**Continuous integration:**

Developers practicing continuous integration merge their changes back to the main branch as often as possible. The developer's changes are validated by creating a build and running automated tests against the build. By doing so, you avoid the integration hell that usually happens when people wait for release day to merge their changes into the release branch.

Continuous integration puts a great emphasis on testing automation to check that the application is not broken whenever new commits are integrated into the main branch.

**Continuous delivery:**

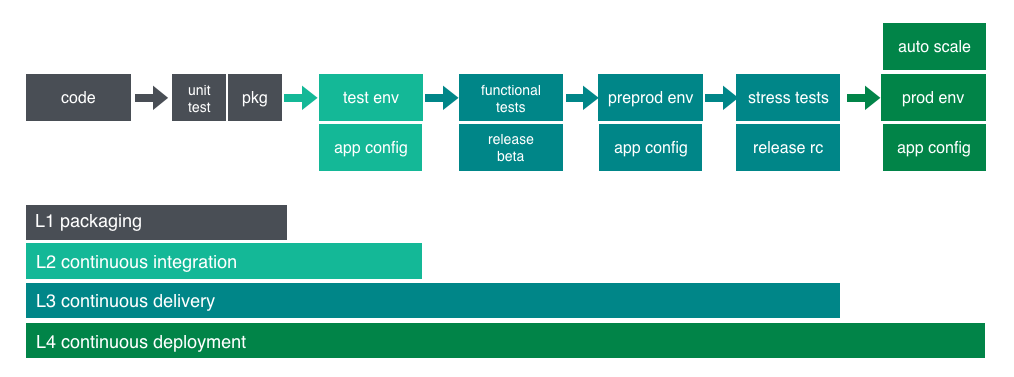
Continuous delivery is an extension of continuous integration to make sure that you can release new changes to your customers quickly in a sustainable way. This means that on top of having automated your testing, you also have automated your release process and you can deploy your application at any point of time by clicking on a button.

In theory, with continuous delivery, you can decide to release daily, weekly, fortnightly, or whatever suits your business requirements. However, if you truly want to get the benefits of continuous delivery, you should deploy to production as early as possible to make sure that you release small batches, that are easy to troubleshoot in case of a problem.

**Continuous deployment:**

Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a Release Day anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.



**What is Jenkins?**

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

**Prerequisites:**

Minimum hardware requirements:

256 MB of RAM

1 GB of drive space (although 10 GB is a recommended minimum if running Jenkins as a Docker container)

**Recommended hardware configuration for a small team:**

1 GB+ of RAM

20 GB+ of drive space

**Software requirements:**

Java 8 - either a Java Runtime Environment (JRE) or a Java Development Kit (JDK) is fine.

In this tutorial we install Jenkins on CentOS 7:

**Install JAVA:**

* Install jdk8 RPM package from the following URL

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

* By using winscp we can copy the RPM package(jdk-8u151-linux-x64.rpm) to the centos machine.
* Now, login as a root to the centos machine and check for the execution permissions of the rpm package

[root ~] ls -l jdk-8u151-linux-x64.rpm

If the execution permissions are not there change with

[root ~] chmod +x jdk-8u151-linux-x64.rpm

* Now, execute the rpm package to install jdk on the machine

[root ~] rpm -Uvh jdk-8u151-linux-x64.rpm

* Once the jdk is installed successfully, now set up the java path as

[root~] alternatives --install /usr/bin/java java /usr/java/latest/bin/java 200000

[root~] alternatives --install /usr/bin/javac javac /usr/java/latest/bin/javac 200000

[root~] alternatives --install /usr/bin/jar jar /usr/java/latest/bin/jar 200000

* Now, export the path in

[root~] nano /etc/rc.local

Add the following line at the end of the file

export JAVA\_HOME="/usr/java/latest"

* Now, check the java with the java, javac command

**Install Jenkins:**

[root~] wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo

[root~] rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key

[root~] yum install -y jenkins

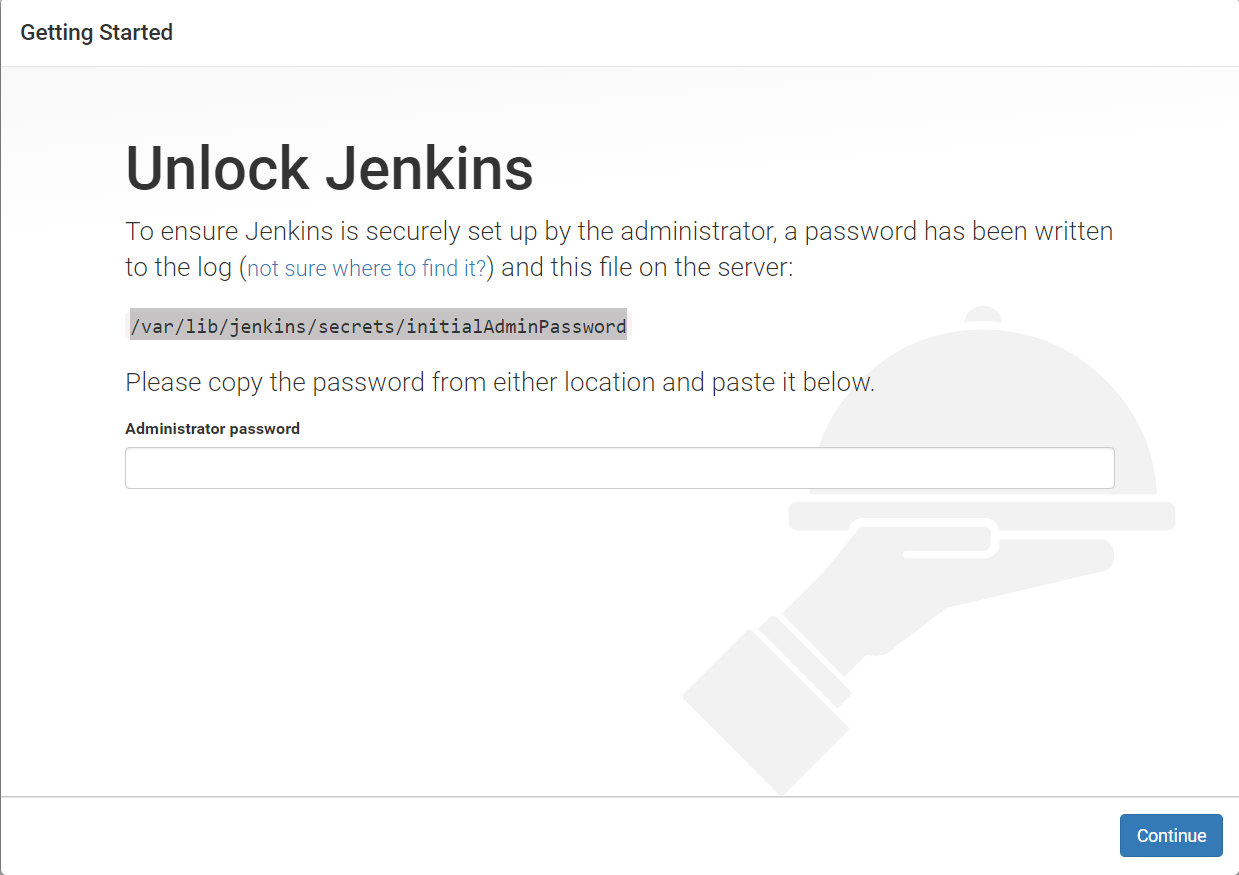
[root~] systemctl start jenkins

[root~] systemctl enable Jenkins

After the Jenkins installation process go to the URL as,

http://<machine-ip>:8080

It shows the Jenkins start up page as,



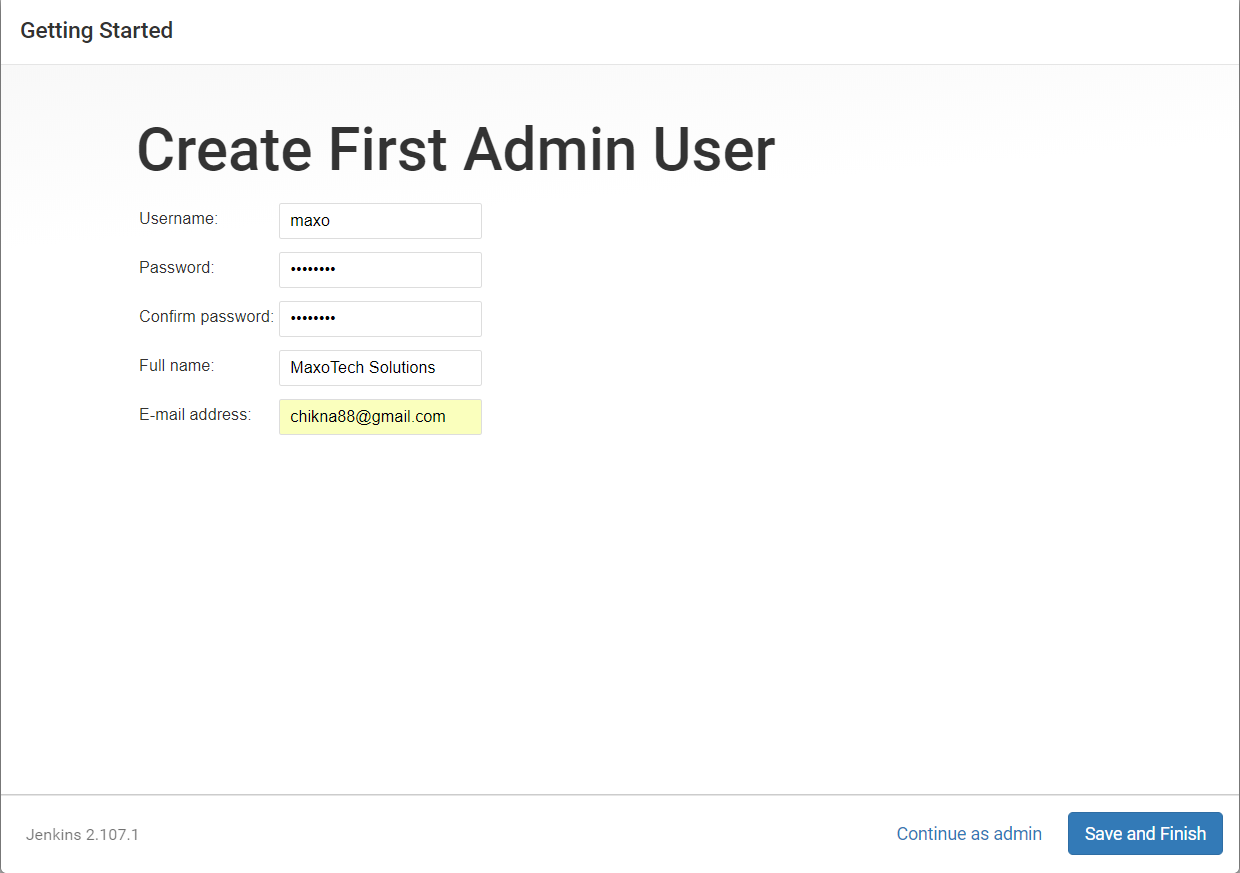
To get the initial password type the following command in the terminal and enter the administrator password here

[root~] cat /var/lib/jenkins/secrets/initialAdminPassword

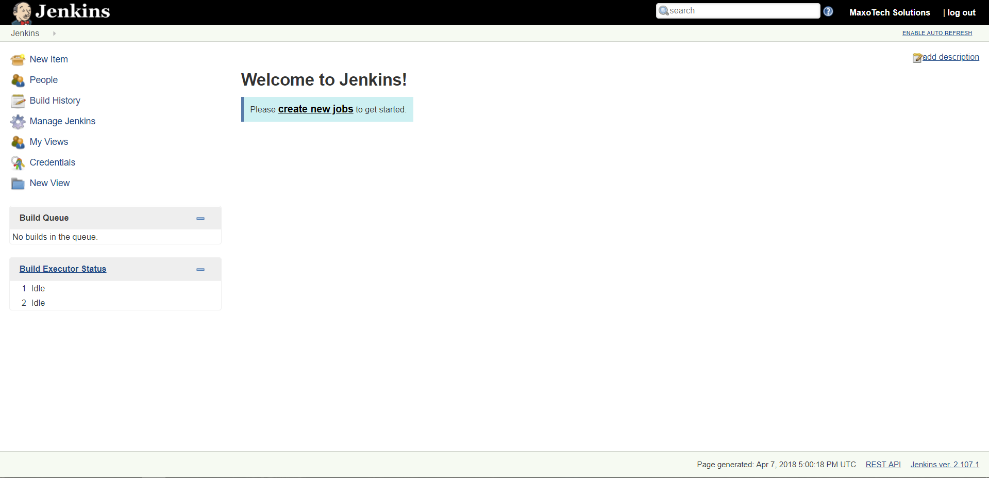
In the next you get the options to install the plugins, select install suggested plugins



Once all the plugins are installed, create the first admin user in the next page



Now, you can start using the Jenkins



**Adding a slave node:**

* First in the Jenkins master terminal, change the user as Jenkins with

[root~] su Jenkins -s /bin/bash

[jenkins~] ssh-keygen

* Then copy the generated key to the remote machine with

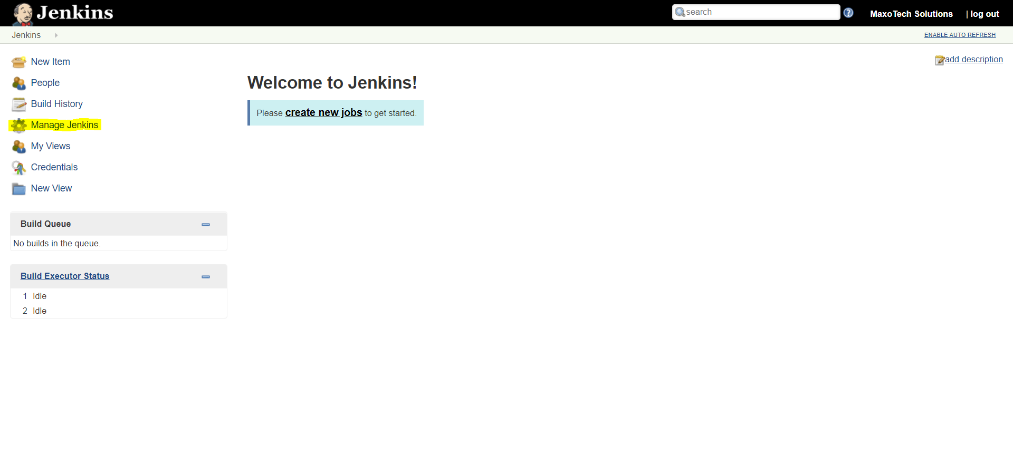
[root~] ssh-copy-id user@<remote-ip>

Enter the password.

