Jenkins file

In Declarative pipeline, the Jenkinsfile start with **pipeline** block.

```
pipeline{
```

Agent

Inside pipeline block we have **agent** block and **stages** block.

agent block is used to tell Jenkins where to execute this job.

```
    agent any
```

Options

buildDiscarder - Persist artifacts and console output for the specific number of recent Pipeline runs.

disableConcurrentBuilds - Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources.

master branch keeps 300 builds and 20 artifacts.

```
    options {
        buildDiscarder(logRotator(
            // number of builds to keep
            numToKeepStr:         env.BRANCH_NAME ==~ /master/ ? '300' :
'20'
            ))
        disableConcurrentBuilds()
    }
```

Stages

Inside stage block we should have at least one **stage** block

stage block is used to group the set of tasks.

```
    stages {
      stage('Build'){
```

Inside stage block we should have **steps** block

steps block is used to group the step

```
          steps {
```

Inside steps block we should have at least one **step (inbuild function name)**

step is the basic unit which executes the command.

- **sh** step to execute any shell commands
- **echo** step to print some data

```
            script {

              //get committer's details and github repo name
            env.GET_BRANCH = sh(returnStdout: true, script: "echo
'${env.BRANCH_NAME}' | cut -f1 -d '-'").trim()
            branch_name = "${env.GET_BRANCH}"
                    committerEmail = sh (script: 'git --no-pager show -s -
-format=\'%ae\'', returnStdout: true).trim()
                commiterName = sh (script: 'git show -s --pretty=%an | awk
\'{print $1}\'', returnStdout: true).trim()
                reponame = sh (script: 'basename -s .git `git config --get
remote.origin.url`', returnStdout: true).trim()
            build_team = '$DEFAULT_RECIPIENTS'
```

Execute on the master branch

- **=~ (Find operator)** creates a matcher against the env.BRANCH_NAME variable, using the pattern on the right hand side, the return type of =~ is a Matcher

- print: 'Working on release branch: ${env.BRANCH_NAME}'
- run prepare.sh file with variable ${env.BRANCH_NAME}
- go to master directory
- run gradlew file

```
              if (env.BRANCH_NAME =~ '^master$|^avengers$') {

              //Initiate build process for main branches
                sh """
                  echo 'Working on release branch: ${env.BRANCH_NAME}'
                  sh prepare.sh ${env.BRANCH_NAME}
                  cd master
                  sh gradlew clean build publish bvt publishImage
                  sh gradlew clean
                """
```

Execute on federal branch

The only difference between master branch and federal branch is we printed different message

- print: 'Working on gmv branch: ${env.BRANCH_NAME}'
- run prepare.sh file with variable ${env.BRANCH_NAME}
- go to master directory
- run gradlew file

```
            } else if (env.BRANCH_NAME =~ 'gmv-federal$') {
              sh """
                echo 'Working on gmv branch: ${env.BRANCH_NAME}'
                sh prepare.sh $branch_name
                cd master
                sh gradlew clean build publish bvt publishImage
                sh gradlew clean
              """
```

Execute elsewhere

- print: 'Working on build branch: ${env.BRANCH_NAME}'
- run prepare.sh file with variable default
- go to master directory
- run gradlew file

```
            } else {
              sh """
                echo 'Working on build branch: ${env.BRANCH_NAME}'
                sh prepare.sh default
                cd master
                sh gradlew clean build bvt
                sh gradlew clean
              """
            }
            }
        }
        }
    }
```

Post

The **post** block is inside a pipeline block.

Even if some stages failed, post block will be executed always. In post block we have three important blocks **always**, **success**, **failure**.

- always - if we trigger a job, whether the sage is success or failure, this block will be always executes.
- success - this block will be executed only if all the stages are passed.
- failure - this block will be executed if any one of the stage is failed.

If failed in master branch, mail to committerEmail and build_team

```
post {
       failure {
         script {
           if (env.BRANCH_NAME =~ '^master$|^avengers$') {
             emailext (
               to: "$committerEmail;$build_team",
               subject: "$reponame ${env.BRANCH_NAME} branch
#${env.BUILD_NUMBER} failed",
               body: "Hi $commiterName," + "\n\nBuild failed for
$reponame project, which is blocking the build. Please fix it." +
"\n\nMerged changes:\n" +getChangeString() + "\nDeveloper should fixed
the issues before ask for the code review and project owner should not
merge the changes when branch build fails.\n" +"\nPlease find attached
build failure logs.\n" + "\nRegards," + "\nBuild Team",
               attachLog: true,
             )
```

If failed in other branch, mail to committerEmail only.

```
            } else {
              emailext (
                to: "$committerEmail",
                subject: "$reponame ${env.BRANCH_NAME} branch
#${env.BUILD_NUMBER} failed",
                body: "Hi $commiterName," + "\n\nBuild failed for
$reponame project. Please fix it." + "\n\nChanges:\n" +getChangeString()
+ "\nPlease find attached build failure logs.\n" + "\nRegards," +
"\nBuild Team",
                attachLog: true,
              )
            }
          }
        }
    }
}
```

Find all changed strings and is used to send through emails if any build step failed in post.

```
@NonCPS
def getChangeString() {
    MAX_MSG_LEN = 100
    def changeString = ""

    echo "Gathering SCM changes"
    def changeLogSets = currentBuild.changeSets
    for (int i = 0; i < changeLogSets.size(); i++) {
        def entries = changeLogSets[i].items
        for (int j = 0; j < entries.length; j++) {
            def entry = entries[j]
            truncated_msg = entry.msg.take(MAX_MSG_LEN)
            changeString += " - ${truncated_msg}\n"
        }
    }

    if (!changeString) {
        changeString = " - No new changes"
    }
    return changeString
}
```

Triggers

The **triggers** block is inside a pipeline block.

triggers block is used to re-trigger the job based on the defined triggers, it can be **cron**, **pollSCM** and **upstream**.

- cron - Accepts a cron-style string to define a regular interval at which the Pipeline should be re-triggered, for example:

  ```
  triggers {
      cron('H/15 * * * *')
  }
  ```

  This will trigger the job every fifteen minutes.
- pollSCM - Accepts a cron-style string to define a regular interval at which Jenkins should check for new source changes. If new change exist, the Pipeline will be re-triggered. For example:

  ```
  triggers {
      pollSCM('* * * * *')
  }
  ```

  This will check for new source changes in git repository every one minute.
- upstream - Accepts a comma-separated string of jobs and a threshold. When any job in the string finishes with the minimum threshold, the Pipeline will be re-triggered. For example:

  ```
  triggers {
      upstream(upstreamProjects:'job1,job2', threshold:
  hudson.model.Result.SUCCESS)
  }
  ```

  This will trigger the job, if job1 or job2 finished with success status.

Parameters

The **parameters** block is inside a pipeline block.

The parameters block is used to pass dynamic paramerts/variables to the job. It has the following types:

- **string** - A parameter of a string type, for example:

```
parameters {
    string(name: 'DEPLOY_ENV', defaultValue: 'staging', description:
'')
}
```

- **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')
}
```

Environment

The **environment** block specifies a sequence of key-value pairs which will be defined as environment variables for all steps, or stage-specific steps, depending on where the environment block is located within the Pipeline or within the stage.

The environment block can be inside the pipeline block or inside the stage block.

```
environment {
    NAME = 'vignesh'
}
```

If environment block is defined within the stage block, then those environment variables will be accessible only within that stage.