A GUIDE TO GIT & GITHUB

Created by: Lorenzo Uriel

Table of Contents

Welcome!

About the Author

My name is Lorenzo Uriel, born in 2000 and I am currently a Database Administrator. I have a degree in Information Systems from Anhanguera Educacional and a Postgraduate in Project Management from Instituto Mackenzie.

I am always doing what I can to share my learnings and knowledge with the community, this e-book is a reflection of that.

You can follow me on my networks and see a little more of my work:

- https://linktr.ee/lorenzo_uriel

PDF Ebook

This ebook was created using the ibis tool, created by: <u>Mohamed Said.</u> - https://github.com/themsaid

Ibis is a PHP tool that helps you write e-books in Markdown and then export them to PDF.

It allows you to create your e-book in Light and Dark themes.

A Guide to Getting Started in the Git & GitHub

Who are they? Where do they live? What do they eat? And, best of all, how to install it?

What is Git?

Git is a **version control system** designed to track changes in software projects and coordinate the work of multiple people on them.

Developed by **Linus Torvalds in 2005**, Git stands out for its efficiency, flexibility, and ability to handle projects of any size.

It records changes to the source code, allows multiple development branches to exist simultaneously, and facilitates the merging of code from different contributors.

The main function of Git is to allow multiple people to work on the same code simultaneously, without causing conflicts or data loss.

Git Installation and Initial Configuration

Installing Git

Windows

1. Download Git from: https://git-scm.com/downloads
2. Run the installer and select default options
3. Open Git Bash and verify installation:

```
git --version
```

macOS:

1. Install Git via Homebrew:

```
brew install git
```

2. Verify installation:

```
git --version
```

Linux (Debian-based):

```
sudo apt update && sudo apt install git
git --version
```

Initial Git Configuration

After installing Git, configure your identity:

```
git config --global user.name "Your Name"
git config --global user.email "your-email@example.com"
```

Check your settings with:

```
git config --list
```

These settings ensure your commits are properly attributed.

What is GitHub?

GitHub is a platform for version control and collaboration using Git. It allows multiple developers to work on the same project, track changes, and manage code repositories efficiently.

With GitHub, you can:

- Host and share repositories
- Collaborate with teams using pull requests and code reviews
- Automate workflows with GitHub Actions
- Manage project tasks with Issues and Projects

GitHub Authentications

GitHub no longer supports password authentication for HTTPS Git operations since August 2021. If you tried to connect, you probably received an error.

To resolve this, you need to use one of the currently supported authentication methods. The most common way is to use a personal access token (PAT) or use an SSH key.

SSH

1. Check for Existing SSH Keys

Before generating a new SSH key, check if you already have one on your system

```
ls -al ~/.ssh
```

If you see files like `id_rsa` and `id_rsa.pub`, you already have an SSH key pair. If not, proceed to generate a new one.

2. Generate a New SSH Key

To generate a new SSH key pair, run the following command:

```
ssh-keygen -t ed25519 -C "your_email@gmail.com"
```

- *Replace your_email@gmail.com with the email you use for GitHub.*

If your system doesn't support the `ed25519` algorithm, use `rsa`:

```
ssh-keygen -t rsa -b 4096 -C "your_email@gmail.com"
```

- *Replace your_email@gmail.com with the email you use for GitHub.*

This command creates a new SSH key using the provided email as a label.

3. Save the SSH Key

When prompted to "Enter a file in which to save the key," press Enter to accept the default

location (`~/.ssh/id_ed25519 or ~/.ssh/id_rsa`).

You'll then be asked to enter a passphrase. You can either:

- Enter a secure passphrase (recommended for better security)
- Press Enter to skip creating a passphrase (less secure but more convenient)

4. Add Your SSH Key to the SSH Agent

To manage your SSH key, you need to ensure the SSH agent is running:

```
eval "$(ssh-agent -s)"
```

Now, add your SSH private key to the agent:

```
ssh-add ~/.ssh/id_ed25519
```

- (Or `~/.ssh/id_rsa` if you used RSA.)

