

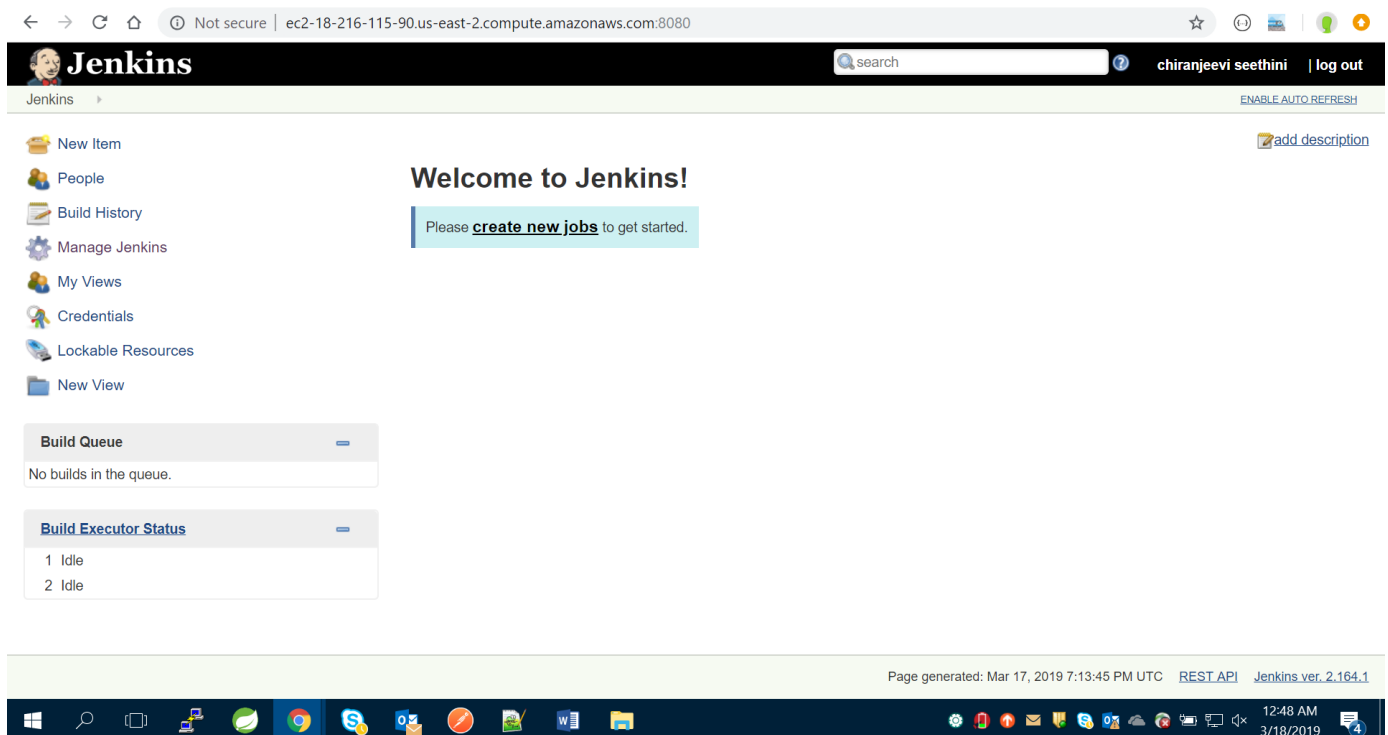
## Getting started with Jenkins

Jenkins is an open source automation tool written in **Java** with plugins built for **Continuous** Integration purpose. Jenkins is used to build and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