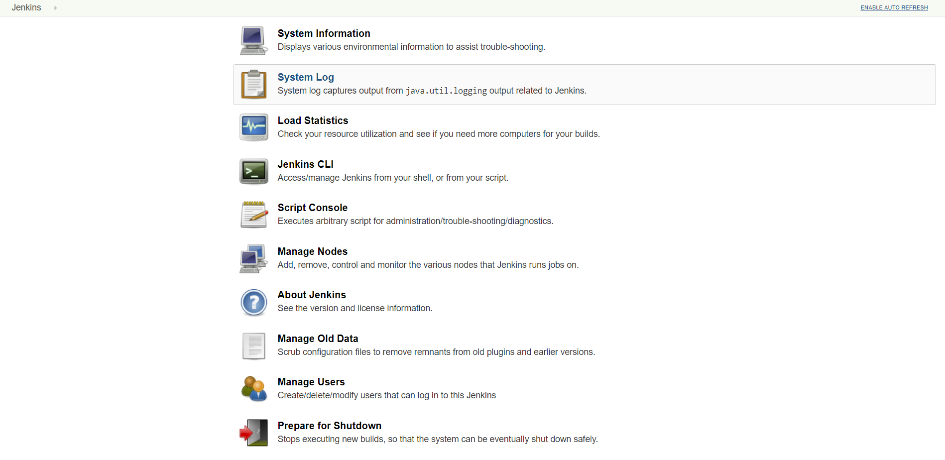
To check the authentication ssh user@<remote-ip>

* Install java and setup the path
* Once the authentication is done go to Jenkins home page and click on manage Jenkins

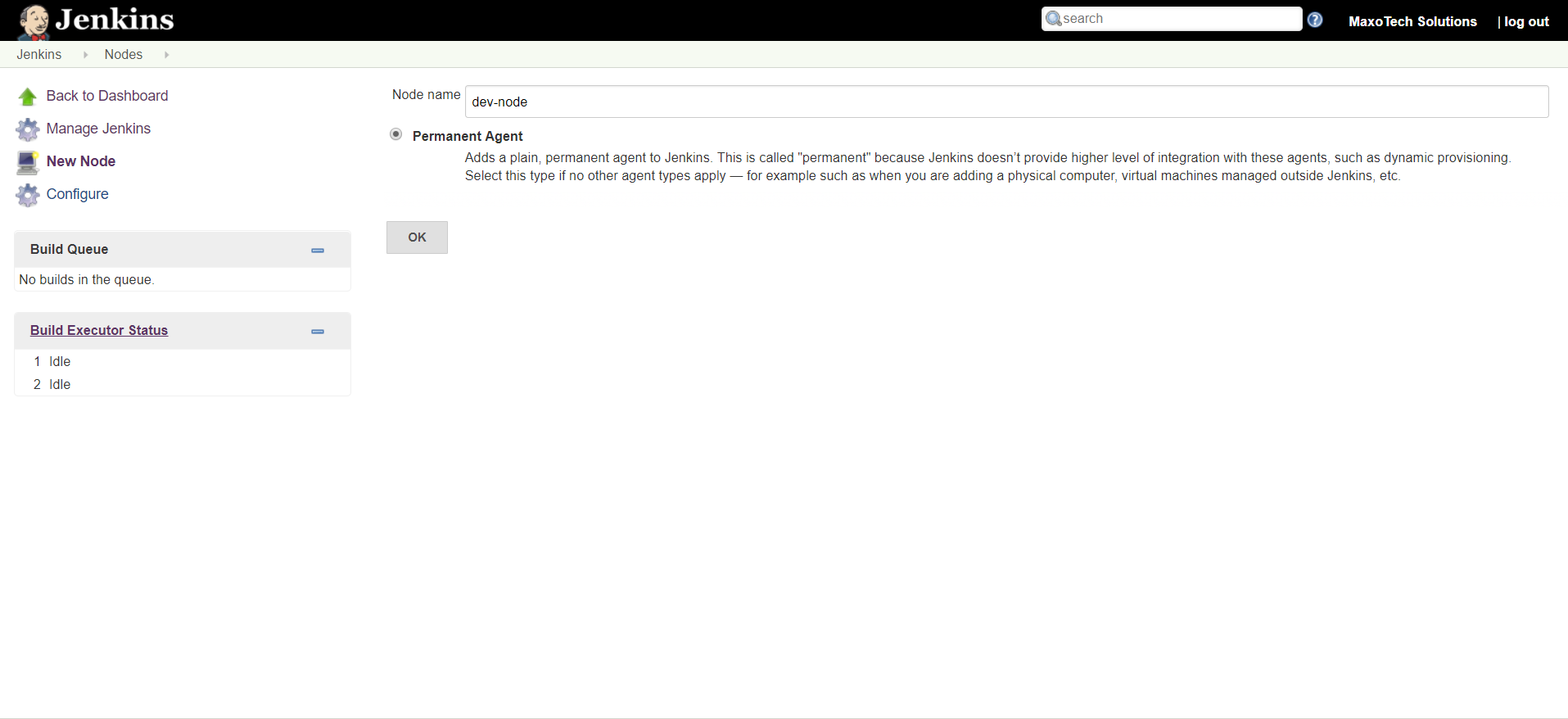
Go to the Jenkins page and click on Manage Jenkins tab



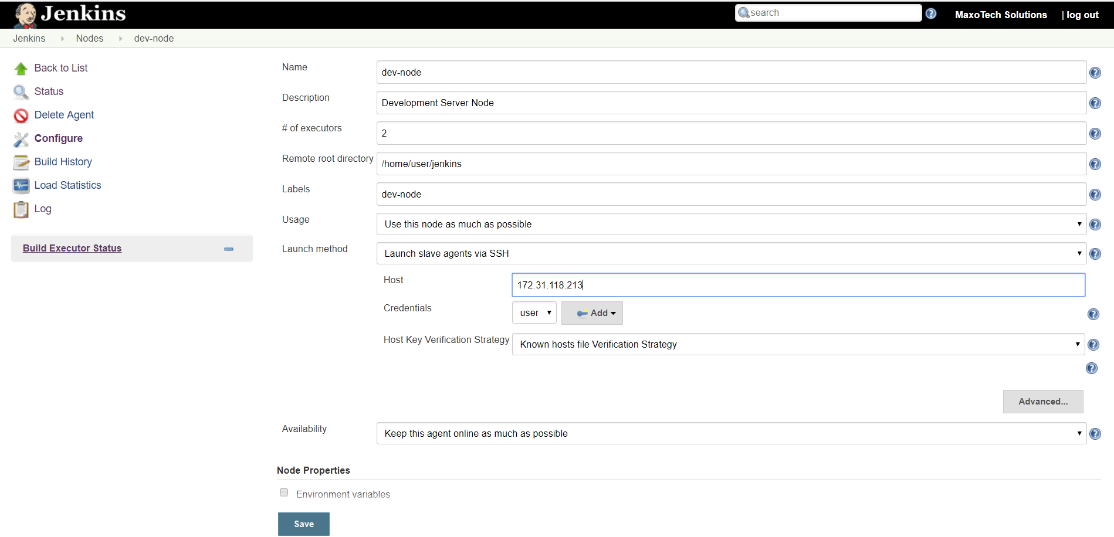
In the manage Jenkins tab you will find the manage nodes option



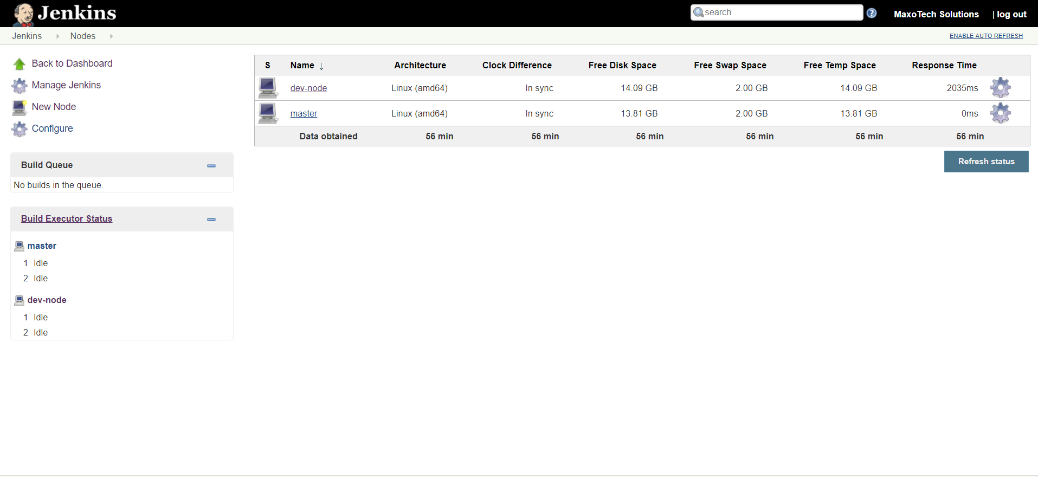
Now, click on the New Node on the left side and enter the name of the node, select as a permanent agent.



Configure the node with the details and save the node (make sure you have the remote root directory available in node)



Now, you can see the node is connected.



* To download the source code of the project we use the version control system Git in slave node

Install git with the command,

[root~] yum install -y git

* For the project we will download the source code from the following GitHub link,

<https://github.com/vemular1/jenkins-maven-pipeline.git>

[root~] git clone <https://github.com/vemular1/jenkins-maven-pipeline.git>

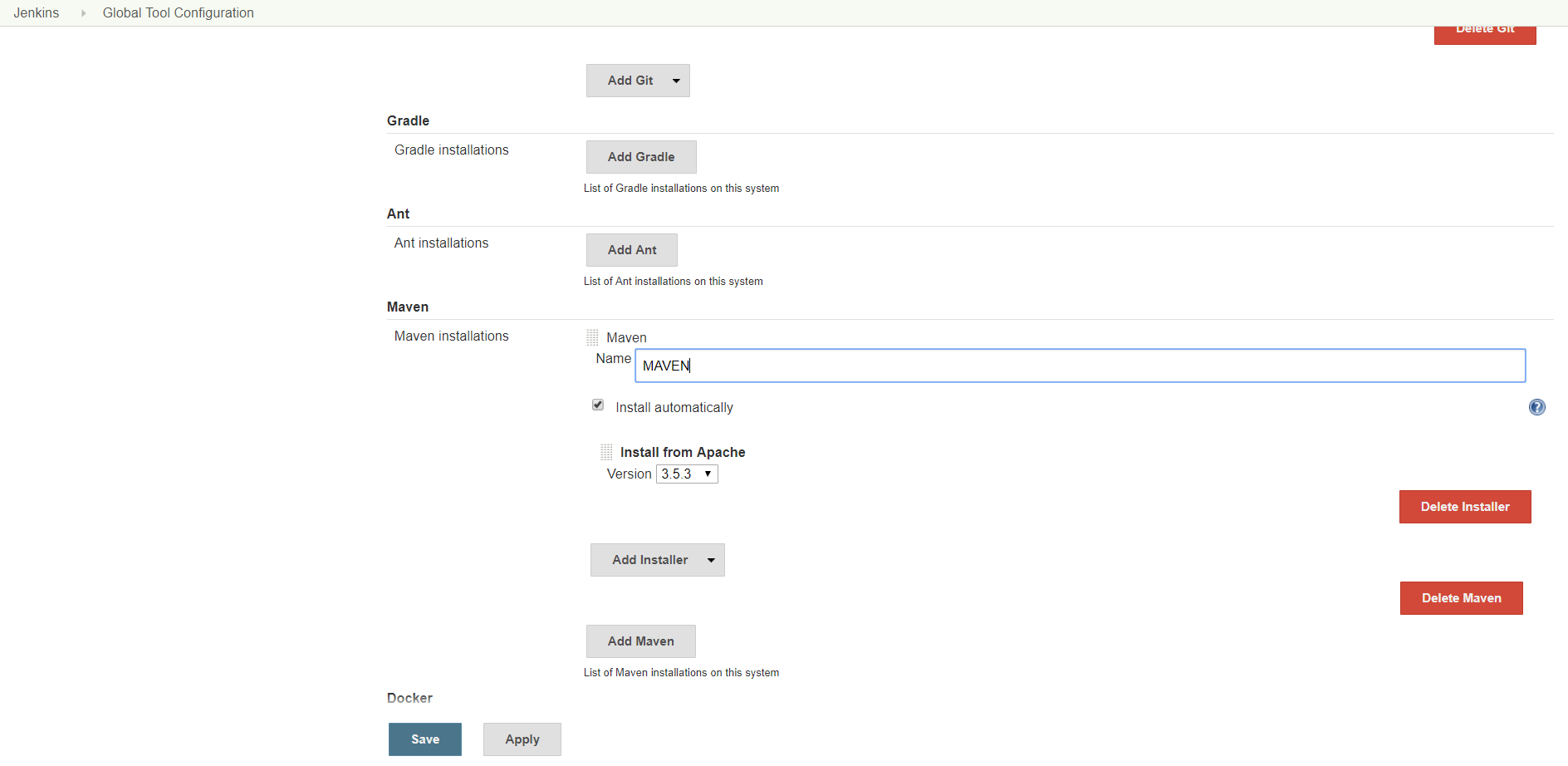
To create a pipeline, we need to configure SonarQube, Nexus and Maven with Jenkins

**Configure Maven:**

To setup the Maven, go to Jenkins home page click on Manage Jenkins->Global Tool Configuration->Maven->Add Maven

Here, you have the option to choose you maven installation automatically or you manually configure the existing maven installation path.

Select “Install Automatically” and apply, save it



Now we need to set up the Sonar Qube for code analysis.

**Configure SonarQube:**

**SonarQube is an open source tool for quality system development. It is written in Java and supports multiple databases. It provides capabilities to continuously inspect code, show the health of an application, and highlight newly introduced issues. It contains code analyzers which are equipped to detect tricky issues. It also integrates easily with DevOps.**

Prerequisites

* A Vultr 64-bit CentOS 7 server instance with at least 2 GB RAM.
* A sudo user.
* Install and configure Java

Install PostgreSQL:

* Install PostgreSQL repository by typing:

sudo rpm -Uvh https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-7-x86\_64/pgdg-centos96-9.6-3.noarch.rpm

* Install PostgreSQL database server by running:

sudo yum -y install postgresql96-server postgresql96-contrib

* Initialize the database:

sudo /usr/pgsql-9.6/bin/postgresql96-setup initdb

* Edit the /var/lib/pgsql/9.6/data/pg\_hba.conf to enable MD5-based authentication.

sudo nano /var/lib/pgsql/9.6/data/pg\_hba.conf

* Find the following lines and change peer to trust and idnet to md5.

# TYPE DATABASE USER ADDRESS METHOD

# "local" is for Unix domain socket connections only

local all all peer

# IPv4 local connections:

host all all 127.0.0.1/32 ident

# IPv6 local connections:

host all all ::1/128 ident

* Once updated, the configuration should look like the one shown below.

# TYPE DATABASE USER ADDRESS METHOD

# "local" is for Unix domain socket connections only

local all all trust

# IPv4 local connections:

host all all 127.0.0.1/32 md5

# IPv6 local connections:

host all all ::1/128 md5

* Start PostgreSQL server and enable it to start automatically at boot time by running:

sudo systemctl start postgresql-9.6

sudo systemctl enable postgresql-9.6

* Change the password for the default PostgreSQL user.

sudo passwd postgres

* Switch to the postgres user.

su - postgres

* Create a new user by typing:

createuser sonar

* Switch to the PostgreSQL shell.

psql

* Set a password for the newly created user for SonarQube database.

ALTER USER sonar WITH ENCRYPTED password 'StrongPassword';

* Create a new database for PostgreSQL database by running:

CREATE DATABASE sonar OWNER sonar;

* Exit from the psql shell:

\q

* Switch back to the sudo user by running the exit command.

Download and configure SonarQube

* Download the SonarQube installer files archive.

wget https://sonarsource.bintray.com/Distribution/sonarqube/sonarqube-6.4.zip

You can always look for the link to the latest version of the application on the SonarQube [download page](https://www.sonarqube.org/downloads/).

* Install unzip by running:

sudo yum -y install unzip

* Unzip the archive using the following command.

sudo unzip sonarqube-6.4.zip -d /opt

* Rename the directory:

sudo mv /opt/sonarqube-6.4 /opt/sonarqube

* Open the SonarQube configuration file using your favorite text editor.

sudo nano /opt/sonarqube/conf/sonar.properties

* Find the following lines.

#sonar.jdbc.username=

#sonar.jdbc.password=

* Uncomment and provide the PostgreSQL username and password of the database that we have created earlier. It should look like:

sonar.jdbc.username=sonar

sonar.jdbc.password=StrongPassword

* Next, find:

#sonar.jdbc.url=jdbc:postgresql://localhost/sonar

* Uncomment the line, save the file and exit from the editor.

