**AFCEPF – Cours du 23 octobre 2018 .**

Table des matières

[AFCEPF – Cours du 23 octobre 2018 2](#_Toc528318199)

[Les cycles Maven 3](#_Toc528318200)

[Construction d’un projet freestyle étape par étape : 9](#_Toc528318201)

[Deuxième projet – Projet MAVEN 12](#_Toc528318202)

[Le Scheduling 19](#_Toc528318203)

[Téléchargement de Nexus 21](#_Toc528318204)

[Création d’identifiants Jenkins 32](#_Toc528318205)

[Création du pipeline de manière graphique avec Jenkins : 33](#_Toc528318206)

[Etape 1 : récupération des sources sur Git, puis sauvegarde 33](#_Toc528318207)

[Etape 2 : build MAVEN 34](#_Toc528318208)

[Installation de Sonar 37](#_Toc528318209)

[Sonar et la qualité du code 44](#_Toc528318210)

[Le profil de qualité 44](#_Toc528318211)

[Quality gate 45](#_Toc528318212)

[Exercice : création et configuration d’un profil 46](#_Toc528318213)

# AFCEPF – Qualimétrie - Cours du 23 octobre 2018

AL32 – [\\192.168.102.11](file:///\\192.168.102.11) – Tt2w4R7f

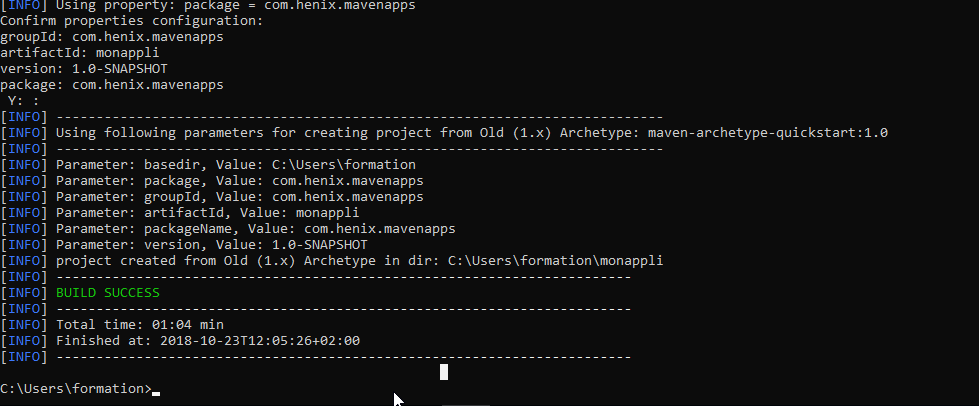
Lignes de commande maven sont instanciées par des -D

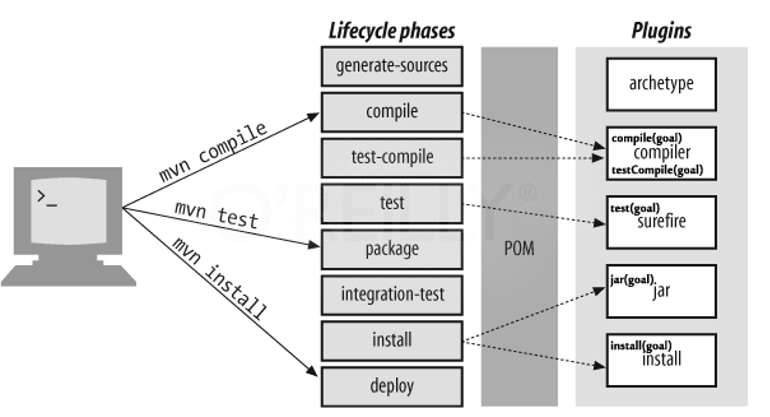
***mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DgroupId=com.henix.mavenapps -DartifactId=monappli -Dversion=1.0-SNAPSHOT -DpackageName=com.henix.mavenapps.monappli***

Shell bash : linux

Power shell : windows

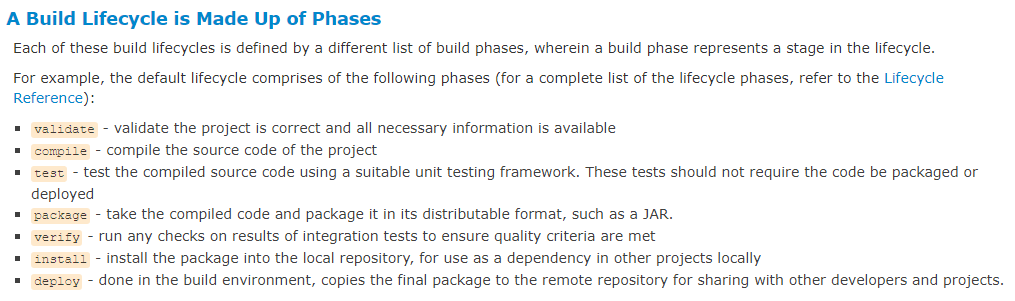
La ligne de commande a créé le repertoire :



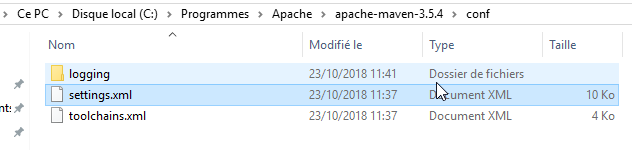


« Maven test » veut dire « go maven, jusqu’à la phase de test »

On peut dans le POM configurer pour sortir des chemins battus (faire exécuter un plugin perso, etc)

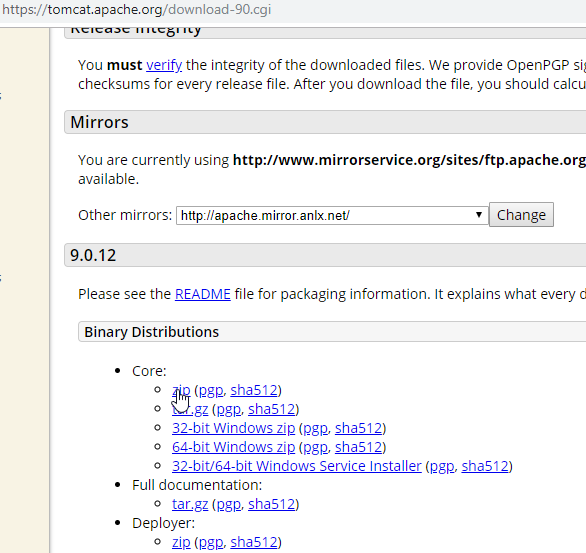
Les cycles Maven  
  


La conf se gère ici :

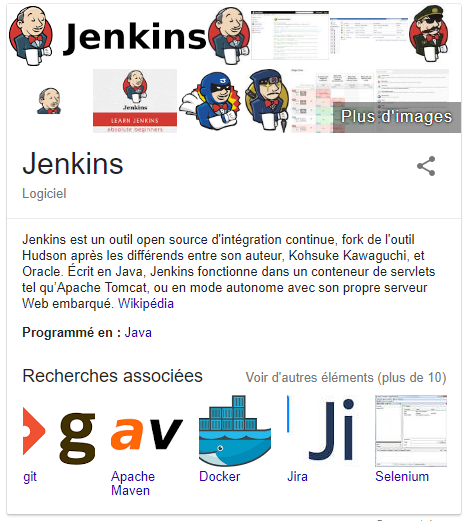


Notes sur le POM (Project Object Model) [clic](https://maven.apache.org/guides/introduction/introduction-to-the-pom.html)

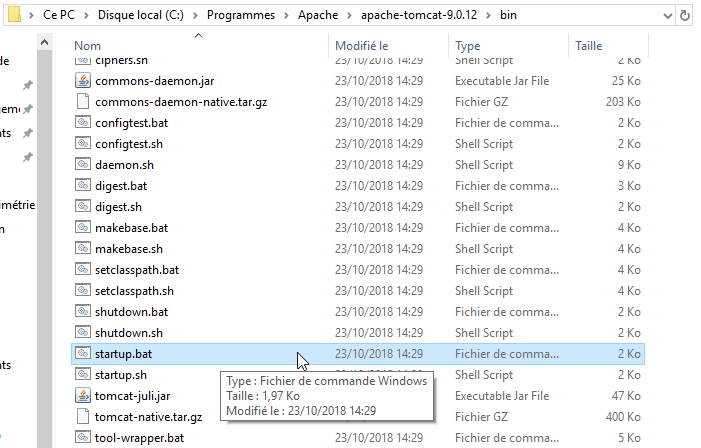
On installe tomcat :



On installe Jenkins



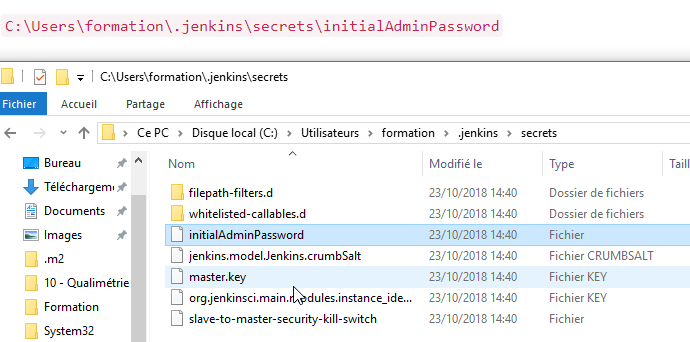
On fou le Jenkins dans Tomcat qu’on a préalablement installé -> Tomcat-9.0.12 dossier webapp



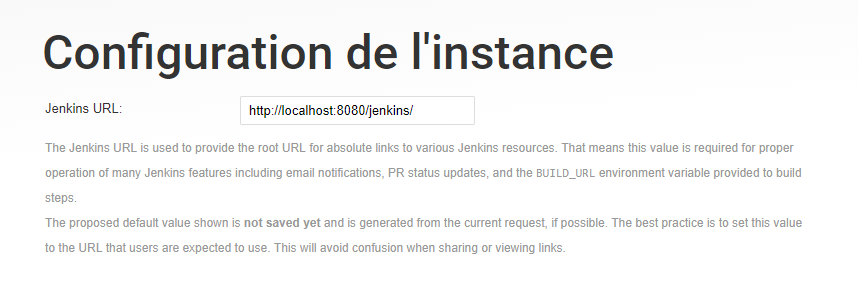
Lancement du startup.bat : ouvrir une cmd en Admin, puis aller sur le chemin d’accès du startup.bat 🡪 ça le lance et lance jenkins.  
  
Ensuite après avoir lancé le startup.bat, on va sur <http://localhost:8080/jenkins> :

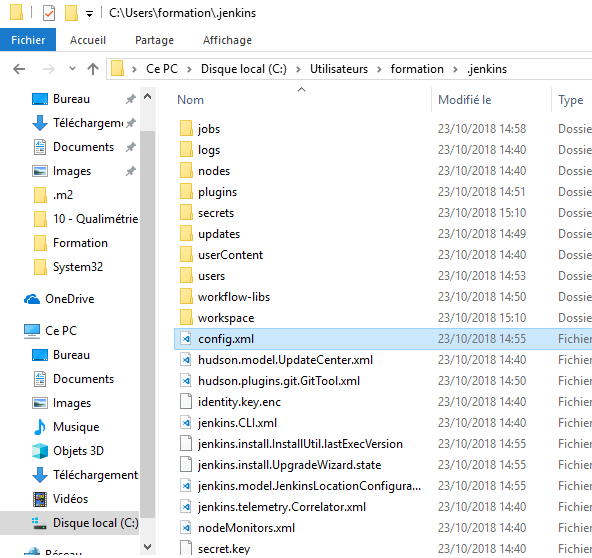


On va chercher les logs générés automatiquement :

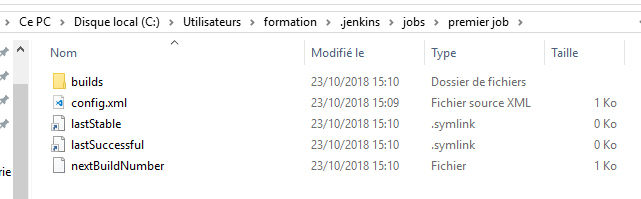


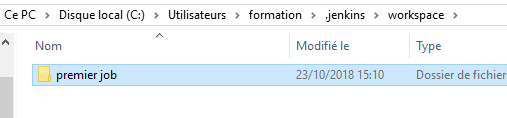
On crée un utilisateur,  
Puis on arrive à l’écran suivant : on doit ici configurer



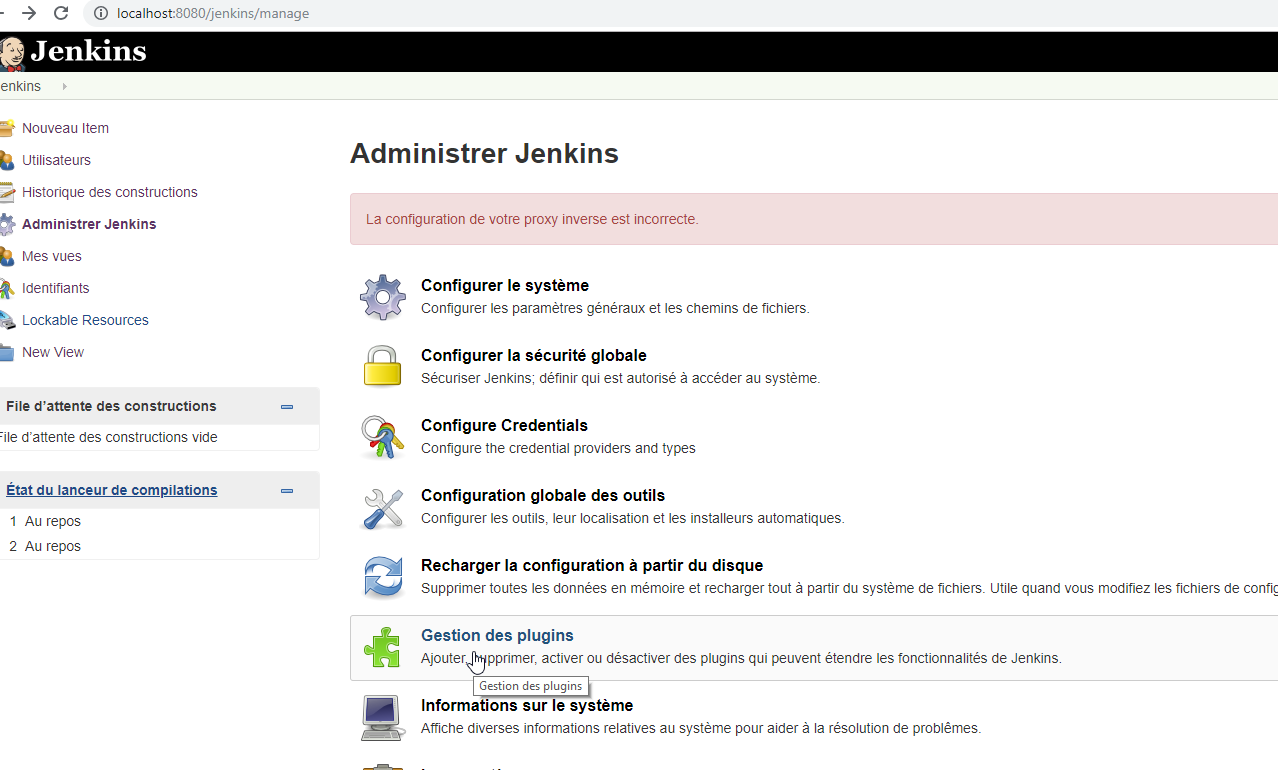


Dans le répertoire jobs :  
les jobs qu’on a créé, dans chaque sous dossier le config.xml, le nextBuildNumber, et les builds :

