



# Tutoriel – Utilisation de Jenkins

*Application principalement écrite en Java*

Le but de ce tutoriel est de vous montrer quelques grandes fonctionnalités disponibles avec Jenkins afin de mettre en place une intégration continue le projet Noumea. Il ne présente en aucun cas un guide détaillé de Jenkins.

## Contenu

Introduction .....	2
Accéder à Jenkins. ....	3
Création d'un Job .....	4
Planification des Build crons (Build, Run, Test) .....	7
Le Trigger Git pour déclencher un build & Push Tag Automatique .....	14
Pipeline.....	18
Analyse de la qualité du code.....	19
Checkstyle : .....	19
Conclusion .....	25



## Introduction

**Jenkins** est un outil open source d'intégration continue, fork de l'outil **Hudson** après les différends entre son auteur, Kohsuke Kawaguchi, et Oracle. Écrit en Java, Jenkins fonctionne dans un conteneur de servlets tel qu'Apache Tomcat, ou en mode autonome avec son propre serveur Web embarqué.

Source : Wikipédia.

Ce tutoriel a pour objectif de montrer les grandes lignes de Jenkins afin de mettre en place une intégration continue du projet Noumea.

Jenkins intègre énormément de modules utilisables selon vos projets (exemple : modules pour Java, Android, iOS, .NET...).

Pour ce tutoriel nous avons décidé d'utiliser, comme langage principale, le java & le JavaScript mais Jenkins peut être utilisé pour bien d'autre langage grâce à des plugins.

Nous allons commencer par vous présenter comment créer un job, configuration General, Planification des Build par des crons (Build, Run et Test) ou à partir d'un dépôt Git, l'utilisation des pipeline et JenkinsFile, puis comment faire exécuter une série de tests unitaires sur du code java ou JavaScript par Jenkins.

Enfin nous mettrons en place un outil d'analyse du code JavaScript.



## Accéder à Jenkins.

Une version de Jenkins est déjà installée dans Docker sur la machine Ubuntu (voir Document Guide installation Jenkins avec Docker).

On démarre Jenkins avec la commande : `docker start jenkins`

Après suffit simplement de se rendre avec Firefox à l'adresse suivante : <http://localhost:8080/>

Vous arriverez alors sur l'interface d'administration.

Jenkins a déjà été configuré. Nous allons donc voir comment intégrer le projet Calculatrice à Jenkins.



## Création d'un Job

Un job est l'équivalent d'un projet. Il va donc contenir un workspace (le répertoire où seront stockés les fichiers sources), des tâches et des builds, que nous paramétrons plus tard.

Sur l'interface d'accueil, on peut ajouter un job en passant par le lien « Nouveau Item » disponible dans le menu de gauche.

Vous arrivez donc sur le formulaire de création d'un job.

Tout d'abord, donnez un nom à votre job.

Différentes options se présentent pour le type de projet. A ses débuts, Jenkins était prévu pour des projets Maven, mais suite à la forte demande des utilisateurs, d'autres projets ont été rendus possibles d'intégrer.

Le premier choix : « **Construire un projet free-style** » permet d'intégrer tout type de projet. C'est ce type de projet que nous allons utiliser.

Une fois le formulaire rempli, vous pouvez cliquer sur « OK ». Vous passez alors à une deuxième configuration.



Pour le moment, nous laissons toutes les valeurs par défaut, à l'exception de « Description » où vous pouvez écrire à quoi correspond votre job. Vous pouvez ensuite cliquer sur le bouton sauver.

Votre premier job est alors créé et vous vous retrouvez sur l'interface de gestion de votre job.

Pour le moment, aucun fichier n'a été relié à votre projet. Si vous vous rendez dans votre workspace par le lien disponible dans le menu de gauche (« Répertoire de travail »), vous serez confronté à cette erreur :



Comme l'erreur l'indique, il faut en premier lieu lancer un premier build qui va initialiser des métadonnées, (et s'il y'a un gestionnaire de versionning, il va alors récupérer pour la première fois les fichiers sur le dépôt configuré)

Pour lancer un build, rien de plus simple, il suffit de cliquer sur « Lancer un build », disponible dans le menu de gauche.

Une fois le build lancé, vous pouvez voir l'historique des builds dans un encadré en dessous du menu de gauche :



 Jenkins

Jenkins > MonPremierJob >

[Retour au tableau de bord](#)

[État](#)

[Modifications](#)

[Répertoire de travail](#)

[Effacer l'espace de travail](#)

[Lancer un build](#)

[Supprimer Projet](#)

[Configurer](#)

[Rename](#)

## Workspace of MonPremierJob on maître



Aucun fichier dans ce répertoire

[Historique des builds](#) [tendance](#)

find

#1 26 déc. 2019 14:06

[Atom feed des builds](#) [Atom feed des échecs](#)

La couleur bleue à gauche de la date du build indique que tout s'est bien passé. La couleur rouge annonce elle que le build a échoué.