Configure Systemd service

* SonarQube can be started directly using the startup script provided in the installer package. As a matter of convenience, you should setup a Systemd unit file for SonarQube.

sudo nano /etc/systemd/system/sonar.service

* Populate the file with:

[Unit]

Description=SonarQube service

After=syslog.target network.target

[Service]

Type=forking

ExecStart=/opt/sonarqube/bin/linux-x86-64/sonar.sh start

ExecStop=/opt/sonarqube/bin/linux-x86-64/sonar.sh stop

User=root

Group=root

Restart=always

[Install]

WantedBy=multi-user.target

* Start the application by running:

sudo systemctl start sonar

* Enable the SonarQube service to automatically start at boot time.

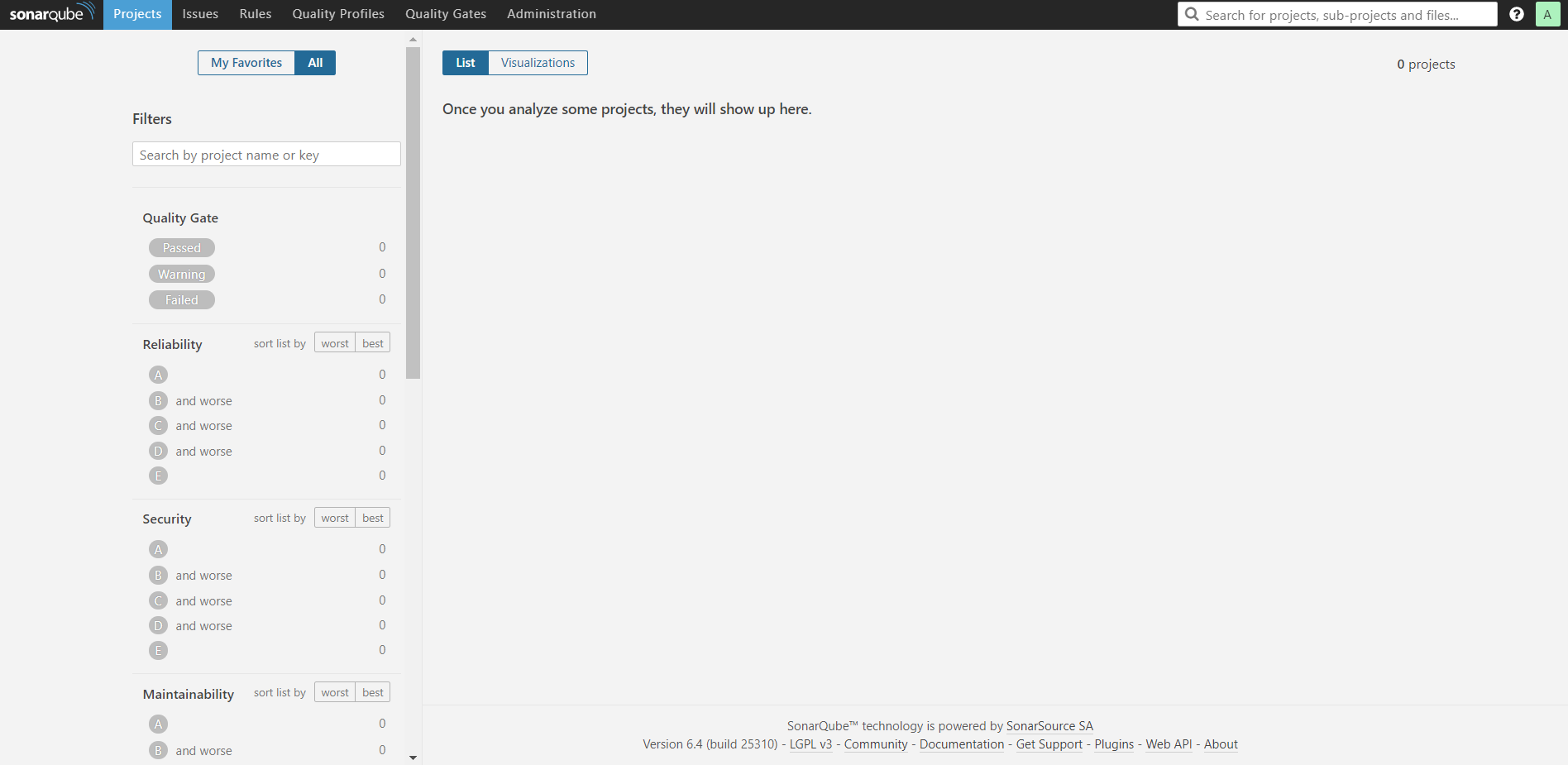
sudo systemctl enable sonar

* To check if the service is running, run:

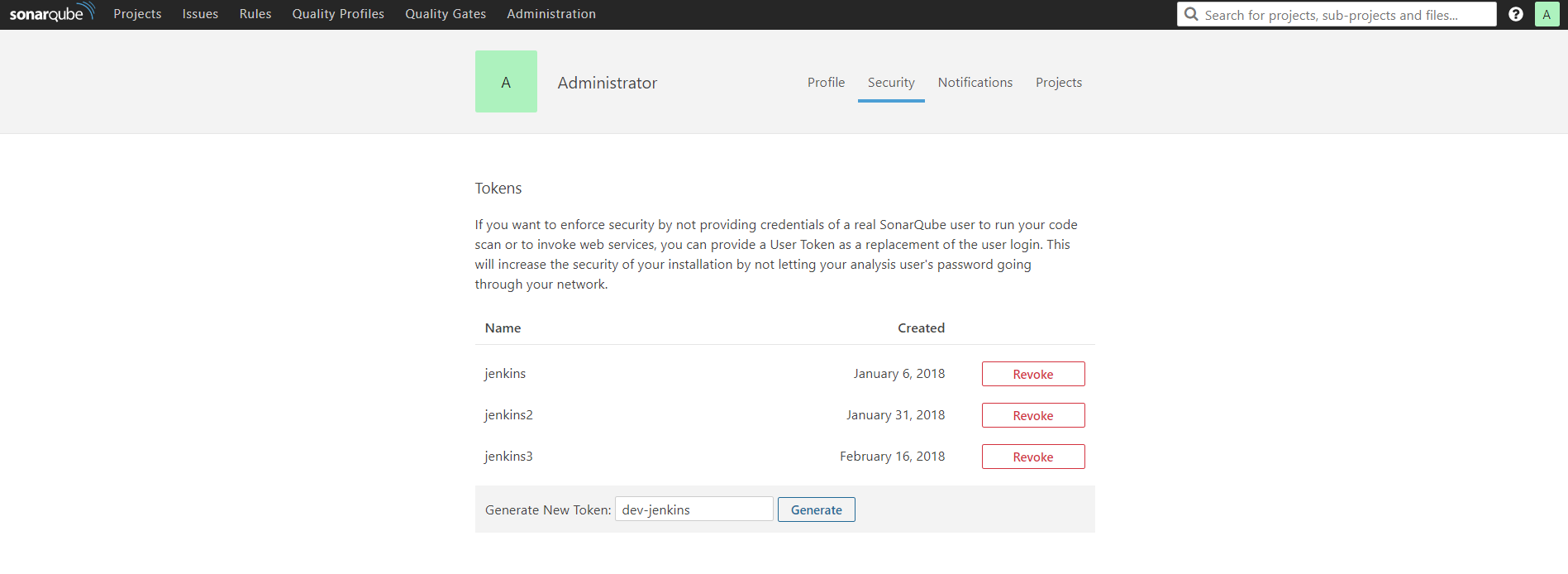
sudo systemctl status sonar

* SonarQube is installed on your server, access the dashboard at the following address.

http://<server-ip>:9000

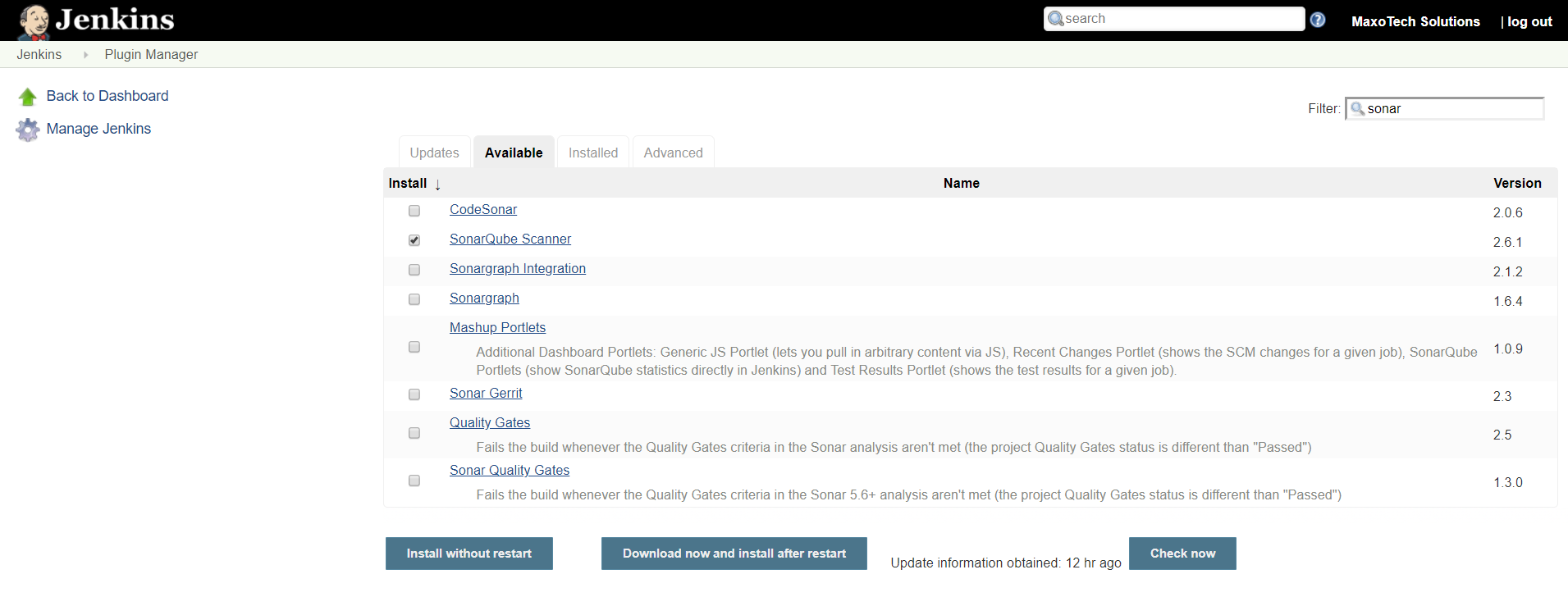


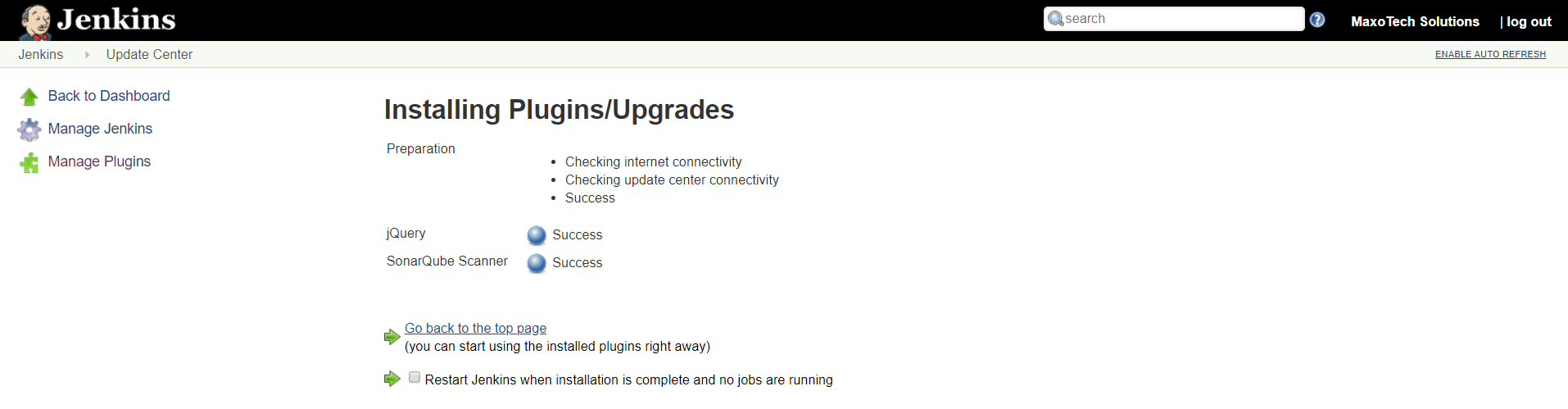
* To integrate SonarQube with Jenkins we need to generate the authentication token from My Account -> Security and generate a new token.



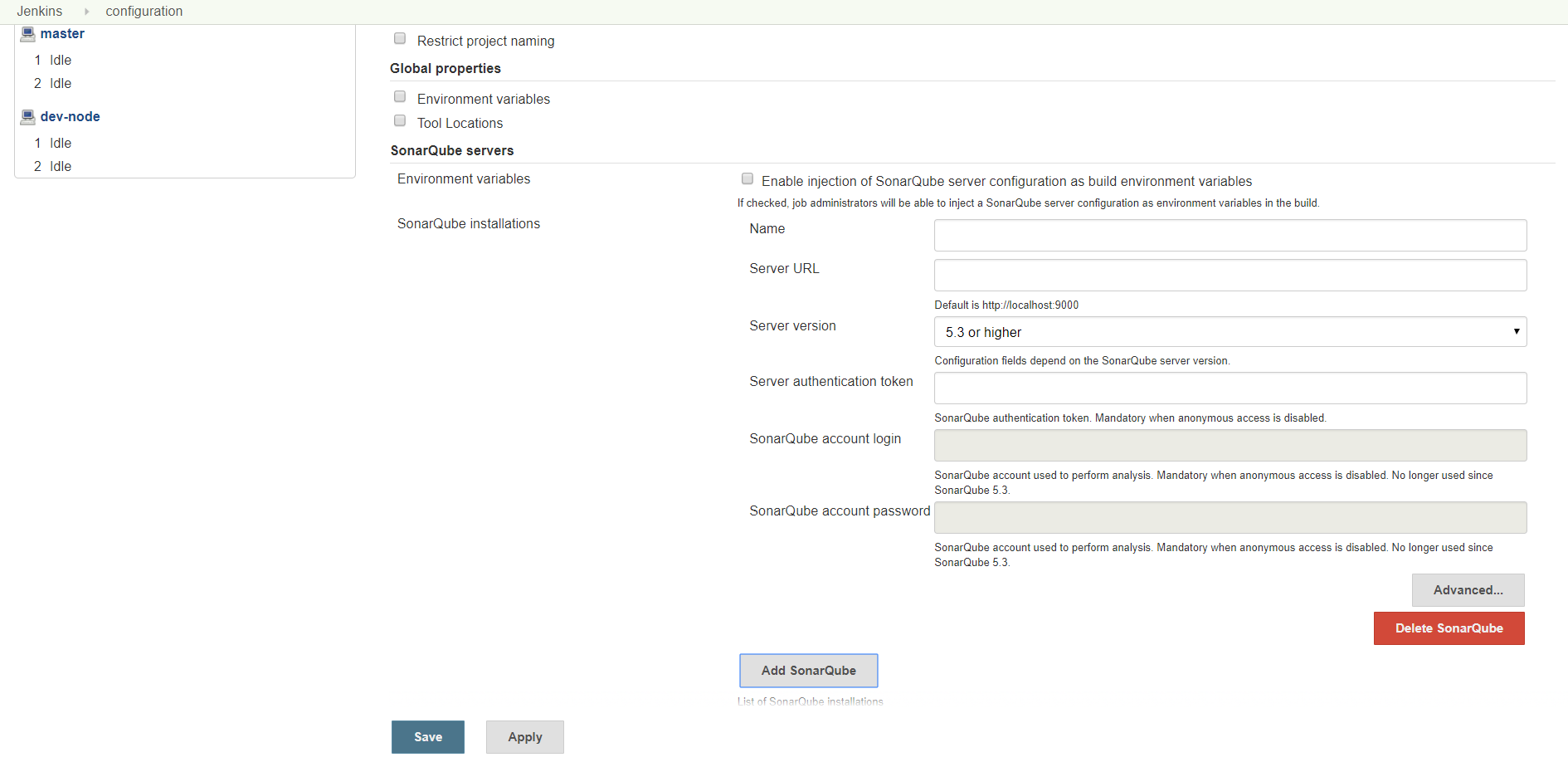
* Now copy the token and go to the Jenkins home page and click on Manage Jenkins -> Manage Plugins -> Available