5. Add the SSH Key to GitHub

Now you need to add the public key to GitHub:

- Display the public key:

```
cat ~/.ssh/id_ed25519.pub
```

- Copy the output (this is your public key).
- Go to GitHub SSH and GPG Keys.
- Click New SSH key, provide a title (e.g., "My Machine SSH"), and paste your public key into the "Key" field.
- Click Add SSH key.

6. Test Your SSH Connection

Test if the SSH key is working by running:

```
ssh -T git@github.com
```

If it's successful, you'll see a message like:

> Hi lorenzouriel! You've successfully authenticated, but GitHub does not
> provide shell access.

7. Change Your Git Remote URL to Use SSH

Now update your Git remote to use SSH:

```
git remote set-url origin git@github.com:lorenzouriel/ebook-git-
github.git
```

8. Push Your Changes

Now try pushing your changes:

```
git push -u origin main
```

This should work without asking for your username or password.

Now you can start working with Git & GitHub!

A Guide to Fundamental Concepts

Without understanding the fundamentals, its history and what it makes up. It doesn't make sense for a man to continue walking the path.

Okay, I didn't need to say that catchphrase. But we'll focus on this chapter as a general introduction to what constitutes Git & GitHub.

Introduction to Repositories and Concepts

A **repository** is where a project's files, history, and changes are stored. It can be:

- **Remote:** Hosted on platforms like GitHub, GitLab, or Bitbucket.
- **Local:** Stored directly on the developer's machine.

Main Concepts

Branches

Branches allow developers to work on different features or fixes without affecting the main codebase. The primary branch is usually called **main**, but you can create new branches for specific tasks.

- **Example:** Suppose you're working on **task 13**. You create a branch called **dev-task-13**, make changes, commit them, and then merge them back into **main** once the work is complete. This keeps your main branch stable.

Commit

A **commit** captures the current state of the project, like a snapshot of all files. Each commit has:

- A unique identifier (`hash`).
- A descriptive message explaining the change.

Merge

A **merge** combines changes from one branch into another, usually integrating a feature branch back into **main**.

Issues

In GitHub, **issues** help track tasks, bugs, or feature requests, providing a structured way to manage development work.

Repository Structure

HEAD

HEAD is a pointer to the most recent commit in the current branch. When you make a new commit, HEAD updates to point to it.

Working Tree

The **Working Tree** contains your project's files, where you modify and create content.

Index (Staging Area)

The **index** is an intermediate storage area where Git tracks changes before committing. To add files to the index:

```
git add file.txt
```

Main Commands

git commit

Commits the staged changes, creating a new version in the repository.

Commit Workflow:

1. Check the status of modified files:

```
git status
```

2. Add files to the commit:

```
git add file1.txt file2.js

# or add all files

git add .
```

3. Create a commit with a message:

```
git commit -m "First Commit"
```

-   The `-m` flag allows you to specify a message

git push

Uploads local changes to the remote repository.

Push Workflow:

1. Push changes to the remote repository

```
git push
```

2. If this is the first push to a new remote branch:

```
git push -u origin main
```

- The `-u` flag sets up tracking for future pushes and pulls.

`git pull`

Fetches and merges changes from the remote repository into the local branch.

Pull Workflow:

1. Get the latest updates:

```
git pull origin main
```

2. If tracking is already set up:

```
git pull
```

`git tag`

A `tag` marks a specific commit in Git history, commonly used for versioning.

Tag Workflow:

1. List existing tags:

```
git tag
```

2. Create a new annotated tag:

```
git tag -a v1.0 -m "Version 1.0"
```

- `-a` creates an annotated tag.
- `v1.0` is the tag name.
- `-m` specifies a message.

