

Jenkins Pipelines

David THIBAU – 2024

david.thibau@gmail.com



Agenda

Démarrage

- Plateforme de CI/CD
- Le projet Jenkins
- Installation
- Interface Utilisateur

Exécution des Jobs

- Configuration Système, Outils et plugins
- Nœuds
- Exécution des Jobs

Pipelines

Approche et concepts
Syntaxe déclarative et script
Groovy et syntaxe script
Librairies partagées

Container

- Docker,
- Kubernetes

Plugins CI/CD

- Publication des tests
- Intégration Sonar
- Intégration Ansible



Démarrage

Pipeline CI/CD

Le projet Jenkins

Installation

Interface Utilisateur



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

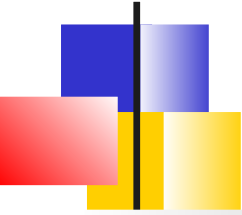
Si ce traitement échoue, l'outil notifie immédiatement les développeurs afin qu'ils **corrigent le problème ASAP**



En continu

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

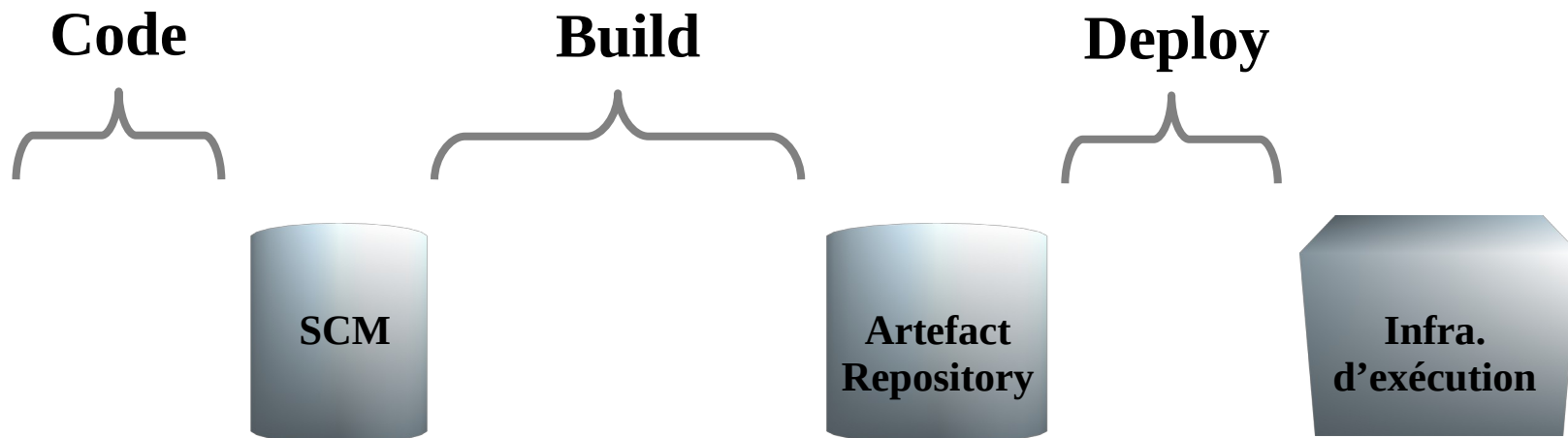
En fonction de leurs succès, l'application est déployée dans les différents environnements (intégration, staging, production)



Cycle de vie du code Build / Release / Run

La pipeline suit le cycle de vie du code.

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





Build is tests !

La construction de l'application consiste principalement à :

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



Pipelines

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

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



Distinction CI/CD





Outil de communication

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

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

Les métriques sont visibles de tous en toute transparence

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



Démarrage

Pipeline CI/CD

Le projet Jenkins

Installation et *JENKINS_HOME*

Interface Utilisateur



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



Atouts

- ✓ Facilité d'installation
- ✓ Interface web intuitive avec aide intégrée
- ✓ Très extensible et adaptable à des besoins spécifiques (Nombreux plugins opensource)
- ✓ Communauté large
- ✓ Release quasi-hebdomadaires ou LTS release (Long Term Support)
- ✓ S'appuie sur un langage de scripting pour exécuter les pipelines



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 :

