Jenkins Pipeline

Pipeline scripts are another way of job configuration with the help of groovy code.

A Pipeline’s code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.

**Advantages:**

* We can divide the jobs into parts like build, deploy and test and we can schedule each stage or part on separate agent
* Parallel execution of stages is easy to configure.
* We can use different versions of tolls in single pipeline like JDK, maven and etc.
* We can check the console and verify the job status.

**Why Pipeline?**

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modeling a series of related tasks, users can take advantage of the many features of Pipeline:

* **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
* **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins controller.
* **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
* **Versatile**: Pipelines support complex real-world CD requirements, including the ability to fork/join, loop, and perform work in parallel.
* **Extensible**: The Pipeline plugin supports custom extensions to its DSL [[1](https://www.jenkins.io/doc/book/pipeline/#_footnotedef_1)] and multiple options for integration with other plugins.

Types of Pipelines:

* **Scripted pipeline**
* **Declarative pipeline**

**Scripted pipeline:**

These are the old and traditional way of job configuration we have limited options here compared to declarative pipeline.The scripted pipeline will start with node. Below is the syntax of sample scripted pipeline.

node{

stage('Preparation') {

echo “hello world”

}

}

**Declarative pipeline:**

It’s recent addition of pipelines here we have many simplified and opinionated syntax when compared to scripted pipeline.

The declarative pipeline will start with pipeline. Below is the syntax of sample declarative pipeline.

pipeline {

agent any

stages {

stage('build'){

steps {

cleanWs()

echo ‘Hello World’

}

}

}

**Pipeline** is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.

[**agent**](https://www.jenkins.io/doc/book/pipeline/syntax#agent) is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.

**Stages** is collection of stage.

**Stage** is a syntax block that describes a [stage of this Pipeline](https://www.jenkins.io/doc/book/pipeline/#stage). stage blocks are optional in Scripted Pipeline syntax.

[**Steps**](https://www.jenkins.io/doc/book/pipeline/syntax#steps) is Declarative Pipeline-specific syntax that describes the steps to be run in this stage.

**sh** is pipeline that execute the shell commands.

Agent: Agent section defines where the entire pipeline or stage to be run.

agent any

agent none

agent { label “my-agent” }

agent { docker 'maven:3.8.1-adoptopenjdk-11' }

scripted : node(label ‘mu-agent’) { }

**any** : Execute the Pipeline, or stage, on any available agent.

**none** : entire Pipeline run and each stage section will need to contain its own agent section.

**label** : Execute the Pipeline, or stage, on an agent available in the Jenkins environment with the provided label.

**Docker** : Execute the Pipeline, or stage, with the given container which will be dynamically provisioned on a [node](https://www.jenkins.io/doc/book/glossary/#node) pre-configured to accept Docker-based Pipelines, or on a node matching the optionally defined label parameter.

**Variables:**

Variables are used to define the values in pipeline.

There are 2 types

* **Pre-defined variables**: These are the default variables we access use these in anywhere in Jenkins, we can see using http://${jenkins\_url}:port/env-vars.html
* **User-defined variables**: These are user defined variables, we can define inside pipeline or stage

environment{

tool = 'jenkins'

}

**Parameters:**

Parameters are used to transfer the values to job dynamically at runtime.

There are many types of parameter

**Text:**

A text parameter, which can contain multiple lines, for example: parameters { text(name: 'DEPLOY\_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '') }

**booleanParam:**

A boolean parameter, for example: parameters { booleanParam(name: 'DEBUG\_BUILD', defaultValue: true, description: '') }

**Choice:**

A choice parameter, for example: parameters { choice(name: 'CHOICES', choices: ['one', 'two', 'three'], description: '') }

**password**

A password parameter, for example: parameters { password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password') }

**Options:**

**Retry:**

On failure, retry the entire Pipeline the specified number of times. For example: options { retry(3) }

**buildDiscarder:**

Persist artifacts and console output for the specific number of recent Pipeline runs. For example: options { buildDiscarder(logRotator(numToKeepStr: '1')) }

**disableConcurrentBuilds:**

Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources, etc. For example: options { disableConcurrentBuilds() }

**Timeout:**

Set a timeout period for the Pipeline run, after which Jenkins should abort the Pipeline. For example: options { timeout(time: 1, unit: 'HOURS') }

**Timestamp:**

adding current time on console output. For example: options { timestamp() }

**Trigger other jobs:**

In this function we can able to trigger the other jobs in same Jenkins with different folder location.