3. Push the tag to the remote repository:

```
git push origin v1.0
```

git init

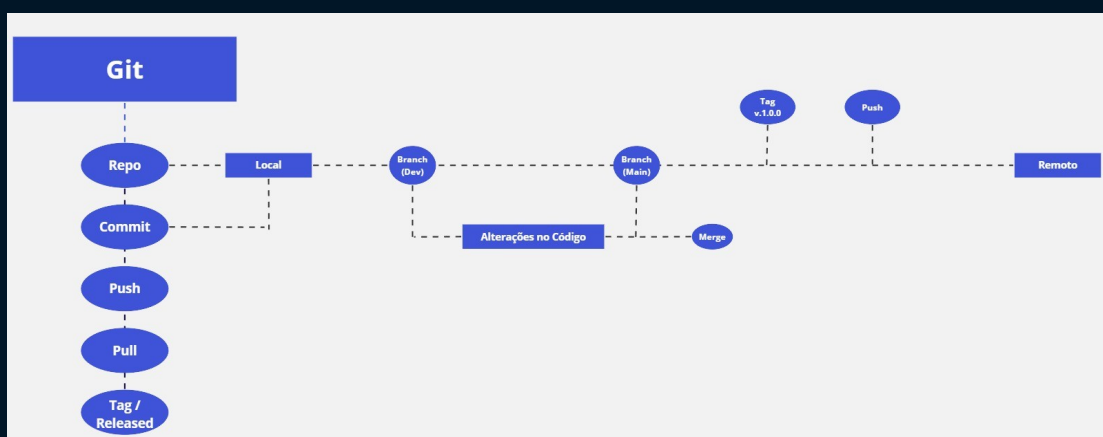Initializes a new Git repository in the current directory.

```
git init
```

git clone

Creates a local copy of a remote repository.

```
git clone https://github.com/lorenzouriel/ebook-git-github.git
```

Architecture, Concepts and Commands:

-

Creating a Local and Remote Repo

1. Go to your GitHub and create a new repository:

   -

2. Navigate to the local directory where you want to create the repository

```
cd c:\path\vagrant
```

3. This command creates a file called README.md and inserts the text #up-website-with-vagrant into it. The >> is a redirection operator that appends the text to the end of the file, or creates the file if it doesn't exist.

```
echo "# up-website-with-vagrant" >> README.md
```

4. Initializes a new Git repository in the current directory.

```
git init
```

5. Adds the file README.md to the index. This prepares the file to be included in the next commit.

```
git add README.md
```

6. Adds all changes (if any)

```
git add .
```

7. Creates the first commit in the repository with a message. The -m allows you to add the commit message directly on the command line.

```
git commit -m "first commit"
```

8. Renames the repository's default branch to `main`. This command is used to update the main branch name to follow the latest naming practices, replacing the old master.

```
git branch -M main
```

9. Adds a remote repository named "`origin`". The term "`origin`" is a standard term used to refer to the main remote repository. The URL is the repository's address on GitHub.

```
git remote add origin
https://github.com/lorenzouriel/up-website-with-vagrant.git
```

10. Pushes the local repository to the remote repository ("`origin`") on the main branch. The -u establishes a tracking relationship, automatically associating the local branch with the remote branch. This is useful for future git pull and git push without having to specify the branch.

```
git push -u origin main
```

We can check our remote repository:

-

What is Versioning and Tags?

Have you ever worked with version releases?

Or with Tags in Git?

Tags are very important in our projects - with them we can identify at what point in time the main releases and versions occurred.

It is a way to organize and document your work.

But when you add tags, will you know the difference between each number?

I'll explain the difference in a simple and practical way:

- **Major:** These are changes that are incompatible with previous versions (Did you restructure everything? Add +1 - v2.0.0)
- **Minor:** These are important changes that are compatible with the previous version (Did you add a new feature? Add +1 v2.1.0)
- **Correction:** Errors and bugs that do not affect the version (Did you find a bug and fix it? Add +1 v2.1.1)
- **Build:** This is an internal control of Git and versioning, it is not necessarily visible when you specify the version.

Example:

-

Introduction to Markdown

Markdown is a lightweight markup language used to format text in a simple and easy-to-read way.

Created by John Gruber and Aaron Swartz in 2004, its goal is to be readable in plain text while allowing automatic conversion to HTML. Markdown is widely used for creating documents, writing blog posts, and producing `README.md` in code repositories.

