

# Jenkins utilisation

# Introduction

---

1. Introduction
2. Jobs
3. Pipeline
4. Scripted Pipeline
5. Fonctionnalités avancées
6. Liens utiles

# Objectifs

---

- Présentation de Jenkins
- Installation de Jenkins
- Présentation de l'interface
- Installation des plugins
- Gestion des jobs
- Jenkins et les APIs

# Présentation

---

Jenkins est un des outils de référence pour le traitement de **CI/CD** (Continuous integration / Continuous delivery).

Jenkins permet d'automatiser et de programmer le lancement de traitement via des jobs. Il propose plusieurs types de jobs pour faciliter leur écriture.

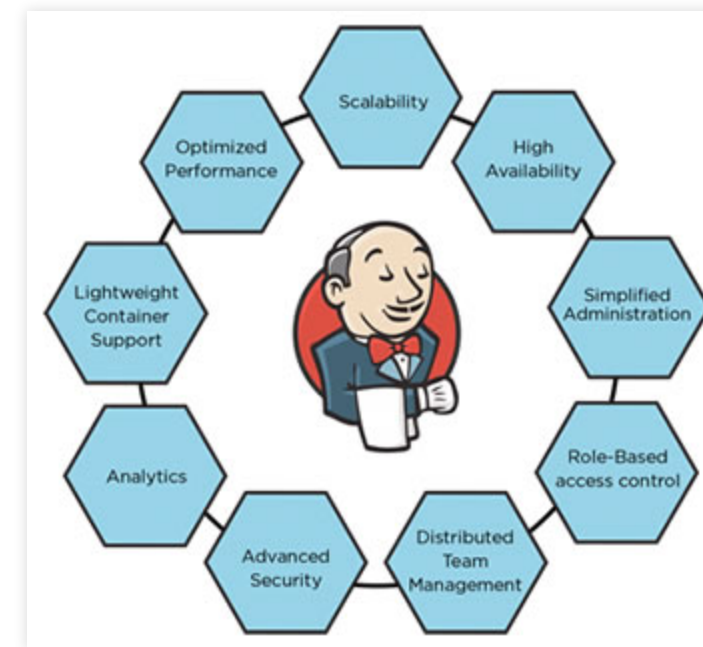
Jenkins acquière de nouvelles fonctionnalités via des plugins, ils en existent un grand nombre. Les formulaires et les champs de l'interface utilisateur dépendent des plugins installés par l'administrateur.

La version 2 de Jenkins sortie en 2016 propose un nouveau type de jobs **pipeline** écrit en groovy (language de programmation du monde java). Le pipeline permet de :

- définir une chaine de traitements très simplement ;
- scripter la définition de jobs, plutôt que de renseigner des formulaires ;
- versionner le script avec le code de l'application ;
- mettre à disposition plus de fonctionnalités : exécution, conditions, gestion des exceptions... ;

# Historique

- 2004 : projet initié par Kohsuke Kawaguchi (Sun) sous le nom *Hudson* ;
- 2010 : principale solution d'intégration continue ;
- 2011 : fork de nom pour *Jenkins*, dépôt du code sur github (<https://github.com/jenkinsci>). Depuis Hudson est également sur github (<https://github.com/hudson>) ;
- 2016 : Jenkins version 2, introduction des pipelines (workflow) et l'interface BlueOcean ;
- 2018 : Jenkins X, projet associant Jenkins et Kubernetes



Historiquement, Jenkins est un outil de CI/CD, mais il peut également être utilisé pour certaines tâches d'administration et de maintenance d'infrastructure (ex : plugin ansible).

# Installation

---

Jenkins est une application java, seul pré-requis.

Elle s'installe très simplement à partir des packages à disposition (<https://jenkins.io/download/>).

Des images docker de Jenkins sont également disponibles.

## Sous Ubuntu/Debian

---

Installation à partir du repo debian :

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -  
sudo apt-get update  
sudo apt-get install jenkins
```

## A partir d'une image docker

---

Vérifier que docker est installé et prêt à l'emploi :

```
docker info
docker version
```

Sinon, installer docker avec le script [install docker on centos](#) .

