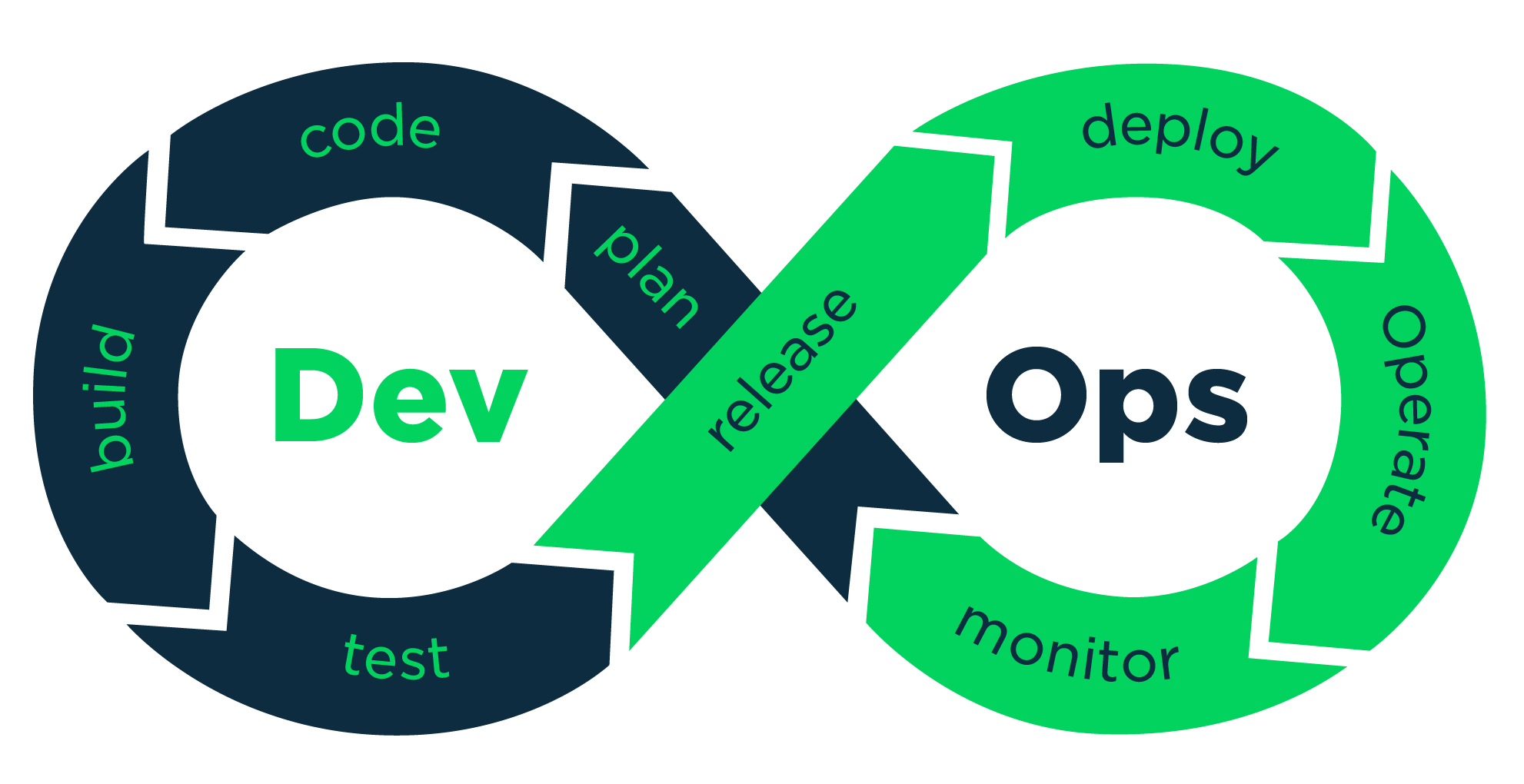
**DEVOPS :**

The term DevOps is a combination of two words namely Development and Operations. DevOps is a practice that allows a single team to manage the entire application development life cycle, that is, development, testing, deployment, and monitoring.

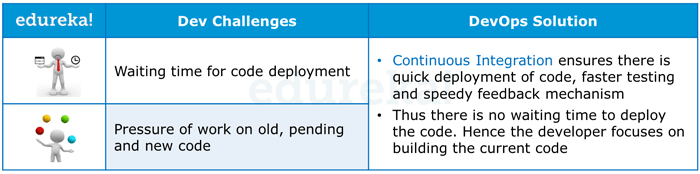
The ultimate goal of DevOps is to decrease the duration of the system’s development life cycle while delivering features, fixes, and updates frequently in close synchronization with business objectives.

**DevOps Life Cycle:**

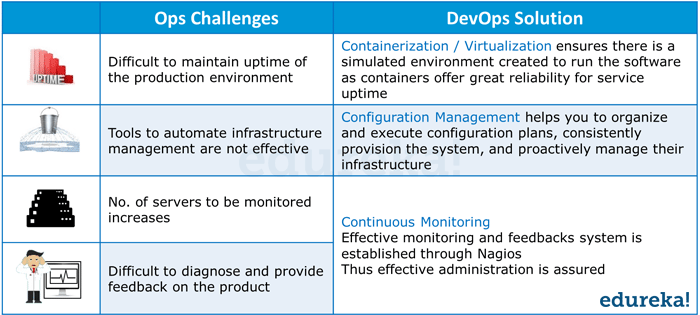


a single group of Engineers (developers, system admins, QA’s. Testers etc turned into DevOps Engineers) has end to end responsibility of the Application (Software) right from gathering the requirement to development, to testing, to infrastructure deployment, to application deployment and finally monitoring & gathering feedback from the end users, then again implementing the changes.

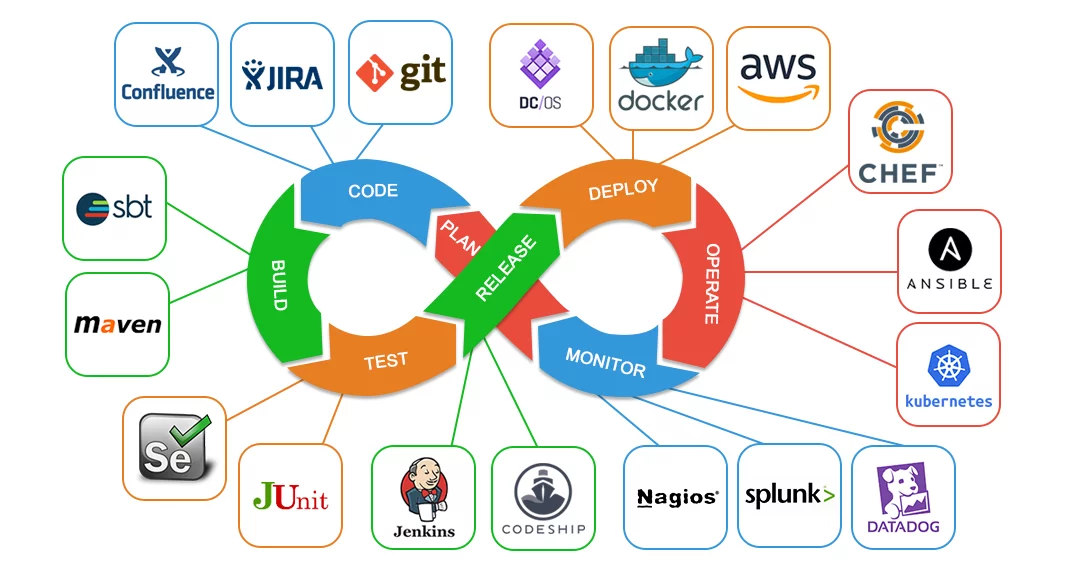
**Below table describes how DevOps addresses Dev Challenges**



