

Jenkins Pipeline to deploy Angular App to AppEngine GCP



vijay kumar

Nov 20, 2018 · 10 min read

This post is all about creating a Continuous Deployment in Jenkins for Angular App to GCP. This could also be used as a reference to deploy the Nodejs apps to GCP since most of the stages are similar with language specific commands for building, packing and deploying the app

Table of Contents

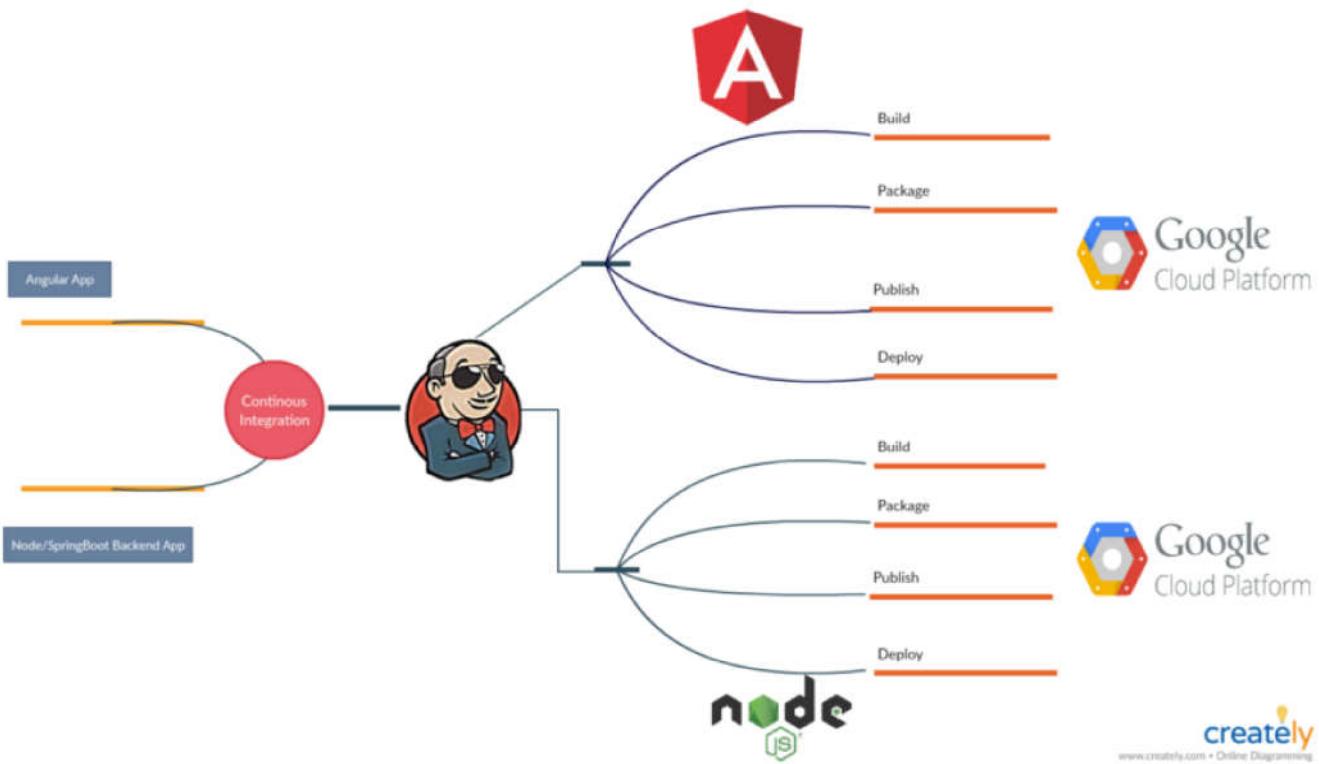
Table of Contents

- A Brief Introduction:
- Why GCP?
- Install Jenkins
- App and Repo Creation
- Pipeline in Jenkins
- * Click on the new item
- * Create a sample pipeline project
- * Describe the pipeline
- * Pipeline script
- CI in Jenkins
- Jenkins Plugins
- Configure Global Tools in Jenkins
- Configure Credentials in Jenkins
- Configure Systems in Jenkins

- Declarative Pipeline
- Understanding Declarative Pipelines
- Initial Declaration
- Stages in Pipeline
- Declaring Deploy stage

A Brief Introduction:

Jenkins is one of the most widely used tools to develop Continuous Integration and Continuous Delivery pipelines. The need for CI/CD cannot be overstated, since the advent of developing agile based applications. That doesn't mean that it should not be used for the more traditional waterfall model though.