Au milieu, vous voyez maintenant que le workspace existe. Il est cependant vide.

Nous allons donc mettre les fichiers dans le workspace.

Naviguez ou saisissez dans l'URL l'adresse suivante : /var/lib/jenkins/jobs/

Vous pourrez y voir le dossier de votre job. Entrez dedans puis dans workspace.

C'est dans ce dernier dossier qu'il va falloir mettre les sources du projet.

Allez copier-coller les sources du projet disponibles dans le répertoire suivant : /home/rdeva/source/

*Si vous rencontrez un problème de droit ne vous permettant pas de copier les fichiers dans le répertoire workspace, ouvrez un terminal et rendez-vous dans le répertoire parent de workspace (/var/lib/jenkins/jobs/MonPremierJob dans notre cas), et saisissez cette commande :*

```
Sudo chmod 777 workspace/ -R
```

Le mot de passe est : rdeva4of6

Vous pourrez alors copier les fichiers

Quand vous retournez dans votre Firefox, vous pouvez actualiser la page (un rafraîchissement a déjà pu se faire le temps de la copie de votre fichier). Vous verrez alors vos fichiers.

<a href="#">is</a>		
<a href="#">calculatrice.html</a>	2,13 KB	
<a href="#">index.html</a>	932 B	
<a href="#">run-jasmine-xml-reporter.js</a>	7,31 KB	
<a href="#">style.css</a>	166 B	
<a href="#">(Tous les fichiers dans un zip)</a>		



## Planification des Build crons (Build, Run, Test)

Nous avons déjà vu comment créer un job, dans cette partie on va découvrir la planification dans Jenkins au niveau du scheduler « c'est qui d'éclanche le build » comme les crons en unix

MINUTE      HOUR      DOM      MONTH      DOW

**MINUTE:** Minutes withing the hour (0-59)

**HOUR:** Hour of the day (0 - 23)

**DOM:** The day of the Month (1-31)

**MONTH:** The Month (1-12)

**DOW:** the day of the week (0 - 7) where 0 and 7 are Sunday

Sur l'interface d'accueil, on peut aller sur le premier job, puis dans la configuration

S	M	Nom du projet ↓	Dernier succès	Dernier échec	Dernière durée
●	●	Git	18 j - #2	s. o.	9.6 s
●	●	MonPremierJob	11 j - #1	s. o.	0.32 s
●	●		18 j - #1	s. o.	3 mn 17 s
●	●		18 j - #2	s. o.	12 s

Icone: S M L

Liens: Atom feed pour tout Atom feed de tous les échecs Atom feed juste pour les dernières compilations

Après se positionner sur **C'est qui d'éclanche le build**, puis cocher la case construire périodiquement

General    Gestion de code source    **Ce qui déclenche le build**    Environnements de Build    Build    Actions à la suite du build

**Ce qui déclenche le build**

Déclencher les builds à distance (Par exemple, à partir de scripts) ?

Construire après le build sur d'autres projets ?

Construire périodiquement ?

Planning    \* \* \* \* \*

**⚠ Voulez-vous vraiment dire "chaque minute" avec l'expression "\*\*\*\*\*"? Peut-être voulez-vous dire "H \*\*\*\*\*"?**

Aurait été lancé à Monday, January 6, 2020 2:25:00 PM UTC; prochaine exécution à Monday, January 6, 2020 2:25:00 PM UTC.

GitHub hook trigger for GITScm polling ?

Scrutation de l'outil de gestion de version ?



Dans notre exemple on va faire 5 étoiles (\* \* \* \* \*) c'est à dire il va d'éclanche le build toute les minutes, puis vous enregistre.

[Retour au tableau de bord](#)

[État](#)

[Modifications](#)

[Répertoire de travail](#)

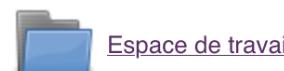
[Lancer un build](#)

[Supprimer Projet](#)

[Configurer](#)

[Rename](#)

## Projet MonPremierJob



[Espace de travail](#)



[Changements récents](#)

### Liens permanents

- [Dernier build \(#1\), il y a 11 j](#)
- [Dernier build stable \(#1\), il y a 11 j](#)
- [Dernier build avec succès \(#1\), il y a 11 j](#)
- [Last completed build \(#1\), il y a 11 j](#)

**Historique des builds** [tendance](#) —

find  x

	<a href="#">#3</a>	6 janv. 2020 14:34
	<a href="#">#2</a>	6 janv. 2020 14:33
	<a href="#">#1</a>	26 déc. 2019 14:06

[Atom feed des builds](#)  [Atom feed des échecs](#)

Comme on peut le constater sur l'historique des builds le premier build a été lancer le 26 dec 2019, mais après la planification du build pour la date du 6 janv. 20, a l'intervalle de chaque minute il se d'éclanche.

