**Getting Started**

**What is a Pipeline?**

Pipelines are a series of steps that allow you to orchestrate the work required to build, test and deploy applications. Pipelines are defined in a file called Jenkinsfile that is stored in the root of your project’s source repository.

# Quick Start

## Python

pipeline {

agent { docker 'python:3.5.1' }

stages {

stage('build') {

steps {

sh 'pip --version'

sh 'python --version'

}

}

}

}

# Create your first Pipeline

To get started quickly with Pipeline (assuming you have already downloaded and installed Jenkins):

1. Copy one of the examples into your repository and name it Jenkinsfile
2. Click the **New Item** menu within Jenkins
3. Provide a name for your new item (e.g., **My Pipeline**) and select **Multibranch Pipeline**
4. Click the **Add Source** button, choose the type of repository you want to use, and fill in the details
5. Click the **Save** button and watch your first Pipeline run!

You may need to modify one of the example Jenkinsfile`s to make it run with your project. Try modifying the `sh command to run the same command you would run on your local machine.

After you have set up your Pipeline, Jenkins will automatically detect any new Branches or Pull Requests that are created in your repository and start running Pipelines for them.

# Running multiple steps

Pipelines are made up of multiple steps that allow you to build, test and deploy applications. Jenkins Pipeline allows you to compose multiple steps in an easy way that can help you model any sort of automation process.

## Multiple Steps

Think of a step as like a single command that does a single action. When a step succeeds it moves onto the next step. When a step fails to execute correctly the Pipeline will fail.

When all the steps in the Pipeline have successfully completed, the Pipeline is considered successfully executed.

### Linux, BSD and macOS

The sh step is used to execute a shell command.

pipeline {

agent { label 'master' }

stages {

stage('build') {

steps {

sh 'echo step1'

sh 'echo step2'

sh '''

echo 'Multiline'

echo 'Example'

'''

echo 'not using shell'

}

}

}

}

### Windows

The bat step is used to execute a batch command.

pipeline {

agent { label 'master' }

stages {

stage('build') {

steps {

bat 'echo step1'

bat 'echo step2'

bat '''

echo 'Multiline'

echo 'Example'

'''

echo 'not using shell'

}

}

}

}

## Timeouts, retries and more

There are some powerful steps that "wrap" other steps that can easily solve problems like retrying (retry) steps until successful or bailing out if a step takes too long (timeout).

pipeline {

agent { label 'master' }

stages {

stage('deploy') {

steps {

retry(3) {

sh 'echo deploying...'

}

timeout(time: 3, unit: 'MINUTES') {

sh 'echo checking health...'

}

}

}

}

}

The deploy stage retries a script 3 times, and then waits for up to 3 minutes for the "health check" script to run. If it did not complete in 3 minutes, the pipeline will have failed at the "deploy" stage.

Steps like timeout and retry contain other steps and you can have any steps run within them as you like - even timeout and retry.

We can compose these steps together. For example, if we wanted to retry our deployment 5 times, but never want to spend more than 3 minutes before failing the pipeline stage:

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh 'echo deploying'

sh 'echo verifying...'

}

}

## Cleaning up after yourself

When the build has run, you may need to run steps to clean up, or take some action based on the outcome. This is where the post section comes in handy:

pipeline {

agent { label 'master' }

stages {

stage('test') {

steps {

sh 'fail me please'

}

}

}

post {

always {

sh 'This will always run'

}

success {

sh 'This will run only if successful'

}

failure {

sh 'This will run only if failed'

}

unstable {

sh 'This will run only if the run was marked as unstable'

}

changed {

sh 'This will run only if the state of the Pipeline has changed')

sh 'For example, the Pipeline was previously failing but is now successful'

}

}

}

You can also run cleanup steps per stage as well if you like:

pipeline {

agent { label 'master' }

stages {

stage('test') {

post {

always {

sh 'This will always run'

}

failure {

sh 'This will run only if failed'

}

changed {

sh 'This will run only if the state of the Pipeline has changed')

}

}

steps {

sh 'fail me please'

}

}

}

}

# Controlling your build environment

In the quick start examples, you will have seen the agent directive in use. This specifies where and how Jenkins will run your build. This is required at the top level and can also be used to override the top-level agent settings for individual stages.

