il exécute tous 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 construction.

# 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 *target/classes*
- ***compile / compiler:compile*** : compile tout le code source présent dans *src/main/java* vers *target/classes*.
- ***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





# Packaging *pom*

---

- L'artefact (l'objet à construire) est alors le POM  
Exemple des projets multi-modules, voir + loin
- Les objectifs par défaut sont alors différents :
  - ***package / site-attach-descriptor*** : Ajoute le descripteur de site (*site.xml*) aux fichiers devant être installés
  - ***install / install:install*** : Installe l'artefact (pom.xml) dans le dépôt local
  - ***deploy / deploy:deploy*** : Installe l'artefact dans le dépôt distant



# Packaging JavaEE

---

- Trois packaging à l'environnement Java EE sont disponibles :
  - *ejb* : EJB (2.1 et 3)
  - *war* : Application web
  - *ear* : Application d'entreprise
- Ils incluent les descripteurs de déploiement dans les objectifs de build



# 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 indiquée 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>
```



# Fichier settings.xml

---

Un autre fichier configure le comportement Maven :  
***settings.xml***

Il se place généralement dans `${HOME}/.m2` et on y trouve principalement des personnalisations de l'utilisateur :

- Identifiants de l'utilisateur pour se connecter à des dépôts
- Définitions des serveurs miroirs
- Personnalisation de l'emplacement du dépôt local
- Définition de groupes de plugins, non standard

Il peut également se trouver dans le répertoire projet ou être indiqué via l'option `-s` de la commande *mvn*



# Éléments de *settings.xml*

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0  
      https://maven.apache.org/xsd/settings-1.0.0.xsd">
```

```
  <localRepository/>  
  <interactiveMode/>  
  <offline/>  
  <pluginGroups/>  
  <servers/>  
  <mirrors/>  
  <proxies/>  
  <profiles/>  
  <activeProfiles/>
```

```
</settings>
```



# Exemple configuration Proxy

---

```
<settings>
.
.
<proxies>
  <proxy>
    <id>example-proxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.example.com</host>
    <port>8080</port>
    <username>proxyuser</username>
    <password>somepassword</password>
    <nonProxyHosts>www.google.com|*.example.com</nonProxyHosts>
  </proxy>
</proxies>
.
.
</settings>
```





# Maven

---

Concepts généraux  
**Gestion des dépendances**  
Adaptation du cycle de build  
Profils de build  
Projets multi-modules



# 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. C'est une de fonctionnalités les plus puissantes de Maven.
- Les dépendances peuvent être utilisées pour la compilation, le test, le packaging, etc.  
Il est donc possible de préciser des **scopes** qui indique à Maven quand utiliser la dépendance

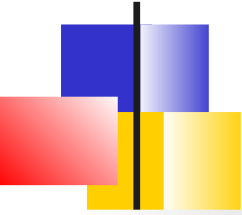


# Dépendances transitives

---

- 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

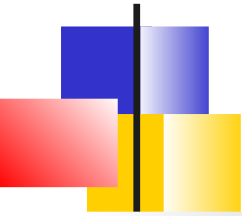


---

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

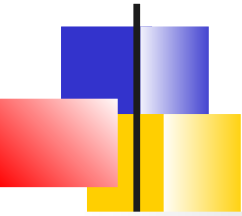
```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
</dependencies>
```

# Exclusion de dépendance transitive



```
<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>
```

# Scopes




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

5 scope sont disponibles :

- ***compile*** (défaut) : Les dépendances sont dans le classpath durant la compilation et les tests. Elles peuvent être packagées dans le jar final
- ***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)


# Example



---

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

# 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





# Version du projet

---

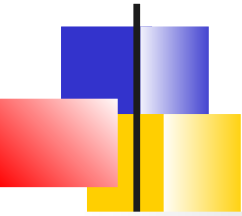
- La version d'un projet Maven permet de grouper et ordonner les différentes releases.
- Elle est constituée de 4 parties :

`<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

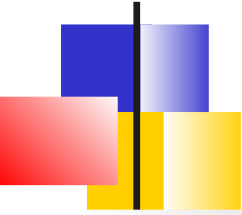


Pour éviter de disperser dans tous les projets de l'entreprise les mêmes dépendances classiques ... et d'avoir des soucis de mis à jour lors d'une montée de version

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

Cela est généralement effectué dans un POM parent

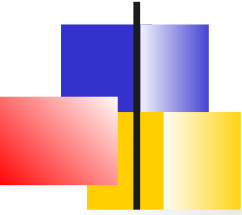
# POM parent



---

```
<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 projet

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 BOM

---

```
<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

---

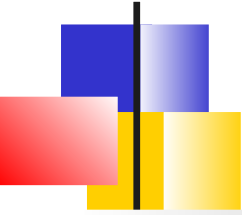
Le même mécanisme est utilisé pour fixer la version des plugins utilisés

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 versions de plugins



# Exemple *pluginManagement*



---

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
      </plugin>
    </plugins>
  </pluginManagement>
  ...
```



# Maven

---

Concepts généraux  
Gestion des dépendances  
**Adaptation du cycle de build**  
Profils de build  
Projets multi-modules



# 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*



# Example

---

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

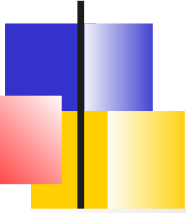


# Association objectif/phase.

## L'exemple du plugin Assembly

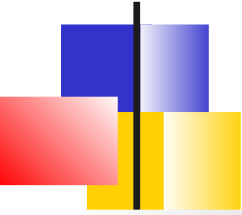
- Le plugin **Assembly** permet de créer des distributions du projets sous d'autres formats que le types d'archives standard (*.jar*, *.war*, *.ear*)
  - Il est ainsi possible de générer un fat jar (exécutable du projet)
- Par défaut, ce plugin n'est pas utilisé par le cycle de construction

# 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

# Example



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





# Maven

---

Concepts généraux  
Gestion des dépendances  
Adaptation du cycle de build  
**Profils de build**  
Projets multi-modules



# 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 profils sont **activés** manuellement, en fonction de l'environnement ou du système d'exploitation

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



# Example

---

```
<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>`

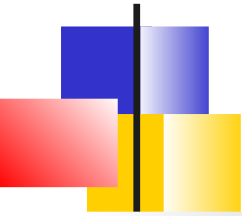
---

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



# Exemple

---

```
<profile>
  <id>dev</id>
  <activation>
    <activeByDefault>false</activeByDefault>
    <jdk>1.5</jdk>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
    <property><name>mavenVersion</name><value>2.0.5</value></property>
    <property><name>!environnement.type</name></property>
  <file>
    <exists>file2.properties</exists>
    <missing>file1.properties</missing>
  </file>
</activation>
...
</profile>
```



# Maven

---

Concepts généraux  
Gestion des dépendances  
Adaptation du cycle de build  
Profils de build  
**Projets multi-modules**

# Projets multi-modules

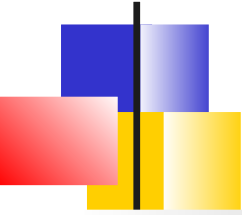


---

- 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>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>
  <modules>
    <module>simple-model</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>
</project>
```

# POM d'un projet multi-modules

- 
- Le POM parent définit les coordonnées Maven
  - Il ne 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>
```

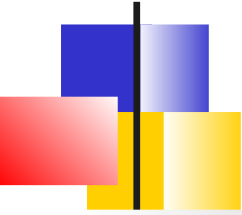
```
    <artifactId>simple-model</artifactId>
```

```
    <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



# Dépôts et Releasing

---

## **Les dépôts d'artefacts**

L'exemple de Nexus  
Processus de Release



# Dépôts

---

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

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



# Dépôt privé

---

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

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

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

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



# Utilisation d'un dépôt interne

---

Il suffit d'indiquer :

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





# Balise repository dans *pom.xml*

---

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



# Balise *<server>*

---

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



# *Deploy* plugin

---

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

En général, un dépôt d'entreprise.

Le plugin offre 2 objectifs principaux :

***deploy*** : attaché à la phase *deploy*, il publie l'artefact dans un dépôt distant

***deploy-file*** : Permet de déployer un fichier qui n'a pas été construit par Maven



# *<distributionManagement>*

---

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

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



# SNAPSHOTS et RELEASES

---

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

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



# Dépôts et Releasing

---

Les dépôts d'artefacts  
**L'exemple de Nexus**  
Processus de Release



# Nexus

---

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

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

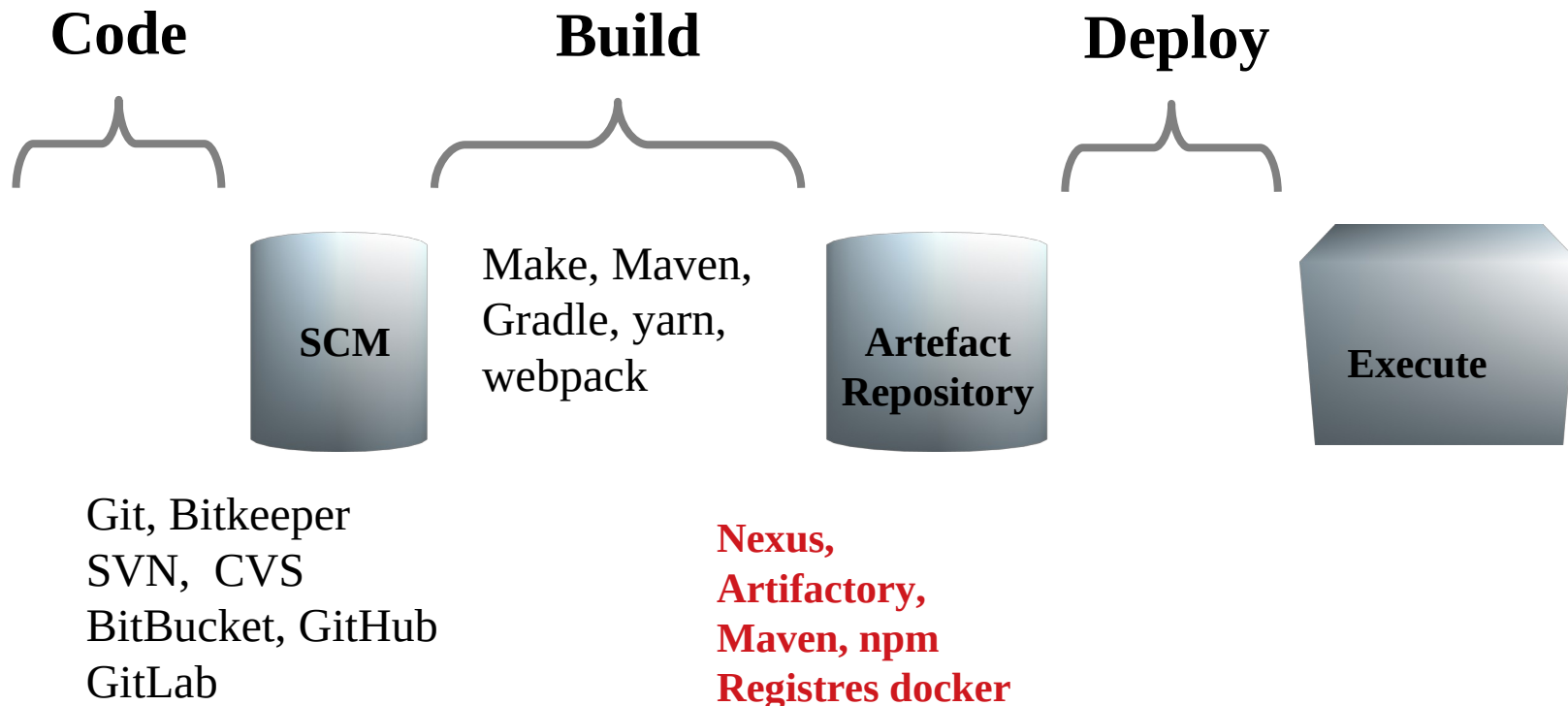
*Nexus* support de nombreux types de dépôts ... et bien sûr les dépôts Maven



# Les gestionnaires de dépôts dans le cycle de vie

Plateforme d'intégration continue

Plateforme de livraison







# Concepts

---

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

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



# Types de dépôt

---

Nexus permet de définir différents types de dépôts :

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



# Interface utilisateur

---

*Nexus* fournit un accès anonyme pour la recherche de composants

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

La fonctionnalité de recherche propose de nombreuses options



# Installation par défaut

---

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

*Administration → repositories → Add*

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

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



# Intégration Maven

---

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

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



# Example

---

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

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



# Configuration projet

---

Au niveau du projet Maven, il faut indiquer

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



# Configuration projet

---

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

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





# Dépôts et Releasing

---

Les dépôts d'artefacts  
L'exemple de Nexus  
**Processus de Release**



# Production d'une release

---

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

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

Ce processus devra également être automatisé dans le contexte de pipeline CD.



# 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
- Certains plugins d'outils de build implémentent tout le processus (*Maven Release plugin, Gradle Release plugin*)



# Maven et SCM

---

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

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

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



# Exemple GIT

---

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



# Plugin *Release*

---

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

Une release est effectuée en 2 étapes principales :

***prepare*** : Préparation

***perform*** : Réalisation de la release



# Préparation

---

La préparation effectue les étapes suivantes :

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



# Exemples de commande

---

Reprend la commande ou elle s'est arrêtée

