

# TP Jenkins

## Objectif

L'objectif de ce TP est d'installer le serveur d'intégration continue Jenkins et de tester l'intégration d'un ensemble d'outils de l'intégration continue dans des jobs dédiés.

## Récupération du code des exemples de cette séance

L'ensemble des projets utilisés dans cette séance est disponible dans le dépôt de code suivant :

[https://git.sr.ht/~toma/jenkins\\_exemples](https://git.sr.ht/~toma/jenkins_exemples)

Ce dépôt contient, pour chaque exemple, un répertoire avec l'ensemble des sources à utiliser.

## Installation de Jenkins

Nous allons utiliser le packaging de Jenkins s'exécutant dans son propre serveur applicatif. Ce packaging n'est pas recommandé pour une véritable mise en production, mais est très utile pour déployer simplement une version de test.

*Jenkins étant basé sur Java, vous devez avoir installé une version récente de la technologie Java. La version courante supporte les versions 17 et 21 de Java.*

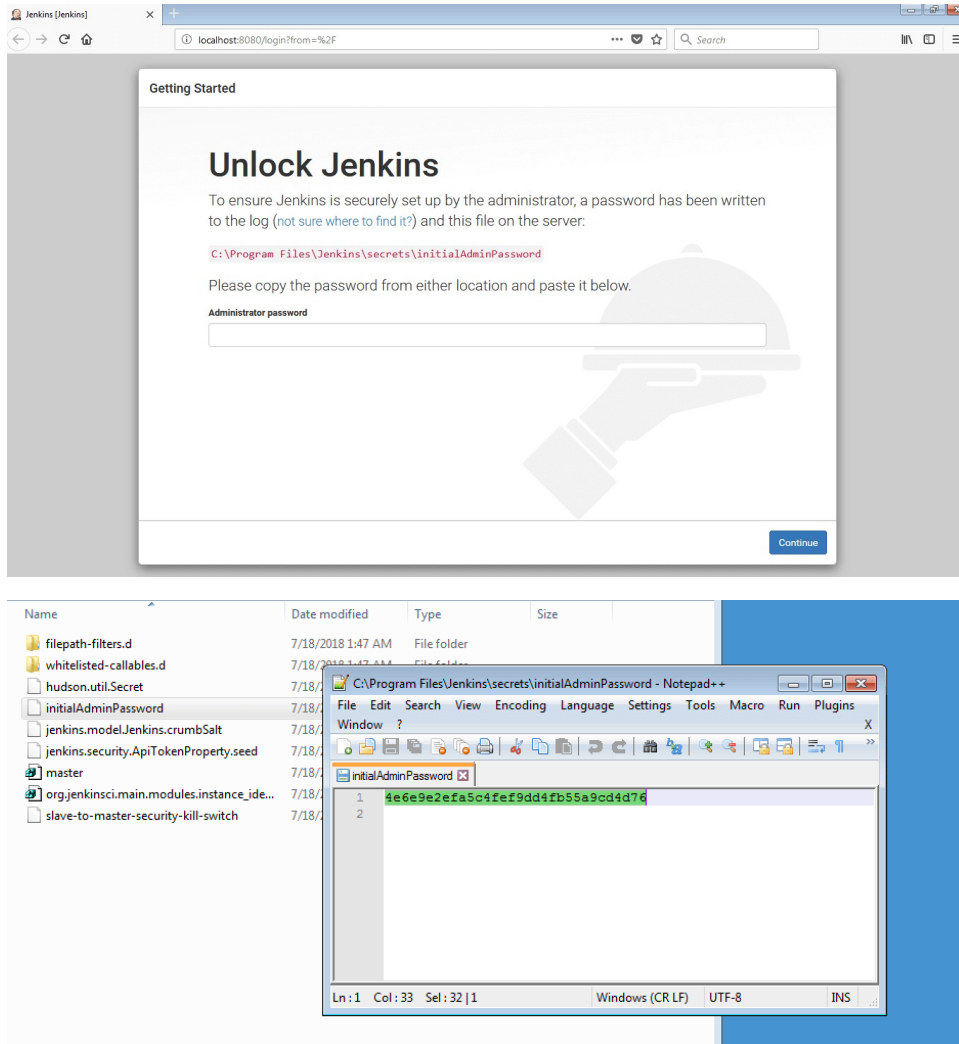
### Installation

1. Récupérer le packaging WAR de la version LTS de Jenkins depuis la page de téléchargement :  
<https://www.jenkins.io/download/>
2. Lancer Jenkins depuis la ligne de commandes :
  - a. Ouvrir un invité de commandes dans le répertoire où a été téléchargé Jenkins
  - b. Depuis ce répertoire, lancer la commande `java -jar jenkins.war` dans le terminal

### Configuration

L'installation ne devrait prendre que quelques secondes. Attendre le message "Jenkins is fully up and running", puis ouvrir le programme en allant à l'URL `localhost:8080` dans votre navigateur par défaut. Cela donne accès à une interface Web, que vous pouvez utiliser pour configurer Jenkins. Mais vous devez d'abord prendre une mesure de sécurité : Jenkins a généré un mot de passe

aléatoire pour vous. Celui-ci se trouve dans le répertoire Jenkins, dans le dossier Secrets et enfin dans le fichier initialAdminPassword. Ce fichier peut être ouvert avec n'importe quel éditeur de texte. Copiez la chaîne de caractères et collez-la dans la zone ad hoc de l'interface Web.



Maintenant, il est temps de configurer Jenkins. L'assistant de configuration vous demandera si vous souhaitez choisir les plugins à installer ou si vous préférez utiliser le paramétrage par défaut intégrant déjà toutes les améliorations majeures. Nous allons choisir les plugins par défaut et installerons les plugins manquants lorsque nécessaire. Patientez pendant l'installation des plugins classiques, avant de passer à la création du compte utilisateur.

# Getting Started

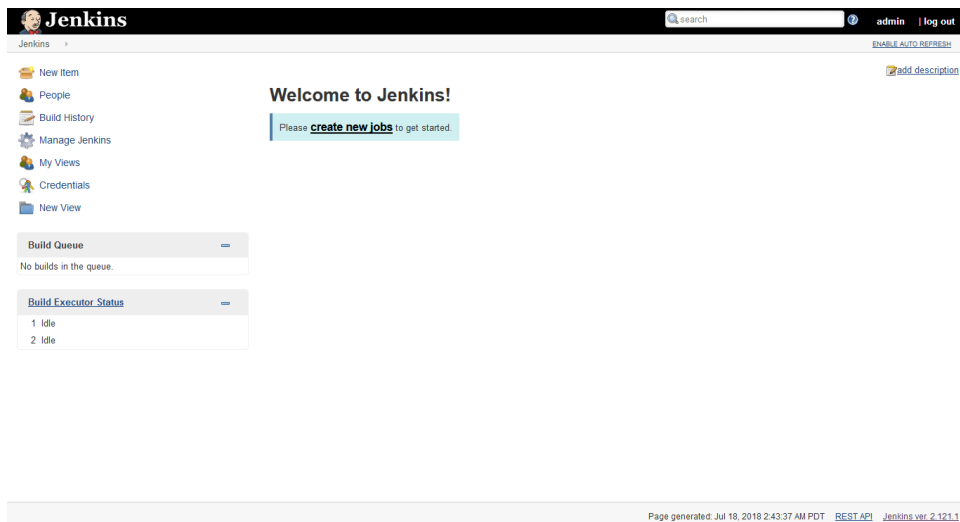
✓ Folders	OWASP Markup Formatter	Build Timeout	Credentials Binding	** JDK Tool ** Script Security ** Command Agent Launcher <b>Folders</b> ** bouncycastle API ** Struts ** Pipeline: Step API ** SCM API
Timestampers	Workspace Cleanup	Ant	Gradle	
Pipeline	GitHub Branch Source	Pipeline: GitHub Groovy Libraries	Pipeline: Stage View	
Git	Subversion	SSH Slaves	Matrix Authorization Strategy	
PAM Authentication	LDAP	Email Extension	Mailer	
				** - required dependency

Jenkins 2.121.1

Remplir la configuration du compte utilisateur que vous souhaitez créer et valider. Jenkins demande de valider l'URL de son serveur : garder la configuration par défaut (<http://localhost:8080>) et valider pour atteindre la page d'accueil du service.