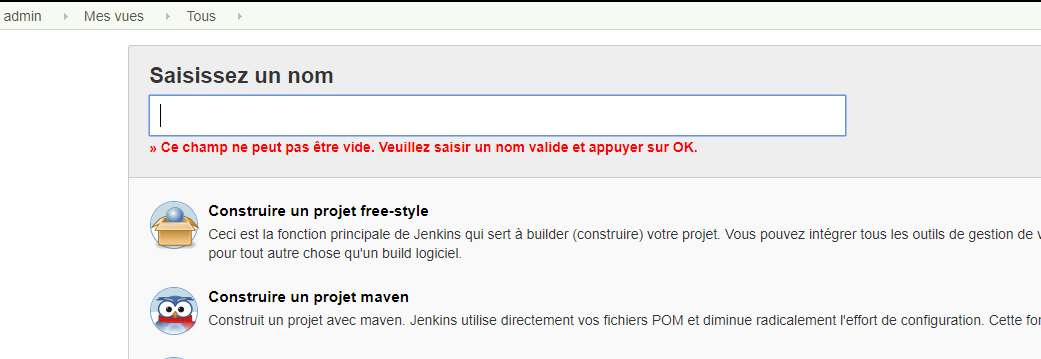


Dans Workspace : là ou travaille Jenkins !  
Il ne peut travailler ailleurs.  


(…)  
On veut tester avec un autre plugin, on télécharge MavenIntegration



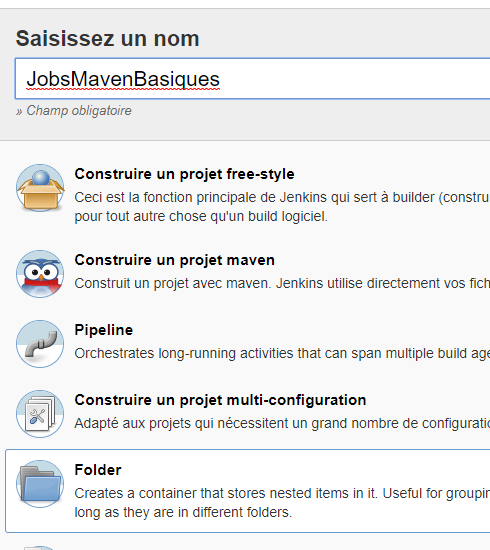
On ajoute le plugin Maven Installer puis on accède à une nouvelle option de création de projet :



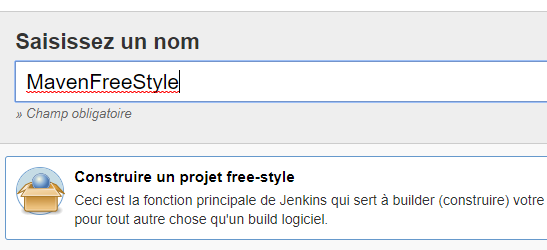
On configure Maven dans Administrer -> conf globale des outils

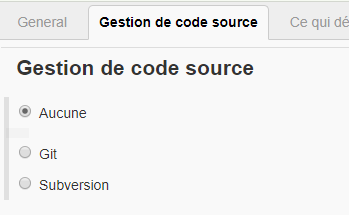
## Construction d’un projet freestyle étape par étape :

Création d’un folder

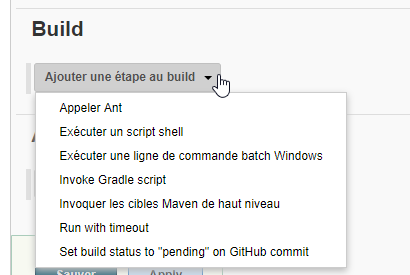


Maven FreeStyle projet :

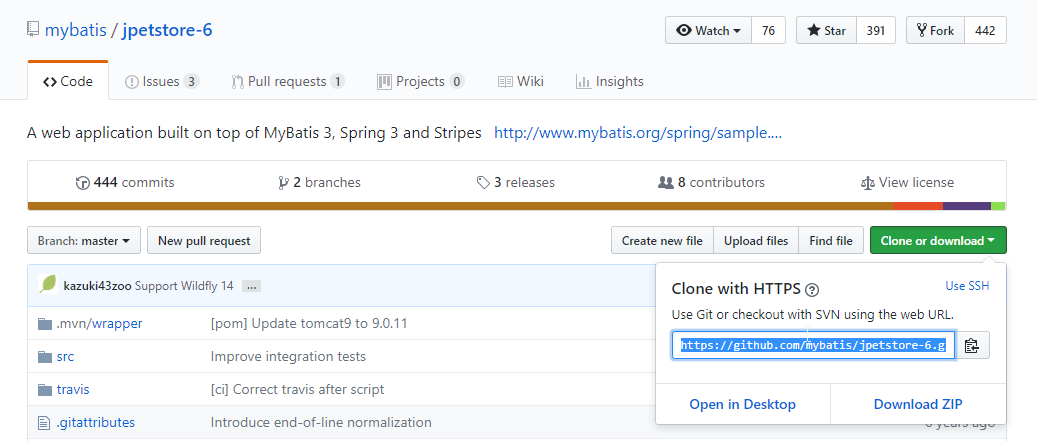


On veut ensuite récupérer les jobs et les builder.  
Ceci va nous intéresser, Git :  


On tire les données de Git.

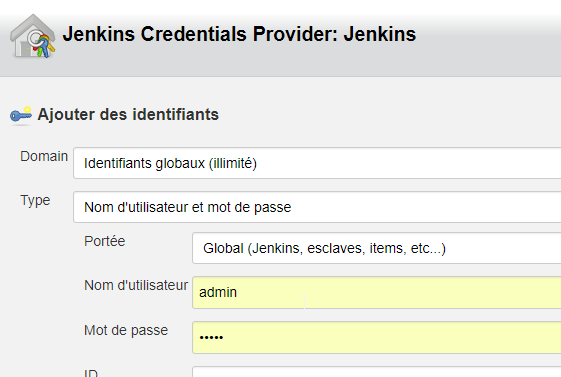
Onglet Build à remplir aussi.  


Maven bien formé est nécessaire en prérequis : il faut donc un pom.xml à la racine, avec src, etc.

On va utiliser jpet-store : [clic](https://github.com/mybatis/jpetstore-6)  


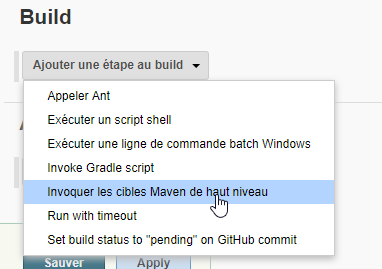
Le projet est visible publiquement, no need de credentials. Si on devait pusher, on en aurait besoin.

Au niveau des crédentials, pour ne pas avoir les infos sensibles en clair, Jenkins possède une partie spéciale, encryption des datas dans une zone protégée de Jenkins, puis utilisation d’alias pour y faire référence pour nous.

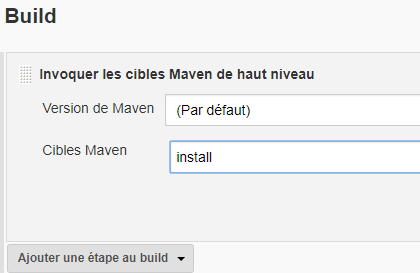


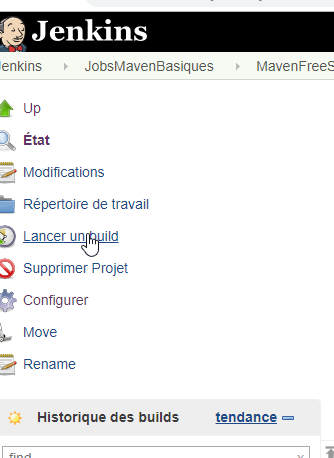
Les vrais infos sont donc protégées, mais nous savons à quoi nous faisons référence.

On renseigne la cible build :

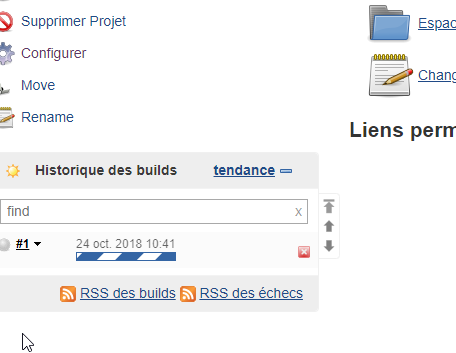


On va jsuqu’à l’Install :

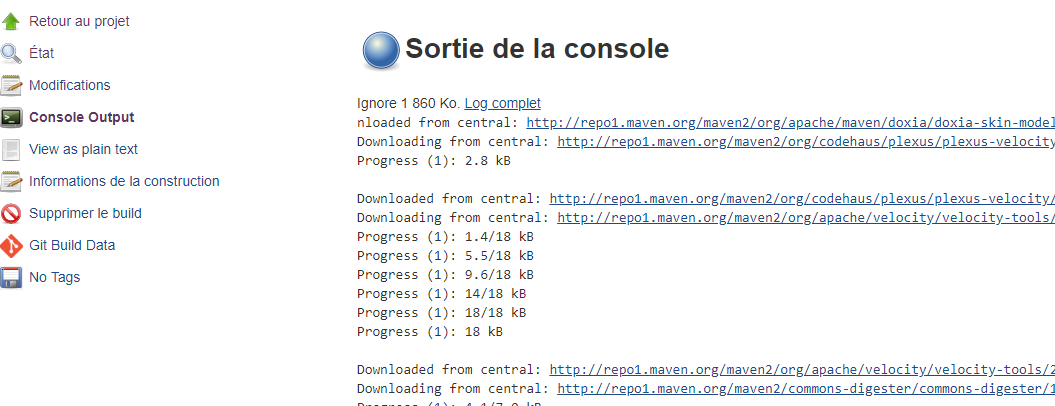


On SAV, puis on lance un build :  


On voit ensuite l’avancement ->

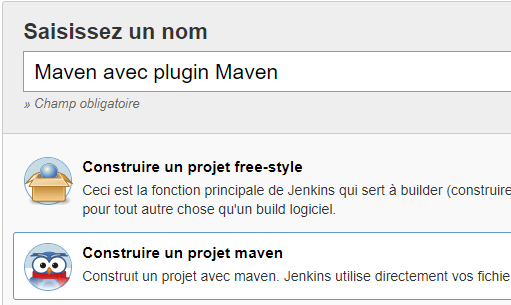


On peut voir la sortie console :



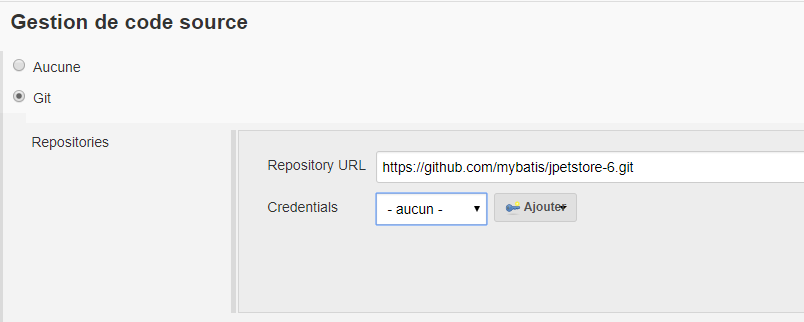
## Deuxième projet – Projet MAVEN

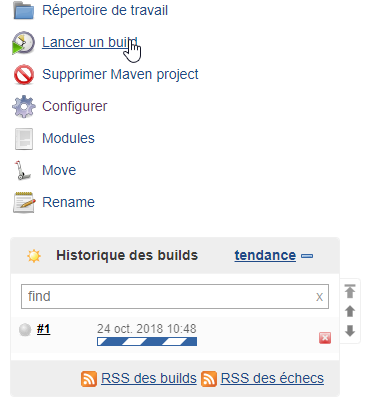
Après avoir installé le plugin Maven Install 🡪



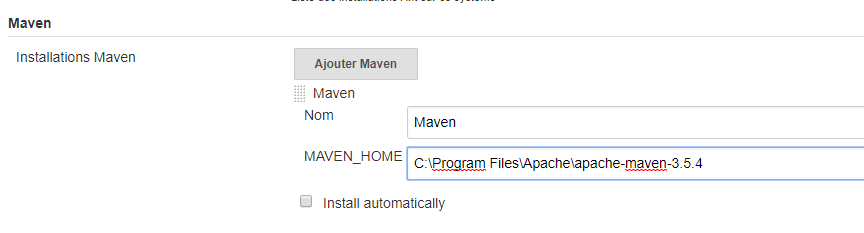
On doit ensuite choisir avec plus d’onglets, dont dans le build le choix du pom  


On renseigne le git, puis on sauvegarde



On lance le build :  


