Table of Contents

[1. **GIT**: 2](#_Toc159946376)

[**2.** **Docker** 22](#_Toc159946377)

[**3.** **Kubernetes** 30](#_Toc159946378)

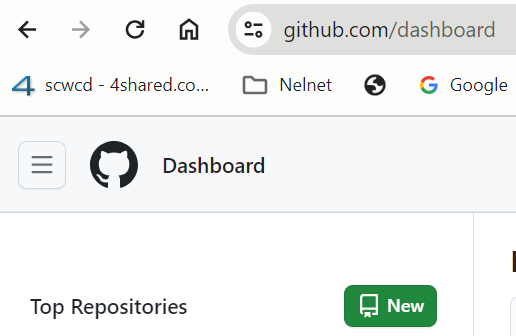
## GIT:

<https://www.youtube.com/watch?v=WAF4bnAk6Qo&list=PLVlQHNRLflP-C43xWt10PB3gKC8DDPB12&index=3>

<https://www.youtube.com/watch?v=Ga-9lzAgeFw&list=PLVlQHNRLflP-C43xWt10PB3gKC8DDPB12&index=4>

Spring Boot Logging- <https://www.youtube.com/watch?v=_A7yBhW8Fx4&list=PLVlQHNRLflP-C43xWt10PB3gKC8DDPB12&index=9>

Create new Repository by Clicking **New option** in <https://github.com/dashboard>

 A screenshot of a computer

Description automatically generated

**New🡪Repository Name(GitPractice)🡪**Click on **Create Repository** button to create Repository in Git Repository.

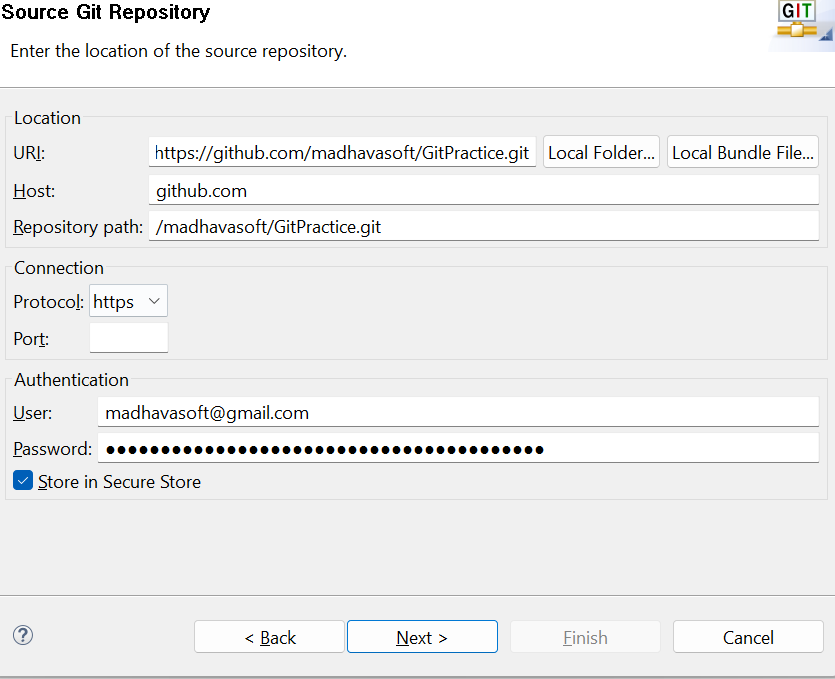
**Open Eclipse** 🡪**Window**🡪**Preferences** 🡪**Version Control**🡪**Git**🡪**Configuration**🡪**Add user emailId and Name.**

**Window🡪Show View🡪Git🡪Select (Git Repositories, Git Staging, Git Tree Compare)**

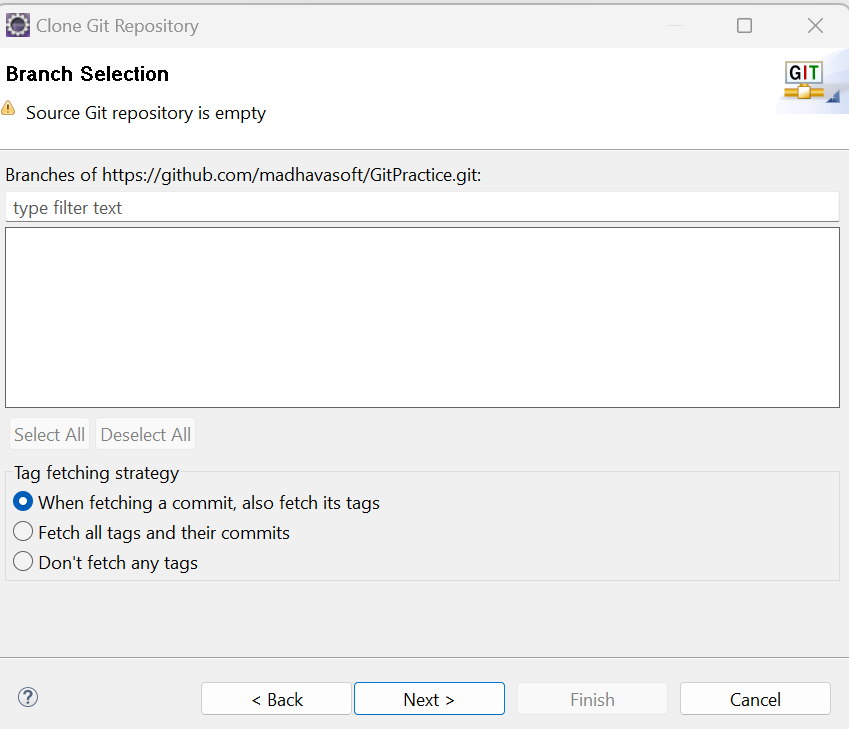
**To Import the Project**

**Git Repositories window🡪**Click on Clone a Git Repository **link🡪**Select **Clone URI🡪Provide Git URI**

A screenshot of a computer

Description automatically generated 

After providing the above details Click on Next🡪 Click on Next(on Branch Selection Window)🡪 Provide the Local Git Repository name(Default Repo name will come but local name change is optional)🡪 Now Server Git will be linked to the Local Git Repository. (Refer Below screenshots)