```
mvn release:prepare
```

Reprend la commande depuis le début

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

Ou

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





# *Perform*

---

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

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



# Process de release

---

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

*release:prepare*

*release:perform*

Pour réussir, on doit s'assurer que :

Le *pom* effectif n'a pas de dépendances sur des versions SNAPSHOTS

Tous les changements locaux ont été commités

Tous les tests passent

Tout cela peut se faire en une seule commande maven :

***mvn release:prepare release:perform -B***



# Qualité avec SonarQube

---

## **Offre et architecture**

Concepts Sonar

Calculs de métriques

Mise en place et configuration

Profils et portes qualité spécifique



# Offre Sonar (1)

---

Modèle Open Source

- <http://www.sonarqube.org/> : développement / support

Analyse complète de la qualité d'une application

- Nombreuses métriques
  - Quantitatives : nombre de classes, duplication de code, ...
  - Qualitatives : couverture et taux de réussite des tests, complexité du code, respect des règles de codage...

Tableau de bord des applications

Historique des statistiques

Plus de 600 règles de qualité

Gestion de profils

Visualisation du code source



# Offre Sonar (2)

---

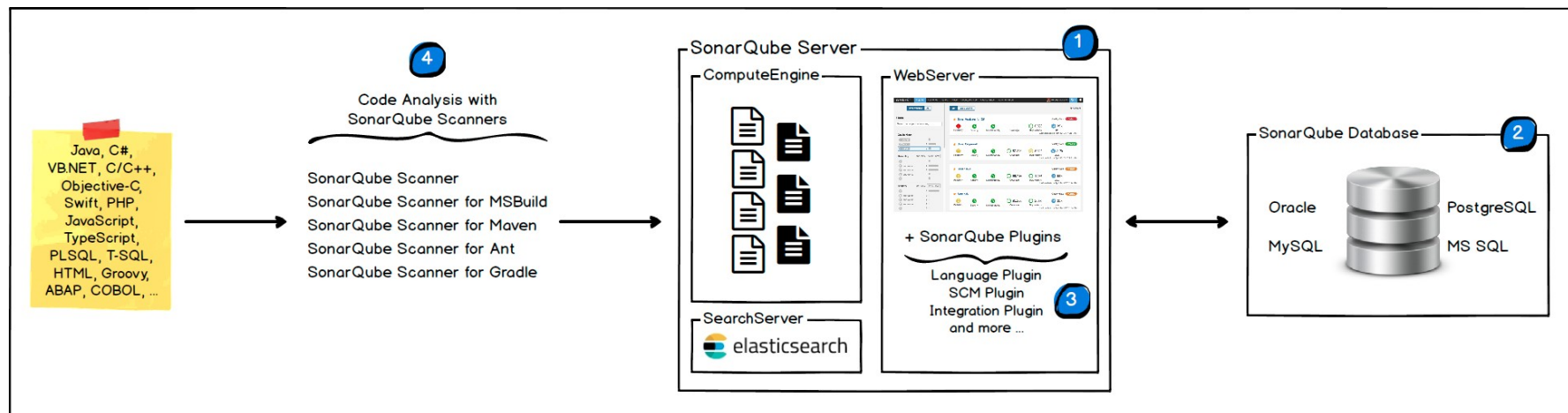
## Utilisation de nombreux outils via des plugins

- CheckStyle
- PMD
- FindBugs
- Cobertura, Jacoco
- Sonar Squid
- ...

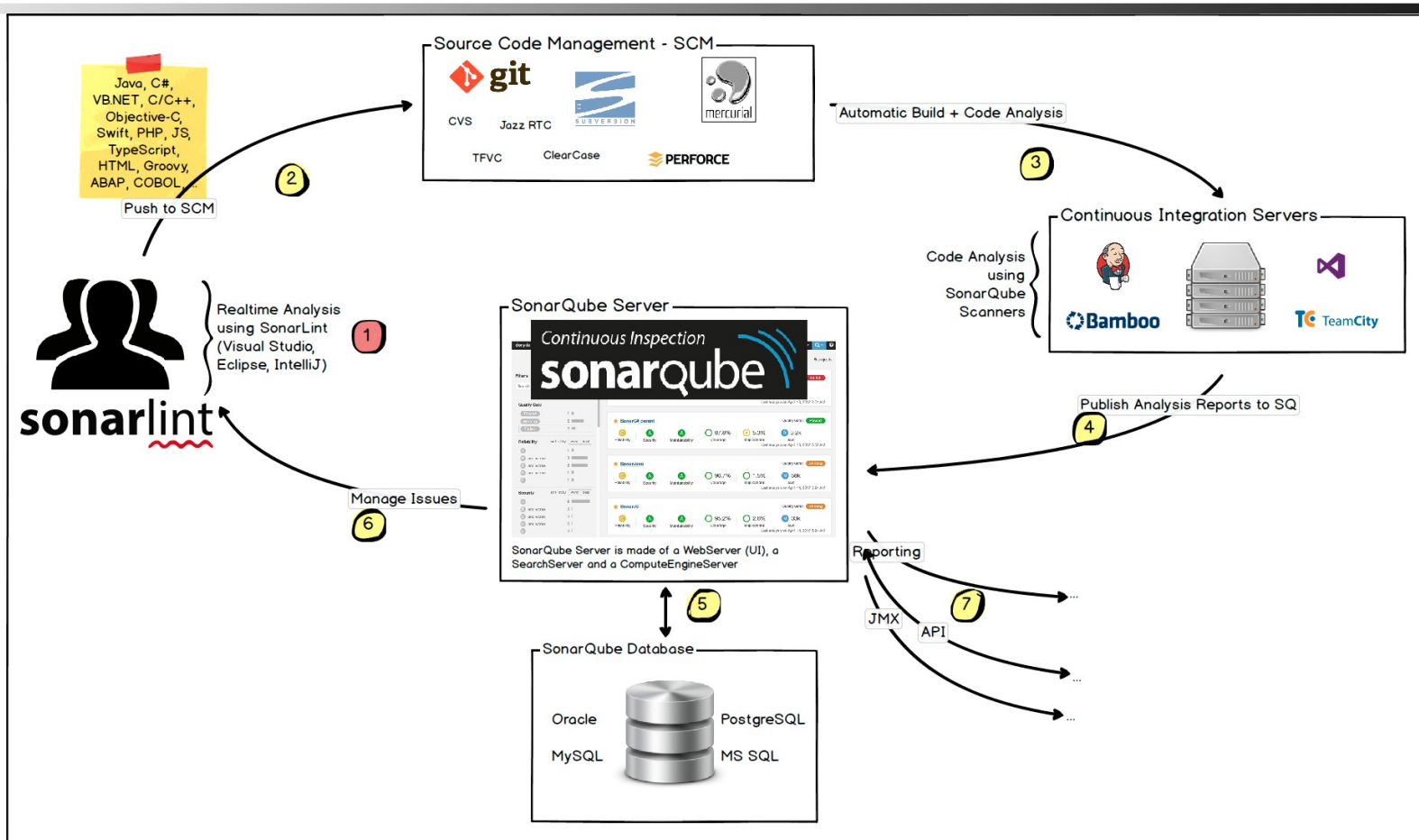
## Gestion de profils

- Configuration des règles de codage
- 1 profil par défaut : *Sonar way*
- Possible de créer ses propres profils

# Architecture



# Analyse continue





# Modes d'exécution

---

## Installation :

- Docker :

```
docker run -d --name sonarqube -p 9000:9000 sonarqube
```

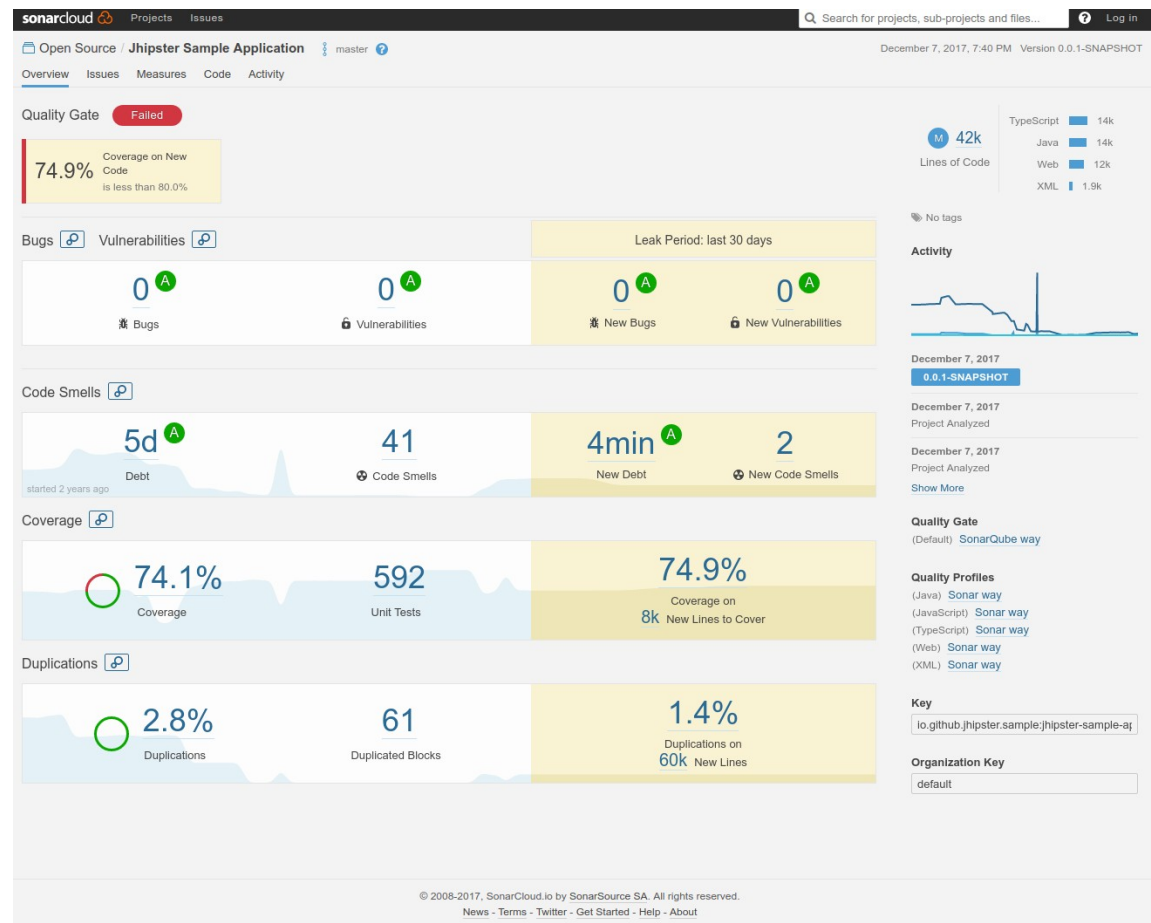
- Distribution sous forme d'archive  
décompresser *sonar-x.x.zip*

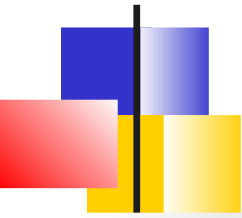
- Exécution

- En standalone
    - Comme service
    - Déployé comme application web sous Tomcat



# Sonar Dashboard





# Qualité avec SonarQube

---

Offre et architecture

**Concepts Sonar**

Calculs de métriques

Mise en place et configuration

Profils et portes qualité spécifique



# Analyse et Scanners

---

*Sonar* permet de démarrer une analyse facilement grâce à ses ***Scanners*** fournis

- MSBuild: Projets .Net
- Maven
- Gradle
- Ant
- Jenkins
- Commande en ligne



# Analyse

---

## Base de règles

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

L'analyse consiste à vérifier les règles du **profil qualité** associé au projet.

Lorsqu'une règle est transgressée, elle produit une **Issue**

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






# Règle

---

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

squid:S2116  

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

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

### Noncompliant Code Example

---

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

---



# Création de règles

---

La création de règles personnalisées (dans un environnement Java) peut se faire de différentes façons :

- Utiliser un plugin permettant d'exprimer des nouvelles règles selon sa syntaxe (Exemple : Checkstyle)
- Coder la règle en Java et l'intégrer dans le plugin Java de Sonar



# Exemple