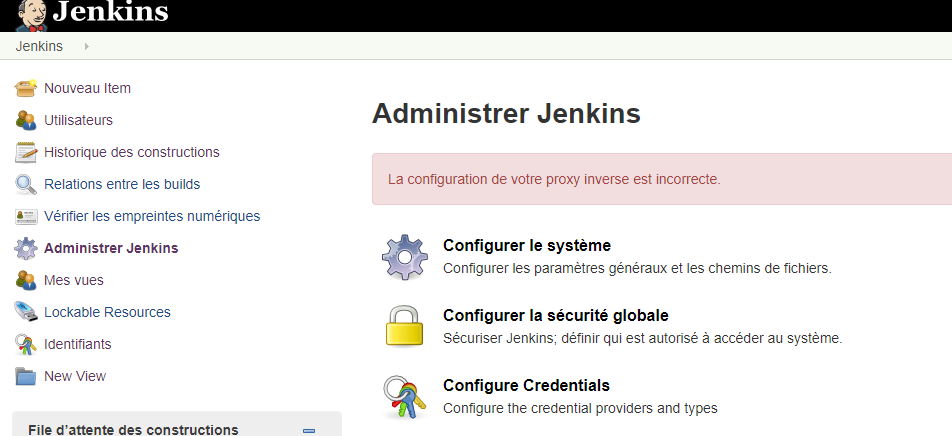
Important d’avoir défini dans la Configuration Globale des Outils



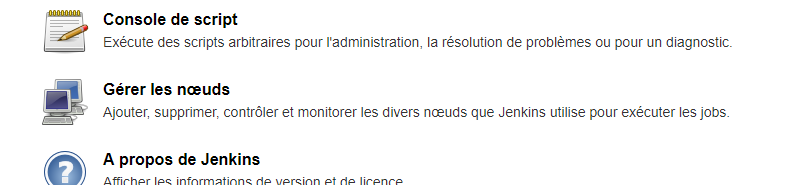


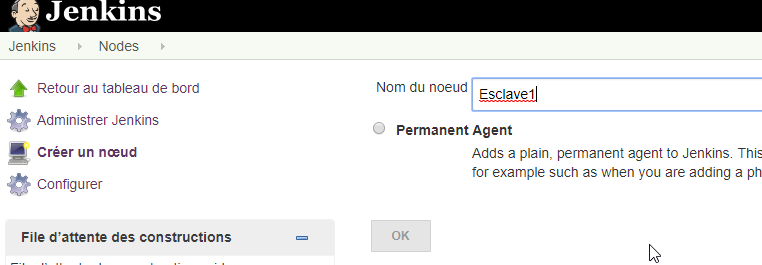
Définition des esclaves :  
On créée un esclave, on lui donne un répertoive de travail, et là on peut jouer sur les configs 🡪 prendre le git par défaut ? le jdk par défaut ?  
On peut le définir ici.

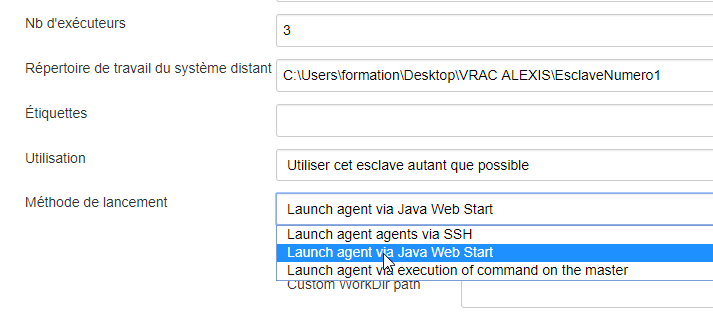
Création dans Jenkins 🡪 administrer 🡪 gérer les nœuds

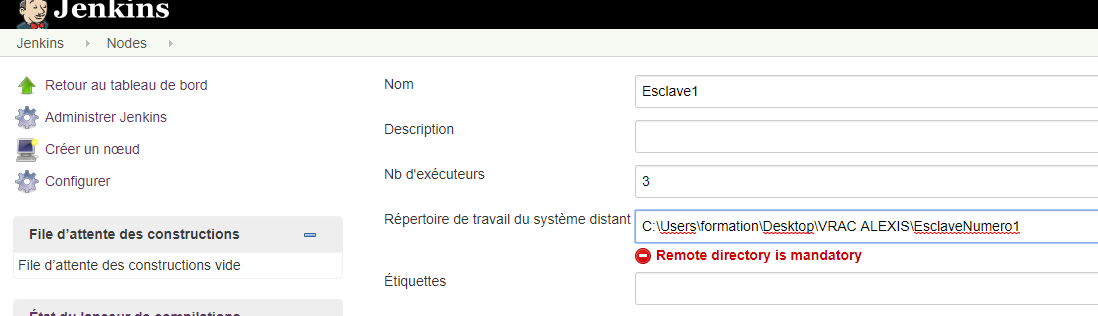


Ici :

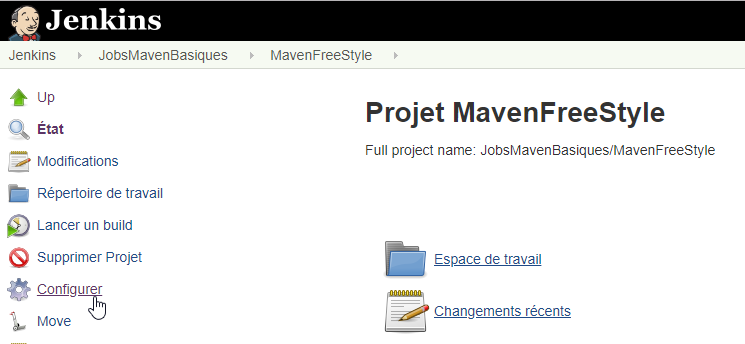




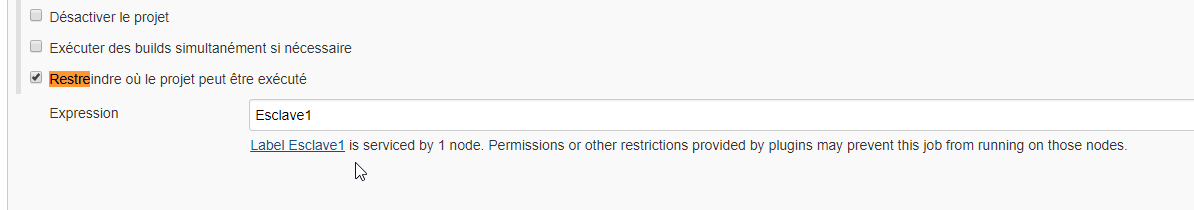
On le met en JavaWebStart  


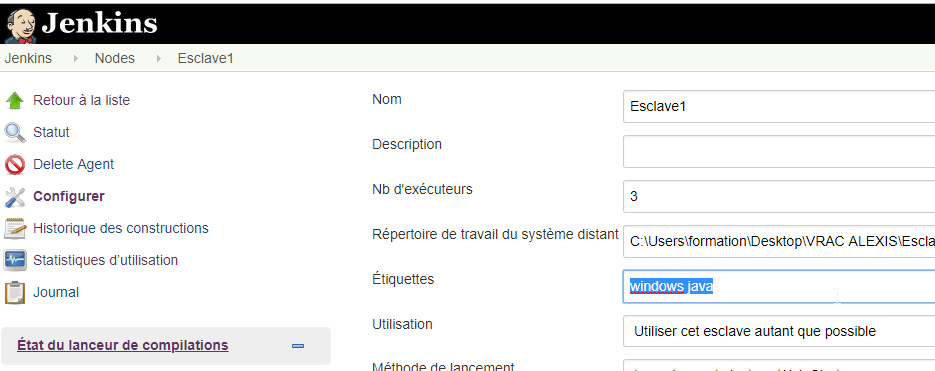
On configure le nœud en lui donnant un chemin d’accès sur un folder sur notre machine en local  


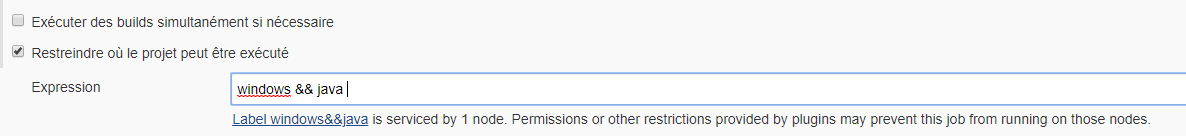
Nb : il faut aussi faire pour éviter le SSH, dans les configurations globales  


On peut lier Job par Job, dans la configuration du Job, ou il doit être lancé 🡪  
A partir de la page du Job  


🡪 Configuration

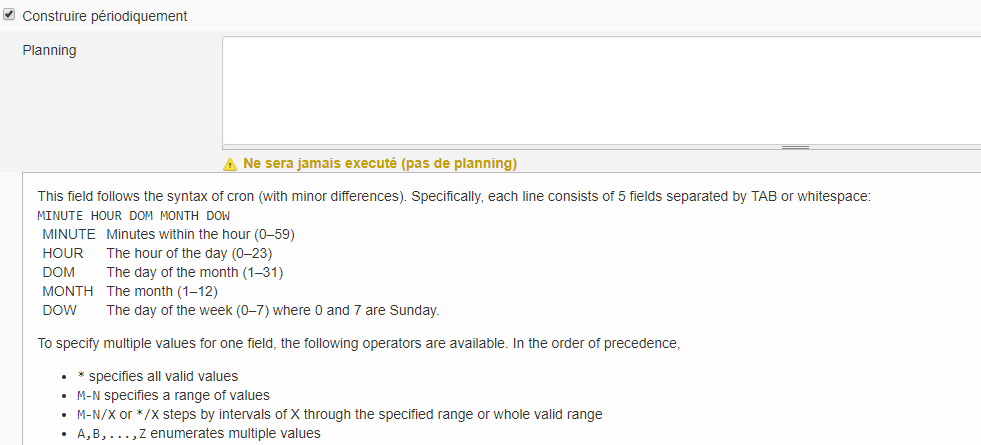


Ajout d’étiquettes :  


Et au niveau du projet, on utilise les liaisons des étiquettes avec && :  


## Le Scheduling

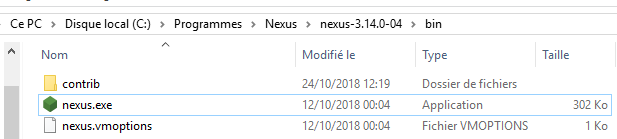
Autre fonctionnalité importante  

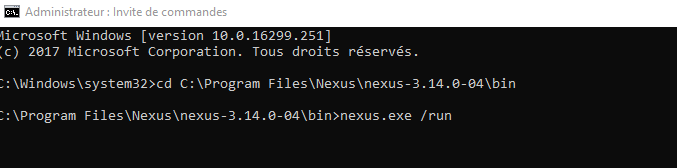

L’aide permet d’avoir des infos :  


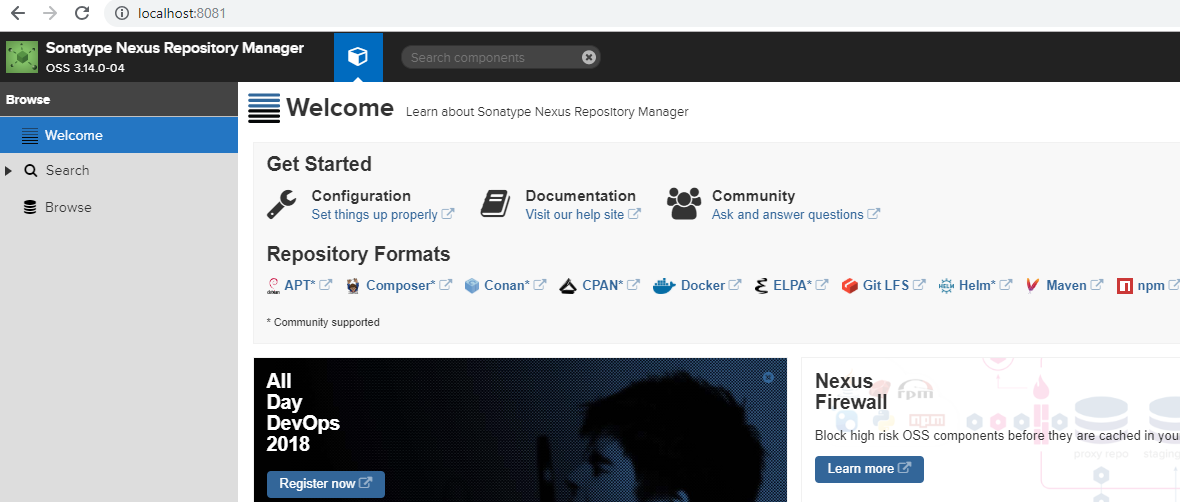
Ainsi :  
 \*\*\*\*\* lance l’exécution toutes les minutes.  
0 10 \* \* \* lance l’exécution à 10h tous les jours (de tous les mois).  
0 10 \* \* 1-5 lance l’exécution à 10h tous les jours en semaine

## Téléchargement de Nexus

Nexus OSS : c’est un repository de binaire.  
Il gère le format Maven, ce qui nous intéresse, mais il gère aussi le Python le Bower et d’autres binaires.  
On choisit la version Windows 🡪  


On le met dans un dossier NEXUS créé pour l’occasion, on y dézippe Sonata et Nexus :  
  
  
On entre en ligne de commande un cd « path to Nexus »,  
On lance run nexus.exe **/run**

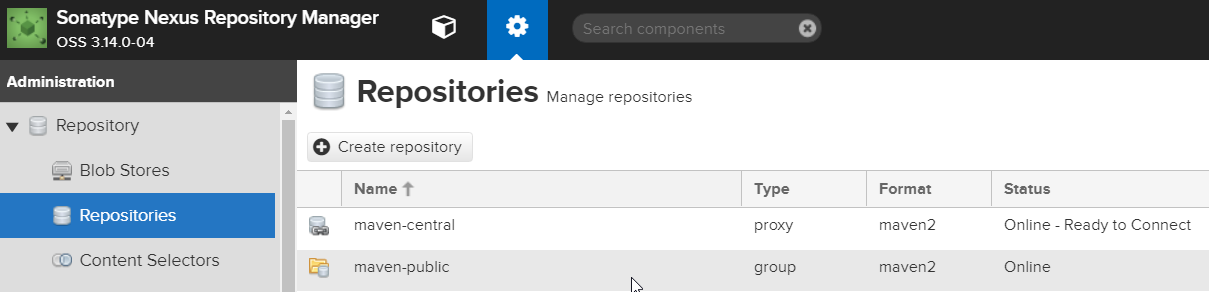


Port par défaut : localhost :**8081**  
On peut donc lancer Jenkins

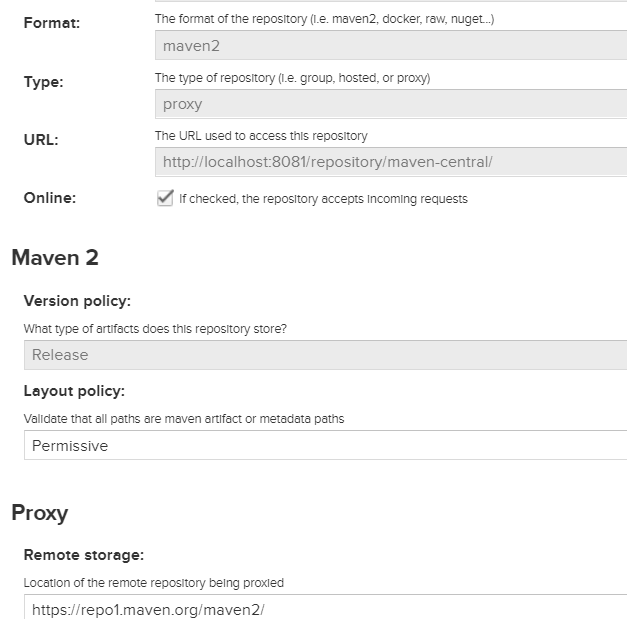
Trois repositories possibles :

* Proxy : copie interne d’un repo externe
* Hosted : création d’un dépôt à partir de nada
* Group : regrouper des dépôts 🡪 on ne sait pas trop, on va chercher dans un groupe rassemblant les éléments.

