

GIT:

1. What is use of Git?

1. **Version Control:** Git allows multiple developers to work on the same project without conflicts by tracking changes over time.
2. **Collaboration:** Teams can collaborate more effectively using branches and merge requests, enabling parallel development.
3. **Backup and Restore:** Changes made in repositories can be committed and stored, allowing easy restoration of previous versions if needed.
4. **Distribution:** Git is a distributed system, so every developer has a full local copy of the repository, enhancing redundancy.
5. **Branching and Merging:** Developers can create branches to develop features independently and then merge them back into the main project seamlessly.
6. **History Tracking:** Git records a complete history of changes, making it easier to audit, review, and understand project evolution.

2. What is the use of a PAT and SSH key in GitHub and how do you implement them?

Uses of PAT and SSH Key in GitHub

Personal Access Token (PAT)

- **Authentication:** A PAT provides a secure way to access GitHub's API and authenticate without using a password.
- **Fine-Grained Access Control:** Users can set specific permissions for the token, limiting its use to certain repositories or actions.
- **Automation:** PATs are useful in scripts or applications that need to interact with GitHub (e.g., CI/CD pipelines).

SSH Key

- **Secure Connection:** SSH keys enable a secure, encrypted connection to GitHub, eliminating the need for password authentication.
- **Ease of Use:** Once set up, users can push and pull changes without entering passwords repeatedly.
- **Access Control:** Each SSH key is tied to a user account, allowing for easy management of access permissions.

Implementing a Personal Access Token (PAT)

1. Generate a PAT:

- Go to **GitHub** and navigate to **Settings**.
- Click on **Developer settings > Personal access tokens**.
- Select **Tokens (classic)**, then click on **Generate new token**.
- Specify required scopes (permissions) based on your needs.
- Click **Generate token** and copy the token string.

2. Use the PAT:

- When performing Git operations that require authentication, you can use the PAT instead of your password.

Implementing an SSH Key

1. Generate an SSH Key

Open your terminal and run:

- `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`

Press **Enter** to accept the default location and enter a passphrase if desired

2. Add SSH Key to GitHub

Copy the public key:

- Go to **GitHub > Settings > SSH and GPG keys.**
- Click on **New SSH key**, paste the key, give it a title, and click **Add SSH key**.

4. Use SSH for Git Operations

Update your Git remote URL:

- `git remote set-url origin git@github.com:username/repo.git`

3. In which case do we get a revert conflict and how do you resolve it?

A revert conflict occurs in Git when you attempt to revert a commit that introduces changes which conflict with subsequent commits in your branch. This means that Git can't automatically apply the changes from the commit you're trying to revert because it would disrupt the changes made after it.

When Revert Conflicts Occur

- **Subsequent Changes:** Modifications made after the commit you're reverting.
- **Merged Changes:** Conflicts arise from merges with diverging branches.
- **Conflicting File States:** Alterations in files affected by the commit.

Resolving Revert Conflicts

1. **Create a File:** Added a line; committed as "c1 cmt."
2. **Add Line:** Added a second line; committed as "c2 cmt."
3. **Add Another Line:** Added a third line; committed as "c3 cmt."
4. **Revert Commit:** Reverted "c2 cmt," creating a revert commit will get a conflict.

```
saich@SFTY-I018 MINGW64 ~/revert-scenario (master)
$ git log --oneline
3e9ef10 (HEAD -> master) c3 cmt
a0d82a6 c2 cmt
ee54528 c1 cmt

saich@SFTY-I018 MINGW64 ~/revert-scenario (master)
$ git revert a0d82a6|
```

```
saich@SFTY-I018 MINGW64 ~/revert-scenario (master)
$ git revert a0d82a6
Auto-merging f1
CONFLICT (content): Merge conflict in f1
error: could not revert a0d82a6... c2 cmt
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
hint: You can instead skip this commit with "git revert --skip".
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
hint: Disable this message with "git config set advice.mergeConflict false"
```

5. **Stage Resolved Files:** Run git add <file>.