There are really 3 ways to use this (from quickest to most powerful):

## 1. With Docker

Use agent { docker 'node:6.0' to specify that you want Jenkins to pull the node:6.0 image from Docker Hub and run the build inside it. This can pretty much be any image you like; your build steps and stages will run inside it. }

Example:

pipeline {

agent {

docker 'node'

}

stages {

stage("testing 123") {

steps {

sh 'node --version'

}

}

}

}

You can also use a Dockerfile in your source repository - with agent { dockerfile true }, the Dockerfile at the root of your repository will be built and the resulting image will be used for a container your build will run in.

### Reusing node/workspace with per-stage Docker agents

Additionally, if you're running a Docker agent for a specific stage while specifying agent { label 'whatever' } at the top level, you can ensure that this stage will use the same node and workspace as the rest of the Pipeline:

pipeline {

agent {

label 'whatever'

}

stages {

stage('build') {

steps {

sh "./build-artifact.sh"

}

}

stage('test in docker') {

agent {

docker {

image 'ubuntu:16.04'

reuseNode true

}

}

steps {

sh "./run-tests-in-docker.sh"

}

}

}

}

## 2. Specifying a Label for a Jenkins agent to run on

This is useful for when you want to specify a particular machine size, or requirement. Use agent { label 'windows' } to specify that you want your build to run on an agent that is labelled as windows. If you make this label '' then it will run on any agent connected to your Jenkins master that is available.

Example (will use the master node):

pipeline {

agent {

label 'master'

}

stages {

stage('testing 123') {

steps {

sh 'echo hello from master node'

}

}

}

}

## 3. I like to drive stick, get out of my way

In this case, put agent none as your agent directive. Pipeline won't try to find an agent to run on, or pull any Docker images. Inside your stage ... directives, you will need to specify what node to run on, should you need one.

Example:

pipeline {

agent none

stages {

stage('say hi') {

steps {

echo "I don't need no node"

}

}

stage('build') {

agent {

label 'master'

}

steps {

checkout scm

sh 'echo from master'

}

}

stage('deploy') {

agent {

label 'deploy-host'

}

steps {

sh './deploy-code-here'

}

}

}

}

In this case, we have a few stages. The last 2 actually get different nodes to execute on (a node is a machine that is running the Jenkins agent). Note the checkout scm step - this actually fetches the code into the nodes workspace (previously it was done automatically for you).

# Environment variables

Environment variables are set using name/value pairs:

pipeline {

environment {

FOO = "BAR"

}

agent { label "master" }

stages {

stage("foo") {

steps {

sh 'echo "FOO is $FOO"'

}

}

}

}

You can add many more variables, one per line, like so:

environment {

FOO = "BAR"

BAZ = "bang"

}

Environment variables can be set globally (like the above) or per stage. If they are set per stage, they only apply to the running stage. Other stages will not get those environment variables.

You can also access credentials (ie. secrets) and set them to environment variables. For example:

environment {

AWS\_ACCESS\_KEY\_ID = credentials('AWS\_ACCESS\_KEY\_ID')

AWS\_SECRET\_ACCESS\_KEY = credentials('AWS\_SECRET\_ACCESS\_KEY')

}

This looks up the *secret text* for each credential id.

If you use username and password style of credentials, this is slightly different:

environment {

SAUCE\_ACCESS = credentials('sauce-lab-dev')

}

This will actually set 3 variables:

* SAUCE\_ACCESS containing <username>:<password>
* SAUCE\_ACCESS\_USR containing the username
* SAUCE\_ACCESS\_PSW containing the password

# Reporting test results and storing artifacts

If you have a test runner (such as maven surefire, or many of the javascript runners) chances are it can output the junit/xml style of test result reports. If so, recording them is easy:

post {

always {

junit "path/to/xml"

}

}

This will always grab the test results and let Jenkins track them, calculate trends, and report on them. Jenkins has plugins for pretty much all other types of test runners too. A build that has failing tests is marked as UNSTABLE (to separate it from FAILED).

If you want to store an artifact for downloading later (i.e., keep a record of it):

post {

success {

archive "target/\*\*/\*"

}

}

You can also run actions, such as notifications, artifact archiving, or any arbitrary step, after individual stages:

stage('some-stage') {

steps {

sh 'do-something.sh'

}

post {

failure {

sh 'rollback.sh'

}

}

}

A more complete example:

pipeline {

agent { docker "java" }

stages {

stage("build") {

steps {

sh 'mvn clean install -Dmaven.test.failure.ignore=true'

}

}

}

post {

always {

archive "target/\*\*/\*"

junit 'target/surefire-reports/\*.xml'

}

}

}

# Notifications

email, HipChat, slack, IRC

The post section of a pipeline can define both where to send a notification, and on what event. Because notifications are steps, making use of Jenkins plugins to send them is doable. In this example, using email:

post {

success {

mail to:"me@example.com", subject:"SUCCESS: ${currentBuild.fullDisplayName}", body: "Yay, we passed."