Connection : par défaut, admin (id) admin123 (mdp)

On va dans les paramètres (la roue crantée en haut à gauche), puis Repositories :  


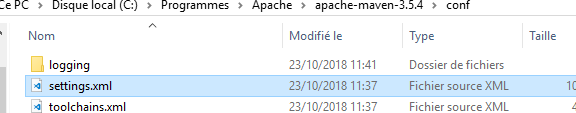
Maven-central



Nexus permet de mettre en place un proxy,  
Ce dernier entasse toutes les dépendencies qui ont déjà été utilisées / téléchargées par tout utilisateur de l’entreprise ;  
Quand un user lance un projet Maven, le pom.xml définit quelles dépendances sont utilisées par le projet ;  
Recherche d’abord dans le repo local .m2/repository  
Si ne trouve pas, on cherche dans le proxy (Nexus – réseau interne entreprise), si on trouve basta.  
Si on ne trouve pas, alors effectivement on recherche sur les internets (repository central)

Avantages du proxy :

* gain de temps de build
* pas d’interaction avec l’extérieur
* sauvegarde interne des dépendances.

On doit ensuite dire à Maven de travailler avec le Nexus qu’on a installé,  
ON doit donc modifier la balise Mirrors du **settings.xml** de Maven.  


On cherche la balise Mirros, qui contient un exemple :

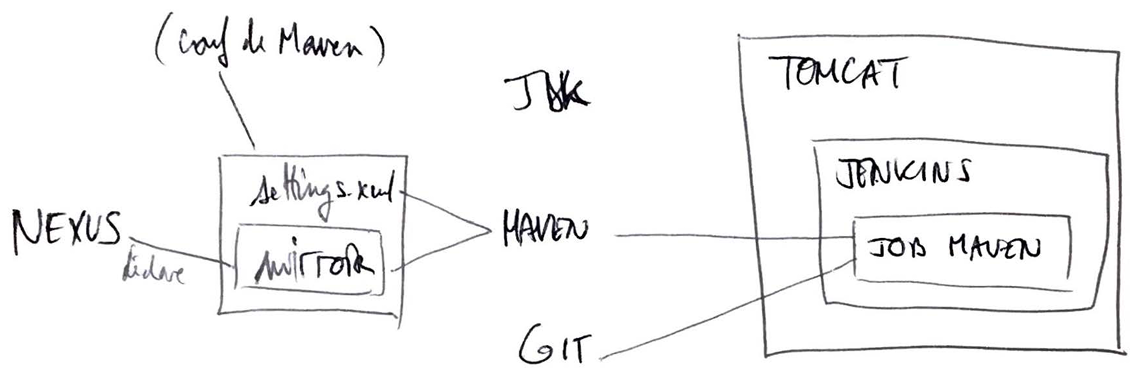


Maven interroge Nexus (le proxy), il faut le lui dire dans ses configs et c’est ce qu’on est en train de gérer.

On rajoute donc celui-ci :

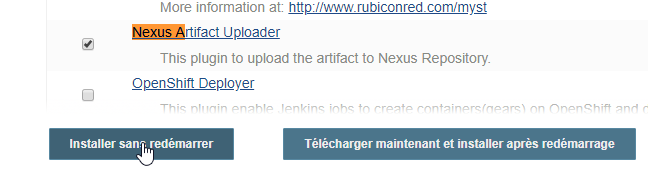


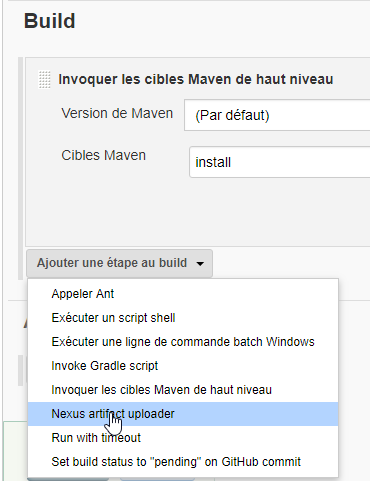
On peut lancer le build, téléchargement dans Nexus du coup !  
Logs du build :  

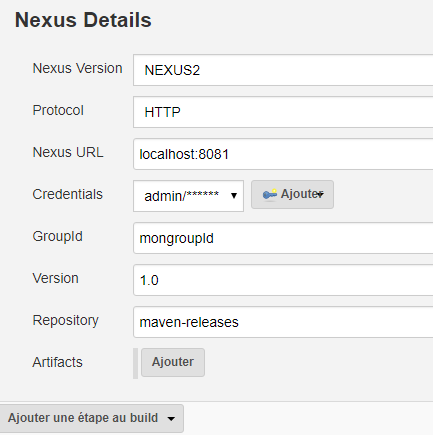
Nexus gère les dépendances Maven

On s’installe ce plugin :

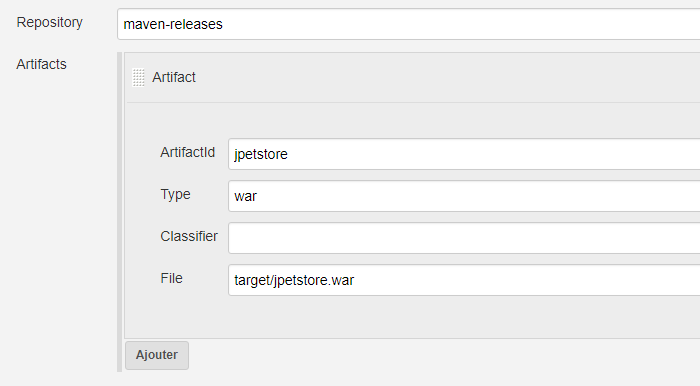


Finalement, on reprend un projet FreeStyle par simplicité, on entre dans ses configs et on rajoute une étape de build :  


On rentre ensuite les paramètres :  
On se souvient qu’en Maven, le SNAPSHOT est détruit quand la version est stable, et on passe à la version suivante.

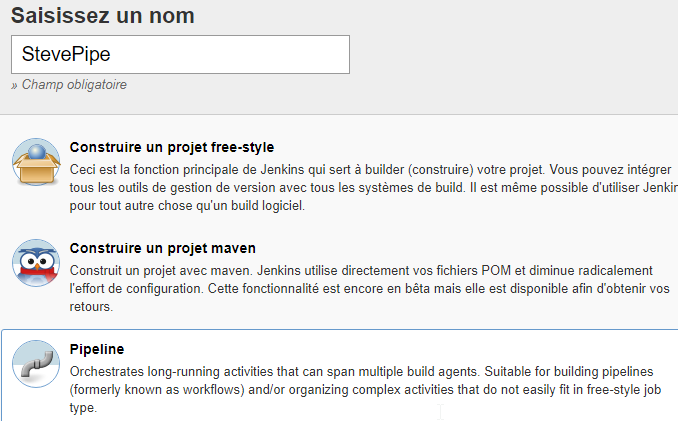


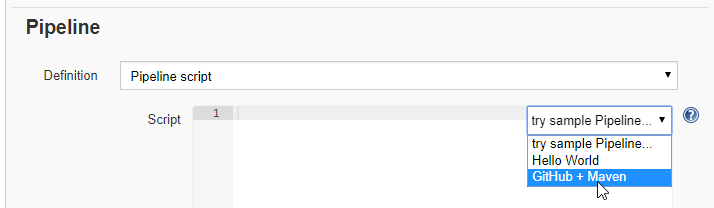
Puis on ajoute une étape au build



On peut ainsi lancer le build.

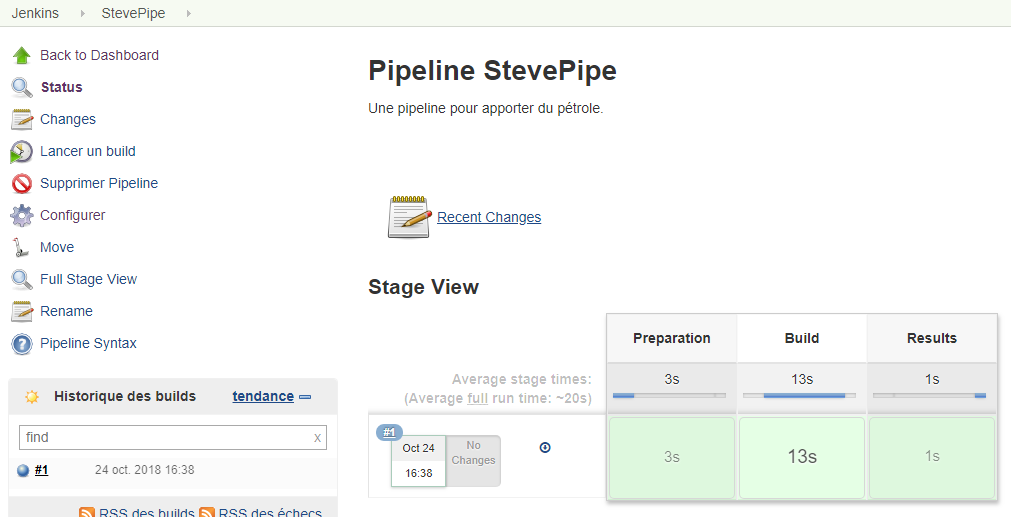
Maven pipeline, aussi dite dans le jargon la « Steve Pipeline », ou la « G. Abitdbol ».



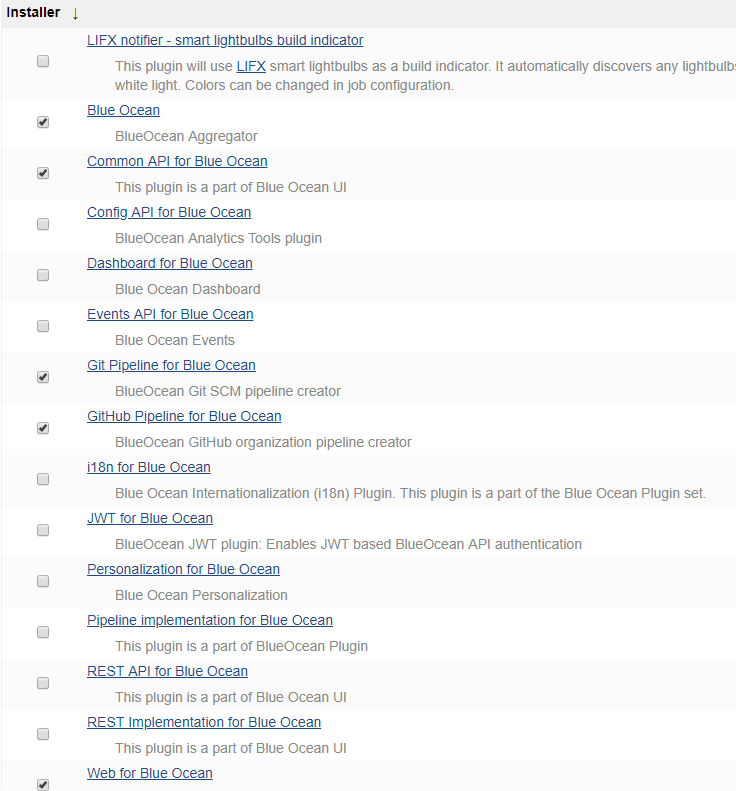
On peut ensuite utiliser un sample :  


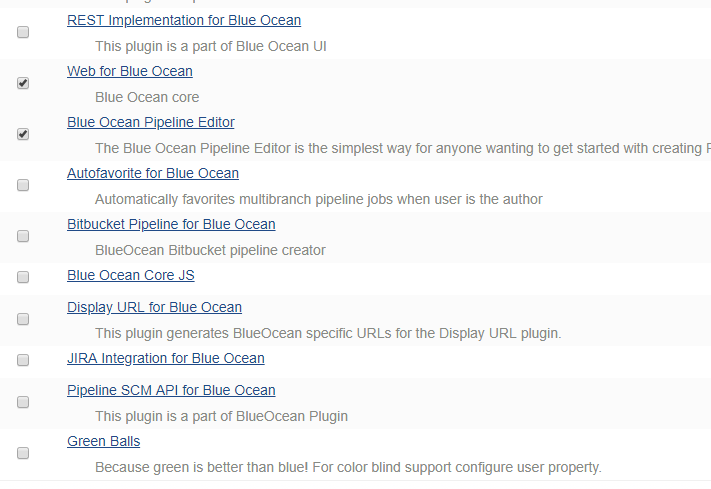
On voit ainsi apparaitre quelques belles lignes, presque des courbes finement dessinées, mmmm..  
On en croquerait !