To start using Jenkins you need to install Jenkins

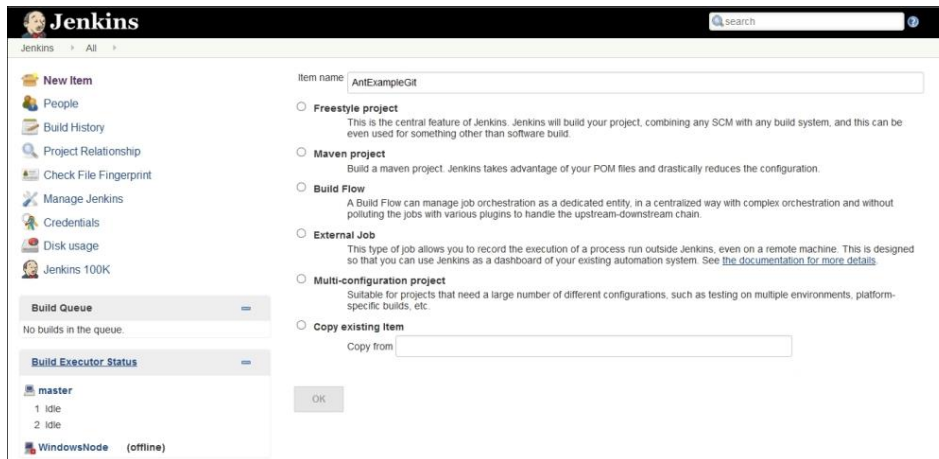
Jenkins by default run on port number 8080.

To open Jenkins the url will be http://<host name/ip >:8080 jenkins home page will looks like this when you launches it prior to configure.



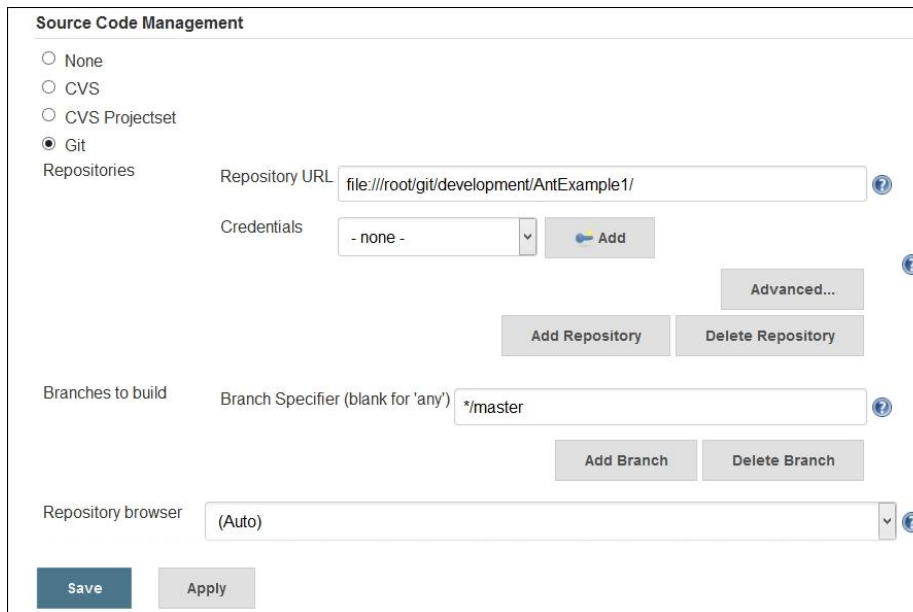
Creating job in Jenkins with git.

1. Installing the required plugin in Jenkins dashboard
  - a. On the Jenkins dashboard, click on **Manage Jenkins** and select **Manage Plugins**. Click on the **Available** tab and write github plugin in the search box.
  - b. Click the checkbox and click on the button, **Download now and install after restart**.
  - c. Restart Jenkins.
2. Click on new item from the left side navigation, choose free style project and provide a name for your job and then click ok.