Continuous Integration is for continually triggering the phases such as build/report/publish/package after every code commit and package the code ready for deployment. This is mandatory since every time when a piece of code is checked in your repo , say bitbucket, every time a build should be triggered and the app should be bundled.

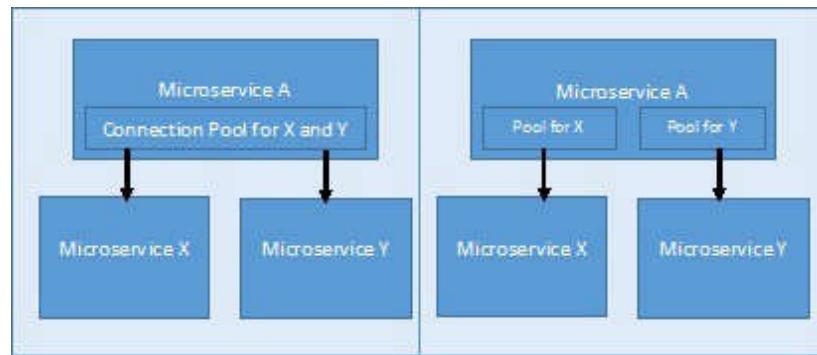
For the QA team to test the applications, they should be available in the sandbox at all times. It is really inefficient to manually deploy the applications to the sandbox/

test environments every time when a patch is made or a bug fix is done.

Enter Jenkins!!!. The world of CICD has been made simple with the introduction of pipeline items, where different phases can be automated to happen after each successful completion of a stage (Build/Package/Publish/Deploy)

Why GCP?

- Micro-services architecture emphasises that each independent modules be hosted as an independent application. GCP supports the micro-services architecture by default where we can have a single project embed multiple applications (which are language agnostic).



- The AppEngine part of GCP supports automatic scaling based on the request throughput and usage
- Kubernetes is supported natively in GCP
- GCP also makes a developer's life much easier by allowing the admin to . SSH into the remote machine and deploy manually.

There are many more advantages like debugging, logging which are supported natively as well as support for a greater set of analytic tools

Install Jenkins

The article in the link explains about installing and using Jenkins in a macOS machine. In a windows or Unix/Linux based machines use the respective package managers to fetch the app. Please follow this link to either startup Jenkins as a standalone or behind a NGINX reverse proxy

App and Repo Creation

I am not going to give an introduction on creating the angular/nodejs/springboot apps and push the code to a remote repo as it is beyond the scope of the article. There are numerous online articles for that . Here are some of the references

Angular App reference:

A Quick Guide to Help you Understand and Create Angular 6 Apps

This post is divided into two parts:

medium.freecodecamp.org

Nodejs App reference

Building a simple REST API with NodeJS and Express.

Have you been working on front-end technologies and been feeling like you're missing out something in the whole...

medium.com

Bitbucket references

Learn Git with Bitbucket Cloud | Atlassian Git Tutorial

Learn Git with Bitbucket Cloud

| Atlassian Git Tutorial Learn Git with Bitbucket Cloud www.atlassian.com

Pipeline in Jenkins

Let us create a sample pipeline project in Jenkins

* Click on the new item

The screenshot shows the Jenkins dashboard. On the left, there is a sidebar with links: 'New Item' (which is highlighted with a red underline), 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', and 'My Views'. The main area has a title bar with 'Jenkins' and a user icon 'vijayakumar'. A search bar and a 'log out' link are also present. Below the title bar, there is a red status bar with the number '2'. The main content area is titled 'test-pipeline-view' and shows a table of pipelines. The table has columns: S, W, Name, Last Success, Last Failure, Last Duration, and Fav. There are four entries in the table:

S	W	Name	Last Success	Last Failure	Last Duration	Fav
1	customName	customName	19 days - #3	N/A	13 sec	
2	myproject	myproject	19 days - #1	N/A	52 sec	
3	project compile	project compile	19 days - #7	N/A	8.4 sec	
4	test	test	17 days - #2	17 days - #2	8 sec	

The screenshot shows the Jenkins dashboard with the following sections:

- Open Blue Ocean**: A link to the Open Blue Ocean plugin.
- Lockable Resources**: A link to lockable resources.
- Credentials**: A link to credentials.
- Splunk**: A link to Splunk integration.
- New View**: A link to create a new view.