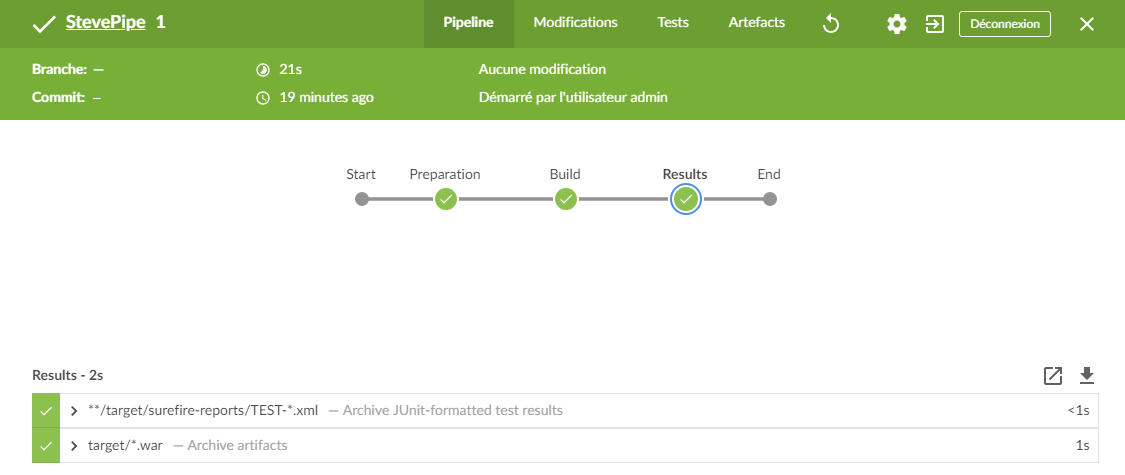
On modifie un peu, on launch puis :  


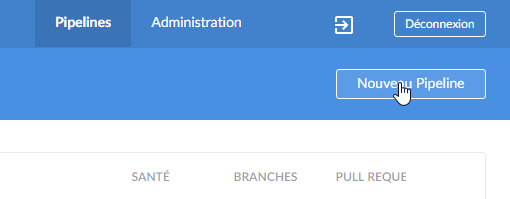
Ajout de plugins !

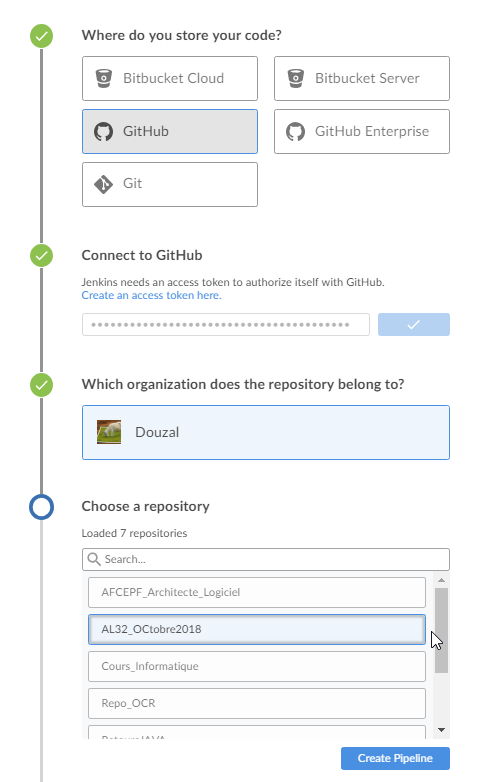




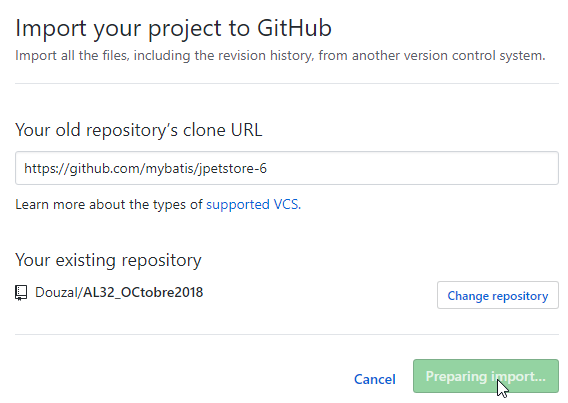
On a ensuite une vue plus stylée :  

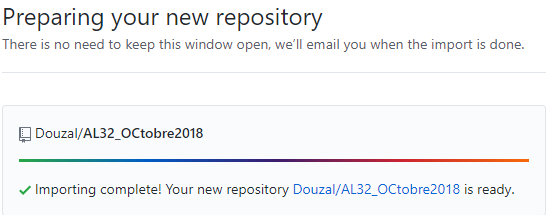

La vue est plus belle, et détails :  


On peut aussi créer un nouveau pipeline !  


On crée un token github, et là c’est le drame ->  


On revient sur notre repo sur Git, on importe avec l’url de jpetStore, le projet dans le repo qu’on a créé pour l’exercice (ici, AL32\_Octobre2018)



On a ainsi les droits d’écriture sur le projet, par cette petite astuce.  


Création d’identifiants Jenkins

ID qu’on va utiliser juste après :

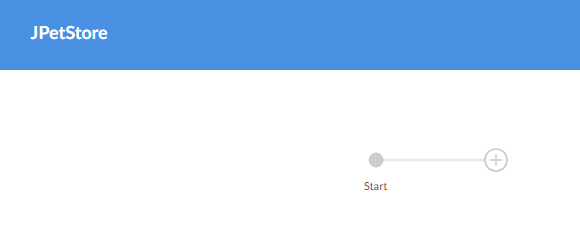


Puis System -> identifiants globaux :

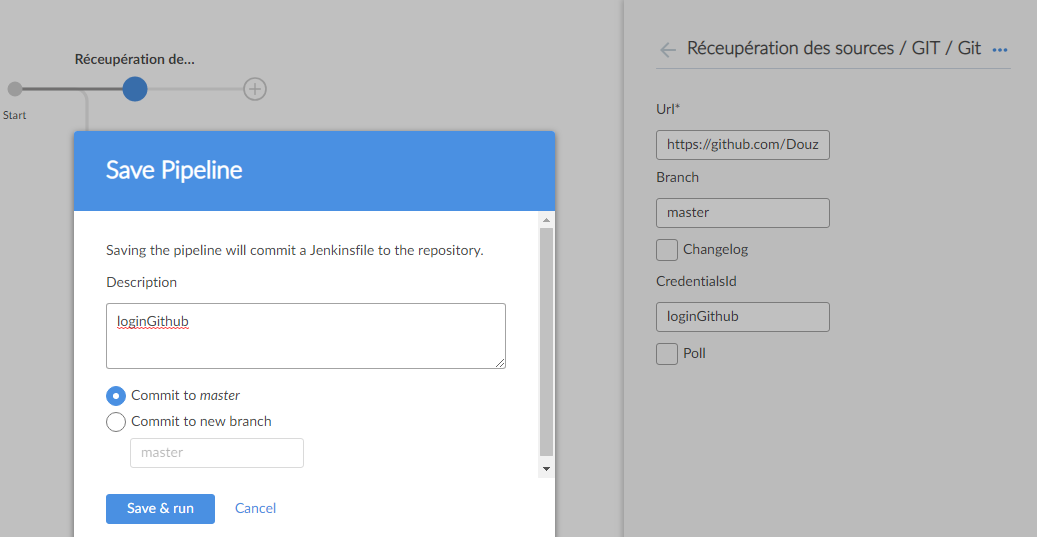


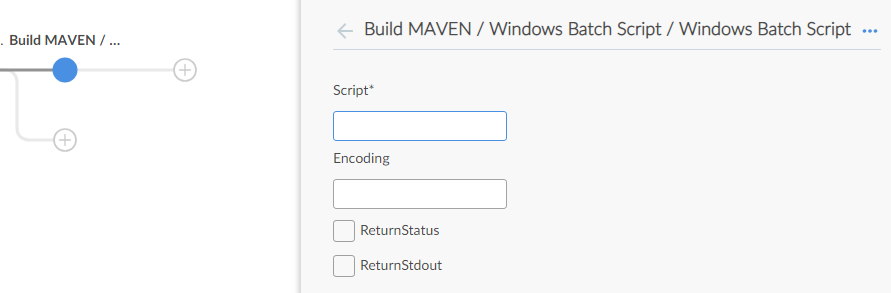
On ajoute donc son login github :  

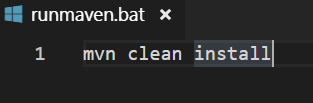

## Création du pipeline de manière graphique avec Jenkins :



### Etape 1 : récupération des sources sur Git, puis sauvegarde

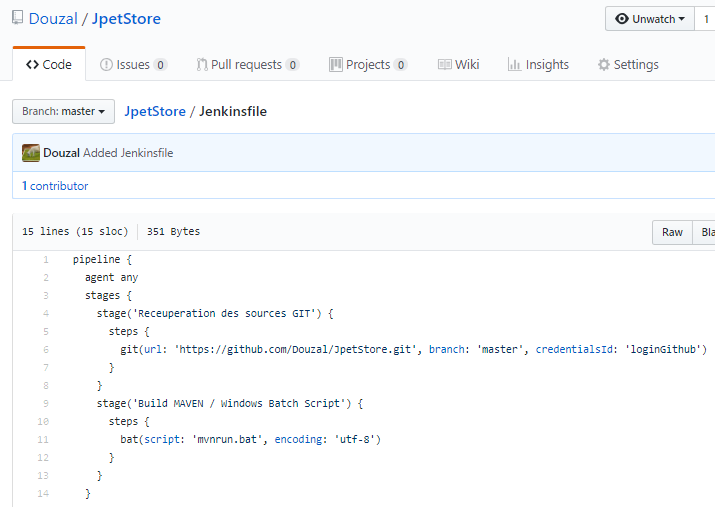


Etape 2 : build MAVEN  
Création d’un Script Windows  


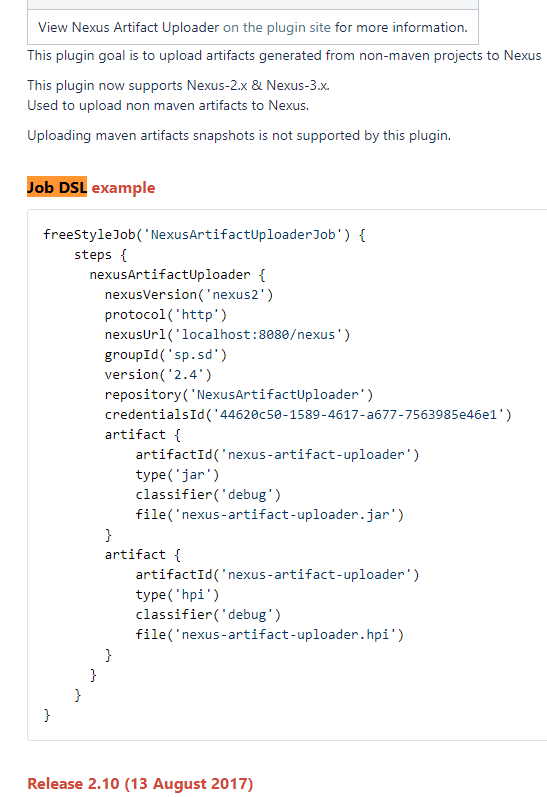
On crée le runmaven.bat  


On l’add au repo local qu’on a cloné préalablement de Git.

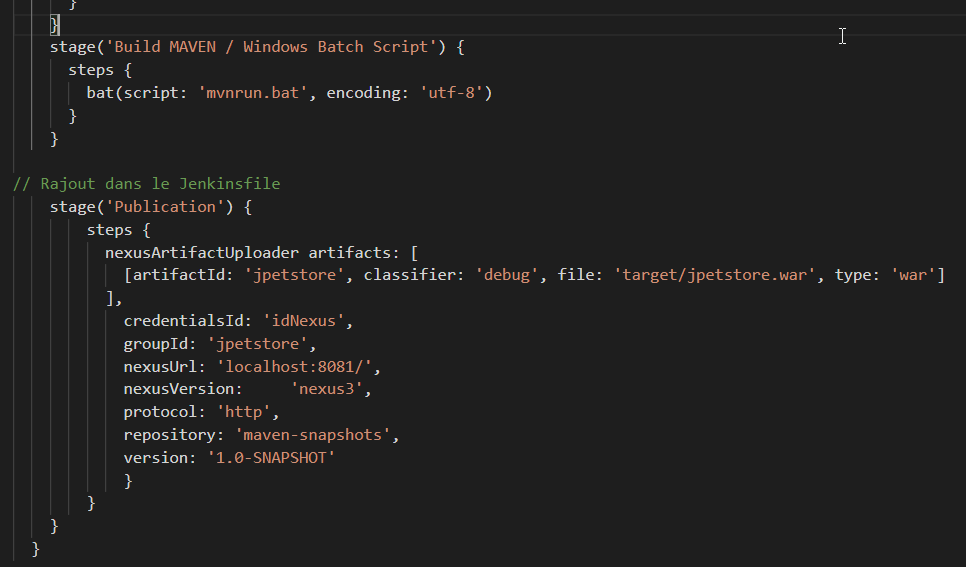
git add -A  
git commit -m « message »  
git push

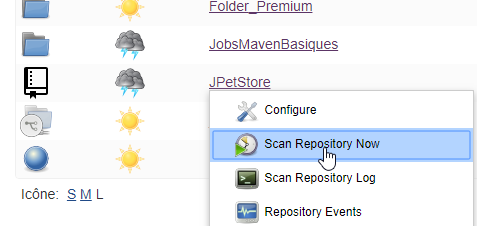
Puis on voit sur le repo git (internet) qu’une file a été ajoutée avec les credentials etc :  