Type sonar qube scanner plugin and install without restart

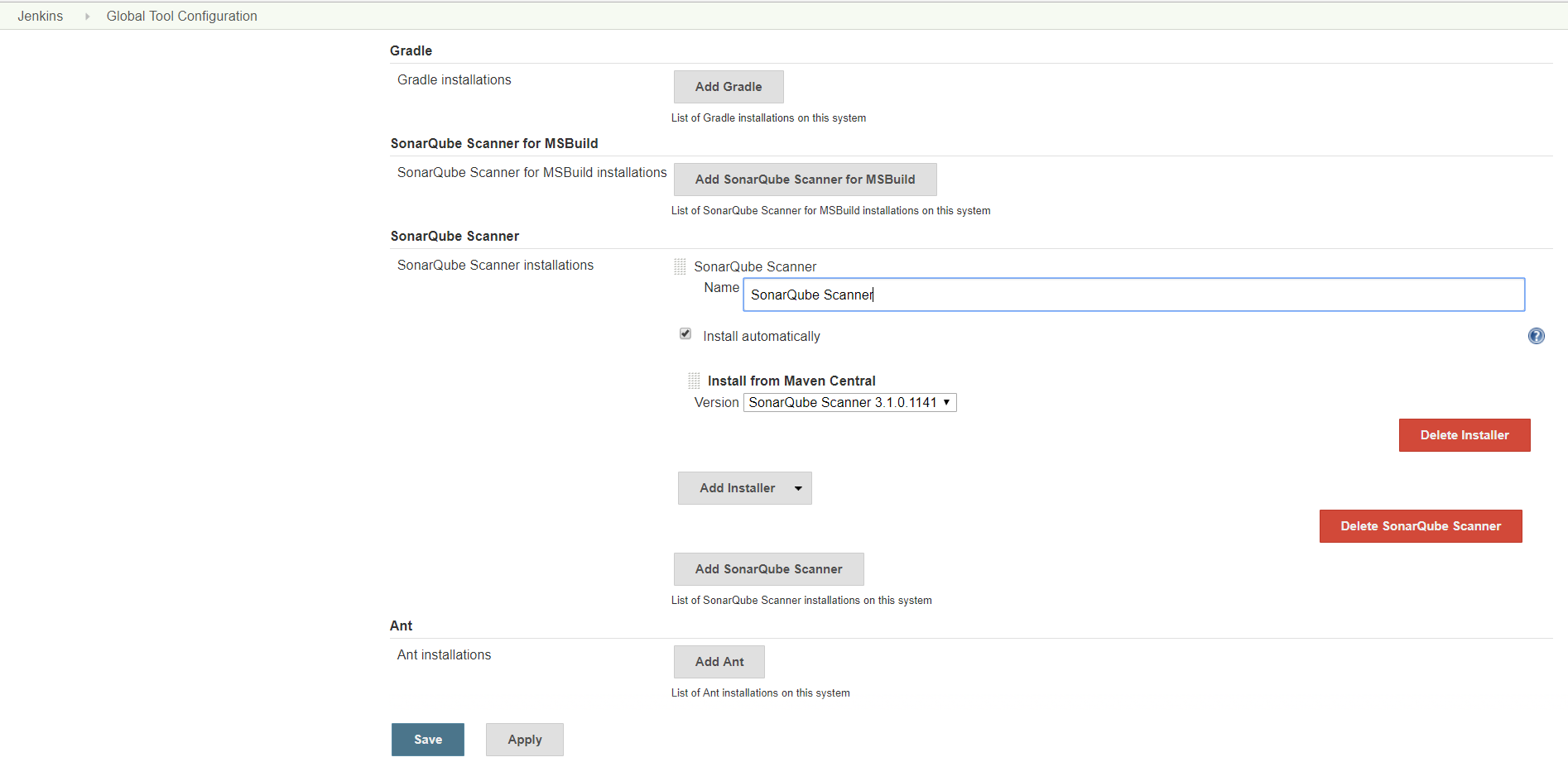




* Configure the SonarQube with Jenkins



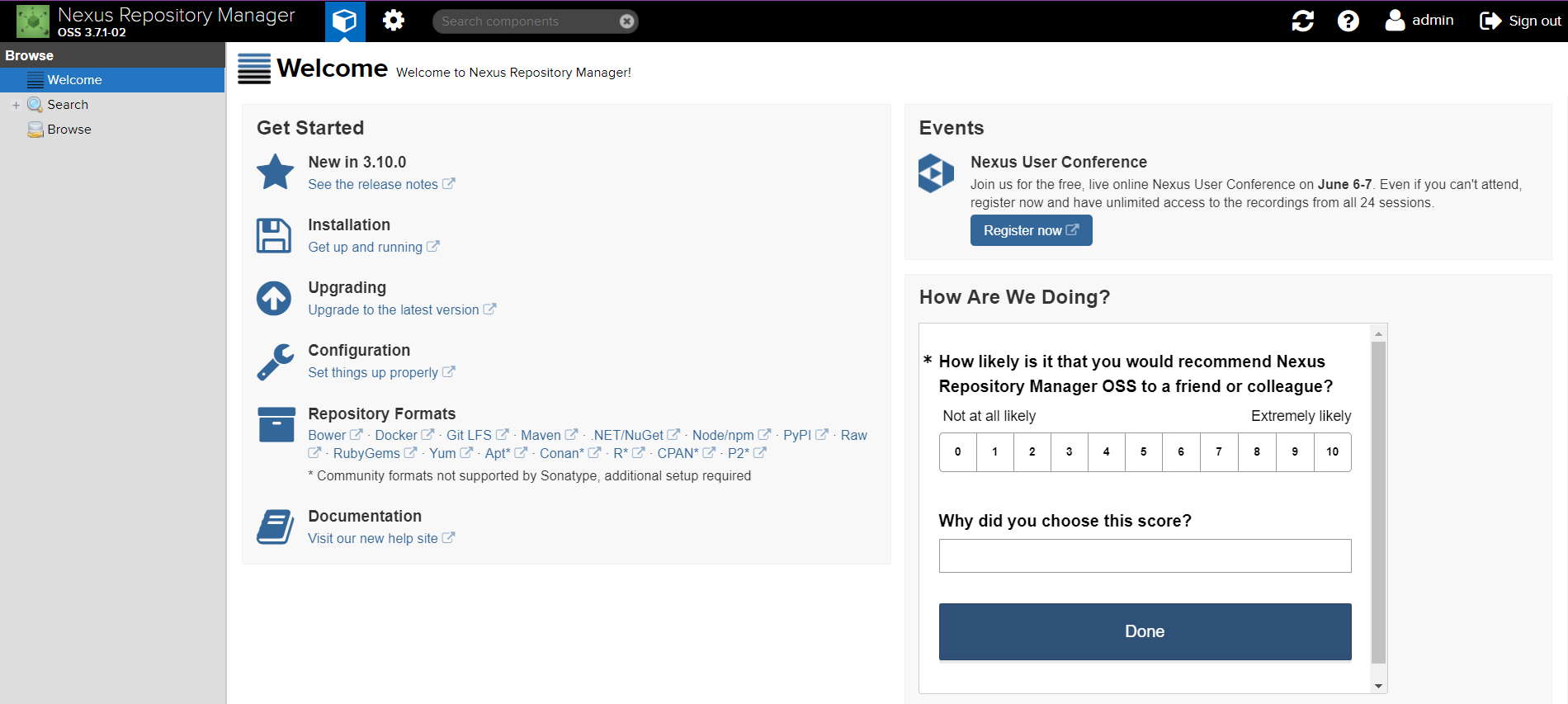
* Add the SonarQube Scanner Manage Jenkins -> Global Tool Configuration ->Add SonarQube Scanner with Install Automatically or Configure manually.



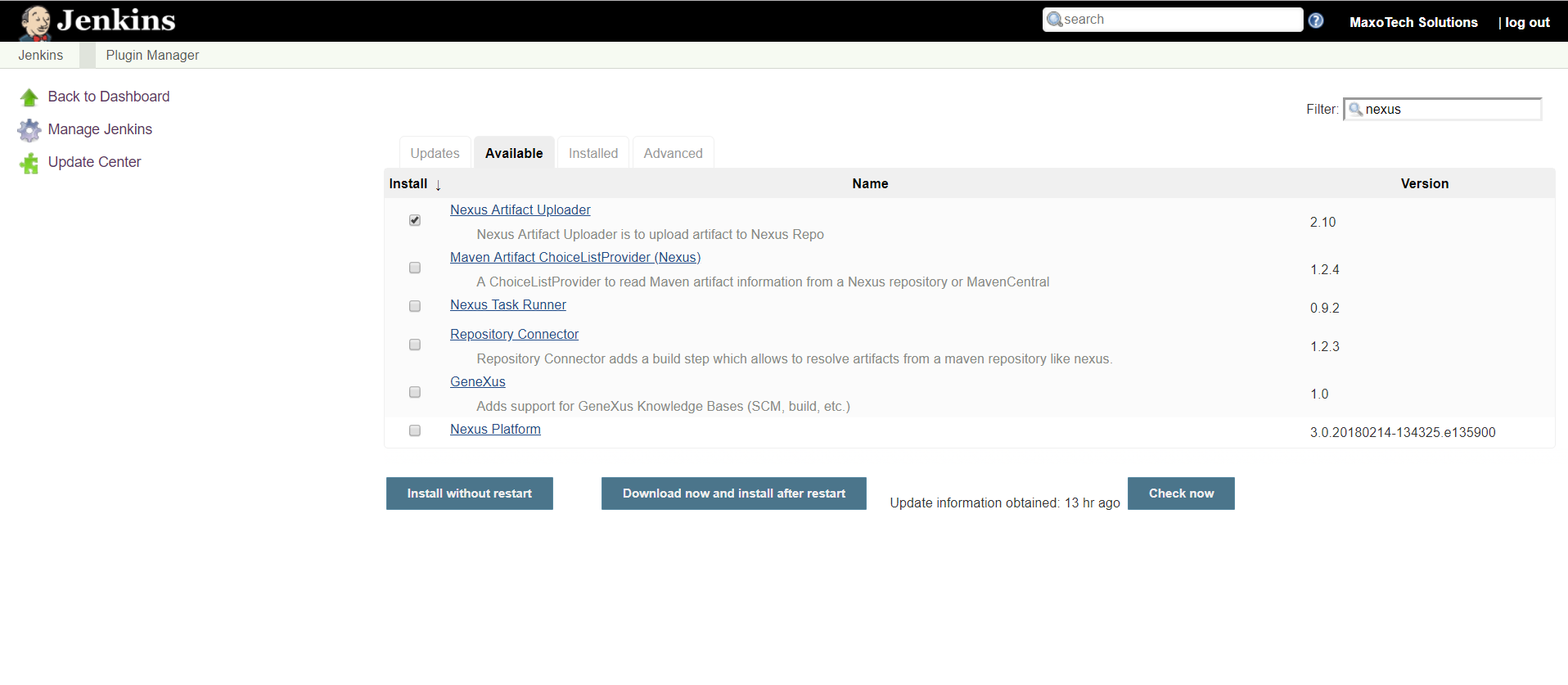
* Install Nexus from the following URL

<https://devopscube.com/how-to-install-latest-sonatype-nexus-3-on-linux/>

* Nexus Home Page



To integrate nexus with Jenkins, install the nexus plugin Manage Jenkins -> Manage Plugins -> Available, search for Nexus



* Once it is installed, configure the Nexus in the Jenkins server by editing the file in the location

sudo nano /var/lib/jenkins/tools/hudson.tasks.Maven\_MavenInstallation/MAVEN/conf/settings.xml

add the following lines before the </servers> tag, (watch out for <!-- -- >)

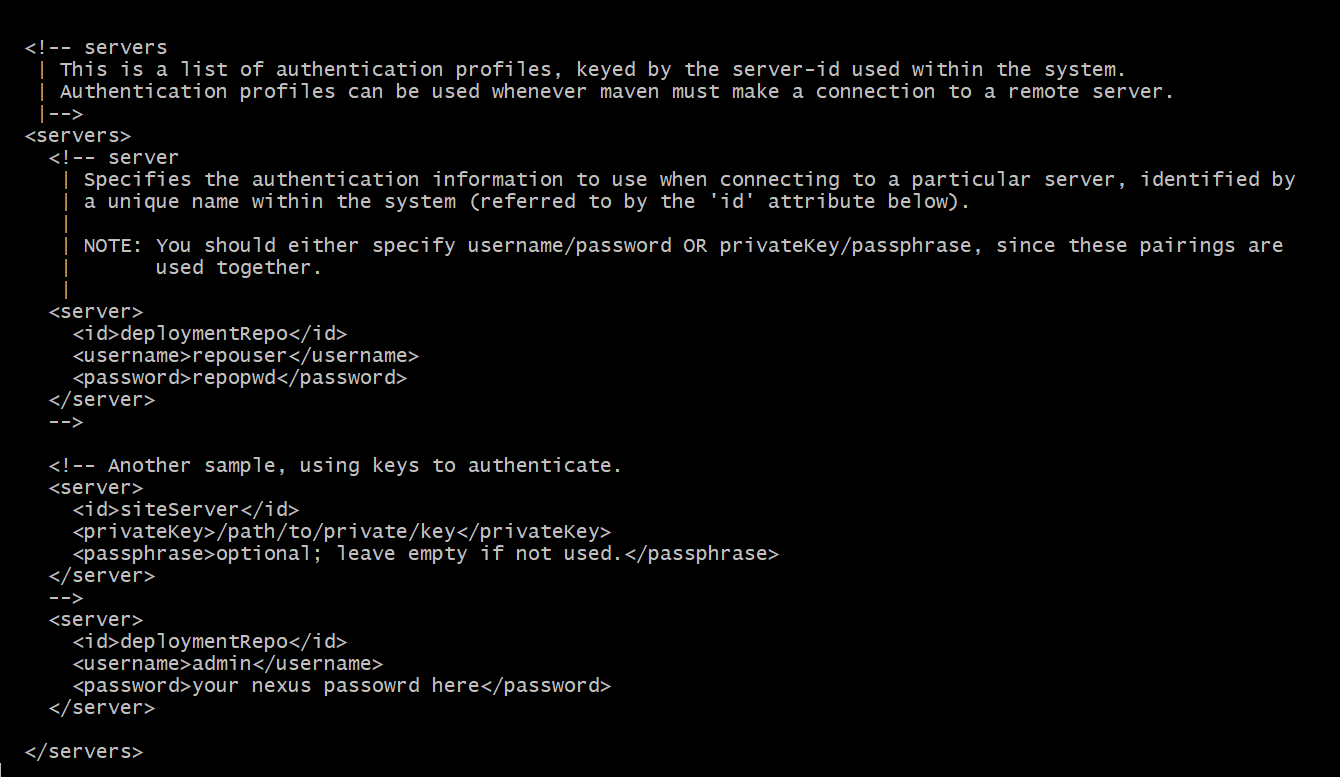
<server>

<id>deploymentRepo</id>

<username>admin</username>

<password>your nexus password here</password>

</server>



* Now edit your POM.xml file to add the distribution management to upload the artifacts to NEXUS.

