**l tm: B m v v t**

At the end of this chapter, you should be comfortable with:

- Setting up your own Git repository
- Saving changes to your repository
- Inspecting your repository
- Traversing your repository
- Concepts of Branching, Merging, Pushing and Pulling

# 1. Introduction

## 1.1 What is Git?

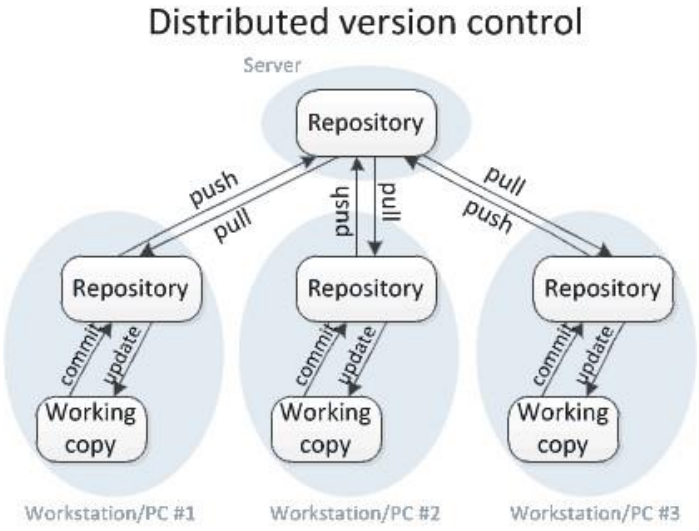
Git is a common and modern version control system for managing and tracing changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development. Git is a distributed version control system (DVCS) that has greater characteristics of performance, security and flexibility than most alternate version control systems.


## 1.2 What is a Version Control System?

- Version control systems, or VCSs, are a category of software instruments that support software development teams, manages changes to source code over time.
- It tracks the history of individual changes by each contributor to code in a special kind of database.
- If a mistake is made or a bug is to be fixed, developers can turn back to an earlier version of the source code to solve the problems without impeding the workflow of other team members.
- If a software team does not use a VCS they are subject to issues such as the creation of incompatible code between two independent parts of a project or ignorance towards the changes that are available to the users.

## 1.3 Why use Git?

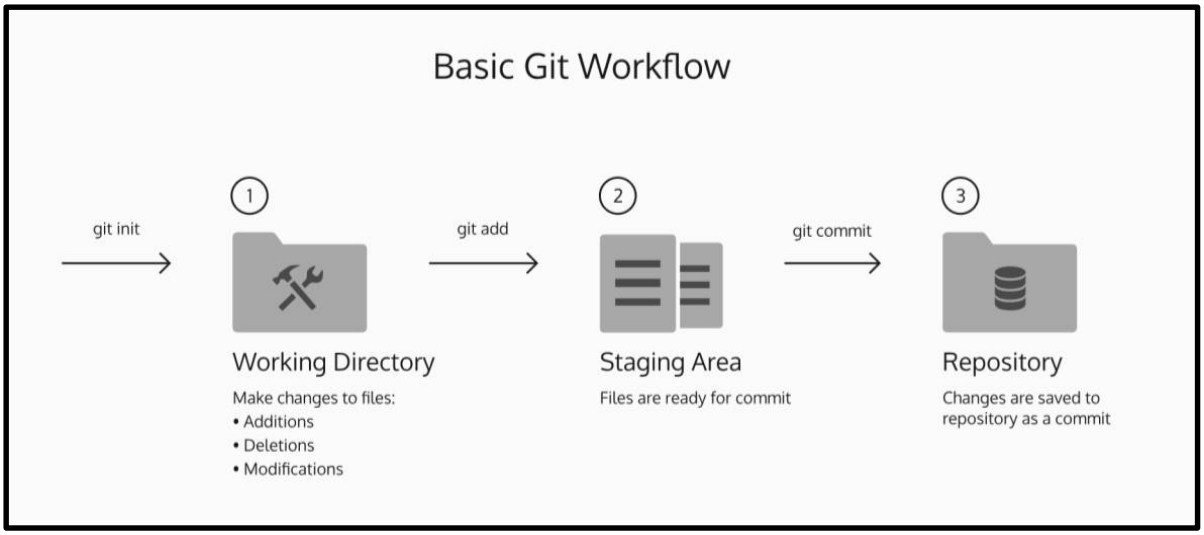
- Git lets developers see the entire timeline of their changes, decisions, and progression of any project in one place.
- With a DVCS like Git, collaboration can happen at any time while maintaining source code integrity. Using branches, developers can safely propose changes to production code.




