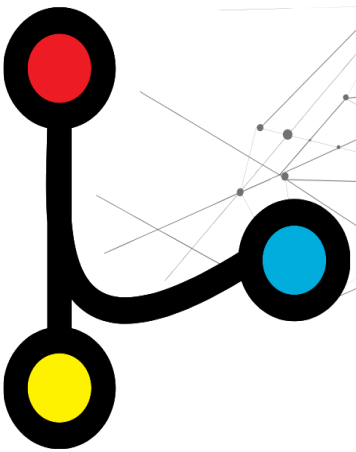
GIT PRODIGY

Learn Git and GitHub For code management

And open-source contributions!



EBENEZER DON

Git Prodigy

Mastering Version Control with Git and GitHub

Ebenezer Don

This book is for sale at http://leanpub.com/git-prodigy

This version was published on 2023-08-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

This book exists for one reason: to **help you master Git**, **GitHub**, **and the open-source landscape**, regardless of your prior knowledge or experience.

When I first encountered Git, I was, frankly, scared of it. The concept of **Version Control** and the commands associated with Git seemed intimidating. The term "**Open Source**" was a mystery, and I couldn't understand how a group of strangers around the globe could use GitHub to efficiently collaborate on a project. However, my perspective changed dramatically when I finally delved in and learned how to use Git. I came to the startling realization of how many thousands of hours and years of lost work I could have saved if I had embraced it earlier. Git became a tool that **empowered me to work confidently**, giving me the freedom to collaborate, experiment, and even break things, all while knowing I had a reliable safety net.

"Git Prodigy" was crafted as a clear, accessible, and practical guide designed to unravel these powerful tools and the vibrant world of open-source software.

In the pages of this book, we will take a journey. It's a journey that begins in the roots of version control,

unfurling the basic concepts of Git. You'll learn how to initiate a new repository, stage and commit changes, manage branches, and merge code—always with an eye on **real-world application**, with plenty of examples to make the concepts come alive.

We'll then explore **GitHub**, a platform that has emerged as a cornerstone of collaborative software development. You'll learn how to work with remote repositories, use pull requests, manage code reviews, and fork projects. But we won't stop at the mechanics: you'll also learn how to work effectively as part of a team, leveraging GitHub's features to facilitate productive collaboration.

Beyond Git and GitHub, *"Git Prodigy"* shines a spotlight on the thriving **open-source community**. This ecosystem is as diverse and colourful as it is influential, shaping much of today's software landscape. Through this book, you'll gain a deep understanding of the customs, rules, and culture of open-source software, with insights into licensing and best practices for making contributions. This knowledge will **empower you to participate confidently, responsibly, and meaningfully**.

As we reach the final chapters, you'll learn **advanced Git features** and real-world troubleshooting techniques. From rebasing to cherry-picking, from handling merge conflicts to recovering lost commits, you'll acquire skills that turn problems into opportunities for learning and

growth.

In this journey, you'll learn more than just commands and features. You'll understand the philosophy behind Git, GitHub, and open source—the **why**, not just the **how**. Whether you're a beginner hoping to grasp the basics, an experienced developer aiming to hone your skills, or an open-source enthusiast seeking to contribute more effectively, you'll find *"Git Prodigy,"* a valuable companion.

Let's get started.

1. Introduction to Version Control

This chapter sets the foundation for understanding the importance and role of version control in software development. We'll delve into Git, one of the most widely used version control systems, and explain the key differences between Git and GitHub. We'll end the chapter with a guide on how to install Git.

What is Version Control?

Imagine you're working on an important document, like an essay or a report, and it's been weeks since you started. You keep restructuring your document as you write, adding and removing paragraphs, and saving your changes.

Then you realize that you've made a mistake. You've deleted a paragraph that you wanted to keep. You try to undo your changes, but you've saved over the same file multiple times, and you can't go back to the previous version. You're stuck!

Now, if you have ever worked on a project and saved different versions at different points with names like "first-draft", "second-draft", "final-draft" or "final-final-draft-for-real", to avoid this problem, instead of overwriting only one saved file, you've actually done some kind of version control. You did this so that you can always go back to a previous version in case you don't like your new changes.

Version Control, in programming, is just a more sophisticated and streamlined version of this concept. It's like a time machine for your project. It keeps snapshots of your work at different points in time, and you can travel back and forth through these snapshots whenever you

want. And the best part? It's all in one place; you don't need multiple versions of the same file cluttering your workspace.

There are two types of version control systems: **Centralized** and **Distributed**.

Centralized Version Control (CVC)

Centralized Version Control is like having one master document on a central server. Users check out parts of the document to their local workspace, make modifications, and then push those changes back to the master document. There's a single place where you can find the history of the document and see all the changes. But, if something happens to that central place, all the history might be lost.

Distributed Version Control (DVC)

Distributed Version Control is different. It's like everyone having a complete history of the document. They can make changes independently and later merge their changes with others. Even if the central place is lost, the document's history is safe because everyone has a copy. This is the type of version control we'll be learning in this

book, and it's the most popular type of version control used today.

Git is a Distributed Version Control System that allows you to keep a history of your project, collaborate with others easily, and make changes without affecting the main project until you are ready.

In the following sections, we'll dive into the fascinating world of Git and learn how to harness its power for managing your projects efficiently and collaboratively.

Why is Version Control Important?

Version control is essential to modern software development for a multitude of reasons. It's like an insurance policy for your project, offering a safety net for potential mistakes and facilitating smooth team collaboration. Let's dive into some specifics of why version control is so indispensable.

Reverting and Rectifying Mistakes

Everyone makes mistakes, and programmers are no exception. Sometimes you'll make changes that either create new bugs or just don't pan out the way you hoped. With version control, there's no need to panic.

Imagine you roll out a new feature for your web app and immediately receive messages about a critical bug. Version control allows you to swiftly revert to a stable version of the code, effectively undoing the changes, so you can address the issue.

Experimentation Without Fear

Being innovative and experimenting with new ideas is at the heart of software development. However, experimenting in your main codebase can be risky. Version control systems allow you to mitigate this risk.

For example, consider a scenario where you are developing an app and want to add a new feature that you're not sure will be enjoyable. Version control allows you to create a separate copy of your code, known as a branch, for experimentation. If your new feature proves to be successful, you can merge it back into the main code. If it's unsatisfactory, you can simply discard the branch without affecting the main codebase.

Simplifying Collaboration

When working on a software project with a team, things can get messy if you don't have a good system in place. Imagine multiple people making changes to the same files - it's a recipe for disaster without version control.

Let's say you and your colleague are both working on different features of a mobile app. Version control enables you both to work on the same codebase without interfering with each other's work. When you're both done,

you can combine your changes into the main project, and if there are any conflicts, the version control system will help you resolve them.

In essence, version control is not just a good-to-have, it's a **must-have for any software development process**. It's like a safety net, a time machine, and a team coordinator all rolled into one. In the next section, we'll start our journey into one of the most popular version control systems - Git.

Introduction to Git

Now that we've established how critical version control is for software development let's delve into Git. Git is a Distributed Version Control System (DVCS) that has become an industry standard for both small and large-scale projects. In this section, we will lay the groundwork for understanding what Git is, how it differs from other version control systems, and why it has become so popular among developers.

What is Git?

Git is a distributed version control system designed to manage the source code history of software development projects. It was created by **Linus Torvalds** in **2005** to manage the development of the Linux kernel. Since then, it has become the most widely used version control system in the world.

The term "distributed" in its architecture means that each user's working copy of the codebase includes the full history of changes and commits. Unlike Centralized Version Control Systems (CVCSs) where there's one central repository, in Git, every developer's working copy is a repository.

A **repository**, often shortened to "repo," is a storage location where all the files of a project and their history of changes are stored. A repository can be local, which means it's stored on your computer, or remote, which means it's stored on a server.

Git's distributed architecture makes it possible to work offline on your local repository, and then push your changes to a remote repository when you're ready. This **remote repository** is typically a version of your project that is hosted on the Internet or network somewhere. This makes collaboration and version control efficient, even with large codebases and teams.

How Does Git Work?

To track changes in your project, Git uses a feature known as a **commit**. Commits in a Git repository are like snapshots of your project at a particular point in time. With commits, you can revert to previous versions of your project, compare changes over time, and identify when and where changes were made.

These snapshots are stored within a directory at the root of your project called `.git`, which is created when a new Git repository is initialized. This directory is hidden by default on most systems and contains all the information

necessary for Git to manage your project's history. To see the contents of the `.git` directory, you might need to enable the option to show hidden files in your file explorer.

Whenever you make a commit, Git saves a snapshot of what all your files look like at that moment. This is different from some other VCSs, which store data as a list of file-based changes.

Why Use Git?

Here are a few reasons you might want to use Git for your projects:

- **Speed**: Git is fast. Since you have the entire repository and history on your local machine, most operations are almost instantaneous.

- **Collaboration**: Git's distributed nature makes it easy for teams to work together. Different members can work on different features in parallel.

- **Data Integrity**: Git uses a cryptographic hash function called "SHA-1" to generate unique identifiers for each commit. This helps ensure data integrity and tracks changes, contributing to the protection of your

files, directory structure, and history from unnoticed modifications or corruption.

- **Flexibility**: Whether you're a solo developer working on a small project or part of a multinational team on an enterprise application, Git can scale to your needs.

- **Open Source**: Git is free and open source. It has a large, active community of contributors, which means it's continually being improved and updated.

- **Integration**: Git integrates well with various development tools and platforms, making it versatile in different workflows.

In Summary:

Git is not just an ordinary tool—it's an entire ecosystem that stands on its own. The combination of its decentralized structure, exceptional efficiency, and flexibility makes it the favored version control system among many developers. As we progress through this book, you'll obtain hands-on knowledge and understand how to harness Git's potential to manage your code effectively.

Git vs GitHub: What's the Difference?

One of the common questions everyone new to version control has is the distinction between Git and GitHub. Although they sound similar and are closely related, they serve different purposes. Let's demystify these two and understand how they complement each other.

Understanding Git

As we discussed in the previous section, Git is a version control system. It is a tool that manages the source code history and allows multiple developers to work on a single project without stepping on each other's toes. Git is software that you install locally on your computer, and it works primarily from the command line (though there are also graphical user interfaces for Git).

Key Points about Git:

- It is a Distributed Version Control System (DVCS).
- It allows you to track changes, create branches, and collaborate in your local code repository.
- Git does not require an internet connection for most operations.

- It is free and open source.

Understanding GitHub

GitHub, on the other hand, is a web-based platform that uses Git for version control. Think of it as a social networking site for coders and their code. It allows you to upload your Git repositories online, making it easier to collaborate with others. GitHub provides additional features such as Pull Requests, Issues, and Wikis for your repositories.

Key Points about GitHub:

- It is a hosting service for Git repositories.
- It facilitates collaboration by making it easy to share repositories and work with others.
- GitHub offers additional features such as Pull Requests, code reviews, and Issue tracking. We'll discuss these in detail in later chapters.
- GitHub can be integrated with third-party tools and services, such as continuous integration systems, code quality measurement tools, and project management applications.

- GitHub provides several security features such as security vulnerability alerts, automated security fixes, and token scanning to protect repositories and user data.

How They Complement Each Other

Git is mainly a local tool; you use it to track changes on your computer. It is powerful on its own, but its capabilities are magnified when you bring GitHub into the mix. With GitHub, you can share your code with the world, collaborate with other developers, and even integrate third-party apps and services.

When to Use Git and When to Use GitHub

- Use Git when you are working on a project (either alone or with a team) and need to keep track of the changes in your code.

- Use GitHub when you want to share your project with others, collaborate more efficiently, or when you need a remote backup of your repository.

In Summary:

Git is the tool, and GitHub is the service that hosts your repositories and enhances collaboration. They are distinct but complementary. While you can use Git without using GitHub, utilizing them together creates a more powerful and efficient workflow, especially for collaboration on larger projects or open-source contributions. Next, we'll look at other version control systems and how they compare to Git.

Other Version Control Systems

While Git is undoubtedly one of the most popular version control systems, it is by no means the only one. There are several other VCSs, each with their unique features and strengths. It's beneficial to know that alternatives exist, as different projects may have different needs. Let's take a brief look at some other prominent VCSs.

Subversion (SVN)

Subversion, also known as SVN, is a centralized version control system initially developed by CollabNet Inc in 2000, and now maintained by the Apache Software Foundation. Unlike Git, which is a distributed system, SVN follows a more linear approach to version control.

While SVN does support branching and merging, the mechanism is quite different and can be more cumbersome compared to Git. SVN requires a connection to the central repository for commits, but it can track changes offline to some extent. Some teams find SVN straightforward and suitable when transitioning from a no-VCS setup, but it lacks some of the robust features offered by distributed systems like Git.

Mercurial

Mercurial is a distributed version control system like Git, but it was designed with simplicity in mind. Its commands and workflow are more straightforward, making it easier to learn, especially for those new to version control.

While Mercurial supports most of the basic features of Git, such as local commits and easy branching, its simplicity does come at the cost of fewer features by default. However, it does have a robust plugin system that allows you to extend its functionality.

Perforce

Perforce, or **Helix Core**, is a version control system that is widely used in the gaming industry. It excels in handling large binary files, making it an excellent choice for projects like game development, where you're dealing with 3D models, textures, and other large assets.

However, Perforce operates as a centralized version control system and is proprietary software. So depending on your project's needs or team size, you might need to purchase a license.

Unlike Git, Perforce lacks a distributed architecture and does not support local repositories in the same way that

Git does.

Concurrent Versions System (CVS)

Concurrent Versions System, first released in 1986, is an older centralized version control system, once widely used in the open-source community. CVS allows multiple developers to work simultaneously on a project, a notable advancement when it was introduced.

However, CVS is less efficient with large projects or binary files and has a more error-prone branching and merging system compared to newer VCS like Git or SVN. Development on CVS has slowed significantly, and it lacks many features found in contemporary systems. Today, CVS is mostly used in projects that adopted it before modern systems were available and have not yet migrated to a newer VCS.

In Summary:

Different version control systems have different strengths and are suited to different types of projects. While Git's robustness, flexibility, and widespread adoption make it the tool of choice for most software development projects, other systems may be more suited to specific use cases.

The key is to understand your project's needs and the capabilities of each VCS. Regardless of the specific system

you use, the important thing is that you use version control. It's an essential tool for modern software development. In the following chapters, we will focus on mastering Git, but many of the principles and practices you'll learn will apply to other version control systems as well.

Installing Git

Now that we've learned what version control is and have a basic understanding of Git, it's time to start using Git practically. The first step is to install Git on your computer. This section will guide you through the process of installing Git on Windows, macOS, and Linux.

Installing Git on Windows

1. Go to the official Git website at git-scm.com[1].
2. Click on the "Download" button for Windows.
3. Once the installer is downloaded, run the .exe file.
4. During the installation, you can leave the default options selected or customize them according to your preferences.
5. Click "Install" to begin the installation process.
6. Once the installation is complete, you may need to restart your computer or your command promp-

[1] https://git-scm.com

t/PowerShell terminal to ensure Git is accessible from any command prompt.

7. You can open the Git Bash application, which is a command-line interface for using Git on Windows.

Installing Git on macOS

1. Git might be preinstalled on macOS. You can check this by typing `git --version` in the terminal. If it's already installed, you'll see the Git version displayed.

2. If you don't have Git or want to install a newer version, the easiest way to install Git is using Homebrew. If you don't have Homebrew, you can install it by visiting brew.sh[2] and following the instructions.

3. After installing Homebrew, you can install Git by typing `brew install git` in the terminal, and pressing Enter.

4. To check if Git was installed successfully, type `git --version` in the terminal. It should display the installed Git version.

[2] https://brew.sh

Installing Git on Linux

1. On a Linux system, you can use the package manager that comes with your distribution to install Git.

2. For Debian/Ubuntu-based systems, open a terminal and type `sudo apt-get update` followed by `sudo apt-get install git`.

3. For Red Hat/Fedora-based systems, use `sudo dnf install git`.

4. Please note, other distributions might require different commands or procedures.

5. To confirm that Git is installed, type `git --version` in the terminal. This will show you the version of Git that's installed.

Configuring Git

After installing Git, it's a good idea to configure it with your information.