Build Queue: No builds in the queue.

Build Executor Status: 1 Idle, 2 Idle.

Eclipse integration is not enabled.

Page generated: 20-Nov-2018 14:18:54 IST REST API Jenkins ver. 2.147

* Create a sample pipeline project

The screenshot shows the Jenkins 'Create New Item' dialog with the following steps:

- Enter an item name**: The input field contains "sample_pipeline".
- Freestyle project**: Description: 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.
- Maven project**: Description: Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Pipeline**: Description: Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job types. This option is selected.
- Multi-configuration project**: Description: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Bitbucket Team/Project**: Description: Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.
- Folder**: Description: 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 workspace, so you can have multiple things of the same name as long as they are in different folders.

GitHub Organization

OK

* Describe the pipeline

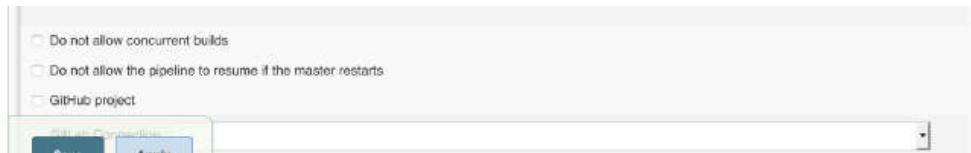
The screenshot shows the Jenkins 'General' configuration page for the "sample_pipeline" project. The General tab is selected, and the configuration includes:

- Description**: Sample pipeline.
- Discard old builds**: Checked.
- Strategy**: Log Rotation.
- Days to keep builds**: 10.
- Max # of builds to keep**: 10.

If not empty, build records are only kept up to this number of days.

If not empty, only up to this number of build records are kept.

Advanced...



Configuring the 'Discard old builds' is optional

* Pipeline script

A screenshot of the Jenkins Pipeline configuration page. The 'Definition' dropdown is set to 'Pipeline script'. The 'Script' field contains the Groovy code: 'try sample Pipeline...'. Below the script field are 'Save' and 'Apply' buttons. A note says 'Use Groovy Sandbox' and 'Pipeline Syntax'. At the bottom, there is a note about Eclipse integration and a footer with page generation details.

Now this is where scripting comes into picture. The user can select the **Definition** of the pipeline to either be a **Pipeline script** or a **Pipeline file** that is stored in SCM

A screenshot of the Jenkins Snippet Generator interface. The left sidebar includes links for Back, Snippet Generator, Declarative Directive Generator, Declarative Online Documentation, Steps Reference, Global Variables Reference, Online Documentation, and IntelliJ IDEA GDSL. The main area shows an 'Overview' of the Snippet Generator, a 'Steps' section with a 'Sample Step' of 'build: Build a job', and configuration options for 'Project to Build' (set to 'test'), 'Wait for completion' (checked), 'Propagate errors' (checked), and 'Quiet period' (empty). Parameters are noted as 'test is not parameterized'.

Snippet Generator

Here we are going to look at Pipeline declarations that are stored and executed directly from SCM. This eliminates the need for maintaining a separate space in Jenkins for the configuration file. The declaration pipeline is also much easier to configure.

CI in Jenkins

Continuous Integration is made possible in Jenkins by means of enabling Webhooks.