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



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
```



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

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

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

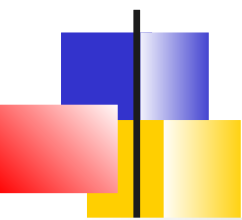
L'intégralité de l'état du serveur est stocké sous le répertoire *JENKINS_HOME*



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, créidentiels, 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

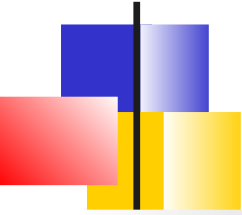


Structure du répertoire jobs

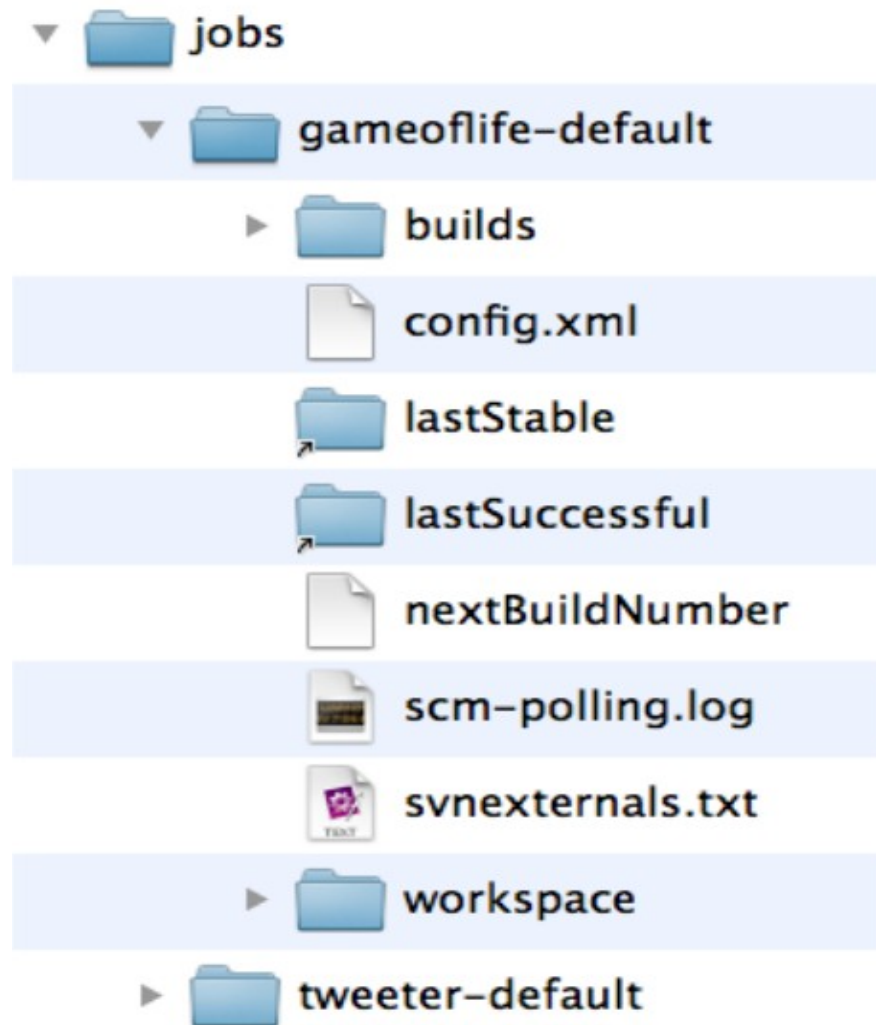
Le répertoire *jobs* contient un répertoire par projet Jenkins

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





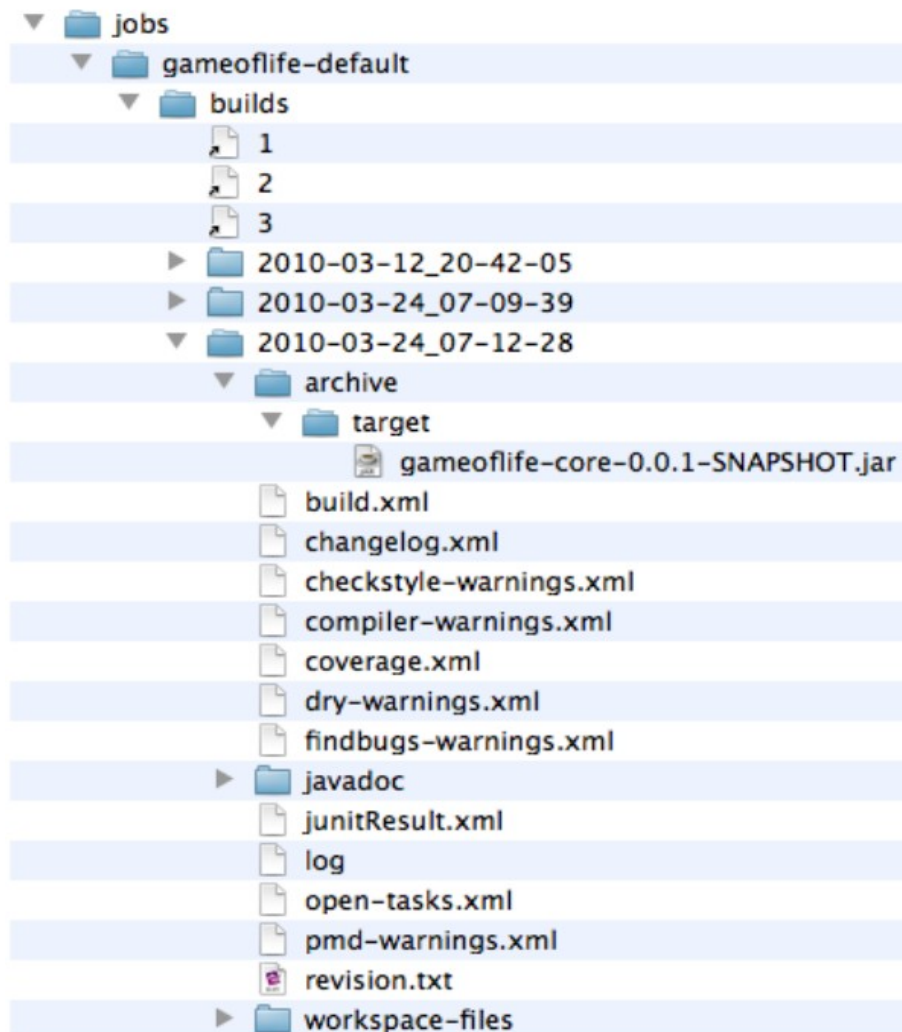
Répertoire build

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





Démarrage

Pipelines CI/CD
Le projet Jenkins
Installation et *JENKINS_HOME*
Interface Utilisateur



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.mymeedingsondemand.com:8080

rechercher S'identifier Créer un compte

Jenkins

Rafraichissement automatique

Ajouter une description

Tous +

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

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](#)

Test Result Trend


count


(just show failures) enlarge


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

Serveur, Outils, Plugins

Noeuds

Exécution des jobs



Point d'entrée

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

Les liens présents sont regroupées en catégorie :

- Configuration système : Intégration, Outils, Plugins, Agents
- Sécurité : Stratégie de sécurité, stockage de crédeniels, utilisateurs
- Monitoring : Statut, traces du serveur. Analyse de la charge
- Autres: Redémarrage, exécution de script



Configuration système

La configuration du système englobe :

- Configurer le système : Fonctionnement global, configuration du contrôleur, mail de l'administrateur, déclaration de serveur tiers ...
- Configuration des outils : Définition de chemins d'accès à des outils utilisés par les builds : JDK, Maven, SonarScanner, ...
- Gestion des plugins : Disponibilité, installation de plugin
- Gérer les nœuds et les clouds : Ajout/suppression d'agents permanents ou provisionné dynamiquement via le cloud
- Configuration As Code : Gérer la configuration de Jenkins via des fichiers



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



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



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



Configuration

Serveur, Outils, Plugins

Nœuds

Exécution des jobs



Introduction

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

- On peut mettre en place une ferme de serveurs (agents) afin de répartir la charge ou d'exécuter les jobs dans différents environnements

Jenkins a suivi les évolutions des technologies d'approvisionnement d'infrastructure et propose donc différentes alternatives :

- Serveur matériels ou VM pré-provisionnés
- VMs associés à des outils de gestion de conf
- Clouds privés ou publics utilisant la containerisation



Provisionnement

Le provisionnement des esclaves consiste à configurer une machine avec les outils de build, les comptes user et les services demandés par un projet.

Différentes alternatives pour le provisionnement sont également possibles :

- Provisionnement manuel ou par outils de gestion de conf
- Installation automatique d'outil
- Utilisation dynamique d'image défini dans la pipeline



Architecture Jenkins

Jenkins utilise une architecture **maître/esclave**

- Le nœud maître ou contrôleur
 - 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 ou agents 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.



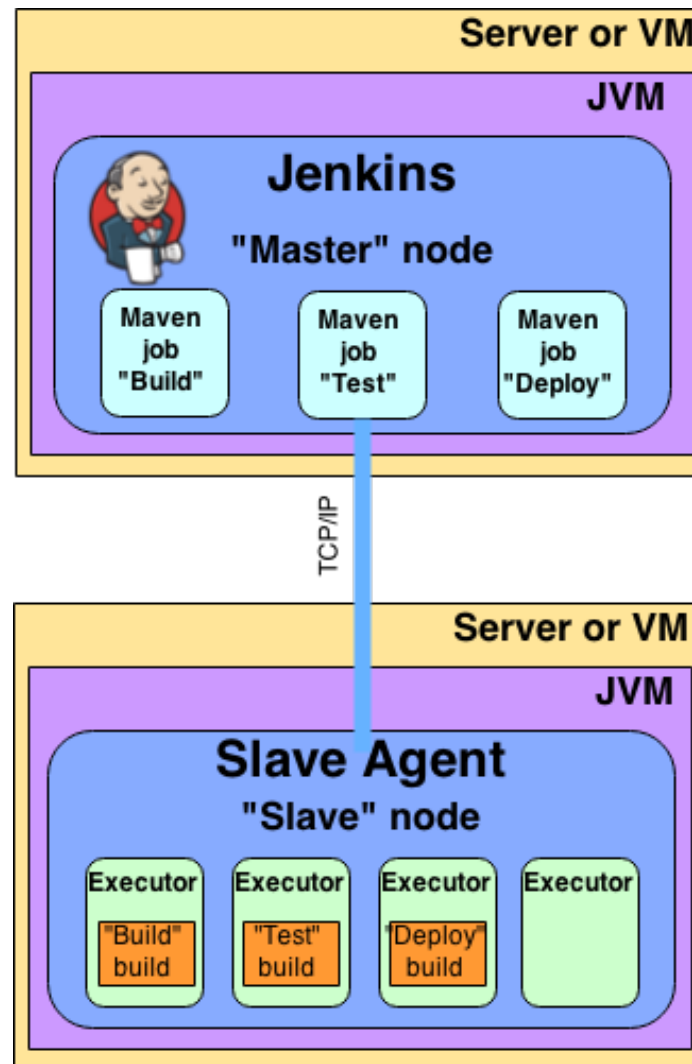
Nœud esclave

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, dans un container
- 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



Netflix 2012

1 master avec 700
utilisateurs

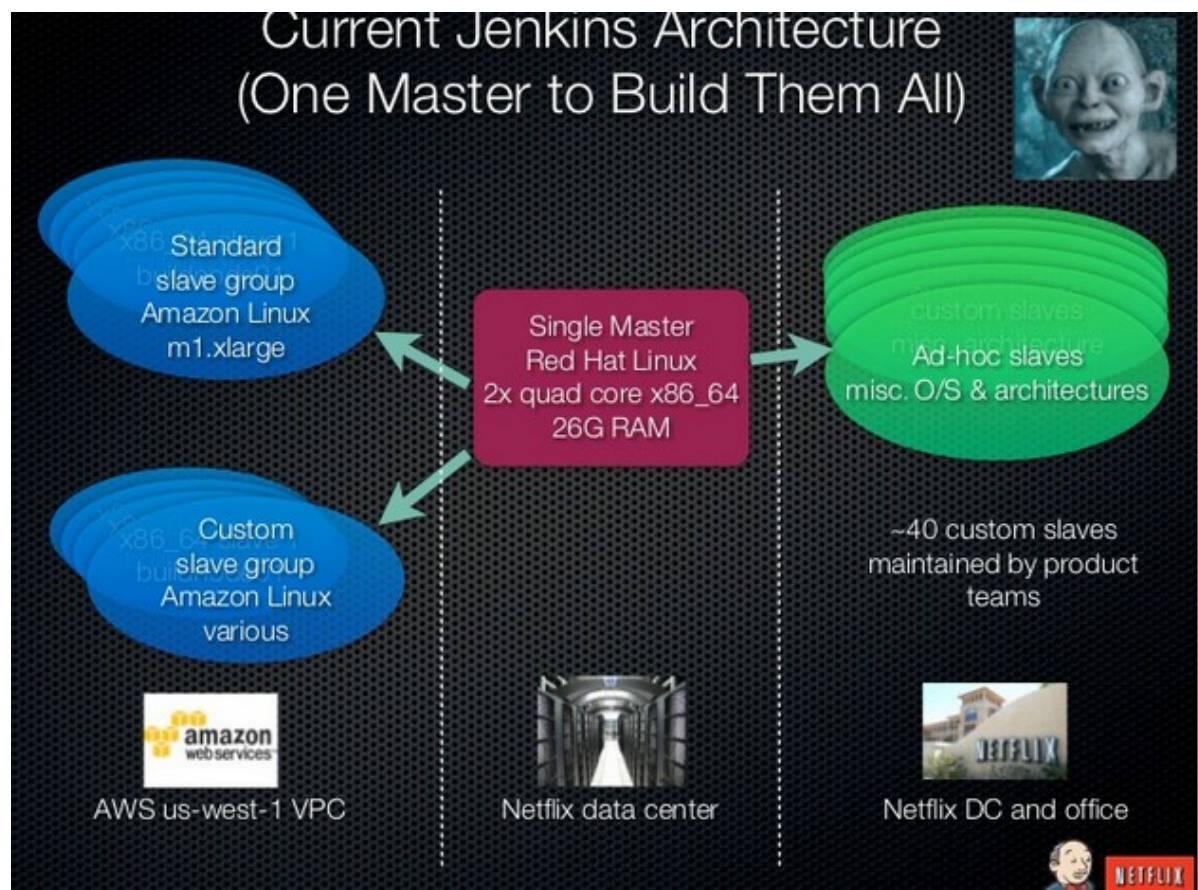
1,600 jobs :

- 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



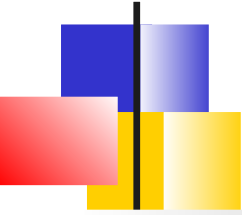


Type de nœud

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

Cela signifie que ce sont des exécutables toujours démarrés (serveur matériel ou VM)

D'autres plugins ajoutent des types de nœuds permettant par exemple le provisionnement dynamique d'esclaves
Dans les dernières versions, ces nœuds sont gérés via le menu *Cloud*



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 agents permanents

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 (JNLP)** / Windows ou Linux
- Le nœud esclave est installé comme **service Windows**



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



Labels/Tags des nœuds

Des **labels** peuvent être associés 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 agents

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



Agents sur le cloud

Il est également possible de provisionner des agents sur le cloud.

- Les agents n'existent que pendant l'exécution du job
- Le maître Jenkins s'y connecte via SSH ou JNLP (Windows)

Exemple de plugins

- Le plug-in **EC2** permet d'utiliser AWS EC2
- Le plugin **JCloud** permet d'utiliser les fournisseurs compatible *Jcloud*.
- Les plugins OpenStack, Kubernetes,



Configuration

Serveur, Outils, Plugins

Nœuds

Exécution des jobs



Types de jobs et Outils de build

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

Job FreeStyle : Script shell ou *.bat*
Windows

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

=> Éviter les espaces et les accents



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)



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



Déclencheurs

Il y a 4 façons de déclencher un job :

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

On peut également démarré un build via l'API Jenkins (REST ou CLI) :

<http://SERVER/job/PROJECTNAME/build?token=SECRET>



Étapes de build

Un job *freestyle* est 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 sont :

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



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



Statuts d'un job

L'exécution d'un job peut avoir différents statuts :

- **SUCCESS** : Tout s'est bien passé. Actions du build et actions post-build
- **UNSTABLE** : Des tests, des seuils qualité ont échoués
- **FAILURE** : Des actions ou des post-actions ont échoués
- **NOT_BUILT** : Le job ne s'est pas exécuté
- **ABORTED** : Le job a été interrompu



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



Introduction

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

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



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

Quelque soit l'approche, 2 syntaxes sont disponibles :

- Syntaxe déclarative
- Syntaxe script



Avantages de l'approche

Pipeline offre de nombreux avantages :

- *Build As Code* : Les pipelines implémentées par du code sont gérées par le SCM
=> Historique des révisions, branches, ...
- Le DSL supporte des pattern de workflow complexes (fork/join, boucle, ...)
- Les Pipelines peuvent s'arrêter et attendre une approbation manuelle, survivent au redémarrage de Jenkins
- Il est possible de profiter de toute la puissance d'un langage de script comme Groovy en particulier de librairies partagées
- Les plugins Jenkins permettent d'étendre le DSL en proposant de nouvelles fonctions facilitant l'intégration



Termes du DSL

Le DSL introduit plusieurs termes et concepts :

- **node ou agent** : Les travaux d'une pipeline sont exécutés dans le contexte d'un nœud ou agent.
 - Plusieurs nœuds peuvent être déclarés dans une pipeline.
 - Chaque nœud a son propre espace de travail
- **stage (phase)** : Une phase définit un sous ensemble de la pipeline.
Par exemple : "Build", "Test", et "Deploy".
Les phases améliorent la compréhension et la visualisation de l'avancement de la pipeline
- **step (étape)** : Une tâche unitaire Jenkins.
Par exemple, exécuter un shell, publier les résultats de test



Exemple *Jenkinsfile*

