1. **Naming Convention**

Project folders

Business Unit 🡪 Meaningful Project name 🡪 1. Builds

2. Deployment

Example-

Credco 🡪 FCRA 🡪 1. Builds

2. Deployment

Jobs within Builds folder-

Projectname\_Build

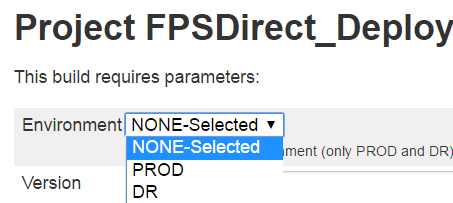
(Trunk and Branch names can be parameterized in the job)

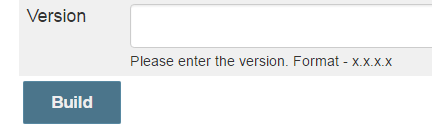
Jobs within Deployment folder-

Projectname\_Deploy

(Environment and Version number can be parameterized in the job)

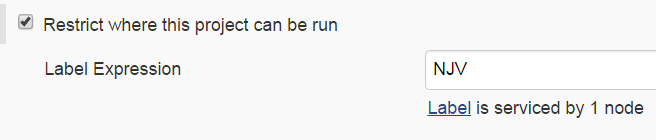
For example-





1. **Use Label in job**

Always use Label name instead of the node name in the “Label Expression” section of the job.



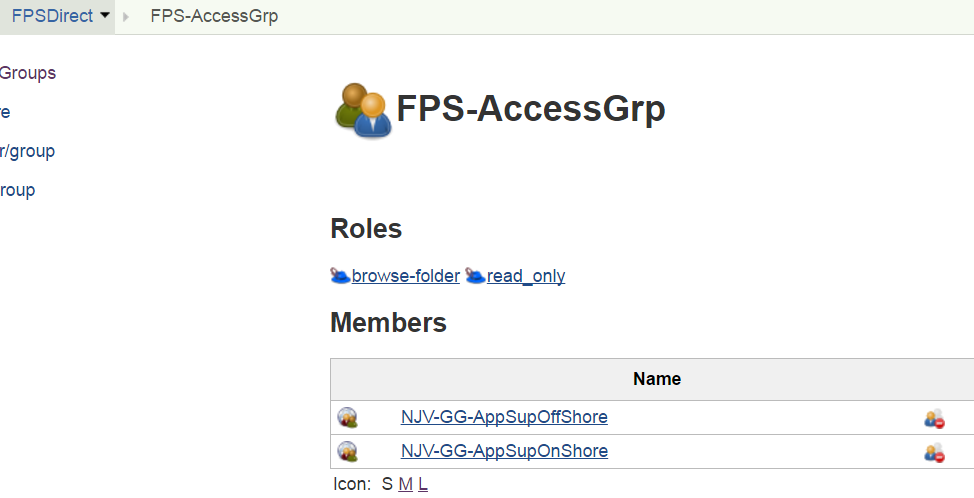
1. **While restarting Jenkins**
2. Drop a mail beforehand to inform everyone regarding Jenkins restart.
3. Go to Manage Jenkins 🡪 Restart Safely

This will restart Jenkins once no jobs are running.

1. Once Jenkins is up, inform everyone over the same mail.
2. **Groups access**

Always grant access to AD Groups instead of individual users for accessing specific projects or jobs.

For example-

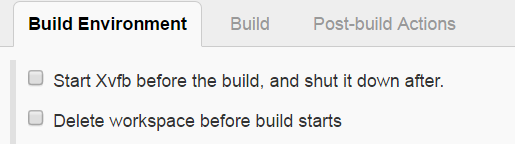


1. **Workspace cleanup**

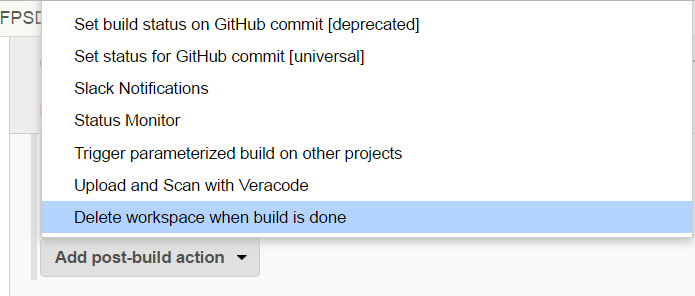
Always cleanup workspace through Jenkinsfile or Freestyle job.

