



CI/CD avec le serveur Jenkins

David THIBAU – 2021

david.thibau@gmail.com



Agenda

Introduction

- Plateforme de CI/CD

Mise en place

- Installation, UI,
- Configuration Système, Outils et plugins
- Configuration de Jobs free-style
- Architecture Maître/esclaves

Relations entre jobs

- Jobs paramétrés et multi-configuration
- Relations amont/aval
- Passage de données entre jobs

Pipelines

Vocabulaire et principe

Syntaxe déclarative et script

Utilisation de Docker

Librairies partagées

Administration

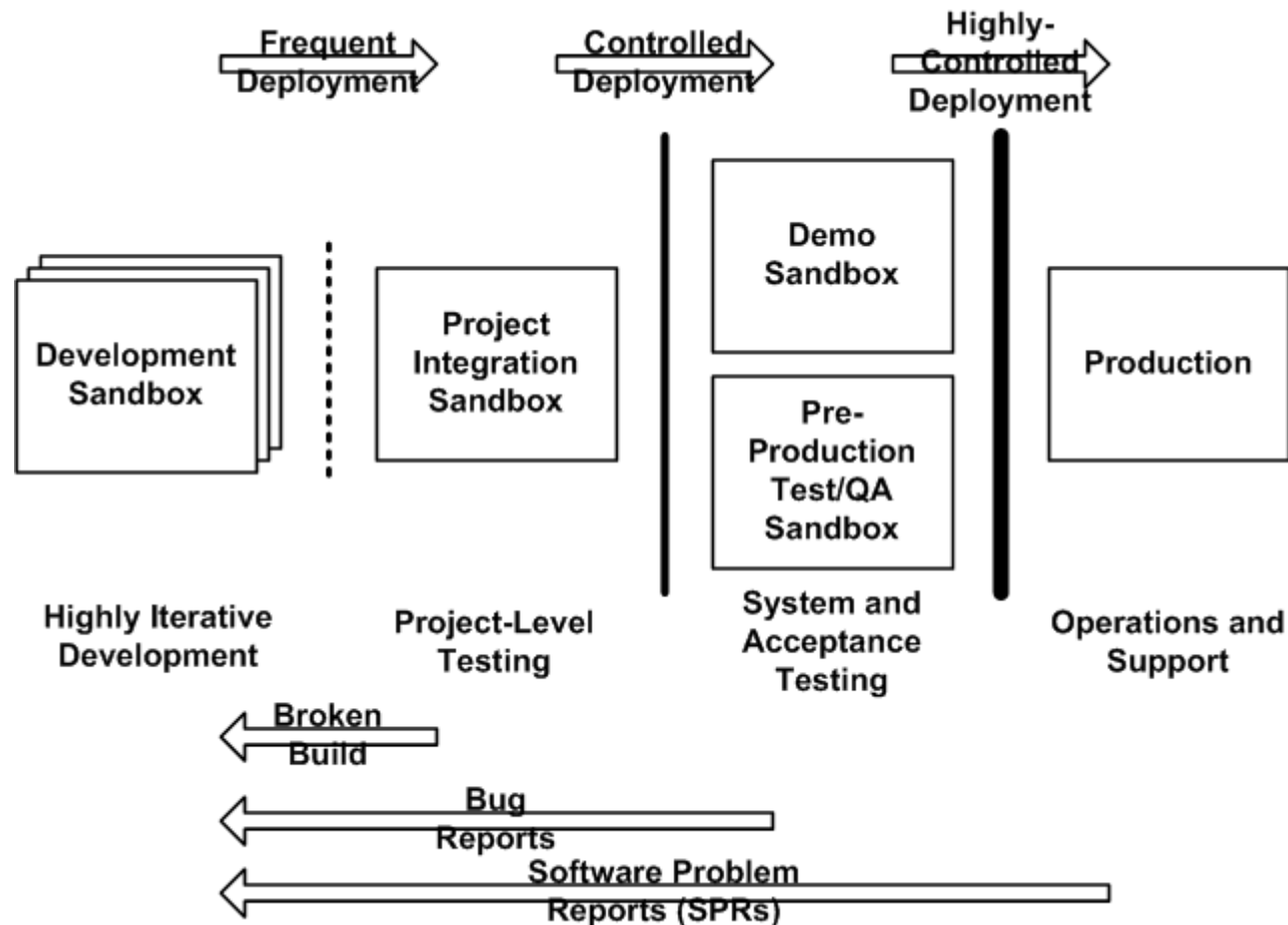
- Sécurité,
- Exploitation et monitoring
- Jenkins CLI et API Rest

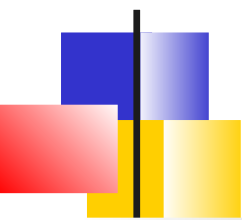


Introduction

DevOps, CI/CD, PIC

Environnements et fréquence de déploiement





Avant l'intégration continue

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

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

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

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



Plateforme d'intégration continue

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

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

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



Build is tests !

La construction de l'application consiste à principalement à :

- Packager le code source dans un format exécutable qui peut être automatiquement déployé
- Exécuter toutes les vérifications automatique permettant d'avoir confiance dans l'artefact généré

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

En fonction du résultat des tests et de la confiance qu'on leur accorde, chaque itération de création de valeur logicielle pourra être poussée dans un des environnements (intégration, QA, production)



Outil de communication et de motivation

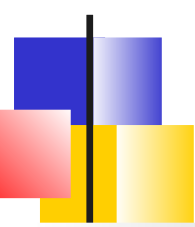
La PIC permet également de publier les résultats des builds (les résultats des tests et analyse):

- Nombre de tests exécutés, succès/échecs
- Couverture des tests
- Métriques Qualité
- Performance : Temps de réponse/débit
- Documentation produit du code source
-

=> Donne de la confiance dans la robustesse du code développé, réduction des coûts de maintenance.

=> Métriques qualité visibles aussi bien par les fonctionnels que par les développeurs

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



Intégration continue et agilité

L'**intégration continue** met automatiquement à disposition sur une plateforme d'intégration l'application en cours de développement

- Dans les méthodes agiles, c'est une nécessité. Les fonctionnels et les développeurs peuvent alors arbitrer les choix fonctionnels en se basant sur du concret.



Livraison continue

Dans une philosophie DevOps, la **livraison continue (Continuous Delivery)** consiste à essayer de produire une release à chaque modification du code source

Produire une release implique :

- Des tests automatiques poussés permettant d'avoir un très haut niveau de confiance dans l'artefact produit
- Tagger l'artefact et le stocker dans un dépôt

Les release peuvent alors être déployés en QA et des tests manuels peuvent être effectués afin de décider d'une mise en production éventuelle



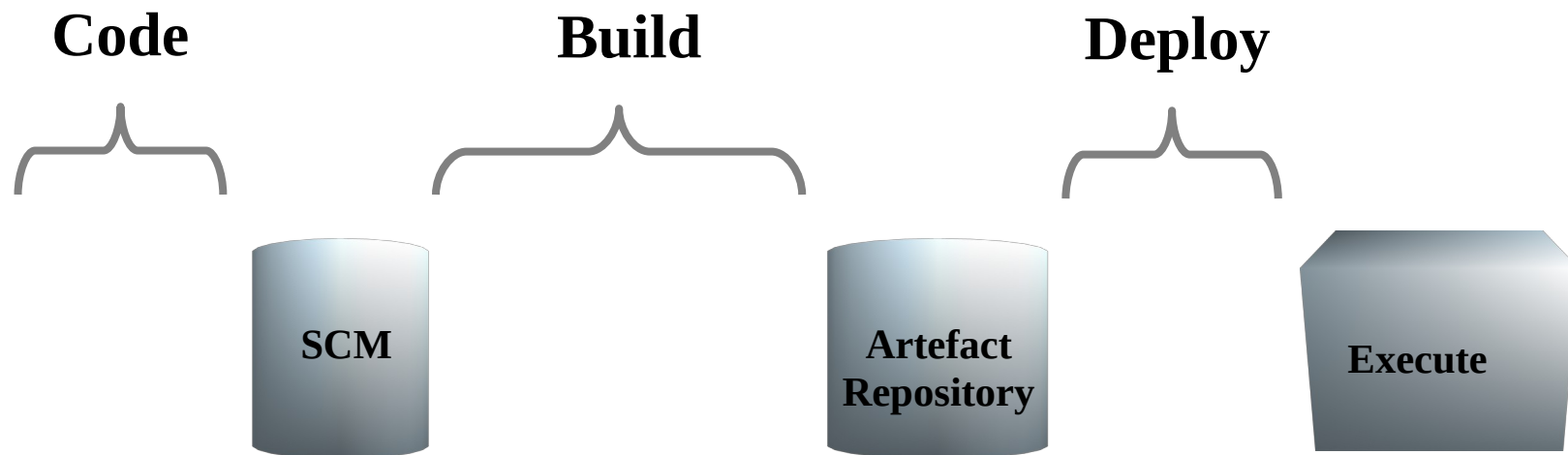
Déploiement continu

Le **déploiement continu (Continuous Deployment)** est le stade ultime de l'intégration continue

La totale confiance dans les tests exécutés lors de la production de release permet de déployer automatiquement en production.



Cycle de vie d'un logiciel





Types d'outils

Différents types d'outils sont utilisés durant le cycle de vie. Citons:

- Les **SCMs** (Source Control Management) qui centralisent le code source, permet le développement courant via les branches
- Les **outils de build** qui permettent de séquencer les tâches de construction : compilation, tests, documentation, packaging
- Les **dépôts d'artefacts** qui stockent et fournissent les différentes releases du logiciel
- La **plate-forme de livraison** permet de contrôler une version à livrer et de provisionner les environnements de production, de QA, ...

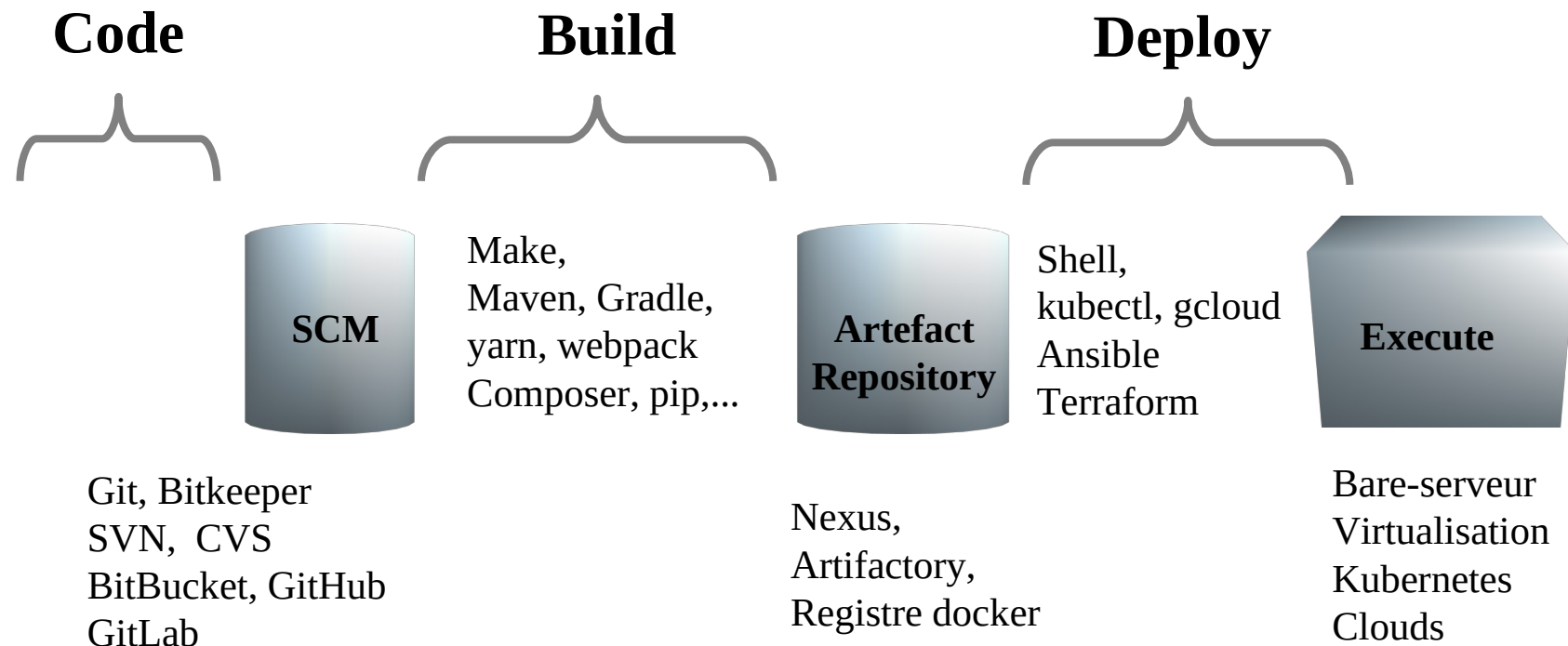
La plateforme d'intégration continue joue alors le rôle de chef d'orchestre entre ses outils,
=> cela nécessite des capacités d'intégrations avec toute la variété des outils utilisés



PIC et outils

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Azure Pipeline





Mise en place

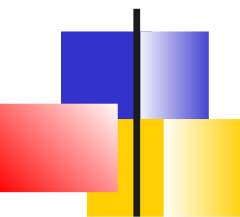
Le projet Jenkins/Hudson

Installation

Interface utilisateur

Configuration du serveur, plugins, outils

Configuration d'un Job FreeStyle

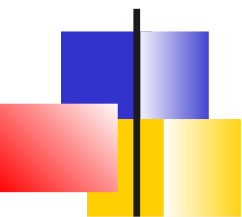


Introduction

Jenkins, à l'origine Hudson, est une plateforme CI/CD écrite 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



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)



Histoire

Démarrage du projet en 2004 par *Kohsuke Kawaguchi* au sein de Sun.

En 2010, 70 % du marché

Rachat de Sun par Oracle et divergences entre l'équipe initiale de développement et Oracle
=> 2011 Fork du projet Hudson et création de Jenkins qui reste dans le mode OpenSource

2014 : La société *CloudBees* emploie la plupart des committers Jenkins et supporte commercialement la solution



Installation



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



Pré-requis

Hardware minimum :

- 256 MB RAM
- 1 GB de disque, 10 GB si exécution dans un conteneur Docker

Hardware recommandé pour une petite équipe :

- 1 GB+ RAM
- 50 GB+ d'espace disque

Software :

- Java 8 : JRE ou JDK



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 Docker

Plusieurs images sont disponibles, l'image recommandée est : ***jenkinsci/blueocean***

Elle intègre une version de Jenkins LTS + tous les plugins nécessaire à l'éco-système Blue Ocean (~Version moderne de Jenkins)

Commande de démarrage :

```
docker run \  
-u root \  
--rm \ # Suppression automatique du conteneur lors de son arrêt  
-d \ # Background  
-p 8080:8080 \ # Publication du port 8080 sur le hôte de Docker  
-v jenkins-data:/var/jenkins_home \ # Montage de fichier sur le hôte  
-v /var/run/docker.sock:/var/run/docker.sock \ # Pour pouvoir lancer des  
images Docker  
jenkinsci/blueocean
```



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
```




Installation package natif

1. Ajouter le dépôt adéquat :
Ex : <http://pkg.jenkins-ci.org/debian>
2. Utiliser *apt-get*, *yum* ou *zypper* pour installer sous *Debian/Ubuntu*, *RedHat/Fedora/CentOS*, *Suse/OpenSuse*



Installation package Linux

L'installation du package Linux a pour conséquences :

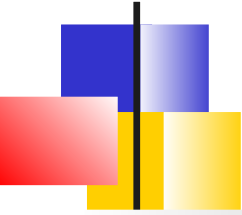
- Un utilisateur *jenkins* est créé
- Le service est démarré par un bash (*/etc/init.d/jenkins*)
- Le service est configurable via un fichier externe (*/etc/default/jenkins*)
- Le JENKINS_HOME est par défaut */var/lib/jenkins*
- Les traces */var/log/jenkins/jenkins.log*



Windows

Jenkins/Cloudbees propose :

- Un installateur Windows qui installe Jenkins en service
 - Le service se configure via le fichier *jenkins.xml*
 - Des documentations existent également pour mettre Apache ou *nginx* en proxy
- Un déploiement sur le Cloud Azure



Configuration service Windows (*jenkins.xml*)