```
@Rule(key = "MyFirstCustomCheck",
      name = "Return type and parameter of a method should not be the same",
      description = "For a method having a single parameter, the types of its return value and its
parameter should never be the same.",
      priority = Priority.CRITICAL, tags = {"bug"})
public class MyFirstCustomCheck extends IssuableSubscriptionVisitor {

    @Override
    public List<Kind> nodesToVisit() { // Quels nœuds on veut traiter
        return ImmutableList.of(Kind.METHOD);
    }

    @Override
    public void visitNode(Tree tree) { // Méthode effectuant la vérification
        MethodTree method = (MethodTree) tree;
        if (method.parameters().size() == 1) {
            MethodSymbol symbol = method.symbol();
            Type firstParameterType = symbol.parameterTypes().get(0);
            Type returnType = symbol.returnType().type();
            if (returnType.is(firstParameterType.fullyQualifiedName())) {
                reportIssue(method.simpleName(), "Never do that!");
            }
        }
    }
}
```



# Profil Qualité

---

Les ***profils qualité*** sont les emplacements ou sont définis les règles pour un projet particulier

*SonarSource* fournit un profil qualité par défaut pour chaque langage

Profil Java : 292 règles

- 93 bugs
- 18 vulnérabilités
- 181 Code smells





# SonarQube

Le travail quotidien des développeurs et *reviewers* consiste à traiter des *issues* (anciennement violation).

Il y en a de 3 types :

- **Code Smell** : Mauvaise pratique de conception rendant difficile la maintenance
- **Bug** : Un problème représentant une erreur dans le code. Cette erreur se manifestera un jour, il faut donc le corriger ASAP
- **Vulnérabilité** : Un problème lié à la sécurité permettant une attaque

Des coûts sont associés au traitement de ces problèmes :

- **Dette technique** : Le temps estimé pour corriger tous les problèmes de maintenabilité
- **Coût de correction** : Le temps estimé pour corriger les bugs et les vulnérabilités



# Sévérité

---

SonarSource définit des sévérités pour les issues :

- **BLOCKER** :  
Haute probabilité d'impacter le comportement en production (fuite de mémoire, de connexion JDBC, ...).  
=> Doit être fixé ASAP
- **CRITICAL** :  
Bug avec faible probabilité d'impact ou défaut de sécurité.  
=> Doit être revu ASAP.
- **MAJOR** :  
Défaut de qualité pouvant impacter la productivité du développeur (code mort, blocs dupliqués paramètres inutilisés, ...)
- **MINOR** :  
Défaut de qualité ayant un léger impact sur la productivité. (Lignes trop longues, "switch" à la place de if, ...)



# Revue d'issues

---

Le travail du *reviewer* consiste à revoir les résultats de l'analyse et de modifier éventuellement les statuts des problèmes :

- **Confirm** : Reconnaissance du problème. Le statut passe de « Ouvert » à « Confirmé »
- **Fausse détection** : Indique que l'issue détectée par Sonar n'est pas réellement un problème
- **Won't fix** : On reconnaît le problème ; mais on l'accepte en tant que dette technique
- **Changement de sévérité** : Le problème n'a pas la sévérité détectée par Sonar. On change sa sévérité et les prochaines mesures de Sonar tiendront compte du changement
- **Résolu** : Le problème est censé être résolu. Lors de la prochaine analyse, Sonar fermera ou ré-ouvrera l'issue



# Métriques

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

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



# SonarQube - Progression

---

SonarQube propose 2 moyens pour évaluer les progrès depuis le démarrage d'une démarche qualité :

- Les **Leak Period** :  
Permet de définir un point d'origine. *Sonar* montre alors des métriques par rapport au code introduit depuis ce point d'origine. (Typiquement release précédente ou début du Sprint)
- Les **portes qualité**  
Permet de définir un ensemble de seuils pour différentes mesures. Cela répond à la question : « Le code peut il aller en production ? » Les seuils progressent au fur et à mesure des releases.



# Porte qualité

---

Les portes qualité sont différentes en fonction des projets

La porte qualité « *SonarQube way* » est fournie par *SonarSource* et elle est activée par défaut

Chaque condition d'une porte qualité est la combinaison de :

- La métrique concerné
- La période : Date de la mesure ou différentiel par rapport au point d'origine (***Leak period***)
- Des seuils pour un avertissement ou pour une erreur



# Exemple : *SonarQube Way*

---

Metric	Over Leak Period	Operator	Warning	Error
Coverage on New Code	Always	is less than		80.0%
Duplicated Lines on New Code (%)	Always	is greater than		3.0%
Maintainability Rating on New Code	Always	is worse than		A
Reliability Rating on New Code	Always	is worse than		A
Security Rating on New Code	Always	is worse than		A



# Qualité avec SonarQube

---

Offre et architecture  
Concepts Sonar

**Calculs de métriques**

Mise en place et configuration  
Profils et portes qualité spécifique





# Complexité

---

## **Complexité cyclomatique :**

- Calculé à partir du nombre de chemins possibles dans le code. Chaque fonction a une complexité minimale de 1.

## **Complexité Cognitive :**

- Difficulté à comprendre les chemins dans le code  
*Voir <https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html>*

## **Mesures dérivées :**

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



# Documentation

---

## **Commentaires :**

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

## **Méthodes publiques et API**

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



# Duplications

---

## **Duplications**

- Nombre de blocs de lignes dupliqués
- Nombre de fichiers impliqués dans les duplications.
- Nombre de lignes impliqués dans les duplications.
- Pourcentage de lignes dupliquées



# Fiabilité Bugs

---

## **Bugs , New Bugs :**

- Nombre de bugs et de nouveaux bugs, par sévérité, par statuts

## **Indicateur de fiabilité** (*Reliability rating*)

- A = 0 Bug
- B = au moins 1 bug mineur
- C = au moins 1 bug majeur
- D = Au moins 1 critique
- E = Au moins 1 bloquant

## **Effort de correction** (*Reliability remediation effort*) :

- L'effort pour fixer tous les bugs ou tous les nouveaux bugs (en minutes et en se basant sur une journée de 8 heures),



# Sécurité

---

## **Vulnérabilités :**

- Nombre de vulnérabilités et de nouvelles vulnérabilités, par sévérité, par statut

## **Indicateur de sécurité** (*Security Rating*)

- A = Aucune vulnérabilité
- B = au moins 1 vulnérabilité mineure
- C = au moins 1 vulnérabilité majeure
- D = au moins 1 vulnérabilité critique
- E = au moins 1 vulnérabilité bloquante

## **Effort de sécurisation** (*Security remediation effort*) :

- Minutes pour corriger les vulnérabilités, sur tout le code, sur le nouveau code



# Maintenabilité

---

## **Code Smells**

- Nombre de code smells, de nouveaux par sévérité, par statut

## **Dette technique :**

- Effort nécessaire pour corriger tous les problèmes de maintenabilité (en minutes).

## **Rapport dette/développement :**

- Rapport entre le coût de développement et le coût pour fixer les problèmes de maintenance (Le coût d'une ligne de développement est estimé à 0,06 jour)

## **Taux de maintenabilité :** Indicateur en fonction du rapport précédent

- $A < 0,05$  /  $B < 0,1$  /  $C < 0,2$  /  $D < 0,5$  /  $E > 0,5$



# Couverture des tests

---

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

- Les lignes
- Les chemins ou conditions

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



# Métriques

---

## **Couverture des lignes :**

- Lignes à couvrir
- Lignes couvertes
- Pourcentage

## **Couverture des conditions :**

- Conditions à couvrir
- Conditions couvertes
- Pourcentage





# Mesures globales

---

**Couverture** : Mélange entre la couverture de lignes et la couverture de conditions

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

- CT = Conditions ayant été évaluées à 'true' au moins une fois
- CF = Conditions ayant été évaluées à 'false' au moins une fois
- LC = Lignes à couvrir - lignes non couvertes
- B = Nombre total de conditions
- EL = Nombre total de lignes à couvrir

## Couverture des conditions

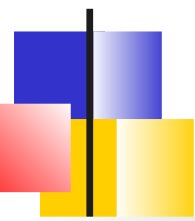
– **Couverture de condition =  $(CT + CF) / (2*B)$**

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

## Couverture des lignes

– **Taux de couverture de ligne =  $LC / EL$**

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



# Rapport d'exécution des tests

---

Sonar remonte également un rapport sur l'exécution des tests unitaires :

- Nombre de tests unitaires
- Nombre de tests unitaires ignorés
- Nombre de tests ayant échoués (Erreur d'assertion)
- Nombre de tests en erreur (Exception).
- Densité de réussite
- Durée des tests
- Nombre de conditions non couvertes par les tests
- Lignes non couvertes



# Qualité avec SonarQube

---

Offre et architecture

Concepts Sonar

Calculs de métriques

**Mise en place et configuration**

Profils et portes qualité spécifique



# Utilisation en développement

---

Déployer à l'identique dans l'IDE (***SonarLint***) et en construction continue

- Ceci permet à tout développeur de savoir à l'avance, au fur et à mesure du développement, quel sera l'effet de ses modifications et si elles seront éventuellement rejetées par les outils d'analyse de code



# Utilisation en construction continue

---

*Sonar* est devenue la solution de facto pour l'intégration continue en particulier avec Jenkins.

L'approche d'intégration prend en général 2 formes :

- Utiliser l'intégration Sonar Maven
- Installer des plugins spécifiques Sonar dans Jenkins

Permet de construire un tableau de bord accessible depuis Jenkins

- Visualise l'état du projet à chaque build
- Conserve l'historique des résultats => Permettra d'affiner ses prédictions pour les futurs projets



# Hiérarchie de configuration

---

Les paramètres pour configurer l'analyse peuvent s'effectuer à plusieurs endroits :

- **Paramètres globaux** : Défini via l'interface, ils s'appliquent à tous les projets  
*Administration > Configuration > General Settings*
- **Paramètres projet** : Défini via l'interface, ils surchargent les paramètres globaux. Au niveau projet :  
*Administration > General Settings*
- **Paramètres d'un analyseur** : Défini dans un fichier configuration propre au scanner, surcharge ceux de l'UI
- **Paramètres de ligne de commande** : Ils surchargent tous les autres



# Paramètres obligatoires

---

## Serveur

***sonar.host.url*** L'URL du serveur  
(http://localhost:9000)

## Configuration projet

***sonar.projectKey*** : Un identifiant de projet. Avec Maven : <groupId>:<artifactId>.

***sonar.sources*** : Liste des répertoires contenant les sources séparés par des virgules. Par défaut, l'emplacement Maven



# Paramètres optionnels

---

## Identité projet

***sonar.projectName*** : Nom du projet affiché sur l'UI. Balise <name> avec Maven

***sonar.projectVersion*** : La version Balise <version> avec Maven.

## Authentification

***sonar.login*** : Le login ou le jeton d'authentification d'un utilisateur avec la permission Execute Analysis .

***sonar.password*** : Le mot de passe si on utilise un login

## Service web

***sonar.ws.timeout*** : Timeout pour les appels webservises durant l'analyse





# Configuration projet

***sonar.projectDescription*** : Description du projet Pas compatible avec Maven

***sonar.links.homepage*** : Page d'accueil du projet. Pas compatible avec Maven

***sonar.links.ci*** : Lien vers serveur d'intégration. Pas compatible avec Maven

***sonar.tests*** : Liste des répertoires contenant le source des tests. Pas compatible avec Maven

***sonar.language*** : Le langage du code source à analyser. Si non spécifié, une analyse multi-langage est effectuée

***sonar.projectBaseDir*** : Si la racine du projet ne correspond au répertoire de démarrage de l'analyse

***sonar.scm.provider*** : Permet d'indiquer explicitement le plugin SCM à utiliser



# Exclusions / Inclusions

---

***sonar.inclusions*** : Liste des gabarits de chemins à inclure dans l'analyse.

***sonar.exclusions*** : Liste des gabarits de chemins à exclure de l'analyse.

***sonar.coverage.exclusions*** : Liste des gabarits de chemins à exclure du calcul de la couverture de code

***sonar.test.exclusions*** : Liste des gabarits de chemins des fichiers de test à exclure de l'analyse.

***sonar.test.inclusions*** : Liste des gabarits de chemins des fichiers de test à inclure dans l'analyse.

***sonar.issue.ignore.allfile*** : Les fichiers contenant cette expression régulière seront ignorés par l'analyse.

***sonar.cpd.exclusions*** : Liste de gabarits de chemins à exclure de la détection de duplication

***sonar.cpd.\${language}.minimumtokens*** : Seuil de détection de duplication en mots. Par défaut : 100

***sonar.cpd.\${language}.minimumLines*** : Seuil de détection de duplication en lignes. Par défaut : 10



# Gabarits

---

Les gabarits présents dans les exclusions/inclusions sont relatifs à la racine du projet.

Ils supportent les caractères suivants :

- \* zéro ou plusieurs caractères
- \*\* zéro ou plusieurs répertoires
- ? un seul caractère

Exemple :