## Exemple 1. Premiers pas dans Jenkins

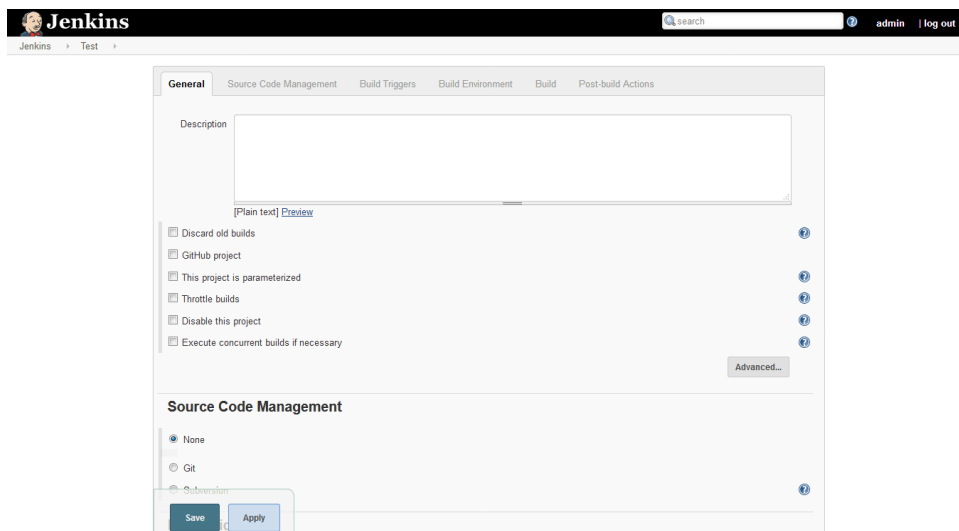
Vous démarrez Jenkins avec un environnement de travail complètement vide. Pour démarrer un nouveau projet d'intégration continue, vous devez créer un nouveau job. Pour ce faire, utilisez l'icône visible au milieu de la fenêtre (« create new jobs ») ou l'option de menu « New Item », que vous trouverez à gauche.



Ensuite, il est nécessaire de donner un nom à votre projet et de choisir ce que vous voulez atteindre :

- Freestyle project : Jenkins associe la gestion des versions à un système de build.
- Pipeline : créez un pipeline entre plusieurs agents de build.
- Projet multi-configuration : Choisissez cette option si vous avez un projet qui nécessite une variété de paramètres, par exemple parce que vous utilisez plusieurs environnements de test.
- Folder : un dossier est un conteneur dans lequel vous pouvez stocker des objets imbriqués.
- GitHub Organization : cette option fait une recherche dans tous les référentiels d'un compte sur GitHub.
- Multibranch Pipeline : vous permet de créer directement plusieurs pipelines.

Jenkins se concentre principalement sur les projets freestyle, c'est pourquoi nous utiliserons ce type de projet pour commencer.



La page suivante propose de nombreuses options de paramétrage regroupées en six onglets.

Commençons par la **gestion du code source**. La plupart du temps, nous utiliserons Git comme référence pour récupérer le code source (il est également possible de connecter directement Jenkins à GitHub, par exemple). Jenkins supporte également de nombreux plugins pour gérer des dépôt Subversion, Mercurial, ... Pour ce premier projet, nous allons copier les sources à la main : choisir “None”.

À l'étape suivante, sélectionnez le **Build Trigger**. Cela déterminera dans quelles situations Jenkins devra effectuer un build. Ceci peut par exemple se faire par l'intermédiaire d'un script spécial d'un autre programme, à la suite d'un autre build, ou périodiquement à des intervalles de temps spécifiques. Vous pouvez également ne rien sélectionner et déclencher un build manuellement lorsque vous le souhaitez - ce que nous ferons dans cet exemple.

**L'environnement de build**, que Jenkins vous laissera configurer par la suite, comprend d'autres options liées aux builds : une interruption, par exemple, si le processus se bloque ? Ou l'affichage d'un horodatage (timestamp) dans la console ? Aucune option n'est obligatoire.

Enfin, il s'agit du **Build** lui-même : ici, vous déterminez comment votre programme doit être construit. Comme vous avez déjà intégré les connexions à Maven et Gradle via la sélection standard de plugins, vous pouvez choisir l'un de ces programmes. Il est toutefois également possible d'utiliser des commandes simples de type ligne de commande. Sélectionnez l'option pour les commandes par batch pour ce premier exemple et entrez le script suivant :

```
del *.class
javac *.java
java -cp . App "2" "3"
```

Après cela, Jenkins vous donne encore la possibilité d'exécuter des **actions postérieures au build**. Par ailleurs : Avec des plugins supplémentaires, vous pouvez également connecter Jenkins à d'autres environnements de test. Vous pouvez même ainsi laisser Jenkins exécuter des tests automatiques. Vous pouvez également vous informer et informer les autres sur le statut du build par courrier électronique. Nous n'utiliserons pas ces options dans ce premier exemple : finir la configuration en sauvegardant les informations et créez votre premier job.


Chaque projet a sa propre sous-page dans Jenkins. Ici, vous pouvez déclencher un build, modifier à nouveau les paramètres et consulter le statut. Sur la page « Statut », Jenkins vous indique également **l'historique des builds**. Le dernier build a-t-il réussi ou non ? Jenkins indique par des points bleus que le build a réussi. Le rouge représente une erreur qui doit être résolue immédiatement, conformément aux principes de l'intégration continue. À cet endroit, vous verrez également quand un build est en cours. En cliquant sur le numéro de build, vous accédez à une page de détails où vous pouvez également afficher le contenu de la console.

**Lancez un premier build** du job que nous venons de créer dans Jenkins. Ce build va échouer car nous n'avons pas encore copié les sources du projet, mais il aura pour effet de créer le répertoire de travail pour le job.