Headings:

- Headings are created using the `#` symbol. The number of `#` indicates the level of the heading.

```
# Heading Level 1
## Heading Level 2
### Heading Level 3
```

Lists:

- Markdown supports both ordered and unordered lists.

```
- Item 1
- Item 2
- Subitem 2.1
- Subitem 2.2

1. First item
2. Second item
1. Subitem 2.1
2. Subitem 2.2
```

Links:

- To create links, use square brackets for the link text and parentheses for the URL.

```
[Link to my website](https://www.example.com)
```

Images:

- Images are inserted similarly to links, but with an exclamation point ! before them.

```
![Project Logo](images/logo.jpg)
```

Emphasis (bold and italics):

- To create emphasis in text, you can use asterisks or underscores.

```
This is an **amazing project** that uses _modern technologies_.
```

Quotes:

- To create a quote, use the > symbol.

```
> "Be the best version of yourself." - Einstein
```

Tables:

- Tables can be created using | to separate columns and - to create the header row.

```
| Name | Role |
|-----------|-----------------|
| John | Developer |
| Mary | Designer |
```

Code:

- To include code blocks, use three backticks before and after the block. You can specify the programming language for syntax highlighting.
- ❌

So far we understand what makes up Git and how to start your first project, let's delve deeper into what we've seen so far!

A Guide to Work with Branches

28

So, you already know what a branch is and why you should have one. But now, how to work with them?

Let's see!

What is a Git Branch?

A Git branch is an independent line of development within a repository. Think of it as a separate workspace where you can make changes without affecting the main (or default) branch. Branches allow developers to work on new features, bug fixes, or experiments without disrupting the actual version of the project.

Why Use Branches?

Branches provide several benefits in a development workflow:

- **Isolation:** Keep different tasks (features, bug fixes, experiments) separate.
- **Collaboration:** Multiple developers can work on different branches without interfering with each other.
- **Safe Experimentation:** Test changes without affecting the production code.
- **Version Control:** Easily roll back or switch between different versions of the project.

In Git, the default branch is usually called main or master. However, new branches can be created for various purposes, such as:

- **Feature branches (`feature/new-ui`):** Used for developing new functionalities.
- **Bugfix branches (`hotfix/login-fix`):** Used for fixing issues in production.
- **Release branches (`release/v1.2.0`):** Used to prepare stable versions for deployment.

Creating and Managing Local Branches

Branches in Git allow you to work on new features, bug fixes, or experiments without affecting the main codebase. Here's how to create, switch, rename, and delete branches efficiently.

1. Create a New Branch

```
git branch feature/create-chapter
```

- This creates a new branch named `feature/create-chapter` but does not switch to it.
- Useful when you need to prepare multiple branches but don't want to switch immediately.

2. Switch to a Branch

```
git checkout feature/create-chapter
```

- This moves you to the specified branch so you can start coding.

Modern Alternative: Since Git 2.23+, use `git switch`:

```
git switch feature/create-chapter
```

It's a cleaner and safer alternative to `checkout`.

3. Create and Switch to a New Branch (Shortcut)

```
git checkout -b feature/create-chapter
```

- Creates a new branch and switches to it immediately.
- Saves time by combining two commands into one.

Modern Alternative:

```
git switch -c feature/create-chapter
```

More intuitive and recommended for newer Git versions.

4. List All Branches

```
git branch
```

- Displays all local branches. The current branch is marked with *.

```
* dev/ebook-v2
  main
  feature/add-search
  bugfix/fix-typo
```

- This helps you track available branches and navigate between them.

To list remote branches as well:

```
git branch -a
```

5. Rename a Branch

```
git branch -m feature/create-chapter feature/create-chapter-branch
```

- Renames the branch `feature/create-chapter` to `feature/create-chapter-branch`.

If you're already on the branch you want to rename:

```
git branch -m new-branch-name
```

6. Delete a Local Branch (Safely)

```
git branch -d feature/create-chapter-branch
```

- Deletes the branch only if it's already merged into another branch.
- If it has unmerged changes, Git will prevent deletion to avoid data loss.

7. Force Delete a Branch (Dangerous)

```
git branch -D feature/create-chapter-branch
```

- This deletes the branch unconditionally, even if it has unmerged changes.
- Use carefully to avoid losing important work.