Sur cette url : https://wiki.jenkins.io/display/JENKINS/Nexus+Artifact+Uploader

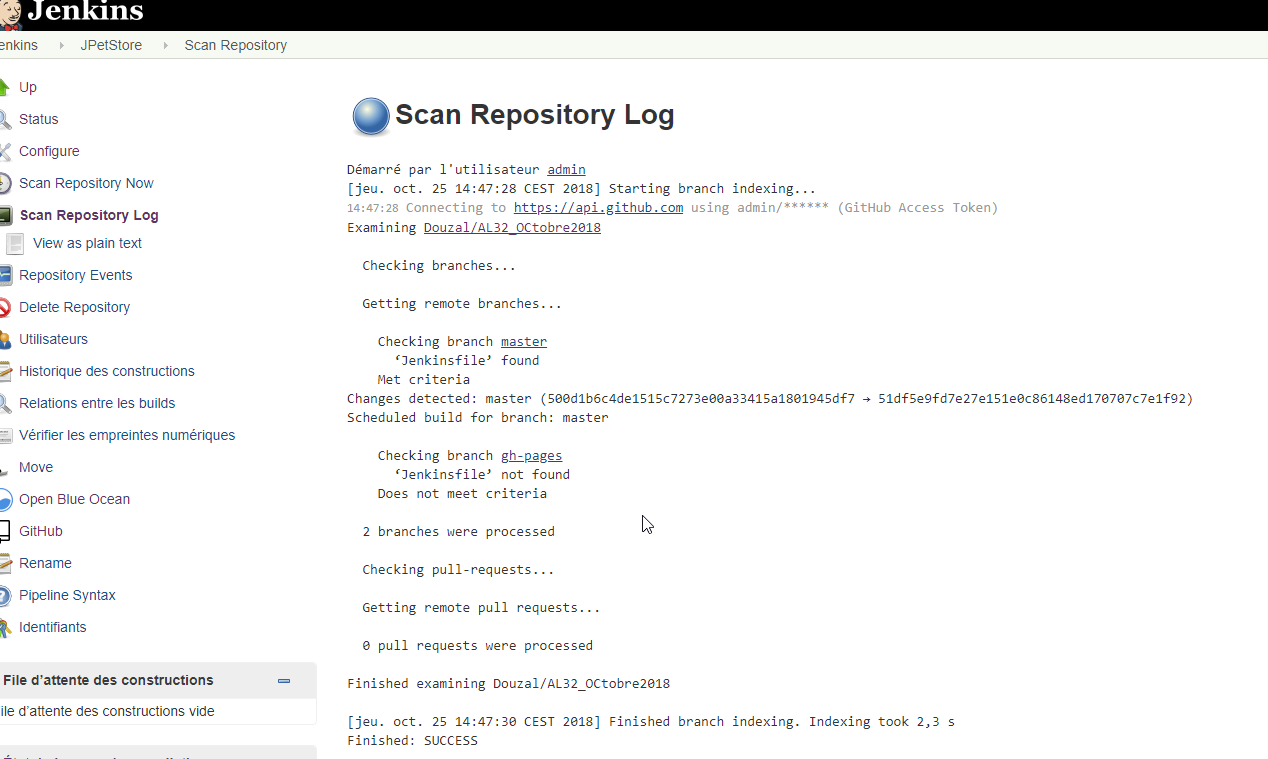
On copie un bloc de code :  


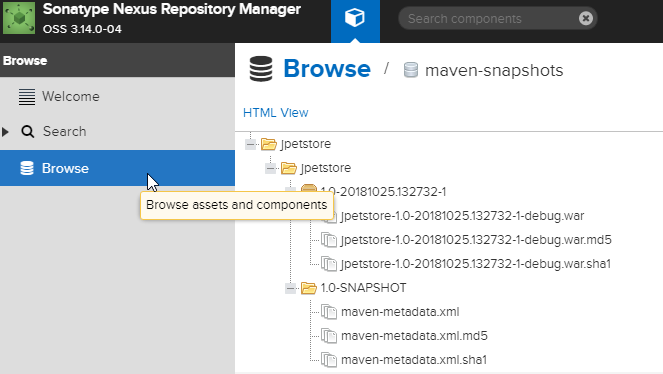
On rajoute ainsi au JenkinsFile pullé du git, on rajoute donc en local :



Comme auparavant, on ajoute donc sur le repo distant le fichier màj.  
On peut ensuite lancer un scan du repo, le système comprend qu’il y a un fichier màj, le prend en compte, et lance le scan :  


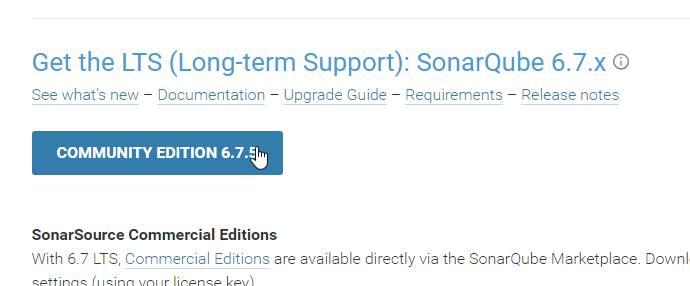
On obtient bien le résultat :



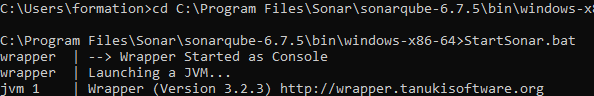
On a dans Nexus, dans le dossier SNAPSHOT:  


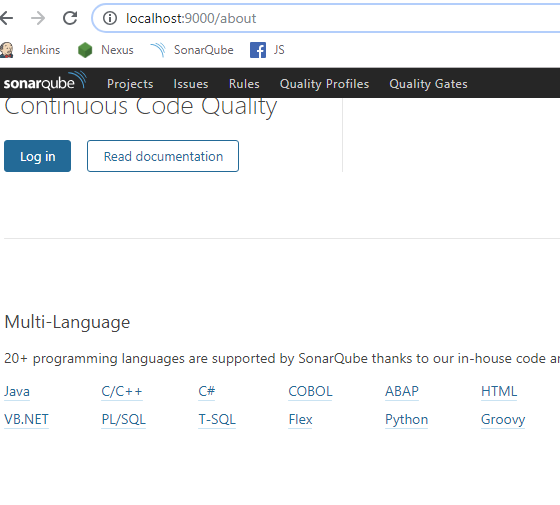
## Installation de Sonar

Sur le site de Sonar, téléchargement de la dernière version stable.

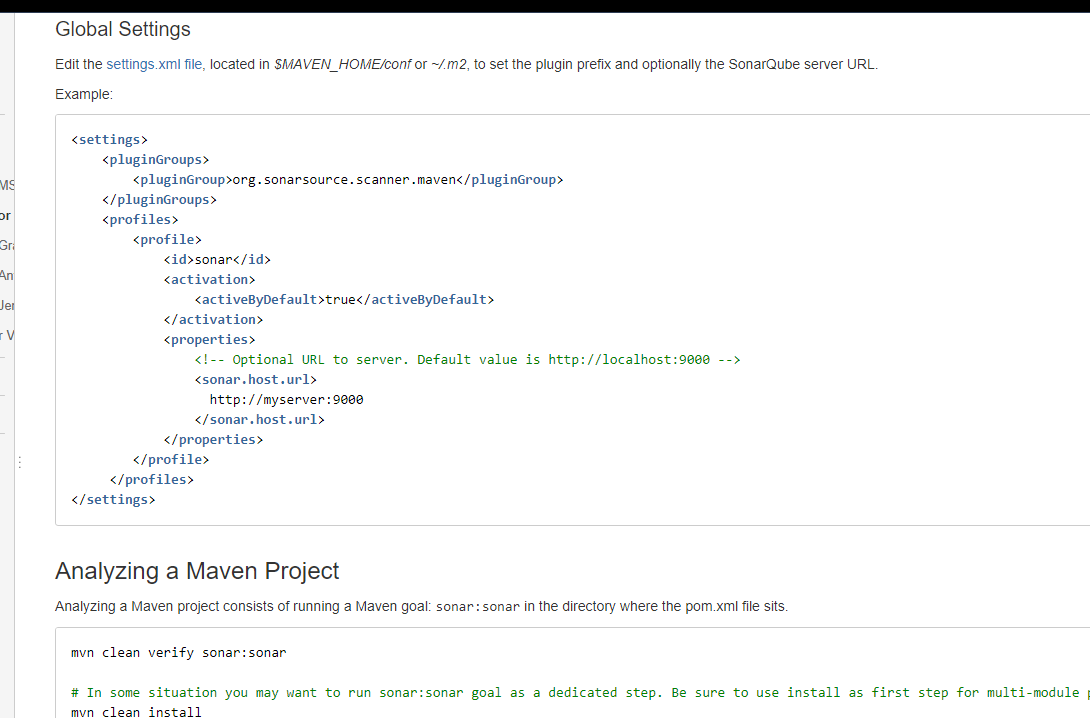


On le place par exemple dans Programmes > Sonar  
On ouvre une commandLine 🡪 cd « path » 🡪 on lance ensuite le Startup.bat

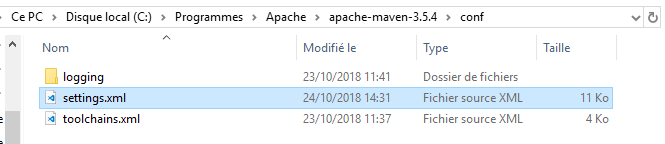


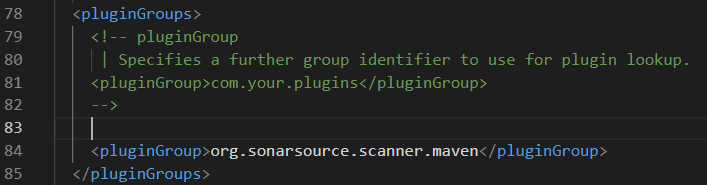
On a donc notre Sonar fonctionne, port localhost :9000  


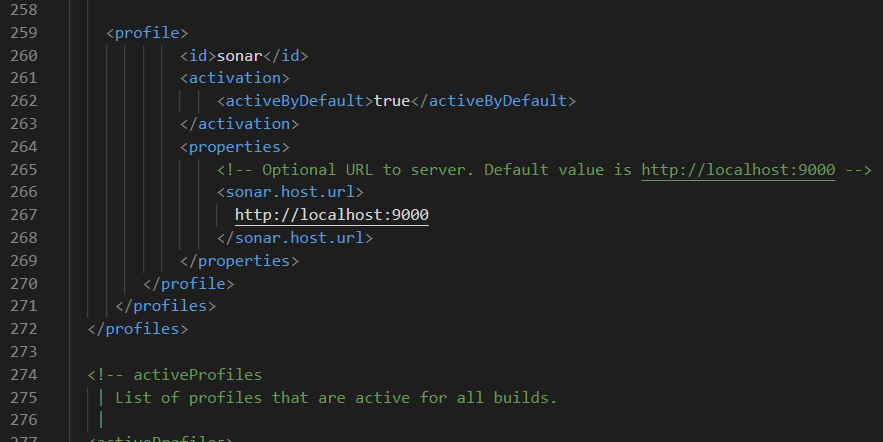
On doit dire à Maven qu’on veut utiliser sonar, on va jouer sur la config again.  
On regarde la doc : <https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Maven>

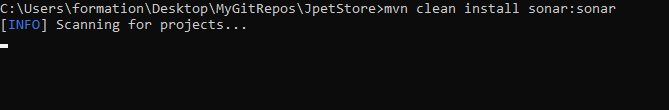
On va adpater le bout de code trouvé :  


On retourne dans le settings.xml

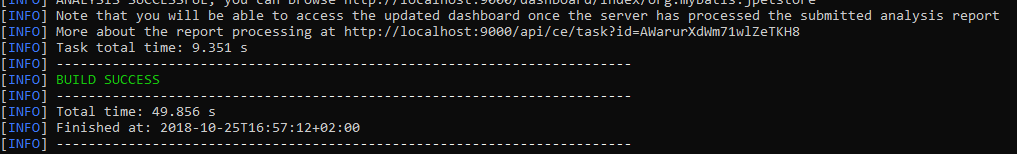


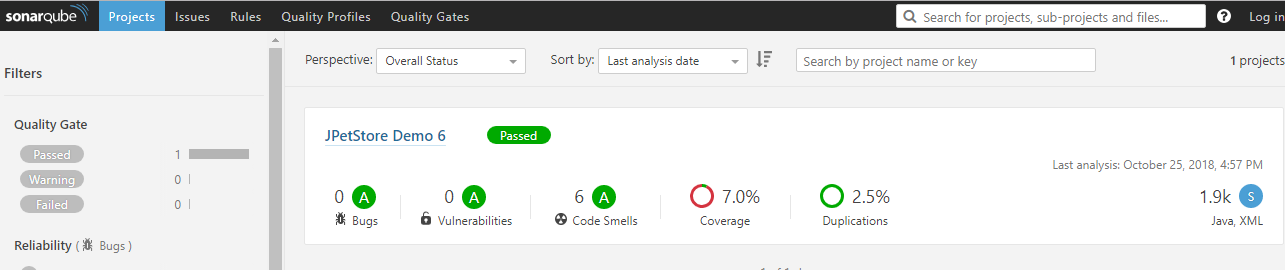
On ajoute :   


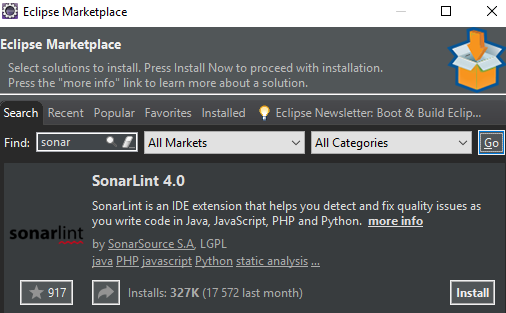
Et dans les profils, toujours dans le settings.xml :  


On peut lancer un **mvn clean install sonar :sonar** 🡪  


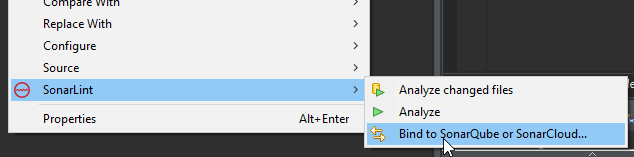
Nb : la commande **mvn clean install sonar :sonar** lance un clean install à travers le plugin sonar **sonar:sonar** pour plugin:tache

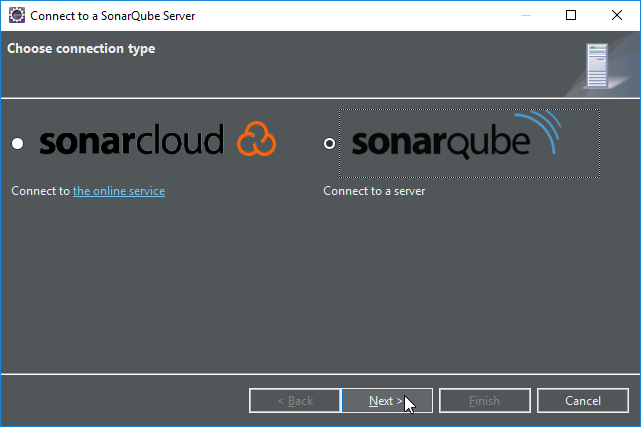
Finalement, ça build !  