```
pipeline {
  agent any

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




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



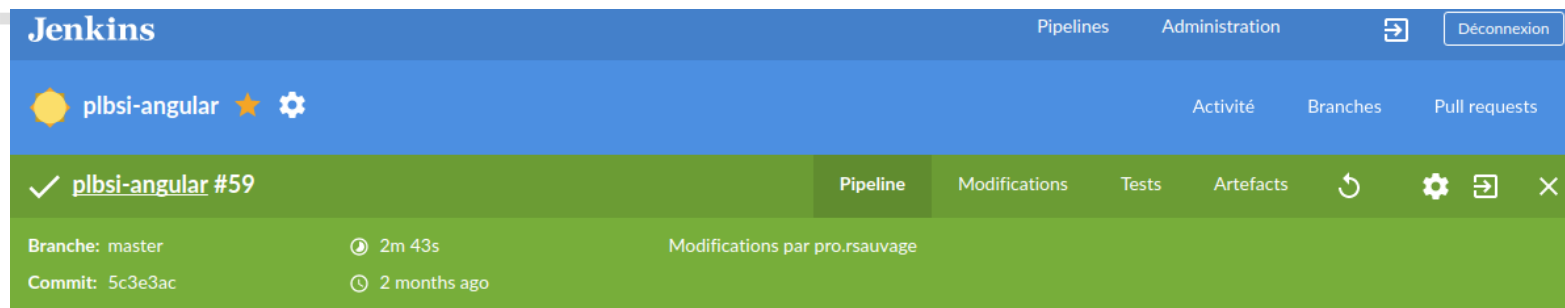
Blue Ocean

Le plugin ***Blue Ocean*** propose une interface utilisateur dédiée aux pipelines :