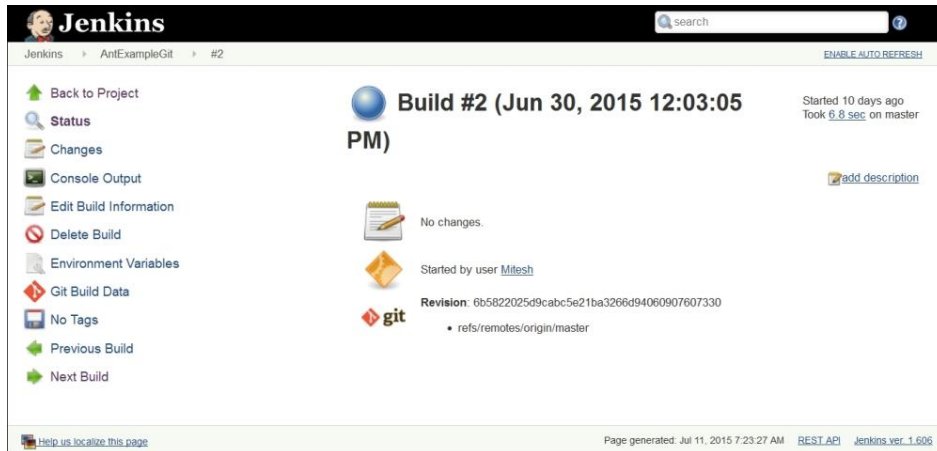
The image shows the Jenkins 'New Item' configuration page. The left sidebar contains links for 'New Item', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', 'Credentials', 'Disk usage', and 'Jenkins 100K'. Below these are sections for 'Build Queue' (showing 'No builds in the queue') and 'Build Executor Status' (showing 'master' with 1 idle and 2 idle executors, and 'WindowsNode' as offline). The main area is titled 'Item name: AntExampleGit'. It lists several project types with radio buttons: 'Freestyle project' (selected), 'Maven project', 'Build Flow', 'External Job', 'Multi-configuration project', and 'Copy existing item'. Each type has a brief description. At the bottom is an 'OK' button.

3. Check the Git hub project checkbox and provide cloning URL of your project code from the git hub.

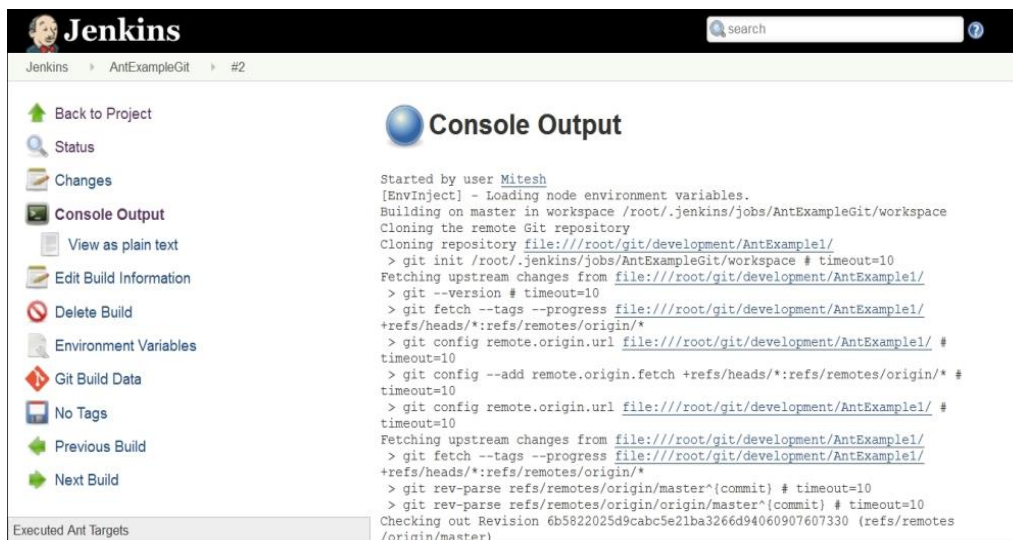


The image shows the 'Source Code Management' configuration page in Jenkins. It has a title bar 'Source Code Management'. Below it are radio buttons for 'None', 'CVS', 'CVS Projectset', and 'Git' (which is selected). Under 'Git', there is a 'Repositories' section with a 'Repository URL' field containing 'file:///root/git/development/AntExample1/' and a 'Credentials' dropdown set to '- none -'. There are 'Add' and 'Advanced...' buttons. Below this are 'Add Repository' and 'Delete Repository' buttons. The 'Branches to build' section has a 'Branch Specifier (blank for 'any')' field containing '\*/master' and 'Add Branch' and 'Delete Branch' buttons. The 'Repository browser' dropdown is set to '(Auto)'. At the bottom are 'Save' and 'Apply' buttons.