<distributionManagement>

<repository>

<id>deploymentRepo</id>

<name>Internal Releases</name>

<url>http://<nexus-ip>:8081/repository/maven-releases/</url>

</repository>

<snapshotRepository>

<id>deploymentRepo</id>

<name>Internal Releases</name>

<url>http://<nexus-ip>:8081/repository/maven-snapshots/</url>

</snapshotRepository>

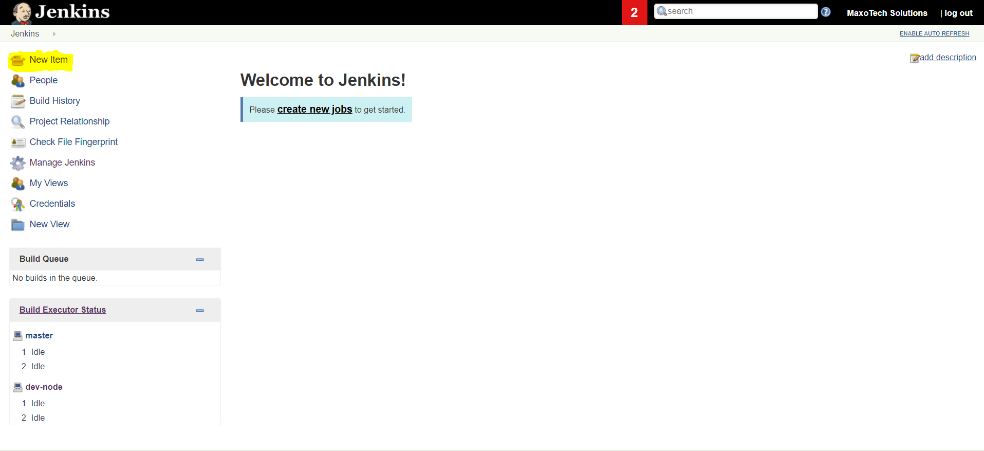
</distributionManagement>

**Jenkins Build Pipeline Jobs:**

**GIT 🡪 SONARQUBE 🡪 MAVEN 🡪 JUNIT 🡪 JAR 🡪 NEXUS**

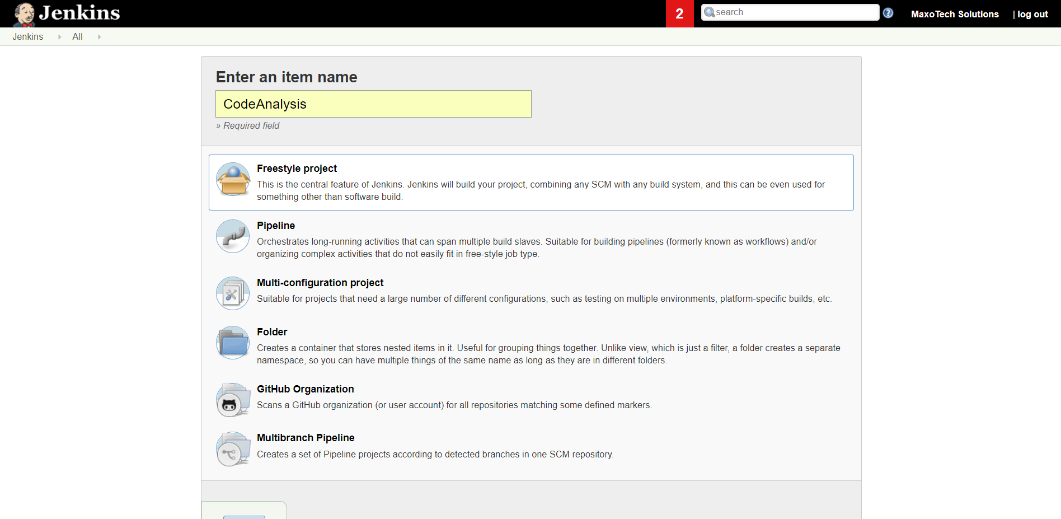
In the Jenkins console create your first job with Code Analysis and configure the steps as below.

* First click on the new item



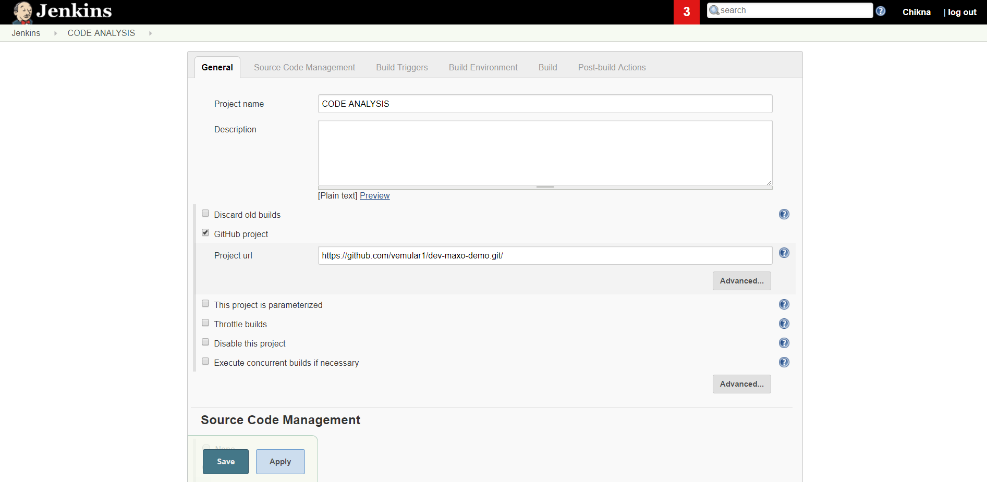
**Code Analysis:**

* Enter an Item name and select a free style job. (For all the remaining jobs we select the same)

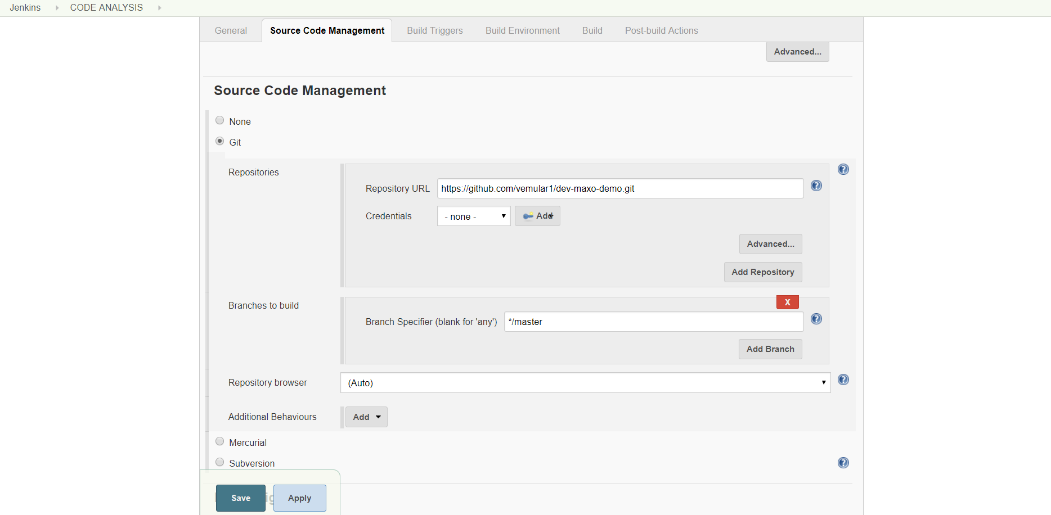


* In the first step we select the GitHub project and enter the URL of our project which is

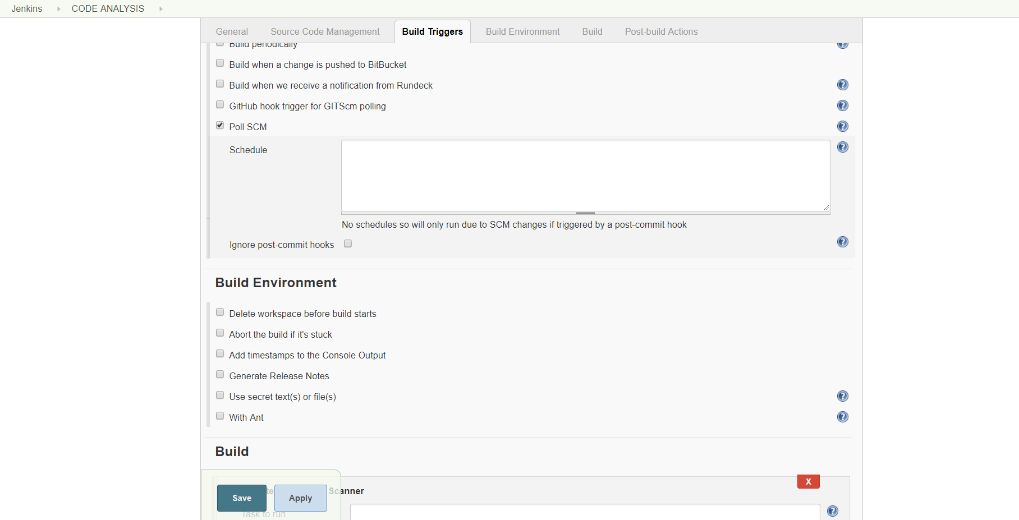
<https://github.com/vemular1/dev-maxo-demo>



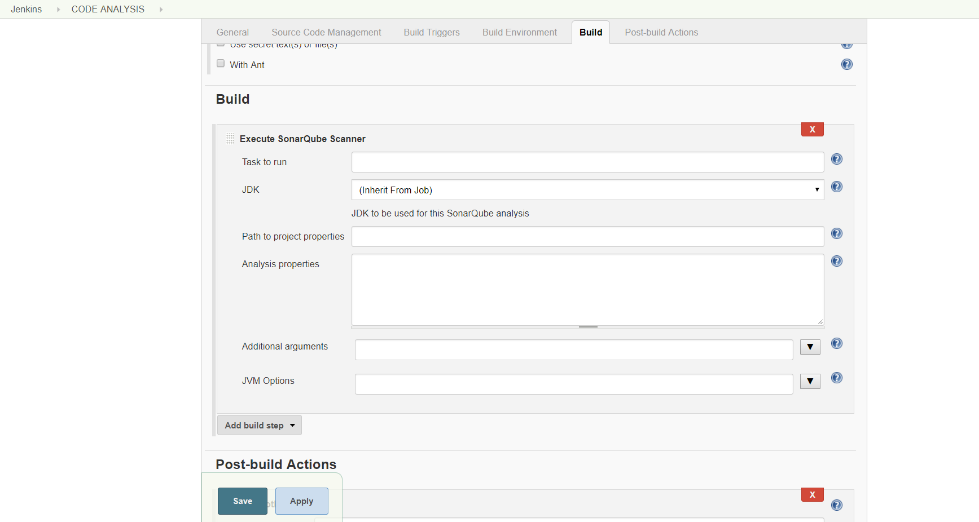
* Now paste the same github project and enter the credentials if it is a private repository and select the branch you want to build.



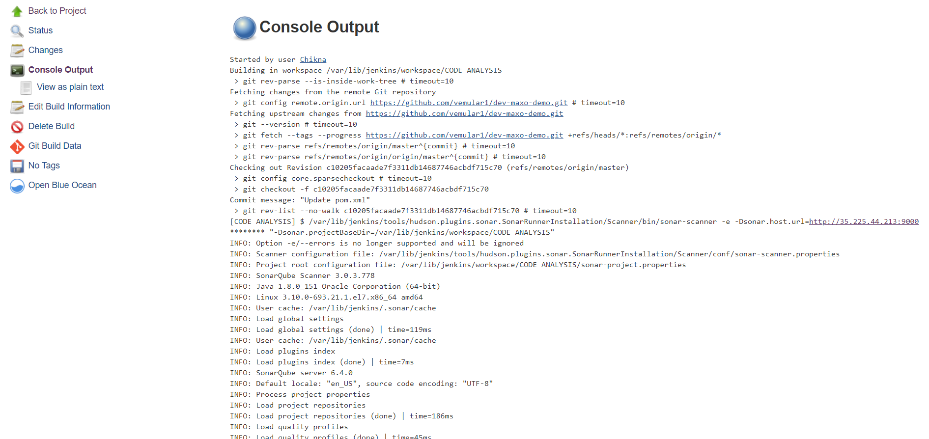
* Now schedule the builds according to your priority (cronjobs)



* In this step select the build step and in the drop down find the sonarqube scanner. (You can define the properties here or leave blank), click apply and save



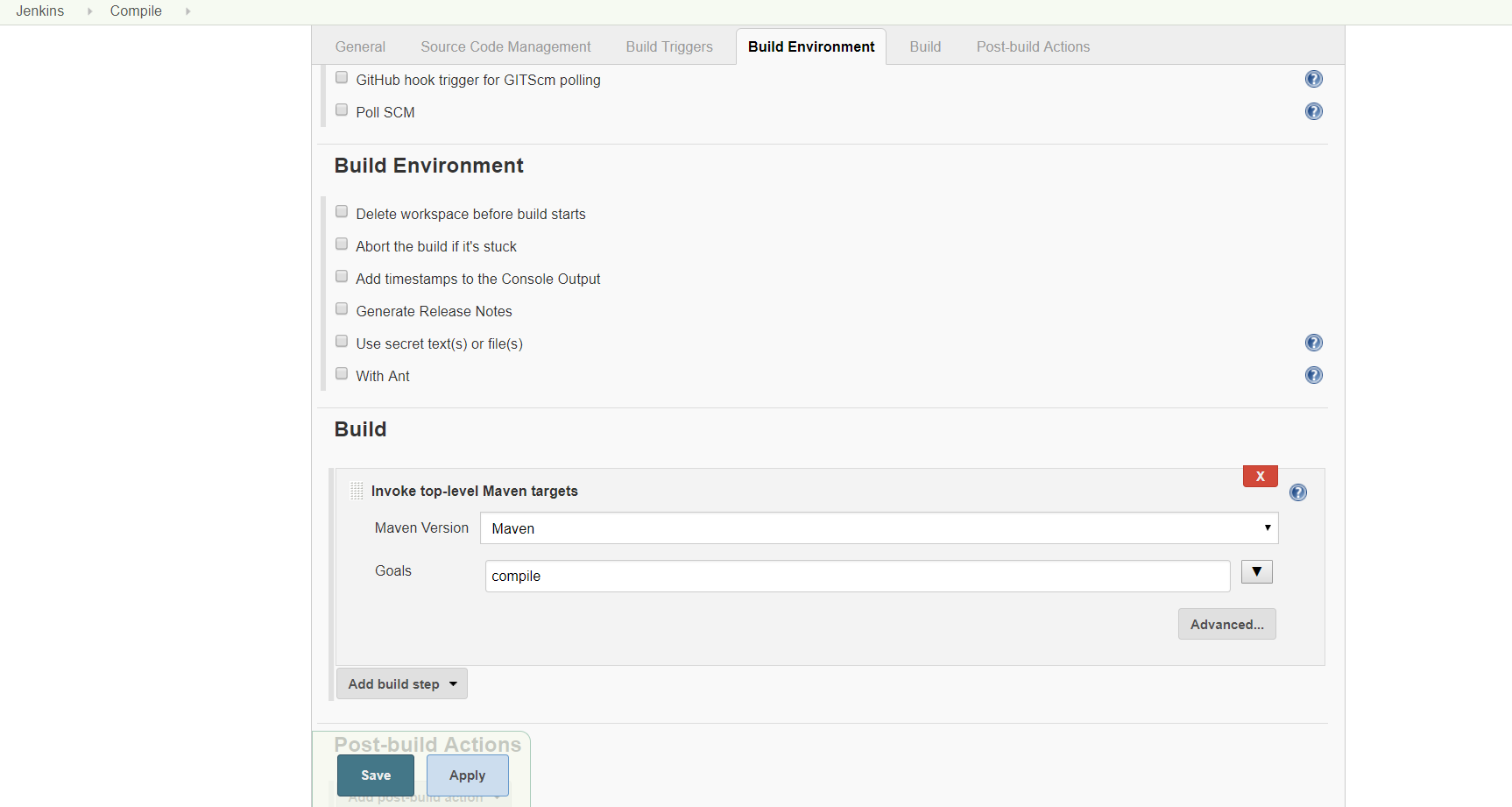
* Now go to the job page and click on build now to verify whether the job runs successfully.