6. **Complete the Revert:** Execute git revert --continue

7. **Commit Changes:** Finalize with git commit -m "Resolved conflicts from reverting <commit>"

4. How to clone the particular folder from git repo?

- Create one Directory
- Navigate to the Directory
- Initialize a New Git Repository
- Add the Remote Repository

```
git remote add -f origin <repository-url>
```

- **Fetch All Data from Remote**

```
git fetch origin
```

- Create a New Branch

```
git checkout -b <new-branch-name> origin/<branch-name>
```

- Sparse Checkout Configuration

```
git config core.sparseCheckout true
```

- Specify Folder to Clone

```
path/to/folder/*
```

- Checkout the Specific Folder

```
git checkout <branch-name>
```

```
ara@SFTY-I007 MINGW64 ~
cd onefile
ara@SFTY-I007 MINGW64 ~/onefile
git init
initialized empty Git repository in C:/users/vara/onefile/.git/
ara@SFTY-I007 MINGW64 ~/onefile (master)
git remote add origin -f https://github.com/raghukadali/pull-repo.git
updating origin
emote: Counting objects: 100% (21/21), done.
emote: Compressing objects: 100% (14/14), done.
emote: Total 21 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
npacking objects: 100% (21/21), 6.37 KiB | 39.00 KiB/s, done.
remote: Processing changes: 100% (21/21), done.
From https://github.com/raghukadali/pull-repo
 * [new branch]      main      -> origin/main
 * [new branch]      new       -> origin/new

ara@SFTY-I007 MINGW64 ~/onefile (master)
git remote -v
origin https://github.com/raghukadali/pull-repo.git (fetch)
origin https://github.com/raghukadali/pull-repo.git (push)
ara@SFTY-I007 MINGW64 ~/onefile (master)
git config core.sparseCheckout true
ara@SFTY-I007 MINGW64 ~/onefile (master|SPARSE)
git sparse-checkout set a.txt
ara@SFTY-I007 MINGW64 ~/onefile (master|SPARSE)
ls
ara@SFTY-I007 MINGW64 ~/onefile (master|SPARSE)
git pull origin main
rom https://github.com/raghukadali/pull-repo
 * branch      main      -> FETCH_HEAD
ara@SFTY-I007 MINGW64 ~/onefile (master|SPARSE)
ls
.a.txt
ara@SFTY-I007 MINGW64 ~/onefile (master|SPARSE)
```

5. Simulate a rebase conflict, merge conflict and resolve it.

```
# Create a new directory and initialize a Git repository
```

```
mkdir conflict
```

```
cd conflict
```

```
git init
```

Configure Git user information

```
git config user.name "chandupolina"
```

```
git config user.email "chandupolina95@gmail.com"
```

Create the initial file and commit

```
echo "first line" > demo.txt
```

```
echo "second line" >> demo.txt
```

```
git add .
```

```
git commit -m "c1"
```

Create a feature branch and add changes

```
git checkout -b feature
```

```
echo "third line" >> demo.txt
```

```
git add .
```

```
git commit -m "f1"
```

Switch back to the main branch and make conflicting changes

```
git checkout main
```

```
echo "Line 3 from main branch" >> demo.txt
```

```
git add demo.txt
```

```
git commit -m "c2"
```

Attempt to merge the feature branch, simulating a merge conflict

```
git merge feature
```

```
saich@SFTY-I018 MINGW64 ~/merge-rebase (main)
$ git merge feature
Auto-merging demo.txt
CONFLICT (content): Merge conflict in demo.txt
Automatic merge failed; fix conflicts and then commit the result.

saich@SFTY-I018 MINGW64 ~/merge-rebase (main|MERGING)
$ vi demo.txt

saich@SFTY-I018 MINGW64 ~/merge-rebase (main|MERGING)
$ git add .

saich@SFTY-I018 MINGW64 ~/merge-rebase (main|MERGING)
$ git commit -m "c3"
[main d1ce29d] c3
```

Resolve merge conflict

```
echo "Merge conflict in demo.txt. Editing to resolve..."
```

Open the demo.txt and modify as needed

```
vi demo.txt # edit the file as you needed
```

```
git add demo.txt
```

```
git commit -m "c3"
```

Create another feature branch and make changes

```
git checkout -b feature1
```

```
echo "Another line from another feature" >> demo.txt
```

```
git add demo.txt
```

```
git commit -m "f1 cmt"
```

Switch back to the main and make further changes

```
git checkout main
```

```
echo "Another change from main" >> demo.txt  
git add demo.txt  
git commit -m "c5 cmt"
```