Best Practices for Managing Branches

1. Use clear and descriptive branch names (`feature/add-login`, `bugfix/fix-typo`).
2. Delete old branches to keep your repository clean.
3. Use `-d` instead of `-D` unless you're sure you want to force-delete.
4. Regularly push branches to remote (`git push origin branch-name`) to prevent accidental loss.

Merging Branches

In Git, `merge` **is the process of integrating changes from one branch into another.** It is commonly used to combine updates from different development branches into a main branch, like `main` or `develop`.

1. Merge a Branch into the Current Branch

To merge changes from another branch into your current branch:

```
git merge feature/create-chapter-branch
```

- This integrates the changes from feature/create-chapter-branch into the branch you are currently on.
- If no conflicts exist, Git will automatically complete the merge.

Ensure you are on the correct branch before merging:

```
git checkout main  # or git switch main
git merge feature/create-chapter-branch
```

2. Resolving Merge Conflicts

During a merge, if there are conflicts, Git will pause the process and inform you of the conflicting files. Edit these files to resolve the conflicts, marked with:

```
<<<<<<< HEAD
(changes in the current branch)
=======
(changes in the branch being merged)
>>>>>>> main
```

- The HEAD section represents changes from your current branch.
- The section below ======= comes from the branch being merged.

Steps to Resolve Conflicts:

1. Open the conflicting file(s) in a text editor.
2. Manually edit and keep the correct version of the code.
3. Mark the files as resolved:

```
git add .
```

4. Complete the merge with a commit:

```
git commit -m "Conflicting files resolved."
```

To abort a merge and return to the previous state:

```
git merge --abort
```

Types of Git Merges

Git supports different merging strategies depending on whether branches have diverged.

1. Fast-Forward Merge (No Divergence)

A fast-forward merge happens when the branch being merged is ahead of the current branch without any changes on the current branch.

Git moves the branch pointer forward instead of creating a merge commit.

Example:

```
git checkout main
git merge feature/create-chapter-branch
```

- This works when `main` has not changed since `feature/create-chapter-branch` was created.

2. Three-Way Merge (Diverging Branches)

A three-way merge occurs when the two branches have different histories and cannot be

fast-forwarded.

Git creates a new merge commit to combine the changes.

```
git checkout main
git merge feature/create-chapter-branch
```

You will see a commit message like:

```
Merge branch 'feature/create-chapter-branch' into main
```

Best Practices for Merging

1. Always pull the latest changes before merging.
2. Test your code after merging to ensure everything works.
3. Use feature branches for development to keep `main` stable.
4. Delete merged branches to keep the repository clean.

Rebasing Branches

Rebasing is a powerful Git feature that **allows you to integrate changes from one branch into another by moving your branch to the latest state of the target branch.** Unlike merging, which creates a new merge commit, **rebasing replays your commits on top of the latest changes, keeping the commit history cleaner.**

When to use rebase?

- To keep a feature branch up to date with the main branch.
- To clean up the commit history before merging.
- To rewrite commit history for better readability.

1. Rebase a Branch onto Another Branch

```
git checkout feature/create-chapter-branch
git rebase main
```

This updates `feature/create-chapter-branch` with the latest commits from `main`, ensuring it is based on the most recent changes.

⚠ *If conflicts occur during rebasing, Git will stop and ask you to resolve them before continuing.*

2. Abort an Ongoing Rebase

If something goes wrong during rebasing and you want to cancel the process, use:

```
git rebase --abort
```

This restores your branch to the state before you started the rebase.

3. Continue an Interrupted Rebase After Resolving Conflicts

When a conflict occurs, Git pauses the rebase and asks you to resolve conflicts manually in the affected files.

After fixing the conflicts, stage the resolved files:

```
git add .
```

Then, continue the rebase:

```
git rebase --continue
```

Git will continue applying the remaining commits. If another conflict occurs, repeat the process until the rebase is complete.

4. Interactive Rebase (modify commit history)

To edit, reorder, squash, or remove commits, use interactive rebase:

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

HEAD~3 means you are interacting with the last 3 commits.