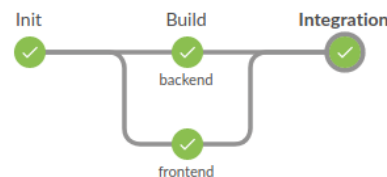
- Visualisation graphique des pipelines
- Éditeur graphique de pipeline
- Intégration des branches et pull-request

Cette interface cohabite avec l'interface classique

Exemple : Détail d'un build



The image shows the Jenkins web interface for a specific build. At the top, the Jenkins logo is on the left, and navigation links for 'Pipelines', 'Administration', and 'Déconnexion' are on the right. Below this, the project name 'plbsi-angular' is displayed with a star and gear icon. Further down, the build number 'plbsi-angular #59' is shown with a green checkmark. To the right of the build number are tabs for 'Pipeline', 'Modifications', 'Tests', 'Artefacts', and a set of icons for refreshing, settings, and closing. Below the tabs, the build details are listed: 'Branche: master' with a clock icon and '2m 43s', and 'Commit: 5c3e3ac' with a clock icon and '2 months ago'. The text 'Modifications par pro.rsauvage' is also visible.



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



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



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

- *Declarative Directive Generator* n'est valable que pour la syntaxe déclarative
- *Snippet Generator*, permettant de générer les steps, est valable pour les 2 syntaxes



Declarative Directive Generator

Sample Directive

agent: Agent

See [the online documentation](#) for more information on the agent directive.

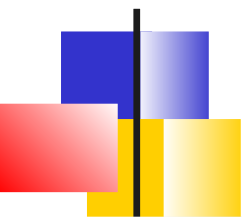
Agent

label: Run on an agent matching a label

Label

jdk8

Generate Declarative Directive



Snippet Generator

Sample Step

archiveArtifacts: Archive the artifacts

archiveArtifacts

Files to archive

Generate Pipeline Script



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 utilisables dans les pipelines

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*



Généralités

La syntaxe déclarative est recommandée car plus simple d'accès et plus lisible

Toutes les pipelines déclaratives sont dans un bloc ***pipeline***.

Elles contiennent toujours les **sections** suivantes :

- ***stages*** : Bloc contenant un ou plusieurs bloc *stage*
- ***stage*** : Bloc contenant un bloc *steps* et éventuellement un bloc *post*
- ***steps*** : Bloc contenant une succession de step unitaire

Des **directives** sont placées soit au niveau pipeline soit au niveau d'un stage. Par exemple : *agent*

Éventuellement, des sections ***post*** spécifient les actions de post build



Structure

```
pipeline {  
  // Directives s'appliquant à toute la pipeline  
  stages {  
    stage('Compile et tests') {  
      // Directives ne s'appliquant qu'au stage  
      steps {  
        // Steps unitaires  
        echo 'Unit test et packaging'  
  
      } post { // Les actions de post-build  
        // Les différentes issues du build (stable, unstable, success, ...)  
      }  
    }  
    stage('Une autre phase') {  
      ....  
    }  
  }  
}
```



Stages et steps

Les sections stages et steps ne font que délimiter un bloc

stages :

- pas de paramètres spécifiques.
- Englobe plusieurs blocs *stage*

stage :

- Un nom
- des directives appliquées au *stage*
- un bloc steps

steps :

- Pas de paramètre
- A l'intérieur de chaque *stage*
- Englobe plusieurs actions unitaires



Section *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 *stages* 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
- ***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
- ***docker, dockerfile*** : Si plugin docker présent
- ***kubernetes*** : Si plugin Kubernetes présent



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 ceux installés par l'administrateur Jenkins



Exemples

```
// Directive,  
// agent avec le label jdk8  
agent {  
    label 'jdk8'  
}
```

```
// Mise à disposition d'un outil sur l'agent  
  
agent any  
tools {  
    maven 'Maven 3.5'  
}
```



Directive *environment*

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}"  
  }  
}
```



Directive *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' }
    }
  }
}
```



Directives

input et *parameters*

La directive ***input*** permet de stopper l'exécution d'une pipeline et d'attendre une saisie 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 {  
  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."  
}
```



Directive *triggers*

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

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



Directive *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'  
  }  
}
```



Directive *parallel*

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
  stages {
    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 : Positionner le répertoire courant, supprimer un répertoire

fileExists, writeFile, readFile

mail, git, build, error

sleep, timeout, waitUntil, retry

withEnv, credentials, tools

cleanWs() : Nettoyer le workspace

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



Lint

La syntaxe autorisée dans Jenkinsfile dépend des plugins installés sur le serveur.

Pour valider la syntaxe d'un Jenkinsfile, Jenkins propose un Linter accessible via Jenkins CLI ou l'API Rest

Exemple :

```
# 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
```

Des extensions d'IDE proposent une validation du Jenkinsfile via l'API Rest de Jenkins



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



Pipelines

Approche et concepts

Syntaxe déclarative

Groovy et Syntaxe script

Librairies partagées



Introduction

Une pipeline scriptée utilise directement la syntaxe Groovy.

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

La syntaxe script n'est pas recommandée car moins lisible.

On l'utilise cependant assez souvent dans un bloc ***script*** à l'intérieur d'une pipeline déclarative

Une pipeline déclarative est en fait un script Groovy !



Exemple 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"
          }
        }
      }
    }
  }
}
```



Déclaration de variables

En Groovy, il n'est pas nécessaire de préciser le type d'une variable lors de sa déclaration.

Il suffit d'utiliser le mot-clé ***def***

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



Script Groovy

Les scripts Groovy contiennent des instructions qui ne sont pas encapsulées par une déclaration de classe.

Ils peuvent contenir des définitions de méthodes et des déclarations de variables

Un script est parsé intégralement avant son exécution

Le fichier Jenkinsfile est un script Groovy



Déclaration méthode et variables dans JenkinsFile

```
def integrationUrl
def dataCenters
pipeline {
    agent none

    .....
    stage {
        steps {
            script {
                checkSonarQualityGate()
            }
        }
    }
}

def checkSonarQualityGate(){
    // Get properties from report file to call SonarQube
    def sonarReportProps = readProperties file: 'target/sonar/report-task.txt'
    def sonarServerUrl = sonarReportProps['serverUrl']
    ....
}
```



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.'
```



Collections

Groovy facilite la manipulation des collections en ajoutant des opérateurs, des instanciations via des littéraux

L'accès aux éléments est cohérent quelque soit le type de la collection (*List*, *Map*, ...)

Il introduit également un nouveau type : *Range* avec la notation ..