```
<service>
<id>jenkins</id>
<name>Jenkins</name>
<description>Jenkins continuous integration
  system</description>
<env name="JENKINS_HOME" value="D:\jenkins" />
<executable>java</executable>
<arguments>-Xrs -Xmx512m
-Dhudson.lifecycle=udson.lifecycle.WindowsServiceLifecycle
-jar "%BASE%\jenkins.war" --httpPort=8081 --ajp13Port=8010
</arguments>
</service>
```



Installation sur serveur applicatif

En général, il suffit de copier *jenkins.war* dans le répertoire de déploiement du serveur.

Pour Tomcat : `$CATALINA_BASE/webapps`



Utilisateur Jenkins

En général, Jenkins s'exécute avec un utilisateur dédié, cela permet de faciliter le monitoring du serveur

- L'installation par packages crée automatiquement l'utilisateur Jenkins
- Dans la plupart des environnements, il faudra configurer l'utilisateur afin qu'il puisse accéder aux outils utilisés par Jenkins (JDK, Ant, Maven, ssh, ...)



Options du script de démarrage

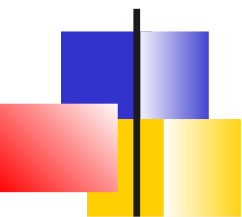
- httpPort** : Port http (Par défaut 8080)
- ajp13Port=8010** : Frontal Apache
- controlPort** : Démarrage/arrêt du serveur Winstone
- prefix** : Chemin de contexte pour l'application web.
- daemon** : Si Unix possibilité de démarrer Jenkins comme daemon.
- logfile** : Emplacement du fichier de log de Jenkins (par défaut le répertoire courant)



Etapes 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_HOME



Recherche du répertoire HOME

Au démarrage de l'application web, Jenkins recherche son répertoire home dans cet ordre :

1. Une entrée dans l'environnement JNDI nommée *JENKINS_HOME*
2. Une entrée dans l'environnement JNDI nommée *HUDSON_HOME*
3. Une propriété système nommée *JENKINS_HOME*
4. Une propriété système nommée *HUDSON_HOME*
5. Une variable d'environnement nommée *JENKINS_HOME*
6. Une variable d'environnement nommée *HUDSON_HOME*
7. Le répertoire *.hudson* dans le répertoire de l'utilisateur
8. Le répertoire *.jenkins* dans le répertoire de l'utilisateur



Changer *JENKINS_HOME*

Une première possibilité est donc de définir la variable d'environnement ***JENKINS_HOME***

```
export JENKINS_BASE=/usr/local/jenkins
```

```
export JENKINS_HOME=/var/jenkins-data
```

```
java -jar ${JENKINS_BASE}/jenkins.war
```

Dans un contexte Tomcat, on peut définir un fichier *jenkins.xml* dans *\$CATALINA_BASE/conf/localhost* qui redéfinit **JENKINS_HOME**

```
<Context docBase="../jenkins.war">
```

```
<Environment name="JENKINS_HOME" type="java.lang.String"  
value="/data/jenkins" override="true"/>
```

```
</Context>
```



Mise à jour

Les mises à jour de Jenkins n'altèrent pas le répertoire HOME

Elles peuvent s'effectuer via :

- Les packages natifs
- L'interface web
- Le remplacement de *jenkins.war* avec la nouvelle version

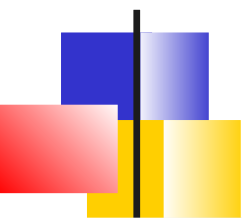
Les plugins peuvent également être mis à jour via l'interface web



Structure de répertoires

Sous JENKINS_HOME, on trouve :

- **jobs** : Configuration des jobs gérés par Jenkins ainsi que les artefacts générés par les builds
- **plugins** : Les plugins installés .
- **tools** : Les outils installés
- **secrets** : Mots de passes, crédits, token
- **fingerprints** : Traces des empreintes des artefacts générés lors des build.
- **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

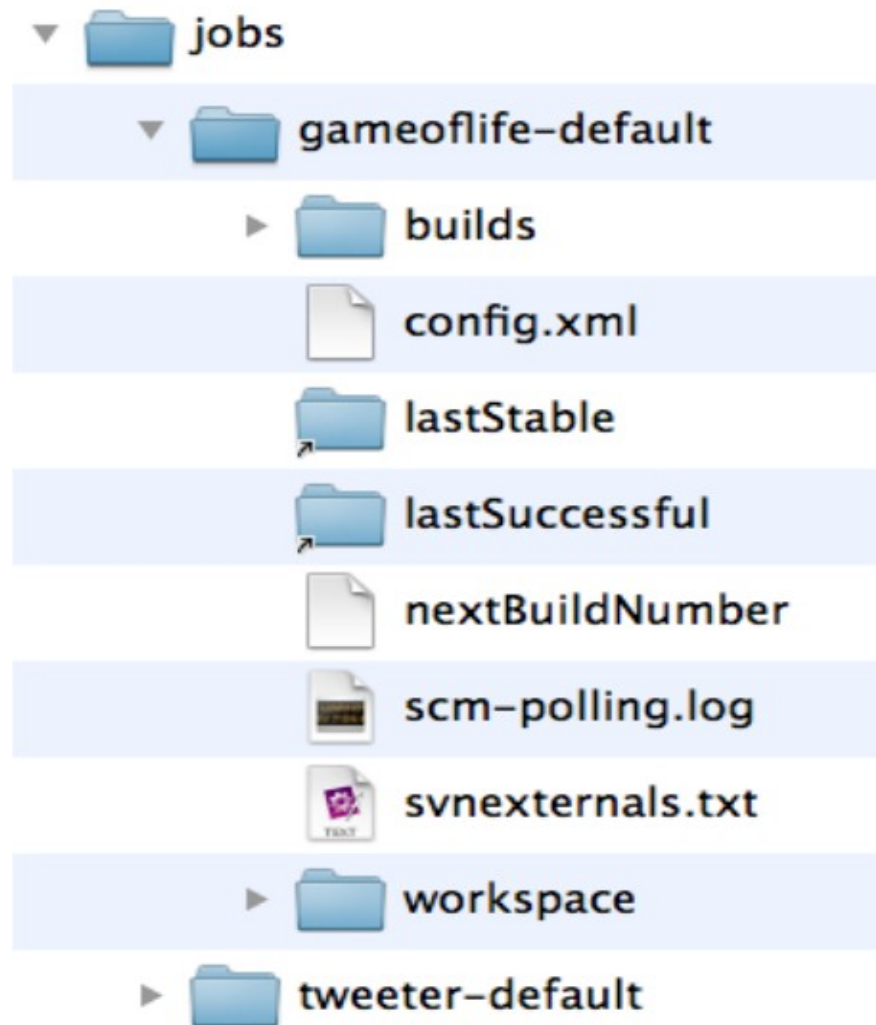
Le répertoire *jobs* contient un répertoire par projet Jenkins

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



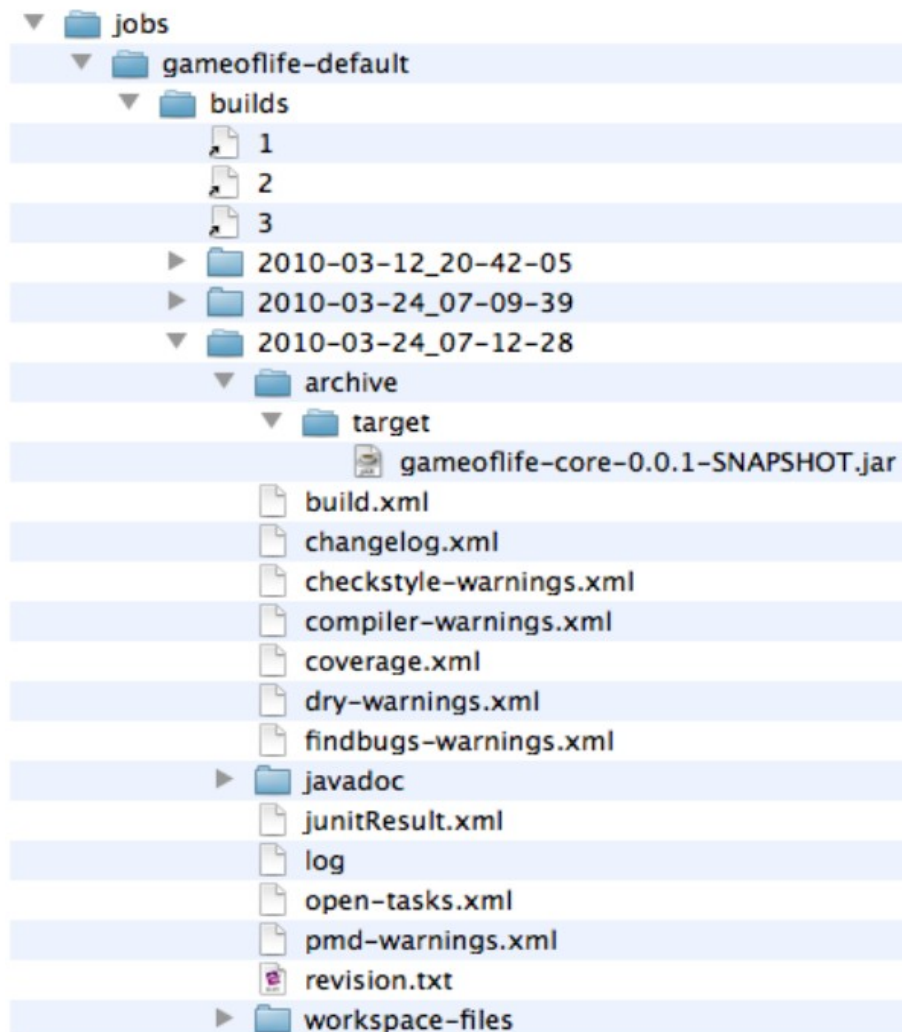


Répertoire build

Jenkins stocke l'historique et les artefacts de chaque build dans un répertoire numéroté

Chaque répertoire de build contient un fichier *build.xml* contenant les informations du build, le fichier de log, les changements par rapport aux dernier build effectué, les artefacts générés, et toutes les données publiées lors des actions de post-build

Exemple répertoire builds





Espace disque

La taille du répertoire de build a tendance a continuellement augmenter.

- => Utiliser une partition suffisamment large pour stocker les informations des builds (fichiers XMLs) et éventuellement les artefacts de l'application archivés (jar, war, ...)
- => Il est recommandé de limiter le nombre de builds stockés pour un job
- => Backup régulier du répertoire JENKINS_HOME



Configuration du serveur



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 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

Il permet de :

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

L'installation généralement ne nécessite pas de redémarrage. Des dépendances existent entre les plugins



Mode d'installation des plugins

Les plugins sont fournis sous forme de fichiers **.hpi** autonomes, qui contiennent tout le code, les images et les autres ressources nécessaires au bon fonctionnement du plug-in.

L'installation d'un plugin peut se faire :

- L'utilisation de l'UI "Plugin Manager"
- L'utilisation de la commande *install-plugin* de Jenkins CLI.
- L'image Docker officielle de Jenkins contient un script *plugin.sh* capable d'installer des plugins via un fichier texte listant les clés des plugins



Interface utilisateur et exécution de Jobs



Interface Web

L'interface utilisateur de Jenkins propose :

- Validation à la volée des champs de formulaire
- Rafraîchissement automatique
- Aide contextuelle
- Internationalisation
- Liens permanents
- URLs REST



Constitution

- Page d'accueil de type *Tableau de bord* qui donne l'état de santé des différents projets/builds
- Page projet : Liste les jobs effectués, affiche des graphiques de tendance
- Page job : Accès aux traces de la console, à la cause de démarrage aux artefacts créés
- Page de configuration : Toutes les configurations possibles : plugins, outils, utilisateurs

Il est possible de personnaliser l'interface en fonction des utilisateurs

Tableau de bord

Tableau de bord [Jenkins] - Mozilla Firefox

Quiz x Tableau de bord [Je... x Tableau de bord [Je... x Can't start apache!... x prroxy_ajp depende... x Renouvelez instanta... x Meetings-stime [Jen... x

integration.mymeeingsondemand.com:8080

rechercher S'identifier Créer un compte

Jenkins

Rafraichissement automatique

Ajouter une description

Nouveau Item

Utilisateurs

Historique des constructions

Administrer Jenkins

Credentials

File d'attente des constructions

File d'attente des constructions vide

État du lanceur de compilations

#	État
1	Au repos
2	Au repos

S	W	Name ↓	Dernier succès	Dernier échec	Dernière durée
		Charge demo	8 mo. 8 j - #453	s. o.	26 mn
		Meetings-coverage	3 mo. 27 j - #27	s. o.	35 s
		Meetings-proto	10 mo. - #1	s. o.	24 s
		Meetings-stime	7 j 0 h - #33	s. o.	35 s
		Meetings-web	3 mo. 27 j - #113	3 mo. 27 j - #110	35 s
		Syndicflow	3 mo. 5 j - #5	10 mo. - #2	43 s
		TestCGINDRE	28 mn - #5714	1 j 20 h - #5670	8 s
		TestChargeCGIndre	6 mo. 23 j - #6	s. o.	5 mn 18 s

Icône: S M L

Légende RSS pour tout RSS de tous les échecs RSS juste pour les dernières compilations

Aidez-nous à traduire cette page

Page générée: 7 juin 2015 20:13:45 [REST API](#) Jenkins ver. 1.574



Page projet

La page projet permet :

- De configurer le projet
- Visualiser les graphiques de tendances (test, temps de build, métriques, ...)
- Accéder à l'espace de travail
- Démarrer manuellement un build
- Voir les changements récents (commit)
- Accéder aux derniers builds (liens permanents)
- Accéder aux builds liés (amont ou aval)

Page projet

The screenshot displays the Jenkins web interface for the 'Meetings-stime' project. The browser is Mozilla Firefox, and the URL is integration.mymeetingsondemand.com:8080/job/Meetings-stime/. The interface includes a sidebar with navigation links, a main content area with project details, and a 'Test Result Trend' chart.

Project Meetings-stime

Build Meetings integration

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

[Email Template Testing](#)

[Subversion Polling Log](#)

[Workspace](#)

[Recent Changes](#)

[Latest Test Result](#) (no failures)

[edit description](#)

[Disable Project](#)

Build History (trend)

Build Number	Timestamp
#33	May 31, 2015 7:45:05 PM
#32	May 15, 2015 6:25:09 PM
#31	May 3, 2015 8:45:06 PM
#30	Apr 12, 2015 12:00:07 PM
#29	Apr 9, 2015 12:50:06 PM
#28	Apr 9, 2015 10:40:05 AM
#27	Apr 4, 2015 6:50:10 PM
#26	Mar 23, 2015 6:45:08 PM

Test Result Trend

count


#26 #27 #28 #29 #30 #31 #32 #33

(just show failures) enlarge


Permalinks


- [Last build \(#33\), 5 days 13 hr ago](#)
- [Last stable build \(#33\), 5 days 13 hr ago](#)
- [Last successful build \(#33\), 5 days 13 hr ago](#)


Page Build


 **Jenkins**


Dashboard > 1_freestyle > #6


 [Back to Project](#)


 [Status](#)


 [Changes](#)


 [Console Output](#)


 [Edit Build Information](#)


 [Delete build '#6'](#)


 [Polling Log](#)


 [Git Build Data](#)



 [Test Result](#)



 [Open Blue Ocean](#)


 [Previous Build](#)

 **Build #6 (Jun 8, 2021, 9:34:50 AM)**


 **Build Artifacts**

 [gs-multi-module-application-0.0.3-SNAPSHOT.jar](#) 17,46 MB  [view](#)

 [gs-multi-module-library-0.0.3-SNAPSHOT.jar](#) 3,15 KB  [view](#)


 **Changes**

1. Ajout chemin jacoco ([details](#))

 **Started by an SCM change**

Revision: 15bef6d0b82f52ef6258f7b3482d776c38f622cb
Repository: /home/dthibau/Formations/Jenkins/MyWork/multi-module

- refs/remotes/origin/master

 **Test Result** (no failures)



Configuration de jobs freestyle



Types de jobs et Outils de build

Sans plugin installé, Jenkins propose un seul type de job :

Job FreeStyle : Script shell ou *.bat*
Windows

En fonction des plugins installés, d'autres types de job peuvent être proposés par l'interface : Maven, Pipeline, ...



Sections de configuration

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 du build (choix dépendant du type de build et des plugins installés)
- Les étapes après le build (choix dépendant des étapes de build et des plugins installés)



Nom du projet

Le nom du projet est utilisé comme
répertoire et dans des URLs

=> Éviter les espaces et les accents



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

Cependant, Jenkins ne détruit jamais les derniers builds stables

On peut également conserver pour toujours un build particulier



Options avancées

L'option « Période d'attente » permet de surcharger la valeur globale défini dans la configuration du serveur Jenkins.
(Temps d'attente avant le démarrage du build)

« Nombre d'essais » : Le nombre de fois ou Jenkins tente de faire un checkout

Les options « Empêcher le build quand un projet en amont est en cours de build » ou « Empêcher le build quand un projet en aval est en cours de build » sont utiles lorsque plusieurs projets sont affectés par un même commit et qu'ils doivent être exécutés selon un ordre particulier

L'espace de travail ou le code source est récupéré peut être surchargé par le champ « Utiliser un répertoire de travail spécifique »



Déclencheurs

Il y a 4 façons de déclencher un build freestyle :

- A la fin d'un autre build
- Périodiquement
- En surveillant le SCM, et en déclenchant le build si un changement est détecté
- Manuellement



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”



Builds périodiques

Dans ce cas, on n'est plus vraiment dans les objectifs de l'intégration continue

Cependant, les *nightly builds* peuvent s'appliquer à des builds longs

Jenkins utilise la syntaxe *cron* constitué de 5 champs séparé par un espace correspondant à :

MINUTE HEURE JOUR_DU_MOIS MOIS JOUR_DE_LA SEMAINE



Syntaxe Cron

"*" représente toutes les valeurs possibles pour un champ

"* * * * *" = chaque minute

"* * * * 1-5" = Chaque minute du Lundi au Vendredi

"*/5 * * * * *" = Toutes les 5mn

"15,45 * * * * *" = A $\frac{1}{4}$ et moins le $\frac{1}{4}$ de chaque heure

Les raccourcis suivants sont autorisés : "@yearly", "@annually", "@monthly", "@weekly", "@daily", "@midnight", et "@hourly".



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
- Déclencher la vérification Jenkins du SCM qui provoquera un build



Déclenchement du build

Le déclenchement du build peut se faire en demandant l'URL :

<http://SERVER/jenkins/job/PROJECTNAME/build>

Avec Subversion (Git), il faut alors écrire un hook se déclenchant après un commit (push).

Les solutions comme GitHub ou Gitlab permettent de faire facilement ce type d'intégration



Sécurité

Si la sécurité Jenkins est activée, il suffit de fournir une URL avec un token associé à un utilisateur :