After running this command, Git opens an interactive editor with options like:

- pick → Keep the commit as-is.
- reword → Change the commit message.
- edit → Modify the commit contents.
- squash → Merge commits together.
- drop → Delete a commit.

Example of an interactive rebase editor window:

```
pick 123abc filename3
squash 456def commit
pick 789ghi first commit

# Rebase db14708..ca382f6 onto db14708 (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                     commit's log message, unless -C is used, in which
case
#                     keep only this commit's message; -c is same as -C
but
#                     opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                     to this position in the new commits. The <ref>
is
#                     updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~
.git/rebase-merge/git-rebase-todo [unix] (20:33 16/02/2025)
```

This will squash the second commit into the first one, combining their changes.

Best Practices for Rebasing:

- Always rebase local branches before merging to keep history clean.
- Avoid rebasing shared branches (`main`, `dev`) as it rewrites commit history.
- Use `git rebase --interactive` to rewrite history in an organized way.
- If unsure, create a backup branch before rebasing.

Remote Branches and Branch Tracking

Remote branches are versions of your branches stored on a remote repository (e.g., GitHub, GitLab, Bitbucket). These branches allow collaboration between multiple developers by keeping their local repositories in sync with the remote.

Why use remote branches?

- To collaborate with others by pushing and pulling changes.
- To keep a backup of your work in a central repository.
- To manage different environments (`main`, `dev`, `staging`).

1. List Remote Branches

To see all branches stored in the remote repository:

```
git branch -r
```

- This lists only remote branches, prefixed by `origin/`.

To list both local and remote branches:

```
git branch -a
```

This helps check which branches exist locally and remotely.

2. Create a Branch Tracking a Remote Branch

If you want to work on a remote branch locally, use:

```
git checkout --track origin/dev
```

- This creates a local branch that automatically tracks the remote one.

The return will be:

```
branch 'dev' set up to track 'origin/dev'.
Switched to a new branch 'dev'
```

This is useful when Git doesn't auto-track the branch.

3. Update Remote Branches

Fetching updates from the remote repository ensures you have the latest branch list:

```
git fetch
```

- This downloads remote changes but doesn't apply them to your working directory.

4. Merge Changes From a Remote Branch Into the Current Branch

To pull the latest updates from a remote branch into your local branch:

```
git pull origin main
```

This is equivalent to running:

```
git fetch
git merge origin/main
```

If the branch is already tracking `origin/main`, you can simply run:

```
git pull
```

⚠ *If conflicts occur, resolve them manually and commit the changes.*

5. Push a New Local Branch to the Remote Repository

After creating a new branch locally, push it to the remote repository:

```
git push -u origin dev
```

- The `-u` flag sets up tracking, so future `git pull` and `git push` commands can be run without specifying the remote branch.

6. Delete a Remote Branch

If a remote branch is no longer needed, delete it using:

```
git push origin --delete dev
```

- This removes `origin dev` from the remote repository.

To delete a local reference to the deleted remote branch:

```
git remote prune origin
```

Best Practices for Remote Branch Management

1. Use descriptive branch names (`feature/login`, `bugfix/navbar-issue`).
2. Always fetch before merging to avoid conflicts.
3. Prune deleted branches regularly
4. Don't push work-in-progress branches unless necessary.

Stashing Changes

Git stash **allows you to temporarily save uncommitted changes without committing them.** This is useful when you need to:

- Switch branches without committing incomplete work.
- Keep your working directory clean while pulling changes.
- Save temporary work that you're not ready to commit.

By stashing, Git stores your changes safely so you can retrieve them later.

1. Stash Uncommitted Changes

To stash all modified and staged files:

```
git stash
```

- This removes changes from the working directory and saves them in a stack.

To stash with a custom message:

```
git stash push -m "Fixing ETL"
```

- Helps you identify what each stash contains.

2. List Stashed Changes

To view all saved stashes:

```
git stash list
```

- Each stash entry is labeled as `stash@{n}`, where `n` is its index.

Example output:

```
stash@{0}: On main: Fixing ETL
stash@{1}: On dev: Debugging API issue
```

- The latest stash is always `stash@{0}`.