1. Open your terminal or Git Bash (on Windows).

2. Type `git config --global user.name "Your Name"` and press Enter.

3. Type `git config --global user.email "youremail@example.com"` and press Enter.

This sets up your name and email as the author of your commits. It's important because every commit you make will include this information.

Wrapping Up

Congrats! You have successfully installed and configured Git on your system. You're now ready to start creating repositories, making commits, and delving deeper into version control. In the next chapter, we'll take a closer look at creating your first repository and making your first commit. Let's dive right in!

2. Getting Started with Git - Command Line and GUI

In this chapter, we'll discuss how to interact with Git, both through the command line Interface (CLI) and Graphical User Interface (GUI) tools, with a focus on understanding when to use each. We'll walk through some basic Git commands and use Visual Studio Code to perform them in a GUI. This chapter concludes by guiding you through creating your first Git repository and making your first commit.

- 2.5 Viewing the Commit History and Reverting To A Previous Commit

Basic Git Commands

Before we dive into creating repositories and making commits, it's essential to familiarize yourself with some basic Git commands. These commands form the bedrock of your interactions with Git and will be used frequently throughout your Git journey.

Let's start by looking at some of the most common Git commands you'll use. Remember, practice makes perfect. So, as you read, try to follow along by typing the commands on your own computer.

`git init`

The `git init` command is used to create a new Git repository. When you run this command in a directory, Git initializes a new repository in it. This means that Git starts keeping track of your files and changes in that directory. You can run this command in any directory to create a new repository there:

```
$ git init
```

The dollar sign ($) is used to indicate the command prompt, so you don't need to type it too.

git clone

The `git clone` command allows you to create a copy of an existing Git repository in your local machine. This is especially useful when you want to work on a project that is hosted on a remote repository, like GitHub.

```
$ git clone <repository-url>
```

git add

The git add command is used to stage changes for a commit. It tells Git that you want to include the updates to a particular file(s) in the next commit. This includes new files, modifications to existing files, and the deletion of files. Here, the term "staging" refers to the process of preparing files for a commit. The "staging area" is where Git keeps track of the changes that are to be included in the next commit.

```
$ git add <file-name>
```

To add all new and modified files in the current directory and its subdirectories to the staging area, use the . (dot) as the file name:

```
$ git add .
```

The `git add` command also has a `-A` flag that allows you to stage all changes, including all new, modified, and deleted files in the entire repository, not just the current directory and its subdirectories:

```
$ git add -A
```

git commit

The `git commit` command is used to save your changes to the local repository. This step takes the files as they are in the staging area and stores a snapshot of these files as a commit. A **commit** in Git is like a checkpoint in your project history that you can return to at any time. Each commit has a unique ID, often referred to as a "hash", which you can use to access it whenever you need to.

```
$ git commit -m "Commit message"
```

The `-m` flag is used to add a commit message. A commit message is a short description of the changes you made in the commit. It's a good practice to write a descriptive commit message that explains your changes accurately. In

the terminal, a flag is a way to specify the behavior of a command. Flags are usually preceded by a hyphen (`-`) or two hyphens (`--`).

The `git commit` command also has an `-a` flag that allows you to automatically stage files that have been modified and deleted, but new files that have not been tracked are not affected:

```
$ git commit -a -m "Commit message"
```

If there are new files that you've never added to the repository before, you will need to add them using `git add <file>` or `git add .` before they can be included in a commit.

If you need to write a longer commit message, you can omit the `-m` flag and just run `git commit`. This will open a text editor where you can write your commit message. When you're done, save the file and close the editor to complete the commit. We'll discuss best practices for writing commit messages in a later chapter.

`git status`

The `git status` command displays the state of the working directory and the staging area. It lets you see

which changes have been staged, which haven't, and which files aren't being tracked by Git. This command gives you a summary of all the changes that have been made since the last commit, and which of these changes are ready to be committed.

```
$ git status
```

git log

The `git log` command shows a list of all the commits made to a repository. The commits are displayed in reverse chronological order, so the most recent commit is at the top. This command provides a history of your project, showing the changes made in each commit, as well as the author of the commit and the date it was made.

```
$ git log
```

These commands are just the tip of the iceberg when it comes to interacting with Git, but they will get you started on your journey. As we move along through the chapters, we'll introduce more commands and dive deeper into how Git works.

CLI vs GUI: Understanding the Differences

When interacting with Git, you have two main options: the command line or a Graphical User Interface (GUI). Both methods have their advantages and disadvantages. Let's explore what each offers and when it might be appropriate to use one over the other.

Command Line Interface (CLI)

The Command Line Interface (CLI) is a text-based way to interact with Git. As you've seen in the previous section, you issue commands by typing them into a terminal or console. This is the original way of interacting with Git, and it's powerful because it gives you direct access to all Git commands.

Advantages:

- Full access to all Git features.
- Automation: CLI allows you to write scripts to automate repetitive tasks.
- Might be faster for experienced users.

Disadvantages:

- Steeper learning curve for those not familiar with a command line.
- Requires memorizing commands and their options.
- Might be more error-prone for beginners.

Graphical User Interface (GUI)

GUI tools for Git are applications that provide a visual interface for interacting with repositories. These tools are often easier for beginners because they don't require memorizing commands.

Advantages:

- User-friendly: Visual representation makes it more intuitive.
- Easier to learn for those not familiar with the command line.
- No need to remember commands; options are usually available through menus.

Disadvantages:

- Might not have support for all Git features.
- Automation is limited compared to the command line.
- Some GUI tools might not be updated as frequently as Git itself.
- Might be more difficult to troubleshoot issues.

When to Use CLI or GUI?

- **Use the CLI** if you're comfortable with text-based commands, need to use advanced Git features, or want to automate tasks using scripts.
- **Use a GUI** if you're new to Git, prefer a visual representation, or don't have a need for advanced features and automation.

Ultimately, many Git users find that a combination of both the command line and a GUI tool is the most effective way to work. You might use a GUI for day-to-day tasks and the command line for more complex operations.

In the next section, we will explore how to perform some basic Git commands using a popular GUI tool: VSCode.

Performing Basic Commands in VSCode

Visual Studio Code, commonly referred to as VSCode, is a popular code editor that comes with built-in support for Git. This makes it an excellent tool for developers who prefer a graphical interface over the command line. In this section, we will explore how to perform some of the basic Git commands we learned earlier, but this time, using VSCode's Git integration.

Setting Up VSCode with Git

Before we begin, make sure you have both Git and VSCode installed on your computer. VSCode should automatically detect Git, but if not, you can set the path to Git in the VSCode settings.

Initializing a Git Repository

In the command line, we used `git init` to initialize a Git repository. In VSCode, you can do this with the following steps:

1. Open the folder (or directory) you want to use as a Git repository in VSCode.

2. Click on the Source Control icon in the Activity Bar on the side of the window. You'll recognize this as the icon with three connected circles, similar to the Git logo.

3. Click on 'Initialize Repository'. This will create a new Git repository in the current folder.

Alternatively, you can also initialize a Git repository from the VSCode Command Palette (`Ctrl + Shift + P` or `Cmd + Shift + P` on Mac). Type 'Git: Initialize Repository' and select the command.

Cloning a Repository

Instead of using `git clone` in the command line, in VSCode you can:

1. Go to the Command Palette (`Ctrl + Shift + P` or `Cmd + Shift + P` on Mac).

2. Type 'Git: Clone' and select the command.

3. Enter the URL of the repository and press `Enter`.

4. Choose the directory where you want to clone the repository.

Staging Changes

Remember `git add` in the command line? Here's how to do it in VSCode:

1. After making changes to your files, go to the Source Control view (the icon with three connected dots in the Activity Bar).

2. In the changes section, you'll see a list of all the modified files. Click on the "+" sign next to each file you want to stage, or click on the "+" sign at the top to stage all changes.

Committing Changes

Instead of using `git commit` in the command line, in VSCode you can:

1. Go to the Source Control view.

2. Enter a commit message in the text box at the top.

3. Click the commit button or press `Ctrl + Enter` (or `Cmd + Enter`) on Mac to commit the changes.

Checking Status and History

In the command line, we used `git status` and `git log`. In VSCode, much of this information is available visually:

1. In the Source Control view, you can see which files have changed.

2. For history, you can install an extension like Git History to view the commit logs.

VSCode provides a visually intuitive and easy way to interact with Git, especially for those who are not comfortable with the command line. However, as you become more experienced, you may find it useful to use a combination of the GUI and command line depending on the complexity of the tasks you are performing. But in most cases, the GUI should be sufficient for your day-to-day Git needs.

Creating Your First Repository and Making Your First Commit

Now that you are familiar with some basic Git commands and how to perform them in both the command line and VSCode, let's create your first Git repository and make your first commit. We'll go through this process step by step in both environments, so you can choose the one you are more comfortable with or even try both to see which one suits you better.

Using the Command Line

1. **Create a New Directory**: First, create a new directory (folder) on your computer where you want your project to live. You can do this using your file explorer or by using the command line.

   ```
   $ mkdir my-first-repo
   $ cd my-first-repo
   ```

2. **Initialize a Git Repository**: Use the `git init` command to initialize a new Git repository in this directory.

```
$ git init
```

3. **Create a File**: Create a new file in the directory, for example, `README.md`. You can do this using a text editor, or from the command line by typing:

```
$ echo "# My First Repository" > README.md
```

4. **Stage the File**: Use the `git add` command to stage the file, which means you are preparing it to be included in the next commit.

```
$ git add README.md
```

5. **Commit the File**: Finally, use the `git commit` command to save the staged changes. Make sure to include a descriptive message about what changes the commit includes.

```
$ git commit -m "Add README file with pro\
ject title"
```

6. **Check the Status**: Use the `git status` command to see the state of your repository. Since you've made a commit, it should tell you that your working directory is clean.

```
$ git status
```

Using VSCode (GUI)

1. **Create a New Directory**: Create a new directory on your computer where you want your project to live. You can use the file explorer for this step.

2. **Open VSCode**: Launch Visual Studio Code.

3. **Open the Directory**: Go to `File > Open Folder` and select the directory you just created.

4. **Initialize a Git Repository**: Click on the Source Control icon in the Activity Bar, and then click on 'Initialize Repository'.

5. **Create a File**: Create a new file in the directory called `README.md`. Type "`# My First Repository`" into the file and save it (`Ctrl + S` or `Cmd + S` on Mac).

6. **Stage the File**: In the Source Control view, click the '+' icon next to the `README.md` file to stage it.

7. **Commit the File**: Enter a commit message, such as "Add README file with project title", in the text

box at the top of the Source Control view. Click the commit button or press `Ctrl + Enter` (or `Cmd + Enter` on Mac) to commit the changes.

Nice! You've just created your first Git repository and made your first commit. Whether you chose the command line or VSCode, you now have a snapshot of your project that you can always revert back to if needed. As you work on more complex projects, you'll find that Git is an invaluable tool for managing and tracking changes.

Viewing the Commit History and Reverting To A Previous Commit

Being able to view the history of your commits and revert to a previous state is one of the most powerful features of version control systems like Git. In this section, we'll learn how to view the commit history and how to revert to a previous commit using both the command line and VSCode. We'll continue with the repository we just created.

Using the Command Line to View the Commit History

1. **Making Additional Changes**: For the purpose of this guide, let's make an additional change to the `README.md` file so that we can then revert it. Open `README.md` in a text editor, add a new line of text, and save the file.

2. **Commit the Change**: Stage the file and commit the change with a message indicating what you modified.

```
$ git add README.md
$ git commit -m "Add a new line to README"
```

3. **Viewing the Commit History**: Now, let's view the commit history by using the `git log` command. This will show you a list of all the commits made in the repository in reverse chronological order.

   ```
   $ git log
   ```

   You will see an output similar to this:

   ```
   commit d3e3f3a8f23f9deee330658f7 (HEAD ->\
    main)
   Author: Your Name <your.email@example.com>
   Date:   Thu Jun 18 11:00:00 2025 +0000

       Add a new line to README

   commit a4d5b8d8ed4b2e0b4567df2b2
   Author: Your Name <your.email@example.com>
   Date:   Thu Jun 18 10:45:00 2025 +0000

       Add README file with project title
   ```

   The commit hash, author, date, and commit message are displayed for each commit. You can navigate

through the log using the arrow keys and exit by pressing `q`.

4. **Reverting to a Previous Commit**: After committing the change, we can now revert it. The `git log` command will now show two commits. To revert to the initial commit, find the hash of the commit you want to revert to (which would be the hash of the "Add README file" commit in this case), and use the `git revert` command followed by the commit hash. This will create a new commit that undoes the changes.

   ```
   $ git revert a4d5b8d8ed4b2e0b456ad6a14b4a\
   99e8627df2b2
   ```

   Save and close the text editor if it pops up requesting a revert commit message. This will effectively undo the changes made in the "Add a new line to README" commit, bringing your `README.md` file back to its initial state.

Using VSCode to View the Commit History

1. **Viewing the Commit History**: To view the commit history in VSCode, it's best to install an extension like

Git History. After installing it, you can access it by clicking on the Git icon on the activity bar and then on the clock icon at the top. This will show you a list of all the commits.

2. **Making Additional Changes**: Similar to the CLI example, open the `README.md` file within VSCode, add a new line of text, and save the file.

3. **Commit the Change**: Go to the Source Control view, stage the change by clicking the '+' icon next to the `README.md` file, enter a commit message, and press `Ctrl + Enter` or `Cmd + Enter` on Mac to commit the change.

4. **Reverting to a Previous Commit**: Go to the Git History view by clicking on the Git icon and then the clock icon. Right-click on the commit you want to revert to, and select 'Revert this Commit'. This creates a new commit that undoes the changes.

And that's it! Now you know how to view and revert commits in your repository, and how you can use this powerful feature to undo mistakes or simply explore what the project looked like at any point in its history. Understanding these concepts is fundamental in using Git effectively for version control. And whether you prefer

the command line or VSCode, Git has you covered.

3. Branching and Merging in Git

This chapter introduces the concepts of branching and merging, two powerful features in Git that enable developers to work on different features or bugs in isolation. You'll learn how to create and switch between branches, merge them, and resolve merge conflicts. You'll also learn how to do all these operations in Visual Studio Code.

Understanding Branches

If you think of your Git repository as a tree, your project starts as a single trunk, growing taller with each commit you make. However, sometimes you might want to try out new ideas, make some experimental changes, or work on a new feature without affecting the main part of the project. In such cases, branches come to the rescue.

Imagine your project as a tree again, but this time, besides the main trunk, it has smaller branches sprouting out. Each branch represents a new timeline, allowing for independent development and experimentation. This also helps with collaboration, as multiple people can work on different branches simultaneously without affecting each other's work. Once you're satisfied with the changes in a branch, you can merge it back into the main trunk.

Main Branch

By default, every Git repository has a main branch, often called the `master` branch (or `main` in more recent repositories). This is usually considered the definitive, stable version of your project. All other branches you create will be based on this branch, or a branch that was based on this one, and so on.

Feature Branches

When you're working on a new feature or an experimental idea, you create a new branch, commonly known as a feature branch. The feature branch is created from the main branch, so it contains all its code and history at the point of creation. This allows you to work on your new ideas separately. If your experiments are successful, you can later integrate them back into the main branch. If not, you can simply discard the branch without any harm done to your project.

Why Use Branches?

1. **Isolated Environment**: Branches provide a safe, isolated environment to work on new tasks or features without disturbing your primary codebase.

2. **Collaboration**: When working in a team, branches make it possible for developers to work on different features at the same time without running into conflicts.

3. **Easy Integration**: When the work on a branch is complete and tested, it can be seamlessly integrated back into the main branch.

4. **Risk Mitigation**: Branches offer a safety net. If something goes wrong in a branch, your main branch remains untouched, and you can always return to it.

In the following sections, we'll learn how to create and switch between branches both from the command line and using Visual Studio Code's graphical interface for Git. This is where Git starts showing its real power in flexibly handling project versions. Let's keep learning!

Creating and Switching Between Branches

Now that you understand what branches are, let's see how you can create and switch between them. Git makes this process straightforward and intuitive.