Syntax: build(‘job1’)

**Triggers:**

The triggers directive defines the automated ways in which the Pipeline should be re-triggered. For Pipelines which are integrated with a source such as GitHub or BitBucket, triggers may not be necessary as webhooks-based integration will likely already be present. The triggers currently available are cron, pollSCM.

* **Cron:** This will trigger the job at particular cron schedule.

Syntax: triggers { cron('H \*/4 \* \* 1-5') }

* **pollSCM:** This will trigger the job at particular cron schedule if the commit’s are done.

Syntax: triggers { pollSCM('H \*/4 \* \* 1-5') }

**Parallel builds:**

Stages in Declarative Pipeline may have a parallel section containing a list of nested stages to be run in parallel, here we can add multiple stages inside the parallel section.

Syntax:

stage("parallel execution od stgaes"){

    parallel{

        stage('test job1'){

            steps{

                echo 'test job 1'

            }

        }

        stage('test job2'){

            steps{

                echo 'test job 2'

            }

        }

    }

}

**Try/catch :**

Another way Scripted Pipeline flow control can be managed is with Groovy’s exception handling support. When [Steps](https://www.jenkins.io/doc/book/pipeline/syntax/#scripted-steps) fail for whatever reason they throw an exception.

**try** {

sh 'exit 1'

}

**catch** (exc) {

echo 'Something failed, I should sound the klaxons!'

}

**Post:**

The post section defines one or more additional [steps](https://www.jenkins.io/doc/book/pipeline/syntax/#declarative-steps) that are run upon the completion of a Pipeline’s or stage’s run (depending on the location of the post section within the Pipeline).

* **Always:** Run the steps in the post section regardless of the completion status of the Pipeline’s or stage’s run.
* **Aborted:** Only run the steps in post if the current Pipeline’s run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.
* **Failure:** Only run the steps in post if the current Pipeline’s or stage’s run has a "failed" status, typically denoted by red in the web UI.
* **Success:** Only run the steps in post if the current Pipeline’s or stage’s run has a "success" status, typically denoted by blue or green in the web UI.
* **Cleanup:** Run the steps in this post condition after every other post condition has been evaluated, regardless of the Pipeline or stage’s status.

When:

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition.

* **allOf:** Execute the stage when all of the nested conditions are true. Must contain at least one condition
* **anyOf:** Execute the stage when at least one of the nested conditions is true. Must contain at least one condition

**Shared libraries:**

As Pipeline is adopted for more and more projects in an organization, common patterns are likely to emerge. Oftentimes it is useful to share parts of Pipelines between various projects to reduce redundancies and keep code "DRY", no can able to see the our code.

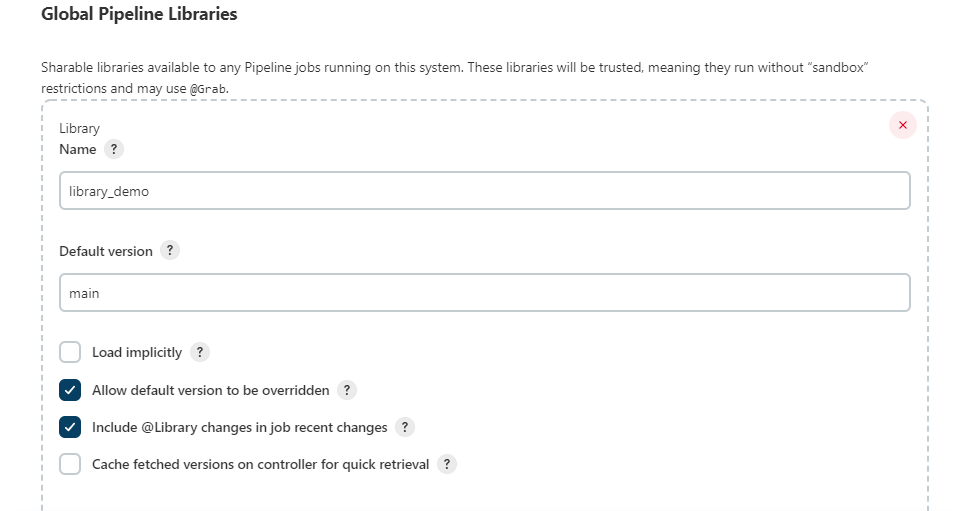
Here we can create the repository in SCM, here we took GitHub as SCM, we created the repo in GitHub and added the files in vars folder,