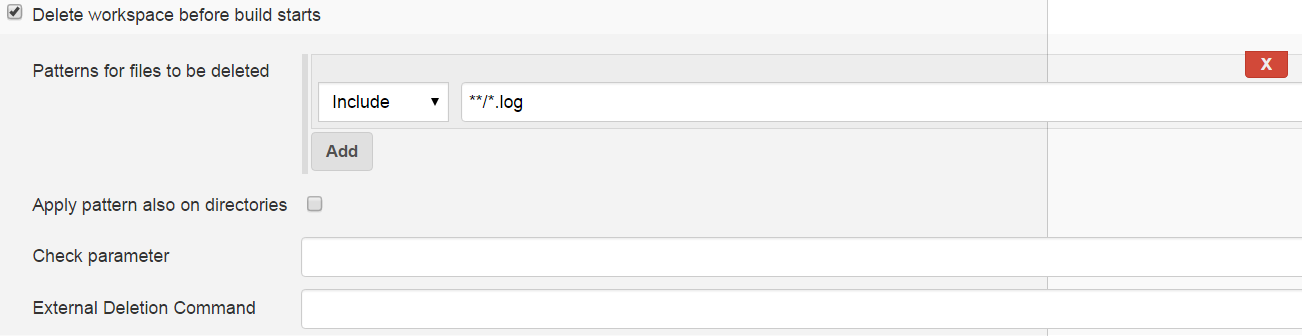
**Steps to clean up workspace using freestyle job-**

There is a “Workspace Cleanup Plugin” installed in Jkci which (when used) deletes the workspace before the build or when a build is finished and artifacts saved.  
Option for deleting workspace before build is in Build Environment section: “Delete workspace before build starts”.



Option for deleting workspace **after** is in **Post-build Actions** section:





**Steps to clean up workspace using jenkinsfile (pipeline job)-**

We need to use [deleteDir()](https://jenkins.io/doc/pipeline/steps/workflow-basic-steps/" \l "deletedir-recursively-delete-the-current-directory-from-the-workspace) as the last step of the pipeline Jenkinsfile.

The deleteDir function recursively deletes the current directory and its contents.

To delete a specific directory of a workspace wrap the deleteDir step in a dir step within the pipeline script -

dir('directoryToDelete') {

deleteDir()