Nous allons maintenant **copier les sources du 1er exemple** : copiez le contenu du répertoire "exemple1" (sans le répertoire lui-même) dans le répertoire de travail : `.jenkins/workspace/{NomDuJobHudson}`


*Le répertoire .jenkins peut varier d'un environnement à l'autre : c'est le même que celui utilisé lors de la configuration pour récupérer le fichier initialAdminPassword*


**Relancez le build** : celui-ci doit maintenant être valide. Vérifiez dans la console que la sortie du programme affiche bien : `The result is 5`


 Retour au projet

 État

 Modifications

 Console Output

 View as plain text

 Informations de la construction

 Delete build '#6'

 Build précédent

## Sortie de la console

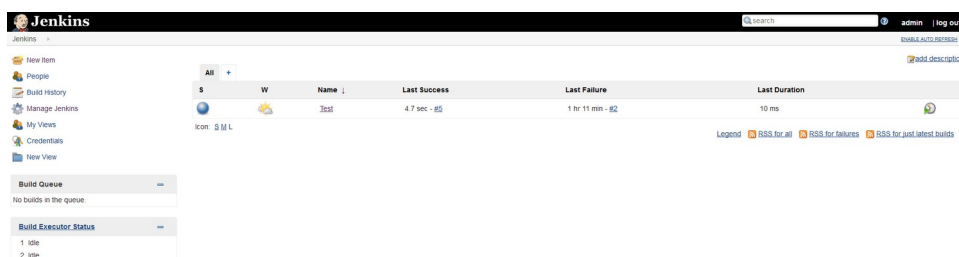
Started by user [Tom A.](#)  
Running as SYSTEM  
Building in workspace /home/tavenel/.jenkins/workspace/projet1  
[projet1] \$ /bin/sh -xe /tmp/jenkins2880260359174772716.sh  
+ cd exemple1  
+ rm App.class CalculatorAlgo.class CalculatorService.class  
+ javac App.java CalculatorAlgo.java CalculatorService.java  
+ java -cp . App 2 3  
The result is : 5  
Finished: SUCCESS

## Exemple 2. Utilisation d'un outil de build (Maven)

Le compilateur javac fourni par java étant extrêmement limité, les projets Java utilisent généralement des outils de build leur permettant de gérer les dépendances. Dans ce deuxième exemple, nous allons utiliser l'outil de build Maven pour compiler et packager notre application. Maven permet de gérer grâce à un fichier `pom.xml` les dépendances d'un projet, la compilation du code et l'exécution par plugins d'un ensemble d'outils : exécution des tests unitaires, ... Maven fonctionne sous forme de cibles : la cible exécute l'outil approprié et exécute si besoin les autres cibles dépendantes (par exemple : compiler le code avant d'exécuter les tests).

*Jenkins est avant tout un orchestrateur capable de lancer différents outils par le biais de plugins, mais il ne réalise pas l'installation des outils eux-mêmes. Si ce n'est pas déjà le cas, assurez-vous d'avoir bien installé Maven sur votre machine avant de poursuivre.*

Dans le tableau de bord de Jenkins, vous pouvez voir tous les projets sur lesquels vous travaillez. Ici aussi, le programme matérialise l'état du projet par une couleur. Vous obtenez également des informations sur la **Stabilité du build** sous la forme d'un bulletin météo. Il s'agit d'une statistique sur la stabilité moyenne des builds du projet. Si plus de 80 % de vos builds réussissent, vous verrez un soleil. En dessous de cette valeur, la météo symbolique se dégrade.



**Créez un nouveau job** de manière similaire au 1er exemple, mais au lieu d'exécuter un script choisissez, dans l'onglet **Build**, d'invoquer des cibles Maven.

compile

```
exec:java -Dexec.mainClass="epsi.App" -Dexec.args="2 3"
```



**Copiez le contenu de l'exemple 2** dans le nouveau job. Attention à bien copier le contenu du répertoire exemple2 directement dans le nouveau job (et pas dans un sous-répertoire exemple2), sinon jenkins ne trouvera pas le fichier pom.xml de Maven.

**Lancez le build** et vérifiez que le programme affiche bien la même valeur que dans le 1er exemple.

```
Building in workspace C:\Users\Presenter\.jenkins\workspace\exemple2
[exemple2] $ cmd.exe /C "mvn compile exec:java -Dexec.mainClass=epsi.App "-Dexec.args=2 3" && exit %%ERRORLEVEL%%"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< epsi:addition >-----
[INFO] Building addition 1.0
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ addition ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Users\Presenter\.jenkins\workspace\exemple2\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ addition ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ addition ---
The result is : 5
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  2.219 s
[INFO] Finished at: 2020-11-26T18:39:52+01:00
[INFO] -----
Finished: SUCCESS
```

## Exemple 3. Utilisation d'un gestionnaire de versions (Git)

Dans cet exemple, nous allons utiliser le gestionnaire de versions Git pour récupérer et valider automatiquement les changements dans le code de notre projet.

### 1. Créez un nouveau dépôt de code git

- a. Ce dépôt peut être créé localement sur la machine de travail, ou en utilisant un hébergement en ligne de type Github, Bitbucket, Gitlab, ...
- b. Copiez les sources données dans cet exemple dans le nouveau dépôt git

### 2. Créez un nouveau job dans Jenkins en utilisant, et configurez l'accès au dépôt git :

- a. Dans la section **Gestion du code source**, sélectionner git et renseignez l'accès à votre dépôt

The screenshot shows the Jenkins 'Repositories' configuration page. It includes a 'Repository URL' field with the value 'https://avenelt@bitbucket.org/avenelt/jenkins\_epsi.git', a 'Credentials' dropdown menu showing 'avenelt/\*\*\*\*\*' and an 'Ajouter' button, and a 'Branches to build' section with a 'Branch Specifier (blank for \'any\')' field containing '\*/master'. At the bottom, there are three buttons: 'Sauver', 'Apply', and 'Add Branch'.

- b. Dans la section Ce qui déclenche le build, sélectionnez Scrutation de l'outil de gestion de version. Cela permet d'analyser le dépôt git suivant une horloge configurée pour chercher d'éventuels changements à analyser, et de lancer un nouveau build dans le cas où des changements seraient détectés.

On pourra per exemple utiliser le planning H/5 \* \* \* \* pour vérifier toutes les 5 minutes l'arrivée de changements.

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

☐ GitHub hook trigger for GITScm polling

☒ Scrutation de l'outil de gestion de version

?

?

?

?

?

Planning

H/5 \* \* \* \*

3. Dans la section **Build**, ajouter une cible Maven **compile** afin de compiler les sources récupérées.

## Exemple 4. Intégration des tests dans Jenkins

Dans ce nouvel exemple, nous allons utiliser les cibles Maven orientées tests pour exécuter les tests unitaires présents dans le répertoire et afficher leurs résultats.

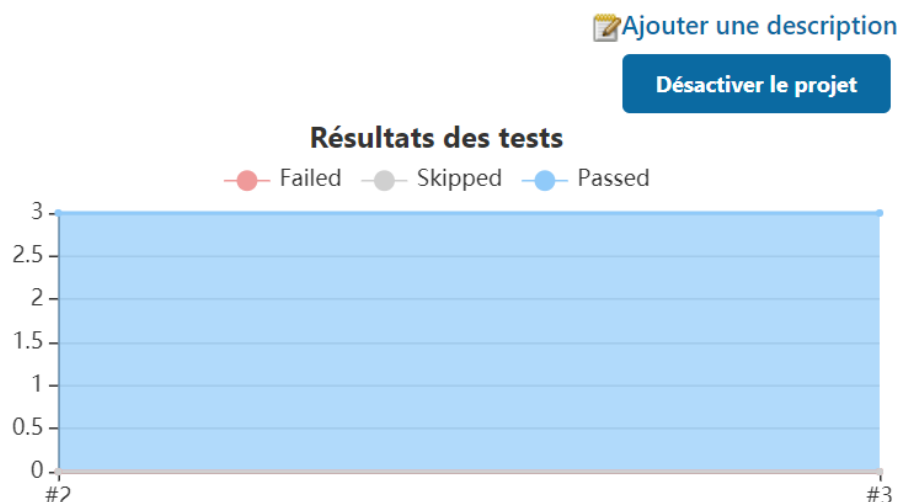
**Créez un nouveau job** en utilisant les sources de cet exemple, et créez :

- Une étape de **Build** invoquant des cibles Maven : `verify`
- Dans la section **Actions à la suite du build**, ajouter une nouvelle étape **Publier le rapport des résultats des tests JUnit**



The screenshot shows the configuration for the 'Publier le rapport des résultats des tests JUnit' step. The 'XML des rapports de test' field is set to 'target/surefire-reports/TEST-\*.xml'. A descriptive text explains the 'Fileset' configuration. The 'Retain long standard output/error' checkbox is unchecked. The 'Health report amplification factor' is set to '1,0'. The 'Allow empty results' checkbox is checked.