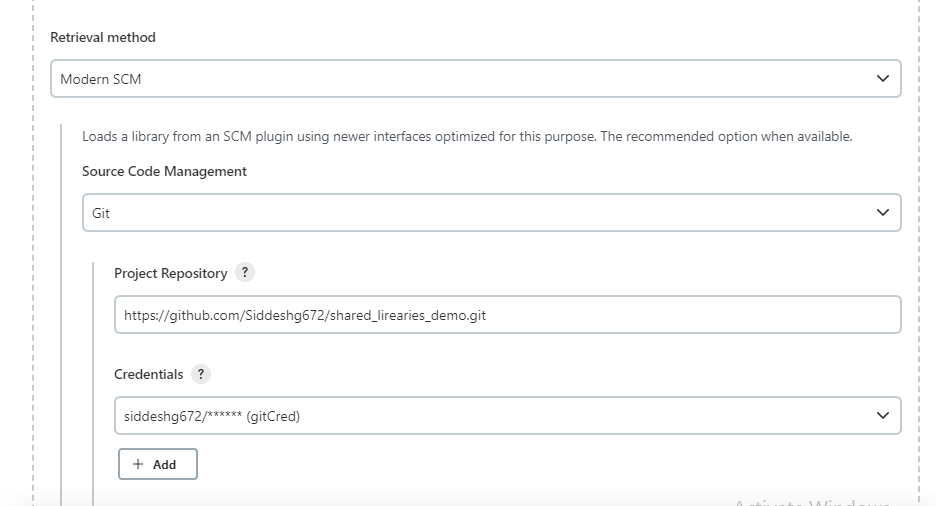
Folder use cases with respect to Git repo.

* src : The src directory should look like standard Java source directory structure
* vars: The vars directory should contains the main shared library scripts.

Once done with the creation of GitHub repo integrate the shared library repo with Jenkins.

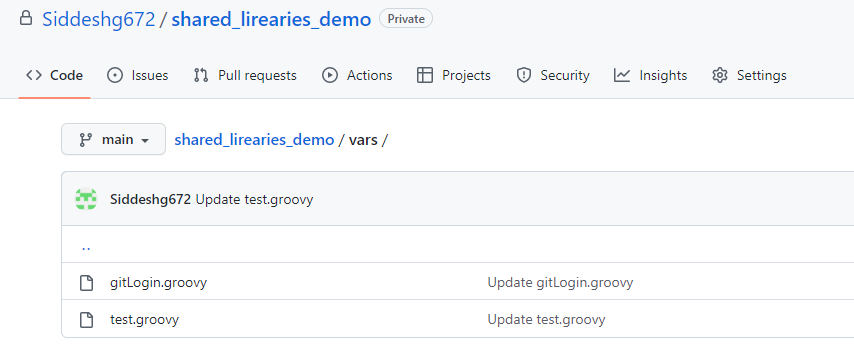
Fallow the below steps to integrate with Jenkins:

* Go to manage Jenkins 🡪 configure system 🡪 Global pipeline libraries
* Fille the required details like
  + Name: name of library
  + Default version: git branch name
  + Select modern SCM in retrieval method
  + Provide the GitHub repo url and select the credential ID which is stored in Jenkins workspace to access git repo.
  + Below are some sn



Once done with the integration check the create the file in github, the files shold be created inside the vars folder and file name should fallow the syntax like nameoflirary.groovy.

Below is some sample file



**Import in pipeline:**

We should import the libarries with the blow keyword

@Library('library\_demo')

And we can call the library like below

stage('Hello') {

steps {

script{

echo 'Hello World'

gitLogin.gitStatus("${branchname}")

gitLogin.gitStatusLogin("${branchname}","${giturl}")

test.docker\_push("siddesh","passwsord","nginx:latest","siddeshg672")

}

}

}