```
sonar.exclusions=**/*Bean.java,**/*DT0.java
```



# Couverture des tests

---

Pour calculer la couverture des tests, SonarQube utilise les rapports d'outils tiers comme *jacoco*

Pour Java, le plugin est indiqué via la propriété :

**`sonar.java.codeCoveragePlugin`**

Il faut en général indiquer l'emplacement des rapports générés par le plugin



# Traces

---

***sonar.log.level*** : Niveau de trace. Par défaut INFO. Plus verbeux DEBUG et TRACE

***sonar.verbose*** : Idem que DEBUG avec en plus les variables d'environnement du client

***sonar.showProfiling*** : Information de profiling. Génère également un fichier *profiler.xml*

***sonar.scanner.dumpToFile*** : Indique un fichier où sont écrites toutes les propriétés passées au scanner



# Qualité avec SonarQube

---

Offre et architecture

Concepts Sonar

Calculs de métriques

Mise en place et configuration

**Profils et portes qualité spécifique**



# Axes de configuration

---

La configuration d'un projet consiste à spécifier :

- La Leak Period : Typiquement la version précédente du projet, sinon 30 jours
- La clé du projet : Permet d'appliquer des permissions par défaut
- Les portes qualité
- Les profils qualité
- Les exclusions



# Profils qualité

---

Même si les profils « *Sonar Way* » permettent de démarrer rapidement, il n'est pas recommandé de les utiliser dans les projets car ils ne sont pas éditables et ne peuvent donc pas être personnalisés

=> Il est donc recommandé de créer un nouveau profil et de copier ou hériter du profil « *Sonar Way* » correspondant au langage





# Héritage des profils

---

Les profils peuvent avoir des relations d'héritage en définissant un profil parent.

Les règles héritées peuvent être surchargées en modifiant le niveau de sévérité

Les mises à jour du profil parent sont répercutés sur les enfants.

- Il peut être intéressant de définir des profils racines pour chaque technologie de l'entreprise
- Hériter du profil *SonarWay* permet de profiter des mises à jour des règles lors d'un upgrade.



# Changement dans les règles

---

A tout moment, il est possible de comparer 2 profils et en particulier son profil avec le profil « Sonar way ».

***Page Quality Profiles > Selection du profil 1 > puis Actions > Compare et sélectionner le profil 2 .***

- Cela permet de détecter de nouvelles règles ou les règles dépréciées.

Une recherche en activant le filtre *Available Since* permet également d'isoler les nouvelles règles

Le filtre *Deprecated Rules* isole les règles dépréciées et permet d'accéder aux profils qui les utilise.



# Exclusions de règles

---

En dehors de SonarWay, les règles des profils qualité peuvent être activées/désactivées.

SonarQube donne la possibilité de finement configurer ce qu'il va être analysé, en excluant des fichiers/répertoires:

- Complètement
- De la détection de transgression (pour toutes les règles ou certaines règles)
- De la détection de duplications
- Du calcul de la couverture de test



# Ignorer des fichiers

---

4 façons d'ignorer complètement des fichiers :

- Utiliser ***sonar.sources*** pour limiter les fichiers analysés
- Définir les suffixes autorisés  
***Administration > General Settings > [Language]***
- ***source.exclusion*** et ***source.test.exclusion***  
pour exclure des fichiers ou des classes de test
- ***source.inclusion*** et ***source.test.inclusion***  
pour inclure des fichiers ou des classes de test



# Ignorer des issues

---

Il est possible d'ignorer des issues dans certains cas seulement . (Fichiers générés par exemple)

***Administration > General Settings > Analysis Scope > Issues***

- Ignorer des fichiers dont le contenu contient une expression régulière
- Ignorer des blocs de texte. Il faut alors spécifier les marqueurs de début et de fin avec des expressions régulières



# Ignorer avec des critères multiples

---

Ce menu permet de désigner la règle à exclure et les fichiers concernés.

Par ex : *Ignorer la règle "cpp:Union" sur tous les fichiers du répertoire object*

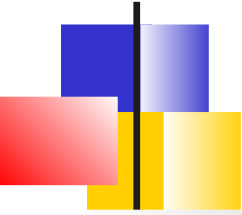
KEY = cpp:Union

PATH = object/\*\*/\*

Cela est également possible par Maven :

```
<sonar.issue.ignore.multicriteria>toto</sonar.issue.ignore.multicriteria>
<sonar.issue.ignore.multicriteria.toto.resourceKey>object/**/*</
  sonar.issue.ignore.multicriteria.toto.resourceKey>
<sonar.issue.ignore.multicriteria.toto.ruleKey>cpp:Union</
  sonar.issue.ignore.multicriteria.toto.ruleKey>
```

Par l'interface, il est également possible de confirmer les règles à appliquer pour un ensemble de fichiers



# Ignorer les duplications ou de la couverture de test

---

Pour ignorer des fichiers de la détection de duplication :

***Administration > General Settings > Analysis Scope > Duplications***

De la couverture de test :

***Administration > General Settings > Analysis Scope > Code Coverage***



# Journal projet

---

Lorsque SonarQube détecte qu'une analyse a été démarré avec un profil différent, un événement **Quality Profile** est ajouté au journal d'événements du projet.

**Quality Profiles > [ Profile Name ] > Changelog.**

Les utilisateurs avec des privilèges d'administration de profil qualité sont notifiés chaque fois qu'un profil prédéfini est mis à jour.





# Porte qualité

---

La porte qualité est le meilleur moyen pour améliorer la qualité dans un organisation.

Elle répond à la question : Est-ce que cette release peut aller en production ?

Elle est constituée d'un ensemble de conditions booléennes basées sur des seuils de mesure

Les portes qualité doivent être adaptées en fonction des projets.



# Sonar Way

---

Sonar fournit une porte qualité par défaut en mode read-only

Elle s'adresse à stopper l'hémorragie qualité en se basant sur les « Leak periods »

Il faut l'adapter par rapport à sa situation et s'en servir comme objectif à moyen terme



# Condition

---

Chaque condition d'une porte qualité est une combinaison :

- D'une mesure
- D'une période : Value (to date) or Leak  
(Valeur différentielle sur la *Leak period*)
- D'un opérateur de comparaison
- D'une valeur d'avertissement
- D'une valeur d'erreur



# Jenkins

---

## **Introduction / Installation**

Architecture maître/esclaves

Configuration

Job Maven

Intégration Sonar / Nexus



# Introduction

---

**Jenkins**, à l'origine Hudson, est un outil d'intégration continue écrit en Java.

Utilisable et utilisé pour des projets très variés en terme de technologie .NET, Ruby, Groovy, Grails, PHP ... et Java

C'est sûrement l'outil de CI le plus répandu

Supporté par la société **CloudBees**



# Atouts

---

- ✓ Facilité d'installation
- ✓ Interface web intuitive
- ✓ Prise en main rapide
- ✓ Très extensible et adaptable à des besoins spécifiques (Nombreux plugins opensource)
- ✓ Communauté très large, dynamique et réactive (blogs, twitter, IRC, mailing list),
- ✓ Release quasi-hebdomadaires ou LTS release (Long Term Support)



# Exécution

---

Jenkins est un **programme exécutable Java** qui intègre un serveur Web intégré

- Il peut également être déployé comme *.war* sur un autre serveur d'application : Tomcat, Glassfish, etc.

Les distributions typiques sont :

- Image Docker
- Packages natif Linux/Mac Os
- Application Java Standalone
- Service Windows



# Releases

---

Jenkins propose 2 types de releases :

- **LTS (Long Term Support) :**

Ce sont des release qui intègrent les développements les plus stables.

Elles sont choisies toutes les 12 semaines, à partir des dernières releases effectuées et déjà bien testées.

Elles n'intègrent que les corrections des bugs majeurs

- **Weekly Release :**

Toutes les semaines, elles intègrent les tous derniers développements





# Installation manuelle

---

Récupérer la distribution et la placer dans le répertoire de votre choix :

- Linux : `/usr/local/jenkins` ou `/opt/jenkins`
- Windows : `C:\Outils\Jenkins`

Pour démarrer, exécuter :