- Allow the webhooks plugin for the bitbucket server to be installed in your bitbucket tool and then go to the repository settings

The screenshot shows the Bitbucket repository settings page. On the left, there's a sidebar with actions like Clone, Create branch, Create pull request, Create fork, Source, Commits, Branches, Pull requests, and Repository settings. Under the Repository settings, the Hooks section is highlighted. The main panel is titled 'Repository details' and shows the following configuration:

- Name: XXXX (input field)
- Default branch: master (dropdown menu)
- Allow forks: checked (checkbox)
- Transcode diffs: unchecked (checkbox)
- Large File Storage (LFS): Allow LFS (checkbox)

At the bottom right of the main panel are 'Save' and 'Cancel' buttons.

- Now go to the **Hooks** section and enable the **Bitbucket Server Webhooks to Jenkins**

The screenshot shows the 'Bitbucket Server Webhook to Jenkins' configuration page. It has a header 'Give the Jenkins URL' with a placeholder 'URL to the Jenkins instance. Example: https://jenkins.example.com'. Below it is a 'Repo Clone URL' section with a dropdown set to 'SSH' and a button 'ADD OWN SSH VALUE'. There are several configuration options:

- Skip SSL Certificate Validation**: A checked checkbox with a note: 'When connecting to Jenkins, allow all certificates to be accepted, including self-signed certs'
- Omit SHA1 Hash Code**: An unchecked checkbox with a note: 'Do not send the commit's SHA1 hash code to Jenkins'
- Omit Branch Name**: An unchecked checkbox with a note: 'Do not send the commit's branch name to Jenkins'
- Omit the Trigger Build Button**: An unchecked checkbox

Do not display the Trigger Build Button on the pull request overview page

Configuration check **Trigger Jenkins**

Test the configured connection with Jenkins.

Advanced Configuration

Committers to

Save **Cancel**

- Once the webhook is enabled , it gets reflected in the repo

ACTIONS

- Clone
- Create branch
- Create pull request
- Create fork

NAVIGATION

- Source
- Commits
- Branches
- Pull requests

SHORTCUTS

- Add shortcut

Repository settings

Repository details

Hooks

Add hook

Pre receive

Pre receive hooks allow you to control which commits go into your repository before pushes are committed or pull requests are merged.

- Reject Force Push**
Reject all force pushes (git push --force) to this repository
Inherited (disabled)
- Verify Commit Signature**
Reject commits and tags without a verified GPG signature
Inherited (disabled)
- Verify Committer**
Reject commits not committed by the user pushing to this repository
Inherited (disabled)

Post receive

Post receive hooks can perform actions after commits are processed.

- Bitbucket Server Webhook to Jenkins**
Webhook for notifying a configured endpoint of changes to this repository
Enabled

Webhooks are nothing but functions that run on the web when something is triggered. In our case, we would want the Jenkins pipeline function to be triggered when we push the code to SCM, each time.

- Now go back to Jenkins and enable the **Build Trigger**

General **Build Triggers** Advanced Project Options Pipeline

Bitbucket Pull Requests Builder

Build after other projects are built

Build periodically

Build when a change is pushed to BitBucket

Build when a change is pushed to Coding, WebHook URL: http://127.0.0.1:1081/coding/sample_pipeline

Build when a change is pushed to GitLab, GitLab webhook URL: http://127.0.0.1:1081/project/sample_pipeline

Generic Webhook Trigger

GitHub Branches

GitHub Pull Request Builder

GitHub Pull Requests

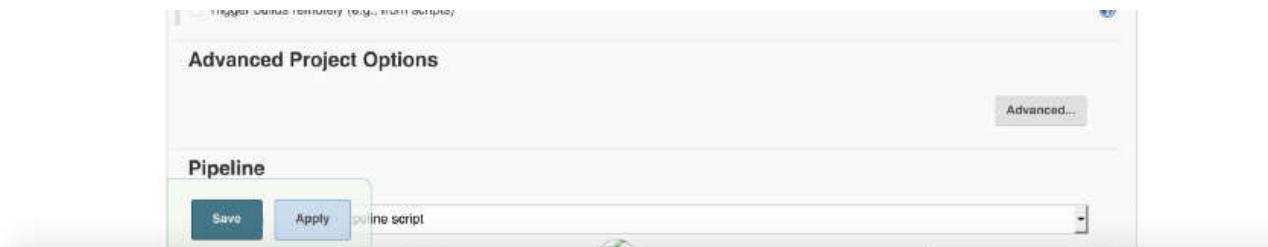
GitHub hook trigger for GITScm polling

Poll SCM

Disable this project

Quiet period

Trigger builds remotely (e.g. from script)



Jenkins Plugins

There will be cases when some of the options in Jenkins may not be available by default. So proper plugins need to be installed for every function.