}

failure {

mail to:"me@example.com", subject:"FAILURE: ${currentBuild.fullDisplayName}", body: "Boo, we failed."

}

unstable {

mail to:"me@example.com", subject:"UNSTABLE: ${currentBuild.fullDisplayName}", body: "Huh, we're unstable."

}

changed {

mail to:"me@example.com", subject:"CHANGED: ${currentBuild.fullDisplayName}", body: "Wow, our status changed!"

}

}

There is also HipChat, Slack, IRC and more (even Yo has a notifier).

(Seriously: <https://github.com/jenkinsci/notify-yo-plugin> - it won't work in current form with pipeline, however, and I am not even sure if Yo still exists as an app).

You can put notifications as top-level declarations (inside pipeline) or on a per stage basis. For example:

pipeline {

agent any

stages {

stage('build') {

post {

success {

mail to:"me@example.com", subject:"SUCCESS: ${currentBuild.fullDisplayName}", body: "Yay."

}

}

steps {

sh 'make install'

}

}

}

}

# Deployment

As your pipeline grows to accommodate deployment, you will have multiple stages. You have likely already seen stages in use in some examples:

## Multiple stages

Adding more stages makes clear what your pipeline is doing.

pipeline {

agent {

docker 'node'

}

stages {

stage('build') {

steps {

sh 'echo building...'

}

}

stage('test') {

steps {

sh 'echo testing...'

}

}

stage('deploy') {

steps {

retry(3) {

sh 'echo deploying...'

}

}

}

}

}

This is a pipeline of build->test->deploy. The deploy stage actually retries a script 3 times.

Retrying things, with a timeout, is a pretty common pattern in deployment:

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh 'echo deploying'

}

}

This gives it 5 chances to get it right, but in 3 minutes (if your app isn't deployed in 3 minutes, it's free!).

You can also sleep between steps:

stage('deploy') {

steps {

sh 'echo deploying...'

sleep 30

sh 'echo 30 seconds should be long enough'

}

}

Default is in seconds.

## Stages as deploy environments

One useful way to use stages is to have them represent deployment environments:

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo building...'

}

}

stage('Test') {

steps {

sh 'echo testing...'

}

}

stage('Deploy - Staging') {

steps {

sh 'echo deploying to staging...'

sh 'echo smoke tests...'

}

}

stage('Deploy - Production') {

steps {

sh 'echo deploying to production...'

}

}

}

}

The final 2 stages represent staging and production environments. Typically you would want to run some smoke tests before things are promoted to the production environment.

## Asking for human input to proceed

Often when passing between stages, especially environment stages, you may want a human to judge whether the application is in a good enough state to promote (you can also ask for input parameters too, but I won't show that here):

pipeline {

agent none

stages {

stage('Deploy - Staging') {

agent {

label 'master'

}

steps {

checkout scm

sh 'echo deploying $APP\_NAME to staging...'

}

}

stage('Sanity check') {

steps {

input "Does the staging environment for ${env.APP\_NAME} look ok?"

}

}

stage('Deploy - Production') {

agent {

label 'master'

}

steps {

sh 'echo deploying $APP\_NAME to production'

}

}

}

environment {

APP\_NAME = 'my-app'

}

}

In this example, the "Sanity check" stage actually blocks for input and won't proceed without a person allowing it to. You may want to do this if you don't trust your test suite, but don't tell anyone that, it can be our secret.

There are also a few other things going on in the human input example above: i.e., agent none means that is is up to each stage to get its own node to run steps on. This also means that when it is waiting for input from a person, it isn't keeping an agent running (i.e., it can wait for hours or days, with no harm done).

# Deployment targets and techniques

## Deploying a serverless application

To make this a little more concrete, this example builds on the excellent <https://serverless.com/> library, which is a tool to automate a lot of stuff around AWS Lambda (<https://aws.amazon.com/lambda/>).