Creating a New Branch

To create a new branch in Git from the command line, you can use the `git branch` command followed by the name of your new branch:

```
$ git branch feature-x
```

In the above example, `feature-x` is the name of the new branch you have created. Branch names should be meaningful and indicative of the work being done on the branch. Running the command `git branch` without any arguments will list all the branches in your repository:

```
$ git branch
```

You should see an output similar to this:

```
  feature-x
* main
```

The current active branch is indicated by an asterisk.

Switching Between Branches

Creating a branch does not automatically switch to it. To move from one branch to another, you need the `git checkout` command. Let's switch to the `feature-x` branch we just created:

```
$ git checkout feature-x
```

Now, you're working on the `feature-x` branch, and any commits you make will be on this branch and not affect the `main` branch. You can switch back to the `main` branch at any time using the same `git checkout` command:

```
$ git checkout main
```

Creating and Switching in One Step

Git also allows you to create a new branch and switch to it in one command using the `-b` flag with the `git checkout` command:

```
$ git checkout -b feature-y
```

The command above creates a new branch named `feature-y` and immediately switches to it. This is a handy shortcut that you will find yourself using frequently.

Let's make a new commit on the `feature-y` branch and then switch back to the `main` branch. First, add a new line to the `README.md` file with the text `Working on feature-y` and save the file. Then, commit the changes to the `feature-y` branch:

```
$ git add README.md
$ git commit -m "Add feature-y text to RE\
ADME.md"
```

Now, switch back to the `main` branch:

```
$ git checkout main
```

When you switch back to the `main` branch, you will notice that the new line you added to the `README.md` file is no longer there. This is a good way to keep your main branch clean and stable while you work on new features.

Let's learn how to merge branches so that we can integrate the changes from the `feature-y` branch into the `main` branch.

Merging Branches

Once you've completed your work on a feature branch, the next step is to incorporate those changes back into the main branch, or another branch if needed. This is known as **merging**.

Let's continue with our example. We made some changes in the `feature-y` branch, and now we want to integrate those changes into the `main` branch. To do this, you need to be on the `main` branch. That's because when we merge, the changes are applied to the currently active branch. If you're not already on the `main` branch, switch to it using the `git checkout` command.

With the `main` branch active, we can use the `git merge` command to merge the changes from the `feature-y` branch:

```
$ git merge feature-y
```

This command combines the content of the `feature-y` branch with the `main` branch.

Fast-Forward Merges

In our example, since there were no new changes in the `main` branch after we created the `feature-y` branch, Git performed a "fast-forward" merge. This means that Git simply moved the `main` branch pointer to the last commit of the `feature-y` branch, without needing to create a separate merge commit.

Merge Commits

However, sometimes both the branch you are merging into and the branch you are merging from might have changes that diverge. In such cases, Git will create a new commit that brings together the changes from both branches. This is known as a merge commit.

Handling Merge Conflicts

There are times when changes in the branches involve the same lines in the same files, and Git isn't sure which version to use. This results in a merge conflict. Git will need your help to resolve these conflicts. We'll delve deeper into resolving merge conflicts in the next section.

Viewing the Merge

After the merge is complete, it's a good practice to review the changes to ensure everything is as expected. You can view the history of your commits to see the merge. The `git log` command is useful for this:

```
$ git log --oneline
```

This command produces an output similar to the following:

```
b1a2b3c (HEAD -> main) Merge branch 'feat\
ure-y'
d4e5f6g Add feature-y text to README.md
h7i8j9k Add a new line to README
l0m1n2o Add README file with project title
```

The `--oneline` flag displays each commit on a single line, making it easier to read. `HEAD` is a pointer to the current branch - in this case, the `main` branch.

Merging branches in Git is a crucial aspect of version control, enabling the integration of different lines of development. In the next section, we'll focus on resolving merge conflicts, an essential skill for maintaining a clean codebase. Later, we'll explore how Visual Studio Code

can simplify the branching and merging process through its intuitive graphical interface.

Resolving Merge Conflicts

In the previous section, we introduced the concept of merge conflicts — situations where Git cannot automatically merge changes because two branches have made different modifications to the same part of a file. Now, let's dive into how you can resolve these conflicts.

What are Merge Conflicts?

Before we dive into resolving merge conflicts, it's essential to understand why they occur. Merge conflicts generally happen when two branches have changed the same line in conflicting ways, or when a file is deleted in one branch but modified in the other. It's like two people trying to edit the same sentence in a document at the same time. Git gets confused and asks for your help to decide which change to keep.

When you attempt to merge these branches, Git will throw a message like:

```
Auto-merging README.md
CONFLICT (content): Merge conflict in REA\
DME.md
```

```
Automatic merge failed; fix conflicts and\
 then commit the result.
```

This means that Git was not able to resolve the differences automatically, and it's waiting for you to step in and make the decision.

How Does Git Indicate Conflicts?

When a merge conflict occurs, Git will modify the affected files to visually indicate the parts causing the conflict. Conflict markers are added to show you the conflicting changes. These markers are:

- `<<<<<<< HEAD` marks the beginning of the conflicting changes in the current branch (usually the branch you are merging into).
- `=======` separates the conflicting changes in the current branch from those in the branch you're trying to merge.
- `>>>>>>> [branch-name]` marks the end of the conflicting changes in the branch you're merging.

It looks something like this:

```
<<<<<<< HEAD
This is the change on the current branch.
=======
This is the conflicting change on the bra\
nch you're merging.
>>>>>>> feature-y
```

Resolving Conflicts Manually

Now that you understand how Git shows you the conflicts, you can resolve them by manually editing the file. Essentially, you have to decide which changes to keep or make a new change that resolves the conflict.

When editing the file, you might wonder if you need to keep the conflict markers. The answer is no. The conflict markers are there to guide you in locating and understanding the conflicts. Once you decide on the changes to keep, you should remove the conflict markers.

For example, if you want to keep the change from the branch you're merging, your file should look like this after editing:

```
This is the conflicting change on the bra\
nch you're merging.
```

After you have made your changes and removed the conflict markers, you need to tell Git that the conflict has been resolved. To do this, stage the file by using the `git add` command:

```
$ git add README.md
```

Finally, it's time to cement the resolution with a commit. This commit will include the resolved changes:

```
$ git commit -m "Resolved merge conflict \
in README.md"
```

Using Tools to Resolve Conflicts

Editing files and removing conflict markers manually isn't the only way to resolve conflicts. There are various tools that provide a more visual or guided interface to help you. For instance, VSCode has a built-in merge conflict resolution tool that highlights conflicts and allows you to easily choose which changes to keep. We will explore this in the last section of this chapter.

The Significance of Merge Conflicts

While merge conflicts can be intimidating, especially for newcomers, it's important to understand that they're not inherently bad or indicative of a mistake. They are merely Git's way of communicating that it needs human input to proceed. With practice, resolving merge conflicts becomes just another routine aspect of collaborative coding.

In the next section, we'll explore how to handle branching and merging using VSCode. This includes using the GUI for creating branches, switching between them, merging them, and even resolving conflicts.

Branching and Merging Using VSCode

The command line is an essential tool for Git, but using a Graphical User Interface (GUI) like Visual Studio Code (VSCode) can offer a more intuitive and visual way to handle Git operations, especially for beginners. Now let's explore how to perform branching and merging using VSCode. It's important to recognize that GUIs can change over time, so depending on when you're reading this, the steps below may not be exactly the same. But the general concepts should still be similar.

Creating a Branch with VSCode

1. Open your Git repository in VSCode.

2. Look at the lower-left corner of the window. You will see the current branch name (default is `main` or `master`). Click on this.

3. A window pops up at the top center of the screen showing a list of branches and some options. Click on "+ Create new branch".

4. Type in the name of your new branch, e.g., `feature-x`, and press Enter.

Switching Between Branches using VSCode

1. Click on the current branch name at the bottom left corner again.

2. A window pops up showing a list of branches. Click on the branch you want to switch to.

Making a Commit using VSCode

1. Make some changes to a file in your project.

2. Click on the Source Control icon on the left sidebar (or press `Ctrl + Shift + G`).

3. You'll see the changes listed. Type in a commit message in the text box at the top.

4. Click the checkmark at the top or press `Ctrl + Enter` to commit the changes.

Merging Branches using VSCode

1. First, switch to the branch that you want to merge **into** (e.g., `main`).

2. Click on the ellipsis (...) on the Source Control title bar and select "Branch", then "Merge Branch".

3. In the popup window, select the branch you want to merge from.

4. If there are no conflicts, the branches will merge automatically. If conflicts arise, you'll get a notification.

Resolving Merge Conflicts using VSCode

1. When a merge conflict happens, you can navigate to the conflicting file by clicking on the file name in the Source Control panel.

2. In the file, you'll see the conflict markers showing the conflicting changes, similar to what you'd see in the command line.

3. VSCode provides buttons for "**Accept Current Change**", "**Accept Incoming Change**", "**Accept Both Changes**", and "**Compare Changes**". Clicking these will resolve the conflict as indicated, and automatically remove the conflict markers.

4. After resolving the conflicts, don't forget to stage (+ button or `git add` in the terminal) and commit the changes to finalize the merge.

Remember that while VSCode provides these handy features for handling Git operations, it's essential to understand the underlying Git commands and concepts to fully grasp what's happening when you click these buttons. This way, you'll be more equipped to resolve any issues that might arise and to use Git more effectively.

4. Introduction to GitHub

In this chapter, we'll introduce GitHub, a popular platform for hosting Git repositories. You'll learn how to create a GitHub account, explore its interface, and create your first GitHub repository.

What is GitHub?

Wow, you've come a long way already! You've taken your first steps into the world of version control, learned about Git, its core concepts like commits, branching, and merging, and even used both the command line and a graphical user interface with VSCode. That's impressive, and you should be proud of yourself!

Now it's time to up the ante and dive into a platform that brings even more power to your version control skills – GitHub. If you recall, we touched on GitHub in the first chapter, but let's dig a bit deeper and see why it's such a crucial tool for developers around the world.

GitHub is an online platform that utilizes Git for version control and collaboration. It allows you to work on projects with other people, keep track of issues, and even host web pages. Essentially, it takes all the version control goodness from Git and wraps it in a user-friendly interface that's accessible from anywhere.

GitHub vs Git - A Quick Recap

It's important to reiterate the distinction between Git and GitHub. Git is a distributed version control system that tracks changes in files. Although Git can facilitate

collaboration between different users, it doesn't provide a visual interface or a centralized place for repositories.

GitHub, on the other hand, is a web-based platform built on top of Git. It integrates the distributed version control and source code management functionality of Git with its own features, such as bug tracking, feature requests, task management, and wikis for each project.

Why is GitHub Needed?

While Git is super powerful, it can be a bit isolated when working on your own. GitHub takes your coding projects to the cloud, allowing for robust collaboration features. This means that whether you're a team of one or one thousand, you can work together effectively. It's especially popular in open-source projects (*projects where the code is available to the public and can be modified and used by anyone*).

Additionally, having a centralized online place for your repositories is a great backup. If something goes wrong locally, you can always get the code from GitHub. And you also have the option of keeping your repositories private on GitHub, so only you and anyone you give permission to can access them.

Alternatives to GitHub

While GitHub is incredibly popular and widely used among developers for hosting Git repositories, it's not the only option available. There are other platforms that offer similar functionalities, with some distinct advantages:

GitLab

GitLab is a highly regarded alternative to GitHub. Like GitHub, it offers features such as issue tracking, continuous integration/continuous deployment (CI/CD) pipelines and a web-based graphical interface. One major advantage of GitLab is that it is open source, which means that you can download and install GitLab on your own infrastructure, giving you full control and customization options. Additionally, GitLab offers self-hosted instances, allowing organizations to maintain their code repositories behind their own firewalls, which is ideal for those with strict data security requirements.

Bitbucket

Bitbucket, owned by Atlassian, is another popular option especially favored by teams already using other Atlassian products like Jira (a project management tool) or Confluence (a team collaboration software). Bitbucket integrates

seamlessly with these products and offers unlimited private repositories for small teams for free. It also supports both Git and Mercurial as version control systems and offers built-in CI/CD pipelines.

SourceForge

SourceForge has been around since the late 1990s, making it one of the pioneers in code repository hosting. Like GitHub, it provides a web-based interface for managing repositories, but in addition to Git, it also supports other version control systems such as Subversion and Mercurial. SourceForge is particularly popular among open-source projects and offers features such as project web hosting, forums, and mailing lists. However, its interface is dated compared to newer platforms like GitHub and GitLab, and it does not offer the same level of integration with modern development tools.

Let's get you set up with GitHub and explore its powerful features! In the next section, you'll learn how to create a GitHub account. This will be the first step in unlocking the collaborative power of version control with GitHub.

Creating a GitHub Account

Alright, it's time to create a GitHub account. This is a straightforward process, and best of all, it's free! Now, just a heads-up: website interfaces tend to evolve, so the exact steps and the look of the pages might change. However, the essential elements usually stay the same. So, if the GitHub signup page looks a bit different from what's described here, don't worry! You should still be able to follow along. Here's a general guide to creating a GitHub account:

1. **Navigate to GitHub's Website**: Open your web browser and go to GitHub's official website - www.github.com[1]. Typically, you can find the option to create an account (or sign up) prominently displayed on the homepage.

2. **Sign Up**: Look for a "Sign up" button or link, usually located at the top-right corner or in the center of the page. Click on it to get started.

3. **Enter your Details**: You'll probably be asked for a few pieces of information:

[1] https://www.github.com

- **Username**: Choose a unique username. This will be public, and others will use it to identify you on GitHub.

- **Email Address**: Provide an email address you frequently check. GitHub will use this for communication, such as sending notifications or verifying your account.

- **Password**: Create a strong password to secure your account. Aim to use a combination of letters, numbers, and special characters.

4. **Complete the Security Check**: There might be a captcha or a similar security measure to ensure that you're a human and not an automated bot creating an account.

5. **Select a Plan**: You might be asked to choose between different plans. For beginners or personal projects, the free plan is usually sufficient. It allows you to create public and private repositories. You can always upgrade to a paid plan later if you need additional features.

6. **Fill in Optional Information**: Sometimes, websites ask for additional information such as your coding

experience or areas of interest. This is usually optional and helps in personalizing your experience on the platform.

7. **Verify your Email Address**: This is an important step. Check your email for a message from GitHub and verify your email address by clicking on the link provided in the email.

8. **Complete the Setup**: Follow any remaining on-screen instructions to complete your account setup.

And there you have it! You're now the proud owner of a GitHub account. Take a moment to explore, customize your profile, and maybe even jot down ideas for your first project.

Next, we'll get acquainted with the GitHub interface. Remember that practice makes perfect, so don't be afraid to click around and learn by doing.

Exploring the GitHub Interface

Once you've set up your GitHub account, the next step is to understand how to navigate and utilize this powerful tool. While GitHub's interface may evolve, the core functionalities tend to remain the same.

GitHub Homepage

Upon logging into GitHub, you land on your dashboard. Here you'll find:

- **Repositories**: A section where you can quickly create a new repository or access your existing ones.
- **Feed**: This shows you the latest activity from repositories you're watching or people you follow.
- **Explore**: A place where GitHub suggests repositories that might interest you based on your activity.

Navigation Bar

Regardless of where you are on GitHub, there's a constant Navigation Bar. It typically contains:

- **Pull Requests**: An area where you can see your pull requests. We'll learn more about pull requests in a later lesson.

- **Issues**: A section where your issues are listed. We'll also cover issues in a later lesson.

- **Notifications**: An overview of all your notifications.

- **Profile and Settings**: Click on your profile picture to find these options.

Profile Page

Access your profile by clicking on your profile picture and selecting "Your profile". Your profile page is where you can:

- **Customize Your Profile**: Add a profile picture, write a short bio, and add a link to your website or portfolio.

- **View Your Repositories and Contributions**: A summary of your repositories and an overview of your contributions to GitHub projects.

Repository Page

Clicking on a repository, yours or someone else's, will take you to its page. This page usually contains:

- **Code**: This is where the files and directories of the project are displayed.
- **Issues**: Here, users can report bugs, ask questions, or suggest enhancements.
- **Pull Requests**: Any proposed changes to the repository are listed here.
- **Actions**: This is where you set up and monitor automated workflows.
- **Security**: This tab provides an overview of any security issues associated with your repository.
- **Insights**: This gives you a detailed analysis of the repository, including contributions and repository traffic.