Exemples Collection

```
// 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

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



Exécuteur

En mode script, la fonction *node()* permet de provisionner un agent

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



Examples

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



Checkout du dépôt

A la différence de la directive `agent`, *node* ne provoque pas le checkout du dépôt

// 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()
```



Plugin Utility Steps

Le plugin ***Pipeline Utility Steps*** ajoute plein d'étapes utilitaires comme :

- Trouver un fichier dans le workspace
- Créer ou vérifier un SHA-1
- Créer des tar gz, des zip
- Lire des fichiers CSV, properties, YAML, JSON
- ...



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 .. */  
    }  
  }  
}
```




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



Pipelines

Approche et concepts
Syntaxe déclarative
Groovy et Syntaxe script
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('myRepo')
```



Variable globale

Fichier ***vars/log.groovy***. Le nom du fichier doit être en *camelCase*.

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

Utilisation dans pipeline déclarative :

```
@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

?



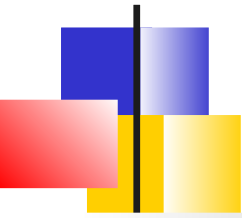
Exemples d'usage

-- 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.Utls.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"
    }
    // ...
}
```



Jenkins et les containers

Docker
Kubernetes



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



Docker pipeline plugin

Docker pipeline permet d'utiliser des containers pour exécuter les steps de la pipeline via la directive ***agent***

Il met également à disposition une variable ***docker*** permettant de construire et publier des images



Agent Docker

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

Il suffit de préciser l'attribut ***image*** à la directive *agent*



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 clean' }
    }
  }
}

// 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, *Jenkins* 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.

Le plugin *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	- 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'
    }
  }
}
```



Jenkins in Docker

Cloudbees fournit une image docker de Jenkins capable de lancer des agents docker : ***docker:dind***

```
docker network create jenkins
docker run --name jenkins-docker --rm --detach \
  --privileged --network jenkins --network-alias docker \
  --env DOCKER_TLS_CERTDIR=/certs \
  --volume jenkins-docker-certs:/certs/client \
  --volume jenkins-data:/var/jenkins_home \
  --publish 2376:2376 docker:dind --storage-driver
overlay2
```



Exemple customisation

```
FROM jenkins/jenkins:2.303.2-jdk11
USER root
RUN apt-get update && apt-get install -y apt-transport-https \
    ca-certificates curl gnupg2 \
    software-properties-common
RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN apt-key fingerprint 0EBFCD88
RUN add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) stable"
RUN apt-get update && apt-get install -y docker-ce-cli
USER jenkins
RUN jenkins-plugin-cli --plugins "blueocean:1.25.0 docker-workflow:1.26"
```



Jenkins et les container

Docker
Kubernetes



Kubernetes plugin

Le plugin ***Kubernetes*** permet de provisionner des agents sur un cluster Kubernetes

Le plugin crée un pod Kubernetes pour chaque agent démarré et l'arrête après chaque build.

Le pod se connecte automatiquement au master Jenkins



Configuration

*Manage Jenkins -> Manage Nodes and Clouds ->
Configure Clouds -> Add a new cloud -> Kubernetes*

Ensuite indiquer

- L'URL du cluster
- Les crédits d'accès
- L'URL de Jenkins



Pod Template

Les pod template permettent de définir les pods qui seront démarrés pour exécuter les builds :

- Ils sont définis par l'administrateur
- Ils peuvent être spécifiés par la directive ***inheritFrom***
- Ils peuvent être redéfinis dans le fichier Jenkinsfile

Le podTemplate doit pouvoir communiquer avec le nœud maître.

Lorsque le maître s'exécute dans le cluster Kubernetes, on peut utiliser ***jenkins/inbound-agent***



Exemple

Utilisation podTemplate par défaut :

```
agent {  
  kubernetes {  
    inheritFrom 'default'  
  }  
}
```



Exemple

Utilisation podTemplate avec 2 containers :

- Jenkins-agent
- jdk17-agent

```
agent {  
    kubernetes {  
        inheritFrom 'jdk17-agent'  
    }  
}  
steps {  
    container(name: 'openjdk-17') {  
        sh 'javac -version'  
    }  
}
```



Exemple

Redéfinition du podTemplate à utiliser

```
pipeline {  
  agent {  
    kubernetes {  
      yamlFile 'KubernetesPod.yaml'  
    }  
  }  
  Stages {  
    container(name: 'openjdk-17') {  
      sh 'javac -version'  
    }  
    ...  
  }  
}
```



Exemple yaml

```
apiVersion: v1
  kind: Pod
  metadata:
    labels:
      some-label: some-label-value
  spec:
    containers:
      - name: openjdk-17
        image: openjdk : 17-alpine
        command:
          - cat
        tty: true
      - name: busybox
        image: busybox
        command:
          - cat
        tty: true
```



Plugins CI/CD

Tests et analyse statique
Infrastructure as Code



Plugin junit

Le plugin *junit* permet de publier les résultats des tests.

- S'applique à tous les outils fournissant des résultats au format XML de junit
- Fournit une visualisation graphique des résultats de test, des historiques ainsi qu'une interface utilisateur Web pour afficher les rapports de test, suivre les échecs, etc.



Paramètres de *JUnit*

Test report XMLs: Plusieurs chemins vers les fichiers XML de résultat séparés par des espaces. Syntaxe Ant

Retain long standard output/error: Si coché, cela permet de conserver les sorties standard et d'erreur complètes des tests.

Health report amplification factor: Comment les résultats des tests affectent le statut du build.

Le facteur par défaut est 1,0. Un facteur de 0,0 désactivera la contribution des résultats de test à l'établissement du score de santé et, à titre d'exemple, un facteur de 0,5 signifie que 10 % des tests échoués obtiendront 95 % de santé.

Allow empty results: Si cochée, des résultats vides ne font pas échouer le build.



Paramètres de *JUnit* (2)

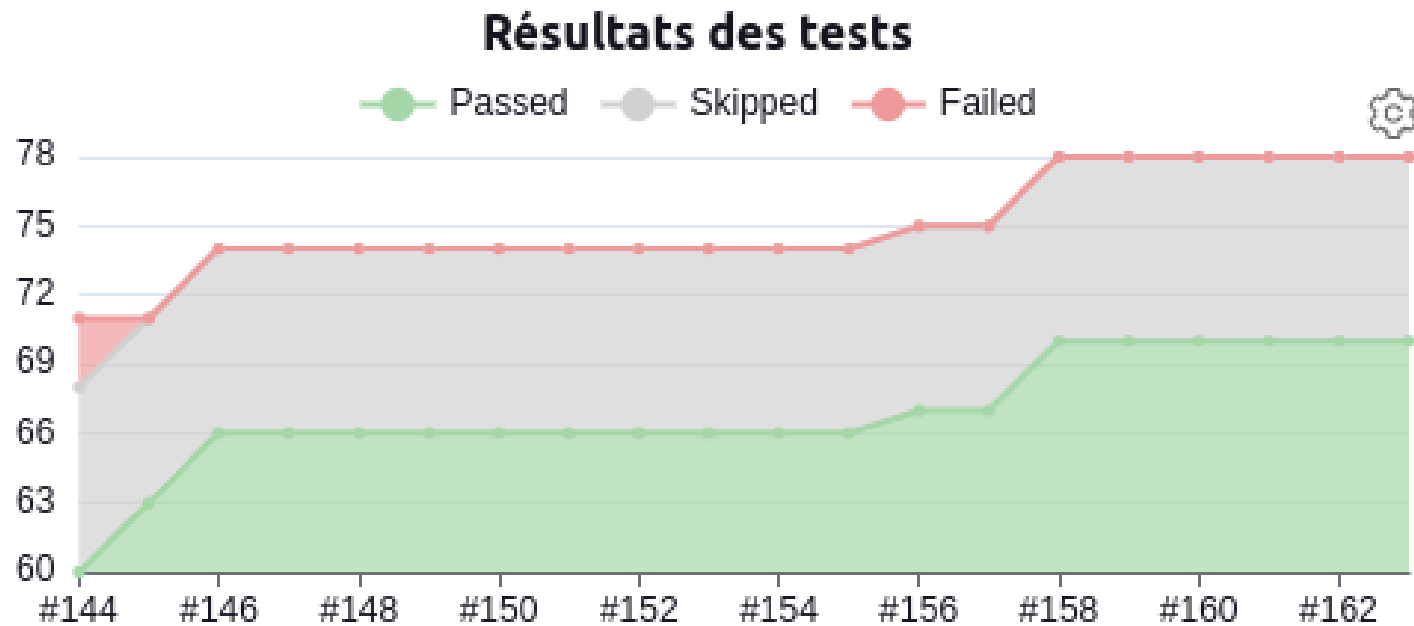
Skip publishing checks for failed builds: Si cochée, les rapports ne sont pas publiés si le build échoue. .

Check name: Le nom utilisé lors de la publication des résultats . Le nom peut également être surchargé avec `withChecks`