```
http://SERVER/jenkins/job/PROJECTNAME/  
build?token=D0IT
```

Ou avec un user/password

```
curl -u scott:tiger  
http://scott:tiger@myserver:8080/jenkins/j  
ob/gameoflife/build
```




Interactions avec le SCM

La plupart des jobs sont reliés à un SCM et le démarrage d'un job consiste en

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

Jenkins propose des plugins pour la plupart des SCMs :

Accurev, Bazaar, BitKeeper, ClearCase, CVS, CMVC, Dimensions, Git, CA Harvest, Mercurial, Perforce, PVCS, StarTeam, SVN, CM/Synergy, Microsoft Team Foundation Server et Visual SourceSafe



Configuration Git

La configuration consiste à spécifier :

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

Par défaut, Jenkins extrait alors la branche spécifiée dans la racine de son workspace.



Options avancées

- Timeout pour les checkout ou les clones
- Checkout vers une branche locale ou un sous-répertoire particulier
- Nettoyage du workspace (avant ou après le checkout)
- Effectuer un merge avec une autre branche avant le build
- Ignorer certains commits pour le déclenchement (users, chemins, message de commits)
- Utiliser l'auteur plutôt que le commiter dans le change log
- ...



Étapes de build

Un *build freestyle* peut être organisé en étapes ayant des incidences sur le résultat de build

Les étapes proposées par l'UI dépendent des plugins installés

Par défaut, les étapes peuvent être de type :

- Maven (2 et 3)
- Bat Windows
- Shell



Étape Maven

1. Sélectionner “*Invoquer les cibles Maven de haut niveau*” parmi les étapes de build proposées
2. Choisir l’outil Maven prédéfini
3. Saisir les objectifs Maven à exécuter
Il est possible de passer des options



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 **!= 0**, le build échoue

=> *Ce type d'étape 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)

Jenkins positionne des variables d'environnements qui peuvent être utilisées dans les jobs :

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.

GIT_COMMIT : Identifiant du commit Git



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
- Pousser une branche, tagger le SCM



Archivage des artefacts

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 d'artefacts comme Nexus ou Artifactory, il est préférable de déployer automatiquement les artefacts dans le repository pendant le job de build



Artefacts via Jenkins

Demander à Jenkins d'archiver des artefacts à comme conséquence :

- Le stockage dans *JENKINS_HOME* de l'artefact généré
- Une URL d'accès à l'artefact et la présence d'un lien dans la page web du build ayant généré l'artefact
- La possibilité (via un plugin) de fournir l'artefact généré à un autre build



Empreintes

Si des projets sont inter-dépendants, i.e. utilisation d'un artefact généré par un autre projet, il est utile de demander à Jenkins d'enregistrer les **empreintes (fingerprints)**

=> Cela permet d'être sûr de quel artefact a été utilisé par tel build

Post-build action → Record fingerprints of files to track usage



Publication des tests

Le format xUnit, utilisés par de nombreux outils de tests contient des informations sur les tests échoués mais également le nombre de tests exécutés et leurs temps d'exécution

Pour remonter ses informations dans Jenkins, sélectionner « Publier le rapport des tests JUnit » et indiquer l'emplacement des fichiers Junit préalablement générés

Les caractères '*' et '**' peuvent être utilisés : (***/target/surefire-reports/*.xml*)

Jenkins agrège tous les fichiers trouvés en un seul rapport



Git Publisher

Exemple typique : Si le build réussit, on tag et push sur master

Post-build Actions

Git Publisher

Push Only If Build Succeeds ☒

Merge Results ☐
If pre-build merging is configured, push the result back to the origin

Force Push ☒
Add force option to git push

Tags
Tags to push to remote repositories

Branches

Branch to push

Target remote name

Branches to push to remote repositories

Notes
Notes to push to remote repositories



Permanent URLs / Build status

Des liens permanents, utilisables dans d'autres build Jenkins ou dans des scripts externes, permettent d'accéder aux artefacts les plus récents.

Les URLs disponibles concernent : le dernier build stable, réussi ou terminé

Réussi : Il n'y a eu aucune erreur de compilation, l'URL est de la forme
`/job/<build-job>/lastSuccessfulBuild/artifact/<path-to-artifact>`

Stable : réussi et aucun des rapports « post-build » (test, couverture de code, métriques qualité) ne l'a marqué comme instable (configuration projet)
`/job/<build-job>/lastStableBuild/artifact/<path-to-artifact>`

Terminé : terminé quelque soient ses résultats
`/job/<build-job>/lastCompletedBuild/artifact/<path-to-artifact>`



Architecture maître/esclaves



Introduction

Une des fonctionnalités les plus puissante de Jenkins est sa capacité à **distribuer les jobs** sur des machines (nombreuses) distantes

Il est aisé de mettre en place une ferme de serveurs afin de répartir la charge ou d'exécuter les jobs dans différents environnements

Des machines peuvent être dynamiquement ajoutées pour absorber des pics de charge.



Architecture Jenkins

Jenkins utilise une architecture **maître/esclave**

- Le nœud maître
 - gère le démarrage des jobs, les distribue sur les esclaves, surveille les esclaves
 - enregistre et présente les résultats des builds.
 - Il peut éventuellement exécuter lui même des jobs.
- Les nœuds esclaves exécutent les jobs qu'on leur a demandé.

Il est possible de configurer un projet afin qu'il s'exécute sur certains nœuds esclaves ou de laisser Jenkins choisir un nœud esclave.



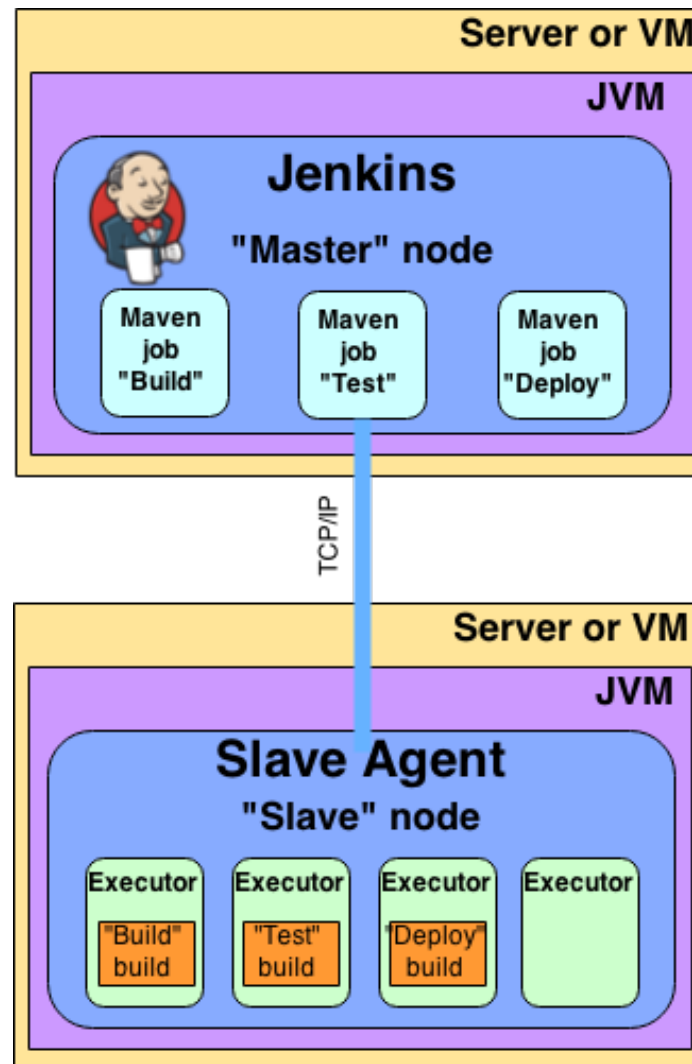
Nœud esclave

Un nœud esclave est un petit **exécutable Java** s'exécutant sur une machine distante et écoutant les requêtes provenant du nœud maître.

- Les esclaves peuvent s'exécuter sur différents systèmes d'exploitation
- Ils peuvent avoir différents outils pré-installés
- Ils peuvent être démarrés de différentes façons selon le système d'exploitation et l'architecture réseau
- Ils proposent un certain nombre d'exécuteurs

Une fois démarrés, ils communiquent avec le maître via des connexions TCP/IP

Architecture Maître/Esclave



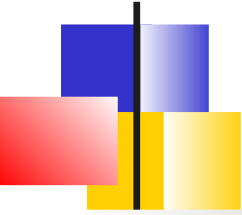


Type de nœud

Par défaut (sans installation de plugins supplémentaire), les nœuds esclaves sont de types « ***Agent permanent*** »

Cela signifie que ce sont des exécutables toujours démarrés

Certains plugins ajoutent des types de nœuds permettant par exemple le provisionnement dynamique d'esclaves.



Champs d'un nœud

Nom : Identifiant unique

Description : Aide pour les administrateurs

Nombre d'exécuteurs : Nombre de jobs en //

Home : Répertoire Home de travail (ne nécessite pas de backup, chemin spécifique à l'OS)

Labels ou **tags** : Permet de sélectionner des nœuds

Usage : Autant que possible ou dédié à un job particulier

Méthode de démarrage : Java Web Start, SSH/RSH, Service Windows

Disponibilité : Le nœud peut être mis offline et démarré seulement lorsque la charge est importante



Démarrage des esclaves

Différentes alternatives sont possibles pour démarrer les nœuds esclaves

- Le maître démarre les esclaves via **ssh** (le plus commun dans un environnement Unix/Linux, avec option pour non-blocking I/O)
- Le nœud esclave est démarré manuellement via **Java Web Start**
- Le nœud esclave est installé comme **service Windows**
- Le nœud esclave est démarré manuellement par une **commande en ligne**



Démarrage via ssh

Il est alors nécessaire :

- d'installer le plugin ***SSH Slaves plugin***.
Le plugin ajoute un nouveau choix dans le champ « démarrage » lors de la configuration du nœud.
- de fournir les **informations de connexions** aux nœuds esclave (hôte, créidentiels)
- Ou installer la **clé publique ssh du maître** dans `~/.ssh/authorized_keys` de l'esclave

Jenkins fait le reste : copie du binaire nécessaire, démarrage et arrêt des esclaves



Agents sur le cloud

Il est également possible de démarrer des agents sur le cloud. Les agents n'existent que pendant l'exécution du job :

- Le plug-in **EC2** permet d'utiliser AWS EC2
- Le plugin **JCloud** permet d'utiliser les fournisseurs compatible *JCloud*.

Si l'on dispose d'une plateforme Docker, les agents peuvent être des conteneurs dockers démarrés pendant l'exécution du job



Labels/Tags des nœuds

Les labels sont des tags que l'on peut associer à des esclaves afin de les différencier

Ils peuvent être utilisés :

- Pour indiquer qu'un nœud a certains outils installés
- Qu'il s'exécute sur tel OS
- Sa situation géographique ou réseau

Exemple :

`jdk windows eu-central docker`

On peut alors fixer des contraintes concernant le nœud pour un job particulier.

- Contrainte simple : `eu-central`
- Contrainte multiple : `docker && eu-central`



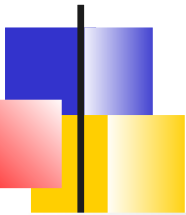
Surveillance des noeuds

Jenkins surveille également les nœuds esclave, il peut déclarer un nœud **offline** si il considère qu'il est incapable d'exécuter un build

3 métriques sont surveillées

- Le **temps de réponse** : un temps de réponse bas peut indiquer un problème réseau ou que la machine esclave est à l'arrêt
- Les **ressources disque** : l'espace disque, l'espace pris par les répertoires temporaires et l'espace de swap disponible
- Les **horloges** : elles doivent rester synchronisées avec l'horloge du maître

=> Si un de ces critères n'est pas correct, Jenkins déclare le nœud offline



Jenkins → Node → Configure

Jenkins

search

John | log out

Jenkins » nodes

Back to Dashboard

New Node

Configure

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Preventive Node Monitoring

☒ Response Time

☒ Free Disk Space

Free Space Threshold 1GB

☒ Architecture

☒ Free Temp Space

Free Space Threshold 1GB

☒ Free Swap Space

☒ Clock Difference

OK



Provisionnement

Les nœuds esclaves forment une sorte de cluster hétérogène qui peut comporter énormément de nœuds.

La gestion de configuration de ces nœuds (outils, comptes, droits sur les répertoires, etc..) peut être complexe et donc nécessiter l'utilisation d'outils de provisionnement ou de gestion de configuration (Puppet, Ansible, Chef, ...)



Installation automatique des outils

Jenkins nécessite de connaître l'emplacement des outils nécessaires aux jobs (Java, Maven, Gradle, etc.)

- Si les outils ont été configurés pour être automatiquement installés, aucune configuration supplémentaire n'est nécessaire
- Sinon, les outils doivent être préalablement installés et leur emplacement doit être indiqués à Jenkins



Dimensionnement de l'architecture



Noeud maître

Un nœud maître a pour principale vocation de :

- Démarrer des jobs de build sur des machines esclaves (pré-provisionnées ou provisionnées dynamiquement)
- Stocker la configuration, les plugins, les historiques des jobs, et éventuellement des artefacts dans `$JENKINS_HOME`
- Publier les résultats des Jobs et les rendre disponible aux équipes (=> estimer le nombre d'utilisateurs)
- Eventuellement,, effectuer lui-même des builds (pas spécialement recommandé dans les gros déploiement)



Espace disque

Le point le plus important est sûrement d'anticiper le grossissement de `$JENKINS_HOME`.

Il vaut mieux favoriser une grosse machine plutôt qu'une machine rapide (En termes de matériel, de la capacité disques extensibles plutôt que de la rapidité)

En fonction de l'OS différentes stratégies pour un stockage extensible :

- Volume fractionné sous Windows (NTFS)
- Volumes logiques pour Linux : LVM permet de redimensionner les volumes logiques à la volée.
- ZFS pour Solaris : Le plus flexible. Cela facilite la création d'instantanés, de sauvegardes, etc.
- Liens symboliques : Si les autres méthodes ne peuvent être utilisées. Des liens symboliques (liens symboliques) peuvent être utilisés pour stocker les dossiers des jobs sur des volumes distincts.



Mémoire

Si le maître dispose d'agents, ses besoins en mémoire sont faibles et dépendent :

- du nombre de jobs en // qu'il peut démarrer (compter 2Mo par jobs)
- et du nombre d'utilisateurs qui accèdent au master)

=> De 200 Mo à 1Go devrait suffire dans la plupart des cas

Si le maître effectue lui-même des builds, le dimensionnement mémoire dépend des build exécutés.



Noeuds agents

Les agents sont généralement des machines x86 génériques disposant de suffisamment de mémoire pour exécuter les builds.

Leur configuration dépend des builds pour lesquels, ils seront utilisés.

Netflix 2012

1 master avec 700
utilisateurs

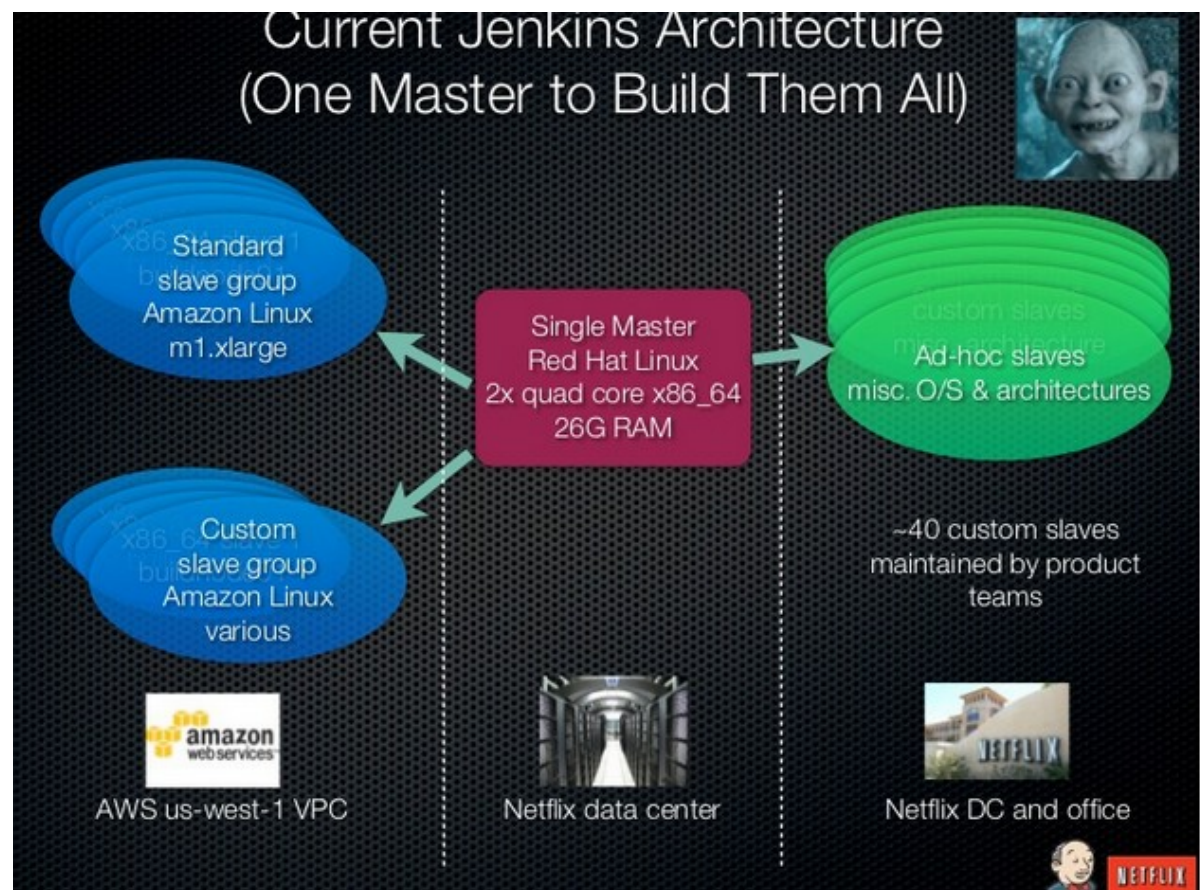
1,600 jobs :

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

=> 1 maître avec 26Go de RAM

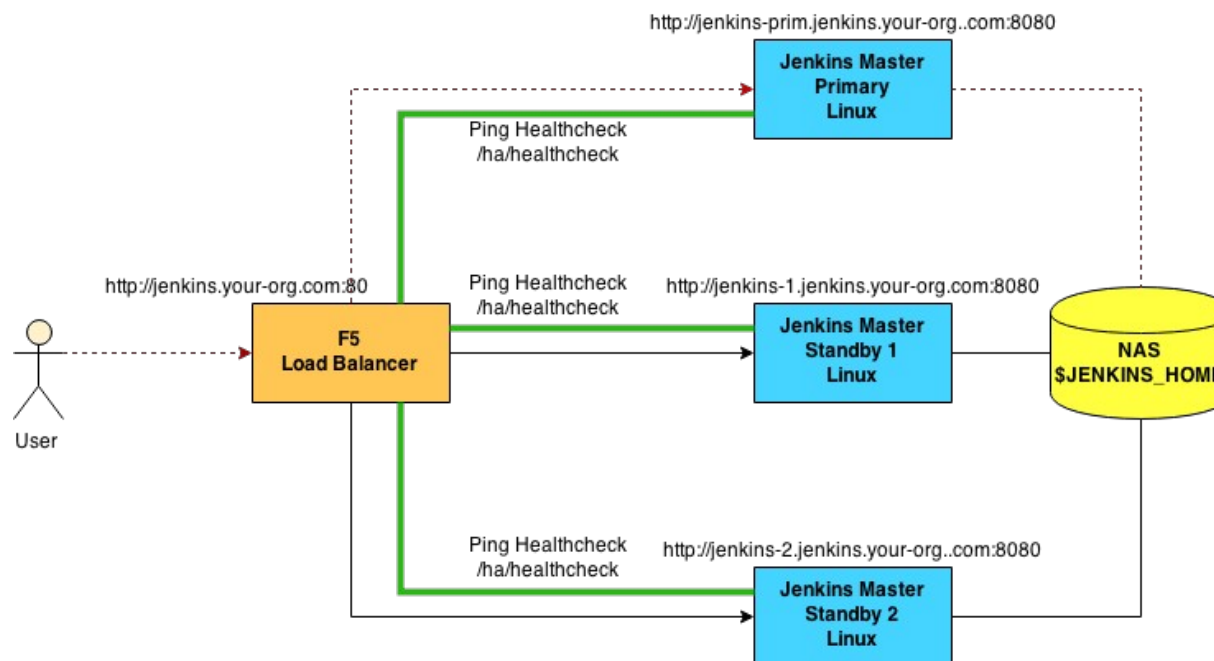
=> Agent Linux sur amazon

=> Esclaves Mac. Dans le réseau interne



Haute disponibilité

La version commerciale CloudBees Jenkins Enterprise permet de mettre en place des architecture HA





Relations entre jobs

Jobs paramétrés
Jobs multi-configurations
Relation amonts/aval et passage de données



Builds paramétrés

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

Ils sont renseignés :

- soit par l'utilisateur qui démarre le job manuellement.
Dans ce cas, Jenkins génère automatiquement l'interface utilisateur
- soit récupérés d'un autre job du build. (**Plugin Parameterized Trigger**)

Les paramètres sont ensuite mis à disposition des jobs 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`



Types de paramètres

- String, password
- Liste à choix fermé
- Booléens
- **Exécution**: Permet de sélectionner un build particulier d'un projet. La valeur du paramètre est alors l'URL d'exécution du build permettant par exemple d'accéder aux artefacts générés
- **File** : Permet de charger un fichier dans l'espace de travail. Le fichier peut alors être récupéré dans un script via `${workspace}/<param_value>`

Exemple Choix

Jenkins » parameterized-builds » unit-tests-build

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

Project name: unit-tests-build

Description:

☐ Discard Old Builds

☒ This build is parameterized

Choice

Name: DATABASE

Choices: mysql, oracle, postgres, derby

Description: Database to be used for the tests

Delete

Build History (trend)

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

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



Paramètre commit (Git)

Via les paramètres, Jenkins permet de construire le projet à partir d'un commit particulier. (ou révision svn)

- A l'exécution, Jenkins propose une liste de choix correspondant aux commit, branches ou tags trouvés dans le dépôt
- Il faut ensuite utiliser le paramètre pour extraire la bonne révision

Plugin : Git Parameter



Jobs multi-modules, multi-configuration




Projet multi-modules Maven

Le plugin ***Maven integration*** prend en charge les projets multi-modules.

L'interface permet de voir les relations entre les modules

Modules

S	W	Name ↓	Last Success
		Multi-Spring_Chapter Parent Project	25 min - #11
		Multi-Spring_Chapter Simple Command Line Tool	25 min - #11
		Multi-Spring_Chapter Simple Object Model	25 min - #11
		Multi-Spring_Chapter Simple Parent Project	25 min - #11
		Multi-Spring_Chapter Simple Persistence API	25 min - #11
		Multi-Spring_Chapter Simple Weather API	25 min - #11
		Multi-Spring_Chapter Simple Web Application	25 min - #11

Icon: [S](#) [M](#) [L](#)



Multiconfiguration jobs

Un job multi-configuration est un job paramétré qui peut être exécuté automatiquement avec toutes les combinaisons de valeurs des paramètres.

Ils sont particulièrement utiles pour les tests car les tests peuvent alors être effectués sous différentes conditions (navigateur, base de données, ...)

Un job multi-configuration est un job classique avec un élément additionnel de configuration : la **matrice de configuration**



Axes de configuration

La matrice de configuration permet de définir différents axes de configuration :

- **Axe des esclaves ou labels** : Par exemple, exécuter les tests sous Windows, Mac OS X, et Linux
- **Axe du JDK** : Jenkins exécute le build avec tous les JDKs installé pour le projet
- **Axe personnalisé** : Un paramètre du build auquel on a fournit toutes les valeurs possibles.



Exécution

Jenkins traite chaque combinaison de la matrice comme un job séparé.

Il affiche les résultats agrégés dans une table où toutes les combinaisons sont montrées. La table permet de naviguer au détail d'un job

- Par défaut, Jenkins exécute les jobs en parallèle, ce comportement peut être évité en cochant l'option « *Run each configuration sequentially* »
- L'option « *Combination Filter* » permet elle de mettre en place des règles qui limite le nombre de combinaison en indiquant les conditions pour lesquelles une combinaison est valable
- Enfin on peut également, indiquer que certains build doivent être exécutés en premier. Si ils échouent, les autres combinaisons ne sont pas exécutées.

Matrice

Jenkins

Jenkins » [acceptance-test-suite](#)

 [Back to Dashboard](#)

 [Status](#)

 [Changes](#)

 [Workspace](#)

 [Build Now](#)

 [Delete Project](#)

























 [Configure](#)

 [Dependency Graph](#)

 Build History	(trend)
 #3	Feb 12, 2011 1:18:52 PM
 #2	Feb 10, 2011 11:36:12 PM 50KB
 #1	Feb 10, 2011 11:34:54 PM 25KB
 for all  for failures	

Project acceptance-test-suite

Configuration Matrix

		OSX	linux	windows
mysql	Java 1.6			
	Java 1.5			
oracle	Java 1.6			
	Java 1.5			
postgres	Java 1.6			
	Java 1.5			
derby	Java 1.6			
	Java 1.5			



Relations amont/aval et passage de données

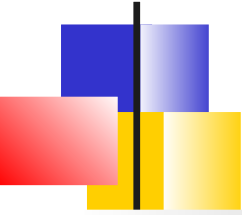


Introduction

Le chaînage de jobs avec ce 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 plugin **Pipeline** et les plugins liés permettant de définir des pipelines complexes en Groovy

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



Déclenchement de build paramétré

Pour qu'un build déclenche un build paramétré, le plugin « ***Parameterized Trigger*** » est nécessaire.

Il permet de configurer le passage de paramètres entre build.

Les paramètres peuvent ainsi être renseignés avec :

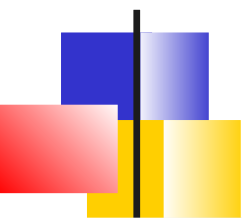
- Des variables d'environnement du build courant
- Des valeurs en *dur*
- Des valeurs provenant d'un fichier *.properties*



Passage de données entre jobs d'une pipeline

Avec le concept de pipeline, il est nécessaire de s'assurer que les jobs travaillent sur les mêmes sources (révision svn ou clé de hash d'un commit Git) ou les mêmes artefacts.

- Les identifiants de commit, branches peuvent être passés en **paramètres** des jobs
- Le plugin « **Copy Artifact** » permet de copier des artefacts construits par un build précédent dans le build courant



Configuration Copy Artifacts

×

Copy artifacts from another project

Project name

1_weather_unit

?

Which build

Latest successful build

▼

?

☐ Stable build only

Artifacts to copy

**/*.war

?

Artifacts not to copy

?

Target directory

.

?

Parameter filters

?

☐ Flatten directories

☐ Optional

☒ Fingerprint Artifacts

?

Advanced...



Fingerprints

Lorsque des jobs utilisent des artefacts d'autres jobs, il est intéressant d'enregistrer les empreintes

=> Il est ainsi facile de retrouver l'archive utilisée par un job dépendant

=> Jenkins utilise ce mécanisme automatiquement lors de build Maven avec les dépendances du projet

Les empreintes sont stockées dans le répertoire ***fingerprint***. (Fichier au format XML contenant le checksum MD5 et les usages du jar).



Plugins Legacy pour relation Amont/Aval

Dependency Graph Viewer : Permet de visualiser graphiquement les relations entre jobs

Pipeline viewer : Vue d'exécution d'une pipeline de jobs

Join : Synchronisation des jobs aval

Lock : Mise en place de verrou



Pipelines

Approche et concepts
Syntaxes déclaratives et Scripts
Types de Jobs, Interface Blue Ocean
Syntaxe déclarative
Syntaxe script
Utilisation de Docker
Librairies partagées



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

an-example


» 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.

OK

Premier script

Pipeline

Definition Pipeline script ▼

Script

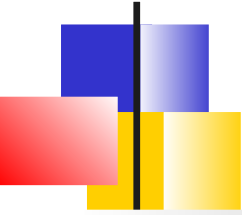
```
1 node {  
2   echo "Hello World"  
3 }
```

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

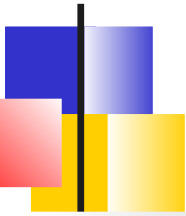
```
stage('Deploy') {  
    // some block  
}
```



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...' }
    }
  }
}

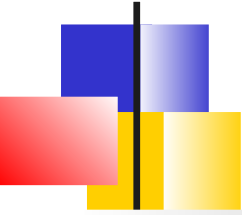
// Script //
node {
  stage('Build') { echo 'Building....' }
  stage('Test') { echo 'Building....' }
  stage('Deploy') { echo 'Deploying....' }
}
```