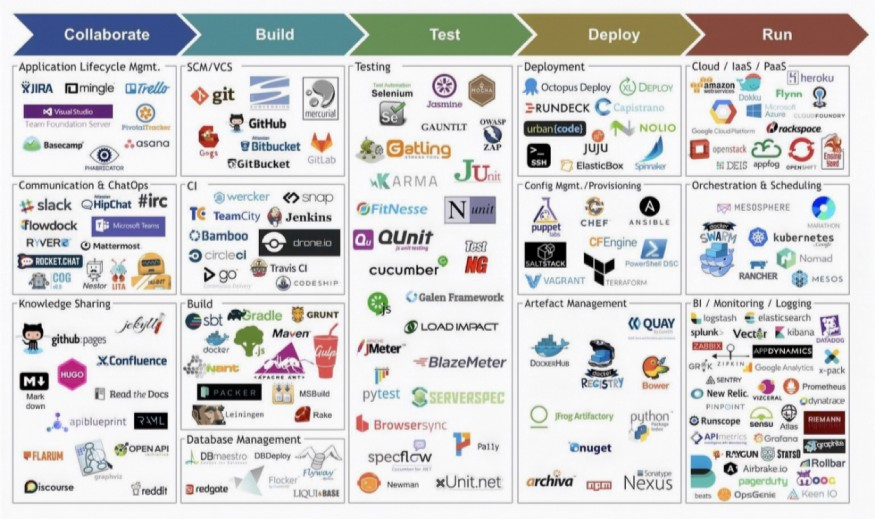
the table below describes how DevOps addresses Ops Challenges.



Now take a look at the below DevOps diagram with various DevOps Tools closely and try to decode it.



You can check the dynamics of DevOps from the picture below



DevOps Lifecycle can be broadly broken down into the below DevOps Stages:

* **Continuous Development**
* **Continuous Integration ( CI )**
* **Continuous Testing**
* **Continuous Monitoring**
* **Virtualization and Containerization**

These stages are the building blocks to achieve DevOps as a whole.

#### **Agile Process:**

* In Agile, a company releases the application with some high priority features in the first iteration.
* After its release, the end-users or the customers give you feedback about the performance of the application.
* Then you make the necessary changes into the application along with some new features and the application is again released which is the second iteration.
* You repeat this entire procedure until you achieve the desired software quality.

Now let us have a look at each of the stages of DevOps life cycle one by one.

**Stage – 1: Continuous Development**

Tools Used: **Git,** SVN, TFS

**Stage – 2: Continuous Integration**

Tools: **Jenkins**, TeamCity, Travis ,Bamboo

**Stage – 3: Continuous Testing**

Tools: Jenkins, Selenium TestNG, JUnit

**Stage – 4: Continuous Deployment**

Tools Used:

Configuration Management – Chef, Puppet, **Ansible**

Containerization – **Docker,** Vagrant

Container Orchestration - **Kubernetes,** Docker swarm

**Stage – 5: Continuous Monitoring**

Tools Used: Splunk, ELK Stack, Nagios, New Relic,**StackDriver**

Now will discuss about Stage1 i.e Continuous Development

Git comes under Continuous Development

Version Control System Tool : VCS

GIT:

Git is an open-source, distributed version control system (VCS), which has a remote repository on the server-side and a local repository on the client-side. This means that the file or code is not present in a central server, but there is a copy of the file stored on the client’s computer.

A distributed version control system enables multiple developers to work in parallel with each other without any code conflicts. Git enables developers to revert and go back to an older version of the code whenever necessary.

Git helps both developers and non-tech professionals by keeping track of their project files. It makes it easier for multiple individuals to work together, and it plays an extremely significant role in big projects that involve large teams.

**GitHub**.com is a **cloud** hosting service that can handle a range of account types: free (public repos only) and paid ($7 per month) developer accounts, teams ($9 per user per month), and businesses ($21 per user per month)

**Git VS SVN:**

| **Git (Distributed VCS)** | **SVN (Centralized VCS)** |
| --- | --- |
| **It's a distributed version control system.** | **It's a Centralized version control system** |
|  |  |
| **Git has a cloned repository.** | **SVN does not have a cloned repository.** |

**Git Bash installation:**

[**https://git-scm.com/downloads**](https://git-scm.com/downloads)

**Github Account creation:**

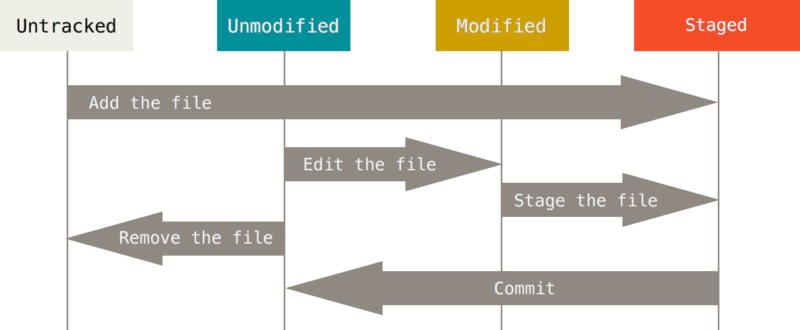
[**https://github.com/signup?return\_to=%2Fsignup%3Fsource%3Dlogin&source=login**](https://github.com/signup?return_to=%2Fsignup%3Fsource%3Dlogin&source=login)

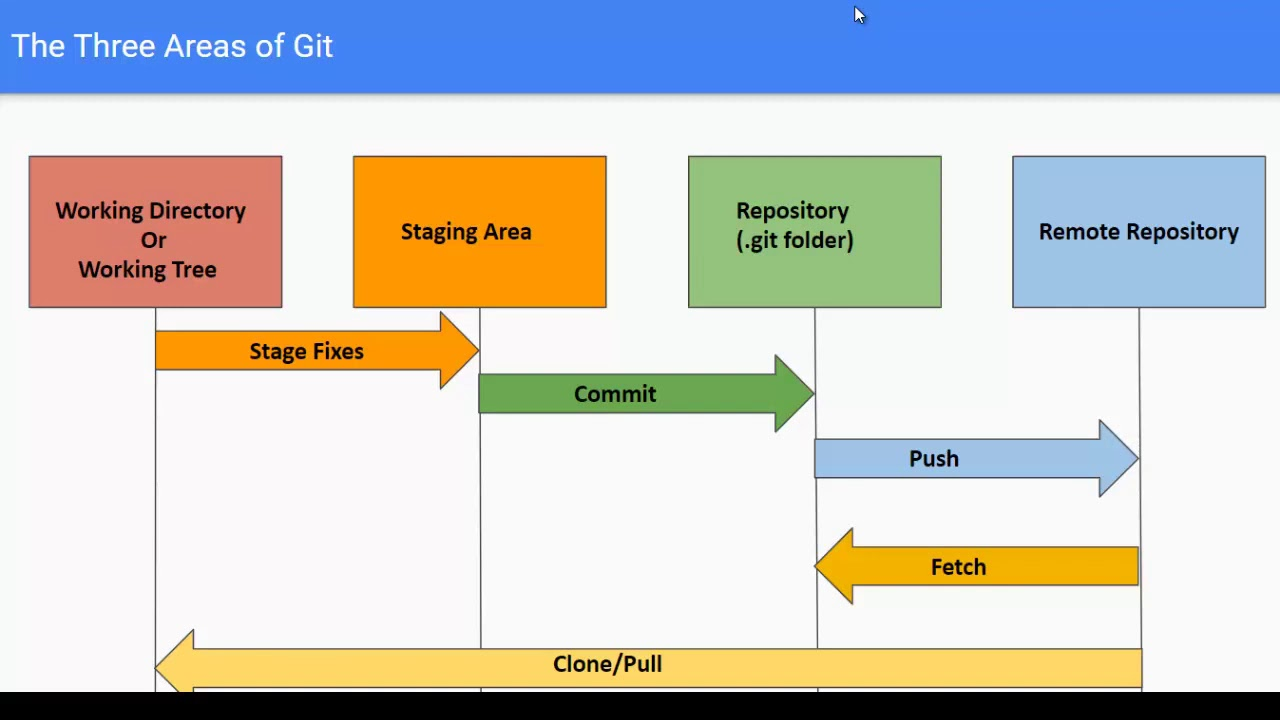
**github account creation**

**What is the difference between git and git bash?**

**Git Bash emulates a bash environment on windows. It lets you use all git features in command line plus most of standard unix commands. Useful if you are used to Linux and want to keep the same habits. Git GUI is a Graphical User Interface letting you use Git without touching command line**