For example if build is to be triggered on every push to bitbucket, then Bitbucket plugin should be installed first. Here is a gist of all the plugins that I happen to install. Some of the plugins may not be needed for your configuration.

- Go to *Manage Jenkins-> Manage plugins*

Script Security sandbox bypass
GitHub Pull Request Builder 1.42.0:
GitHub access tokens stored in in build.xml

Credentials

Splunk

New View

Build Queue

No builds in the queue.

Build Executor Status

1 Idle
2 Idle

Configure System

Configure global settings and paths.

Configure Global Security

Secure Jenkins; define who is allowed to access/use the system.

Configure Credentials

Configure the credential providers and types

Global Tool Configuration

Configure tools, their locations and automatic installers.

Reload Configuration from Disk

Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

Manage Plugins

Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
⚠ There are updates available

System Information

Displays various environmental information to assist trouble-shooting.

System Log

System log captures output from java.util.logging output related to Jenkins.

Load Statistics

https://localhost:1443/pluginManager

- Go to *Available -> Search the plugins needed and install them. Make sure you select Install without restart every time you wanted a plugin to be installed*

Back to Dashboard

Manage Jenkins

Update Center

Updates Available Installed Advanced

Install ↴

Name	Version
aws-device-farm	1.22
This plugin schedules test runs on AWS Device Farm.	
AWS Lambda	0.5.10
This Plugin allows you to upload a zip file or folder to AWS Lambda	
AWS Elastic Beanstalk Deployment	0.3.19
This Plugin allows you to deploy into one or many AWS Elastic Beanstalk environments.	

AWS.CodeDeploy	This plugin provides a "post-build" step for AWS CodeDeploy.	1.21
jenkins-cloudformation-plugin	Adds a build wrapper that can spawn an AWS Cloud Formation recipe at the start of the build and take it down at the end.	1.2
AWS.SQS.Build.Trigger	This plugin triggers builds on all events received via Amazon Web Services Simple Queue Service (AWS SQS) on a specified Queue.	2.0.1
Amazon.EC2	This plugin integrates Jenkins with Amazon EC2 or anything implementing the EC2 API's such as an Ubuntu.	1.41
Misc (aws)		
AWS.CloudWatch.Logs.Publisher	Publishes build logs to Amazon CloudWatch Logs .	1.2.0
AWS.CodeBuild	Build your project on AWS CodeBuild.	0.32

Here is a gist of all plugins that I installed

Configure Global Tools in Jenkins

These are the building blocks of the pipeline allowing the user to execute build/publish and deploy commands. We are going to make use of three major tools — **Java, Git & Node**. Since Jenkins runs from local, it is important that the local machine has the necessary tools installed in it. The other option is that we can instruct Jenkins to download the tools over the internet.

- Go to *Manage Jenkins -> Global Tool Configuration*



All the three tool configurations are given here

- Configure the executable paths of the tools by locating the bin folder for each

which java

which node

which git

Configure Credentials in Jenkins

- Go to *Jenkins -> Credentials*

A screenshot of the Jenkins Credentials management interface. On the left, there is a sidebar with various Jenkins navigation links like Jenkins, New Item, People, Build History, etc. The main area is titled 'Credentials' and shows a table of stored credentials. The table columns are T, P, Store, Domain, ID, and Name. There are 10 entries listed:

T	P	Store	Domain	ID	Name
Jenkins	(global)	VV		vijayakumar.shanmugam/*****	
Jenkins	(global)	bb2540bb-4782-4393-9944-f0fc640b3931		Vijayakumar_psg587/*****	
Jenkins	(global)	ae5a54f3-f402-44f3-8dbb-c2ad97d7914d		vijayakumar.shanmugam/*****	
Jenkins	(global)	bby-stash		vijayakumar.shanmugam/*****	
Jenkins	(global)	mypriv		vijayakumar_psg587/***** (mypiv)	
Jenkins	(global)	new-bby-passwd		vijayakumar.shanmugam/*****	
Jenkins	(global)	0bba1bf1-a6b1-4c01-a345-59260720cc09		vijayakumar.shanmugam/***** (new-bby-passwd)	
Jenkins	(global)	service_account_key		test-prj-crnd-d83de65910ea.json (service_account_key)	
Jenkins	(global)	usrPasswd		vijayakumar.psg/test587@gmail.com/***** (usrPasswd)	

