Git Repository Management in 30 Days

Learn to manage code repositories like a pro

Sumit Jaiswal

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-071

www.bpbonline.com

Dedicated to

My beloved wife

Kanika

&

My daughter

Anika

About the Author

Sumit Jaiswal has been engaged in software development for over 11 years, serving as a technical leader and software engineer on several projects utilizing Open-Source Technologies. He is currently a Principal Engineer at Ansible by RedHat. Meanwhile, he has obtained multiple Kubernetes and Security certifications. Furthermore, the author speaks at international conferences and writes technical blogs on Open-Source-related topics.

About the Reviewer

Paul Oluyege is an Innovative, Result and Data-driven Software Engineer and Manager. His professional experience of about 10 years cut across software development, process management, people management, project management and product management. He currently leads and manages a cross-functional development team of 20 members building and maintaining both new and existing products in multiple domains (Fintech, E-commerce B2B, B2C, SAAS, Logistics). Alongside, he is a tech author, tech coach and mentor.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my wife Kanika and my daughter Anika.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Git Repository Management in 30 Days welcomes you!

This book will give you a complete and practical approach to managing Git repositories. This book will help you grasp Git and take control of you are code, whether you're a newbie or an experienced developer.

Git is a critical tool for code management in modern software development. It lets engineers effectively track changes, collaborate with others, and manage code versions. Git, on the other hand, can be difficult and overwhelming for people who are new to it. This is where this book comes into play. This guide has been created to help you learn Git systematically and logically, with lessons that will take you from novice to expert in 30 days.

Each chapter of this book delves into a different facet of Git, beginning with the fundamentals of version control and progressing to more advanced topics like branching, merging, and rebasing. Collaboration, troubleshooting, and best practices for optimizing your productivity will also be covered. By the end of this book, you'll be able to confidently manage code repositories, interact with others, and streamline your development process.

With clear explanations, real-world examples, and step-by-step directions, this book is intended to be practical and approachable. We've also included challenges to help you put what you've learned into practise and improve your skills. This book is for you if you are a student, a nonexpert, or a professional developer.

Thank you for your interest in "Git Repository Management in 30 Days". We hope you find it useful and educational, and we look forward to assisting you in mastering Git and advancing your development abilities!

Chapter 1: Introduction to Git and GitHub - This is the introductory chapter. Source control is one of the key concepts and tools that is widely used in the software development process and without which DevOps makes little sense as it helps to bring collaboration and transparency between the development and operation teams. One of the most popular and well-liked source control systems is GIT, which is elaborated and extended by GitHub. This chapter covers the configuration and setup of GIT on various operating systems, as well as the creation of a GitHub account.

Chapter 2: Getting Started and Understanding Git and GitHub - Git and GitHub go hand in hand, but users should be aware of the differences that define each other's roles in the software development process. Any system or tool used to store and manage changes to projects over time is referred to as version control. The key advantages of source control include standardizing coding practices, parallelizing development activities, and eliminating dependencies. This chapter covers all the details around version control and goes on to examine Git in depth and detail, allowing you to clear a few basics about Git and make the learning process go more smoothly. Discussing Git gradually leads to the distinctions between Git and GitHub.

Chapter 3: Git Branching, Merging, and Rebasing - This chapter focuses on the essential capabilities of Git and GitHub, as well as how they complement each other in the software development and DevOps processes. It addresses the essential ideas of GIT as well as the basic day-to-day processes and commands that you may encounter while using the Git source control.

Chapter 4: Deleting, Renaming, and Ignoring Files in Git - This chapter builds on what readers learned in the previous chapter and allows you to make the final decision before pushing and committing changes to source control. This process of committing changes to the GitHub repo may include renaming, deleting, and ignoring files in the project.

Chapter 5: Collaborating Towards Your/Other Larger Projects over GitHub - This chapter discusses all of the process-related and critical aspects that should be kept in mind and followed before attempting to contribute to an open-source project that is being followed and used by a larger community from all over the world, as opposed to maintaining and contributing to a repo maintained by a single user.

Chapter 6: Contributing Towards Open-Source Project Repo - As one of the most important applications of using Git and GitHub together is how users can contribute to open-source projects that are part of GitHub, and having worked in open-source projects for quite some time, I've gained insights into how one should approach their contributions towards an open-source way of working and process, and one very important aspect of this is raising PR and issues over GitHub open source projects in a way that can get the most traction and help.

Chapter 7: Tags and Releases Using Git - This chapter goes over all of the Git and GitHub processes and important points to remember. Git only stores four kinds of objects in its object store: blobs, trees, commits, and tags. Managing releases using

Git and GitHub is a basic and straightforward procedure, and we will learn all about the principles and underlying commands involved.

Chapter 8: Undo or Refresh all the Work Done - This chapter focuses on Git's undo/refresh functionality, discussing all of the principles that Git exposes to assist users to achieve similar functionalities, as well as how employing GitHub processes and workflows can aid and make the corresponding job seamless and efficient.

Chapter 9: Most Commonly Used Git Commands - This chapter is essentially a summary of all of the chapters that we have gone through together. It will assist all users, whether beginner, intermediate, or advanced, in referring to this chapter whenever they may find it useful.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/aqascyr

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Git-Repository-Management-in-30-Days**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Introduction to Git and GitHub

Source control is one of the key concepts and tools used extensively in software development. With it, DevOps makes more sense as it helps bring collaboration and transparency between the development and the operation teams. The tracking and management of code changes are known as source control, and it ensures that developers are constantly working on the correct version of the source. One of the most used and loved by community source control is Git, which is elaborated and extended by GitHub. This chapter is about the configuration and setup of Git over different flavors of **Operating System** (**OS**) and setting up an account over GitHub.

Structure

In this chapter, we will cover the following topics:

- Version Control
- Introducing Git and GitHub
- Getting started with Git
    - Linus/Unix
    - Mac OS
    - Windows
- Creating and configuring the GitHub account

Objectives

After reading this chapter, you will get an understanding of What is source version control, and the Git version control. You will also get equipped with introducing Git and GitHub. You will also understand the running instance of Git and the difference between Git and GitHub. By the end of this chapter, you will have learned how to create an account on GitHub.

Once completed, you will learn about different types of version control systems, and how they evolved and resulted in the creation of Git. And as we progress through the chapter, we will go through all the information and requirements needed to follow along and complete all the examples and concepts discussed in the upcoming chapters, making the reading and development process easily consumable.

What is version control

Version control, also known as source control, refers to tracking and managing changes to code. This ensures that developers are always working on the right version of the source code.

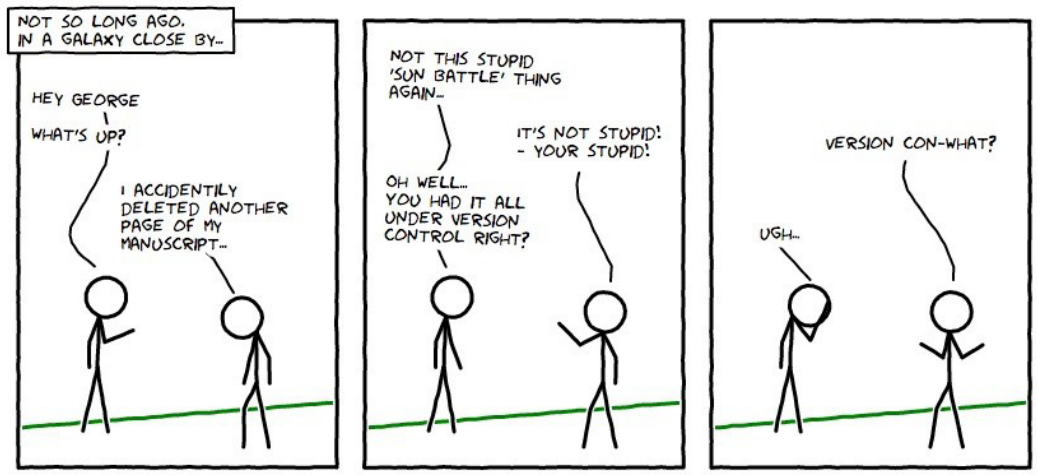

***Figure 1.1**: Why version control (credit: smutch)*

Why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

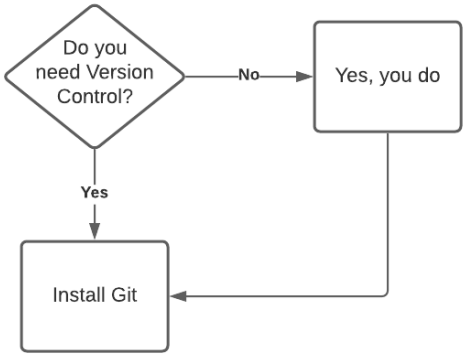

***Figure 1.2**: Version control importance*

Version control allows the developers the flexibility of making mistakes without worrying that they will have to start over the project / work. Basically, version control keeps track of all the changes at any time. If there is a need to undo any particular change, it can be done on the fly. Version control systems went through a series of evolutions with time and as project complexity grew.

Local Version Control Systems

The local version control system approach is very basic and simple, but it is also incredibly error prone. That is because it is extremely easy for the user to forget which directory they are in, and thus, they can mistakenly either write to the wrong file or copy over the entire files they do not mean to.

To avoid the above discussed issue, developers worked on the concept of local **Version Control Systems** (**VCSs**), which is a local database located on your local computer, in which every file change is stored as a patch. Every patch set contains only the changes made to the file since its last version.

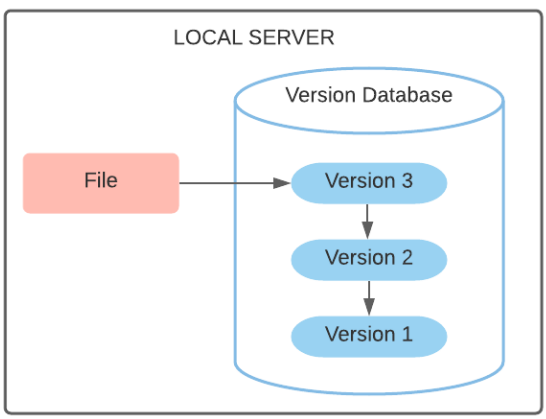

***Figure 1.3**: Local version control*

Centralized Version Control Systems

Centralized Version Management Systems (**CVCSs**) resolve the likely issues and challenges local version control systems face. The requirement to collaborate with developers on alternative systems became a recurring issue that developers and creators noticed over time. Systems (such as CVCS, Subversion, and Perforce) have a single server that contains all the versioned files. Various users use it to check out the files from a central location, so CVCS is still popular and is alternatively used instead of the local version control system.

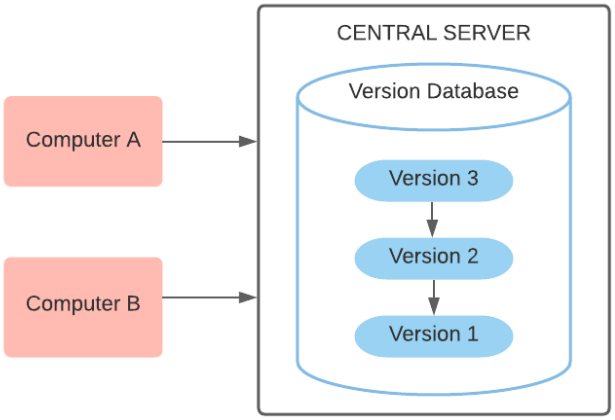

***Figure 1.4**: Centralized version control*

Distributed Version Control Systems

Distributed Version Control Systems (**DVCSs**), such as Git, Mercurial, Bazaar, or Darcs, each clone of the repository is the full backup of the repository data. This, in turn, means that when the user takes the latest snapshot of the files, DVCS takes the full mirror back of the repository. This includes the complete history of the repository. It helps when the server hosting the repository crashes, and any of the users' repositories can be copied back up to the server to help restore the repository content onto the server.

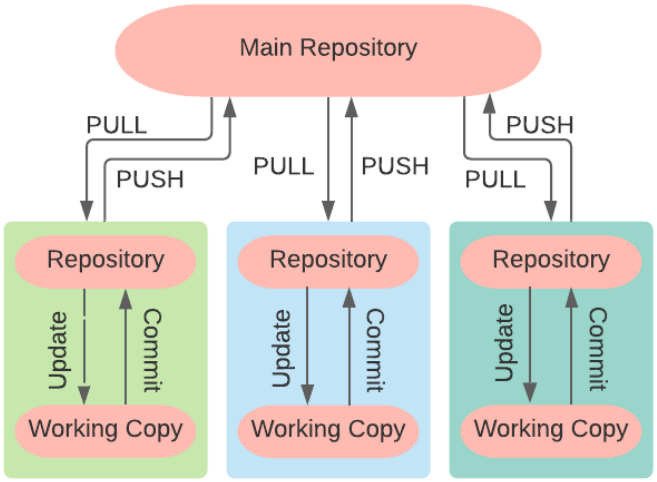

***Figure 1.5**: Distributed version control*

Git History

Like numerous extraordinary things in everyday life, Git started with a touch of innovative obliteration and blazing debate.

The Linux OS kernel is a free and open-source OS with extreme opportunities. For a large portion of the lifetime of Linux piece support (1991–2002), changes to the operating system were passed around as patches and archived files. In 2002, the Linux OS project started utilizing a restrictive DVCS called BitKeeper.

In 2005, the agreement between the DVCS system BitKeeper and the community that had worked on its Kernel got revoked, and thus BitKeeper being used as a free tool got renounced too. This resulted in the Linux community (particularly Linus Torvalds, the creator of Linux) working and developing their own tool based on the learnings when using BitKeeper. Linux community also prioritized the goals which they wanted in the new system. They are as follows:

Design that is simpler and easier to use:

Well-rounded support for non-linear development (that is, working on thousands of parallel branches)

Distributed Completely:

Should be able to handle large projects like the Linux kernel efficiently and with zero tolerance (speed and data size)

Git became self-hosted on April 7 with this commit:

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

Initial revision of "git", the information manager from hell
```

***Figure 1.6**: Git first commit*

Shortly thereafter, the first Linux commit was made:

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

Linux-2.6.12-rc2

Initial git repository build. I'm not bothering with the full history, even though we have it. We can create a
separate "historical" git archive of that later if we want to, and in the meantime it's about 3.2GB when
imported into git - space that would just make the early git days unnecessarily complicated, when we don't
have a lot of good infrastructure for it.

Let it rip!
```

***Figure 1.7**: Linux first commit*

From the time Git came into existence around 2005, it has evolved and matured into a tool that is easy to use and yet inherited and extends all the capabilities of DVCS. It also ticked all the initial use cases and principles it was built upon, which is why it is lightning-fast and very efficient for large projects. It also has an incredible branching system for non-linear development.

What is Git

This section is a key to understanding the underlying concept and principles upon which Git is built. If you follow this section keenly, you can learn how Git works fundamentally and use the concepts and knowledge to use Git effectively when you start using the same for your projects. As discussed previously in the chapter Git is a distributed version control system. Other version control systems are also available in the market, but Git functions differently and stores the information differently.

***Figure 1.8**: Git User experience*

A major difference between Git and any other **version control system** (**VCS**) is how they store information. Other VCS store data as a rundown of record-based changes. These different frameworks like (Central Version control system, Subversion, Perforce, Bazaar, and so on) think about the data they store like a bunch of records and the progressions made to each of the records over the long haul (this is more commonly represented as delta-based variant control).

On the other hand, Git considers its record information more like a series of smaller snapshots of the filesystem. With Git, each time users commit or save the state of the user's project, Git essentially snaps a photo of what every one of your records resembles at that point and stores a reference to that depiction. To be effective, if records have not changed, Git does not store them again. Simply a connection to the past indistinguishable records has already been effectively stored. Git considers its data to be more like a stream of snapshots.

Simply put, every time a change is made to the filesystem, Git just merges those changes with the already present ones instead of replacing or overriding the existing content.

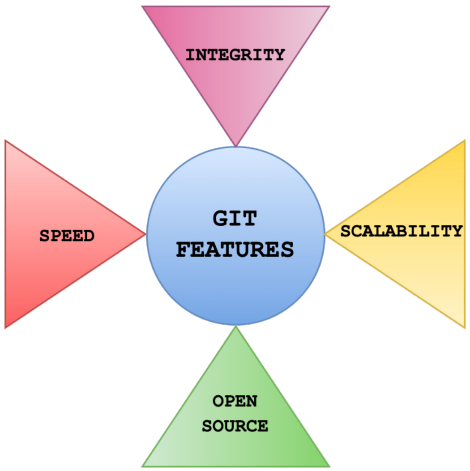

***Figure 1.9**: Git Features*

Git has a high level of performance and integrity:

Most Git operations are done locally and only need local files and objects. On the contrary, other VCSs have network latency overhead which gives Git a major performance boost as the entire history of the project files is stored over your local disk and is thus available instantaneously.

This also gives you the freedom to work remotely or without network connectivity as you can save your changes to your local copy. Once back online, you can push the required changes to the repo, whereas other VCSs do not have this flexibility.

When it comes to Integrity and Git knowing things, there is no way any file can be updated or modified without Git knowing it, and this is taken care of by Git's checksum in place. Everything in Git is check summed before it is stored and is then referred to by that checksum.

SHA-1 hash is used for the checksum by Git, and you will see these hash values all the time as, Git stores everything in its database not by file name but by the hash value of its contents.

Git three States

Git principally works based on three stages, as depicted in *Figure 1.10*, and the files stored under Git can either be in a **modified**, **staged,** or **committed** state:

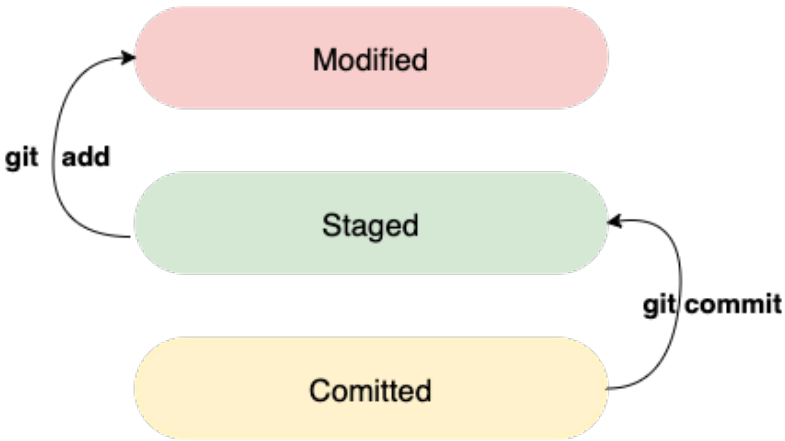

***Figure 1.10**: GIT 3 states*

Modified: It means that the user has changed/edited the file but has not made the changes to their GitHub repository database yet.

Staged: It means that the user has marked a modified file in its current version, which is supposed or will go into the user's next commit snapshot.

Committed: The users' GitHub repository data changes are safely stored in the users' local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.

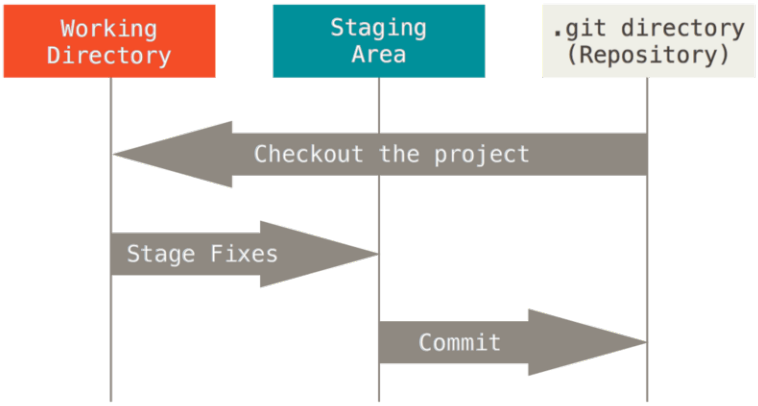

***Figure 1.11**: Working tree, staging area, and Git directory (source:* Git*)*

The Git working tree is a single checked-out version of the project, where the files are pulled out from a compressed database from the Git directory. And this is then placed over the user's disk for them to use, update and modify.

The staging area is a file, generally contained in the user's Git directory, which generally stores the information about what would go into the next commit. Technically and more precisely, it is called Git parlance, which is the "index," but the phrase "staging area" also works.

The Git repository directory is the place where Git stores the metadata and item information base for the user's project. This is the most important aspect of Git, duplicated when users clone a Git repository from a different computer.

The usual Git workflow is a process that goes as follows:

Users modify the files in their working tree. Users then selectively stage those changes they want to be part of their next commit, adding only those changes to their respective staging area. Users do a commit, which will take the files as they are in the staging area and then stores that particular snapshot permanently in their Git repository directory.

Getting started with Git

In this section of the chapter, we will go through some of the basics of Git, its installation and setup procedure on your work machine.

This chapter will cover Git installation on the three most used OS platforms:

- Linux/Unix
- Windows
- Mac OS

We will go through the installation steps for each installation process via screenshots. All installation shown here will be done through command line, that is, CLI, as Git and GitHub are coupled. All its functions can easily be controlled via CLI. Working over CLI is more efficient than working over UI based.

We will start the installation process in the preceding order of OS platforms:

Linux/Unix

For **Linux/Unix** platform, the installation process is very similar for all the available platforms with different commands based on the flavors of **Linux/Unix.**

1. For all the commands, you can visit the Git website (refer to *Figure 1.1*):

   https://git-scm.com/

   https://git-scm.com/downloads

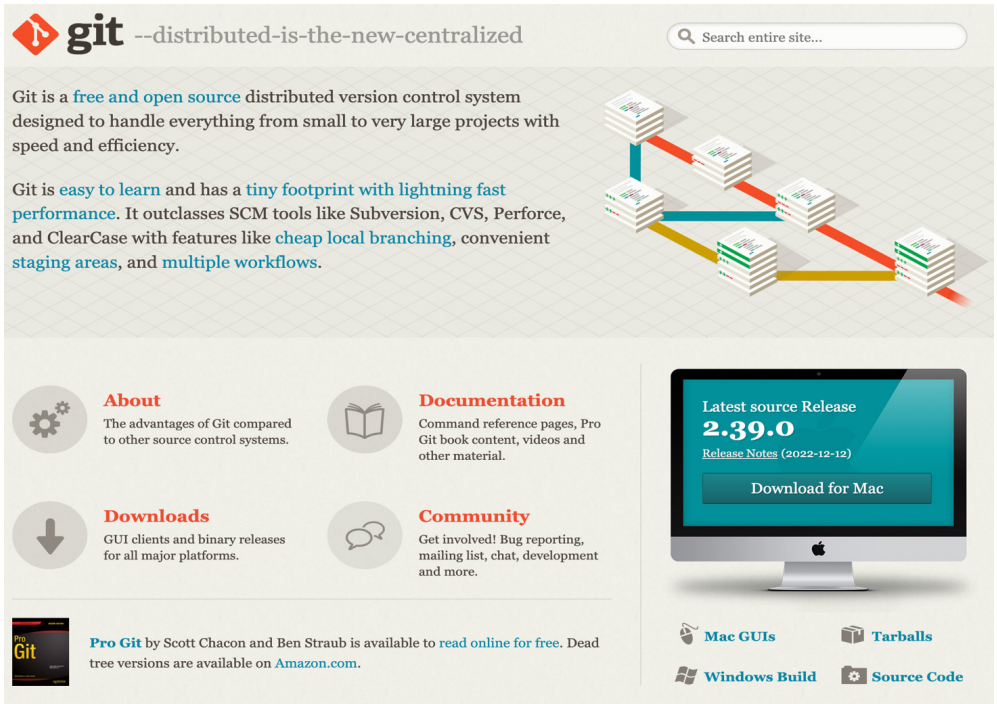

***Figure 1.12**: Git Homepage*

2. Then, browse to the **Download** section. It will open the page as shown in the following figure:

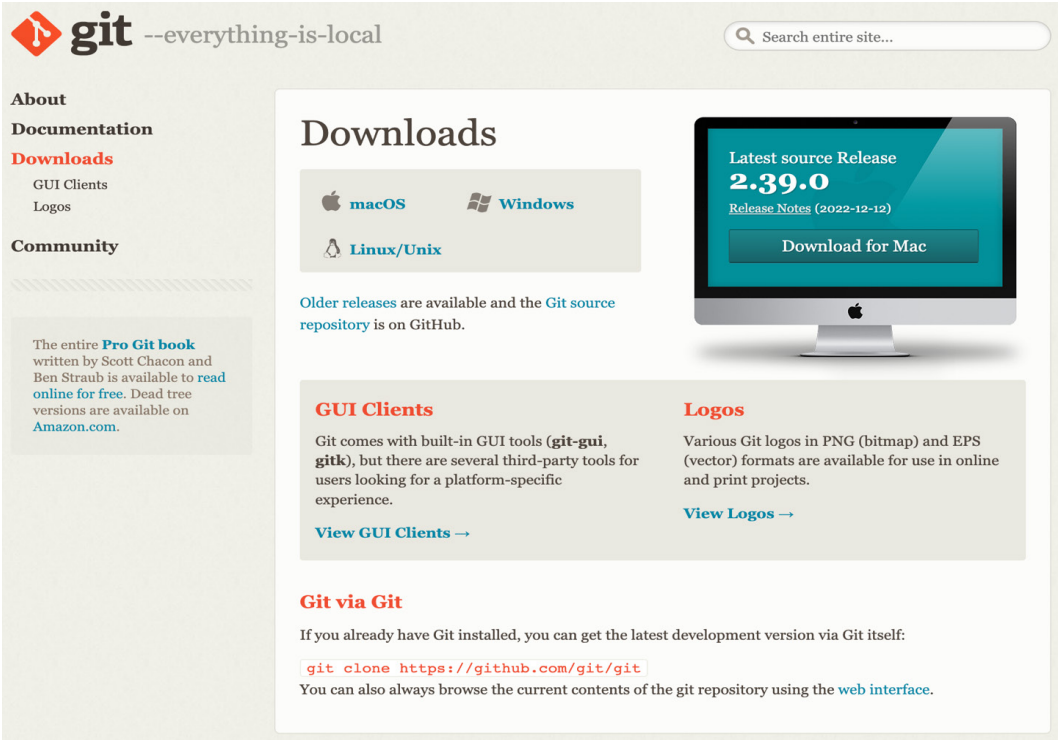

***Figure 1.13**: Downloads page*

3. From there, click on  **Linux/Unix** download section, which will take you to the page shown in the following figure. It guides on how to install Git over different flavors of **Linux/Unix:**

Download for Linux and Unix

It is easiest to install Git on Linux using the preferred package manager of your Linux distribution. If you prefer to build from source, you can find tarballs on kernel.org. The latest version is 2.31.1.

Debian/Ubuntu

For the latest stable version for your release of Debian/Ubuntu

```
# apt-get install git
```

For Ubuntu, this PPA provides the latest stable upstream Git version

```
# add-apt-repository ppa:git-core/ppa # apt update; apt install git
```

Fedora

```
# yum install git
```
(up to Fedora 21)
```
# dnf install git
```
(Fedora 22 and later)

Gentoo

```
# emerge --ask --verbose dev-vcs/git
```

Arch Linux

```
# pacman -S git
```

openSUSE

```
# zypper install git
```

Mageia

```
# urpmi git
```

Nix/NixOS

```
# nix-env -i git
```

FreeBSD

```
# pkg install git
```

Solaris 9/10/11 (OpenCSW)

```
# pkgutil -i git
```

Solaris 11 Express

```
# pkg install developer/versioning/git
```

OpenBSD

```
# pkg_add git
```

Alpine

```
$ apk add git
```

Red Hat Enterprise Linux, Oracle Linux, CentOS, Scientific Linux, et al.

RHEL and derivatives typically ship older versions of git. You can download a tarball and build from source, or use a 3rd-party repository such as the IUS Community Project to obtain a more recent version of git.

Slitaz

```
$ tazpkg get-install git
```

***Figure 1.14**: Git Download version*

Here, we have used the Ubuntu example, a Linux-based Operating System that belongs to the Debian family of Linux. Since it is **Linux** based, it is freely available for use and is open source. Here, Ubuntu version 20.04 is used.

We will not go through the installation procedure for Ubuntu, so I assume you are up and running with the Ubuntu box, which needs to install and configure Git. Let us begin the installation procedure:

1. Check if the ubuntu box is updated and upgraded by running the following:

   `sudo apt-get update`

   `sudo apt-get upgrade`

2. Once the Ubuntu image is updated and upgraded, try running git command. If the box is freshly installed or does not have Git installed, you should see the results shown in the following screenshot:

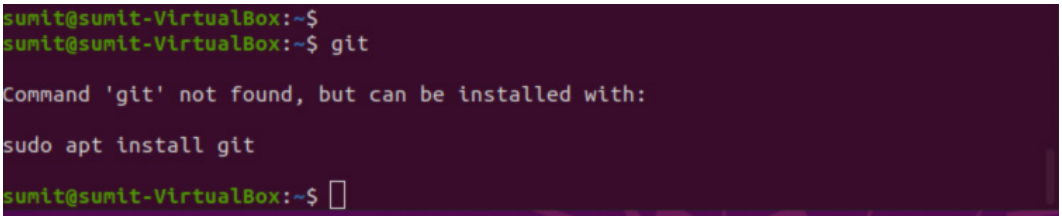

***Figure 1.15**: Git Ubuntu installation (a)*

3. Now, for installing Git, we need to run the following command:

   `sudo apt install git`

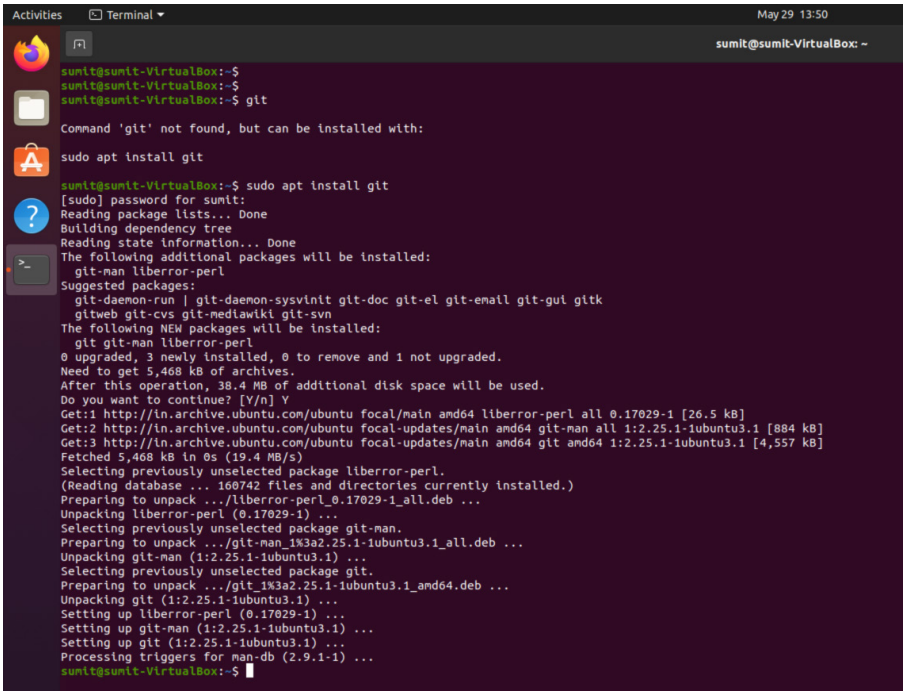

***Figure 1.16**: Git Ubuntu installation (b)*

4. To verify if Git is successfully installed, run the following command:
   `git --version`

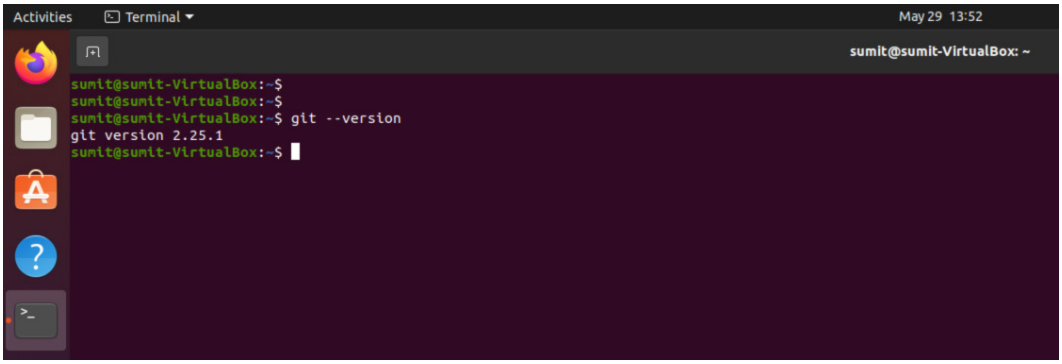

***Figure 1.17**: Git Ubuntu installation (c)*

5. And, if you run the Git command, it should show all the GIT options:
   `git`

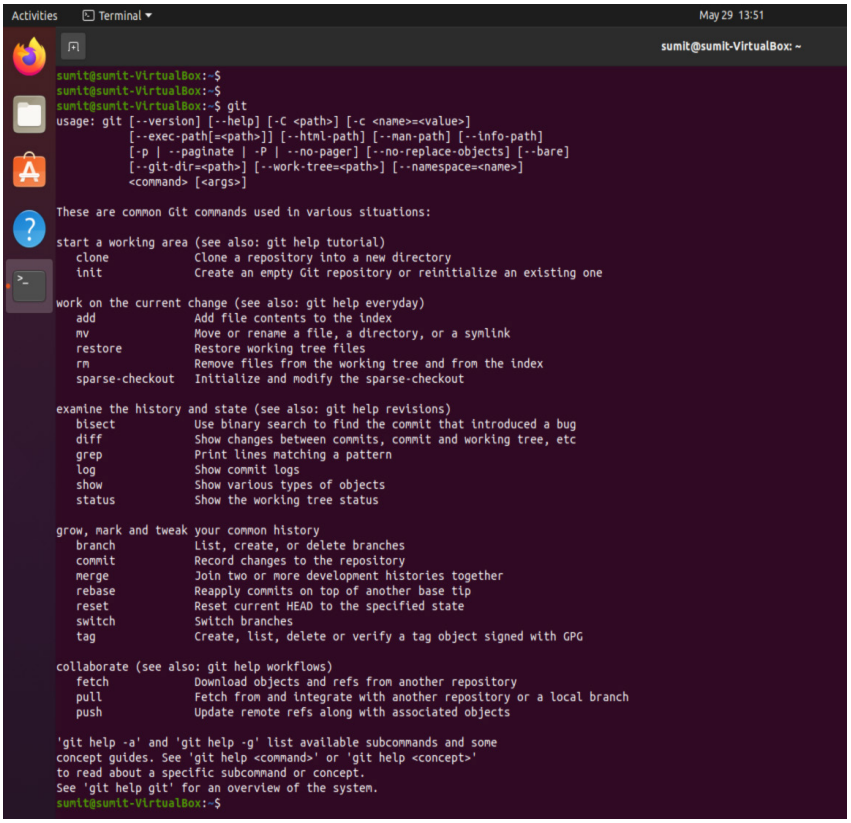

***Figure 1.18**: Git Ubuntu installation (d)*

Mac OS

We have seen that the installation process for Linux and Mac OS are very similar. Here, we will function through the command line Git, not the GUI application.

For Mac OS installation either you can directly run the command:

- brew install git,

or

Browse down to the Git website and click on Mac OS download information. Git will show the command that you can use to download the Git. At the time of writing this book, the following is the screenshot of the Git Mac OS download page:

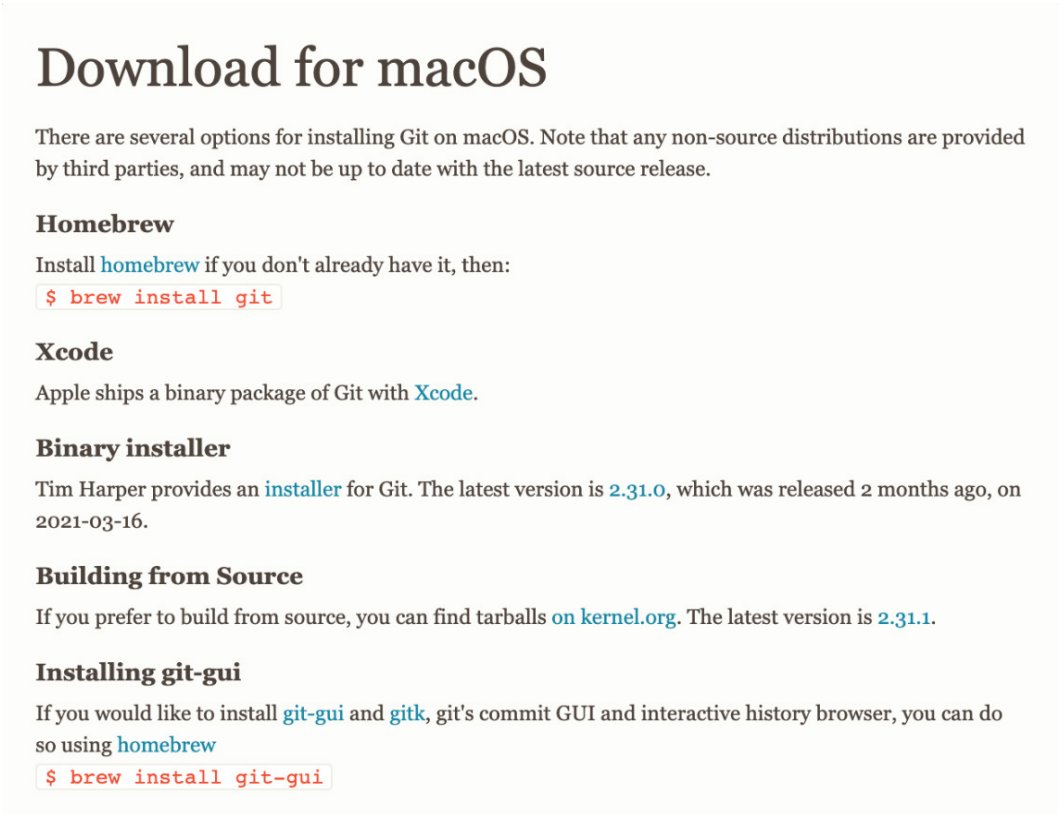

***Figure 1.19**: Git Mac installation instructions*

Once the installation via brew completes, to verify if the installation is working as expected, run the following command:

- **Git**
- **Git -version**

Also, if the installation was done successfully and as expected, you should see similar results as shown in the following screenshot:

```
➜  ~ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone     Clone a repository into a new directory
   init      Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add       Add file contents to the index
   mv        Move or rename a file, a directory, or a symlink
   restore   Restore working tree files
   rm        Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect    Use binary search to find the commit that introduced a bug
   diff      Show changes between commits, commit and working tree, etc
   grep      Print lines matching a pattern
   log       Show commit logs
   show      Show various types of objects
   status    Show the working tree status

grow, mark and tweak your common history
   branch    List, create, or delete branches
   commit    Record changes to the repository
   merge     Join two or more development histories together
   rebase    Reapply commits on top of another base tip
   reset     Reset current HEAD to the specified state
   switch    Switch branches
   tag       Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch     Download objects and refs from another repository
   pull      Fetch from and integrate with another repository or a local branch
   push      Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
➜  ~
➜  ~ git version
git version 2.24.3 (Apple Git-128)
➜  ~
```

***Figure 1.20**: Mac post Git installation*

Windows

With windows, the process is slightly different since Git provides the UI-based installer, where users can select the default settings with which users want their Git installed and work out of the box. Please follow the steps as shown via screenshots, and at the end of this, the reader should have Git working over Windows box as it worked in Linux/Mac via the Windows command line.

1. To verify if the windows box has Git already installed, use the command as follows:
   - `git,` or
   - `git -version`

   For either of the commands, if Git is not already installed, you should see a Windows error of `git` not being recognized.

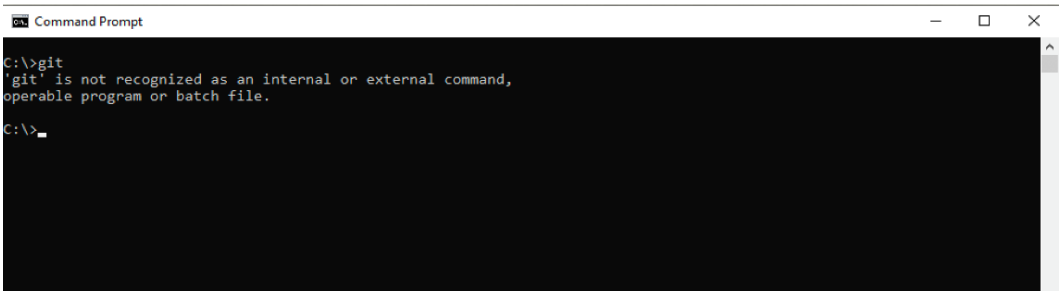

***Figure 1.21**: Git check via Windows command prompt*

2. To install Git on your windows box, visit the Git official website page and click on the **Download** tab to traverse the download page for downloading the Windows installer. Click on download via Windows icon, and your installer should start to download as shown in the following screenshot:

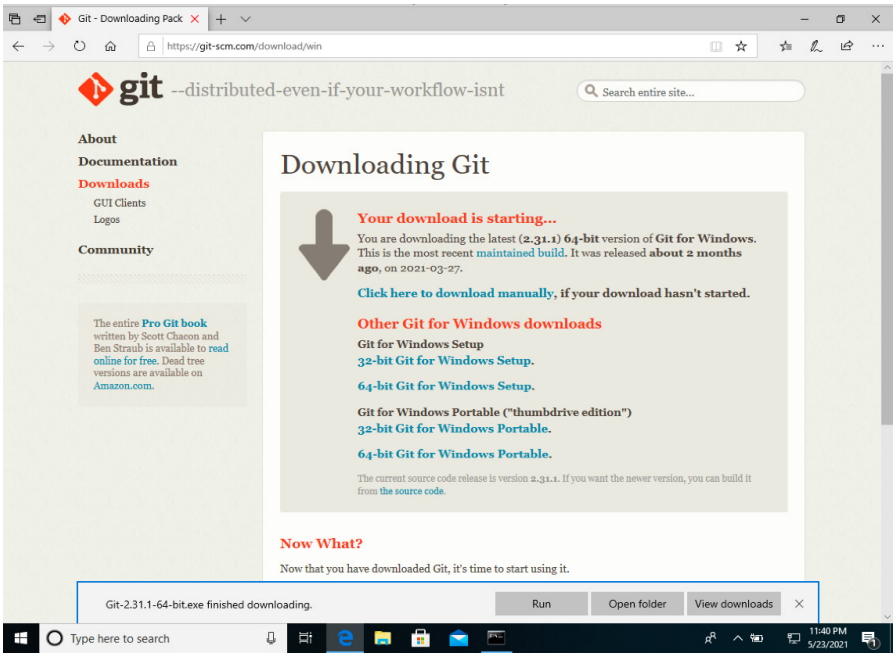

***Figure 1.22**: Git Windows installer download*

3. Once downloaded, the Git installer should be visible in the download folder or any specific path you might have set as follows:

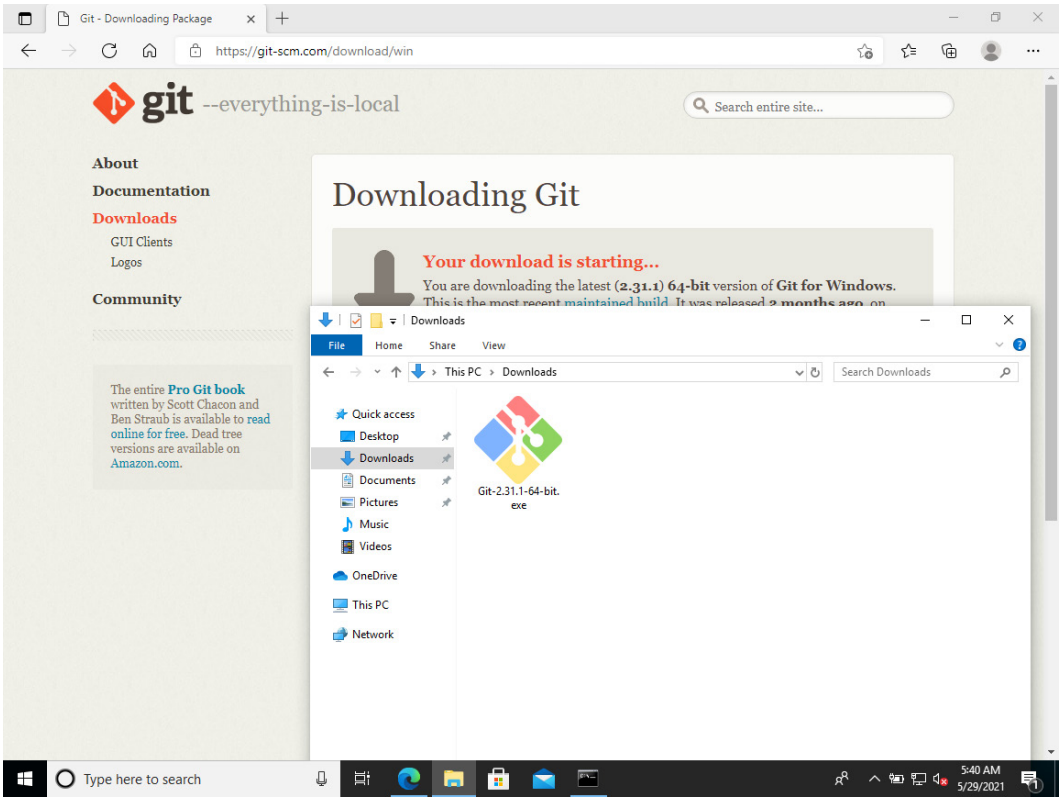

***Figure 1.23**: Git Windows installer*

4. Before beginning the installation process via the Git windows installer, please know that, unlike Linux/Mac installation, the Windows installer will ask for defaults that you need to set before the installation can begin. While most default settings are already selected, the installer should be kept as is unless you need to update specific changes for your respective Git installation. Here we will discuss a few defaults you can edit as suggested or keep the installer's defaults.

***Figure 1.24**: Git installer Windows installation (a)*

5. In the following installer page, you can select/deselect various options based on your need. For example, you can wish for Git to install the GUI or not, but here, we will continue with the default installer settings. However, you can make changes based on your requirements, as all options are configurable.

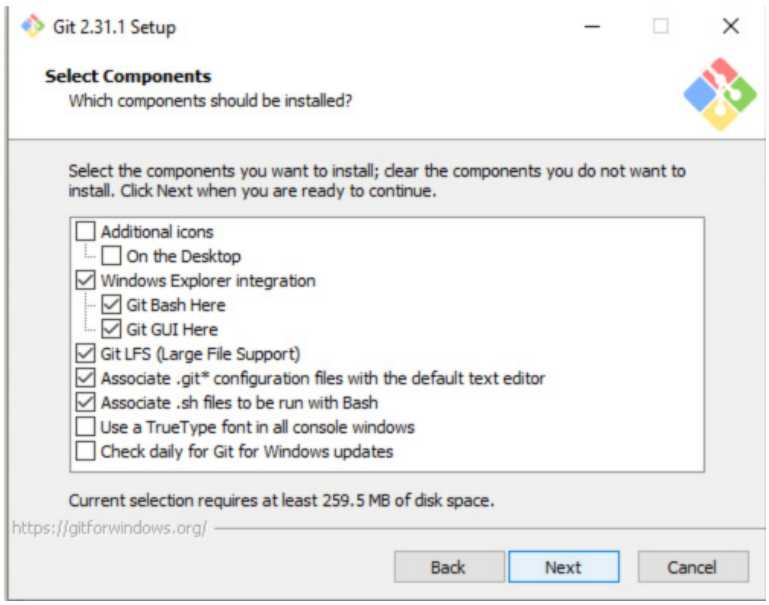

***Figure 1.25**: Git installer Windows installation (b)*

6. Now, we have updated the default from **Let Git decide** to **Override the default branch name for new repository**. This modification was made because I want my new default Git repository branch to be called main, which is the new standard. In the past, many Git repositories you encountered for your open-source contribution, were labeled as master or some other name. Also, at this point, please do not stress out by seeing the ***default*** branch, as we will discuss what they are and what purpose they serve in depth in the upcoming chapters.

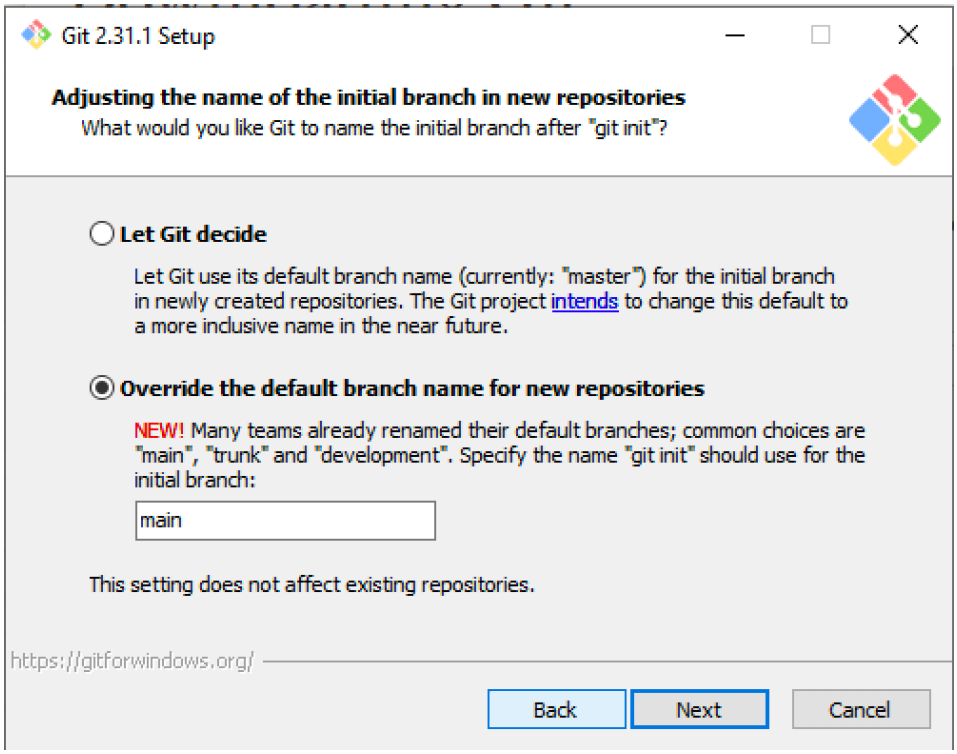

***Figure 1.26**: Git installer Windows installation (c)*

7. At this step of the installation, you will be asked for the Git HTTPS connection library. If you want to update the default, which is the OpenSSL library, to

Windows native, you can certainly do that.

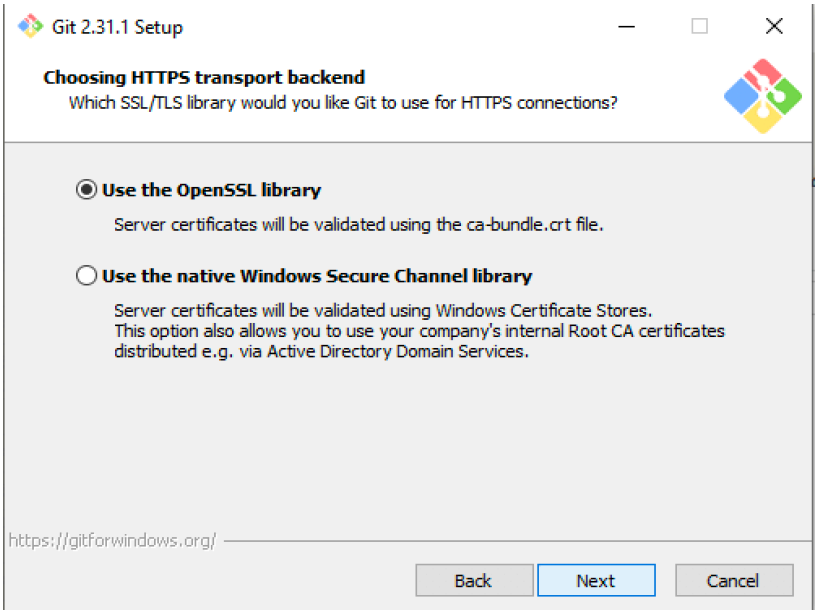

***Figure 1.27**: Git installer Windows installation (d)*

8. The installation process also checks if the user wants to install MinTTY for all the Git operations or just wants to continue with the Windows command prompt. Here, I have restarted the installation with the Windows default console, which is also more convenient for integration prospects.

