

DevOps

Practical 1: Version Control System

Q1. What is a Version Control System? Define Git. How Git is used in Version Control System. Install Git.

A version control system is a software that tracks changes to a file or set of files over time so that you can recall specific versions later. It also allows you to work together with other programmers.

The version control system is a collection of software tools that help a team to manage changes in a source code. It uses a special kind of database to keep track of every modification to the code.

Developers can compare earlier versions of the code with an older version to fix the mistakes.

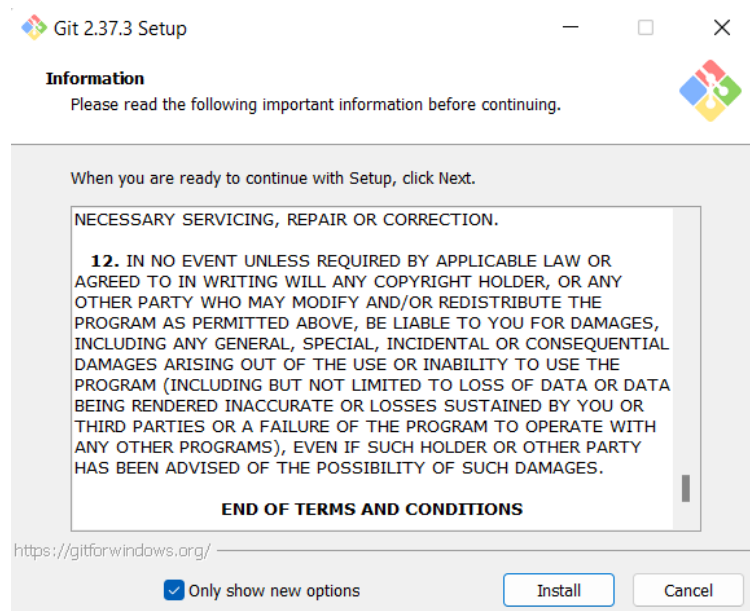
Git is an open-source distributed version control system. It is designed to handle minor to major projects with high speed and efficiency. It is developed to coordinate the work among the developers.

Git is the foundation of many services like GitHub and GitLab, but we can use Git without using any other Git services. Git can be used privately and publicly.

Git was created by Linus Torvalds in 2005 to develop the Linux Kernel. It is also used as an important distributed version-control tool for DevOps.

Steps to install Git:

1. Downloading installer from the official website



2. Follow the prompts during the installation. Default options will suffice for the majority of the task, however additional configurations can be made during the installation

3. Verify installation by the command line

```
C:\Users\prana>git --version
git version 2.37.3.windows.1
```

4. Configure git username and email which will be associated with the commits to be made.

Commands: git config – global user.name “Name Here”

Git config –global user.email “user@email.com”

```
C:\Users\prana>git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=pranavp778@gmail.com
user.name=Pranav Patel
core.editor="D:\Microsoft VS Code\bin\code" --wait
```

Q2. Make a list of various Git Commands. Explain the Git Flow with a diagram.

Git commands can be executed via the command line interfaces supported by it. There are various commands available in git, however, only a handful of them are used on a daily basis, while others are intended for special situations.

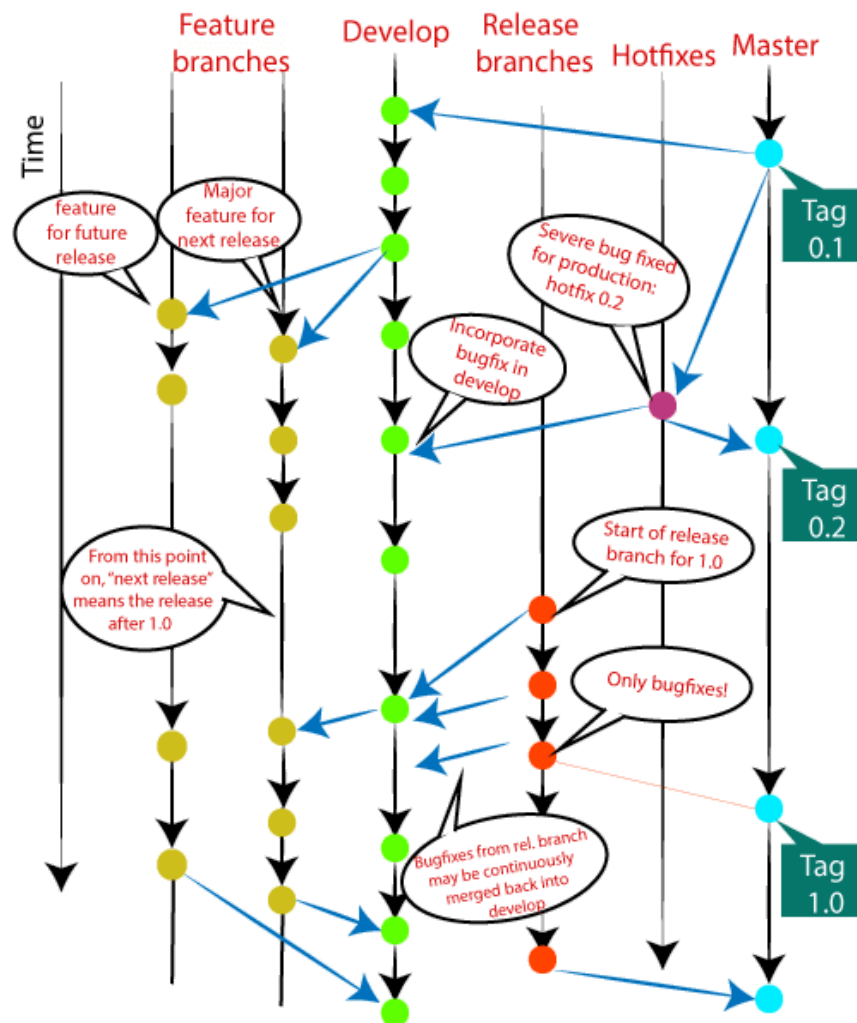
Here is a list of most essential Git commands that are used regularly:

- Git Config command
- Git init command
- Git clone command
- Git add command
- Git commit command
- Git status command
- Git push Command
- Git pull command
- Git Branch Command
- Git Merge Command
- Git log command
- Git remote command

Git Flow:

Git flow is the set of guidelines that developers can follow when using Git. They are standards for an ideal project.

It is referred to as Branching Model by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.



There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

- Master: Master and Develop are the main branches. Master contains the final changes or stable versions of the project. The contents of the master branch will be reflected in the final version.

They should not be messed with otherwise the changes would affect everyone else and merge conflicts may arise.

- Develop: Works parallel to master branch. Contains the development changes for next changes. Also called “Integration Branch”. When a stable version of the product is developed in this branch, we need to merge it to master branch, tag with release version for release.
- Hotfixes: Similar to Release branch. Created for instant fix of a version needed due to a critical bug in the production version. Can be merged with master after fix, and subsequent version tagging
- Release branches: Supports a new version release. Minor bug fixes and meta-data preparation is conducted. Merged with the develop branch after the features are created. Deployed to staging server for testing.
- Feature branches: Involved in development of new features. Limited existence and is deleted after features are merged with develop branch

Q3. Explain and Implement all the staging and commit commands in Git.

i. Git Init:

Initialise a blank repository. Creates a repository in the current directory and creates a .git subdirectory with metadata as well. One can create files in the repository after creation. Can also be used in a directory where the project exists to share it on a VCS.

```
MINGW64:/d/Projects/git/Demo-repo3
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (main)
$ git init
Initialized empty Git repository in D:/Projects/git/Demo-repo3/.git/
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ touch file1.txt
```

ii. Git Add

Adds files to staging area. Updates content of current working tree. Every update requires forwarding updates to the staging area. Adds one file but many can also be added via options.

Git add filename : Adds specified file to staging area

Git add -A : Adds all files to staging area

Git add . : Same as above

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ touch file2.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ touch file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ ls
file1.txt file2.txt file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git add file1.txt
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF
at time Git touches it

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file2.txt
    file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git add -A

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   file1.txt
    new file:   file2.txt
    new file:   file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
```

iii. Git Commit

The staging and committing are co-related to each other. Staging allows us to continue making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes.

Commits are the snapshots of the project. Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version. Two different commits will never be overwritten because each commit has its own commit-id. This commit-id is a cryptographic number created by SHA (Secure Hash Algorithm) algorithm.

Options:

Git commit: Commit changes and create commit-id. Opens a text editor for commit message

Git commit -a : Only commits files in the staging area.

Git commit -m : To add a message to the commit

Git commit -amend : Serves to edit wrongly entered commit message.

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git commit -m "First commit made"
[master (root-commit) 4d33fbf] First commit made
3 files changed, 3 insertions(+)
create mode 100644 file1.txt
create mode 100644 file2.txt
create mode 100644 file3.txt
```

iv. Git Clone:

To create a local copy of any target repository. The target repository will be termed as origin

One can download a repository from Github or BitBucket or select the “clone with HTTP” section and copy the link.

The same link will be provided as an argument to the git clone command.

Options:

Cloning into a specific directory,

Git clone -b <Branch name><Repository URL> : To create copy of specific branch of a repository.

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 31, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 31 (delta 6), reused 5 (delta 5), pack-reused 23
Receiving objects: 100% (31/31), 7.62 KiB | 600.00 KiB/s, done.
Resolving deltas: 100% (7/7), done.

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ ls
Git-Example/  file1.txt  file2.txt  file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ ls Git-Example
Demo  README.md  index.html  'new file'  newfile2
```

v. Git Stash

To store all data temporarily without committing. It stores the incomplete work that one may not want to commit, while switching branches.

It stores messy state of a branch and stores it for continuation for later

Options:

Git stash : To stash untracked files or unstaged files.

Git stash save “<Message>”: Save stash with a message.

Git stash list: List stored stashes

Git stash apply ; Apply changes back into repository to continue work. Can also specify specific stash id.

Git stash show : To track stash and changes.

Git stash show -p : Shows what changes are made on the file.

Git stash pop : To re-apply previous commit

Git stash drop : Deletes recent stash item. Can also specify stash id

Git stash clear : To empty the stash completely

Git stash branch : To stash work in a separate branch

```
MINGW64:/c/Gitproject

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   p4.txt

no changes added to commit (use "git add" and/or "git commit -a")

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git stash
Saved working directory and index state WIP on master: 276d3c1 4rth File

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git stash list
stash@{0}: WIP on master: 276d3c1 4rth File

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   p4.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (bb6d77dd97ae2f24942b2d1391a5839ed54e8450)

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git stash list
```

vi. Git Ignore

To specify certain files to be left untrack. Used when files aren't to be sent to GitHub.

Ignored files can be tracked on the .gitignore file. Steps:

1. Create a .gitignore file if not exists
2. Open the file and type filenames and directory names to ignore here.
3. Add the .gitignore file to the staging area and commit it.

List ignored files with : `git ls-files -i --exclude-standard`

```
prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ touch .gitignore

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ touch p5.txt

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ ls
README.md p1.txt p2.txt p3.txt p4.txt p5.txt

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ cat > .gitignore
p5.txt

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ cat .gitignore
p5.txt
```

```

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   p4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

no changes added to commit (use "git add" and/or "git commit -a")

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git add .
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        modified:   p4.txt