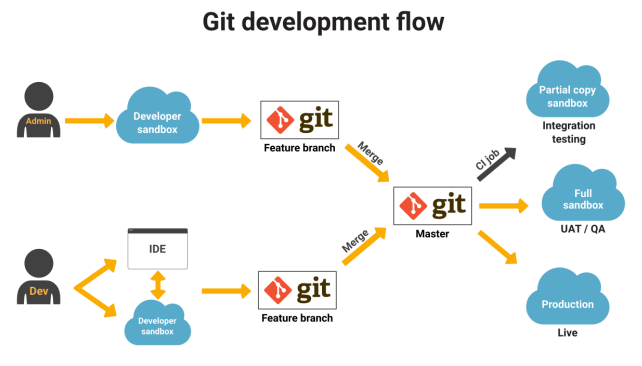
**GIT Workflow:**

****

****

**Git Flow:**

**Developers create feature branches from master (or another integration branch) to work on new features. These branches keep track of all of a feature’s changes and, when complete, they’re merged back into master. From master, the feature can be deployed out to any number of environments, following a clearly-defined and largely automated release process. A typical Git workflow looks like this:**

****

**GIT commands:-**

**Git init (or) git init - -shared ---->It will create shared repository---it has .git directory--- By default it creates Master branch.**

**Git init - - bare : it creates bare repo—it doesn’t have .git directory—it has sub directories as usual ---It is a remote repository**

**File states----untracked, tracked, modified, staged**

**Git ignores empty directory by default. If you want to add a file that is already in .gitignore, then we can add file forcefully.**

**Git config file -- /etc/gitconfig or .git/config**

**Git –version –to find the git version**

**Which git --- to find location of git**

**HEAD ---latest commit in repo**

**Files are stored in a git repo with hash alg names,files stored in repo in compressed format.**

**Git add filename - ----add file to staging**

**Git commit -m “message” ---add to stage**

**Git commit –amend “message” file ---add & commit at a time**

**Git commit - - amend -m “new message”---amending most recent commit message**

**OR**

**Git commit –amend –no-edit -àchnages/modify latest commit mistakes—creates new commit ID,doesn’t change its snapshot.**

**GIT CONFIGURATION:**

**Git config –global user.name “kiran”**

**Git config –global user.email logintokiran@gmail.com**

**Git config –local user.name “kiran”**

**Git config –list ----**list config settings

**Git config –global –list ---** list only global settings

**GIT STATUS:**

**The git status command is used to display the state of the repository and staging area. It allows us to see the tracked, untracked files and changes**

**Git status file --** --gives status about files(untracked/new files)

**Git status -s file**  ---gives short output

**GIT LOG :**

**The git log command displays a record of the commits in a Git repository. By default, the git log command displays a commit hash, the commit message, and other commit metadata.**

**Git log file ---> gives log details**

**Git log –oneline ---> to see more compact version in one line.**

**Git log –author=kiran ---> show logs done by kiran**

**Git log –grep= “kiran” --> show logs done by kiran/kiran\*\*\***

**Git log –before**

**Git log –since**

**Git log –after**

**Git log –since**

**Git log – file ---see commit history of deleted file.**

**GIT DIFF:**

**Diff command is used in git to track the difference between the changes made on a file.**

**Git diff file ---> show diff between working dir and Repo**

**Git diff –stat > summary of changes instead of full diff.**

**Git diff –cached/staged –> show diff of index&latest commit.**

**Git diff HEAD àshow difference of working dir & latest commit.**

**Git diff master origin/master –compare remote branches with local branch.**

**Git Ignore:**

**Git can specify which files or parts of your project should be ignored by Git using a .gitignore file. Git will not track files and folders specified in .gitigonre**

**HOW TO IGNORE FILES/DIR in GIT:**

**Vi .gitignore**

**\*.txt**

**ignore files with started with txt and then u can try to add .gitignore file**

**Git add .gitignore**

**Git commit -m “message”**

**Git push origin master**

**Then you can start to add .txt file and Git won’t track \*.txt files.**

**If you want to .txt files , you should add by adding -f to the git add command as below**

**Git add .txt -f**

**Git Show:**

**git-show is a command line utility that is used to view expanded details on Git objects such as blobs, trees, tags, and commits**

**Git show ID ---> show details about commit ID.**

**Git show HEAD pom.xml ---> address pom.xml in HEAD.**

**GIT CHERRY-PICK**

**Git cherry-pick --> choose commit from one branch & apply it onto another branch.**

**Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes**

**GIT STASH:**

**git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them when ever you wanted.**

**Git stash ---> record current state of working tree &index.**

**Git stash list -- > list of availbale stashes.**

**Git stash clear –> clear or delete all available stashes.**

**Git stash apply stash@{0} ---> apply stash@{0}**

**Git stash drop stash@{0} -- > remove stashed change without apply it.**

**Git stash pop ---> apply latest stash &delete it after from stash list.**

**Git stash drop ----> discard most recently stashed change**

**GIT BRANCHES:**

**Branching is used in Version control and software management to maintain stability while isolated changes are made to code. Branching facilitates the development of bug fixes, the addition of new capabilities and the integration of new versions after they have been tested in isolation.**

**Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone.**

**Branches are part of the every day development process.**

**Local branches are changed by user. updated by merge command.[.git/refs/heads]**

**Remote branches are not changed by user.updated by fetch command.[.git/refs/remotes]**

**In git trunk becomes master, master is always ready to release.**

**Default branch is master**

**Origin/master ---remote repo.**

**Branch Related Commands:**

**Git branch ---> list all available branches**

**Git branch - a - - -> list all branches local+ remote.**

**Git branches -r –>list all remote branches.**

**Git branch -v ---> see last commit on each branch**

**Git branch branchname –> create new branch**

**Git branch -d name – > delete local branch**

**Git branch -D name ----> delete all branches.**

**Git branch branchname commitID –> create particular branch for that commit.**

**Git branch –merged –> to see which branches are merged/not**

**Git branch - - no-merged –>to see which branches are not merged**

**Git remote show origin --->to see branches on remote origin.**

**Git branch –remotes ->To see all remote tracking branches**

**Git checkout branch— >switch to particular branch.**

**Git checkout -b newbranch ID ---> create new branch from commit ID.**

**Git branch -m oldbranch newbranch ---> rename branch.**

**Git Branch creation Reatime:**

**Kiran@LAPTOP-LT369VEJ MINGW64 ~/Github/docker-k8s (master)**

**$ git checkout -b designing**

Create and Switched to a new branch 'designing' from master branch

Master is the default branch

**kiran@LAPTOP-LT369VEJ MINGW64 ~/Github/docker-k8s (designing)**

**$ ls**

**CONTRIBUTING.md Jenkinsfile LICENSE README.md docs/ jenkins/ sample-app/ tests/**

**kiran@LAPTOP-LT369VEJ MINGW64 ~/Github/docker-k8s (designing)**

**$ vi Jenkinsfile**

**kiran@LAPTOP-LT369VEJ MINGW64 ~/Github/docker-k8s (designing)**

**$ git add \***

**$ git commit -m "this is for new design DEVOPS-1234"**

**[designing 28e5648] this is for new design DEVOPS-1234**

**1 file changed, 1 insertion(+), 1 deletion(-)**

**$ git push origin designing**

**Enumerating objects: 5, done.**

**Counting objects: 100% (5/5), done.**

**Delta compression using up to 4 threads**

**Compressing objects: 100% (3/3), done.**

**Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.**

**Total 3 (delta 2), reused 0 (delta 0), pack-reused 0**

**remote: Resolving deltas: 100% (2/2), completed with 2 local objects.**

**remote:**

**remote: Create a pull request for 'designing' on GitHub by visiting:**

**remote: https://github.com/kirankumar77/docker-k8s/pull/new/designing**

**remote:**

**To https://github.com/kirankumar77/docker-k8s.git**

**\* [new branch] designing -> designing**

**kiran@LAPTOP-ETEV1HL9 MINGW64 ~/Github/payment (audio)**

**$ vi test.sh**

**kiran@LAPTOP-ETEV1HL9 MINGW64 ~/Github/payment (audio)**

**$ git status**

**On branch audio**

**Untracked files:**

**(use "git add <file>..." to include in what will be committed)**

**test.sh**

**nothing added to commit but untracked files present (use "git add" to track)**

**kiran@LAPTOP-ETEV1HL9 MINGW64 ~/Github/payment (audio)**

**$ git add test.sh**

**warning: LF will be replaced by CRLF in test.sh.**

**The file will have its original line endings in your working directory**

**kiran@LAPTOP-ETEV1HL9 MINGW64 ~/Github/payment (audio)**

**$ git status**

**On branch audio**

**Changes to be committed:**

**(use "git restore --staged <file>..." to unstage)**

**new file: test.sh**

**SQUASHING:**

Git squash is a technique that helps you to take a series of commits and condense it to a few commits. For example, assume that you have a series of n commits. By squashing you can make all the n-commits to a single commit

**CHERRY-PICK :**

**git cherry-pick is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD. Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes.**

**Example:**

**Git checkout master**

**Git cherry-pick commitID --c> hoose specific commit from master &apply It onto another branch.**

**GIT REMOTE:**

**When you clone a repository with git clone , it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits.**

**Git remote -- > list of remote repos**

**Git remote -v –> list of remote repos with URLs**

**Git remote show remotename – > show info about remote.**

**Git remote remove name –> remove remote**

**Git remote show name ---> show info about repo**

**Git remote rename kiran Kumar –> rename kiran with Kumar**

**Git remote add localrepo –> add local repo to remote**

**GIT PUSH:**

**The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch , but whereas fetching imports commits to local branches, pushing exports commits to remote branches**

**Git push ---> push to remote repo**

**Git push origin branchname –> push branch to origin.**

**Git push -f remote kiran –> push kiru forcefully to remote repo**

**Git push –all ----> push all branches to remote.**

**GIT PULL:**

git pull is a Git command used to update the local version of a repository from a remote

**Git pull – > fetches from origin by default &merges into current branch.**

**Git pull origin master ---> download changes directly merge/integrate into HEAD.**

**GIT FETCH:**

The git fetch command downloads commits, files, and refs from a remote repository into your local repo

**Git fetch origin ----> update remote tracking branches from Rmote repo.**

**Git fetch origin –tags –> download tags form remote repo.**

**GIT MERGE:**

The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

**Git checkout master**

**Git merge kiran ---> merge kiran branch into master branch**

**Git merge kiran Kumar –merge kiran branch to Kumar branch**

**GIT MERGE PRACTICE:**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (watsap)**

**$ git add abc.java**

**warning: LF will be replaced by CRLF in abc.java.**

**The file will have its original line endings in your working directory**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (watsap)**

**$ git commit -m "addedfile"**

**[watsap d9fada2] addedfile**

**1 file changed, 1 insertion(+)**

**create mode 100644 abc.java**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (watsap)**

**$ git push origin watsap**

**Enumerating objects: 4, done.**

**Counting objects: 100% (4/4), done.**

**Delta compression using up to 8 threads**

**Compressing objects: 100% (2/2), done.**

**Writing objects: 100% (3/3), 341 bytes | 113.00 KiB/s, done.**

**Total 3 (delta 0), reused 0 (delta 0), pack-reused 0**

**remote:**

**remote: Create a pull request for 'watsap' on GitHub by visiting:**

**remote: https://github.com/kirankumar77/gitops/pull/new/watsap**

**remote:**

**To https://github.com/kirankumar77/gitops.git**

**\* [new branch] watsap -> watsap**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (watsap)**

**$ git checkout master**

**Switched to branch 'master'**

**Your branch is up to date with 'origin/master'.**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (master)**

**$ git merge watsap**

**Updating e0491a5..d9fada2**

**Fast-forward**

**abc.java | 1 +**

**1 file changed, 1 insertion(+)**

**create mode 100644 abc.java**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (master)**

**$ ls**

**abc.java gitops-tutorial-master/ test.txt**

**KIRAN KUMAR REDDY@LAPTOP-J01UHMRN MINGW64 ~/github/gitops (master)**

**$ git push origin master**

**Total 0 (delta 0), reused 0 (delta 0), pack-reused 0**

**To https://github.com/kirankumar77/gitops.git**

**e0491a5..d9fada2 master -> master**

**After this , you can see same changes in Github Repo and this procedure is alternative mechanism for Git Pull request and merge.**

**GIT REMOVE:**

The easiest way to delete a file in your Git repository is to execute the “git rm” command and to specify the file to be deleted. Note that by using the “git rm” command, the file will also be deleted from the filesystem

**Git rm file ----> remove file from working dir &index**

**Git commit -m “deleted file” ---**

**Git rm -f filename — > remove file in working dir**

**Git rm –cached file –> remove files in index&working dir**

**[still these files are untracked files]**

**GIT CLONE:**

**The "clone" command downloads an existing Git repository to your local computer.**

**Git clone path**

**Git clone https://github.com/project.git**

**GIT REVERT/RESET:**

The git revert command is **used for undoing changes to a repository's commit history**.

A revert is safer than a reset because it will not remove any commits from a shared history

**Git Interview Questions:**

**What is Git in simple words?**

**Git is a type of version control system (VCS) that makes it easier to track changes to files. For example, when you edit a file, git can help you determine exactly what changed, who changed it, and why. ... Git isn't the only version control system out there, but it's by far the most popular.**

**What exactly is GitHub?**

**GitHub is a Git repository hosting service, but it adds many of its own features. While Git is a command line tool, GitHub provides a Web-based graphical interface. It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.**

**How to add a new SSH key to your GitHub account:**

**Generate ssh keys with ssh-keygen -t rsa command and copy public key**

1. **Copy the SSH public key to your clipboard. ...**
2. **In the upper-right corner of any page, click your profile photo, then click Settings.**
3. **In the user settings sidebar, click SSH and GPG keys.**
4. **Click New SSH key or Add SSH key.**
5. **In the "Title" field, add a descriptive label for the new key. ...**
6. **Paste your key into the "Key" field.**

**How to Clone a repository using the command line:**

1. **From the repository, click + in the global sidebar and select Clone this repository under Get to work.**
2. **Copy the clone command (either the SSH format or the HTTPS). ...**
3. **From a terminal window, change to the local directory where you want to clone your repository.**

**What is Pull request in Github: (PR)**

**A pull request is a method of submitting contributions to an open development project. ... A pull request occurs when a developer asks for changes committed to an remote repository to be considered for inclusion in a project's main repository.**