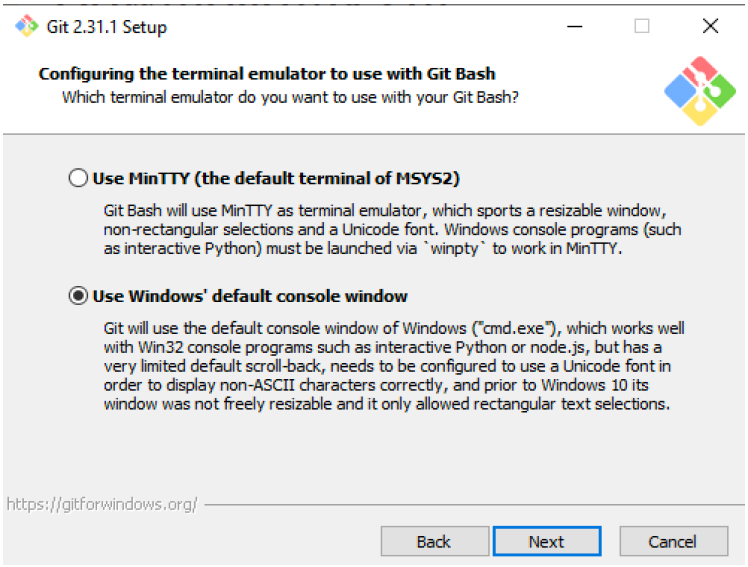

***Figure 1.28**: Git installer Windows installation €*

9. There are other installer steps, and I have continued with the installer's default settings. Once you press install, Git will start the installation in either the default directory or any specific directory you might have chosen during installation.

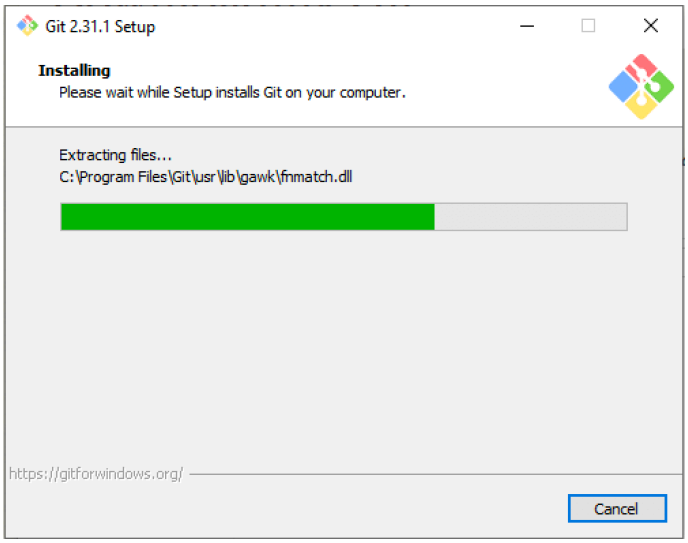

***Figure 1.29**: Git installer Windows installation (f)*

10. Once the installation completes, you can check for Git commands directly via the Windows command line as follows:
    - `git -version`
    - `Git`

    Please refer to the following screenshot:

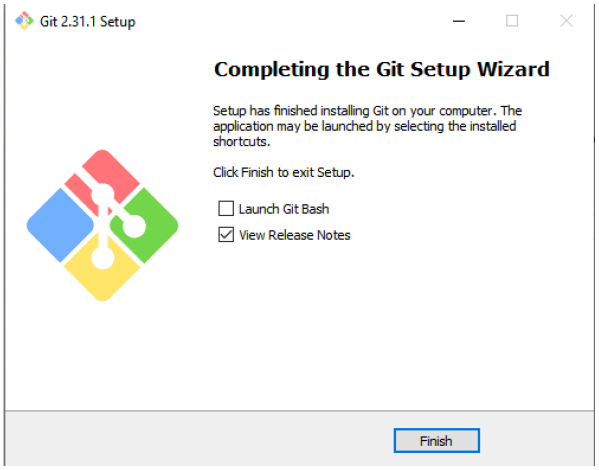

***Figure 1.30**: Git installer Windows installation (g)*

11. Once the installation is complete, the user can check if it has taken effect by running **`git version`** command. This will also help check the version of Git installed via the Windows console, as shown in the following figure:

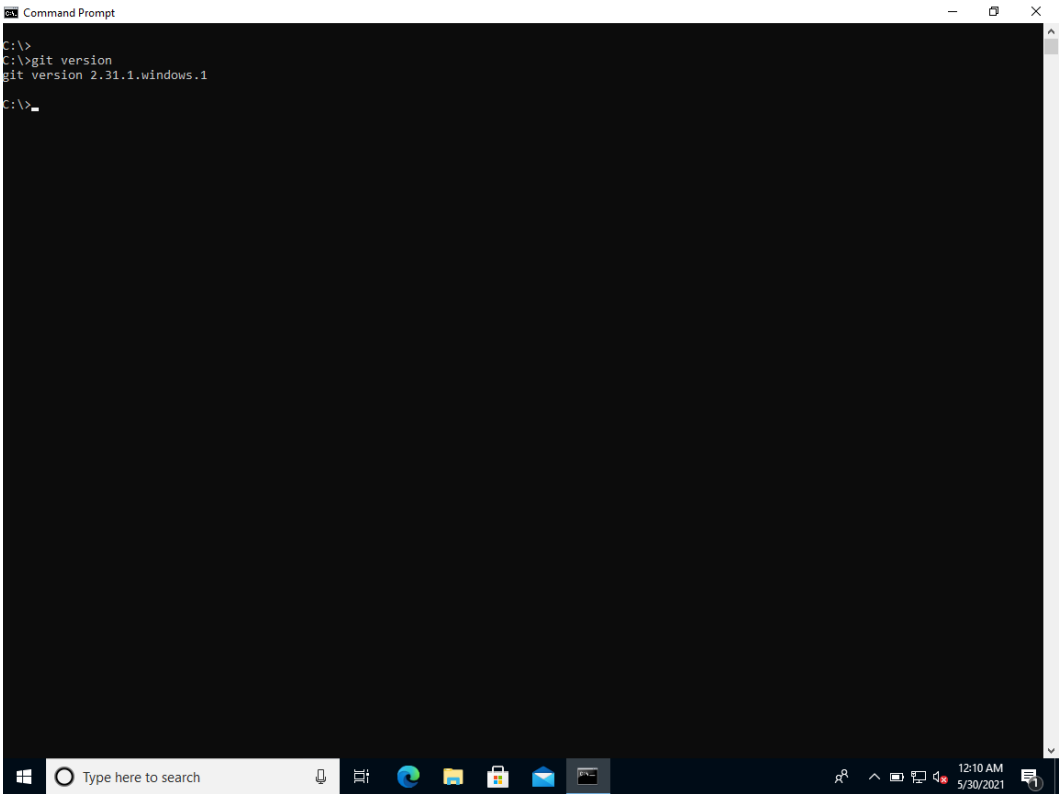

***Figure 1.31**: Git check via Windows command prompt*

12. Post successful installation of Git, if the user runs **`git`** command, it should result in supported commands as shown in the following screenshot:

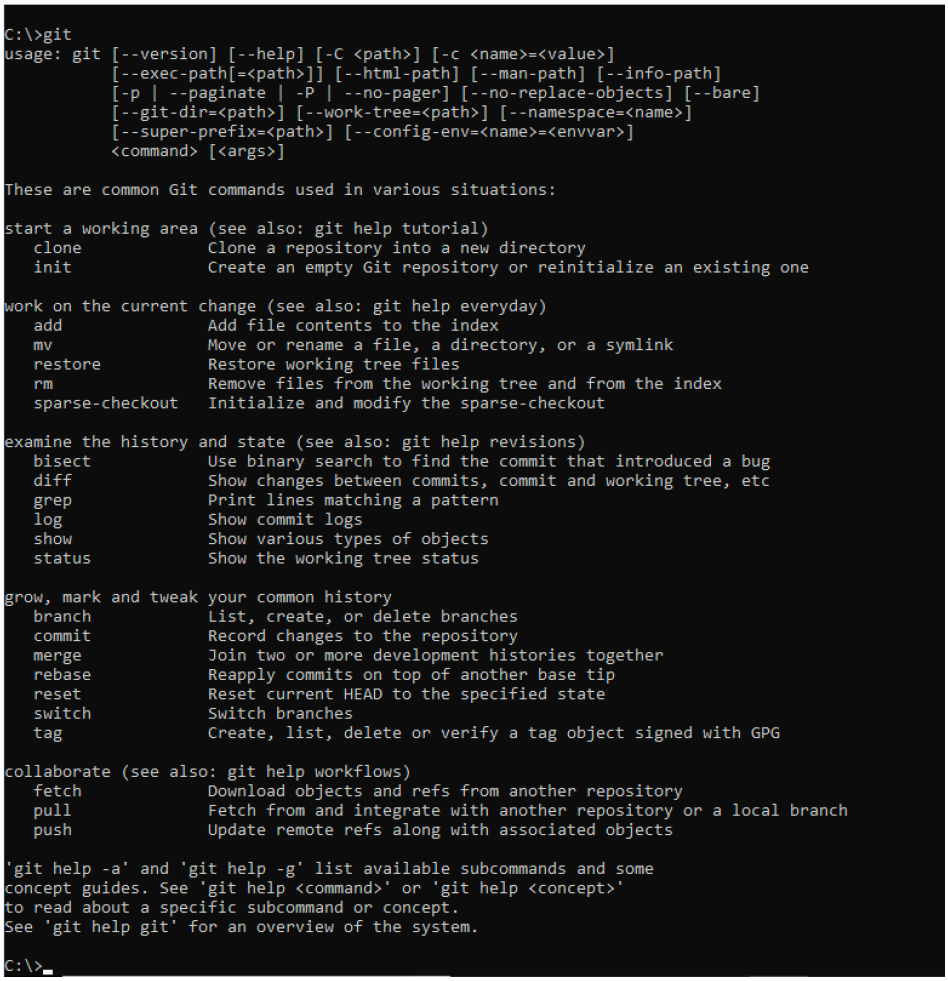

***Figure 1.32**: Git windows installation successful*

Now, you are all set to run Git and work as expected on your desired OS platform. Let us now check how you can set up an account at GitHub and then manage your Git via GitHub.

Introducing GitHub

You must understand some key underlying concepts to work efficiently and effectively with Git and GitHub. Below discussed features are the key features and most common features/terms that you will come across. Each of the discussed features has a short description and an example of they might be used in the project.

Creating and configuring the GitHub account

A GitHub account is needed to manage your code repository. Internally, it manages your repository concerning all the core functionality of your repository content using Git.

Before you can do all sorts of work/automation on your repository for your Open-source work, you need to create an account, and the following screenshot-based steps will help you create and manage your GitHub login account.

1. You can skip this section or follow along if you already have a login account over GitHub. Please visit the GitHub homepage and click on the sign-up link:

   https://github.com/

   Once you click the sign-up link, you should get a page like the following screenshot, where you are expected to enter your desired username, email, and password. There will also be a puzzle for you to solve to confirm that some automated Bot is not creating the account.

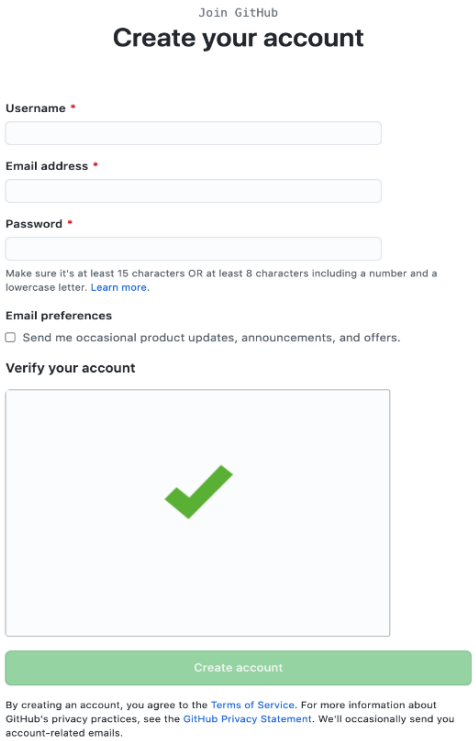

***Figure 1.33**: GitHub account creation (a)*

Fill in all the required details for creating the GitHub account, as shown in *figure 1.34*:

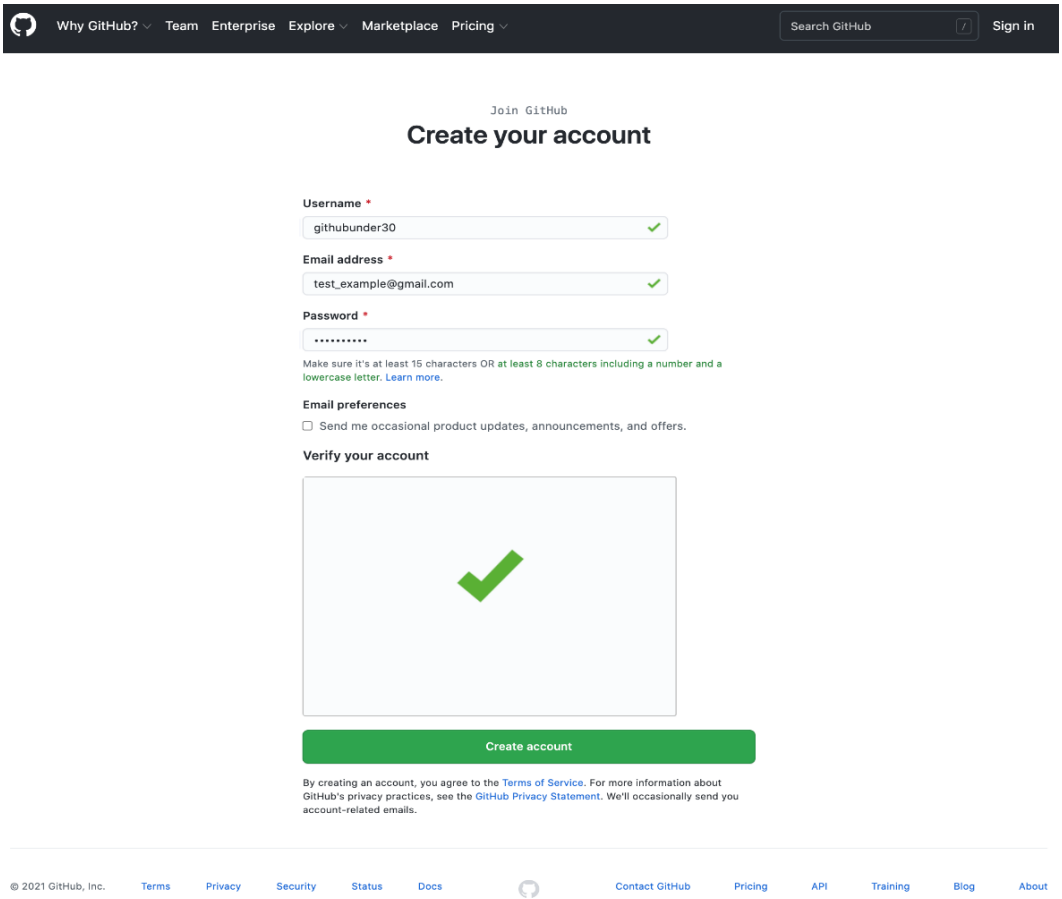

***Figure 1.34**: GitHub account creation (b)*

2. Once you have signed up successfully, you will be logged in to your account, and you should see the welcome page as follows:

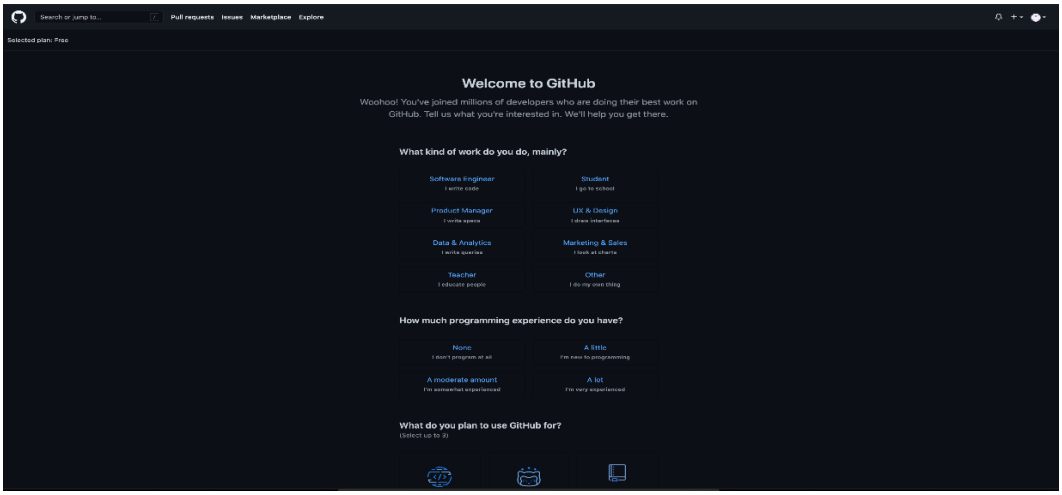

***Figure 1.35**: GitHub account creation (c)*

3. Now, traverse to the upper right corner of the page and select the setting under your accounts page. You should see the page as shown in the following screenshot and then click on **account security**. Once you click that from the left pane, your profile will appear. You should see the option for changing your existing password and enabling two-factor authentication for your account. It is advisable to enable two-factor authentication wherever the respective option is available. In case of data breach, it will keep your account doubly protected.

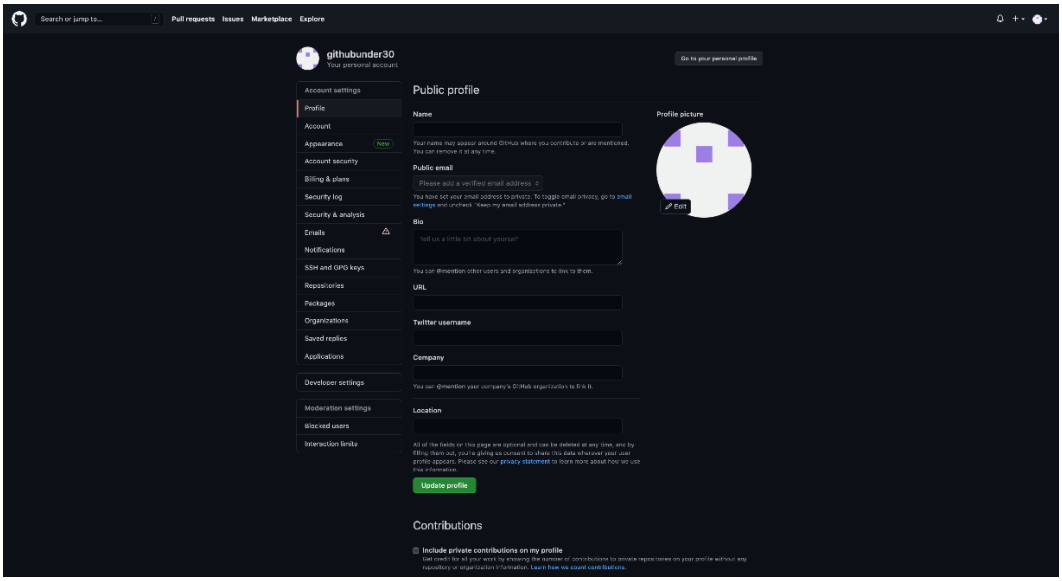

***Figure 1.36:** GitHub account creation (d)*

4. Browse to the account security tab to enable 2FA for your GitHub account, as shown in the following figure:

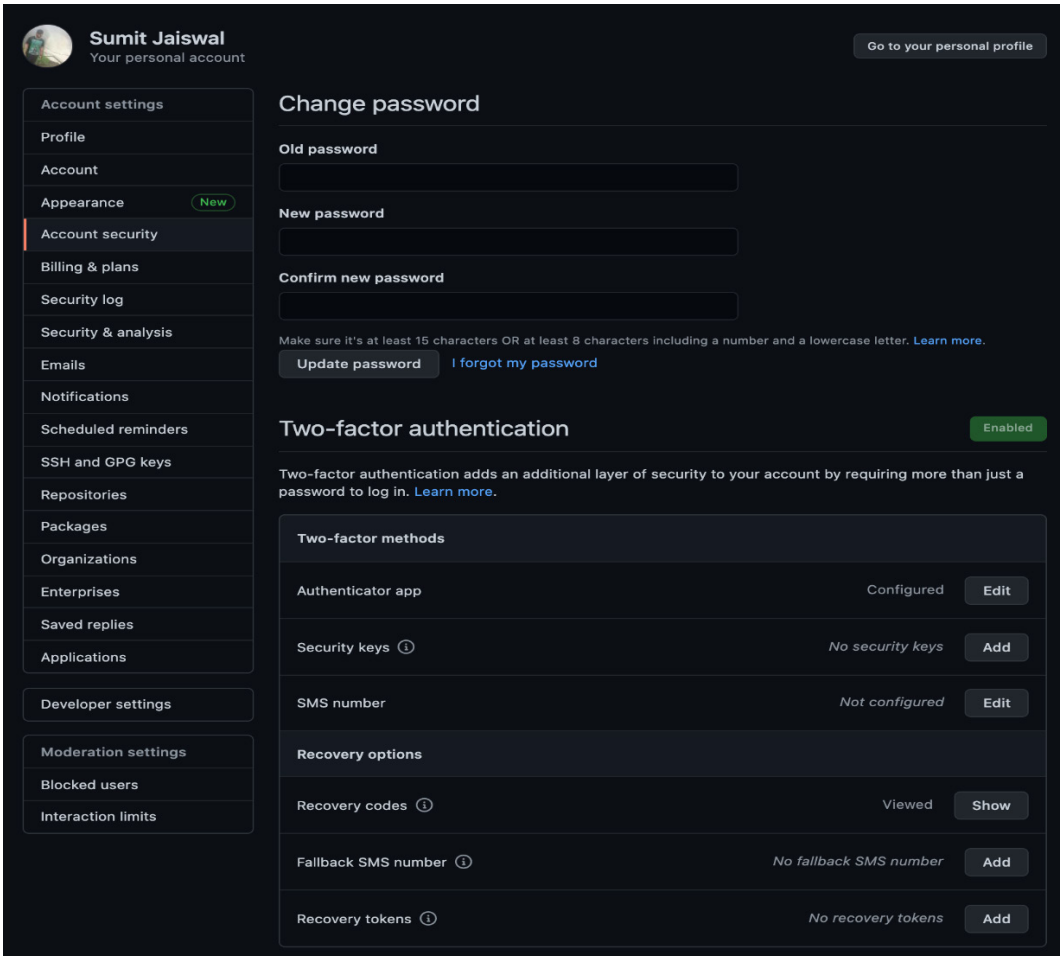

***Figure 1.37**: GitHub account creation (e)*

5. Once you enable 2FA, you must log in with your login password and authentication token.

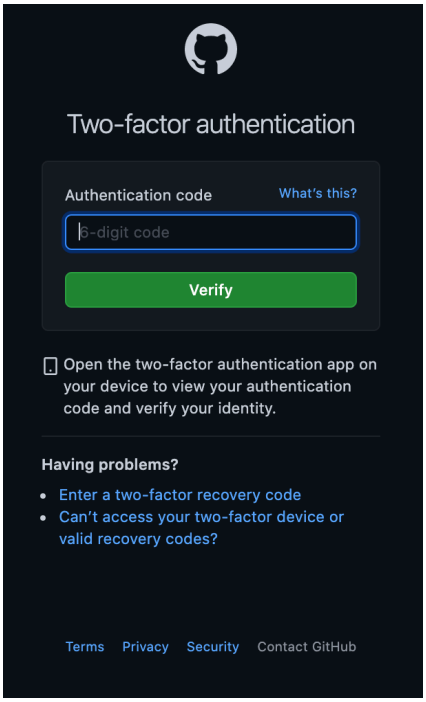

***Figure 1.38**: GitHub account creation (f)*

6. Once you enable two-factor authentication, most of the other configurations are requirement based, which you can enable/disable based on your need. The following screenshot of my account, which might seem cluttered as I have been using GitHub for more than 5 years now:

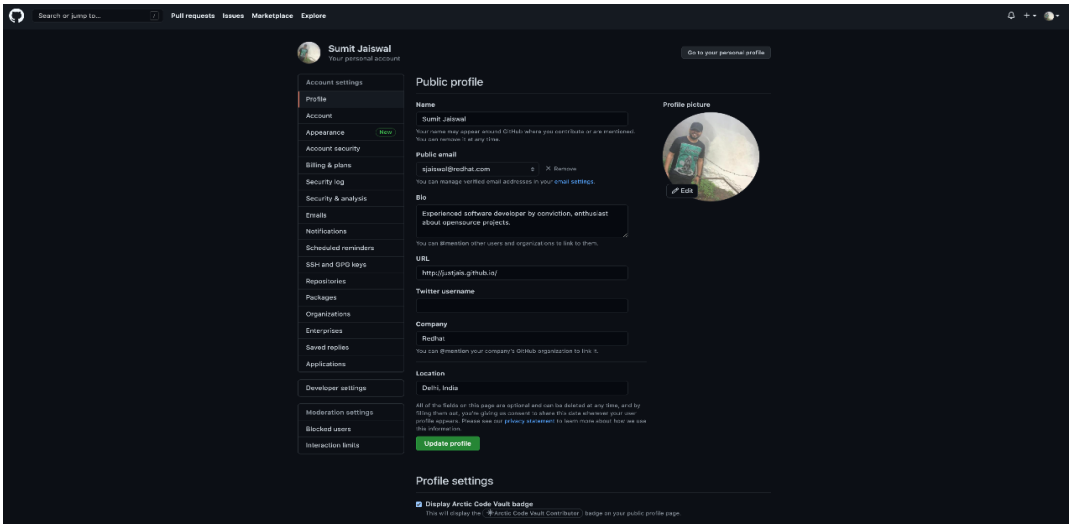

***Figure 1.39**: GitHub account (a)*

7. In the following figure, you can see the home page of the GitHub account once you login:

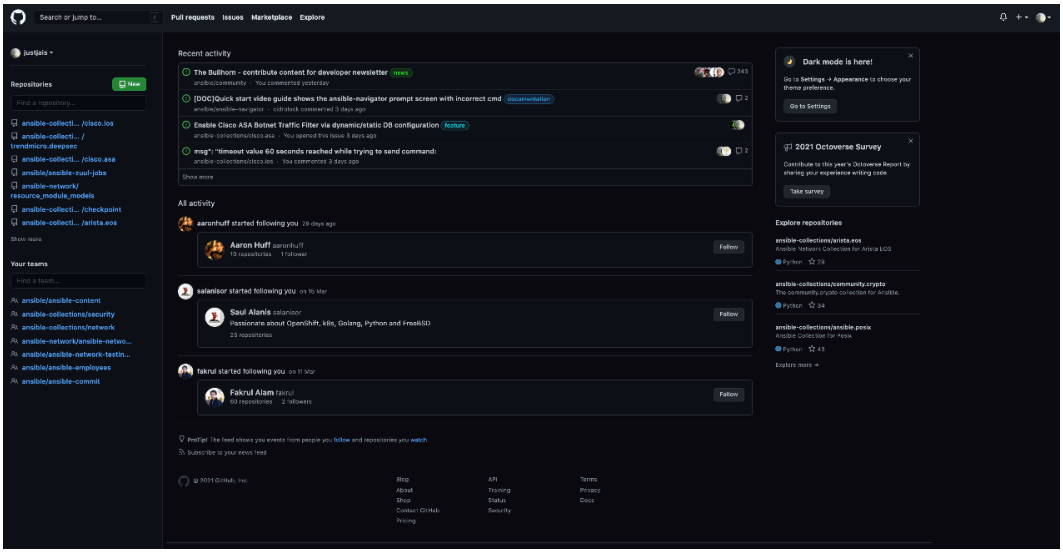

***Figure 1.40**: GitHub account (b)*

Conclusion

This is the end of this chapter, where we were able to set up our Git and our GitHub login account. In the next chapter, the stage is set for us to dive deeper into the content of Git and how GitHub encapsulates the working and functionalities of Git.

Multiple choice questions

1. **Which one of the following is true with respect to version control system?**
    a. Keeping file names and hierarchies consistent and organize
    b. Documenting changes
    c. Automatically creating a backup of your work
    d. All of the above

2. **Choose the correct statement for Git?**
    a. A commit containing one small change to a project is not practical
    b. Each version of the project is called a branch
    c. Git is a developer's IDE
    d. Git helps manage the history of the project

3. **Choose the correct statement for Git branches**
    a. By default, a commit does not belong to a branch
    b. The default branch is named "devel"
    c. The default branch is named "main"
    d. A branch contains a small part of the project

4. **Which one of the following is true with respect to Git?**
    a. Git implements distributed version control
    b. Git implements centralized version control
    c. Git does not scale large projects
    d. Git is owned by a single company

5. **Git documentation URL address?**
    a. www.github.com
    b. www.git-scm.com/
    c. www.git-scm.com/doc
    d. www.git-scm.com/downloads

6. **2FA is an acronym for?**
    a. Two-factor addresses
    b. Two-factor authentication
    c. Two-factor attributes
    d. Two-factor arms

7. **Command to check the installed version of Git?**
    a. git version
    b. git --version
    c. git -version
    d. Both a and b

8. **In Git, what is the location called where the commit history of a project is kept?**
    a. Staging area
    b. Remote repository
    c. Branch
    d. Working tree

Answers

1. **d**
2. **d**
3. **c**
4. **a**
5. **c**
6. **b**
7. **d**
8. **a**

Key terms

- Version control:
    - **Local version control systems** (**LVCS**)
    - **Centralized version control systems** (**CVS**)
    - **Distributed version control systems** (**DVCS**)
- Git 3 States:
    - Modified
    - Staged
    - Committed

Points to remember

- Git installation is supported on all the three major OS platforms macOS, windows, Linux/Unix.
- Git installation is required to make it useful as intended by GitHub, as GitHub makes use of all Git features exposed.
- You should enable GitHub 2FA for your account to be sure that your account is not compromised easily.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2

Getting Started and Understanding Git and GitHub

Now that you have Git and GitHub up and running on your system, let us understand why Git and GitHub are required, with their importance and relevance when it comes to open-source ways of development or development across teams and locations in general.

This chapter is divided into inter-related sub-topics and follows a flow that makes it easier for readers to grasp the background and concept in an informative and simpler way.

Structure

In this chapter, we will cover the following topics:

- Difference between Git and GitHub
- GitHub fundamental
- Creating a repository on GitHub
- Committing changes to your repository

Objectives

After reading this chapter, you will be able to differentiate between Git and GitHub, as both are often used in context with each other. At the end of the chapter, we will create a GitHub repository and do our 1st commit to the newly created repo.

Difference between Git and GitHub

Git and **GitHub** are two entities that work hand in hand to give way to a centralized development process. A creative illustration of the same can be seen in the following figure:

***Figure 2.1**: Git is not GitHub*

Git is a source/version control system, a version control system is software designed to keep track of all the changes made to a record over a period of time. As discussed in previous chapter, Git is a distributed version control system type, where each of the users working on a project over Git has their copy which internally has a complete history of the project and not just the current state of the records.

GitHub is a platform where you can upload a copy of your Git repo (shorthand for repository), hosted on **GitHub.com.** GitHub thus acts as a cloud-based hosting service. GitHub, along with the maintenance of your Git repo does various other tasks, which ultimately allows you to collaborate easily with other people on your projects across all geographies. It provides a centralized location to share the repository, and a web-based interface to access and operate. In the upcoming chapters, we will discuss multiple practical and functional features like forking, Pull Requests, Issues, Projects, and GitHub Wikis. These features allow you to specify, discuss, and review changes with your team efficiently and effectively. The Git logo can be seen in the following figure:

***Figure 2.2**: Git logo*

There are numerous benefits of using Git, even when users are working independently and editing text files. The benefits are as follows:

- Undo changes

  One of the most relevant features is the ability to undo the change, which means if a user makes any mistake, they can revert the changes to a previous point in time and recover their past version of their work.

- Complete history of the changes

  If the user wants to verify the state of the project in a past time like a day, week, month, or even a year ago, the user can go ahead and check out any previous version of the project to verify exactly the state of project files at that point in time.

- Documenting the changes made

  For real-time project scenarios where different people collaborate, it becomes hard to track why a particular change was made. This issue is resolved by having commit messages in Git, which makes it easy to document for future reference.

- Change anything or everything

  Git gives the user the power to recover a previous version of the project, and as the process is straightforward and easy, the user gets the confidence to make any changes that the user wants. Users can always revert to an earlier work version if the changes do not work out.

- Multiple streams of history

  Users can maintain multiple branches of history to experiment with changes in their work content or build different features independently. Once the changes are finalized, they can be merged back into the main branch or deleted if the changes do not work out.

If you are working with a larger team, users get added advantages and a wider range of benefits when using Git to keep track of their project-related changes. Some of the key advantages of Git when working with a team are as follows:

- Ability to resolve conflicts

  With Git, multiple people working on the project can collaborate and work on the same file simultaneously. In most cases, Git can automatically merge the changes from different team members. Sometimes if it cannot, Git will display all the user conflicts. Then, the user can resolve the conflicts and merge the intended changes.

- Independent streams of history

  As different people work and collaborate on the project, each user can work on different branches, which allows other users to work independently on separate features. When the changes are ready, they can be merged without any issues.

***Figure 2.3**: GitHub logo*

GitHub is significantly more than simply a spot to store your Git repositories. It gives multiple advantages, including the ability to do the following:

- Document requirements

  GitHub provides multiple ways to document the team roadmaps where the team can report bugs or track upcoming features that team members are supposed to work on.

- Collaborate on independent streams of history

  GitHub branches and pull requests help the team to collaborate and work on different branches and features.

- Review process

  Users can get to the list of the **pull request** (**PR**). They can track all the different features currently worked on by the team. By clicking any available

pull requests, they can easily track the latest progress and all the discussions about the PR changes. A team can have their **Continuous Integration** (**CI**) server in place, which can run the tests for the changes and help the reviewer approve them before they are accepted and merged into the main repository.

- Track team progress

  Traversing through the pull requests commit history allows the team to see what the team has been working on and its progress.

GitHub fundamental

Users must first understand fundamental concepts to work efficiently and effectively with Git and GitHub. The key features and most common features/terms that you will come across are discussed below; each of these have a short description and an example of how they might be used in the project.

- Commit

  At whatever point users save their work changes in either one or more files, users can create a new commit in Git. A commit is, therefore, a snapshot of the user's entire repository, not just of one or two files. So more commonly, after the user has changed files, the user will want to update the repository by taking a new snapshot. *Example usage: "We should submit respective progressions and finally push them up to GitHub."*

- Commit message

  Whenever a user makes a commit to the PR, a user is expected to make the commit with a commit message, and this commit message might seem like an overhead if you are starting anew. Still, it is invaluable information when it comes to tracking the commit history and checking the reason for a particular commit to the branch. *Example usage: "Committing the change to fix the recursive base condition to avoid infinite loop error. "*

- Branch

  A branch is an autonomous series of commits out of the way that a user can use to evaluate an analysis or work on creating a new feature. *Example usage: "It's better to create a branch to implement the idea of new search functionality."*

- *main* branch

  When a user creates a new Git project, there's a default branch that gets created which is called main. This branch is the parent branch where finally all the commit land, and once the changes are finalized and verified, the main branch from where changes are pushed to production. *Example usage: "It's highly recommended that user never makes commit directly to main."*

- Feature (or topic) branch

  When users start working on new functionality, users should create a new branch called a feature or a topic branch. *Example usage: "We have got too many feature branches; we should now focus on getting one or two of these finished and into production."*

- Release branch

  For a project where the team must maintain multiple software versions, in turn, supporting old, released versions for customers. A release branch is needed, which would be the place where all required fixes and updates will get merged. Technically, there is no difference between a feature and a release branch, but the segregation is useful when the project is discussed within the team. *Example usage: "We need to release a patch to fix the security bug on all of the supported release branches."*

- Merge

  A merger is the final push of the completed work from one branch to the main or different branch to incorporate changes into the respective branch. Most of the time, you will merge the content from your feature/topic branch to your main branch. *Example usage: "Awesome work on the 'security' feature. Could you merge it into the main so that we can push it to the production?"*

- Tag

  A specific tag is a reference point to a specific historic commit. Tags are most used to tag a particular production release, letting the team/users know which versions of the code went to production with the particular tagged release. *Example usage: "Let us tag this release as 1.0 and push it to production."*

- Checkout

  Checking out simply means that the user is trying to get to a different version of the project to check and verify the file's state at the time of commit. Usually, users will check out from a branch to verify the work done, which means that anything committed in Git can also be checked out. *Example usage: "I need to check out the last released tag. As there's a bug in production that I need to replicate and fix."*

- Pull request

  Initially, a PR meant that someone from the team or community would assess the work and fix that your PR was supposed to resolve, and then provide feedback before it would be merged into the corresponding project main branch.

However, pull requests are now commonly utilized before the above process and can either discuss a potential feature or the actual fix. *"You should go ahead and write a new pull request for the voting feature; this will allow the team and the community to provide comments on the PR feature."*

- Issue

  An issue, as the name suggests, is the GitHub feature used to discuss features, track bugs, or both in the project. *Example usage: "the login feature is not working as intended over supported mobile platforms. Please go ahead and create an issue on GitHub documenting the steps to reproduce the bug?"*

- Clone

  Frequently, you will need to download a duplicate of the venture from GitHub to deal with it locally. The way towards duplicating the GitHub repository to your local PC is called cloning. *Example usage: "Please clone the GitHub repository, fix the bug, and then push the fix back up to GitHub repo."*

- Fork

  Now and then, you do not have the necessary consent and permission to make changes straightforwardly to the project. Also, maybe it is an open-source project composed of individuals where you do not have a clue, or a task composed by one more gathering at your organization that you do not work with a lot. On the off chance that you need to submit changes to such a project, first, you need to duplicate the project under your client account on GitHub. That cycle is called forking the GitHub repository. You could then clone it, make changes, and submit them back to the project using a pull request. *Example usage: "It'd be lovely to see how you'd rewrite the home page marketing copy. You can fork the project* GitHub *repository and submit a pull request with your proposed changes."*

At this point, going through all this terminology might seem a bit overwhelming but do not worry! All this worry would start to fade when you start working on some real projects, and this would start making more sense, and you will be comfortable using the above-discussed terminology and its use case. In the upcoming chapter, we will go through the various elements you might have seen/observed in a GitHub project and how to use the discussed features to get a sense of progress on a project.

Creating a repository on GitHub

Before we can start playing around with Git and operate the same via GitHub, we need to create a repository. You need to first make it on GitHub, and as you have already created an account over GitHub, creating the GitHub repo using the account

is a few steps task. It is a step-by-step process which is listed as follows, along with screenshots for you to follow along in an easier and more precise manner:

1. First, log in to your GitHub account (**https://github.com/login**), and once you are logged in, you will be displayed with the GitHub home page. If you have recently created your account, it should look similar to the following screenshot.

   Despite that, if you log in to your GitHub account, the layout should be almost similar.

   To create a new repository, tap the **Create repository** button from the GitHub home page.

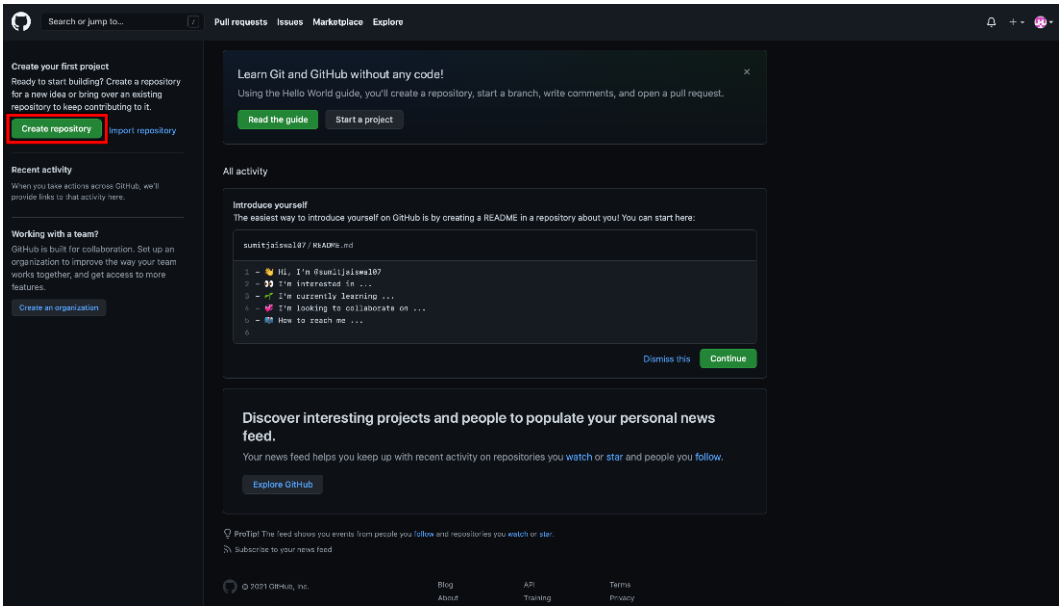

***Figure 2.4**: To create New GitHub repo - 1*

2. You can also create a new repository using the '+' on the upper right-hand side and then clicking on the new repository, as shown in the following screenshot:

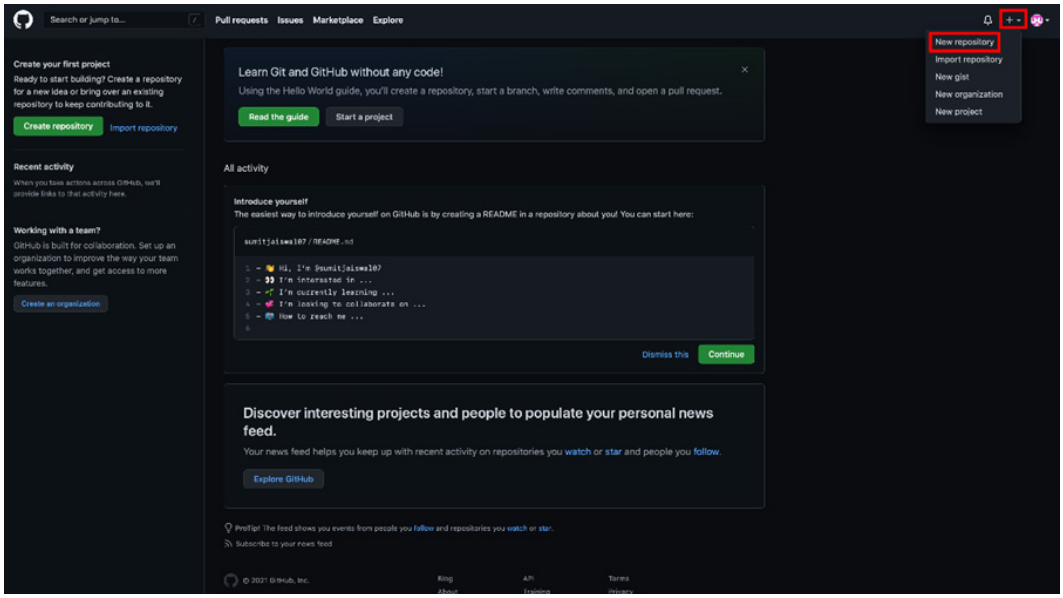

***Figure 2.5**: To create New GitHub repo – 2*

3. On the new repository, you will see a few options that you can select based on your project's needs.

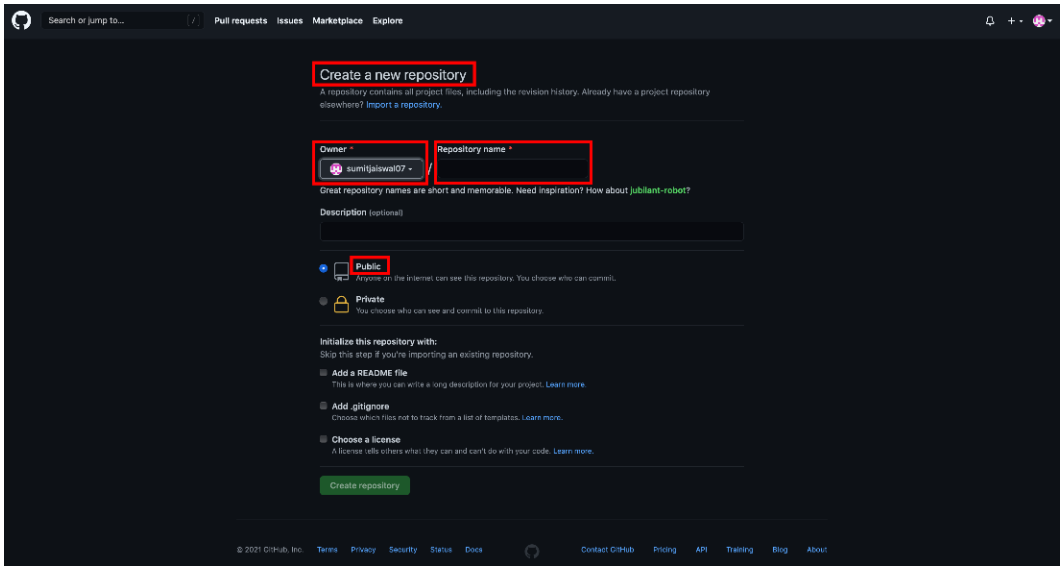

***Figure 2.6**: To create New GitHub repo - 3*

4. Each of the options available on the "**Create a new repository"** page has its own purpose, as discussed as follows:

a. **Repository name**: This will be the name of the GitHub repository, and once created, you cannot rename the respective repo to something else, so be careful while naming your GitHub repository.

b. Type of GitHub repository, it can either be a **Public** or **Private** repo. As the name suggests, if you work with an open-source project, you will go with Public. If you want the GitHub repo to not be open for the world to access, you can make your GitHub repo Private and make the project part of your organization. Only people who are a part of your organization or to whom you, as a project administrator, give access, can access the project. For the rest of the world, the project and its content will not be accessible.

c. **Readme file**: This is a very basic option if you select GitHub. While creating your repository, you would also create a dummy Readme file with the GitHub project name and can update the file's content later when you start working on your project. This is optional, as you can also check in your Readme file later after the project is created with all the required content.

d. **.Gitignore**: If you wish to configure Git to ignore files and not check in to the GitHub repository, pick this option. When you commit, .Gitignore informs GIT which files and directories to ignore.

e. **License**: Public repositories on GitHub are often used to share open-source software. For your repository to truly be open source, you must license it so that other people are free to use, change, and distribute the software. There are various license types available, and you can choose one based on your requirement. To read further about the available licenses, refer to this link:

https://docs.GitHub.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository# searching-GitHub-by-license-type

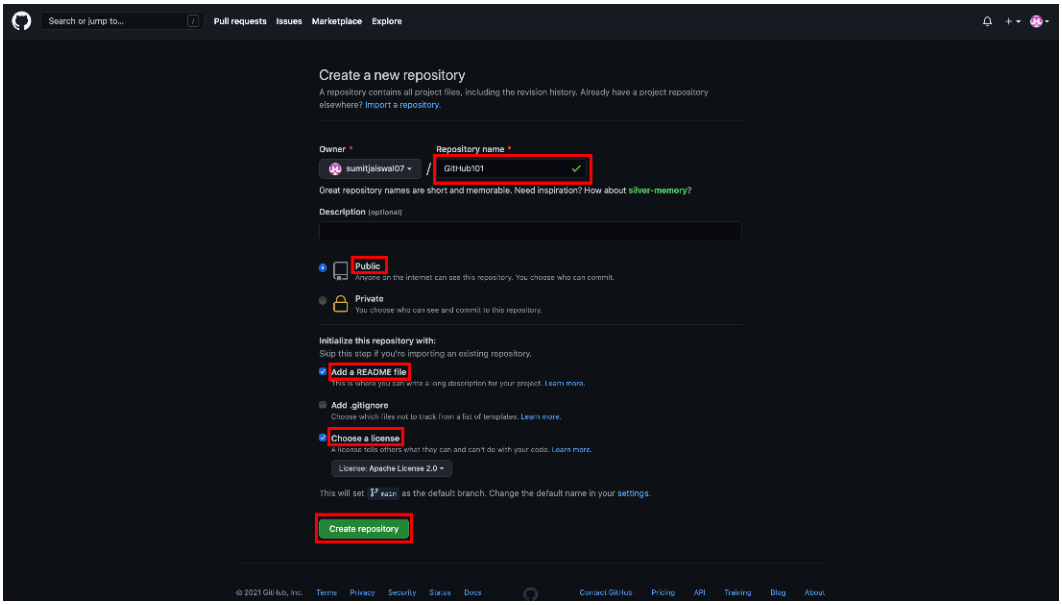

***Figure 2.7**: Create a new GitHub repository page*

5. Once you fill the above-discussed field and press **Create repository**, GitHub will create a new project repository under your GitHub. Now that GitHub has created a new repository for your project with the given name, you can start working on your project and start pushing content to your project.

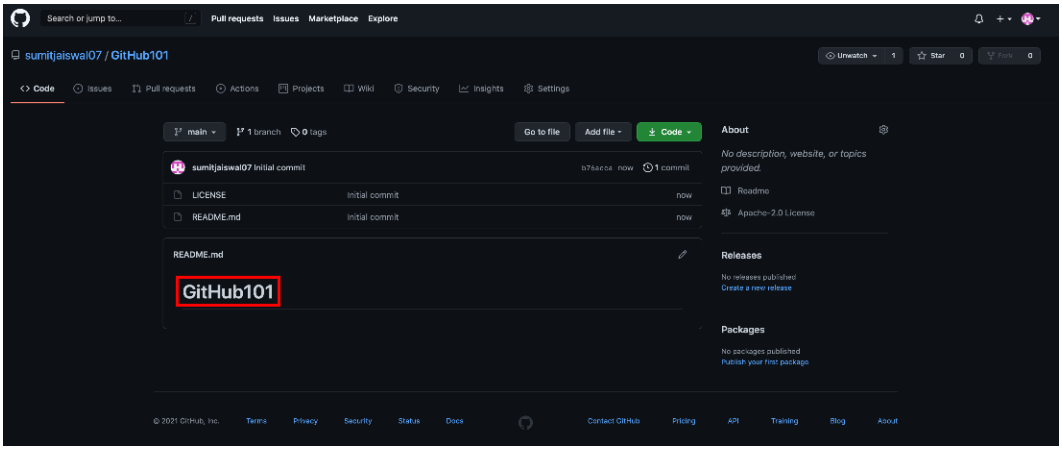

***Figure 2.8**: Newly created GitHub repository page*

The above-discussed steps will help you create your own GitHub repo. Once created, you can explore all the available options and tabs that GitHub gives you with the project. Also, we will discuss the project's available tabs and its available options in the upcoming chapters.

Committing changes to your repository

As the new GitHub repository is in place now, let us check how you can make commit changes to your newly created repository directly via GitHub **Graphical User Interface** (**GUI**) without involving any command line or firing any of the available Git commands.