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)
$ git commit -m "Gitignore file created"
[master 2686a57] Gitignore file created
 2 files changed, 2 insertions(+)
 create mode 100644 .gitignore

prana@LAPTOP-1P08FSRM MINGW64 /c/Gitproject (master)

```

vii. Git Fork

A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project. One of the excessive use of forking is to propose changes for bug fixing. To resolve an issue for a bug that you found, you can:

- Fork the repository.
- Make the fix.
- Forward a pull request to the project owner.

Forking is not a Git function; it is a feature of Git service like GitHub.

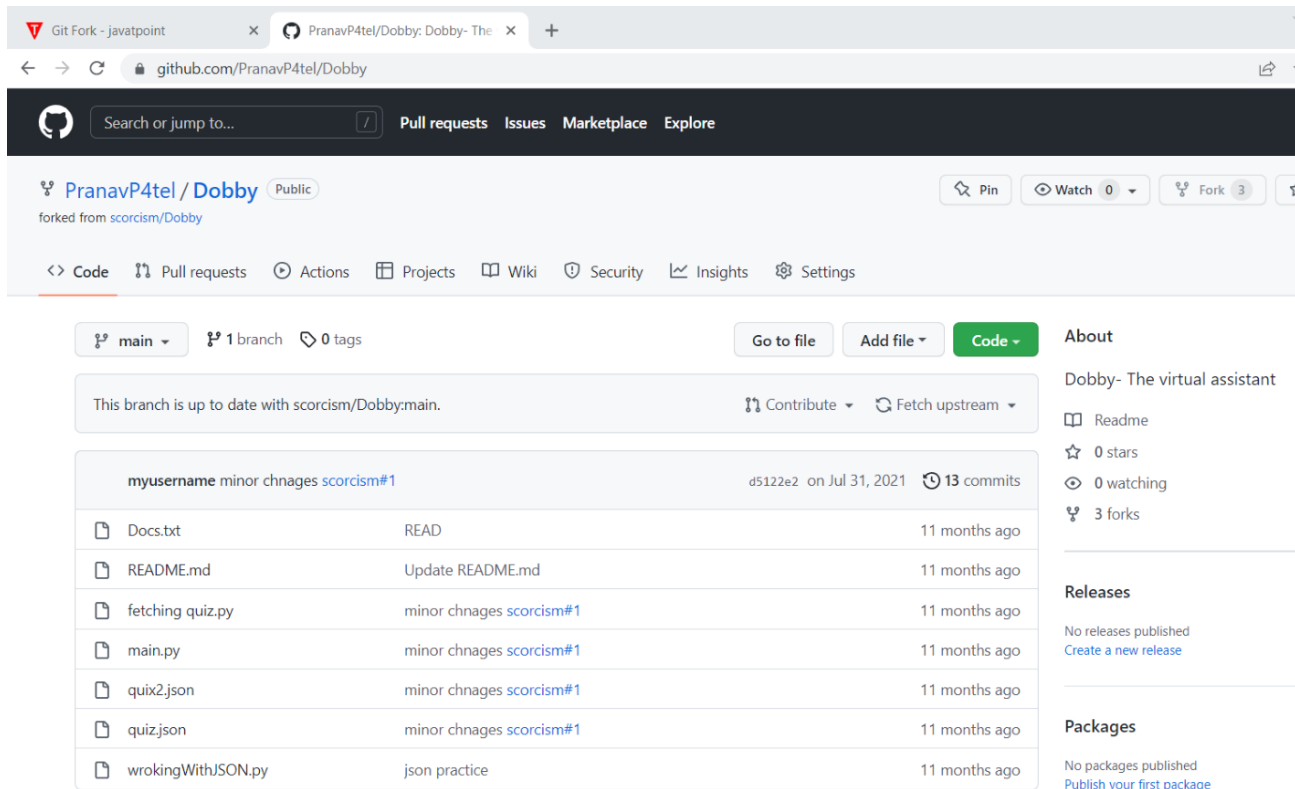
To fork a repository, simply go to it in GitHub and click on the Fork Options.

Check your GitHub repositories which will show the forked repository

The screenshot shows the GitHub interface for the repository 'scorcism/Dobby'. The repository is public and has 13 commits, 4 stars, 1 watch, and 2 forks. The file list includes:

File	Commit	Time
Docs.txt	READ	11 months ago
README.md	Update README.md	11 months ago
fetching quiz.py	minor chnages #1	11 months ago
main.py	minor chnages #1	11 months ago
quix2.json	minor chnages #1	11 months ago
quiz.json	minor chnages #1	11 months ago
wrokingWithJSON.py	json practice	11 months ago

The repository details on the right show 'Dobby- The virtual assistant' with 4 stars, 1 watch, and 2 forks. There are no releases published.



viii. Git Repository

In Git, the repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repository in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

One can create a repository as:

1. Git init command : Creates empty repository as explained above
2. Git clone : Creates a copy of remote repository on local machine as explained above

ix. Git Index

The Git index is a staging area between the working directory and repository. It is used to build up a set of changes that you want to commit together.

Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

The git status command shows the index file. Information about the files is present in the index file which includes:

1. Mtime : Time of last update
2. File : Name of file
3. Wdir : Version of file in working directory
4. Stage: Version of file in the index
5. Repo : Version of file in the repository

```
prana@LAPTOP-1P08FSRM MINGW64 /c/GitProject (master)
$ ls
README.md p1.txt p2.txt p3.txt p4.txt p5.txt p6.txt

prana@LAPTOP-1P08FSRM MINGW64 /c/GitProject (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        p6.txt

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


x. Git Head

The HEAD points out the last commit in the current checkout branch. It is like a pointer to any reference. The HEAD can be understood as the "current branch." When you switch branches with 'checkout,' the HEAD is transferred to the new branch.

Git show head : Used to check the status of the Head

Detached Head : When the head is not pointing to the last commit

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo3 (master)
$ git show head
commit 4d33fbf1a9daf61b2502513885edd7f2b5a70ccf (HEAD -> master)
Author: Pranav Patel <pranavp778@gmail.com>
Date:   Mon Oct 3 21:05:55 2022 +0530

    First commit made

diff --git a/file1.txt b/file1.txt
new file mode 100644
index 0000000..6a257ff
--- /dev/null
+++ b/file1.txt
@@ -0,0 +1,3 @@
+Hello There
+This is a git tutorial
+Plus Ultra!
diff --git a/file2.txt b/file2.txt
new file mode 100644
index 0000000..e69de29
diff --git a/file3.txt b/file3.txt
new file mode 100644
index 0000000..e69de29
```

xi. Git Origin Master

The term "git origin master" is used in the context of a remote repository. It is used to deal with the remote repository. The term origin comes from where the repository originally situated and master stands for the main branch. Let's understand both of these terms in detail.

Git Master is a naming convention for a Git branch. It's a default branch of Git. After cloning a project from a remote server, the resulting local repository contains only a single local branch. This branch is called a "master" branch. It means that "master" is a repository's "default" branch.

In Git, The term origin is referred to the remote repository where you want to publish your commits. The default remote repository is called origin, although you can work with several remotes having a different name at the same time. It is said as an alias of the system.

Some commands in which the term origin and master are widely used are as follows:

- Git push origin master
- Git pull origin master

Git has two types of branches called local and remote. To use git pull and git push, you have to tell your local branch, on which branch is going to operate. So, the term origin master is used to deal with a remote repository and master branch. The term push origin master is used to push the changes to the remote repository. The term pull origin master is used to access the repository from remote to local.

xii. Git Remote

In Git, the term remote is concerned with the remote repository. It is a shared repository that all team members use to exchange their changes. A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion, and more. In the case of a local repository, a remote typically does

not provide a file tree of the project's current state; as an alternative, it only consists of the .git versioning data.

The developers can perform many operations with the remote server. These operations can be a clone, fetch, push, pull, and more.

To check the configuration of the remote server, run the git remote command. The git remote command allows accessing the connection between remote and local. If you want to see the original existence of your cloned repository, use the git remote command. It can be used as:

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote
origin
```

Git remote supports a specific option -v to show the URLs that Git has stored as a short name. These short names are used during the reading and write operation. Here, -v stands for verbose. We can use --verbose in place of -v. It is used as:

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote -v
origin https://github.com/PranavP4tel/Demo-repo2 (fetch)
origin https://github.com/PranavP4tel/Demo-repo2 (push)
```

One can also use git remote add <short name><url name> : To add a remote for a repository

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote add demorepo2 https://github.com/PranavP4tel/Demo-repo.git

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote
demorepo2
origin
```

Git remote rm <destination> or git remote remove <destination> : To remove a repository

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote rm demorepo2

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote
origin
```

Git remote rename <old name> <new name> : To rename a remote

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote rename demorepo2 demorepo
```

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git remote
demorepo
origin
```

There is also the: Git remote show <remote> : To show information about the remote server

Q4. Explain and Implement all the commands for Undoing changes in Git.

i. Git Checkout

In Git, the term checkout is used for the act of switching between different versions of a target entity. The git checkout command is used to switch between branches in a repository. Be careful with your staged files and commits when switching between branches.

The git checkout command operates upon three different entities which are files, commits, and branches. Sometimes this command can be dangerous because there is no undo option available on this command.

It checks the branches and updates the files in the working directory to match the version already available in that branch, and it forwards the updates to Git to save all new commit in that branch.

Check all branches : git branch

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git branch
  master
* quick-test
```

Checkout branches : git checkout branchname

Create and switch branch : git checkout -b <branchname>

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (quick-test)
$ git checkout master
Switched to branch 'master'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (master)
$ ls
README.md  newfile.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (master)
$ git checkout -b newbranch
Switched to a new branch 'newbranch'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch
  master
* newbranch
  quick-test
```

ii. Git Revert

In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a commit.

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Moreover, we can say that git revert records some new changes that are just opposite to previously made commits.

To revert to old commit : git revert commit-id

To revert to old commit and change commit message : git revert -r commit-id

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ touch newfile.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ touch f1.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ ls
README.md  f1.txt  newfile.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git add .

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git commit -m "new commimt"
[newbranch b46cfaf] new commimt
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 f1.txt
create mode 100644 newfile.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git log
commit b46cfaff9b01294227a3fd53699a99f2b95f39 (HEAD -> newbranch)
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Oct 15 12:22:06 2022 +0530

    new commimt

commit 1e51b59370a728a073a9f02ef49bb1a978079690 (origin/feature-readme
ons, quick-test, master)
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Jan 29 17:13:28 2022 +0530

    updated readme with more instructions

commit 6fa23e399afd201a9a32a1fab9d1eabfbdf2e4e5
Author: Pranav Patel <pranavp778@gmail.com>
Date: Fri Jan 28 18:07:25 2022 +0530

    Created README

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ touch file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git add .

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git commit -m "Added file3"
[newbranch e9f3165] Added file3
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git revert b46cfaff9b01294227a3fd53699a99f2b95f39
hint: Waiting for your editor to close the file...

Run with 'code -' to read output from another program (e.g. 'echo Hello
code -').
[newbranch 2edeb3a] Revert "new commimt"
2 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 f1.txt
delete mode 100644 newfile.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ ls
README.md  file3.txt

```

The term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.

- Soft
- Mixed
- Hard

If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state. It is a sophisticated and versatile tool for undoing changes. It acts as a time machine for Git. You can jump up and forth between the various commits. Each of these reset variations affects specific trees that git uses to handle your file in its content.

Additionally, git reset can operate on whole commits objects or at an individual file level. Each of these reset variations affects specific trees that git uses to handle your file and its contents.

Git Reset Hard :

It will first move the Head and update the index with the contents of the commits. It is the most direct, unsafe, and frequently used option. The --hard option changes the Commit History, and ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory need to reset to match that of the specified commit. Any previously pending commits to the Staging Index and the Working Directory gets reset to match Commit Tree. It means any awaiting work will be lost.

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ touch fil4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git add fil4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   fil4
```

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ ls
README.md  fil4  file3.txt

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git log
commit 2edeb3a7dd70db274fdc10dc2e7c86551693dd30 (HEAD -> newbranch)
Author: Pranav Patel <pranavp778@gmail.com>
Date:   Sat Oct 15 12:25:29 2022 +0530

    Revert "new commimt"

    This reverts commit b46cfaff9b01294227a3fd53699a99f2b95f39.
```

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git reset --hard
HEAD is now at 2edeb3a Revert "new commimt"

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
nothing to commit, working tree clean
```

Git reset mixed :

A mixed option is a default option of the git reset command. If we would not pass any argument, then the git reset command considered as --mixed as default option. A mixed option updates the ref pointers. The staging area also reset to the state of a specified commit. The undone changes transferred to the working directory. Let's understand it with an example.

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ touch file4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git add .

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git reset --mixed

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file4

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

```

Generally, the reset mixed mode performs the below operations:

- It will move the HEAD pointer
- It will update the Staging Area with the content that the HEAD is pointing to.

It will not update the working directory as git hard mode does. It will only reset the index but not the working tree, then it generates the report of the files which have not been updated.

Git reset Head (Soft):

The soft option does not touch the index file or working tree at all, but it resets the Head as all options do. When the soft mode runs, the refs pointers updated, and the resets stop there. It will act as git amend command. It is not an authoritative command. Sometimes developers considered it as a waste of time.

iv. Git Rm

In Git, the term rm stands for remove. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.

The files being removed must be ideal for the branch to remove. No updates to their contents can be staged in the index. Otherwise, the removing process can be complex, and sometimes it will not happen. But it can be done forcefully by -f option.