```
docker container run -d -u 1001 -p 8080:8080 -v /var/lib/jenkins -e JENKINS_PASSWORD=my_password openshift/jenkins-2-centos7:v3.11
```

Accès à Jenkins dans un navigateur : <http://127.0.0.1:8080>



# Interface utilisateur

Démonstration :

- comment consulter un job
- comment lancer un job manuellement
- comment consulter/modifier la configuration d'un job

## Note

Projet OpenShift : la trace de l'exécution d'un job se voit directement dans le dashboard d'OpenShift, il n'est pas nécessaire d'aller sur l'interface graphique de Jenkins, mais elle reste accessible.

# Installation de plugins

Jenkins de base propose peu de fonctionnalités. Pour personnaliser l'outil en fonction des besoins du projet, il faut installer les plugins nécessaires.

Dans la partie *Manage Jenkins/Plugin Manager* : dans l'onglet *Available*, rechercher et sélectionner le(s) plugin(s).

The screenshot shows the Jenkins Plugin Manager interface. The top navigation bar includes the Jenkins logo, a red notification badge with the number '3', a search bar, and links for 'Jenkins Admin' and 'log out'. The breadcrumb trail shows 'Jenkins' > 'Plugin Manager'. On the left sidebar, there are links for 'Back to Dashboard' and 'Manage Jenkins'. The main content area has tabs for 'Updates', 'Available', 'Installed', and 'Advanced'. The 'Available' tab is selected, and a search filter 'slack' is applied. A table lists available plugins with columns for 'Install' (checkbox), 'Name', and 'Version'. The 'Slack Notification' plugin is selected with a checked checkbox. Below the table, there are buttons for 'Install without restart' and 'Download now and install after restart', along with a timestamp 'Update information obtained: 4 min 29 sec ago'. The footer indicates 'Page generated: 10-Aug-2018 13:26:14 UTC' and 'Jenkins ver. 2.121.2'.

Install	Name	Version
<input type="checkbox"/>	<a href="#">Slack Upload</a> A post-build uploader that uploads files to slack generated during build process	1.7
<input type="checkbox"/>	<a href="#">cucumber-slack-notifier</a> This plugins posts summarised Cucumber report information to Slack	0.8.3
<input checked="" type="checkbox"/>	<a href="#">Slack Notification</a> This plugin is a Slack notifier that can publish build status to Slack channels.	2.3
<input type="checkbox"/>	<a href="#">Build Notifications</a> Send build notifications through Telegram, Pushover, Boteco or Slack.	1.5.0
<input type="checkbox"/>	<a href="#">Global Slack Notifier</a> This plugin post to slack after any build completed without any job setting.	1.3

Exemple : installation d'un plugin pour effectuer des notifications avec l'outil *slack*

L'installation d'un plugin peut modifier l'interface : ajout de nouvelles icônes et de nouvelles options.

Certains mettent également à disposition des variables et des fonctions à utiliser dans les fichiers de description des jobs (ex: git, docker ...)

#### Note

Attention au choix des plugins : il est conseillé d'avoir le minimum de plugins utiles.

Catalogue des plugins : <https://plugins.jenkins.io>

# Configuration de la gestion des jobs

---

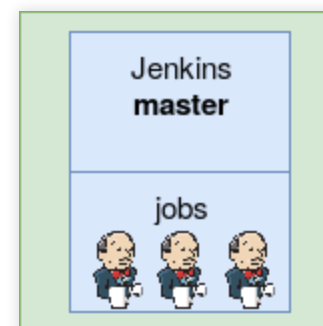
## Mode Standalone

---

Une seule instance qui lance les jobs en local. Il est nécessaire d'installer sur le serveur Jenkins tous les outils et paquets nécessaires pour l'exécution des jobs. (Ex: docker, ansible, python, ...)

### Avertissement

Toutes les actions sont effectuées sur le même environnement (donc pas d'isolation).



## Mode master & nodes

---

Le master porte la gestion des jobs et l'interface graphique, mais l'exécution se fait sur d'autres serveurs : les nodes (ou slaves). Cette organisation permet d'avoir des nodes dédiés :

- à certaines opérations : installation d'applications particulières pour le build, le test...
- à avoir différentes distributions pour réaliser les tests fonctionnelles d'une application : nodes sous linux et des nodes sous windows ;
- simplement décharger la machine master s'il y a beaucoup de jobs à gérer ;

### Note

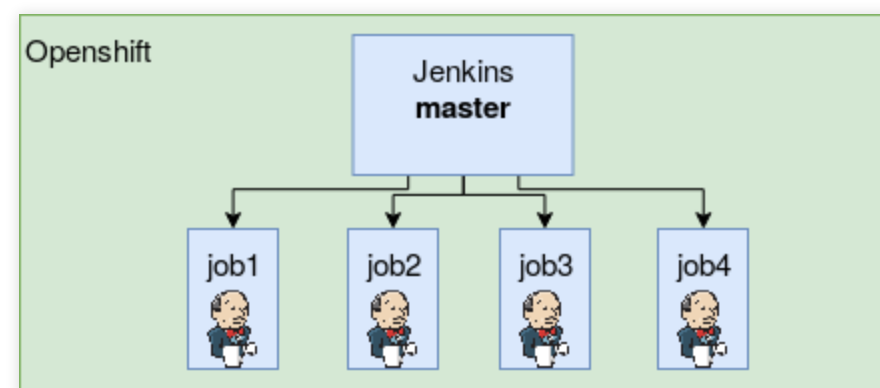
Avec les jobs de type pipeline, il est possible d'utiliser plusieurs nodes pour un même job.

## Projet OpenShift

---

Dans le cadre de **BeAPI**, la chaîne de traitement sera créée automatiquement avec une instance Jenkins master prête à l'emploi pour les utilisateurs, et des slaves instanciés à la volée dans Openshift.

Elle est configurée dans OpenShift et répond aux besoins du projet en terme de droits d'accès et de plugins.



# Jenkins & les APIs

---

Jenkins expose une API REST : <http://127.0.0.1:8080/api/>

Plusieurs outils permettent de consommer cette API :

- `jenkins-cli` (java) : CLI fourni par Jenkins ;
- `python-jenkins` (python) : wrapper python développé par la communauté OpenStack ;
- `JJB Jenkins Job Builder` (python) : CLI basé sur `python-jenkins` permettant la création automatique de jobs à partir de fichiers de description en yaml ;



# Jobs

---

*1. Introduction*

**2. Jobs**

3. Pipeline

4. Scripted Pipeline

5. Fonctionnalités avancées

6. Liens utiles

# Objectifs

---

- Présentation des jobs
- Présentation des *jobs builds triggers*
- Composants communs à tous les jobs
  - création de paramètres
  - création de credentials
  - création de variables globales

# Définition de jobs

---

Les **jobs** sont les principaux objets manipulés dans Jenkins.

Un job est la description d'une ou plusieurs **tâche(s)** à effectuer. Un job se caractérise par :

- Un nom
- Un type

## Type de jobs

---

Il existe plusieurs types de jobs, la liste varie en fonction des plugins installés :

- **freestyle** : historiquement un des premiers type, le job est configuré à partir de formulaires ;
- **pipeline** : permet de créer des jobs à partir de fichiers de script ;


Note

Les scripts de **pipeline** sont généralement versionnés avec le code de l'application. Ils peuvent être utilisés dans plusieurs jobs

Vous pouvez également créer des objets de type **folder**, cela permet de créer une arborescence de jobs.

Note


Projet OpenShift : utilisation uniquement des jobs pipeline

 **Jenkins** 3  [Jenkins Admin](#) | [log out](#)


Jenkins ▾ ▸ All ▸ ▹

### Enter an item name


» Required field

**Freestyle project**


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

**Pipeline**


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

**Multi-configuration project**


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

**Bitbucket Team/Project**


Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

**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.

If you want to create a new item from other existing, you can use this option:

 Copy from

127.0.0.1:8099

Grandes parties d'un job :

1. **General** : nom et description du job et ajout de paramètres ;
2. **Build Trigger** : règle pour déclencher le lancement d'un build du job ;
3. Définition du job :
  - soit avec des formulaires : pre-build, build, post-build, méthode historique ;
  - soit avec un script, pour le job de type pipeline : **méthode à privilégier** ;

Le code source de l'application avec les scripts est récupéré depuis un SCM (exemple : *github entreprise*).

## Lancement des jobs *builds trigger*

---

Un **job** correspond à une ou plusieurs tâches exécutées *séquentiellement*.

Un **build** correspond à une exécution d'un job, par défaut si une tâche échoue, l'exécution du job s'arrête en ignorant les tâches suivantes.

Chaque build possède un status final :

- **SUCCESS** : toutes les tâches se sont exécutées correctement ;
- **FAIL** : une des tâches a échoué, les suivantes sont ignorées ;
- **ABORTED** : le job n'a pas pu être lancé ou a été interrompu ;

Dans le dashboard, chaque job possède un logo *météo* pour illustrer le pourcentage de builds ayant réussis.

[Back to Dashboard](#)
[Status](#)
[Changes](#)
[Build Now](#)
[Delete Pipeline](#)
[Configure](#)
[Full Stage View](#)
[Open Blue Ocean](#)
[Rename](#)
[Pipeline Syntax](#)
[Polling Log](#)

## Pipeline demo

[add description](#)

[Recent Changes](#)

[Disable Project](#)

### Stage View

Average stage times:  
(Average full run time: ~8s)

	01- clean	02- function sh	03- modification resultat	04- boucle for	05- execution conditionnelle	06- gestion des erreurs
#5	647ms	1s	799ms	1s	638ms	195ms
#4	730ms	2s	609ms	1s	698ms	152ms
#3	671ms	1s	616ms	1s	830ms	198ms
#2	776ms	2s	602ms	2s	716ms	182ms
#1	981ms	1s	983ms	2s	755ms	207ms

#### Build History

find x

- #5 10-Aug-2018 14:11
- #4 10-Aug-2018 14:08
- #3 10-Aug-2018 14:08
- #2 10-Aug-2018 14:07
- #1 10-Aug-2018 14:06

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



Plusieurs modalités de *build trigger* :

Le lancement manuel est disponible pour tous les jobs depuis l'interface graphique.

- **avec une API** : jenkins-cli ou python-jenkins ;
- **avec un cron** : lancement à interval régulier. La syntaxe utilisée est la même que pour la définition d'un cron sous Linux ;
- **avec un poll** : solution adaptée pour des projets sous git ; contact du dépôt git à interval régulier (même syntaxe que le cron) :
  - si des changements ont été effectués dans le code (nouveaux commits) : lancement d'un build ;
  - si pas de changement, rien à faire;
- **avec un hook** : configure dans un projet git un hook vers un ou plusieurs jobs de Jenkins, en fonction de la survenue de certains évènements (commit, merge, pull-request...) le job est lancé. Le statut du job est alors affiché dans le projet git.

## Exemples de syntaxe *cron*