3. Apply the Latest Stashed Changes

To apply the most recent stash without removing it from the stash list:

```
git stash apply
```

- This restores the stashed changes but keeps them in the stash.

To apply a specific stash:

```
git stash apply --index 2
```

- Replace 2 with the desired stash index.

4. Apply and Remove the Latest Stashed Changes

To restore and remove the most recent stash:

```
git stash pop
```

To pop a specific stash:

```
git stash pop --index 1
```

- This removes `stash@{1}` after applying its changes.

5. Drop a Specific Stash

To delete a specific stash without applying it:

```
git stash drop --q 1
```

- Removes `stash@{1}` from the stash list.

6. Clear all Stashes

To delete all stashes at once:

```
git stash clear
```

- Use with caution—this permanently deletes all stashed changes.

7. Create a New Branch From a Stash

To create a branch with the stashed changes:

```
git stash branch feature-fix
```

- This creates a new `feature-fix` branch from the most recent stash and applies the stash.

Best Practices for Stashing

1. Use meaningful stash messages.
2. Apply stash before switching branches if necessary.
3. Clear old stashes to keep your repository clean.
4. Use stash branches for complex changes.

Workflow with Branches (Feature Branch, Hotfix, and Release)

This guide outlines the typical workflow using Git branches for features, hotfixes, and releases. It ensures that development, bug fixes, and release processes are streamlined.

1. Feature Branch

Feature branches allow you to work on new features independently without affecting the main codebase.

- Create a new branch for the feature:

```
git checkout -b feature/create-chapter-branch
```

- Work on the feature and make commits, regularly commit your changes as you progress.
- Merge the feature branch into the main branch (`main` or `dev`).

```
git checkout main
git merge feature/create-chapter-branch
```

2. Hotfix Branch

Hotfixes are used for urgent bug fixes in the production environment, typically based on the main branch.

- Create a new branch for the hotfix from the main branch:

```
git checkout -b hotfix/main-app-filter main
```

- Work on the hotfix and commit.
- Merge the hotfix branch into the main branch:

```
git checkout main
git merge hotfix/main-app-filter main
```

- Also merge the hotfix into the dev branch: If you have a dev branch, ensure that the hotfix is merged there too, to keep the development branch up to date.

```
git checkout dev
git merge hotfix/main-app-filter main
```

3. Release Branch

Release branches are created for preparing a new version of the software for production. They allow for testing, final tweaks, and versioning.

- Create a new release branch from the dev branch: The release branch should be based on the dev branch to include all the new features.

```
git checkout -b release/1.0.0 dev
```

- Test and prepare the release, committing as needed. Make final adjustments, perform testing, and commit any necessary changes.
- Merge the release branch into the main branch and tag the release: After testing, merge the release branch into main and tag it with the version number to mark the official release.

```
git checkout main
git merge release/1.0.0
git tag -a v1.0.0 -m "Release 1.0.0"
```

- Merge the release branch back into the dev branch: To ensure any final changes made during the release process are reflected in the development branch, merge the release branch back into dev.

```
git checkout dev
git merge release/1.0.0
```
49

Best Practices for Workflows

1. Use Descriptive Branch Names
2. Keep Branches Short-Lived
3. Commit Frequently and Logically
4. Keep the main Branch Stable
5. Use dev for Ongoing Development
6. Merge Instead of Direct Pushes to main or dev
7. Use Tags for Releases
8. Ensure Proper Testing on Release Branches
9. Clean Up Branches After Merging

Cherry-Picking Commits

Cherry-picking **allows you to selectively apply specific commits from one branch to another.** Instead of merging or rebasing an entire branch, you can pick only the commits you need.

This is useful when:

- A bug fix is in a feature branch and needs to be applied to main.
- A commit from another developer's branch should be added to yours.
- A commit was mistakenly added to the wrong branch and needs to be moved.

1. Apply a Specific Commit From Another Branch

To apply a commit from another branch to your current branch:

```
git cherry-pick a1b2c3d
```

- This applies commit with the hash `a1b2c3d` to your current branch.