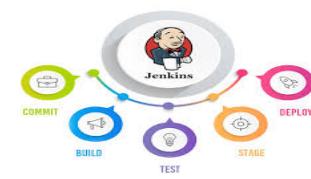
Maintenant on va créer notre premier Build -> Run ->Test.

Dans le build on va écrire un script si tout est OK il exécute le Run qui dépend du Build puis le Test si le Run est OK

Nous allons utiliser un exemple simple comme un **Hello World** avec trois (3) job :

- 1-Build
- 1-Run
- 1-Test

On va créer le job 1-Build qui sera de type free-style tout comme les deux autres jobs.



**Saisissez un nom**

1-Build

» Champ obligatoire

**Construire un projet free-style**  
Ceci est la fonction principale de Jenkins qui sert à builder (construire) votre projet. Vous pouvez intégrer tous les outils de gestion de version avec tous les systèmes de build. Il est même possible d'utiliser Jenkins pour tout autre chose qu'un build logiciel.

**Construire un projet maven**  
Construit un projet avec maven. Jenkins utilise directement vos fichiers POM et diminue radicalement l'effort de configuration. Cette fonctionnalité est encore en bêta mais elle est disponible afin d'obtenir vos retours.

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

**Construire un projet multi-configuration**  
Adapté aux projets qui nécessitent un grand nombre de configurations différentes, comme des environnements de test multiples, des binaires spécifiques à une plateforme, 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**

# Build

Ajouter une étape au build ▾

- Conditional step (single)
- Appeler Ant
- Conditional steps (multiple)
- Exécuter un script shell**
- Exécuter une ligne de commande batch Windows
- Invoke Gradle script
- Invoquer les cibles Maven de haut niveau
- Run with timeout
- Set build status to "pending" on GitHub commit
- Trigger/call builds on other projects



**Build**

Exécuter un script shell

Commande

```
mkdir -p /tmp/Boubacar
rm -rf /tmp/boubacar
echo '
public class Main{
    public static void main(String[] args){
        System.out.println("hello world");
    }
}
' >/tmp/Boubacar/Main.java
javac /tmp/Boubacar/Main.java
```

Voir [la liste des variables d'environnement disponibles](#)

[Avancé...](#)

Ajouter une étape au build ▾

**Actions à la suite du build**

Ajouter une action après le build ▾

[Sauver](#) [Apply](#)

Dans la partie commande shell on va exécuter ce script pour affiche un hello world, puis on enregistre.

- [Retour au tableau de bord](#)
  - [État](#)
  - [Modifications](#)
  - [Répertoire de travail](#)
  - [Lancer un build](#)
  - [Supprimer Projet](#)
  - [Configurer](#)
  - [Rename](#)
- On lance le Build

## Projet 1-Build

- [Espace de travail](#)
- [Changements récents](#)

## Liens permanents



## Historique des builds

tendance ▾

find X

#2 6 janv. 2020 15:11

**Modifications**

**Console Output**

**échecs**

Informations de la construction

Delete build '#2'

Maintenant on va vérifier dans la console si le script c'est bien exécuter ..

[Retour au projet](#)

[État](#)

[Modifications](#)

**Console Output**

[View as plain text](#)

[Informations de la construction](#)

[Delete build '#2'](#)

[Build précédent](#)

## Sortie de la console

```

Started by user Boubacar
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/1-Build
[1-Build] $ /bin/sh -xe /tmp/jenkins3001669685671317740.sh
+ mkdir -p /tmp/Boubacar
+ rm -rf /tmp/boubacar
+ echo
public class Main{
    public static void main(String[] args){
        System.out.println("hello world");
    }
}

+ javac /tmp/Boubacar/Main.java
Finished: SUCCESS

```

Nous allons créer les deux autres jobs avec la même procédure :

**1-Run : dans la partie script shell on ajoute :** on construit le Run à partir du build

### Ce qui déclenche le build

Déclencher les builds à distance (Par exemple, à partir de scripts)

Construire après le build sur d'autres projets

Projet à surveiller 1-Build,

- Déclencher que si la construction est stable
- Déclencher même si la construction est instable
- Déclencher même si la construction échoue



Après avoir enregistré, il lancer le **1-Buil**, qui va automatiquement déclencher le 1-Run.