- Businesses using Git can break down communication barriers between teams and keep them focused on doing their best work

## 1.4 Basic Git Workflow

The basic Git workflow is shown below. A more detailed discussion will be given in the next section.




- The **Working Directory** is where all changes will be made to the file.
- The **Staging Area** where you will mention all the changes made to the working directory.
- A **Repository** in which Git stores all changes made as different versions of the project.

# 2. Getting Started with Git

## 2.1 Installing Git

Before we start using git, we have to make sure it is there on your computer. If it is installed it is good to update it to the latest version.

To check if git is installed on your computer use [ ] command.

| |
|---|
| |

If it is not installed, follow the instructions below to install it.

**Installing on Linux**

To install git on Linux systems copy and paste the commands below to your terminal.

| |
|---|
| |
| |
| |

**Installing on Mac**

To make installation of software easier on Mac we download Homebrew. If you already have Homebrew you can skip the step below. If you do not have Homebrew copy the commands written below and confirm the installation.

| |
|---|
| |
| |

Copy and paste the code below to download git.

| |
|---|
| |

**Installing on Windows**

To install git on Windows, visit the link below and download it.

https://gitforwindows.org/

## 2.2 Initialising a new repository

A repository, or git repository, contains the entire collection of files, folders, directories, etc., along with a history of the changes made in the project.

There are two ways to create your local git repository. You can either initialise it or clone an existing remote directory.

**To Initialise a directory**

1. Open your terminal and browse to the directory of the root project folder using the `cd` command.
2. In this folder, we will use the command [ ] to initialize a new repository.

   | |
   |---|
   | |

   After executing this command, a new *.git/* subdirectory will be created in your current working directory. The command sets up all the tools Git needs to begin tracking changes made to the project.

3. Now we can create a file that we want to work on. For example, we can create *work.txt* as follow.

   | |
   |---|
   | |

[ ] has a few other command line options which you may find useful:

| Command | Meaning |
|---|---|
| | Creates a new local repository |
| | warnings. All other output is silenced |
| | Creates a bare repository |
| | Specifies directory from which templates will be used. |
| | Creates a text file containing the path to the actual repository |

**To Clone an existing remote directory**

1.

**NOTE :** If a *.git* directory is present, the repositories will be cloned there. If we do not have a *.git* directory then the repository will be cloned to your pwd (present working directory).

2. We type the command [ ] to clone a copy of the remote repository. In the example below, we copy the remote repository from the Internet to the current directory.

| |
|---|
| |

Some additional commands of [ ] are:

| Command | Meaning |
|---|---|
| | Specifies a specific branch to clone instead of the entire master branch |
| | Similar to [ ] creates a copy of the remote repository with an omitted working directory |
| | Clones all extended references of the remote repository and implicitly calls the argument |
| | Clones the repo and applies the template from to the newly created local branch |

## 2.3 Inspecting the working directory

As you keep making changes in your working directory (only *work.txt* at the moment), you can track all the changes made by using the command [ ].

```
```

Notice that your file *work.txt* is in red and under the untracked files. This means that Git can see the file but has not started tracking changes on that file yet.

## 2.4 Staging changes

In order for Git to start tracking the changes you make in the working directory, you need to add those files to the staging area first. This can be done by using the command [ *filename* ], where *filename* is the name of the *work.txt* file for us.

This command adds all changes in the working directory to the staging area. It is used to save a copy of the current state of your project.

After adding the file to the staging area, you can check the status of the files again using [ ]

```

```

Now you can see how Git shows that a new file was added to the staging area, in green.

Some useful commands are shown below:

| Command | Meaning |
|---|---|
| | Adds all files under the current directory |
| | Finds all new or updated files that are present throughout the project and adds them to the staging area |

## 2.5 Tracking changes in the working directory

If we make changes the *work.txt* by adding a third line.

To check the difference between the working directory and the staging area, we can use the command `filename`, where *filename* is the name of the file that you are checking.

The text indicated that the first two lines of text in the working directory are also in the staging area, and the last line of text is added in the working directory.

We can then add the changes made to the staging area using the command .

Below are some useful commands using .

| Command | Meaning |
|---|---|
| | View the conflicts against the base file (the point where the two branches started diverting the considered file) |
| | Preview changes before merging |

* Both commands are useful before merging. Merging is explained in the next section.