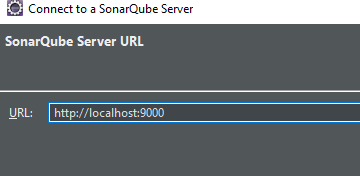
Résultat, sur la page localhost :9000 🡪  


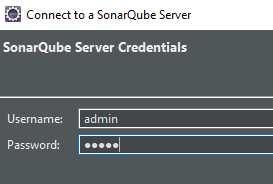
On ouvre ecplise, on improte un projet Maven ;  
On télécharge sur la marketplace SonarLint 🡪  


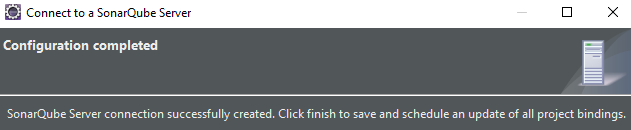
(restart de Eclipse)

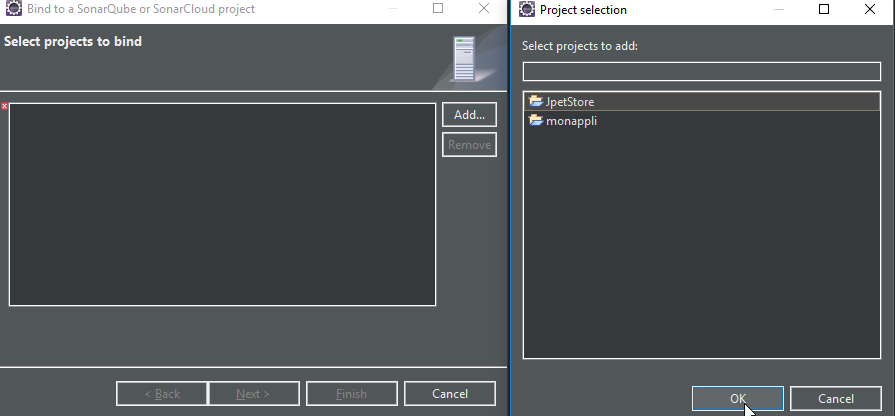
On clic droit sur le projet, puis :  


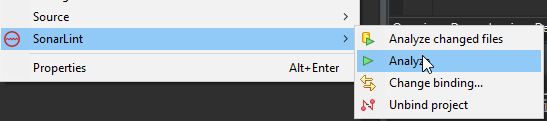
Puis :  


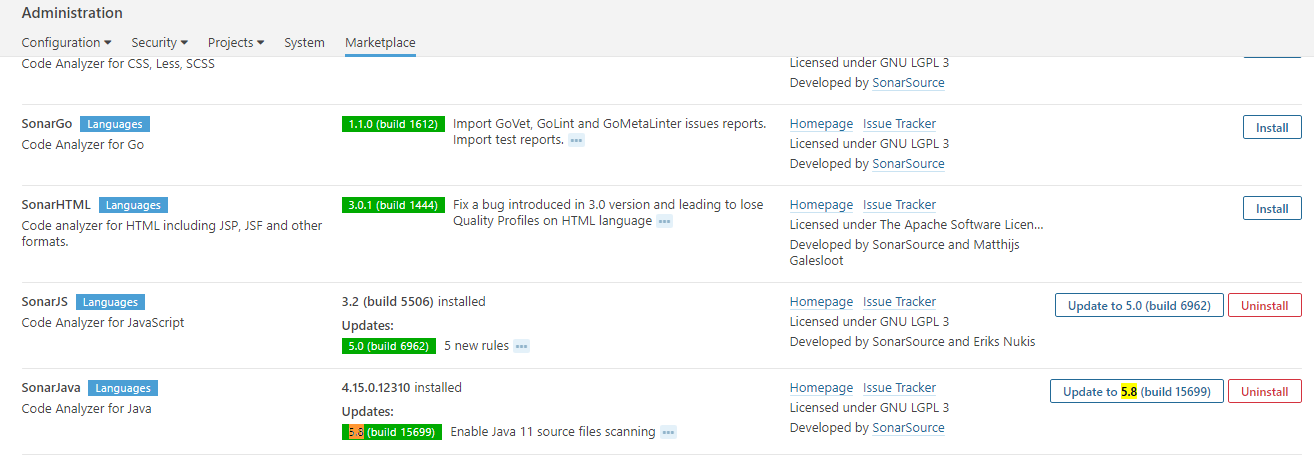
Puis  


Puis admin / admin  


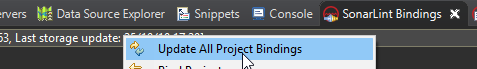


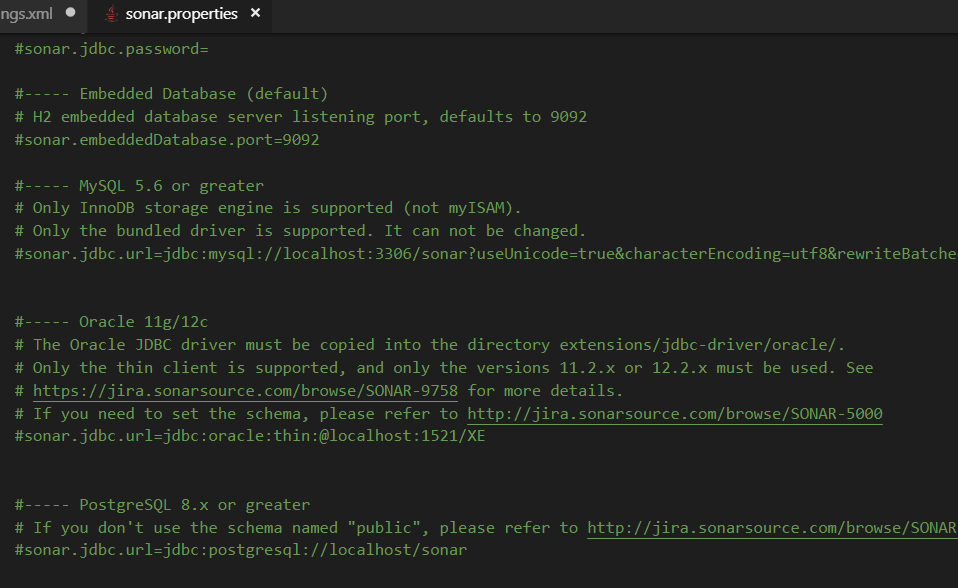
On add le projet JPetStore 🡪  


On peut ensuite terminer la config easy, puis lancer une analyse :  


Dans Sonar 🡪 administration 🡪 MarketPlace on peut màj les plugins ;  
On màj le SonarJava  


On peut restart ensuite pour prendre en compte.

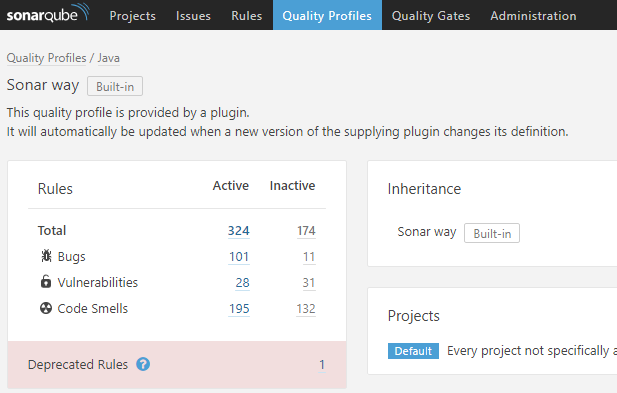
On met à jour les bindings, le plus simple pour le faire à côté de la console onglet **SonarLintBindings 🡪  
**

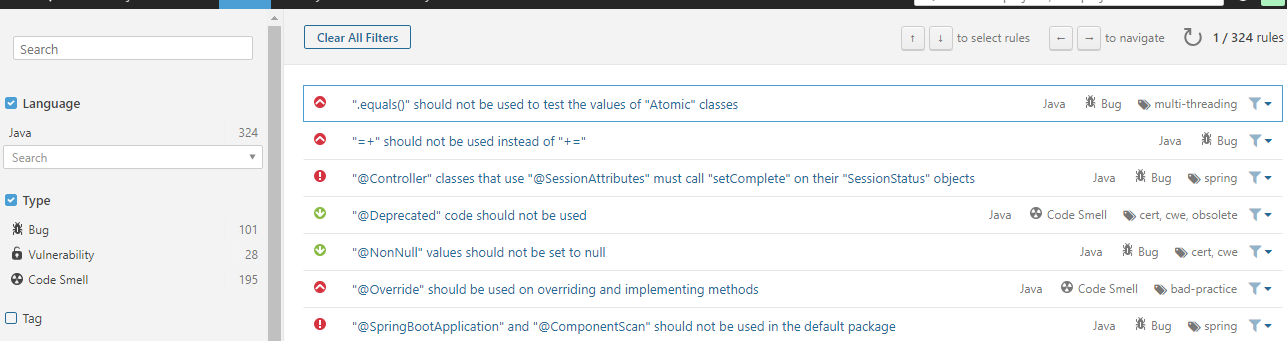
Nb : SonarQube vient avec une bdd embarquée, on peut gérer une la connexion avec une bdd perso dans le fichier de configurations de sonarqube (sonar.properties) :  


## Sonar et la qualité du code

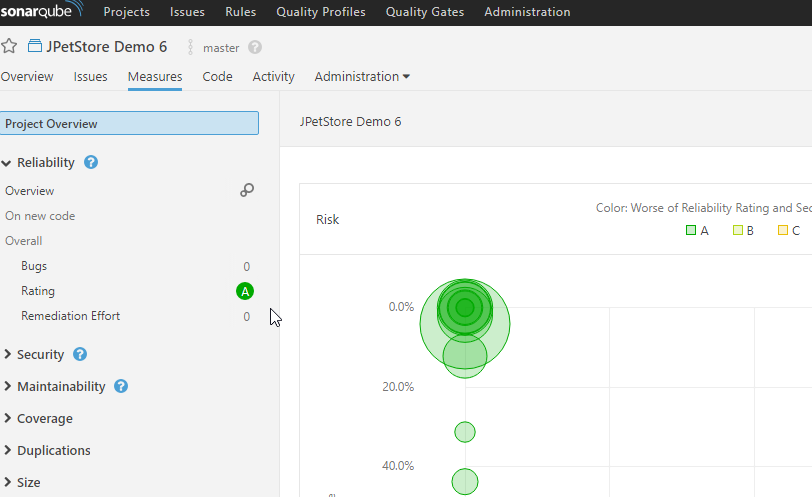
### Le profil de qualité

Les profils sont dispo dans Quality Profile (ruban en haut)



On peut voir le langage, le framework concerné, etc :  


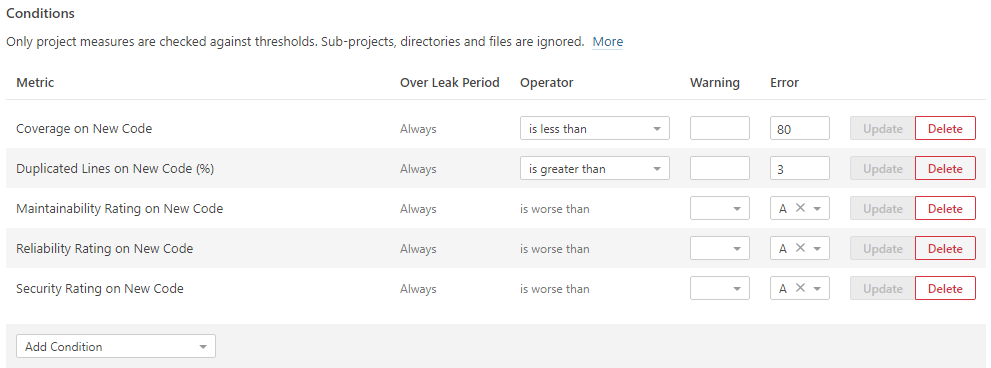
En cliquant sur le projet, on peut avoir des statistiques dessus, et une note. La **Remediation Effort** permet d’évaluer le temps pour monter à la note A.



### Quality gate

Quality Gate is the set of conditions the project must meet before it can be released into production.

C’est comme la couverture de tests minimum de 75% pour passer du code APEX en production dans SalesForce.



### Exercice : création et configuration d’un profil

1. Profil de règle JAVA hérité de sonar-way

* Ajouter 3 règles de votre choix
* Modifier le code JAVA pour introduire des violations
* Associer le nouveau profil au projet
* Relancer une analyse

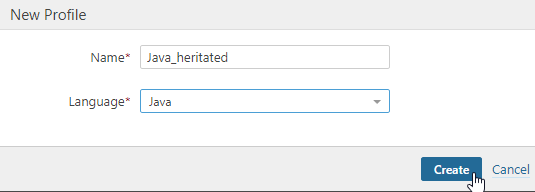
1. Quality gate

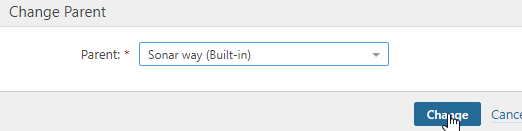
* Créer un nouveau quality gate avec pour but de provoquer un échec
* Associer le nouveau quality gate au projet
* Relancer une analyse

1. Ignorer des fichiers

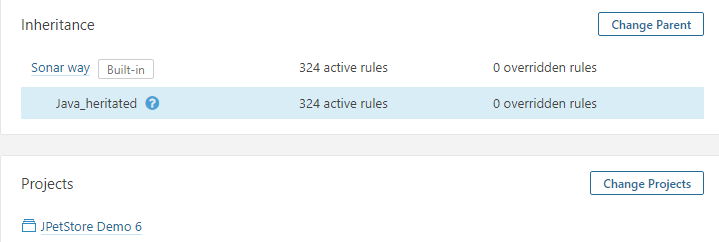
* Configurer le projet pour ignorer les fichiers contenant les problèmes
* Relancer une analyse qui devra être un succès

#### 1. Après s’être connecté en admin, création du profil.

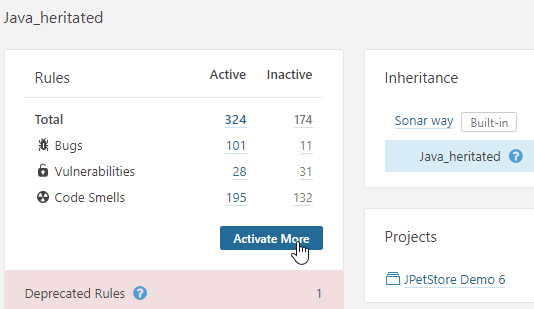


On associe au profil de base sonar-way :  


Les évolutions sonar-way impacteront ce profil automatiquement (màj).