 A screenshot of a computer

Description automatically generated

A screenshot of a computer

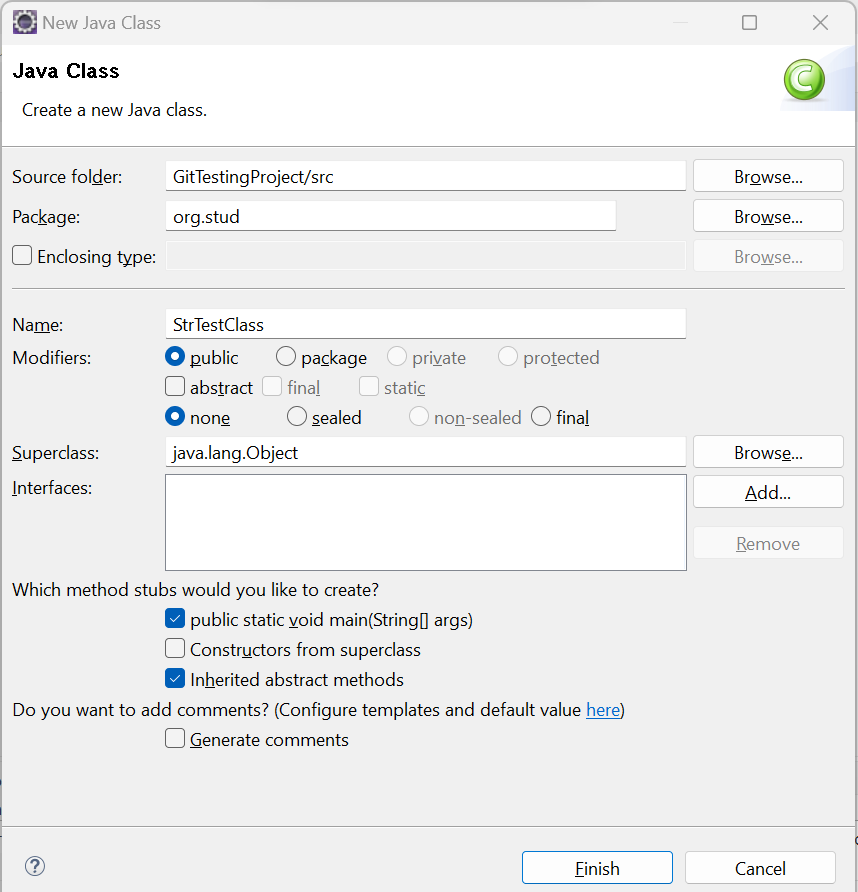
Description automatically generated

Now Create a new Java Project in Project Explorer🡪Java Project🡪provide project name🡪Click on Next🡪Finish

A screenshot of a computer

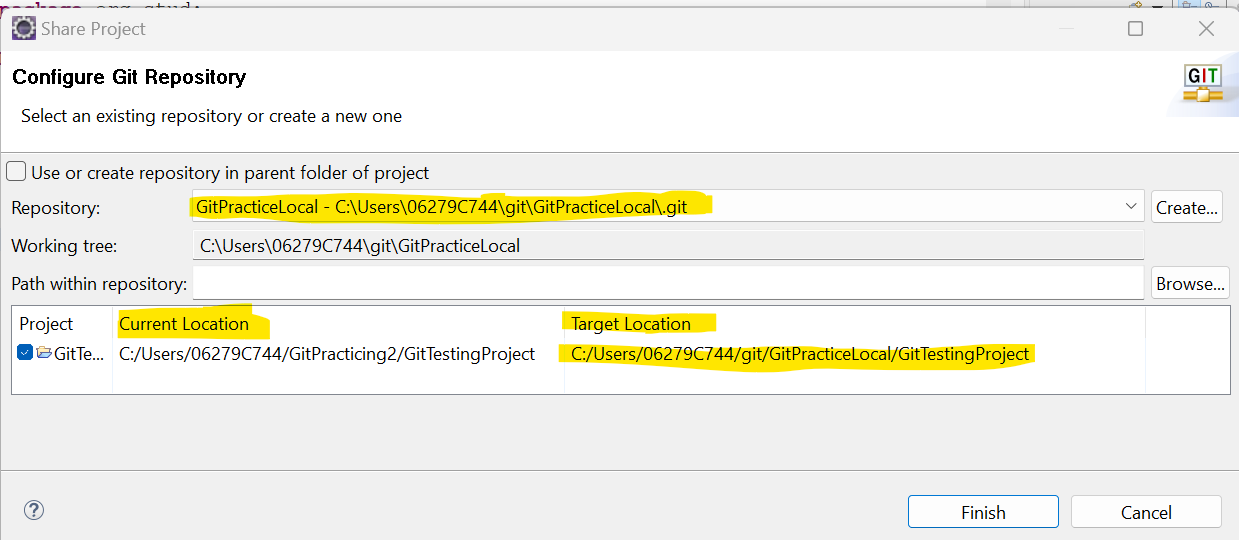
Description automatically generated

Create Test Class for Testing purpose.



**Now we need to map the local workspace with local git Repository**.

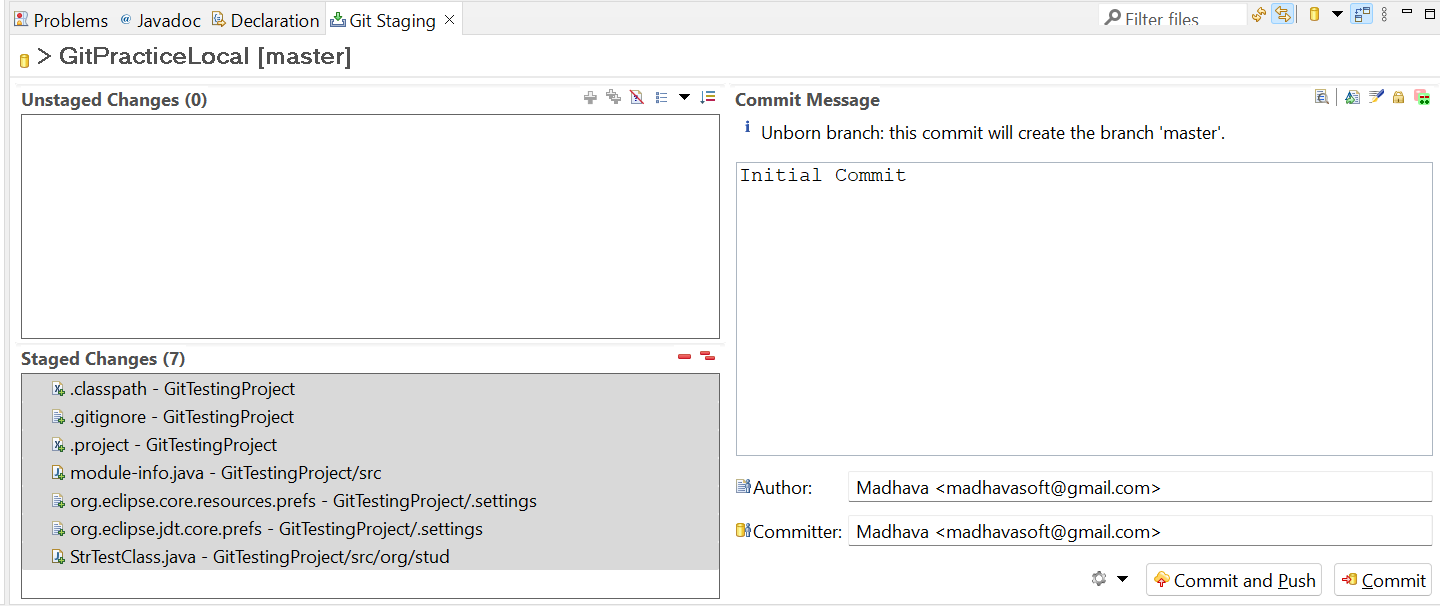
**Right Click on the project**🡪Team🡪Click on Share Project(First time only we will get this Option)🡪 Select the Repository name from Dropdown as shown below which should be mapping to the Target Location