**Compile:**

* Now create the next stage in the pipeline as Compile to compile the code. To create the compile stage we follow the same steps as above and change the build step as compile.

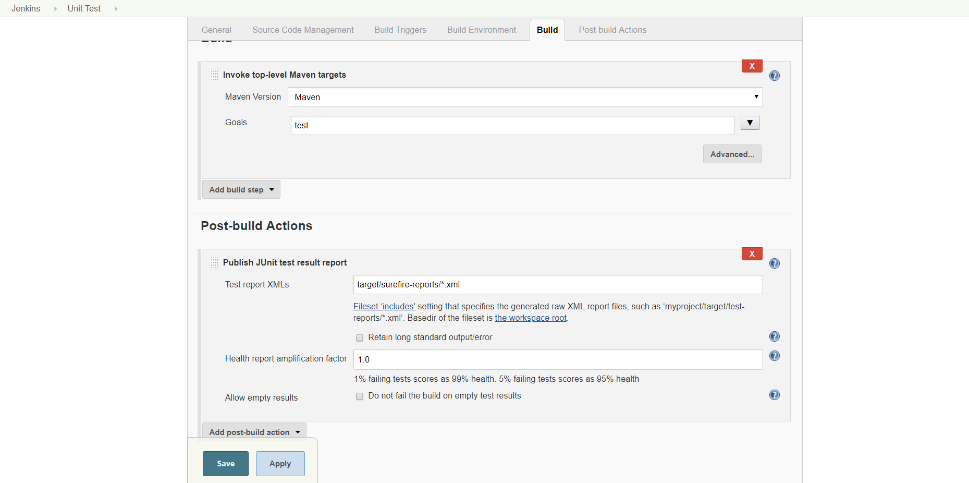


* Click apply and save the job, now build the job to check whether runs successfully.

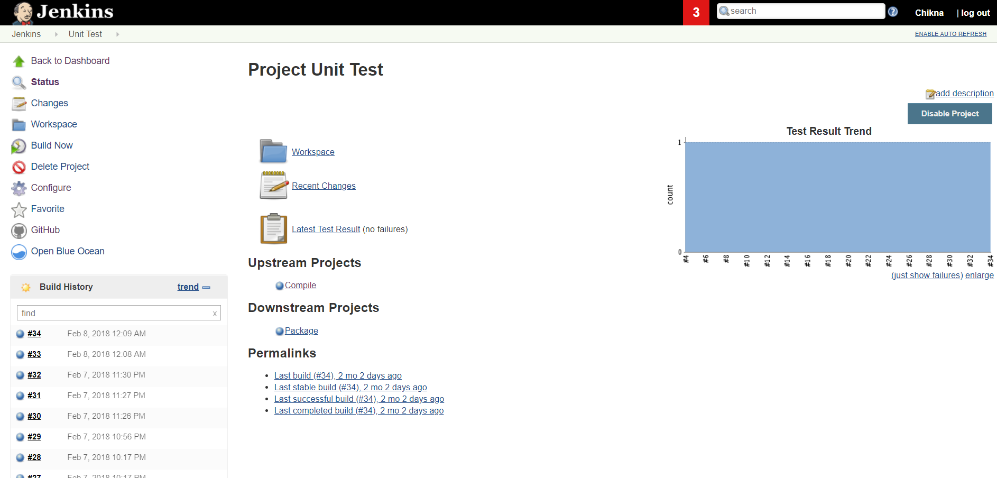


**Unit Test:**

* In the next stage we perform the Unit Test, for this stage we specify the same steps as above and in place of compile we specify as test to perform the unit test cases and we generate the report for unit test with surefire reports plugin.

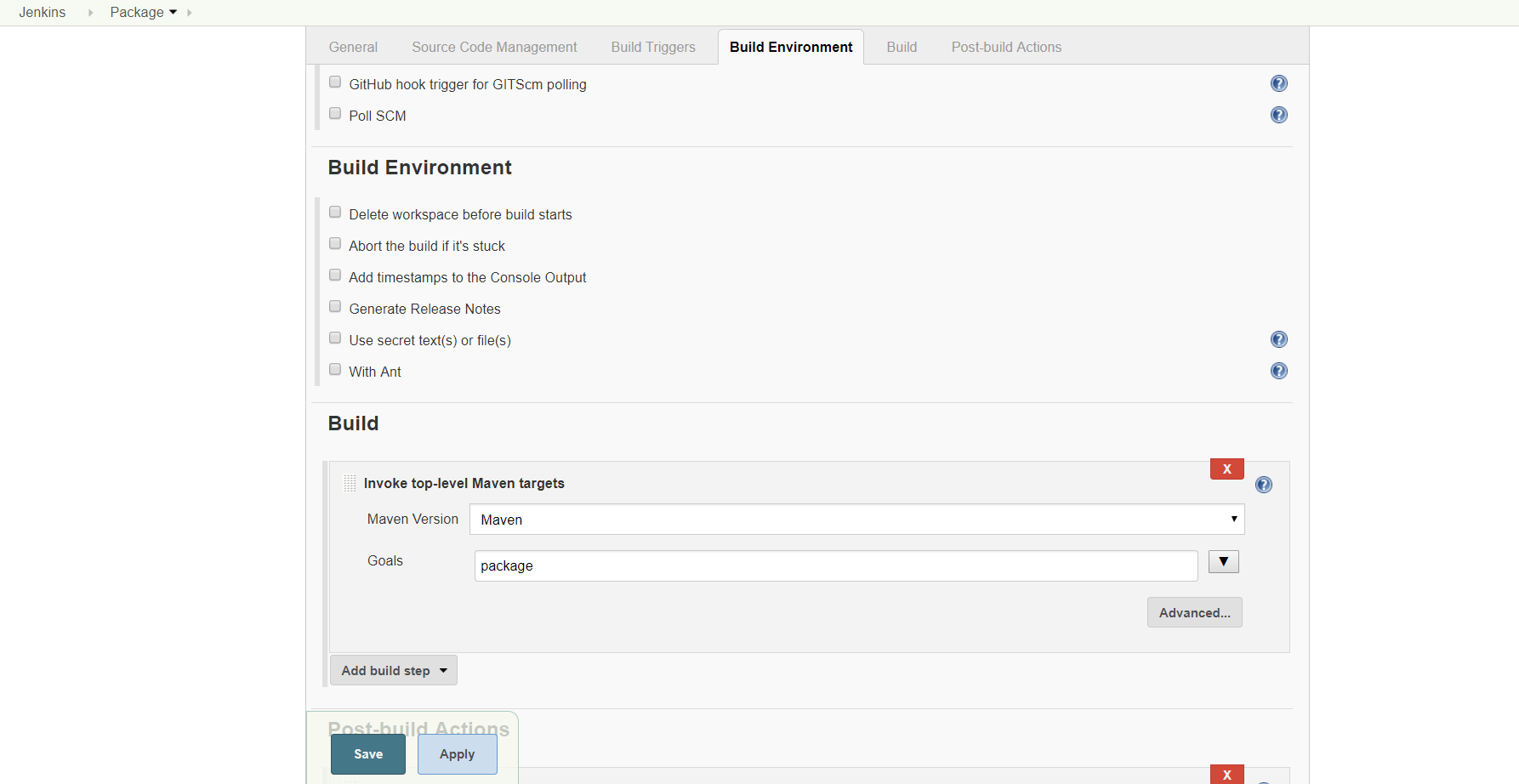


* Now click on the build now to check the build status of the job and check the output in the terminal and output page.



**Package:**

* In this job we specify the same and we in place of test we specify as package to package the application.

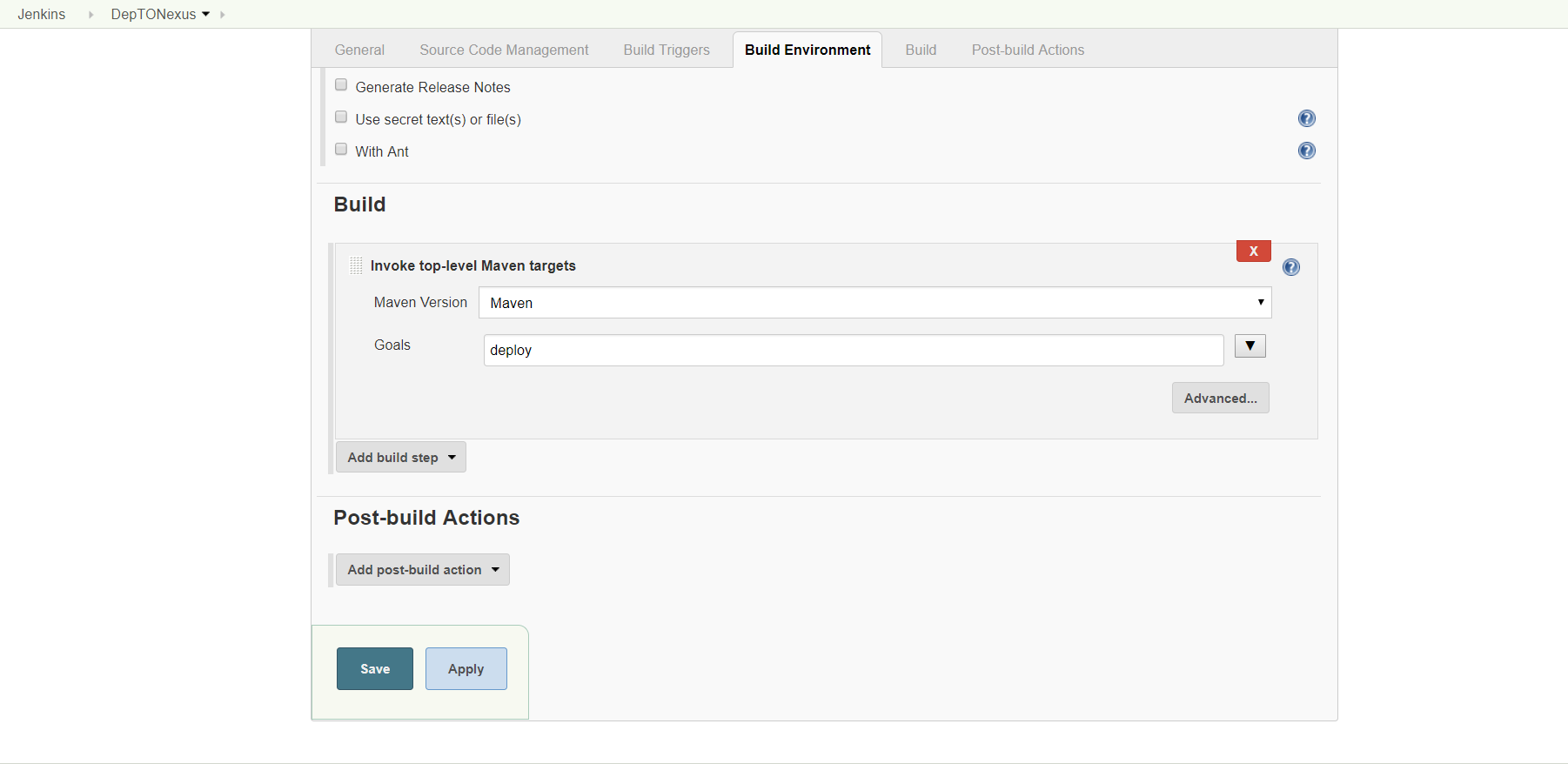


* Now check the build status with build now.



**Deploy to Nexus**

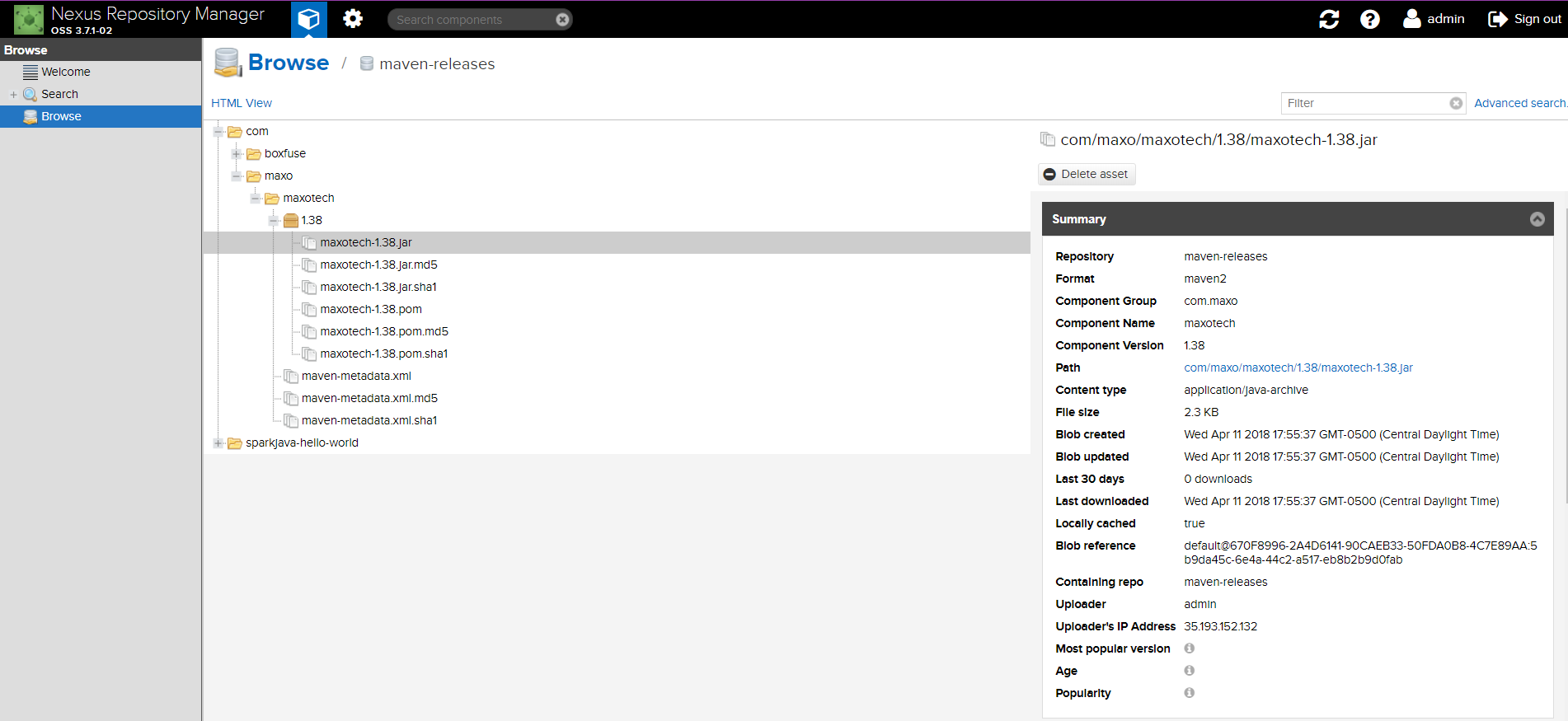
* Now we will deploy our generated artifacts to Nexus, we create the job same as previous ones and change the build step as below.