Field	Value
XML des rapports de test	target/surefire-reports/TEST-*.xml
Retain long standard output/error	<input type="checkbox"/>
Health report amplification factor	1,0
Allow empty results	<input checked="" type="checkbox"/>

Lancer un build et vérifier que les résultats de tests sont affichés dans le sommaire du projet Jenkins :



*Notez au passage que l'utilisation de cibles Maven nous a permis d'exécuter automatiquement les tests unitaires à chaque build, sans aucune configuration supplémentaire.*

 Informations de la construction Delete build '#4' Build précédent

```
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ addition ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/tavenel/.jenkins/workspace/projet2/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ addition ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.0.2:testResources (default-testResources) @ addition ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/tavenel/.jenkins/workspace/projet2/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ addition ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ addition ---
[INFO]
[INFO] T E S T S
[INFO]
[INFO] Running epsi.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in epsi.AppTest
[INFO] Running epsi.CalculatorServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in epsi.CalculatorServiceTest
[INFO] Running epsi.CalculatorAlgoTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in epsi.CalculatorAlgoTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ addition ---
[INFO]
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ addition ---
[INFO] The result is : 5
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 2.019 s
[INFO] Finished at: 2020-11-24T17:13:57+01:00
[INFO]
Finished: SUCCESS
```

## Exemple 5. Développement piloté par les tests

Nous allons maintenant réaliser le développement d'une nouvelle fonctionnalité en pilotant ce développement par les tests. Cette méthode de développement permet une augmentation significative de la qualité et de la rapidité de développement.

1. Créez un nouveau job dans Jenkins en utilisant le code source de cet exemple. Configurer le gestionnaire de versions git et l'exécution et les rapports de tests dans Jenkins, de manière similaire aux exemples précédents.
2. Ajoutez au programme fourni le squelette d'une nouvelle fonctionnalité de multiplication sans en coder l'implémentation (on pourra par exemple limiter l'implémentation à "return 0").
3. Ajoutez les tests correspondants en s'inspirant des tests de l'addition, et vérifiez que les tests sont lancés automatiquement dans Jenkins mais échouent puisque l'implémentation n'est pas encore écrite.
4. Ajoutez l'implémentation de la fonctionnalité de multiplication et vérifiez que les tests ne retournent plus d'erreur.

## Exemple 6. Intégration avec Gradle

Dans cet exemple, nous allons utiliser l'outil de build Gradle pour construire et tester un projet. Gradle est très similaire à Maven (et utilise le même système de dépendances) mais utilise un fichier `build.gradle` de configuration du build au format Groovy ou Kotlin.

1. Créez un nouveau job dans Jenkins en utilisant le code source de cet exemple.
2. Ajoutez une étape de **Build** en choisissant **Invoke Gradle Script**.
  - a. Sélectionner **Use Gradle Wrapper** pour utiliser le programme gradle fourni dans le projet.
  - b. Dans le champ **Tasks**, choisissez d'exécuter la tâche **clean test**.
3. Ajoutez une **Action à la suite du Build** en choisissant de **Publier le rapport des résultats des tests JUnit**.
  - a. Spécifier le chemin vers les rapports de test générés par Gradle : **build/test-results/test/TEST-\*.xml**
4. Lancez un nouveau build. Vérifier que les résultats de tests sont reportés dans la page du projet.
5. Afficher la sortie de la console enregistrés durant le build. Vérifier que les différentes tâches Gradle ont bien été extraites dans l'arbre de gauche.



## Exemple 7. Mise en place de jobs chaînés

Dans cet exemple, nous allons voir un aperçu d'une des fonctionnalités les plus puissantes de Jenkins : celle de créer des liens logiques de dépendances entre les jobs.

1. Créez un premier job dans Jenkins (on gardera la configuration par défaut pour créer un job vide). Ce job sera exécuté automatiquement après un job gérant ses pré-requis.
2. Créez un second job. Ce job sera le pré-requis à l'exécution du job précédent :
  - a. Ajouter une **Action à la suite du Build** pour **Construire d'autres projets en aval**.
  - b. Renseigner le projet précédent dans les **Projets à construire**
  - c. Sélectionner l'exécution du premier job seulement si la construction est stable.
3. Lancer un nouveau build du second job. Vérifier qu'à la fin de ce build, une exécution du premier job est lancée automatiquement.

DANS LA PRATIQUE, LES DÉPENDANCES ENTRE JOBS SONT UTILISÉES :

- POUR APPELER DES JOBS DE PUBLICATION DE RÉSULTATS OU TOUT AUTRE PROCESS QUI PEUVENT ÊTRE COMMUNS À UN ENSEMBLE DE BUILDS D'INTÉGRATION CONTINUE
- POUR GÉRER DES OPÉRATIONS DE NETTOYAGE DU SYSTÈME EN CAS D'ÉCHEC LORS D'UN BUILD
- POUR SÉPARER LES DIFFÉRENTES ÉTAPES D'UN PROCESS D'INTÉGRATION CONTINUE : COMPILATION DES SOURCES, EXÉCUTION DES TESTS, ANALYSE STATIQUE, ...

## Exemple 8. Les plugins “Warnings Next Generation” et “Git Forensics”

Dans cet exemple, nous allons installer et utiliser deux plugins dans Jenkins :

Le plugin “Warnings Next Generation” qui permet de répertorier les erreurs et avertissements renvoyés par différents outils de compilation et d’analyse statique

Le plugin “Git Forensics” qui permet d’identifier le commit et le développeur ayant créé les modifications à la base des erreurs reportées

### Installation d’un plugin dans Jenkins :

*Dans le tableau de bord Jenkins, choisir “Administrer Jenkins” puis “Gestion des plugins”. Cliquer sur l’onglet “Disponibles” et rechercher le plugin à installer. Jenkins supporte mal l’installation à chaud de plugins : choisir l’option “Télécharger maintenant et installer après redémarrage”*

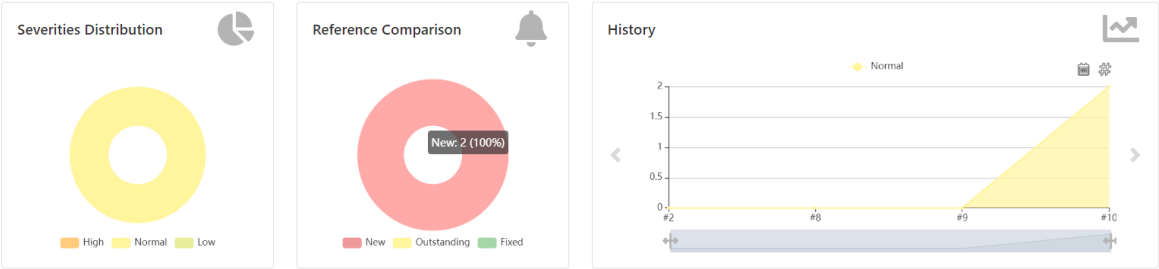
*Redémarrer Jenkins pour prendre en compte les changements.*

*Note : il est recommandé d’installer les plugins l’un après l’autre et de redémarrer Jenkins entre chaque installation, pour éviter des problèmes d’interdépendances.*

1. Installez les plugins “Warnings Next Generation” et “Git Forensics”.
2. Créez un nouveau job dans Jenkins, en utilisant le dépôt git des exemples fournis.
  - a. Configurer l’étape de **Build** avec une unique étape exécutant la compilation du programme en affichant tous les avertissements :

```
javac -Xlint:all App.java
```
  - b. Ajouter une **Action à la suite du Build** de type **Record compiler warnings and static analysis results**. Sélectionner l’outil **Java**.
3. Lancez un nouveau build. Vérifiez que les avertissements du compilateur sont bien affichés et que le commit contenant les changements incriminés est bien reporté.