1. Currently, the newly created GitHub repository GitHub101 does not have any content, and the license and the readme files are generated by the GitHub engine while creating a new GitHub repository. Now, let us try to update the README.md file content directly from GitHub **User Interface** (**UI**). For that, you only need to click the edit (pencil) button available on the right-hand side of the Readme file, as shown in the following screenshot:

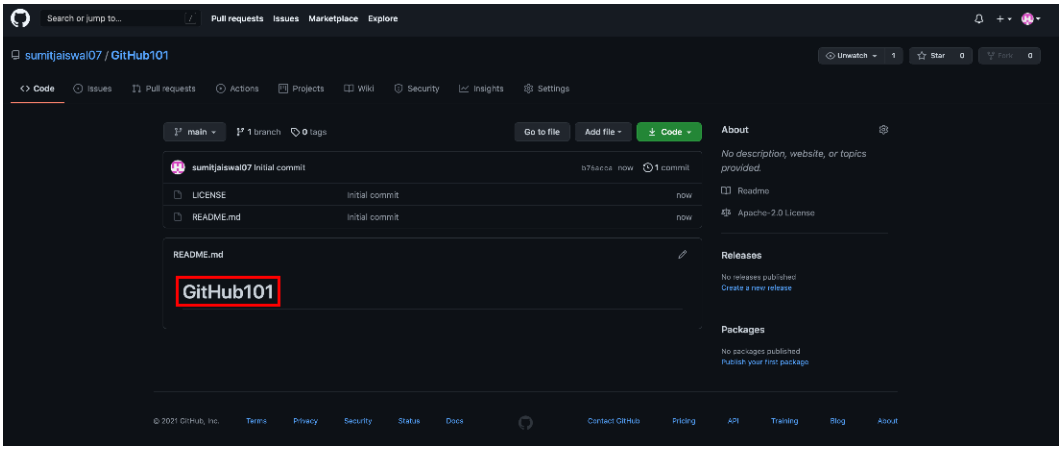

***Figure 2.9**: GitHub101 project page*

2. With the edit page opened for GitHub101, I have added a short description to the **`README.md`** file ***"- This is TEST Description"*** in addition to the # GitHub101 heading added by GitHub when the project was created, as shown in the following screenshot:

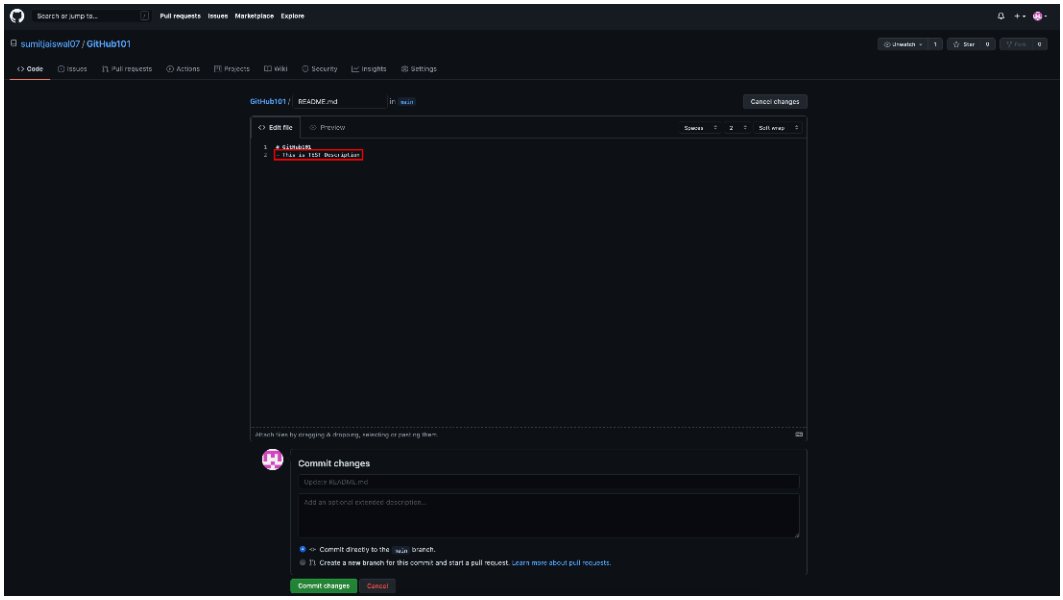

***Figure 2.10**: Edit README.md for the GitHub101 project*

3. Using the GitHub preview tab, you can verify the changes, and if the changes you have made are intended and as expected, you can push the changes to take effect on your GitHub repository. You can see in the screenshot below *(Figure 2.11)* how the changes made to the readme file are rendered and how they will look when the changes take effect.

   As I am satisfied and have verified the intended changes, I can push/commit the changes directly to my GitHub101 repo **main** branch or create a new branch for the change and create a **Pull request** (**PR**). You can merge the PR once someone reviews and approves the changes. This is the preferred/advised way to make changes to your README. But, at this point, since we have verified the intended changes, we can go ahead and commit the

changes directly, and once done, respective changes would take effect to the GitHub101 repo, as shown in the following screenshot:

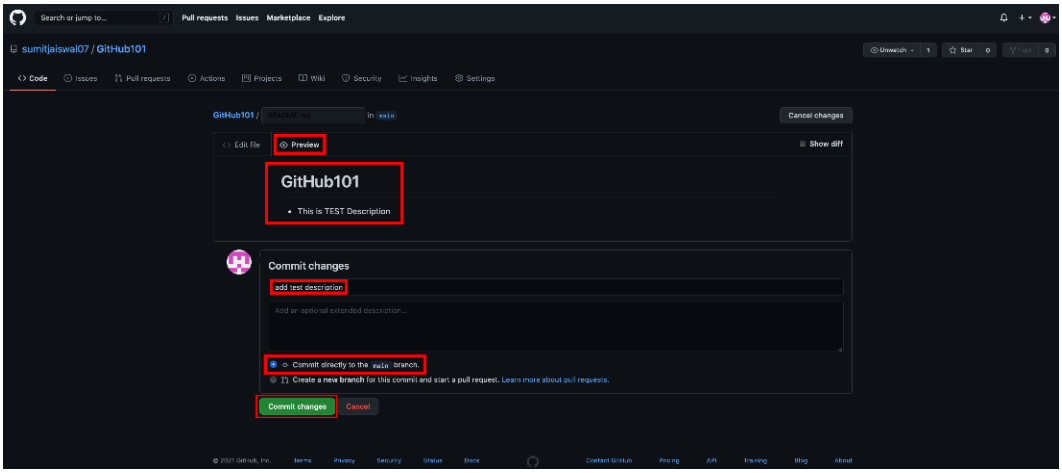

***Figure 2.11**: Edit the readme file and commit changes*

4. The Committed description change in the Readme file has now taken effect and is displayed as updated in the following screenshot:

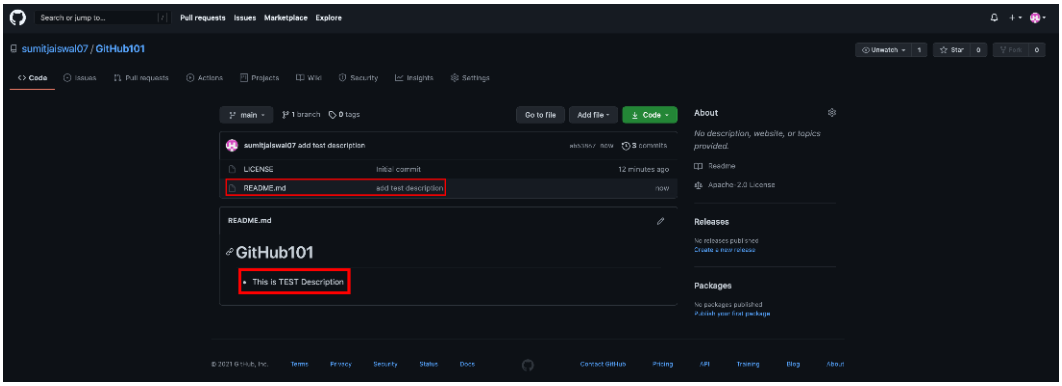

***Figure 2.12**: GitHub101 repo with updated Readme description*

Going by the process of committing changes to the GitHub repository, now you should have gained a fair understanding of GitHub's new repository and how to maintain it.

Conclusion

This is the end of this chapter where we were able to get things started with Git and GitHub and have fun creating a new GitHub repository and make changes to the newly created GitHub repository.

In the next chapter, the stage is set for us to dive deeper into the content of Git and how GitHub encapsulates the working and functionalities of Git.

Multiple choice questions

1. **Git remote repository is offline. In that reference which one of the following is true?**
    a. You cannot continue an offline project, and you'll need to start over
    b. You must wait for the remote repository to become available.
    c. You can continue to work, but only with the project's current version.
    d. You can continue to work with the local repository.

2. **Which is true for GitHub?**
    a. Version control system
    b. Cloud-based hosting service
    c. Keep track of the changes in the project over a period of time
    d. Helps to maintain Git repo and in collaborating amongst teams in different geographies

Answers

1. **c**
2. **d**

Key terms

- GitHub commonly used terms:
    - Commit
    - Commit message
    - Branch
    - *main* branch
    - Feature (or topic) branch
    - Release branch
    - Merge
    - Tag
    - Checkout

    - Pull request
    - Issue
    - Wiki
    - Clone
    - Fork
- Commonly used Acronyms
    - GH – GitHub
    - User Interface (UI)
    - Graphical User Interface (GUI)

Points to remember

- Git is a version control system of type distributed version control system.
- Git and GitHub are two entities that work hand in hand to give way to a centralized development process.
- The default branch of a GitHub repo is *"main,"* and all the Pull request/PR that gets merged to the GitHub repository gets merged to the *"main"* branch.

Further reading

- For more history and reference around the discussed topics in this chapter, you can check out the Git official documentation for getting started and GitHub documentation:
    - Getting started with Git: **https://Git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
    - Getting started with GitHub: **https://docs.GitHub.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 3

Git Branching, Merging, and Rebasing

Now that you have Git and GitHub installed, let us explore why they are required, as well as their value and relevance when it comes to open-source software development or software application development across teams and regions in general.

This chapter is organized into interconnected sub-topics and follows a flow that allows readers to grasp the background and concept in an informative and simplified manner.

This chapter is about the core functionality of Git and GitHub and how one complements the other in the software development life cycle and DevOps.

Structure

In this chapter, we will cover the following topics:

- Introducing Git options
- Git commands
    - To start a working area
    - To work on the current change
    - To examine the history and state of the repository

- To grow, mark and tweak your repo history
- To collaborate over repository

Objectives

The key concepts discussed in the chapter lets you use Git and its available commands for regular day-to-day Git operations and processes. Git commands that you will come across in this chapter will serve as the building blocks and lay the foundation for your learning and working with Git as your version source control.

This chapter assumes you are up and running with Git on your Linux, Windows, or Mac machines.

Introducing Git options

This section of the chapter deals with all and everything about Git.

Git is a fast, scalable, distributed revision control system with an extensive command set that allows high-level operations and full access to the internals.

Now that you have a basic understanding of how Git works and its usefulness when it comes to the version control system, it is time to dive deep into the Git operations and commands that makes Git a prominent tool when it comes to the software development lifecycle.

Git options

Multiple command options are available with Git. Just fire the **`git`** command that would result in the **`git`** command options as:

```
→  ~ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

***Figure 3.1**: Git options*

We will first go through the Git options commands and then discuss Git's remaining output later in the chapter. Git options are as follows:

- **`-version`**: This git command outputs the Git version that is currently installed on the machine.

- **`--help`**

  This git command outputs an overview and a list of the most frequently used commands. All the available commands are printed if the option —all or -a is used.

  If a Git command is run with **`--help`**, it will result in the respective Git command's manual page.

- **-C <path>**

  This option lets the user run the Git command from the specified <path> instead of the current working directory. When several –C <path> options are specified, each non-absolute –C <path> is evaluated in relation to the –C <path> before it. If the Git option is present but empty, for example, `-C ""`, the current working directory remains unchanged.

  This option affects the options that expect path names, such as **`–git-dir`** and **`–work-tree`**, in that the path names are interpreted relative to the working directory because of the **`-C`** option.

  The following invocations, for example, are equivalent:

```
git --git-dir=a.git --work-tree=b -C c status
git --git-dir=c/a.git --work-tree=c/b status
```

Figure 3.2: Git path specifier

- **`-c <name>=<value>`**

  Give the command a configuration parameter. Values from configuration files will be overridden by the value provided.

  The **`<name>`** must follow the format specified by Git config (subkeys separated by dots).

  NOTE: If the user omits = From the option command as git -c foo.bar ...

  It's an allowed operation, and Git in such case, sets **`foo.bar`** to the Boolean true value. But, in case of empty value like **`git -c foo.bar=`** ... the **`foo.bar`** will be set to an empty string where Git will convert it too false.

- **`--exec-path[=<path>]`**

  Wherever your core Git programs are installed is the path. The Git EXEC PATH environment option can also be used to regulate this.

  If no path is specified, Git will display the current configuration before exiting.

- **`--html-path`**

  This Git command will exit after printing the path where Git's HTML documentation is installed, without the trailing slash.

- **`--man-path`**

  This Git option command results into the "manpath" for the man pages in the version of Git installed on the machine.

- **`--info-path`**

  This command results into the path where the Information files document the version of Git installed on the machine and exits.

- **`-p | --paginate`**

  This will pipe all output into less (or if set, $PAGER) if standard output is a terminal. This overrides the pager.<cmd> configuration options (see the *Configuration Mechanism* section below).

  If you want to Pipe all the output into less, the standard output is terminal.

- **`-P | --no-pager`**

  This tells Git to not pipe the output into a pager.

- **`--no-replace-objects`**

  This tells Git to not use replacement reference to replace Git objects.

- **`--bare`**

  This will treat the given Git repository as Bare repository and if **`GIT_DIR`** env is not set, it will be set to the current working directory.

- **`--git-dir=<path>`**

  Set the repository's path (**`".git"`** directory). The **`GIT_DIR`** environment variable can also be used to regulate this. It can be a relative or absolute path to the current working directory.

  Specifying the location of the **`". git"`** directory using this option (or **`GIT_DIR`** environment variable) turns off the repository discovery that tries to find a directory with "**`. git`**" subdirectory (which is how the repository and the top-level of the working tree are discovered) and tells Git that you are at the top level of the working tree. If you are not at the top-level directory of the working tree, you should tell Git where the top-level of the working tree is, with the **`--work-tree=<path>`** option (or **`GIT_WORK_TREE`** environment variable).

If you just want to run Git as if it was started in <path> then use **`git -C <path>`**.

- **`--work-tree=<path>`**

  Create a shortcut to the working tree. It can be a relative or an absolute path to the current working directory. The **`GIT_WORK_TREE`** environment variable, as well as the core, can be used to control this.

  It is a variable for work-tree configuration.

- **`--namespace=<path>`**

  This is equivalent to setting **`GIT_NAMESPACE`** environment variable

  There are few other Git options that are available, and you can read more about it in the Git's official document page: **https://git-scm.com/docs/git#_options**

Git commands

The following section of this chapter deals with various Git commands that you will come across during your day-to-day Git operations and working. Each of these commands are categorized based on its working as follows:

- To start a working area:
  - **init**: Creating an empty Git repository or reinitialize an existing one.
  - **clone**: Cloning a repository into a new directory.
- To work on the current change:
  - **`add`**: Adding file contents to the index
  - **`mv`**: Moving or renaming a file, a directory, or a symlink
  - **`restore`**: Restoring working tree files
  - **`rm`**: Removing files from the working tree and the index
  - **`sparse-checkout`**: Initializing and modifing the sparse-checkout
- To examine the history and state of the repository:
  - **`bisect`**: Using binary search to find the commit that introduced a bug
  - **`diff`**: Showing changes between commits, commit and working tree, etc.
  - **`grep`**: Printing lines matching a pattern

    - **`log`**: Showing commit logs
    - **`show`**: Showing various types of objects
    - **`status`**: Showing the working tree status
- To grow, mark and tweak your repo history:
    - **`branch`**: List, create, or delete branches
    - **`commit`**: Recording changes to the repository
    - **`merge`**: Joining two or more development histories together
    - **`rebase`**: Reapplying commits on top of another base tip
    - **`reset`**: Reset current HEAD to the specified state
    - **`switch`**: Switching branches
    - **`tag`**: Create, list, delete or verify a tag object signed with GPG
- To collaborate over repository:
    - **`fetch`**: Downloading objects and refs from another repository
    - **`pull`**: Fetching from and integrating with another repository or a local branch
    - **`push`**: Updating remote refs along with associated objects

This seems to be too clustered information, but need not get worried as these are the commands which you will find most familiar by the end of this book.

We will discuss and go over each of these commands individually and, go over the commands in different use-case scenarios.

Let us now discuss each of the Git commands one-by-one to get a better understanding regarding the working and importance of the respective commands:

Starting a working area

As the name suggests, this Git command section will walk you through the process of setting up a Git repository for a new or existing project.

This guide assumes a basic understanding of a command-line interface, and will cover the following high-level topics:

- Creating a new Git repository
- Cloning a pre-existing Git repository
- Adding a modified version of a file to the repository
- Setting up a Git repository for remote collaboration

What is a Git repository?

A Git repository is a virtual location for your project's files.

It allows you to save versions of your code that you can access whenever you want.

Git init - Initialize Git repository

The ***git init*** command initiates the creation of a new Git repository. It can be used to convert an un-versioned project to a Git repository or create a new, empty repository. Since most other Git commands are inaccessible outside an initialized repository, this is typically the first command you will run in a new project.

This will also result in the creation of a new main branch.

When you run **`git init`** for the first time, it creates a **`.git`** subdirectory in your current working directory that will contain all the Git metadata with respect to the newly created repository.

Subdirectories for objects, references, and template files are included in this metadata. In addition, a HEAD file is created, which points to the currently checked-out commit.

Sometimes there might be projects or user's need where **`.git`** subdirectory needs to be outside your current working directory. In that case, you can mention the custom path either by setting the **`$GIT_DIR`** under your environment or else, pass **`--separate-git-dir`** along with the custom path.

Git Init usage:

`git init` command is the most straightforward way of setting up your repository when compared to other version-controlled systems. There is no explicit need to generate your project repository, input files, and so on. There are two ways in which you can initialize your directory:

1. To get a working Git repository, simply cd into your project subdirectory and run the **`git init`** command in your terminal.

```
~ mkdir git_repo/
~
~ ls -l
total 4
drwxrwxr-x 2 ubuntu ubuntu 4096 Jul 13 11:42 git_repo
~
~ cd git_repo/
~
~ git init
Initialized empty Git repository in /home/ubuntu/git_repo/.git/
~ ls -la
total 12
drwxrwxr-x 3 ubuntu ubuntu 4096 Jul 13 11:43 .
drwxr-xr-x 5 ubuntu ubuntu 4096 Jul 13 11:42 ..
drwxrwxr-x 7 ubuntu ubuntu 4096 Jul 13 11:43 .git
```

***Figure 3.3**: Git init working – option 1*

2. To keep track of project changes, convert the directory into a Git repository. To generate a new **.git** subdirectory, create a new Git repository in a specific directory.

```
~ ls -l
total 0
~
~ git init git_repo
Initialized empty Git repository in /home/ubuntu/git_repo/.git/
~
~ ls -la git_repo
total 12
drwxrwxr-x 3 ubuntu ubuntu 4096 Jul 13 13:31 .
drwxr-xr-x 5 ubuntu ubuntu 4096 Jul 13 13:31 ..
drwxrwxr-x 7 ubuntu ubuntu 4096 Jul 13 13:31 .git
```

***Figure 3.4**: Git init working – option 2*

After running **git init** on your project directory, if you see a **.git** subdirectory with the repository, meta-information gets created. Run the **init** command again on the same project directory, it will not overwrite the existing **.git** configuration.

Options:

We will be discussing the most commonly used Git INIT options. Those are as follows:

```
git init [-q | --quiet] [--bare] [--template=<template_directory>]
                [--separate-git-dir <git dir>] [--object-format=<format>]
                [-b <branch-name> | --initial-branch=<branch-name>]
                [--shared[=<permissions>]] [directory]
```

***Figure 3.5**: Git init options*

- **-q, --quiet**

  As the name suggests, when appended with **git init** it will only print error and warning messages; all other output will be suppressed.

- **--bare**

  To create a shared repository **-bare** flag should be passed with git init. Repository created using **-bare** flag does not have a working directory, which is the reason why committing or making changes to the repository is not possible.

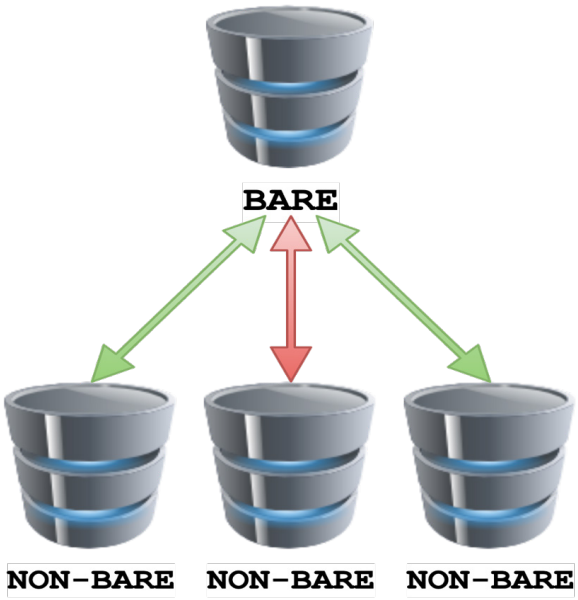

Figure 3.6: Git bare command

Most commonly, bare repositories are used to create a remote central repository.

- **--template**

  This command specifies the directory from which the templates will be used:

```
git init <directory> --template=<template_directory>
```

Figure 3.7: Git template command

  Git Templates enable you to create a new repository with a predefined **.git** subdirectory.

  A template can be configured to have default directories and files that will be copied to a new repository **.git** subdirectory.

  The default Git templates are typically located in the **/usr/share/git-core/templates** directory, but this may differ depending on your machine operating system.

- **--separate-git-dir**

  `Separate git dir` option is used to make a text file with the path to the actual repository :

```
git init <directory> --seperate-git-dir=<git dir>
```

Figure 3.8: Git separate git dir command

  This file serves as a filesystem-independent Git symbolic link to the repository.

  It is commonly used in scenarios where you want to keep system configuration dotfiles like .bashrc, .viminfo, .ssh, etc in the system home directory while keeping **`.git`** over a separate location.

  If the Git history has grown quite large, the project demands to move it to different high-capacity drives.

- **-b/--initial-branch**

  In the newly created repository, use the specified name for the first branch.

  If no name is specified, the default name is used (currently main, the name can be customized via the init.defaultBranch configuration variable).

Examples:

Begin a new Git repository for an existing code base:

```
~ cd /path/to/my/codebase
~ git init
~ git add .
~ git commit
```

Figure 3.9: Git init working example

1. Make a directory called **`/path/to/my/codebase/.git`**.
2. Fill in the index with all existing files.
3. Consider the repository first state to be the first commit in history.

Git clone - Clone a Git repository into a new directory

In many instances, you may want to download a copy of a project from GitHub so that you can work on it locally. Cloning is the process of copying the repository to your computer.

It is simple and easy to use this command. All we need is the URL of the repository we want to clone. GitHub provides the URL in the bottom-right corner of the repository home page, as shown in the following screenshot:

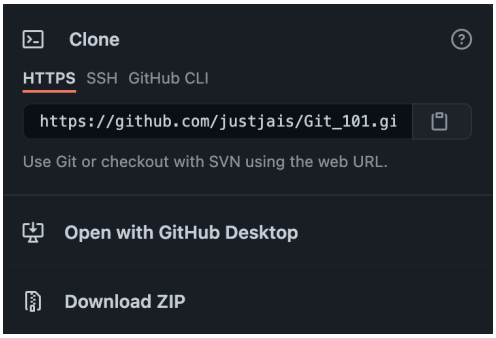

***Figure 3.10**: Git clone GitHub repository*

To copy the GitHub repository link, click on the clipboard icon on right hand side of the link.

Now, to clone the repository you can follow these steps:

1. Open the command line and browse to the location where you want to clone the respective GitHub repository.
2. I have taken the demo Git_101 GitHub repo and will be cloning the repo under my **`home_folder/Git_101`** folder
3. Now, at the final step run the command as:

```
~ git clone https://github.com/justjais/Git_101.git
```

***Figure 3.11**: Git clone GitHub repository – command line - a*

4. Once you run the respective command, Git will clone the mentioned repository from the GitHub URL provided as:

```
➜  ~ git clone https://github.com/justjais/Git_101.git
Cloning into 'Git_101'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 12.50 KiB | 639.00 KiB/s, done.
➜  ~
➜  ~ cd Git_101
➜  Git_101 git:(main)
```

***Figure 3.12**: Git clone GitHub repository – command line - b*

With this, Git created a new **`Git 101`** folder containing our downloaded repository.

Inside, we will find a **`README.md`** file, which is standard for a GitHub repository.

You can describe your repository to users who come across it in this file using the common markdown markup language.

Git's collaboration

If a project is already set up in a central repository, the most common way for users to obtain a development copy is with the **`git clone`** command.

Cloning, like **`git init`**, is typically a one-time operation.

All version control operations and collaborations are managed through a developer's local repository once they have obtained a working copy.

Working with Git is fundamentally different from working with source version control tools like **Apache Subversion** (**SVN**) as shown in the following figure:

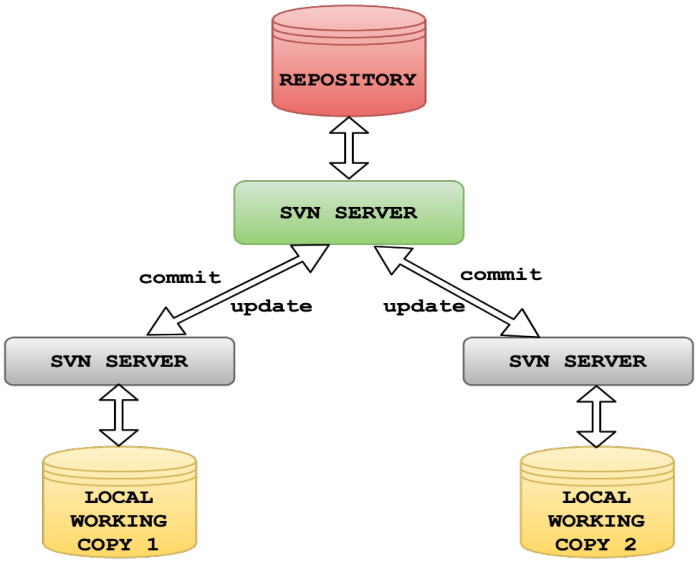

***Figure 3.13**: SVN (Apache Subversion) repository architecture*

Unlike SVN, which is based on the relationship between the central repository and the working copy, Git's collaboration model is built on repository-to-repository interaction.

Users can push or pull commits directly from one repository to another instead of checking a working copy into SVN's central repository, as shown in the following figure:

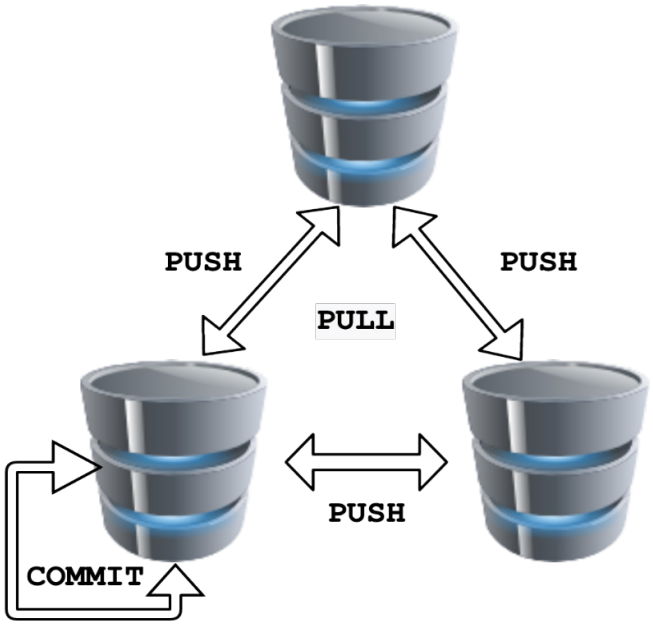

Figure 3.14: Git repo collaboration

Git Clone usage:

Git clone makes a duplicate clone copy of the already existing Git repository. The original repository can be either on the local filesystem or on a remote machine that supports the supported protocols.

This is why Git clone is majorly used to point to an existing repository and create a clone or copy of that repository in a different directory of the user's choice.

It is very similar to SVN checkout, but here the "*working copy*" is a full-fledged Git repository that maintains its history and files independently from the original Git repository which might be present on a remote server or more conveniently over GitHub.

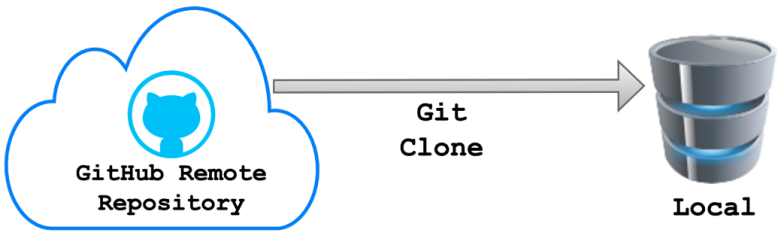

Figure 3.15: Git clone

Options:

We will be discussing the most commonly used Git CLONE as follows:

```
git clone [--template=<template_directory>]
          [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
          [-o <name>] [-b <name>] [-u <upload-pack>] [--reference
               <repository>]
          [--dissociate] [--separate-git-dir <git dir>]
          [--depth <depth>] [--[no-]single-branch] [--no-tags]
          [--recurse-submodules[=<pathspec>]] [--[no-]shallow
                -submodules]
          [--[no-]remote-submodules] [--jobs <n>] [--sparse]
          [--filter=<filter>] [--] <repository>
          [<directory>]
```

***Figure 3.16**: Git clone options*

- **-l/--local**

  When the repository to clone from is on a local machine, this flag bypasses the normal "Git aware" transport mechanism and clones the repository by creating a copy of HEAD as well as everything in the objects and refs' directories.

  To save space, the files in the **`.git/objects/`** directory are hard linked whenever possible.

- **-b <name>/--branch <name>**

  If there is a need to clone the Git repository based out of a particular tag/name; then, this option is utilized. This is the branch that will be checked out in a non-bare repository.

- **--bare**

  Create a bare Git repository, that is, rather than creating and storing administrative files in **`/.git`**, make it in **`$GIT DIR`**.

  Since there is nowhere to check the working tree, this obviously implies --no-checkout option is used.

  The remote branch heads are also copied directly to the corresponding local branch heads, without being mapped to **`refs/remotes/origin/`**.

  This option is commonly used when you have to make a repository where there are no remote-tracking branches or when no configuration variables are needed.

- **--depth <depth>**

  To make a shallow clone with a history limited to the number of commits specified. Unless user specifies **`--no-single-branch`**, which thus fetching the histories near the tips of all branches, and implies Git **`--single-branch`**.

```
git clone -depth=1 <repo>
```

***Figure 3.17**: Git clone shallow copy*

  In the preceding example, I am trying to clone a repository where depth is set to 1 which internally means that only the latest commit is included in the cloned repo. Shallow cloning is used or becomes necessary for the Git repository which has an extensive history, that might sometimes result in scaling issue or disk space limit issue.

  Shallow cloning is thus commonly used to avoid the above-discussed issue.

- **--mirror**

  Create a copy of the source repository.

  In contrast to **`--bare`**, **`--mirror`** not only maps local branches of the source to local branches of the target, but it also maps all refs (including remote-tracking branches, notes, and so on) and configures a ref-spec so that all these refs are overwritten by a git remote update in the target repository.

- **--template**

  Passing the template parameter with **`git clone`** implies that it clones the repository at **`<repo location>`** and applies the template from **`<template directory>`** to the newly created local branch.

```
git clone --template=<template_directory> <repo location>
```

***Figure 3.18**: Git clone template*

Visit the official Git clone documentation page for a comprehensive list of other git clone options.

Git Init vs Git Clone

It is important to note that the terms **`git init`** and **`git clone`** are frequently used interchangeably. They can both be used to "initialize a new git repository" at a high level.

`Git clone`, on the other hand, is reliant on **`git init`**. To make a copy of an existing repository, use **`git clone`**. To create a new repository, Git clone first calls **`git init`**. The data is then copied from the existing repository, with a new set of working files.

Work on the current change

This section of the book discusses the commands that are relevant with respect to Git branch current changes. Let us go through all the commands that are commonly used to work with Git branch current changes.

Git add - Adding file contents to the index

The command **`git add`** adds changes in the working directory to the staging area. This command is used to tell Git that you want to include updates to a specific file in the next commit. However, to record changes, you must also run **`git commit`**. Both work in conjunction with each other. Along with **`git push`**, you can compose fundamental Git workflow.

In addition to the commands listed above, the **`git status`** command is required to determine the current state of the working directory and the staging area it is in as shown in the following screenshot:

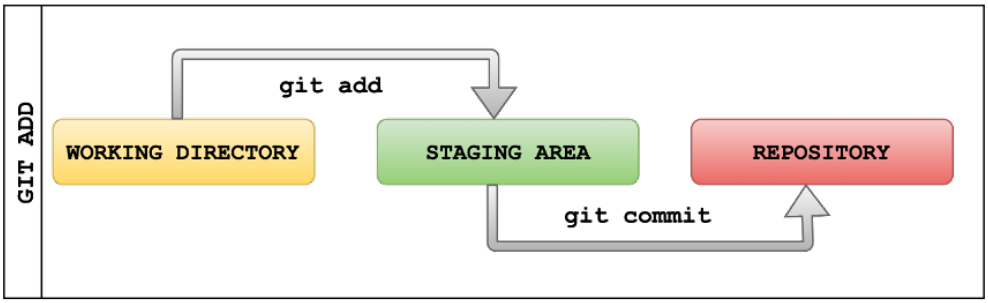

***Figure 3.19**: Git add flow*

As we already know, basic edit/stage/commit pattern governs project development. To understand the above figure, we need to understand the difference between the working directory and staging area.

- **Working area**

  In a nutshell, the working area contains all the files that are not in the staging area. Git is not in charge of them. They are pretty much just in your local directory. They are also known as untracked files.

  The working area is similar to the scratch space; it is where you can add new content, modify existing, and delete content. If the content you modify or delete is in your repository, you will not lose it.

- **Staging area**

  Git staging area is one of Git's unique and characteristic features and is also considered a part of Git's 3 trees along with the commit and working directory. You can think of it as a buffer between the current working directory and project commit history.

  The staging area allows the user to have all the relevant and related changes group into highly focused snapshots and commit them as one to project history.

  Options:

  We will be discussing the most commonly used Git ADD options as follows:

```
git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
        [--edit | -e] [--[no-]all | --[no-]ignore-removal | [--update | -u]]
        [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing] [--renormalize]
        [--chmod=(+|-)x] [--pathspec-from-file=<file> [--pathspec-file-nul]]

        [--] [<pathspec>...]
```

***Figure 3.20**: Git add available options*

- **-n, --dry-run**

  This option will not add the file(s) and will simply indicate whether the file exists and/or will be ignored.

- **-f, --force**

  This option allows adding otherwise ignored files.

- **-p, --patch**

  Select the chunk patch between the index and the work tree interactively and add them to the index. This allows the user to evaluate the differences before adding changed information to the index.

  This also executes **`add --interactive`** but skips the initial command menu and goes straight to the patch sub-command. Thus, it helps the user select a chunk of changes and stage the same for the next commit.

Examples:

1. Git add stages all the changes in the <file> for the next commit.

```
git add temp_file.txt
git commit
```

***Figure 3.21**: Git add filename*

2. Git Add stages all the changes in the <directory> for the next commit.

```
git add temp_directory
git commit
```

***Figure 3.22**: Git add directory*

3. Git Add stages all the changed file in the directory.

```
git add .
git commit
```

***Figure 3.23**: Git add all*

Mv - Move or rename a file, a directory, or a symlink

This command helps rename the file, giving the previous file name and the new name you want to give the file. This will mark your modification as ready for commit.

Options:

```
git mv <options>... <args>...
```

***Figure 3.24**: Git mv options*

We will be discussing the most commonly used Git CLONE options as follows:

- **-f, --force**

  Even if the file exists, force renaming or relocating a file.

- **-k**

  Moving or rename operations that would result in an error situation should be avoided. An error occurs when a source does not exist or is not controlled by Git, or when it would overwrite an existing file unless the **`-f`** option is used.

Examples:

1. Change the **`old_filename`** to **`new_filename`**

```
git mv old_filename new_filename
```

***Figure 3.25**: Git mv file*

2. To check the status if the effect has taken place and registered, check with **`git status`** command:

```
git status
> # On branch temp-branch
> # Changes to be committed:
> #   (use "git reset HEAD ..." to unstage)
> #
> #     renamed: old_filename -> new_filename
> #
```

***Figure 3.26**: Git mv status*

Restore - Restore working tree files

The **`git restore`** command is used to restore or delete uncommitted local modifications to files.

Options:

```
git restore [<options>][--source=<tree>][--staged]
            [--worktree][--] <pathspec>...
git restore [<options>][--source=<tree>][--staged]
            [--worktree] --pathspec-from-file=<file>
            [--pathspec-file-nul]
git restore (-p|--patch)[<options>][--source=<tree>]
            [--staged][--worktree][--][<pathspec>...]
```

***Figure 3.27**: Git restore options*

We will be discussing the most commonly used Git RESTORE options as follows:

- -s <tree>, --source=<tree>

  Restore the working tree files with the provided tree's content.

  It is a standard practice to describe the source tree by specifying a commit, branch, or tag that corresponds to it.

  If no arguments are supplied, the contents are restored from HEAD if **`--staged`** is specified, otherwise from the index.

  In a particular scenario, where there is just one merge base, you can use "A...B" as a shorthand for the merge base of A and B. You can leave out only one of A and B, in which case HEAD is used.

- -p, --patch

  Select chunks in the difference between the restore source and the restore location interactively.

  To understand how to use the **`--patch`** mode, go to the "Interactive Mode" section of **`git-add`**.

  It should be noted that **`--patch`** accepts no path-spec and will urge you to restore any changed paths.

- -W, --worktree, -S, --staged

  Set the location of the restoration. If neither option is given, the working tree is restored by default. The **`--staged`** option will simply restore the index.

  Working trees are restored when parameters are specified.

Examples:

1. To restore a file, use **`git restore <file_name>`**.

```
git restore temp_example.txt
```

***Figure 3.28**: Git restore file*

Below is the working example output and you can see that, we used git restore to remove a file's local modification.

```
→  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:

  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
deleted:    temp/__init__.py

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

→  Git_101 git:(main) ✗
→  Git_101 git:(main) ✗ git restore temp/__init__.py
→  Git_101 git:(main)
→  Git_101 git:(main) git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

***Figure 3.29**: Git restore file restore status*

2. To Un-stage a file from the staging area use this:

```
git restore --staged temp_example.txt
```

***Figure 3.30**: Git restore –staged syntax*

In the working example output, you can see that the specified file is un-staged, but the files' local modifications are still present.

```
➜  Git_101 git:(main) vi temp/__init__.py
➜  Git_101 git:(main) ✗
➜  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.

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:   temp/__init__.py

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

➜  Git_101 git:(main) ✗
➜  Git_101 git:(main) ✗ git add temp/__init__.py
➜  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:

  (use "git restore --staged <file>..." to unstage)
modified:   temp/__init__.py

➜  Git_101 git:(main) ✗ git restore --staged temp/__init__.py

➜  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.

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:   temp/__init__.py

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

➜  Git_101 git:(main) ✗
```

***Figure 3.31**: Git restore –staged working example*

3. To restore a file from a particular commit follow this command:

```
git restore --source e0f98da test_example.txt
```

***Figure 3.32**: Git restore –source using a specific commit*

The source option can be used to restore a file from a certain commit. By default, restore will use the contents of the HEAD directory.

In the preceding command, we are restoring a file from a previously mentioned commit.

rm - Remove files from the working tree and from the index

The **`git rm`** command is used to delete a single file or a group of files. Git rm's primary job is to delete tracked files from the Git index. It is also used to remove files from the staging index as well as the working directory.

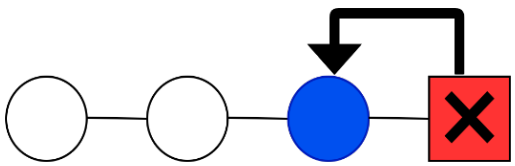

***Figure 3.33**: Git rm*

Options:

```
git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch]
       [--quiet] [--pathspec-from-file=<file> [--pathspec-file-nul]]
       [--] [<pathspec>...]
```

***Figure 3.34**: Git rm options*

We will be discussing the most commonly used Git RM options and those are as follows:

- **`-f`, `--force`**

  This option overrides the up-to-date check.

- **`-n`, `--dry-run`**

  Do not delete anything. Instead, just display if they exist in the index and would be deleted by the command otherwise. Therefore, dry run acts as a safeguard which tells the user what all files it would have removed.

- **`-r`**

  `-r` here signifies recursive delete, when a directory is given to delete. It will delete the directory and all the content inside it.

- **`--cached`**

  This option is used to unstage and delete routes from the index alone. Working tree files will be left alone, whether they have been changed or not.

- **`--ignore-unmatch`**

  Even when no files were found, exit with a status of zero. This is commonly used when **`git rm`** is a part of a script, and it gracefully fails in error circumstances.

- **`-q`**, **`--quiet`**

  For each file deleted, **`git rm`** usually generates one line (in the form of a **`rm`** command). This option disables that output.

Examples:

1. Delete all the filenames with **`.txt`** extension from the index under **`Documentation`** directory as shown in the following figure:

```
git rm Documentation/\*.txt
```

***Figure 3.35**: Git rm directory*

   In this example, the asterisk **`*`** is quoted from the shell, which allows Git, rather than the command-line shell, to extend the pathnames of files and subdirectories in the **`Documentation/`** directory.

2. This example applies the force option to target all **`git-*.sh`** files using a wildcard.

```
git rm -f git-*.sh
```

***Figure 3.36**: Git rm file*

   Note, the force option removes the target files from the working directory as well as the staging index.

3. **`git rm`** changes are not permanent, and if user needs to undo the changes it can be done as the update is made to staging index. The changes need to be

available under **git log** history, and needs to be committed first. Command to undo the changes is as follows:

```
git reset HEAD
```

Figure 3.37: Git reset

git reset head undo the changes made by **git rm**, and will roll back the current staging index and working directory to the **HEAD** commit.

Similar, undo functionality can be achieved via **git checkout .**, which restores the most recent version of a file from **HEAD**.

```
git checkout .
```

Figure 3.38: Git checkout

sparse-checkout - Initialize and modify the sparse-checkout

Git sparse-checkout initializes and customizes the sparse-checkout configuration, which restricts the checkout to a set of routes defined by a pattern list.

Options :

```
git sparse-checkout <subcommand> [options]
```

Figure 3.39: Git sparse-checkout file

To examine the history and state of the repository

For examining the Git History, we can use the following command line Git commands:

bisect

To locate the commit that caused a problem, use binary search. **Git bisect** gives you a way out of the tedious task of checking out recent commit to search for broken commit. Instead, Git bisect lets you walk through recent commits and lets you verify if the commit contains the broken commit.

Options:

```
git bisect <subcommand> <options>
```

***Figure 3.40**: Git bisect - 1*

The goal of **`git bisect`** is to detect a specific regression by running a binary search through history.

Let us assume you have the following history of development:

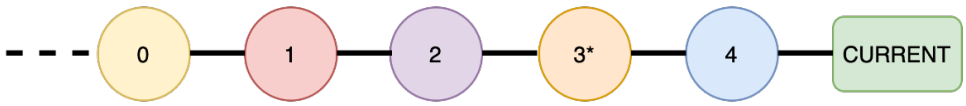

***Figure 3.41**: Git bisect - 2*

You are aware that your application is not operating properly at the current revision, even though it was working perfectly at revision **0**.

As a result, regression was most likely introduced in commits **1, 2, 3, 4,** and **current**.

You may try checking each commit, build it, and seeing whether the regression is still present.

This can take a long time if there are a lot of commits. This is a linear search, which can be significantly improved by using a binary search.

The **`git bisect`** command accomplishes this. It seeks to cut the number of possibly problematic revisions in half at each level.

```
git stash save
git bisect start
git bisect bad
git bisect good 0
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[< ... sha ... >] 3
```

***Figure 3.42**: Git bisect - 3*

Git will checkout a commit after this command. It will commit 3 in our instance. You must create your script and validate that the regression is present. If the regression

is present, you will additionally need to notify git the status of this revision with **`git bisect bad`**, or **`git bisect good`** when it is not.

```
git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[< ... sha ... >] 4
```

***Figure 3.43**: Git bisect - 4*

After that, it will check another commit, **3** or **4** (as there are only two commits). Assume it chose number four. We update the script after it has been built to ensure that the regression is present.

```
git bisect bad
Bisecting: 0 revisions left to test after this (roughly 1 step)
[< ... sha ... >] 3
```

***Figure 3.44**: Git bisect - 5*

We put the most recent revision, **3**, to the test. We inform Git because that is the source of the regression:

```
git bisect bad
< ... sha ... > is the first bad commit
< ... commit message ... >
```

***Figure 3.45**: Git bisect - 6*

We just had to evaluate three variants (**2, 3, 4**) in this basic circumstance, rather than three (**1, 2, 3**,). This is a minor victory, but is significant because our history is so brief. We should expect to test **1 + log2 N** commits with **`git bisect`** instead of roughly **N/2** commits with a linear search if the search range is **N** commits.

You can examine the commit that caused the regression once you have identified it. After that, you may run **`git bisect reset`** to restore everything to its previous condition before using the **`git bisect`** command.

diff

`git diff` is a multi-purpose Git command that performs a diff on Git data sources when run. Commits, branches, files, and other data sources are examples of data sources.

The **`git diff`** command, along with **`git status`** and **`git log`**, is frequently used to examine the current state of a Git repository.

Options:

```
git diff [<options>] [<commit>] [--] [<path>...]
git diff [<options>] --cached [<commit>] [--] [<path>...]
git diff [<options>] <commit> [--merge-base] [<commit>...] <commit> [--] [<path>...]
git diff [<options>] <commit>...<commit>] [--] [<path>...]
git diff [<options>] <blob> <blob>]
git diff [<options>] --no-index [--] <path> <path>]
```

Figure 3.46: Git diff - 1

- **Compare all changes and changes since last commit**

  Without specifying a file path, **`git diff`** compares changes across the whole repository. By default, git diff will display any changes that have not been committed since the last commit.

```
git diff
```

Figure 3.47: Git diff - 2

- **Compare files from two separate git commits**

  Git refs can be provided to git diff to compare commits. HEAD, tags, and branch names are some examples of refs.

  Every Git commit has a commit ID, which you can find by running a **`git log`**. You may also use **`git diff`** with this commit ID.

```
commit c21ff5c97f52e265e3974027f9a36158937eb9ec
Author: Sumit Jaiswal
Date:   Fri Aug 27 12:32:49 2021 +0530

    fix comment

commit f41ce854974252bd8f6a99c2f507c1c27f2c4022
Author: Sumit Jaiswal
Date:   Fri Aug 27 12:09:58 2021 +0530

    add changelog

commit e186d6d31822dfe63837af8dcae46d943a4a0638
Author: Sumit Jaiswal
Date:   Fri Aug 27 12:07:22 2021 +0530

    fix code for the issue 120
```

Figure 3.48: Git diff – 3

To get the diff between two Commit IDs, we can do as done in the following figure:

```
git diff c21ff5c97f52e265e3974027f9a36158937eb9ec f41ce854974252bd8f6a99c2f507c1c27f2c4022
```

Figure 3.49: Git diff - 4

- **Compare Git Branch**

  Git Branches are compared in the same way as all other ref inputs are in git diff.

  The dot operator is demonstrated in the following snippet.

  The two dots indicate that the diff input is the tip of both branches. If the dots are removed and a space is utilized between the branches, the result is the same.

  By adjusting the first input parameter **`branch1`**, the three-dot operator starts the diff. It transforms branch1 into a reference to the shared common ancestor commit between the two diff inputs, branch1's shared ancestor and the ancestor of the other-feature-shared branch.

  As the tip of the other feature branch, the last parameter input parameter remains intact.

```
git diff <branch1>..<branch2>
git diff <branch1>...<branch2>
```

Figure 3.50: Git diff - 5

- Compare two files from different branch

  To compare a specific file across different branches, provide **`git diff`** the file's path as the third parameter.

```
git diff <branch 1> < branch 2> <path/filename>
```

Figure 3.51: Git diff - 6

An explicit file path parameter can be provided to the **`git diff`** command.

The diff operation will be scoped to the provided file when a file path is passed to **`git diff`**. This is illustrated in the examples below.

```
git diff HEAD <path/filename>
```

***Figure 3.52**: Git diff - 7*

This will compare the specific changes in the working directory to the index, presenting the changes that have not yet been staged.

By default, **`git diff`** performs the comparison against the **HEAD** branch.

```
git diff --cached <path/filename>
```

***Figure 3.53**: Git diff - 8*

When using the --cached option with **`git diff`**, the diff compares the staged changes to the local repository. The options **`--cached`** and **`--staged`** are interchangeable.

grep

Git grep searches for filter/pattern in the work tree monitored files, blobs in the index file, or blobs of certain tree items. Filters are a collection of one or more search phrases separated by newlines. When used as a search expression, an empty string matches all lines.

Options:

```
git grep [<options>] [-e] <pattern> [<rev>...] [[--] <path>...]
```

***Figure 3.54**: Git grep - 1*

To find commit content (that is, real lines of code rather than commit messages and the like), perform the following:

```
git grep <regexp> $(git rev-list --all)
```

***Figure 3.55**: Git grep - 2*

If you wish to confine the search to a certain subtree (for example, "**`plugins/modules`**"), use the **`rev-list`** subcommand and grep together:

```
git grep <regexp> $(git rev-list --all -- plugins/modules) -- plugins/modules
```

***Figure 3.56**: Git grep - 3*

This will look for regexp patterns in all your commit content.

Since rev-list will return the revisions list where all the changes to **`plugins/modules`** occurred, you must additionally give the path to grep so that it will just search in **`plugins/modules`**.

Consider the scenario: The same **`<regexp>`** could be found by grep on additional files in the same revision given by rev-list (even if there was no change to that file on that revision).

Few other relevant and useful example are:

```
git grep <regexp>
# Search for regex in working tree
git grep -l -e <regexp1> --and -e <regexp2>
# Search working tree for lines of text matching regexp1 and line of text matching regexp2
git grep -l -e <regexp1> --and -e <regexp2>
# Look through the working tree for files that include lines of text that match the regexp1 and regexp2 regular expressions.
git grep <regexp> $(git rev-list --all)
# Search all revisions for text matching <regexp>
```

***Figure 3.57**: Git grep - 4*

log

The **`git log`** command displays snapshots that have been committed. It is used to list and filter the project's history, as well as search for specific modifications.

In contrast to **`git status`**, which controls the working directory and staging area, **`git log`** only operates on the committed history.

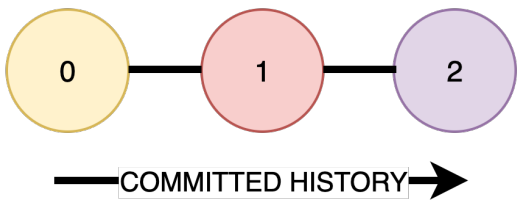

***Figure 3.58**: Git log – 1*

Options:

Git Log options:

```
git log [<options>] [<revision-range>] [[--] <path>...]
git show [<options>] <object>...
```

***Figure 3.59**: Git log – 2*

The following example will provide a complete diff of all the modifications made by the author to the file **learning_git.txt**.

```
git log --author="Sumit Jaiswal" -p learning_git.txt
```

***Figure 3.60**: Git log – 3*

When comparing branches, the **..** syntax is utilized.