* Now we will build the job with build now and check the output on the console.

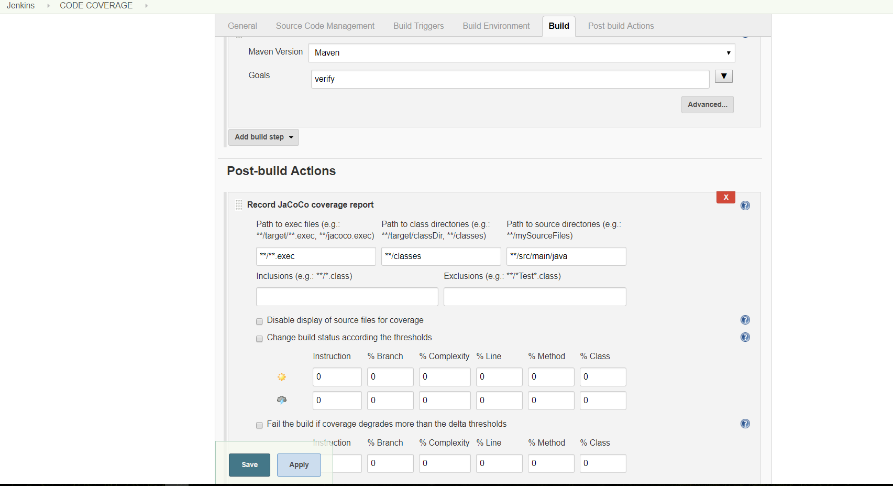


In the output screen we can see that the jar file is uploaded to the nexus repo location.



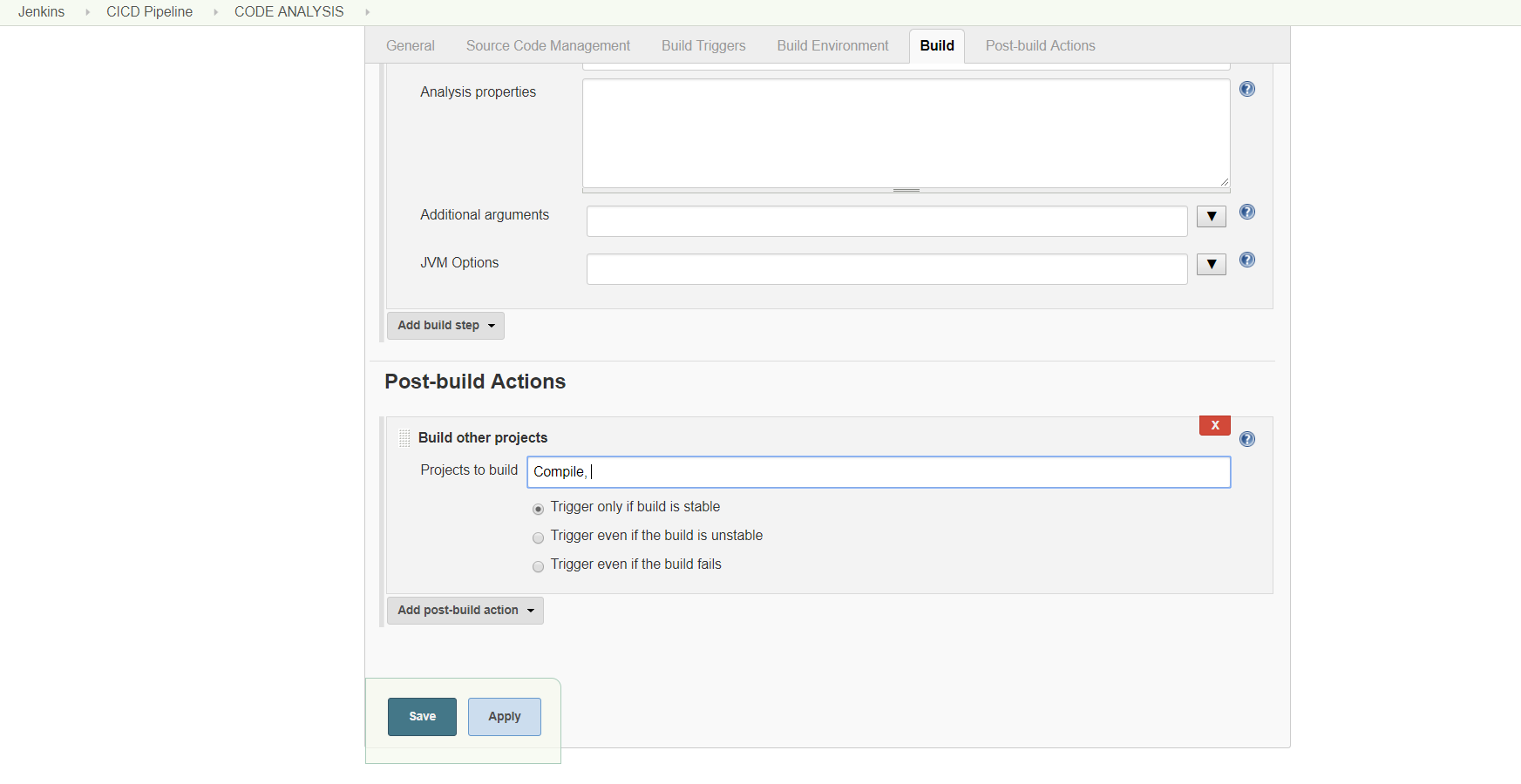
**Code Coverage:**

* Create another job called “Code Coverage” and configure it same as previous jobs and in the build step we specify it as “maven verify” and configure the post build actions as Record JaCoCo Coverage Report as shown in the figure.

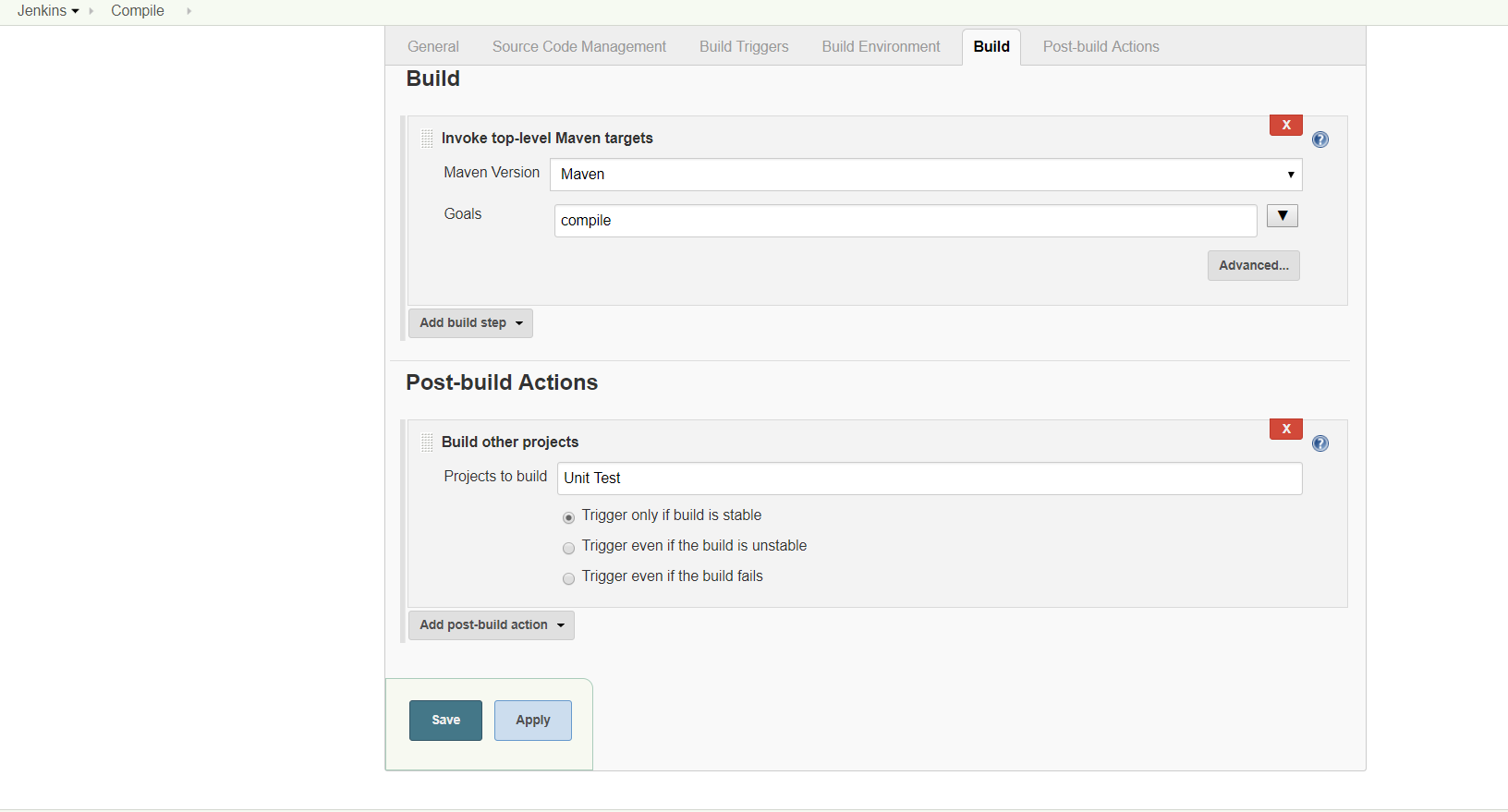


* To create a Pipeline view, go to the manage plugins page and search for “Build Pipeline Plugin” and install without restart.
* In order to execute the jobs in pipeline we specify the upstream and downstream projects in the job configurations.
* In each job post build actions, we specify the next job to be executed in pipeline.
* **Code Analysis 🡪Compile 🡪UnitTest🡪Package🡪DeptoNexus🡪CodeCoverage**

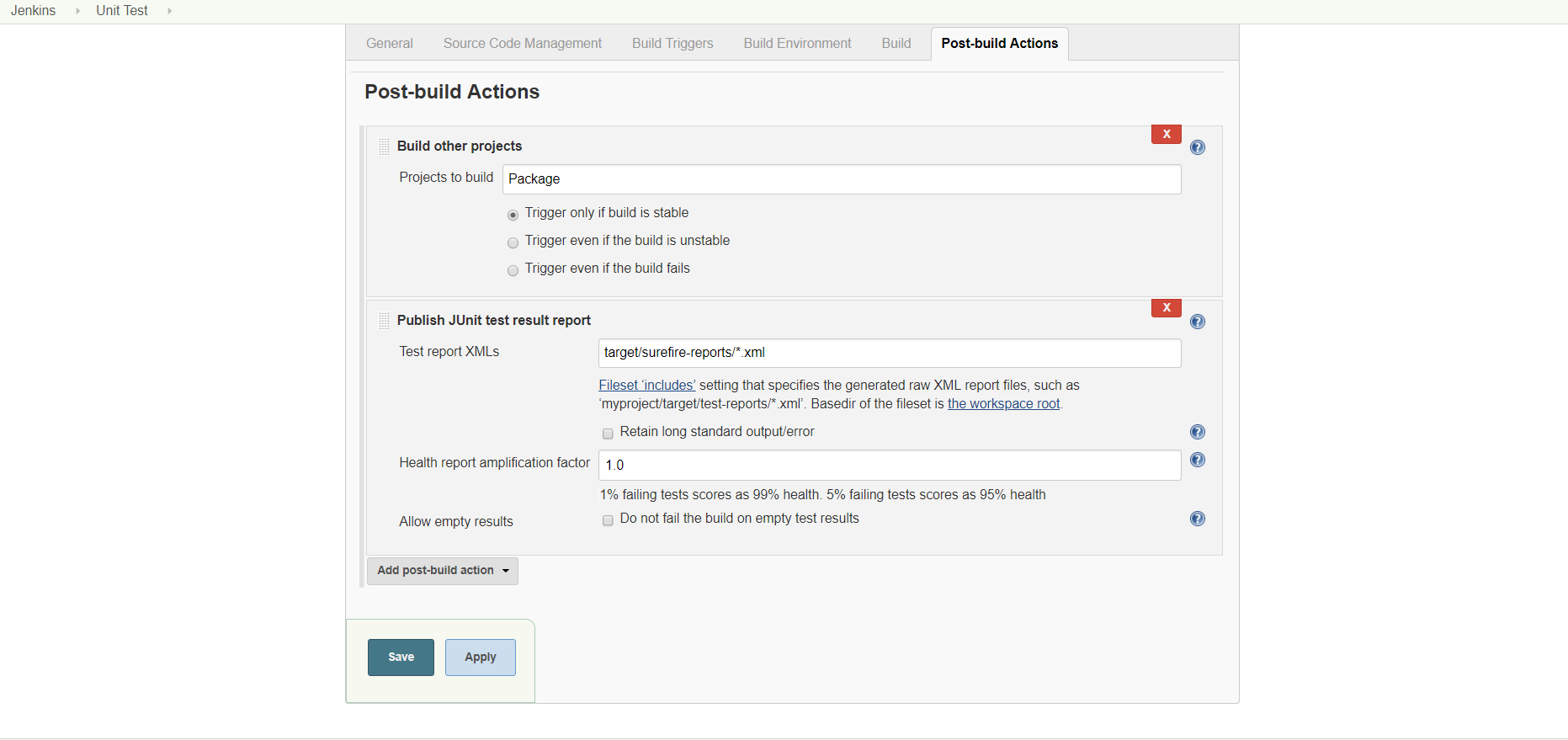
Go to first job and in the post build actions, select build other projects and specify as “Compile”