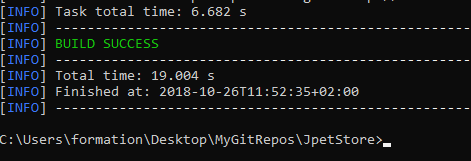
On add aussi le projet :  


#### Activer des règles en plus pour le profil :

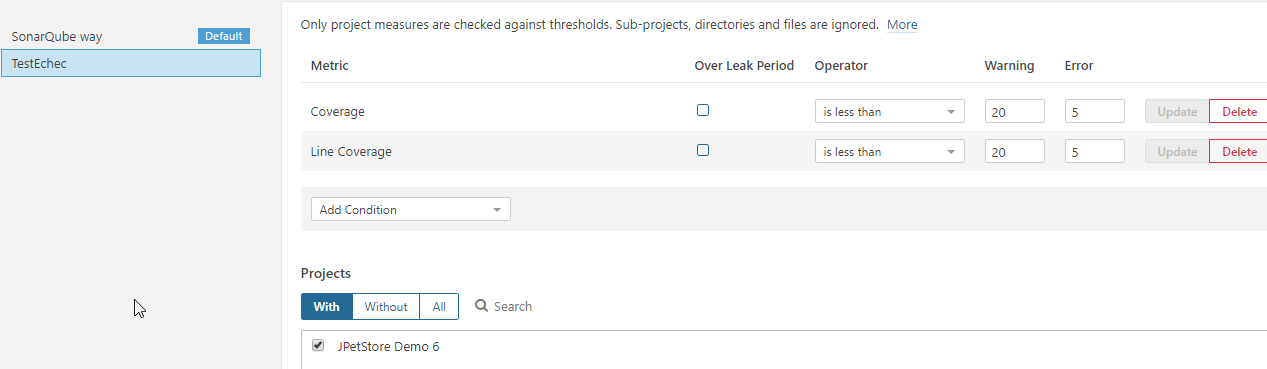


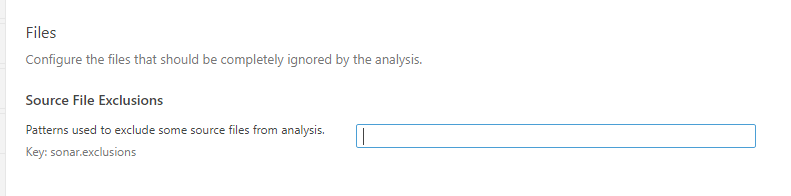
On choisit la règle :  

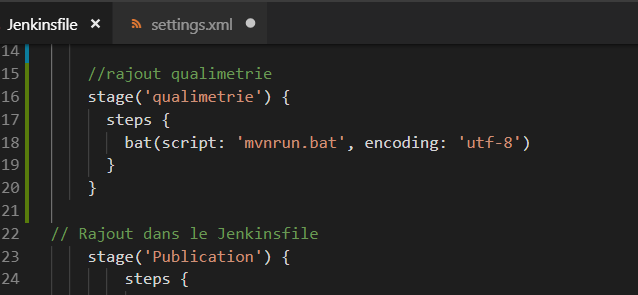

#### Lancer une analyse

Toujours pareil 🡪 cmd 🡪 cd « pathProjet » 🡪 mvn clean install sonar :sonar  


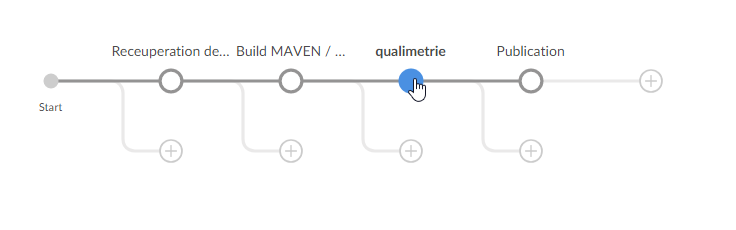
#### 2. Création nouveau profil quality gate

Le profil créé avec des conditions à l’arrache, parce que je le riz de veaux bien:  


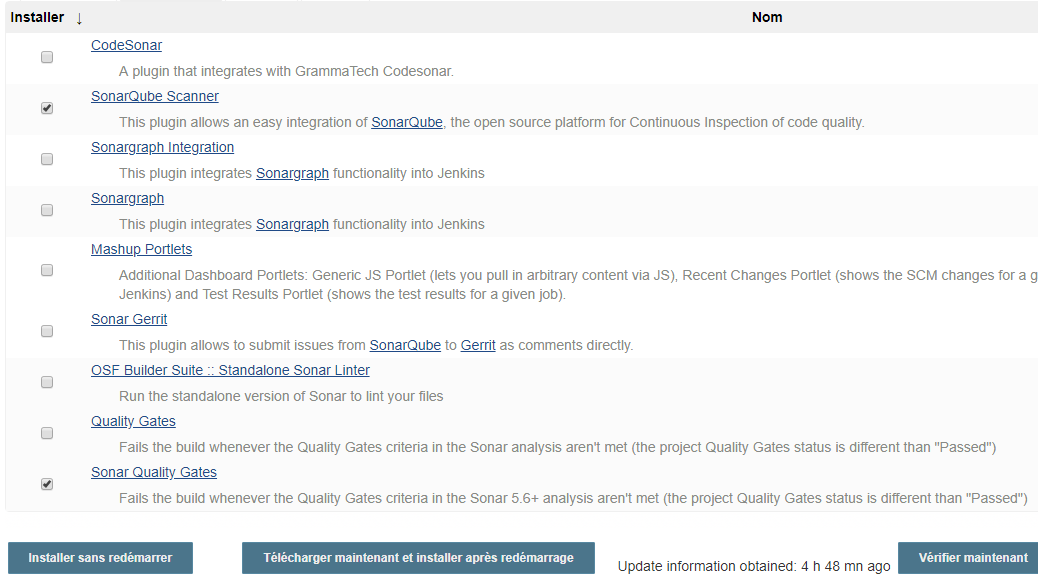
Dans l’onglet d’administration, on peut jouer sur les files à ignorer. Pour la syntaxe, google is your friend !  


On rajoute dans le pipeline l’étape de qualimétrie :  


Ensuite on doit pusher sur les www après avoir git add -A puis commit :  
git commit -m"modification du Jenkinsfile pour creer l etape de qualimetie"  
  
On push, puis HOP !  
Ca crée graphiquement les bayous

On a enfin le bail dans le Bifty !!  
  


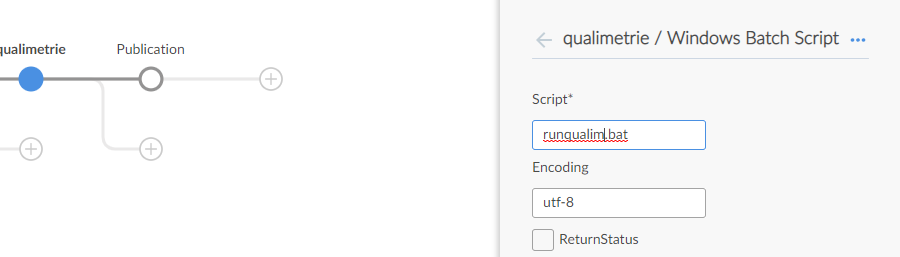
On va ensuite dans l’admin Jenkins tchecker s’il existe des p’tits plugins de grande qualité afin d’améliorer de façon significative notre expérience avec SonarQube.



On installe ces deux plugins :

* SonarQube Scanner : fournit un support pour le scanner Sonar, qu’on n’utilisera pas dans un premier temps car on build avec Maven ;
* Sonar Quality Gates

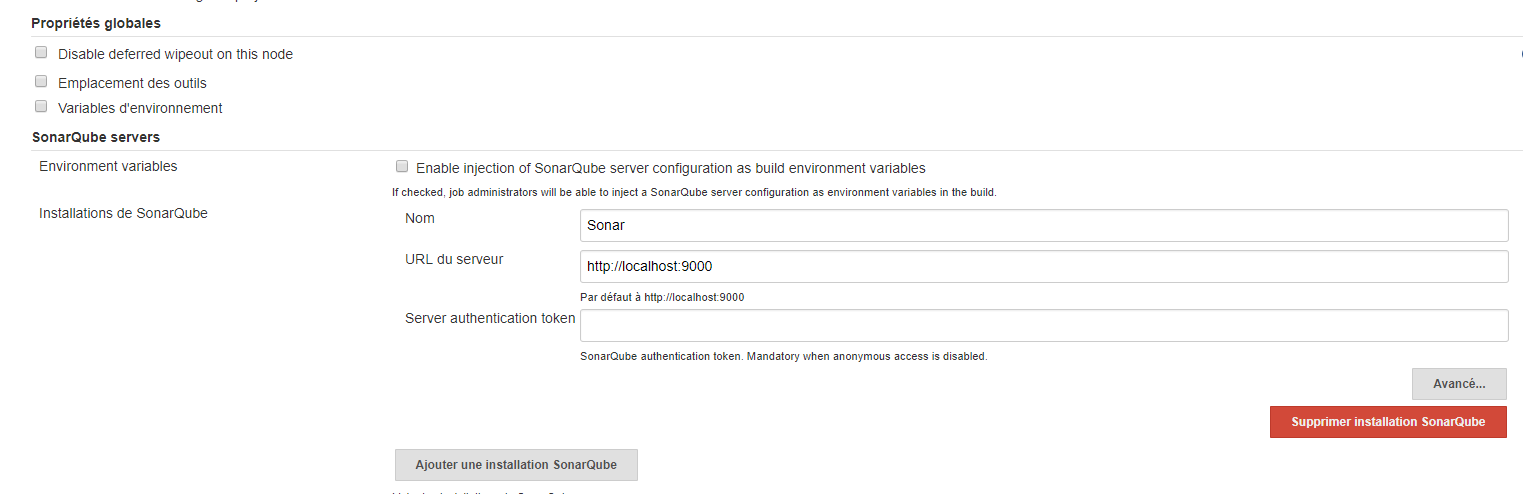
On se crée une étape de plus :



On va devoir créer le **runqualim.bat**

On doit ajouter au git, comme à l’époque du temps jadis, on push.

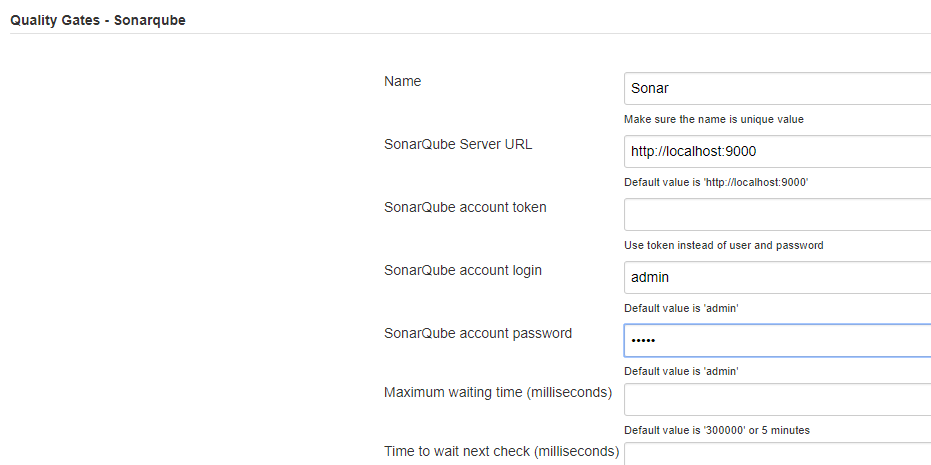
On ajoute Sonar dans la configuration de Jenkins



Il faut modifier le JenkinsFile pour prendre en compte les derniers changements.



On doit évidemment faire une tartine de Git pull, add, commit, push



## Jacoco – Java Code Coverage

Quand on démarre une JVM, on la démarre avec une option particulière, qui démarre un JaCoCo en même temps !  
C’est un espion qui va relever tous les appels ByteCode effectués 🡪 génère un jacoco.exec  
C’est donc un gros fichier de log, de tous les appels Bytecode.

L’utilisation la plus simple de JaCoCo est de rajouter l’option au démarrage :  
-javaagent:*[yourpath/]*jacocoagent.jar=*[option1]*=*[value1]*,*[option2]*=*[value2]*

clic ici <https://www.jacoco.org/jacoco/trunk/doc/agent.html>

On retrouve le fichier Jacoco.exec dans ce chemin :  
C:\Users\formation\.jenkins\workspace\JPetStore\_master-BLIMBXZFN3IOVONF6FPBZWLPCNJM75HFGJIHGT3BH6UT3GQBBMKA\target

Le projet doit donner ce qu’il faut à Sonar pour qu’il crée la couverture.

Bon là je dors, j’en peux plus c’est vendredi,  
En gros à revoir mais il faut déclarer dans le pom.xml du projet la version du plugin qu’on souhaite utiliser, et voir la doc pour rajouter les goals qu’on veut déclarer (pour qu’il sache quand il doit bosser).

## Pitest

Parcourt les classes et méthodes, rencontre des if i > j, va muter la classe en inversant le change de l’opérateur, en rajoutant un égal, les tests unitaires passent ensuite et si Pitest a muté le code et que les tests passent encore 🡪 les tests sont déclarés comme insuffisants !!

Ca se passe avec une intégration Maven : plugin Pitest qui génère dans le build des mutations, avant le passage des tests.