Maintenant nous allons vérifier s'il a bien d'éclanche le 1-Run

- [Retour au tableau de bord](#)
- [État](#)
- [Modifications](#)
- [Répertoire de travail](#)
- [Lancer un build](#)
- [Supprimer Projet](#)
- [Configurer](#)
- [Rename](#)

## Projet 1-Run



[Espace de travail](#)



[Changements récents](#)

### Projets en amont



[1-Build](#)

### Liens permanents

- [Dernier build \(#1\), il y a 1 mn 32 s](#)
- [Dernier build stable \(#1\), il y a 1 mn 32 s](#)
- [Dernier build avec succès \(#1\), il y a 1 mn 32 s](#)
- [Last completed build \(#1\), il y a 1 mn 32 s](#)

Historique des builds [tendance](#)

find

#1	6 janv. 2020 15:37
Modifications	<a href="#">échecs</a>
Console Output	
Informations de la construction	
Delete build '#1'	

- [Retour au projet](#)
- [État](#)
- [Modifications](#)
- [Console Output](#)
- [View as plain text](#)
- [Informations de la construction](#)
- [Delete build '#1'](#)



## Sortie de la console

```
Started by upstream project "1-Build" build number 3
originally caused by:
Started by user Boubacar
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/1-Run
[1-Run] $ /bin/sh -xe /tmp/jenkins8273460414156314534.sh
+ cd /tmp/Boubacar/
+ java Main
Finished: SUCCESS
```

Il a bien exécuté le 1-Run et il se rendu dans le dossier /tmp/Boubacar et il a exécuter le java Main.



1-Test : dans la partie script shell on ajoute :

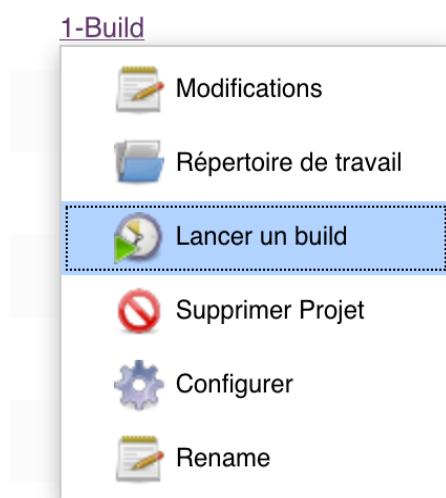
### Ce qui déclenche le build

- Déclencher les builds à distance (Par exemple, à partir de scripts)
  - Construire après le build sur d'autres projets
- Projet à surveiller  ?
- Déclencher que si la construction est stable  
 Déclencher même si la construction est instable  
 Déclencher même si la construction échoue

Ici le 1-Test dépend de 1-Run

```
cd /tmp/Boubacar
echo ##### le contenu du fichier file est #####
cat test.file | grep -ri hello | wc -l
echo "OK"
```

Enregistre le job puis lancer à nouveau le 1-Build à son tour il va exécuter le 1-Run puis le 1-Test



### Sortie de la console

```
Started by upstream project "1-Run" build number 5
originally caused by:
Started by upstream project "1-Run" build number 7
originally caused by:
Started by user Boubacar
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/1-Test
[1-Test] $ /bin/sh -xe /tmp/jenkins4453527344729228415.sh
+ cd /tmp/Boubacar
+ echo #### le contenu du fichier file est #####
##### le contenu du fichier file est #####
+ cat test.file
+ grep -ri hello
+ wc -l
3
+ echo OK
OK
Finished: SUCCESS
```



## Le Trigger Git pour déclencher un build & Push Tag Automatique

Précédemment nous avons puis planifié des builds crons (buil -> Run -> Test), dans cette partie on va utiliser un trigger qui vas scruter en permanence l'état d'un dépôt git, dès lors qu'on va avoir des nouveaux commit, un job qui va se lancer automatiquement à un intervalle de temps réguliers.

- Nous allons utiliser un dépôt existant sur github qui contient juste un code java helloworld : <https://github.com/buba4r>HelloWorld>
- Build sans utilisation du plugin Git.

On vas créer un job dans lequel on vas faire :

- Clone :
  - Git clone https://github.com/buba4r>HelloWorld
  - cd jenkins-helloworld
- Compilation :
  - Javac Main.java
- Lancement un run :
  - Java Main

Nous allons créer un job avec le **Git**

Dans la section **générale** de la page de configuration du job vous sélectionner github projet puis dans url vous mettez le lien du dépôt git.