The following example gives a quick summary of all the commits that are in **<branch_name>** *instead of the main* branch.

```
git log --oneline main..<branch_name>
```

***Figure 3.61**: Git log – 4*

show

The **git-show** is a command-line utility that displays additional information about Git objects such as blobs, trees, tags, and commits.

git-show reacts differently depending on the type of object.

When we work with Git, we see the **.git** folder, which contains many subdirectories, one of which is the **.git/objects** directory, which contains information about various types of objects such as blobs, trees, commits, and tags.

- Blob object: stores the file's contents.
- Tree object: a list of all the files in our repository, each with a pointer to the blob object assigned to it.
- Commit object returns a pointer to the object's tree.
- Tag object: Display the tag message as well as any other objects associated with the tag (object name, object type, tag name).

We use the command **`git show`** to see more information about these objects.

We have made changes to the file **`pull_101.md`** and committed the changes, now using the **`git show`** command directly, we can check in the below figure what all information is shown in the command output:

```
➜  Git_101 git:(main) git show
commit fdad70b728f67241fd97ae498d270f4feb1a61b1 (HEAD->main, tag: 1.1.0, tag:1.0.0, origin/main, origin/HEAD)
Merge: efd4ea4 cdb75e0
Author: Sumit Jaiswal <justjais@gmail.com>
Date:   Mon Jun 27 14:48:16 2022 +0530

    Merge pull request #2 from justjais/test_workPR

    Adding text to pull_101 readme file
```

***Figure 3.62**: Git show*

We can infer two things from the output of the **`git show`** command.

- The commit message and the pointer pointing to the HEAD.
- The second thing that can be seen is the various versions of the file.

If we pass the specific commit ID which is not the latest commit, we will not get **`HEAD->main`** as the respective commit is not pointing to the **`HEAD`**.

```
➜  Git_101 git:(main) git show 5fc03926898f2bd531c49e38584c2036c9df08ff
commit 5fc03926898f2bd531c49e38584c2036c9df08ff
Author: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Thu Jun 23 18:19:42 2022 +0530

    Adding text to pull_101 readme

diff --git a/temp/pull_101.md b/temp/pull_101.md
index f214544..cc13842 100644
--- a/temp/pull_101.md
+++ b/temp/pull_101.md
@@ -1 +1,2 @@
# This file is a placeholder for PULL request.
+Pull 101: Starting point to Pull request
```

***Figure 3.63**: Git show with commit ID*

`Git show` is a powerful command for inspecting objects in a Git repository. It is possible to use it to target specific files at specific revisions.

When you run **`git show`** on a commit range, it will show you the individual commits that fall within that range.

status

The **`git status`** command displays information about the current working directory and staging area. It allows you to observe staged changes as well as files that Git does not track.

There is no information about the committed project history in the Status output. Use the **`git log`** command to do this. The **`git status`** command simply shows the status of the **`git add`** and **`git commit`** commands.

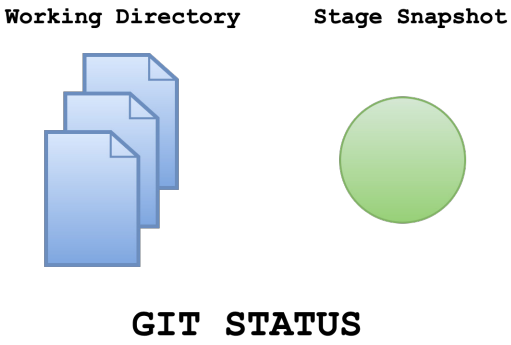

***Figure 3.64**: Git status – 1*

Options:

Git status options:

```
git status [<options>] [--] <pathspec>...
```

***Figure 3.65**: Git status – 2*

- **Clean working tree**

  Before we get started with the **`git status`** command, let us have a look at what the **`git status`** looks like when no modifications have been made. To check the status, launch git bash and execute the status command on the directory you want to check.

```
➜  Git_101 git:(main) git status
On branch main
nothing to commit, working tree clean
```

***Figure 3.66**: Git status – 3*

- **New file in working tree**

  When we add a file to the repository, the repository's state changes.

Let us use the touch command to make a file. Now use the status command to check the status. Please refer the following terminal output:

```
➜  Git_101 git:(main) touch learn_git.txt
➜  Git_101 git:(main) ✗ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        learn_git.txt

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

Figure 3.67: Git status – 4

The status is **nothing added to commit but untracked files present (use "git add" to track)** as we can see from the preceding output. The suggestions are also displayed by the status command.

It suggests using the **git add** command to track the file, as seen in the above result.

```
➜  Git_101 git:(main) ✗ git status
On branch main

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   learn_git.txt
```

Figure 3.68: Git status – 5

We can observe from the preceding output the file's state after the staging **changes to be committed.** We can verify the status before making a quick decision. This command will assist us in avoiding modifications we do not wish to make. Let us commit it and then see how it is doing. Consider the following result:

```
➜  Git_101 git:(main) git status
On branch main
nothing to commit, working tree clean
```

Figure 3.69: Git status – 6

We can verify the current condition of the file after it has been committed as clean as it was before.

- **Modify file in Working tree**

  Let us see what happens when an existing file is changed. To alter a file, use the echo command to update an existing file, which will add the text to the given file. Next, verify the repository's status.

```
→  Git_101 git:(main) echo "Modify existing file content" > learn_git.txt
→  Git_101 git:(main) ✗ git status
On branch main

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:   learn_git.txt

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

***Figure 3.70**: Git status – 7*

Once the changes are staged, added to repository, and modified files are committed, we can verify the status of the Git repository again using **`git status`**. Once again, it should be clean.

- Delete file in Working tree

  Now, if we go and delete the file that we created and committed to the earlier step using **`rm`** command and check the status of the working tree using **`git status`**, we can verify that git reports one of the files is deleted from the working tree.

```
→  Git_101 git:(main) rm learn_git.txt
→  Git_101 git:(main) ✗ git status
On branch main

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    learn_git.txt

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

***Figure 3.71**: Git status – 8*

To grow, mark and tweak your repo history

In this section, we will go through the Git command to grow and mark the Git history.

branch

The **`git branch`** command makes it possible to create, list, and remove branches. It does not enable you to move between branches or reassemble a branched history.

As a response, **`git branch`** is integrated with the operations **`git checkout`** and **`git merge`**.

Branches indicate a single development line which in case of Git is **`main`**. They can be used to request a new working directory, staging area, or project history.

Creating separate branch lines for two different features by two different developers in branches will allow developers to work on them in parallel while keeping the *main* branch free of under-development or problematic code.

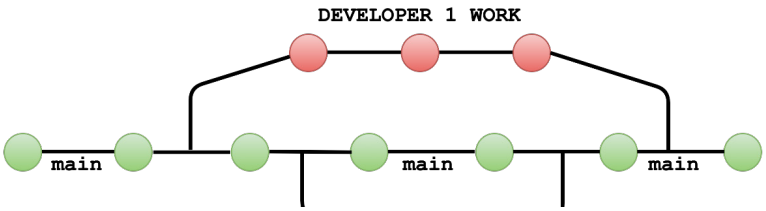

***Figure 3.72**: Git branch – 1*

Most version control systems provide branching as a feature. Git branches serve as a link to a snapshot of your modifications. When you plan to fix issues or add new features, you create a new branch to encapsulate the changes. This assists you in cleaning up the past of the future before integrating it.

Git branches are an important component of every developer's workflow. Git saves the branch as a reference to a commit, rather than copying files from one location to another.

Options:

```
git branch [<options>] [-r | -a] [--merged] [--no-merged]
git branch [<options>] [-l] [-f] <branch-name> [<start-point>]
git branch [<options>] [-r] (-d | -D) <branch-name>...
git branch [<options>] (-m | -M) [<old-branch>] <new-branch>
git branch [<options>] (-c | -C) [<old-branch>] <new-branch>
git branch [<options>] [-r | -a] [--points-at]
git branch [<options>] [-r | -a] [--format]
```

***Figure 3.73**: Git branch – 2*

- **Create Branch**

  The **`git branch`** command can be used to create a new branch:

```
git branch <branch_name>
```

***Figure 3.74**: Git branch – 3*

  This command will be executed as follows:

```
➜  Git_101 git:(main) git branch dev_branch_1
➜  Git_101 git:(main)
```

***Figure 3.75**: Git branch – 4*

  This will create the **`dev_branch_1`** branch in the Git directory locally.

- **List Branch**

  To list the available branches in the repository, we may either use **`git branch --list`** or the **`git branch`** command:

```
➜  Git_101 git:(main) git branch
   dev_branch_1
 * main
➜  Git_101 git:(main) git branch --list
   dev_branch_1
 * main
```

***Figure 3.76**: Git branch – 5*

  Both these commands are listing the repository's available branches. The sign * denotes the branch that is presently active.

- **Rename Branch**

  The Git branch command may be used to rename the branch. Use the following command to rename a branch:

```
git branch -m <branch old name> <branch new name>
```

***Figure 3.77**: Git branch – 6*

If we execute the git branch with rename tag, we will be able to rename the input branch as:

```
➜  Git_101 git:(main) git branch -m dev_branch_1 rename_dev_branch_1
 * main
   rename_dev_branch_1
```

***Figure 3.78**: Git branch – 7*

You can see from the above console output that the branch **`dev_branch_1`** got renamed to **`rename_dev_branch_1`**.

- **Delete Branch**

  You may delete a remote or local git branch using the Git application. The local branch is the Git branch that is available on your local development system; deleting the local branch will only affect the local modifications on your machine, not your GitHub project.

  If you want to delete/remove a branch remotely, this will have an impact on your GitHub project main branch.

  - Delete/Remove a Remote branch

    The Git desktop application allows you to remove a remote branch. To remove a remote branch, execute the following command:

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

***Figure 3.79**: Git branch – 8*

Run output:

```
➜  Git_101 git:(main) git push origin --delete update_readme
To https://github.com/justjais/Git_101.git
 - [deleted]         update_readme
```

***Figure 3.80**: Git branch – 9*

This deletes the branch **`update_readme`** remotely from GitHub, and you can verify the same by checking the available **`git branch`** and your local **`update_readme`** branch should remain intact.

  - Delete/Remove a Local branch

The Git desktop application allows you to remove a local branch. To remove a local branch, execute the following command:

```
git branch -D <branch name>
```

***Figure 3.81**: Git branch – 10*

Run output:

```
➜  Git_101 git:(main) git branch -D update_readme
Deleted branch update_readme (was 056bb29).
```

***Figure 3.82**: Git branch – 11*

Now, if you run the **`git branch`** on your local machine, **`update_readme`** will not be shown under available git branches.

- Switch Branch

  You may swap between branches without committing with Git. The Git checkout command allows you to move back and forth between git branches. The following command is used to switch between the branches:

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

***Figure 3.83**: Git branch – 12*

You can switch from **`main`** to any other branch available on your git repository without making any commit.

```
➜  Git_101 git:(main) git checkout dev_branch
Switched to branch 'dev_branch'
➜  Git_101 git:(dev_branch)
```

***Figure 3.84**: Git branch – 13*

The branch is switched from **`main`** to **`dev_branch`** without committing, as shown in f*igure 3.81*.

- Merge Branch

  You may merge the other branch with the one that is presently active in Git. The **`git merge`** command can be used to merge two branches.

To merge the branches, use the following command:

```
git merge <branch name>
```

Figure 3.85: Git branch – 14

Run output:

```
➜  Git_101 git:(main) git merge dev_branch
Already up to date.
```

Figure 3.86: Git branch – 15

The **main** branch merged with **dev_branch** as seen in the output above. Since I did not commit anything before merging, the output appears to be up to date.

Commit

The Git commit command saves the project's current staged changes. Commits are used to document a project's current status. Git asks before altering committed snapshots, so they are regarded as safe versions of a project.

Git add is used before executing **git commit** to promote changes to the project, which will subsequently be saved in a commit.

Commits are the project's snapshots. Every commit is saved in the repository's main branch. We can undo the commits or go back to an earlier version. Since each commit has its unique commit-id, two separate commits will never overwrite each other.

The **Secure Hash Algorithm** (**SHA**) algorithm generates this commit-id, which is a cryptographic number.

Options :

```
git commit [<options>] [--] <pathspec>...
```

Figure 3.87: Git commit – 1

The commit command saves the modifications and assigns a commit-id to them. Without any arguments, the commit command will launch the default text editor and prompt for a commit message.

In this editor, we may write our commit message. In this example, I have updated the content of the file **learn_git.txt** and am trying to commit the changes.

```
→  Git_101 git:(dev_branch) ✗ git commit
```

***Figure 3.88**: Git commit – 2*

The command will launch a default text editor and prompt us with a commit message as we type it. Here, **updating file content** is my commit message.

```
updating file content
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch dev_branch_1
# You are currently bisecting, started from branch 'main'.
#
# Changes to be committed:
#       modified:   learn_git.txt
#
```

***Figure 3.89**: Git commit – 3*

To verify the successful **git commit**, we can check the logs using **git log** command:

```
→  Git_101 git:(dev_branch) ✗ git commit
commit 3d84d5b97dcea8bed29b033074bbed8bc3345e98 (HEAD -> dev_branch_1)
Author: Sumit Jaiswal
Date:   Mon Aug 30 00:17:48 2021 +0530

    updating file content
```

***Figure 3.90**: Git commit – 4*

We can verify the output with **commit-id**, **author detail**, **date** and **time**, and the **commit message** information.

- **Git commit –a**

  The **-a** option of the **commit** command allows you to specify specific commits. It is used to commit all changes' snapshots. This option solely takes into account files that have previously been uploaded to Git. The newly generated files will not be committed.

Check the repository's status and issue the commit command as follows:

```
git commit -a
```

Figure 3.91: Git commit – 5

Firing git commits with **-a** tag will only commit the files that have already been added. The files that have not been staged will not be committed.

- **Git commit –m**

  You may write the commit message on the command line using the **-m** option of the commit command. The text editor will not be prompted by this command :

```
git commit -m "commit message"
```

Figure 3.92: Git commit – 6

Run output:

```
➜  Git_101 git:(dev_branch) ✗ git commit -m "updating file content"
[dev_branch 056bb39] Introduced "learn_git"
1 file changed, 1 insertions(+), 0 deletions(-)
create mode 100644 learn_git.txt
```

Figure 3.93: Git commit – 7

In the above output, a **learn_git.txt** is committed to our **dev_branch** repository with a commit message. The same can be verified using **git log** command.

Merge

Git merge is a process to connect the branched histories. It combines the development histories of two or more things. You can easily take the data produced by git branch and merge it into a single branch using the **git merge** command.

A group of commits will be combined into a single, unified history using git merge. Typically, two branches are combined using git merge.

The idea behind Git Merge is to combine multiple sequences of commits stored in different branches into a unified history, or, to put it another way, in a single branch.

When we try to merge two branches, Git takes two commit pointers and looks for a common base commit in the two git branches.

When Git locates the common base commit, it automatically generates a **`merge commit`** and merges each pending merge commit sequentially. Git performs all these processes, and if there are any conflicts, Git displays them using appropriate merging algorithms.

Rebase

Rebasing is the process of applying commitments once more on top of a base trip. It is employed to combine several commits from several branches into a single commit. It serves as a substitute for the **`git merge`** command. The process of merging is linear. A series of commits are moved or combined into a new base commit throughout this process.

Rebasing works best and is most understandable when used in conjunction with a feature branching methodology.

Git rebase is regarded as a possible replacement for the git merge command. Merge is always a record that moves forward. In contrast, Rebase is a powerful history-rewriting tool in git. It sequentially merges several commits.

Consider that you have two commits in your feature branch and four commits in your main branch. If you merge this, all commits will be merged at once. However, if you rebase it, it will be merged linearly, as depicted in the following figure:

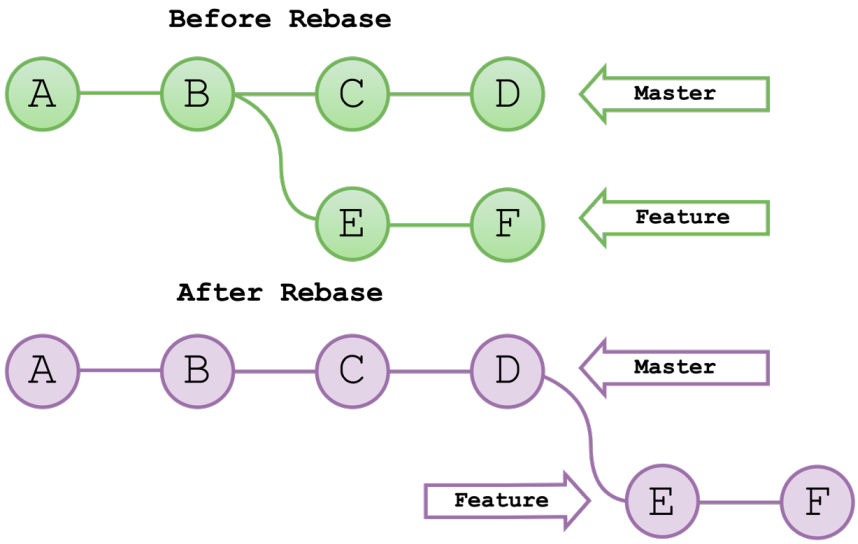

***Figure.3.94**: Git rebase*

Rebasing is the process of altering the base of your branch from one commit to another, giving the impression that your branch was created from a different commit. Git does this internally by generating new commits and applying them to the chosen base.

It is important to understand that, despite the branch looking similar, it contains entirely new changes.

Options:

```
git rebase [-i | --interactive] [<options>] [--exec <cmd>]
           [--onto <newbase> | --keep-base] [<upstream> [<branch>]]
git rebase [-i | --interactive] [<options>] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase (--continue | --skip | --abort | --quit | --edit-todo | --show-current-patch)
```

***Figure 3.95**: Git rebase options*

When we make some changes to the main branch and feature branch, respective branches, whether the main or feature branch, can both be rebased.

Track the modifications with the **`git log`** command. You should check out the branch you want to rebase to. To perform the rebase we need to use the following syntax:

```
git rebase  <branch name>
```

***Figure 3.96**: Git rebase syntax*

Resolve any conflicts in the branch and then execute the following commands to continue making modifications. To determine the status:

```
git status
```

***Figure 3.97**: Git status check*

To continue with the changes made, we need to run the following command:

```
git rebase --continue
```

***Figure 3.98**: Git rebase continue*

The changes can be skipped using the following command:

```
git rebase --skip
```

***Figure 3.99**: Git rebase skip*

Once the rebasing is completed and there is no merge conflict, we can force push the commit to the origin as:

```
git push --force-with-lease origin <branch name>
```

***Figure 3.100**: Git force push*

If another user has rebased and force-pushed to the branch you are committing to, a git pull will override all changes you have made based on that prior branch with the forcibly pushed tip.

Fortunately, you can obtain the record of the remote branch using **`git reflog`**. A ref before it was rebased can be found in the remote branch's reflog.

You can then use the **`--onto`** option to rebase your branch against that remote reference.

Git rebase can be invoked using the **`--onto`** command-line parameter. When you use **`git rebase --onto`**, the command expands to:

```
git rebase --onto <new base-branch name> <old base-branch name>
```

***Figure 3.101**: Git rebase --onto*

The **`--onto`** command enables a more sophisticated type of rebase that permits specific refs to be passed as rebase recommendations.

Git Interactive Rebase

Git supports Interactive Rebase, a powerful tool that allows you to edit, rewrite, reorganize, and more, on existing commits. Interactive Rebase can only be used on the currently checked-out branch. As a result, place your local HEAD branch in the sidebar.

Git interactive rebase can be used with the rebase command; simply type **-i** after the rebase command. The letter i here stands for interactive. This rebase-interactive syntax is as follows:

```
git rebase -i
```

***Figure 3.102**: Git rebase –I*

Standard Vs Interactive Rebase

Standard and Interactive are the two distinct Git rebase modes:

In standard mode, Git rebase automatically collects commits from your active working branch and applies them to the passed branch's head.

On the other hand, instead of simply gathering everything and dumping it into the passed branch, Git's interactive rebase mode allows you to change specific commits as the process progresses. You can edit the history and remove, divide, or change previous commits if you use interactive mode.

Merge Vs Rebase

When to use the merge command and rebase is a frequently asked question among Git users. Both commands are similar since they merge changes from various branches of a repository.

Rebasing is not encouraged on a shared branch because it results in inconsistent repositories. Individuals may find that rebasing is more beneficial than merging. If you want to see the entire history, utilize the merge. Merge keeps track of all previous commits, whereas rebase creates a new one.

Another important point one should keep in mind is that merge can be used with both public and private branches, while rebase is not done ideally over public branches. During the process, merge preserves the Git history, while rebase re-writes it.

Reset

The Git reset command is a powerful and flexible tool for reverting changes. The commit tree (HEAD), the staging index, and the working directory are Git's three internal state management mechanisms.

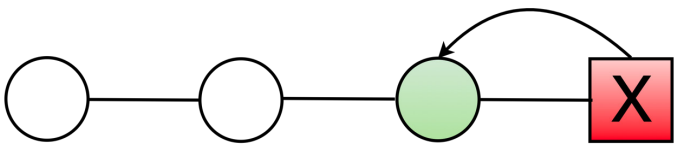

***Figure 3.103**: Git reset*

Options:

```
git reset [-q] [<tree-ish>] [--] <pathspec>...
git reset [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]
git reset (--patch | -p) [<tree-ish>] [--] [<pathspec>...]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

***Figure 3.104**: Git reset options*

There are three primary ways to invoke the **`git reset`** command:

- **`--hard`**
- **`--mixe`**d
- **`--soft`**

Git is a tool that restores HEAD's state to a predetermined state. For Git, it serves as a time machine. You can hop back and forth among different commits. The specific trees that Git uses to manage your file's content are affected by each of these reset modifications. Git reset can also work on complete commit objects or on a file-by-file basis.

Each of these reset changes has an impact on the specific trees that git uses to manage your file and its contents.

Tag

Git Tags highlight a particular period in Git history. A commit stage can be marked as relevant by using tags. It is mostly used to indicate the starting point of a project. A commit can be tagged for later use.

Like branches, once launched, tags do not alter. Any number of tags can be placed on a single branch or several branches.

A tag is analogous to a branch that never changes. On the other hand, Tags, unlike branches, have no history of commits after being created.

There are numerous branches in the following figure. In the repository, each of these versions has a tag.

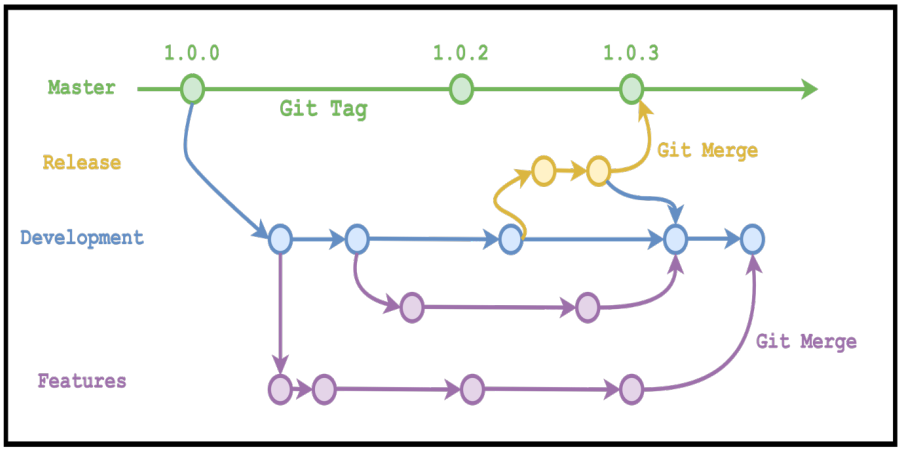

***Figure 3.105**: Git Tag – 1*

Options:

Git Tag options:

```
git tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <file>] [-e]
           <tagname> [<commit> | <object>]
git tag -d <tagname>...
git tag [-n[<num>]] -l [--contains <commit>] [--no-contains <commit>]
           [--points-at <object>] [--column[=<options>] | --no-column]
           [--create-reflog] [--sort=<key>] [--format=<format>]
           [--merged <commit>] [--no-merged <commit>] [<pattern>...]
git tag -v [--format=<format>] <tagname>...
```

***Figure 3.106**: Git tag options*

Two types of available Git Tags are:

- Annotated tag
- Light-weighted tag

These two tags are comparable, but they differ in terms of the number of stored Metadata.

The recommended strategy is to treat Lightweight tags as private and Annotated tags as public. Extra meta data is kept in annotated tags, such as the tagger's name, email address, and date.

For a public release, this information is crucial. Lightweight tags, which are basically "bookmarks" on a commit, are excellent for building rapid links to pertinent commits because they only provide a name and a hyperlink to the commit.

- **Creating a Light-Weighted Tag**

  Lightweight tags serve the purpose of designating a location in the repository. It is often a commit that is kept in a file. To keep it lightweight, it does not save extraneous data.

  In a light-weighted tag, a command-line option like **`-a`**, **`-s`**, or **`-m`** is not provided; instead, a tag name is passed like:

```
git tag <tag name>
```

***Figure 3.107**: Git Lightweight Tag*

  To tag the Git_101 project as 1.0.0, we can use the following lightweight syntax:

```
→ Git_101 git:(main) git tag 1.0.0
```

***Figure 3.108**: Git Lightweight Tag example*

  Once the tagging is successful, the user can check the tag via **`git show`** command:

```
→ Git_101 git:(main) git tag 1.0.0
→ Git_101 git:(main) git show 1.0.0
commit fdad70b728f67241fd97ae498d270f4feb1a61b1 (HEAD -> main, tag: 1.0.0, origin/main, origin/HEAD)
Merge: efd4ea4 cdb75e0
Author: Sumit Jaiswal <justjais@gmail.com>
Date:   Mon Jun 27 14:48:16 2022 +0530

  Merge pull request #2 from justjais/test_workPR

  Adding text to pull_101 readme file
```

***Figure 3.109**: Git show Tag example*

- **Creating an Annotated tag**

  Annotated tags provide supplementary Metadata like the name of the developer, their email address, the date, and more. They are kept in the Git database as a collection of objects.

  It is advised to create an annotated tag if you are pointing and storing a final version of any project. However, you can construct a lightweight tag if you

only want to make a temporary mark or do not want to disclose information. For the project to be released to the public, the information given in annotated tags is crucial.

There are other choices for annotation, such as the ability to add a message for the project. To create the annotated tag, we use the following syntax:

```
git tag <tag name> -m <tag message>
```

***Figure 3.110**: Git Annotated Tag*

Users can tag the project stating the message and verify if the project is tagged successfully using the **`git show`** command:

```
→ Git_101 git:(main) git tag 1.1.0 -m "Tagging to mark 1.1.0 release"
→ Git_101 git:(main) git show 1.0.0
tag 1.1.0
Tagger: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Thu Dec 1 12:10:25 2022 +0530

Tagging to mark 1.1.0 release
  commit fdad70b728f67241fd97ae498d270f4feb1a61b1 (HEAD -> main, tag: 1.1.0, tag: 1.0.0, origin/main,
  origin/HEAD)
Merge: efd4ea4 cdb75e0
Author: Sumit Jaiswal <justjais@gmail.com>
Date:   Mon Jun 27 14:48:16 2022 +0530

  Merge pull request #2 from justjais/test_workPR

  Adding text to pull_101 readme file
```

***Figure 3.111**: Git Annotated Tag working example*

- **List Tags**

  The most popular choice for listing all the repository's accessible tags is via **`git tag`** command:

```
→ Git_101 git:(main) git tag
 1.0.0
 1.1.0
```

***Figure 3.112**: Git Tag example*

- **Git Push Tag**

  Tags can be pushed to a GitHub server project. Other team members will benefit from knowing where to choose an update.

  On a GitHub server account, it will appear as a release point. The **`git push`** command makes it easy to push tags by providing a few particular arguments.

- **Git push origin**

  To push any particular tag by using the **`git push`** command, we can use the following syntax:

```
git push origin <tagname>
```

***Figure 3.113**: Git push origin Tag*

- **Git Delete Tag**

  A tag can always be removed from the repository using Git. Use the command listed below to delete an existing tag:

```
git push --d <tagname> # or,
git push --delete <tagname>
```

***Figure 3.114**: Git delete Tag*

We can also delete multiple tags using the following command:

```
git push -d <tagname 1> <tagname 2>
```

***Figure 3.115**: Git delete multiple Tag*

In conclusion, tagging is an extra method to take a snapshot of a Git repository. Tags are the semantic version numbers that serve as identifying tags and correlate to software release cycles. Creation, modification, and deletion of tags are primarily controlled by the Git tag command.

Annotated and lightweight tags are the two different categories. Annotated tags are often preferable since they hold more useful meta-information about the tag. The Git commands, **`git push`** and **`git checkout`** were also addressed in this chapter.

To collaborate over repository

Git collaboration is done via available Git commands: Fetch, Pull, and Push as shown in the following figure:

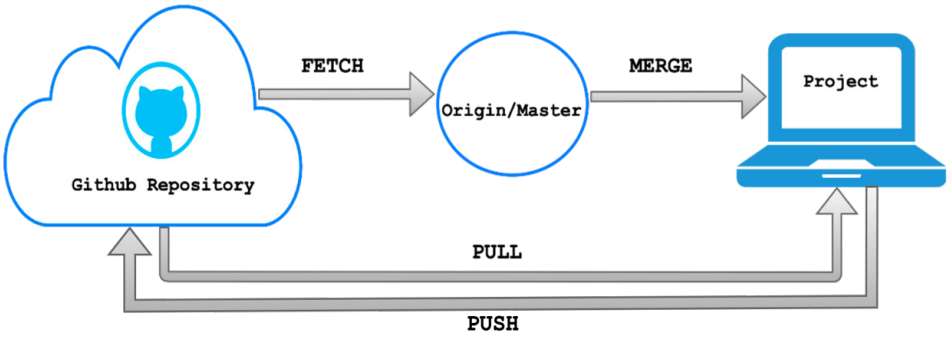

Figure 3.116: Git collaboration Fetch, Pull, Push

fetch

This command downloads the objects and refs. (A Git reference (git ref) is a file that contains a Git commit SHA-1 hash) From another Git repository, Git *"fetch"* is a command that downloads commit, objects, and references from another repository. It pulls tags and branches from one or more repositories.

It contains repositories as well as the objects required to complete their histories to keep remote-tracking branches up to date.

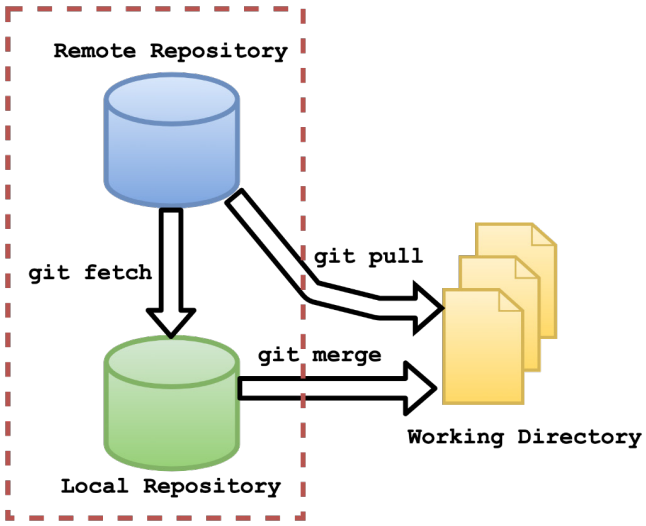

Figure 3.117: Git fetch

Options:

```
git fetch [<options>] [<repository> [<refspec>...]]
git fetch [<options>] <group>
git fetch --multiple [<options>] [(<repository> | <group>)...]
git fetch --all [<options>]
```

***Figure 3.118**: Git fetch CLI options*

The update from remote-tracking branches is pulled using the **`git fetch`** command. We may also download updates that have been pushed to our remote branches onto our local development machine.

As we all know, a branch is a subset of our repository's core code. Thus, remote-tracking branches are those that have been configured to pull and push from a remote repository.

Git fetch remote repository:

We can use the fetch command to get the whole repository from a repository URL, just like we can with the pull command.

```
git fetch <repository URL>
```

***Figure 3.119**: Git fetch syntax*

Run output:

```
➜  Git_101 git:(main) git fetch https://github.com/justjais/Git_101.git
From https://github.com/justjais/Git_101
 * branch            HEAD       -> FETCH_HEAD
```

***Figure 3.120**: Git fetch output*

The entire repository was retrieved from a remote URL in the above result, as the main branch is already up to date, fetch command did not update the local repository.

Git fetch specific branch:

From a repository, we may retrieve a specific branch. It will only use a certain branch to access the element. Take a look at the following output:

```
git fetch <branch URL> <branch name>
```

***Figure 3.121**: Git fetch over specific branch*

Run output:

```
➜  Git_101 git:(main) git fetch https://github.com/justjais/Git_101.git dev_branch
remote: Enumerating objects: 34, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 19 (delta 13), reused 7 (delta 4), pack-reused 0
Unpacking objects: 100% (19/19), 3.37 KiB | 93.00 KiB/s, done.
From https://github.com/justjais/Git_101.git
 * branch            HEAD       -> FETCH_HEAD
```

***Figure 3.122**: Git fetch output over specific branch output*

The specific branch **`dev_branch`** in the report has fetched data from a remote URL.

Git fetch all branch:

The Git fetch command allows you to get all branches from a remote repository at the same time.

```
git fetch --all
```

***Figure 3.123**: Git fetch all*

Run output:

```
➜  Git_101 git:(main) git fetch --all
Fetching origin
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 625 bytes | 312.00 KiB/s, done.
From https://github.com/justjais/Git_101
   6c30e21..9a1b8ff  main       -> origin/main
```

***Figure 3.124**: Git fetch all output*

All the branches have been fetched from the **`Git_101`** repository.

Git fetch to synchronize local repository

Assume that a member of your team has merged some new functionality to your remote repository.

Use the **`git fetch`** command to add these updates to your local repository. The following fetch command help update your local repository with remote branch changes and updates.

```
git fetch origin
```

***Figure 3.125**: Git fetch origin*

Run output:

```
➜  Git_101 git:(main) git fetch --all
remote: Enumerating objects: 34, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 19 (delta 13), reused 7 (delta 4), pack-reused 0
Unpacking objects: 100% (19/19), 3.37 KiB | 93.00 KiB/s, done.
From https://github.com/justjais/Git_101
   46b06f9..433bc48  main             -> origin/main
 * [new branch]      dev_branch -> origin/dev_branch
```

***Figure 3.126**: Git fetch output*

New features of the remote repository have been updated to my local system in the presented output. The branch **`dev_branch`** and associated objects are added to the local repository in this output. Git fetch can retrieve from a single named repository or URL, or many repositories at the same time.

It is possible to think of it as a safe version of the **`git pull`** commands. The **`git fetch`** command downloads the remote material but does not change the working state of your local repo. It will fetch the origin remote by default if no remote server is provided.

Pull

The **`git pull`** command downloads and merges changes from a remote repository into the local repository.

`Git pull` is a command that combines the commands **`git fetch`** and **`git merge`**. When receiving data from GitHub, the word "pull" is used. It gets changed from

the remote server and merges them into your local working directory. To pull a repository, use the **`git pull`** command.

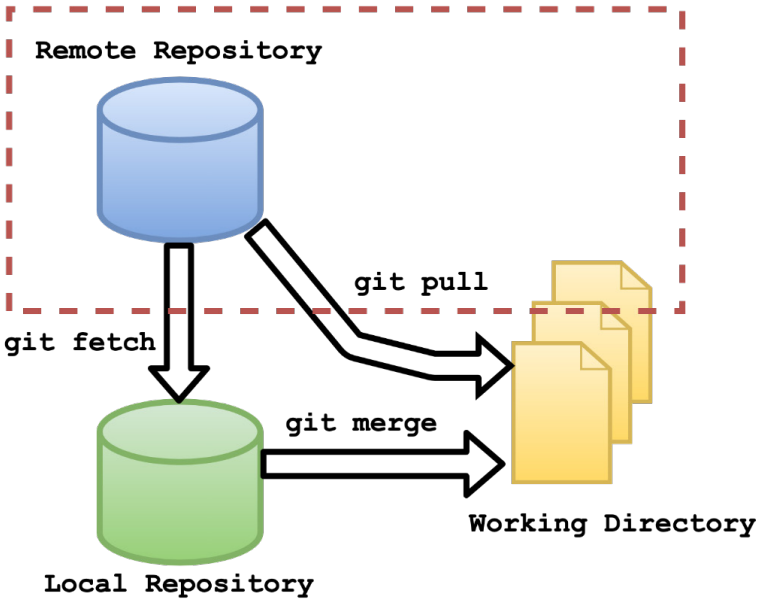

***Figure 3.127**: Git Pull request*

A pull request is a way for the developer to update the rest of the team that they have completed a feature.

The developer submits a pull request using their remote server account once their feature branch is complete. The pull request informs all team members that they must evaluate and integrate the work into the master branch.

The following figure shows how pull works across different places and how it differs from other related operations:

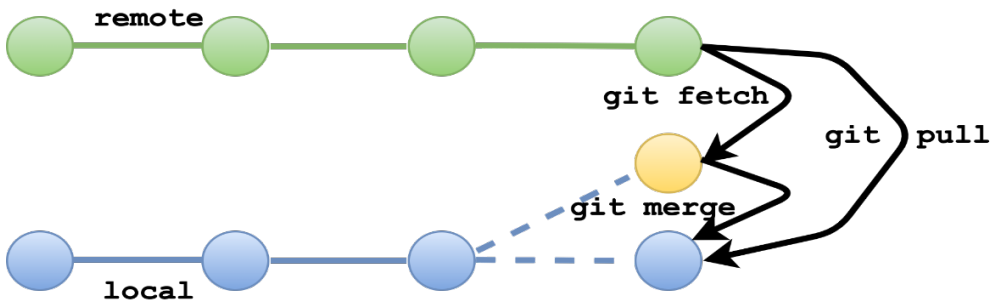

***Figure 3.128**: Git Pull request example view*

Push

The Git push command syncs the contents of the local and remote repositories.

The opposite of fetching is pushing. Git push exports the material to the remote branches, whereas git fetch imports it to the local branches.

The term "push" refers to the process of copying material from a local repository to a distant repository.

The act of pushing commits from your local repository to a remote repository is known as pushing. Pushing has the potential to overwrite modifications; care should be exercised when pushing.

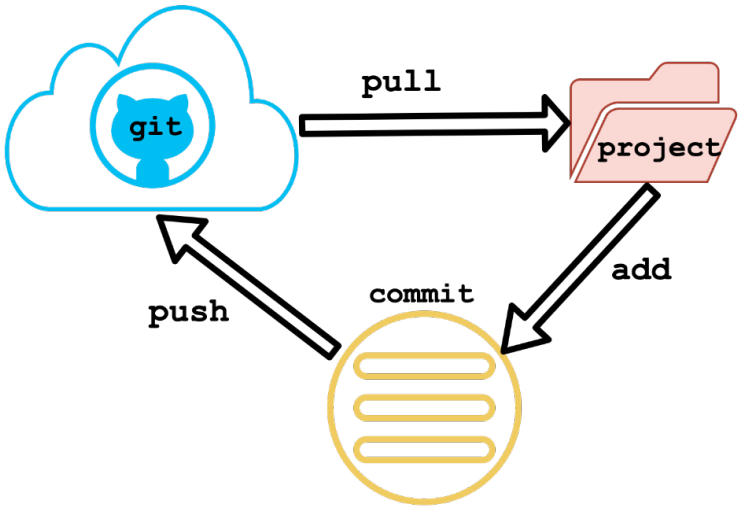

***Figure 3.129**: Git Push*

Options :

Pushing into the repository is done with the **`git push`** command. Push may be thought of as a mechanism for transferring commits between local and remote repositories.

***Figure 3.130**: Git Push options*

- **`--all`**

  When used, this will push refs for all branches.

- **`--prune`**

  It gets rid of remote branches that do not have a local equivalent.

This means that if a remote branch, such as a dev_branch, does not exist locally, it will be deleted.

- **`--force`**

  Even if the result is a non-fast-forward merging, the push is forced. Before using the, **`--force`** option make sure none have pulled the commits.

- **`--mirror`**

  It is used to replicate the repository to a remote location. Local references that have been updated or generated will be pushed to the remote end. On the remote end, it may be forced to update.

  The remote end will be cleared of the deleted references.

- **`--dry-run`**

  The commands are being tested by Dry Run. All this is done except for the repository original update.

- **`--tags`**

  To push all local tags.

- **`--delete`**

  It deletes the specified remote branch.

Git push origin main:

Git push origin main is a command-line method that allows you to specify a remote branch and directory. This command aids you in selecting your primary branch and repository when you have several branches and directories.

The word origin often refers to the remote repository, whereas main is the primary branch. As a result, the complete command **`git push origin main`** moves the local content to the remote location's main branch.

```
git push origin main
```

***Figure 3.131**: Git Push origin*

Let us understand this via working scenario. We will first verify the branch status using **`git status`** command to check if repository status is clean and then make update in repository content.

```
→  Git_101 git:(main) git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

***Figure 3.132**: Git status check*

Once verified that branch has no pending changes, we can move to the step where we can modify the file or do required changes in repository content.

```
→  Git_101 git:(main) mv learn_git.txt temp
→  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    learn_git.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        temp/learn_git.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

***Figure 3.133**: Git status check*

Here, we will move the **`learn_git.txt`** file to **`temp`** folder and once done we can verify the same using **`git status`** command which confirms the expected change.

```
→  Git_101 git:(main) ✗ git add .
→  Git_101 git:(main) ✗ git commit
[main b1b4cca] move file dir
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename learn_git.txt => temp/learn_git.txt (100%)
```

***Figure 3.134**: Git commit*

Now, as we want to commit the changes to the remote repository, we will use **`git add .`** which stages multiple files in one step. We are trying to make two changes in this scenario, moving the file from one directory to another and removing the file from the existing directory. We can now simply commit to the changes.

In the local repository, the file is moved to the new directory and is completely tracked. We can now push it to origin main as:

```
→  Git_101 git:(main) git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 388 bytes | 388.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/justjais/Git_101.git
   febe99d..b1b4cca  main -> main
```

***Figure 3.135**: Git push origin*

Git force push :

You can use Git force push to push a local repository to a remote server without dealing with conflicts.

```
git push <remote branch> -f
or, git push <remote branch> --force
```

***Figure 3.136**: Git push --force*

The **-f** suffix is used to abbreviate the word force. The branch may be of any branch name, and the remote can be any remote site like GitHub, Subversion, or any other git service. We may use **git push origin main -f** as an example.

Both the remote and the branch can be omitted.

When both the remote and the branch are missing, the default action is decided by the git config variable push.default.

```
git push --force
```

***Figure 3.137**: Git push --force*

Forcing a repository to be pushed has several effects, one of which is that it may replace work that you wish to preserve. If there are fresh commits on the remote that you did not expect, the force pushing with a lease option can cause the push to fail.

If we think of it in terms of Git, we can argue that if the remote includes untracked commits, it will fail.

```
git push <remote branch> --force-with-lease
```

***Figure 3.138**: Git push –force-with-lease*

Conclusion

This is the end of this chapter and the contents we discussed in this chapter are the building block and soul of the Git version control system. The more fluent you get will all the discussed terms and their respective usages and application, the efficient you will become in using Git and GitHub.

In the next chapter, we will continue to build on the knowledge we gained in this chapter and discuss a few of the Git concepts in more depth.

Multiple choice questions

1. **How to delete git branch locally?**
    a. git push --force
    b. git push origin –delete <branch name>
    c. git branch –D <branch name>
    d. git pull origin <branch name>

2. **Choose the correct syntax for Git force push which fails when there are untracked commits?**
    a. git push --force
    b. git push –force-with-lease
    c. git push origin –delete <branch name>
    d. git branch –D <branch name>

3. **Git command to move files from the working tree and from the index?**
    a. delete
    b. push
    c. rm
    d. mv

4. **Git command to switch between two git branches?**
    a. rebase
    b. switch
    c. merge
    d. init
5. **Git downloads remote git repo locally.**
    a. Init
    b. clone
    c. pull
    d. mv
6. **Git collaborate commands?**
    a. Bisect, Diff, Log
    b. Add, Remove, Restore
    c. Fetch, Pull, Push
    d. Merge, Commit, Rebase

Answers

1. **c**
2. **b**
3. **d**
4. **b**
5. **b**
6. **c**

Key terms

- Start a working area
    - init
    - clone
- Work on the current change:
    - add

  - mv
  - restore
  - rm
  - sparse-checkout
- Examine the history and state of the repository:
  - bisect
  - diff
  - grep
  - log
  - show
  - status
- Grow, mark and tweak your repo history:
  - branch
  - commit
  - merge
  - rebase
  - reset
  - switch
  - tag
- Collaborate over repository:
  - fetch
  - pull
  - push

Points to remember

- Git is a version control system of type Distributed version control system.
- **Git** and **GitHub** are two different entity which work hand-in-hand to give way to a centralized development process.
- Default branch of a GitHub repo is *"main"* and all the Pull request/PR that gets merged to **GitHub** repo gets merged too *"main"* branch.

Further reading

For more history and reference around the discussed topics in this chapter, you can check out the Git official documentation for getting started and GitHub documentation:

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 4

Deleting, Renaming, and Ignoring Files in Git

Now, that you have GIT and GitHub up and running on your system let us understand why Git and GitHub are required. We will also understand its importance and relevance when it comes to open-source way of development or development across teams and location in general.

This chapter is divided into sub-topics which are inter-related and follows a flow which makes it easier for the readers to grasp on to the background and concept in an informative and simpler way. This chapter builds on what we learned in the previous chapter and allows you to make a final decision before pushing and committing changes to source control.

Structure

In this chapter, we will cover the following topics:

- Delete the Git file
- Git rm cached
- Git rename files
- Git branching
- Ignoring the files using .gitignore
- Git commit : save the staged changes

Objectives

This chapter builds on the process of committing changes to the GitHub repo that may entail renaming, deleting, or ignoring files in the project. All these ideas will be covered in full in this chapter.

This chapter assumes that you are up and running with Git on your Linux, Windows, or Mac machines. The examples in this chapter were run on a Mac, but they should work similarly on a Linux or Windows machine.

Delete the Git file

The term rm stands for remove in Git. It is used to delete a single or a group of files. Git rm's primary job is to delete tracked files from the Git index. It can also be used to delete files from the working directory as well as the staging index. An illustration of Git delete can be seen in the following figure:

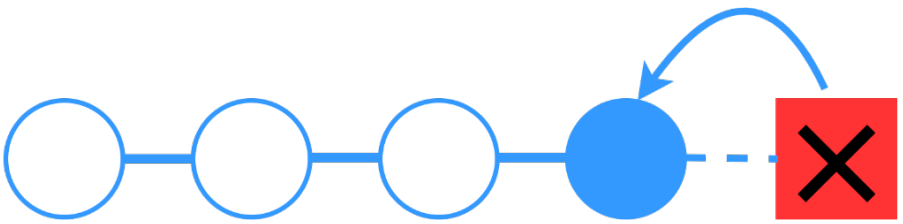

***Figure 4.1**: Git Delete*

The files that are being removed must be appropriate for the branch. In the index, no changes to their content can be staged. Otherwise, the removal process can be difficult, and it may not occur at all. However, the *–f* option can be used to force it.

Options

Git delete command line options are as follows:

```
git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch]
                [--quiet] [--pathspec-from-file=<file> [--pathspec-file-nul]]
                [--] [<pathspec>...]
```

***Figure 4.2**: Git Delete command line options*

We will be discussing the most used Git rm options as follows:

- `-f`, `--force`

Git's safety check to ensure that the files in HEAD match the current content in the staging index and working directory is overridden using the **`-f`** option.

- **`-n`**, **`--dry-run`**

  The **`dry run`** option provides a safety net that runs the **`git rm`** command but does not remove the files. Simply put, it will display the files that would have been deleted.

- **`-r`**

  When using **`git rm`** in recursive mode, it will remove a target directory and all of its contents. When a leading directory name is specified, **`-r`** tag allows recursive removal.

- **`--cached`**

  The cached option specifies that only the staging index should be removed. The files in the working directory will be left alone even if modified.

- **`--ignore-unmatch`**

  Even if no files were found, this command will terminate with a 0 sigterm unix status. The value 0 indicates that the command was successfully executed.

  When using **`git rm`** as part of a larger shell script that needs to fail gracefully, the **`-ignore-unmatch`** option can be very useful.

- **`-q`**, **`--quiet`**

  For each file removed, **`git rm`** usually prints one line (in the form of a **`rm`** command). This setting disables the output.

Examples

It is easy to remove files from our repository; simply delete them and commit.

`learn_git.txt`, **`temp_1.txt`**, **`temp_2.txt`**, **`rm_temp.txt`** are the four files in the repository.

```
➜  Git_101 git:(main) ls temp
__init__.py   learn_git.txt rm_temp.txt   temp_1.txt    temp_2.txt
➜  Git_101 git:(main)
➜  Git_101 git:(main) git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
➜  Git_101 git:(main)
```

***Figure 4.3**: Git status check*

Now, we want to delete the **rm_temp.txt** file from temp directory:

```
→  Git_101 git:(main) ls temp
 __init__.py  learn_git.txt  rm_temp.txt  temp_1.txt  temp_2.txt
→  Git_101 git:(main) git rm rm_temp.txt
 rm 'temp/rm_temp.txt'
→  Git_101 git:(main) ✗ git status
 Your branch is up to date with 'origin/main'.

 Changes to be committed:
    (use "git restore --staged <file>..." to unstage)
  deleted:  rm_temp.txt
```

***Figure 4.4**: Git status once file is deleted*

However, the **rm_temp** file is still there in our repository:

```
→  Git_101 git:(main) git commit -am "rm_temp deleted"

 [main 7a4caa8] rm_temp deleted
  1 file changed, 19 deletions(-)
  delete mode 100644 rm_temp.txt
```

***Figure 4.5**: Git commit the changes with message*

It is no longer there, and the **rm_temp** has been deleted.

So, deleting a file from a repository is as simple as deleting and committing.

Git rm cached

At times, you may want to remove files from Git but keep them in your local repository. In other words, you do not want your file to be shared on Git.

Git allows you to do so. The cached option is used in this scenario.

It specifies that the removal will only affect the staging index and not the repository. Let us say we wish to remove a file from Git. We will use the **git rm** command with the cached option to remove **rm_temp.txt**.

The file will be deleted from the version control system but can still be tracked in the repository using the appropriate command. It can also be added to the version control system again. Use the **status** command to check the file's status:

```
→  Git_101 git:(main) git rm --cached rm_temp.txt

rm rm_temp.txt

→  Git_101 git:(main) git status

On branch main

Changes to be committed:

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

        deleted: rm_temp.txt

Untracked files:

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

        rm_temp.txt
```

***Figure 4.6**: Git rm cache*

The **`rm_temp.txt`** file is erased from the repository / version control system, but may be tracked in the repository, as seen in the above report.

Undo before Commit command

The **`git rm`** command is not permanent and can be undone after it has been run. The most common and straightforward method to do so is to use the **`git reset`** command.

These changes will not be saved until the repository is updated with a new commit.

The following is how the **`git reset`** command will be used:

```
→  Git_101 git:(main) git reset HEAD

or,

→  Git_101 git:(main) git reset --hard
```

***Figure 4.7**: Git reset*

The above instruction will return the head to its original position.