To remove files from the working tree and the index : git rm <filename>

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git rm file4
rm 'file4'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    file4

```

The git rm is operated only on the current branch. The removing process is only applied to the working directory and staging index trees. It is not persisted in the repository history until a new commit is created.

Q4. Explain and Implement all the commands for Inspecting changes in Git

i. Git Log

The advantage of a version control system is that it records changes. These records allow us to retrieve the data like commits, figuring out bugs, updates. But, all of this history will be useless if we cannot navigate it. At this point, we need the git log command.

Git log is a utility tool to review and read a history of everything that happens to a repository. Multiple options can be used with a git log to make history more specific.

Generally, the git log is a record of commits. A git log contains the following data:

- A commit hash, which is a 40 character checksum data generated by SHA (Secure Hash Algorithm) algorithm. It is a unique number.
- Commit Author metadata: The information of authors such as author name and email.
- Commit Date metadata: It's a date timestamp for the time of the commit.
- Commit title/message: It is the overview of the commit given in the commit message.

To check the history : git log

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git log
commit 2fb288d1415762e43bff61e5e96ea5a08e62bf04 (HEAD -> newbranch)
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Oct 15 12:37:37 2022 +0530

    Added file4

commit 2edeb3a7dd70db274fdc10dc2e7c86551693dd30
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Oct 15 12:25:29 2022 +0530

    Revert "new commimt"

    This reverts commit b46cfa9f9b01294227a3fd53699a99f2b95f39.

commit e9f31652e2ee6f25007f762561e9c1f6ebe2b249
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Oct 15 12:22:42 2022 +0530

    Added file3
```

To check one commit per line : git log --oneline

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git log --oneline
2fb288d (HEAD -> newbranch) Added file4
2edeb3a Revert "new commimt"
e9f3165 Added file3
b46cfaf new commimt
1e51b59 (origin/feature-readme-instructions, quick-test, master) update
6fa23e3 Created README
```

To check modified files and summary line of records changed : git log --stat

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git log --stat
commit 2fb288d1415762e43bff61e5e96ea5a08e62bf04 (HEAD -> newbranch)
Author: Pranav Patel <pranavp778@gmail.com>
Date: Sat Oct 15 12:37:37 2022 +0530

    Added file4

file4 | 0
1 file changed, 0 insertions(+), 0 deletions(-)
```


ii. Git Diff

Git diff is a command-line utility. It's a multi use Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits, and more. It is used to show changes between commits, commit, and working tree, etc.

It compares the different versions of data sources. The version control system stands for working with a modified version of files. So, the diff command is a useful tool for working with Git.

However, we can also track the changes with the help of git log command with option -p. The git log command will also work as a git diff command.

To track changes that have not been staged : git diff

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ cat > file5
Hello World
Learning diff commands

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git diff
warning: in the working copy of 'file5', LF will be replaced by CRLF the
me Git touches it
diff --git a/file5 b/file5
index e69de29..4511e5e 100644
--- a/file5
+++ b/file5
@@ -0,0 +1,2 @@
+Hello World
+Learning diff commands
```

To track the changes stages but not committed : git diff --staged

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git add .
warning: in the working copy of 'file5', LF will be replaced by CRLF the
me Git touches it

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git diff --staged
diff --git a/file4 b/file4
deleted file mode 100644
index e69de29..0000000
diff --git a/file5 b/file5
new file mode 100644
index 0000000..4511e5e
--- /dev/null
+++ b/file5
@@ -0,0 +1,2 @@
+Hello World
+Learning diff commands
```

Track changes after committing a file : git diff HEAD

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git commit -m "Edited file5"
[newbranch 286769c] Edited file5
2 files changed, 2 insertions(+)
delete mode 100644 file4
create mode 100644 file5

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git diff HEAD

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
nothing to commit, working tree clean

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git diff HEAD

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ cat >> file5
Hello again

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git diff HEAD
warning: in the working copy of 'file5', LF will be replaced by CRLF t
me Git touches it
diff --git a/file5 b/file5
index 4511e5e..24cf821 100644
--- a/file5
+++ b/file5
@@ -1,2 +1,3 @@
Hello world
Learning diff commands
+Hello again

```

Also allows one to track changes between commits and branches

iii. 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. This command will not show any commit records or information.

Mostly, it is used to display the state between Git Add and Git commit command. We can check whether the changes and files are tracked or not.

Can be used when:

1. Working tree is clean
2. Status when file is created
3. Status when file is edited
4. Status when file is deleted

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git status
On branch newbranch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file5

no changes added to commit (use "git add" and/or "git commit -a")

```

Q5. Explain and Implement all the Branching & Merging commands in Git.

i. Git Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.

Operations on branches :

- Create : `git branch <name>`
- List : `git branch`
- Rename : `git branch -m <old name> <new name>`
- Delete: `git branch -d <name>`

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch newbranch2

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch
master
* newbranch
newbranch2
quick-test

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch -m newbranch2 newbranch3

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch
master
* newbranch
newbranch3
quick-test

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch -d newbranch3
Deleted branch newbranch3 (was 286769c).

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git branch
master
* newbranch
quick-test
```

ii. Merge & Merge Conflict

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The `git merge` command facilitates you to take the data created by `git branch` and integrate them into a single branch. `Git merge` will associate a series of commits into one unified history. Generally, `git merge` is used to combine two branches.

`Git merge` : `git merge <query>`

Can be used to:

1. Merge specific commit to current branch : `git merge <commit>`
2. Merge commit in master branch : `git merge master`
3. Merge branches

```

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git checkout newbranch2
Switched to branch 'newbranch2'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ ls
README.md  file3.txt  file5

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ touch file1 file4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ git add .

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ git commit -m "Added 2 files"
[newbranch2 df42563] Added 2 files
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file1
create mode 100644 file4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ git checkout newbranch
Switched to branch 'newbranch'

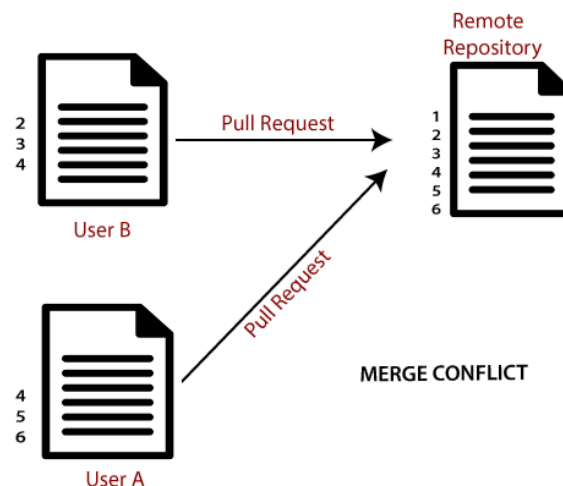
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git merge newbranch2
Merge made by the 'ort' strategy.
file1 | 0
file4 | 0
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file1
create mode 100644 file4

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)

```

Merge conflict :

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



To resolve the conflict, one can use the tool as : `git mergetool`

iii. Git Rebase

Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.

In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.

To rebase a branch: `git rebase branch-name`

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch)
$ git checkout newbranch2
Switched to branch 'newbranch2'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ git rebase newbranch
Successfully rebased and updated refs/heads/newbranch2.
```

Can also be used in interactive mode with: `git rebase -i`

Rebasing is not recommended in a shared branch because the rebasing process will create inconsistent repositories. For individuals, rebasing can be more useful than merging. If you want to see the complete history, you should use the merge. Merge tracks the entire history of commits, while rebase rewrites a new one.

Git rebase commands said as an alternative of git merge.

Q6 .Explain and Implement all the Collaborating commands in Git.

i.Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.

To pull updates from remote tracking branches : `git fetch <branch name><url>`

Branch name is optional

To fetch all branches : `git fetch -all`

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

Thus `git pull = git fetch + git merge`

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (newbranch2)
$ git checkout master
Switched to branch 'master'

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo2 (master)
$ git fetch https://github.com/PranavP4tel/Demo-repo.git
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 10 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), 1.50 KiB | 2.00 KiB/s, done.
From https://github.com/PranavP4tel/Demo-repo
* branch                HEAD      -> FETCH_HEAD
```

ii.Git Pull

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The git pull command is used to pull a repository.

Pull request is a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.

The pull command is used to access the changes (commits) from a remote repository to the local repository. It updates the local branches with the remote-tracking branches. Remote tracking branches are branches that have been set up to push and pull from the remote repository. Generally, it is a collection of the fetch and merges command. First, it fetches the changes from remote and combined them with the local repository.

Git pull <option> [<repository url><refspec>]

<options> : Can be quiet -q, verbose -v, and more

<repository url> : url of the repository

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ git pull https://github.com/PranavP4tel/Demo-repo.git
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 10 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), 1.50 KiB | 2.00 KiB/s, done.
From https://github.com/PranavP4tel/Demo-repo
* branch                HEAD      -> FETCH_HEAD
```

There is also the force pull used to overwrite the files, if one updates it on local machine but team members have updated on remote too.

One can also use git pull origin master to pull the master branch of remote repository

Pull Request:

Pull request allows you to announce a change made by you in the branch. Once a pull request is opened, you are allowed to converse and review the changes made by others. It allows reviewing commits before merging into the main branch.

Pull request is created when you committed a change in the GitHub project, and you want it to be reviewed by other members. You can commit the changes into a new branch or an existing branch.

Once you've created a pull request, you can push commits from your branch to add them to your existing pull request.

iii. Git Push

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.

Moreover, we can say the push updates the remote refs with local refs. Every time you push into the repository, it is updated with some interesting changes that you made. If we do not specify the location of a repository, then it will push to default location at origin master.

The "git push" command is used to push into the repository. The push command can be considered as a tool to transfer commits between local and remote repositories. The basic syntax is given below:

Git push <option> [<remote url><branch name><refspec>]

Let's try : git push origin master : to push the changes to remote repository

The screenshot shows a GitHub repository page for 'Demo-repo' (Private). The repository has 1 branch (main) and 0 tags. The commit history shows a commit by 'PranavP4tel' titled 'Added the new file on main branch' with commit hash 'fc557c4' on Jan 28, containing 3 commits. The commit details show two files: 'README.md' and 'index.html', both added on Jan 28, 9 months ago. The 'README.md' content is visible, showing a header '#Some header' and some lines of text. The right sidebar shows the repository's statistics: 0 stars, 1 watch, and 0 forks. The 'Releases' section shows 'No releases' and a link to 'Create a new release'.

```
prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ ls
README.md  index.html

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ cat index.html
<h1>Hello World</h1>

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ cat > demo.html
<!DOCTYPE html>
<html>
<head><h1>Hello world</h1></head>
<body>
<p> Hello Again</p>
</body>
</html>

prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ git add .
warning: in the working copy of 'demo.html', LF will be replaced by
t time Git touches it


prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ git commit -m "Added a demo html file for push"
[master bec900d] Added a demo html file for push
1 file changed, 7 insertions(+)
create mode 100644 demo.html
```







```



prana@LAPTOP-1P08FSRM MINGW64 /d/Projects/git/Demo-repo4 (master)
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 419 bytes | 419.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/PranavP4tel/Demo-repo/pull/new/master
remote:
To https://github.com/PranavP4tel/Demo-repo.git
 * [new branch]      master -> master




```

 master had recent pushes 1 minute ago
 Compare & pull request

 master ▾
  2 branches
  0 tags
 Go to file
Add file ▾
Code ▾

This branch is [1 commit ahead](#) of main.
  Contribute ▾

 **PranavP4tel** Added a demo html file for push
 bec900d 2 minutes ago
 4 commits

 README.md	Added the new file on main branch	9 months ago
 demo.html	Added a demo html file for push	2 minutes ago
 index.html	Added the new file on main branch	9 months ago

One can also force push by using : `git push <remote> <branch> -f`

To delete a remote branch using git push: `git push origin -delete <branch name>`

DevOps

Practical 2: Continuous Integration and Deployment

Q1. Install and Setup Java and Tomcat Setup for Jenkins.

Since Jenkins is written in Java, hence one needs to install Java on the system.