```
stage('Ignored') {  
    withChecks('Integration Tests') {  
        junit 'yet-more-test-results.xml'  
    }  
}
```

Skip marking build unstable: Si pas coché le build devient instable si au moins un échec. Si coché, le build est en succès même si il y a des erreurs

Graphe de tendance



Résultats des tests

1 échecs (±0) , 8 non passés (±0)



92 tests (±0)

A pris 5 mn 58 s.

Ajouter une description

Tous les tests qui ont échoué

Nom du test	Durée	Age
<div><div>– com.plb.plbsiapi.offre.gescof.GescofSynchroServiceTest.testCheckFamilleProduit</div><div>– Error Details</div><div>403 Forbidden: [{"code":403,"message":"Vous n'avez pas les droits suffisants pour effectuer cette action."}]</div><div>+ Stack Trace</div><div>+ Standard Output</div></div>	0.66 s	8

Tous les tests

Package	Durée	Échec	(diff)	Sauté	(diff)	Pass	(diff)	Total	(diff)
com.plb.plbsiapi	8 ms	0		0		1		1	
com.plb.plbsiapi.cms	0 ms	0		0		1		1	
com.plb.plbsiapi.core.service	1.3 s	0		0		2		2	
com.plb.plbsiapi.elk.offre.service	1.8 s	0		0		2		2	



Plugin Performance

Performance Plugin permet :

- d'exécuter des tests de performances lors d'une étape de construction avec l'outil Taurus
- de créer des rapports à partir de fichiers de résultats de tests préexistants.
Supporte les formats de Taurus, JMeter, HPE LoadRunner, wrk

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

[Performance Trend](#)

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

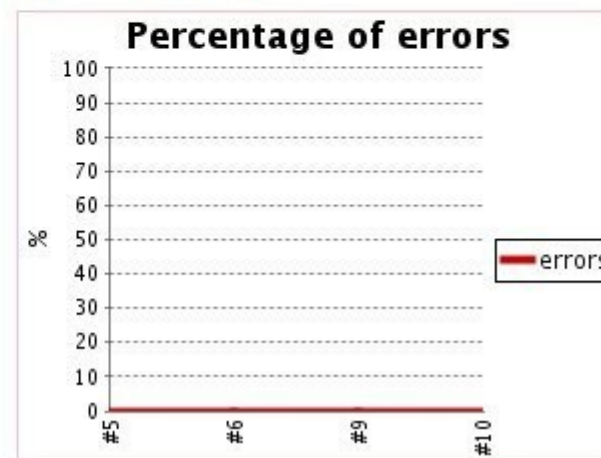
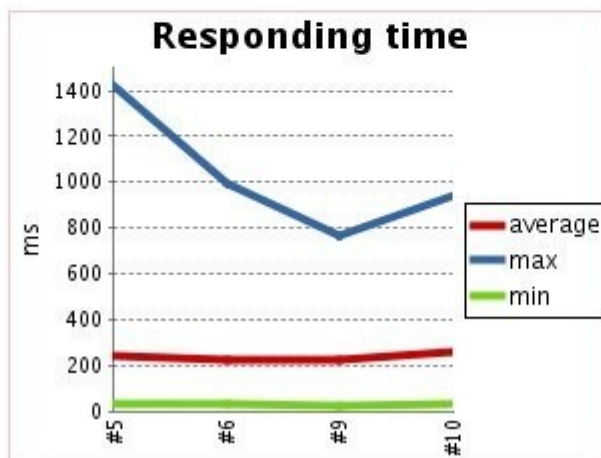
- [#10 Mar 22, 2010 10:36:48 AM](#)
- [#9 Mar 22, 2010 9:59:28 AM](#)
- [#8 Mar 22, 2010 9:46:45 AM](#)
- [#7 Mar 22, 2010 9:38:15 AM](#)
- [#6 Mar 9, 2010 1:22:57 PM](#)
- [#5 Mar 9, 2010 12:09:36 PM](#)
- [#3 Mar 9, 2010 11:06:32 AM](#)
- [#2 Mar 9, 2010 10:47:59 AM](#)
- [#1 Mar 9, 2010 10:38:19 AM](#)

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

Performance Trend

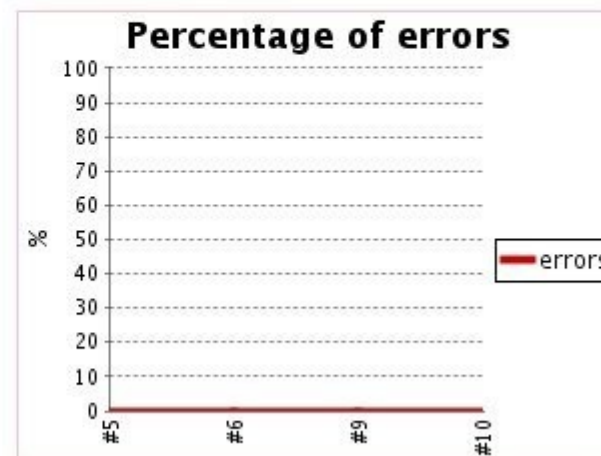
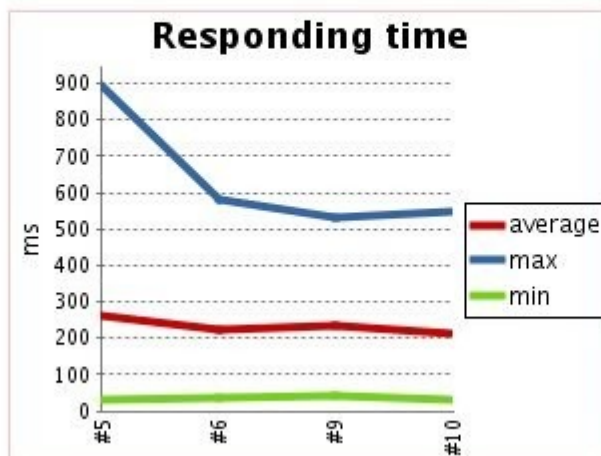
[Last Report](#)
[Filter trend data](#)

Test file: myTests1.jtl



[Trend report](#)

Test file: myTests2.jtl





Autres plugins

Cucumber Report : Publication des rapports de tests Cucumber

Robot Framework Plugin : Publie les résultats de Robot

Selenium Plugin : Publie les résultats Selenium



Plugin SonarQube Scanner

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

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

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



Configuration

La mise en place consiste à :

- Définir l' ou les instances de SonarQube

Manage Jenkins → Configure System → SonarQube configuration → Add SonarQube

- Définir les scanners à utiliser (Exemple Maven)

Manage Jenkins → Configure Tools → SonarQube scanner

- Configurer un projet pour utiliser le scanner



Steps du plugin Sonar

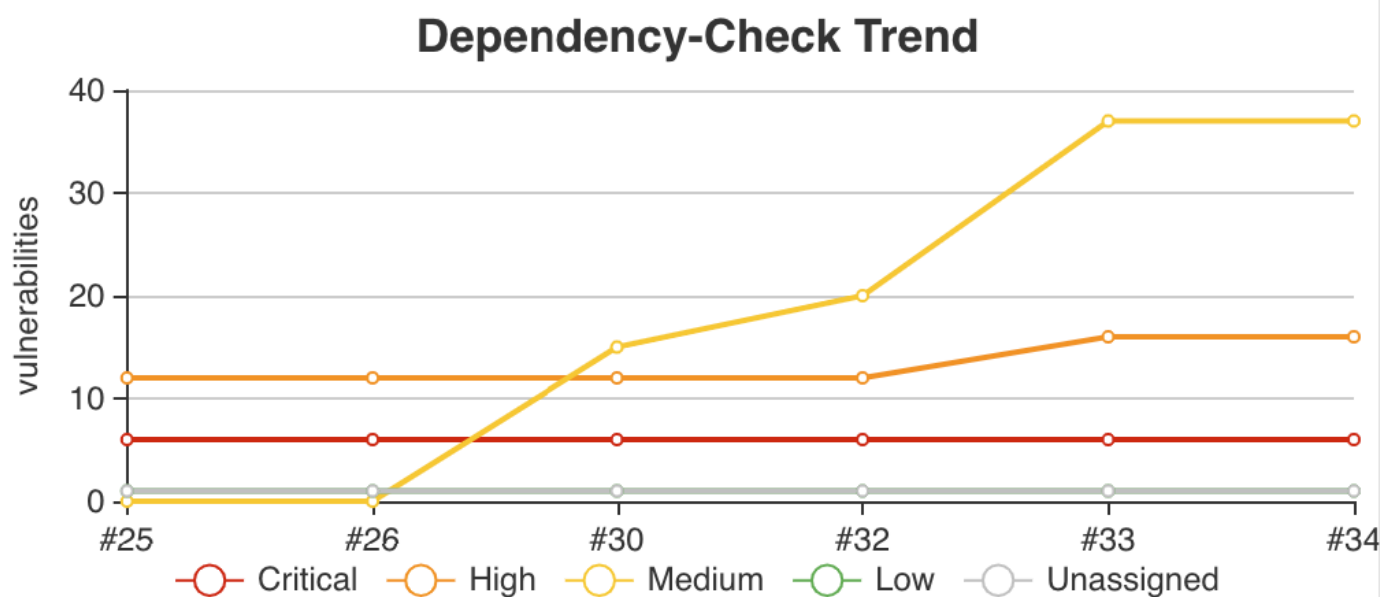
```
// Positionne les variables d'environnement
// SONAR_CONFIG_NAME, SONAR_HOST_URL, SONAR_AUTH_TOKEN .
withSonarQubeEnv('My SonarQube Server', envOnly: true) {
    println ${env.SONAR_HOST_URL}
}