As a result, it will revert to its prior position. Consider the following result:

```
➜  Git_101 git:(main) ✗ git reset HEAD
➜  Git_101 git:(main) git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
➜  Git_101 git:(main)
```

***Figure 4.8**: Git reset status*

Only the current branch is affected by **`git rm`**. The working directory and staging index trees are the only ones that get removed. Until a new commit is made, it is not saved in the repository history.

The following tables shows the difference between **`git revert`** and **`git reset`**:

git revert	git reset
This command is used to create a fresh commit that undoes the prior commit's changes.	This command is used to undo any modifications made locally in the git repository.
Using this command adds a new history to the project without modifying the existing history.	This command operates on the commit history, git index, and the working directory.

***Table 4.1**: Git revert Vs Git reset*

Git rename files

There are two ways in which we can rename our files in the git repository which are as follows:

Method 1

1. In a working folder, *rename* a file. We would like to rename **`the temp`** to **`chapter04`**.
2. First, we rename the file in our working directory as follows:

```
➜  Git_101 git:(main) mv temp.txt chapter04.txt
➜  Git_101 git:(main) ls
chapter01.txt  chapter02.txt  chapter03.txt   chapter04.txt
```

***Figure 4.9**: Git rename files - 01*

3. With the above changes done, we can check the repository status using the **`git status`** command:

```
→  Git_101 git:(main) ✗ git status
On branch main
Changes not staged for commit:
  (use "git add/rm ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)

    deleted:    temp.txt

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

    chapter04.txt

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

***Figure 4.10**: Git rename files - 02*

4. The git believes that **`temp.txt`** has been removed and that **`chapter04.txt`** has been inserted.

5. On the repo, we should now remove **`temp.txt`** and add **`chapter04.txt`**:

```
→  Git_101 git:(main) ✗ git rm temp.txt
rm 'temp.text'
→  Git_101 git:(main) ✗ git add chapter04.txt
```

***Figure 4.11**: Git rename files - 03*

6. Check the status again, and you will see that Git has figured out that all we did was rename the file!

```
→  Git_101 git:(main) ✗ git status
On branch main
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    renamed:    temp.txt -> chapter04.txt
```

***Figure 4.12**: Git rename files - 04*

7. Now, we can commit the changes to the repository as:

```
→  Git_101 git:(main) ✗ git commit -m "renamed temp to chapter04"
[main ed1m7f8] renamed temp as chapter04
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename temp.txt => chapter04.txt (100%)
```

***Figure 4.13**: Git rename files - 05*

Method 2

Another way of renaming the file is using **`git mv`** command, as:

```
→  Git_101 git:(main) git mv temp.txt chapter04.txt
```

***Figure 4.14**: Git rename files - 06*

Now when we check the status again, Git understands that we have renamed it:

```
→  Git_101 git:(main) ✗ git status
On branch main
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

   renamed:    temp.txt -> chapter04.txt
```

***Figure 4.15**: Git rename files - 07*

We can now just commit the changes with the commit message and check the status to verify if the commit is done as expected.

```
→  Git_101 git:(main) ✗ git commit -m "renamed temp to chapter04"
[main ed1m7q4] temp->chapter04
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename temp.txt => chapter04.txt (100%)
→  Git_101 git:(main)
→  Git_101 git:(main) git status
On branch main
nothing to commit, working directory clean
```

***Figure 4.16**: Git rename files - 08*

Git branching

Git branches are an essential element of any developer's workflow. Branches are a reference to a snapshot of the Git changes you have made. Branching can help tidy up the history of changes made before combining/merging them into the main branch. Branches represent a single development item and can be used to request a new working directory, staging area, or project history.

The segregation of development for different features in different branches allows you to work on them simultaneously, while keeping the main branch free of problematic code.

The git branch command creates, lists, and deletes branches, but does not allow you to switch between them or reassemble a splintered history.

Local vs remote Git branch

A local branch is a snapshot that exists on the developer's system. Only the local user has access to it. A remote branch is a branch that is located at a remote location which can be accessed by multiple users. A locally cached copy of a remote branch is referred to as a remote tracking branch.

The **`git push`** command with the **`-u`** option can be used to push a newly made branch to a remote repository. This will establish a remote tracking branch on your machine.

Using the **`git fetch`** or **`git pull`** commands, update and sync the remote-tracking branch with the remote branch.

Using the **`git fetch`**/**`git merge`** or **`git pull`** commands, update and sync the local remote-tracking branch with the remote branch.

A local tracking branch tracks another branch locally. The majority of local tracking branches follow a remote-tracking branch.

When using **`git push -u`** to push a local branch to the origin, the local branch <NewBranch> follows the remote-tracking branch <origin/NewBranch>.

When working on a project with a team, it is important to rename or remote branches in Git.

Let us now rename a local branch:

1. Use the **`git branch`** command with the **`-m`** option to rename the local branch to the new name as:

```
→ Git_101 git:(main) git branch -m <old-name> <new-name>
```

***Figure 4.17**: Git branch syntax*

2. Use the following command to delete the old branch on remote (assuming the remote name is origin, which is by default):

```
→ Git_101 git:(main) git push origin --delete <old-name>
```

***Figure 4.18**: Git delete old branch - 01*

3. Alternatively, you can speed up the process of deleting the remote branch by doing something like this:

```
→ Git_101 git:(main) git push origin :<old-name>
```

***Figure 4.19**: Git delete old branch - 02*

4. Now, we have to push the new name of the repost that we want to update:

```
→ Git_101 git:(main) git push origin <new-name>
```

***Figure 4.20**: Git push the new name*

5. Use the **`-u`** parameter with the **`git push`** command to reset the upstream branch for the

   new-name local branch:

```
→ Git_101 git:(main) git push origin -u <new-name>
```

***Figure 4.21**: Upstream new name via Git push*

Working of Git commit

The local repository is where Git snapshots are committed.

Git allows you to collect commits in a local repository rather than making a change and instantly committing it to the central repository. Splitting a feature across commits, grouping similar commits, and cleaning up local history before committing it to the central repository all have their advantages.

This also allows the developers to work in a private environment.

Ignoring the files using .gitignore

A .gitignore file indicates which files Git should ignore since they are intentionally untracked.

Git's file system is divided into three categories:

- **Tracked**: Tracked files are the files that have been previously staged or committed.
- **Untracked**: Untracked files are the files that have not been previously staged or committed.

Ignored: Ignored files are those that git has decided to ignore.

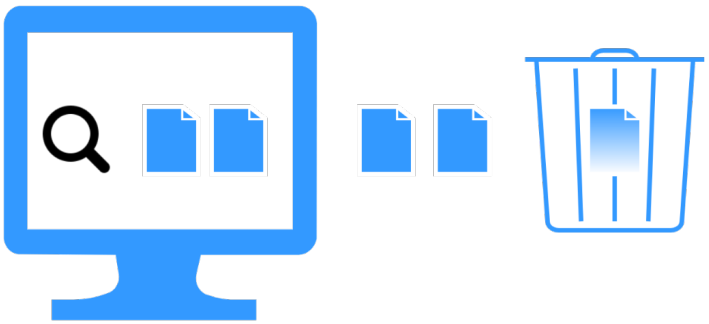

***Figure 4.22**: Git ignore*

User may not always wish to send files to a remote repository over GitHub. In Git, we can select which files to ignore. We must instruct Git to disregard those files.

Ignored files are usually remnants and machine-generated files. These files should not be committed unless they are derived from your repository source.

The following are some files that are frequently overlooked:

- Runtime files like log, lock, cache, or temporary files.

- Files with sensitive information, such as passwords or API keys.
- Compiled code, like file extension of **`.o`**, **`.pyc`**, and **`.class`** files.
- Build output directories, like: **`/bin`**, **`/out`**, or **`/target`**.
- Files generated at runtime, like: **`.log`**, **`.lock`**, or **`.tmp`**.
- Development IDE config files, like: **`.idea/workspace.xml`**.
- Hidden system files, like: **`.DS_Store`** or **`Thumbs.db`**

You can tell Git which files and directories to ignore when you commit by placing a **`.gitignore`** file in the root directory of your project.

Commit the **`.gitignore`** file to your repository to share the ignore rules with other users that clone it.

The .gitignore files

Git ignore files is a file or a folder that contains all of the files we want to ignore. Files that are not needed to complete the project are ignored by the developers. Many ignored files are created by Git itself and are almost always hidden. You can specify the ignore files in a variety of ways.

A **`.gitignore`** file in the repository's root folder can be used to keep track of the files that are ignored. To disregard a file, there is no specific command. When you have new files that you want to ignore, you must manually change and commit the **`+`** file.

There is no explicit **`git ignore`** command; instead, anytime you have new files to ignore, you must change and commit the **`.gitignore`** file manually.

Patterns in the **`.gitignore`** files are compared against file names in your repository to decide whether they should be ignored or not.

The .gitignore patterns, that is, file structure

Each line of the .gitignore file contains a pattern for which files or folders should be ignored. It matches filenames with wildcard characters using globbing patterns.

If you have files or folders that include a wildcard pattern, you can escape the character with a single backslash (`\`).

- Comments

  Comments are lines that begin with a hash mark (`#`) and are disregarded.

  Empty lines can be used to improve the file's readability and to group relevant pattern lines.

- Slash

  A directory separator is represented by the slash symbol (/). The slash at the start of a pattern refers to the directory where the **`.gitignore`** file is located. If the pattern begins with a slash, it only matches files and directories in the repository's root directory.

  The pattern matches files and folders in any directory or subdirectory if it does not begin with a slash. When a pattern ends in a slash, it only matches directories.

  All files and subdirectories in a directory are disregarded when it is ignored.

- Literal file names

  A literal file name with no special characters is the most straightforward pattern as shown in the following table:

Pattern	Example matches
`/access.log`	`access.log`

`access.log`	`access.log`
	`logs/access.log`
	`var/logs/access.log`

`build/`	`build`

***Table 4.2**: Literal file name*

- Wildcard symbols

  (*) - The asterisk symbol matches zero or more characters as shown in the following table:

Pattern	Example matches
`*.log`	**`error.log`** **`logs/debug.log`** **`build/logs/error.log`**

***Table 4.3**: Wildcard symbols - 01*

  (**) - Two adjacent asterisk icons denote any file or zero or more folders.

It only matches directories when preceded by a slash (/) as shown in the following table:

Pattern	Example matches
`logs/**`	Matches anything inside the log's directory.

`**/build`	`var/build`
	`/build`
	`build`

`foo/**/bar`	`foo/bar` `foo/*/bar`

***Table 4.4**: Wildcard symbols - 02*

(?) - The question mark matches any single character.

Pattern	Example matches

`access?.log`	`access0.log`
	`access0_1.log`
	`accessABC.log`

`temp??`	`temp0`
	`temp0_1`
	`tempABC`

`foo/**/bar`	`foo/bar` `foo/*/bar`

***Table 4.5**: Wildcard symbols - 03*

- Negating patterns

  Any file that was ignored by the previous pattern is negated (re-included) by a pattern that starts with an exclamation mark (!).

  The only exception is the re-inclusion of a file if its parent directory is excluded.

Pattern	Example matches
`*.log` `!error.log`	Files which will not be ignored: error.log or logs/error.log

***Table 4.6**: Negating Pattern table*

.gitignore sample

Following files are considered as **.gitignore** files:

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
# Ignore the node_modules directory
node_modules/
# Ignore Logs
logs
*.log
# Ignore the build directory
/dist
/temp*
# The file containing environment variables
.env
# Ignore IDE specific files
.idea/
.vscode/
*.sw*
```

***Figure 4.23**: Git ignore files*

Global .gitignore

We have already established that a project can contain multiple **.gitignore** files. However, Git enables us to create a global **.gitignore** file that can be used throughout the project. Run the following command on the terminal to generate a global **.gitignore**:

```
git config --global core.excludesfile ~/.gitignore_global
```

***Figure 4.24**: Git ignore global*

Global ignore rules are very handy for disregarding specific files that you never want to commit, such as sensitive data files or compiled executables.

Ignoring a previously committed file

You have a file called **bug_fix_sample.py** in your Git repository that you previously required but is not needed anymore, since you have changed the design of the implementation. Now, the fix is more efficient and passes all the required test cases as well.

Adding a file to the **.gitignore** file is an excellent approach to ignore untracked files, but you may override this by using **git add -f** on the file to force its inclusion in the index.

But how can you get Git to ignore committed files and stop them from showing up as available to commit every time you make a change?

Solution to the issue is putting the file to **.gitignore**:

- To begin, add the file to your **.gitignore** file
- The file will be removed from the index as a result of this action

(Specified by the **--cached** parameter). This also deletes it, but not from your hard drive:

```
git rm --cached sample/bug_fix_sample.py
```

***Figure 4.25**: Git ignore committed files*

- Using the following command will delete all files in the specified folder:

```
git rm -r --cached sample
```

***Figure 4.26**: Delete files from folder*

- If you want to verify the changes beforehand you can run the above command with **--dry-run** tag as:

```
git rm --dry-run -r --cached sample
```

***Figure 4.27**: Git rm dryrun command*

- Continue to make changes to other files and commit.

The ignored file will remain on your hard drive and will be ignored in the future:

```
git commit -m "Ignoring the required file"
```

***Figure 4.28**: Git commit ignored changes*

Stashing an ignored file

Git stash is a useful Git tool for temporarily stacking and undoing local changes so that you can apply them again later. As you might expect, by default, git stash ignores ignored files and only stores changes to Git-tracked files.

You can use the **`--all`** option with git stash to save changes to ignored and untracked files as well.

Debugging .gitignore File

At times it can be difficult to figure out why a file is being ignored, especially if you are working with multiple **`.gitignore`** files or complex patterns.

The **`git check-ignore`** tool with the **`-v`** option, that allows Git to display details about the matching pattern, is useful in this situation.

For example, to check why the **`/temp`** folder is ignored you would run:

```
git check-ignore -v /temp_project
```

***Figure 4.29**: Verify ignored files - 01*

In the path to the **`.gitignore`** file, the number of the matching line, and the actual pattern are all displayed in the output:

```
www/.gitignore:12:/temp*   /temp_folder
```

***Figure 4.30**: Verify ignored files - 02*

To get all of the ignored files, the **`--ignored`** option is used with **`git status`** command:

```
git status --ignored
```

***Figure 4.31**: Check all the ignored files*

You can use the **`.gitignore`** file to prevent files from being checked into the repository. The file contains globbing patterns that advise you which files and directories to ignore.

Git commit: save the staged changes

The **`git commit`** command saves all of the projects presently staged changes. Commits are used to document a project's current state. Git asks before modifying committed snapshots, so they are regarded as safe versions of a project.

The **`git commit`** command takes a snapshot of the current state of the project's changes. Committed snapshots are "secure" versions of a project that Git will never modify unless you specifically request it to. The git adds command used to promote or 'stage' changes to the project that will be saved in a commit before running the **`git commit`**.

In Git, when we add a file, it goes into the staging area. To fetch updates from the staging area to the repository, use the commit command. The staging and committing processes are intertwined. Staging allows us to continue making changes to the repository and committing allows us to record these changes in the version control system when we want to share them.

Git add is used before running **`git commit`** promoting changes to the project, which will then be saved in a commit. Git commit and **`git add`** are two of the most frequently used commands.

Commits are the project's snapshots. Every commit is saved in the repository's master branch. We can undo the commits or go back to an earlier version. Since each commit has its commit-id, two separate commits will never overwrite each other.

The **Secure Hash Algorithm** (**SHA**) algorithm generates this commit-id, which is a cryptographic number.

How Git commits differs from SVNs

A commit in SVN pushes changes from a local SVN client to a centralized shared SVN repository at a remote location.

Repositories are distributed in Git, Snapshots are committed to the local repository, and no interaction with other Git repositories are required. Git uses snapshots, whereas SVN keeps track of file differences. A diff is created and compared to the original file added to the repository in a svn commit. Every commit in Git saves the content of all the files. Git takes a snapshot of how the current file looks and saves a reference to it when committing or saving the project's state. Git does not store a file again if it has not changed.

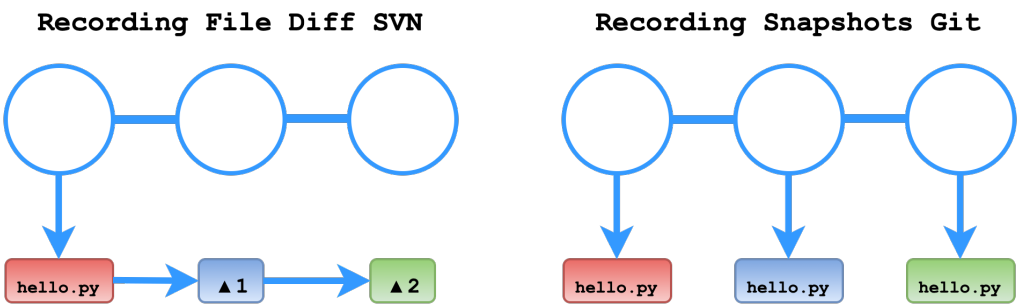

Figure 4.32: SVN commit Vs Git Commit

Git commits can be pushed to any remote repository at any time.

The commit models of SVN and Git are very distinct, but because of the common terminology, they are sometimes mistaken. If you are switching from SVN to Git, it is important to understand that commits are cheap in Git and should be utilized frequently.

Unlike SVN commits, which need a remote request, Git commits are performed locally with a more efficient algorithm.

By using the same set of settings, the **`--dry-run`** option can be used to acquire a summary of what is included by any of the respective **`git commit`** options for the next commit (options and paths).

Options

Git commit command line options are as follows:

```
git commit [<options>] [--] <pathspec>...
```

Figure 4.33: Git commit command line options

- **`-a`**, **`--all`**

  Commits a snapshot of the working directory's changes. Only changes to files that have been tracked are included.

- **`-m`**, **`--message`**

  Creates a commit using the commit message supplied. By default, **`git commit`** launches the locally configured text editor and prompts you to type a commit message.

- **`--interactive`**

  To add files interactively.

- **`-n`**, **`--no-verify`**

  To bypass pre-commit and commit-msg hooks.

- **`-am`**

  Creates a commit for all staged changes and takes an inline commit message by combining the **`-a`** and **`-m`** arguments.

- **`--amend`**

  Changes the previous commit. The preceding commit is updated with staged changes. This command modifies the previously provided commit message and opens the system's predefined text editor.

- **`--dry-run`**

  Does not create a commit, instead, displays a list of routes to be committed, pathways with local modifications that will be left uncommitted, and untracked paths.

Examples

The **`git101.txt`** file with altered content on the current branch is shown in the following example. To commit the staged snapshot of the file, use the **`git add`** command to stage it first:

```
→  Git_101 git:(main) git add git101.txt
```

***Figure 4.34**: Git add file*

The **`git101.txt`** file will be moved to the Git staging area after running **`git add`**.

To see the output, run the **`git status`** command.

```
→  Git_101 git:(main) git status
On branch main
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: git101.txt
```

***Figure 4.35**: Git status after adding file*

`git101.txt` will be saved with the next commit, according to the green status. The commit is made by running the **`git commit`** command as:

```
➜  Git_101 git:(main) git commit git101.txt -m "checkin the git101 file"
```

***Figure 4.36**: Git commit changes after adding file*

If there are multiple files to commit to stage, all the modified/new files, you can add them to the stage by running the command **`git add .`**, and once all the required files are staged you can commit them as:

```
➜  Git_101 git:(main) git commit
```

***Figure 4.37**: Git commit*

This results into commit window like:

```
updating two files
# Enter the commit message of your changes. Lines that start
# with '#' will be ignored, an empty message breaks off the commit.
# On branch main
# Changes needed to be committed:
# (use "git reset HEAD ..." to unstage)
#
#modified: git101.txt, temp.txt
```

***Figure 4.38**: Git commit changes output*

Here, you can enter the commit message (example, updating two files) and save the changes. This will commit the changes to your Git repository.

The first line of the commit message is commonly used as the subject line, similar to an email. The remainder of the log message is referred to as the body and is used to provide information about the commit change set. Many developers choose to use the present tense in their commit messages as well.

This makes them read more like repository actions, making many history-rewriting processes easier to understand:

```
➜  Git_101 git:(main) git commit -am "update and ammend"
```

***Figure 4.39**: Git commit update and amend*

A shortcut command for power users that combines the **-a** and **-m** parameters. This combination creates an inline commit message and quickly commits all of the staged modifications.

```
➜  Git_101 git:(main) git commit --amend
```

***Figure 4.40**: Git commit amend*

The commit command has a new level of functionality with **--amend** parameter. Passing this option modifies the previous commit. Staged changes will be applied to the previous commit rather than creating a new one.

This command will launch the system's preset text editor and prompt you to alter the commit message you previously specified.

The **git commit** command is one of Git's most important features. To pick the modifications that will be staged for the next commit, use the git add command first.

Then, using **git commit**, a snapshot of the staged changes along a timeline of a Git project's history is created. On the next page, you may learn more about how to use **git add**. The **git status** command can be used to see how the staging area and pending commits are doing.

Conclusion

This is the end of this chapter and the contents we discussed in this chapter are the building block and soul of the Git version control system. The more fluent you get will all the discussed terms and their respective usages and application, the more efficient you'll become in using Git and GitHub as well.

In the next chapter, we will continue to build on the knowledge we gained in this chapter and discuss a few of the GitHub concepts including the best practices for writing and raising the pull request for any open-source project over GitHub in more depth.

Multiple choice questions

1. **How to amend to git commit?**
    a. git commit -m
    b. git commit -am
    c. git commit --amend
    d. git add .

2. **How to add multiple files to git staged area**
    a. git commit -m
    b. git push --force-with-lease
    c. git branch –D <branch name>
    d. git add .

3. **Command to verify if the file removed using git rm is as expected?**
    a. git rm
    b. git commit
    c. git rm --dry-run
    d. git status

4. **Git command to check the status of files which are staged to the repository?**
    a. git commit
    b. git status
    c. git add .
    d. git commit --amend

5. **How to undo a bad commit that has already been pushed?**
    a. git add .
    b. git rm <filename>
    c. git commit
    d. git revert <commit name>

6. **How to remove the file from git index without actually removing it from the local file system?**
    a. git rm
    b. git stash
    c. git reset
    d. git commit

Answers

1. **b**
2. **d**
3. **c**
4. **b**
5. **d**
6. **c**

Key terms

- Git rename
- Git delete:
  - rm
- Gitignore files
- Git commit
- Git revert Vs Git reset

Points to remember

- To avoid making the same error again, instead of using the **`git rm`** command, we can use the **`git reset`** command to remove the file from the staged version and then add it to the **`.gitignore`** file.
- To revert a bad commit that is already pushed, a new commit can be created that reverts changes done in the bad commit. It can be done using **`git revert <name of bad commit>`**.
- It is preferable to create a new commit rather than amend an existing one:
  - It is acceptable only if the commit message is changed or destroyed, but, there may be instances where the contents of the commits are changed. As a result, vital information linked with the commit is lost.
  - Excessive use of **`git commit --amend`** can have serious consequences, since the modest commit amend might grow and accumulate unrelated modifications over time.
  - The command **`git reset --mixed`** is used to undo changes to the working directory and **`git index`**.

Further reading

For more history and reference around the discussed topics in this chapter, you can check out the Git official documentation for getting started and GitHub documentation:

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 5
Collaborating Towards Your/Other Larger Projects over GitHub

Now, that you have Git and GitHub up and running on your system, let us understand why Git and GitHub are required and its importance and relevance when it comes to open-source way of development or development across teams and location in general.

This chapter discusses all of the process-related and critical aspects that should be considered and followed before contributing to an open-source project that is followed and used by a larger community from all over the world; as well as how this differs significantly from maintaining and contributing to a single-user repository.

Structure

In this chapter, we will cover the following topics:

- Clone and fork the GitHub repository
- Why forking repository is needed
- Creating a Pull request from forked repository
- Contributing to single repository
- Collaborating on Pull request
- Git Aliases

Objectives

This chapter builds on what we learned in the previous chapter and allows you to make a final decision before pushing and committing changes to source control. This process of committing changes to the GitHub repo may entail renaming, deleting, or ignoring files in the project. All these ideas will be covered in detail in this chapter.

This chapter assumes that you are up and running with Git on your Linux, Windows or Mac machines.

Clone and fork the GitHub repository

Cloning is the act of creating a copy of any target repository in Git. The destination repository might be either remote or local. You can create a local duplicate of your repository by cloning it from the remote repository. You can also sync any two repositories.

A partial replica of a repository is a fork. When you fork a repository, you can freely test and debug modifications without harming the original project. The most common use of forking is to propose changes for bug fixes in remote repositories.

You will often work on repositories where you are neither the owner nor a collaborator. If you wish to do anything other than browsing these files, you will need to fork the repository.

This section defines forking, demonstrates the process, and contrasts forking with cloning and duplicating.

We will discuss about forking code and contributing to it. If the user has already made changes to a clone before forking, this section also shows you how to get the code you want to contribute into a fork. Before we get into cloning and forking details, we need to understand the difference between the respective terms and duplication.

Cloning, forking, and duplicating

You can make a local copy of the project on your computer when you clone a GitHub repository. A GitHub repository can be cloned by forking it, which copies it onto your GitHub.com account. The original GitHub repository and the one you forked will still be linked, enabling you to push changes you make to the original copy and pull changes made to the original repository into your copy.

A repository gets duplicated when a copy is created that no longer contains a connection to the original repository. Duplication makes it more challenging to submit changes back into the original GitHub repository, it is not typically a component of an open-source workflow.

However, there are situations when duplicating a repository might be practical, such as when the original project is no longer active, and you want to keep it going with your fork.

Cloning repository

A GitHub repository can be copied locally by cloning it. By doing this, you can avoid editing the source files of the original repository directly and instead make all of your changes locally. To clone a GitHub repository, you can either use:

- GitHub desktop client (**https://desktop.github.com/**) or
- Operating System command line

Here, I will be using command line for all the GitHub operation.

Let us consider that you want to clone a repository. You can either work on the project together or clone it for your learning.

Find that project on GitHub and select the **Code** drop-down to reveal the repository's URL as shown in the following screenshot:

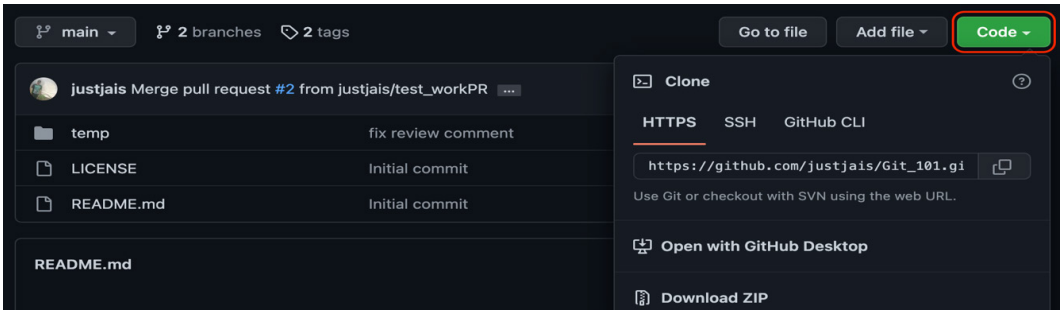

***Figure 5.1**: Get Github repo clone link via Code*

Any public repository can be cloned, allowing you to execute the code on your machine and edit it. However, if you do not have push rights to the remote repository, you would not be able to push those changes back to it.

Copy the repository's URL, then launch a terminal window and type the following command:

```
→ Self_test git clone https://github.com/justjais/Git_101.git
  Cloning into 'Git_101'...
  remote: Enumerating objects: 109, done.
  remote: Counting objects: 100% (109/109), done.
  remote: Compressing objects: 100% (66/66), done.
  Recremote: Total 109 (delta 32), reused 94 (delta 24), pack-reused 0
  Receiving objects: 100% (109/109), 22.41 KiB | 717.00 KiB/s, done.
  Resolving deltas: 100% (32/32), done.
```

***Figure 5.2**: Clone GitHub repo using code clone link*

Once a repository has been copied to your local machine, you can examine and edit its metadata in your terminal. Go to a directory where you have a GitHub repository in the terminal after opening it.

With the following command, you can find out where the remote/target repository is:

```
Git_101 git:(main) git remote -v
 origin https://github.com/justjais/Git_101.git (fetch)
 origin https://github.com/justjais/Git_101.git (push)
```

***Figure 5.3**: GitHub repo remote target information*

You should see the same origin URLs to retrieve and push if you had cloned the **`Git_101`** repository. You should be able to see that user ***"justjais"*** is the owner of the remote repository, not your username.

You will not get any information back if you attempt to use the command on a Git repository that lacks a remote origin (that is, one that is not hosted on GitHub.com or another remote location).

Forking repository

A repository is copied in a fork. To experiment with modifications without impacting the original project, fork a repository.

When you fork a repository, you create your own unique copy of it that is a part of an account belonging to a separate organisation or individual, and the modifications you make to the fork have no impact on the original repository unless you use a pull request to merge your changes back into the respective GitHub repository. Your fork of the GitHub repository is entirely independent of the source repository.

There are three main arguments for forking another person's repository:

- First, download a copy of the repo and play around with it. In that situation, you would clone the repository to your local hard drive using GitHub desktop/command line after forking the repository to your own account on the GitHub website.
- Secondly, using the fork and clone method will allow you to switch between branches in the forked repository and possibly feed edits back to the original repository using the Git system.
- Lastly, the ability to contribute to a project for which you do not have the write (that is, push) access is the other prime motivation for forking. When working on major open-source projects, many contributors—some of whom may not even be known to the repository owners—may be involved.

For forking a repository of your choice, go to the repo's home page and click the **Fork** button on the top right to fork the repository. Use **https://github.com/justjais/Git 101** to practice forking and contributing to a public repository if you would like.

Duplicating repository

You can use a specific command to clone a repository, mirror-push to the new repository to keep a mirror of it without forking it.

A repository without a working directory is known as a **`bare`** git repository. In essence, you can only change the contents in the **`.git`** folder of your ordinary git repository, not the checked-out files.

It is helpful for server-based repositories when no work should be taking place at all. This refers to a directory that is not used to create your project, run tests, or change files. Cloning, pushing, pulling, and fetching is still possible while using less disc space. Since it lacks a working directory, it performs add and commit poorly.

To duplicate GitHub repository, we need to perform the following steps via **`git`***:*

1. Create a bare clone of the repository:

```
Self_Test git clone --bare https://github.com/justjais/Git_101.git
 Cloning into bare repository 'Git_101.git'...
 remote: Enumerating objects: 109, done.
 remote: Counting objects: 100% (109/109), done.
 remote: Compressing objects: 100% (66/66), done.
 remote: Total 109 (delta 32), reused 94 (delta 24), pack-reused 0
 Receiving objects: 100% (109/109), 22.41 KiB | 132.00 KiB/s, done.
 Resolving deltas: 100% (32/32), done.
```

***Figure 5.4**: Github bare repo*

2. Mirror-push to the new repository:

```
→ Self_Test cd Git_101.git
→ Git_101 git:(main) git push --mirror https://github.com/justjais/Git-101-new.git
```

***Figure 5.5**: Github mirror push repo*

3. Remove the local temporary repository you set up in the earlier step:

```
→ Git_101 git:(main) cd ..
→ Self_Test rm -rf Git_101
```

***Figure 5.6**: Delete the old GitHub repo*

We can also mirror a GitHub repository to another location by following these steps:

1. Make a repository bare-metal mirrored clone:

```
→ Self_Test git clone --mirror https://github.com/justjais/Git-101.git
```

***Figure 5.7**: Bare metal clone*

2. Update the push location to your mirror:

```
→ Self_Test git clone --mirror https://github.com/justjais/Git-101.git
→  Git_101 git:(main) git remote set-url --push origin https://github.com/justjais/mirrored
```

***Figure 5.8**: Bare metal clone*

A mirrored clone includes all remote branches and tags, just like a bare clone does, but all local references are replaced every time you fetch, keeping the copy identical to the original repository.

Pushing to your mirror is made easier by setting the URL for pushes.

3. Update the push location to your mirror:

```
→ Git_101 git:(main) git fetch -p origin
→  Git_101 git:(main) git push --mirror
```

***Figure 5.9**: Mirror repo push location*

Why forking repository is needed

Adding code to repositories where you are not the owner, or an explicit collaborator is an important part of the GitHub process and mission. As the goal of open source is to promote cooperation among software professionals all over the world.

You can contact the repository owner and ask to be a collaborator to join an open-source project. However, since adding you as a collaborator would grant you push permissions to the repository, the owner is unlikely to do so if they are unaware of your identity.

As a contributor, you will first need to earn the repository owner's trust. However, to fork a repository, you do not need any permission. To demonstrate how you can benefit the project, you can make your contributions and share them with the owner.

Go to the repo's home page and click the **Fork** button in the top right to fork the repository. To begin with, you can fork the Git_101 repo, ref: **https://github.com/justjais/Git_101**, to test forking and contributing to a public repository.

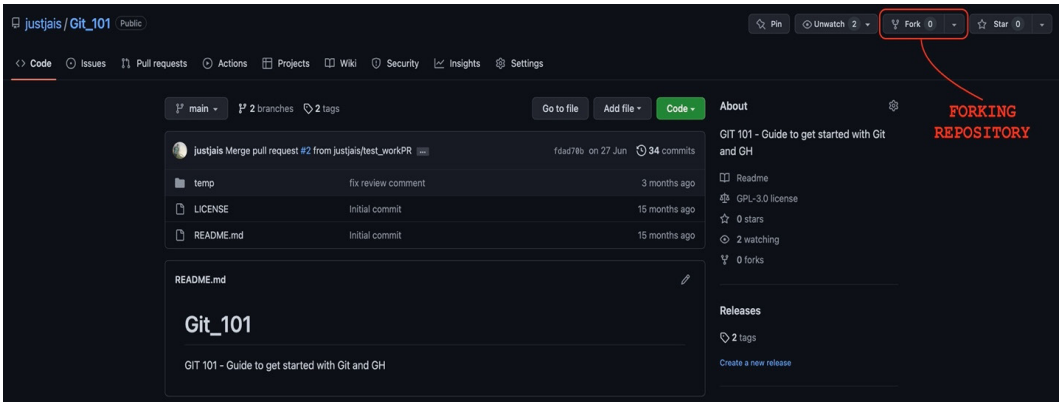

***Figure 5.10**: Forking GitHub repo push button*

Once you click on the **Fork** button, you will be taken to a page where you can provide information against the repo you are trying to fork, as shown in the following figure:

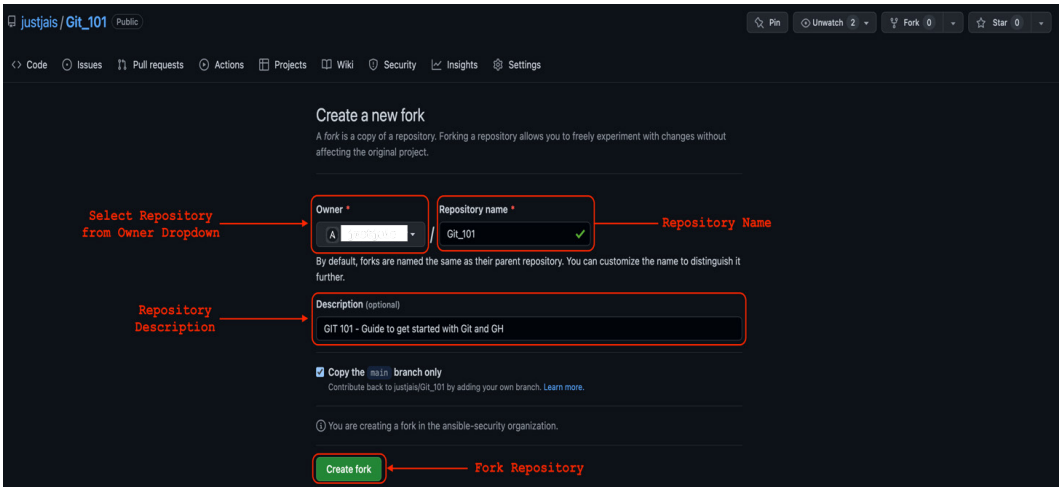

***Figure 5.11**: Forking GitHub repo details page*

After filling in the required details for forking the parent repository, the user can click on **the Create fork** button to go ahead with the forked repository creation. The forked repository once created will appear as shown in the following screenshot:

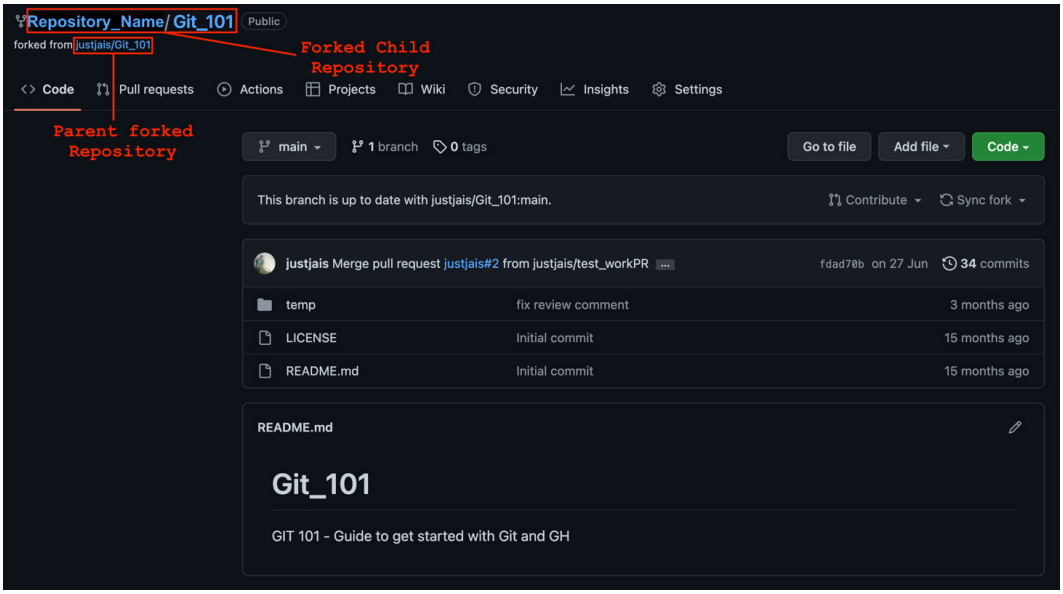

***Figure 5.12**: Forked GitHub Child repo*

Once you have your forked copy of the repository, you can clone it on your local computer to begin editing it.

Creating a Pull request from forked repository

Those who are unfamiliar with distributed version control systems like Git may find the idea of having it remote to be puzzling. When you clone a GitHub repository, the entire repository is present on your local computer. You may be tempted to think that the copy of the repository on GitHub is the official one. Git, however, does not have the concept of official. The official copy is whatever the project team decides.

Git does support the idea of a remote file, which is the pointer to a different location's hosting of the same Git repository.

A remote can also be a path to the location that contains a copy of the repository; often, remote is a URL to a Git-hosting platform like GitHub. Git adds a remote named origin with the location (often a URL) from where you cloned a repository when you do so. However, you can add more than one remote to a Git repository to represent other places from where you might want to push and pull updates.

For instance, you might wish to have a remote named upstream that points to the original repository if you clone a fork of a repository. You might wish to set the upstream remote if you clone the repository using the command line.

Run the **`git remote -v`** command in the directory where you cloned the repository to see both the forked remote origin and original remote upstream:

```
→ Git_101 git:(main) git remote -v
 origin https://github.com/<your github ID>/Git_101.git (fetch)
 origin https://github.com/<your github ID>/Git_101.git (push)
```

***Figure 5.13**: Forked GitHub repo remote config*

Now, to add the upstream parent repo to this forked repository we use the following command:

```
→ Git_101 git:(main) git remote add upstream https://github.com/justjais/Git_101.git
→ Git_101 git:(main) git remote -v
 origin https://github.com/<your github ID>/Git_101.git (fetch)
 origin https://github.com/<your github ID>/Git_101.git (push)
 upstream https://github.com/justjais/Git_101.git (fetch)
 upstream https://github.com/justjais/Git_101.git (push)
```

***Figure 5.14**: Add Parent forked GitHub repo as upstream remote repository*

Your username is stored in the origin, which is where you fetch/pull changes from and push changes to. (**<your github ID>** in this example).

The upstream holds the username of the original author, which is where the original code is located and where you eventually want to contribute the code (***justjais*** in this example).

Contributing to single repository

Your forked repo must be linked to the upstream repo. To ensure that you get the most recent version when you make modifications, you should be able to retrieve or pull any updates made to the original code into your own.

For instance, imagine you cloned and forked a *website* project a week ago to change the *Product* page. Someone else altered the *Product* page while you were working on those alterations.

Their changes can clash with yours, or they might add something fresh that you wish to incorporate into your adjustments. It makes it reasonable to pull those modifications into your repository before submitting your own to the original repository. It increases the likelihood that the owner will accept your modifications and decreases the likelihood that they will conflict with those others are making to the *Product* page.

```
→ Git_101 git:(main) git fetch upstream
 remote: Enumerating objects: 7, done.
 remote: Counting objects: 100% (7/7), done.
 remote: Compressing objects: 100% (3/3), done.
 remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
 Unpacking objects: 100% (3/3), done.
 from https://github.com/justjais/Git_101.git
```

***Figure 5.15**: Update the forked repo with upstream changes*

Now, to pull any upstream changes that might not be updated to local repository we can run the following command:

```
→ Git_101 git:(main) git pull -v --progress upstream main
 POST git-upload-pack (321 bytes)
    From https://github.com/justjais/Git_101
     * branch            main       -> FETCH_HEAD
     * [new branch]      main       -> upstream/main
    Already up to date.
```

***Figure 5.16**: Pull upstream branch changes to local repository*

Once we have pulled all the latest available changes from the parent repository to our local repository, we can check out a new branch for creating a pull request from the local repository as:

```
→ Git_101 git:(main) git checkout -b new_branch
 Switched to a new branch 'new_branch'
→ Git_101 git:(new_branch)
```

***Figure 5.17**: Checkout new branch*

Forgetting to fork a repository before attempting to contribute is a typical mistake people make. One example of how you might end up in this circumstance is described in the scenario that follows.

You clone a repository onto your local machine, edit the source code, push your changes to main, and you are done. However, a sinister-looking error message follows.

You are informed by the error message that you lack the authorization to push to this repository. The repository should have been forked first.. Making all of your modifications in a branch is a smart approach, as we advise in the book. To correct this error, fork the repository, modify the remote URLs for your local repository to link to your fork, and then push your changes.

These steps can assist you get out of this pickle even if this procedure can be challenging:

Moving your changes to new branch

Move your changes to a new branch as soon as you realize you are aiming for the wrong remote repository. You do not want to unintentionally add the laborious work you have just accomplished to the original, upstream branch's modifications.

This phase can be challenging, but fortunately, you can make use of many useful Git Aliases (discussed in detail in next section of the chapter) to assist.

```
→ Git_101 git:(main) git migrate new_branch
 Switched to a new branch 'new_branch'
 Branch 'main' set up to track remote branch 'main' from 'origin'.
 Current branch new_branch is up to date.
```

***Figure 5.18**: Git migrate new branch*

Verify the existence of the new branch using **`git status`** command, as:

```
→ Git_101 git:(new_branch) git status
 On branch new-branch
 nothing to commit, working tree clean
```

***Figure 5.19**: Status check for the new branch*

Make the source repository the upstream remote setting

Type the following command over console to add an upstream remote branch to your local repository as:

```
→ Git_101 git:(main) git remote add upstream https://github.com/justjais/Git_101.git
→ Git_101 git:(main)
```

***Figure 5.20**: Add remote branch to local repository*

Verify that the upstream remote was properly added:

```
→ Git_101 git:(main) git remote -v
 origin https://github.com/justjais/Git_101.git (fetch)
 origin https://github.com/justjais/Git_101.git (push)
 upstream https://github.com/justjais/Git_101.git (fetch)
 upstream https://github.com/justjais/Git_101.git (push)
```

***Figure 5.21**: Verify for remote repo*

Fork the repo

If you go to Github.com, browse the original repository and select Fork from the GitHub drop-down menu on the repo's home page.

When the page reloads, your customized version of the repository with a link to the original repository should appear.

Set your forked repository as the origin remote:

You can modify your remote origin to be your version after you have your fork of the repository:

```
→ Git_101 git:(main) git remote set-url origin https://github.com/temp-user/Git_101.git
```

***Figure 5.22**: Set remote origin*

Check the remote settings for the local repository now and verify if the settings are as expected:

```
→ Git_101 git:(main) git remote -v
 origin https://github.com/temp-user/Git_101.git (fetch)
 origin https://github.com/temo-user/Git_101.git (push)
 upstream https://github.com/justjais/Git_101.git (fetch)
 upstream https://github.com/justjais/Git_101.git (push)
```

***Figure 5.23**: Verify remote repository settings*

Send your branch to the forked copy

You are currently in the same situation as if you had first forked the repository before cloning. You can publish your branch once again in the editor.

Create a new pull request

Now, with all the settings in place as expected you can easily create a pull request using your forked repository and once the **Pull request** (**PR**) gets created the PR will be shown at the upstream repository as discussed in earlier chapter.

Collaborating on pull request

The purpose of pull requests is to initiate a discussion about a suggested change, which is typically either a new feature or a bug fix. Pull requests were once only made after coding was finished to ask someone to incorporate a finished set of changes, but they are now used in a few other ways as well.

If you are confident in a change, you can still start a new branch, make your changes, and then wait to submit a pull request until you are finished. The aim of the pull request in this situation is simply to confirm that the rest of your team supports the modifications you made before they are merged into main branch and deployed to production.

Pull requests can also be used in different ways. Team members/collaborators frequently create pull requests for features they would like to discuss at their workspaces.

Therefore, if you have an idea for a modification but are unsure, think about creating a branch which we have discussed in *Chapter 4, Deleting, Renaming, and Ignoring Files in Git*, starting with the least amount of work possible, perhaps just a brief text file outlining it. A pull request can be made after a commit has been made to the branch to start a conversation regarding the concept.

Generic steps involved before a pull request is ready to get merged to the main repository are:

Collaborators' involvement in the pull request

When making a pull request, ***@mention*** the team members you would like to receive comments from. To achieve this, start a sentence with @ and then the GitHub username in the pull request or in a comment on the pull request.

As you start typing, the username will automatically finish if the individual is either the owner or a collaborator on the repository. Additionally, you can start entering the user's displayed name (which can be set in your public profile).

You could make a comment saying something like, "Hey @brntbeer, mind looking at this PR and letting me know what you think?" You can also send messages like, ''f you needed my opinion on some work", "How you have been doing."

Depending on the people you are working with, the language's formality will vary, although pull request comments are generally written in a formal manner as you might know the collaborator who might be your team member, or you might contribute to an open-source project where you want your contribution to be accepted by moderators.

Pull request review process

If you go to the main page and click the **Pull requests** button at the top, you will find a list of all the open pull requests. This will allow you to see what people are working on within a repository.

A reasonable rule of thumb is to limit the number of open pull requests per developer in a private repository. In general, you want to have as few open pull requests as possible because it is more beneficial to keep the team focused on closing up existing features than beginning new ones.

Additionally, to make them simpler to review, pull requests should be for tiny, iterative modifications. A branch will live longer and be more challenging to properly examine as more changes are added to it. Even though you cannot always avoid them, you should keep an eye out for these "long-lived" branches.

Thing to be note here is, since anybody can submit a pull request, it can occasionally take a while for the core project team to examine, accept, and/or close them, the number of open pull requests for open-source projects will often be substantially higher.

Click on a pull request to visit the pull request detail page when you discover one you want to examine.

Commenting over a pull request

Reviewing any pull request that you might be interested in is a crucial element of working with a development team. Nothing is more demoralizing than working on a feature for a few days, submitting a pull request, and receiving zero response.

Remember to take the time to examine other people's work so they are not tempted to merge it in without at least one or more people having a look at it. By default, anyone can merge their pull request into the upstream main branch as long as they have the write access, but this practice should ideally be avoided.

Alternatively, you can utilize protected branches to enforce certain approval workflows if you would prefer that this not be the case. Make sure to check out any pull requests that you have been @mentioned in as soon as you can and offer some insightful criticism.

Letting the person know you will give it a favourable review soon is sometimes valuable feedback. Even if you are not specifically mentioned, it is a good idea to take a little time out of your day to make sure you examine any pending pull requests and offer your opinions to ensure that everyone is aware of the project's direction.

Commenting on pull requests is one of the more frequent opportunities for interaction in a team, especially one that does not always work in the same office. As a result, it is frequently a good idea to inject some humour into the encounters.

Animated GIFs are another option to give your GitHub comments some extra colour. While most animated GIFs are far larger and more visually striking than emojis, which are often modest, they are a terrific way to truly lighten the atmosphere or express strong support (or opposition) for a change or opinion.

Drag and drop an animated GIF (or any other picture) into the comment box of a pull request, and it will be immediately posted.

Contributing to a pull request

It is straightforward to make changes to someone else's pull request. You may want to alter a pull request directly on occasion.

To improve how a newly added page appears in your preferred browser, you may wish to alter the marketing content, legal disclaimer, or even the CSS.

Testing pull request

Before approving a pull request that includes significant code changes that you cannot just inspect visually if you have the necessary permissions, you should download a copy of the repository.

Check out the branch to which the pull request pertains, run the automated tests to see that they are all passing, then execute the code and perhaps perform some manual testing to ensure it appears to be solid.

Setting up automated testing that will execute for you and submit its results back to the pull request is a simpler and best practice choice. You may ask for artifacts showing testing done as part of the CI/CD pipeline or manual testing from the original developer who raised the PR.

This can be set up in a repository's protected branch settings along with needed reviews.

Also, GitHub has introduced GitHub actions which can actually take care of pull request testing and can run set of tests designed by repo owner/organization in the form of Unit, Integration and Sanity tests as shown in *figure 5.24*. Once this GitHub action tasks turns green either the reviewer can merge the pull request manually or it can also be merged automatically using GitHub actions if the rules are set accordingly. We will not be discussing GitHub actions here as it is beyond the scope of this book.

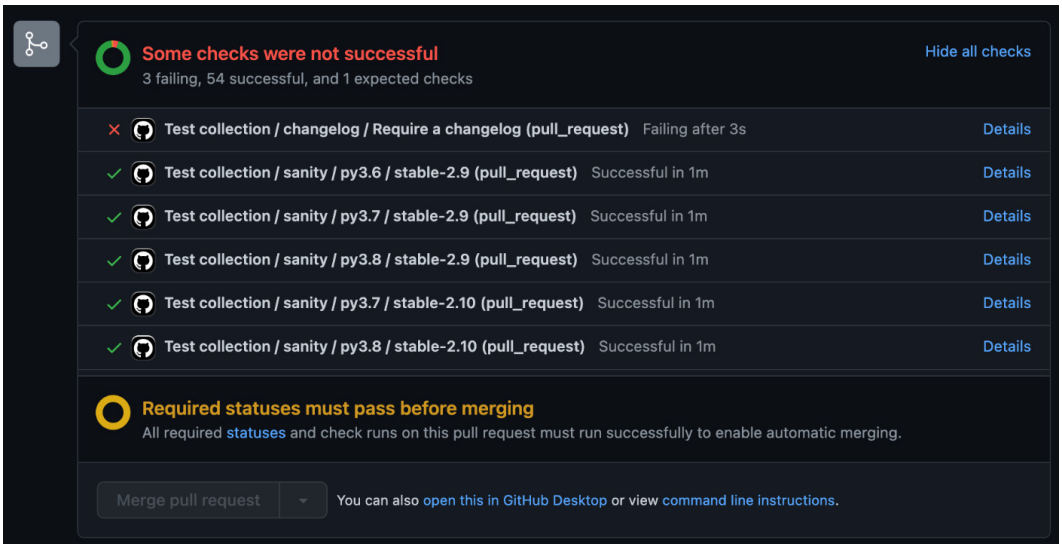

***Figure 5.24**: Testing done via GitHub actions*

Merging pull request

Users can click the big green **Merge pull request** button when they are ready to merge a pull request.