1. Java:

- 1) Install Java from Oracle website. Can be Java 8 or 11. Follow the steps to setup Java.
- 2) Check version of java in command line.

```
C:\Users\prana>java --version
java 11.0.15.1 2022-04-22 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.15.1+2-LTS-10)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.15.1+2-LTS-10, mixed mode)
```

2. Tomcat:

- a. Download the Tomcat binary distribution zip file and unzip the contents
- b. Copy Jenkins.war file into the webapps folder of tomcat directory
- c. Open terminal and traverse to bin folder in the tomcat folder. type
Startup.bat
- d. Allow Tomcat in the security permissions.
- e. Open browser and type <http://localhost:8080/jenkins>

Q2. Integrate GitHub with Jenkins by fetching the source code from GitHub and build it using Jenkins.

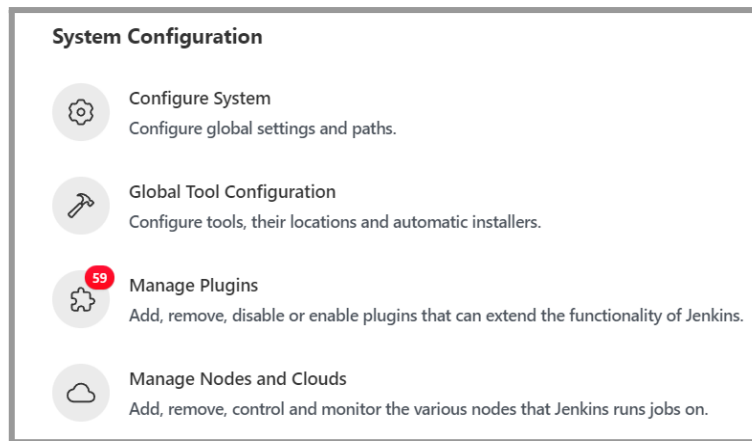
Jenkins is a CI (Continuous Integration) server and this means that it needs to check out source code from a source code repository and build code. Jenkins has outstanding support for various source code management systems like Subversion, CVS etc.

Github is the fast becoming one of the most popular source code management systems. It is a web based repository of code which plays a major role in DevOps. GitHub provides a common platform for many developers working on the same code or project to upload and retrieve updated code, thereby facilitating continuous integration. Jenkins works with Git through the Git plugin.

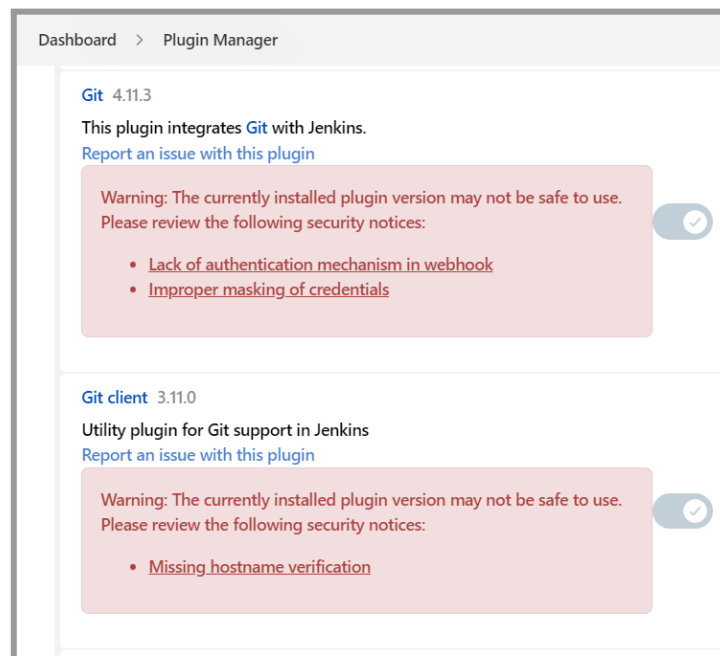
Connecting a GitHub private repository to a private instance of Jenkins can be tricky. To do the GitHub setup, make sure that internet connectivity is present in the machine where Jenkins is installed.

Steps:

1. Select Manage Jenkins in the menu in the left
2. Click on Manage Plugins



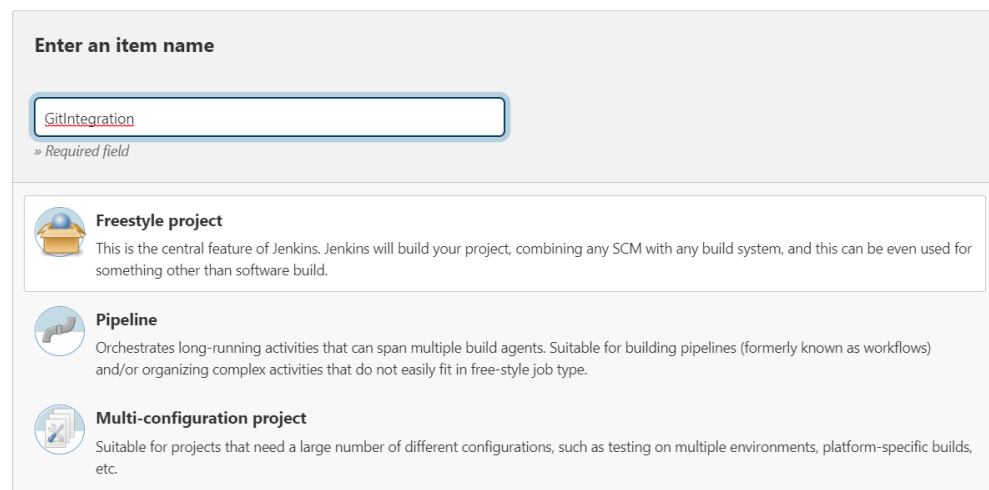
3. Select the Available option, and select Git Plugin and install without restart.
4. In the installed Tab, the git plugin will be installed



5. Restart / Relaunch Jenkins.

To fetch source code and build, the steps are:

1. Create a new job as FreeStyle Project



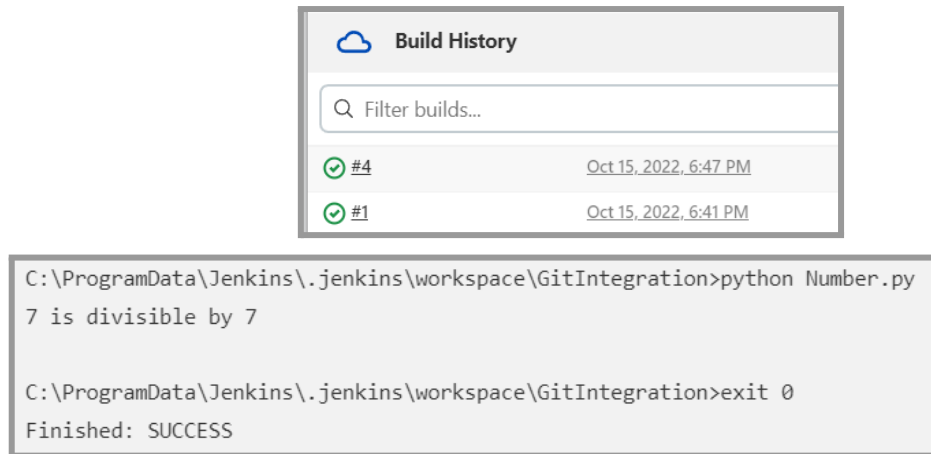
2. Enter project information
3. Under Source Code Management, Select Git option.
4. Enter repository URL and add the credentials. Also select branch

The screenshot shows the 'Source Code Management' configuration panel. At the top, there are two radio buttons: 'None' and 'Git'. The 'Git' option is selected. Below this, there is a section for 'Repositories'. Inside this section, there is a 'Repository URL' field containing 'https://github.com/PranavP4tel/Git-jenkins.git', a 'Credentials' field containing 'PranavP4tel/*****', and an 'Add' button. Below the 'Add' button is an 'Advanced...' button. At the bottom of the 'Repositories' section is an 'Add Repository' button. Below the 'Repositories' section is a 'Branches to build' section. It contains a 'Branch Specifier (blank for \'any\')' field with the value '*/firstbranch' and an 'Add Branch' button. At the bottom of the panel is a 'Repository browser' field with the value '(Auto)'.

5. Add any extra build step

The screenshot shows the 'Build' configuration panel. It contains a section for 'Execute Windows batch command'. Below this section is a 'Command' field with the text 'python Number.py'. There is a link 'See the list of available environment variables' above the command field. The panel also features a red 'X' icon in the top right corner.

6. Execute project and check everything is working.



Q3. Install and set up Maven for build automation.

Maven is a powerful project management and comprehension tool that provides complete build life cycle framework to assist developers. It is based on the concept of a POM (Project Object Model) that includes project information and configuration information for Maven such as construction directory, source directory, test source directory, dependency, Goals, plugins etc.

Maven is build automation tool used basically for Java projects, though it can also be used to build and manage projects written in C#, Scala, Ruby, and other languages. Maven addresses two aspects of building software: 1st it describes how software is build and 2nd it describes its dependencies.

Installing Maven:

1. Go to apache's official website and download maven zip file.
2. Extract the maven zip.
3. Setup Java Home and Maven Home
4. Check installation by the following

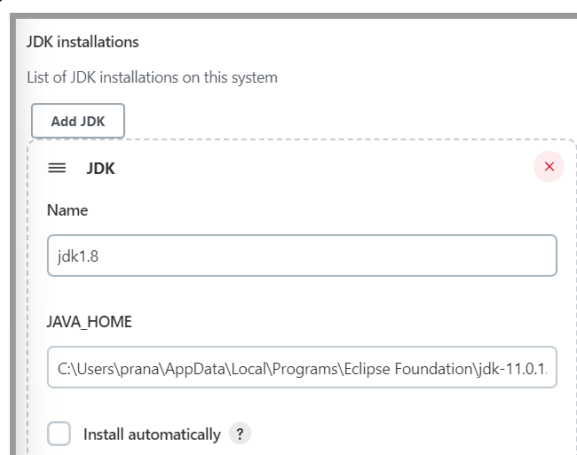
```

C:\Users\prana>mvn -version
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: C:\Program Files\apache-maven-3.8.6
Java version: 11.0.12, vendor: Eclipse Foundation, runtime: C:\Users\prana\AppData\Local\Programs\Eclipse Foundation\jdk-11.0.12-hotspot
Default locale: en_IN, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

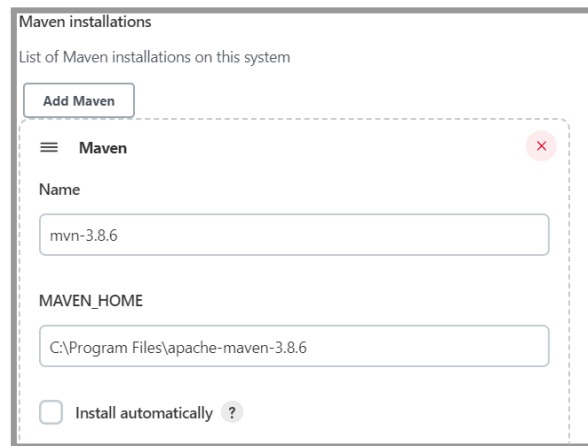
```

Install Maven in Jenkins:

1. Click on Manage Jenkins
2. Select Global tool configuration
3. Add a JDK by adding the Name and the Path or select install automatically.



4. Select Add Maven option and follow the same as above



Maven installations

List of Maven installations on this system

Add Maven

Maven

Name

mvn-3.8.6

MAVEN_HOME

C:\Program Files\apache-maven-3.8.6

☐ Install automatically ?

5. Save the changes. Now we can create a job with Maven Project.

Q4. Install and setup Jenkins.

Jenkins is an open source automation tool written in Java programming language that allows continuous integration.

Jenkins builds and tests our software projects which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

It also allows us to continuously deliver our software by integrating with a large number of testing and deployment technologies. Jenkins offers a straightforward way to set up a continuous integration or continuous delivery environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks.

With the help of Jenkins, organizations can speed up the software development process through automation. Jenkins adds development life-cycle processes of all kinds, including build, document, test, package, stage, deploy static analysis and much more.

Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

For example: If any organization is developing a project, then Jenkins will continuously test your project builds and show you the errors in early stages of your development.

Possible steps executed by Jenkins are for example:

- Perform a software build using a build system like Gradle or Maven Apache
- Execute a shell script
- Archive a build result
- Running software tests

Installing Jenkins:

Requirements :

- Only requires a JDK or a JRE.