Structure déclarative

La syntaxe déclarative est donc encapsulée par un bloc ***pipeline***.

Ensuite des **directives** permettent d'encapsuler des fonctionnalités classiques des pipelines qui nécessiteraient plusieurs lignes de code dans la syntaxe script.



Directives structurelles requises (déclaratif)

Certaines directives sont toujours présentes :

- La directive ***agent*** indique à Jenkins d'allouer un exécuteur et un espace de travail et d'effectuer un checkout du SCM
 - La directive est placée au niveau du bloc *pipeline* mais peut également être placée au niveau *stage*
- Les directives ***stages*** et ***steps*** (également requises) indiquent à Jenkins ce qui doit être exécuté et dans quelle phase.



Example

```
pipeline {  
  agent { docker 'maven:3.3.3' }  
  stages {  
    stage('build') {  
      steps {  
        sh 'mvn --version'  
      }  
    }  
  }  
}
```



Directives fonctionnelles

Les autres directives apportent des fonctionnalités souvent requises par une pipeline :

- ***tools*** : Ajouter un outil défini dans Jenkins dans le PATH
- ***parameters*** : Définir un paramètre au job
- ***input*** : Posez une question à un utilisateur
- ***post*** : Effectuer une étape de post-build
-



Bloc script

La syntaxe déclarative est plus restrictive que la syntaxe script. En particulier, on ne peut pas définir de variables.

Pour contourner cette restriction, on peut toujours utiliser le bloc ***script***

```
pipeline {
  agent any
  stages {
    stage('Example') {
      steps {
        echo 'Hello World'
        script {
          def browsers = ['chrome', 'firefox']
          for (int i = 0; i < browsers.size(); ++i) {
            echo "Testing the ${browsers[i]} browser"
          }
        }
      }
    }
  }
}
```



Syntaxe script

Une pipeline script a généralement
comme première ligne

#!groovy

Il a ensuite un ou plusieurs blocs **node**,
un ou plusieurs blocs **stage**

Dans ces blocs, des appels à des
taches/steps (identique à la syntaxe
déclarative)



Exemple script

```
node {
  def mvnHome
  stage('Preparation') { // for display purposes
    // Get some code from a GitHub repository
    git 'file:///home/dthibau/Formations/Jenkins/MyWork/weather-project'
    // Get the Maven tool.
    // ** NOTE: This 'M3' Maven tool must be configured
    // **      in the global configuration.
    mvnHome = tool 'M3'
  }
  stage('Build') {
    // Run the maven build
    if (isUnix()) {
      sh "'${mvnHome}/bin/mvn' -Dmaven.test.failure.ignore clean package"
    } else {
      bat("/"${mvnHome}\bin\mvn" -Dmaven.test.failure.ignore clean package/)
    }
  }
  stage('Results') {
    junit '**/target/surefire-reports/TEST-*.xml'
    archive 'target/*.jar'
  }
}
```



Interface pipeline et outils de mise au point



Jobs pipeline

Le plugin Pipeline ajoute de nouveaux types de Jobs :

- **Pipeline** : Définition d'une pipeline in-line ou dans un Jenkinsfile
- **Multi-branch pipeline** : On indique un dépôt et Jenkins scanne toutes les branches à la recherche de fichier Jenkinsfile. Un job est démarré pour chaque branche
- **Bitbucket/Team, Github** : On indique un compte et Jenkins scanne toutes les branches de tous les projet du serveur Bitbucket ou Github à la recherche de Jenkinsfile. Il démarre un job pour chaque Jenkinsfile trouvé



Variables d'environnement additionnelles

Les builds d'une pipeline multi-branches ont accès à des variables additionnelles :

- `BRANCH_NAME` : Nom de la branche pour laquelle la pipeline est exécutée
- `CHANGE_ID` : Identifiant permettant d'identifier le changement ayant provoqué le build

Les builds d'une pipeline multi-projet ont accès à des variables additionnelles identifiant le projet Github ou Bitbucket



Interface Blue Ocean

CloudBees met a disposition une interface dédié à la gestion des pipelines

It's a brand new way to use Jenkins that even your boss's boss can understand.