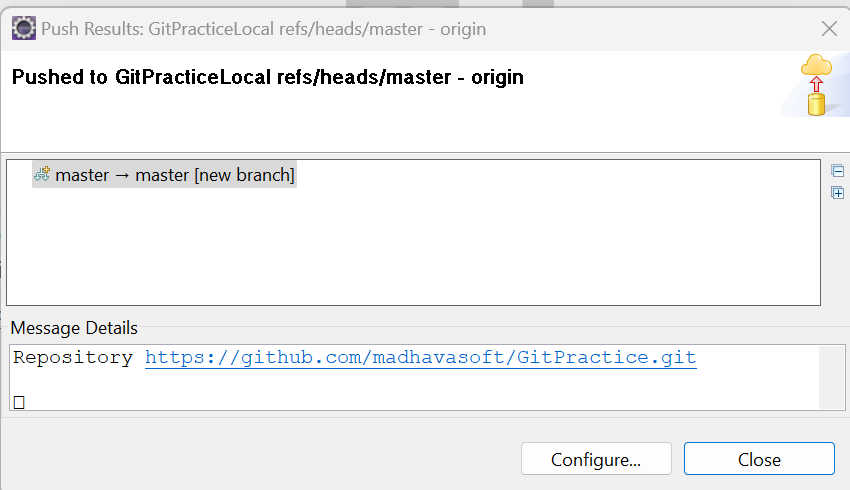
Now all the files will be reflected in **Git Staging** window.

Commit the Code by right clicking on Project 🡪 Team🡪Commit🡪all the files will be reflected in Git Staging🡪select the all the files and click on ++ (Plus) symbol to add to the Saved Changes window also provide commit message🡪Click on Commit and Push.

**Note: We must select .classpath,.project,.gitignore and others**



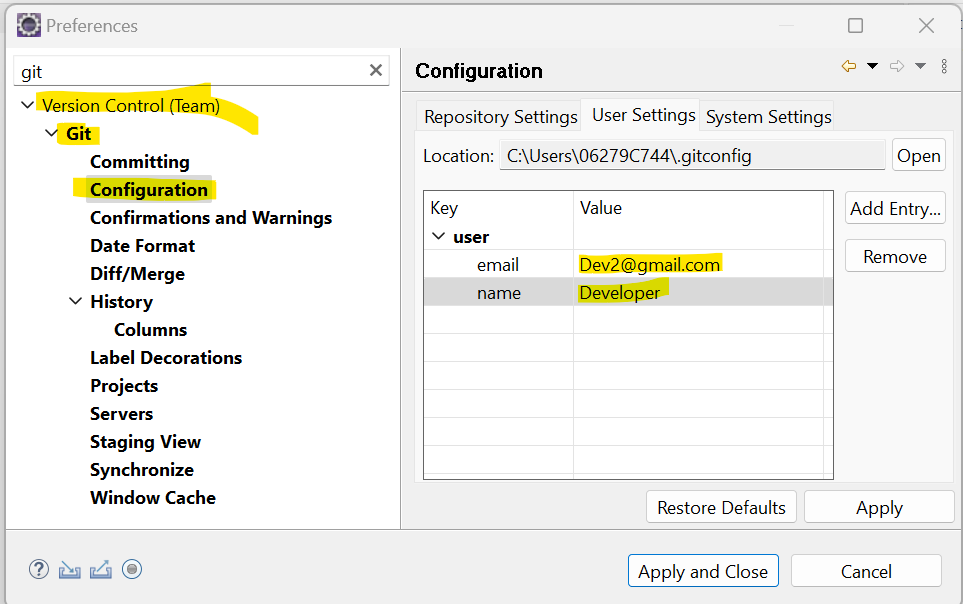
Now the Code with pushed to the Master Repository.



Now the Git Head master/Trunk is up to date with our latest code.

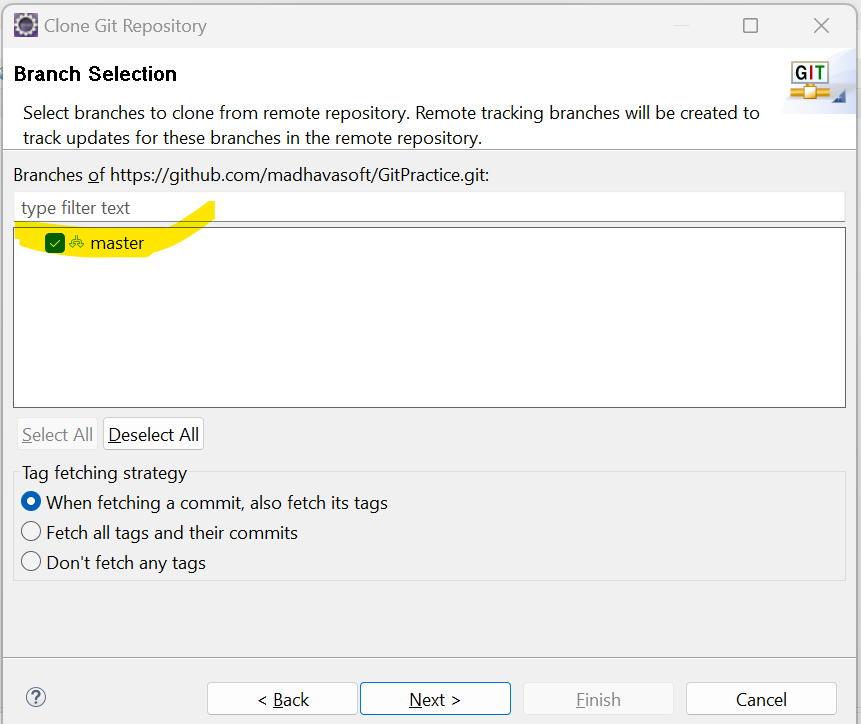
Now we can import the project in Devepoper2 system directly in Project Explorer.