- Space : Atleast 1 GB
- Ram : Atleast 2GB

Steps :

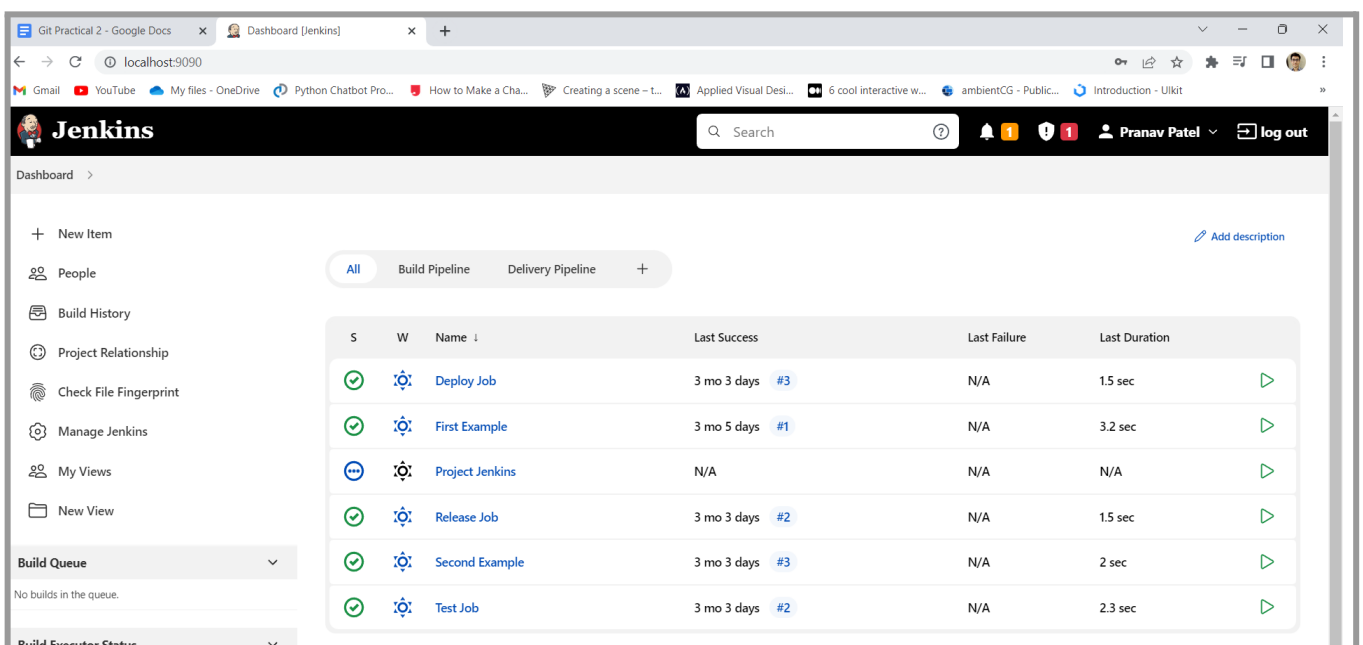
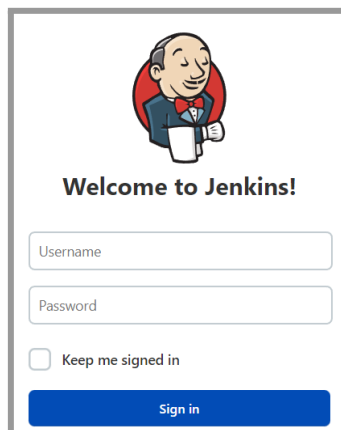
1. Install Java

- Install Java from Oracle website. Can be Java 8 or 11. Follow the steps to setup Java.
- Check version of java in command line.

```
C:\Users\prana>java --version
java 11.0.15.1 2022-04-22 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.15.1+2-LTS-10)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.15.1+2-LTS-10, mixed mode)
```

2. Install Jenkins

- Download the war file from it's official website
- Start the installer. Select the default location
- Setup the username and password
- Select port number
- Select directory where java was installed
- Select features to install and finish installation
- Jenkins is now downloaded. Access it via <http://localhost:9090>



Q5. Create Jenkins Continuous integration/ Continuous deployment pipeline, Set up the build jobs in the order of code ,build ,test ,deploy .

In Jenkins, a pipeline is a collection of events or jobs which are interlinked with one another in a sequence.

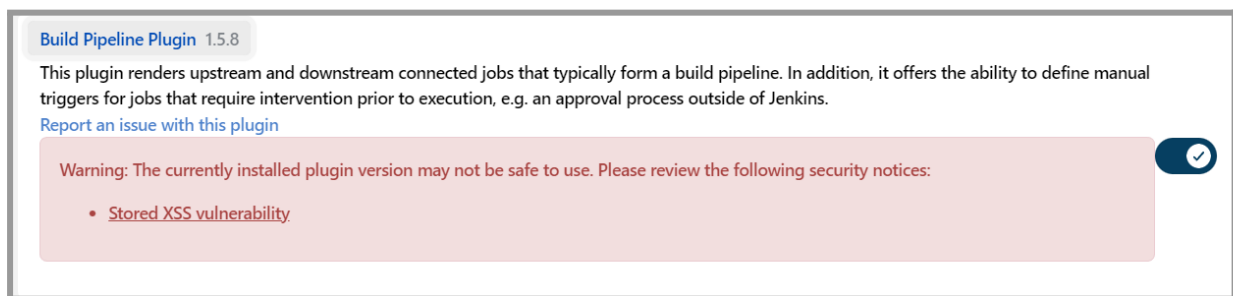
It is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins.

In other words, a Jenkins Pipeline is a collection of jobs or events that brings the software from version control into the hands of the end users by using automation tools. It is used to incorporate continuous delivery in our software development workflow.

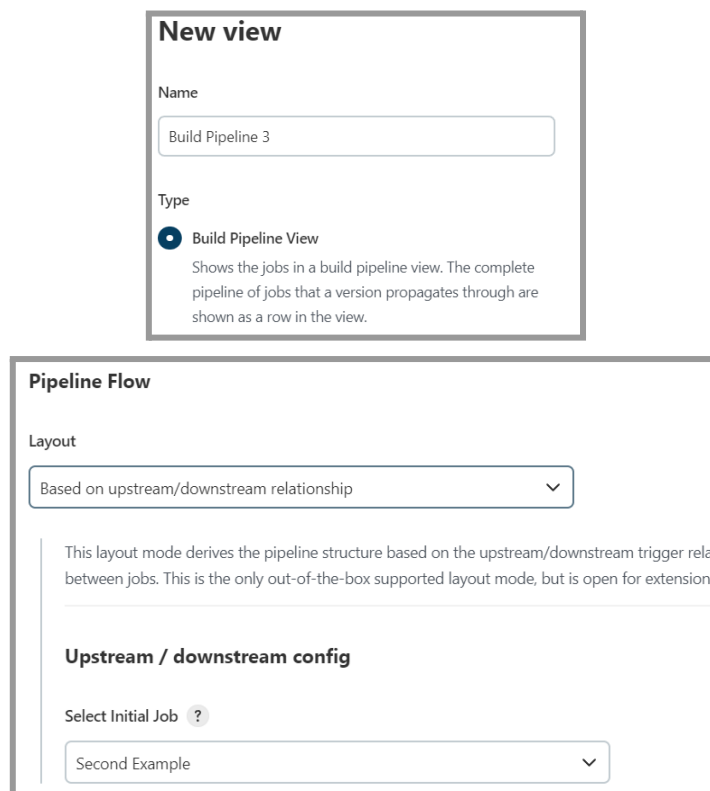
A pipeline has an extensible automation server for creating simple or even complex delivery pipelines "as code", via DSL (Domain-specific language).

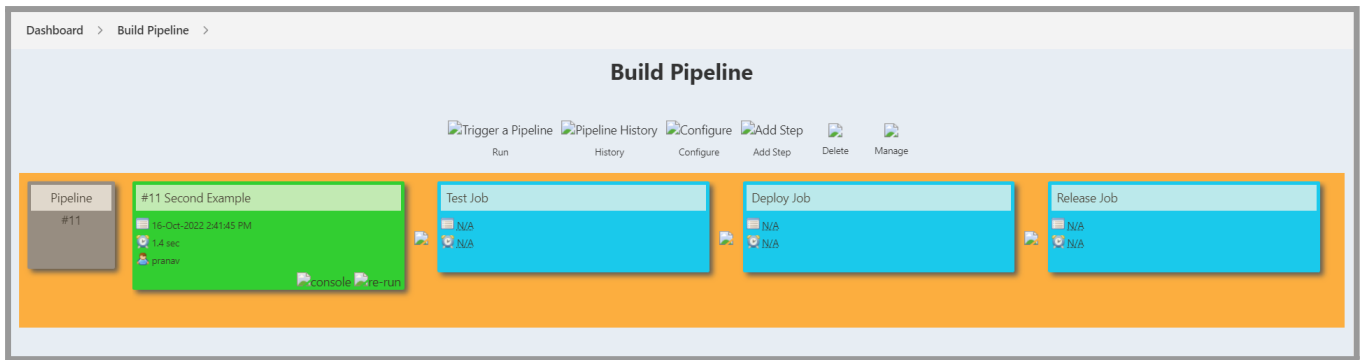
To create a CI CD Pipeline in Jenkins, we will first execute one task and let the others be executed sequentially. Steps:

1. Install build pipeline plugin in Jenkins.



2. Create a Build Pipeline View by clicking on the “+” sign in the Dashboard





3. Create the Delivery Pipeline view in the similar fashion

New view

Name

Type

☐ Build Pipeline View

Shows the jobs in a build pipeline view. The complete pipeline of jobs that a version propagates through are shown as a row in the view.

☒ Delivery Pipeline View

Continuous Delivery pipelines, perfect for visualization on information radiators. Shows one or more delivery pipeline instances, based on traditional Jenkins jobs with upstream/downstream dependencies.

Pipelines

Components

Component

Name ?

Initial Job ?

Second Example

Final Job (optional) ?

4. Create a Job which will be executed first. (Select No SCM and no Build Triggers). The job will contain a windows batch command to print that the job is completed.

Build

Execute Windows batch command ?

Command

See [the list of available environment variables](#)

```
echo "Build process successfully done: %date% : %time%"
```

5. Create a Test job. No Source code Management is required here. Select a build trigger to build after Job 1 is completed. Also add the windows batch command.

Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☒ Build after other projects are built ?

Projects to watch

Second Example,

☒ Trigger only if build is stable

☐ Trigger even if the build is unstable

☐ Trigger even if the build fails

☐ Always trigger, even if the build is aborted

Build

≡ Execute Windows batch command ?

Command

See [the list of available environment variables](#)

echo "Deployment is done: %date% : %time%"

6. Create a deployment job which will be the same as above but it will be build after Job 2.

Dashboard > Deploy Job >

General Source Code Management **Build Triggers**

Build Environment Build Post-build Actions

Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☒ Build after other projects are built ?

Projects to watch

Test Job,

☒ Trigger only if build is stable

☐ Trigger even if the build is unstable

☐ Trigger even if the build fails

☐ Always trigger, even if the build is aborted

Build

≡ Execute Windows batch command ?

Command

See [the list of available environment variables](#)

echo "Deployment is done: %date% : %time%"

7. Create a release job, same as earlier, which will be built after Job 3.

Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☒ Build after other projects are built ?

Projects to watch

Deploy Job,

☒ Trigger only if build is stable

☐ Trigger even if the build is unstable

☐ Trigger even if the build fails

☐ Always trigger, even if the build is aborted

Build

≡ Execute Windows batch command ?