```
# Execution toutes les 5 minutes
cron(H/5 * * * *)

# Toutes les 4 heures du lundi au vendredi
cron(H */4 * * 1-5)
```

Type	Content
MINUTE	Minutes within the hour (0–59)
HOUR	The 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.

Documentation : [https://en.wikipedia.org/wiki/Cron#CRON\\_expression](https://en.wikipedia.org/wiki/Cron#CRON_expression)

Note

Projet OpenShift : le lancement des jobs est g  r   par OpenShift

Jenkins » demo »

General **Build Triggers** Advanced Project Options Pipeline

### Build Triggers

☐ Build after other projects are built ?

☒ Build periodically ?

Schedule  ?

Would last have run at Friday, August 10, 2018 12:33:25 PM UTC; would next run at Friday, August 10, 2018 4:33:25 PM UTC.

☐ GitHub hook trigger for GITScm polling ?

☒ Poll SCM ?

Schedule  ?

Would last have run at Friday, August 10, 2018 1:53:41 PM UTC; would next run at Friday, August 10, 2018 1:58:41 PM UTC.

Ignore post-commit hooks ☐ ?

☐ Disable this project ?

☐ Quiet period ?

☐ Trigger builds remotely (e.g., from scripts) ?

# TP1

---

## Objectifs

---

- Créer son premier job ;
- Lancer un job ;
- Consulter le log d'un build ;

## Sujet

---

- Créer dans Jenkins un job de type Freestyle ;
- Ajouter le code suivant dans le partie build :