Creating and Editing Files

Inside a repository, you can create new files or edit existing ones. Look for options to add a file or edit an existing one.

Account Settings

To reach your account settings, click on your profile picture and select "Settings". This is where you can set your preferences, such as email notifications and account security.

As you spend more time on GitHub, you'll become more comfortable with the platform. It's a powerful tool, so take your time and explore its many features. We'll discuss creating your first GitHub repository and using GitHub for version control in the next section. Get ready!

Creating Your First GitHub Repository

Now that you've explored the GitHub interface, it's time to dive into the practical part – creating your first GitHub repository. A GitHub repository, or "repo" for short, is a place where you store your project files and file-revision history. Repositories can also have associated features, such as wikis for documentation, and issue trackers for bug reports and feature requests.

Step 1: Starting the Creation Process

Log into your GitHub account and click on the "+" icon at the top right corner of the page. From the drop-down menu, select "New repository."

Step 2: Set Up Your Repository

You'll be taken to a page where you can set up your new repository. Here's what you need to do:

- **Repository Name**: Choose a name for your repository. It should be something descriptive so that you

(and others) can tell what the project is about at a glance.

- **Description (optional)**: Though optional, it's good practice to add a short description of your project.

- **Visibility**: Here, you can decide whether your repository should be public (accessible to anyone) or private (only accessible to you, and people you grant access to).

- **Initialize this repository with**: You can choose to add a README file, a .gitignore file, or choose a license. For now, just tick the box next to "Add a README file."

  - A **README file** is a document that explains what the project is, how to use it, and other vital information. GitHub displays the contents of your repo's README file on the repository's front page.

  - A **.gitignore file** specifies intentionally untracked files that Git should ignore - this is handy for files or folders that you don't want to track or share via GitHub.

  - A **license file** is a document that lets others

know what they can and can't do with your source code.

Once you've filled out this page, click on the "Create repository" button.

Step 3: Explore Your New Repository

Nice! You've created your first GitHub repository. You will now be redirected to your new repository's main page, where you can see your README file displayed.

The README is written in a language called Markdown (.md), which is a lightweight syntax for creating formatted text documents. It includes conventions for styling text, tables, headers, links, and more. You can learn more about Markdown here[2].

Remember that your repository is more than a storage space for your files – it's also a space for managing your project. So, it's important to get familiar with the different features of your repository that GitHub provides.

You've successfully created your first GitHub repository! This is a significant step in your journey. As we proceed, you'll learn more about collaborating with others, man-

[2] https://guides.github.com/features/mastering-markdown/

aging changes, and integrating your local Git knowledge with GitHub.

5. Remote repositories with GitHub

It's time to connect the dots between the local version control you've been working with and the online collaboration that platforms like GitHub allow.

This chapter focuses on remote repositories hosted on GitHub. You'll learn how to connect a local Git repository to GitHub, clone a repository from GitHub, and push your changes to GitHub.

Understanding Remote Repositories

You've learned about repositories, or "repos", in the context of Git. These repositories contain your project's files and the history of changes made to them. When working with Git, you've been dealing with local repositories that reside on your computer. Remote repositories, on the other hand, are versions of your project that are hosted on the internet or another network.

Remote repositories serve several purposes:

1. **Backup and versioning**: Remote repositories act as a backup for your code, safeguarding it in case something happens to your local machine. In addition to this, they also store the version history of your project. This means you can retrieve your latest code from the remote repository and, if necessary, revert to a previous version of your code, providing an additional layer of security for your project's evolution.

2. **Collaboration**: Remote repositories facilitate collaboration among developers. By pushing code to or pulling code from the remote repository, developers

can share their work, integrate changes from others, and keep their local repositories aligned with the collective project progress.

3. **Centralization**: Despite Git's decentralized structure, a remote repository serves as a centralized hub for the project's codebase. This becomes the main reference point for the latest, stable version of the project, providing a clear overview for all collaborators.

4. **Accessibility**: Remote repositories can be accessed from different machines and locations. This flexibility allows you to work on the go or collaborate with a distributed team, ensuring the codebase is always within reach.

How Do Remote Repositories Work?

Remote repositories work hand in hand with local repositories. When you make changes to your project locally, you have to explicitly "push" those changes to the remote repository to update it. Conversely, if someone else has made changes to the remote repository, or if you have pushed changes from a different machine, you would need to "pull" these changes into your local repository.

While Git enables the use of multiple remote repositories, the typical practice, particularly among smaller teams, is to maintain a single primary remote repository. This primary repository often serves as the main hub for the codebase, ensuring everyone has access to the most recent, stable version of the project. However, having multiple remote repositories can be beneficial in complex projects, for instance, when you want to have separate repositories for development, staging, and production environments.

Git also allows you to assign names to your remote repositories for easy reference. By default, Git assigns the name "origin" to the primary remote repository. This name can be changed as per your project needs or left as is for consistency with common Git practices.

Now that you've understood what remote repositories are, let's see how to connect your local Git environment to a GitHub remote repository. In the next section, you'll learn how to connect Git with GitHub, push local changes to a remote repository, and pull updates from it. We'll be using GitHub as our remote repository host, but the same principles apply to other Git hosting services like GitLab and Bitbucket, even though the steps may vary.

Connecting Git with GitHub

To take advantage of the collaborative features offered by GitHub, we need to establish a link between your local Git repository and a GitHub repository, effectively transforming the latter into a remote repository. Let's walk through the steps required to set up this connection.

Adding a Remote Repository

Assuming you have a GitHub account and a local Git repository (from previous sections), here's how you link the two:

1. **Create a new repository on GitHub**: Log in to your GitHub account and click the '+' icon in the top right corner, then select 'New repository'. Give it a name (preferably the same name as your local repository for consistency), an optional description, and choose whether you want it to be public (anyone can see it) or private (only you and those you invite can see it). Skip the 'Initialize this repository with a README' option for now. Click on 'Create repository'.

2. **Connect your local repository to GitHub**: Navigate to your local repository on your computer

via the command line. Here, you can tell Git where your remote repository is using the `git remote add` command followed by the alias you want to give to the remote repository (commonly "origin") and the URL of the repository. You can get this URL from your newly created GitHub repository page; it should look like "https://github.com/yourusername/yourrepositoryname.git". Here's the command:

```
git remote add origin https://github.com/\
yourusername/yourrepositoryname.git
```

3. **Verify the remote repository**: To ensure that the remote repository has been added correctly, you can use the `git remote -v` command. This will list all the remote repositories connected to the local one, along with their URLs. If everything went well, you should see your GitHub repository URL listed under the alias "origin".

You have now linked your local Git repository to a remote GitHub repository! This connection is an important foundation for collaborating on projects and syncing your work between your local machine and GitHub.

Connecting with VSCode

If you prefer a graphical user interface, Visual Studio Code (VSCode) provides a seamless way to connect your local Git repository to GitHub. Here's how you can do it:

1. **Open your local repository in VSCode**: Launch VSCode and open the folder containing your local Git repository. To do this, click on 'File' and then 'Open Folder'. Navigate to the folder containing your local repository and select it.

2. **Install the GitHub Extension**: VSCode has a rich ecosystem of extensions that can add extra functionality to your workflow. For GitHub integration, you'll need to install the official GitHub extension. You can find this by going to the Extensions view (click the square icon in the sidebar or use the shortcut `Ctrl + Shift + X`), and then searching for "GitHub". Install the extension by clicking on the 'Install' button.

3. **Sign in to your GitHub account**: Once you've installed the GitHub extension, VSCode may prompt you to sign in to your GitHub account. If not, you can initiate this by clicking the 'Sign In' button in the bottom-left corner of VSCode.

4. **Add the Remote Repository**: With your local repository open in VSCode, go to the Source Control view by clicking the branch icon in the sidebar. At the top, click on the '...' (More Actions) button, and then choose 'Add Remote'. A prompt will ask you for the remote's name. It's customary to name your main remote "origin", but you can choose another name if you prefer. The prompt will then ask for the URL of the remote repository. This is the URL you copied from GitHub earlier when you created the repository, for example, "https://github.com/yourusername/yourrepositoryname.git".

5. **Verify the Remote Repository**: You can confirm that the remote repository has been successfully added by checking the bottom-left corner of VSCode. You should see a cloud icon with an arrow pointing upwards (push) and another pointing downwards (pull). These icons indicate that VSCode is now connected to your remote repository on GitHub.

And that's it! You've now connected your local Git repository to GitHub using VSCode.

In the following sections, we will cover how to clone a repository (which is another way to get a local copy of a project), how to push your local changes to GitHub, and how to pull updates from a remote repository into your

local repository. These are key workflows in the use of Git and GitHub for version control and collaboration, so let's keep moving!

Cloning a GitHub Repository

We've talked about cloning a little bit in chapter 2. Now, let's dive into the depths and understand exactly what it means to clone a repository, the various scenarios in which cloning is useful, and how to effectively clone repositories using both the command line and VSCode.

What Does Cloning Mean?

Cloning a repository essentially means creating a local copy of a remote repository on your machine. This includes not just the files and code, but also the history of commits and branches. It's like taking a snapshot of the repository at that moment and downloading it to your local machine, where you can work on it as if it was just another Git repository.

Why and When to Clone a Repository?

Contributing to a Project:

If you come across an open source project or any project on GitHub to which you'd like to contribute, you would start by cloning it. This gives you a local working copy

that you can modify. After making your changes, you can push them back to the remote repository (if you have permissions), or open a pull request.

Working Across Multiple Machines:

If you work on multiple machines, such as a desktop at home and a laptop on the go, cloning can keep your work synchronized. You can clone your repository on both machines, and as long as you regularly pull the latest changes and push your updates, you'll have consistent code on both.

Creating a Backup:

Cloning a repository can also serve as a form of backup. By having a copy of your code and its history on your local machine, you have an extra layer of security against data loss.

Starting a New Project:

If you're starting a new project, you can clone a repository as a starting point. This is useful if you want to use a project template or boilerplate code to get started quickly.

Cloning with the Command Line

Let's go through the process of cloning a repository using the command line:

1. **Copy the Repository URL**: Go to the main page of the repository on GitHub. Click the 'Code' button and then copy the URL from the 'Clone' section.

2. **Clone the Repository**: Open your command line or terminal, and use the `git clone` command followed by the URL of the repository you want to clone:

   ```
   git clone https://github.com/username/rep\
   ository.git
   ```

   Replace the URL with the one you copied from GitHub. This command creates a new directory with the same name as the repository and downloads the code and version history into this new directory.

Cloning with VSCode

You can also clone repositories directly from VSCode:

1. Open VSCode.

2. **Clone the Repository**: In the 'Start' view, click on 'Clone Git repository...'. Alternatively, use the Command Palette (`F1` or `Ctrl + Shift + P`) and search for 'Git: Clone'. Enter the URL of the repository you want to clone.

3. **Choose a Location**: VSCode will ask you to select a directory for the repository. Navigate to where you'd like the cloned repository to be located and confirm the location.

Once the clone is complete, VSCode will ask if you'd like to open the cloned repository. Click 'Open' to start working on it.

You now have a good grasp of what cloning is, why it's important, and how to do it using both the command line and VSCode. In the next section, we will tackle how to keep your local and remote repositories in sync by pushing to and pulling from GitHub. Congrats on making it this far!

Pushing to and Pulling from GitHub

Now that we've explored cloning, it's time to tackle two other crucial operations that will ensure the synchronization between your local and remote repositories: pushing and pulling. These operations allow your local Git repository and your GitHub repository to communicate and share changes, ensuring that both repositories are up-to-date with the latest changes.

Pushing to GitHub{!: Pushing}

When you make changes to your local repository—like committing a change to a file—those changes are only reflected in your local repository. The remote repository on GitHub isn't automatically updated. To update the remote repository with your local changes, you use the `git push` command.

In the command line, this would look like:

```
$ git push origin main
```

The `git push` command tells Git to push your commits to the remote repository. The word 'origin' is an alias that

stands for the URL of your remote repository on GitHub. And 'main' is the branch that you're pushing.

After running the `git push` command, your changes from the `main` branch in your local repository are now also in the `main` branch of your remote repository on GitHub.

Pulling from GitHub{!: Pulling}

While `git push` is used to send commits from your local repository to your remote repository, `git pull` is used to fetch changes from the remote repository and merge them into your current local working branch.

Let's say you're working on a project with a team, and one of your teammates has pushed some changes to the repository on GitHub. To get those changes into your local repository, you'd use the `git pull` command. Here's what that command looks like:

```
$ git pull origin main
```

The `git pull` command tells Git to fetch the changes from the remote repository (origin) and merge them into your current local working branch (main).

Using VSCode for Pushing and Pulling

VSCode makes it easy to push and pull changes to and from GitHub without having to leave your editor or use the command line. Here's how to do it:

Pushing with VSCode:

1. **Commit your changes**: Make some changes to your code, then stage and commit those changes using the Source Control panel.

2. **Push the changes**: Clicking the three dots at the top of the Source Control panel will open a menu. Click on the 'Push' button to push your changes to GitHub. Alternatively, After making your commit in the Source Control panel, you might see a button that says "Sync Changes". Click it to push your changes to GitHub. You can also click the circular button with an up arrow beside it at the bottom-left of VSCode, where you have the branch name. This will either push or pull depending on whether you have new commits in the local or remote repository.

Pulling with VSCode:

1. **Open the Source Control Panel**: On the left side of VSCode, click on the 'Source Control' icon to open the Source Control panel.

2. **Pull the changes**: Clicking the three dots at the top of the Source Control panel will open a menu with a 'Pull' button. You can also click the circular button with an a down beside it at the bottom-left of VSCode, where you have the branch name.

Remember, regular pushing and pulling are important to keep your repositories in sync, especially when collaborating with others.

6. Collaborating on GitHub

In this chapter, we'll cover the collaborative features of GitHub, including adding collaborators to a repository, forking a repository, raising pull requests and creating issues. We'll also discuss the process of code reviews and merging pull requests on GitHub.

Understanding Collaborators and Permissions

By now, you've gained a solid understanding of how to work with Git and GitHub. You've learned to make commits, manage branches, and synchronize your local repository with a remote one on GitHub. As we've mentioned before, one of the main advantages of using GitHub is its capacity to facilitate collaboration. So, let's dive into that aspect, starting with the concept of collaborators and permissions.

In GitHub, a collaborator is a person who has been granted permission to a repository. This role isn't limited to coding; collaborators can contribute to the project in various ways, including writing documentation, testing, and managing issues and pull requests (which we'll discuss in detail later in this chapter).

Permissions control what collaborators can and can't do in a repository. There are two main types of access permissions in GitHub:

- **Owner**: The repository owner has full control over the repository. They can add and remove collaborators, change permissions, and delete the repository.

- **Collaborator**: Collaborators have different levels of access depending on the permissions granted to them by the owner. They can typically view and modify the repository's contents, but they may not be able to make administrative changes.

In addition to these two main types of permissions, there are also a few other permissions that can be granted to users:

- **Maintainer**: Maintainers have the same permissions as collaborators, but they can also add and remove other collaborators.

- **Triage**: Triage users can view and manage issues and pull requests, but they cannot make changes to the repository's contents.

- **Auditor**: Auditors can view all of the activity in a repository, including commit history, pull requests, and issue discussions.

Note that the permissions for a repository are managed at the individual repository level. This means that a user can have different access levels to different repositories.

Repository Visibility and Collaborators

When you create a new repository on GitHub, you can choose to make it either public or private. This setting determines who can see and access the repository.

Another important aspect to consider is repository visibility, which can be set to either **public** or **private**.

- **Public repositories** are visible to everyone and can be forked by anyone. However, only collaborators with the appropriate permissions can make changes that directly affect the repository.

- **Private repositories** are only visible to you and the collaborators you invite. The access to a private repository is more controlled, and you can precisely manage who has access to the repository and what they can do.

Understanding how collaborators and permissions work is crucial for a smooth collaboration process. It helps ensure that every team member has an appropriate level of access to contribute effectively while maintaining the integrity and security of the project.

In the next section, we'll discuss how you can invite collaborators to a repository and manage their permissions.