Update the user details in Window🡪Preferrences🡪Version Control🡪Git🡪Configuraiton and update the user email id and name.



Click on Window🡪Show View🡪Others🡪Select git and **Select (Git Repositories, Git Staging, Git Tree Compare)**

Git Staging🡪Clone a Git Repository 🡪 and URI🡪 Click on Next and now we can see Master Repository as shown Below. Click on Next🡪Update the Directory name(Optional)🡪Finish

 A screenshot of a computer

Description automatically generated

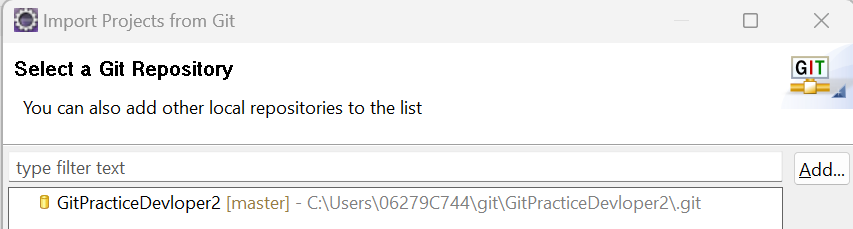
Goto Project Explorer to import the project from LocalGit Repo🡪Git🡪Projects fro Git🡪Next🡪Select Existing Local Reposotory🡪

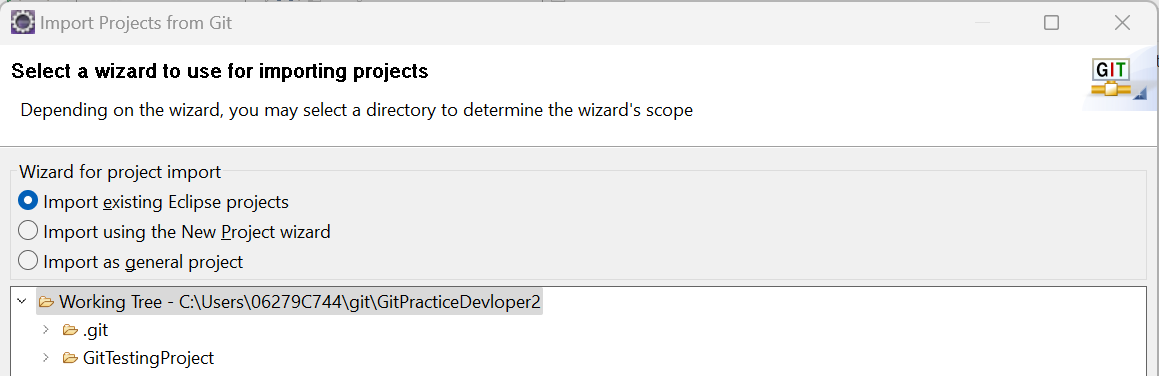
A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Select the Git Local Project as below.-->Next🡪Next🡪Finish(Below)





A screenshot of a computer

Description automatically generated

Team Ignore🡪While coming the code if we select the ignore option , those files will be ignored to check in.

**Creating Branches and Merging Branches :**

Click on Git Repository tab🡪Click on Branches🡪Local(select master)🡪Right click on master(Under Local)🡪Select on Create Branch🡪Provide Branch name🡪Finish

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Now the new Branch will be reflected under Local.

**A yellow text on a white background

Description automatically generated**

Now we should push the new Branch to Git Server

Right click on new Branch(The above selected one) 🡪Click on Push Branch🡪 Click on Push🡪Finish

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Now Developer can Checkout the code by pulling from his System.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

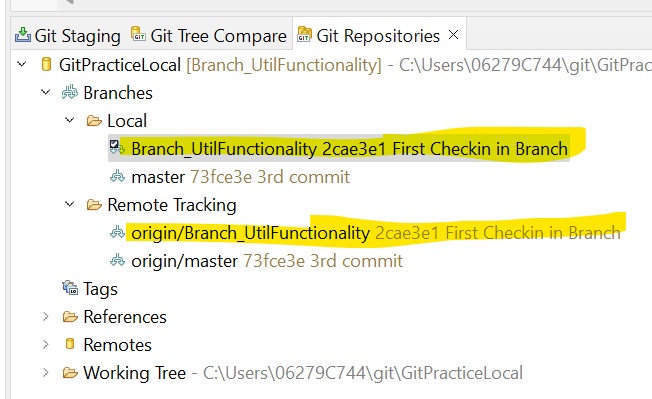
Description automatically generated A screenshot of a computer

Description automatically generated

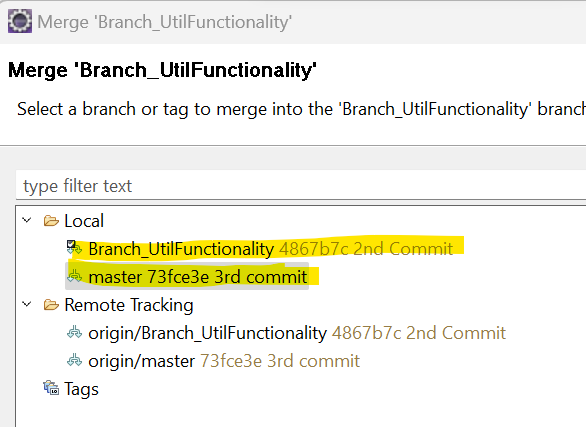
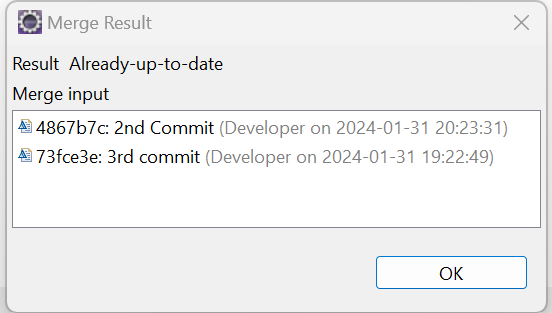
Now in Branch code , the Developer added some code and checked.