Command

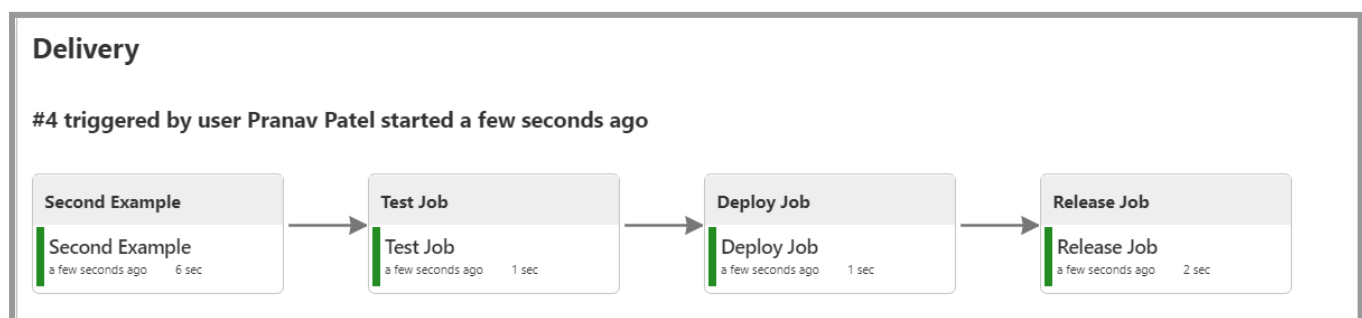
See [the list of available environment variables](#)

echo "Deployment is done: %date% : %time%"

8. Now the pipeline is ready. Execute Job 1 and observe how others get executed

S	W	Name ↓	Last Success	Last Failure	Last Duration	
		CI CD Pipeline	N/A	N/A	N/A	
		Deploy Job	3 mo 4 days	N/A	1.5 sec	
		First Example	3 mo 5 days	N/A	3.2 sec	
		GitIntegration	19 hr	N/A	8.6 sec	
		Project Jenkins	N/A	N/A	N/A	
		Release Job	3 mo 4 days	N/A	1.5 sec	
		Second Example	3 mo 4 days	N/A	2 sec	
		Test Job	3 mo 4 days	N/A	2.3 sec	

In the delivery pipeline, one can check the status of the tasks. Here each job is build only after the previous has been successfully executed.



Finally, the Console outputs of the tasks can be verified.

```

C:\ProgramData\Jenkins\.jenkins\workspace\Release Job>echo "Deployment is done: 16-10-2022 : 14:32:10.58"
"Deployment is done: 16-10-2022 : 14:32:10.58"

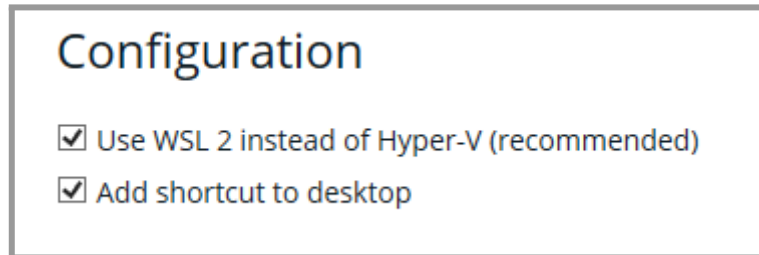
C:\ProgramData\Jenkins\.jenkins\workspace\Release Job>exit 0
Finished: SUCCESS
  
```

DevOps**Practical 3: Containerization****Q1. Install and configure docker**

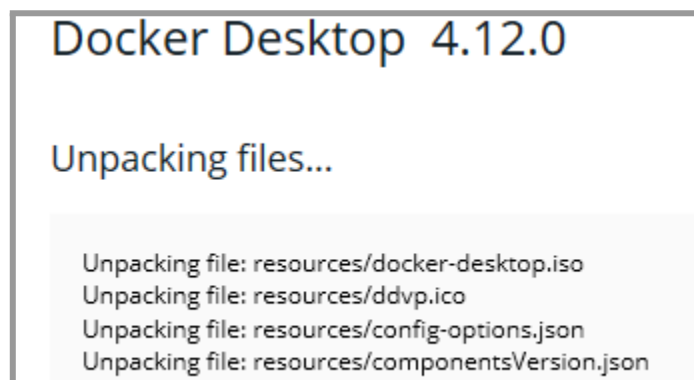
Step 1: Download Docker Desktop from <https://docs.docker.com/desktop/install/windows-install/>

Step 2: Start the installer downloaded earlier

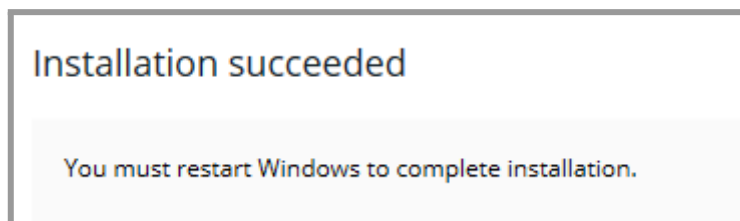
Step 3: Configure docker desktop by selecting the below options



Step 4: Docker will take some time to unzip the files and install.



Step 5: Upon successful installation, restart your computer



Step 6: If your admin account is different to your user account, you must add the user to the docker-users group. Run Computer Management as an administrator and navigate to Local Users and Groups > Groups > docker-users. Right-click to add the user to the group. Log out and log back in for the changes to take effect.

To install from the command line:

- After downloading Docker Desktop Installer.exe, run the following command in a terminal to install Docker Desktop: "Docker Desktop Installer.exe" install
- If you're using PowerShell you should run it as:
Start-Process 'Docker Desktop Installer.exe' -Wait install
- If using the Windows Command Prompt:

start /w "" "Docker Desktop Installer.exe" install

To check if docker is installed:

```
PS C:\Users\ADMIN> docker --version
Docker version 20.10.17, build 100c701
PS C:\Users\ADMIN> |
```

Q2. Create docker image using Dockerfile, Start docker container

Creating DockerFile:

PS C:\Users\Admin> cd MyDockerImages

PS C:\Users\Admin\MyDockerImages> New-item Newdockerfile

Directory: C:\Users\Admin\MyDockerImages

Mode	LastWriteTime	Length	Name
----	-----	-----	
-a----	02-08-2022 11:15	0	Newdockerfile

PS C:\Users\Admin\MyDockerImages> Set-Content C:\Users\Admin\MyDockerImages\Newdockerfile
cmdlet Set-Content at command pipeline position 1

Supply values for the following parameters:

Value[0]: FROM UBUNTU:20.04

Value[1]: RUN apt update

Value[2]: RUN apt install default-jdk -y

Value[3]: COPY ..

Value[4]: RUN ["javac","Demo.java"]

Value[5]: CMD ["java","Demo"]

Value[6]:

PS C:\Users\Admin\MyDockerImages> cat Newdockerfile

FROM UBUNTU:20.04

RUN apt update

RUN apt install default-jdk -y

COPY ..

RUN ["javac","Demo.java"]

CMD ["java","Demo"]

Building the Docker File and creating an image:

PS C:\Users\Admin\MyDockerImages> docker build .

[+] Building 191.4s (7/7) FINISHED

=> [internal] load build definition from Dockerfile 0.0s

=> => transferring dockerfile: 31B 0.0s

=> [internal] load .dockerignore 0.0s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/ubuntu:latest 4.4s

```

=> [auth] library/ubuntu:pull token for registry-1.docker.io          0.0s
=> [1/2] FROM
docker.io/library/ubuntu@sha256:34fea4f31bf187bc915536831fd0afc9d214755bf700b5cdb1336c82516
d154e 10.0s
=> => resolve
docker.io/library/ubuntu@sha256:34fea4f31bf187bc915536831fd0afc9d214755bf700b5cdb1336c82516
d154e 0.0s
=> => sha256:42ba2dfce475de1113d55602d40af18415897167d47c2045ec7b6d9746ff148f 529B /
529B 0.0s
=> => sha256:df5de72bdb3b711aba4eca685b1f42c722cc8a1837ed3fbd548a9282af2d836d 1.46kB /
1.46kB 0.0s
=> => sha256:d19f32bd9e4106d487f1a703fc2f09c8edadd92db4405d477978e8e466ab290d 30.43MB /
30.43MB 8.6s
=> => sha256:34fea4f31bf187bc915536831fd0afc9d214755bf700b5cdb1336c82516d154e 1.42kB /
1.42kB 0.0s
=> => extracting sha256:d19f32bd9e4106d487f1a703fc2f09c8edadd92db4405d477978e8e466ab290d
1.1s
=> [2/2] RUN apt-get update 176.7s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image
sha256:6336f24393cd69d17de3a3a19cd57d1b44ec652dd88273e548b8c6b6ab4c9b63 0.0s
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

```

Image created:

```

PS C:\Users\Admin\MyDockerImages> docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
<none> <none> 6336f24393cd About a minute ago 113MB
<none> <none> 2243a5e562a0 3 days ago 97.9MB

```

Running the Image:

```

PS C:\Users\Admin\MyDockerImages> docker run 6336f24393cd
Hello World

```

Pulling the Ubuntu Image:

```

PS C:\Users\Admin\MyDockerImages> docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
d19f32bd9e41: Already exists
Digest: sha256:34fea4f31bf187bc915536831fd0afc9d214755bf700b5cdb1336c82516d154e
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest

```


Running the container:

```
PS C:\Users\Admin\MyDockerImages> docker run -it -d ubuntu
930ffc6736b2a9c53c66f0676b3fab46c6f5617f35a3fb57495f6139b786f2d9
```

```
PS C:\Users\Admin\MyDockerImages> docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
930ffc6736b2	ubuntu	"bash"	15 seconds ago	Up 14 seconds	
mystifying_elgamal					
280d1270e43a	6336f24393cd	"echo 'Hello World'"	6 minutes ago	Exited (0) 6 minutes ago	
flamboyant_bose					

```
PS C:\Users\Admin\MyDockerImages> docker stop 280d1270e43a
280d1270e43a
```

```
PS C:\Users\Admin\MyDockerImages> docker start 280d1270e43a
280d1270e43a
```

Q3. Connect to docker container, Copy the website code to the container.

```
PS C:\Users\Admin\MyDockerImages> docker exec -it 930ffc6736b2 bash
root@930ffc6736b2:/# echo hello
hello
root@930ffc6736b2:/# exit
exit
```

Q4. Use docker management commands to

List the images:

```
PS C:\Users\Admin\MyDockerImages> docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
<none> <none> 6336f24393cd 15 minutes ago 113MB
ubuntu latest df5de72bdb3b 5 hours ago 77.8MB
<none> <none> 2243a5e562a0 3 days ago 97.9MB
```

List the containers:

```
PS C:\Users\Admin\MyDockerImages> docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
930ffc6736b2	ubuntu	"bash"	15 seconds ago	Up 14 seconds	
mystifying_elgamal					
280d1270e43a	6336f24393cd	"echo 'Hello World'"	6 minutes ago	Exited (0) 6 minutes ago	
flamboyant_bose					

Start and stop container:

```
PS C:\Users\Admin\MyDockerImages> docker start 280d1270e43a
280d1270e43a
```

```
PS C:\Users\Admin\MyDockerImages> docker stop 280d1270e43a
```

280d1270e43a

Remove container and image:

```
PS C:\Users\Admin\MyDockerImages> docker rm 930ffc6736b2
930ffc6736b2
```

```
PS C:\Users\Admin\MyDockerImages> docker rmi df5de72bdb3b
```

Untagged: ubuntu:latest

Untagged: ubuntu@sha256:34fea4f31bf187bc915536831fd0afc9d214755bf700b5cdb1336c82516d154e