```
$ java -jar jenkins.war
```



# Structure de répertoires

---

**.jenkins** : Le répertoire home par défaut (compatibilité lors de mise à jour Jenkins)

**fingerprints** : Traces des empreintes des artefacts générés lors des build.

**jobs** : Configuration des jobs gérés par Jenkins ainsi que les artefacts générés par les builds

**plugins** : Les plugins installés .

**updates** : Répertoire interne à Jenkins stockant les plugins disponibles

**userContent** : Répertoire pour déposer son propre contenu  
(<http://myserver/hudson/userContent> ou  
<http://myserver/userContent>).

**users** : Les utilisateurs Jenkins si l'annuaire Jenkins interne est utilisé

**war** : L'application web Jenkins décompressée



# Structure du répertoire jobs

---

Le répertoire *jobs* contient un répertoire par projet Jenkins

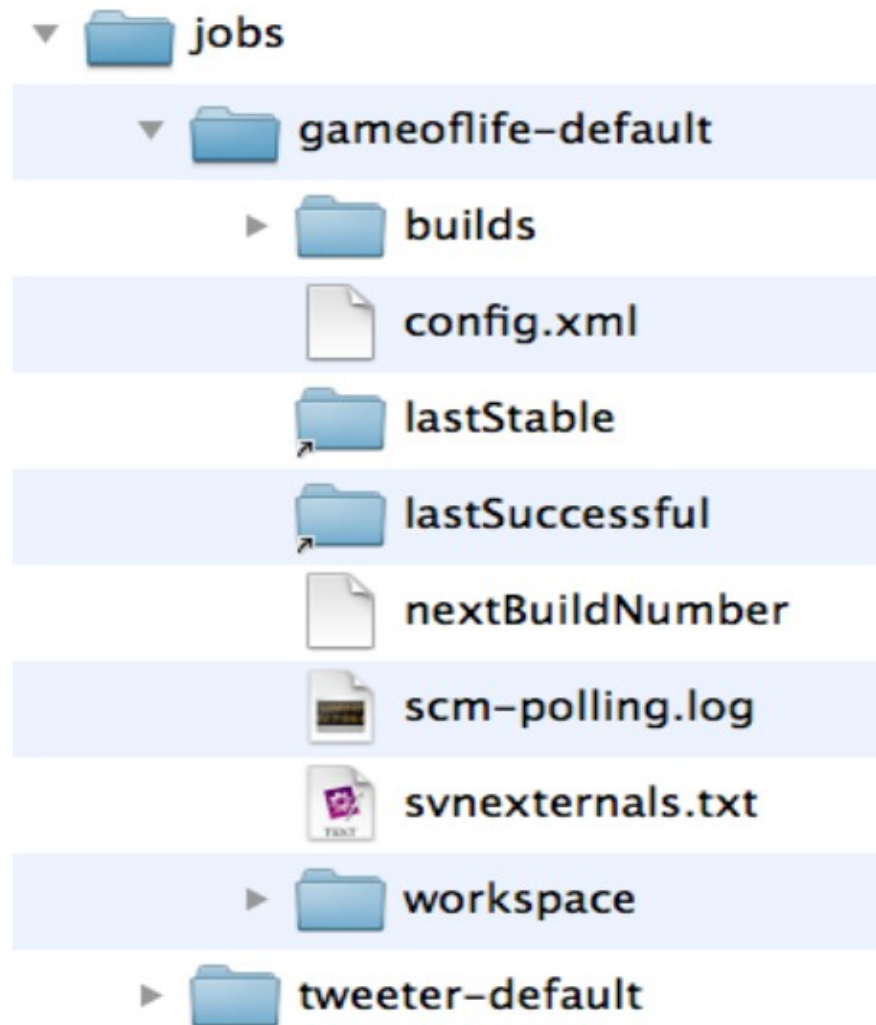
Chaque projet contient lui-même 2 sous répertoires :

- ***builds*** : Historique des builds
- ***workspace*** : Les sources du projet + les fichiers générés par le build



# Exemple structure jobs

---





# Répertoire build

---

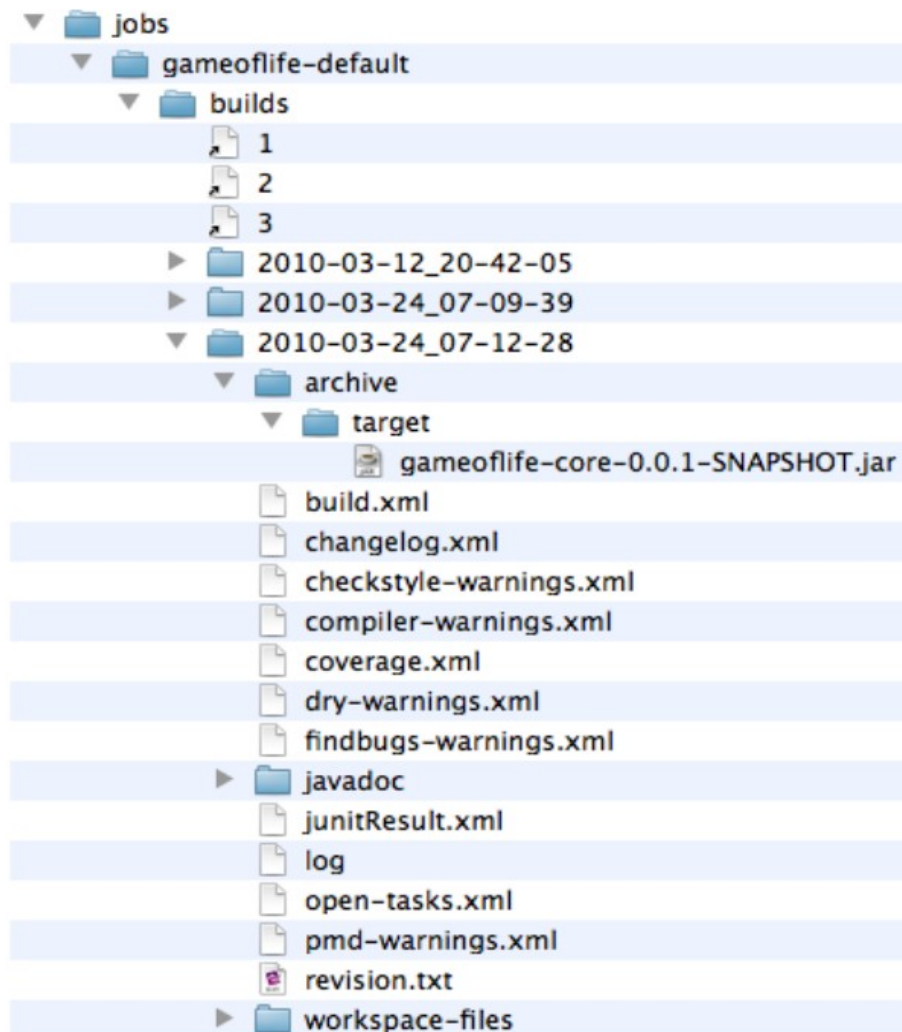
Jenkins stocke l'historique et les artefacts de chaque build dans un répertoire libellé avec un timestamp (“2010-03-12\_20-42-05”).

Des liens symboliques libellés avec les numéros de build pointent également sur ces répertoires

Chaque répertoire de build contient le fichier de log, le numéro de révision du SCM, les changements qui ont déclenché le build, les binaires et fichiers générés, et toutes les données ou métriques configurées pour ce build



# Exemple répertoire builds





# *JENKINS\_HOME*

---

Au démarrage, Jenkins recherche son répertoire *JENKINS\_HOME* dans cet ordre :

1. Une entrée dans l'environnement JNDI
2. Une propriété système
3. Une variable d'environnement
4. Le répertoire *.jenkins* dans le répertoire de l'utilisateur

Le répertoire *JENKINS\_HOME* contient l'intégralité des données du serveur



# Étapes post-installation

---

Quelques étapes terminent  
l'installation :

- Déverrouillage de Jenkins (via un mot de passe généré)
- Création d'un administrateur
- Installation de plugins. L'assistant propose d'installer les plugins les plus répandus.





# Jenkins

---

Introduction / Installation  
**Architecture maître/esclaves**  
Configuration  
Job Maven  
Intégration Sonar / Nexus



# 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 **conteneurs** qui sont alors provisionnés/exécutés lors du build et qui disparaissent ensuite

# Netflix 2012

1 master avec 700  
utilisateurs

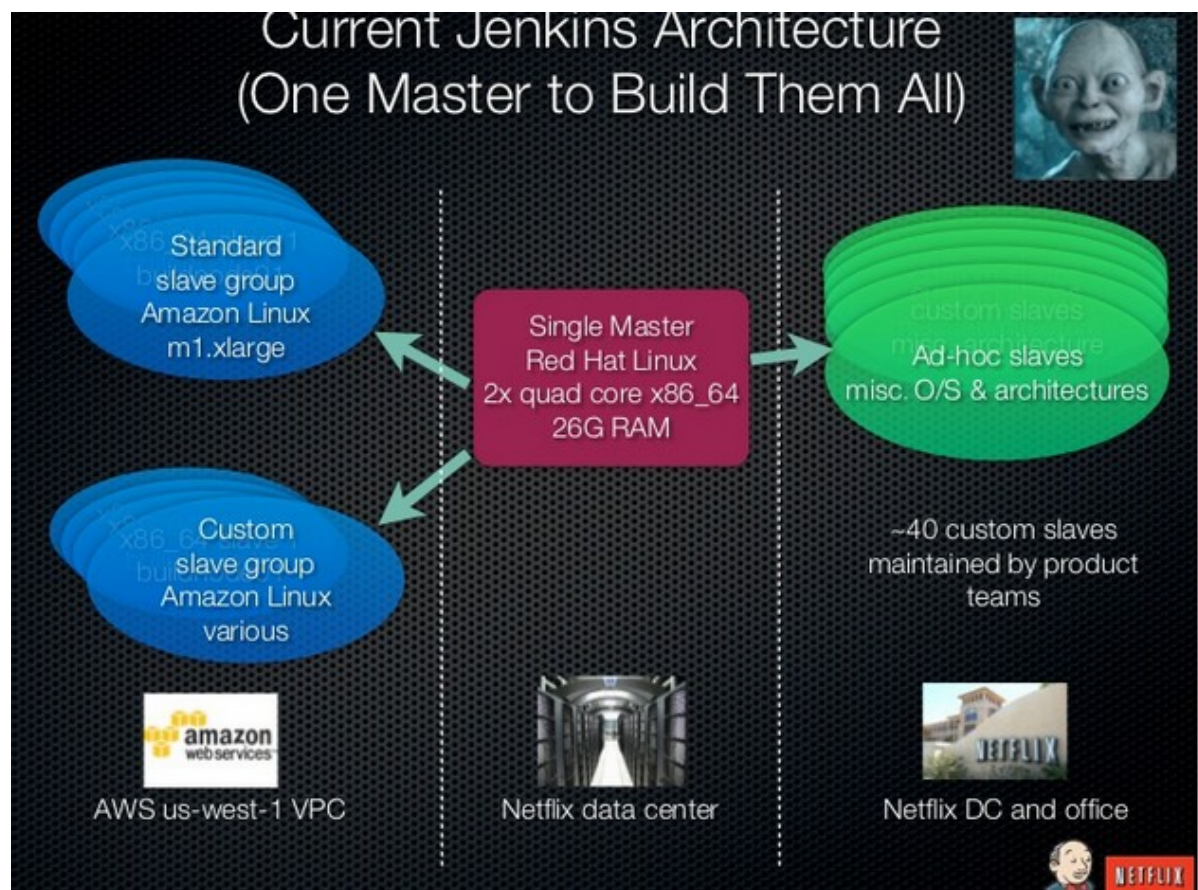
1,600 jobs :

- 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





# Jenkins

---

Introduction / Installation  
Architecture maître/esclaves  
**Configuration**  
Job Maven  
Intégration Sonar / Nexus



# Point d'entrée

---

Le point d'entrée est la page web « ***Administrer 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, ...
- Sécurité globale : Annuaire utilisateur et permissions
- 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 utilisateurs : Ajouter ou supprimer les utilisateurs



# Configuration système

---

La constitution de la page dépend des plugins utilisés, Citons :

- L'emplacement Jenkins : URL et mail de l'administrateur
- Adresse du serveur de mail pour notifier les utilisateurs (Plugin Jenkins Mailer Plugin)
- Emplacement du dépôt local Maven (Maven Integration Plugin)
- ....



# Configuration notification email

La technique principale de notification utilisée par Jenkins se base sur les emails.

Typiquement, Jenkins envoie un email au développeur ayant committé les changements qui ont provoqué l'échec du build

**E-mail Notification**

SMTP server	<input type="text" value="smtp.plbformation.com"/>
Default user e-mail suffix	<input type="text"/>
<input checked="" type="checkbox"/> Use SMTP Authentication	
User Name	<input type="text" value="stageojen@plbformation.com"/>
Password	<input type="password" value="....."/>
Use SSL	<input type="checkbox"/>
SMTP Port	<input type="text"/>
Reply-To Address	<input type="text"/>





# Configuration globale des outils

---

Certains outils utilisés lors des builds peuvent être configurés dans cette page.

- Si l'outil est installé sur la machine exécutant le build, il faut spécifier l'emplacement du répertoire HOME
- Sinon, il faut demander à Jenkins de l'installer automatiquement (répertoire *`$JENKINS_HOME/tools`*)

Différentes versions d'un outil peuvent être configurées

# Configuration outils : JDKs

## JDK

JDK installations

Add JDK

JDK

Name

JDK8

JAVA\_HOME

/usr/lib/jvm/java-8-openjdk-amd64

☐ Install automatically



Delete JDK

JDK

Name

JDK11

JAVA\_HOME

/usr/lib/jvm/java-11-openjdk-amd64

☐ Install automatically



Save

Apply



# Configuration Maven

---

Jenkins peut installer automatiquement une version spécifique de Maven ou on peut indiquer le home d'une installation locale.

Différentes versions peuvent être configurées et utilisées

L'installation automatique s'effectue à partir du site Apache ou d'une URL fournie

Possibilité de fournir les options à Maven via *MAVEN\_OPTS*



# Configuration Git/SVN

---

Jenkins doit se connecter à des SCM  
(SVN, GIT, ...)

Pour cela, il doit avoir accès à un client.

En général, le client est pré-installé sur  
les machines de build



# Gestion des plugins

---

Une page spécifique est dédiée à la gestion des plugins.

L'instance du serveur se connecte au dépôt

***[updates.jenkins-ci.org](https://updates.jenkins-ci.org)***

Il permet de :

- Voir les plugins installés
- Voir les plugins disponibles
- Voir les mise à jour disponibles

L'installation ne nécessite en général pas de redémarrage et des dépendances existent entre les plugins



# Jenkins

---

Introduction / Installation  
Architecture maître/esclaves  
Configuration  
**Job Maven**  
Intégration Sonar / Nexus



# Types de jobs et Outils de build

---

Par défaut, Jenkins propose un seul type de job

- FreeStyle : Script shell ou *.bat* Windows

En fonction des plugins installés, d'autres types de job peuvent être mis en place : Maven, Gradle, Gant, Grails, MSBuild, ...

# Gestion des modules

Jenkins comprend la structure en module d'un projet multi-modules Maven et ajoute des entrées dans l'interface pour chaque module

=> Possibilité de visualiser la structure en modules, d'accéder au détail d'un module, de lancer le build d'un module particulier en isolation, ou de configurer spécifiquement un module



The screenshot shows the Jenkins Hudson interface for a project named 'game-of-life'. The left sidebar contains navigation links: Back to Dashboard, Status, Changes, Workspace, Build Now, Delete Project, Configure, Modules (highlighted), Promotion Status, and Subversion Polling Log. The main content area is titled 'Modules' and displays a table with the following data:

S	W	Job ↓	Last Success	Last Failure	Last Duration	
		<a href="#">gameoflife</a>	21 hr ( <a href="#">#73</a> )	1 mo 13 days ( <a href="#">#41</a> )	1.7 sec	
		<a href="#">gameoflife-core</a>	21 hr ( <a href="#">#73</a> )	2 days 12 hr ( <a href="#">#68</a> )	11 sec	
		<a href="#">gameoflife-web</a>	21 hr ( <a href="#">#73</a> )	2 days 12 hr ( <a href="#">#68</a> )	21 sec	
		<a href="#">gameoflife-webservice</a>	21 hr ( <a href="#">#73</a> )	2 days 12 hr ( <a href="#">#68</a> )	0.61 sec	
		<a href="#">gameoflife-cli</a>	21 hr ( <a href="#">#73</a> )	2 days 12 hr ( <a href="#">#68</a> )	0.32 sec	





# Sections de configuration d'un job Maven

---

La configuration d'un job consiste en

- Des configurations générales : Nom, conservation des vieux builds, ...
- L'association à un SCM
- La définition des déclencheurs de build
- Les étapes avant le build : Initialisation, etc.
- Les étapes du build : Appel d'une cible Maven
- Les étapes après le build : Fermeture des ressources, ...
- Actions à la suite du build : Archivage d'artefacts, Publication des résultats, démarrage d'autre build, ...



# Gestion de l'historique des builds

L'option « ***Supprimer les anciens build*** » permet de limiter le nombre de builds conservés dans l'historique

- On peut indiquer un nombre de jours ou un nombre de builds
- On peut également conserver pour toujours un build particulier

Par exemple, garder tous les rapports sur la qualité du code et ne garder que les derniers builds qui déploient sur un serveur de test

Jenkins ne détruit jamais les derniers builds stables

=> Si on a spécifié de garder les 2 derniers build et que les 2 derniers build ont échoué. Jenkins garde les 2 derniers builds + le dernier build stable



# Déclencheurs

---

Il y a plusieurs façons de déclencher un build Maven :

- A chaque fois qu'une dépendance snapshot est construite
- A la fin d'un autre build
- Périodiquement
- En surveillant le SCM, et en déclenchant le build si un changement est détecté
- Via un hook déclenché par GitHub ou autre
- Manuellement



# Déclenchement via Sanapshot

---

Si cette option est sélectionnée, Jenkins examine le *pom.xml* afin de vérifier si aucun autre job construit une version SNAPSHOT d'une des dépendances.

Si un job met à jour une version SNAPSHOT le projet est reconstruit.

L'option est cochée par défaut



# Scrutation du SCM

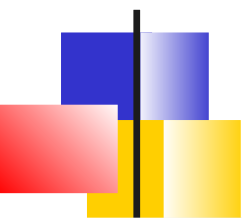
---

Le polling du SCM consiste à vérifier à intervalle régulier si des changements sont survenus et démarrer un build si besoin.

Le polling est une opération légère. Plus fréquemment il est effectué, plus rapidement le feedback sera réalisé

- Il faut cependant faire un compromis entre la fréquence des commits et la capacité à enchaîner les builds
- La surcharge réseau peu devenir un problème si de nombreux jobs utilisent cette technique

Le polling se configure également via la syntaxe *cron*



# Déclenchement à distance

---

Un autre approche consiste à déléguer directement au SCM le rôle de déclencher le build

Par exemple, avec Subversion ou Git, écrire un script accédant au serveur Jenkins et s'exécutant après un commit (hook-script). Le script a alors 2 alternatives :

- Déclencher directement le build Jenkins :  
`http://SERVER/jenkins/job/PROJECTNAME/build`
- Déclencher la vérification Jenkins du SCM qui provoquera un build  
`http://SERVER/jenkins/job/PROJECTNAME/notifyCommit`



# Interactions avec le SCM

---

La plupart des jobs étant reliés à un SCM leur démarrage consiste en

- Effectuer un check-out complet du projet dans un espace de travail de jenkins
- Lancer le build (compilation, test unitaires, ...)



# Configuration Git

---

La configuration consiste à spécifier :

- L'URL du dépôt
- La branche à construire
- De nombreuses options additionnelles





# Options Maven dans un contexte de CI

---

- B, --batch-mode** : Pour éviter que Maven bloque le build. Pas de prompt pour les données d'entrée, les valeurs par défaut sont utilisées.
- U, --update-snapshots** : Force Maven à vérifier les mises à jour des dépendances. Cela garantit que le build est effectué avec les dernières versions des dépendances.
- Dsurefire.useFile=false** : Force maven à écrire les sorties de Junit sur la console plutôt que dans des fichiers texte. Ainsi, les tests en échec sont directement visibles dans le console du job. Les fichiers XML utilisés pour le reporting sont quand même générés.



# Configuration Build

---

Dans un job Maven, la configuration consiste à :

- Spécifier la version de Maven à utiliser
- Éventuellement, l'emplacement du *pom.xml* file
- Le ou les objectifs Maven
- Les options de la ligne de commande



# Configuration Build avancée

---

La configuration avancée permet de renseigner d'autres champs :

- **Emplacement** du *pom.xml* ( $\Leftrightarrow$  l'option `-f` ou `-file` de Maven).
- Les **propriétés** : propriétés passées à Maven via `-D`
- **Options** de la JVM
- Utilisation d'un **dépôt dédié**. ( $\Rightarrow$  build effectué en isolation)
- Par défaut, Jenkins archive tous les artefacts générés le build  
 $\Rightarrow$  Option « Désactive l'archivage automatique des artefacts »
- L'option "*Construction incrémentale*" est intéressante pour les gros projets multi-modules.  
Lorsqu'un changement est détecté dans un des modules du projet, seul ce module et les modules dépendants sont reconstruits
- L'option « *Construire les modules en parallèle* » est intéressante seulement si les modules ne sont pas dépendants



# Pre ou Post : Exécuter un shell

---

Il est possible d'exécuter une commande système spécifique ou d'exécuter un script (typiquement stocké dans le SCM)

- Le script est indiqué relativement à la racine du répertoire de travail
- Les scripts Shell sont exécutés avec l'option -ex. La sortie des scripts apparaît sur la console
- Si une commande retourne une valeur  $\neq 0$ , le build échoue

=> Ce type d'étapes rend (au minimum) votre build dépendant de l'OS et quelquefois de la configuration du serveur. Une autre alternative est d'utiliser un langage plus portable comme Groovy ou Gant



# Variables d'environnement Jenkins (1)

---

Des informations sur le job Jenkins peuvent être facilement récupérées et utilisées dans les scripts Ant ou *pom.xml* Maven :

**BUILD\_NUMBER** : N° de build.

**BUILD\_ID** : Un timestamp de la forme YYYY-MM-DD\_hh-mm-ss.

**JOB\_NAME** : Le nom du job

**BUILD\_TAG** : Identifiant du job de la forme jenkins-\${JOB\_NAME}-\${BUILD\_NUMBER}

**EXECUTOR\_NUMBER** : Un identifiant de l'exécuteur

**NODE\_NAME** : Le nom de l'esclave exécutant le build ou "" si le maître

**NODE\_LABELS** : La liste des libellés associés au nœud exécutant le build



# Variables d'environnement Jenkins (2)

---

**JAVA\_HOME** : Le home du JDK utilisé

**WORKSPACE** : Le chemin absolu de l'espace de travail

**HUDSON\_URL** : L'URL du serveur Jenkins

**JOB\_URL** : L'URL du job, par exemple

<http://ci.acme.com:8080/jenkins/game-of-life>.

**BUILD\_URL** : L'URL du build, par exemple

<http://ci.acme.com:8080/jenkins/game-of-life/20>.

**SVN\_REVISION** : La version courante SVN si applicable.

**CVS\_BRANCH** : La branche CVS ou "" si trunk



# Accès à partir des builds

---

Les variables d'environnement Jenkins sont accessibles via Maven avec le préfixe ou directement le nom de la variable :