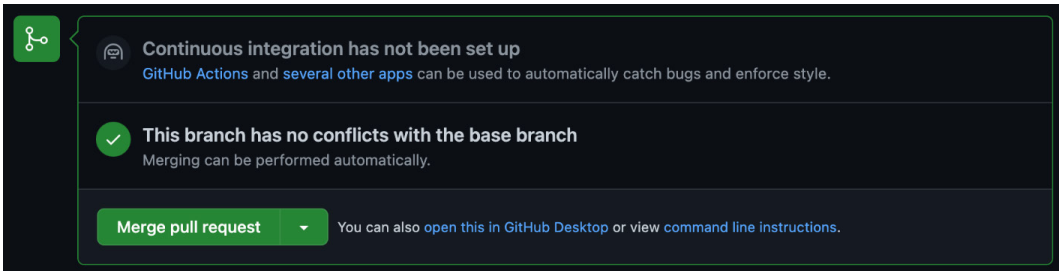

***Figure 5.25**: Merging Pull request*

Then, GitHub will require a commit message (the default will be the title of the pull request and an indication that this commit came in from a pull request merge).

After entering, the pull request will be merged and closed as soon as you click the **Confirm merge** button.

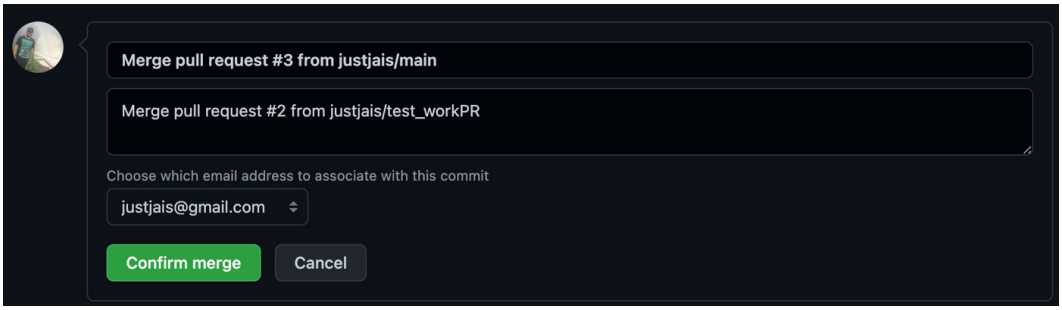

***Figure 5.26**: Confirm merging Pull request*

Most open-source projects have a procedure in place for closing pull requests. Before a pull request is merged, project owners/moderators will request that one or two individuals who are not the pull request's primary author submit +1 to approve the PR changes.

In general, it is preferable to "move fast and break things" than to have multiple people who need to approve every pull request before it can be merged. Keep in mind that you can always undo a merge.

Who should merge the pull request

One question that frequently arises is whether the person who created the pull request should merge it or someone else. It is often a good practice to block the person who authored the pull request should be able merge the pull request.

The policy that "the individual who created a pull request cannot merge it" is one that many organizations adopt.

The individual who initiated the pull request is typically the one with the most knowledge of the changes.

As a result, whenever their work gets merged in, that individual should be ready in case something unforeseen breaks. Asking PR author to perform the merging is one of the simplest methods to ensure that they are available.

As a result, PR author should be allowed to merge in their own pull requests but making sure that they do so only when they have received at least a few +1/ approvals from the team or until all other necessary workflows and statuses have been completed.

Git Aliases

Git Aliases is a feature that can make using Git easier, simpler, and more acquainted to you. We would not use aliases anywhere in this book for clarity, but if you plan to use Git frequently, you should be aware of them.

Git config makes it simple to create an alias for each command if you do not want to input the complete text of each one.

It should be noted that there is no **`git alias`** command. The **`git config`** command and the Git configuration files are used to create aliases. The creation of aliases can be done in a local or global scope, just like other configuration parameters.

Let us understand this with some examples for better understanding:

```
→ Git_101 git:(main) git config --global alias.co checkout
→ Git_101 git:(main) git config --global alias.br branch
→ Git_101 git:(main) git config --global alias.ci commit
→ Git_101 git:(main) git config --global alias.st status
```

***Figure 5.27**: Git Aliases examples*

This implies that you only need to type **`git ci`** rather than **`git commit`**. You will likely use other commands regularly as you continue to use Git; do not be afraid to create new aliases.

Using this method, you may also create commands that you believe ought to be there. You can create your unstage alias to Git, for instance, to fix the usability issue you observed with unstaging a file:

```
→ Git_101 git:(main) git config --global alias.unstage 'reset HEAD --'
```

***Figure 5.28**: Git Aliases unstage alias*

Consequently, these two commands are equivalent:

```
→ Git_101 git:(main) git unstage README.md
→ Git_101 git:(main) git reset HEAD -- README.md
```

***Figure 5.29**: Git Aliases unstage alias git equivalent*

As you can see, Git substitutes the new command for the value of the alias. But, instead of executing a Git subcommand, you could prefer to execute an external command. In that situation,"**!**" character is used to begin the command. This is helpful if you want to create your own tools that integrate with a Git repository.

```
→ Git_101 git:(main) git config --global alias.visual "!gitk"
```

***Figure 5.30**: Git Aliases external command example*

In conclusion, Git aliases are an effective workflow tool that enables the creation of shortcuts for commonly used Git commands. Git aliases will help you develop code more quickly and effectively.

Aliases can be used to combine several Git commands into a single more developer-friendly command. The **`git config`** command, which essentially edits local or global Git config files, is used to build git aliases.

Conclusion

This is the end of this chapter and the topics discussed are the building blocks and soul of the Git version control system. The more fluent you get will all the discussed terms and their respective usages and application, the more efficient you will become in using Git and GitHub as well.

In the next chapter, we will continue building on the knowledge we gained in this chapter and discuss a few of the Git concepts in more depth.

Multiple choice questions

1. **Which git command gets your repository off GitHub and onto your computer?**
    a. git push
    b. git fork
    c. git commit
    d. git clone

2. **How do you duplicate a repository, so you may solve it on your own GitHub account?**
    a. Fork a GitHub repo using GitHub interface
    b. git fork
    c. git clone
    d. git pull-request

3. **What is the use of forked repo?**
    a. Utilize someone else's project as a foundation for your own
    b. Used to iterate on ideas or modifications before submitting them to the upstream repository
    c. Both a and b
    d. None of the above

4. **How can you determine whether your local Git repository has changed since your last commit?**
    a. git status
    b. git diff
    c. git commit
    d. git check

5. **Git command to give the frequently used Git command a generic alias instead of using the long usual command line syntax?**
    a. git alias
    b. git config --global alias
    c. git push
    d. git add

Answers

1. d **2. a** **3. b** **4. a** **5. b**

Key terms

- Start a working area
    - init
    - Clone

- Work on the current change:
    - add
    - mv
    - restore
    - rm
- sparse checkout examine the history and state of the repository:
    - bisect
    - diff
    - grep
    - log
    - show
    - status
- Grow, mark and tweak your repo history:
    - branch
    - commit
    - merge
    - rebase
    - reset
    - switch
    - tag
- Collaborate over repository:
    - fetch
    - pull
    - push

Points to remember

Recommended pull request strategies. When handling pull requests, it is a good idea to keep the following in mind:

- For everything, create pull requests.

  Make sure to work on a branch if you wish to fix a bug or add a new feature and then submit a pull request to gather feedback before merging your changes into master.

- Make the PR headings informative.

  To understand what is going on, other team members will examine the pull requests. They should be able to tell what you are working on from the title.

- Spend some time commenting.

  Even if no one mentions you, take this action. It will increase the overall quality of the job and provide you with a clear understanding of the project's status.

- Mention relevant users in the PR.

  By @mentioning the relevant users, you may ensure that marketing, legal, and the operations team notice your pull request and increase the likelihood that you will receive feedback.

- Run all the tests.

  Ensure that at least one developer checks out the proper branch, downloads the most recent changes from a pull request and executes your automated tests.

  This should be a part of the CI CD pipeline and should be used as a gateway by the reviewer(s).

- Simply scanning the code visually for non-trivial changes is insufficient.

  Clearly define your approval process for pull requests.

- Before a pull request is merged, most open-source projects demand that one or two authors who are not the pull request's lead author examine it and give their approval.

Further reading

For more history and reference around the discussed topics in this chapter you can check out the Git and GitHub's official documentation for getting started:

- Pull request over GitHub: **https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 6
Contributing Towards Open-Source Project Repo

Introduction

Now that you have Git and GitHub up and running on your system, let us understand why Git and GitHub are required and their importance and relevance when it comes to open-source ways of development or development across teams and location in general.

This chapter discusses all the process-related and critical aspects that should be kept in mind and followed before simply trying to contribute to an open-source project that is being followed and used by a larger community from all over the world, as well as how this varies significantly from maintaining and contributing to a single-user repository.

Structure

In this chapter, we will cover the following topics:

- Understanding a pull request
- Open a pull request over GitHub
- Writing a great bug report
- Pushing code and opening a pull request over GitHub

Objectives

This chapter builds on what we learned in the previous chapter and allows you to make a final decision before pushing and committing changes to source control.

This process of committing changes to the GitHub repo may entail renaming, deleting, or ignoring files in the project. All these ideas will be covered in detail in this chapter.

This chapter assumes you have Git up and running under your Linux, Windows, or Mac machines.

Understanding a pull request

A pull request is a request for a repository's maintainer to pull in some code.

Pull requests are a way for a developer to notify the rest of the team that they have finished working on a feature/bug fix. The developer submits a pull request to their GitHub account after their feature branch is complete. This lets the developer inform teammates concerned that the code has to be reviewed and merged into the main branch.

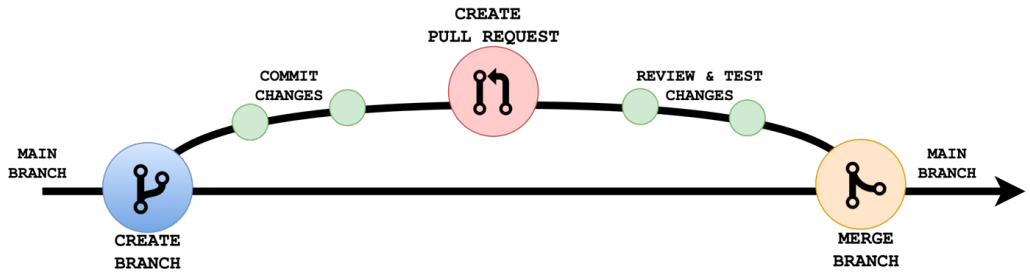

***Figure 6.1**: To demonstrate the Pull request flow*

This formal technique for sharing commits has a considerably more efficient workflow than existing collaboration models. With a simple script, SVN and Git, both can send notice emails; however, when it comes to discussing changes, developers often have to rely on email threads. This can become haphazard, especially when it comes to follow-up commits.

Nature of a pull request

If you are proposing that another developer (example, the project maintainer) pull a branch from your repository into their repository when you submit a pull request. To submit a pull request, you must supply four pieces of information:

- Source repository
- Source branch
- Destination repository
- Destination branch

Git pull

The pull request is used to fetch changes (commits) from a remote repository and store them locally. It synchronizes the local and remote-tracking branches.

```
→ Git_101 git:(main) git pull <option> [<repository URL><refspec>..]
```

***Figure 6.2**: Pull request syntax*

Remote tracking branches have been configured to push and retrieve data from a remote repository. It is just a compilation of the fetch and merges commands. It begins by retrieving changes from a remote repository and combining them with the local repository where syntax parameters are as follows:

- <option>

  Options are commands that can be used as an additional option in a specific command.

  `-q` (quiet), **`-v`** (verbose), **`-e`** (edit), and more options are available.

- <repository URL>

  A repository URL is the URL of your remote repository or an alias for the same, such as GitHub or another Git site, where you have placed your original repositories.

This is how you can implement Git pull:

1. Go to your GitHub account and pick the repository you wish to clone to get this URL.
2. Then, from the repository menu, select the clone or download option.

3. A new pop-up window will appear; from the list of options, select **Clone** with **HTTPS** as shown in the following screenshot:

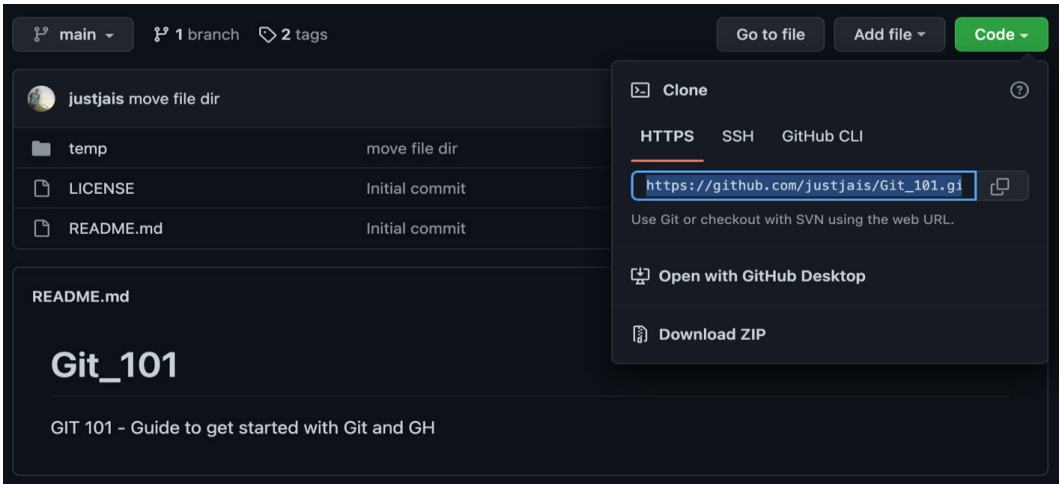

***Figure 6.3**: Git_101 clone repository*

4. Copy the repository highlighted from the *Clone->HTTPS*
5. It is critical to understand how Git works.

Let us look at an example to see how it works and how to put it to use. Assume, I have added a new file to my remote repository of project **`Git_101`** called **`pull_101.md`** under project temp directory.

1. To begin, select the **Create a File** option from the repository sub-functions.
2. After that, select the file name and make any necessary changes. Consider the following screenshot:

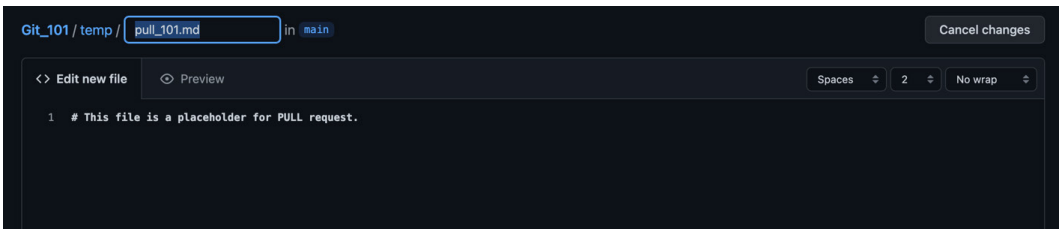

***Figure 6.4:** Add new file to Git_101 repository - I*

3. Select a commit message and file description at the bottom of the page.
4. Choose whether to establish a new branch or commit it to the main branch directly. Consider the following screenshot:

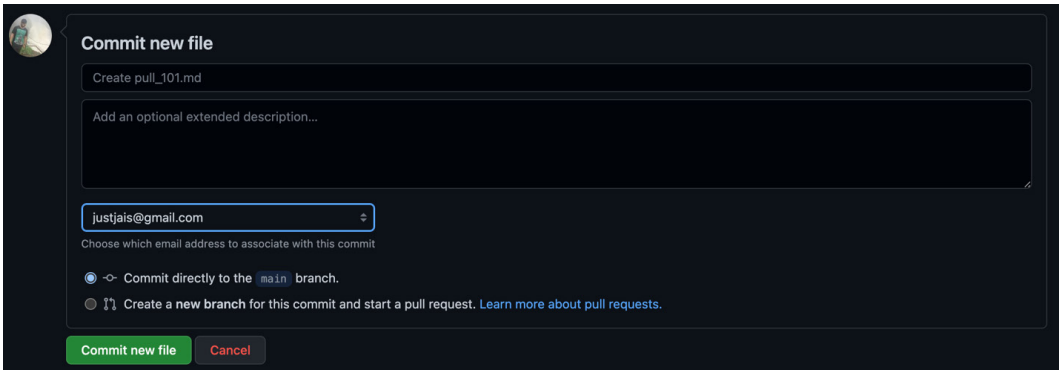

***Figure 6.5**: Add new file to Git_101 repository - II*

5. The changes have now been successfully made.

Perform a git pull on your cloned repository to pull these changes into your local repository. For the pull command, there are a variety of choices.

Using the **`Git pull`** command, we can pull a remote repository. It is the default setting:

```
→ Git_101 git:(main) git pull
```

***Figure 6.6**: Git pull default*

The **`git pull`** command is used to obtain the repository's freshly updated items in the output below. It is the **`git pull`** command's default version. It will update the newly produced file **`pull_101.md`** in the local repository, as well as any linked objects.

```
→ Git_101 git:(main) git pull
 remote: Enumerating objects: 6, done.
 remote: Counting objects: 100% (6/6), done.
 remote: Compressing objects: 100% (3/3), done.
 remote: Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
 Unpacking objects: 100% (4/4), 796 bytes | 113.00 KiB/s, done.
 From https://github.com/justjais/Git_101
   b1b4cca..efd4ea4  main       -> origin/main
 Merge made by the 'recursive' strategy.
   temp/pull_101.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 temp/pull_101.md
```

***Figure 6.7**: Git pull in action*

The **pull_101.md** file is added to the local repository, as shown in the output below. Git pull is the same as **git fetch origin head** and **git merge origin head**. The present branch's head is referred to as the ref:

```
→ Git_101 git:(main) tree
.
├── LICENSE
├── README.md
└── temp
    ├── __init__.py
    ├── learn_git.txt
    ├── new_main_file.txt
    └── pull_101.md

1 directory, 6 files
```

***Figure 6.8**: Git pull output*

There is another way to get to the repository. Using the **git pull** command, we can pull the repository.

```
→ Git_101 git:(main) git pull <option> <remote branch name>
→ Git_101 git:(main) git pull origin main
```

***Figure 6.9**: Git pull origin syntax*

The term origin refers to the repository's physical location from which the remote repository can be accessed. The primary branch of the project is called Main. (Previously called Master).

```
→ Git_101 git:(main) git fetch -all
 From https://github.com/justjais/Git_101
 * branch            main       -> FETCH_HEAD
 Already up to date.
```

***Figure 6.10**: Git pull origin main in action*

It will replace the data in the local repository with data from a distant repository. You can look up your repository's remote location.

Use the following command to check the repository's remote location:

```
→ Git_101 git:(main) git remote -v
```

***Figure 6.11**: Git remote syntax*

Both fetch and push locations are displayed in the output.

```
→ Git_101 git:(main) git remote -v
 origin https://github.com/justjais/Git_101.git (fetch)
 origin https://github.com/justjais/Git_101.git (push)
```

***Figure 6.12**: Git remote in action*

Git pull from remote branch

Git allows you to fetch a specific branch. Using the **`git pull`** command to fetch a remote branch is identical to the technique described in the previous section of Git remote. The only difference is that we must copy the URL of the branch we wish to pull followed by the branch name. For example, git pull origin feature/bugFix. To accomplish so, we will pick a particular branch:

```
→ Git_101 git:(main) git pull <remote branch URL>
```

***Figure 6.13**: Git pull remote branch syntax*

Git force pull

Git force pull allows you to pull your repository at any time and at any cost. Consider the following scenario:

- You made a local change to a file and another team member made a remote change to it. So, when are you going to fetch the repository? This could cause a dispute.
- Force pull is a technique for overwriting files. If we wish to undo all the modifications in the local repository, we can pull it influentially and replace it. To force pull a repository, follow the steps below:
    1. Download the latest updates from the remote using the **`git fetch`** command without *merging or rebasing*.

```
→ Git_101 git:(main) git fetch -all
```

***Figure 6.14**: Git fetch*

2. If everything is already updated in the branch, you should receive the output below, else the local repository branch will be updated with all remote branches:

```
→ Git_101 git:(main) git fetch -all
 Fetching origin
```

***Figure 6.15**: Git fetch in action*

3. To reset the master branch with the updates you fetched from the remote, use the **`git reset`** command. The hard option is used to replace all the files in the local repository with those from the remote repository. This comes handy in scenarios like, you have committed changes in your local branch that you want to revert.

```
→ Git_101 git:(main) git reset --hard <branch name>
→ Git_101 git:(main) git reset --hard main
```

***Figure 6.16**: Git reset –hard*

`git reset` actual command run:

```
→ Git_101 git:(main) git reset --hard main
HEAD is now at bb76e56 Merge branch 'main' of https://github.com/justjais/Git_101
```

***Figure 6.17**: Git reset –hard in action*

A complete GitHub workflow

The GitHub workflow manages cooperation using a shared GitHub repository, and developers create features in isolated branches. Rather than merging them directly into *main* branch, developers should file a pull request to start a conversation about the feature before it is merged into the *main* codebase.

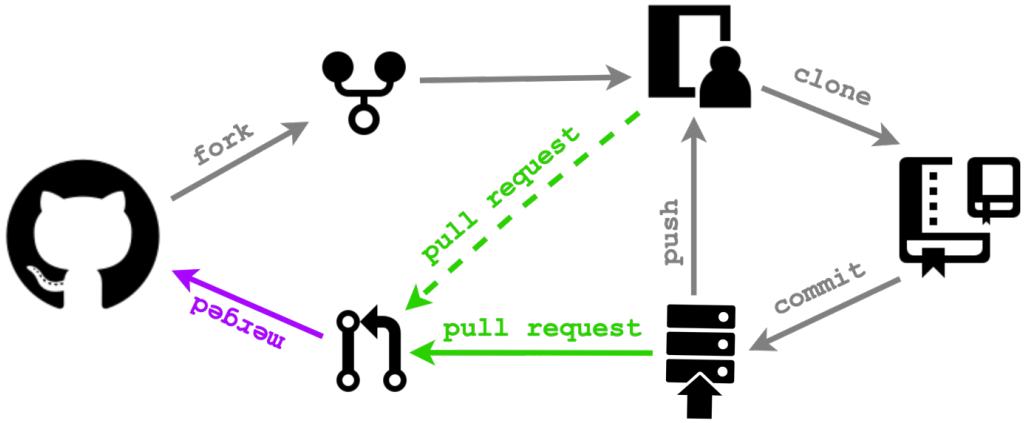

***Figure 6.18**: GitHub Workflow*

The destination repository and the source repository of a pull request will always be the same because there is only one public repository in the GitHub Workflow. The primary branch is usually designated as the destination branch, and the developer will typically designate their feature branch as the source branch.

GitHub Workflow with pull requests

The GitHub Workflow specifies a rigid branching model built around the project release. Developers can easily discuss a release or maintenance branch while they are working on it by adding pull requests to the GitHub Workflow.

When a feature, release, or hotfix branch has to be reviewed, a developer simply submits a pull request, and the rest of the team is notified via GitHub. This is exactly how pull requests work in the GitHub Workflow.

Release and hotfix branches are often merged into both checked out branch and *main*. All of these mergers can be explicitly managed using pull requests.

Fork Workflow with pull requests

Instead of pushing a finished feature to a shared repository, a developer uses the forking workflow to publish it to their public repository. Once it is prepared for review, they submit a pull request to the project maintainer.

Since the project maintainer has no means of knowing when another developer has submitted changes to their GitHub repository, the notification feature of pull requests is especially helpful in this scenario.

The source repository for a pull request will be different from its destination repository because each developer has their forked public repository. The source

branch is the one that has the suggested changes, and the source repository is the developer's open repository.

The official project is the destination repository, and the main branch is the destination branch if the developer is attempting to integrate the feature into the primary codebase.

GitHub for Code distribution

The Feature branch workflow manages cooperation using a shared GitHub repository, and developers create features in isolated branches. Rather than merging them into the main, developers should file a pull request to start a conversation about the feature before it is merged into the main codebase.

The first step is to make a new branch. When using Git, it is usual practice to create a new branch before writing new code. We directly committed code to the *main* branch in the examples while learning about Pull requests. We offered that as a shortcut to keep things simple.

Now, we will work on a new branch and do things the right way in this section. This is for a very crucial reason. A pull request is not made up of a random collection of modifications or commits.

A branch is always associated with a pull request. A pull request, in other terms, is a request to merge two branches into one.

While a pull request can target any branch (except its own), the most common situation is to target the repository's primary branch, which is usually named main.

```
→ Git_101 git:(main) git checkout -b origin/understand_PR
```

***Figure 6.19**: GitHub Workflow*

This link between pull requests and branches is why, when starting new work, you should create a new branch. For this example, we will call the branch **`understand_PR`**, but you may call it whatever you wish by replacing **`understand_PR`** with your branch name in the command shown in figure 6.19

Open a pull request over GitHub

You need a repository to push code to GitHub. Select a repository and open it. Clone your forked copy to your local workstation after forking the BPB repository.

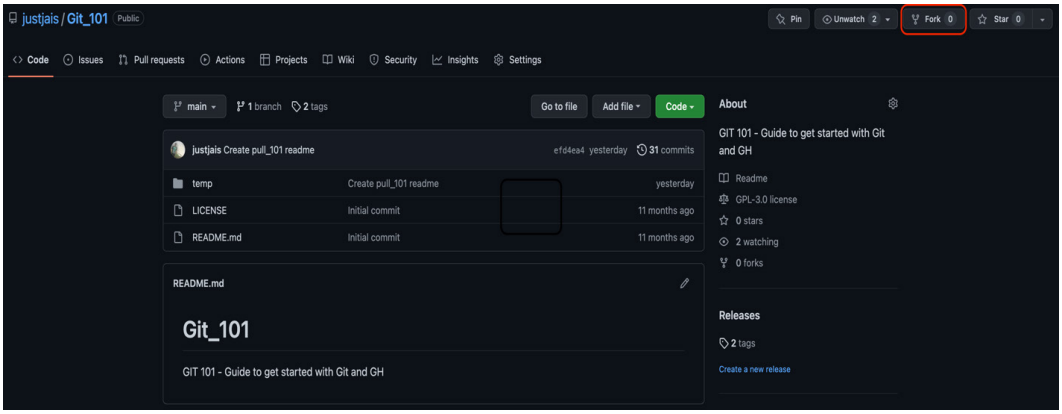

***Figure 6.20**: Git fork the repo*

Clone the forked repo to your local system and once done check for the set repo via the command **`git remote -v.`** It should show fetch and push repo as your cloned fork repo link. Now, you may want to add the parent repo as an upstream repo to your local repo to keep your forked local repository with the latest changes made to the upstream parent repo by other developers on the team. To add upstream branch to your forked local repository, use the following command:

```
→ Git_101 git:(main) git remote add upstream https://github.com/justjais/Git_101.git
```

***Figure 6.21**: Git remote add upstream repository*

Now if you run the **`git remote -v`** command, it should give you the following output:

```
→ Git_101 git:(main) git remote -v
 origin https://github.com/<your github_id>/ibm.qradar.git (fetch)
 origin https://github.com/<your github_id>/ibm.qradar.git (push)
 upstream https://github.com/justjais/Git_101.git (fetch)
 upstream https://github.com/justjais/Git_101.git (push)
```

***Figure 6.22**: Git remote in action*

The first step is to make a new branch. When using Git, it is a usual practice to create a new branch before writing new code. While a pull request can target any branch (except its own), the most common situation is to target the repository's primary branch, which is usually named **`main`** *(*previously referred as master).

This link between pull requests and branches is why, when starting new work, you should create a new branch. For this example, the branch is referred to as **`test_`**

workPR; however, you are free to give it any other name by substituting your branch name for **test_workPR** in the command below:

```
→ Git_101 git:(main) git checkout -b test_workPR origin/main
```

***Figure 6.23**: Git checkout in action*

We can now make a commit in the branch we created. The particulars of the commit's contents are not relevant for this example. Any file that you choose can be edited, for example, by adding text at the end. You can also manually edit the **pull_101.md** file located in the Git_101 temp directory if you are using the practice repository we set up.

```
→ Git_101 git:(test_workPR) echo "Pull 101: Starting point to Pull request" >> temp/pull_101.md
→ Git_101 git:(test_workPR) ✗ cat temp/pull_101.md
 # This file is a placeholder for PULL request.
 Pull 101: Starting point to Pull request
```

***Figure 6.24**: Update pull_101.md file content*

Commit the modifications you have made to the file. To commit all your modifications, for instance, use the **git add .** command. The commit message should be something readable for the commit made. The most crucial element is to have a commit in a branch other than the main branch that you can work with.

```
→ Git_101 git:(test_workPR) git add .
→ Git_101 git:(test_workPR) git commit -m "Adding text to pull_101 readme"
 [test_workPR 5fc0392] Adding text to pull_101 readme
 1 file changed, 1 insertion(+)
```

***Figure 6.25**: Git commit in action*

Now that the changes are committed to the **test_workPR** branch, we will push the changes to the same branch in the remote repository to open a pull request against the parent repository as:

```
➜ Git_101 git:(test_workPR) git push origin test_workPR
 Enumerating objects: 7, done.
 Counting objects: 100% (7/7), done.
 Delta compression using up to 12 threads
 Compressing objects: 100% (4/4), done.
 Writing objects: 100% (4/4), 401 bytes | 401.00 KiB/s, done.
 Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
 remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
 remote:
 remote: Create a pull request for 'test_workPR' on GitHub by visiting:
 remote:      https://github.com/justjais/Git_101/pull/new/test_workPR
 remote:
 To https://github.com/justjais/Git_101.git
  * [new branch]      test_workPR -> test_workPR
```

***Figure 6.26**: Git push in action*

Git is instructed to push local commits to a remote repository using the **`git push`** command.

Opening a pull request

Your GitHub.com repository needs to have at least one branch aside from the default branch before you can start a pull request.

The branch in our example is called **`test_workPR`** branch, but you might have given yours a different name. To start a pull request, go to the **`Git_101`** repository's page on *GitHub.com* as shown in the following screenshot:

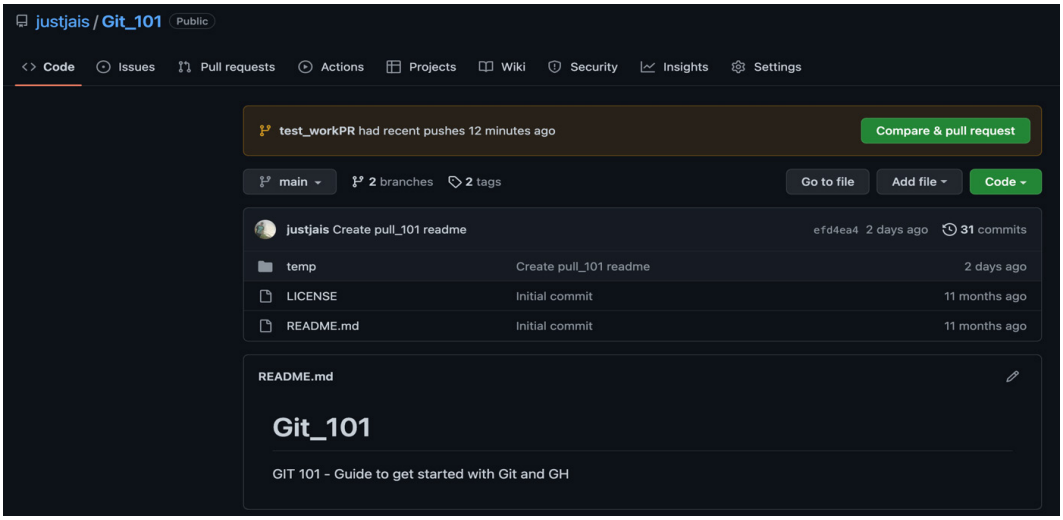

***Figure 6.27**: GitHub repo pull request notification*

To access the **Open a pull request** page, click the **Compare & pull request** button as seen in the screenshot below. The default branch for the repository is the target branch, into which you want to merge your modifications.

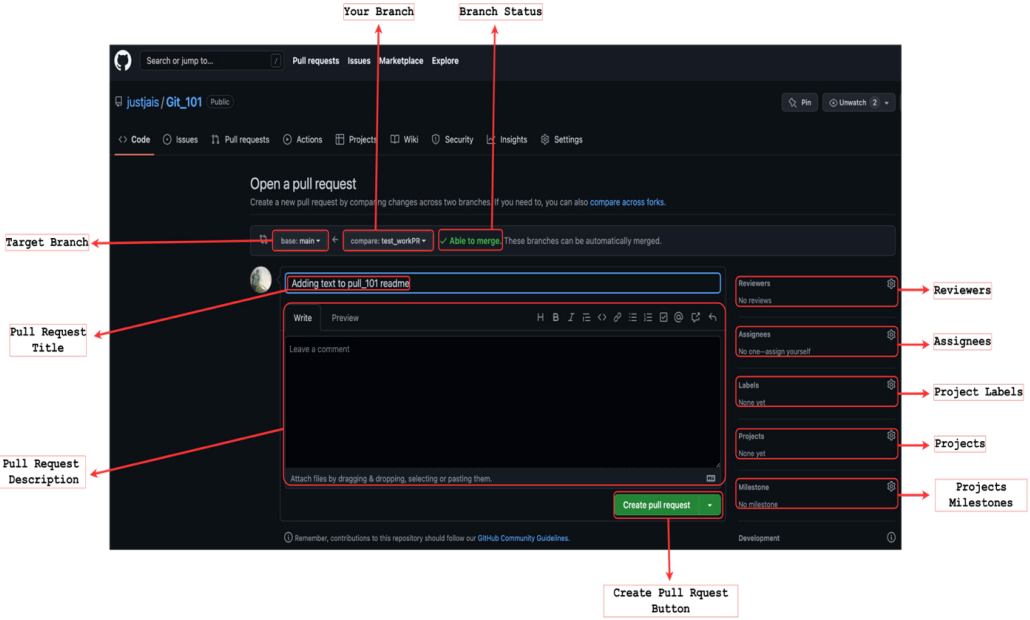

***Figure 6.28**: GitHub Pull Request*

The target branch is listed next to your branch, which is followed by a status indicating whether your branch can be merged into the target branch. The pull request title is the same as the most recent commit message, which in this case is, **Adding text to pull_101 readme**, and your pull request description is blank.

Describing the pull request

You can enter a summary and description from the Open a pull request page. For commit messages, GitHub has established conventions.

The majority of these practices, such as mentioning people using the **`@USERNAME`** format, are also supported in pull requests. So, if I use **`@justjais`** in the PR description, I will get a notification as **`@justjais`** is my GitHub Username/ID.

The **`#ISSUEID`** format can be used to refer to problems and other pull requests.

A good pull request requires a lot of work. Most projects already have a set of norms that developers adhere to when making a pull request; if not, we will examine the considerations and points to keep in mind in a later portion of this chapter.

Adding reviewers

A collection of pull request choices is located to the right of the summary and description boxes.

You can designate one or more individuals to review your pull request using the first option, **Reviewers**. Adding reviewers:

1. To get a list of people you can mention, click the gear. You can begin typing to narrow down the list of users in repositories with a lot of users.

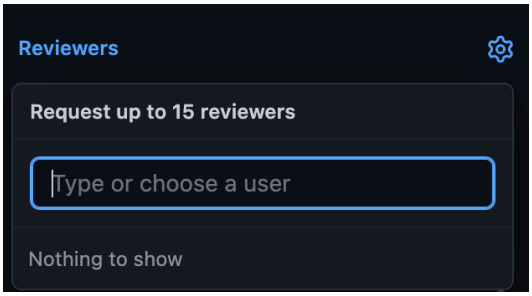

***Figure 6.29**: GitHub Pull Request reviewers' tab*

2. To add a user to the list of reviewers, click on each one.
3. When you finish creating the pull request, the reviewers you have added are informed right away.
4. When clicked on reviewers' tab, it displays the list of reviewers' who are a part of the respective GitHub project repository.

Since I am the only contributor to the Git 101 project currently, it is a personal project. However, like all open-source projects, when you click on the reviewers' page, members of the project are automatically populated as suggestions.

Adding assignees

An option to specify assignees comes after the Reviewer as a choice. The individual who needs to act on the pull request is known as an assignee.

A pull request frequently signifies work in progress rather than the finished product of some task. You will assign the pull request to the appropriate person if extra work needs to be done on it.

To list the assignees:

1. To view a list of assignees, click the **Assignees** tab. The assignee dialogue box functions exactly like the reviewers' dialogue box, which was covered in the part before this one. You have the option of choosing one or more assignees.

2. To add a user to the list of reviewers, click on each one.

It is usually better to designate just one person to oversee the following action.

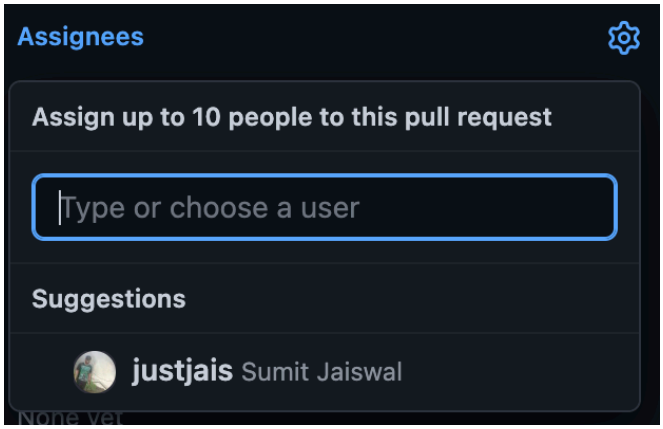

***Figure 6.30**: GitHub Pull Request assignees' tab*

In the likelihood that, many assignees believe the other assignees oversee, the job is decreased when only one person is assigned.

Adding labels

Labels function the same manner for pull requests as the title implies as a method of determining what to work on next. To make it easier for you to choose what to work on, or what to review next, they offer handy categorization and context.

Although you can use the same set of labels on issues and pull requests, some labels make more sense for issues than for requests, and vice versa. For instance, many repositories feature a label called "*ready for review*" just for pull requests.

Additionally, labels can be used to initiate any response specified in a label action.

Adding projects and milestones

You can define the project board and milestone that this Pull request belongs to using the last Projects and Milestones options.

Creating the pull request

Click the **Create pull request** button to save all your work and to start the pull request once you have written the pull request and chosen all of its options.

A pull request I made on my repository is shown in the following screenshot:

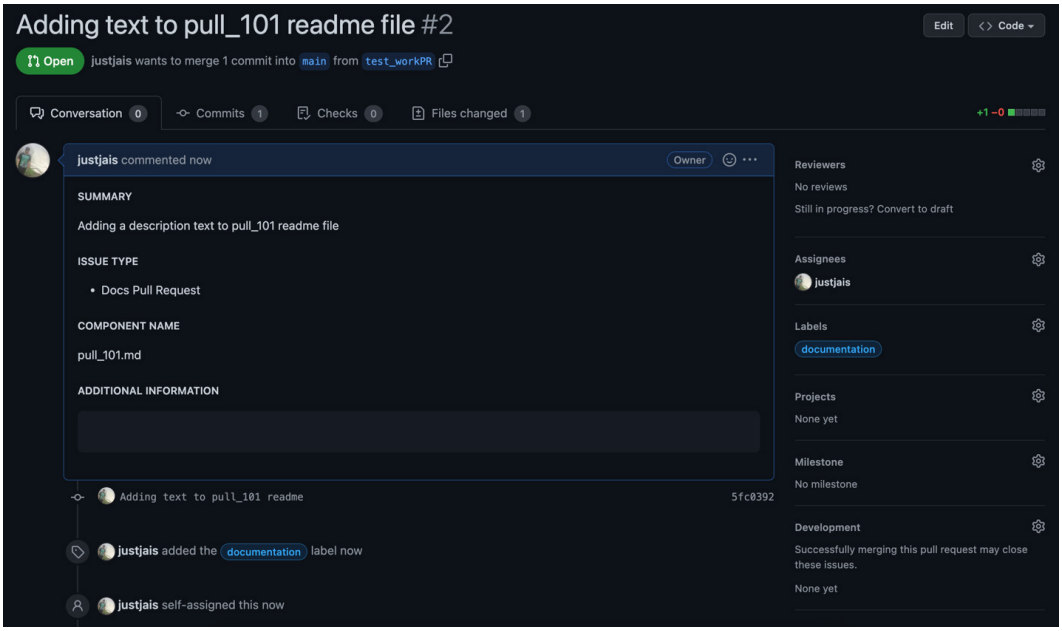

***Figure 6.31**: GitHub Pull Request*

Writing a good pull request

Pull requests are where an open-source project's interaction with users takes place.

When you contribute to a project, your pull request is your opportunity to convince the project's developers that your code deserves to be merged into the main branch. So, be sure to put your best foot forward.

- Clear purpose

  You should aim to be both succinct and informative. The synopsis, for instance, needs to be explicit about the pull request's goal. On the page listing pull requests, only the summary is displayed. Readers should be able to get a sense of the PR's content.

  Here are a few illustrations of effective pull request summaries:

  - Adds the website's **About** page.
  - Reduce boilerplate JavaScript library setup code.
  - Extract and separate the error handling from the internals.

  The goal of the pull request should be more thoroughly explained in the description. Make it obvious what the pull request aims to do but do not write a book about it.

Maintaining the focus

A pull request should not include a cluster of irrelevant changes, just like a commit should not. Multiple commits may be included in a pull request, but they must all be pertinent to the job at hand. You may also use the rebase command to merge multiple commits in your local repository to make it more understandable. For example, a Commit for correcting a spelling mistake is not something that you would want to have as a separate commit while submitting your PR.

You can often tell that a pull request is doing too much when crafting a clear description of what the pull request does is tough. Keep the pull request to an acceptable size, even if it just contains one significant change. It is challenging to review a big pull request.

Break the task down into smaller steps and submit multiple smaller pull requests for each one if the pull request pertains to a particularly broad task.

Be aware of your audience

Knowing your audience before you start writing can assist you to concentrate your words on the information that will be most valuable to them. Many audiences could benefit from a pull request. While it is vital to consider your entire audiences, your main attention should be on those who will examine your pull request and decide whether to merge it. They tend to be very busy, so you want to make their life easy.

Although the project maintainers are your main target audience, you must always keep in mind that the pull request will be read by a large number of people. That audience may be the entire world for an open-source project. So, be sure to speak in a respectful, cordial, and inclusive manner.

Defining a call to action

When asking for feedback on a pull request, you must be very specific.

Make it explicit from the beginning, for instance, if the pull request is a work in progress to prevent people from wasting their time examining it. Tags like **`<WIP>`** for Work in Progress or **`<DNM>`** Do not merge are often used.

Reviewing a pull request

When reviewing modified files, you can add comments to particular lines of code to highlight issues, provide improvements, or simply acknowledge someone's amazing coding expertise. If the destination branch has changed significantly after the source branch was rebased, it is recommended to request a rebase before reviewing.

Positive and motivating remarks create a friendly and cooperative atmosphere. Maintainers frequently overlook how intimidating it might be to submit your first piece of code to a project.

We advise initiating a review rather than adding individual comments as you go because a review results in a more comprehensive and useful code review. When evaluating code, it frequently happens that something you read later in the review prompts you to realize that a previous comment needs to be revised or even removed.

Before sending anything to the author, a review enables you to make these modifications. You can review your review before publishing it using this option.

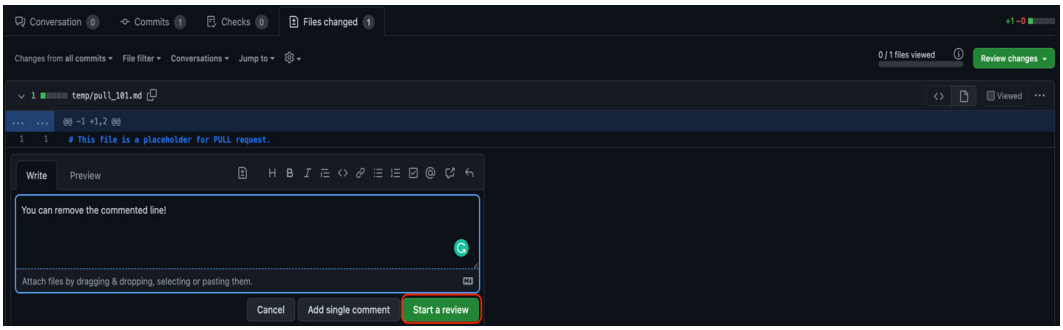

***Figure 6.32**: GitHub Pull Request review comment*

Suggesting changes

Sometimes when reviewing code, you come across a portion where you think it would be quicker to just fix the code rather than try to convey what needs to be addressed verbally.

Or perhaps you find several little mistakes, like typos, where correcting them instead of commenting on each one of them will make less errors and consume less time.

For these situations, GitHub supports suggesting a change via a pull request comment that the author can apply with a click.

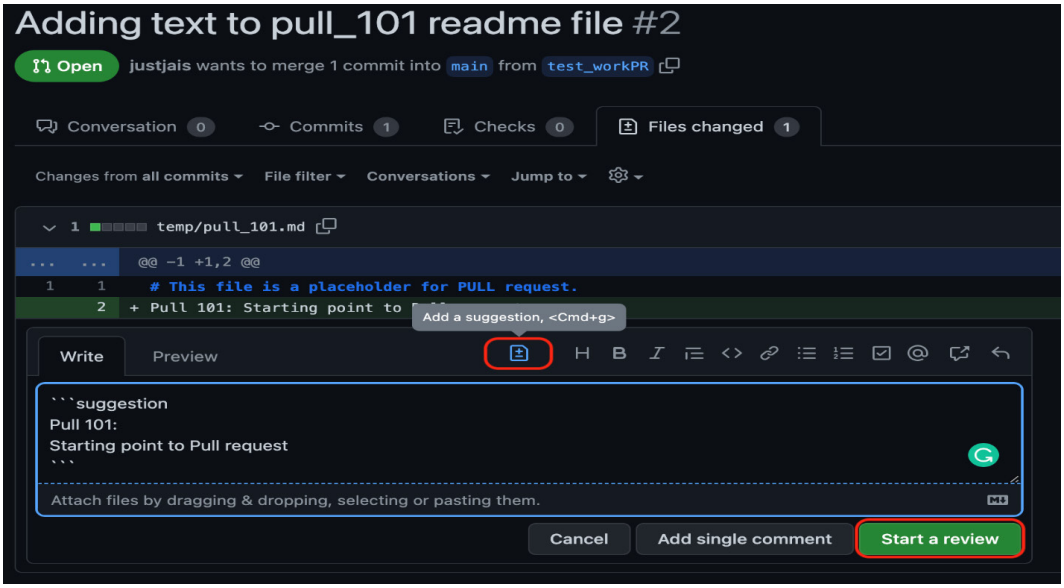

Figure 6.33: *GitHub Pull Request review suggested change tab*

After you have entered the suggested changes, if you click on preview tab, your suggested suggestion will be displayed as shown in the following screenshot:

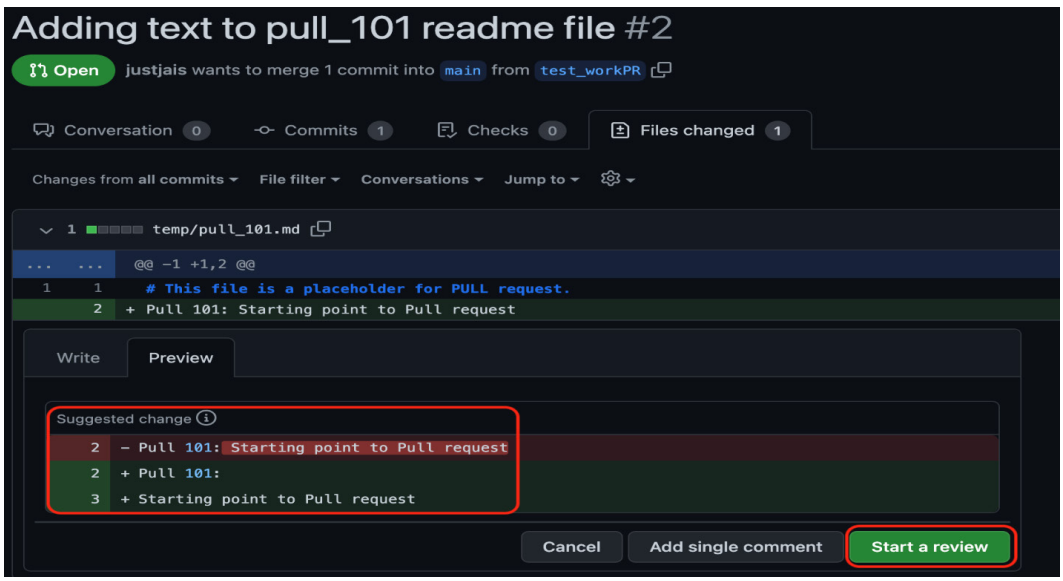

Figure 6.34: *GitHub Pull Request review suggested change*

Finish review

Once you have offered all your suggestions, you may wrap up the review so that the author can start addressing all of your insightful feedback. Click the **Finish your review** button at the bottom of each outstanding remark to complete the review.

You will come across a form where you can enter a general overview of the pull request. This form gives you the chance to mention any review remarks that are not related to any particular lines of code. Additionally, it is an excellent chance to express general approval, voice general concerns, recommend follow-up actions, and so forth.