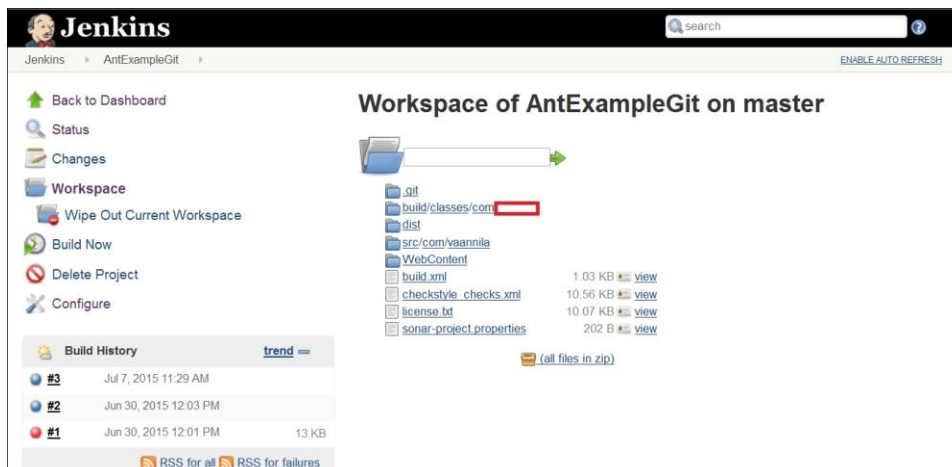
4. From the **source code management** tab select Git and provide details like project repository URL, branch and credentials.
5. You can select build tool as well from the build section.
6. Click on save button.
7. Execute the build.



8. Click on Console Output to see the progress of the build.



9. Once the build has succeeded, verify Workspace in the build job.



Done!

## Creating Jenkins pipeline

Both declarative and scripted pipeline are Domain Specific Languages to describe portions of your software delivery pipeline. Scripted Pipeline is written in a limited form of Groovy sandbox.

A Pipeline can be created in one of the following ways:

- **Through Blue Ocean** - after setting up a Pipeline project in Blue Ocean, the Blue Ocean UI helps you write your Pipeline's Jenkinsfile and commit it to source control.
- **Through the classic UI** - you can enter a basic Pipeline directly in Jenkins through the classic UI.
- **In SCM** - you can write a Jenkinsfile manually, which you can commit to your project's source control repository.

The syntax for defining a Pipeline with either approach is the same, but while Jenkins supports entering Pipeline directly into the classic UI, it is generally considered best practice to define the Pipeline in a Jenkinsfile which Jenkins will then load directly from source control.

### Through Blue Ocean

If you are new to Jenkins Pipeline, the Blue Ocean UI helps you set up your Pipeline project, and automatically creates and writes your Pipeline (i.e. the Jenkinsfile) for you through the graphical Pipeline editor.

As part of setting up your Pipeline project in Blue Ocean, Jenkins configures a secure and appropriately authenticated connection to your project's source control repository. Therefore, any changes you make to the Jenkinsfile via Blue Ocean's Pipeline editor are automatically saved and committed to source control.

Read more about Blue Ocean in the Blue Ocean chapter and Getting started with Blue Ocean page.









### Through the classic UI

A Jenkinsfile created using the classic UI is stored by Jenkins itself (within the Jenkins home directory).

To create a basic Pipeline through the Jenkins classic UI:


1. If required, ensure you are logged in to Jenkins.
2. From the Jenkins home page (i.e. the Dashboard of the Jenkins classic UI), click **New Item** at the top left.




-  [New Item](#)
-  [People](#)
-  [Build History](#)
-  [Manage Jenkins](#)
-  [My Views](#)
-  [Open Blue Ocean](#)
-  [Credentials](#)
-  [New View](#)

3. In the **Enter an item name** field, specify the name for your new Pipeline project.  
**Caution:** Jenkins uses this item name to create directories on disk. It is recommended to avoid using spaces in item names, since doing so may uncover bugs in scripts that do not properly handle spaces in directory paths.
4. Scroll down and click **Pipeline**, then click **OK** at the end of the page to open the Pipeline configuration page (whose **General** tab is selected).


**Enter an item name**  
  
» Required field

**Freestyle project**


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

**Pipeline**


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

**Multi-configuration project**


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

**Bitbucket Team/Project**


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

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**GitHub Organization**

Scans a GitHub organization (or user account) for all repositories matching some defined markers.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.  
**Note:** If instead you are defining your Jenkinsfile in source control, follow the instructions in In SCM below.
6. In the **Pipeline** section, ensure that the **Definition** field indicates the **Pipeline script** option.
7. Enter your Pipeline code into the **Script** text area.  
For instance, copy the following Declarative example Pipeline code (below the *Jenkinsfile* heading) or its Scripted version equivalent and paste this into the **Script** text area. (The Declarative example below is used throughout the remainder of this procedure.)