Rebase the another-feature-branch onto main, simulating a rebase conflict

```
git checkout another-feature-branch  
git rebase main || true
```

Resolve rebase conflict

```
echo "Rebase conflict in demo.txt. Editing to resolve..."
```

Open the demo.txt and modify as needed

```
vi demo.txt # open the demo.txt and modify the changes as you needed  
git add demo.txt  
git rebase --continue
```

6. What is the use of a fork in GitHub, why do we need it, and how do you implement it?

The Use of Forks in GitHub

A fork in GitHub is a personal copy of someone else's repository that allows you to freely experiment with changes without affecting the original project. It is essential for collaborative workflows, particularly in open-source projects.

Reasons for Using a Fork

1. **Experimentation:** Work on features or fixes without impacting the original codebase.
2. **Contributions:** Make changes to a repository you don't have write access to, allowing you to propose changes via pull requests.
3. **Independent Work:** Personalize or modify a project as needed for your own use.

How to Implement a Fork:

- **Fork the Repository:** Click the "Fork" button on the original repo.

- **Clone Your Fork:**

```
git clone https://github.com/chandupolina/newrepo.git
```

```
cd newrepo
```

- **Set Up Remote:**

```
git remote add upstream https://github.com/chandupolina/newrepo.git
```

- **Fetch and Update:**

```
git fetch upstream
```

```
git checkout main
```

```
git merge upstream/main
```

- **Make Changes:** Create a new branch, make changes, and commit.

```
git checkout -b feature
```

- **Push Changes:**

```
git push origin feature
```

- **Create a Pull Request:** Submit your changes to the original repository via GitHub.

```
aich@SFTY-I018 MINGW64 ~
$ cd newrepo/
aich@SFTY-I018 MINGW64 ~/newrepo (main)
$ git remote add upstream https://github.com/chandupolina/newrepo.git

aich@SFTY-I018 MINGW64 ~/newrepo (main)
$ git fetch upstream
from https://github.com/chandupolina/newrepo
 * [new branch] main      -> upstream/main

aich@SFTY-I018 MINGW64 ~/newrepo (main)
$ git checkout main
Already on 'main'
Your branch is up to date with 'origin/main'.

aich@SFTY-I018 MINGW64 ~/newrepo (main)
$ git merge upstream/main
Already up to date.

aich@SFTY-I018 MINGW64 ~/newrepo (main)
$ git checkout -b feature
Switched to a new branch 'feature'

aich@SFTY-I018 MINGW64 ~/newrepo (feature)
$ git push origin feature
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:     https://github.com/chandupolina/newrepo/pull/new/feature
remote:
To https://github.com/chandupolina/newrepo.git
 * [new branch] feature -> feature

aich@SFTY-I018 MINGW64 ~/newrepo (feature)
```

7. Reset a file to its last committed state.

To reset a file to its last committed state in Git, you can use the following command:

Create one file and make three commits on that file c1 , c2 , c3

And again modify the file but dont commit it

Currently the file is in Untracked State

If you execute:

- git checkout -- f1

The latest changes that we made on f1 file will be removed and set the file to its last committed state

```
saich@SFTY-I018 MINGW64 ~/check (master)
$ git checkout -- f1

saich@SFTY-I018 MINGW64 ~/check (master)
$ git log --oneline
f36f338 (HEAD -> master) c3
d43787c c2
c7f630d c1

saich@SFTY-I018 MINGW64 ~/check (master)
$ vi f1

saich@SFTY-I018 MINGW64 ~/check (master)
```

8. How can give access to others for private repo.

To give access to others for a private repository in GitHub, follow these steps:

Granting Access to a Private Repository

1. Navigate to Your Repository

- Go to your private repository on GitHub.

2. Open the Settings

- Click on the "**Settings**" tab located in the repository menu.

3. Manage Access

- On the left sidebar, click on "**Manage access**".

4. Invite Collaborators

- Click on the "**Invite teams or people**" button.
- Enter the username or email of the person you wish to invite.

5. Set Permissions

- Choose the appropriate permission level:
 - **Read**: Can view the repository.
 - **Write**: Can view and edit the repository.
 - **Admin**: Can manage settings and users.

6. Send Invitation

- Click "**Invite**" to send the invitation.