Inviting Collaborators to a Repository

As you work on your projects, you'll likely want to invite others to collaborate with you. This could be colleagues or friends who you want to work with on a project. Let's learn how you can invite collaborators to a repository and manage their permissions right from GitHub. Remember, these steps might vary slightly depending on how much GitHub changed its interface since this guide was written, but the general idea should be the same.

1. **Navigate to your repository**. Start by logging in to your GitHub account and opening the repository to which you want to add collaborators.

2. **Open the repository settings**. Once inside your repository, click on the "Settings" tab, which is located in the row of tabs at the top of the repository page.

3. **Open the 'Collaborators' tab**. In the left sidebar, click on the "Collaborators" tab. Here, you'll see a list of all the collaborators who have access to the repository. If you haven't added any collaborators yet, this list will be empty.

4. **Invite a collaborator**. Click the "Add people" button. Here, you'll be asked to enter the username, full name, or email address of the person you want to invite. As you type, GitHub will suggest matches from users with whom you've interacted before, within your organization or outside of it.

5. **Choose permissions**. Once you've added the person, you can select the level of access they should have from the dropdown menu. Make sure to give only the necessary permissions based on what they need to do.

6. **Send the invitation**. After you've selected the permissions, click the "Add [username] to [repository name]" button. They will receive an email with the invitation, and once they accept it, they will have access to the repository with the permissions you've specified.

Inviting collaborators is a crucial step in opening up your project for contributions from others. In the next sections, we'll explore even more ways to collaborate on GitHub by forking repositories and creating pull requests. Keep up the great work!

Forking a Repository

Forking is one of the fundamental concepts on GitHub that allows you to create a copy of a repository under your own account. This enables you to experiment with changes without affecting the original project. It's especially popular in open-source projects, where it's often used as a way to contribute. Let's dive into what forking is, why you might want to do it, and how it's done.

What Does Forking Mean?

When you fork a repository, you're making a complete copy of the repository and all its history onto your own GitHub account. This copy is independent of the original repository. You have full control over this forked repository, and you can make any changes without affecting the original repository.

Why Fork a Repository?

Here are a few common scenarios in which forking is extremely useful:

1. **Contributing to Open Source**: You might come

across an open-source project that you are enthusiastic about and want to contribute to. By forking the repository, you can make changes in your fork and then submit a pull request to the original repository to merge your contributions.

2. **Experimentation**: If you find a project interesting but want to experiment with changes without the risk of damaging the original code, a fork allows you to do this safely.

3. **Using Someone Else's Project as a Starting Point**: Sometimes, instead of starting from scratch, you might find a repository that is close to what you want to achieve. Forking allows you to use it as a starting point for your own project.

4. **Preserving a Project**: If there's a project you rely on and you're worried it might get abandoned or go in a direction you're not happy with, forking ensures you have a copy that you can maintain yourself.

How to Fork a Repository?

Forking a repository on GitHub is a simple process:

1. **Go to the Repository**: Navigate to the repository you want to fork on GitHub.

2. **Click the Fork Button**: In the top-right corner of the page, you will see a button labeled "Fork." Click this button. GitHub will create a copy of the repository in your account.

3. **Clone Your Fork**: You can now clone this forked repository to your local machine and work on it just like any other repository.

What About Updates to the Original Repository?

One common question is, "What happens if the original repository gets updated? How do I get those updates into my fork?" You can sync your fork with the original repository to pull in updates. This is done by adding the original repository as a remote and then pulling changes from this remote into your fork.

To add the original repository as a remote, run the following command in your forked repository:

```
$ git remote add upstream <original repos\
itory URL>
```

You can then pull changes from the original repository by running the following command:

```
$ git pull upstream main
```

This will pull changes from the original repository's main branch into your forked repository. You can also pull changes from other branches by replacing `main` with the name of the branch you want to pull from.

We used "upstream" as the name of the remote in the above example instead of the default "origin." This is a common convention, but you can use any name you want.

Can I Fork My Own Repository?

Yes, it is possible to fork your own repositories, which might be useful if you want to create a separate line of development.

Forking is a powerful feature for code sharing and collaboration, particularly in open-source projects. It allows you to freely experiment with projects, contribute to them, or adapt them to your own needs. However, before you

fork a repository, make sure you understand the license under which it is released. Some licenses might not allow you to fork a repository, or they might require you to release your fork under the same license, or not use it for commercial purposes. So make sure you understand the license before you fork a repository. You'll usually find the license information in the repository's README file or in a file called LICENSE. We'll discuss licenses in more detail in a later chapter.

As you work with forks, you'll develop a deeper understanding of how collaborative development flows operate, particularly on GitHub.

Pull Requests

Pull requests are an essential part of the collaborative development process, especially on GitHub. They are a way to propose changes from one repository to another, typically from a fork back to the original repository, or between branches in the same repository. They not only suggest changes but also provide a way to review and discuss these proposed changes before they are integrated.

Understanding Pull Requests

In the simplest terms, a pull request is a proposal to merge a set of changes from one branch to another. It is a request for the maintainer of the project to 'pull' your changes into their branch.

Pull requests show the differences between the content in the two branches. The changes are presented in a diff format, which shows what has been added or removed from files.

Components of a Pull Request

A pull request generally contains the following elements:

1. **Title and Description**: The title is a brief summary of changes, while the description is a more detailed explanation. It is a good practice to provide as much detail as possible about the changes, why you made them, and how they work. This helps the project maintainer understand your intentions and the context of your changes.

2. **Commits**: All the commits that are included in the branch and thus in the pull request. Each commit has a message explaining what it does, making it easier for others to follow your work.

3. **Changes or 'diff'**: This is the detailed view of the changes you've made to the code, shown in a before (red) and after (green) format. This view makes it clear what has been removed, what has been added, and what has been modified.

4. **Comments**: Pull requests also have a section for discussion. Project maintainers or collaborators can leave comments on the pull request or specific lines of code in the diff. This is a great way to ask questions, suggest changes, or discuss potential issues.

Raising a Pull Request on GitHub

Here's a step-by-step guide to raising a pull request on GitHub:

1. **Fork and Clone the Repository (if contributing to someone else's project)**: If you are contributing to a project that you do not own, you'll need to fork the repository to your GitHub account. After forking, clone the forked repository to your local machine.

2. **Create a New Branch**: Before making any changes, it's a good practice to create a new branch in your local repository. This keeps your changes organized and separated from the main branch. You can create a new branch using the `git checkout` command.

3. **Make Your Changes**: Make the changes to the code in your local repository. Be sure to make small, focused commits with informative commit messages that explain what each change does.

4. **Push Your Branch to the Remote Repository**: Once you are satisfied with the changes, push your branch to the remote repository (your fork, if you are contributing to someone else's project) by using the `git push` command.

5. **Create a Pull Request**: Go to the GitHub page of your forked repository and click on "New pull request." Select the branch you pushed from the "compare" dropdown. Review the changes and ensure they are what you intend to submit.

6. **Add Title and Description**: It is important to provide a meaningful title and a detailed description. If there are any related issues, reference them in the description.

7. **Submit the Pull Request**: After reviewing your changes and providing the necessary information, click on "Create pull request" to submit it.

8. **Address Review Comments**: The maintainers of the repository or other contributors may provide feedback or ask for changes. Be sure to address these comments. You can make additional changes to your branch and push them; they will automatically become part of the existing pull request.

9. **Wait for Approval and Merging**: Once all feedback has been addressed, wait for the maintainers to review and approve your pull request. They may merge it directly, or they may ask for further changes.

10. **Clean Up After Merge (Optional)**: After your pull request is merged, you might want to delete the branch you created for it, both locally and on the remote repository. This keeps your repository clean and organized.

These steps should help you create and submit a pull request to either a project you are contributing to or even to your projects when working with branches.

Best Practices for Pull Requests

Here are some best practices to follow when creating a pull request:

- **Make small, single-purpose pull requests**: This makes your changes easier to understand and review. A giant pull request with multiple changes across various aspects of the project is harder to review and has a higher chance of conflicts.

- **Write a good description**: The more information you provide, the easier it is for the reviewer to understand your changes. Explain what you did, why you did it, and any other relevant details.

- **Link to any relevant issues**: If your pull request is solving an open issue in the repository, make sure to link it in your description. This helps keep project management organized. In GitHub, **Issues** function as a way to track tasks, enhancements or bugs associated with a project. We'll discuss them in the next section.

- **Keep your branch up to date with the main branch**: If changes have been made to the main branch after you created your branch, make sure to pull these changes into your branch before submitting the pull request. This reduces the chance of merge conflicts.

Common Issues and Recommendations

One of the most common issues with pull requests is merge conflicts. These occur when the same part of the code has been modified in both your branch and the target branch. GitHub will alert you of the conflict, and you will have to resolve it manually.

Another common issue is submitting a pull request too early. It's best to fully test your changes and ensure

everything works as expected before requesting others to review your code.

Finally, be patient and open to feedback. Especially in open-source projects, it might take time for maintainers to review your pull request. They might also have suggestions for improvements. Remember, the goal is to improve the project, and constructive feedback is a part of the process.

> Pull requests are the heart of collaboration on GitHub. Understanding how to create and manage them effectively will make you a better collaborator and open-source contributor. They are a way to contribute to the projects you care about, improve your coding skills, and collaborate with others in the developer community.

Issues

Now that we have a good understanding of pull requests and their importance in proposing and reviewing changes, let's discuss another essential aspect of project collaboration on GitHub - Issues.

Issues are a great way to keep track of tasks, enhancements, and bugs for your projects. They act like a conversation thread where you can discuss these items with your team or contributors, assign people to tasks, and keep track of progress.

How to Open an Issue

To open an issue, you can navigate to the main page of the repository and click on the "Issues" tab, then click on the "New Issue" button. Here, you can write a title for your issue and a description. It's good practice to be as clear and descriptive as possible to help others understand the issue.

Labels, Assignees, and Milestones

When creating an issue, you can assign it to one or more people, add it to a milestone, or categorize it using labels.

- **Assignees** are individuals responsible for working on the issue.

- **Labels** help you categorize and filter issues. GitHub starts with a standard set of labels, but you can create your own.

- **Milestones** are a way to group issues and pull requests by goals or timelines. They can be particularly helpful in tracking progress toward a project phase or version release.

Linking Issues with Pull Requests

One of the great things about issues and pull requests is that they can be linked. This means when a pull request that resolves a particular issue gets merged, the issue gets automatically closed, and vice versa.

To link an issue to a pull request, you can use certain keywords followed by the issue number in the pull request description. Keywords include `close`, `closes`, `closed`, `fix`, `fixes`, `fixed`, `resolve`, `resolves`, and `resolved`.

For example, if you have an "issue number 12" and you've created a pull request that solves this issue, in your pull request description, you could write `Fixes #12`. This

will create a reference link in the issue, and once the pull request is merged, issue number 12 will automatically be closed. The issue number is different from the issue title. You'll see the issue number beside the title after you open an issue.

GitHub Issues and Your Workflow

GitHub Issues integrate smoothly with your Git workflow. When you're ready to work on an issue, you might create a new branch with the issue number in the name, make your changes, commit them referencing the issue in your commit message, push the branch and open a pull request.

Using issues in this way helps maintain a clear connection between the discussion about what should be done and the actual changes made in the code. As a result, your project's history becomes not only a record of what changes were made but also why they were made.

By leveraging the combination of issues and pull requests, you can ensure a more organized and efficient workflow, making collaboration on GitHub more effective and enjoyable.

Code Review and Merging on GitHub

After creating issues and pull requests, the next steps in the collaborative development workflow are reviewing the proposed changes and then merging them if they are suitable. Both tasks can be done within GitHub's interface, making the process streamlined and efficient.

Code Review

Code reviews are a critical practice in modern software development, designed to catch bugs, improve code quality, and spread knowledge among team members. On GitHub, code reviews happen in the form of comments on a pull request.

When a pull request is opened, it's common for the maintainers of the project or other collaborators to review the changes. This can involve several steps:

- **Understanding the Changes**: Reviewers start by reading the description of the pull request and looking at the files changed. This gives them an overview of what the pull request is trying to achieve and the means used to achieve it.

- **Reading the Code**: The reviewers then go through the proposed changes line by line, understanding the new code, and making sure it follows the project's coding standards.

- **Leaving Comments**: If the reviewers have questions about a piece of code or if they think something could be improved, they can leave comments. Comments can be about the pull request as a whole or about specific lines of code.

- **Requesting Changes**: If there are parts of the code that need to be fixed before it can be merged, reviewers can mark the pull request as needing changes. The person who opened the pull request can then make those changes and request another review.

- **Approving the Pull Request**: Once the reviewers are satisfied with the changes, they can approve the pull request. This indicates that the pull request is ready to be merged.

Merging on GitHub

Once the code review is done and any necessary changes have been made, it's time to merge the pull request.

Merging is the act of integrating the changes from the pull request's branch into the target branch.

On GitHub, merging a pull request is simple. If there are no conflicts between the pull request's branch and the target branch, you can just click the "Merge pull request" button, and GitHub will do the rest.

When merging, you can choose between three strategies:

- **Merge commit**: This strategy combines all the commits from the source branch into a single merge commit in the target branch. The commit history of the source branch is preserved.

- **Squash and merge**: This strategy squashes all the commits from the source branch into one and then merges that commit into the target branch. This leads to a cleaner commit history but loses the context provided by individual commits.

- **Rebase and merge**: This strategy moves or "rebases" the commits from the source branch onto the tip of the target branch, creating a linear commit history.

The right strategy to use depends on your project and how you want to structure your commit history.

Code reviews and merging are the final steps in the collaborative development workflow on GitHub. They are crucial practices for maintaining high-quality code and a healthy project. By knowing how to review and merge code effectively, you can contribute more efficiently to your projects or any open-source project on GitHub. Remember to be considerate and constructive when reviewing people's code; collaboration is all about helping each other to grow and improve.

7. Best Practices for Git and GitHub

Throughout the journey so far, you've gained a solid understanding of Git and GitHub. You've learned how to make commits, create branches, open pull requests, manage issues and collaborate with others.

Knowing how to use a tool is one thing; using it efficiently is another. As you continue to make progress in your development career, you'll find that adopting best practices is just as important as understanding how to code.

This chapter provides guidelines on how to use Git and GitHub effectively. We'll discuss best practices, such as writing good commit messages, managing branches, organizing your repository, and other key points to remember when working with Git and GitHub.

- 7.4 Things to keep in mind when working with Git and GitHub

Take note that these aren't to be followed blindly; they are guidelines that have evolved based on the experiences of many developers. They might not be a perfect fit for every case or preference, but they're invaluable as a starting ground to shape your code of practice.

Writing Good Commit Messages

Let's now delve into the art of crafting quality commit messages. Good commit messages are crucial to creating an understandable history for your project. They not only document what has changed but also provide context and reasoning behind those changes. In this section, we'll explore why this is important and how to achieve it.

Why Write Good Commit Messages?

Commit messages serve as a snapshot of the changes made in each commit, providing a succinct summary of the modifications. They're not merely about describing what changes were made - they also convey why those changes were necessary. These messages become a crucial communication tool for you and others working on the project, helping understand the rationale behind each commit.

The importance of a clear commit message amplifies when you're working with a team or contributing to open source projects, where people from different backgrounds collaborate. Here, commit messages offer crucial context that helps everyone understand each other's changes, intentions, and the project's evolution.

Guidelines for Good Commit Messages

Let's learn some best practices for writing commit messages that communicate effectively.

Be Clear and Descriptive

A clear, descriptive commit message can save a lot of time when you or others are looking back through the project's history. Compare a vague message like "Updated code" with a descriptive one like "Fixed bug causing application crash when user inputs negative numbers". The second example immediately conveys what the commit is about, reducing the need to dig through the code to understand the changes.

For example, if you've fixed a bug, briefly explain the bug's impact. If it's a new feature, describe what the feature adds to the application. The goal is to provide context and make the commit message meaningful.

Use the Imperative Mood

When it comes to commit messages, the convention is to write in the imperative mood, as if you're giving orders to the codebase to alter its behavior. This style matches with Git's built-in messages for operations like `git merge`.