A screenshot of a computer

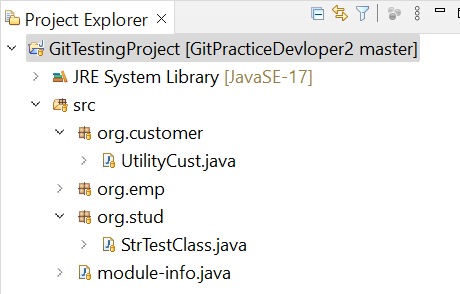
Description automatically generated



Merging Code: Refer above screen shot under Git Repositories🡪Select Local Branch and Right Click 🡪Select Merge 🡪Select master Git Repository under Local🡪 Click Merge🡪Ok(Now the Branch Code has been merged with Local Master/Truck Repo

To Verify that, Come to Master Repo and Pull the latest Code



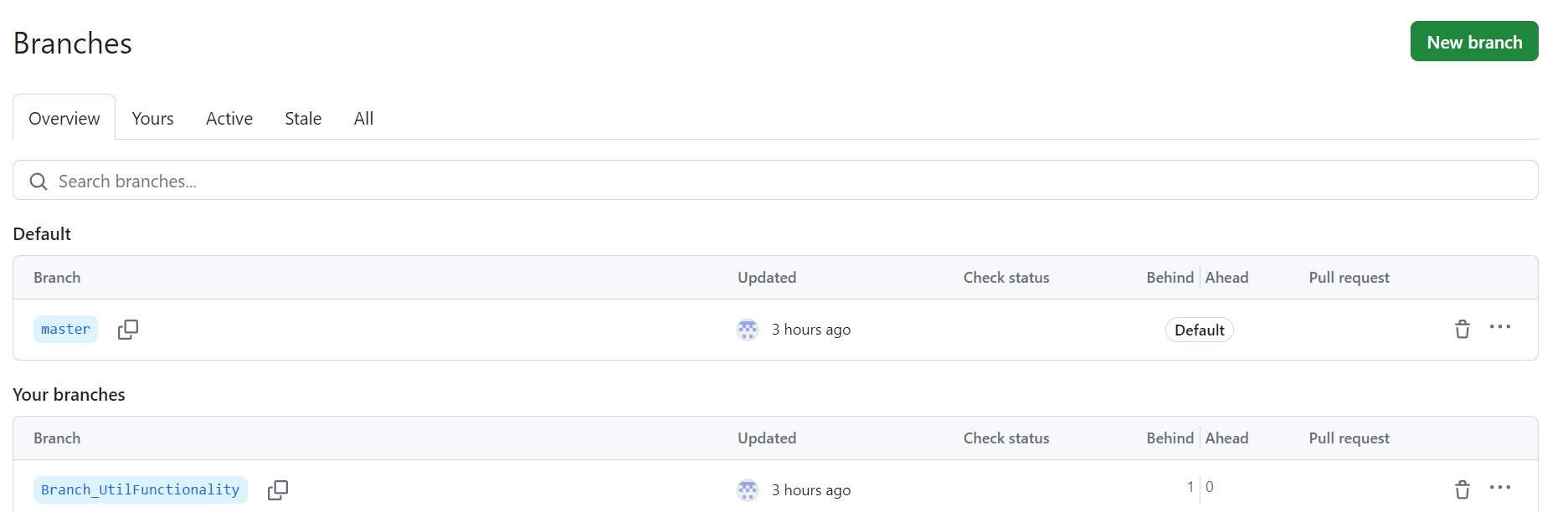
**Creating a Git Branch from GIT UI**

**Select the project which we need to create a Branch.**

**A screenshot of a computer

Description automatically generated**

**Click on Branches🡪Click on New Branch Button**

****

**Provide the New Branch Name and select the Source repo from Dropdown🡪 Click on Create new Branch**

**A screenshot of a computer

Description automatically generated**

**New Brach has been created.**

**A screenshot of a computer

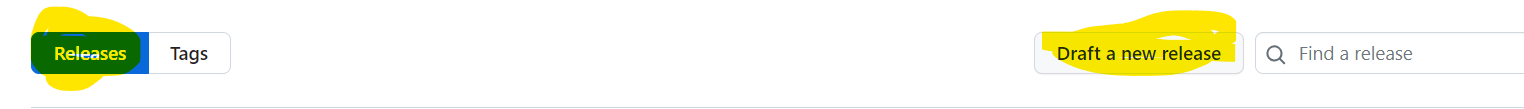
Description automatically generated**

**Creating Tags from Git UI:**

**Goto the Repository(**[**https://github.com/madhavasoft/GitPractice)--> Click**](https://github.com/madhavasoft/GitPractice)--%3e%20Click) **on Tags🡪 Click on Releases**

**A screenshot of a computer

Description automatically generated**

**🡪Click on Draft a new release🡪Click on Choose a tag🡪and Enter new tag Name🡪Click on + Create new Tag🡪 Click on Push Release Button after providing details🡪New tag will be created with hashcode number**

**A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated**

**A green square with black text

Description automatically generated A screenshot of a computer

Description automatically generated**

**Creating Tag from Eclispe Git Repositories :**

**Right Click on Tags🡪 Select Create Tag🡪**

**A screenshot of a computer

Description automatically generated**

**Provide the Tag Name🡪Click on Create Tag and Start Push button🡪Click on Next after Verifing the details**

**A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated**

**Click on Finish to Push the tag to Master Repo**

**A close-up of a confirmation

Description automatically generated**

**Reverting the Code:**

Right Click on Project🡪Team🡪Show in History🡪 Select the version which we need to revert🡪

**A screenshot of a computer

Description automatically generated**

Right Click on that version🡪Revert Commit🡪 Now that commit will be reverted in Local Git Repo🡪Now Push the code by right clicking the project🡪Team🡪Push to Origin🡪 The Code will be reverted.

**A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated**

**Stashing:**

If the developer don’t want to commit particular modified file and need to use for future purposes, we will use Stashing.