7. Acceptance of Invitation

- The invited user needs to accept the invitation via their email or GitHub notifications.

9. Cherry-pick a specific commit from another branch into the current branch.

- Create one directory
- Switch in to that directory
- Create one file and make one commit on that file
- And Create one feature branch and switch in to it and make 2 commits on the same file

- Again switch into master branch and execute
git cherry-pick <commit-hash>
- Means we are merging a single commit from feature branch in to the master branch instead of merging a complete branch

```
saich@SFTY-I018 MINGW64 ~/cherry (feature)
$ git log --oneline
3b848ad (HEAD -> feature) f2
91783f9 f1
e1aa4fc (master) m1

saich@SFTY-I018 MINGW64 ~/cherry (feature)
$ git checkout master
Switched to branch 'master'

saich@SFTY-I018 MINGW64 ~/cherry (master)
$ git log --oneline
e1aa4fc (HEAD -> master) m1

saich@SFTY-I018 MINGW64 ~/cherry (master)
$ git cherry-pick 91783f9
[master b3f49d9] f1
Date: Fri Oct 10 10:56:24 2025 +0530
1 file changed, 1 insertion(+)

saich@SFTY-I018 MINGW64 ~/cherry (master)
$ git log --oneline
b3f49d9 (HEAD -> master) f1
e1aa4fc m1

saich@SFTY-I018 MINGW64 ~/cherry (master)
```

10. How do you retrieve files from a specific commit?

- Create one directory
- Switch to that directory
- Initialize git
- Create one file and make three commits on that file m1 , m2 , m3
- And Execute this command

```
git checkout <commit-hash> – filepath
```

```
saich@SFTY-I018 MINGW64 ~/retrieve
$ git init
Initialized empty Git repository in C:/Users/saich/retrieve/.git/
saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ vi f1

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ git add . && git commit -m "m1 cmt"
warning: in the working copy of 'f1', LF will be replaced by CRLF the next time Git touches it
[master (root-commit) 8b4d963] m1 cmt
 1 file changed, 1 insertion(+)
 create mode 100644 f1

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ vi f1
saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ git add . && git commit -m "m2 cmt"
warning: in the working copy of 'f1', LF will be replaced by CRLF the next time Git touches it
[master 9a9a620] m2 cmt
 1 file changed, 1 insertion(+)

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ vi f1

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ git add . && git commit -m "m3 cmt"
warning: in the working copy of 'f1', LF will be replaced by CRLF the next time Git touches it
[master eb957a9] m3 cmt
 1 file changed, 1 insertion(+)

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ git log --oneline
eb957a9 (HEAD -> master) m3 cmt
9a9a620 m2 cmt
8b4d963 m1 cmt

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ git checkout 9a9a620 -- f1

saich@SFTY-I018 MINGW64 ~/retrieve (master)
$ vi f1
```

11. How do you push files to a private GitHub repository

Steps to Push Files to a Private GitHub Repository

1. Create a Private Repository on GitHub

- Log in to your GitHub account.
- Click on the "+" icon in the top right corner and select "New repository."
- Give your repository a name, set it to private, and click "Create repository."

2. Initialize Local Repository (if not already done)

- Open your terminal and navigate to the folder containing your project.
- If you haven't initialized a Git repository yet, run:

```
git init
```

3. Add Remote Repository

- Connect your local repository to the GitHub repository by adding a remote URL.
Replace <username> with your GitHub username and <repository> with your repository name:

```
git remote add origin https://github.com/<username>/<repository>.git
```

4. Stage Files for Commit

- Add the files you want to push to the staging area:

```
git add .
```

- This command stages all files in the directory. You can specify specific files instead of using . if needed.

5. Commit the Changes

- Commit your changes with a descriptive message:

```
git commit -m "Initial commit with my files"
```

6. Push to GitHub

- Push your changes to the private GitHub repository:

```
git push -u origin master
```

- If you're pushing to a different branch (e.g., main), replace master with main.

7. Provide Authentication

- If prompted, enter your GitHub username and password or personal access token for authentication.

8. Verify the Push

- Visit your GitHub repository in your web browser to confirm that your files have been successfully uploaded.

12. How to Create a Pull Request and Why We Need It

Steps to Create a Pull Request

1. Push Your Branch:

- Ensure your branch with changes is pushed to the remote repository:

```
git push origin <branch-name>
```

2. Go to GitHub:

- Navigate to the repository on GitHub.