If the output of is too long, git will use a pager (using command or ), in that case, you need to press to exit the pager. Alternatively, you can add option to ask git not to use a pager. This option can be used in most of the git commands.

## 2.6 Committing changes

After staging all changes, the last step of the Git workflow would be to permanently save the changes from the staging area to the repository. Every time we do this, we create a commit that you can refers to in the future. For this, we use the command [ ] .

The option will be used to specify a commit message. The message should be enclosed in a pair of double-quotes( ). The message should describe the purpose or the changes in the commit.

[ ]

Notice the message above regarding your name and email address configuration. You can follow the instructions to make changes to the configuration file accordingly.

As a good development practice and for a good repository stewardship, always specify a meaningful commit message.

To access older version of the project, you can use the command [ ]. It lets you see the list of all previous commits, filter it, and also search for specific changes. All commits are stored in chronological order in the repository and can be accessed using this command.

[ ]

In each record, the following information will be given.

- A 40-character code called SHA (hash), that is used to uniquely identify the commit, typically seen in orange.
- The commit author, being yourself.
- The date and time of the commit
- The commit message

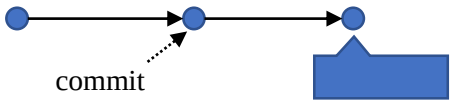
# 3. Collaborating with others

We have learnt about the basics of setting up a repository, inspecting it and saving the changes in a repository. We will now go over the basics of the mechanism used to collaborate with other users such as **Branching**, **Merging**, **Push**, **Pull**.

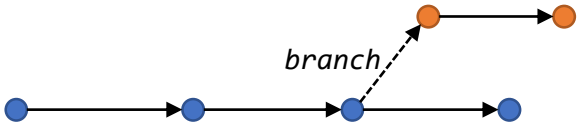
## 3.1 Branching

**What is Branching?**

A branch can be thought of as a pointer to the latest commit in your Git repository. When we initialise a repo, we are working on a single branch called the master branch. The commit we are working at is called HEAD, which is usually the latest commit of a branch. We can visualize the branch as a list.

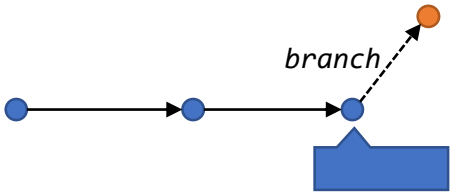



Branches can be created to allow changes to be made on different parts of a project simultaneously. This increases efficiency and allows for abstraction of work.

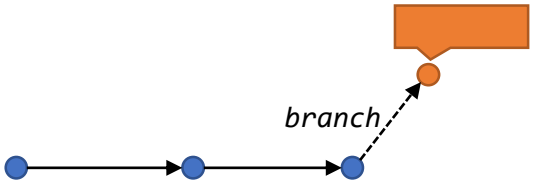



**Creating a Branch**

We will be using our local repository to create the branch. Type the command [ ] to create a new branch called .




Note that we are still working in the master branch. We need to switch to the new branch using the command [ ].




| |
|---|
| |

**Traversing your repository**

Above we have created a new branch named and used the command to switch to that branch.

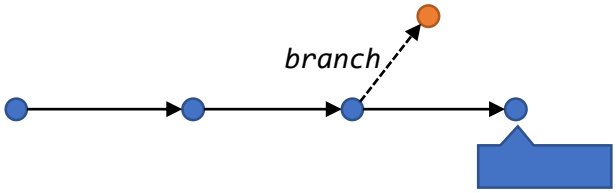
We can use the command to move our HEAD to a previous commit as well. If we do so, we restore the states of the files in this commit as well. For example, we can move to commit , which is the previous commit in the branch .

To move to a specific commit, use to find the hash.

Below are some useful commands involving branches:

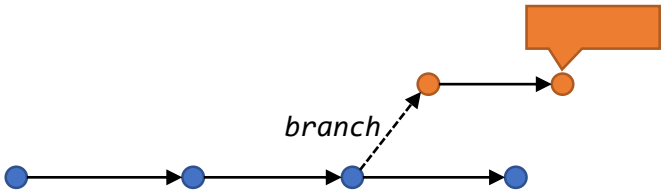
| Command | Meaning |
|---|---|
| | List all branches |
| | Create a branch |
| | Change to a branch |
| | Create and change to a new branch |
| | Rename branch |
| | Show all completely merged branches with current branch |
| | Delete merged branch (only possible if not HEAD) |
| | Delete not merged branch |