**How do I accept a pull request?**

**To accept the pull request, click the Pull Requests tab to see a summary of pending pull requests. If you are happy with the changes, click Merge Pull request to accept the pull request and perform the merge.**

**You can add in a comment if you want. Once you click Merge Pull request, you will see a button Confirm merge.**

**What is the difference between pull request and merge request?**

**GitLab's "merge request" feature is equivalent to GitHub's "pull request" feature. Both are means of pulling changes from another branch or fork into your branch and merging the changes with your existing code. ... A "merge request" should not be confused with the git merge command**

**MERGE Conflicts: (what can happen if won’t perform git pull command before starting development on GIT ?)**

**A merge conflict is an event that takes place when Git is unable to automatically resolve differences in code between two commits. Git can merge the changes automatically only if the commits are on different lines or branches.**

**Let’s assume there are two developers: Developer A and Developer B. Both of them pull the same code file from the remote repository and try to make various amendments in that file. After making the changes, Developer A pushes the file back to the remote repository from his local repository. Now, when Developer B tries to push that file after making the changes from his end, he is unable to do so, as the file has already been changed in the remote repository.**

**To prevent such conflicts, developers work in separate isolated branches. The Git merge command combines separate branches and resolves any conflicting edits.**

**How do I combine multiple commits into one/ How to Combine Multiple Git Commits into One ?**

1. **Steps to merging multiple commits. Running git rebase in interactive mode. Typing "squash" Choosing between commit messages. Pushing changes.**
2. **Squashing.**

**Can you push multiple commits?**

**For your first question, no, there's nothing wrong with pushing multiple commits at once. Many times, you may want to break your work down into a few small, logical commits, but only push them up once you feel like the whole series is ready.**

**How do I create a pull request for multiple commits?**

**A pull will pull all the commits, including their dependencies - it won't cherry-pick individual commits. So if you want to request that only your commits be pulled, and there are other people's commits in the same branch, you have to first separate your commits into a different branch.**

**Difference between Git Merge and Rebase:**

**Git rebase and merge both integrate changes from one branch into another. ... Git rebase moves a feature branch into a master. Git merge adds a new commit, preserving the history.**

**Or**

**rebase “reapplies commits on top of another base branch”, whereas merge “joins two or more development histories together”.**

**Why is git push rejected?**

**If your push is rejected, what has most likey happened is that someone else pushed some changes to the remote master while you were making your changes, and you need to pull them down to your repo before you can push your changes up. So do a 'git pull –rebase', then push again**

**Is there a difference between Git and GitHub?**

| **GIT** | **SVN** |
| --- | --- |
| **Git is a Distributed VCS.** | **SVN IS a Centralized VCS.** |
| **In git every user has their own copy of code on their local like their own branch.** | **In SVN there is central repository has working copy that also make changes and committed in central repository.** |

**What are merge conflicts in git?**

**merge conflict is an event that occurs when Git is unable to automatically resolve differences in code between two commits. When all the changes in the code occur on different lines or in different files, Git will successfully merge commits without your help.**

**How do I fix merge conflicts in GitHub /Resolving a merge conflict on GitHub?**

1. **Under your repository name, click Pull requests.**
2. **In the "Pull Requests" list, click the pull request with a merge conflict that you'd like to resolve.**
3. **Near the bottom of your pull request, click Resolve conflicts.**

**Difference between git pull and fetch?**

**The git fetch command downloads commits, files, and refs from a remote repository into your local repo. ... git pull is the more aggressive alternative; it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content.**

**Explain about your Branching Strategy:**

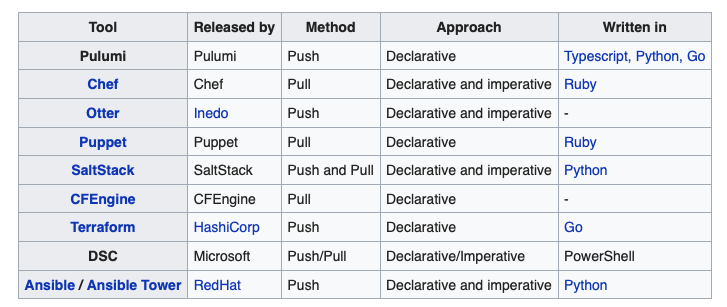
* **Code in Master branch is deployable at all times.**
* **When you want to start working on a new task or new functionality, create a new branch and give it a descriptive name.**
* **Commit to that branch locally and regularly send your work to the same-named branch on the server.**
* **Open a pull request (PR ) when you feel your changes are ready to be merged (or even if you aren't so sure, but would like some feedback).**
* **After the new feature is revised and approved, you can merge it into Master Branch.**
* **Once your changes are merged and pushed to the Master Branch, you can and *should* deploy immediately(Jenkins will trigger the build )**

**The GitHub flow is also known for encouraging** [**continuous delivery (CD)**](https://rollout.io/blog/continuous-integration-continuous-delivery-continuous-deployment/)**. As soon as your changes are merged, you should deploy to production.**

**TERRAFORM:**

**Infrastructure as a code(IaC) is a process of managing and provisioning mechanisms for authenticating, planning and implementing servers and data centers in the cloud and private network through machine-readable and understandable configuration files rather than physically configuring the hardware.**

**Various tools for IaC**

****

**Terraform is an open-source infrastructure as a code software tool created by HashiCorp. It enables users to define and provision a data center infrastructure using a high-level configuration language known as Hashicorp Configuration Language (HCL), or optionally JSON.Terraform supports a number of cloud infrastructure providers such as Amazon Web Services, IBM Cloud (formerly Bluemix), Google Cloud Platform, Linode, Microsoft Azure, Oracle Cloud Infrastructure, or VMware vSphere as well as OpenStack.**

****

**Terraform Installation:**

**wget** [**https://releases.hashicorp.com/terraform/0.12.24/terraform\_0.12.24\_linux\_amd64.zip**](https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip)

**unzip terraform\_0.12.24\_linux\_amd64.zip**

**mv terraform /usr/local/bin/terraform**

**terraform version**

#### Let's Get Our Application Up and Running!

**Terraform validate - > validate the terraform file syntax**

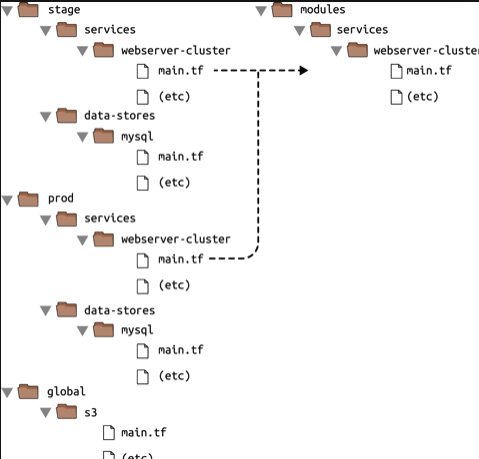
**terraform init** -- > This command sets up the environment. (mean download the required plugins )

**terraform plan** → This command reports which configuration will be applied.(before

**terraform apply -auto-approve** → This command approves the changes automatically and applies the configuration defined on Terraform files.