3. Open the Pull Requests Tab:

- Click on the "Pull requests" tab.

4. Click "New Pull Request":

- Click on the "New pull request" button.

5. Select Base and Compare Branches:

- Set the base branch (usually main or master).
 - Select the compare branch (the branch with your changes).
6. Add Details:
- Provide a title and description for your pull request, explaining the changes made.
7. Create Pull Request:
- Click the "Create pull request" button.

Why We Need Pull Requests

- Code Review: Allow for feedback and discussion on changes before merging, promoting code quality.
- Collaboration: Facilitate teamwork, making it easier for multiple developers to work on the same project.
- Version Control: Help maintain a history of changes, which is crucial for tracking progress and debugging.

13. How to Perform Version Control Using Git

Key Points of Version Control Using Git

1. Initialize a Repository:
 - Use git init to start a new repository.
2. Commit Changes:
 - Use git add to stage changes and git commit to save them with informative messages.
3. Branching:
 - Create branches using git branch <branch-name> to manage features, bug fixes, or experiments independently.

4. Merging:
 - Merge branches with git merge <branch-name> to incorporate changes into the main codebase.
5. Tracking Changes:
 - Use commands like git log to view commit history and git status to see the current state of the repository.
6. Collaboration:
 - Use git pull and git push to synchronize changes with the remote repository.

14. How to Create a Branch Ruleset

Steps to Create a Branch Ruleset (GitHub)

1. Navigate to Your Repository:
 - Go to your repository on GitHub.
2. Access Settings:
 - Click on the "Settings" tab in the repository.
3. Branches Section:
 - In the left sidebar, click on "Branches."
4. Add Branch Protection Rules:
 - Click on "Add branch protection rule."
5. Specify Branch Name Pattern:
 - Enter the branch name pattern (e.g., main, release/*) to apply the rules to.
6. Select Protection Settings:
 - Choose the protections you want, such as requiring pull requests, requiring review before merging, or preventing force pushes.
7. Save Changes:
 - Click "Create" or "Save changes" to apply the ruleset.

Docker:

1. Package a Python or Node.js app into a Docker image for easy sharing.

Dockerfile

1) Package a Python or Node.js app into a Docker image for easy sharing.

Dockerfile

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY ..

EXPOSE 5000

CMD ["python", "app.py"]
```

BUILD

```
docker build -t my-python-app
```

RUN THE CONTAINER

```
docker run -d -p 5000:5000 my-python-app
```

2) Run a container with mounted volume to store logs or user data.

Create a Volume:

```
docker volume create volume1
```

Attach the created volume using this command:

- Create one image or use an image available in central repository
- And Run this command to attach the created volume to the container

- docker run -d --name container1 -P --mount source=vol1,target=/data/app img:v1
- After Running this command check the available running containers
- docker ps

```
root@vm-instance:~# docker run -d --name container1 -P --mount source=vol1,target=/data/app img1:v1
dad78604006ed41d3eb01b49d534317d63119b2778ce5d161f04cb32754fdbd1
root@vm-instance:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
dad78604006e img1:v1 "python app.py" 5 seconds ago Up 4 seconds 0.0.0.0:49153->5000/tcp container1
e0aa7c3a26dd img1:v1 "python app.py" 3 hours ago Up 3 hours 0.0.0.0:6578->5000/tcp h1con
root@vm-instance:~#
```

- Check this created volume in this path /var/lib/docker/volumes in the host machine
- To inspect the volume details
docker volume inspect vol1

3. Use Docker Compose to run multi image at a time (python image and java or web image)

Docker-compose File:

```
services:
  python_app:
    build: ./python_app
    ports:
      - "5000:5000"
    volumes:
      - ./python_app:/app
    networks:
      - my_network

  java_app:
    build: ./java_app
    networks:
      - my_network
    ports:
      - "8081:8080"

networks:
  my_network:
    driver: bridge
```

Run the Docker-Compose:

```
docker-compose up -d
```

Down the Docker-Compose:

```
docker-compose down (it will delete the containers)
```

4. Demonstrate the image optimization process in Docker.**Image Optimization Process in Docker**

1. Use Multi-Stage Builds:

- Create multiple stages in a Dockerfile to separate build and runtime dependencies.
- Only copy necessary artifacts to the final image, reducing size.

2. Choose a Minimal Base Image:

- Start with a smaller base image (e.g., alpine or distroless) to minimize the initial size.

3. Reduce Layers:

- Combine commands using && to minimize the number of layers created by the Dockerfile, as each command creates a new layer.

4. Limit the Use of RUN Commands:

- Instead of multiple RUN commands, group them together to consolidate layer duplication.

5. Optimize Dependencies:

- Remove unnecessary packages and dependencies, especially during the build process.
- Use the --no-cache option when installing packages to avoid downloading cache files.

6. Utilize .dockerignore File:

- Exclude unnecessary files and directories (like .git, node_modules, etc.) from being copied into the image with a .dockerignore file.

7. Minimize Environment Variables:

- Use only the required environment variables to limit information stored in the image.

8. Use Specific Versions:

- Specify exact versions of packages in the Dockerfile instead of using latest, which can lead to larger images when updates occur.

9. Clean Up Unused Packages:

- Use cleanup commands (e.g., apt-get clean, rm -rf /var/lib/apt/lists/*) to remove temporary and cached files after installations.

10. Rebuild Regularly:

- Regularly rebuild images to pick up optimizations and remove outdated dependencies.

5) Push Docker image to private registry with versioning for release control.

```

root@ansible-host:~# docker tag nginx:latest cpolina/python-images:v1
root@ansible-host:~# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
cpolina/python-images   v1      07ccdb783875  2 days ago   160MB
nginx               latest   07ccdb783875  2 days ago   160MB
hello-world          latest   1b44b5a3e06a  2 months ago  10.1kB
root@ansible-host:~# docker login

USING WEB-BASED LOGIN

[ Info → To sign in with credentials on the command line, use 'docker login -u <username>' ]

Your one-time device confirmation code is: LPHK-WZTM
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate

Waiting for authentication in the browser...

WARNING! Your credentials are stored unencrypted in '/root/.docker/config.json'.
Configure a credential helper to remove this warning. See
https://docs.docker.com/go/credential-store/

Login Succeeded
root@ansible-host:~# docker push cpolina/python-images:v1
The push refers to repository [docker.io/cpolina/python-images]
d6eb78ef52a2: Mounted from library/nginx
d009686a1d10: Mounted from library/nginx
8117ecf0e00c: Mounted from library/nginx
98da25895b87: Mounted from library/nginx
96d86bc8de59: Mounted from library/nginx
d61356d6b00c: Mounted from library/nginx
1d46119d249f: Mounted from library/nginx
v1: digest: sha256:2a32ec90b80e9c0ced57db0da93969f0ae71840945e7e874a154de511bf973c4 size: 1778
root@ansible-host:~# 
```

- Here we have one image nginx in the local and we need to push that image to dockerhub private registry
- Create one private registry in dockerhub (“cpolina/python-images”)
- We have 2 images

```
root@ansible-host:~# docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
nginx          latest    07ccdb783875  2 days ago   160MB
hello-world    latest    1b44b5a3e06a  2 months ago  10.1kB
```

- We need to push the nginx image to the private registry
- Tag that image
- `root@ansible-host:~# docker tag nginx:latest cpolina/python-images:v1`
- And login to the docker hub account
- And finally push the image to remote registry .

6) Demonstrate the networking concept with real time example.

The Docker network is a virtual network created by Docker to enable communication between Docker Containers. If two containers are running on the same host they can communicate with each other without the need for ports to be exposed to the host machine

Network Drivers

There are several default network drivers available in Docker and some can be installed with the help of plugins, Command to see the list of containers in Docker mentioned below.

- docker network ls

Types of Network Drivers

1. **bridge:** If you build a container without specifying the kind of driver, the container will only be created in the bridge network, which is the default network.
2. **host:** Containers will not have any IP address they will be directly created in the system network which will remove isolation between the docker host and containers.
3. **none:** IP addresses won't be assigned to containers. These containments are not accessible to us from the outside or from any other container.

4. **overlay**: overlay network will enable the connection between multiple Docker demons and make different Docker swarm services communicate with each other.
5. **ipvlan**: Users have complete control over both IPv4 and IPv6 addressing by using the IPvlan driver.
6. **macvlan**: macvlan driver makes it possible to assign MAC addresses to a container.

1. Understanding the Docker Network Command

With the help of the "Create" command, we can create our own docker network and can deploy our containers in it.

- `sudo docker network`

2. Using Docker Network Create command

With the help of the "Create" command, we can create our own docker network and can deploy our containers in it.

- `sudo docker network create –driver <driver-name> <bridge-name>`

3. Using the Docker Network Connect command

Using the "**Connect**" command, you can connect a running Docker Container to an existing Network.

- `sudo docker network connect <network-name> <container-name or id>`

4. Using the Docker Network Inspect command

Using the Network Inspect command, you can find out the details of a Docker Network.

- `sudo docker network inspect <network-name>`

You can also find the list of Containers that are connected to the Network.

5. Using the Docker Network ls command

To list all the Docker Networks, you can use the **list** command.

- sudo docker network ls

7. Using the Docker Network rm command

You can remove a Docker Network using the **rm** command.

- sudo docker network rm <network-name>

Note that if you want to remove a network, you need to make sure that no container is currently referencing the network.

8. Using the Docker Network prune command

To remove all the unused Docker Networks, you can use the **prune** command.

- sudo docker network prune

Terraform:

1. Create multiple VM instances at once using count.