#### *Jenkinsfile (Declarative Pipeline)*

```
pipeline {
  agent any
  stages {
    stage('Stage 1') {
      steps {
        echo 'Hello world!'
      }
    }
  }
}
```

- a. agent instructs Jenkins to allocate an executor (on any available agent/node in the Jenkins environment) and workspace for the entire Pipeline
- b. echo writes simple string in the console output.
- c. node effectively does the same as agent (above).

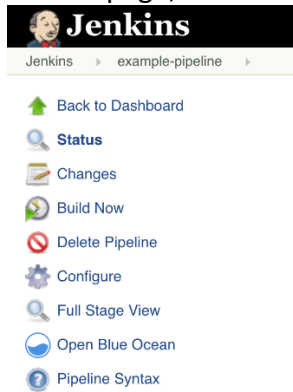
The screenshot shows the Jenkins configuration page for a Pipeline. The 'Pipeline' tab is active. Under the 'Definition' section, 'Pipeline script' is selected. The 'Script' text area contains the following code:

```
1= pipeline {
2=   agent any
3=   stages {
4=     stage('Stage 1') {
5=       steps {
6=         echo 'Hello world!'
7=       }
8=     }
9=   }
10= }
```

Below the script area, there is a checkbox labeled 'Use Groovy Sandbox' which is checked. At the bottom of the page, there are two buttons: 'Save' and 'Apply'.

**Note:** You can also select from canned *Scripted* Pipeline examples from the **try sample Pipeline** option at the top right of the **Script** text area. Be aware that there are no canned Declarative Pipeline examples available from this field.

8. Click **Save** to open the Pipeline project/item view page.
9. On this page, click **Build Now** on the left to run the Pipeline.



10. Under **Build History** on the left, click **#1** to access the details for this particular Pipeline run.
11. Click **Console Output** to see the full output from the Pipeline run. The following output shows a successful run of your Pipeline.

## Console Output

```
Started by user Alex
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/example-pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Stage 1)
[Pipeline] echo
Hello world!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

### In SCM

Complex Pipelines are difficult to write and maintain within the classic UI's **Script** text area of the Pipeline configuration page.

To make this easier, your Pipeline's Jenkinsfile can be written in a text editor or integrated development environment (IDE) and committed to source control. Jenkins can then check out your Jenkinsfile from source control as part of your Pipeline project's build process and then proceed to execute your Pipeline.

To configure your Pipeline project to use a Jenkinsfile from source control:

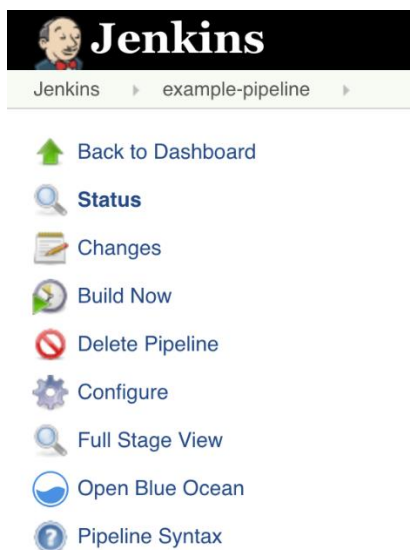
1. Follow the procedure above for defining your Pipeline through the classic UI until you reach step 5 (accessing the **Pipeline** section on the Pipeline configuration page).
2. From the **Definition** field, choose the **Pipeline script from SCM** option.
3. From the **SCM** field, choose the type of source control system of the repository containing your Jenkinsfile.
4. Complete the fields specific to your repository's source control system.
5. In the **Script Path** field, specify the location (and name) of your Jenkinsfile. This location is the one that Jenkins checks out/clones the repository containing your Jenkinsfile, which should match that of the repository's file structure. The default value of this field assumes that your Jenkinsfile is named "Jenkinsfile" and is located at the root of the repository.

When you update the designated repository, a new build is triggered, as long as the Pipeline is configured with an SCM polling trigger.

### Built-in Documentation

Pipeline ships with built-in documentation features to make it easier to create Pipelines of varying complexities. This built-in documentation is automatically generated and updated based on the plugins installed in the Jenkins instance.

The built-in documentation can be found globally at: localhost:8080/pipeline-syntax/, assuming you have a Jenkins instance running on localhost port 8080. The same documentation is also linked as **Pipeline Syntax** in the side-bar for any configured Pipeline project.





## Snippet Generator

The built-in "Snippet Generator" utility is helpful for creating bits of code for individual steps, discovering new steps provided by plugins, or experimenting with different parameters for a particular step.


The Snippet Generator is dynamically populated with a list of the steps available to the Jenkins instance. The number of steps available is dependent on the plugins installed which explicitly expose steps for use in Pipeline.


To generate a step snippet with the Snippet Generator:

1. Navigate to the **Pipeline Syntax** link (referenced above) from a configured Pipeline, or at localhost:8080/pipeline-syntax.
2. Select the desired step in the **Sample Step** dropdown menu
3. Use the dynamically populated area below the **Sample Step** dropdown to configure the selected step.
4. Click **Generate Pipeline Script** to create a snippet of Pipeline which can be copied and pasted into a Pipeline.

**Steps**

Sample Step stage: Stage ▼

Stage Name  



**Generate Pipeline Script**