Right Click on Project🡪Team🡪Stashes🡪Stash changes🡪Enter Comment 🡪Click on Stash🡪Now that uncommitted code will be disappeared/Stashed

**A screenshot of a computer

Description automatically generated**

The Stashed Code will not goto Repository even after commit as well. If the Developer wanted to work on that Stashed Code then Right Click on Project🡪Team🡪Stashes🡪Select the Stashed Comment version🡪Click on Apply Stashed Changes(below highlighted)🡪Now those stashed changes will be reflected in Workspace🡪Just Pull the code once

**A screenshot of a computer

Description automatically generated**

**GIT VS SVN**

GIT is distributed, SVN is not:

This is by far the \*core\* difference between GIT and other non-distributed version control systems like SVN, CVS etc. If you can catch this concept well, then you have crossed half the brldge. To add a disclaimer, GIT is not the first or only distributed VCS(versiOn control system) currently available. There are other tools like Bitkeeper, Mercurial etc. which also work on distributed mode. But, GIT does it better and comes with much more powerful features.GIT like SVN do have centralized repository or server. But, GIT is more intended to be used in distributed mode which means, every developer checking out code from central repository/server will have their own cloned repository installed on their machine. Let’s say if you are stuCk somewhere where you don't have network connectivity, like inside the flight, basement, elevator etc. :), you will still be able to cOmmlt files, look at 1 evision hlstory, create branches etc. This may sound trivial for lot of people but, it is a big deal when you olten bump into no-network scenario. And also, the distributed mode of oper ation is a biggest blessing for open-source softwdre development community. lnstead of creating patches & sending it through emails, you can create a branch & send a pull request to the project team. It will help the code stay streamlined without getting lost in transport. GitHub.com is an awesome working example of that.There were some rumors that the future version of subversion will be working on distributed mode. But, it’s still dn unknown at this point.

Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit theii changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do com mit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

**Let as see the basic workflow of Git.**

**Step** 1:You modify a file from the working airectory.

**Step** 2:You add these files to the staging area.

**Step** 3:You perform commit operation that moves the files from the staging area. After pu operation, it stores the changes permanently to the Git repository.

BLOSB:

Blob stands forBinaryLargeObject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file and in Git database, it is named as SHAI hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub- directories. A tree is a binary file that stores references to blobs and trees which are also named asSHAlhash of the tree object.

Commits

Commit holds the current state of the repository. A commit is also named bySHAlhash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

Branches

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Su bversion. Usually, a branch is created to work on a new feature. Once the feature 's completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repositor '. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

Pull

Pull operation copie5 the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

Push

Push operation copies changes from a local repository instance to a remote one. ThÎS lS used to store the changes permanently into the Cuit repository. This is same as the commit operation in Subversion.

HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you maxe a commit, HEAD i5 updated with the latest commit. The heads of the branches are stored in.git/refs/heads/directory.

'[CentoS]$ ls -1 .git/refs/heads/ master

[CentoS]$ cat .git/refs/heads/master 570837e7d58fa4bccd8Scb575d884502188b0c49

Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by SHAl secure hashes.

Registering

You can register for a GitHub account at https://github.com/. However, it is strongly recommended that you get a free “student” account. A normal, free GitHub account does not allow you to create “private” repositories. Any code you push to GitH ub is automatically public and accessible by anyone. This is okay in general, however many of your courses will have Academic Integrity policies that will require you to not share code. A student account allows you up to 5 private repositories (normally $7/month as of this writing) so that you can comply with Academic Integrity policies.

To get a student account first register at GitHub using an email account that ends in

.edu (to “prove” you’re a student). Then go to https://education.github.com/pack and register for a “student pack.” Sign up early as some have reported long wait times to receive their student pack. The student pack contains a lot of other free and reduced cost software packages, tools and services that may be of interest.

Installing Git on Your Machine

If you want to use Git on your own personal machine, then you may need to lnstall a Git client. There are many options out there and you are encouraged to explore them, however the following suggestions are all free and open source.Git has released its own graphical user interface clients which are available for free for both Windows and Mac: - Windows: <https://windows.github.com/>

— Mac: <https://mac.github.com>

**See section 2 for instructions on using the client.**

If you will be using the Eclipses IDE (<http://www.eclipse.org/downloads/)> for development, the most recent versions already come with a Git client. Eclipse will work on any system. See Section 4 for using Git with Eclipse.If you use Windows and prefer to use a command line interface, you can clownload and install TortoiseGit https://code.goog1e.com/p/tortoisegit/) a Windows Shell Interface to Git. See Section 3 for using Git via the command line interface.

If you use Mac and want the command line version of Git, you can download and install here: [http://www.git-scm.com/download/mac.](http://www.git-scm.com/download/mac) Alternatively, you can install Git using a tool like MacPorts:

<http://iamphioxus.org/20> 3/04/20/installing-git-via-macports-on-mac-osx/. See Section 3 for using Git via the command line interface.

Creating a Repository on Github

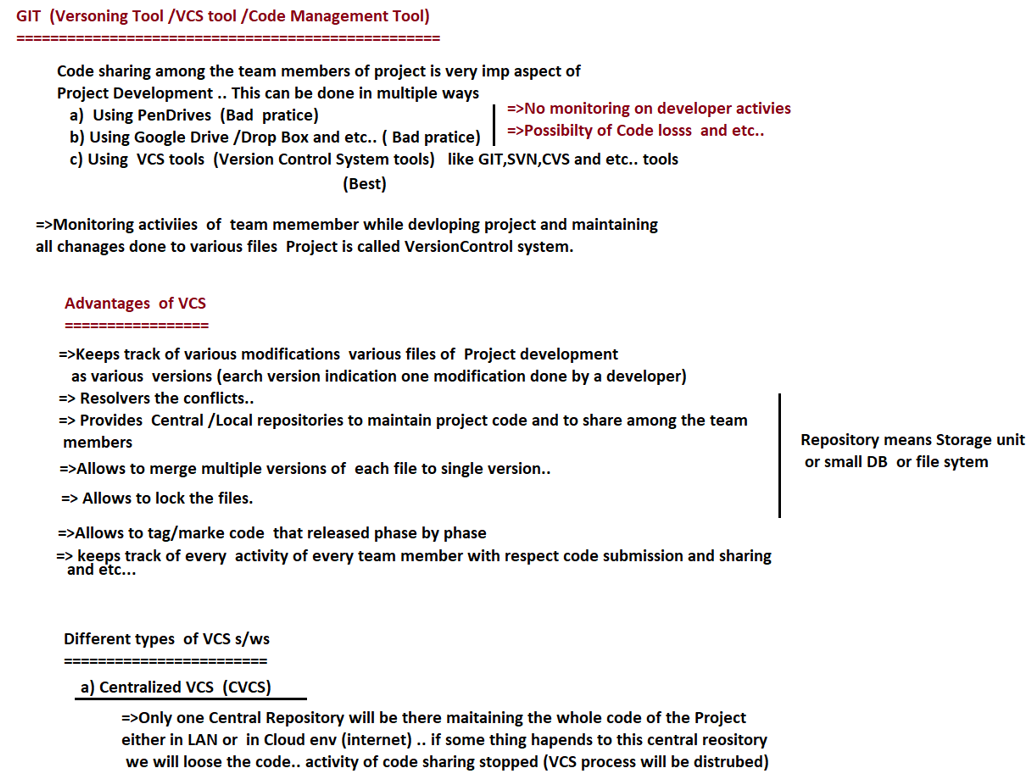
You will eventually want to publish (“push”) your project code to Github. To do this you’ll first need to create a repository on Github’s site:

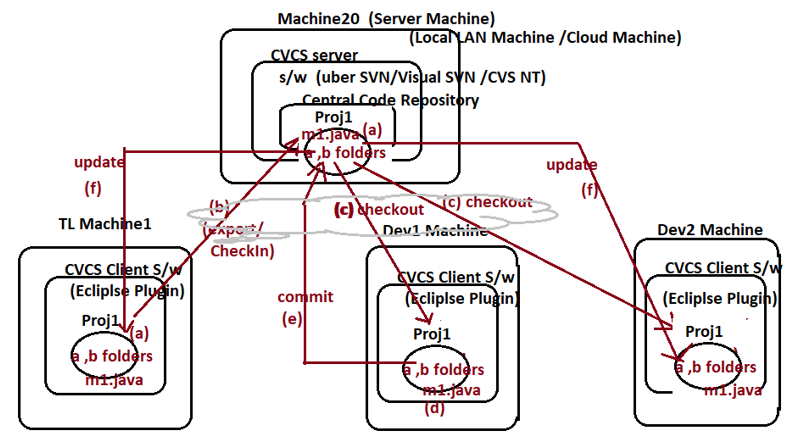
Login tO Github (https://github.com/) and click on the “repositories” tab.

Create a new repositoi y (see Figure 2) with the name that will match your project folder (the names do not have to match, but it keeps things ol ganized). Provide a short description and choose whether or not to make it public or private depending on whether ol not you are allowed to share your code with your peers. You may choose to include a README file arid/ or establish a license (which creates a LICENSE file). However, for this tutorial we will assume that you start with an empty repo on Github. If you choose to create these files some extra steps may be necessary.

Using Git via Git’s Clients

In this section we’ll explore the basic uses of Git by using Git’s rlient which provides a Graphical User Interface (GUI) to Git. A complete online help guide is available here: https://mac.github.com/help.ntml(Mac) and here: https://windows. github.com/help.html(Windows).Though the clients should be almost identical for Mac and Windows, there may be some slight differences;.





A close up of text

Description automatically generated

A diagram of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A black text on a white background

Description automatically generated

A diagram of a tree

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generatedA close-up of words

Description automatically generated

A white paper with black text

Description automatically generatedA black text on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generatedA white background with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

## Docker

A computer program written on a white background

Description automatically generatedA screenshot of a computer program

Description automatically generatedA diagram of software application

Description automatically generatedA close up of a text

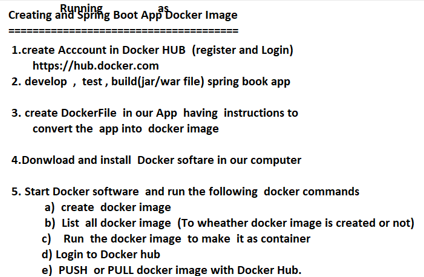
Description automatically generatedA group of black and white text

Description automatically generated with medium confidenceA white paper with black text

Description automatically generatedA close-up of a logo

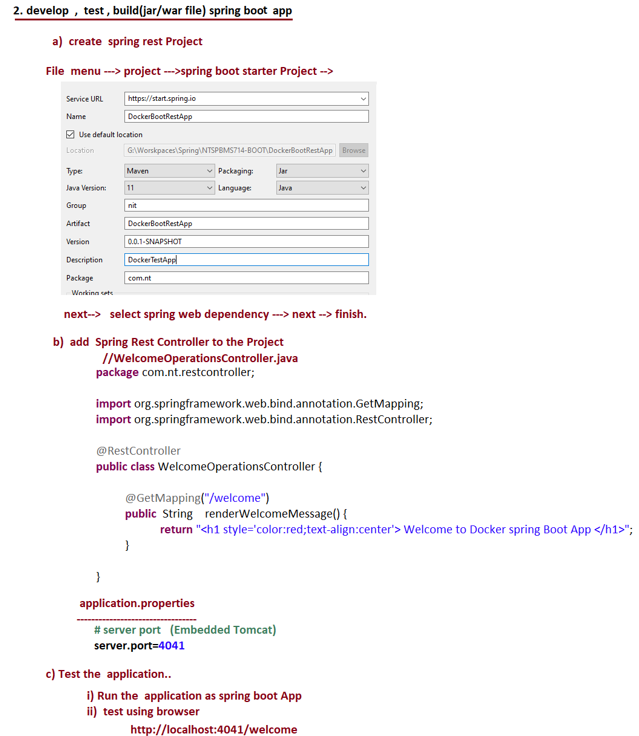
Description automatically generated

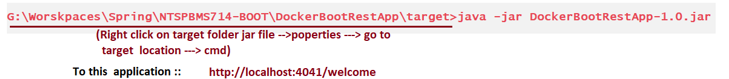
A diagram of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

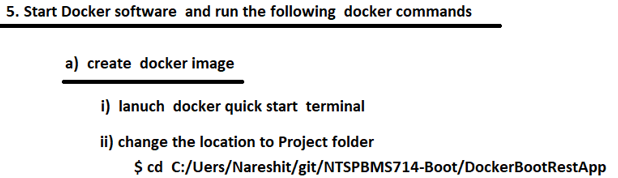
A close-up of a computer code

Description automatically generated

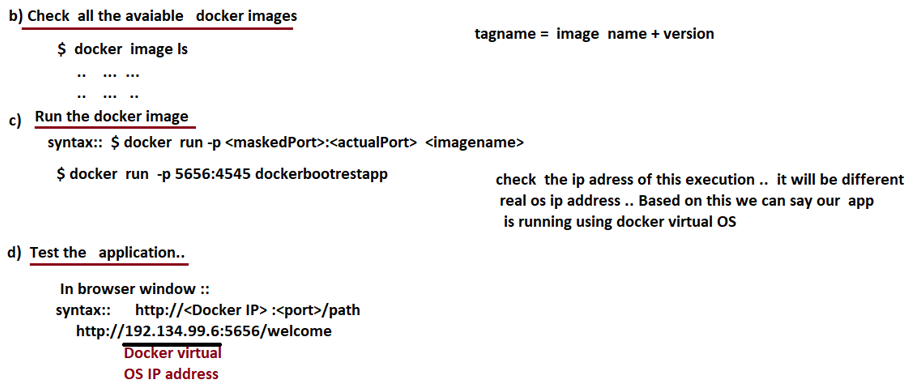
A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated A close up of a computer code

Description automatically generated



A screenshot of a computer

Description automatically generated

A black text on a white background

Description automatically generated

**Here are some basic Docker commands that you can use to manage Docker containers:**

**Managing Containers:**

1. docker run: This command is used to create and start a new container based on a Docker image.