Check one of the options after you have typed your remark to show where you stand on the pull request. The comment form with review options is shown in the following screenshot:

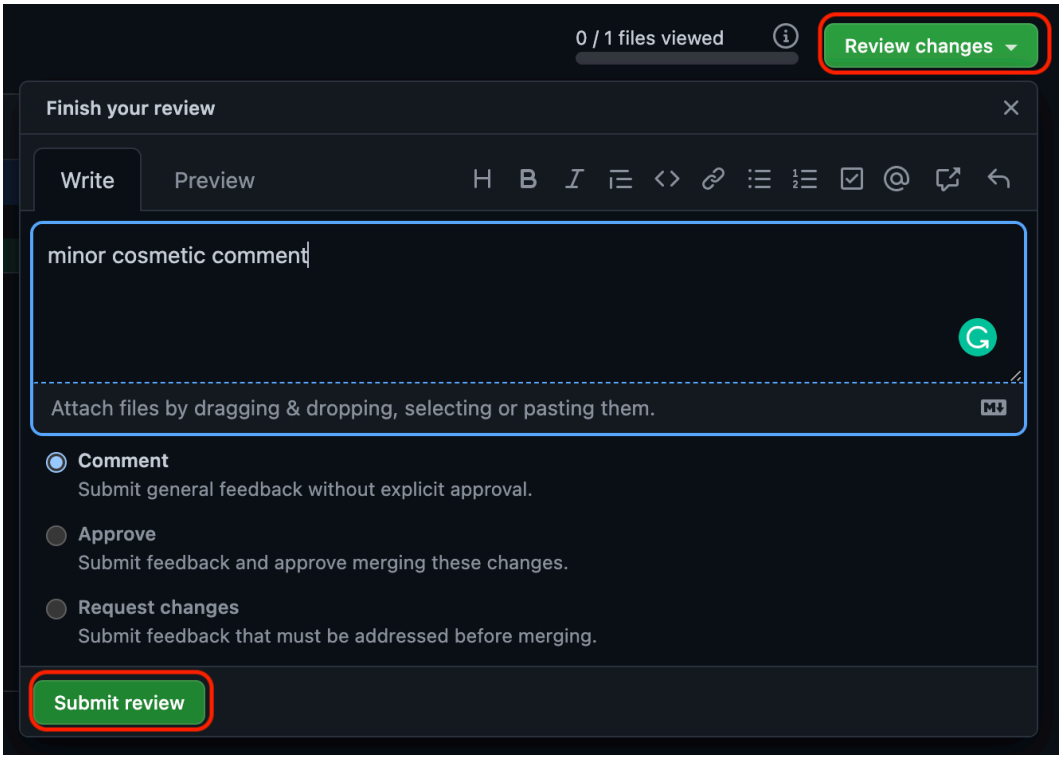

Figure 6.35: *GitHub Pull Request review complete*

Once the review is pushed, it will be displayed as the following screenshot:

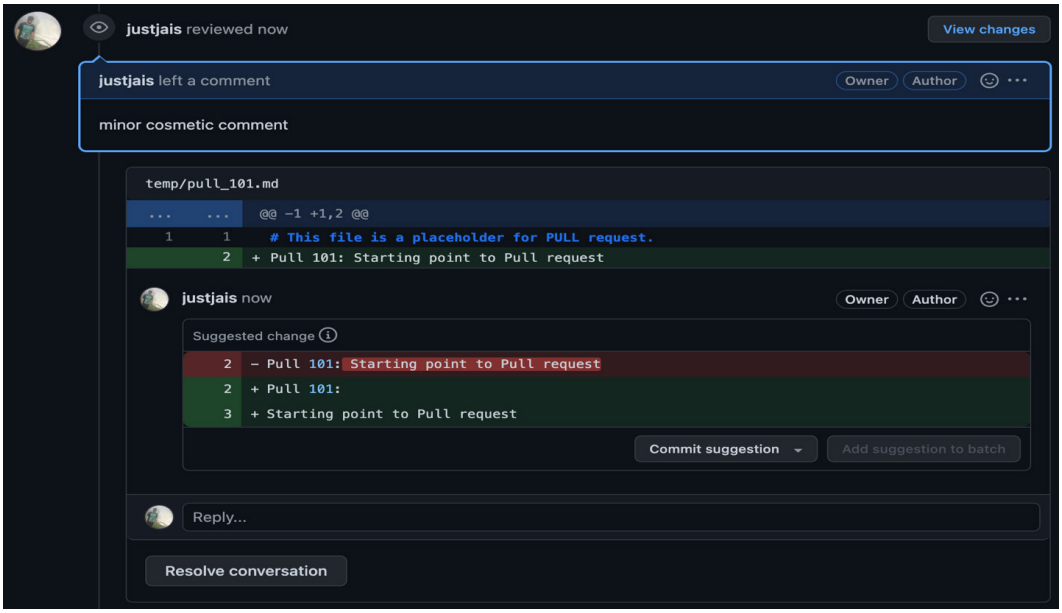

Figure 6.36: *GitHub Pull Request review comment*

Merging Pull Request

The developer who submitted the **Pull request** (**PR**) must fix the reviewer's comment, upload the changes, and wait for the reviewer to approve before merging the PR.

The developer will be able to merge the PR if he has permission to do so. In larger open-source projects, manual merge permissions are not permitted, and the PR is only merged automatically once the project's continuous integration setup has completed its processing. In some projects, the reviewer also has permission to merge the project.

```
→ Git_101 git:(test_workPR) vi temp/pull_101.md
→ Git_101 git:(test_workPR) ✗ git add .
→ Git_101 git:(test_workPR) ✗ git commit -m "fix review comment"
 [test_workPR cdb75e0] fix review comment
 1 file changed, 2 insertions(+), 2 deletions(-)
→  Git_101 git:(test_workPR) git push origin test_workPR
 Enumerating objects: 7, done.
 Counting objects: 100% (7/7), done.
 Delta compression using up to 12 threads
 Compressing objects: 100% (3/3), done.
 Writing objects: 100% (4/4), 353 bytes | 353.00 KiB/s, done.
 Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
  remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
 To https://github.com/justjais/Git_101.git
  5fc0392..cdb75e0  test_workPR -> test_workPR
```

Figure 6.37*: Fixing review comment and pushing the change*

With the review comment fix uploaded, and if everything looks good, we are OK to merge the pull request by clicking on the **Merge pull request** button as shown in the following screenshot:

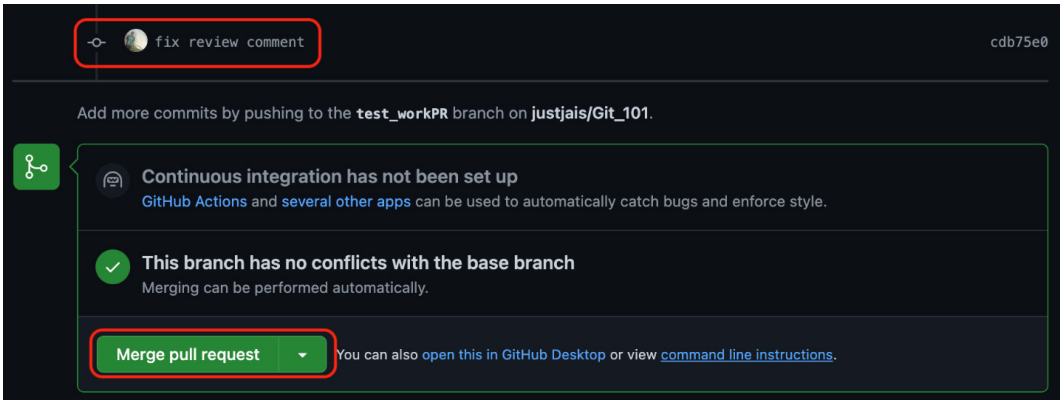

***Figure 6.38**: Merging Pull request*

Writing a great bug report

Your bug report's chances of being fixed increase if it is effective. Therefore, how well you report a defect will determine whether it is fixed. The ability to report a bug is only a skill, and in this section, we will explain to the reader how to develop it.

***Figure 6.39**: Bug report*

The good, the bad, and the ugly of bug reports will all be discussed in this section. Additionally, we will provide some advice on how to create bug/PR reports that your developers will adore.

The developer will most likely reject a bug reported incorrectly by a bug reporter and label it irreproducible.

Characteristics of a quality software bug report

A bug report can be written by anyone. But not everyone can create a good bug report. An ordinary bug report and a good bug report should be distinguishable from one another.

How do you tell a good bug report from a bad one? Apply the traits and methods listed below to report an issue:

- **Having a clearly defined bug number**

  Give each bug report a special identification number. You may then use this to locate the bug record. This special number will be produced automatically each time you report a bug if you are using an automated bug-reporting program. Keep track of how many bugs you reported, along with a brief explanation of each one.

- **Reproducible**

  A bug that cannot be reproduced will never be fixed. The procedures to reproduce the bug should be stated in unambiguous terms. Make no assumptions or shortcuts when reproducing. It is simple to reproduce and fix the bug that is presented step by step.

- **Be Particular:** Avoid writing an essay about the issue.

  To ensure that the bug reports adhere to the best practices established by testers and developers throughout time, you can use the simple checklist.

- **Title**

  Be short and precise. Make sure the bug's description is concise and includes the location or category. The developer will have an easier time subsequently finding and merging any duplicates if your report has a clear title.

  When developers analyze it, they will be able to identify the problem right away and determine whether or not to investigate it by examining the other components of the bug report.

- **Summary**

  You might include a brief report summary if you feel your title is insufficient.

  By brief we mean, give the time and circumstances around the bug in as few words as you can. Important keywords should be included in your title and description because they might be used in searches, as was already explained.

- **Issue Type**

  Giving the person who triages the bug report information about the bug report—whether it is a bug request, feature request, or doc fix update—gives them an early sense of whom to allocate the relevant bug.

  This facilitates quick action against the reported bug.

- **Component and version information**

  The brief name of the following module, plugin, task, or feature; if in doubt, make your best assumption.

  Reporting the software version, you noticed the bug in will also assist developers prioritize fixing it in the version you reported.

- **Expected vs. actual results**

  Now, spend some time explaining to your developer what you anticipated (this is commonly referred to as the user story) and what occurred. For example:

  ```
  Expected result: The format for the date should be "dd/mm/yyyy".

  Actual result: The date is displayed in the format "mm/dd/yyyy"
  instead.
  ```

- **Steps to reproduce**

  This is your chance to share the instructions required to reproduce the bug! Always assume that your developer is unaware of the bug you have discovered, and how they can fix it.

  The steps to be taken should be detailed, simple to comprehend, and condensed. The main objective of this stage is for your developer to encounter the bug.

- **Screenshot**

  There are a thousand words in a picture. Take a screenshot of the failure with the appropriate captions to show the flaw. Light red hue is used to highlight unexpected error notifications. This highlights the necessary area.

- **Environment**

  Depending on the environment, websites and apps might behave extremely differently. For developers, this is very crucial information.

- **Severity and priority**

  Your developer can determine how quickly a bug should be repaired by specifying the issue's severity or priority. The extent of the impact your bug has on your website or product can be used to gauge its severity.

After determining this, you can categorize it as:

- Critical
- Major
- Minor
- Trivial
- Enhancement

Repository/Content developer can choose which bug to look at and address first, based on the priority. Here, you have three options:

- High
- Medium
- Low

You will often be in charge of determining the severity and priority as the bug reporter.

Effective bug reporting

An essential component of software testing is bug reporting. Effective bug reports communicate clearly with the development team to prevent misunderstandings or errors.

A good bug report should be concise and unambiguous, without any important details being omitted. Any ambiguity causes miscommunication and slows down the development process. One of the most crucial yet underappreciated aspects of the testing life cycle is the documenting and reporting of defects.

When reporting an issue, excellent writing is crucial. The main thing a bug reporter should remember is to avoid writing in a directive manner in the report. This undermines morale and fosters unproductive working relationships.

Do not presume that the developer erred and that you can be tough with them. It is crucial to confirm whether the bug has already been reported before filing a complaint.

The testing process is hampered by *duplicate* bugs. View the complete list of known bugs. The developers may occasionally be aware of the problem and choose to

disregard it in the next releases. It is also possible to use any of the available bug tracking tools, which check for duplicate bugs automatically. To find any *duplicate* bugs, it is better to search manually.

You need to ensure that the report should provide information about how and where the crucial details are. The report should specify in detail how the test was conducted and the precise location of the issue. The bug should be simple for the reader to recreate and locate.

Remember that the goal of creating a bug report is to help the developer see the issue. They need to understand the flaw from the bug report clearly. Do not forget to give the developer all the information they need.

Additionally, keep in mind that a bug report will be saved for later use and should be properly written with the necessary details. To describe your bugs, use clear sentences and straightforward language. Avoid making comments that are unclear and waste the reviewer's time.

Each bug should be *reported separately*. If a bug report has numerous issues, you cannot close it until every issue is fixed.

Therefore, it is best to *break* the *problems* into *different bugs*. This guarantees that every bug can be dealt with individually. A developer can replicate a bug at their terminal with the aid of a well-written bug report. They can also diagnose the problem with the aid of this.

Pushing code and opening a pull request over GitHub

The Feature Branch Workflow, the *Gitflow* Workflow, and the Forking Workflow can all be used with pull requests. Pull requests, on the other hand, necessitate two independent branches or repositories, thus they would not work with the Centralized Workflow.

Pull requests are used differently in each of these workflows, but the general procedure is as follows:

1. On their local repo, a developer creates the feature in a dedicated branch.
2. The branch is pushed to a public GitHub repository by the developer.
3. The developer submits a GitHub pull request.
4. The reviewer/repositories maintainer of the team looks at the code, debates it, and makes changes. The ideal flow should be to incorporate most of the change in a branch, test, and then incorporate back to the main branch.

5. The feature is merged into the official repository, and the pull request is closed by the project maintainer.

You generate and submit a pull request when you write code that you want to contribute to a repository. Some proposed changes to the target repository are included in your code. A pull request is how you submit these modifications to the repository's maintainer. It allows repository administrators to assess the modifications and decide whether to accept, reject, or request additional changes.

Summary

Bug report must without a doubt be a professional-grade document. Since this is the primary means of communication between the tester, developer, and manager, concentrate on producing effective bug reports and give it some time.

The fundamental duty of every tester is to provide a quality bug report, and managers should make their staff aware of this. Your efforts to produce a quality bug report will not only conserve corporate resources but also foster a positive working relationship between you and the developers.

Conclusion

The topics we have discussed in this chapter are the building block and soul of the Git pull request and its associated used terms.

The software development lifecycle includes two crucial processes: pull requests and code reviews. As a result, there is also plenty of excellent advice available for carrying out these tasks properly, more than we can cover in a single chapter.

In the next chapter, we will continue to build on the knowledge we gained in this chapter and discuss a few of the Git concepts in more depth.

Multiple choice questions

1. **Command to check all the remote branch in your local Git repository?**
    a. git remote
    b. git remote -v
    c. git remote -a
    d. git pull

2. **Command to checkout branch from your main branch?**
    a. git checkout
    b. git checkout -a
    c. git checkout -b
    d. git checkout –b <branch_name> origin/main
3. **How can you tag a Bug ID under your PR description?**
    a. #BUGID
    b. !BUGID
    c. @BUGID
    d. &BUGID
4. **How can you notify a member from the team in the pull request?**
    a. #USERNAME
    b. &USERNAME
    c. @USERNAME
    d. %USERNAME
5. **Can a developer who has raised a PR approve the PR?**
    a. True
    b. False
6. **What are the usual bug priority levels?**
    a. high
    b. medium
    c. low
    d. All the above

Answers

1. **b**
2. **d**
3. **a**
4. **c**
5. **b**
6. **d**

Further readings

For more information and reference around the discussed topics of this chapter you can check out the Git official documentation for getting started and GitHub documentation

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 7

Tags and Releases Using Git

This chapter discusses all the Git and GitHub related processes and critical aspects that should be kept in mind. Git places only four types of objects in the object store: the blobs, trees, commits, and tags.

Managing releases using Git and the GitHub is a simple and uncomplicated process, and we will learn all about the concepts and underlying commands used to achieve the process.

Structure

In this chapter, we will cover the following topics:

- Release tags versus release branches
- Git tag
- Git list tag
- Git branch
- Cherry-Pick commit for reuse
- Git Stash for code reusability

Objectives

This chapter builds on what we learned in the previous chapter and allows us to make a final decision before pushing and committing changes to source control.

This process of committing changes to the GitHub repo may entail renaming, deleting, or ignoring files in the project, and all these ideas will be covered in detail in this chapter.

This chapter assumes that you have an up and running Git on your Linux, windows or Mac machines.

Release tags versus release branches

When you are working on many features or have a team working on multiple features/bugs, branches come in handy. In such a situation, say a team of 5, each individual can be allocated a different issue. They can work on and fix that issue on his or her branch (most Git hosting services can do this automatically).

If you are done developing the feature and tested it, you can simply make a merge request to have it merged with the main branch. This helps in ensuring that:

- Before merging the code to the main branch, the project owner can inspect it and do a code review before pushing it to the main branch.
- Only well-reviewed work is pushed to the main branch, making sure that the code is always stable.

Assume you have coded numerous features this way and merged them into the main branch. Your product is now ready to be released as a new version. Tagging comes handy in this situation. After a few modifications have been merged into main, users can tag the most recent merged commit with a new release tag, such as version-1.1.

The added benefit of tagging is that you may include change logs and release notes in each of the tags you work on, ensuring that your development cycle remains consistent over time.

Git has CI functionality that can produce tagged releases to a container and deploy them automatically.

Tagging can substantially aid your "release cycle" in the following ways:

1. Merge features into main from branches.
2. Add tags when you have a large number of features.
3. Automatic deployment to a staging server.

4. Test the release.
5. Release it to production if everything looks good.

So, branches come in handy for scenarios where you have a large team, Branching can help you code unique features or fix bugs (while keeping the main branch clean).

Tags are useful for remembering certain points in your commit history, such as releases. If something goes wrong after a release, you can always go back to the previous tag, fix the problem, and re-release it.

Git Tag

Tags mark a particular point in Git's history. Tags are used to indicate whether a commit stage is relevant. A commit can be tagged for future reference, and can be mostly used to indicate the beginning of a project, such as v1.0.

Tags are similar to branches, they do not alter. A branch or separate branches can have any number of tags. The tags on various branches are shown in the following figure:

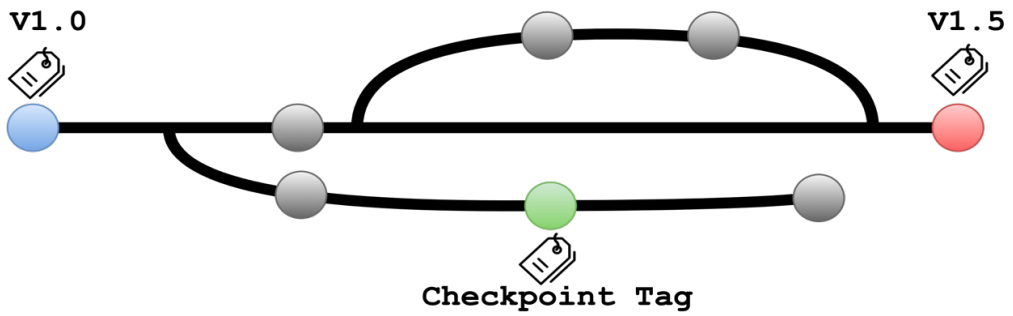

***Figure 7.1**: Git tag*

There are two types of tags:

- Annotated tag
- Light-weighted tag

Both tags are similar, but they differ in terms of the number of Metadata they can store.

Git Create tag

To make a tag, go to the branch where you wish to make a tag and checkout.

```
→  Git_101 git:(main) git tag <tagname>
```

Figure 7.2: Git create tag

Replace **<tagname>** with a meaningful identifier for the repo's current state at the time the tag is created. The use of version numbers, such as git tag v1.0, is a typical pattern. Annotated and lightweight tags are the two types of tags supported by Git. The lightweight tag was produced in the preceding example.

The quantity of associated metadata stored by Lightweight tags and Annotated tags differs. Annotated tags should be considered public, while Lightweight tags should be considered private. Extra metadata is stored in annotated tags, such as the tagger's name, email, and date. This is critical information for a public release. Lightweight tags are simply committed 'bookmarks,' consisting of only a name and a pointer to a commit.

Annotated tag

Annotated tags are tags that store additional metadata such as the developer's name, email address, date, and other information. In the Git database, they are saved as a collection of objects.

It is advisable to create an annotated tag when pointing and storing a final version of any project. You can create a lightweight tag if you only want to make a momentary mark or do not want to share information.

The data given in annotated tags are required for the project's public release. There are more annotation options available, such as adding a message for project annotation.

```
→ Git_101 git:(main) git tag <tag_name> -m "<Tag Message>"
```

Figure 7.3: Git annotated tag

The command above will make a tag with a message. Annotated tags include extra information such as the author's name and other project-related details.

```
→  Git_101 git:(main) git tag git101v1.0 -m "release point for git101v1.0"
```

Figure 7.4: Git annotated tag in action

In the main branch of my project's repository, the above command will produce an annotated tag git101v1.0.

When we present an annotated tag, extra information about the tag will be displayed.

```
➜  Git_101 git:(main) git show git101v1.0
tag git101v1.0
Tagger: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Sun Sep 12 13:34:48 2021 +0530

 release point for git101v1.0

commit 652ec3f1aac1afccc93e12e6068d665f07525ba5 (HEAD -> main, tag: git101v1.0)
Author: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

   move file dir

   interactive new msg in file

diff --git a/learn_git.txt b/temp/learn_git.txt
similarity index 100%
rename from learn_git.txt
rename to temp/learn_git.txt
```

***Figure 7.5**: Git show tag*

Light-weighted tag

Another sort of tag supported by Git is the Light-weighted tag. Both tags have the same goal: to mark a location in the repository. It is usually a commit that is saved in a file. To make it light-weight, it does not store unwarranted data.

This command generates a lightweight tag with the name **`v1.0`**. The **`-a`**, **`-s`**, and **`-m`** parameters are not used to create lightweight tags. Lightweight tags create a new tag checksum and store it in the **`.git/`** directory of the project's repo.

```
➜  Git_101 git:(main) git tag git101v1.0
```

***Figure 7.6**: Git lightweight tag*

The above output will generate the projectv1.0 light-weight tag. It will have a smaller output than a tag that has been annotated.

```
➜  Git_101 git:(main) git show git101v1.0
commit 652ec3f1aac1afccc93e12e6068d665f07525ba5 (HEAD -> main, tag: git101v1.0)
Author: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

    interactive new msg in file

diff --git a/learn_git.txt b/temp/learn_git.txt
similarity index 100%
rename from learn_git.txt
rename to temp/learn_git.txt
```

Figure 7.7: Git show lightweight tag

Git list tag

We can make a list of the tags that are available in our repository. There are three ways for displaying the tags in the repository:

- git tag
- git show
- git tag -l ".*"

Git tag is the most common way to get a list of all the tags available in the repository.

```
➜  Git_101 git:(main) git tag
git101v1.0
(END)
```

Figure 7.8: Git tag output

To check details about specific tag, **`git show`** command comes in handy as:

```
➜  Git_101 git:(main) git show git101v1.0
commit 652ec3f1aac1afccc93e12e6068d665f07525ba5 (HEAD -> main, tag: git101v1.0)
Author: Sumit Jaiswal <sjaiswal@redhat.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

    interactive new msg in file

diff --git a/learn_git.txt b/temp/learn_git.txt
similarity index 100%
rename from learn_git.txt
rename to temp/learn_git.txt
```

***Figure 7.9**: Git show output*

It is also a one-of-a-kind command-line utility that uses a wildcard pattern to display the available tags. Assume we have ten tags, such as v1.0, v1.1, v1.2, and so on, up to v1.10. Then, using the tag pattern v, we can list all the v patterns.

```
➜  Git_101 git:(main) git tag -l "<pattern>.*"
```

***Figure 7.10**: Git tag pattern*

As you can see from the above CLI output, I was able to extract the two tags beginning with **`git`** as output and filter out the tag named **`test project`**.

```
➜ Git_101 git:(main) git tag
 git101v1.0
 git101v1.1
 test_projectv1.0
➜ Git_101 git:(main) git tag -l "git*"
 git101v1.0
 git101v1.1
```

***Figure 7.11**: Git tag pattern in action*

Tagging old commits

Git tag will create a tag on the commit that **`HEAD`** is referencing by default. Alternatively, you can use git tag to refer to a specific commit. Instead of defaulting to **`HEAD`**, the given commit will be tagged.

Use the **`git log`** command to get a list of older commits:

```
→ Git_101 git:(main) git log --prety=oneline
 652ec3f1aac1afccc93e12e6068d665f07525ba5 (HEAD -> main) move file dir
 febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453 reset
 136f2a9877ee78d70347fda4c33740ac3d92e7fa move file directory
 9a1b8ff6a66c971ea802011b285326b1138d4b3b Merge pull request #1 from justjais/dev_branch_1
 09cd056e67c3e96dba761bcd18d38a0c44f3fbdd (origin/dev_branch_1, dev_branch_1) remove unwanted
```

***Figure 7.12**: Git log output*

The command git log produces a list of commits. We will use the top-most commit Merge branch 'feature' for the new tag in this example. To tag the Git, we will need to refer to commit SHA hash:

```
→ Git_101 git:(main) git tag -a test_v1.0 652ec3f1aac1afccc93e12e6068d665f07525ba5
```

***Figure 7.13**: Git tag commit*

The **`Git tag`** command above will produce a new annotated commit for the commit we picked in the previous git log example, labelled **`test_v1.0`**.

```
→ Git_101 git:(main) git log --pretty=oneline
 652ec3f1aac1afccc93e12e6068d665f07525ba5 (HEAD -> main, tag: test_v1.0) move file dir
 febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453 reset
 136f2a9877ee78d70347fda4c33740ac3d92e7fa move file directory
 9a1b8ff6a66c971ea802011b285326b1138d4b3b Merge pull request #1 from justjais/dev_branch_1
 09cd056e67c3e96dba761bcd18d38a0c44f3fbdd (origin/dev_branch_1, dev_branch_1) remove unwanted
```

***Figure 7.14**: Git commit id tagged*

Git Push tag

We can add tags to a project on a remote server. It will aid other team members in determining where to look for an update.

On a remote server account, it will appear as a release point. The git push command makes it easier to push tags by providing some special arguments.

- Git push origin **`<tagname>`**

Using the **`git push`** command, we may push any specific tag, as:

```
→ Git_101 git:(main) git push origin <tagname>
```

***Figure 7.15**: Git push origin*

The command above creates a release point for the supplied tag name. Consider the following scenario:

I have added some tags to my local repository that I want to push to my GitHub account. Then I must execute the command listed above. Consider the following figure, which depicts the present state of my remote repository.

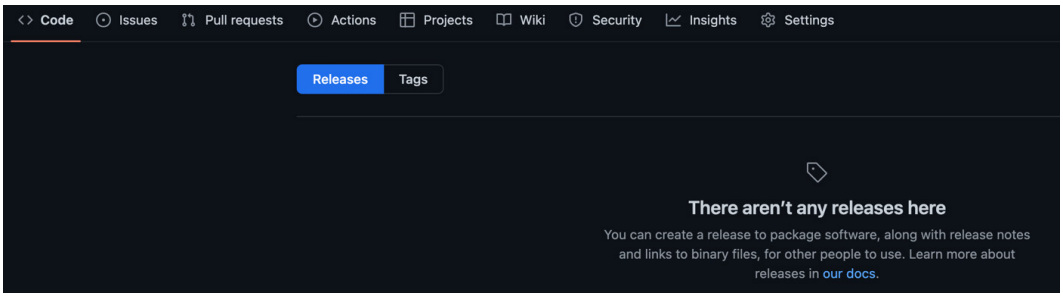

***Figure 7.16**: GitHub repo page (remote repository)*

Release at this point is shown as none. Now, if we push the tag to our remote repository as:

```
→ Git_101 git:(main) git push origin testProject_v1.0
 Enumerating objects: 6, done.
 Counting objects: 100% (6/6), done.
 Delta compression using up to 12 threads
 Compressing objects: 100% (4/4), done.
 Writing objects: 100% (4/4), 555 bytes | 555.00 KiB/s, done.
 Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
 To https://github.com/justjais/Git_101.git
  * [new tag]         testProject_v1.0 ->
```

***Figure 7.17**: GitHub push origin tag output*

My **`testProject_v1.0`** tag has been uploaded to the remote repository. The repository's current state will be changed.

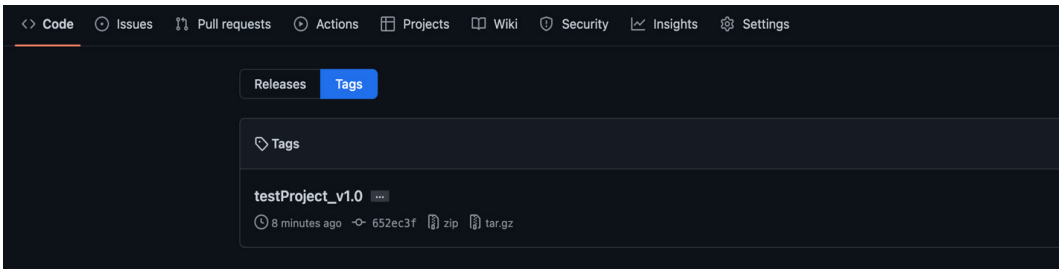

***Figure 7.18**: GitHub remote repo is tagged*

We can download it as a zip and tar file.

- The git push origin --tag/ git push --tags:

This command will simultaneously push all the available tags. It will generate as many release points as there are tags in the repository.

```
→ Git_101 git:(main) git push origin --tags
```

***Figure 7.19**: GitHub push origin –tags*

All accessible tags from the local repository will be pushed to the remote repository using the preceding command:

```
→ Git_101 git:(main) git tag
 git101v1.0
 git101v1.1
 testProject_v1.0
 (END)
→ Git_101 git:(main) git push origin --tags
 Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
 To https://github.com/justjais/Git_101.git
 * [new tag]         git101v1.0 -> git101v1.0
 * [new tag]         git101v1.1 -> git101v1.1
```

***Figure 7.20**: GitHub push origin –tags output*

The release point has been updated as tags have been pushed to the remote server origin. Take a look at the following repository snapshot:

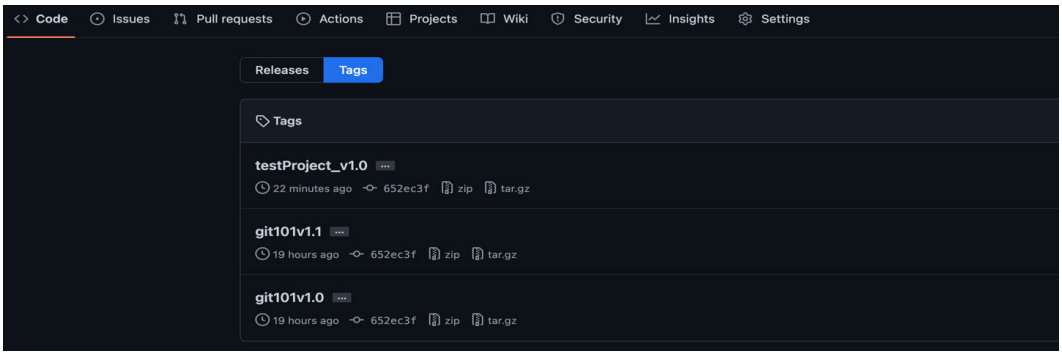

***Figure 7.21**: GitHub remote repo updated with all tags*

In the above result, the release point is modified based on tags. All 3 tags are updated now as expected.

Git Delete tag

At any time, Git allows you to remove a tag from the repository. Run the following command to remove a tag:

```
→ Git_101 git:(main) git tag -d <tagname>
→ Git_101 git:(main) git tag --delete <tagname>
```

Figure 7.22: Git delete tag command

The command above will remove a specific tag from the local repository. If I wish to remove my tag **`git101v1.0`**, it can be done as:

```
→ Git_101 git:(main) git tag -d git101v1.0
 Deleted tag 'git101v1.0' (was 652ec3f)
```

Figure 7.23: Git delete tag output

The tag **`git101v1.0`** has been deleted from the local repository.

Delete remote repository tag

A tag can also be removed from the remote server. Run the following command to remove a tag from the remote repository:

```
→ Git_101 git:(main) git push origin -d <tagname>
→ Git_101 git:(main) git push origin --delete <tagname>
```

Figure 7.24: Git tag delete from remote repo

The command above will remove the tag supplied from the remote repository.

```
→ Git_101 git:(main) git push origin --delete git101v1.0
 To https://github.com/justjais/Git_101.git
 - [deleted]         git101v1.0
```

Figure 7.25: Git tag delete from remote repo output

Delete multiple tags

With a single command, we may erase multiple tags. To delete multiple tags at the same time we use the following command:

- From Local Repository

```
→ Git_101 git:(main) git tag -d <tag1> <tag2>
→ Git_101 git:(main) git tag --delete <tag1> <tag2>
```

Figure 7.26: Git multiple tag delete from local repo

- From remote repository:

```
→ Git_101 git:(main) git push origin -d <tag1> <tag2>
→ Git_101 git:(main) git push origin --delete <tag1> <tag2>
```

***Figure 7.27**: Git multiple tag delete from remote repo*

Git checkout tags

Using the **`git checkout`** command, you may see the current status of a repo at a certain tag.

```
→ Git_101 git:(main) git checkout <tagname>
```

***Figure 7.28**: Git checkout tags output*

The repo is placed in a detached HEAD state after checkout. This implies that any modifications you make will not be reflected in the tag. They will start over with a new detached commit. This new detached commit will not belong to any branch and will only be accessible via the SHA hash of the commit.

As a result, whenever you make changes in a detached HEAD state, it is advisable to create a new branch.

Retagging/Replacing old tags

Git will throw an error if you try to create a tag with the same identifier as an existing tag:

```
→ Git_101 git:(main) git tag git101v1.1
  fatal: tag 'git101v1.1' already exists
```

***Figure 7.29**: Git tag on existing tags*

Git will also throw an error if you try to tag a previous commit with an existing tag identifier.

If you need to change an existing tag, you must use the **`-f FORCE`** option.

```
→ Git_101 git:(main) git tag -a -f git101v1.1 652ec3f1aac1afccc93e12e6068d665f07525ba5
```

***Figure 7.30**: Git tag on existing tags output*

The **`652ec3f1aac1afccc93e12e6068d665f07525ba5`** commit will be mapped to the **`git101v1.1`** tag identifier when the above command is run.

It will overwrite any existing **`git101v1.1`** tag content.

Tagging is another way for creating a snapshot of a Git repository. Tagging has long been used to construct meaningful version number identifier tags for software release cycles. The primary driver of tag generation, modification, and deletion is the **`git tag`** command.

Annotated tags and lightweight tags are the two types of tags. Annotated tags are often preferable since they hold more useful meta data about the tag.

Git branch

Most current version control systems include branching as a feature. Branching in other VCSs can be a time and disc-space-consuming procedure. Branches are an integral aspect of the Git development process. They provide a shortcut to a snapshot of your modifications.

You create a new branch to encapsulate your changes when you wish to add a new feature or solve a bug, no matter how big or tiny. This makes it more difficult for unstable code to be merged into the main code base, and also allows you to clean up the history of your future branch before merging it into the main branch.

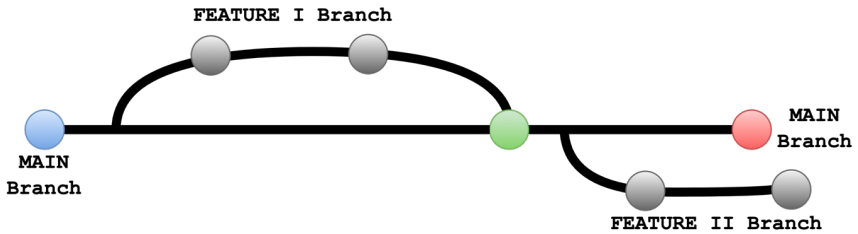

***Figure 7.31**: Git Branch*

The preceding figure depicts a repository with two development lines, one for feature I and the other for a feature II. It keeps the main branch clean of problematic code.

Git branches' implementation is far more lightweight than other version control system architectures. Git keeps a branch as a reference to a commit, rather than transferring files from directory to directory. A branch, in this sense, indicates the end of a sequence of commits rather than a container for commits. The commit relationships are used to derive a branch's history.

Git main branch

In Git, the default branch name for new repositories created on GitHub is now the *main*. It is created when the project's first commit is made. You are assigned a main branch to the starting commit point when you make your first commit. When you start committing, the main branch pointer advances automatically. There can only be one main branch in a repository.

The main branch is the one where all the changes are eventually merged back in. It is possible to think of it as your project's official working version.

Operations on branches

On Git branches, we may conduct a variety of tasks. Create, list, rename, and delete branches with the **`git branch`** command. The git checkout and git merge commands perform a variety of operations on branches. As a result, the git branch works in tandem with the git checkout and git merge commands.

The following are the operations that can be carried out on a branch:

- **Create branch**

  The git branch command can be used to create a new branch.

```
→ Git_101 git:(main) git branch <branch name>
```

***Figure 7.32**: Git Branch command*

We can use the following command to create the B1 branch in the Git directory locally:

```
→ Git_101 git:(main) git branch test_branch
```

***Figure 7.33**: Git Branch command working*

- **List branch**

  We can list all the available branches in your repository by using any of two git branch command:

```
→ Git_101 git:(main) git branch --list
→ Git_101 git:(main) git branch
```

***Figure 7.34**: Git Branch list*

Both commands display a list of the repository's available branches. The symbol * denotes a branch that is currently active which in this case is the main branch.

```
→ Git_101 git:(main) git branch
  dev_branch_1
  * main
  test_branch
→ Git_101 git:(main) git branch --list
  dev_branch_1
  * main
  test_branch
```

***Figure 7.35**: Git Branch list output*

In this case we have just three branches, one named as **`dev_branch_1`**, other as **`test_branch`** which we just created and the main branch.

- **Delete Branch**

  You can delete a branch without losing any history once you are done working on it and merged it into the *main* branch.

```
→ Git_101 git:(main) git branch -d <branch name>

→ Git_101 git:(main) git branch --delete <branch name>
```

***Figure 7.36**: Git Branch delete supported commands*

`--delete/-d/-D` any of the 3 delete supported commands will delete the existing branch **`test_branch`** from the local repository.

```
→ Git_101 git:(main) git branch --delete test_branch
  Deleted branch test_branch (was 652ec3f).
```

***Figure 7.37**: Git Branch delete command run*

- Delete Remote Branch

  To delete a remote branch from the Git remote repository, we need to use the following command:

```
→ Git_101 git:(main) git push origin --delete <branch name>
```

***Figure 7.38**: Git Remote Branch delete*

Using the remote branch delete command, we have deleted the **dev_branch_1** from the remote repository.

```
→ Git_101 git:(main) git push origin --delete dev_branch_1
 To https://github.com/justjais/Git_101.git
 - [deleted]         dev_branch_1
```

***Figure 7.39**: Git Remote Branch delete in action*

- Switch Branch

  You can switch between branches without committing with Git. The git checkout command allows you to swap between two branches.

```
→ Git_101 git:(main) git checkout <branch name>
```

***Figure 7.40**: Git Switch branch syntax*

To switch from **main** branch to **test_branch**, we need to **checkout** the **test_branch:**

```
→ Git_101 git:(main) git checkout test_branch
 Switched to branch 'test_branch'

→ Git_101 git:(test_branch) git branch
   main
 * test_branch
```

***Figure 7.41**: Git Switch main->test_branch*

Vice-versa, if we need to switch from **test_branch** to **main** branch, we need to *checkout* the **main** branch as:

```
→ Git_101 git:(test_branch) git checkout main
 Switched to branch 'main'

→ Git_101 git:(main) git branch
 * main
   test_branch
```

***Figure 7.42**: Git Switch test_branch->main*

- **Rename branch**

  Using the **`Git branch`** command, we may rename the branch:

```
→ Git_101 git:(main) git branch -m <old branch name> <new branch name>
```

***Figure 7.43**: Git rename branch syntax*

To rename the existing **`test_branch`** to **`rename_test_branch`**, we can run the following command:

```
→ Git_101 git:(main) git branch -m test_branch rename_test_branch
→ Git_101 git:(main) git branch
 * main
   rename_test_branch
```

***Figure 7.44**: Git rename branch in action*

The **`Git branch`** command and Git's branching behavior were discussed in the above section. The major functions of the git branch command are to create, list, rename, and delete branches.

Cherry-Pick commit for reuse

`git cherry-pick` is a command that allows you to pick random Git commits by reference and append them to the current working HEAD. Picking a commit from one branch and applying it to another is known as cherry-picking.

For undoing modifications, **`git cherry-pick`** can be handy. Let us imagine a commit is made to the wrong branch. You can now switch to the correct branch and cherry-pick the commit for placement.

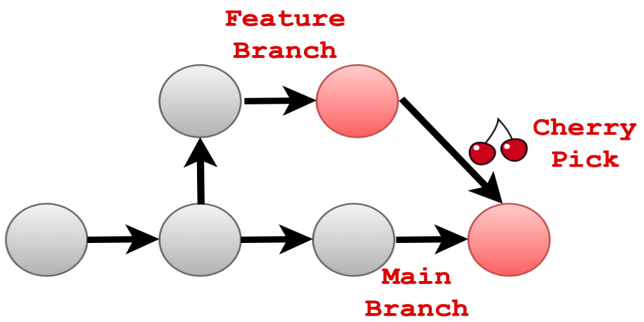

***Figure 7.45**: Git cherry-pick*

The primary goal of a cherry-pick is to apply the modifications made by a previous commit. A cherry-pick examines a prior commit in the repository's history and applies the changes from that commit to the current working tree. Although the definition is simple, cherry-picking a commit or even cherry-picking from another branch becomes more difficult.

Cherry-picking is a useful tool but is not always the best decision. It may result in duplicate commits and other situations in which alternate merges are favored over cherry-picking. It comes in handy in a few situations. It differs from other methods, such as the merge and rebase commands. Merge and Rebase can be applied to many commits in another branch.

Need for Cherry-Picking

Assume you are working on a medium- to large-scale project with a team of developers. You wish to apply some of the changes suggested by another team member to your main project, but not all of them. Since coordinating changes across several Git branches can be difficult, you may not want to merge one entire branch into another.

Only one or two explicit commits are required. Cherry-picking is the process of incorporating changes from other branches into your main project branch.

Cherry-Picking scenarios:

- **Accidently make a commit in a wrong branch**

  Git cherry-pick is useful for applying changes that were made on the wrong branch. Assume we wish to commit to the master branch, but accidentally commit to another branch.

```
→ Git_101 git:(test_branch) touch temp/new_main_file.txt
→ Git_101 git:(test_branch) git add temp/new_main_file.txt
→ Git_101 git:(test_branch) git commit -m "new proposed changes meant for main branch"
 [test_branch a6edc53] new proposed changes meant for main branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 temp/new_main_file.txt
```

***Figure 7.46**: Git cherry-pick in action - I*

  In the preceding example, I intended to commit to the **`main`** branch, but mistakenly committed to the **`test_branch`**. We will use **`git pull`** to merge all the changes from the **`test_branch`** into the **`main`** branch, but for this particular commit we will use **`git cherry-pick`**.

```
→ Git_101 git:(test_branch) git log
 commit a6edc534744533472450ec924fcccea41425709b (HEAD -> test_branch)
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sun Sep 12 23:49:07 2021 +0530

   new proposed changes meant for main branch

 commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (origin/main, origin/HEAD)
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Mon Aug 30 12:11:13 2021 +0530

   move file dir
```

***Figure 7.47**: Git cherry-pick in action - II*

To verify the commit history, I used the **`git log`** command. Copy the commit-id for the main branch that you want to make. Switch to the main branch now and cherry-pick it.

```
→ Git_101 git:(test_branch) git checkout main
 Switched to branch 'main'
→ Git_101 git:(test_branch) git cherry-pick a6edc534744533472450ec924fcccea41425709b
 [main 31b84f0] new proposed changes meant for main branch
 Date: Sun Sep 12 23:49:07 2021 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 temp/new_main_file.txt
→ Git_101 git:(test_branch) git log
 commit 31b84f0025e2c149810df792c63d765e7b5d080e (HEAD -> main)
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sun Sep 12 23:49:07 2021 +0530

  new proposed changes meant for main branch
```

***Figure 7.48**: Git cherry-pick in action – III*

We can see from the output that I used the **`git cherry-pick`** command to paste the commit id and added it to my **`main`** branch. We can verify it using the **`git log`** command.

- **Changes suggested by another team member were implemented**

  Adjusting as suggested by another team member is another example of cherry-picking. Assume one of my team members made a change to the primary project and recommends it to the rest of the team.

  After you have reviewed it, you can cherry-pick it.

  Cherry picking is a powerful and convenient command that comes in handy in a variety of situations.

Git Stash for code reusability

Git Stash stores (or stashes) modifications you have made to your working copy so you can work on something else and then come back to the changes made earlier. When you wish to transfer branches but are working on an unfinished part of your current project, it is sometimes difficult to stash current branch changes. You do not want to commit to work that is not finished. You can do this with Git stashing. You can switch branches without committing the current branch with the **`git stash`** command.

The attributes and role of stashing in terms of repository and working directory are shown in the following figure:

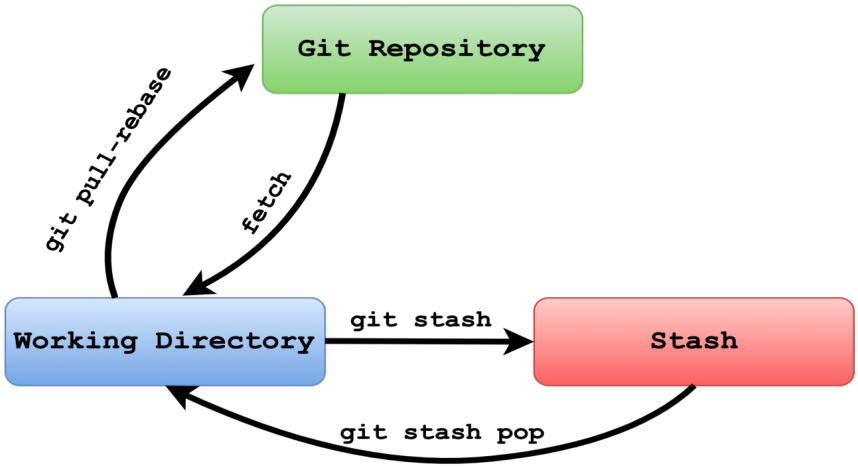

***Figure 7.49**: Git Stash*

The term "stash" means "to safely store something in a secret spot." Git works in the same way as **`git stash`** saves your data safely without committing it.

Stashing saves the state of your working directory, which is currently incomplete, for later use. With **`git stash`**, you have a lot of possibilities. The following are some relevant options:

- Git stash
- Git stash save
- Git stash list
- Git stash apply
- Git stash changes

- Git stash pop
- Git stash drop
- Git stash clear

Git stash branch

Let us look at a real-life example. I made modifications to my **`Git_101`** project in two files from two different branches. I am in shambles, and I have not finished editing any files yet.

As a result, I would like to keep it for later usage. We can save it in its current state by stashing it. Let us take a peek at the repository's present state before we stash. Run the **`git status`** command to see the repository's current status.

Let us now alter the file content in the repo, save the changes, and run the **`git status`** command as follows:

```
➜ Git_101 git:(test_branch)✗ git status
 On branch main
 Your branch and 'origin/main' have diverged,
 and have 2 and 1 different commits each, respectively.
 (use "git pull" to merge the remote branch into yours)

 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:   temp/learn_git.txt

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

***Figure 7.50**: Git status*

You can see from the above result that there is 1 untracked file in the repository: **`temp/learn_git.txt`**. We can use the **`git stash`** command to save it temporarily.

```
➜ Git_101 git:(test_branch)✗ git stash
 Saved working directory and index state WIP on main: 31b84f0 new proposed changes meant for main branch
➜ Git_101 git:(test_branch) git status
 On branch main
 nothing to commit, working tree clean
```

***Figure 7.51**: Git stash in action*

Above result shows that the work is saved with the **`git stash`** command in the specified output. We may look at the repository's status. The output is just stored in its current location and the directory has now been cleaned. We can flip between the branches and work on them.

Save Git Stash

Saving Stashes with the message. Changes can be saved with a message in Git using following command:

```
→ Git_101 git:(test_branch) git stash save <stashing message>
```

***Figure 7.52**: Git stash save syntax*

Following stash will be saved with the provided message, as:

```
→ Git_101 git:(test_branch)✗ git stash save "Stash the updated file changes"
 Saved working directory and index state On main: Stash the updated file changes
→ Git_101 git:(test_branch) git status
 On branch main
 nothing to commit, working tree clean
```

***Figure 7.53**: Git stash save in action*

List Git Stash

To check the Stored Stashes, we can run the following command:

```
→ Git_101 git:(test_branch) git stash list
 stash@{0}: On main: Stash the updated file changes
 stash@{1}: WIP on main: 31b84f0 new proposed changes meant for main branch
```

***Figure 7.54**: Git stash list in action*

As we can see from the above screenshot, since we made two calls to **`git stash`**, both the changes can be tracked via **`git stash list`** command as **`stash@{0}:`** **`stash@{1}:`** and so on.

Apply Git Stash

Using the **`git stash`** command, you can re-apply the modifications you just saved. Use the **`git stash`** command followed by the apply option to apply the commit.

```
→ Git_101 git:(test_branch) git stash apply
 On branch main
 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:   temp/learn_git.txt

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

***Figure 7.55**: Git stash apply in action*

The preceding output restores the previous stash. If you check the repository's status now, you will see the changes that have been made to the file.

To apply a certain commit to several stashes, use the **`git stash apply`** command followed by the stash index id. It is employed in the following ways:

```
→ Git_101 git:(test_branch) git stash apply <stash id>
```

***Figure 7.56**: Git stash apply with id syntax*

If we apply the **`git stash apply`** command with specific ID again, git will throw an error to add/stash the previous stash and then proceed with the changes of stash with specific ID, as:

```
→ Git_101 git:(test_branch) ✗ git stash apply stash@{1}
 error: Your local changes to the following files would be overwritten by merge:
        temp/learn_git.txt
 Please commit your changes or stash them before you merge.
 Aborting
```

***Figure 7.57**: Git stash apply with id in action*

Git stash changes

We can keep track of where the stashes are and how they have changed. Run the following command to observe the changes in the file before and after the stash operation:

```
→ Git_101 git:(test_branch) ✗ git stash show
  temp/learn_git.txt | 2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
```

***Figure 7.58**: Git stash show in action*

We can keep track of every change made to the file. Use the following command to see the file's altered content:

```
→ Git_101 git:(test_branch) ✗ git stash show -p
  diff --git a/temp/learn_git.txt b/temp/learn_git.txt
  index 403df72..910713e 100644
  --- a/temp/learn_git.txt
  +++ b/temp/learn_git.txt
  @@ -1 +1 @@
  -Modify existing file content
  + Modify existing file content
```

***Figure 7.59**: Git stash show partial in action*

Re-applying your stashed changes

With **`git stash pop`**, you can re-apply previously stashed changes, as:

```
→ Git_101 git:(main) git status
 On branch main
 nothing to commit, working tree clean

→ Git_101 git:(main) git stash pop
 On branch main
 Your branch and 'origin/main' have diverged,
 and have 2 and 1 different commits each, respectively.
 (use "git pull" to merge the remote branch into yours)

 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:   temp/learn_git.txt

 no changes added to commit (use "git add" and/or "git commit -a")
 Dropped refs/stash@{0} (a7b001101f82345f8bb247dddb7a69889466eba2)
```

***Figure 7.60**: Git stash pop*

When you pop your stash, the changes are removed from your stash and reapplied to your working copy. Alternatively, yo can use **`git stash apply`** to reapply the changes to your working copy and keep them in your stash:

The **`git stash apply`** and **`git stash pop`** commands are very similar.

The **`stash pop`** command, which deletes the stash from the stack after it is applied, is the key distinction between these two commands.

- **`Stashing Ignored/Untracked files`**

  When you run **`git stash`** by default, it will save the following files:

  - Alterations you have made to your index (staged changes)
  - Alterations to files currently being tracked by Git (unstaged changes)

However, it will **`not`** stash:

- Fresh files that have not been staged in your working copy
- Files that were ignored

If we create a new file temp/new test file.txt but do not stage it and instead do a git stash on the branch, but as the file was not added to the working stage, **`git stash`** will not stash it.

```
→ Git_101 git:(main) ✗ git status
 On branch main

 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:   temp/learn_git.txt

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

 no changes added to commit (use "git add" and/or "git commit -a")
→ Git_101 git:(main) ✗ git stash
 Saved working directory and index state WIP on main: 31b84f0 new proposed changes meant for main branch
```

***Figure 7.61**: Git stash in action for untracked change*

If we check the status again, we can see the newly added file under untracked changes:

```
→ Git_101 git:(main) ✗ git status
 On branch main

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

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

***Figure 7.62**: Git status for untracked change*

The **`-u`** option (or --include-untracked) instructs **`git stash`** to save your untracked files as well:

```
→ Git_101 git:(main) ✗ git stash -u
 Saved working directory and index state WIP on main: 31b84f0 new proposed changes meant for main branch

→ Git_101 git:(main) git status
 On branch main
 no changes added to commit (use "git add" and/or "git commit -a")
```

***Figure 7.63**: Git stash –u in action*

Git stash branch

If you have saved some work on a certain branch and want to keep working on it. Then, during the merging process, it may cause a conflict. As a result, it is a good idea to keep working on a distinct branch.

To avoid conflicts, the **`git stash branch`** command allows the user to stash work on a separate branch.

This command will establish a new branch and transfer all the previously saved work to it.