Release 1.0.0 GA : Avril 2017

Cette interface cohabite avec l'interface classique et est installable via un plugin



Blue Ocean

Blue Ocean a pour objectif de fournir une interface claire à tous les membres d'une équipe DevOps :

- Visualisations de pipelines de livraison continue
- Editeur de pipeline
- Personnalisation
- Attirer l'attention là où une pipeline nécessite une intervention
- Intégration native des branches et pull request pour faciliter la collaboration autour du SCM



Barre de navigation

La barre de navigation principale de Blue Ocean propose :

- Jenkins : Rechargement du Dashboard Legacy
- Pipelines : Dashboard des pipelines
- Administration Jenkins : Gestion via l'UI classique
- Basculement vers l'UI classique
- Déconnexion

Tableau de bord (multi-branches)



JenkinsPipelinesAdministrationDéconnexion

Tableau de bordNouveau Pipeline

Favorites

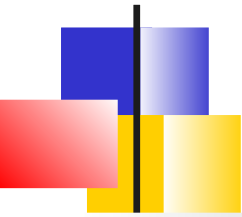
✓ plbsi-angular🔗 master🔑 #5c3e3ac🕒 2 months ago🔄 ⏮ ⭐

✓ plbsi-angular🔗 develop🔑 #d07b3cb🕒 4 months ago🔄 ⏮ ⭐

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

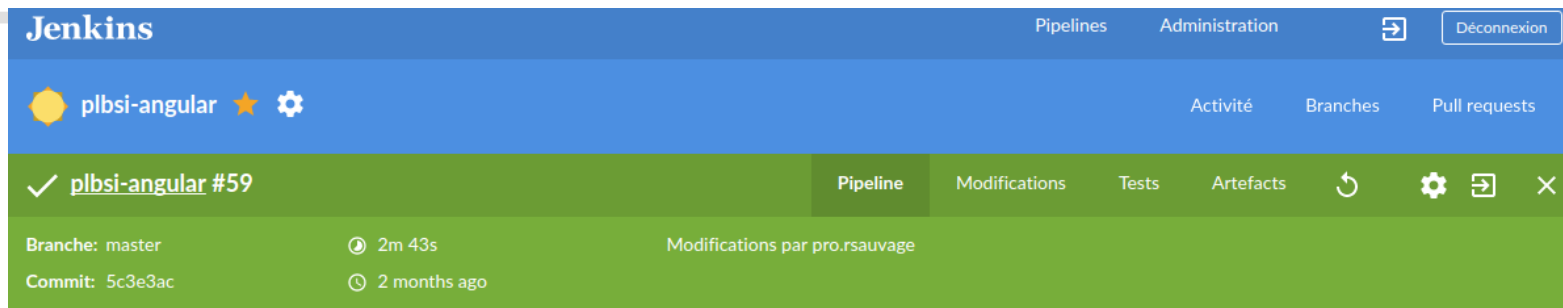
Détail pipeline

(*Git comments*)



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

Détail d'un build



The Jenkins UI header shows the 'plbsi-angular' project. The build status is 'plbsi-angular #59' with a green checkmark. The build is on the 'master' branch, commit '5c3e3ac', and was last modified '2 months ago'. The build duration is '2m 43s'. The build was triggered by 'Modifications par pro.rsauvage'. The UI includes tabs for 'Pipeline', 'Modifications', 'Tests', 'Artefacts', and a 'Déconnexion' button.

Jenkins

Pipelines Administration

plbsi-angular

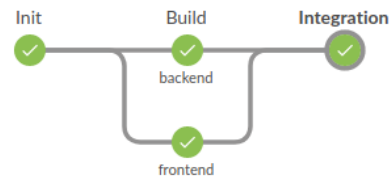
Activité Branches Pull requests

✓ plbsi-angular #59

Pipeline Modifications Tests Artefacts

Branche: master 2m 43s Modifications par pro.rsauvage

Commit: 5c3e3ac 2 months ago



Étapes - Integration

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



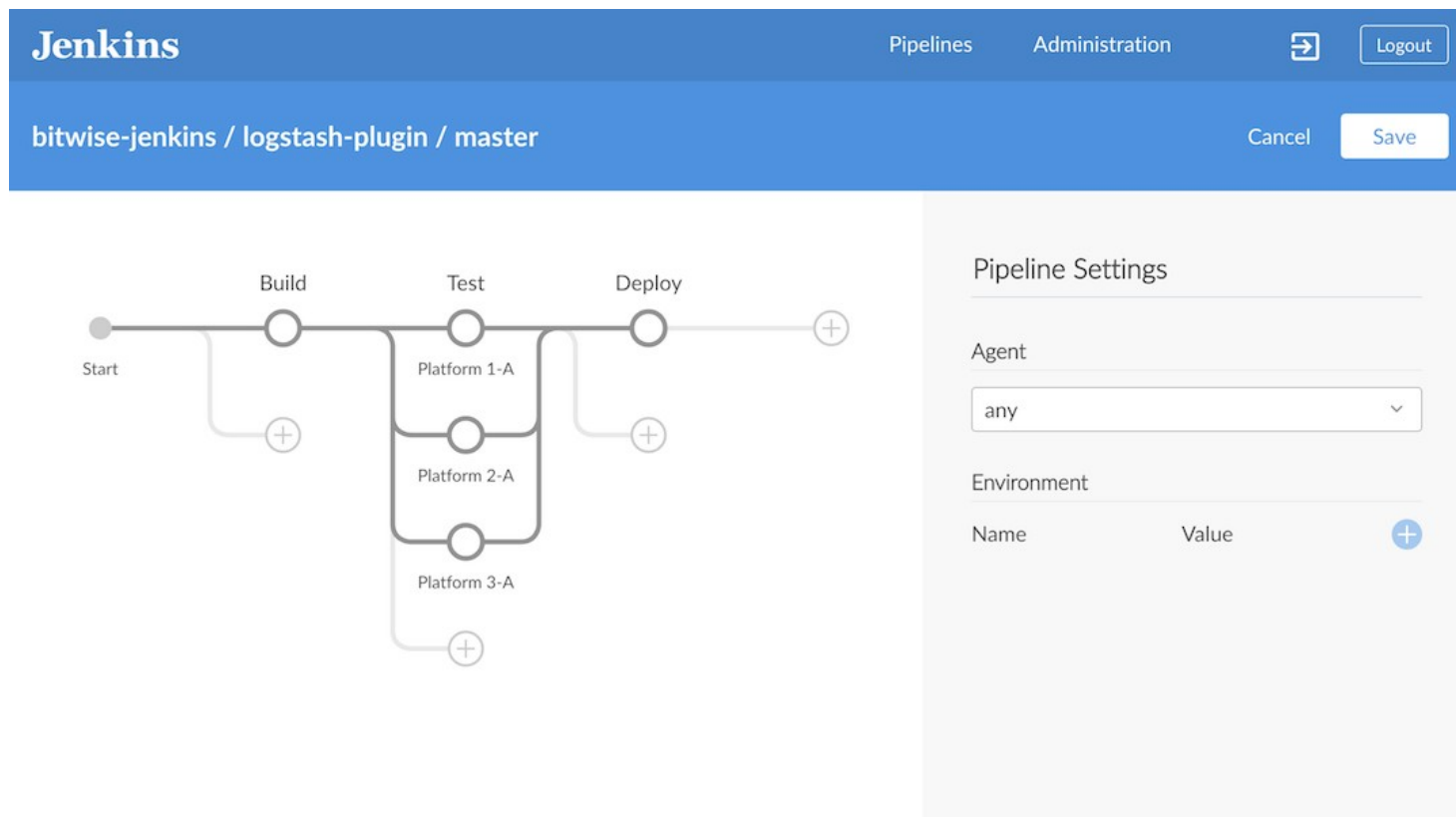
Editeur Blue Ocean

L'éditeur **Blue Ocean** fournit un outil WYSIWYG pour créer des pipelines déclaratives.

Il offre une vue structurée de toutes les étapes, et branches parallèles.

L'éditeur valide les changements dès lorsqu'ils sont effectués et avant qu'ils soient commités.

Illustration





Lint

En utilisant Jenkins CLI ou des requêtes HTTP POST, il est possible de valider une pipeline avant de l'exécuter :

```
# ssh (Jenkins CLI)
# JENKINS_SSHD_PORT=[sshd port on master]
# JENKINS_HOSTNAME=[Jenkins master hostname]
ssh -p $JENKINS_SSHD_PORT $JENKINS_HOSTNAME
    declarative-linter < Jenkinsfile
```



Rejouer une pipeline

Sur un build exécuté, le lien "**Replay**" permet de le rejouer en apportant des modifications (sans changer la configuration de la pipeline et sans committer)

Après plusieurs essais, il est possible de récupérer les modifications pour les committer



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/>



Syntaxe script



Introduction

Une pipeline scriptée est un DSL construit avec Groovy.

La plupart des fonctionnalités du langage Groovy sont disponibles rendant l'outil très flexible et extensible



String

En Groovy, les littéraux String peuvent utiliser les simples ou double-quotes.

La version double-quotes permet l'utilisation d'expressions qui sont résolues à l'exécution

```
def nick = 'ReGina'  
def book = 'Groovy in Action, 2nd ed.'  
assert "$nick is $book" == 'ReGina is Groovy in Action, 2nd  
ed.'
```



Pas de types primitifs

En Groovy, tout est objet.

- Tout nombre, booléen est converti en une objet Java.
- Les notations des types primitifs sont cependant toujours supportées :

```
def x = 1
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```



Collections

Groovy facilite la manipulation des collections en ajoutant des opérateurs, des instanciations via des littéraux et de nouvelles méthodes

Il introduit également un nouveau type :
Range



Exemples

```
// roman est une List
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
// On accède à un élément comme un tableau Java
assert roman[4] == 'IV'
// Il n'y a pas d'ArrayIndexOutOfBoundsException
roman[8] = 'VIII'
assert roman.size() == 9
// Les maps peuvent être facilement instanciées
def http = [
  100 : 'CONTINUE',
  200 : 'OK',
  400 : 'BAD REQUEST'
]
// Accès aux éléments avec la notation Array Java
assert http[200] == 'OK'
// Méthode put simplifiée
http[500] = 'INTERNAL SERVER ERROR'
assert http.size() == 4
```




Closures

Les ***closures*** Groovy permettent la programmation fonctionnelle.

Un bloc d'instructions peut être passé en paramètre à une méthode.

Un objet de type *Closure* travaille en sous-main

```
[1, 2, 3].each { entry -> println entry }
```

```
// Variable implicite it
```

```
[1, 2, 3].each { println it }
```



Structures de contrôle

```
if (false) assert false // if sur une ligne
if (null) { // null est false
    assert false
}
// Boucle while classique
def i = 0
while (i < 10) {
    i++
}
assert i == 10

// for in sur un intervalle
def clicks = 0
for (remainingGuests in 0..9) {
    clicks += remainingGuests
}
assert clicks == (10*9)/2
```



Structures de contrôle (2)

```
// for in sur une liste
```

```
def list = [0, 1, 2, 3]
for (j in list) {
    assert j == list[j]
}
```

```
// each avec une closure
```

```
list.each() { item ->
    assert item == list[item]
}
```

```
// Switch
```

```
switch(3) {
    case 1 : assert false; break
    case 3 : assert true; break
    default: assert false
}
```



Examples

```
node {  
  stage('Example') {  
    if (env.BRANCH_NAME == 'master') {  
      echo 'I only execute on the master branch'  
    } else {  
      echo 'I execute elsewhere'  
    }  
  }  
}
```



Limitations Groovy

Les pipeline, pour pouvoir survivre à des redémarrage, doivent sérialiser leurs données vers le master.

Ainsi certaines constructions Groovy ne sont pas supportées. Par exemple :

```
collection.each {  
    item → /* perform operation */  
}
```



Exécuteur

```
// Script
// Un nœud esclave taggé 'Windows'
node('Windows') {
    // some block
}
```



Exécution //

// Exemple script

```
stage('Test') {  
  parallel linux: {  
    node('linux') {  
      checkout scm  
      try {  
        unstash 'app'  
        sh 'make check'  
      }  
      finally {  
        junit '**/target/*.xml'  
      }  
    }  
  },  
  windows: {  
    node('windows') {  
      /* .. snip .. */  
    }  
  }  
}
```



Commandes basiques

// Bat Windows

```
bat 'dir'
```

// Shell

```
sh 'ls'
```

// Checkout branche master de Git

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],  
  doGenerateSubmoduleConfigurations: false, extensions: [],  
  submoduleCfg: [], userRemoteConfigs: [[url:  
    '/home/dthibau/Formations/MavenJenkins/MyWork/weather-  
    project']]])
```

// Plus simple, clone du repo :

```
git '/home/dthibau/Formations/MavenJenkins/MyWork/weather-  
project'
```

// Positionner la clé de hash dans une variable

```
gitCommit = sh(returnStdout: true, script: 'git rev-parse  
HEAD').trim()
```




Commandes basiques

// Nettoyer le workspace

```
cleanWs()
```

// Afficher un msg sur la console

```
echo 'Coucou'
```

// Changer de répertoire courant

```
dir('target') {
```

```
    // some block
```

```
}
```

// Archiver les résultats des tests

```
junit '**/target/test-reports/*.xml'
```



Exemple complet

```
#!/groovy

stage('Init') {
    node {
        git 'file:///home/dthibau/Formations/MyWork/MyProject/.git'
        echo 'Pulling...' + env.BRANCH_NAME
        sh(returnStdout: true, script: 'git checkout '+ env.BRANCH_NAME)
        gitCommit = sh(returnStdout: true, script: 'git rev-parse HEAD').trim()
    }
}

stage('Build') {
    parallel frontend : {
        node {
            checkout([$class: 'GitSCM', branches: [[name: gitCommit ]], userRemoteConfigs: [[url:
'file:///home/dthibau/Formations/MyWork/MyProject/']]])
            dir("angular") {
                sh 'nvm v9.5.0'
                sh 'ng build -prod' }
            dir ("angular/dist") {
                stash includes: '**', name: 'front'}
        }}, backend : {
        node {
            checkout([$class: 'GitSCM', branches: [[name: gitCommit ]], userRemoteConfigs: [[url:
'file:///home/dthibau/Formations/MyWork/MyProject']]])
            sh 'mvn clean install'
        } } }
```



Exemple complet (2)

```
stage('DockerImages') {  
    node { sh 'docker-compose build'    }  
}
```

```
stage('Integration') {  
    node { sh 'docker-compose up -d' }  
}
```



Exemple Tâche manuelle

```
stage 'Build to Stage' {
  node {
    sh "echo building"
    stash 'complete-workspace'
  }
}

stage 'Promotion' {
  timeout(time: 1, unit: 'HOURS') {
    input 'Deploy to Production?'
  }
}

stage 'Deploy to Production' {
  node {
    unstash 'complete-workspace'
    sh "echo deploying"
  }
}
```



Utilisation de Docker



Jenkins et Docker

Plusieurs cas d'usage de Docker dans un contexte Jenkins :

- Utiliser des images pour exécuter les builds
- Construire et pousser des images pendant l'exécution d'une pipeline
- Utiliser des images pour exécuter des services nécessaires à une étape de build (Démarrer un serveur lors de test d'intégration/fonctionnel)
- Dockeriser des configurations Jenkins



Agent Docker

Pipeline permet une utilisation facile des images Docker comme environnement d'exécution pour un Stage ou pour toute la Pipeline.



Exemple

```
// Declarative //
pipeline {
    agent {
        docker { image 'node:7-alpine' }
    }
    stages {
        stage('Test') {
            steps { sh 'node --version' }
        }
    }
}

// Script //
node {
    /* Nécessite le plugin Docker Pipeline */
    docker.image('node:7-alpine').inside {
        stage('Test') { sh 'node --version' }
    }
}
```




Gestion de cache

Les outils build téléchargent généralement les dépendances externes et les stocke localement pour les réutiliser.

- Pour réutiliser les téléchargements entre 2 builds, il faut monter des volumes persistant sur les nœuds exécutant les builds

Le bloc *docker{}* permet de passer des arguments à la commande de démarrage du container



Example

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

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



Dockerfile

Pipeline permet également de construire des images à partir d'un Dockerfile du dépôt de source.

Il faut utiliser la syntaxe déclarative :

```
agent { dockerfile true }
```

```
// Declarative //
```

```
pipeline {  
  agent { dockerfile true }  
  stages {  
    stage('Test') { steps {sh '--version'}}  
  }  
}
```



Docker Label

Par défaut, *Pipeline* assumes que tous les agents sont capables d'exécuter une pipeline Docker, ce qui peut poser problème si certains agents ne peuvent pas exécuter le daemon Docker.

Pipeline fournit une option globale permettant de spécifier un label pour les agents acceptant Docker

Pipeline Model Definition

Docker Label	<input type="text"/>	?
Docker registry URL	<input type="text"/>	?
Registry credentials	<input type="text" value="- none -"/>	Add



Construction d'image

Le plugin Pipeline mais à disposition la variable docker qui permet entre autres de :

- Déclarer un registre
- Construire ou récupérer une image
- Tagger, Pousser, Tirer des images
- Découvrir le mapping du port d'un conteneur en exécution
- ...



Example

```
script {  
    def dockerImage  
    = docker.build('dthibau/multi-module', '.')  
  
    docker.withRegistry('https://registry.hub.docker.com',  
                        'dthibau_docker') {  
        dockerImage.push 'latest'  
    }  
}
```



Example avancé – side car pattern

```
node {
  checkout scm
  docker.image('mysql:5').withRun('-e "MYSQL_ROOT_PASSWORD=my-secret-pw"') { c ->
    docker.image('mysql:5').inside("--link ${c.id}:db") {
      /* Wait until mysql service is up */
      sh 'while ! mysqladmin ping -hdb --silent; do sleep 1; done'
    }
    docker.image('centos:7').inside("--link ${c.id}:db") {
      /*
       * Run some tests which require MySQL, and assume that it is
       * available on the host name `db`
       */
      sh 'make check'
    }
  }
}
```



Librairies partagées



Introduction

Pipeline permet la création de **librairies partagées** pouvant être définies dans des dépôts de sources externes et chargées lors de l'exécution d'une Pipelines.