```
resource "google_compute_instance" "tf_vm" {
  name = "tf-vm-${count.index}"
  machine_type = "e2-medium"
  zone = "us-central1-a"
  count = 2
  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }
  network_interface {
    network = "default"
    access_config {}
  }
}
```

2. Create a Cloud Storage bucket, push a local file into it and make show all files must access.

```
resource "random_id" "tf_randomid" {
  byte_length = 5
```

```

}

#bucket creation
resource "google_storage_bucket" "tf_bucket" {
  name = "${var.project_id}-${random_id.tf_randomid.hex}"
  location = "US"
  #public_access_prevention = true
}

# public access to bucket
resource "google_storage_bucket_iam_member" "public_permission" {
  bucket = google_storage_bucket.tf_bucket.name
  role = "roles/storage.objectviewer"
  member = "allUsers"
}

# file uploading to my local to bucket
resource "google_storage_bucket_object" "tf_object" {
  name = "chandu.txt"
  bucket = google_storage_bucket.tf_bucket.name
  source = "./chandu.txt"
}

```

3. Create a VM that must deploy the website at time of creation.

```

resource "google_compute_instance" "tf_gce" {
  name = "use-case-vm1"
  zone = "us-central1-a"
  machine_type = "e2-medium"
  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-its"
    }
  }
  network_interface {
    network = "default"
    access_config {
      }
  }
  desired_status = "RUNNING"
  metadata_startup_script = file("script.sh")
}

```

```
# output block
output "jenkins_url" {
  value    =
"http://${google_compute_instance.tf_gce.network_interface[0].access_config[0].nat_ip}:8080"
  description = "Jenkins UI URL"
}
```

script.sh:

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

```
# Update system
sudo apt update -y
sudo apt install fontconfig openjdk-21-jre -y
```

```
# Verify Java installation
java -version
```

```
# Add Jenkins repository and key
sudo wget -O /etc/apt/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
```

```
echo "deb [signed-by=/etc/apt/keyrings/jenkins-keyring.asc]" \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
```

```
sudo apt-get update -y
sudo apt-get install jenkins -y
```

4. Create a custom VPC and firewall rule, then attach to a VM.

```
resource "google_compute_network" "custom_network" {
  name        = "custom-network"
  auto_create_subnetworks = false
}
resource "google_compute_subnetwork" "custom_subnetwork" {
  name      = "custom-subnetwork"
  ip_cidr_range = "10.0.0.0/24"
```

```

network      = google_compute_network.custom_network.id
region       = "us-central1"
}
resource "google_compute_firewall" "custom_firewall" {
  name    = "custom-firewall"
  network = google_compute_network.custom_network.name

  allow {
    protocol = "tcp"
    ports    = ["22", "80"]
  }

  source_ranges = ["0.0.0.0/0"]
  target_tags   = ["http"]
}

resource "google_compute_instance" "custom_vm" {
  name      = "custom-vm"
  machine_type = "e2-medium"
  zone      = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    subnetwork = google_compute_subnetwork.custom_subnetwork.id
    access_config {}
  }
  tags = ["http"]
}

```

5. While creating the VM, the local files should be pushed into it. Software like Docker and Nginx should be installed, and the Docker service should be started.