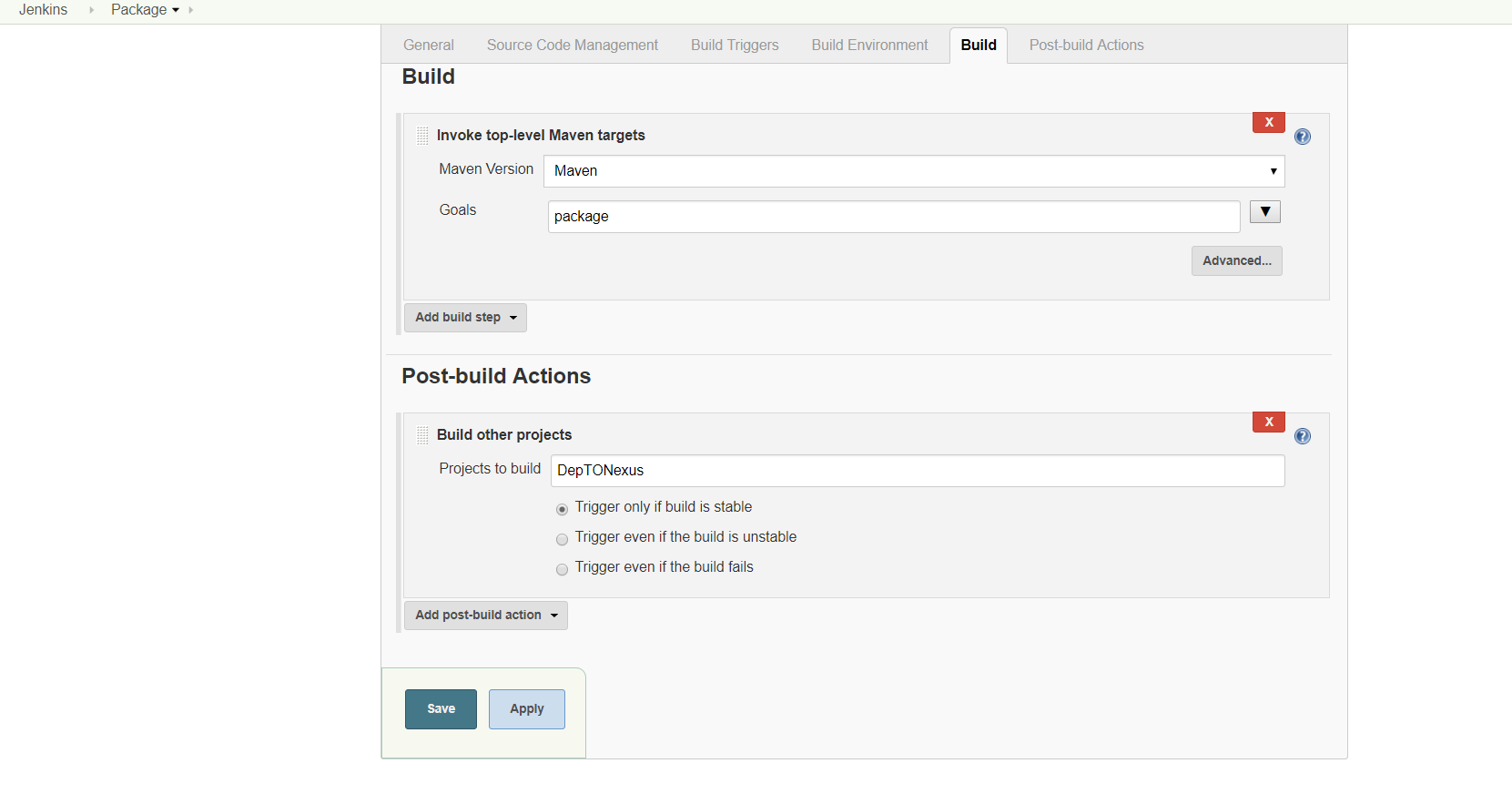
Go to compile job and now specify build other projects as “Unit Test”



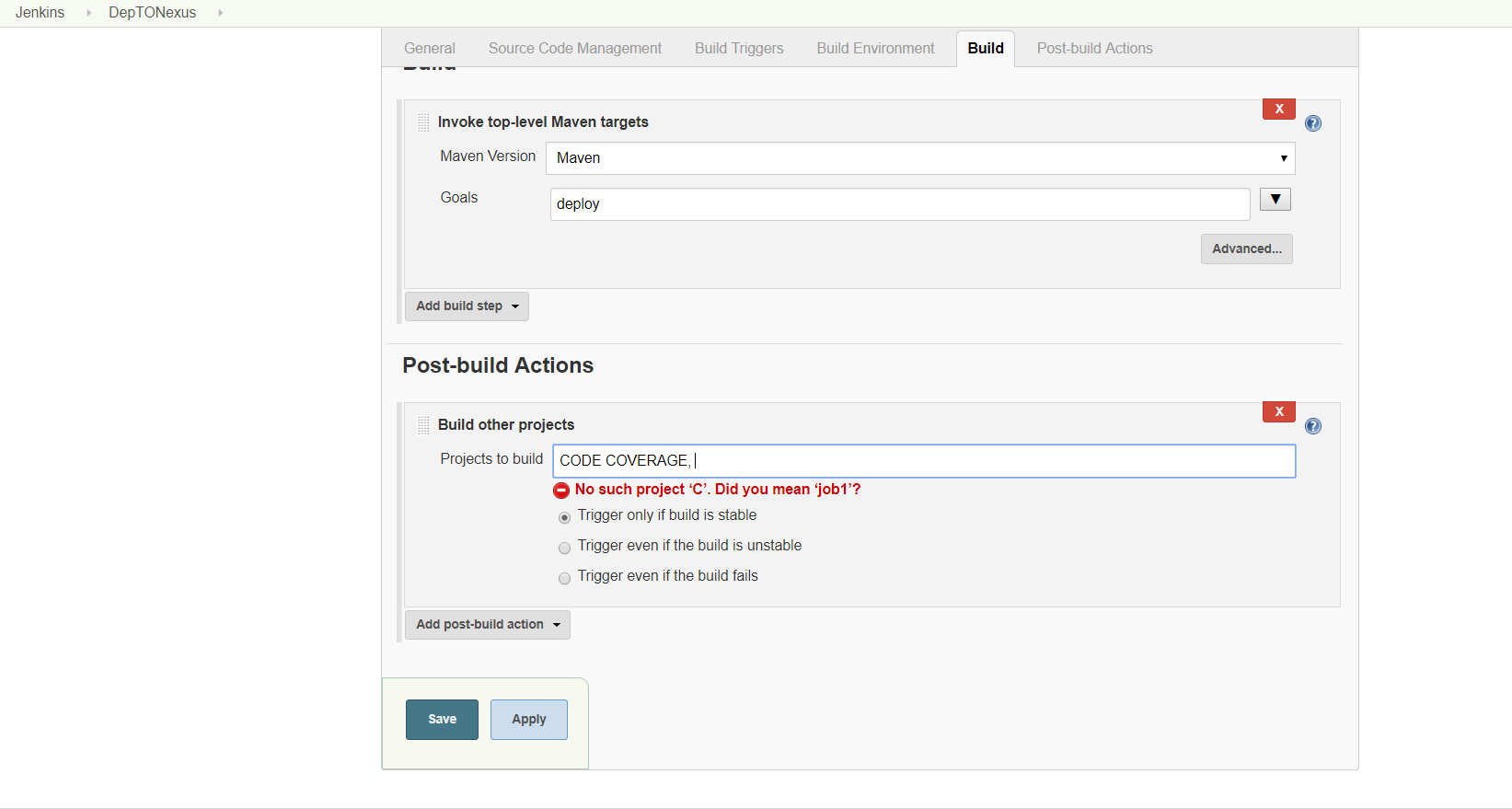
Go to Unit Test job and now specify build other projects as “Package”



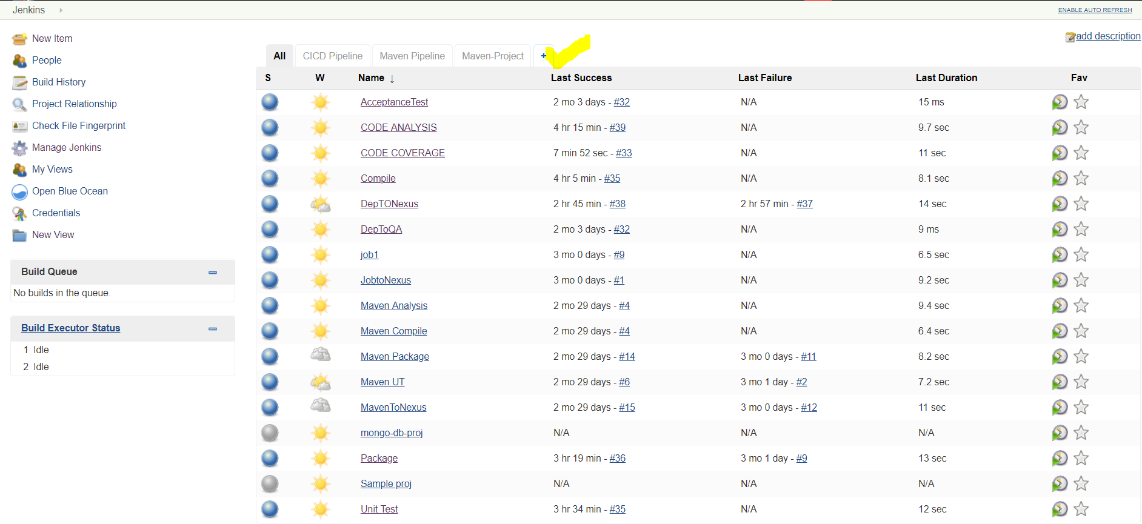
Go to Package job and now specify build other projects as “DepToNexus”



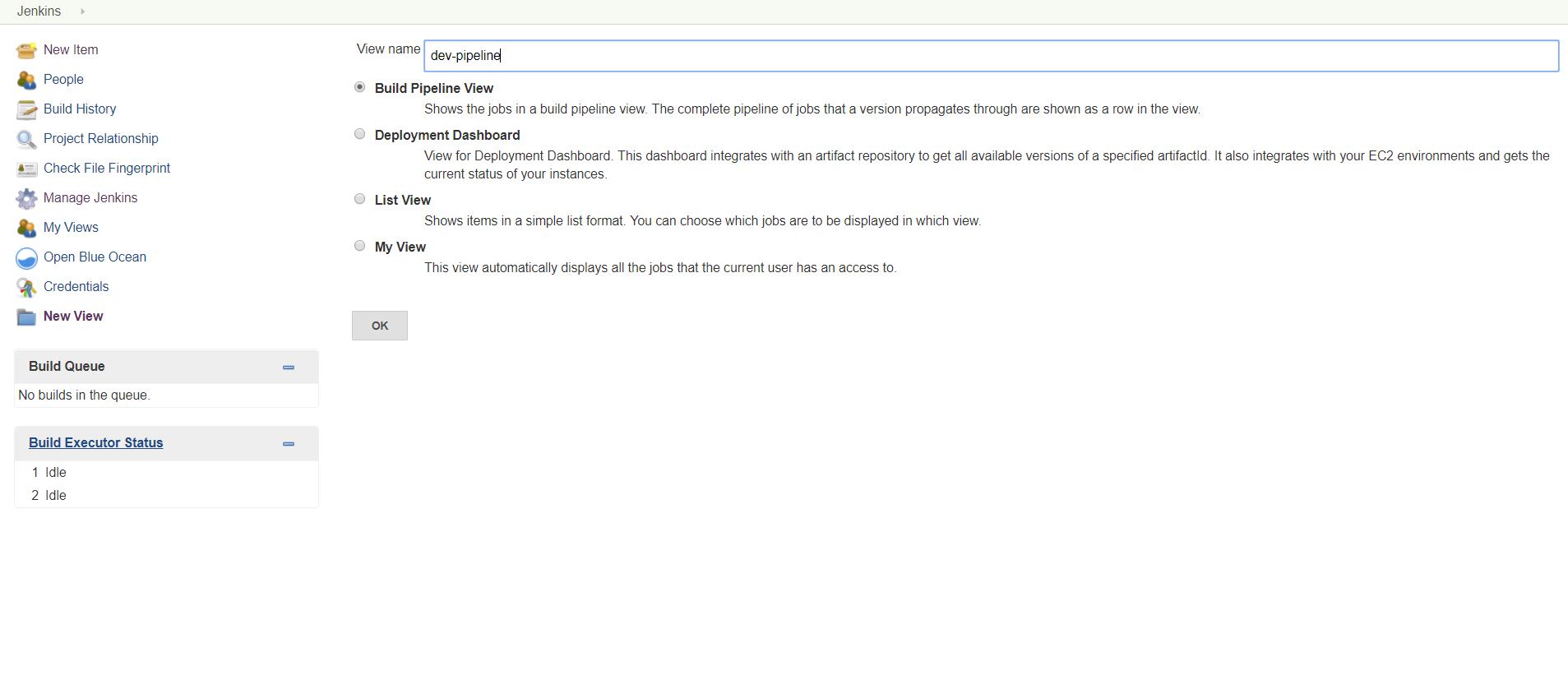
Go to DepToNexus job and now specify build other projects as “Code Coverage”



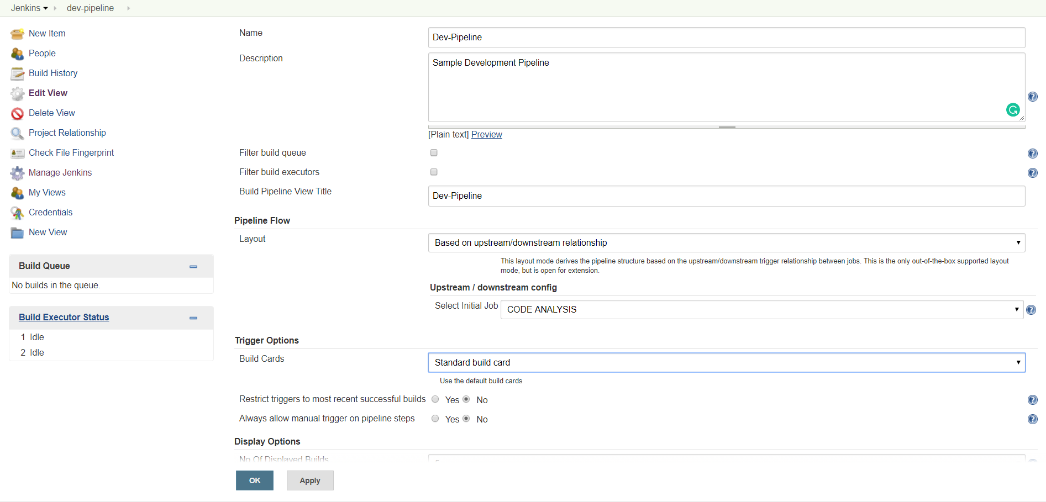
Finally, to create a pipeline view by clicking the “+” sign on the Jenkins home page



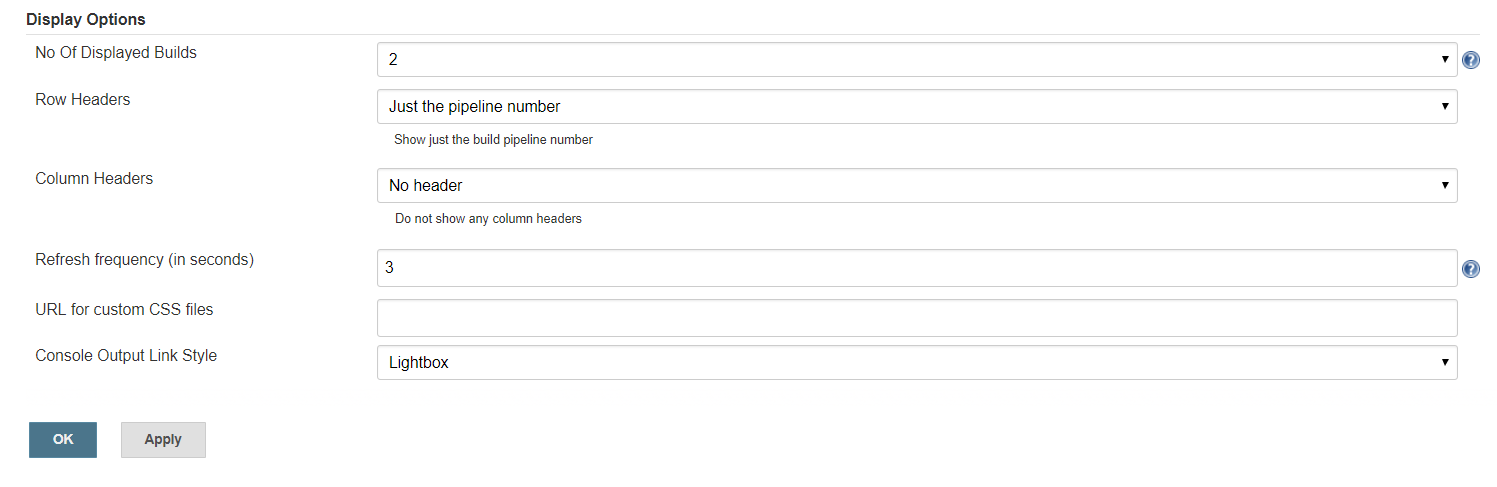
In the next page give a name to the pipeline view and select build pipeline view



In the next page give the Description of the view, Select the flow and specify the initial job as “Code Analysis”



We can also specify no of displayed builds. Here we specify as “2”



The final view is,