Une librairie est constituée de fichiers Groovy



Étapes de mise en place

La mise en place consiste en :

- 1) Créer les scripts groovy en respectant une arborescence projet et committer dans un dépôt
- 2) Définir la librairie dans Jenkins
Administrer Jenkins → Shared Libraries :
 - Un nom
 - Une méthode de récupération
 - Une version par défaut
- 3) L'importer dans un projet en utilisant l'annotation ***@Library***



Code groovy

Différents types de codes peuvent être développés dans une librairie :

- Classes groovy classique, définissant des structures de données, des méthodes.
Pour les utiliser, il faudra les instancier ;
Pour interagir avec les variables de la pipeline (env par exemple), il faudra les passer en paramètre.
Utilisable dans pipeline script
- Définir des variables globales. Jenkins les instancie automatiquement comme singleton et elles apparaissent dans l'aide.
Utilisable dans pipeline script
- Définir des nouvelles steps. Idem variable globale + mise à disposition de la méthode *call()*
Dans ce cas, utilisable en script et déclaratif



Structure projet

```
(root)
+- src                                     # Classes Groovy classiques
|   +- org
|       +- foo
|           +- Bar.groovy                # Classe org.foo.Bar
+- vars
|       +- foo.groovy                    # Variable globale 'foo'
|       +- foo.txt                       # Aide pour la variable 'foo'
+- resources                             # Fichiers ressources
|       +- org
|           +- foo
|               +- bar.json              # Données pour org.foo.Bar
```



Exemple Code classique

Fichier *src/org/foo/Zot.groovy*

```
package org.foo
```

```
def checkoutFrom(repo) {  
    git url: "git@github.com:jenkinsci/${repo}"  
}  
return this
```

Utilisation dans une pipeline

```
def z = new org.foo.Zot()  
z.checkoutFrom(repo)
```



Variable globale

Fichier ***vars/log.groovy***

```
def info(message) {  
    echo "INFO: ${message}"  
}  
  
def warning(message) {  
    echo "WARNING: ${message}"  
}
```

Utilisation dans pipeline declarative :

```
@Library('utils') _  
  
pipeline {  
    agent none  
    stage ('Example') {  
        steps {  
            script {  
                log.info 'Starting'  
                log.warning 'Nothing to do!'  
            } } } }  
}
```



Nouvelle step

Fichier *vars/sayHello.groovy*

```
def call(String name = 'human') {  
    // N'importe quelle steps peut être appelé dans ce bloc  
    // Scripted Pipeline  
    echo "Hello, ${name}."  
}
```

Utilisation

```
sayHello 'Joe'
```



Définition de librairies

Les librairies une fois développées peuvent être installées de différentes façons :

- **Global** Pipeline Libraries :
Manage Jenkins → Configure System → Global Pipeline Libraries
- **Folder** : Une librairies peut être définies au niveau d'un dossier
- Certains **plugins** ajoute des façons de définir des librairies.
Ex : *github-branch-source*



Utilisation des librairies

Les librairies marquées « ***Load Implicitly*** » sont directement disponibles.

=> Les classes et les variables définies sont directement utilisables

Pour les autres, le Jenkinsfile doit explicitement les charger en utilisant l'annotation ***@Library***

Depuis la version 2.7, le plugin *Shared Groovy Libraries* permet de définir une ***step*** « ***library*** » qui charge dynamiquement la librairie.


- Avec cette méthode, les erreurs ne sont pas détectées à la compilation.



Option Load Implicitly

Global Pipeline Libraries


Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.



Library


Name

my-shared-library




Default version

master




Load implicitly

☐




Allow default version to be overridden

☒



Retrieval method

☒ Modern SCM





Exemples d'usage

-- Annotations

```
@Library('my-shared-library') _  
/* Avec une version, branch, tag, ou autre */  
@Library('my-shared-library@1.0') _  
/* Plusieurs librairies */  
@Library(['my-shared-library', 'otherlib@abc1234'])  
/* Typiquement devant la classe importée */  
@Library('somelib')  
import com.mycorp.pipeline.somelib.UsefulClass
```

-- Plugin

```
library('my-shared-  
    library').com.mycorp.pipeline.Utils.someStaticMethod()
```



Récupération de la librairie

La meilleure façon de référencer la librairie est d'utiliser un plugin de SCM supportant l'API **Modern SCM** (Supporté par Git, SVN)

Cela se fait :

- via la page d'administration pour les librairies globales
- Via les options de *@Library*
- Ou dynamiquement :

```
library identifier: 'custom-lib@master', retriever:  
modernSCM( [$class: 'GitSCMSource', remote:  
'git@git.mycorp.com:my-jenkins-utils.git',  
credentialsId: 'my-private-key'] )
```



Librairies de tiers

Il est possible également de charger une librairie à partir du dépôt Maven Central en utilisant l'annotation **@Grab**

```
@Grab('org.apache.commons:commons-math3:3.4.1')
import org.apache.commons.math3.primes.Primes
void parallelize(int count) {
    if (!Primes.isPrime(count)) {
        error "${count} was not prime"
    }
    // ...
}
```

Exploitation d'un serveur Jenkins

Sécurité / Utilisateurs et permissions
Monitoring du serveur
Sauvegarde, archivage et migration
Jenkins CLI et Rest API



Introduction

Les tâches d'exploitation d'un serveur Jenkins sont principalement :

- La sécurité
- La sauvegarde
- Surveillance de l'utilisation disque
- Comment archiver les jobs ou les migrer d'un serveur à un autre serveur
- Surveillance de la charge => dimensionnement de l'architecture maître/esclave



Sécurité

Utilisateurs et autorisations



Introduction

Jenkins supporte plusieurs modèles de sécurité et s'intègre avec différents types d'annuaire

La mise en place de la sécurité s'effectue en plusieurs étapes :

- 1.Activation de la sécurité
- 2.Spécification de la base utilisateurs
- 3.Définition des autorisations



Activation

Sur la page de configuration principale, cocher la case
« *Activer la sécurité* » qui fait apparaître d'autres champs

Si une erreur de manipulation empêche l'accès au serveur,
modifier directement le fichier *config.xml* dans le répertoire
Jenkins et redémarrer le serveur :

```
<hudson>
```

```
<version>1.391</version>
```

```
<numExecutors>2</numExecutors>
```

```
<mode>NORMAL</mode>
```

```
<useSecurity>true</useSecurity>
```

```
...
```



Base utilisateurs

La base utilisateurs peut prendre la forme de :

- Base de données interne Jenkins.
(Dans ce cas pas de notion de groupe d'utilisateurs)
- Annuaire LDAP, Microsoft Active Directory
- Utilisateurs et groupe Unix
- Servlet Container (*Tomcat*)

Le contenu de l'annuaire est visible via la page *People*



Base Jenkins

Les utilisateurs sont gérés via la page

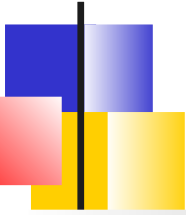
Manage Jenkins → Manage Users

L'administrateur peut configurer Jenkins afin de permettre aux utilisateurs anonymes de se créer un compte (signup)

Jenkins ajoute automatiquement les utilisateurs du SCM lorsqu'ils committent. Leur identifiant est associé au build qu'ils ont provoqué

- Cependant, Le fait d'être committers ne donne pas forcément le droit de se logger sur Jenkins. (Il n'a pas forcément de password « Jenkins »)

Fiche utilisateur



Jenkins [john](#) | [log out](#)

[Jenkins](#) » [John Smart](#)

- [People](#)
- [Status](#)
- [Builds](#)
- [My Views](#)
- [Configure](#)

Your name ?

Description ?

Password

Password:


Confirm Password:

E-mail

E-mail address
Your e-mail address, like joe.chin@sun.com

My Views

Default View
The view selected by default when navigating to the users private views





Autorisations

Différents modèles d'autorisation sont disponibles :

- Tout le monde peut tout faire (Pas de sécurité)
- Les utilisateurs loggés peuvent faire tout
- Sécurité matricielle globale
- Sécurité matricielle par projets



Sécurité matricielle









La sécurité matricielle permet d'assigner différents droits à différents utilisateurs avec une approche par rôle (administration, gestionnaire de jobs, architecte système, ...)

1. La première chose lorsque l'on met en place la sécurité matricielle est de créer un administrateur (pas nécessairement un utilisateur du SCM)
2. Ensuite, activer la sécurité matricielle et donner à l'administrateur tous les droits
3. Se logger avec l'administrateur et définir les permissions pour les utilisateurs ou groupes

L'utilisateur anonyme représente un utilisateur non loggé

Sécurité matricielle

Matrix-based security

User/group	Overall		Slave		Job							Run		View			SCM		
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag	
 administrator	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
 bob	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
 joe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
 kate	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Anonymous	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

User/group to add:

Add



Permissions (1)

Permissions globales :

- Administration : Configuration Jenkins, toutes opérations
- Read : Accès en consultation à toutes les pages
- Run scripts : Exécution de scripts
- Upload plugin : Installation de plugin
- Configure Update Center : Configuration du centre de mise à jour

Slave : permissions concernant les nœuds esclaves

- Build : Exécuter des jobs sur les nœuds
- Configure : Créer et configurer des nœuds esclaves
- Delete : Supprimer des nœuds
- Create : Créer des nœuds
- Disconnect / connect : Se connecter sur les nœuds



Permissions (2)

Job : Permissions sur les jobs

- Create : Créer un nouveau job
- Delete : Supprimer un job existant
- Configure : Mise à jour de jobs existants
- Read : Consultation des jobs
- Build : Démarrer un job
- Workspace : Visualisation et téléchargement de l'espace de travail d'un build job
- Release : Démarrer une release Maven pour un projet configuré avec le plugin M2Release



Permissions (3)

Run : Permissions relatives à des builds de l'historique

Delete : Supprimer un build de l'historique

- Update : Mise à jour de la description et d'autres propriétés dans l'historique. (Une note concernant l'échec d'un build par exemple)

View : Permissions relatives aux vues

- Create : Créer une nouvelle vue
- Delete : Supprimer une vue existante
- Configure : Configurer une vue existante

SCM : Permissions relatives au SCM

- Tag : Créer un nouveau tag dans le tag

Others : Rubrique dépendant des plugins installés

- Promote : Permet de promouvoir un build manuellement

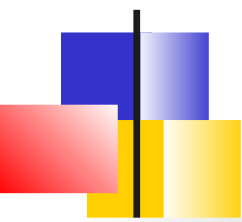


Sécurité sur les projets

La sécurité matricielle s'appliquant de façon globale peut être surchargée par des permissions définies projet par projet

- "*Enable project-based security*" dans l'écran de configuration principale
- Dans la configuration projet, activer également "*Enable project-based security*" et surcharger les permissions globales

Les permissions sont cumulatives, les permissions globales ne peuvent donc pas être révoquées au niveau d'un projet



Sécurité basée sur les rôles








Une autre alternative est d'utiliser le plugin « **Role Strategy** » qui permet de définir des rôles globaux ou projet puis d'assigner les utilisateurs sur les rôles

Les rôles sont associés à des projets via une expression régulière



Assign Roles

Global roles

User/group	admin	read-only	
 administrator	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
 authenticated	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
 johnsmart	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	

User/group to add

Add

Project roles

User/group	deployment-developer	game-of-life-developer	game-of-life-run-build	production-deployment	uat-deployment	
 bob	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
 joe	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 kate	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
 rob	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

User/group to add

Add

Save



Audit et traces

Il est possible de garder des traces de toutes les actions de configuration grâce au plugin « *Audit Trail* »

- Les traces peuvent être écrites dans un fichier, sur la console, dans un syslog

Le plugin « *JobConfigHistory* » permet de garder une historique des configurations Jenkins et de comparer 2 versions entre elles



Gestion des créden*t*iels

Jenkins nécessite de nombreux créden*t*iels afin de s'authentifier sur les outils associés (SCM, ssh, Serveurs LDAP, ...)

Un administrateur système peut configurer des créden*t*iels pour une utilisation dédiée par Jenkins

- Des scopes sont associés aux créden*t*iels et limitent ainsi leur utilisation
- Les jobs et les pipelines peuvent ensuite avoir accès à ces créden*t*iels via des Ids

Ces fonctionnalités sont apportées par le plugin *Credentials Binding Plugin*



Scopes

Les créidentiels stockés par Jenkins peuvent être utilisés :

- Globalement (partout dans Jenkins)
- Par un projet spécifique (et ses sous projets pour un projet Dossier par exemple)
- Par un utilisateur Jenkins particulier



Types de crédits

Jenkins peut stocker des crédits de type suivant :

- Texte secret : comme un token par exemple (exemple Token GitHub),
- Username et password : Traité comme des composants séparé ou comme une chaîne séparé avec un :
- Secret file : Une chaîne secrète stockée dans un fichier
- Un utilisateur SSH avec sa clé privé
- Un certificat de type PKCS#12 et un mot de passe optionel
- Certificat d'un hôte Docker



Usage disque : surveillance et sauvegarde



Données d'historique

L'historique de build occupe beaucoup d'espace disque. Jenkins analyse l'historique lorsqu'il charge une configuration de projet
=> plus l'historique est long, plus long dure le chargement

La méthode la plus simple pour garder une taille de disque raisonnable est de limiter le nombre de build conservés dans l'historique


Une configuration avancée permet de garder le XML et de ne pas conserver les artefacts



Plugin Disk Usage

Ce plugin enregistre la taille disque utilisé par les projets.

Les rapports générés permettent d'isoler les projets qui occupent un trop grand espace disque


 [Back to Dashboard](#)

Disk usage

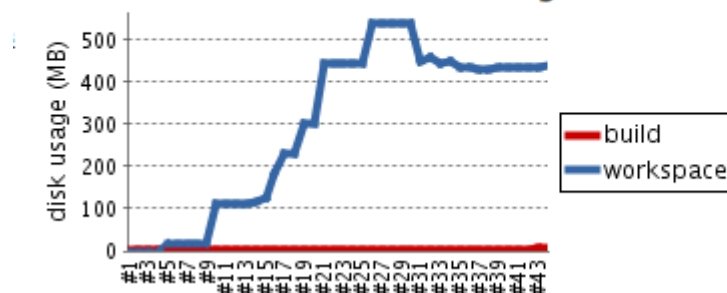
Builds: 167MB **Workspace:** 1GB

Project name	Builds	Workspace
ci-with-hudson-book-default	88MB	1GB
thucydides-code-quality	71MB	17MB
thucydides-sonar	6MB	8MB
thucydides-default	2MB	8MB
Total	167MB	1GB

Disk usage is calculated each 360 minutes. If you want to trigger the calculation now, click on the button.

 **Disk Usage:** Workspace 442MB, Builds 25MB

Disk Usage Trend





Projet Maven

Les projets Maven sont par défaut très gourmands en espace disque

Les jobs archivent automatiquement les artefacts

L'option “Disable automatic artifact archiving” permet de modifier le comportement par défaut



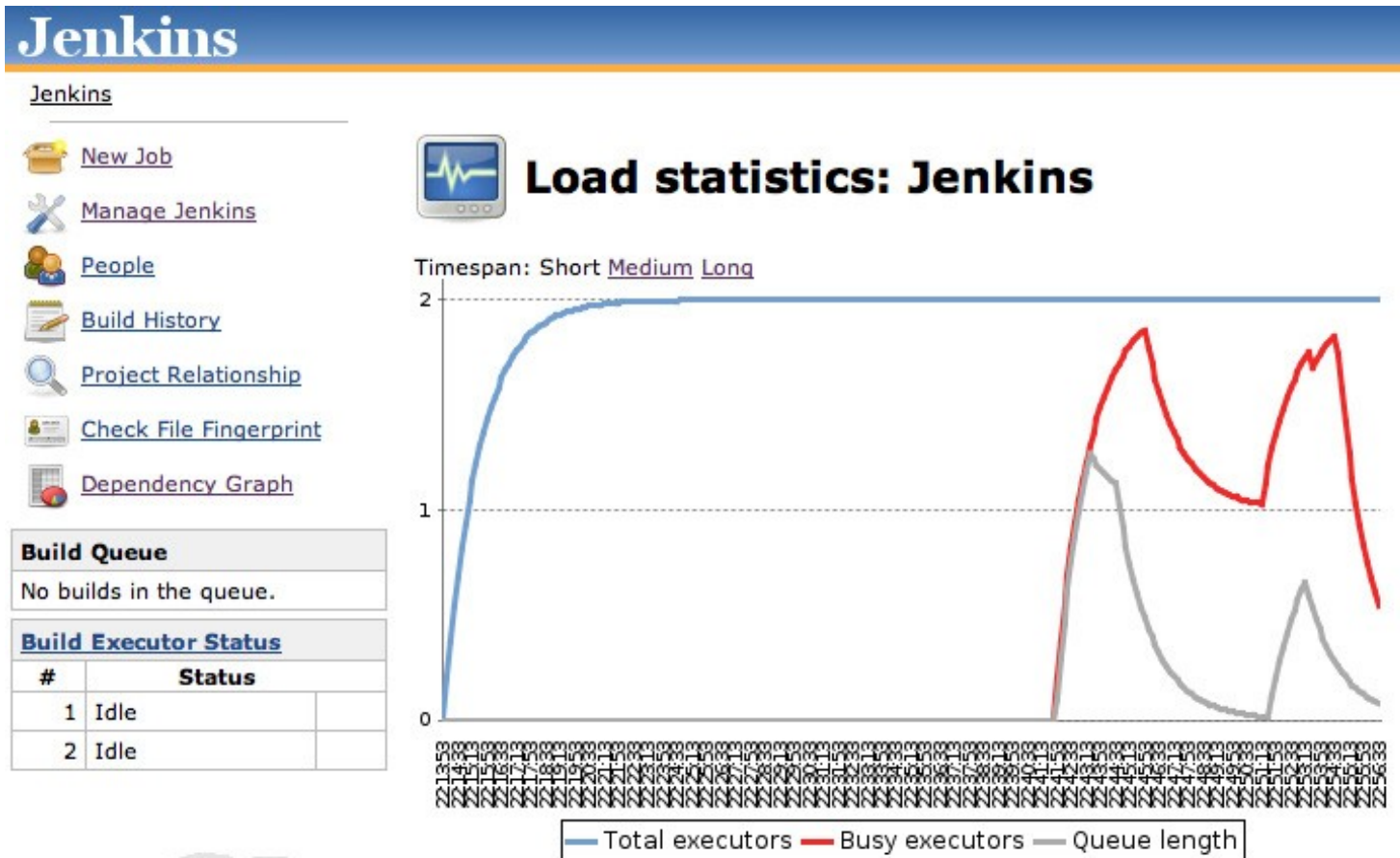
Surveillance de la charge

Jenkins surveille l'activité du serveur. Sur la page de configuration, le lien « *Statistiques d'utilisation* » permet d'afficher un graphique de la charge du serveur maître. Ce graphique présentent 3 mesures :

- Le **nombre total d'exécuteurs** qui inclut les exécuteurs du nœud maître et des nœuds esclave actifs
- Le **nombre d'exécuteurs occupé** à exécuter des builds. Si tous les exécuteurs sont constamment occupés, il vaut mieux ajouter des exécuteurs sur les nœuds esclave
- La **longueur de la file** représente le nombre de jobs qui attendent qu'un exécuteur soit libre

Il est possible d'avoir le même graphique pour un nœud esclave particulier.