```
<url>${JOB_URL}</url>
```

```
<url>${env.JOB_URL}</url>
```



# Actions « Post-build »

---

Lorsque le build est terminé, d'autres actions peuvent être enclenchées :

- Archiver les artefacts générés
- Créer des rapports sur l'exécution des tests
- Notifier l'équipe sur les résultats
- Démarrer un autre build





# Archivage des résultats

---

Un build construit des artefacts (Jar, war, javadoc, ...)

- Un job peut alors stocker un ou plusieurs artefacts, ne garder que la dernière copie ou toutes
- Il suffit d'indiquer les fichiers à archiver (les wildcards peuvent être utilisés)
- Possibilité d'exclure des répertoires

Dans le cas où on utilise un gestionnaire de repository d'entreprise comme Nexus ou Artifactory, il est préférable de déployer automatiquement les artefacts dans le repository pendant le job de build



# Déploiement dans un repository d'entreprise

---

Avec les jobs Maven, il est possible de déployer les artefacts dans un repository d'entreprise :

- Les artefacts sont déployés de façon atomique (tous ou aucun si le build échoue), ce qui peut présenter un avantage sur l'objectif *mvn deploy* dans un contexte multi-modules
- Le repository peut être indiqué via une URL ou mieux son *id* Maven défini dans *settings.xml*
- Des plugins existent pour Nexus, Artifactory, etc.



# Reporting sur les tests

---

Le format XML de JUnit contient des informations sur les tests échoués mais également combien de tests ont été exécutés et le temps d'exécution

Pour activer le reporting, sélectionner « Publier le rapport des tests JUnit » et indiquer l'emplacement des fichiers Junit générés

Le caractère '\*' peut être utilisé  
(`**/target/surefire-reports/*.xml`)

Jenkins agrège tous les fichiers trouvés en un seul rapport



# Jenkins

---

Introduction / Installation  
Architecture maître/esclaves  
Configuration  
Job Maven  
**Intégration Sonar / Nexus**



# Introduction

---

Dans un contexte Jenkins, Maven  
l'intégration avec Nexus et Sonar peut  
se faire :

- Soit en utilisant les plugins Maven
- Soit en utilisant des plugins Jenkins



# Plugin SonarQube Scanner

---

Le plugin ***SonarQube Scanner*** permet de centraliser la configuration de SonarQube dans Jenkins

L'analyse d'un projet peut alors être définie comme étape d'un build

Une fois l'analyse terminée, un statut de qualité est remonté sur l'UI Jenkins et un lien permet d'accéder aux tableaux de bord Sonar



# Configuration

---

La mise en place consiste à :

- Définir l' ou les instances de SonarQube

*Manage Jenkins → Configure System → SonarQube configuration → Add SonarQube*

- Définir les scanners à utiliser (Exemple Maven)

*Manage Jenkins → Configure Tools → SonarQube scanner*

- Configurer un projet pour utiliser le scanner



# Plugin Nexus

---

Le *Nexus Platform Plugin* permet d'ajouter des étapes de build :

- Publier vers un dépôt Nexus
- Associer des Tag
- Créer des Tag
- Déplacer des composants
- Supprimer des composants

Il nécessite une version commerciale de Nexus Repository Manager





# Configuration


La configuration du plugin consiste à indiquer l'instance de Nexus dans la configuration globale du système

**Sonatype Nexus**


Nexus Repository Manager Servers

**Nexus Repository Manager 3.x Server**

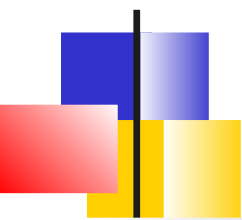
Display Name

Server ID  

Server URL

Credentials   Add

☐ Hide messages about what's coming to the Nexus Platform Plugin



# Pipelines CI/CD avec Jenkins

---

## **Notions de pipeline**

Ordonnancement legacy

Le plugin pipeline

Syntaxe déclarative



# Introduction

---

Le chaînage de jobs, i.e. pipeline, avec Jenkins peut se faire de 2 façons :

- Définir des **relations amont/aval** entre les jobs et utiliser des plugins legacy de visualisation de graphe, de gestion de pipeline, de fork, de join, ...(approche dépréciée mais fonctionnelle)
- Utiliser le nouveau plugin **Pipeline** et les plugins liés (Version Blue Ocean) permettant de définir des pipelines complexes en Groovy (Approche DevOps)

Quelque soit l'approche retenue, une problématique commune est le passage de paramètres ou d'artefacts entre les jobs



# Principes

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

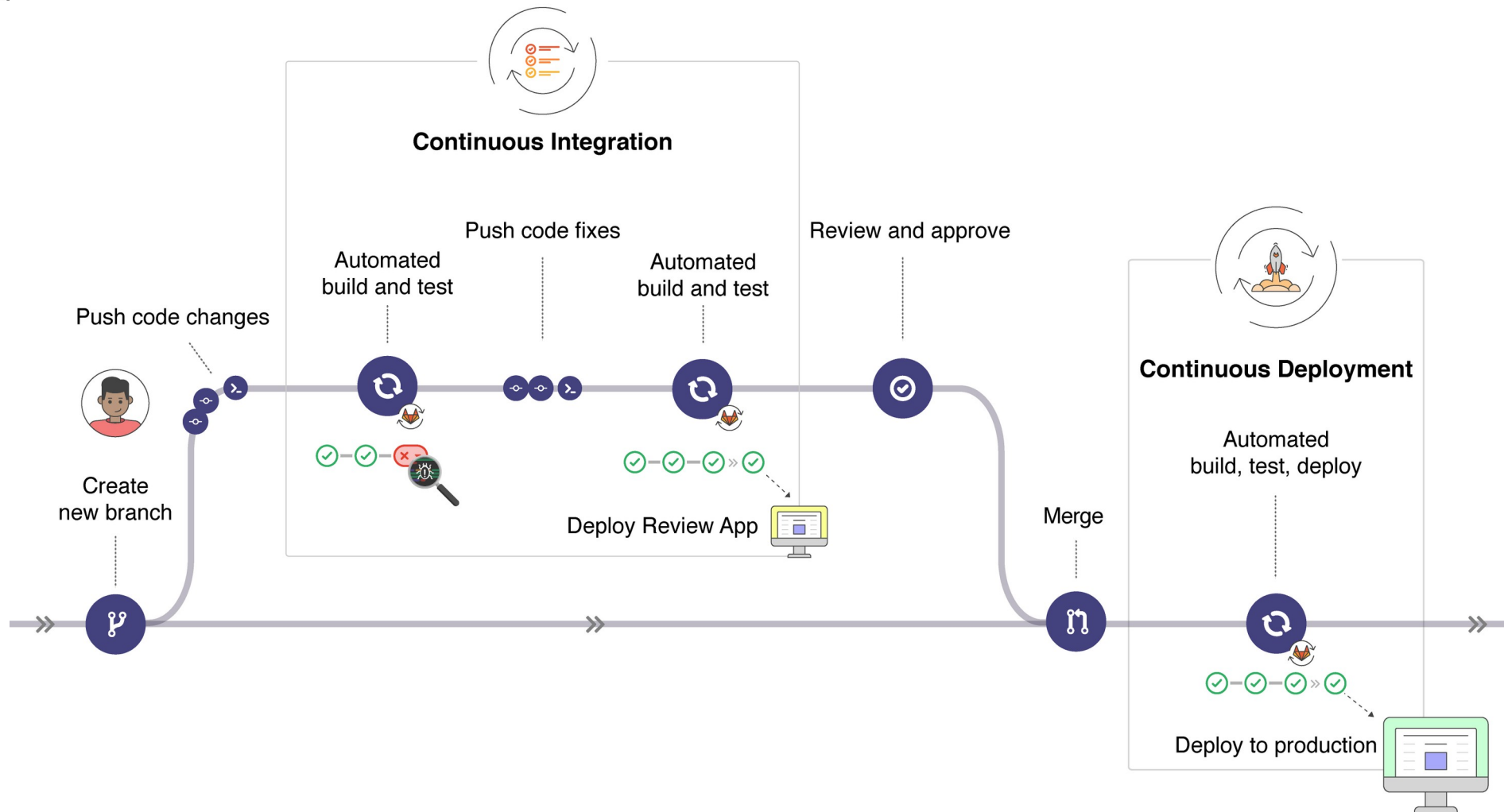
Par exemple : Build, Test, QA, Production

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

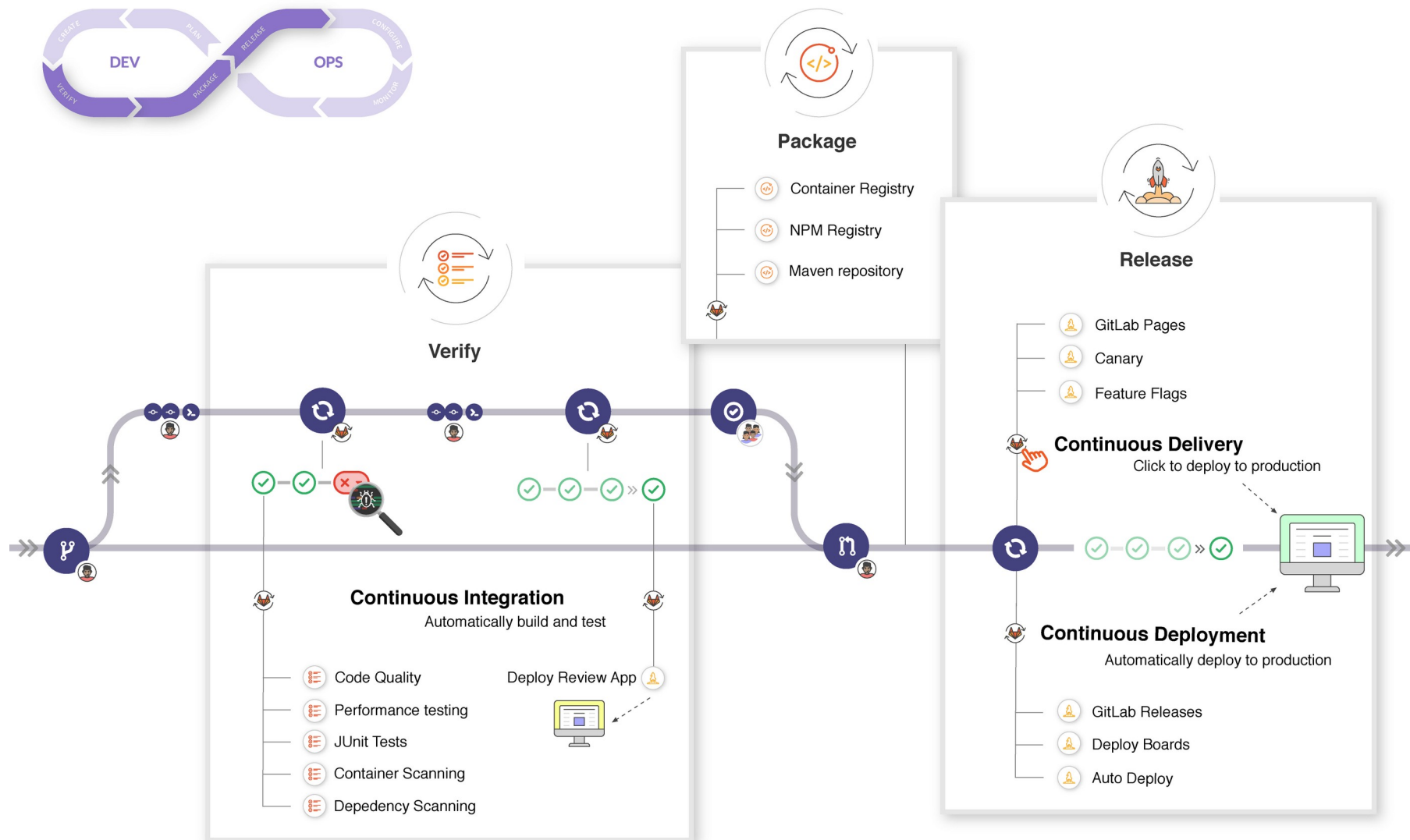
Les jobs exécutent des actions à partir du dépôt ou réutilisent des artefacts précédemment construits

Les jobs publient des rapports résultats

# Exemple Gitlab CI



# En plus détaillé





# Buils paramétrés

---

Des **paramètres** peuvent être configurés pour un job donné

Ils sont renseignés soit par l'utilisateur qui démarre le job manuellement soit récupérés d'un autre job du build

- Jenkins génère automatiquement l'interface utilisateur permettant de saisir les paramètres dans le cas d'un démarrage manuel.
- Il passe les paramètres au build appelé lorsque le passage de paramètres est automatique.