```
echo "Formation ADEO"
```

- Lancer le job manuellement et consulter le log

# Ajout de paramètres dans un job

---

Dans plusieurs types de jobs Jenkins, il est possible de définir des paramètres, dans la partie **General**. Il permet de passer des valeurs aux jobs à exécuter.

Paramètres les plus utilisés :

- **string** ;
- **boolean** : case à cocher ;
  - décocher => False ;
  - cocher => True ;
- **choices** : liste de choix restreints ;
- **credentials** : choix parmi les credentials déjà définis ;

Jenkins > demo2 >

**General** Build Triggers Advanced Project Options Pipeline

☐ GitHub project

☒ This project is parameterised ?

String Parameter X ?

Name param\_name ?

Default Value undefined ?

Description Description consultable depuis la page du job ?

[Plain text] [Preview](#)

☐ Trim the string ?

Add Parameter ▼

☐ Throttle builds ?

**Build Triggers**

☐ Build after other projects are built ?

**Save** **Apply**



# Création de credentials

---


Jenkins est capable de stocker de manière sécurisée différents types de données dont :


- des utilisateurs/mots de passe ;
- des clés privées (pour les connexions SSH) ;
- des fichiers ;
- du texte ;
- des certificats ;


La portée des credentials peut être :


- globale : utilisable par tous les jobs dans Jenkins ;
- propre à un sous-domaine : équivalent à des folders ou dossier au sein de Jenkins. Les credentials créés dans un sous-domaine ne sont pas accessibles ailleurs ;

Jenkins > Credentials > System > Global credentials (unrestricted) > admin/\*\*\*\*\* (git-access)

 [Back to Global credentials \(unrestricted\)](#)

 [Update](#)

 [Delete](#)

 [Move](#)

Scope

Global (Jenkins, nodes, items, all child items, etc) ?

Username

admin P ?

Password

..... P ?

ID

git-acces ?

Description

git-access ?

Save

#### Note

Projet OpenShift : les credentials sont créés dans OpenShift via des fichiers yaml et chargés automatiquement dans Jenkins.

# Création de variables globales

Jenkins permet de définir des variables globales dans la partie *configuration*. Ces variables sont disponibles dans tous les jobs.

The screenshot shows the Jenkins configuration page for 'Global properties'. The breadcrumb navigation at the top indicates 'Jenkins > configuration'. Under the 'Global properties' section, the 'Environment variables' checkbox is checked. Below this, there is a 'List of variables' section containing two entries. Each entry has a 'Name' field and a 'Value' field, with a red 'Delete' button to the right. The first entry has the name 'GLOBAL\_JENKINS\_VARIABLE' and the value 'env\_jenkins\_variable'. The second entry has the name 'PYTHONPATH' and the value '/usr/bin/python2.7', with a blue help icon to its right. An 'Add' button is located below the list. At the bottom left, there is an unchecked 'Tool Locations' checkbox. At the bottom, there are 'Save' and 'Apply' buttons.

Name	Value	Action
GLOBAL_JENKINS_VARIABLE	env_jenkins_variable	Delete
PYTHONPATH	/usr/bin/python2.7	Delete

Buttons: Add, Save, Apply

# TP2

---

## Objectifs

---

- créer un secret Git

## Sujet

---

- Créer un secret Git **global** de type utilisateur/mot de passe
- Renseigner :
  - Utilisateur : nom d'utilisateur GitHub
  - Mot de passe : Token GitHub

# Pipeline

---

*1. Introduction*

*2. Jobs*

**3. Pipeline**

4. Scripted Pipeline

5. Fonctionnalités avancées

6. Liens utiles

# Objectifs

---

- Présentation des jobs de type *pipeline*
- Comparaison de deux syntaxes de jobs possibles
  - *scripted-pipeline* ;
  - *declarative-pipeline* ;
- Premiers jobs



# Introduction

---

Les jobs de type pipeline sont des scripts, par convention les fichiers s'appellent des **Jenkinsfile** ou utilisent seulement l'extension *.groovy*.

Le pipeline est découpé en une ou plusieurs étapes. Ils apparaissent dans le rendu d'exécution du job dans Jenkins.

Saisie en directe :

The screenshot shows the Jenkins web interface for configuring a pipeline. The breadcrumb navigation at the top indicates 'Jenkins' > 'demo'. The configuration tabs are 'General', 'Build Triggers', 'Advanced Project Options', and 'Pipeline', with 'Pipeline' being the active tab. The main section is titled 'Pipeline' and contains a 'Definition' subsection. A dropdown menu labeled 'Pipeline script' is set to 'Pipeline script'. Below this, a 'Script' editor is shown with a single script block labeled '1'. The editor contains a text area for writing the pipeline script. To the right of the script block is a dropdown menu labeled 'try sample Pipeline...' and a help icon. Below the script editor, there is a checkbox labeled 'Use Groovy Sandbox' which is checked. A link for 'Pipeline Syntax' is located below the checkbox. At the bottom left of the configuration area, there are two buttons: 'Save' and 'Apply'.

# Syntaxe des scripts Pipeline (Jenkinsfile)

Le script décrit une suite d'étapes *stage* qui seront exécutées séquentiellement.

Chaque *stage* contient le code à exécuter grâce à des fonctions mises à disposition par les plugins pipelines.

## Note

La liste des fonctions est dépendantes des plugins installées.

Ex : plugin git apporte sa fonction *git* pour faciliter l'écriture des commandes git.

Exemple de la fonction **sh** pour exécuter des commandes shell :

- sh "python -version" : exécution de la commande **python -version**
- Si python est présent la version est affichée et le job continue
- Sinon la commande échoue, le job s'arrête et le build du job est au status FAILED

*Scripted pipeline*

```
node {
  stage ('Build') {
    sh 'echo Commands to build application'
  }
  stage ('Test') {
    sh '''
    echo Commands to test application
    echo Other tests
    '''
  }
  stage ('Deploy') {
    sh 'echo Command to deploy application'
  }
}
```

*Declarative pipeline*

```
pipeline {
  agent any

  stages {
    stage('Build'){
      steps{
        sh "echo Command to build Application"
      }
    }
    stage('Test'){
      steps{
        sh "echo Command to test Application"
      }
    }
    stage('Deploy'){
      steps{
        sh "echo Command to deploy Application"
      } // end steps
    } // end stage
  } // end stages
} // end pipeline
```

## Avertissement

Le nom des stages doit être unique dans un pipeline.

# TP3

---

## Objectifs

---

- Créer un job pipeline

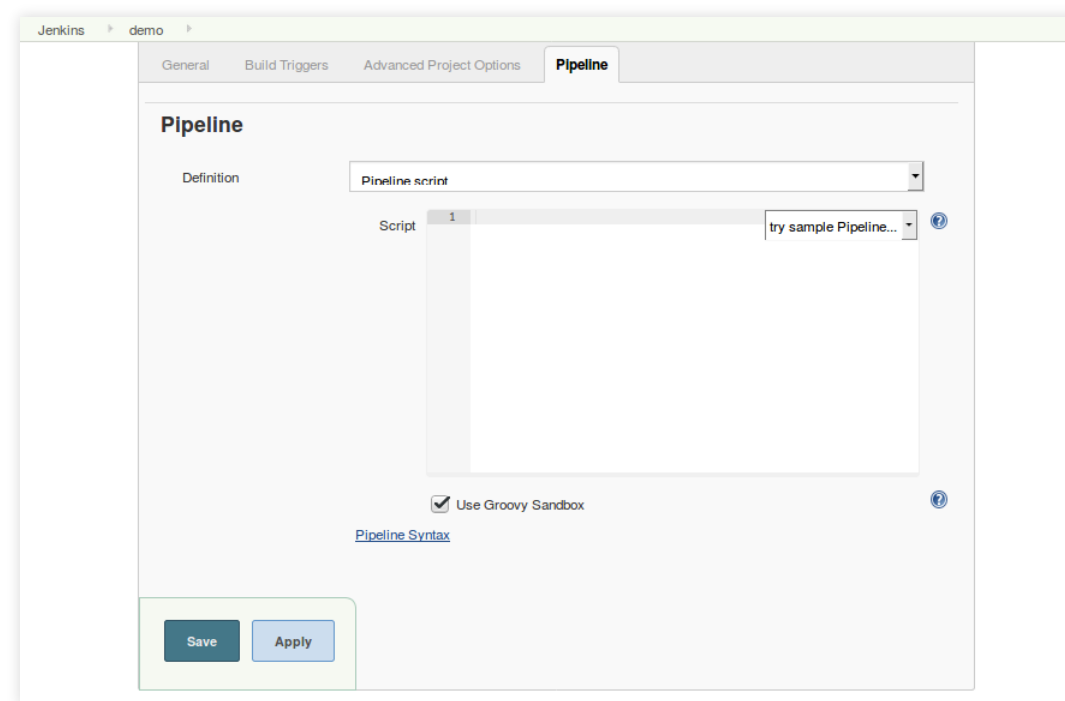
## Sujet

---

- Créer un job pipeline avec le script :