The imperative mood is a grammatical mood that is used to express commands or requests. So, instead of "added feature X" or "fixed bug Y", use "add feature X" or "fix bug Y". It's a subtle change, but it aligns with the Git language and keeps commit messages consistent.

Limit the Subject Line

The subject line of a commit message (the first line) should ideally be kept under 50 characters. This is not just a random rule; it's about usability. Many tools, including command-line tools and GitHub, are designed around this convention and will truncate longer messages. Keeping messages short ensures that they can be easily read at a glance.

If a commit can't be sufficiently described within this limit, that's a good sign that it probably contains too many changes and should have been broken down into smaller commits.

Use the Body to Explain 'What' and 'Why', Not 'How'

If a commit is more complex and the subject line isn't enough to fully describe it, you can use the body of the message to elaborate. In the body, focus on explaining what you changed and why you made those changes.

Remember, the code tells the story of how something was changed - the commit message should provide context to understand these changes.

Here's a simple example of a well-structured commit message:

```
Add input validation for user registratio\
n form

Previously, the registration form was acc\
epting invalid inputs such as empty email
 fields and weak passwords. This commit i
ntroduces input validation to enhance dat
a integrity and improve user experience.
```

Reference Relevant Issues or Tickets

If your commit pertains to a particular issue or ticket, reference it in the commit message. This is especially helpful in a team setting where you are likely using an issue tracker.

Example:

```
Add error handling for invalid user input

Ensure that the application doesn't crash\
 when an invalid input is
submitted by the user.

Related issues: #45, #67
```

Why These Practices Matter

Codebase Scalability: As a codebase grows, good commit messages help new members understand the code's evolution.

Code Reviews: During code reviews, reviewers can understand the context of changes which leads to more insightful feedback.

Debugging: When you're trying to find the source of an issue, being able to read through the history effectively can often provide insights into where a bug was introduced.

Rollbacks: If a bug is found and you need to do a rollback, good commit messages can help you identify which commit is safe to roll back to.

Good commit messages communicate your intentions. They are especially important in a collaborative environment but can be just as valuable even when you're working on solo projects. Make it a habit to write meaningful commit messages; your teammates (and your future self) will thank you for it.

Managing Branches Effectively

Branching is a fundamental aspect of Git and one that lends itself to a multitude of strategies and approaches. With so much flexibility, how can we ensure that we're managing our branches effectively? Let's explore some guidelines to help maintain an efficient and organized branch structure.

Keep Branches Focused and Short-Lived

Every branch in your repository should have a clear and focused purpose. It could be for adding a new feature, fixing a bug, or even experimenting with some ideas. By keeping branches focused on a single task, you make it easier to understand what changes are contained in each branch and reduce the likelihood of conflicts when it's time to merge.

Short-lived branches are also easier to work with. The longer a branch lives separate from the main line of development (often the `main` or `master` branch), the more likely it is to fall behind the current state of the project and incur merge conflicts. Aim to merge branches back into the main line of development as soon as their purpose has

been fulfilled.

Use Meaningful Branch Names

Branch names should convey the purpose of the branch. While Git doesn't impose any strict rules on naming branches, having a consistent and descriptive naming convention makes it easier for you and others to understand what's going on in a repository.

Common branch naming conventions often include the type of work (feature, bugfix, hotfix, etc.), the issue or task number if available, and a short description. For example, `feature/42-add-login-button` or `bugfix/108-fix-navigation-bar`.

Regularly Update Branches

You should frequently sync your branches with the main branch to prevent them from becoming too far out of date. This can be achieved by either merging the main branch into your branch or rebasing your branch onto the main branch. Regular updates can help catch integration issues early and reduce the scope of merge conflicts.

Delete Merged Branches

Once a branch has been merged and its changes are part of the main line of development, it's usually safe to delete the branch. This keeps the list of branches in your repository clean and manageable.

On GitHub, you can easily delete branches after a pull request has been merged. If you're using the command line, the `git branch -d` command can be used to delete branches that have been merged.

Use Branch Protection Rules (GitHub)

For important branches like `main` or `master`, consider using branch protection rules on GitHub. You'll find these settings under the "Branches" tab in your repository's settings.

These rules can enforce certain policies like requiring pull request reviews before merging, requiring status checks to pass before merging, and preventing force pushes. With branch protection rules, it'll be easier to maintain the quality of code in your primary branches and avoid accidental modification or deletion.

Branch management might seem like a lot to take in, but with consistent practice, it'll soon become second nature.

It's all about staying organized, keeping communication clear, and maintaining the quality and understandability of your project's history.

Repository Organization and READMEs

Maintaining a well-organized repository and crafting a comprehensive README file are important aspects of any successful project. They help ensure that your project is understandable and accessible, not only to you but also to other developers who may want to contribute to or use your project. Let's discuss these two areas in more detail.

Repository Organization

A well-structured repository makes your project easy to understand and navigate. Here are a few recommendations for organizing your repository:

- **Directory Structure**: Keep a logical and intuitive directory structure. This often means grouping related files in directories. A good rule of thumb is that someone should be able to guess the function of a file by its location in the directory structure.

- **Naming Conventions**: Use clear and descriptive names for your files and directories. This helps others understand what each file or directory is for. Avoid

using spaces in your file names as this can sometimes cause issues. Instead, use hyphens (-), underscores (_) or camel casing (myFile.js) to separate words. This usually depends on the language or framework you're using and the conventions that are common in that community.

- **Ignored Files**: Use a `.gitignore` file to specify files or directories that should not be tracked by Git. This is typically used for files that are generated during the execution of your code (like log files or cached data), secret information (like API keys), or dependency folders (like `node_modules` in JavaScript projects). Ignoring these files helps keep your repository clean and prevents them from being accidentally committed. Here's what a `.gitignore` file might look like for a JavaScript project:

```
# Ignore node_modules directory
node_modules/

# Ignore environment-specific files
.env
.env.local
.env.\*.local
```

```
# Ignore dependency lock files

yarn.lock
package-lock.json
```

README Files

The README is one of the most important documents in your repository. It serves as the first point of contact for anyone who stumbles upon your project and as such, should be clear, informative, and inviting. Here are some things you should include in your README:

- **Project Title and Description**: Start with the name of your project and a brief description. What does the project do? Why does it exist?

- **Installation and Usage Instructions**: How can someone get your project running on their local machine? Are there any specific steps they need to take? What commands do they need to run? Providing these instructions removes guesswork and makes it easy for others to use your project.

- **Contribution Guidelines**: If your project is open to contributions, provide clear instructions on how

someone can contribute. What steps do they need to take to submit a contribution? Are there any specific coding standards or conventions they need to follow?

- **License Information**: If your project is open source, include license information to inform others about what they can and cannot do with your code.

Creating a README that covers these areas helps make your project inviting to both users and potential contributors. It shows that you care about your project and the people who interact with it.

The way you organize your repository and craft your README can greatly influence the accessibility and success of your project. Strive for clarity and simplicity, and always consider the perspectives of different users.

Things to Keep in Mind when Working with Git and GitHub

As you continue your journey with Git and GitHub, some key considerations can help ensure you maximize your potential and maintain healthy, effective workflows. Here's a recap of what you should keep in mind:

1. **Commit Often, Push Regularly**: Remember to save your changes frequently by committing often. This helps preserve your progress and lets you go back to different versions of your project whenever necessary. Pushing your changes to GitHub regularly safeguards your work and facilitates collaboration.

2. **Stay Synced**: Ensure you're always working with the most recent version of the project by pulling changes from the remote repository frequently. This is particularly important when collaborating with others to avoid merge conflicts or working on outdated code.

3. **Understand Before You Type**: Before running Git commands, make sure you understand what they do. Git is powerful, but with that power comes the ability to erase your work if not handled correctly. If you're unsure about a command, look it up before using it.

4. **Leverage Branches and Pull Requests**: Use branches

to isolate changes for specific features or issues. Pull Requests are not just for contributing to projects; they can also be used for discussing changes, reviewing code, and maintaining a clean project history.

5. **Write Clear, Descriptive Messages**: Whether you're committing changes or creating a pull request, always provide a clear and concise explanation of what has changed and why. This makes it easier for others to follow along and understand your thought process.

6. **Keep Your Repositories Clean and Organized**: A well-organized repository with a comprehensive README file is inviting and user-friendly. It shows respect for your project and for those who interact with it.

7. **Embrace Collaboration and Feedback**: Git and GitHub are designed for collaboration. Be open to contributions from others, listen to their feedback, and be respectful in your interactions. Collaboration is as much about community as it is about code.

Remember, mastering Git and GitHub is a journey. Don't be discouraged if things seem complicated at first. Keep practicing, stay curious, and don't be afraid to ask for help.

8. Licensing and Open Source

Open source is a term that we've mentioned a few times in this guide, and have also briefly explained. It's time we address it properly. In this chapter, we'll dive into the philosophy behind open source, its benefits, and some well-known open source projects. We'll also look at the licenses that protect the freedoms associated with open-source software and how you can use them to define the ways others can study, modify, and distribute the software projects you create.

What is Open Source?

Open source refers to software whose source code is made available to the public, allowing anyone to view, modify, and distribute the software. This concept is based on collaboration and community engagement, as it allows multiple individuals to work together to improve software and create innovative solutions.

The Philosophy Behind Open Source

Open source is not just a development model; it's also a philosophy. The open-source philosophy believes in the free exchange of ideas and knowledge. It is built on the premise that when developers can read, modify, and distribute source code, the software evolves. Through this community-driven development model, open source fosters innovation and creates more robust and reliable software.

Origin and Adoption of the Term "Open Source"

The term "Open Source" was proposed in 1998 as an alternative to "Free Software," a movement started in

the mid-1980s by Richard Stallman. The term "Free Software" sometimes led to misunderstandings, as it was often assumed to mean free of cost. A group, including Eric S. Raymond and Bruce Perens, convened to create a term that emphasized the practical benefits of collaborative development. They settled on "Open Source" and established the Open Source Initiative (OSI) to promote and manage this new approach.

It is also important to mention Linus Torvalds, who is another key figure in the open-source movement. He created the Linux kernel in 1991, which eventually became a part of the GNU Project initiated by Richard Stallman. The Linux kernel, along with various GNU tools, forms the GNU/Linux operating system, which is one of the most prominent examples of open-source software.

Open Source Initiative and The Key Tenets

The Open Source Initiative was formed to steward the Open Source Definition (OSD)[1], a document that specifies what criteria a software license must meet to be considered open source. This definition was based on the Debian Free Software Guidelines, with contributions from the

[1] https://opensource.org/osd/

Free Software Movement. The OSD ensures that software can be freely accessed, used, modified, and shared.

The key tenets of open source, as outlined in the OSD, include:

1. **Free Redistribution**: The software can be freely shared and distributed by anyone.

2. **Source Code Availability**: The source code must be available and provided with the distributed software.

3. **Derived Works**: Users must be allowed to modify the software and distribute it as their own derived work.

4. **Integrity of The Author's Source Code**: If a user modifies the source code and distributes it, they may be required to change the name or version of the derived work to protect the integrity of the original software.

5. **No Discrimination Against Persons or Groups**: The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor**: The license must not restrict anyone from using the

software in a specific field of endeavor. For example, it should not restrict the software from being used in a business.

7. **Distribution of License**: The rights attached to the software must apply to all to whom the software is redistributed without the need for execution of an additional license.

8. **License Must Not Be Specific to a Product**: The rights attached to the software should not depend on the software being part of a particular software distribution.

9. **License Must Not Restrict Other Software**: The license must not place restrictions on other software that is distributed along with the licensed software.

10. **License Must Be Technology-Neutral**: The license should not be based on a specific technology or style of interface.

These tenets form the philosophical foundation of open source, ensuring that it remains a collaborative and inclusive field. They represent values such as collaboration, transparency, and community engagement, which have been crucial to the success and adoption of open source software worldwide.

Benefits of Open Source

1. **Collaboration and Community Contribution**: A vast community of developers can contribute, which leads to diverse perspectives and expertise being utilized.

2. **Transparency**: Everyone can inspect an open source codebase, which builds trust and allows for bugs to be found and fixed more efficiently.

3. **Cost-effective**: Often free, and because it's customizable, businesses can adapt the software without the cost of proprietary software licenses.

4. **Continuous Improvement**: With many developers working on a project, updates and improvements can happen at a rapid pace.

5. **Learning and Skill Improvement**: For budding developers, it's a treasure trove of learning, as they can see how various coding challenges are addressed in real-world projects.

Some Well-Known Open Source Projects

- **Linux**: A family of open-source Unix-like operating systems based on the Linux kernel.
- **Python**: A popular programming language known for its simplicity and versatility.
- **Git**: The version control system we have been learning about.
- **WordPress**: A content management system used for creating websites and blogs.

Now that you have an understanding of what open source means and its core principles, let's delve into how open-source licensing works and how you can participate in or start your own open-source project.

Open Source Licensing

Licensing plays a key role in open-source software. It's the legal instruments through which the freedoms and rules associated with the software are defined. Without an open-source license, the software, even if its source code is publicly available, isn't truly open source – it doesn't guarantee users the rights to study, change, and distribute the software to anyone for any purpose.

There are many open-source licenses available, each with its own rules and obligations. However, they all adhere to the Open Source Definition. Let's explore a few of the most common open-source licenses:

MIT License

The MIT License is one of the most permissive and widespread open-source licenses. It permits users to do almost anything with the code, including using it in proprietary software, provided that the original copyright notice and the license's text is included in all copies or substantial uses of the software.

GNU General Public License (GPL)

The GPL is a copyleft license, meaning it requires derived works to be made available under the same license. This ensures the freedoms granted by the license are preserved in all derivative works. It's worth noting that the GPL has different versions (e.g., GPLv2 and GPLv3), each with different terms and conditions.

Apache License 2.0

The Apache License 2.0 is another popular choice. It's permissive like the MIT License but also provides an express grant of patent rights from contributors to users.

Creative Commons Zero v1.0 Universal (CC0)

CC0 is a public domain dedication from Creative Commons. A work released under CC0 waives all copyright and related rights to the fullest extent allowed by law, essentially making the work as close to public domain as possible.

These are just a few examples of the many licenses available. When creating your open-source projects, it's crucial

to understand the licenses' implications and choose one that aligns with your intentions for the project. GitHub makes it easy to choose a license when creating a new repository, and they provide a handy guide[2] to understand the differences between the licenses.

In the next section, we'll discuss some best practices for contributing to open-source projects. These practices will help you become an effective participant in the open-source community and help you make meaningful contributions.

[2] https://choosealicense.com/

Best Practices for Contributing to Open Source Projects

Contributing to open-source projects can be a rewarding experience that allows you to learn from others, gain practical experience, and improve the technology you use and love. However, contributing effectively requires an understanding of the dynamics and etiquette within the open-source community. Here are some best practices to keep in mind when contributing to open-source projects:

Understand the Project and Its Community

Before diving in and making changes, spend some time understanding the project and its community. Read the project's documentation, check out its issue tracker, and look at the project's history and recent changes. Each open-source community can have its own style, conventions, and norms, and it's important to understand these before contributing.

Follow the Contribution Guidelines

Most open-source projects have contribution guidelines, often found in a CONTRIBUTING file or in the project's README. These guidelines can include how to submit a bug report, how to submit a pull request, the process for reviewing changes, and coding standards and conventions. Make sure to read and follow these guidelines.

Start Small and Choose the Right First Issue

If you are new to a project, it's a good idea to start with smaller issues. Many projects use tags like "good first issue" or "help wanted" to indicate issues that are beginner-friendly. Starting small will help you understand the project's codebase and how the maintainers prefer contributions to be structured.

Communicate Effectively

Communication is key in the open-source community. Be clear and concise in your messages, whether in issues, pull requests, or mailing lists. If you are proposing a new feature or change, explain why it's necessary and how you

plan to implement it. Being proactive in communication shows that you are engaged and respectful of the maintainers' and other contributors' time.

Respect the Maintainers' Decisions

Maintainers are usually very invested in the project, and they might not always agree with the changes you propose. It's important to respect their decisions, even if they decide not to incorporate your contributions. Stay open to feedback and be willing to make revisions as needed.

Test Your Changes

Before submitting your changes, make sure they work as expected and don't introduce new bugs. Add tests if appropriate. Projects often have automated testing set up, and your changes should pass these tests.