Les paramètres sont ensuite mis à disposition des jobs appelés via des variables d'environnement :

Shell : `$paramName`, Maven : `${env.paramName}`



# Configuration

---

La configuration consiste à cocher l'option "*Ce build a des paramètres*"

- Puis ajouter des paramètres un à un en indiquant leur type et leur valeur par défaut

=> Au démarrage manuel du build, Jenkins propose une page permettant de saisir les paramètres

On peut également déclencher le build par une URL

`http://jenkins.acme.org/job/myjob/buildWithParameters?  
PARAMETER=Value`



# Paramètres utilisateur

Jenkins » [parameterized-builds](#) » [unit-tests-build](#)

[Back to Dashboard](#)  
[Status](#)  
[Changes](#)  
[Workspace](#)  
[Build Now](#)  
[Delete Project](#)  
[Configure](#)  
[Dependency Graph](#)

Project name

Description

☐ Discard Old Builds

☒ This build is parameterized

**Choice**

Name

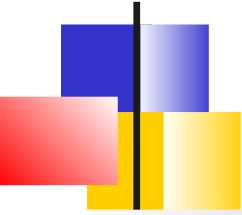
Choices

Description

**Build History** [\(trend\)](#)

#7	<a href="#">Feb 7, 2011 10:00:15 PM</a>	2KB
#6	<a href="#">Feb 7, 2011 10:00:07 PM</a>	2KB
#5	<a href="#">Feb 7, 2011 9:56:55 PM</a>	2KB
#4	<a href="#">Feb 7, 2011 9:14:42 PM</a>	2KB
#3	<a href="#">Feb 7, 2011 9:13:38 PM</a>	2KB
#2	<a href="#">Feb 7, 2011 9:13:12 PM</a>	2KB
#1	<a href="#">Feb 7, 2011 9:11:37 PM</a>	2KB

[for all](#) [for failures](#)



# Déclenchement de build paramétré

---

Le plugin « *Parameterized Trigger* » est nécessaire.

Il permet de configurer le passage de paramètres entre jobs. Les paramètres peuvent alors provenir :

- Des paramètres du build courant
- Des paramètres additionnels
- Des paramètres provenant d'un fichier *.properties*



# Passage de données entre jobs d'une pipeline

---

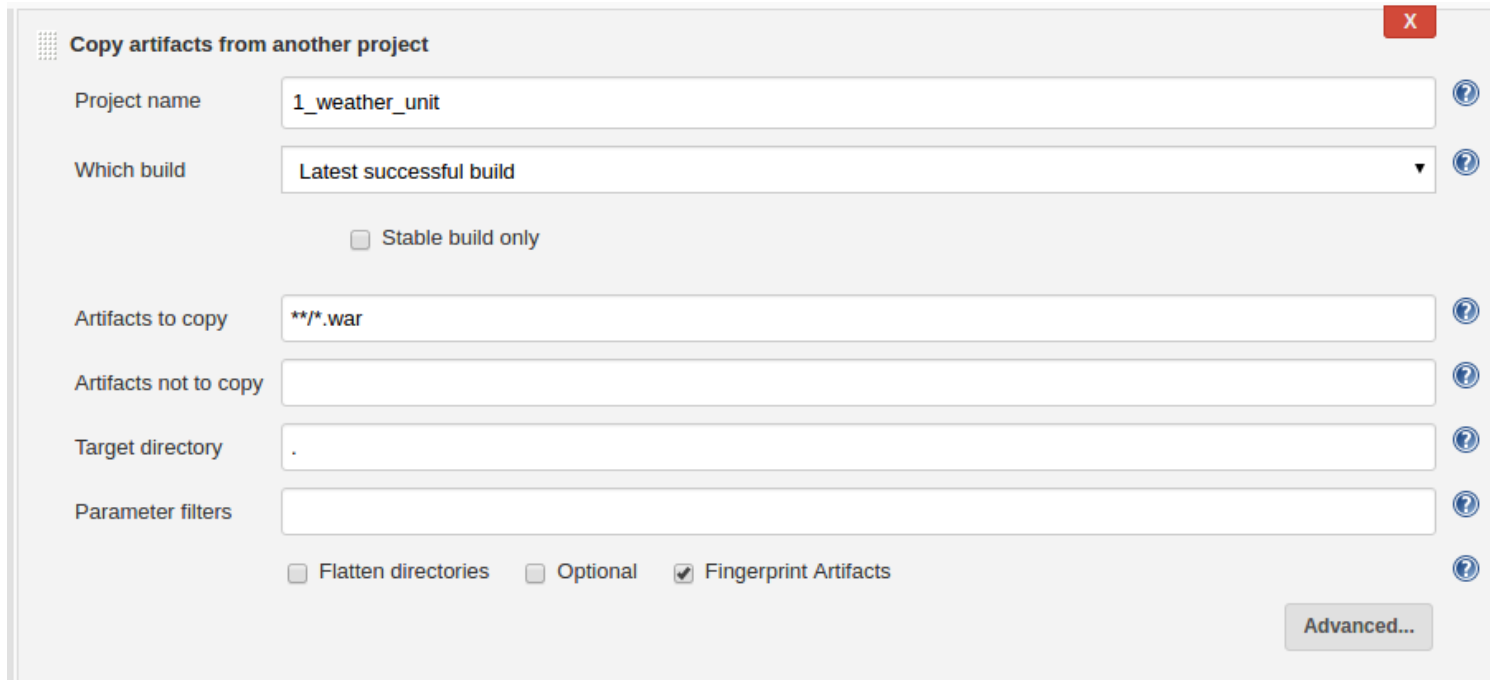
Avec le concept de pipeline, il est nécessaire de s'assurer que les jobs travaillent puissent s'échanger des données.

- Les **paramètres** peuvent être transférés entre jobs
- Le plugin « ***Copy Artifact*** » permet de copier des artefacts construits par un build précédent dans le build courant



# Copie d'artefacts

Une fois le plugin installé, il permet d'ajouter une nouvelle étape de build nommé “Copy artifacts from another project”



**Copy artifacts from another project** [X] [?]

Project name:  [?]

Which build:  [?]

☐ Stable build only

Artifacts to copy:  [?]

Artifacts not to copy:  [?]

Target directory:  [?]

Parameter filters:  [?]

☐ Flatten directories ☐ Optional ☒ Fingerprint Artifacts [?]

[Advanced...](#)



# Pipelines CI/CD avec Jenkins

---

Notions de pipeline

**Ordonnancement legacy**

Le plugin pipeline

Syntaxe déclarative



# Séquencement des jobs

---

Le séquencement de jobs peut s'effectuer via 2 champs de configuration symétriques

- **Du côté du projet aval :**

*“Ce qui déclenche le build → Construire à la suite d'autres projets”*

Dans ce cas, une option permet de démarrer même si le build en amont est instable

- **Du côté du projet amont**

*“Actions à la suite du build → Construire d'autres projet”*



# Parallélisation

---

Lorsqu'un job démarre, Jenkins l'assigne automatiquement au premier nœud disponible, ainsi il peut avoir autant de jobs en parallèle qu'il y a de nœuds disponibles

Lorsque l'on définit plusieurs builds à exécuter après la fin d'un build, ceux-ci sont exécutés en parallèle



# Plugins legacy d'ordonnancement

---

Le plugin « ***Dependency Graph View*** » analyse la configuration des jobs pour afficher un graphe de dépendance entre les jobs

Le plugin « ***Joins*** » permet de configurer des points de synchronisation entre jobs parallèles

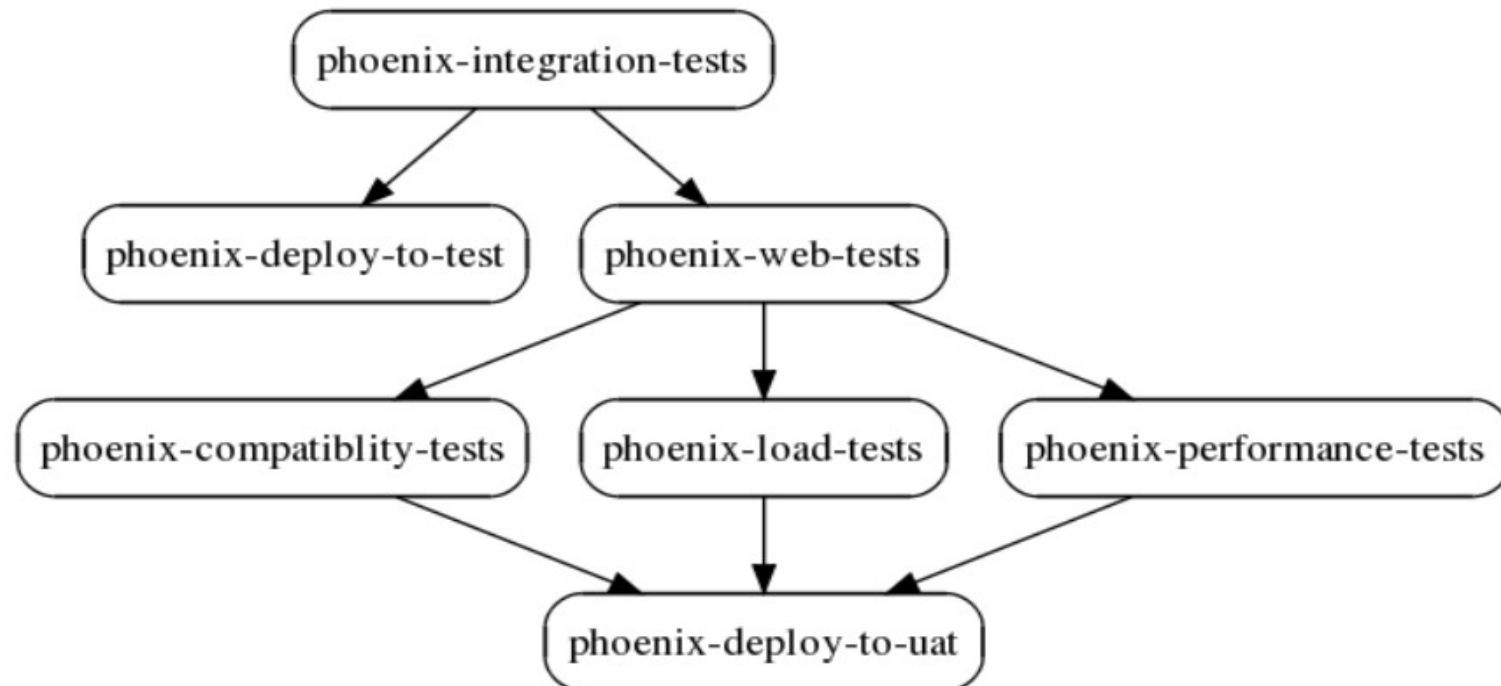
Le plugin « ***Locks et Latches*** » permet de positionner des verrous sur certaines ressources

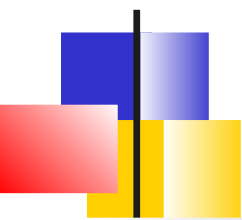




# Graphe de dépendance

---





# Pipelines CI/CD avec Jenkins

---

Notions de pipeline  
Ordonnancement legacy  
**Le plugin pipeline**  
Syntaxe déclarative



# Introduction

---

Jenkins Pipeline est une **suite de plugins** qui permettent d'implémenter et d'intégrer des pipelines de livraison continue avec Jenkins.

- Chaque changement committé dans le SCM provoque un processus complexe dont le but est une release.

Grâce à Pipeline, les processus de build sont modélisés **via du code et un langage spécifique (DSL)**



# Avantages de l'approche

---

Pipeline offre plusieurs avantages :

- Les pipelines implémentées par du code peuvent être gérées par le SCM  
=> Historique des révisions, adaptation au changement du projet
- Les Pipelines survivent au redémarrage de Jenkins
- Les Pipelines peuvent s'arrêter et attendre une approbation manuelle
- Le DSL supporte des pattern de workflow complexes (fork/join, boucle, ...)
- Le plugin permet des extensions et l'intégration de tâche spécifique à un build  
(Par exemple, interagir avec une solution de cloud)



# *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.



# 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*