```
node{
  stage('Hello'){
    echo "Hello ADEO"
  }
}
```

- Utiliser l'entrée directe :



- Lancer le job

# TP4

---



## Objectifs

---

- Utiliser Git comme source pour un job pipeline

## Sujet

---

- Créer un projet sur GitHub, et pousser un fichier `tp4.groovy` contenant :

```
node{
  stage('Hello'){
    echo "Hello ADEO"
  }
}
```

- Créer un job pipeline qui utilise le projet git (*Pipeline from SCM*) et renseigner le script `tp4.groovy`, ainsi que le secret Git créé au *TP 2*
- Exécuter le job manuellement

# Scripted vs. Declarative pipeline

---

**Scripted pipeline** : Première solution disponible

La description des opérations commence par **node**, les scripts sont écrits en groovy.

Les possibilités sont infinies, on peut utiliser les bibliothèques de Jenkins et toutes les structures de contrôle et de condition (boucle for, les conditions if, la gestion des erreurs ...).

**Documentation scripted pipelines**

## Declarative pipeline : Nouvelle solution

La description des opérations commence par **pipeline**, les scripts sont écrits dans un langage déclaratif plus simple à prendre en main pour débiter ou réaliser des opérations simples.

### Note

Il est possible d'inclure du code groovy dans un declarative pipeline.

Elle offre en plus :

- une aide pour l'écriture du Jenkinsfile. Dans chaque job pipeline, le lien **Pipeline Syntax** ;
- un validateur accessible via l'API de Jenkins :

```
curl -X POST -F "jenkinsfile=<Jenkinsfile" http://127.0.0.1:8080/pipeline-model-converter/validate
```

## Documentation declarative pipelines

## Scripted pipeline

---

Structure minimal :

```
node{  
  // code  
}
```

Structure globale d'un fichier, seule la partie node est obligatoire :

```
// import de classe
import jenkins.model.*

// définition de variable globale
def project_name = 'test1'

// définition de fonction
def affichage () { // code }

properties([
    // Configuration du job: build trigger, log rotate
])

node('slave_1'){
    // code groovy exécuté sur le slave_1
}

node{
    // code groovy
}
```

## Scripted pipeline **node**

- Indique que le code est écrit avec un script groovy ;
- Permet de préciser sur quel node exécuter le code en fonction les labels associés au node ;

```
node {  
  
    // code groovy  
    // definition de variable, accessible à tous les stages suivants  
  
    // liste des stages  
}  
  
node ('slave_1') { // code }
```

### Note

À la création d'un node (ou slave) dans Jenkins, on lui associe un ou plusieurs labels. Un même label peut renvoyer vers un ou plusieurs nodes.

[Documentation node](#)

## Scripted pipeline **stage**

- Constitue un block de code, doit être compris dans une instruction *node* ;
- Représente une colonne dans le tableau de rendu des builds ;

```
stage('nom_unique'){  
    // code groovy  
  
    // Appel de la fonction groovy echo pour afficher un message  
    echo 'Hello'  
}
```

## Documentation stage



## Declarative pipeline

---

L'organisation du code est beaucoup plus figée, mais la lecture est plus intuitive.

Exemple de syntaxe minimale obligatoire :

```
pipeline {  
    agent any  
  
    stages {  
        stage('Build'){  
            steps{ sh 'echo hello'}  
        }  
    }  
}
```

Détails de la syntaxe : <https://jenkins.io/doc/book/pipeline/syntax/>

## Declarative pipeline *pipeline*

Exécution séquentielle :

```
pipeline {  
    // code déclaratif  
}
```

## Declarative pipeline **agent**

Permet de :

- renseigner le node qui va executer le code ;
- renseigner l'image docker à lancer pour exécuter le code ;

Note

Il peut être surchargé à l'intérieur d'un stage pour certaines opérations

```
# N'importe quel node  
agent any
```

```
# Spécifie un node avec son label  
agent { label 'labelName' }
```

```
# Utilisation d'un conteneur docker
agent {
  docker { image 'node:7-alpine' }
}
```

```
# dans un stage particulier
stage('Back-end') {
  agent {
    docker { image 'maven:3-alpine' }
  }
  steps { sh 'mvn --version' }
}
```

Documentation agent

## Declarative pipeline **stages/stage**

- **stages** : contient un ou plusieurs blocks *stage* ;
- **stage** : possède un paramètre obligatoire, le nom du stage qui doit être unique ;
- Permet de définir un contexte propre au stage (Ex: en surchargeant l'agent principale) ;

```
stages {
  stage('Etape') {
    steps { echo "Etape: actions" }
  }
}

stages {
  stage('Etape 1') {
    steps { echo "Etape 1: actions" }
  }
  stage('Etape 2') {
    steps { echo "Etape 2: actions" }
  }
}
```

[Documentation stages/stage](#)

## Declarative pipeline **steps**

- Obligatoirement inclus dans un block stage ;
- Ne possède pas de paramètres ;
- Contient le code à exécuter ;

Liste de toutes les références : <https://jenkins.io/doc/pipeline/steps/>

Celui dédié à OpenShift Pipeline sera présenté plus tard : <https://jenkins.io/doc/pipeline/steps/openshift-pipeline/>

[Documentation steps](#)

# TP5

---

## Objectifs

---

- Créer un scripted pipeline avec plusieurs stages



## Sujet

---

- Créer un job tp5 de type pipeline en lien avec le projet git ;
- Créer un fichier tp5.groovy :
  - reprendre le stage Hello du tp4 ;
  - ajouter un nouveau stage pour afficher les variables d'environnement du node (fonction shell env) ;
- Pusher le code sur Github et lancer le job ;

# Variables

---

Les scripts utilisent des variables pour rendre leur utilisation plus générique et réutilisable dans différents contextes.

On peut manipuler quatre type de variables dans un job :

- variable d'environnement ;
- variable Groovy ;
- paramètre du job ;
- credentials ;

## Variable d'environnement

---

Jenkins fournit par défaut un ensemble de variables d'environnement.

Ces variables peuvent être utilisées :

- En shell :

```
sh "echo $BUILD_NUMBER"
sh "echo ${env.BUILD_NUMBER}"
```

- En groovy :

```
echo BUILD_NUMBER
echo env.BUILD_NUMBER
echo "build number: $BUILD_NUMBER"
echo "build number: ${env.BUILD_NUMBER}"
```

Liste des variables d'environnement fournies par votre instance Jenkins :

<http://127.0.0.1:8080/env-vars.html>

Ex :

- WORKSPACE : dossier d'exécution du job ;
- GIT\_BRANCH : branch du projet git cloné ;
- BUILD\_NUMBER : numéro du build courant ;

Il est possible de définir ses propres variables d'environnement.

*Scripted pipeline :*

utilisation de la fonction `withEnv()` avec une liste de définition de variables :