}

1. **Jenkins Pipeline Best Practices**

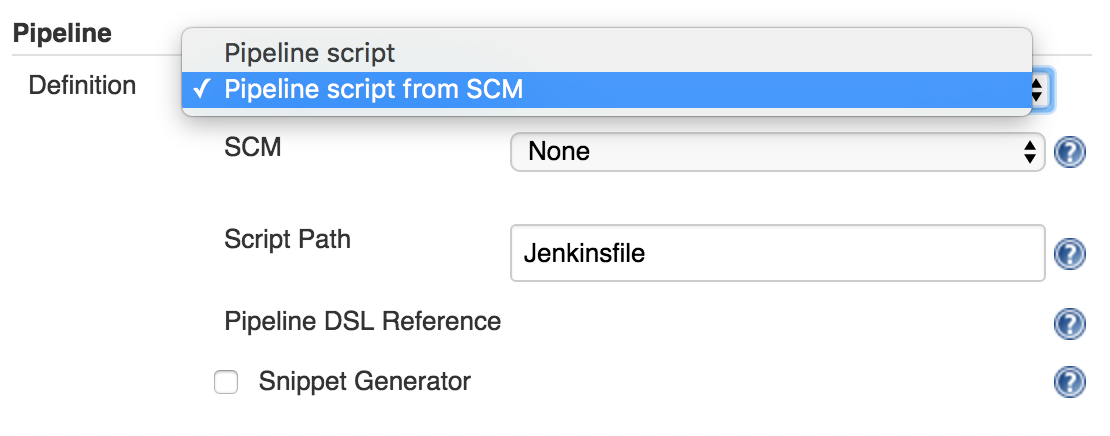
***1. Use the real Jenkins Pipeline***

Don’t use older plugins like Build Pipeline plugin or Buildflow plugin. Instead, use the real [Jenkins Pipeline suite of plugins](https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin).

This is because, unlike freestyle jobs, Pipeline is resilient to Jenkins master restarts and also has built-in features that supersede many older plugins previously used to build multi-step, complex delivery pipelines.

***2. Develop your pipeline as code***

Use the feature to store your Jenkinsfile in SCM. This is because treating your pipeline as code enforces good discipline.



You should also call your Pipeline script the default name: Jenkinsfile and start the following script header, so your IDE, GitHub and other tooling recognize it as Groovy and enable code highlighting:

#!groovy​

​

***3. All work within a stage***

Stages are the logical segmentation of a pipeline. Separating work into stages allows separating your pipeline into distinct segments of work.

Any non-setup work within your pipeline should occur within a stage block.

Example:

stage 'build'

//build

stage 'test'

//test

The Pipeline Stage View plugin visualizes stages as unique segments of the pipeline:



***4. All material work within a node***

Any material work within a pipeline should occur within a node block.

By default, the Jenkinsfile script itself runs on the Jenkins master, using a lightweight executor expected to use very few resources. Any material work, like cloning code from a Git server or compiling a Java application, should leverage Jenkins distributed builds capability and run on an agent node.

Example:

stage 'build'

node{

checkout scm

sh 'mvn clean install'

}

***5. Work within a parallel step***

Pipeline offers a straight-forward syntax for branching your pipeline into parallel steps.

Branching work in parallel will allow your pipeline to run faster, shifting your pipeline steps to the left, and getting feedback to developers and the rest of your team faster.

Example:

parallel 'shifting':{

//everything

}, 'left':{

//I can

}

Use the [Parallel Test Executor plugin](https://wiki.jenkins-ci.org/display/JENKINS/Parallel+Test+Executor+Plugin) to have Jenkins automatically determine how to run your tests in optimally parallel buckets.

***6. Acquire nodes within parallel steps***

You should generally aim to acquire a node within the parallel branches of your pipeline.

Example:

parallel 'integration-tests':{

node('mvn-3.3'){ ... }

}, 'functional-tests':{

node('selenium'){ ... }

}

***7. Don’t use input within a node block***

While you can put an input statement within a node block, you definitely shouldn’t.

The input element pauses pipeline execution to wait for an approval - either automated or manual. Naturally these approvals could take some time. The node element, on the other hand, acquires and holds a lock on a workspace and heavy weight Jenkins executor - an expensive resource to hold onto while pausing for input.

So, create your inputs outside your nodes.

Example:

stage 'deployment'

input 'Do you approve deployment?'

node{

//deploy the things

}

***8. Wrap your inputs in a timeout***

Pipeline has an easy mechanism for timing out any given step of your pipeline. As a best practice, you should always plan for timeouts around your inputs.

Wrapping your inputs in a timeout will allow them to be cleaned-up (i.e., aborted) if approvals don’t occur within a given window.

Example:

timeout(time:5, unit:'DAYS') {

input message:'Approve deployment?', submitter: 'it-ops'

}

***9. Don’t set environment variables with env global variable***

While you can edit some settings in the env global variable, you should use the withEnv syntax instead because the env variable is global, changing it directly is discouraged as it changes the environment globally, so the withEnv syntax is recommended.

Example:

withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {

sh "mvn clean verify"

}