New Warnings



Details

Categories Issues

Show 10 entries Search:

Category	Total	Distribution
Rawtypes	1	
Unchecked	1	
Total	2	

Showing 1 to 2 of 2 entries

Details

Categories Issues SCM Blames SCM Forensics

Show 10 entries Search:

Details	File	Age	Author	Email	Commit	Add
+	App.java:9	1	Tom Avenel	tomavenel@gmail.com	e217c23a9796e3b55303a747d6959beece6b1521	1 da
+	App.java:10	3	Tom Avenel	tomavenel@gmail.com	e217c23a9796e3b55303a747d6959beece6b1521	1 da

Showing 1 to 2 of 2 entries

## Exemple 9. Création de pipelines

Dans cet exemple, nous allons utiliser la fonctionnalité de pipeline de Jenkins. Les pipelines offrent la possibilité d'ordonnancer simplement des jobs complexes, permettant la construction de projets aux dépendances multiples.

Créez un nouveau Job dans Jenkins, mais cette fois choisissez le type de job Pipeline.

Dans la section 'Build Triggers', cocher 'Poll SCM' pour scruter régulièrement le gestionnaire de versions (on pourra utiliser ``* * *`` en test pour une vérification toutes les minutes).

- Note: lorsque c'est possible, il est préférable d'utiliser un `_webhook_` pour pusher les changements depuis l'hébergeur Git (``Github``, ``Gitlab cloud``, ``Bitbucket``, ...) et déclencher le build. Cependant ``Jenkins`` n'est pas toujours accessible publiquement, il faut alors utiliser un pull et du polling pour scruter périodiquement si de nouveaux commits sont arrivés.

Sélectionner le template "Pipeline Script" et, en utilisant les exemples fournis par Jenkins générer un pipeline en cinq étapes :

- Une étape de préparation récupérant les sources du projet sur un dépôt distant (Bitbucket, Github, Gitlab, ...). On pourra s'inspirer de la configuration de l'exemple 3 (Utilisation d'un gestionnaire de versions Git)
- Une étape compilant les sources du projet (on utilisera la cible Maven compile)
- Une étape exécutant les tests (`mvn test`)
- Une étape réalisant le packaging de l'application (`mvn package -DskipTests`)
- Une étape agrégeant les résultats de tests et archivant le jar créé à l'étape précédente

Pour s'aider à l'écriture du script, on pourra utiliser le bouton ``Pipeline Syntax`` qui permet de générer le script du pipeline depuis les interfaces graphiques des différentes étapes.

Exemple de pipeline :

```

```jenkins
pipeline {
    // Utiliser n'importe quel agent (worker) disponible
    agent any

    stages {
        // Liste des étapes

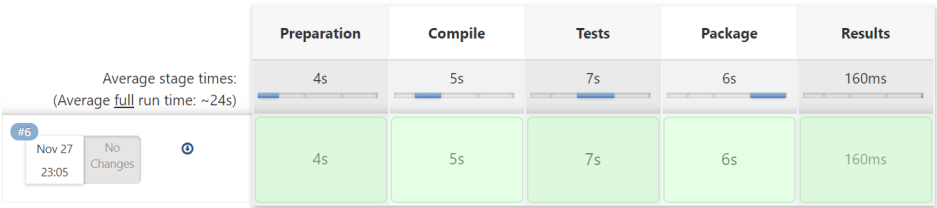
        // Découpage logique en `stage {}` (correspond aux
        grandes étapes du pipeline)
        stage('Checkout') {
            steps {
                // Liste des étapes - une étape par ligne
                git 'https://...'
            }
        }

        stage('Build') {
            steps {
                sh mon_script.sh
            }
            post {
                success {
                    archiveArtifacts 'mon_build.zip'
                }
            }
        }
    }
}
```

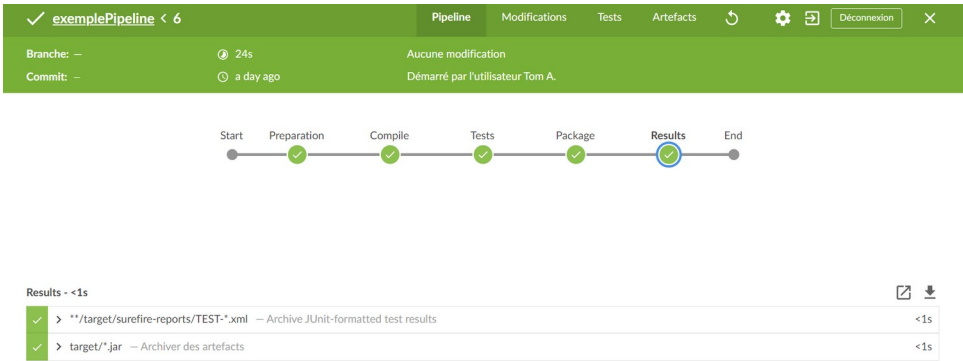
```

Lancez un nouveau build et vérifiez le bon déroulement du pipeline.

# Stage View



LES PIPELINES SONT UNE FONCTIONNALITÉ PUISSANTE DE JENKINS, MAIS LEUR UTILISATION DANS L'INTERFACE CLASSIQUE N'EST PAS TRÈS ADAPTÉE. ON POURRA UTILISER LES PLUGINS BLUE OCEAN QUI FOURNISSENT UNE INTERFACE PLUS OPTIMISÉE POUR LES PIPELINES.



## Exemple 10. Réutilisation d'artefacts

Dans cet exemple, nous verrons comment réutiliser dans un job des objets déjà créés dans un build précédent.

1. Installez le plugin "Copy Artifact". Ce plugin va nous permettre de transférer un objet créé pendant un build dans un second build.
2. Créez un premier job artefactA utilisant les sources de l'exemple avec les configurations suivantes :
  - a. Le **Build** exécute une unique tâche Gradle : **build**
  - b. Accorder la permission d'accéder aux artefacts à un futur job artefactB
  - c. A la suite du build, archiver l'artefact :  
`build/distributions/*.zip`
3. Créez un second job artefactB avec les configurations suivantes :
  - a. Construire après le build d'artefactA
  - b. Dans la section Build :
    - i. La première tâche sera de type **Copy artifacts from another project**. Choisir le projet artefactA, le build **Upstream build that triggered this job** (afin de récupérer l'artefact du build précédent), et cocher **Flatten directories**
    - ii. La deuxième tâche sera une commande batch permettant de dézipper l'artefact :  
  
**`tar.exe -x -f .\artefactA.zip`**
    - iii. La dernière tâche sera l'exécution du programme dans une commande batch :  
  
**`.\artefactA\artefactA\bin\artefactA 3 5`**
4. Lancez un build du projet artefactA. Vérifiez que le build d'artefactB est lancé et que le résultat de l'addition est affiché :