```
node {  
  
  stage('Stage avec variable') {  
  
    withEnv([  
      "ENV_VARIABLE_NAME='value'",  
      "MODE_DEBUG=true",  
    ]){  
      // Block de code qui peut  
      // utiliser ces variables  
      sh 'env'  
    }  
  }  
}
```

*Declarative pipeline :*

utilisation du block `environment{}` :

```
pipeline {  
  agent any  
  environment {  
    ENV_VARIABLE_NAME = 'value'  
    MODE_DEBUG = true  
  }  
  stages {  
    stage('Example stage 2') {  
      steps {  
        // Affichage de toutes  
        // les variables d'environnement  
        sh 'env'  
      }  
    }  
  }  
}
```

## Variables Groovy

---

De nombreuses variables groovy sont accessibles dans les pipelines :

<http://127.0.0.1:8080/pipeline-syntax/globals>

Exemple : **currentBuild** donne des informations sur le build courant.

```
// Affichage
echo currentBuild.description
echo "Result: ${currentBuild.result}"

// Changement de la valeur de la variable
currentBuild.result = 'FAILED'
```

Les variables donnent parfois accès à des fonctions :

```
echo "${currentBuild.resultIsBetterOrEqualTo('UNSTABLE')} "
```

La liste des variables disponible varie en fonction des plugins installés et de la version de Jenkins.

Exemple :

- **Variables pour OpenShift** : [http://URL\\_JENKINS/pipeline-syntax/globals#openshift](http://URL_JENKINS/pipeline-syntax/globals#openshift). Elles sont présentes dans notre instance Jenkins car les plugins OpenShift sont installés
- **Variables pour Docker** : [http://URL\\_JENKINS/pipeline-syntax/globals#docker](http://URL_JENKINS/pipeline-syntax/globals#docker). Elles sont présentes dans notre instance Jenkins car le plugin Docker est installé

Il est possible de définir des variables locales à utiliser dans un script groovy :

### *Scripted Pipeline*

```
node {
    def GLOBAL_VAR = "set_global_var"

    stage("first") {
        def LOCAL_VAR = "set_local_var"
        sh "echo $GLOBAL_VAR and $LOCAL_VAR"
    }
    stage("second") {
        def LOCAL_BOOL = false
        print GLOBAL_VAR + " and " + LOCAL_BOOL
    }
}
```

*Declarative pipeline*: obligation de passer par un block `script{}`

```
pipeline {
    agent any

    stages {
        stage("foo") {
            steps {
                script{
                    def String LOCAL_VAR = "set_local_var"
                    def Boolean LOCAL_BOOL = true

                    sh "echo Local variable, string : $LOCAL_VAR"
                    sh "echo Local variable, bool : $LOCAL_BOOL"
                }
            }
        }
    }
}
```



## Paramètres de jobs

---

La variable `params` permet de récupérer des paramètres définis :

- Dans le Job Jenkins
- Dans le Jenkinsfile

```
echo params.PARAM_NAME  
echo "${params.PARAM_NAME}"  
  
// Récupération dans une variable  
def myvar = params.PARAM_NAME
```

### Note

Avec la fonction `sh`, ils sont accessibles comme variable d'environnement, sans le préfix *params* :

```
sh 'echo $PARAM_NAME'
```

## Credentials

---

Jenkins permet de stocker et chiffrer des données sensibles:

Les principaux :

- login/password, 2 formats de sorties :
  - *conjoined* : une variable contenant **login:password**
  - *separated* : 2 variables, une pour **login**, une pour **password**
- clé ssh ;
- fichier ;
- token ;

Lors de leur création, les credentials ont un **scope** limitant leur domaine d'utilisation :

- pour Jenkins (tous les jobs) ;
- pour un seul dossier (seuls les jobs définis dans ce dossier peuvent utiliser le credential) ;

Les credentials peuvent être déchiffrés dans des variables d'environnement lors des builds afin de pouvoir les utiliser.

#### Note

Les credentials ne sont jamais affichés dans les logs (remplacés par \*\*\*\*\*)

## Scripted pipeline

On utilise `withCredentials([])` qui prend en paramètre une liste de credentials à récupérer.

Exemple de récupération d'un login/password et d'un fichier :

```
stage ('stage avec credentials'){

  withCredentials([
    usernamePassword(
      credentialsId: 'credentials_name_for_login',
      usernameVariable: "variable_name_for_user",
      passwordVariable: "variable_name_for_password"
    ),
    file(
      credentialsId: 'credentials_name_for_file',
      variable: 'secret_file')
  ]) {

    // Block de code qui peut utiliser ces variables
    // Connexion à une application avec login/mot de passe

    // et passage du fichier de configuration avec connexion à une base de données
  }
}
```

## Declarative pipeline

Les variables sont définies dans un block **environment**.

On utilise la fonction **credentials()** pour déchiffrer les credentials dans des variables.

Exemple récupération de login/password et d'un fichier :

```
pipeline {
  agent any

  // accessible globalement
  environment {
    variable_for_login = credentials('credentials_name_for_login')
  }

  stages {
    stage('Example stage 1') {
      // accessible dans le stage uniquement
      environment {
        variable_for_file = credentials('credentials_name_for_file')
      }
      steps {
        // affichera *****
        echo variable_for_login
        echo variable_for_file
      }
    }
  }
}
```

Le credentials *credentials\_name\_for\_login* contient deux données, le login et le password. L'appel à la fonction credentials("credentials\_name\_for\_login") va instancier trois variables :

- variable\_for\_login contient **login:password** ;
- variable\_for\_login\_USR contient **login** ;
- variable\_for\_login\_PSW contient **password** ;

[Documentation credentials](#)

# Lancement d'un autre job

---

Comment dans un job lancer un autre job ?

Avec la fonction `build()`.

## Job simple

---

```
stage('Hello') {  
  steps {  
    build 'folder_name/job_name'  
  }  
}
```



## Job avec paramètres

---

Exemple avec 3 paramètres de type string, boolean, credentials, passés sous la forme d'une liste.