```

resource "google_compute_instance" "tf_vm1" {
  name = "chandu-vm"
  zone = "us-central1-a"
  machine_type = "e2-medium"

```

```

boot_disk {
  initialize_params {
    image = "ubuntu-os-cloud/ubuntu-2204-lts"
  }
}
network_interface {
  network = "default"
  access_config {

  }
}
metadata = {
  enable-osconfig = "TRUE"
  ssh-keys="chandupolina95:${file("C:/Users/saich/.ssh/id_ed25519.pub")}"
  startup-script = file("docker-nginx.sh")
}

provisioner "file" {
  source = "./chandu.txt"
  destination = "/home/chandupolina95/demo.txt"
}

connection {
  type = "ssh"
  user = "chandupolina95"
  private_key = file("C:/Users/saich/.ssh/id_ed25519")
  host = google_compute_instance.tf_vm1.network_interface[0].access_config[0].nat_ip
}
}

```

[docker-nginx.sh](#)

```

sudo apt update
sudo apt upgrade -y
sudo apt install openjdk-21-jdk -y
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \

```

```

$(./etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt upgrade -y
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin -y
apt install tree -y
# nginx instalaltion
sudo apt update
sudo apt install nginx -y

```

6. Create two VM's each VM in different project.

```

# first-vm

resource "google_compute_instance" "tf_instance" {
  name = "vm1"
  zone = "us-central1-a"
  machine_type = "e2-medium"
  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-lts"
    }
  }
  network_interface {
    network = "default"
    access_config {

    }
  }
}

#second-vm
resource "google_compute_instance" "tf_instance1" {
  name = "vm2"
  zone = "us-central1-a"
  machine_type = "e2-medium"
  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-2204-lts"
    }
  }
  network_interface {
    network = "default"
  }
}

```

```
access_config {  
}  
}  
provider=google.hari-proj  
}
```

7. How to get the state file of existing resource in GCP

```
provider "google" {  
  project = "terraform-473207"  
  region  = "us-central1"  
  zone    = "us-central1-a"  
}  
  
resource "google_compute_instance" "my_instance" {  
  # Terraform will populate this after import  
}
```

Run the below command to import the existing VM into the state file:
terraform import google_compute_instance.my_instance
projects/terraform-473207/zones/us-central1-a/instances/instance-1

8. How to store the state file in remote location.

```
terraform {  
  required_providers {  
    google = {  
      source = "hashicorp/google"  
      version = "~> 6.0"  
    }  
  }  
  backend "gcs" {  
    bucket = "bucket1project1-466607"  
    prefix = "/state/dev"  
  }  
}
```

```
}
```

9. I need to configure the two VMs in different zones with a single codebase, and I need separate state files?

```
terraform init
terraform workspace new zone-a
terraform workspace new zone-b
terraform workspace select zone-a
terraform apply -var "zone=us-central1-a"
terraform workspace select zone-b
terraform apply -var "zone=us-central1-b"

variable "zone" {
  description = "GCP zone"
  type        = string
}

provider "google" {
  project = "terraform-473207"
  region  = "us-central1"
  zone    = var.zone
}

resource "google_compute_instance" "vm" {
  name      = "my-vm-${terraform.workspace}"
  machine_type = "e2-medium"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    network = "default"
    access_config {}
  }
}
```

10. Demonstrate the module concept?

Modules are used to create reusable components inside your infrastructure.

You write some code once (like a function)

Then you reuse it multiple times without copying it

structure of modules:

terraform-project/

```
└── main.tf
└── modules/
    └── vm/
        └── main.tf
```

modules/vm/main.tf:

```
resource "google_compute_instance" "vm" {
  name = var.instance_name
  machine_type = var.machine_type
  zone = var.zone

  boot_disk {
    initialize_params {
      image = var.image
    }
  }

  network_interface {
    network = "default"
    access_config {}
  }
}
```

main.tf:

```
module "vm" {
  source = "./modules/vm"
  instance_name = "my-vm"
  machine_type = "e2-medium"
  zone = "us-central1-a"
  image = "debian-cloud/debian-11"
}
```