**terraform destroy -auto-approve** → Counteracting the command above, this removes everything created.

**The folder structure could be as shown below:**

****

**A *Terraform configuration* consists of a *root module*, where evaluation begins, along with a tree of child modules created when one module calls another.**

**The main purpose of the Terraform language is declaring** [**resources**](https://www.terraform.io/docs/configuration/resources.html)**.**

**A group of resources can be gathered into a** [**module**](https://www.terraform.io/docs/configuration/modules.html)**, which creates a larger unit of configuration. A resource describes a single infrastructure object, while a module might describe a set of objects.**

**The Terraform language uses configuration files that are named with the .tf file extension. There is also** [**a JSON-based variant of the language**](https://www.terraform.io/docs/configuration/syntax-json.html) **that is named with the .tf.json file extension.**

**A *module* is a collection of .tf or .tf.json files kept together in a directory.**

# Comments

**The Terraform language supports three different syntaxes for comments:**

* [**#**](https://www.terraform.io/docs/configuration/index.html#) **begins a single-line comment, ending at the end of the line.**
* [**//**](https://www.terraform.io/docs/configuration/index.html#-1) **also begins a single-line comment, as an alternative to #.**
* [**/\***](https://www.terraform.io/docs/configuration/index.html#-2) **and \*/ are start and end delimiters for a comment that might span over multiple lines.**

**Provider:**

**The provider block is used to configure the named provider, in our case "gcp”**

**The resource block defines a resource that exists within the infrastructure. A resource might be a physical component such as an gce vm instance, or it can be a logical resource such as a Heroku application.**

# Initialization

**The first command to run for a new configuration — or after checking out an existing configuration from version control — is terraform init, which initializes various local settings and data that will be used by subsequent commands.**

# APPLY

**If the plan was created successfully, Terraform will now pause and wait for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure. In this case the plan looks acceptable, so type yes at the confirmation prompt to proceed.**

# **Destroy:**

**We’ve now seen how to build and change infrastructure. Before we move on to creating multiple resources and showing resource dependencies, we’re going to go over how to completely destroy the Terraform-managed infrastructure.**

***Modules:***

***Modules* in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components, improve organization, and to treat pieces of infrastructure as a black box**

**Practice:**

**root@docker-ubuntu:~/terraform# ls**

**main.tf**

**root@docker-ubuntu:~/terraform# terraform plan**

**Plugin reinitialization required. Please run "terraform init".**

**Reason: Could not satisfy plugin requirements.**

**Plugins are external binaries that Terraform uses to access and manipulate**

**resources. The configuration provided requires plugins which can't be located,**

**don't satisfy the version constraints, or are otherwise incompatible.**

**1 error(s) occurred:**

**\* provider.docker: no suitable version installed**

**version requirements: "(any version)"**

**versions installed: none**

**Terraform automatically discovers provider requirements from your**

**configuration, including providers used in child modules. To see the**

**requirements and constraints from each module, run "terraform providers".**

**Error: error satisfying plugin requirements**

**root@docker-ubuntu:~/terraform# terraform init**

**Initializing provider plugins...**

**- Checking for available provider plugins on https://releases.hashicorp.com...**

**- Downloading plugin for provider "docker" (2.7.2)...**

**The following providers do not have any version constraints in configuration,**

**so the latest version was installed.**

**To prevent automatic upgrades to new major versions that may contain breaking**

**changes, it is recommended to add version = "..." constraints to the**

**corresponding provider blocks in configuration, with the constraint strings**

**suggested below.**

**\* provider.docker: version = "~> 2.7"**

**Terraform has been successfully initialized!**

**You may now begin working with Terraform. Try running "terraform plan" to see**

**any changes that are required for your infrastructure. All Terraform commands**

**should now work.**

**If you ever set or change modules or backend configuration for Terraform,**

**rerun this command to reinitialize your working directory. If you forget, other**

**commands will detect it and remind you to do so if necessary.**

**root@docker-ubuntu:~/terraform# ls -a**

**. .. .terraform main.tf**

**root@docker-ubuntu:~/terraform# cd .terraform/**

**root@docker-ubuntu:~/terraform/.terraform# ls**

**plugins**

**root@docker-ubuntu:~/terraform/.terraform# cd plugins/**

**root@docker-ubuntu:~/terraform/.terraform/plugins# ls**

**linux\_amd64**

**root@docker-ubuntu:~/terraform/.terraform/plugins# cd linux\_amd64/**

**root@docker-ubuntu:~/terraform/.terraform/plugins/linux\_amd64# ls**

**lock.json terraform-provider-docker\_v2.7.2\_x4**

**root@docker-ubuntu:~/terraform/.terraform/plugins/linux\_amd64# ls -lart**

**total 24108**

**drwxr-xr-x 3 root root 4096 Aug 2 03:03 ..**

**-rwxr-xr-x 1 root root 24674304 Aug 2 03:03 terraform-provider-docker\_v2.7.2\_x4**

**-rwxr-xr-x 1 root root 82 Aug 2 03:03 lock.json**

**drwxr-xr-x 2 root root 4096 Aug 2 03:03 .**

**Refreshing Terraform state in-memory prior to plan...**

**The refreshed state will be used to calculate this plan, but will not be**

**persisted to local or remote state storage.**

**-----------------------------------------------------------------------**

**An execution plan has been generated and is shown below.**

**Resource actions are indicated with the following symbols:**

**+ create**

**Terraform will perform the following actions:**

**+ docker\_image.image\_id**

**id: <computed>**

**latest: <computed>**

**name: "ghost:latest"**

**Plan: 1 to add, 0 to change, 0 to destroy.**

**------------------------------------------------------------------------**

**Note: You didn't specify an "-out" parameter to save this plan, so Terraform**

**can't guarantee that exactly these actions will be performed if**

**"terraform apply" is subsequently run.**

**As of now we cant see ghost image in our VM and below are the details**

**root@docker-ubuntu:~/terraform# docker images**

**REPOSITORY TAG IMAGE ID CREATED SIZE**

**kirankumar77/busybox-test v1 8fef1ef1e3d5 3 days ago 1.24MB**

**gcr.io/devops-test-316311/kirankumar77/busybox-test v1 8fef1ef1e3d5 3 days ago 1.24MB**

**kirankumar77/jboss-counterwebapp latest d892d81ad69b 5 weeks ago 730MB**

**kirankumar77/busybox latest 69593048aa3a 7 weeks ago 1.24MB**

**kirankumar77/centos latest 300e315adb2f 7 months ago 209MB**

**centos latest 300e315adb2f 7 months ago 209MB**

**jboss/base-jdk 11 a240ec4882cb 10 months ago 471MB**

**Now we can apply terraform script to pull the ghost image and use below command for the same.**

**root@docker-ubuntu:~/terraform# terraform apply**

**An execution plan has been generated and is shown below.**

**Resource actions are indicated with the following symbols:**

**+ create**

**Terraform will perform the following actions:**

**+ docker\_image.image\_id**

**id: <computed>**

**latest: <computed>**

**name: "ghost:latest"**

**Plan: 1 to add, 0 to change, 0 to destroy.**

**Do you want to perform these actions?**

**Terraform will perform the actions described above.**

**Only 'yes' will be accepted to approve.**

**Enter a value: yes**

**docker\_image.image\_id: Creating...**

**latest: "" => "<computed>"**

**name: "" => "ghost:latest"**

**docker\_image.image\_id: Still creating... (10s elapsed)**

**docker\_image.image\_id: Creation complete after 17s (ID: sha256:3d84021f2c1e800f7a62c16ad0937f77...10b37c5fc939efd4925bfecc41ghost:latest)**

**Apply complete! Resources: 1 added, 0 changed, 0 destroyed.**

**After applying the terraform script , we can ghost docker images in our VM.**

**If you run the same terraform apply command one more time, it won't apply anything new because ghost image pulled already.**

**root@docker-ubuntu:~/terraform# terraform apply**

**docker\_image.image\_id: Refreshing state... (ID: sha256:3d84021f2c1e800f7a62c16ad0937f77...10b37c5fc939efd4925bfecc41ghost:latest)**

**Apply complete! Resources: 0 added, 0 changed, 0 destroyed.**

**Terraform state file:**

**Terraform stores information about your infrastructure in a state file. This state file keeps track of resources created by your configuration and maps them to real-world resources.**

**Terraform state backup file:**

**By default, a backup of your state file is written to terraform. tfstate. backup in case the state file is lost or corrupted to simplify recovery. The state file is used by Terraform to keep track of resources and metadata information about your infrastructure.**