Keep Your Forks and Branches in Sync

As you work on your contribution, the main project may continue to evolve. Regularly sync your fork and branch with the upstream project to ensure that your changes can be merged without conflicts.

Be Patient and Persistent

Open source maintainers are often volunteers and may not have time to immediately review your contribution. Be patient and don't get discouraged if it takes a while for them to respond. Also, don't be disheartened by constructive criticism. It's a learning process, and every contribution helps you grow.

Participate Beyond Code Contributions

Contributing to open source isn't just about code. You can also contribute by improving documentation, helping with translations, reporting bugs, and participating in discussions.

Learn From Feedback

Once you've submitted your changes, maintainers or other community members may give you feedback. This feedback is a valuable learning opportunity. Be open to feedback and willing to make changes to your contribution if necessary.

Contributing to open-source projects is a process of continuous learning. Don't be afraid to ask questions or ask for help. The open-source community is generally welcoming and supportive, and everyone was a beginner at some point.

Conclusion

The open-source movement is a testament to the power of collaboration and shared knowledge. It has shaped the software industry in unimaginable ways and continues to be a driving force for innovation. As a contributor, you have the opportunity to be a part of this change.

Remember, open source is not just code. It's a community. It's about mutual respect, collaboration, and helping each other to achieve common goals. By participating in open source, you are joining a global community of individuals who believe in the power of openness and collaboration.

Whether you're contributing to improve your coding skills, to give back to tools you use, or to make a positive impact, your contributions are valuable. Always be open to learning and adapting to the practices and conventions of the community you are joining.

9. Advanced Git Features

Congratulations on making it this far! By now, you've gained a solid understanding of the basics of Git and GitHub. You've mastered the art of commits, branching, merging, and collaborating in open source. But the beauty of Git is its depth - there's always more to learn. In this chapter, we'll dive into some of the more advanced features of Git, such as stashing changes, rebasing and rewriting history, using Git hooks, and tracking changes with Git blame and bisect.

Stashing Changes

Imagine you are in the middle of developing a new feature, and your working directory is full of changes when a critical bug is reported. Now, you need to switch branches to work on a fix. But you can't switch branches with uncommitted changes. You don't want to commit half-done work, and you don't want to lose your changes either. What do you do?

This is where Git stashing comes in. Stashing allows you to temporarily save changes that you have made but do not want to commit yet. It's like putting your changes in a drawer so you can work on something else, and then taking them back out when you're ready. Since you can't switch branches with a "dirty" working directory, stashing is a great way to save your work so that you can come back to it later.

Using Git Stash

To stash your changes, first stage the files you want to stash. Then, run the following command:

```
$ git stash push
```

Or:

```
$ git stash push -m "optional stash messa\
ge"
```

This takes all the modified tracked files and stages them and saves them on a stack of unfinished changes that you can reapply at any time. When you stash your changes, Git resets your working directory to the last commit. At first, it might look like you've lost all your changes, but don't worry! They are still there, and you can get them back whenever you want.

It's important to note that the behavior of the `git stash` or `git stash push` command can vary depending on the version of Git you are using and your particular Git configuration. In many typical scenarios and newer versions of Git, these commands will stash both staged and unstaged changes, meaning you don't necessarily need to stage your changes before stashing.

If you are using an older version of Git (earlier than version 2.35), then the git stash command will only stash staged changes by default. In this case, you will need to stage your changes before stashing.

To set the `stash.default` configuration setting to "stash-all", so that both staged and unstaged changes will be stashed by default, you can run the following command:

```
$ git config stash.default stash-all
```

You can check the default behavior of the git stash command by running the following command:

```
$ git config stash.default
```

```
# output
stash-all
```

You can also check the specific settings in your Git environment that affect the behavior of the git stash command by running the following command:

```
$ git config --list | grep stash
```

```
# output
stash.default=stash-all
```

Managing Your Stashes

Your stashes are saved in a stack, which means the last stash you create is the first one to come out. To see a list of all your stashes, use the following command:

```
$ git stash list
```

This will show you a list of all your stashes, along with a unique identifier for each stash. You can use this identifier to reapply or delete a specific stash. The output will look something like this:

```
stash@{0}: WIP on <branch-name>: <last-co\
mmit-hash> <last-commit-message>
stash@{1}: WIP on feature-x: 50a6b7a add \
number to log
stash@{2}: On feature-x: custom stash mes\
sage
```

The template at the top is a general representation of what each entry in the list might look like:

- `stash@{0}`: This is the stash identifier.
- `WIP on`: This stands for "Work In Progress on" and indicates that the changes were stashed midway.
- `<branch-name>`: The branch where the stash was created.
- `<last-commit-hash>`: The hash of the last commit made on that branch before the stash.

- `<last-commit-message>`: The message of that last commit.

The `stash@{2}` entry shows what a stash message looks like when you provide a custom message (`custom stash message` in this case) during stashing.

Remember, `stash@{0}` represents the latest stash, `stash@{1}`, `stash@{2}`, etc., represent earlier stashes. The stash identifier is what you'll use if you want to apply, drop, or do other operations with a specific stash.

If you want to re-apply the changes from your most recent stash to your working directory, you can use:

```
$ git stash apply
```

Alternatively, if you want to apply a specific stash from the list, you can pass the stash name like this:

```
$ git stash apply stash@{2}
```

Once you have successfully reapplied a stash, you might want to remove it from your stack. To do this, use the `git stash drop` command followed by the stash name:

```
$ git stash drop stash@{2}
```

There's also a handy command that applies the latest stash and immediately drops it from the stack:

```
$ git stash pop
```

`pop` is a combination of `apply` and `drop`. It "pops" the latest stash off the stack to your working directory.

Partial Stashing

Sometimes you might not want to stash all the changes in your working directory. Git allows you to stash just the changes in certain files. Use the following command and list the files you want to stash:

```
$ git stash push file1.txt file2.txt
```

Use-Cases for Stashing

Stashing is particularly useful in scenarios where:

- You need to quickly switch contexts, e.g., moving from a feature branch to a hotfix branch.

- You want to pull the latest changes from a remote repository but have uncommitted local changes.

- You began some experimental changes and want to put them aside temporarily.

While stashing is incredibly useful, it's not a replacement for committing your changes. Stashes are not stored as securely as commits and can be lost. It's best to use stashing as a temporary solution.

Now that you have a good understanding of stashing in Git, you're well-equipped to manage your working directory more effectively. Up next, we'll dive into another advanced Git feature - rebasing.

Rebasing and Rewriting History

As you continue to develop and collaborate on projects using Git, you'll often encounter situations where multiple contributors are working on the same branch or different branches that will eventually be merged. Rebasing is an advanced Git feature that can help you streamline the commit history and simplify complex branching workflows. It's a powerful tool, but with great power comes great responsibility, as it allows you to change the history of your repository. In this section, let's explore what rebasing is, why you might use it, and how to go about it.

What is Rebasing?

Rebasing is a way of integrating changes from one branch into another. If that sounds like merging, you're not wrong. Both merging and rebasing are methods to integrate changes from one branch into another. However, they do it in different ways and the result can be quite different.

In Git, a branch is essentially a pointer to a specific commit. Each commit, in turn, points to its parent commit (the commit that came before it). This forms a chain that

leads back to the initial commit.

When you use `git merge` to integrate changes, Git takes the contents of the feature branch and integrates them with the main branch. This is done by creating a new "merge commit" that points to both the previous commit and the latest commit of the feature branch.

On the other hand, `git rebase` works a little differently. Rather than creating a new commit that brings together the two branches, `git rebase` moves or combines the changes onto the tip of the main branch.

Essentially, it takes the changes made in the feature branch, saves them as temporary files, then reapplies them on top of the main branch, one by one, in the order they were made.

This creates a new commit for each change, giving the impression that the changes were made linearly on top of whatever is present in the main branch.

> Rebasing can be a powerful tool, but it also comes with a big warning: it rewrites commit history. This can make it hard to follow what happened in the history of your project, which is often the whole point of using a version control system! As such, it's generally

> recommended to use rebasing judiciously, particularly when collaborating with others.

```
$ git checkout feature-x
$ git rebase main
```

The commands above take the changes in `feature-x`, and reapply them on top of the changes in `main`.

If conflicts occur between the changes you've made in `feature-x` and the changes made in `main` since you branched off, Git will pause and allow you to resolve those conflicts before continuing.

Rewriting History

Along with rebasing, Git also allows you to alter the commit history in other ways. Using commands such as `git commit --amend` and `git rebase -i` (interactive rebase), you can change commit messages, combine commits, reorder commits, and even remove commits entirely.

Just like rebasing, rewriting history changes the existing commit history. This can be dangerous and is not recommended if you're working with others, as it can

create conflicting histories and make collaboration more difficult.

```
$ git commit --amend -m "New commit messa\
ge"
```

The command above replaces the last commit with a new commit that has the same changes but a new commit message.

```
$ git rebase -i HEAD~3
```

The command above starts an interactive rebase session for the last 3 commits. Git will open a text editor listing the last 3 commits and possible commands. You can then choose to pick, reword, edit, squash, or fix up each commit. When you save and close the editor, Git will apply the actions you chose, altering the last 3 commits as you specified.

Rebasing and rewriting history can be complex, but they also offer precise control over the commit history. As long as you understand their risks and use them carefully, they can be powerful tools in your Git toolbox.

Using Git Hooks

Git hooks are an advanced feature that can make your life much easier by automating tasks in your repository. They are scripts that Git executes before or after events such as `commit`, `push`, and `receive`. These hooks can be used to automate various tasks you might want to perform when these events happen. For example, you might want to run tests before a commit is applied, or send a notification after a push operation.

Git hooks reside in the `hooks` subdirectory of the `.git` directory in every Git repository. By default, this directory contains some sample scripts that are disabled (they have the `.sample` extension). To enable a hook script, you must remove the `.sample` extension.

Let's say you want to ensure all your commit messages follow a certain format. You can create a `commit-msg` hook that checks the message and rejects the commit if it doesn't conform. Here's a basic example:

```
#!/bin/sh

# This script checks if the commit message
# contains a ticket number.

if grep -q "[A-Z]\+-[0-9]\+" "$1"; then
    exit 0
else
    echo "ERROR: Commit msg does not cont\
ain a ticket no."
    exit 1
fi
```

This script uses a regular expression to check if the commit message contains a string that looks like a ticket number (e.g., "PROJ-123"). If it does, the script exits with a status of 0, indicating success, and the commit proceeds. If it doesn't, the script prints an error message, exits with a status of 1, indicating failure, and the commit is aborted.

To use this script as a commit-msg hook:

1. Save it in the file `.git/hooks/commit-msg` in your repository (create the file if it doesn't exist).

2. Make the script executable by running `chmod +x .git/hooks/commit-msg`.

Now, every time you try to commit, this script will check the commit message.

Note: Make sure your hook scripts are always tested and maintained. A broken hook can cause problems with the repositories it's installed in.

As powerful as Git hooks are, remember that they are local to your Git repository and not versioned. This means they won't be shared when you push to or clone a repository. If you need to share automated tasks with others, you'll need to use a different method, such as continuous integration/continuous delivery (CI/CD) pipelines, which are commonly used for tasks like running tests or deploying code.

In the next section, we'll dive into more advanced features to trace changes and debugging in Git. These are essential skills for when things start to go wrong.

Git Blame and Bisect: Tracing Changes and Debugging

When working with code, it's inevitable that bugs will pop up. This can become particularly challenging when you're dealing with a large codebase with many contributors. Two of Git's powerful features, `git blame` and `git bisect`, can help you trace changes and debug more efficiently.

Git Blame

`git blame` is a command that helps you determine who made changes to a file and what those changes were, line by line. This can be particularly useful when you're trying to understand why a line of code exists or who to ask about it.

Let's say you have a file called `my_script.py`, and you want to see the revision history for this file. You can use the following command:

```
$ git blame my_script.py
```

This will print each line of the file, along with the commit

hash, author, timestamp, and the line number in the original commit.

Here's an example output:

```
^fd79278 (Author Name 2023-03-23 12:00:00\
 +0000   1) import os
2da6ef3 (Another Author 2023-04-30 16:00:\
00 +0000 2) import sys
^fd79278 (Author Name 2023-03-23 12:00:00\
 +0000   3)
2da6ef3 (Another Author 2023-04-30 16:00:\
00 +0000 4) def main():
```

While the term '*blame*' might sound accusatory, remember that this tool is not meant to point fingers but to aid understanding and collaboration.

Git Bisect

`git bisect` is another powerful debugging tool in Git, designed to help you find the commit that introduced a bug into your project. It uses a binary search algorithm, allowing you to efficiently pinpoint the problematic commit, even among hundreds or thousands of commits. The binary search algorithm operates by dividing the data set

(in this case, your commits) in half repeatedly until the search is narrowed down to one element.

Here's how you can use `git bisect`:

1. **Start the Bisect Session**:

   To begin, navigate to the root of your Git repository in the terminal. Start a bisect session by executing:

   ```
   $ git bisect start
   ```

2. **Mark the Known Bad Commit**

   Next, you need to tell Git which commit is known to be bad. This is usually the current commit. Run:

   ```
   $ git bisect bad
   ```

   For example, if you know that the bug exists in commit with hash `abc123`, you could also specify it explicitly:

   ```
   $ git bisect bad abc123
   ```

3. **Mark the Known Good Commit**

   Now, specify a commit where you know the code was still working as expected. This should be a commit before the bug was introduced. Use:

```
$ git bisect good <commit-hash>
```

For instance, if commit `def456` is the last known good commit:

```
$ git bisect good def456
```

4. **Test the Code**

   Git will now checkout a commit that's approximately in the middle of the bad and good commits. Compile and test your code. If the commit has the bug, mark it as bad:

   ```
   $ git bisect bad
   ```

   If it doesn't have the bug, mark it as good:

   ```
   $ git bisect good
   ```

5. **Repeat as Necessary**

   Git will continue to checkout commits, narrowing down the range. Keep testing and marking them as good or bad until Git points out the first bad commit.

6. **Ending the Bisect Session**

   Once you've found the problematic commit, you can use `git bisect reset` to end the bisect session and return your HEAD and working directory to the normal state:

```
$ git bisect reset
```

Remember that `git bisect` is most effective when used in conjunction with a good suite of automated tests, which can quickly and reliably tell you whether a commit is good or bad.

These tools, along with the others we've covered, can make managing even large and complex projects much easier and more efficient. It's okay if these tools seem overwhelming at first. You might not need them right away, but it's good to know they're there when you do.

10. Troubleshooting Common Git and GitHub Issues

Now that you've delved into the more complex functionalities of Git and GitHub, it's time to look at how to tackle some common issues you may encounter along the way. We've already addressed a common situation in Chapter 3 - Merge Conflicts. In this chapter, we'll explore other issues that may arise when working with Git and GitHub, such as being in a detached HEAD state, authentication problems, and how to recover lost commits. You'll learn solutions and preventive measures for each issue.

Detached HEAD State

As you venture deeper into Git, you may sometimes come across a message saying you are in a 'detached HEAD' state. This can happen when you check out a commit that is not the latest on its branch. Let's see what this means and how to handle it.

Understanding Detached HEAD

In Git, HEAD is a reference to the currently checked-out snapshot of your project. This is usually the latest commit on the branch you are working on. The term 'detached HEAD' means that the HEAD is pointing directly to a commit instead of a branch.

When you're in a detached HEAD state, you can still make commits. However, since there's no branch referencing those commits, you might lose your work once you switch to a different branch.

How to Get Into a Detached HEAD State

You can end up in a detached HEAD state if you check out to anything other than the latest commit on a branch. This

could be a previous commit, a tag, or a different branch. For example, running `git checkout <commit_hash>` will put you into a detached HEAD state.

How to Fix a Detached HEAD

If you've made some commits in the detached HEAD state and want to keep these changes, the simplest way is to create a new branch while still in the detached HEAD state. You can do this using `git checkout -b <new_branch_name>`. This will create and switch to a new branch that includes your detached commits.

If you haven't made any new commits in the detached HEAD state, or if you don't want to keep your changes, you can simply checkout to an existing branch (like `git checkout main`). This will leave the HEAD detached commits behind, and they will be cleaned up by Git's garbage collection eventually.