Graphique de charge



Monitoring plugin

Statistics of JavaMelody monitoring taken at 6/5/10 11:22 AM from 5/31/10 10:56 AM on

(Hudson)

[Update](#) [PDF](#) [Online help](#) Choice of period: [Day](#) [Week](#) [Month](#) [Year](#) [All](#) [Customized](#)





Sauvegarde, archivage et migration



Sauvegarde

La configuration la plus simple consiste à sauvegarder périodiquement le répertoire ***JENKINS_HOME***

Le répertoire contient toutes les configurations de build, l'historique des builds.

La sauvegarde peut se faire lorsque Jenkins s'exécute



Sauvegarde

Le répertoire *JENKINS_HOME* peut contenir beaucoup de données

Si cela devient un problème, il est possible de gagner un peu de place en ne sauvegardant pas les répertoires pouvant être recréés :

- ***\$JENKINS_HOME/war*** : Le fichier WAR non compressé
- ***\$JENKINS_HOME/cache*** : les outils téléchargés
- ***\$JENKINS_HOME/tools*** : Les outils extraits



Sauvegarde

On peut être encore plus sélectif dans les données à sauvegarder

Sous le répertoire *jobs*, il y a un sous-répertoire pour chaque build contenant 2 sous-répertoires :

- Il n'est pas nécessaire de sauvegarder le répertoire ***workspace***
- Dans le répertoire ***builds***, les données d'historique sont stockées. Les artefacts générés sont eux stockés dans le répertoire *archive* et peuvent prendre beaucoup de place. Il n'est peut-être pas nécessaire de les sauvegarder



Thin Backup Plugin

Si seule la configuration doit être sauvegardée le plugin « *Thin Backup* » permet de planifier des backup incrémentaux ou complet des fichiers de configuration



Configuration Backup/Restore

Backup Configuration

Backup settings

Backup directory	<input type="text" value="/var/data/backup/thin"/>	?
Backup schedule for full backups	<input type="text" value="0 0 * * 1-5"/>	?
Backup schedule for differential backups	<input type="text" value="0 * * * 1-5"/>	?
Max number of stored full backups	<input type="text" value="40"/>	?
<input checked="" type="checkbox"/> Clean up differential backups		?

Save

Restore Configuration

Restore options

restore backup from [?](#)

Restore



Archivage

Une autre façon de gagner de l'espace disque sur le serveur consiste à archiver les projets qui ne sont plus actifs

L'archivage permet de pouvoir facilement restaurer un projet si nécessaire

L'archivage consiste à déplacer le répertoire projet en dehors du répertoire *job* et de le compresser



Migration

Il est facile de migrer les instances de projet entre serveur Jenkins

Il suffit de déplacer le répertoire projet ; le serveur Jenkins n'a pas besoin de redémarrer, il suffit de recharger la configuration à partir du disque



Jenkins CLI et Rest API



CLI

La console CLI est téléchargeable sur votre installation Jenkins.

```
wget http://localhost:8080/jnlpJars/jenkins-cli.jar
```

La syntaxe des commandes est ensuite :

```
java -jar jenkins-cli.jar -s <jenkins_http_url> <options>  
<commande>
```

Par exemple, pour afficher les commandes disponibles :

```
java -jar jenkins-cli.jar -s http://localhost:8080/jenkins help
```



Sécurité

Si le serveur est sécurisé il faut utiliser un mécanisme d'authentification :

- Ouvrir un port *ssh* et utiliser une paire clé privé/publique
- Utiliser un jeton préalablement généré dans la page de configuration d'un utilisateur particulier



Exemples

#Authentication via token et liste des jobs

```
java -jar jenkins-cli.jar -s http://localhost:8080/ -auth  
admin:119703af95554ff2891bf4dc5006f5e5af \  
list-jobs > jobs.txt
```

Backup then restore

```
java -jar jenkins-cli.jar -s http://localhost:8080/ get-job test2 >  
config.xml
```

```
java -jar jenkins-cli.jar -s http://192.168.111.110:8080/ \  
create-job test3 < config.xml
```

Installation de plugin

```
java -jar jenkins-cli.jar -s http://localhost:8080/jenkins install-plugin \  
http://updates.jenkins-ci.org/latest/build-monitor-plugin.hpi -restart
```



API Rest

Jenkins offre également une API Rest basée sur XML, JSON ou Python.

<https://wiki.jenkins.io/display/JENKINS/Remote+access+API>

L'API peut être utilisée pour :

- Récupérer de l'information de Jenkins (jobs, plugins, tools, ...)
- Démarrer un job
- Créer, copier des jobs

La sécurité est basée sur un token qui se configure sur la page de configuration d'un utilisateur



API pour chaque objet

IL n'y a pas de point d'entrée unique à l'API REST de Jenkins

La plupart des objets Jenkins fournissent une API d'accès via l'URL **`/.../api/`** où "..." est l'objet concerné.

Par exemple :

- Pour le job myJob
`localhost:8080/job/myJob/api`
- Pour le dossier myFolder
`localhost:8080/job/myFolder/api`
- Pour le build n
`localhost:8080/job/myJob/n/api`



Exemple

#Sauvegarde d'un job

```
curl "http://localhost:8080/job/test2/config.xml" > configtst2.xml
```

Création de job

```
curl -X POST -H "Content-Type:application/xml" -d @config.xml \  
http://localhost:8080/createItem?name=test6
```

Déclenchement

```
curl -X POST http://localhost:8080/job/test2/build \  
--data token=0123456789abcdefghijklmnopqrstuvwxyz
```

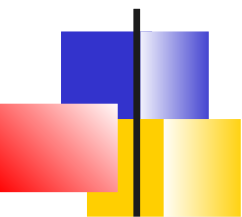
Déclenchement avec paramètres

```
curl -X POST JENKINS_URL/job/JOB_NAME/build \  
--user USER:TOKEN \  
--data-urlencode json='{ "parameter": [{ "name": "id", "value": "123"},  
{"name": "verbosity", "value": "high"}] }'
```



Références

Jenkins : The Definitive Guide
Jenkins Wiki
Cloudbees



Annexes



SCM



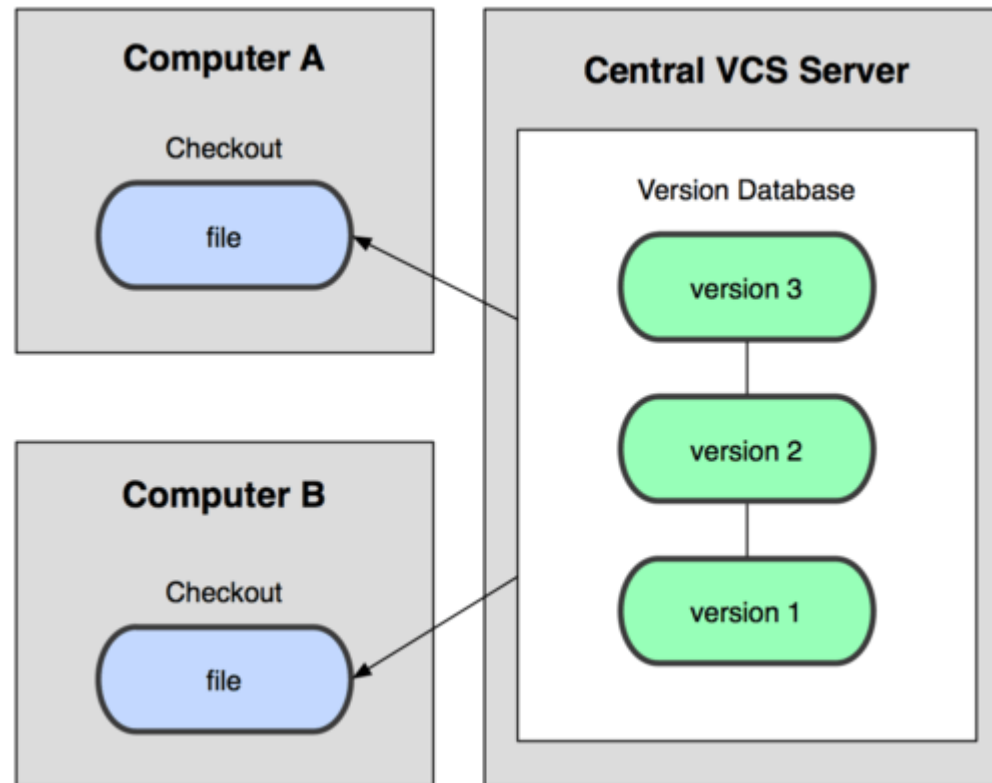
SCM

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

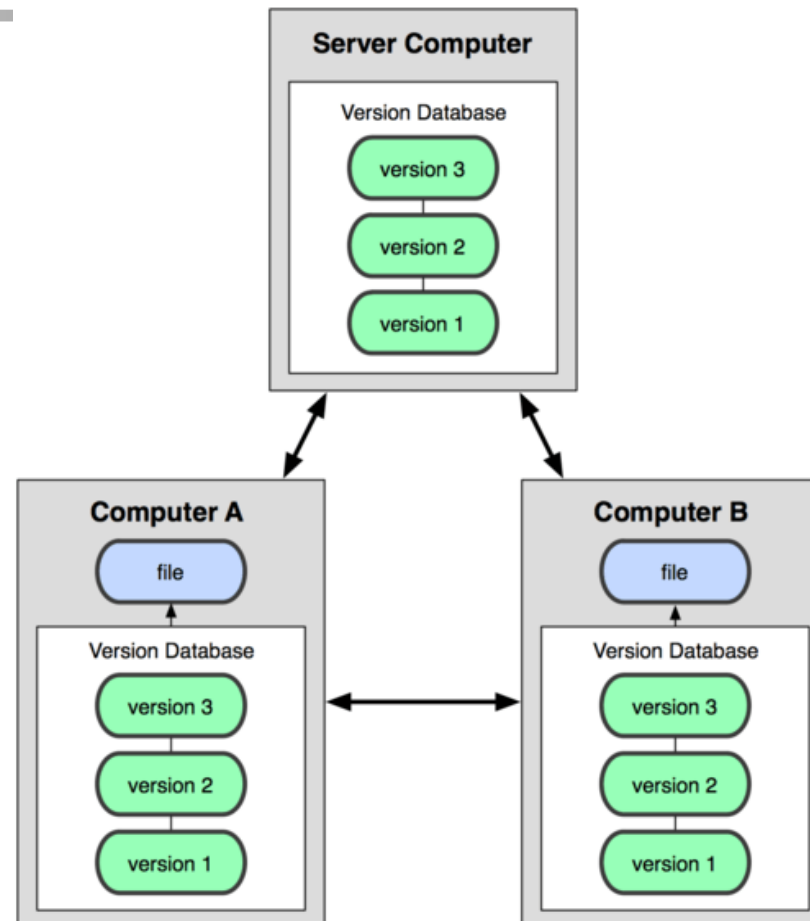
Le système permet :

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

SCM centralisés : SVN, CVS, Perforce



SCM distribués : Git, Bitbucket





Principales opérations

clone, copy : Recopie intégrale du dépôt

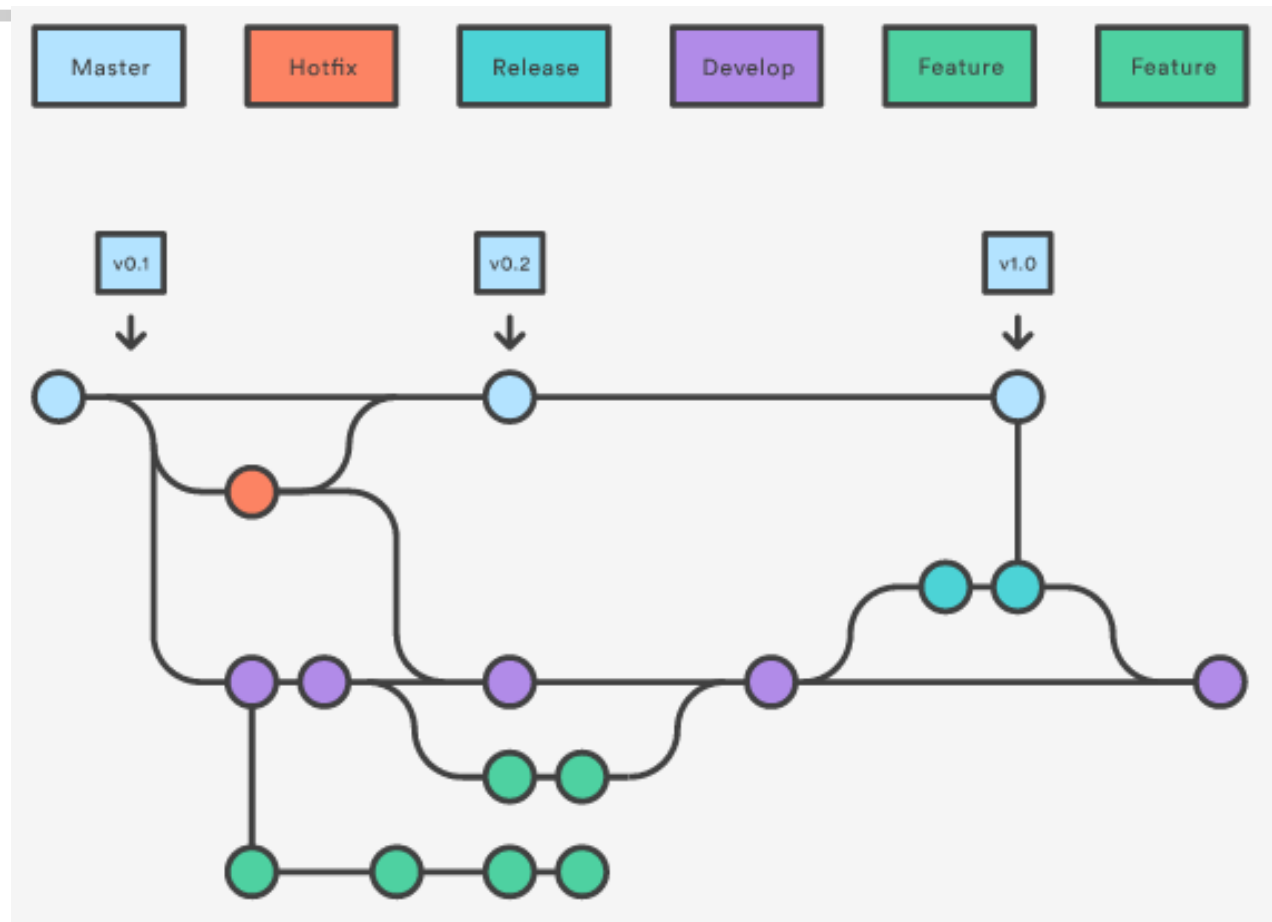
checkout : Extraction d'une révision particulière

commit : Enregistrement de modifications de source

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

log : Accès à l'historique

Gestion des multiples branches (Exemple Gitflow)