Deleted: sha256:df5de72bdb3b711aba4eca685b1f42c722cc8a1837ed3fbd548a9282af2d836d

```
PS C:\Users\Admin\MyDockerImages> docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

DevOps

Practical 4: Configuration Management

Q1. Software Configuration Management and provisioning tools using Puppet.

Configuration management occurs when a configuration platform is used to automate, monitor, design and manage otherwise manual configuration processes. System-wide changes take place across servers and networks, storage, applications, and other managed systems.

An important function of configuration management is defining the state of each system. By orchestrating these processes with a platform, organizations can ensure consistency across integrated systems and increase efficiency. The result is that businesses can scale more readily without hiring additional IT management staff. Companies that otherwise wouldn't have the resources can grow by deploying a DevOps approach.

Configuration management is closely associated with change management, and as a result, the two terms are sometimes confused. Configuration management is most readily described as the automation, management, and maintenance of configurations at each state, while change management is the process by which configurations are redefined and changed to meet the conditions of new needs and dynamic circumstances.

A number of tools are available for those seeking to implement configuration management in their organizations. Puppet has carried the torch in pioneering configuration management, but other companies like Chef and Red Hat also offer intriguing suites of products to enhance configuration management processes. Proper configuration management is at the core of continuous testing and delivery, two key benefits of DevOps.

Based on what we've discussed, you may have already gleaned that configuration management takes on the primary responsibility for three broad categories required for DevOps transformation: identification, control, and audit processes.

Identification:

The process of finding and cataloging system-wide configuration needs.

Control:

During configuration control, we see the importance of change management at work. It's highly likely that configuration needs will change over time, and configuration control allows this to happen in a controlled way as to not destabilize integrations and existing infrastructure.

Audit:

Like most audit processes, a configuration audit is a review of the existing systems to ensure that it stands up to compliance regulation and validations.

Like DevOps, configuration management is spread across both operational and development buckets within an organization. This is by design. There are primary components that go into the comprehensive configuration management required for DevOps:

- Artifact repository
- Source code repository
- Configuration management data architecture

What is Puppet?

- Puppet is a **DevOps configuration management tool**. This is developed by Puppet Labs and is available for both open-source and enterprise versions. It is used to centralize and automate the procedure of configuration management.
- This tool is developed using Ruby DSL (domain-specific language), which allows you to change a complete infrastructure in code format and can be easily managed and configured.
- Puppet tool deploys, configures, and manages the servers. This is used particularly for the automation of hybrid infrastructure delivery and management.
- With the help of automation, Puppet enables system administrators to operate easier and faster.
- Puppet can also be used as a deployment tool as it can deploy software on the system automatically. Puppet implements infrastructure as a code, which means that you can test the environment for accurate deployment.
- Puppet supports many platforms such as Microsoft Windows, Debian/Ubuntu, Red Hat/CentOS/Fedora, MacOS X, etc.
- Puppet uses the client-server paradigm, where one system in any cluster works as the server, called the puppet master, and other works as a client on nodes called a slave.

Puppet Blocks

Puppet provides the flexibility to integrate Reports with third-party tools using Puppet APIs.

Four types of Puppet building blocks are

1. Resources
2. Classes
3. Manifest
4. Modules

Puppet Resources:

Puppet Resources are the building blocks of Puppet.

Resources are the inbuilt functions that run at the back end to perform the required operations in puppet.

Puppet Classes:

A combination of different resources can be grouped together into a single unit called class.

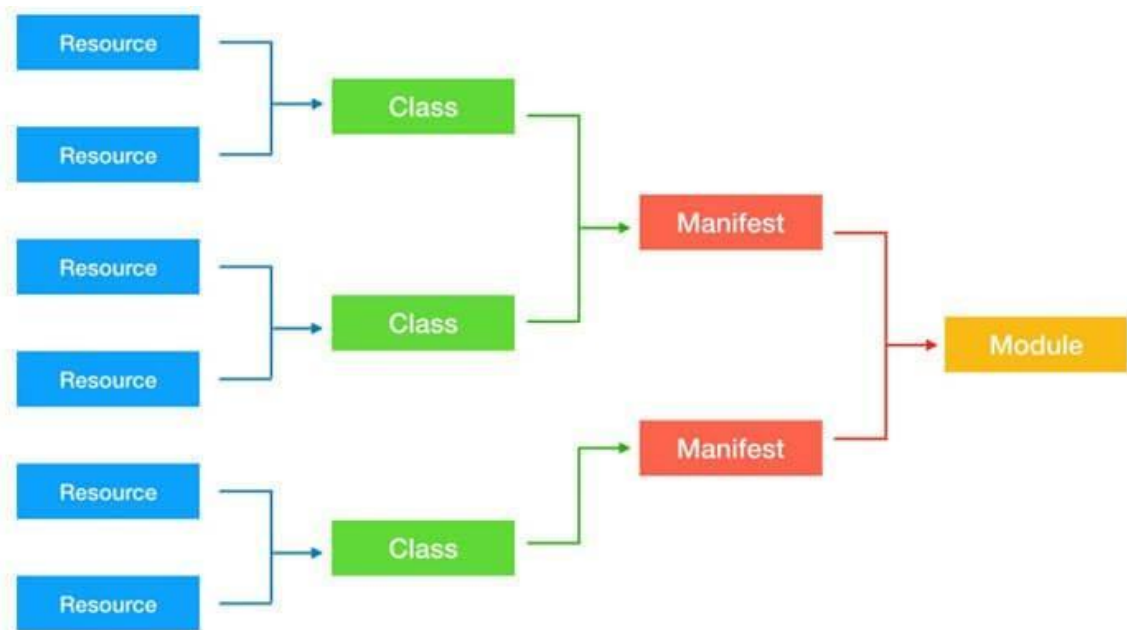
Puppet Manifest:

Manifest is a directory containing puppet DSL files. Those files have a .pp extension. The .pp extension stands for puppet program. The puppet code consists of definitions or declarations of Puppet Classes.

Puppet Modules:

Modules are a collection of files and directories such as Manifests, Class definitions. They are the reusable and shareable units in Puppet.

For example, the MySQL module to install and configure MySQL or the Jenkins module to manage Jenkins, etc..



Windows nodes cannot serve as puppet master servers.

- If your Windows nodes will be fetching configurations from a puppet master, you will need a *nix server to run as puppet master at your site.
- If your Windows nodes will be compiling and applying configurations locally with puppet apply, you should disable the puppet agent service on them after installing Puppet.

Installing Puppet

To install Puppet, simply download and run the installer, which is a standard Windows .msi package and will run as a graphical wizard.

The installer must be run with elevated privileges. Installing Puppet **does not** require a system reboot.

Install puppet client from <http://downloads.puppetlabs.com/windows/puppet-agent-x64-latest.msi>

Check installation via:

```

PS C:\WINDOWS\system32> puppet config print config
C:/ProgramData/PuppetLabs/puppet/etc/puppet.conf
PS C:\WINDOWS\system32> puppet config print modulepath
C:/ProgramData/PuppetLabs/code/environments/production/modules;C
:/ProgramData/PuppetLabs/code/modules
PS C:\WINDOWS\system32>
  
```

Manifest Components

Puppet manifest has the following components:

- o **Files:** Files are the plain text files that can be directly deployed on your puppet clients. Such as yum.conf, httpd.conf, etc.
- o **Resources:** Resources are the elements that we need to evaluate or change. Resources can be packages, files, etc.
- o **Templates:** This is used to create configuration files on nodes and which we can reuse later.
- o **Nodes:** Block of code where all the information and definition related to the client are defined here.
- o **Classes:** Classes are used to group different types of resources.

Writing Manifests

Working with Variables

Puppet provides many in-built variables that we can use in the manifest. As well as we can create our own variable to define in puppet manifest. [Puppet](#) provides different types of variables. Some frequently used variables are strings or an array of string.

Let's see an example for string variable:

```
$package = "vim"
package { $package:
  ensure => "installed"
}
```

Example

Writing a Manifest

As we know, we can create our resources. Let's start with common resources like notify resources.

```
notify { 'greeting':
  message => 'Hello, world!'
}
```

In the above code, the **notify** is the resource, and the message is the attribute. The **message** has attributes that are separated from their values by a comma, which is a very general way of identifying key-value pairs in [Ruby](#), [PHP](#), [Perl](#), and other scripting languages.

Applying a Manifest

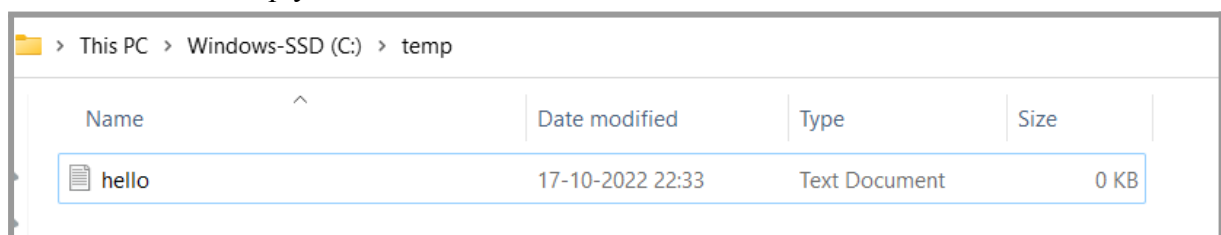
The main quality of the puppet is the ease of testing your code. In this, to work on puppet manifests, there is no need to configure complicated testing environments.

To apply the manifest, puppet uses apply command, which tells puppet to apply a single puppet manifest:

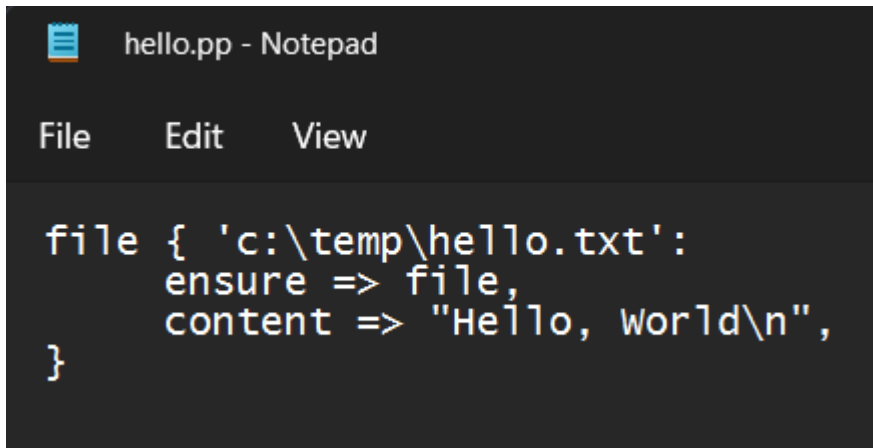
```
# puppet apply helloworld.pp
Notice: Compiled catalog for puppetclient in environment production in 0.03 seconds
Notice: Hello, World!
Notice: /Stage[main]/Main/Notify[greeting]/message:
  defined 'message' as 'Hello, World!'
Notice: Applied catalog run in 0.01 seconds
```

Writing Puppet Hello Program:

1. Create an empty hello.txt file



2. Create a file named hello.pp which has the following contents.

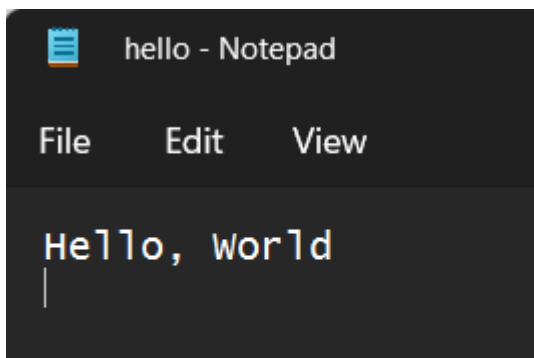


```
file { 'c:\temp\hello.txt':  
  ensure => file,  
  content => "Hello, world\n",  
}
```

3. Execute the command below in the same folder the hello.pp file is saved in

```
PS C:\temp> puppet apply hello.pp  
Notice: Compiled catalog for laptop-1p08fsrm in environment production in 0.16 seconds  
Notice: /Stage[main]/Main/File[C:\temp\hello.txt]/content: content changed '{md5}d41d8cd98f00b204e9800998ecf8427e' to '{md5}9af2f8218b150c351ad802c6f3d66abe'  
Notice: Applied catalog in 0.19 seconds  
PS C:\temp> _
```

4. Observe the changes in the hello.txt file



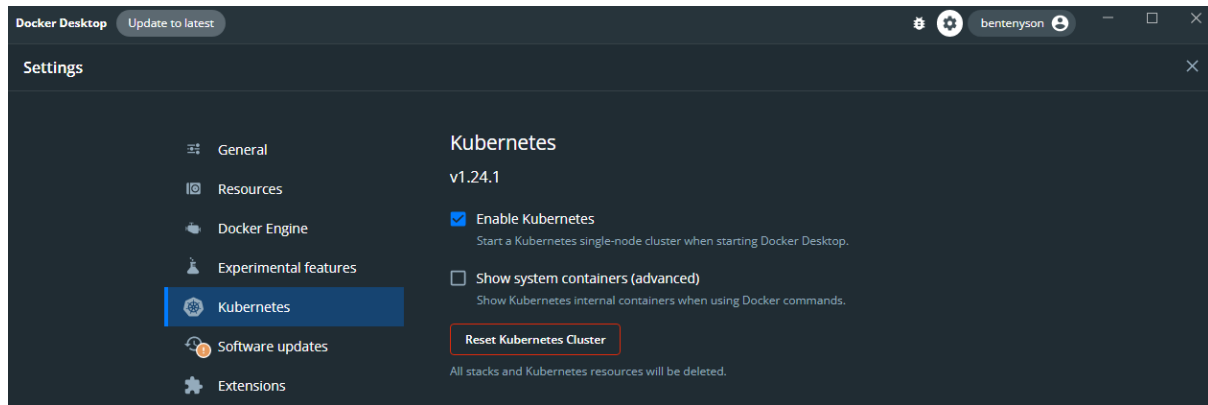
```
Hello, world
```

Q2. Configure Kubernetes, Configure Kubernetes Dashboard.

Dashboard is a web-based Kubernetes user interface. We can use Dashboard to deploy and troubleshoot containerized applications to a Kubernetes cluster, and to manage the cluster resources. We can also use Dashboard to get an overview of applications running on our cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). For example, we can scale a Deployment, initiate a rolling update, restart a pod, etc.

Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.

To start with the dashboard creation process, we have to enable the kubernetes feature in docker and ensure that it is running smoothly.



2: Then we run the following command which will create the required components for the dashboard
`kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.1/aio/deploy/recommended.yaml`
 This `kubectl` command is basically referring to the `recommended.yaml` manifest file stored in the github page for kubernetes dashboard and performing the operations