## Sortie de la console

```
Démarré par l'utilisateur Tom A. ▼
Running as SYSTEM
Building in workspace C:\Users\Presenter\.jenkins\workspace\artefactB
Copied 1 artifact from "artefactA" build number 4
[artefactB] $ cmd /c call C:\Users\PRESEN~1\AppData\Local\Temp\jenkins9751310508497176738.bat

C:\Users\Presenter\.jenkins\workspace\artefactB>tar.exe -x -f .\artefactA.zip

C:\Users\Presenter\.jenkins\workspace\artefactB>exit 0
[artefactB] $ cmd /c call C:\Users\PRESEN~1\AppData\Local\Temp\jenkins15295110364489220591.bat

C:\Users\Presenter\.jenkins\workspace\artefactB>.\artefactA\artefactA\bin\artefactA 3 5
The result is : 8
Notifying upstream projects of job completion
Finished: SUCCESS
```

EN PRATIQUE, CETTE RÉUTILISATION D'ARTÉFACTS EST TRÈS UTILE DANS DES PROCESS D'INTÉGRATION CONTINUE DE PROJETS CONTENANT DE NOMBREUX COMPOSANTS N'AYANT PAS TOUS LE MÊME CYCLE DE VIE (CE QUI EST TYPIQUE D'UNE ARCHITECTURE MICROSERVICES). DANS CE CAS, ON SÉPARE AU MAXIMUM LES FONCTIONNALITÉS DU PROJET DANS DES JOBS JENKINS DÉDIÉS. CELA PERMET DE NE PAS AVOIR À TESTER DE NOUVEAU LE RESTE DU PROJET EN CAS DE CHANGEMENT DANS UNE FONCTIONNALITÉ ISOLÉE (ON UTILISERA UNIQUEMENT LES OBJETS DÉJÀ TESTÉS). PAR EXEMPLE : SI LE SERVICE DE LOGIN N'A PAS ÉVOLUÉ, ON POURRA DIRECTEMENT INTÉGRER LE DERNIER BUILD DE CE SERVICE À LA CONSTRUCTION DU PROJET.



## Exemple 11. Administration de Jenkins

### Sauvegarde et restauration

#### Sauvegarde manuelle

Jenkins enregistre toutes ses données dans le répertoire JENKINS\_HOME. La forme de sauvegarde la plus simple consiste donc à sauvegarder l'ensemble de ce répertoire. Celui-ci pouvant cependant être très volumineux, on pourra choisir d'omettre certains sous-répertoires : historique des jobs, artéfacts, binaires des plugins, ...

De même qu'il est sage de réaliser des sauvegardes fréquentes, il est également sage de tester régulièrement cette procédure. Jenkins permet cela très facilement : il suffit de changer la variable JENKINS\_HOME pour pointer vers le répertoire de sauvegarde :

```
$export JENKINS_HOME=/tmp/jenkins-backup
```

```
$ java -jar jenkins.war --httpPort=8888
```

1. Réalisez une sauvegarde manuelle du répertoire JENKINS\_HOME (par défaut : .jenkins dans le répertoire utilisateur)
2. Testez la sauvegarde en démarrant une nouvelle instance de Jenkins utilisant cette sauvegarde

#### Le plugin Backup

Une solution de facilité pour gérer les sauvegardes est de déléguer ce procédé à un plugin dédié.

1. Installez le plugin Backup
2. Configurez la sauvegarde depuis la page d'administration de Jenkins
3. Testez la restauration depuis la sauvegarde effectuée

### Gestion de la sécurité

1. Créez un nouvel utilisateur depuis la page d'administration de Jenkins. Cet utilisateur sera dédié à l'exécution d'un job.
2. Ouvrez les options de sécurité depuis la page d'administration.
  - a. Jenkins permet de déléguer la gestion des comptes utilisateurs à des services de login dédiés (LDAP, ...). C'est généralement cette option qui est utilisée, mais pour éviter le déploiement d'un annuaire LDAP nous continuerons d'utiliser des comptes locaux dans nos exemples.

- b. Changer le mode d'autorisations par défaut ("Les utilisateurs connectés peuvent tout faire") pour **Stratégie d'autorisation matricielle basée sur les projets**. Cette stratégie permet une gestion fine des autorisations, en ajoutant de nouvelles options dans chaque job définissant quel utilisateur peut accéder à ce job.
  - i. *On pensera à ajouter les droits d'administration à l'utilisateur courant pour pouvoir continuer à configurer des jobs dans Jenkins !*
  - ii. Donner le droit **Global:Read** à l'utilisateur créé à la première étape
- c. Ouvrir la configuration du job de l'exemple 1. Sélectionner **Activer la sécurité basée projet** et ajouter l'utilisateur créé à la première étape en lui accordant uniquement les droits **Job:Read** et **Job:Build**
- 3. Connectez-vous avec l'utilisateur créé à la première étape. Vérifiez que cet utilisateur a uniquement accès au job précédent et que cet utilisateur peut lancer un nouveau build.

## Exécution des jobs dans des nœuds distants

1. Déployez une machine virtuelle Debian qui fournira un environnement d'exécution Linux :
  - a. Installer VirtualBox  
<https://www.virtualbox.org/wiki/Downloads>
  - b. Créer une machine virtuelle de type Linux (debian, ubuntu)
  - c. Dans les paramètres réseau de la machine virtuelle, choisir de configurer la carte réseau en 'Pont' (et non en NAT par défaut) pour que la machine hôte et la machine virtuelle puissent communiquer entre eux
  - d. Démarrer la machine virtuelle
  - e. Installer Java dans la machine virtuelle (requis pour installer un agent Jenkins)
2. Depuis la page d'administration de Jenkins, cliquez sur **Créer un nœud**
  - a. Utiliser **/home/osboxes/jenkins** comme répertoire de travail distant
  - b. Choisir **Launch agents via SSH comme** méthode de lancement
    - i. Entrer l'adresse IP de la machine virtuelle
    - ii. Entrer le nom d'utilisateur / mot de passe de la machine virtuelle

R  pertoire de travail du syst  me distant ?

/home/osboxes/jenkins

  tiquettes ?

Utilisation ?

Utiliser ce noeud autant que possible

M  thode de lancement ?

Launch agents via SSH

Host ?

192.168.1.25

Credentials ?

osboxes/\*\*\*\*\*\* Ajouter

Host Key Verification Strategy ?

Known hosts file Verification Strategy

- c. Valider, s  lectionner l'agent et afficher les logs pour v  rifier que l'installation se d  roule correctement
3. Cr  ez un nouveau job    ex  cuter sur ce nouvel agent :
  - a. Cr  er un nouveau job dans Jenkins
  - b. Choisir **Restreindre o   le projet peut   tre ex  cut  ** et entrer le nom du nouvel agent
  - c. Ajouter une   tape de **Build** de type **script shell** :  
**uname -a**
4. Lancez un nouveau build : v  rifier que le build tourne bien sur le nouvel agent

## Sortie de la console

D  marr   par l'utilisateur **Tom A.**

Running as SYSTEM

Construction    distance sur **debianAgent** in workspace /home/osboxes/jenkins/workspace/exempleUnix

[exempleUnix] \$ /bin/sh -xe /tmp/jenkins1169607429784634306.sh

+ uname -a

Linux osboxes 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5 (2019-06-19) x86\_64 GNU/Linux

Notifying upstream projects of job completion

Finished: SUCCESS

## Exemple 12. Intégration avec SonarQube

Dans cet exemple, nous allons installer un serveur d'analyse SonarQube. Nous intégrerons ensuite les rapports de SonarQube dans Jenkins pour afficher une vue centralisée du projet.