Tests et Métriques Qualité

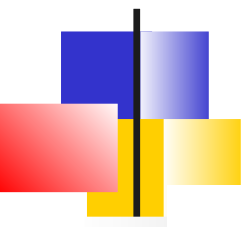


Introduction

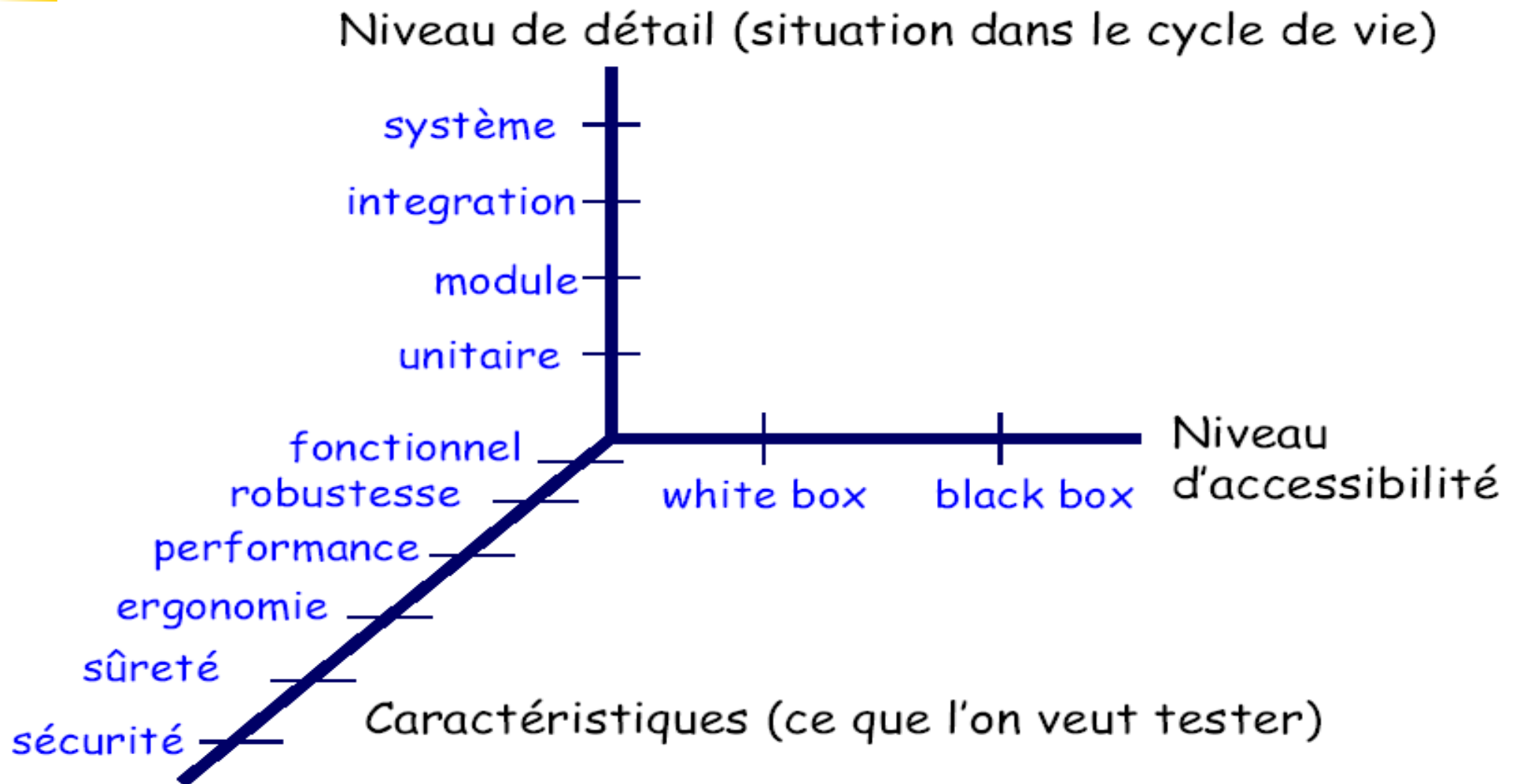
Les tests permettent d'avoir confiance en un système

Ils sont indispensables dans l'approche CI et dans les approches méthodologiques modernes (XP programming, Agilité, DevOps)

Ce sont les tests qui permettent d'obtenir des métriques qui donne une vision objective à l'état du projet. (Performance, couverture de test, nombre de bugs,)



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 ?



Les frameworks de test

Tests Unitaires : xUnit, Junit, TestNG, PHPUnit, cppUnit, Mockito

Tests d'intégration : Injection de dépendance, Serveurs embarqués, Initialisation BD

Tests fonctionnels : Selenium, WebTest, HttpUnit, JMeter, Protractor

Test d'acceptance : Concomber, JBehave

Test de charge : JMeter, Gatling



Qualité : la Norme ISO-9126

ISO-9126 est le premier standard qui a proposé un modèle de référence pour la **qualité** des logiciels

- Il normalise a grand nombre de critères qualité
- Et plus particulièrement des métriques et la façon de les mesurer

La norme a été ensuite complétée par la série de standards 25010 dénommée ***Software Product Quality Requirements and Evaluation (SQuaRE)***.

ISO 25010

SOFTWARE PRODUCT QUALITY

Functional Suitability

- Functional Completeness
- Functional Correctness
- Functional Appropriateness

iso25000.com

Performance Efficiency

- Time Behaviour
- Resource Utilization
- Capacity

Compatibility

- Co-existence
- Interoperability

Usability

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

Reliability

- Maturity
- Availability
- Fault Tolerance
- Recoverability

Security

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

Maintainability

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

Portability

- Adaptability
- Installability
- Replaceability



Outils Qualité

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

L'analyse intègre 2 aspects :

- Définitions des règles de codage du projet, détection des transgressions et estimation de la dette technique
- Calculs des métriques internes

Des *portes qualité* précisant les objectifs qualité pour un projet se configure dans l'outil



Porte qualité

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

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

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



Provisionnement de l'infrastructure



Introduction

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

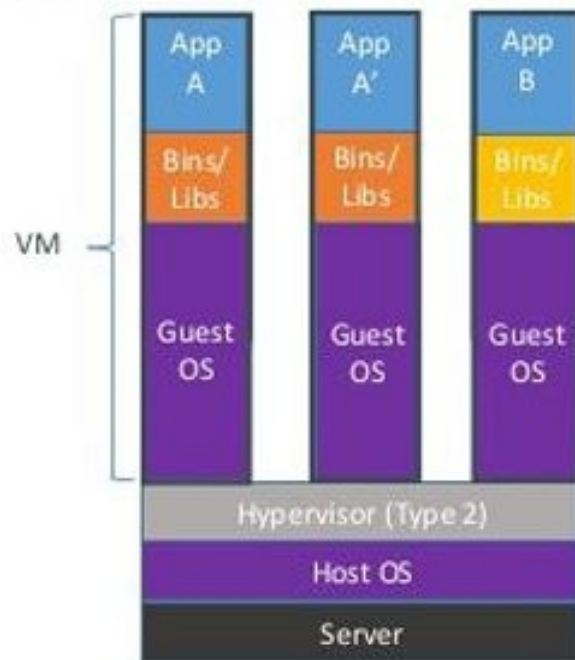
En cours de projet, l'infrastructure est affinée

- => Nécessité de pouvoir provisionner rapidement et à la demande l'infrastructure.

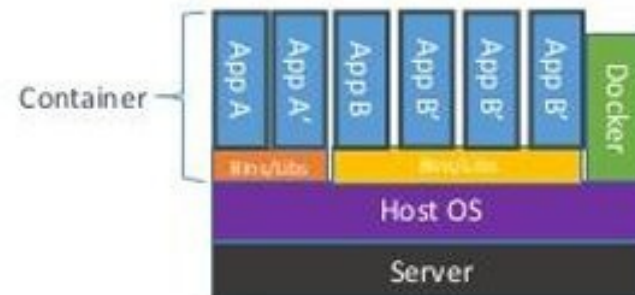
Cette difficulté est résolue par les avancées récentes dans les techniques de virtualisation , de containerisation et les offres Cloud.

Virtualisation et Containerisation

Containers vs. VMs



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





Docker et orchestrateurs

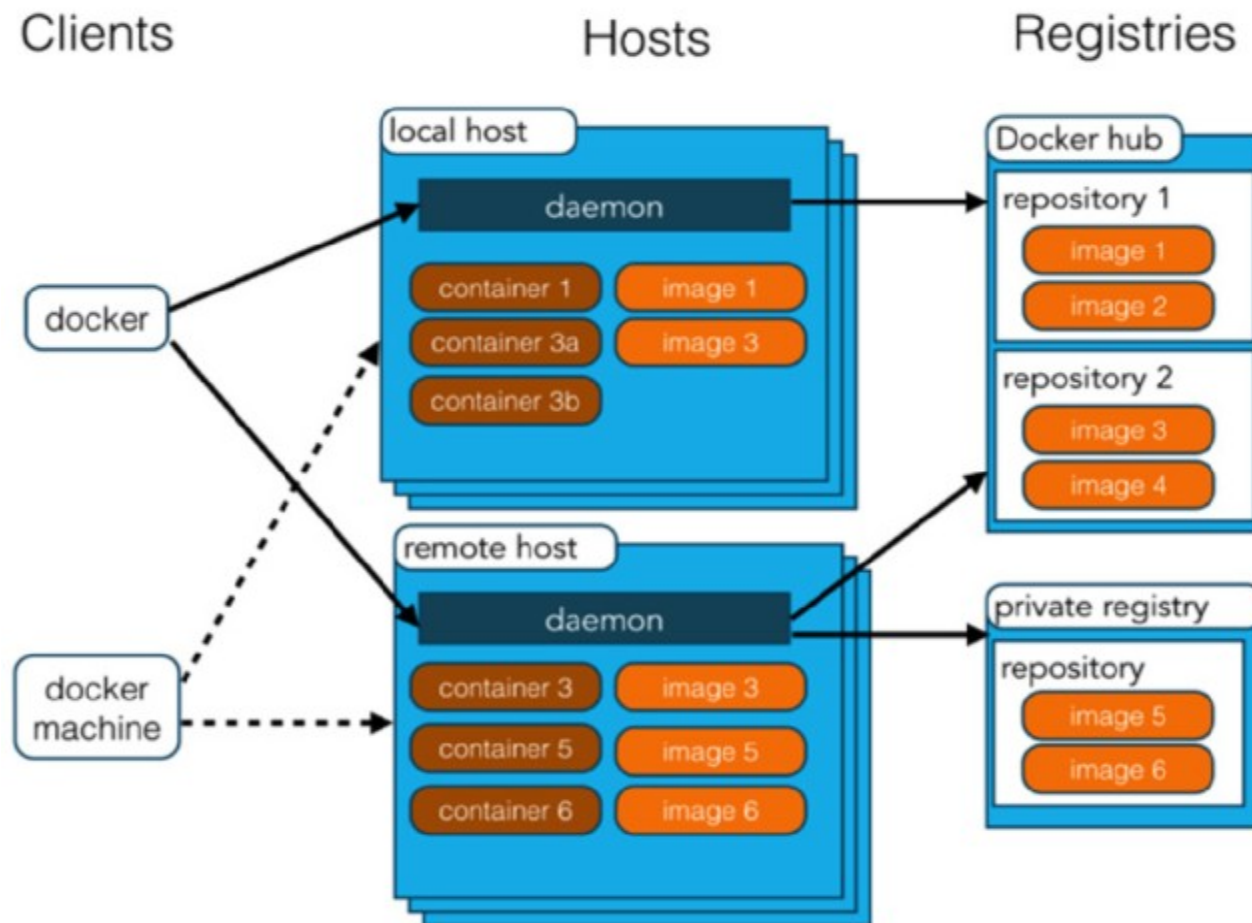
Les « conteneurs » ont été popularisés par **Docker**.

- Les développeurs peuvent définir de façon déclarative leur application en incluant l'infrastructure.
Docker est alors capable d'instancier les différentes instances applicatives sur un seul OS.

Des orchestrateurs de conteneurs comme **docker-swarm**, **Kubernetes** ou les **solutions cloud** permettent d'instancier à la demande des clusters de conteneurs qui collaborent.

- L'orchestration peut être statique ou dynamique (en fonction de la charge par exemple).

Docker architecture





Exemple DockerFile

FROM ubuntu

MAINTAINER Kimbro Staken

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

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

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

RUN apt-get update

RUN apt-get install -y nodejs

RUN mkdir /var/www

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

EXPOSE 8080

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



Exemple docker-compose

```
version: '3'
services:
  catalog:
    build: catalog
    networks:
      - back
    ports:
      - 8080:8080
  catalog-postgresql:
    image: postgres:9.6.5
    volumes:
      - catalog_db:/var/lib/postgresql
      - catalog_data:/var/lib/postgresql/data
    networks:
      - back
    environment:
      - POSTGRES_USER=catalog
      - POSTGRES_PASSWORD=catalog
    ports:
      - 5434:5432

volumes:
  catalog_data:
  catalog_db:

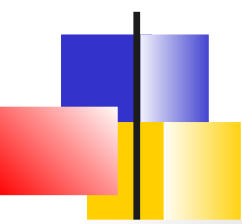
networks:
  back:
```



Offres Cloud

PaaS (Platform As A service) : Offre permettant de développer, exécuter et gérer les applications sans la complexité de mise en place de l'infrastructure. Se base sur la virtualisation et l'orchestration de conteneurs

- Amazon Web Services
- Google App Engine
- OpenShift de RedHat
- Microsoft Azure
- Cloud Foundry, Heroku, Digital Ocean



Jenkins et la containerisation

Les travaux les plus actifs de Jenkins sont l'intégration avec l'outil Docker. On peut distinguer 3 axes :

- Utiliser les images Docker pour les serveurs d'intégration
- Utiliser les images Docker comme des serveurs de build
- Dockeriser des configurations Jenkins



Premiers éléments de syntaxe



Interpolation de String

Jenkins a accès au mécanisme d'interpolation de chaîne de Groovy

Les chaînes de caractères peuvent être définies avec des simple ou double quotes :

```
def singlyQuoted = 'Hello'  
def doublyQuoted = "World"
```

Seules les doubles quotes supportent l'interpolation :

```
def username = 'Jenkins'  
echo "I said, Hello Mr. ${username}"
```



Variables d'environnement

Les variables d'environnement sont accessibles en lecture via :

```
// Declarative and Script //  
${env.BUILD_ID}
```

L'écriture d'une variable d'environnement est différent selon le type de syntaxe

- Directive ***environment*** (declaratif)
- Etape ***withEnv*** (script)



Accès aux variables d'environnement

```
// Declarative //
pipeline {
  agent any
  stages {
    stage('Example') {
      steps {
        echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
      }
    }
  }
}

// Script //
node {
  echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
}
```



Exemple écriture de variable d'environnement

```
// Declarative //
pipeline {
  agent any
  stages {
    stage('Example') {
      environment { DEBUG_FLAGS = '-g' }
      steps { echo "Running ${env.DEBUG_FLAGS}" }
    }
  }
}

// Script //
node {
  withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
  }
}
```




Paramètres

Les pipeline déclaratives supportent directement les paramètres fournis par les utilisateurs via la directive ***parameters***

Les pipelines scriptées implémentent les paramètres avec l'étape ***properties*** (voir Snippet Generator)

Si la pipeline a été configurée avec l'option *Build with Parameters*, les paramètres sont accessibles via la variable ***params***



Exemple Paramètres

```
// Declarative //
pipeline {
    agent any
    parameters {
        string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I
greet the world?')
    }
    stages {
        stage('Example') {
            steps { echo "${params.Greeting} World!" }
        }
    }
}

// Script //
properties([parameters([string(defaultValue: 'Hello', description: 'How should
I greet the world?', name: 'Greeting'))])
node {
    echo "${params.Greeting} World!"
}
```



Gestion des erreurs (déclaratif)

Les pipeline déclaratives supportent le traitement des erreurs via sa section **post** qui permet de déclarer des conditions sur l'issue de la phase : *always*, *unstable*, *success*, *failure* et *changed*

```
// Declarative //
pipeline {
  agent any
  stages { stage('Test') { steps { sh 'make check' } } }
  post {
    always {
      junit '**/target/*.xml'
    }
    failure {
      mail to: 'team@example.com', subject: 'The Pipeline failed :('
    }
  }
}
```



Gestion des erreurs

Les pipeline déclaratifs peuvent traiter les erreurs via la section **post** qui permet de déclarer plusieurs "post conditions" comme :

- *always, unstable, success, failure, et changed*

Les pipelines scriptées reposent sur le mécanisme de Groovy **try/catch/finally** .



Exemple déclaratif

```
// Declarative //
pipeline {
  agent any
  stages {
    stage('Test') {
      steps {
        sh 'make check'
      }
    }
  }
  post {
    always {
      junit '**/target/*.xml'
    }
    failure {
      mail to: team@example.com, subject: 'The Pipeline failed :('
    }
  }
}
```



Gestion des erreurs (script)

Les pipeline scriptées s'appuient sur les construction Groovy ***try/catch/finally*** pour le traitement des erreurs.

```
// Script //  
node {  
    /* .. snip .. */  
    stage('Test') {  
        try {  
            sh 'make check'  
        } finally {  
            junit '**/target/*.xml'  
        }  
    }  
    /* .. snip .. */  
}
```



Différents exécuteurs

Pipeline permet l'utilisation de différents agents ou exécuteurs durant l'exécution d'un build

Cela permet par exemple d'exécuter le même test sous différents OS



Affectation d'exécuteur

Il est possible de spécifier un label permettant de sélectionner un exécuteur particulier

```
// declarative
stage('Test on Linux') {
  agent { label 'linux' }
  steps { ... }
  post { ..}
}
// Scripted
stage('Test') {
  node('linux') {
    checkout scm
  }
}
```