Let’s dive in to the sample, don't worry if it looks a bit long, it will be explained below:

pipeline {

agent any

stages {

stage('Unit test') {

steps {

sh 'serverless --help' // to ensure it is installed

}

}

stage('Integration test') {

steps {

sh 'serverless deploy --stage dev'

sh 'serverless invoke --stage dev --function hello'

}

}

stage('Production') {

when {

branch "master"

}

steps {

parallel (

'us-east-1' : {

sh 'serverless deploy --stage production --region us-east-1'

sh 'serverless invoke --stage production --region us-east-1 --function hello'

},

'ap-southeast-2' : {

sh 'serverless deploy --stage production --region ap-southeast-2'

sh 'serverless invoke --stage production --region ap-southeast-2 --function hello'

}

)

}

}

stage('Teardown') {

steps {

echo 'No need for DEV environment now, tear it down'

sh 'serverless remove --stage dev'

}

}

}

environment {

AWS\_ACCESS\_KEY\_ID = credentials('AWS\_ACCESS\_KEY\_ID')

AWS\_SECRET\_ACCESS\_KEY = credentials('AWS\_SECRET\_ACCESS\_KEY')

}

}

This requires that the "serverless" tool is installed where the build runs (you could run it anywhere, it runs on all platforms). Critical to this is the environment section that pulls out some credentials - in this case they are 2 "secret text" types of secrets stored encrypted in Jenkins. These are set to environment variables (and masked in any output so logs/etc don't leak secrets).

### Unit tests stage

This is really a placeholder to check that serverless is setup correctly. You could also run some in-container unit tests here as well (there isn't really compiling to do for a javascript app like this, so no compile stage).

### Integration test stage

In this stage we actually push the code to a real AWS region, but in a "dev" capacity (so it doesn't replace production just yet). We then run a "real" test by invoking the "hello" function to check that it works.

### Production stage

In this stage we push in parallel to all the regions we want to support this application in (in this case, just 2) and do a quick "smoke test" in each.

Note this stage only happens on the master branch (by using the when conditional). If you push to any other branch this stage will be skipped.

### Teardown stage

In this stage (should it make it this far without failure) it will remove the dev environment which is used for integration testing (in theory to save money, no need to run code you know you won't need).

### Running this sample

If you want to try this sample for real:

* Get an AWS account (and create some access keys)
* Install the serverless library <https://serverless.com/>
* Setup 2 credentials for the secrets in the environment section
* Fork this repository: <https://github.com/michaelneale/sample-serverless>, and setup a multibranch pipeline pointing to your fork
* You can use the <https://serverless.com/> tool to create your own start serverless app and adapt the above to your needs.

## Other deployment targets

You could see how the patterns established above could be adapted to many deployment targets. Jenkins is used every day to deploy to every known deployment target on earth, including, often, home grown internal deployment tools.

Over time you should be able to find more starters/samples on these docs.

# Parallelism

How to use parallel and why (e.g. for running the same tests over multiple browser platforms)

You can use parallelism inside pipelines for a few reasons:

* Split up a test suite on a powerful machine
* Run tests across different operating systems or environments
* Distribute work across a cluster

The aim of all this is to shorten the length of the pipeline run.

## Split up a test suite

In this case, we will split our tests into two:

pipeline {

agent { label '' }

stages {

stage("test") {

steps {

parallel (

"Firefox" : {

sh "echo testing FFX"

sh "echo more steps"

},

"Chrome" : {

sh "echo testing Chrome"

sh "echo more steps"

}

}

)

}

}

}

This makes use of one single agent and spins off 2 runs of the steps inside each parallel branch, as separate processes.

The steps:

sh "echo testing FFX"

sh "echo more steps"

Are run at the same time as:

sh "echo testing Chrome"

sh "echo more steps"

## Testing across operating systems

Let's run steps on different platforms at the same time:

pipeline {

agent none

stages {

stage("distribute") {

steps {

parallel (

"windows" : {

node('windows') {

bat "print from windows"

}

},

"mac" : {

node('osx') {

sh "echo from mac"

}

},

"linux" : {

node('linux') {

sh "echo from linux"

}

}

}

)

}

}

}

The key difference here is that there is image none and then in each parallel section, we use node to go and fetch an agent that suits and run steps on it remotely.

## Distributing work

You don't have to use different node labels. You can use the same one as many times as you like and Jenkins will try to schedule the work on free agents that are suitable:

"Browser Testing" : {

node('linux') {

sh "echo browser testing"

}

},

"Unit Testing" : {

node('linux') {

sh "echo unit testing"

}

}

There is a lot more you can do with parallelism, but this is a start.