**root@docker-ubuntu:~/terraform# cat terraform.tfstate**

**{**

**"version": 3,**

**"terraform\_version": "0.11.13",**

**"serial": 1,**

**"lineage": "90f4ba5c-02f3-5fb6-6a37-f8c45647c0f3",**

**"modules": [**

**{**

**"path": [**

**"root"**

**],**

**"outputs": {},**

**"resources": {**

**"docker\_image.image\_id": {**

**"type": "docker\_image",**

**"depends\_on": [],**

**"primary": {**

**"id": "sha256:3d84021f2c1e800f7a62c16ad0937f77b693ac10b37c5fc939efd4925bfecc41ghost:latest",**

**"attributes": {**

**"id": "sha256:3d84021f2c1e800f7a62c16ad0937f77b693ac10b37c5fc939efd4925bfecc41ghost:latest",**

**"latest": "sha256:3d84021f2c1e800f7a62c16ad0937f77b693ac10b37c5fc939efd4925bfecc41",**

**"name": "ghost:latest"**

**},**

**"meta": {},**

**"tainted": false**

**},**

**"deposed": [],**

**"provider":** "provider.docker"

**}**

**},**

**"depends\_on": []**

**}**

**]**

**}**

**Now if you want to destroy the same Infra ( pull image ) which was created by Terraform , we can use the below command.**

**root@docker-ubuntu:~/terraform# terraform destroy**

**docker\_image.image\_id: Refreshing state... (ID: sha256:3d84021f2c1e800f7a62c16ad0937f77...10b37c5fc939efd4925bfecc41ghost:latest)**

**An execution plan has been generated and is shown below.**

**Resource actions are indicated with the following symbols:**

**- destroy**

**Terraform will perform the following actions:**

**- docker\_image.image\_id**

**Plan: 0 to add, 0 to change, 1 to destroy.**

**Do you really want to destroy all resources?**

**Terraform will destroy all your managed infrastructure, as shown above.**

**There is no undo. Only 'yes' will be accepted to confirm.**

**Enter a value: yes**

**docker\_image.image\_id: Destroying... (ID: sha256:3d84021f2c1e800f7a62c16ad0937f77...10b37c5fc939efd4925bfecc41ghost:latest)**

**docker\_image.image\_id: Destruction complete after 2s**

**Destroy complete! Resources: 1 destroyed.**

**Now we can’t see ghost images in our VM because we performed the destroy command.**

**GCP VM Instance creation:**

[**https://github.com/kirankumar77/terraform-gcp**](https://github.com/kirankumar77/terraform-gcp)

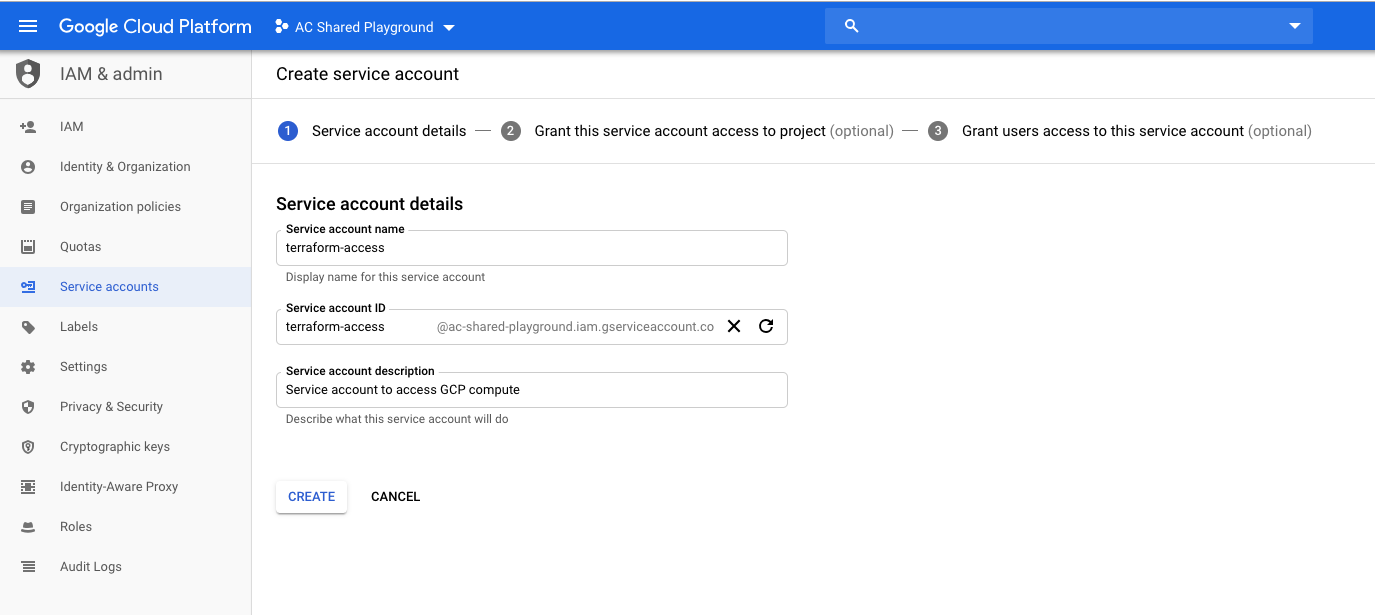
##### Configuring Our Service Account on Google Cloud Platform

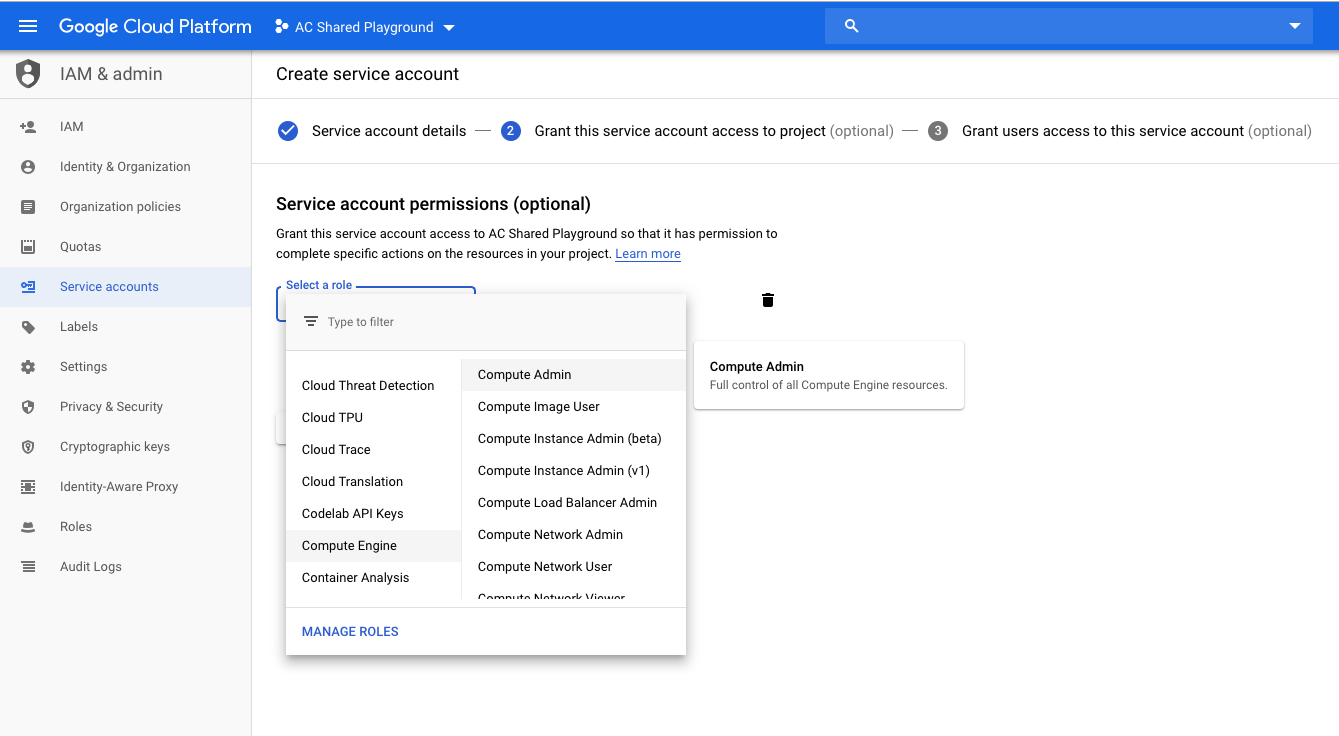
**A service account is a way to give granular access to a vendor or someone else. It's really useful when you want to give specific resources to a group or user.**