Puis on ajoute le code ci-dessous dans <la partie **build Executer un script shell**> puis enregistre le job:

```
git clone https://github.com/buba4r>HelloWorld
cd jenkins-helloworld
Javac Main.java
Java Main
```



Et cela fonctionne bien mais c'est n'est pas le plus pratique, maintenant nous allons utiliser le plugin Git intégré dans Jenkins.

La seule différence ici dans la Gestion de code source il faut cocher Git au lieu Aucune et renseigner url du repositories , si le dépôt est privé ajouter un credentials sinon aucun puis spécifier la branche du dépôt exemple : \*/master comme indiqué sur la fig ci-dessous

#### Dans <la partie build : Executer un script shell>

Build reduit :

```
Javac Main.java
Java Main.
```

Enregistre le job et tester le build .

- LE DECLANCHEURS DE BUID - Trigger:

Un Trigger permet de checker à un intervalle de temps régulier sur un dépôt Git.

Une fois que vous avez configuré votre système de gestion de version, vous devez dire à Jenkins quand démarrer un build. Vous pouvez configurer cela dans la section Ce qui déclenche le build.

Dans un build free-style, il y a trois manières basiques de déclencher une tâche de build (voir fig)



Les tâches de build périodiques ne sont généralement pas la meilleure stratégie pour la plupart des tâches de build d'Intégration Continue. La valeur du retour d'information est proportionnelle à la vitesse à laquelle vous recevez ce retour et il n'y a pas d'exception en ce qui concerne l'Intégration Continue. C'est pour cette raison que scruter le SCM est généralement une bien meilleure option.

La scrutation implique l'interrogation à intervalles réguliers du serveur de contrôle de version pour savoir si des changements ont été ajoutés. Si des changements ont été faits dans le code source du projet, Jenkins lance alors un build. Scruter est habituellement une opération peu coûteuse, vous pouvez donc le faire fréquemment pour vous assurer qu'un build sera lancé rapidement après tout commit de code source. Plus vous scruterez fréquemment, plus votre tâche démarrera rapidement et plus précis sera le retour d'information lié aux changements effectués dans le cas où le build échoue.

Dans Jenkins, la scrutation du SCM est très facile à configurer et utilise la même syntaxe cron précédemment présentée.

Naturellement vous aurez l'envie de scruter le SCM le plus souvent possible (par exemple en utilisant “\* \* \* \* \*” pour chaque minute). Comme Jenkins n'utilise que des requêtes simples et ne lance de build que lorsque le code source a été modifié, cette approche est souvent raisonnable pour de petits projets. Cela montre cependant des limites quand il y a un grand nombre de tâches de build car cela pourrait saturer le serveur SCM et le réseau avec les requêtes dont beaucoup sont inutiles. Dans ce cas, une approche plus précise sera plus appropriée avec un déclenchement de la tâche de build directement par le SCM lorsqu'il reçoit un changement. Si des changements sont commis très fréquemment et dans un grand nombre de projets, cela peut causer la création d'une longue liste d'attente de tâches de build et ainsi retarder le retour d'information par la suite. Vous pouvez donc partiellement réduire la file d'attente de builds en scrutant moins régulièrement le SCM mais au prix d'un retour d'information moins précis.

Enregistre votre job puis modifier votre dépôt, après une minute un build va se déclencher.

- **PUSH TAG AUTOMATIQUE :**

Précédemment nous avons vu comment utiliser le plugin Jenkins pour tirer des modifications d'un dépôt git de GitHub ou gitLab vers notre outil de build, notre job fera en sorte dès qu'ils auront des nouveaux commit qu'on est un trigger pour scruter et lance le build. Là nous allons faire l'inverse c'est à dire dès lors qu'on a un build avec succès on va vouloir pousser sur notre dépôt quelque chose.

Pour faire nous allons commencer à créer un credential, si le build est ok avec succès nous allons pousser un nouveau tag sur la branche c'est à dire on va checker si une version et update on va la tagger mais on peut faire aussi avec les branches de git .



**Git**

**Repositories**

Repository URL: <https://github.com/buba4r/HelloWorld.git>

Credentials: buba4r\*\*\*\*\*

Name: HelloWorld

Refspec:

**Branches to build**

Branch Specifier (blank for 'any')

Pour faire cela il faut rajouter deux lignes dans notre build :

**Exécuter un script shell**

Commande

```
git config --global user.email "bubakar24@gmail.com"
git config --global user.name "buba4r"

javac Main.java
java Main
```

Après le build si succès nous allons tagger avec un nouveau message.

**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**

Tag to push: VERSION-\$BUILD\_ID

Tag message: Jenkins build

Create new tag

Update new tag

Target remote name: HelloWorld



Branch: master ▾    New pull request

Switch branches/tags

Find a tag

Branches    Tags

VERSION-10

VERSION-9

Lancer le build et sur git le numéro de version tagger avec le même numéro de build.