Below the table, there is a section titled 'Stores scoped to Jenkins' with a table showing Jenkins stores and their domains. A blue 'Add credentials' button is located at the bottom of this section.

- Select domain *global* -> *Add credentials*
- Select *Secret file* option and upload the service account key generated from GCP

The screenshot shows the Jenkins Global credentials (unrestricted) configuration page. In the 'Kind' dropdown menu, 'Secret file' is selected. Other options listed include Username with password, Docker Host Certificate Authentication, GitHub API token, Google Service Account from private key, OpenShift OAuth token, OpenShift Token for OpenShift Sync Plugin, OpenShift Username and Password, and SSH Username with private key.

The screenshot shows the Jenkins Global credentials (unrestricted) configuration page for a 'Secret file' credential. The 'Kind' field is set to 'Secret file'. The 'Scope' field is set to 'Global (Jenkins, nodes, items, all child items, etc.)'. The 'File' field contains the path 'Browse... test-prj-cmd-d83de65910ea.json'. The 'ID' field is set to 'service_account_key1'. The 'Description' field is set to 'service_account_key1'.

Make sure the Credentials and the Plain Credentials plugin are installed

Configure Systems in Jenkins

The next step would be to configure the systems such as the slack channels that Jenkins interacts with, Also specify the Artifactory where the bundled apps will be published to, most preferably your Jfrog artifactory

- Go to *Manage Jenkins-> Configure System*

The screenshot shows the Jenkins 'Configure System' page. On the left, there's a sidebar with various Jenkins management links like 'New Item', 'People', 'Build History', etc. The main area is titled 'Maven Project Configuration' and contains fields for 'Global MAVEN_OPTS', 'Local Maven Repository', '# of executors', 'Labels', 'Usage', 'Quiet period', 'SCM checkout retry count', and 'Default view'. Below this is a 'Build Queue' section showing 'No builds in the queue.' and a 'Build Executor Status' section showing '1 Idle' and '2 Idle'. At the bottom are 'Save' and 'Apply' buttons.

Below the configuration section is an 'Artifactory' configuration panel. It includes an 'Add' button, a 'Server ID' field containing 'Any Artifactory server ID', a 'URL' field containing 'URL of the artifactory', a 'Username' field containing 'Username of artifactory', a 'Password' field containing 'Password', and a 'Test Connection' button. There are also checkboxes for 'Enable Push to Bintray (deprecated)', 'Use the Credentials Plugin', and 'Use Different Resolver Credentials', along with an 'Advanced...' link and a 'Delete' button.

| Make sure that the Artifactory plugin is installed

Declarative Pipeline

Pipelines are nothing but a stream of different stages that we want the Jenkins to create. These are provided as Jenkins interpretable configurations directives to create the projects . Jenkins needs to know what commands are to be

executed at a certain stage. Since Jenkins is built using java, its compliers are integrated to interpret ‘Groovy’ — its close ally. Groovy has a lot of embedded features of java along with in-built closures. It is a statically typed, dynamic language which makes coding much more easier. Thus the support for Groovy scripts by default

But many found that learning a whole new language for configuration was not worthwhile and Jenkins came up with the solution — Declarative Jenkins Pipeline

I personally find declarative pipelines more appealing because of its ability to allow the user to write the configuration as directives

Understanding Declarative Pipelines

Initial Declaration

Advantage of using declarative pipelines is the ability to store the Jenkins configuration over in SCM

The first statement starts with defining the directive for **pipeline**. All our directives should be inside the **pipeline** block

```
pipeline{  
    agent any  
}
```

Specifying the **agent** as **any** allows the Jenkins master to select any slave node under its command to execute the *pipeline script*