```
stage('Deploy') {  
    // some block  
}
```

To access additional information and/or documentation about the step selected, click on the help icon (indicated by the red arrow in the image above).

## Global Variable Reference

In addition to the Snippet Generator, which only surfaces steps, Pipeline also provides a built-in "**Global Variable Reference**." Like the Snippet Generator, it is also dynamically populated by plugins. Unlike the Snippet Generator however, the Global Variable Reference only contains documentation for **variables** provided by Pipeline or plugins, which are available for Pipelines.

The variables provided by default in Pipeline are:

## **env**

Environment variables accessible from Scripted Pipeline, for example: `env.PATH` or `env.BUILD_ID`. Consult the built-in Global Variable Reference for a complete, and up to date, list of environment variables available in Pipeline.

## **params**

Exposes all parameters defined for the Pipeline as a read-only Map, for example: `params.MY_PARAM_NAME`.

## **currentBuild**

May be used to discover information about the currently executing Pipeline, with properties such as `currentBuild.result`, `currentBuild.displayName`, etc. Consult the built-in Global Variable Reference for a complete, and up to date, list of properties available on `currentBuild`.

## **Declarative Directive Generator**

While the Snippet Generator helps with generating steps for a Scripted Pipeline or for the steps block in a stage in a Declarative Pipeline, it does not cover the sections and directives used to define a Declarative Pipeline. The "Declarative Directive Generator" utility helps with that. Similar to the Snippet Generator, the Directive Generator allows you to choose a Declarative directive, configure it in a form, and generate the configuration for that directive, which you can then use in your Declarative Pipeline.

To generate a Declarative directive using the Declarative Directive Generator:

1. Navigate to the **Pipeline Syntax** link (referenced above) from a configured Pipeline, and then click on the **Declarative Directive Generator** link in the sidepanel, or go directly to `localhost:8080/directive-generator`.
2. Select the desired directive in the dropdown menu
3. Use the dynamically populated area below the dropdown to configure the selected directive.
4. Click **Generate Directive** to create the directive's configuration to copy into your Pipeline.

The Directive Generator can generate configuration for nested directives, such as conditions inside a when directive, but it cannot generate Pipeline steps. For the contents of directives which contain steps, such as steps inside a stage or conditions like always or failure inside post, the Directive Generator adds a placeholder comment instead. You will still need to add steps to your Pipeline by hand.

### *Jenkinsfile (Declarative Pipeline)*

```
stage('Stage 1') {
  steps {
    // One or more steps need to be included within the steps block.
  }
}
```

Example code

```
node {

    stage '1. Git checkout APP NAME'
        git branch: 'Branch name', credentialsId: 'Credentials_id', url: ' git
repository project URL '
    }

    stage '2. Prepare Environment'
        tool name: 'Maven 3.3.9', type: 'maven'
        env.JAVA_HOME="${tool 'OpenJDK 1.8'}"
        env.PATH="${env.JAVA_HOME}/bin :${ env.PATH}"