```
PS C:\Windows\system32> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.1/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
```

To protect your cluster data, Dashboard deploys with a minimal access control configuration by default. i.e. is why Currently, Dashboard will only support logging in with a Bearer Token.

For this, we will have to create a new user using the Service Account mechanism of Kubernetes, grant this user admin permissions and login to Dashboard using a bearer token tied to this user.

We will create a new manifest file named `dashboard-adminuser.yaml` and use it to perform the mentioned operations.

Creating a Service Account

We are creating Service Account with the name `admin-user` in namespace `kubernetes-dashboard` first.

File content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

The we use the `kubectl` command to execute the operations from the manifest file

```
kubectl apply -f dashboard-adminuser.yaml
```

```
PS C:\Users\user\OneDrive\Desktop\Assignments\Year 3\DevOps> kubectl apply -f dashboard-adminuser.yaml
```

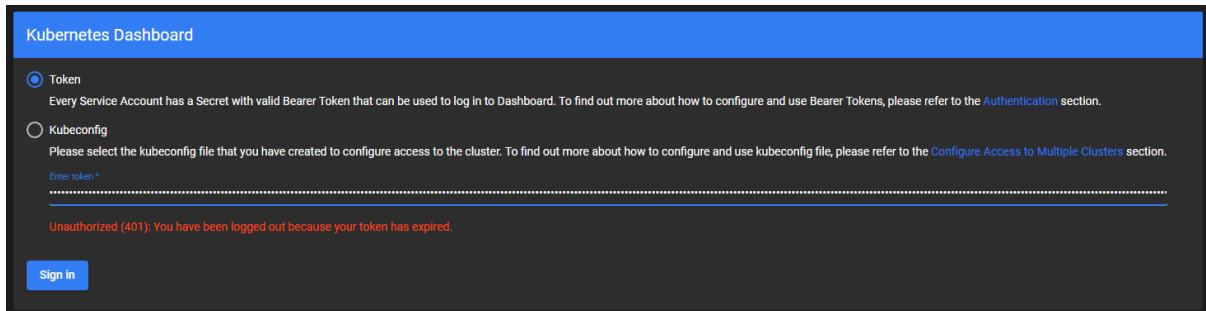
Output: `serviceaccount/admin-user created`

Creating a ClusterRoleBinding

Kubectl will make Dashboard available

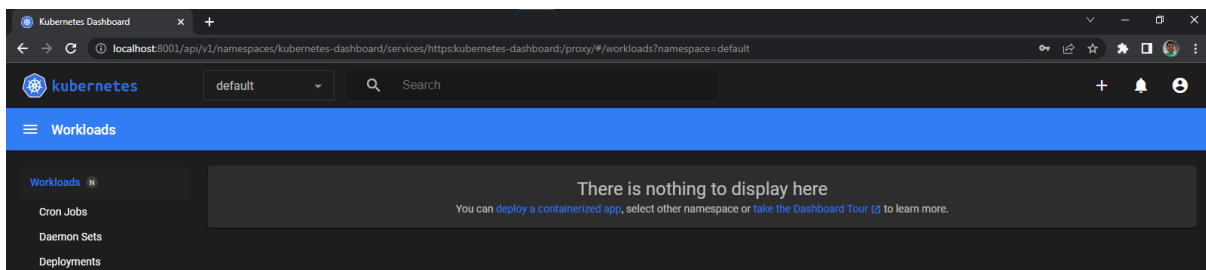
at <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>.

The UI can *only* be accessed from the machine where the command is executed.



Welcome view

When we access the Dashboard on an empty cluster, it will display the welcome page. This page contains a button to deploy our first application. In addition to it, we can view which system applications are running by default in the kube-system [namespace](#) of our cluster, for example the Dashboard itself.



Q3. Setup a Kubernetes Cluster, Access application using Kubernetes service.

A Kubernetes service can be used to easily expose an application deployed on a set of pods using a single endpoint.

A service is both a REST object and an abstraction that defines:

1. A set of pods
 2. A policy to access them
- Pods in a Kubernetes deployment are regularly created and destroyed, causing their IP addresses to change constantly. This will create discoverability issues for the deployed, application making it difficult for the application frontend to identify which pods to connect.
 - This is where the strengths of Kubernetes services come into play: services keep track of the changes in IP addresses and DNS names of the pods and expose them to the end-user as a single IP or DNS.
 - Kubernetes services utilize selectors to target a set of pods:
 1. **For native Kubernetes applications** (which use Kubernetes APIs for [service discovery](#)), the endpoint API will be updated whenever there are changes to the pods in the service.
 2. **Non-native applications** can use virtual-IP-based bridge or load balancer implementation methods offered by Kubernetes to direct traffic to the backend pods.

Discovering Kubernetes services

- **DNS:** Here, the DNS server is added to the Kubernetes cluster that watches the Kubernetes API and creates DNS records for each new service.

- **Environment Variables:** In this method, kubelet adds environment variables to Pods for each active service.

Creating services in Kubernetes

- Creating a deployment with the help of a yml configuration file:

```

my-deployment.yml - Notepad
File Edit Format View Help
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80

```

- We then execute the following command to create the deployment:

kubectl create -f my-deployment.yml

O: deployment.apps/nginx created

- We can get the details of deployment, replicaset and pod using the following commands:

kubectl get deployment

O: NAME READY UP-TO-DATE AVAILABLE AGE

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	4m48s

kubectl get replicaset

O: NAME DESIRED CURRENT READY AGE

NAME	DESIRED	CURRENT	READY	AGE
nginx-6c8b449b8f	3	3	3	5m1s

kubectl get pod

O: NAME READY STATUS RESTARTS AGE

NAME	READY	STATUS	RESTARTS	AGE
nginx-6c8b449b8f-jz9t7	1/1	Running	0	5m15s
nginx-6c8b449b8f-nqk7q	1/1	Running	0	5m15s
nginx-6c8b449b8f-w22d8	1/1	Running	0	5m15s

- Next, we will create a Service definition and create it using the following command:

```
my-service.yml - Notepad
File Edit Format View Help
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
spec:
  externalTrafficPolicy: Local
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: NodePort
```

kubectl create -f my-service.yml

O: service/nginx created

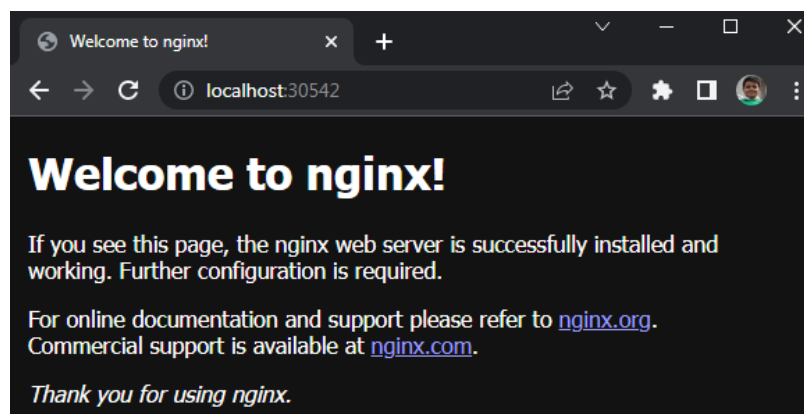
- We use the get service command to obtain to information, especially port number on which the service is running:

kubectl get service nginx

O: NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
nginx NodePort 10.110.47.19 <none> 80:30542/TCP 3m50s

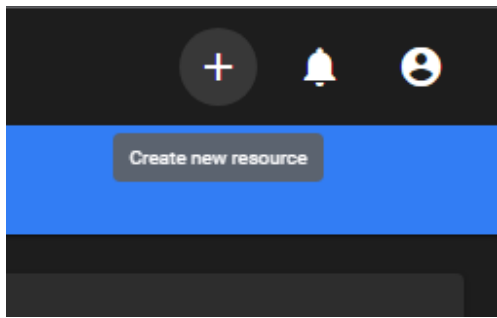
- Now, we can see that it is running on port 30542
- The nginx application can be accessed through this service on NodeIp:NodePort
- We are running the service locally, that is why we go to:

localhost:30542



Q4. Deploy the website using Dashboard.

Dashboard lets us create and deploy a containerized application as a Deployment and optional Service with a simple wizard. We can either manually specify application details, or upload a YAML or JSON *manifest* file containing application configuration or even write the file directly and execute it. To open that prompt, we first have to Click the **CREATE** button in the upper right corner of any page to begin.



Create from input Create from file **Create from form**

App name * 0 / 24 An 'app' label with this value will be added to the Deployment and Service that get deployed. [Learn more](#)

Container image * Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. [Learn more](#)

Number of pods *
 1 A Deployment will be created to maintain the desired number of pods across your cluster. [Learn more](#)

Service *
 None Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. [Learn more](#)

Namespace *
 default Namespaces let you partition resources into logically named groups. [Learn more](#)

Deploy Preview Cancel [Show advanced options](#)

Specifying application details

The deploy wizard expects that you provide the following information:

- **App name** (mandatory): Name for your application. A [label](#) with the name will be added to the Deployment and Service, if any, that will be deployed.

The application name must be unique within the selected Kubernetes [namespace](#). It must contain only lowercase letters, numbers and dashes (-). And is limited to 24 characters. Leading and trailing spaces are ignored.

- **Container image** (mandatory): The URL of a public Docker [container image](#) on any registry, or a private image (commonly hosted on the Google Container Registry or Docker Hub). The container image specification must end with a colon.
- **Number of pods** (mandatory): The target number of Pods you want your application to be deployed in. The value must be a positive integer.

A [Deployment](#) will be created to maintain the desired number of Pods across your cluster.

- **Service** (optional): For some parts of your application (e.g. frontends) you may want to expose a [Service](#) onto an external, maybe public IP address outside of your cluster (external Service).

Other Services that are only visible from inside the cluster are called internal Services.

Irrespective of the Service type, if you choose to create a Service and your container listens on a port (incoming), you need to specify two ports. The Service will be created mapping the port (incoming) to the target port seen by the container. This Service will route to your deployed Pods. Supported protocols

are TCP and UDP.

Service *

External

Port * **Target port *** **Protocol *** TCP

TCP

UDP

- **Namespace:** Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called [namespaces](#). They let you partition resources into logically named groups.

If needed, you can expand the **Advanced options** section where you can specify more settings:

Description

Labels

key	value
k8s-app	test-app

8 / 253

key

value

0 / 253

Image Pull Secret

CPU requirement (cores)

Memory requirement (MiB)

Run command

Run command arguments

☐ Run as privileged

Environment variables

Environment variables

Name	Value
------	-------

- **Description:** The text you enter here will be added as an [annotation](#) to the Deployment and displayed in the application's details.
- **Labels:** Default [labels](#) to be used for your application are application name and version. You can specify additional labels to be applied to the Deployment, Service (if any), and Pods, such as release, environment, tier, partition, and release track.

Example:

release=1.0

tier=frontend

environment=pod

track=stable

- **Image Pull Secret:** In case the specified Docker container image is private, it may require [pull secret](#) credentials.

- **CPU requirement (cores) and Memory requirement (MiB):** You can specify the minimum [resource limits](#) for the container. By default, Pods run with unbounded CPU and memory limits.
- **Run command and Run command arguments:** By default, your containers run the specified Docker image's default [entrypoint command](#). You can use the command options and arguments to override the default.
- **Run as privileged:** This setting determines whether processes in [privileged containers](#) are equivalent to processes running as root on the host. Privileged containers can make use of capabilities like manipulating the network stack and accessing devices.

Environment variables: Kubernetes exposes Services through [environment variables](#). You can compose environment variable or pass arguments to your commands using the values of environment variables. They can be used in applications to find a Service. Values can reference other variables using the \$(VAR_NAME) syntax.