## Pipeline

Voir documentation [ici](#) :

Dans cette partie nous allons réaliser notre premier pipeline Jenkins. Un pipeline est une chaîne d'action un ensemble de job que l'on peut décrire par du code donc en occurrence du groovy c'est ce qui est utilisé au niveau Jenkins.



## Analyse de la qualité du code

Maintenant, intéressons-nous à l'analyse de qualité du code source. Il y a beaucoup d'outils open source qui peuvent aider à identifier les mauvaises pratiques de codage.

Dans le monde Java, trois outils d'analyses statiques ont résisté à l'épreuve du temps, et sont largement utilisés de manière très complémentaire. Checkstyle excelle dans la vérification des conventions et normes de codage, les pratiques de codage, ainsi que d'autres mesures telles que la complexité du code. PMD est un outil d'analyse statique similaire à Checkstyle, plus focalisé sur les pratiques de codage et de conception. Et FindBugs est un outil innovant, issu des travaux de recherche de Bill Pugh et de son équipe de l'université du Maryland, qui se focalise sur l'identification du code dangereux et bogue. Et si vous êtes en train de travailler avec Groovy ou Grails, vous pouvez utiliser CodeNarc, qui vérifie la norme et les pratiques de codage de Groovy.

Tous ces outils peuvent être facilement intégrés dans votre processus de build. Dans les sections suivantes, nous verrons comment configurer ces outils pour générer des rapports XML que Jenkins peut ensuite utiliser dans ses propres rapports.

### Checkstyle :

[Checkstyle](#) est un outil d'analyse statique pour Java. A l'origine conçu pour faire respecter un ensemble de normes de codage hautement configurable, Checkstyle permet aussi maintenant de vérifier les mauvaises pratiques de codage, ainsi que le code trop complexe ou dupliqué. Checkstyle est un outil polyvalent et flexible qui devrait avoir sa place dans n'importe quelle stratégie d'analyse de code basé sur Java.

Checkstyle supporte un très grand nombre de règles, incluant celles liées aux normes de nommage, annotations, commentaires javadoc, taille de méthode et de classe, mesures de complexité de code, mauvaises pratiques de codage, et beaucoup d'autres.

Le code dupliqué est un autre problème important de la qualité de code — le code dupliqué ou quasi-dupliqué est plus difficile à maintenir et à déboguer. Checkstyle fournit un certain soutien pour la détection de code dupliqué, mais des outils plus spécialisés comme CPD font un meilleur travail dans ce domaine.

Un des plugins de qualité de code les plus utiles est le [plugin Violations](#). Ce plugin n'analysera pas le code source de votre projet (vous devez configurer le build pour faire cela), mais il fait un excellent travail en élaborant des rapports sur les mesures de la qualité du code pour les builds individuels et les tendances au fil du temps. Le plugin s'adresse aux rapports sur les mesures de qualité de code venant d'une large gamme d'outils d'analyse statique, comprenant :

Pour Java :

[Checkstyle](#), [CPD](#), [PMD](#), [FindBugs](#), and [jcreport](#)



Installer ce plugin est d'une simplicité. Il suffit d'aller sur l'écran du Plugin Manager et de sélectionner le plugin Violations de Jenkins. Une fois que vous installé le plugin et redémarré Jenkins, vous serez capable de l'utiliser pour vos projets.

Le plugin Violations ne génère pas de mesures de qualité de code lui-même — vous devez configurer votre build pour faire cela.

NB : dans cette partie je vous invite à utiliser l'exemple du dépôt git suivante : [https://github.com/buba4r/app\\_Jenkins.git](https://github.com/buba4r/app_Jenkins.git) pour utilisation de maven.

### Dans l'action suite du build :

CHECKSTYLE. Il permet d'analyser tout type d'XML au bon format. On peut alors créer une tâche (action) après le build. Choisissez : « Publier les résultats de l'analyse Checkstyle » dans « Actions à la suite du build », toujours dans la configuration. Mettez le fichier XML dans le champ que vous avez paramétré au-dessus dans l'autre tâche.



**[Deprecated] Publish Checkstyle analysis results**

Checkstyle results `**/target/checkstyle-result.xml`

Fileset includes setting that specifies the generated raw CheckStyle XML report files, such as `**/checkstyle-result.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/checkstyle-result.xml` is used. Be sure not to include any non-report files into this pattern.