## 3.2 Merging




| |
|---|
| |
| |
| |

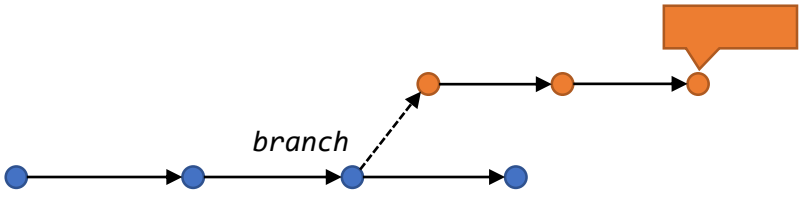
Next, we add another file in branches .




| |
|---|
| |
| |
| |

Here we mimic the case when someone is fixing a bug in branch and at the same time, some other continue the development in the master branch. After the bug is fixed, we would like the change in branch be reflected in the master branch which is our main project. To do this we can use the command.

There are two direction of merging, we could merge the changes in master to or the other way around. Depending on the situations, it may be safer to merge the changes in master to so that you can check if the merge is successful without touching the master branch.

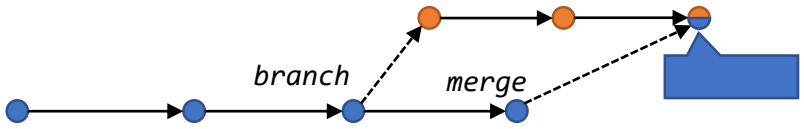


Since we are merging the changes in master to , we stay in the branch and apply the merge. As a new commit is created, you should specify a commit message using option , otherwise Git will ask you to input a message.

| |
|---|
| |

If the master branch has made a change that affect the change in branch , there will be a **conflict**. Git will ask you to resolve the conflicts before it can continue merging.

Now if the merge is successful, we may apply the bug fix on master. Suppose we checkout the master branch and do a merge again. Since there is only one path from master to , by default Git will simply move the master branch to the last commit in so they both share the same commit. This is called **fast forward**. Note that there is no commit created in this case.




| |
|---|
| |

Below are some useful commands involving merge:

| Command | Meaning |
|---|---|
| | Merge a branch (will fast forward if possible) |
| | Merge a branch only with fast forward |
| | Merge a branch without fast forward (force a new commit) |
| | Stop merge in case of conflicts |
| | Merge only one specific commit |

There is an alternative to merging. This command is called rebasing. To learn more about it, go to https://git-scm.com/docs/git-rebase.

## 3.3 Working with Remote Repositories

Till now we have only been working on our local repository. To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

**Showing Remote Repositories**

Use the command [ ] to see all the remote repositories according to the shortnames you have given them. [ ] allows you to look at the URL of the remote repository as well.

| |
|---|
| |
| |
| |

**Adding Remote Repositories**

To add a remote repository, use the command [ ]

| |
|---|
| |

Now, you can see a remote repository pb is added to the local repository.

**Inspecting a Remote Repository**

If you want to see more information about a particular remote, you can use the [ ] command.

| |
|---|
| |

Here are some of the important commands regarding remote repositories:

| Command | Meaning |
|---|---|
| | Viewing git remote configurations |
| | Viewing git remote configurations along with associated URLs |
| | Change to branch |
| | Remove remote repository |
| | Rename remote repository |
| | Show all completely merged branches with current branch |
| | Delete merged branch (only possible if not HEAD) |
| | Delete not merged branch |

## 3.4 Pushing

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. [ ] is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.

| |
|---|

This command works only if you cloned from a server to which you have to write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream,

allowed to push.

Here are some of the important commands regarding remote repositories:

| Command | Meaning |
|---|---|
| | Push the specified branch to <remote> |
| | Same as the above command, but force the push even if it results in a non-fast-forward merge |
| | Push all of your local branches to the specified remote. |
| | Sends all of your local tags to the remote repository |

### 3.5 Pulling

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration workflows.

| |
|---|
| |

Below are some commonly used commands with pull:

| Command | Meaning |
|---|---|
| | immediately merge it into the local copy. |
| | Fetches the remote content but does not create a new merge commit. |
| | Gives verbose content while pulling from the remote directory |

## 4. Further Reading

We have introduced the Git here. You will get familiar with them when you spend more time using it. The following webpages contains a very good introduction to working in Git. You are highly recommended to read it once.

**O 5o ot i L k u mv i v** _____

## 5. References


- Book. (n.d.). Retrieved from https://git-scm.com/book/en/v2