```
#!/groovy

stage('Build') { // Phase
    node { // <=> job
        checkout scm // step
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app'
    }
}

stage('Test') {
    node('linux') { // Label de noeuds
        checkout scm
        try {
            unstash 'app' // Réutilisation des artefacts sauvegardés sous le nom app
            sh 'make check'
        } finally {
            junit '**/target/*.xml'
        }
    }
    node('windows') {
        checkout scm
        try {
            unstash 'app'
            bat 'make check'
        } finally {
            junit '**/target/*.xml'
        }
    }
}
```



# Définir une pipeline

---

Une Pipeline peut être créée :

- En saisissant un script directement dans l'interface utilisateur de Jenkins.
- En créant un fichier *Jenkinsfile* qui peut alors être enregistré dans le SCM.

Approche recommandée


Quelquesoit l'approche 2 syntaxes sont disponibles :

- Syntaxe déclarative
- Syntaxe script




# Example UI


**Enter an item name**  
  
» Required field




**Freestyle project**  
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.




**Pipeline**  
Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.




**External Job**  
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.




**Multi-configuration project**  
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.




**Folder**  
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



**GitHub Organization**  
Scans a GitHub organization (or user account) for all repositories matching some defined markers.



**Multibranch Pipeline**  
Creates a set of Pipeline projects according to detected branches in one SCM repository.



# Premier script

**Pipeline**

Definition Pipeline script ▼

Script

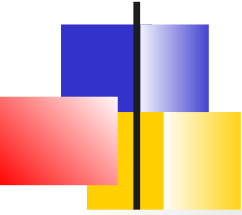
```
1 node {  
2   echo "Hello World"  
3 }
```

try sample Pipeline... ▼ ⓘ

☒ Use Groovy Sandbox ⓘ

[Pipeline Syntax](#)

**Save** Apply



# 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  
}
```

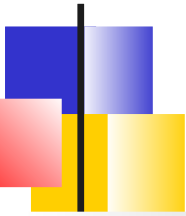


# Références Variables globales

---

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 : 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



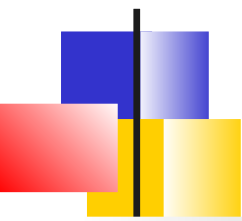


# Illustration

---

```
// 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....' }
}
```



# Pipelines CI/CD avec Jenkins

---

Notions de pipeline  
Ordonnancement legacy  
Le plugin pipeline  
**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

- ***stages*** : Une séquence d'une ou plusieurs sections *stage*
- ***stage*** : Un nom, des directives appliquées au stage et un bloc *steps*
- ***steps*** : Une ou plusieurs étapes à exécuter à l'intérieur d'une directive *stage*
- ***post*** : Steps à exécuter à la suite de la pipeline ou d'un stage



# Stages et steps

---

Les sections stages et steps ne font que délimiter un bloc

## ***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*



# *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



# Exemple

---

```
stage('Compile et tests') {
    agent any
    steps {
        echo 'Unit test et packaging'
        sh 'mvn -Dmaven.test.failure.ignore=true clean package'
    }
    post {
        always {
            junit '**/target/surefire-reports/*.xml'
        }
        success {
            archiveArtifacts artifacts: 'application/target/*.jar', followSymlinks: false
        }
        failure {
            mail bcc: '', body: 'http://localhost:8081/job/multi-branche/job/dev', cc: '',
from: '', replyTo: '', subject: 'Packaging failed', to: 'david.thibau@gmail.com'
        }
    }
}
```



# Directives

---

Les directives se placent généralement soit

- Sous le bloc pipeline  
=> Il s'applique à tous les stage de la pipeline
- Sous un bloc stage  
=> Il ne s'applique qu'au stage concerné





# Directive *agent*

---

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

- ***any*** : N'importe quel agent.
- ***label*** : Agent ayant été labellisé par l'administrateur
- ***node*** : Idem que label mais avec plus d'options
- ***docker, dockerfile*** : Image docker
- ***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



# Directive *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'  
}
```



# 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 crédits définis dans Jenkins.

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



# *options*

La directive ***options*** permet de configurer des options globale à la pipeline.

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



# Example : 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*





# *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



# Example

---

```
stage('Example Deploy') {  
  when {  
    allof {  
      branch 'production'  
      environment name: 'DEPLOY_TO',  
                   value: 'production'  
    }  
  }  
  steps {  
    echo 'Deploying'  
  }  
}
```



# Parallélisme

---

Avec la directive ***parallel***, les *stages* peuvent déclarer des *stages* imbriqués qui seront alors exécutés en parallèle

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



# Example

---

```
// 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 a une step ***script*** qui peut inclure un bloc dans la syntaxe script



# Steps

---

**// Exécuter un script**

sh, bat

**// Copier des artefacts**

```
copyArtifacts(projectName: 'downstream', selector: specific("$  
    {built.number}"));
```

**// Archive the build output artifacts.**

```
archiveArtifacts artifacts: 'output/*.txt', excludes: 'output/*.md'
```

**// Step basiques**

stash, unstash : Mettre de côté puis reprendre

dir, deleteDir, pwd

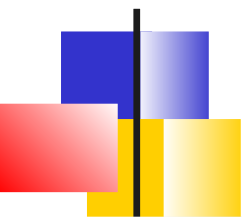
fileExists, writeFile, readFile

mail, git, build, error

sleep, timeout, waitUntil, retry

withEnv, credentials, tools

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



# Exemple plugin Nexus

```
freeStyleJob('NexusArtifactUploaderJob') {  
  steps {  
    nexusArtifactUploader {  
      nexusVersion('nexus2')  
      protocol('http')  
      nexusUrl('localhost:8080/nexus')  
      groupId('sp.sd')  
      version('2.4')  
      repository('NexusArtifactUploader')  
      credentialsId('44620c50-1589-4617-a677-7563985e46e1')  
      artifact {  
        artifactId('nexus-artifact-uploader')  
        type('jar')  
        classifier('debug')  
        file('nexus-artifact-uploader.jar')  
      }  
      artifact {  
        artifactId('nexus-artifact-uploader')  
        type('hpi')  
        classifier('debug')  
        file('nexus-artifact-uploader.hpi')  
      }  
    }  
  }  
}
```





# Exemple Sonar Script

---

```
stage("build & SonarQube analysis") {
    node {
        withSonarQubeEnv('My SonarQube Server') {
            sh 'mvn clean package sonar:sonar'
        }
    }
}

stage("Quality Gate"){
    timeout(time: 1, unit: 'HOURS') {
        def qg = waitForQualityGate()
        if (qg.status != 'OK') {
            error "Pipeline aborted due to quality gate failure: $
{qg.status}"
        }
    }
}
```