```
stage('Hello') {
  steps {
    build job: 'folder_name/job_name',
      parameters: [
        string(name: 'parameter_name', value: 'default_value'),
        booleanParam(name: 'boolean_param_name', value: true),
        credentials(description: 'description', name: 'credential_parameter_name', value: '')
      ]
  }
}
```

# TP6

---

## Objectifs

---

- Apprendre à identifier des erreurs
- Utiliser des paramètres
- Utiliser des variables d'environnement
- Utiliser des secrets

## Sujet

---

Télécharger le tp6 :

- Créer un job pipeline pour le tp6
- Exécuter le job, et corriger les erreurs
- Modifier la section *1 - Variables d'environnement* pour afficher le numéro de build du job.
- Modifier la section *3 - Récupération des secrets* pour afficher :
  - L'utilisateur et le mot de passe
  - Le chemin du fichier de secret

# Scripted Pipeline

---

*1. Introduction*

*2. Jobs*

*3. Pipeline*

**4. Scripted Pipeline**

5. Fonctionnalités avancées

6. Liens utiles

# Objectifs

---

- Instruction de boucle
- Instruction conditionnelle
- Gestion des erreurs
- Définition de fonctions

# Instruction de boucle

---

## Utilisation de **for**

---

Syntaxe: **for** (item : list) { // code }

```
node {
    // Parcours d'un fichier ligne à ligne
    sh 'env > env.txt'
    for (String i : readFile('env.txt').split("\n")) {
        println i
    }

    // Parcours d'une liste
    def my_list = ['val1', 'val2', 'val3']
    for (String val in my_list){
        println val
    }
}
```



## Utilisation de **each**

---

Le rendu sera le même avec for ou each mais la syntaxe est plus compacte.

Syntaxe: **list.each { // code }**

```
node {  
  
    // Parcours d'une liste  
    def my_list = ['val1', 'val2', 'val3']  
  
    // Parcours de la liste  
    my_list.each { val -> println val }  
  
}
```

# Instruction conditionnelle

---

## Utilisation de **if**

---

Syntaxe: **if (condition) { // code si vrai } else { // code si faux }**

```
node {  
  
    // Contenu du job  
  
    stage('last'){  
  
        if (currentBuild.result == "FAILURE"){  
  
            println "Job ${env.JOB_NAME} [${env.BUILD_NUMBER}] FAILED"  
  
        } else if (currentBuild.result == "SUCCESS") {  
  
            println "Job ${env.JOB_NAME} [${env.BUILD_NUMBER}] SUCCESS"  
            println "Le code peut être tagger et l'application déployée"  
  
        } else {  
  
            println "Job ${env.JOB_NAME} [${env.BUILD_NUMBER}] status UNDEFINED"  
  
        }  
    }  
}
```

## Utilisation de **switch**

---

```
node {
    stage('05- execution conditionnelle'){

        sh 'env > env.txt'

        for (String i : readFile('env.txt').split("\n")) {

            // Extraction du prefix de chaque variable
            def prefix = i.split('_')[0]

            switch (prefix){
                case 'JENKINS':
                    // Code : commande ou appel de fonctions
                    println "Variable préfixée par jenkins " + i
                    break

                case 'GIT':
                    println "Variable préfixée par GIT " + i
                    break

                case 'BUILD':
                    println "Variable préfixée par BUILD " + i
                    break

                default :
                    // Rien à faire
                    break

            }
        }
    }
}
```

# Gestion des erreurs

Si une commande renvoie une erreur ou lève une exception, le job passe en échec et le reste du code est ignoré.

Pour gérer les erreurs, on dispose des directives `try/catch/finally`, la partie *finally* est optionnelle.

La partie *catch* permet de traiter l'exception.

Pour faire échouer le job explicitement, on peut utiliser la fonction `error("message")` et redéfinir le statut du `currentBuild.result`.

```
try {  
    // Code pouvant levé une erreur  
    // si une erreur est levée le code suivant est ignoré  
  
} catch (Exception e){  
    // Traitement de l'erreur  
    // Action à faire, message à l'utilisateur  
  
} finally {  
    // Code executée avec ou sans erreur  
  
}
```

```
node {
  stage('06- gestion des erreurs'){

    println "Commande avant le try/catch"
    try {

      println "Commande pouvant générer une erreur"
      // Récupération d'un nombre
      def number = env.BUILD_NUMBER as Integer

      if (number%2) {
        println "Number " + number + " is impair"
        // levee une exception
        throw new Exception()
      }
      else { println "Number "+ number +" is PAIR" }

    } catch (Exception e) {
      println "FAIL : la commande échoue, traiter ce cas"
    } finally {
      println "Commande toujours exécutée"
    }
  }
}
```

**NB** : le try/catch peut être inclus dans un block stage ou peut inclure tous les blocks stages, dans ce cas le traitement au cas par cas des exceptions est plus compliquées.

# Utilisation de fonctions

---

Les scripted pipeline en groovy donnent accès à un ensemble de fonctions.

Cas de la fonction `sh` :

- présentation des différentes syntaxes possibles ;
- passage de paramètres ;
- récupération des valeurs de sortie ;

**NB** : la fonction `sh` n'a pas accès aux variables groovy définies dans le script. Pour les afficher utiliser *println*.

```
node {
    stage('02- function sh'){

        println "Syntaxe 1"
        sh 'git config remote.origin.url'

        println "Syntaxe 2"
        sh(script: 'git config remote.origin.url')

        println "Syntaxe 3"
        sh(script: 'git config remote.origin.url', returnStdout: true)

        println "Syntax 4"
        url = sh(returnStdout: true, script: 'git config remote.origin.url').trim()
        println url

        println "Syntax 5"
        def url2 = sh(returnStdout: false, script: 'git config remote.origin.url')
        println url2
    }

    stage('03- modification resultat'){

        // Utilisation de variable
        def git_cmd = 'git config remote.origin.url'

        // Extraction du nom du projet git
        gitname = sh(returnStdout: true, script: git_cmd).trim().split('/').last()

        println "Nom du projet git => " + gitname
    }
}
```



Il est possible de créer ses propres fonctions la directive `def`, comme les variables.

Elles peuvent être créées :

- à l'extérieur du block node ;
- à l'intérieur du block node : avant de l'utiliser ;

```
// Job
node{
    // Appel de la fonction
    function_without_params()
    def result = function_with_params(arg_value)
}