Note that parallelism happens inside a running pipeline - you can also have multiple instances of a pipeline running concurrently, even if you don't use parallel.

# Trigger runs

**Triggers**

Runs of a pipeline are normally triggered by a SCM change. However, you can specify other triggers like a cron-based schedule:

pipeline {

agent any

triggers {

cron '@daily'

}

...

}

This will run the pipeline once per day. The Unix cron style syntax is supported.

**Other available triggers**

Any trigger with a @Symbol on its class is available - if you put in an invalid trigger name, you'll get a validation error telling you that it's invalid and listing other possible triggers. Noteworthy triggers include:

* upstream 'project-name,other-project-name', hudson.model.Result.SUCCESS - run after either of the other projects have built successfully.
* pollSCM '@hourly' - poll SCM for changes hourly.
  + *NOTE*: this trigger is only properly available in Jenkins 2.22 or later. Previously, it used the scm symbol, which did not work due to Pipeline internals.

**Additional repositories**

A pipeline, even one that is defined a Jenkinsfile, can also bring in other source repositories as needed.

For example:

pipeline {

agent none

stages {

stage('build') {

agent any

steps {

// run main build here and collect the results via "stashing" for later use

stash includes: 'app/\*.py', name: 'stuff'

}

}

stage('testing') {

agent { label 'master' }

steps {

// use a different repo to get the test suite:

git 'https://github.com/jenkinsci/acceptance-test-harness'

// we unstash our stuff from before (because this may be a different node)

unstash 'stuff'

sh 'run-test-suite.sh'

}

}

}

}

Jenkins will track changes from all repositories you mention this way and allow the pipeline run to be triggered accordingly.

# Parametrized pipelines

You can define pipelines as requiring some parameters (from the user, or from an api call) before they can start. These parameters can control what the pipeline does, for example what environment it may be deploying applications into.

# Parameters

Valid parameter types are booleanParam, choice, file, text, password, run, or string.

pipeline {

agent any

parameters {

booleanParam(defaultValue: true, description: '', name: 'userFlag')

}

stages {

stage("foo") {

steps {

echo "flag: ${params.userFlag}"

}

}

}

}

This will ask for a true/false value when the pipeline is run.

You can ask for string input:

string(defaultValue: true, description: '', name: 'userFlag')

You can also ask for multiple choice items, and stack up the input required:

parameters {

string(defaultValue: "TEST", description: 'What environment?', name: 'userFlag')

choice(choices: ['US-EAST-1', 'US-WEST-2'], description: 'What AWS region?', name: 'region')

}

Although parameters are available in env they currently are created before the first time the pipeline is run, therefore you should access them via params:

sh "echo ${params.region}"

They might eventually be available in env. ([JENKINS-40241](https://issues.jenkins-ci.org/browse/JENKINS-40241))

# Pipeline properties

One important option you may want to set up is the log rotation, meaning how long pipeline runs are retained for:

pipeline {

agent any

options {

buildDiscarder(logRotator(numToKeepStr:'5'))

}

...

}

There are other advanced properties that you may need to set from time to time - they go in this section. It is likely you won't need more than what is shown above.

# Validating (or linting) a Declarative Jenkinsfile from the command line

See <https://jenkins.io/doc/book/pipeline/development/#linter>.

# Advanced

The pipeline you have seen so far has been mostly declarative. You can do a lot more with pipeline (in fact there isn't really anything you can't do, if you are careful).

# Using pipeline scripting inside stages

If you need to use lower-level pipeline script inside stages, you can use script - this is a powerful tool not to be used lightly. The script block allows you to use the full power of pipeline.

pipeline {

agent any

stages {

stage('build') {

steps {

sh 'echo hello world'

script {

def thing = someLibrary.someMethod("hi");

.. do something with thing..

.. with great power comes great responsibility ...

}

}

}

}

}

# Wrapping steps around pipelines

You can use block-scoped steps like retry or timeout in a stage, but you can also use those steps across the whole build using the options section.

pipeline {

agent any

options {

retry(3)

timeout time:5, units:'MINUTES'

}

stages {

stage('build') {

sh 'run-build.sh'

}

}

}

# Controlling when stages execute

Using the when construct, you can specify a true/false condition. Depending on the outcome of this condition, the stage will be "skipped".

The most common usage of this would be to choose what branch to run a stage on:

stage('production') {

when {

branch "master"

}

steps ...

}

You can use many other environment variables to programmatically decide if a stage should run.