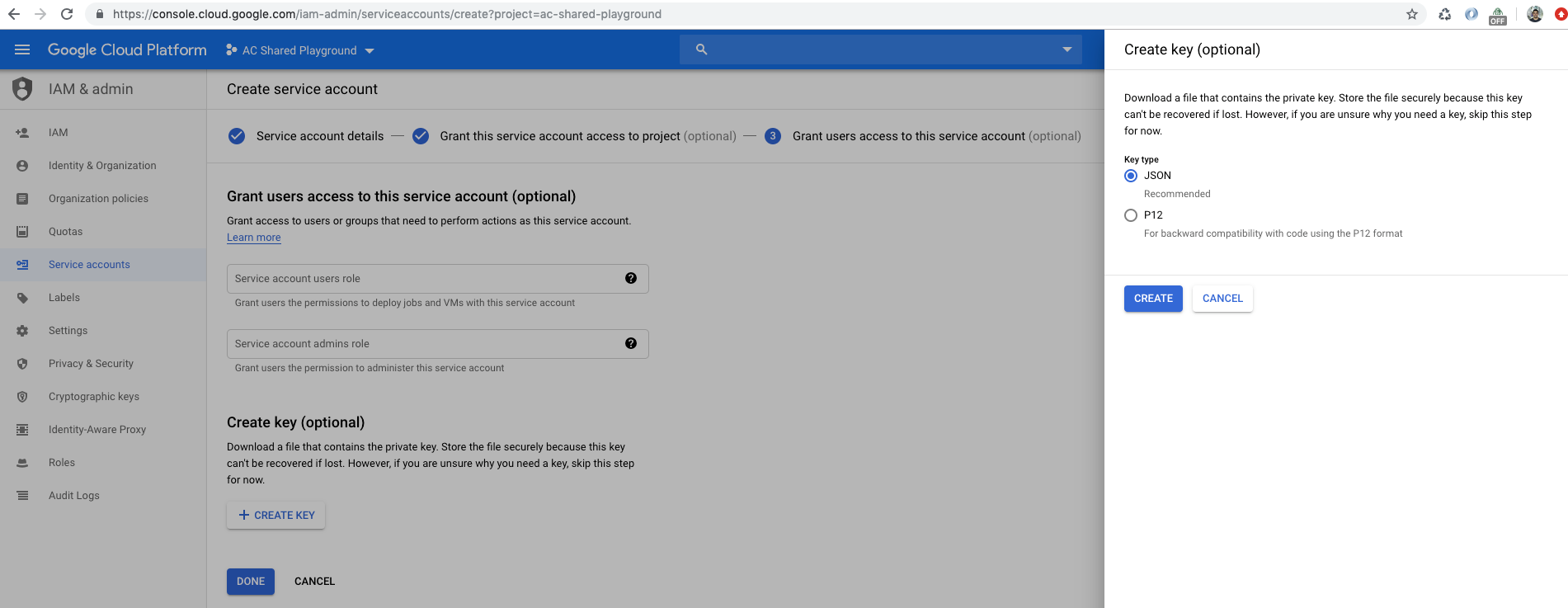
**After you create your account on Google Cloud, you should create a service account that will access Google Compute Engine (GCE). This is needed to create and handle a virtual machine.**

* **Create a service account and specify the compute admin role. Then, download the generated JSON file, rename it credentials.json, and save it to your project’s directory (cloned above).**

**The JSON file you just downloaded should be protected from non-authorized users. This is a private key or password to manage your infrastructure’s resources. For development purposes, we can add a .gitignore file to our project, adding credentials.json so that it’s not versioned to our repository. These steps are shown below:**

****

****

****

#### **Creating a Virtual Machine with Terraform:**

**Now we’re almost able to create a virtual machine instance!**

**Perhaps you noticed that the project cloned above contains files that end in something.tf. These files belongs to Terraform. This name format allows Terraform to know which files to work with when initializing, planning, applying, and destroying.**

**Provider.tf:**

**This file contains the configurations needed for provisioning a resource on GCP. Notice that credentials.json is not versioned in our project; it was built during previous steps.**

**# Specify the provider (GCP, AWS, Azure)**

**provider "google" {**

**credentials = "${file("credentials.json")}"**

**project = "devops-test-316311d"**

**region = "us-central1"**

**}**

**This file contains the resource's configurations on GCP that we want to run.**

**In the first section**

**resource "google\_compute\_instance" "default"**

**We're describing information about our virtual machine, such as type of image, scripting to execute when bootstrapping, and tags to identity this resource.**

**In the second section**

**resource "google\_compute\_firewall" "http-server"**

**We're describing information about our firewall and allowing access to a specific port and its protocol. If you noticed, there is information about our target. This target aims all resources that contain this tag. In other words, our virtual machine instance will be accessed through the internet because we allowed this through the firewall configuration.**

**In the last section**

**output "ip"**

**Our output will be an external ip that will print a message configured on metadata\_startup\_script**

[**https://blog.avenuecode.com/how-to-use-terraform-to-create-a-virtual-machine-in-google-cloud-platform**](https://blog.avenuecode.com/how-to-use-terraform-to-create-a-virtual-machine-in-google-cloud-platform)

**Main.tf file:**

**#download the ubuntu image**

**resource "docker\_image" "image\_id" {**

**name = "ubuntu:latest"**

**}**

**# Start a container**

**resource "docker\_container" "container\_id" {**

**name = "ubuntu\_blog"**

**image = docker\_image.image\_id.latest**

**}**

[**https://jhooq.com/how-to-setup-virtual-machine-on-google-cloud-platform/**](https://jhooq.com/how-to-setup-virtual-machine-on-google-cloud-platform/)

**kirankumarrv0990@cloudshell:~/gcp/vm (devops-test-316311)$ terraform init**

**Initializing the backend...**

**Initializing provider plugins...**

**- Reusing previous version of hashicorp/google from the dependency lock file**

**- Installing hashicorp/google v3.78.0...**

**- Installed hashicorp/google v3.78.0 (signed by HashiCorp)**

**Terraform has been successfully initialized!**

**You may now begin working with Terraform. Try running "terraform plan" to see**

**any changes that are required for your infrastructure. All Terraform commands**

**should now work.**

**If you ever set or change modules or backend configuration for Terraform,**

**rerun this command to reinitialize your working directory. If you forget, other**

**commands will detect it and remind you to do so if necessary.**

**After DELETING Terraform state file:**

**I am trying to Destroy Infra which was created by Terraform apply command but No luck because we dont have terraform state file here.**

**kirankumarrv0990@cloudshell:~/gcp/vm (devops-test-316311)$ terraform destroy**

**No changes. No objects need to be destroyed.**

**Either you have not created any objects yet or the existing objects were already deleted outside of Terraform.**

**Destroy complete! Resources: 0 destroyed.**