* **docker run [OPTIONS] IMAGE [COMMAND] [ARG...]**

1. docker start: Start one or more stopped containers.

* **docker start [OPTIONS] CONTAINER [CONTAINER...]**

1. docker stop: Stop one or more running containers.

* **docker stop [OPTIONS] CONTAINER [CONTAINER...]**

1. docker restart: Restart one or more containers.

* **docker restart [OPTIONS] CONTAINER [CONTAINER...]**

1. docker ps: This command lists all running containers.

* **docker ps [OPTIONS]**

1. docker ps -a: This command lists all containers, including those that are stopped.

* **docker ps -a [OPTIONS]**

1. docker rm: This command is used to remove one or more containers.

* **docker rm [OPTIONS] CONTAINER [CONTAINER...]**

1. docker exec: This command is used to run a command inside a running container.

* docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

1. docker logs: This command is used to view the logs generated by a running container.

* **docker logs [OPTIONS] CONTAINER**

**Managing Images:**

1. docker images: List all images available locally.

* **docker images [OPTIONS] [REPOSITORY[:TAG]]**

1. docker pull: Pull an image or a repository from a registry.

* **docker pull [OPTIONS] NAME[:TAG|@DIGEST]**

1. docker build: Build an image from a Dockerfile.

* **docker build [OPTIONS] PATH | URL | -**

1. docker rmi: Remove one or more images.

* **docker rmi [OPTIONS] IMAGE [IMAGE...]**

1. docker tag: Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE.

* **docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]**

**Managing Networks:**

1. docker network ls: List all networks.

* **docker network ls [OPTIONS]**

1. docker network create: Create a network.

* **docker network create [OPTIONS] NETWORK**

1. docker network inspect: Display detailed information on one or more networks.

* **docker network inspect [OPTIONS] NETWORK [NETWORK...]**

**Managing Volumes:**

1. docker volume ls: List all volumes.

* **docker volume ls [OPTIONS]**

1. docker volume create: Create a volume.

* **docker volume create [OPTIONS] [VOLUME]**

1. docker volume inspect: Display detailed information on one or more volumes.

* **docker volume inspect [OPTIONS] VOLUME [VOLUME...]**

1. docker volume rm: Remove one or more volumes.

* **docker volume rm [OPTIONS] VOLUME [VOLUME...]**

**Managing Swarm (Docker Swarm mode):**

1. docker swarm init: Initialize a new Docker Swarm.

* **docker swarm init [OPTIONS]**

1. docker swarm join: Join a Docker Swarm as a node.

* **docker swarm join [OPTIONS] HOST:PORT**

1. docker service: Manage services in Docker Swarm mode.

* **docker service [OPTIONS] COMMAND [ARGS...]**

1. docker stack: Manage stacks in Docker Swarm mode.

* **docker stack [OPTIONS] COMMAND [ARGS...]**

**Managing Docker Compose:**

1. docker-compose up: Build and start containers as defined in the docker-compose.yml file.

* **docker-compose up [OPTIONS] [SERVICE...]**

1. docker-compose down: Stop and remove containers, networks, and volumes defined in the docker-compose.yml file.

* **docker-compose down [OPTIONS]**

These are some of the most commonly used Docker commands. You can use docker --help or docker COMMAND --help to get more information about each command and its options.

## Kubernetes

Kubernetes (often abbreviated as K8s) is a powerful container orchestration platform used for automating deployment, scaling, and management of containerized applications. Below are some commonly used Kubernetes commands:

1. **kubectl version**: Check the Kubernetes client and server version.
2. **kubectl cluster-info**: Display cluster details, such as the Kubernetes master and services running in the cluster.
3. **kubectl get**: Retrieve information about resources.
   * Example: **kubectl get pods**,
   * **kubectl get deployments**,
   * **kubectl get services**
4. **kubectl describe**: Display detailed information about a specific resource.
   * Example: **kubectl describe pod <pod\_name>**,
   * **kubectl describe service <service\_name>**
   * **kubectl describe deployment <deployment\_name>**
5. **kubectl create**: Create a resource from a file or from stdin.
   * Example: **kubectl create -f deployment.yaml**,
   * **kubectl create deployment <deployment\_name> --image=<image\_name>**
   * **kubectl create deployment nginx --image=nginx:latest**
6. **kubectl apply**: Apply changes to a resource.
   * Example: **kubectl apply -f deployment.yaml**,
   * **kubectl apply -f config-map.yaml**
7. **kubectl delete**: Delete resources.
   * Example: **kubectl delete pod <pod\_name>**, **kubectl delete deployment <deployment\_name>**
   * **kubectl delete pod <pod\_name>**
   * **kubectl delete deployment <deployment\_name>**
   * **kubectl delete service <service\_name>**
8. **kubectl exec**: Execute a command in a container.
   * Example: **kubectl exec -it <pod\_name> -- /bin/bash**
   * **kubectl exec -it <pod\_name> -- /bin/**
9. **kubectl logs**: Display logs from a container.
   * Example: **kubectl logs <pod\_name>**
10. **kubectl scale**: Scale the number of replicas of a deployment.
    * Example: **kubectl scale deployment <deployment\_name> --replicas=3**
11. **kubectl rollout**: Manage deployments and perform rollouts.
    * Example: **kubectl rollout status deployment/<deployment\_name>**, **kubectl rollout undo deployment/<deployment\_name>**
12. **kubectl port-forward**: Forward one or more local ports to a pod.
    * Example: **kubectl port-forward <pod\_name> local\_port:pod\_port**
13. **kubectl get events**: Display cluster events.
14. **kubectl config**: Manage Kubernetes configuration files.
    * Example: **kubectl config view**, **kubectl config current-context**
15. **kubectl proxy**: Run a proxy to the Kubernetes API server.

These are just a few examples of Kubernetes commands. There are many more commands available, and you can explore them further by checking the official Kubernetes documentation or using kubectl --help for a list of available commands and options. Additionally, understanding Kubernetes concepts like Pods, Deployments, Services, ConfigMaps, and Secrets is crucial for effectively using these commands.