# Breaking out of declarative pipeline

Any script you put outside of the pipeline construct will be the same as if it were in a script block. This is non-declarative Groovy CPS DSL that pipeline uses under the covers. If those words sound scary, you probably don't want to do this. With great power comes great responsibility. There are a lot fewer restrictions and validations when you use script outside of pipeline. There is also more verbosity to do common things...

# Finding what steps are available

To see what steps are available, if you go to the page for a pipeline in Jenkins, on the left there will be a link that says "Pipeline Syntax" - click on that and you get a syntax builder that can enumerate what other steps (other than sh) are available on your Jenkins instance (normally installed via plugins).

# Read more

TODO: where should this link to?

# Syntax Reference

// This is the full syntax for Jenkins Declarative Pipelines as of version 0.8.1.

pipeline {

// Possible agent configurations - you must have one and only one at the top level.

agent any

agent none

agent {

label "whatever"

}

// Note that you can define a default label for docker and dockerfile in either your Jenkins

// global configuration, or in the configuration for a folder, like a GitHub Organization Folder.

// If that's defined, that label will be used when you don't specify one here explicitly.

agent {

docker "ubuntu:16.04"

}

agent {

dockerfile true

}

agent {

docker {

image "ubuntu:16.04"

label "docker-nodes"

args "-v /tmp:/tmp"

}

}

agent {

dockerfile {

filename "someOtherDockerfile"

label "docker-nodes"

args "-v /tmp:/tmp"

}

}

// Environment

environment {

FOO = "bar"

OTHER = "${FOO}baz"

SOME\_CREDENTIALS = credentials('some-id')

}

// Tools - only works when \*not\* on docker or dockerfile agent

tools {

// Symbol for tool type and then name of configured tool installation

maven "maven3.3.9"

jdk "jdk8"

}

options {

// General Jenkins job properties

buildDiscarder(logRotator(numToKeepStr:'1'))

// Declarative-specific options

skipDefaultCheckout()

// "wrapper" steps that should wrap the entire build execution

timestamps()

timeout(time: 5, unit: 'MINUTES')

}

triggers {

cron('@daily')

}

// Access parameters with 'params.PARAM\_NAME' - that'll get you default values too.

parameters {

booleanParam(defaultValue: true, description: '', name: 'flag')

// Newer core versions support "stringParam" as well

string(defaultValue: '', description: '', name: 'SOME\_STRING')

}

stages {

stage("first stage") {

// All sections within stage other than steps are optional.

environment {

// Overrides or adds to the existing environment

FOO = "notBar"

}

tools {

// Overrides tools of the same type defined globally

maven "maven3.3.3"

}

agent {

// Overrides the top-level agent. "agent none" at the stage level does nothing.

label "some-other-label"

}

// Conditional execution of this stage - only run this stage if the when condition is true.

when {

// One and only one condition is allowed.

// Only run if the branch matches this Ant-style pattern

branch "master"

// Only run if the environment contains this given variable name with this given value

environment name: "FOO", value: "notBar"

// Only run if this Scripted Pipeline expression doesn't return false or null

expression {

echo "You can run any Pipeline steps in here"

return "foo" == "bar"

}

}

// Runs at the end of the stage, depending on whether the conditions are met.

post {

// always means, well, always run.

always {

echo "Hi there"

}

// changed means when the build status is different than the previous build's status.

changed {

echo "I'm different"

}

// success, failure, unstable all run if the current build status is successful, failed, or unstable, respectively

success {

echo "I succeeded"

archive "\*\*/\*"

}

}

// steps is required and is where you put your stage's actual work

steps {

echo "I'm doing things, I guess."

retry(5) {

echo "Keep trying this if it fails up to 5 times"

}

// the script block allows you to run any arbitrary Pipeline script, even if it doesn't fit the Declarative subset.

script {

if ("sky" == "blue") {

echo "You can't actually do loops or if statements etc in Declarative unless you're in a script block!"

}

}

}

}

stage("second stage") {

steps {

// You can only use the parallel step if it's the \*only\* step in the stage.

parallel(

firstBlock: {

echo "I'm on one parallel block"

},

secondBlock: {

echo "I'm on the other block"

}

)

}

}

}

post {

// always means, well, always run.

always {

echo "Hi there"

}

// changed means when the build status is different than the previous build's status.

changed {

echo "I'm different"

}

// success, failure, unstable all run if the current build status is successful, failed, or unstable, respectively

success {

echo "I succeeded"

archive "\*\*/\*"

}

}

}