Next, let's look at how to handle authentication issues in Git and GitHub.

Authentication Issues

When working with remote repositories, especially on platforms like GitHub, authentication is a key aspect of ensuring that only authorized users have access to certain repositories and operations. While it is crucial for security, authentication can sometimes be a source of issues and frustrations. In this section, we'll explore common authentication issues and how to resolve them.

Understanding Authentication in Git

Authentication is the process of proving who you are. When you interact with a remote repository, especially when pushing code, Git usually requires authentication. There are two main methods of authenticating with a remote repository:

1. **SSH Keys**: Secure Shell (SSH) keys are a pair of cryptographic keys that can be used to authenticate yourself without a password. The private key stays with you and should be kept secret, while the public key can be shared with GitHub or any other Git server.

2. **Username and Password/Token**: This method involves using your username along with a password. However, note that as of August 13, 2021, GitHub no longer accepts account passwords when authenticating Git operations and has instead switched to using personal access tokens.

Common Authentication Issues

Permission Denied

If you try to push to a repository and see a "Permission denied" error, it could mean that your SSH key is not being recognized by the Git server, or you do not have permission to push to the repository.

Solution:

- Double-check that your SSH key is added to your GitHub account (or another Git server you are using).
- Make sure you are a collaborator on the repository or are using an account with sufficient permissions.
- If using SSH, make sure your SSH agent is running and that your key is added to the SSH agent.

HTTPS Repository - Unsupported Credential Request

If you're using HTTPS to interact with a repository and see an "unsupported credential request" error, it likely means you are using a password for authentication, which is no longer supported.

Solution:

- Generate a new personal access token on GitHub and use it as your password when authenticating.

Too Many Authentication Attempts

If you enter the wrong credentials too many times, you may be temporarily locked out.

Solution:

- Wait for a few minutes and then try again with the correct credentials. Make sure to have them at hand to avoid another lockout.

Tips for Avoiding Authentication Issues

- Ensure you're using the correct authentication method. Switching to SSH keys can often be more reliable and secure.
- Regularly check if any tokens or keys have expired or were revoked and need to be renewed.
- Keep your authentication credentials secure.

Now that we've tackled authentication issues, in the next section, we'll explore how to recover lost commits - a life-saver when you accidentally delete or overwrite commits.

Recovering Lost Commits

In the process of working with Git, there may be instances where you might lose commits. This could happen when you run a command that unexpectedly changes your repository's history, or delete a branch that had commits you needed. This can be frustrating, but the good news is that Git provides powerful tools that can help you recover lost commits in most scenarios.

Unreachable Commits and Recovering a Deleted Branch

Unreachable or 'dangling' commits are those commits that are no longer part of any branch or tag due to branch deletion, `git reset`, or other operations. Luckily, Git does not immediately remove these commits and they can usually be recovered.

The `git reflog` command is particularly helpful in these cases. `reflog` stands for 'reference logs', and it records all actions that move the `HEAD` pointer. So, it's a history of everything you've done in your repository.

For example, let's say you've accidentally deleted a branch that had some critical bug fixes. To recover the branch:

```
$ git reflog
```

You will see a list of every change you've made, each with its index, and the most recent operation at the top.

To recover a commit, find its SHA (commit hash) in the list, and then create a new branch with that commit as the head.

```
$ git branch recover-branch <sha-of-commi\
t>
```

Cherry-Picking a Commit

Sometimes, you might want to apply a specific commit from one branch into another without merging the entire branch. This is where the `cherry-pick` command comes in handy. It allows you to apply a commit from one branch to another.

Cherry-picking can be useful when, for example, when you make a hotfix on a release branch and need to apply that fix to the main development branch.

```
$ git checkout mybranch
$ git cherry-pick <commit-to-apply>
```

Resetting a Branch

The `reset` command is a powerful tool that can be used to undo changes to a branch. It can be used to revert commits, unstage files, and even reset the entire branch to a previous commit.

For example, you can use `reset` to undo the last commit on a branch:

```
$ git reset HEAD~1
```

This will undo the last commit and move the `HEAD` pointer to the previous commit. The `--hard` flag can be used to discard any changes in the working directory.

```
$ git reset --hard HEAD~1
```

You can also reset a branch to a specific commit. This can be useful if you want to discard all commits after a certain point. For example, if you want to reset a branch to a commit with the hash `abc123`:

```
$ git reset --hard abc123
```

Quick Tips to Avoid Losing Commits

- Commit early and often! Small, frequent commits are easier to manage and troubleshoot.
- Push to your remote repository regularly. This serves as an offsite backup of your code.
- Avoid using commands that alter commit history on public branches.

Caution and Best Practices

- Before performing operations like resets or force pushes, it's a good idea to make sure you understand the implications. These commands can alter the history of your repository.
- If you are trying to recover lost work, it's often a good idea to create a new branch; this way, you don't accidentally overwrite other changes.

- Remember that the reflog is local to your Git repository; if you're trying to recover changes that were only present on a different clone of the repository, the reflog won't help

Dealing with lost commits can be intimidating, but remember that Git is designed to preserve data integrity, making it difficult to lose committed work.

Git provides the means to turn what could be a catastrophe into only a temporary setback. However, use these powers wisely and again, make sure to practice caution when using commands that can alter your repository's history.

11. Conclusion and Next Steps

What a journey it's been! We've delved into Git and GitHub, starting from the fundamental concepts and gradually making our way to some of the advanced features. Before we talk about the future and where you can go from here, let's take a moment to reflect on what we've learned throughout this book.

Reflecting on Git and GitHub Concepts

Beginning with the Basics

We kicked off our adventure with an introduction to version control systems and an understanding of how critical they are in modern software development. We learned about Git and its capabilities in tracking changes, maintaining history, and facilitating collaboration. Setting up Git and getting to grips with its basic commands was our first practical dive.

Branching, Merging, and Beyond

We discussed the power of branching and merging in Git, allowing us to work on features separately and integrate them seamlessly. You learned how to handle merge conflicts and keep your repositories clean and manageable.

GitHub: Taking Collaboration to the Next Level

GitHub took center stage as we explored how it complements Git. We talked about repositories, collaboration, and the open-source movement. From creating your first repository to managing pull requests and reviewing code, GitHub showed us the power of community and collaboration.

Advanced Features and Troubleshooting

Git, as versatile as it is, comes with a wealth of advanced features. We discovered how to stash changes, rebase, use hooks, and track changes using git blame and bisect. Moreover, understanding how to troubleshoot common issues like detached HEADs and recovering lost commits equipped you with the tools needed to stay calm under pressure.

Best Practices and Contributions

We emphasized the importance of best practices in Git and GitHub, from writing meaningful commit messages

to managing branches effectively. Open source was an area we paid special attention to, understanding licenses and the best ways to contribute to open-source projects.

All Along the Way

Remember, the learning didn't just happen in isolation. Throughout the book, you've been encouraged to try things out, experiment, and get a feel for the tools and concepts. Practical experience is invaluable in cementing this knowledge.

Now, as we stand at the end of this book, it's essential to realize that this is just the beginning of your journey. The world of version control and collaboration is ever-evolving. With the foundation you've built here, you are well-equipped to continue learning and growing.

In the next sections, we'll talk about further resources and suggested learning paths to continue on your journey.

Let's keep the momentum going!

Further Resources and Learning Paths

Congratulations! You've made it to the end of this book, but your journey with Git and GitHub doesn't have to end here. The tech world is ever-evolving and there's always more to learn. To help you continue on your path, here are some resources and learning paths you can follow:

- **NewDev**[1]: NewDev is my learning and collaborative platform for developers that offers a wide range of courses, including Git and GitHub.
- **Git Documentation**[2]: The official Git documentation is an excellent resource. It's highly detailed and can help you understand the finer aspects of Git.
- **GitHub Blog**[3]: This is where GitHub announces new features and provides articles on best practices.

[1] https://wwww.newdev.io
[2] https://git-scm.com/doc
[3] https://github.blog/

Explore Alternative Version Control Systems

While Git is the most popular version control system, it's not the only one. Understanding alternative systems can provide different perspectives and might be useful in certain situations:

- **Mercurial**: Similar to Git but with simpler commands. Some find it easier to understand.
- **Subversion**: It's a centralized version control system. Knowing this might be useful if you work with legacy code.

Contributing to Open Source

The best way to learn is by doing. Contributing to open-source projects allows you to practice your skills in a real-world setting:

- **NewDev**[4]: NewDev also offers several open-source projects that you can contribute to. On NewDev,

[4] https://wwww.newdev.io

you'll find developers of all skill levels working together to build cool, useful projects. Plus, you'll get feedback from experienced developers to help you improve.

- **First Timers Only**[5]: This is a website dedicated to helping newcomers find open-source projects to contribute to.

- **Good First Issues**: GitHub allows project maintainers to tag issues as good for newcomers. Search for these to find something that suits your interests.

Explore Git Integrations

Understand how Git integrates with different development environments and tools:

- **CI/CD Integration**: Learn how Git integrates with CI/CD tools like Jenkins, Travis CI, or GitHub Actions.

- **Integration with IDEs**: Explore how Git can be used efficiently with Integrated Development Environments like IntelliJ, Eclipse, or Visual Studio.

[5] https://www.firsttimersonly.com

Join a Community

Being part of a community can greatly enhance your learning experience:

- **NewDev**[6]: Join the NewDev community to connect with developers, collaborate and share your daily progress updates.
- **Stack Overflow**[7]: A Q&A site where you can ask questions and share your knowledge.
- **Reddit**: Subreddits like r/git[8] and r/github[9] are great communities for Git and GitHub users.

Remember that the key to mastering any skill, including using Git and GitHub, is practice and continual learning. Don't be afraid to experiment, break things and fix them again. That's how you learn.

It has been an absolute pleasure to be your guide on this journey. I would love to hear your feedback about this book. You can find all my contact details on my NewDev profile[10], including my email address and social media handles (You can create your own NewDev profile too!).

[8] https://www.reddit.com/r/git/
[9] https://www.reddit.com/r/github
[10] https://www.newdev.io/ebenezer

Don't hesitate to reach out to me, send me an email or tag me in a tweet (or a post on NewDev) if you have any questions or feedback.

Keep coding, keep collaborating, and keep pushing yourself to new heights. This is just the beginning!

Appendix A: Git Command Cheat Sheet

Here's a quick reference guide to the most commonly used Git commands you might encounter in your day-to-day work. It's designed to be a quick reference guide for when you need a refresher on specific commands or options. Remember not to add the dollar sign ($) when running these commands. It's only there to indicate the command prompt.

Set Up and Configuration:

- **Configure user name**

```
$ git config --global user.name "Your Nam\
e"
```

- **Configure user email**

```
$ git config --global user.email "yourema\
il@example.com"
```

- **Initialize a new repository**

```
$ git init
```

- **Clone an existing repository**

```
$ git clone <repo_url>
```

Staging Changes:

- **Stage a file**

```
$ git add <file>
```

- **Stage all changes**

```
$ git add .
```

Committing Changes:

- **Commit staged changes with a message**

```
$ git commit -m "Your commit message"
```

Viewing History:

- **View commit history**

```
$ git log
```

- **View changes in the working directory**

```
$ git status
```

- **View differences between staged changes and the last commit**

```
$ git diff --staged
```

Branches:

- **Create a new branch**

```
$ git branch <branch_name>
```

- **Switch to a different branch**

```
$ git checkout <branch_name>
```

- **Create and switch to a new branch in one command**

```
$ git checkout -b <branch_name>
```

- **Delete a branch**

```
$ git branch -d <branch_name>
```

- **List all branches**

```
$ git branch
```

Merging:

- **Merge another branch into the current branch**

```
$ git merge <branch_name>
```

Remote Repositories:

- **Add a remote repository**

```
$ git remote add <remote_name> <remote_ur\
l>
```

- **View configured remote repositories**

```
$ git remote -v
```

- **Fetch the latest changes from a remote repository**

```
$ git fetch <remote_name>
```

- **Pull the latest changes from a remote repository**

```
$ git pull <remote_name> <branch_name>
```

- **Push commits to a remote repository**

```
$ git push <remote_name> <branch_name>
```

Stashing:

- **Stash changes in the working directory**

```
$ git stash
```

- **List all stashed changesets**

```
$ git stash list
```

- **Apply the latest stashed changes**

```
$ git stash apply
```

- **Remove the latest stash from the list**

```
$ git stash drop
```

- **Apply and remove the latest stash in one command**

```
$ git stash pop
```

Undoing Changes:

- **Unstage a file**

```
$ git reset <file>
```

- **Revert a commit**

```
$ git revert <commit>
```

- **Reset the repository to a specific commit**

```
$ git reset --hard <commit>
```

Remember, this cheat sheet is just a quick guide. For a deeper understanding of each command

and its available options, always refer back to relevant chapters in this book.

Appendix B: Glossary of Git and GitHub Terms

Here is a collection of some common terms we've encountered throughout this book:

- **Bisect**: A functionality in Git that helps you search through commits to find the one that introduced a bug, using a binary search algorithm.

- **Blame**: A Git command that allows you to see who last modified each line of a file and when, helping track the origin of changes.

- **Branch**: A unique set of code changes with a unique name. Each repository can have one or more branches, allowing you to work on different features isolated from each other.

- **Cherry-picking**: The action of choosing specific commits from one branch and applying them to another. It allows you to selectively apply changes.

- **Clone**: Creating a local copy of a repository that exists on a remote server. This allows you to work on the codebase locally.

- **Collaborator**: A person who has been given access to a repository, enabling them to contribute by making changes and commits to the project.

- **Commit**: A snapshot of your work at a certain point in time in Git. Commits include a unique ID (hash), a message describing the changes, and who made them.

- **Contributor**: Any person who makes changes and commits them to a project.

- **Detached HEAD**: A state in Git where you are not on any branch, you're just working directly with a specific commit.

- **Fetch**: The process in Git of retrieving commits, files, and other data from a remote repository to your local one.

- **Fork**: A personal copy of another user's repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

- **Git**: A distributed version control system, commonly used for tracking changes in source code during software development. It allows multiple developers to work on the same codebase without overwriting each other's changes.

- **Git Hooks**: Custom scripts designed to run when specific events happen in a Git repository. They allow the automation of workflows.

- **GitHub**: A web-based hosting service for Git repositories. It provides a graphical interface for managing repositories and includes additional features like bug tracking, feature requests, task management, and more.

- **Head**: In Git, HEAD is a reference to the last commit in the currently checked-out branch.

- **Hook**: A way to execute custom scripts when certain key events occur in Git. They're used for the automation of workflows.

- **Issue**: A way to track bugs, enhancements, or other requests related to the project. It is part of GitHub's tracking system.

- **Master**: Traditionally the default branch name in Git for the initial branch. However, many projects now use 'main' as the default branch name.

- **Merge**: The action in Git of integrating changes from one branch into another. It keeps the commit history intact.

- **Origin**: The default name Git gives to the server where your repository was originally cloned from.

- **Pull**: A Git command that fetches changes from a remote repository and merges them into the current local branch.

- **Pull Request**: A feature on GitHub that allows developers to propose changes to a repository. It's a way of submitting contributions to a project.

- **Push**: The Git command used to upload local repository content to a remote repository.

- **Rebase**: A Git command that moves or combines a sequence of commits to a new base commit. It's used to maintain a linear project history, but changes commit history.

- **Remote**: In Git, a remote is an alias that stores the URL of your remote repository.

- **Repository**: In Git, a repository, or 'repo' for short, is a central file storage location. It is used to keep track of all changes to files in a project.

- **Stage**: In Git, 'stage' is a step in the process of committing changes, where you add specific changes from the working directory to a staging area in preparation for a commit.

- **Stash**: A command in Git that temporarily stores uncommitted changes to clean the working directory without losing ongoing work.

- **Tag**: In Git, a tag is a reference point to a specific commit, typically used to capture significant points in development, like version release points.

- **Upstream branch**: The branch which was cloned from the remote repository, or the branch that is synced with the remote repository.

Remember, the purpose of this glossary is to serve as a handy reference when you come across a term you don't recognize. Don't worry if you don't have all these terms

memorized right away. With time, they'll will become part of your everyday vocabulary as you use Git and GitHub more frequently.

Index