**[Deprecated] Publish FindBugs analysis results**

FindBugs results `**/findbugsXml.xml`

Fileset includes setting that specifies the generated raw FindBugs XML report files, such as `**/findbugs.xml` or `**/findbugsXml.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/findbugsXml.xml` or `**/findbugs.xml` are used for maven or ant builds, respectively. Be sure not to include any non-report files into this pattern.

Use rank as priority

Uses the bug rank when evaluating the priority of the warnings (otherwise the FindBugs priority is used).

**[Deprecated] Publish PMD analysis results**

PMD results `**/pmd.xml`

Fileset includes setting that specifies the generated raw PMD XML report files, such as `**/pmd.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/pmd.xml` is used. Be sure not to include any non-report files into this pattern.

**Report Violations**

Help for feature: Actions à la suite du build

	checkstyle	codenarc	cpd	cpplint	csslint	findbugs	fxcop	gendarme	jcreport	jslint	pep8	perlritic
checkstyle	10	100	999			**/target/checkstyle-result.xml						
codenarc	10	999	999									
cpd	10	999	999									
cpplint	10	999	999									
csslint	10	999	999									
findbugs	5	10	999	**/target/findbugs.xml								
fxcop	10	999	999									
gendarme	10	999	999									
jcreport	10	999	999									
jslint	10	999	999									
pep8	10	999	999									
perlritic	10	999	999									
	Sauver	Apply	200	999	**/target/pmd.xml							

En cliquant sur avancée, vous pouvez définir une valeur qui permet de dire à Jenkins d'échouer le build au-delà d'un certain nombre d'erreur. Allons le paramétrer à 20 :



## Detect modules



Determines if Ant or Maven modules should be detected for all files that contain warnings. Activating this option may increase your build time since the detector scans the whole workspace for 'build.xml' or 'pom.xml' files in order to assign the correct module names.

## Health thresholds



Configure the thresholds for the build health. If left empty then no health report is created. If the actual number of warnings is between the provided thresholds then the build health is interpolated.

## Health priorities



Determines which warning priorities should be considered when evaluating the build health.

## Status thresholds (Totals)

Toutes les priorités	Priority high	Priority normal	Priority low
----------------------	---------------	-----------------	--------------



If the number of total warnings is greater than one of these thresholds then a build is considered as unstable or failed, respectively. I.e., a value of 0 means that the build status is

Vous pouvez maintenant sauver la configuration et lancer un build.

## Regarder le résultat :

- [Retour au projet](#)
- [État](#)
- [Modifications](#)
- [Console Output](#)
- [Informations de la construction](#)
- [Delete build #34\\*](#)
- [Git Build Data](#)
- [No Tags](#)
- [Résultats Checkstyle](#)
- [Résultats de FindBugs](#)
- [Résultats PMD](#)
- [Résultats des tests](#)
- [Violations](#)
- [Build précédent](#)

**Construction #34 (14 mars 2020 04:46:16)**

Démarrée il y a 28 j.  
A duré 3 mn 6 s

[Ajouter une description](#)

---

No changes.

Lancé par l'utilisateur Boubacar

**git**  
Revision: 14a872b1b634272ce10059fc30f52353c080508a  
refs/remotes/origin/master

Checkstyle: 820 warnings from one analysis.

FindBugs: 6 warnings from one analysis.

PMD: 11 warnings from one analysis.

Résultats des tests (aucune erreur)

checkstyle 820 findbugs No reports pmd 11

## Cliquer sur les warnings

Vous obtenez des détails du : Checkstyle , findBugs , PMD et le Résultats des tests.



## Résultats de CheckStyle

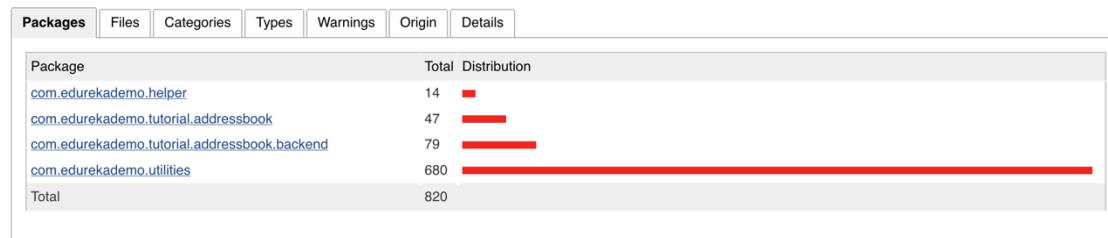
### Tendance des Alertes

Toutes les Alertes	Nouvelles Alertes	Alertes résolues
820	0	0

### Résumé

Total	Priorité haute	Priorité normale	Priorité basse
820	<a href="#">820</a>	0	0

### Détails



Vous pouvez maintenant cliquer sur les fichiers en bas, et vous obtenez un détail précis pour le fichier :

```

082         // Save DAO to backend with direct synchronous service API
083         getUI().service.save(contact);
084
085         String msg = String.format("Saved '%s %s'.",
086             contact.getFirstName(),
087             contact.getLastName());
088         Notification.show(msg, Type.TRAY_NOTIFICATION);
089         getUI().refreshContacts();
090     } catch (FieldGroup.CommitException e) {
091         // Validation exceptions could be shown here
092     }
093
094     public void cancel(Button.ClickEvent event) {
095         // Place to call business logic.
096         Notification.show("Cancelled", Type.TRAY_NOTIFICATION);
097         getUI().contactList.select(null);
098     }
099