### 1. Installer le serveur SonarQube

- a. Télécharger le serveur SonarQube (choisir la version community) :  
<https://www.sonarsource.com/products/sonarqube/downloads/>
- b. Démarrer le serveur SonarQube, par exemple : C:\Program Files\sonarqube-9.6.1\sonarqube-9.6.1\bin\windows-x86-64\StartSonar.bat
- c. Vérifier le bon démarrage du serveur :  
<http://localhost:9000> (admin/admin)

### 2. Installer et configurer le scanner SonarQube pour Jenkins

- a. Installer le plugin **"SonarQube Scanner"**
- b. Configurer le scanner :

Dans le panel "Administrer Jenkins", choisir "Configuration Globale des Outils". Dans la section "SonarQube Scanner", cliquer sur le bouton **"Ajouter SonarQube Scanner"**. Vérifier que l'installation automatique depuis le Maven Central est bien sélectionnée :

Ajouter SonarQube Scanner

SonarQube Scanner

Name

Jenkins Sonar

☒ Install automatically

Installer depuis Maven Central

Version

SonarQube Scanner 4.5.0.2216

Ajouter un installateur

Supprimer un installateur

Supprimer SonarQube Scanner

Ajouter SonarQube Scanner

Liste des installations SonarQube Scanner sur ce système

Enregistrer Appliquer

### 3. Configurer le serveur SonarQube dans Jenkins

Dans le panel “Administrer Jenkins”, choisir “Configurer le système”. Remplir la section “SonarQube servers” (spécifier notamment **l’injection de variables SonarQube dans le build**) :

The screenshot shows the 'SonarQube servers' configuration page in Jenkins. It includes a checkbox for 'Environment variables' which is checked, with a note that it enables injection of SonarQube server configuration as build environment variables. Below this, there's a section for 'Installations de SonarQube' with fields for 'Nom' (SonarQubeServer), 'URL du serveur' (http://localhost:9000), and 'Server authentication token' (set to '- aucun -'). A button 'Ajouter' is also visible.

4. **Configurer SonarQube dans le job Jenkins.** Créer un nouveau job dans Jenkins utilisant les sources de l'exemple SonarQube, avec la configuration suivante :
  - a. Dans la section “Environnements de Build”, cocher l'option **“préparer l'environnement pour SonarQube Scanner”**
  - b. Dans la section “Build” :
    - i. Ajouter une étape de build Maven lançant la cible **verify**
    - ii. Ajouter une étape de build Maven lançant la cible **sonar:sonar**
5. **Lancer le build**, vérifier que celui-ci n'échoue pas et que la page du projet dans Jenkins affiche un succès d'analyse dans Sonar. Afficher les résultats de l'analyse dans Sonar.

## SonarQube Quality Gate

jenkinsSonarExample **Passed**  
server-side processing: **Success**

The screenshot shows the SonarQube Quality Gate dashboard for the project 'jenkinsSonarExample'. It displays various metrics with their status: Bugs (A), Vulnerabilities (A), Hotspots Reviewed (A), Code Smells (A), Coverage (40.0%), Duplications (0.0%), and Lines (133). The overall status is 'Passed'.

|                       |                   |                              |               |          |              |                  |
|-----------------------|-------------------|------------------------------|---------------|----------|--------------|------------------|
| ☆ jenkinsSonarExample | Passed            | Last analysis: 2 minutes ago |               |          |              |                  |
| 🐛 Bugs                | 🔒 Vulnerabilities | 🔍 Hotspots Reviewed          | 👤 Code Smells | Coverage | Duplications | Lines            |
| A                     | A                 | A                            | A             | 40.0%    | 0.0%         | 133 XS XML, Java |

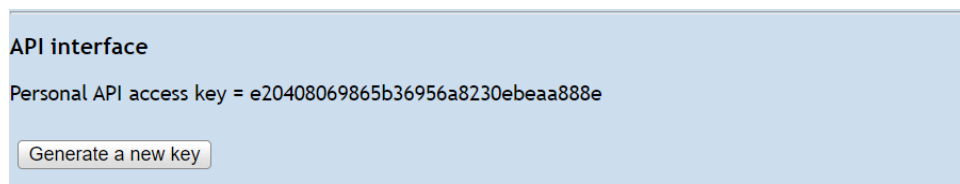
6. **Analyser le fichier pom.xml dans les sources fournies.** Pour réaliser la couverture des tests, le plugin **jacoco** a été ajouté au projet. L'intégration du projet dans Sonar est réalisée par le biais du plugin **sonar** pour Maven.

Pour plus d'information sur l'utilisation des scanners SonarQube,  
on pourra visiter :  
<https://docs.sonarqube.org/latest/analysis/overview/>

## Exemple13. Intégration avec TestLink

Dans cet exemple, nous allons intégrer Jenkins avec l'outil TestLink pour enregistrer les résultats des tests exécutés dans Jenkins à l'intérieur de TestLink.

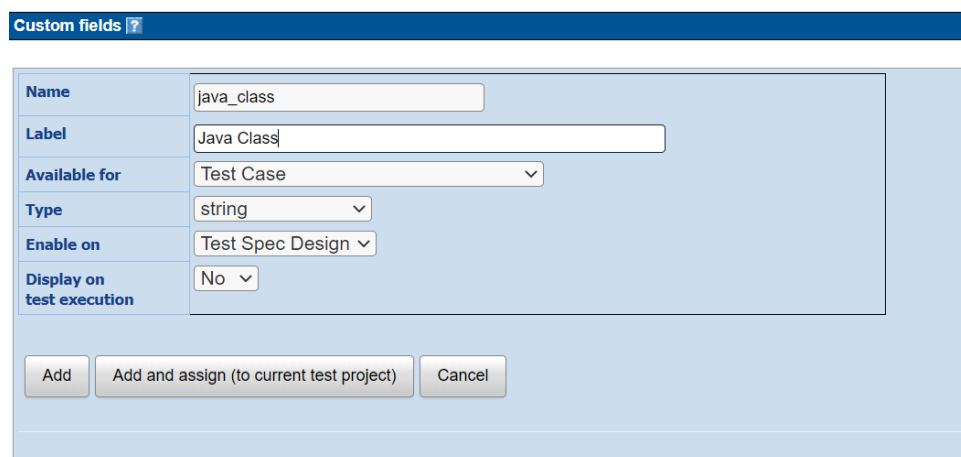
1. **Installer une instance de TestLink.** Testlink est une application PHP - On pourra utiliser ce tutoriel pour l'installer :  
[https://www.tutorialspoint.com/testlink/testlink\\_installation.htm](https://www.tutorialspoint.com/testlink/testlink_installation.htm)
2. **Configurer l'instance de TestLink :**
  - a. Se connecter sur l'instance de TestLink. Créer un nouveau projet (cocher **Enable Test Automation, Active et Public**)
  - b. Créer une clé pour utiliser l'API de TestLink : dans les paramètres de TestLink (bouton MySettings tout en haut), créer une nouvelle clé permettant d'accéder à l'API



API interface

Personal API access key = e20408069865b36956a8230ebeaa888e

- c. Nous allons maintenant définir un champ personnalisé afin de faire le lien entre un test dans TestLink et la classe d'un test unitaire exécuté dans Jenkins. Pour cela :
  - i. Depuis la page principale, cliquer sur "Define Custom Fields"
  - ii. Créer un nouveau champ, choisir un nom (par exemple : java\_class) et un label pour l'affichage, garder les autres paramètres par défaut.