// Pause pour récupérer le statut de la porte qualité
stage("Quality Gate") {
    steps {
        timeout(time: 1, unit: 'HOURS') {
            // true = pipeline est UNSTABLE si la porte qualité échoue
            waitForQualityGate abortPipeline: true
        }
    }
}
```

Vulnérabilités

Le plugin OWASP Dependency-Check permet :

- D'exécuter des tests de vulnérabilités
- De publier des tests déjà présents





Plugins CI/CD

Tests et analyse statique
Infrastructure as Code



Ansible

Le plugin Ansible permet d'exécuter des tâches Ansible en tant qu'étape de création de tâche.

Ansible doit être dans le PATH (Jenkins Global Tool Configuration)



Commandes Ad Hoc

Les commandes ad hoc permettent d'effectuer des opérations simples sans écrire un playbook complet.

```
ansibleAdhoc(credentialsId: 'private_key',  
inventory: 'inventories/a/hosts', hosts:  
'hosts_pattern', moduleArguments:  
'module_arguments')
```



Playbook

Les playbook Ansible peuvent s'exécuter avec le plugin. Cela apporte :

- Utilisation de crédentiel Jenkins
- une sortie couleur sans tampon dans le journal
- Utilisation des variables d'environnement Jenkins dans les playbook

```
ansiblePlaybook(credentialsId: 'private_key',  
inventory: 'inventories/a/hosts', playbook:  
'my_playbook.yml')
```



Vault

La plupart des opérations Ansible Vault peuvent être effectuées avec le plugin.

- Les opérations interactives telles que la création, la modification et l'affichage ne sont pas prises en charge via le plugin.

Permet aux développeurs de crypter des valeurs secrètes tout en gardant secret le mot de passe du vault.

```
ansibleVault(action: 'encrypt', input:  
  'vars/secrets.yml', vaultCredentialsId:  
  'ansible_vault_credentials')
```



Autres plugins

VMware vSphere Plugin: Création et gestion de machine virtuelle

Vagrant Plugin : Idem

Plugins fournisseur de cloud :

- ***AWS Step Plugin***
- ***Google Cloud Build Plugin***
- ***DigitalOcean Plugin***

Kubernetes CLI plugin permet de configurer *kubect*



Références

Jenkins : The Definitive Guide
Jenkins Wiki
Cloudbees



Annexes

Le plugin JTE
Syntaxe script
Pipeline legacy



Principes

Au lieu que chaque projet d'une entreprise passe du temps à développer sa propre pipeline.

Une équipe transverse fournit :

- Un gabarit de pipeline définissant les phases nécessaires à la construction d'un projet
- Des bibliothèques qui implémentent ces phases pour les différents outils de l'entreprise.

Les équipes projets n'ont plus qu'à fournir un fichier de configuration qui spécifie les bibliothèques à utiliser pour son projet.



Exemple Gabarit

```
on_pull_request to: develop, {  
  continuous_integration()  
}
```

```
on_merge to: develop, {  
  continuous_integration()  
  deploy_to dev  
  penetration_test()  
  integration_test()  
  performance_test()  
}
```

```
on_merge to: main, {  
  deploy_to prod  
}
```



Librairies

Les librairies sont poussées dans un dépôt séparé respectant une structure de fichiers Groovy