```

Maintenant, pour avoir l'analyse précise, cliquer sur les liens en bas.

NOTE : l'infobulle se déclenche sur le survol de la ligne orange.

Maintenant, nous vous invitons à résoudre le problème pour que le build marche. La meilleure solution serait d'exclure deux fichiers car ils sont dépendants d'autres, ce qui rend l'analyse peu efficace.



## Résultats de FindBugs

### Tendance des warnings

Tous les warnings	New this build	Warnings résolus
6	0	0

### Résumé

Total	Priorité haute	Priorité normale	Priorité basse
6	0	6	0

### Details



## Résultat de PMD

### Tendance des Alertes

Toutes les Alertes	Nouvelles Alertes	Alertes résolues
11	0	0

### Résumé

Total	Priorité haute	Priorité normale	Priorité basse
11	0	11	0

### Details



## Résultats des tests

0 échecs ( $\pm 0$ )

23 tests ( $\pm 0$ )

A pris 0.22 s.

Ajouter une description

### Tous les tests

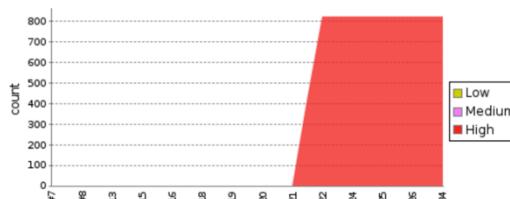
Package	Durée	Échec (diff)	Sauté (diff)	Pass (diff)	Total (diff)
com.edurekademo.utilities	0.51 s	0	0	23	23



## ⚠️ Violations Report /job/Review\_code/34 for build 34

Type	Violations	Files in violation
checkstyle	820	13
pmd	11	5

### checkstyle



filename	I	m	h	number ↑
src/main/java/com/edurekademo/utilities/StringUtilities.java	0	0	145	145
src/main/java/com/edurekademo/utilities/PropertyLoader.java	0	0	144	144
src/main/java/com/edurekademo/utilities/GenericComparator.java	0	0	132	132
src/main/java/com/edurekademo/utilities/CASEInsensitiveComparator.java	0	0	86	86
src/main/java/com/edurekademo/utilities/PropertyHelper.java	0	0	72	72
src/main/java/com/edurekademo/utilities/LoggerStackTraceUtil.java	0	0	62	62
src/main/java/com/edurekademo/tutorial/addressbook/backend/Contact.java	0	0	45	45
src/main/java/com/edurekademo/tutorial/addressbook/backend/ContactService.java	0	0	34	34
src/main/java/com/edurekademo/utilities/HexAsciiConvertor.java	0	0	34	34

### Build Pipeline

Run History Configure Add Step Delete Manage

Pipeline #4

- #4 compile Mar 13, 2020 11:35:09 PM 22 s admin
- #6 Review\_code Mar 13, 2020 11:35:40 PM 1 mn 43 s
- #2 Test Mar 13, 2020 11:37:30 PM 18 s

## Conclusion

On a pu donc voir comment on peut gérer les régressions de codes avec des plugins comme Checkstyle pour la qualité du code Java.

Des plugins d'analyse de code sont aussi présents dans d'autres langages pour détecter du code dupliqué, ...

Vous pourrez trouver une liste exhaustive de plugins à installer sous Jenkins [ici](#).

**Pour résumer :** pour mettre un projet sous Jenkins on **crée un job** et on **effectue un premier build** pour créer le workspace (répertoire des fichiers sources).

Ensuite on **configure le build** ou **on crée des tâches** (des actions) à effectuer lors du build. Par exemple, dans notre projet nous avons effectué une tâche pour générer un rapport des tests puis une autre tâche pour générer un rapport sur l'analyse du code Java.

Après cette modification, à chaque build ces tâches seront effectuées.

Si le **build réussit**, toutes les tâches ont été acceptées, **sinon** cela signifie qu'une tâche a échoué et on peut voir d'où vient cette erreur facilement.