    Stage '3. Add <distributionManagement>'
        dir('service name') {
            sh """
            echo '<distributionManagement>
            <snapshotRepository>
            <id>snapshots</id>
            <name>Snapshots</name>

            <url>http://hostname:8081/nexus/content/repositories/snapshots</url>
            </snapshotRepository>
            </distributionManagement>
            </project>' > append

            sed -i 's|</project>||g' pom.xml
            cat append >> pom.xml
            tail -n 50 pom.xml
            """
            sh "cat pom.xml"
        }

    stage '4. Execute Unit Tests'
        withMaven(globalMavenSettingsConfig: 'a3c77df0-d448-414b-8c6c-4879598a99e1',
jdk: 'OpenJDK 1.8', maven: 'Maven 3.3.9') {
            dir('Project name/service name') {
                sh "mvn clean install -X"
            }
        }

    stage '5. SonarQube analysis'
        dir('service name') {
            def ACCESS_TOKEN = 'access token of sonarqube available in sonarqube
properties';
            def SONAR_URL = 'sonarqube host location ex. http://hostname:9000'
            def scannerHome = tool 'SonarQube scanner';
            def pom = readMavenPom file: 'pom.xml';
            sh """
                ${scannerHome}/bin/sonar-scanner \
                -Dsonar.host.url=${SONAR_URL} \
                -Dsonar.login=${ACCESS_TOKEN} \
            """
        }
    }
```

```

        -Dsonar.projectKey=${pom.artifactId} \
        -Dsonar.projectName=${pom.artifactId} \
        -Dsonar.projectVersion=${pom.version} \
        -Dsonar.sources=. \
        -Dsonar.java.binaries=. \
        -Dsonar.java.source=1.8 \
        -Dsonar.language=java \
        -X
    """
}

    stage '6. Build and deploy service to Nexus Snapshots repository'
    withMaven(globalMavenSettingsConfig: 'credentials id', jdk: 'OpenJDK 1.8',
maven: 'Maven 3.3.9') {
        dir('service name') {
            sh "mvn deploy"
        }
    }

    Stage '7. Docker build and push to aws'
    withCredentials(bindings: [usernamePassword(credentialsId: 'credentials
id, usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD')]) {
        sh "docker login -u ${USERNAME} -p ${PASSWORD} ${DOCKER_REPO}"
        script {
            sh "docker build -t ${AWS_DOCKER_TAG}:${DOCKER_VERSION} ."
            sh "docker push ${AWS_DOCKER_TAG}:${DOCKER_VERSION}"
        }
        sh "docker logout ${DOCKER_REPO}"
    }
}

    Stage '8. Deploy to aws'
    withCredentials(bindings: [usernamePassword(credentialsId: '<credentials_id>,
usernameVariable: '<USERNAME>', passwordVariable: '<PASSWORD>')]) {
        sh "docker login -u ${USERNAME} -p ${PASSWORD} ${DOCKER_REPO}"
        sh 'chmod +x ./build/deploy-service.yaml'
        sh 'chmod +x ./build/ingress.yaml'
        sh """
            cd build
            export CONFIGMAP=configmap-${AWS_REGION}-dev
            export TARGET_HOST=aws
            export DOCKER_VERSION=${DOCKER_VERSION}
            cp configmap-${AWS_REGION}-dev.yaml configmap-${AWS_REGION}-
dev-aws.yaml
            cp deploy-service.yaml deploy-service-aws.yaml
            cp ingress.yaml ingress-aws.yaml
            sed -i -e
\"s|IMAGE_NAME_VAR|${AWS_DOCKER_TAG}:${DOCKER_VERSION}|g\" deploy-service-aws.yaml
            sed -i -e
\"s|INTERNAL_SVC_HOSTNAME_VAR|${INTERNAL_SVC_HOSTNAME}|g\" ingress-aws.yaml
            cd ./dev-ucp-bundle-admin
            . ./env.sh
            cd ..
            . ./deploy.sh
        """
        sh "docker logout ${DOCKER_REPO}"
    }
}

```