```
exampleLibraryName #
├── steps #
│   ├── step1.groovy # Implémentation de la step1
│   └── step2.groovy
├── resources # Fichiers ressources accessibles par les steps
│   ├── someResource.txt #
│   └── nested
│       └── anotherResource.json #
├── src #
│   └── example
│       └── Utility.groovy # Classe utilitaire utilisable par les steps
└── library_config.groovy # Permet de valider l'utilisation de la lib
```



Step.groovy

Par défaut, le nom de l'étape est basé sur le nom du fichier sans l'extension .groovy

Elle doit implémenter la méthode
`void call(){}`

qui peut prendre des paramètres¹

Des variables sont automatiquement mis à disposition :

- **`config`** : Map de propriétés de configuration
- **`stepContext, stageContext, hookContext`** : Contexte d'exécution de la step

1. Attention, toutes les implémentations doivent avoir la même signature de méthode



Configuration de pipeline

La configuration de pipeline d'un projet consiste principalement à indiquer les librairies utilisées par le projet.

Cela peut se faire dans l'interface Jenkins ou via un fichier `pipeline_config.groovy`



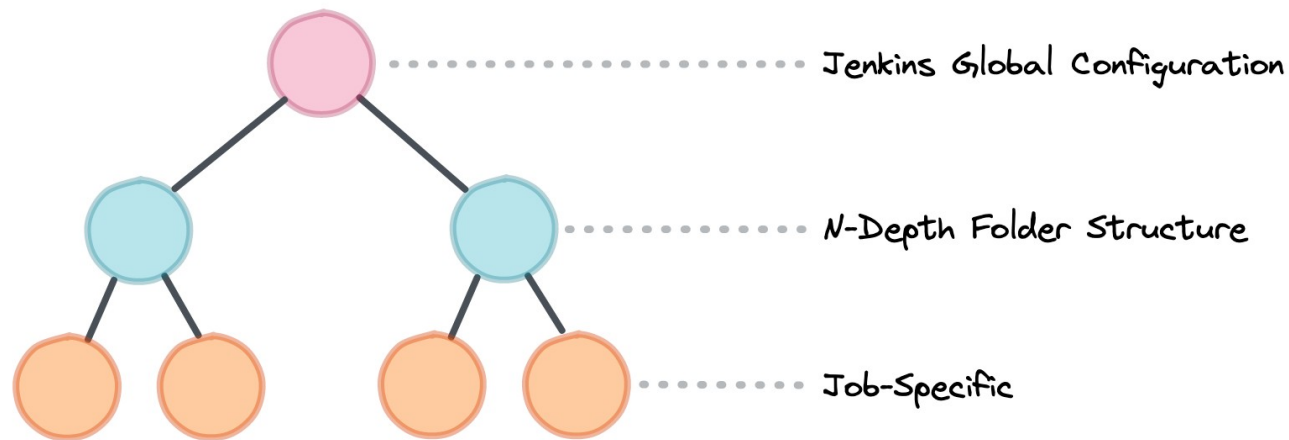
Catalogue de pipelines

Il est peu probable que toutes les pipelines d'une organisation correspondent à un seul modèle de pipeline.

JTE prend en charge plusieurs modèles de pipeline parmi lesquels les équipes peuvent choisir.

Ces modèles sont organisés en catalogues de pipelines ayant des relations hiérarchiques.

Governance tier



Les nœuds de la hiérarchie de configuration nommé Governance tiers stockent :

- Un ensemble de gabarits de pipeline
- Une configuration de pipeline héritée
- Les sources des librairies



Annexes

Le plugin JTE
Syntaxe script
Pipeline legacy



Interpolation de String

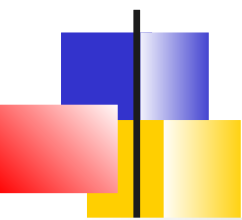
Jenkins a accès au mécanisme d'interpolation de chaîne de Groovy

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
  }
  ...
}
```



Annexes

Le plugin JTE
Syntaxe script
Pipeline legacy



Buils paramétrés

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

Ils sont renseignés :

- soit par l'utilisateur qui démarre le job manuellement.
Jenkins génère alors l'interface de saisie
- Soit par l'API :
`<job_url>/buildWithParameter?param1=value`
- Soit déclenchés par 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}`



Types de paramètres

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

Les types possibles sont alors :

- **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)

#	Time	Size
#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



Job Multi-configuration

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

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

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 installés pour le projet
- **Axe personnalisé** : Un paramètre du build auquel on a fourni 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

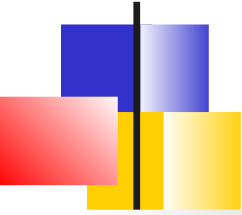
		OSX	linux	windows
mysql	Java 1.6			
	Java 1.5			
oracle	Java 1.6			
	Java 1.5			
postgres	Java 1.6			
	Java 1.5			
derby	Java 1.6			
	Java 1.5			



Chaînage de jobs

Le chaînage de jobs avec ce Jenkins peut se faire de 2 façons :

- Définir des **relations amont/aval** et utiliser les variables d'environnement, les plugin *Parameterized Trigger* ou *Copy Artefact* pour s'échanger des données
- Utiliser le plugin **Pipeline** et les plugins liés permettant de définir des pipelines complexes en Groovy



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.

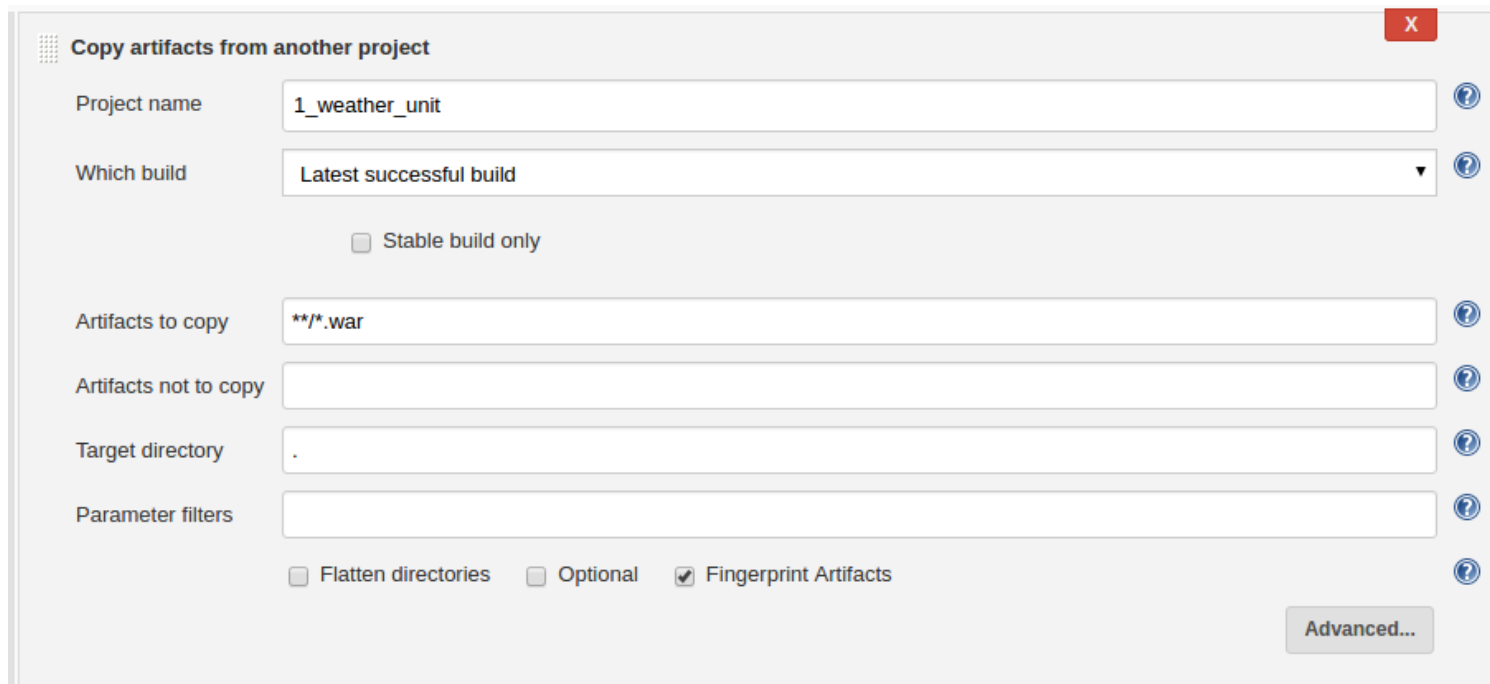
Les paramètres peuvent ainsi être renseignés avec :

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



Plugin Copy Artifacts

Le plugin « ***Copy Artifacts*** » permet de copier des artefacts construits par un build précédent dans le build courant



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).