// Déclaration de fonction
def function_without_params(){
    // code
}

// Déclaration de fonction avec paramètres
def function_with_params(String arg1){
    // code
}
```

## Quelques fonctions utiles

fonction	action
<code>string_variable.trim()</code>	supprime les espaces en trop
<code>list = string_variable.split(character)</code>	découpe une chaine de caractère en liste de chaine de caractères en fonction du caractère de délimitation donnée
<code>string_variable.toUpperCase()</code>	met en majuscule
<code>list.size()</code>	retourne le nombre d'éléments
<code>list = findFiles</code>	permet de rechercher des fichiers dans un dossier, retourne les fichiers trouvés dans une liste
<code>timeout(delay){ code }</code>	arrête l'exécution du code après le temps
<code>retry(number){ code }</code>	relance le code un certain nombre de fois

## TP 7-2

---

## Objectifs

---

- Gestion des erreurs
- Création d'une fonction

## Sujet

---

Télécharger le script tp7-2

1. Créer un job pipeline pour le tp7-2;
2. Exécuter le job tp7 plusieurs fois, il génère une erreur si le numéro de build est impair ;
3. Décommenter les codes pour capturer l'erreur et relancer le job plusieurs fois ;
4. Modifier le stage « Gestion erreurs », remplacer les appels à la fonction groovy *println* par une fonction locale :
  - pour afficher le résultat du *if* ;
  - pour afficher un message d'erreur ;

# Fonctionnalités avancées

---

*1. Introduction*

*2. Jobs*

*3. Pipeline*

*4. Scripted Pipeline*

**5. Fonctionnalités avancées**

6. Liens utiles

# Objectifs

---

- Comment exécuter des traitements en parallèle
- Comment avoir des fonctions partagées entre jobs
- Comment notifier le status d'un build de job

# Exécution en parallèle

Lancement de traitement en parallèle pour gagner du temps d'exécution.

Exemples:

- lancement les mêmes tests unitaires / fonctionnels sur deux environnements :
  - différents navigateurs
  - différents systèmes d'exploitation
- lancement des tâches indépendantes.

```
node {
  stage('Parallel'){

    parallel (
      windows: { node {
        sh "echo building on windows now"
      }},
      mac: { node {
        sh "echo building on mac now"
      }}
    )
  }
}
```



# Utilisation des shared-libraries

---

Les *shared-libraries* sont des scripts groovy pouvant être utilisés dans tous scripts groovy, ainsi le code est écrit une fois et réutilisable partout.

1. créer un projet git pour stocker les scripts groovy ;
2. dans la partie admin de Jenkins: configurer le dépôt dans Jenkins, dans configuration de l'instance ;

3. dans le projet git créer un dossier *vars* , créer le fichier groovy contenant la première fonction ;

ex: vars/hello.groovy

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

def call(String name = 'human') {
    echo "Hello, ${name}."
}
```

ex: vars/clean\_projects.groovy

```
def call(body) {

    def config = [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = config
    body()

    node {
        stage ('Clean') {
            deleteDir()
            checkout scm
        }
    }
}
```

4. utiliser cette fonction dans un script groovy, elles sont chargées en haut du fichier puis appelées à partir du nom du fichier groovy ;

```
@Library("libraries/lib_name") _

node {
    // Appel de la fonction partagée
    clean_projects {}

    stage('First step'){
        sh 'ls -l'
        sh 'touch hello.txt'
    }

    stage('Hello'){
        // Appel de la fonction
        hello 'ADEO'
    }
}
```

# Notifications

Jenkins s'interface avec différents outils de communication pour l'envoi des notifications :

- une messagerie en configurant un serveur SMTP ;
- un outil de tchat : hitchat, slack, rocket ;

Exemple : envoi d'un email en cas d'échec, on reçoit un message qui décrit le job qui a échoué avec un lien vers Jenkins.

```
(...)  
  
if (currentBuild.result == 'FAILURE') {  
  
    emailtext (  
        subject: "FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'",  
        body: ""<p>FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]':</p>  
            <p>Check console output at "<a href="${env.BUILD_URL}">${env.JOB_NAME} [${env.BUILD_NUMBER}]</a>"</p>""",  
        recipientProviders: [[class: 'DevelopersRecipientProvider']]  
    )  
}
```

## Note

Projet OpenShift : les notifications fonctionnent si l'instance Jenkins a été configuré pour avoir accès à la messagerie ou au chat.

# TP 8

---

## Objectifs

---

- Utilisation de fonction partagée

## Sujet

---

Télécharger le script tp7-2

1. Créer un job pipeline pour le tp8 ;
2. Dupliquer tp7.groovy en tp8.groovy. Au lieu de créer des fonctions locales pour afficher les messages,
  - créer une fonction partagée **notify** pour afficher les messages ;
  - utiliser le même projet git.

# Liens utiles

---



# Jenkins

---

- <https://jenkins.io/doc>
- <https://github.com/jenkinsci>

# Variables et syntaxe (documentation instance Jenkins)

---

- [http://INSTANCE\\_JENKINS/env-vars.html](http://INSTANCE_JENKINS/env-vars.html)
- [http://INSTANCE\\_JENKINS/pipeline-model-converter/validate](http://INSTANCE_JENKINS/pipeline-model-converter/validate)
- [http://INSTANCE\\_JENKINS/pipeline-syntax/globals](http://INSTANCE_JENKINS/pipeline-syntax/globals)
- [http://INSTANCE\\_JENKINS/pipeline-syntax/globals#currentBuild](http://INSTANCE_JENKINS/pipeline-syntax/globals#currentBuild)

# Syntaxe (documentation en ligne)

---

- <https://jenkins.io/doc/book/pipeline/syntax/>
- <https://jenkins.io/doc/pipeline/steps/>
- <https://jenkins.io/doc/pipeline/steps/pipeline-utility-steps/>
- <https://github.com/jenkinsci/pipeline-examples>

# Plugins

---

- <https://plugins.jenkins.io>

# Jenkins & openshift

---

- [http://INSTANCE\\_JENKINS/pipeline-syntax/globals#openshift](http://INSTANCE_JENKINS/pipeline-syntax/globals#openshift)
- <https://github.com/jenkinsci/openshift-pipeline-plugin>
- <https://plugins.jenkins.io/openshift-pipeline>
- <https://jenkins.io/doc/pipeline/steps/openshift-pipeline/>

# Jenkins & API

---

- [http://INSTANCE\\_JENKINS/cli](http://INSTANCE_JENKINS/cli)
- <http://python-jenkins.readthedocs.io/en/latest/>
- <https://docs.openstack.org/infra/jenkins-job-builder/>