```
→ Git_101 git:(main) git stash branch <branch name>
```

***Figure 7.64**: Git stash branch*

Git stash cleaning

You can delete a stash using **`git stash drop`** if you decide you do not need it any longer. Git stash list - forget the stash index to be deleted, followed by **`git stash drop stash@`** and the desired index - stash@{index}

```
→ Git_101 git:(main) git stash drop stash@{0}
 Dropped stash@{0} (4f8f08b0e3278b03bd0d55df25269f4bc58b130a)
```

***Figure 7.65**: Git stash drop in action*

Conclusion

This is the end of this chapter and the contents we discussed in this chapter are the building block and soul of the Git version control system. The more fluent you get will all the discussed terms and their respective usages and application, the efficient you will become in using Git and GitHub as well.

In the next chapter, we will continue building on the knowledge we gained in this chapter and discuss a few of the Git concepts in more depth.

Multiple choice questions

1. **Command to filter tag by name/pattern from available Git tags?**
    a. git show
    b. git tag list
    c. git list tag
    d. git tag -l ".*"

2. **Command to delete remote branch from remote Git repository?**
    a. git branch –D
    b. git push origin --delete
    c. git delete branch
    d. git branch origin --delete

3. **Git command to switch between different branches?**
    a. git checkout
    b. git branch
    c. git log
    d. git push

4. **What are the two types of Git tags?**
    a. Hard-weighted tag and Light-weighted tag
    b. Light-weighted tag and Annotated tag
    c. Annotated tag and Hard-weighted tag
    d. None of the above

5. **Command to stash untracked changes?**
    a. git stash pop
    b. git stash clear
    c. git stash apply
    d. git stash –u

6. **Git command to pick random Git commits by reference?**
    a. git branch
    b. git cherry-pick
    c. git tag
    d. git log

Answers

1. **d**
2. **b**
3. **a**

4. **d**
5. **d**
6. **b**

Key terms

- Tag a particular point in Git History
    - Tag
- Relevant Stash options are:
    - Git stash
    - Git stash save
    - Git stash list
    - Git stash apply
    - Git stash changes
    - Git stash pop
    - Git stash drop
    - Git stash clear
    - Git stash branch
- Delete a local branch:
    - git branch -D
    - git branch --delete
- Delete a remote branch from remote repository:
    - git push origin --delete

Points to remember

- Annotated tags and lightweight tags are the two types of tags. Annotated tags are often preferable since they hold more useful meta data about the tag.
- Branches come in handy for scenarios where you have a large team, Branching can help you code unique features or fix bugs.
- Git cherry-pick is useful for applying changes that were made on the wrong branch by mistake.
- To delete all your stashes at once, user can use **`git stash clear`** command.

Further reading

For more history and reference around the discussed topics in this chapter you can check out the Git official documentation for getting started and GitHub documentation:

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 8
Undo or Refresh all the Work Done

The ability to "undo" your mistakes is one of the most useful aspects of any version control system. In Git, the term "undo" can refer to a variety of things.

Git saves a snapshot of your repository at that precise point in time when you make a new commit; later, you can use Git to go back to an earlier version of your project.

This chapter discusses all the process-related and critical aspects that should be kept in mind and followed before undoing changes in Git.

Structure

In this chapter, we will cover the following topics:

- Undo and refresh changes in Git
- Git revert
- Git reset
- Amend Git commit
- Interactive rebase

Objectives

This chapter focuses on Git's undo/refresh feature and covers all the principles that Git exposes to assist users achieve similar functionality, as well as how GitHub processes and workflows can aid in making the job more fluid and efficient.

This chapter assumes that you are up and running with Git on your Linux, Windows or Mac machines. After reading this chapter, you should have all the resources necessary to reverse changes made to a Git repository.

The commands covered in this chapter can be confusing at times, but if you consider how they affect the working directory, staged snapshot, and commit history, it should be simpler to determine which command is appropriate for the current development task.

Undo and refresh changes in Git

At any stage of your development lifecycle using Git and GitHub, you can undo something at any time. We will go over a few basic tools for reversing changes in this section.

However, you cannot always undo some of these undo's', so be careful.

One of the most common undos' occurs when you commit too soon and may forget to add some files or make a mistake in your commit message. This chapter deals with all the underlying concepts for undoing changes in Git.

The advantage of using a version control system is that it keeps track of changes. These records allow us to obtain information such as commits, discover who contributed what, determine where issues were introduced, and undo incorrect changes.

However, all of this information will be useless if we are unable to explore it. The **`git log`** command can help us with this.

Git log is a command for reviewing and reading the history of everything that happens in a repository. This command displays previous commits, allowing you to know who made what changes in the repository.

A **`git log`** can be used with a variety of parameters to make history more specific. The **`git log`** command can be used to list, filter, and display commit history in a variety of ways.

Type the following command into the terminal to run a simple **`git log`** command:

```
→  Git_101 git:(main) git log
```

***Figure 8.1**: Git log*

This will display the entire commit history in an interactive terminal for us to view and navigate as shown in the following screenshot:

```
commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:10:26 2021 +0530

    reset

commit 136f2a9877ee78d70347fda4c33740ac3d92e7fa
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:09:06 2021 +0530

    move file directory

commit 9a1b8ff6a66c971ea802011b285326b1138d4b3b
Merge: 6c30e21 09cd056
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 01:23:20 2021 +0530

    Merge pull request #1 from justjais/dev_branch_1

    Edit Dev branch 1
```

***Figure 8.2**: Git log output*

The **`git log`** is a record of commits in general. These commits are displayed in reverse chronological sequence (the last commit will be shown on the top). It also contains a variety of other details that are incredibly valuable when more than one person is working on the repository. The following information can be found in a git log:

- The **Secure Hash Algorithm** (**SHA**) algorithm generates a 40-character checksum data called a commit hash. It is a one-of-a-kind number.
- Commit Author metadata includes information such as the author's name and email address.
- **Commit Date metadata**: This is a date timestamp for the commit time.
- The commit title/message is a summary of the commit provided in the commit message.

The default log is useful for getting a fast overview of what is going on in the repository. However, it takes up a lot of space, and you can only see a few commits at a time.

Navigating log

Git scrolls through the commit history using the Less terminal pager. You can use the following commands to traverse it:

- To scroll down by one line, press j
- To scroll up by one line, press k
- To scroll down by one page, press the spacebar or the *page down* button
- To scroll up by one page, press the spacebar or the *Page Up* button
- To stop the log, press b or the *Page Up* button

Git log Oneline

If we simply need to list the unique section of the commit id together with the author's message, we may use the ***--oneline*** option to print a single line about each commit. This is a great way to gain a high-level overview of your project.

The output of **`git log --oneline`** will usually look like this:

```
b1b4cca (HEAD -> main, origin/main, origin/HEAD) move file dir
febe99d reset
136f2a9 move file directory
9a1b8ff Merge pull request #1 from justjais/dev_branch_1
```

***Figure 8.3**: Git log –oneline output*

In most cases, the —oneline flag forces the git log to show:

- One commit per line
- The first seven characters of the SHA
- The message of a commit

Using the **`git log`** command, we may limit the number of output commits. If you want fewer commits, use this command to reduce the complexity. Include the **`<n>`** option to limit the output of the git log.

If we just want to see the past **3** commits, we may use the **`git log`** command with the **`-3`** parameter and for condensed information same can be passed with **`--oneline`** command option as shown in the following screenshot:

```
➜  Git_101 git:(main) git log -3
➜  Git_101 git:(main) git log -3 --oneline
```

***Figure 8.4**: Git log limited output*

The **skip** command, on the other hand, will remove the top **<n>** commits.

```
➜  Git_101 git:(main) git log --skip 3
➜  Git_101 git:(main) git log --skip 3 --oneline
```

***Figure 8.5**: Git log --skip*

Git log Log-Size

The log-size option in Git informs you about the log size. After typing the command, it generates an additional line with log-size <number>:

```
➜  Git_101 git:(main) git log --log-size
```

***Figure 8.6**: Git log –log-size*

When you run this command, you will see that a new line called **log size** and number appears in the console. This is the length of the commit message in bytes, as required by many tools. They can allot exact space for saving the commit message in advance by reading this number.

```
commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
log size 102
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
log size 94
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:10:26 2021 +0530

    reset
```

***Figure 8.7**: Git log –log-size output*

Git log Stat

The **log** command shows you which files have been changed. It also includes a summary line showing the total records that have been modified, and the number of lines.

```
➜  Git_101 git:(main) git log --stat
➜  Git_101 git:(main) git log --stat <commit_id>
```

***Figure 8.8**: Git log –stat*

The **stat** option is used to show the following information:

- The modified files
- Number of lines that have been added or removed from the document.
- Summary line of the total number of records that are modified.
- The lines that have been added to or removed from the document.

In *Figure 8.9*, we can see that all the commits reported are repository modifications:

```
commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

 learn_git.txt => temp/learn_git.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:10:26 2021 +0530

    reset

 temp/learn_git.txt | 1 -
 1 file changed, 1 deletion(-)

commit 136f2a9877ee78d70347fda4c33740ac3d92e7fa
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:09:06 2021 +0530

    move file directory

 temp/learn_git.txt | 1 +
 1 file changed, 1 insertion(+)
```

***Figure 8.9**: Git log –stat output*

Git log graph

The **Git log** command displays your Git log as a graph. Run the **git log** command with the **--graph** option to see the commits as a graph.

```
→  Git_101 git:(main) git log --graph
→  Git_101 git:(main) git log --graph --oneline
```

***Figure 8.10**: Git log –graph*

One of the advantages of using this command is that it allows you to see how commits have merged and git history was built.

```
* commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
| Author: Sumit Jaiswal <sumit@email.com>
| Date:   Mon Aug 30 12:11:13 2021 +0530
|
|     move file dir
|
* commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
| Author: Sumit Jaiswal <sumit@email.com>
| Date:   Mon Aug 30 12:10:26 2021 +0530
|
|     reset
|
* commit 136f2a9877ee78d70347fda4c33740ac3d92e7fa
| Author: Sumit Jaiswal <sumit@email.com>
| Date:   Mon Aug 30 12:09:06 2021 +0530
|
|     move file directory
|
```

***Figure 8.11**: Git log –graph output*

The asterisk indicates which branch the commit was made on, therefore, the graph above shows that the **`b1b4cca0`** commit was made on the feature **`xyz`** branch, whereas the others were made on other branches.

Filtering the commit history

We can screen the output to meet our requirements. It is a unique Git feature. On output, we can apply a variety of filters such as amount, date, author, and more. Each filter has its own set of requirements. They can be utilized to implement some output navigation operations.

- Filter commits by author

  We may need to filter commits based on the author's name in some circumstances. To filter and show only the provided author, we will use **`--author`** and enter the author's name as shown in the following screenshot:

```
➜  Git_101 git:(main) git log --author="user_name"
```

***Figure 8.12**: Git log –author*

This command accepts a regular expression and returns a list of authors' commits that match that pattern. You can also use the specific name rather than the pattern as shown in the following screenshot:

```
➜  Git_101 git:(main) git log --author="Sumit"
 commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
 Author: Sumit Jaiswal <sumit@email.com>
 Date:   Mon Aug 30 12:11:13 2021 +0530

     move file dir

 commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
 Author: Sumit Jaiswal <sumit@email.com>
 Date:   Mon Aug 30 12:10:26 2021 +0530

     reset

➜  Git_101 git:(main) git log --author="@email.com"
 commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
 Author: Sumit Jaiswal <sumit@email.com>
 Date:   Mon Aug 30 12:11:13 2021 +0530

     move file dir

 commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
 Author: Sumit Jaiswal <sumit@email.com>
 Date:   Mon Aug 30 12:10:26 2021 +0530

     reset
```

***Figure 8.13**: Git log –author filter output*

We can see that all the commits by the author are listed in the above report. As we know that the author's email is linked to the author's name, we can use the author's email as a pattern or an exact search.

We can achieve this by using wildcards like *"@email.com."* We can track the commits of authors who use different email service like Gmail, Outlook, Yahoo, and so on.

- Filter commits by content

  We may filter commits based on the content of the commit. If we want to search and filter for a specific modification, this will be quite handy. We will use the **`-S`** option and a filter phrase. Keep in mind that this could take a while because it will search all commits that are not indexed for quick search.

  You can use **`git log -S`** to find all occurrences of a string in your repo's history.

It is great for recovering deleted code.

```
➜  Git_101 git:(main) git log -S"value_pattern"
```

***Figure 8.14**: Git log content filter*

- Filter commits by date and time

  The output can be filtered by date and time. To specify the date, we must use the **`--after`** or **`--before`** arguments. Both arguments accept a variety of date formats.

```
➜  Git_101 git:(main) git log --after="yy-mm-dd"
➜  Git_101 git:(main) git log --before="yy-mm-dd"
```

***Figure 8.15**: Git log date and time filter*

Consider the following output:

```
➜  Git_101 git:(main) git log --after="21-08-01"
commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (HEAD -> main, origin/main, origin/HEAD)
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:10:26 2021 +0530

    reset
```

***Figure 8.16**: Git log date and time filter after output*

All commits after "**`2021-08-01`**" are listed in the above command.

We can also keep track of the commits that occurred between the two dates. Pass a statement reference **`--before`** and **`-after`** the date to track the commits made between two dates.

Let us say we want to keep track of commits from "**2021-08-01**" to "**2021-08-21**" The command will be executed as follows:

```
→  Git_101 git:(main) git log --after="21-08-01" --before="21-08-21"
 commit 063eafa082decad7c2b66d4e3dc1313ebdaae061
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sun Aug 1 15:57:23 2021 +0530

     added 8th line

 commit 89ced37e366fbd18775de5ca3524018a85d2c1d2
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sun Aug 1 15:57:13 2021 +0530

     added 7th line

 commit 68a7832616ff9e76d10b97200421521245da5456
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sun Aug 1 15:56:58 2021 +0530

     added 6th line
```

***Figure 8.17**: Git log date and time filter after and before output*

- Filter commits by commit message

```
→  Git_101 git:(main) git log --grep="commit search message"
```

***Figure 8.18**: Git log grep*

The commit message will be used to filter the commits. We can use the grep option, which will function similarly to the author option.

```
→  Git_101 git:(main) git log --grep="reset"
 commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Mon Aug 30 12:10:26 2021 +0530

     reset
```

***Figure 8.19**: Git log grep output*

The result above shows all the commits with the term commit in their commit statement.

There are a variety of other filtering options available, including file name, commit numbers, and more but the above discussed ones are used most.

Git reflog versus Git log

The main difference between **`git reflog`** and **`git log`** is that the log is a public record of the repository's commit history, while the reflog is private.

The **`git log`** is duplicated as a part of the git repository after a push, fetch, or pull. On the other side, the **`git reflog`** is not included. The reflog is a file in **`.gitlogsrefsheads`** that keeps track of local commits for a given branch and excludes any commits that may have been cut away by git trash collection.

The **`git log`**, on the other hand, provides a historical commit traversal for a branch that starts with the most recent commit and ends with the branch's very first commit.

While **`git reflog`** is extremely useful for recovering lost branches and commits, it also has several drawbacks. For example, reference logs are stored locally and cannot be pushed or fetched from a remote repository. They normally expire or are erased after a certain length of time to save disc space.

Git revert

The **`git revert`** command is an 'undo' command and is not a standard undo procedure. Rather than removing the commit from the project history, it determines how to invert the modifications made by the commit and adds a new commit with the inverse content.

This prevents Git from losing history, which is critical for maintaining the integrity of your revision history and collaborating with confidence.

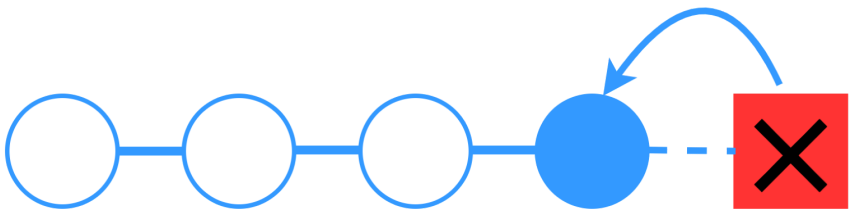

***Figure 8.20**: Git revert in action*

When you wish to apply the inverse of a commit from your project history, reverting is the way to go. This is useful if you are trying to trace down a bug and discover that it was caused by a single commit. Rather than manually going in, fixing it, and committing a fresh snapshot, you may automate it.

```
➜  Git_101 git:(main) git revert
```

***Figure 8.21**: Git revert command*

Furthermore, we can state that git revert records certain new changes that are the polar opposite of earlier commits. Git revert has the following options that are available to the users for reverting the changes.

- **Options**:

  Procedures such as editing, no editing, cleanup, and more are available with Git revert. Let us have a look at these possibilities in more detail:

  `<commit>`

  To undo a commit, use the commit option. The commit reference id is required to roll back a commit. It can be accessed using the **`git log`** command.

  `-e, <--edit>`

  Before retracting the commit, it is useful to change the commit message.

  In the **`git revert`** command, it is a default option.

  `-n/--no edit`

  This option does not initiate the use of a text editor. It will reverse the most recent commit.

  `-n/--no-commit`

  The **`revert`** command, in general, commits by default. The **`no-commit`** option prevents you from committing automatically. Furthermore, if you use this option, your index does not have to match the HEAD commit.

  The no-commit option is useful for undoing the effects of multiple commits in a row on your index.

  `-m parent-number /--mainline parent-number`

  It is used to undo the merger process. We cannot revert a merge in most cases since we do not know which side of the merging should be regarded as the mainline.

We can give the parent's number and revert allows us to undo the change concerning that parent.

Git revert to previous commit

Let us assume you have made a change to your project **`learn_git.txt`** file. Later you remember that you made a mistake by committing to the wrong file or branch.

If you now want to revert the changes, you can. Git gives you the ability to go back and fix your mistakes.

```
→  Git_101 git:(main) ✗ git status
On branch main
Your branch is up to date with 'origin/main'.

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:   temp/learn_git.txt

no changes added to commit (use "git add" and/or "git commit -a")
→  Git_101 git:(main) ✗ git add .
→  Git_101 git:(main) ✗ git commit -m "add new msg in file"
[main 4e1e05d] add new msg in file
1 file changed, 1 insertion(+), 1 deletion(-)
```

***Figure 8.22**: Git status check*

I have made changes to **`learn_git.txt`**, as you can see from the output above. The **`git revert`** command can be used to undo it. We will need a commit to undo the changes.

```
→  Git_101 git:(main) git log
commit 4e1e05d26409e816fcab722694a503c4c3bb3ec3 (HEAD -> main)
Author: Sumit Jaiswal <sumit@email.com>
Date:   Sat Sep 11 19:36:51 2021 +0530

    add new msg in file

commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (origin/main, origin/HEAD)
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:11:13 2021 +0530

    move file dir

commit febe99ddc9d1a8a2b8e75a826c2ad29c2dec8453
Author: Sumit Jaiswal <sumit@email.com>
Date:   Mon Aug 30 12:10:26 2021 +0530

    reset
```

***Figure 8.23**: Git log output*

I have copied the most recent commit-ish to roll back in the above output. On this commit, I will now do a revert operation. It will function as follows:

```
→  Git_101 git:(main) git revert 4e1e05d26409e816fcab722694a503c4c3bb3ec3
[main 939ad08] Revert "add new msg in file"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

***Figure 8.24**: Git revert in action*

The changes made to the repository have been reverted, as you can see from the output above.

The **`git revert`** command is a forward-moving undo operation that allows you to undo changes securely.

A revert creates a new commit that reverses the modifications indicated, rather than deleting or orphaning commits in the commit history.

Git reset

`Git reset` command is a powerful tool for reverting changes. There are three main types of invocation. The command-line parameters corresponding these forms are:

- **`--soft`**
- **`--mixed`**
- **`--hard`**

The three arguments are The Commit Tree (HEAD), The Staging Index, and The Working Directory, which are Git's three internal state management mechanisms.

To put it another way, Git is a tool that resets the current state of HEAD to a specific state. It is a complex and flexible tool for reversing modifications. Git can use it as a time machine. You can navigate between the many commits by jumping up and down. Each of these reset options has an impact on the specific trees that Git uses to process your file's content.

Additionally, **`git reset`** can be used to reset entire commit objects or individual files. Each of these reset options has an impact on the trees that git uses to manage your file and its contents.

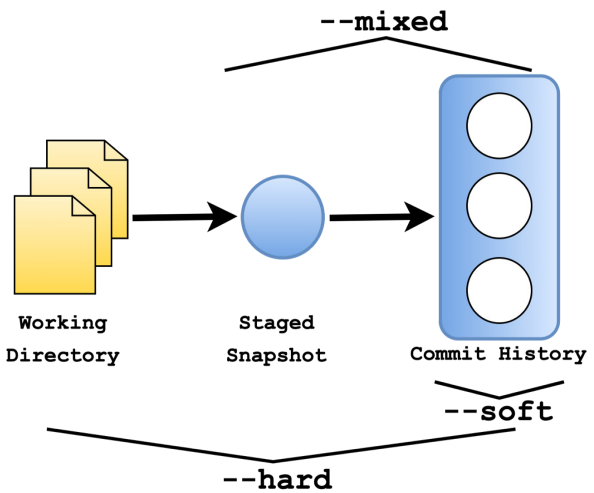

***Figure 8.25**: Git reset*

For creating and reverting commits, Git employs an index (staging area), HEAD, and working directory. If you are not sure what Head, trees, and index mean, check out Git Index and Git Head.

You can change the file and stage into the index from the working directory. You can choose what you wish to include in your next commit in the staging area. A commit object is a hashed version of the information that has been cryptographically signed.

It contains Metadata and points that are needed to turn on earlier commits.

Git reset hard

It will first update the index with the contents of the commits and then move the Head. It is the simplest, riskiest, and most commonly employed method as shown in the following screenshot:

```
➜  Git_101 git:(main) touch temp/test_reset.txt
➜  Git_101 git:(main) ✗ git add temp/test_reset.txt
➜  Git_101 git:(main) ✗ git status
 On branch main
 Your branch is ahead of 'origin/main' by 2 commits.
   (use "git push" to publish your local commits)

 Changes to be committed:
   (use "git restore --staged <file>..." to unstage)

 new file:   temp/test_reset.txt
➜  Git_101 git:(main) ✗ git log
 commit 939ad0854a17191ab2afd5f630a60271ddc1c850 (HEAD -> main)
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sat Sep 11 19:48:30 2021 +0530

     Revert "add new msg in file"

     This reverts commit 4e1e05d26409e816fcab722694a503c4c3bb3ec3.

 commit 4e1e05d26409e816fcab722694a503c4c3bb3ec3
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Sat Sep 11 19:36:51 2021 +0530

     add new msg in file

 commit b1b4cca0cb4009b0be5332b248c8b782e18f302f (origin/main, origin/HEAD)
 Author: Sumit Jaiswal <sjaiswal@redhat.com>
 Date:   Mon Aug 30 12:11:13 2021 +0530

     move file dir
```

***Figure 8.26**: Git reset –hard in action*

The **`--hard`** option modifies the Commit History and updates the ref pointers to the selected commit. The staging index and working directory must then be reset to reflect the commit that was selected. The working directory and any previously outstanding commits to the staging index are reset to match the commit tree. Any pending work will be lost as a result.

1. I have added a file named **`test_reset.txt`** to the output above and double-checked the repository's status. Because I have not committed the

modifications, we can see that the current head position has not altered. I am going to use the reset **--hard** option now.

2. The **--hard** option operates on the repository that is currently available. This option will undo the adjustments and restore the Head's position prior to the most recent alterations. The available adjustments will be removed from the staging area.
3. Following the hard reset, we can see that there is nothing to commit in my repository because the reset hard option deleted all the changes to match the current Head's status with the prior one. So, the file **test_reset.txt** has been removed from the repository.

In general, the reset hard mode does the following tasks:

- The HEAD pointer will be moved.
- The staging area will be updated with the material that the HEAD is pointing to.
- The working directory will be updated to match the staging area.

Git reset mixed

The **git reset** command has a mixed option as a default. If no arguments are passed, the **--mixed** option is used by default by the **git reset** command as shown in the following screenshot:

```
➜  Git_101 git:(main) touch temp/test_reset.txt
➜  Git_101 git:(main) ✗ git status
 On branch main
 Your branch is ahead of 'origin/main' by 2 commits.
   (use "git push" to publish your local commits)

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

 nothing added to commit but untracked files present (use "git add" to track)
➜  Git_101 git:(main) ✗ git add .
➜  Git_101 git:(main) ✗ git status
 On branch main
 Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

 Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
 new file:   temp/test_reset.txt
```

Figure 8.27: Git reset –mixed in action – I

The ref points are updated with a mixed choice. In addition, the staging area was reset to the state of a certain commit. The changes that were left undone were copied to the working directory.

```
➜  Git_101 git:(main) ✗ git reset --mixed
➜  Git_101 git:(main) ✗ git status
 On branch main
 Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

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

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

***Figure 8.28**: Git reset –mixed in action – II*

The above command will reset the Head's status, but it will not erase any data from the staging area in order to match the Head's location.

We can see from the above output that we used the **`git reset -mixed`** command to reset the position of the Head. We also double-checked the repository's status. As we can see, this command has had no effect on the repository's status. As a result, it is evident that mixed mode does not remove any data from the staging area.

The reset mixed mode, in general, performs the following tasks:

- The HEAD pointer will be moved.
- The material that the HEAD is pointing to will be updated in the staging area.

Unlike Git hard mode, it does not update the working directory. It merely updates the index, not the working tree, and then creates a report of the files that haven't been updated.

Git reset soft

The soft option has no effect on the index file or working tree, but it does reset the Head, like all other choices. When the soft mode is activated, the ref points are updated, and the resets come to a halt. It will work in the same way as the **`git amend`** command. It is not a command from a higher authority. It was once seen to be a waste of time by developers.

```
➜  Git_101 git:(main) git reset --soft
```

***Figure 8.29**: Git reset –soft*

It is typically utilized to adjust the head's position.

Git reset to commit

Sometimes, we may need to reset a specific commit from time to time, and Git allows us to do so. We can go back to a previous commit. To reset it, use the **`git reset`** command with any parameter supported by reset. It will reset the provided commit by using the default behavior of a command.

The following is the syntax for resetting commit:

```
→  Git_101 git:(main) git reset <option> <commit-sha-id>
```

***Figure 8.30**: Git reset to commit*

Here the option can either be: **`--hard`**, **`--mixed`**, or **`-soft`**

Resetting versus reverting

It is vital to remember that git revert just undoes a single commit; it does not "revert" a project to its former state by removing all future commits. This is referred to as a reset rather than a revert in Git.

Compared to resetting, reverting has two significant advantages:

- It is a "safe" procedure for commits that have already been published to a shared repository because it does not modify the project history.
- Git revert can target a specific commit at any point of time in the history, but **`git reset`** can only go backward from the current commit. If you wanted to undo an old commit with **`git reset`**, for example, you must erase all commits that came after the target commit, then remove the target commit and re-commit all subsequent commits. Needless to say, this is not a really elegant undo method.

Amend Git commit

The **`git commit --amend`** command makes it easy to make changes to the most recent commit. Instead of initiating a new commit, it allows you to integrate staged modifications with the prior one. It can also be used to make minor changes to a prior commit statement without affecting the snapshot. However, amending does not simply change the most recent commit; it completely replaces it, making the altered commit a new entity with its ref.

```
➜  Git_101 git:(main) git commit --amend
```

***Figure 8.31**: Git commit --amend*

It will appear to Git as a fresh new commit. There are a few frequent instances in which you might want to use **`git commit --amend`**.

Changing most recent Git commit message

Let us imagine you just committed, and your commit log message had an error. You can edit the preceding commit's message without affecting the snapshot if you use this command when nothing is staged.

In the course of your daily development, you will make premature commitments regularly. It is all too simple to forget to stage a file or mis-format your commit message.

The **`--amend`** flag is a handy technique to correct these minor errors.

```
➜  Git_101 git:(main) git commit --amend -m "updated commit msg"
```

***Figure 8.32**: Git commit –amend –m*

Using the **`-m`** option, you can send a new message directly from the command line without having to open an editor.

Changing committed files

The scenario depicted in the following example is a common one in Git-based development.

Let us imagine we have updated a few files and want to commit them all in a single snapshot, but we fail to include one of them the first time. It is as simple as staging the other file and committing with the **`–amend`** command to fix the error

To summarise, **`git commit --amend`** allows you to add new staged modifications to the most recent commit. With a **`--amend`** the commit, you can add or remove modifications from the Git staging area.

Even if no modifications have been staged, a **`--amend`** command will prompt you to edit the last commit message log. When using **`--amend`** on commits that are shared with other team members, be cautious. Amending a commit that has been shared with another user may need perplexing and time-consuming merge conflict resolutions.

Interactive rebase

The term "interactive rebasing" is a misnomer for a very valuable Git feature. Rebase and interactive rebase have little in common from the user's perspective.

Git rebase helps in rebasing the commits from one branch to another, one by one, in order. Git offers an interesting option called **`--interactive (-i)`**, which opens an editor with a list of commits that are about to be updated. This list accepts commands, allowing the user to make changes to the list before rebasing.

You can use interactive rebase to clean your commits before pushing them to the server. You can use interactive rebasing to:

- **`Git Squash`** your commits to make the commit history more compact and easier to read. Before distributing modifications to others, squash is a good tool for group-specific alterations.
- With the intriguing interactive rebase command, you may combine many commits into a single commit. If you use Git, you are probably aware of the necessity of squashing a commit. Many times, especially if you are on an open-source contributor, you will need to submit a **pull request** (**PR**) with squashed commits. You can even squash commits if you have previously created a PR.
- Make changes to the message for your commits.
- Fixup is similar to squash, only it does not pause to ask for a new message.
- Delete to remove a commit.

Interactive rebasing at work

Let us say we want to rephrase one of the check-in comments of the past four commits.

We did **`git rebase -i HEAD4`** and got the following output:

```
➜  Git_101 git:(main) git rebase -i HEAD~4
 pick febe99d reset
 pick b1b4cca move file dir
 pick 4e1e05d add new msg in file
 pick 939ad08 Revert "add new msg in file"

 # Rebase 136f2a9..939ad08 onto 136f2a9 (4 commands)
 #
 # Commands:
 # p, pick <commit> = use commit
 # r, reword <commit> = use commit, but edit the commit message
 # e, edit <commit> = use commit, but stop for amending
 # s, squash <commit> = use commit, but meld into previous commit
 # f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
 #                    commit's log message, unless -C is used, in which case
 #                    keep only this commit's message; -c is same as -C but
 #                    opens the editor
 # x, exec <command> = run command (the rest of the line) using shell
 # b, break = stop here (continue rebase later with 'git rebase --continue')
 # d, drop <commit> = remove commit
 # l, label <label> = label current HEAD with a name
 # t, reset <label> = reset HEAD to a label
 # m, merge [-C <commit> | -c <commit>] <label> [# <online>]
 # .       create a merge commit using the original merge commit's
 # .       message (or the oneline, if no original merge commit was
 # .       specified); use -c <commit> to reword the commit message
 #
 # These lines can be re-ordered; they are executed from top to bottom.
 #
 # If you remove a line here THAT COMMIT WILL BE LOST.
 #
 # However, if you remove everything, the rebase will be aborted.
```

***Figure 8.33**: Git rebase -i HEAD~4 - I*

From oldest to newest, we can observe the four most recent commits.

Examine the comment just below the list of commits. The action is set to "pick" by default, implying that the commit will be reapplied with no changes to its content or message.

The repository will be unaffected if you save and execute this file. If I include "rephrase" (abbreviated as "r") in a commit, it signals that I want to make changes to the subsequent changes.

```
➜  Git_101 git:(main) git rebase -i HEAD~4
 pick febe99d reset
 pick b1b4cca move file dir
 r 4e1e05d add new msg in file
 pick 939ad08 Revert "add new msg in file"
```

***Figure 8.34**: Git rebase -i HEAD~4 – II*

When I save and exit the editor, git will do the procedures outlined above, bringing me back into the editor as if I had changed commit ***4e1e05d***.

```
interactive new msg in file

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Sep 11 19:36:51 2021 +0530
#
# interactive rebase in progress; onto 136f2a9
# Last commands done (3 commands done):
#    pick b1b4cca move file dir
#    reword 4e1e05d add new msg in file
# Next command to do (1 remaining command):
#    pick 939ad08 Revert "add new msg in file"
# You are currently editing a commit while rebasing branch 'main' on '136f2a9'.
#
# Changes to be committed:
#       modified:   temp/learn_git.txt
```

***Figure 8.35**: Git rebase -i HEAD~4 – III*

I make changes to the commit message from *add -> interactive*, save, and exit the editor. The following is the result:

```
[detached HEAD f7c8c40] interactive new msg in file
 Date: Sat Sep 11 19:36:51 2021 +0530
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/main.
```

***Figure 8.36**: Git rebase -i HEAD~4 – III*

The commit message for commit 4e1e05d has been changed to "**`interactive new msg in file`**."

Now that you have learned how to edit the message of any commit, you can try it on your own.

Squash commits together

Rebase interactive also provides us with the following commands:

- squash (s) merges the commit into the previous one (the one in the line before)
- Fixup (f) acts similarly to squash (s) but discards the message of this commit

We will keep working on the rebase example started earlier as shown in the following screenshot:

```
➜  Git_101 git:(main) git rebase -i HEAD~4
 pick febe99d reset
 pick b1b4cca move file dir
 s 4e1e05d interactive new msg in file
 f 939ad08 Revert "add new msg in file"
```

***Figure 8.37**: Git rebase -i HEAD~4 – I*

The editor would have added the third commit message, which had already been commented out for us when it was saved, as follows:

```
# This is a combination of 3 commits.
# This is the 1st commit message:

move file dir

# This is the commit message #2:

interactive new msg in file

# The commit message #3 will be skipped:

# Revert "add new msg in file"
#
# This reverts commit 4e1e05d26409e816fcab722694a503c4c3bb3ec3.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Mon Aug 30 12:11:13 2021 +0530
#
# interactive rebase in progress; onto 136f2a9
# Last commands done (4 commands done):
#    squash f7c8c40 interactive new msg in file
#    fixup ee9651f Revert "add new msg in file"
# No commands remaining.
# You are currently rebasing branch 'main' on '136f2a9'.
#
# Changes to be committed:
#       renamed:    learn_git.txt -> temp/learn_git.txt
```

***Figure 8.38**: Git rebase -i HEAD~4 – II*

Save, and outputs:

```
[detached HEAD 652ec3f] move file dir
 Date: Mon Aug 30 12:11:13 2021 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename learn_git.txt => temp/learn_git.txt (100%)
Successfully rebased and updated refs/heads/main.
```

***Figure 8.39**: Git rebase -i HEAD~4 – III*

Rebase on top of main

We fork an open-source library, begin work on a feature branch, and the upstream project's main branch progresses.

This is what our history looks like:

```
      A---B---C feature
     /
D---E---F---G upstream/main
```

***Figure 8.40**: Git rebase main - I*

The library maintainer requests that we "rebase on top of main" so that any merge conflicts between the two branches may be resolved and our changeset remains intact. The maintainer would want to see a history that looks somewhat like this:

```
             A'--B'--C' feature
            /
D---E---F---G upstream/main
```

***Figure 8.41**: Git rebase main – II*

We wish to reapply our commits to upstream's main one by one, in order. Sounds like the rebase command's description!

Let us look at the commands that will get us to the desired scenario:

```
# Point our `upstream` remote to the original fork
git remote add upstream https://github.com/justjais/Git_101.git

# Fetch latest commits from `upstream` (the original fork)
git fetch upstream

# Checkout our feature branch
git checkout feature

# Reapply it onto upstream's master
git rebase upstream/master

# Fix conflicts, then `git rebase --continue`, repeat until done
# Push to our fork
git push --force origin feature
```

***Figure 8.42**: Git rebase main – III*

Re-writing history risks

Did you notice the **`--force`** option in the previous **`git push`** command? This signifies that the repository's history is being overwritten. This is always safe to do in commits that we do not share with other team members, or on our branches (see my initials in the example of this blog post).

However, if you forcibly push editions that have already been shared with the team, that is, commits that exist outside of my repository, Git will signal that something is wrong. Users will receive unexpected error messages and may accidentally roll back past commits while attempting to address the associated merge conflicts.

This issue generates a real message, and if you're concerned, you can always try it on a temporary copy of your repositories.

Rewriting history entails deleting old commits and replacing them with new ones that are similar but not identical. Your team members will have to re-merge their work if others base their work on your earlier commits, and then you rewrite and force-push your commits (if they notice the potential loss).

Conclusion

This is the end of this chapter and the contents we discussed is the building block and soul of the Git version control system. The more fluent you get will all the discussed terms and their respective usages and application, the more efficient you will become in using Git and GitHub.

In the next chapter, we will continue to build on the knowledge we gained in this chapter and discuss few Git concepts in more depth.

Multiple choice questions

1. **How to amend to git commit?**
   a. git commit -m
   b. git commit -am
   c. git commit --amend
   d. git add .

2. **How to squash files while doing interactive rebase?**
   a. git squash
   b. git add .
   c. git rebase
   d. git rebase -i HEAD~3 -> s

3. **Git reset changes that effects only staging snapshot and commit history?**
   a. git revert
   b. git reset --hard
   c. git reset --mixed
   d. git reset --soft

4. **Git command that is extremely useful for recovering lost branches and commits?**
   a. git log
   b. git reflog
   c. git add .
   d. git commit --amend

5. **How to check condensed log via git log command?**
   a. git log
   b. git log --oneline
   c. git log --stat
   d. git log --log-size

6. **How to remove the file from git index without actually removing it from the local file system?**
   a. git rm
   b. git stash
   c. git reset
   d. git commit

Answers

1. **c**
2. **d**
3. **b**
4. **b**
5. **b**
6. **c**

Points to remember

- To avoid making the same error again, instead of using the **`git rm`** command, we can use the **`git reset`** command to remove the file from the staged version and then add it to the **`.gitignore`** file.
- To revert a bad commit that is already pushed, a new commit can be created that reverts changes done in the bad commit. It can be done using **`git revert <name of bad commit>`**.
- It is usually preferable to create a new commit rather than amend an existing one:
    - It is acceptable if only the commit message is changed or destroyed, but there may be instances where the contents of the commits are changed. As a result, vital information linked with the commit is lost.
    - Excessive use of **`git commit --amend`** can have serious consequences, since the modest commit amend might grow and accumulate unrelated modifications over time.
- The command **`git reset --mixed`** is used to undo changes to the working directory and **`git index`**.

Further readings

For more history and reference around the discussed topics in this chapter you can check out the Git official documentation for getting started and GitHub documentation

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 9
Most Commonly Used Git Commands

Git is widely used in the software industry. It is a crucial component of daily development and is more prominent if you work in a team.

Learning how to utilize all the many commands in Git takes some time and getting used to. Some commands, however, are employed more frequently (some daily).

Therefore, we will go over the most popular and commonly used Git commands in this chapter that every developer should be familiar with.

Structure

In this chapter, we will cover the following most used Git commands:

- Git config
- Git init
- Git clone
- Git status
- Git add
- Git commit
- Git push

- Git branch
- Git checkout
- Git merge
- Git pull
- Git log
- Git show
- Git diff
- Git tag
- Git rm
- Git stash
- Git reset
- Git revert
- Git remote
- Git fetch

Objectives

This chapter essentially serves as a summary of the most frequently used Git commands that we have studied together. All users, whether novice, experienced, or expert, can refer to this chapter whenever they have questions about how to use Git commands because it tries to address most common use case scenarios.

This chapter assumes you are up and running Git under your Linux, Windows or Mac machines.

Git config

Git needs to be configured before you can use it. You can enter the login and email address that will be associated with your commits using this command.

This command specifies the author's name and email address to be used with your commits.

```
# sets up Git with users' name
→ Git_101 git:(main) git config --global user.name "<Users' Full-Name>"
# sets up Git with users' email id
→ Git_101 git:(main) git config --global user.email "<Users' Email ID>"
```

***Figure 9.1**: git config command*

Git init

You cannot do anything with a Git repository until you create one in order to make commits or do anything else with it. A new Git repository will be created using the **`git init`** command. The **`init`** subcommand, is useful because it performs all the initial setup for a repository. It stands for ***"initialise".***

We will look at what it does shortly. Git keeps track of everything by creating the required directories and files with the **`git init`** command.

Note the **"."** at the beginning of the directory name; this denotes that it will be hidden on Mac/Linux systems. All of these files are saved in this directory.

In order to keep its files structured in other subdirectories, Git will create a hidden. git directory and use it.

```
# initialise a Git repo
→ ~ git init
```

***Figure 9.2**: git init command*

Git clone

Git Clone helps clone an existing repository into a new directory. It is the command we will be running on the terminal. The Git repository you want to clone is specified via a path (usually a URL) in the **`git clone`** command. Git clone is the command, and the argument is the location of the Git repository you wish to clone.

The website where we will be working on this project is located on the GitHub repository. Git clone creates a local working copy of the source code from a remote repository. The code is automatically downloaded to your local system when you clone a repository. This will transfer project files from a remote repository to our local system.

The original location will be added as a remote location if you have permission to do so, enabling you to pull changes from and submit changes to it.

To get a repository from an existing URL, use this command:

```
# Clone a remote Git repo
→ ~ git clone https://github.com/<git repo-url>
```

***Figure 9.3**: git init command*

Git status

Our key to understanding Git is the **`git status`** command. It will let us know what Git is thinking and how Git perceives the current state of our repository.

Always use the **`git status`** command when you are first starting. Starting it after any other command is a good idea. This will help you learn Git and prevent you from assuming (perhaps incorrectly) about the status of your files or repository.

The information that the **`git status`** tool displays will vary depending on the condition of your files, working directory, and repository as shown in the following screenshot:

```
# Get the Git status of repo
➜  Git_101 git:(main) git status
```

***Figure 9.4**: git status command*

Git add

Use the **`git add`** command to move files from the working directory to the staging index. You can contrast your local version with the version of the remote repository by using the **`git add`** command, which stores your modifications in a file in the staging area.

Use the **`git add`** command to add your new or modified file to the staging area before committing it.

To add particular files, using the following command, files are added to the staging area:

```
# Add particular files to staging area
➜  Git_101 git:(main) git add <file1> <file2>...<fileN>
```

***Figure 9.5**: git add files – I*

To add all the files, use the following command which adds one or more files to the staging area in order:

```
# Add all changed files to staging area
➜  Git_101 git:(main) git add .
```

***Figure 9.6**: git add files - II*

Git commit

The most often used Git command can be this one. It is used when we have finished developing anything and want to save our changes (maybe after a specific task or issue).

Git commit is comparable to creating a development checkpoint that you can return to at a later time. This command stores the commit id of the changes made to the Git repository along with a log message.

With **`git commit`**, the changes are recorded in your repository. Every time you commit your code changes, you must also add a brief explanation of the modifications that were done. This commit statement makes the modifications easier for others to understand.

```
# commit the changes to remote repo
➜  Git_101 git:(main) git commit -m "commit message for the changes"
```

***Figure 9.7**: git commit changes*

Git push

The contents of your local repository are pushed to the remote repository you added with this command. Pushing commit will send the commits from your main branch to the remote repository.

A named branch will be created in the remote repository if it does not already exist as shown in the following screenshot:

```
# git push changes
➜  Git_101 git:(main) git push <remote> <branch-name>
# Newly created branch, need to be uploaded as
➜  Git_101 git:(main) git push -u origin <branch-name>
➜  Git_101 git:(main) git push --set-upstream <remote> <branch-name>
```

***Figure 9.8**: git push changes*

Git branch

With the **`git branch`**, you can add a new branch to an existing one, display all existing branches, and remove a branch. This command is used to carry out actions on the selected branch. Git will withdraw all the files and directories from the commit that the branch refers to, when you execute this command. Git will also delete all files and folders from the Working Directory it is tracking (files that Git tracks are preserved in the repository, so nothing is lost).

1. Create a new local branch by using the following command:

```
# git branch
➜  Git_101 git:(main) git branch <branch-name>
```

***Figure 9.9**: git branch*

2. Use the following command to push the new branch into the remote repository:

```
# git push branch
➜  Git_101 git:(main) git push -u <remote> <branch-name>
```

***Figure 9.10**: git push remote branch*

3. To view branches:

```
# list branch
➜  Git_101 git:(main) git branch/ git branch --list
```

***Figure 9.11**: List git branch*

4. To delete the branches:

```
# Delete branch
➜  Git_101 git:(main) git branch -d <branch-name>
```

***Figure 9.12**: Delete git branch*

Git checkout

The **`git checkout`** command can be used to switch to an already-existing branch or create a new branch and switch to it. To do this, you need to have the branch you wish to switch to in your local system and need to commit or stash any changes you have made to your current branch.

You can check out files by using **`git checkout`** command. Git will withdraw all the files and directories from the commit that the branch refers to from the repository when you execute this command. Git will also delete all files and folders from the Working Directory it is tracking (files that Git tracks are preserved in the repository, so nothing is lost).

```
# Checkout branch
➜  Git_101 git:(main) git checkout <branch-name>
```

***Figure 9.13**: Checkout git branch*

You must take care of the following issues to effectively switch between branches:

- Before switching, the modifications in your current branch must be committed or stored.
- The branch you want to visit should be available in your local repository.

Git merge

This command merges the history of the selected branch into the current branch. Your branch is joined to the parent branch using the **`git merge`** command.

The parent branch may be a development branch or a main branch, depending on your workflow. It will automatically create a new commit if there are no conflicts.

Before using the **`git merge`** command, you must be on the branch you want to merge with your parent branch. This command merges the history of the selected branch into the current branch as shown in the following screenshot:

```
# merge branch
➜  Git_101 git:(main) git merge <branch-name>
```

***Figure 9.14**: Merge git branch*

Before merging your branches, make sure your development branch has the most recent version; otherwise, conflicts or other undesirable issues may arise.

Git pull

The **`git pull`** command fetches the contents of the remote repository and integrates them into your local repository.

Git pull makes sure you have the most recent information from your colleagues by bringing the most recent changes from the remote server into the local repository.

```
# merge remote changes to local
➜  Git_101 git:(main) git pull
```

***Figure 9.15**: Git pull*

Git log

To view every commit made to a repository, use the **`git log`** command. This command shows a history of each commit made to the active branch.

```
# check git logs
→  Git_101 git:(main) git log
```

***Figure 9.16**: Git log*

Git show

This command displays the modifications to the commit's metadata and content:

```
# show git metadata based on id
→  Git_101 git:(main) git show <commit-id>
```

***Figure 9.17**: Git show commit metadata*

Git diff

`Git diff` displays the differences between your current working directory and your staging directory.

Commits, branches, files, and other types of data sources are examples where Git diff comes handy. When examining the current state of our Git repository, the **`git diff`** command is frequently used in conjunction with the **`git status`** and **`git log`** commands as shown in the following screenshot:

```
# show diff based on commit-id
→  Git_101 git:(main) git diff <commit-id>
```

***Figure 9.18**: Git diff*

To obtain the specifics of commit IDs, we use **`git log`** as shown in the following screenshot:

Git tag

`Git tag` command is used to add tags to the provided commit referenced by an ID:

```
# tag a commit-id
→  Git_101 git:(main) git tag <commit-id>
```

***Figure 9.19**: Git tag*

Git rm

This command stages the deletion while removing the file from your working directory as shown in the following screenshot:

```
# To remove a file
→  Git_101 git:(main) git rm <file-name>
```

***Figure 9.20**: Git rm*

Git stash

Git stash is used to store (changes) safely in a hidden place (the stash stack).

- All the changes in the tracked files are temporarily stored by this command:

```
→  Git_101 git:(main) git stash save
```

***Figure 9.21**: Git stash – I*

- This command restores the most recently saved files:

```
→  Git_101 git:(main) git stash pop
```

***Figure 9.22**: Git stash – II*

- This command lists each change set that has been stored:

```
→  Git_101 git:(main) git stash list
```

***Figure 9.23**: Git stash – III*

- The most recent stored files are deleted by this command:

```
→  Git_101 git:(main) git stash drop
```

***Figure 9.24**: Git stash – IV*

Git reset

`git reset` is used to restore the working tree to its most recent committed state.

- This command unstages the file while maintaining its contents:

```
→  Git_101 git:(main) git reset <file-name>
```

***Figure 9.25**: Git reset – I*

- This command restores the local changes and undoes all commits that came before the one that is provided:

```
→  Git_101 git:(main) git reset <commit-id>
```

***Figure 9.26**: Git reset – II*

- The following command returns to the given commit while erasing all previous history:

```
→  Git_101 git:(main) git reset --hard <commit-id>
```

***Figure 9.27**: Git reset – III*

Git revert

We occasionally need to reverse the adjustments we have made. Depending on what we require, there are several ways to reverse our modifications locally or remotely, but we must use these commands carefully to prevent unintended deletions.

Then, all we have to do is write the hash code next to the commit we want to undo:

```
→  Git_101 git:(main) git revert <commit-id>
```

***Figure 9.28**: Git revert*

Utilizing **`git revert`** has the benefit of not affecting the commit history. This indicates that every commit, including those that have been reverted, is still visible in your history.

Everything, in this case, happens locally unless we push them to the remote repository, which is another safety step.

Git revert is the best method for undoing our commits because it is safer to use.

Git remote

To connect your local repository to the remote server, use this command:

```
→  Git_101 git:(main) git remote add <origin/upstream> <remote repository URL>
```

***Figure 9.29**: Git remote*

Git fetch

Git gathers any commit from the target branch that is not already in our current branch when we run the **`git fetch`** command and stores it in our local repository.

But it does not integrate it into our current branch as shown in the following screenshot:

```
→  Git_101 git:(main) git fetch -all
```

***Figure 9.30**: Git fetch*

This is very helpful when working on something that would fail if we updated our files but still need to maintain our updated repository as current. We utilize merge to include the commits into our master branch.

It pulls every branch from the repository. Additionally, it gets all the necessary files and commits from another repository.

Conclusion

This is the end of the chapter. We discussed the most frequent Git commands that developers and testers use in their daily work.

Given that this is the book's final chapter, I want to emphasize the fact that learning GitHub inside and out takes time. You must first become proficient in a variety of specialized GitHub features. One component of this knowledge is knowing how to automate the creation of a pull request or the linking of issues to a project board. To contribute to projects efficiently and meaningfully, it is also crucial to start mastering facets of software engineering as a whole.

A further strategy to master GitHub is to become a productive community participant.

Simply trying is one of the best methods to learn anything. Trying always results in learning.

When you try something and fail, you learn from your mistakes and gain an understanding of how it operates.

Success teaches you what to do the next time!

Multiple choice questions

1. **How to check the remote config in your local repository?**
    a. git remote
    b. git remote -v
    c. git revert
    d. git status

2. **How to add multiple files to git staged area?**
    a. git commit -m
    b. git push --force-with-lease
    c. git branch –D <branch name>
    d. git add .

3. **Command to get the commit from the target branch that isn't already in our current branch?**
    a. git fetch
    b. git commit
    c. git stash
    d. git status

4. **Git command to configure user credentials to use the git as expected?**
    a. git commit
    b. git config
    c. git add .
    d. git commit --amend

5. **Command to undo a bad commit that was pushed by mistake?**
    a. git add .
    b. git revert <commit name>

c. git rm <filename>
d. git commit

6. **Git command to remove the file from git index without actually removing from the local file system?**
    a. git rm
    b. git stash
    c. git reset
    d. git commit

Answers

1. **b**
2. **d**
3. **a**
4. **b**
5. **b**
6. **c**

Key terms

- Start a working area
    - init
    - Clone
- Work on the current change:
    - add
    - mv
    - restore
    - rm
- sparse-checkoutExamine the history and state of the repository:
    - bisect
    - diff
    - grep
    - log

    - show
    - status

- Grow, mark and tweak your repo history:
    - branch
    - commit
    - merge
    - rebase
    - reset
    - switch
    - tag

- Collaborate over repository:
    - fetch
    - pull
    - push

Further reading

For more history and reference around the discussed topics in this chapter you can check out the Git official documentation for getting started and GitHub documentation

- Getting started with Git: **https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control**
- Getting started with GitHub: **https://docs.github.com/en/get-started/quickstart**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index