Custom fields ?

|                           |                  |
|---------------------------|------------------|
| Name                      | java_class       |
| Label                     | Java Class       |
| Available for             | Test Case        |
| Type                      | string           |
| Enable on                 | Test Spec Design |
| Display on test execution | No               |

- iii. Choisir "Add and assign (to current test project)"

- d. La prochaine étape est l'ajout d'un scénario de test dans TestLink. Ce scénario sera utilisé pour la mise à jour des résultats d'exécution depuis Jenkins. Pour ajouter ce scénario :
  - i. Dans la page d'accueil, cliquer sur **Test Specification**
  - ii. Nous allons commencer par créer une suite de tests qui contiendra le scénario de test. Dans la page qui s'affiche, cliquer sur la roue pour afficher les options de création, cliquer sur le bouton "+" et choisir un nom pour la nouvelle suite de tests.

- iii. Une fois la nouvelle suite de tests créée, sélectionnez-la dans le menu de gauche
  - iv. Nous allons maintenant créer un scénario de test dans cette suite de tests. De la même manière que précédemment, cliquer sur la roue puis sur le bouton "+" à la suite de **Test Case Operations**.
    1. Choisir un titre pour le scénario de test
    2. Changer le type d'exécution à **Automatique**
    3. Le nouveau champ personnalisé (par exemple : java\_class) ayant été associé au projet, un nouveau champ est disponible à la création du scénario. Remplir ce champ avec le nom de la classe de test que nous utiliserons dans ce projet : **epsi.AppTest**
    4. Valider la création du scénario de test
- e. Dernière étape : TestLink utilise des plans de tests pour décrire et reporter l'exécution de tests.
  - i. Créez un nouveau plan de test dans TestLink (**Test Plan Management / Create**) Sélectionner **Active** et **Public**.
  - ii. Nous allons maintenant ajouter le scénario de test créé précédemment dans le plan de test.
    1. Depuis la page d'accueil, choisir **Add / Remove Test Cases**
    2. Sélectionner le scénario de test créé précédemment
    3. Cliquer sur **Add / Remove selected**



- iii. Il n'est pas nécessaire de créer un nouveau build dans TestLink pour chaque exécution de test : Jenkins s'en chargera pour nous à chaque fois qu'un build sera lancé

### 3. Installer et configurer le plugin TestLink dans Jenkins.

- a. Installer le plugin **TestLink** dans Jenkins.
- b. Configurer le plugin : aller dans la **Configuration du système** de Jenkins et configurer la section TestLink.
  - i. On prendra soin de bien remplir la clé d'accès à l'API de TestLink définie à l'étape précédente

TestLink

TestLink Installation

Name

TestLink

URL ?

http://localhost/testlink/lib/api/xmlrpc/v1/xmlrpc.php

Developer Key

e20408069865b36956a8230ebeeaa888e

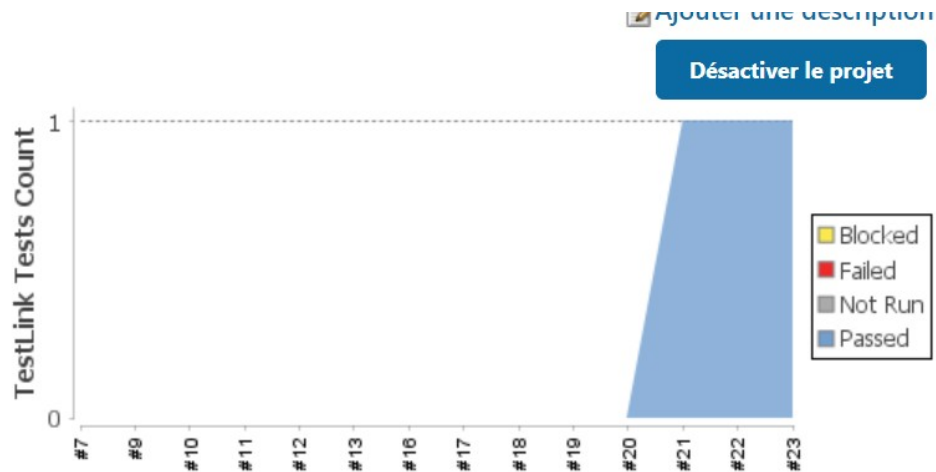
TestLink Java API comma separated properties ?

Supprimer

### 4. Créer un nouveau job dans Jenkins utilisant les sources de l'exemple. Dans la phase de **Build**, on ajoutera un unique step "**Invoke Test Link**" avec la configuration suivante :

- a. Test Project Name : utiliser le nom du projet créé dans TestLink
- b. Test Plan Name : utiliser le nom du plan créé dans TestLink
- c. Build Name : choisir un nom de build. Pour garder un historique des exécutions, on pourra utiliser un nom de build paramétré, par exemple : **build-\$BUILD\_NUMBER**
- d. Dans les champs **Custom Fields** et **Test Plan Custom Fields**, ajouter le champ personnalisé créé dans TestLink (par exemple : java\_class)
- e. Sans quitter la configuration du step TestLink, ajouter un sous-step Maven dans la section Test Execution. Exécuter les cibles **clean test**
- f. Ajouter une extraction des résultats de tests JUnit : dans la section **Result Seeking Strategy**, ajouter une stratégie **JUnit Class name** avec la configuration suivante :
  - i. Dans **Include Pattern**, ajouter : **target/\*\*/TEST-\*.xml**

- ii. Dans **Key Custom Field**, ajouter le champ personnalisé créé dans TestLink (par exemple : `java_class`)
5. **Lancer un build du job Jenkins.** Vérifier que les résultats de TestLink sont bien intégrés dans la page du projet dans Jenkins :



6. **Ajouter l'intégration des autres tests du projet dans TestLink.**

Pour plus d'information sur l'intégration de TestLink dans Jenkins : <https://plugins.jenkins.io/testlink/>

## Exemple 14. Tests d'interface utilisateur

Dans cet exemple, nous allons utiliser Jenkins pour réaliser des tests d'interface utilisateur sur une application existante. Nous utiliserons Sélénium pour réaliser les tests et TestLink pour les rapports de tests.

1. Choisir une application Web à tester (par exemple : <https://fr.wikipedia.org> ).
2. Créer des tests d'interface utilisateur en utilisant l'outil de test d'interface graphique Sélénium. On pourra :
  - a. Intégrer directement Sélénium dans le framework de tests JUnit
  - b. Ou bien : utiliser un framework procurant une abstraction au-dessus de Sélénium (par exemple : Geb <https://gebish.org/> )
3. Intégrer l'exécution des tests Sélénium dans Jenkins. On utilisera une matrice de tests pour réaliser des tests sur différents navigateurs : Edge, Firefox, Google Chrome
4. Intégrer les rapports de tests dans TestLink

DANS LA PRATIQUE, IL EST COURANT D'UTILISER UN DÉPÔT DE CODE DÉDIÉ ET SÉPARÉ DES SOURCES POUR LES TESTS D'INTERFACE GRAPHIQUE (ET/OU TOUS LES TESTS D'INTÉGRATION).

## Exemple 15. Intégration des outils d'analyse statique

Dans cet exemple, nous allons intégrer dans Jenkins les rapports des outils d'analyse statique générés par des tâches dédiées des outils de build, par exemple Maven, Gradle, NPM.

1. Vérifier que l'exemple de code intègre bien les outils d'analyse statique directement dans l'outil de build Gradle (fichier build.gradle ) pour y ajouter les plugins d'intégration continue spotbugs, pmd, cpd et checkstyle.
2. Créer un nouveau job dans Jenkins exécutant les tâches Gradle correspondant à ces outils
3. A l'aide des plugins correspondants dans Jenkins, intégrer les rapports de qualité (par exemple : spotbugs, pmd, cpd, checkstyle) dans le job du projet

On pourra utiliser la documentation : <https://jenkins-le-guide-complet.github.io/html/sect-code-quality-tools.html>