Now the next step is specifying the **tools** to be used in Jenkins. Tools are nothing but the build tools that are installed in the Jenkins instance. Before using the tools, we should configure them as shown above

```
pipeline {  
    agent any  
  
    environment {  
        GOOGLE_PROJECT_ID = 'test-prj-cmd';  
  
        GOOGLE_SERVICE_ACCOUNT_KEY =
```

```

credentials('service_account_key');
}

tools {

    git 'localGit'
    jdk 'localJava'
    nodejs 'localNode'

}

```

The environment variables to be used in Jenkins can be declared inside the environment section like above. The usage of **credentials** directive is only possible if the Credentials/Plain Credentials plugin have been installed. The Declaration pipeline requires the instances of that particular class available in the environment. Now to use the service account key that was uploaded in the **Credentials** page, we declare an environment variable like above

The tools to be used can be declared specifying the named aliases of the executable paths (see the Global tool section)

Stages in Pipeline

All pipelines involve multiple stages and declarative pipelines allow us to use stage convention.

```

pipeline {
    agent any

    environment {
        GOOGLE_PROJECT_ID = 'test-prj-cmd';
        GOOGLE_SERVICE_ACCOUNT_KEY =
        credentials('service_account_key');
    }

    tools {

        git 'localGit'
        jdk 'localJava'
        nodejs 'localNode'

    }
    stages{
        stage('Init') {
            steps{
                sh '''#!/bin/bash
echo "JAVA_HOME = ${JAVA_HOME}";
echo "PATH = ${PATH}";'''
            }
        }
    }
}

```

```

echo "MAVEN_HOME = ${M2_HOME}";

echo "this is the project id environment"+GOOGLE_PROJECT_ID;
npm install -g @angular/cli@6.0.8;
npm install
''

echo "THis is the credentails:${GOOGLE_SERVICE_ACCOUNT_KEY}";
//readFileStep();
println "Init success..";

}

}

stage('Build') {
steps{

echo "Starting build ...."
sh '''#!/bin/bash
    ng build --aot --prod
'''
println "BUILD NUMBER = $BUILD_NUMBER"
println "Build Success.."
}
post {
    always {

        -----Post build steps -----
    }
}
}
}
}

```

There is a lot of info here. I'll try to explain the necessary directives.

Stages are the provisions to provide every phase that Jenkins has to go through. We have the ability to provide multiple stages through **stage** declarative.

If a script needs to be executed then those can be provided inside the **steps** scope. We are trying to interpret the script as **bash** command instead of the normal **shell** script (it can also be done in **bat** mode for windows) and hence we declare the interpret to be used as

```
#!/bin/bash
```

The **post** declarative provides provision for the user to specify any actions to be taken after each successive phase.

println is a groovy command to print statements and **echo** is the command to print statements in the console inside a script block

Declaring Deploy stage

The above gist gives a detail description of how to deploy an Angular app to the GCP App Engine. Again the presence of ***app.yaml*** is required and the article assumes it to be present implicitly. We require ***gcloud sdk*** to interact and most of its components to interact with GCP

- Use curl to get the ***gcloud sdk***. Specify the required version to be fetched
- Unzip it using the ***tar*** command to */tmp/folder*
- Now install ***gcloud sdk*** in jenkins by from installation script
- Set the ***Path*** variable for cloud specifying where to find the bin location
- Configure the project to be used using the following command

```
gcloud config set project <project_id>
```

- Authenticate into the ***GCP*** using the following command

```
gcloud auth activate-service-account --key-file <the account json file, in this case the credentials provided in environment>
```

- Now use the ***gcloud*** command to deploy

```
gcloud app deploy
```

Note: App.yaml contains the configuration for deploying the app in GCP

There are other parts I have added where in the user can publish the artifactory and also send email notifications of every stage but chose not to explain them since the article's main focus is to deploy to google cloud

Now come back to Jenkins and either trigger a build manually using – ***Build Now*** or by checking in some changes to SCM. You should see all the stages of the pipeline successfully executed





Each stage with the build result

I hope this gives a good idea to get started with Jenkins. Shoot me up with any questions and I will try to answer them as best as possible. Happy reading!!!