2. Apply Multiple Commits

To cherry-pick multiple commits in one command:

```
git cherry-pick a1b2c3d e4f5g6h
```

- This applies both commits in sequence.

3. Abort a cherry-pick Operation

If you encounter a conflict and want to cancel the operation:

```
git cherry-pick --abort
```

- This restores your branch to the state before you started cherry-picking.

4. Cherry-Picking Without Committing

If you want to apply a commit's changes without committing:

```
git cherry-pick -n a1b2c3d
```

- This applies the changes, but doesn't create a commit.
- You can modify the files before committing manually.

5. Cherry-Picking with a Custom Commit Message

To use a custom message instead of the original commit message:

```
git cherry-pick -e a1b2c3d
```

- This opens an editor where you can modify the commit message.

Best Practices for Cherry-Picking

1. Ensure you're on the correct branch before cherry-picking.
2. Use cherry-picking can lead to duplicate commits.
3. Verify commit history after cherry-picking to avoid conflicts.
4. Consider merging or rebasing instead of cherry-picking when appropriate.

Undoing Changes

Sometimes, mistakes happen, and you need to undo changes in Git. Depending on the scenario, you might want to:

- Keep changes staged while undoing a commit.
- Completely erase a commit and its changes.
- Revert a commit while keeping history intact.
- Restore files to a previous state.

1. Undo the Last Commit (Keep Changes Staged)

If you want to undo the last commit but keep the changes staged:

```
git reset --soft HEAD~1
```

- The commit is undone, but changes remain in the staging area.
- Useful when you accidentally committed too soon and want to modify the commit before pushing.

Example:

```
git reset --soft HEAD~1
git commit -m "Updated commit message"
```

2. Undo the Last Commit (Discard Changes)

If you want to undo the last commit and discard all changes:

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

- This completely removes the commit and its changes.

Example:

```
git reset --hard HEAD~1
git log --oneline
```

3. Restore a Deleted File

If you accidentally deleted a file and haven't committed the deletion yet:

```
git checkout -- filename.txt
```

- This restores the file to its last committed state.

4. Reset a File to the Last Committed State

If a file has been modified but not staged and you want to discard the changes:

```
git checkout HEAD -- filename.txt
```

- This restores the file to its last committed version, discarding local modifications.

5. Revert a Commit

If you want to undo a commit but keep history intact:

```
git revert 0a91ea2
```

- This creates a new commit that reverses the changes from the specified commit.
- Unlike `reset`, it does not remove history, making it safer for shared repositories.

6. Undo a Pushed Commit

If you accidentally pushed a commit and need to undo it before others pull it:

```
git reset --hard HEAD~1
git push --force
```

- *git push --force overwrites history and can cause problems if others have already pulled the commit.*

If the commit has already been shared, use revert instead:

```
git revert 0a91ea2
git push
```

7. Undo All Local Changes

If you want to reset everything to the last committed state:

```
git reset --hard HEAD
```

- This removes all uncommitted changes. Use with caution.

Best Practices for Undoing Changes

- Use `reset --soft` when you want to redo a commit but keep your changes staged.
- Use `reset --hard` with caution it deletes changes permanently.
- Use `revert` instead of `reset` when working in a shared repository.
- Always double-check commit history (`git log --oneline`) before making destructive changes.

A Guide to Synchronization and Collaboration

Here is the main point for using GitHub, synchronization and collaboration between teams or personal projects with a friend. Focus on using it this way and you will understand how functional it is.

Setting Up Remote Repositories

Remote repositories are versions of your project hosted on the internet or network, allowing multiple developers to collaborate. They serve as the central hub for storing and sharing project changes.

To work with remote repositories, you need to add, modify, or remove them as needed. Below are the key commands for managing remote repositories:

1. Add a Remote Repository

```
git remote add origin
https://github.com/lorenzouriel/ebook-git-github.git
```

- Replace with the actual URL of your remote repository.

2. Verify Configured Remote Repositories

```
git remote -v
```

- This command lists all configured remote repositories along with their fetch and push URLs.

3. Change the URL of a Remote Repository

```
git remote set-url origin
https://github.com/lorenzouriel/ebook-git-github.git
```

- Useful when changing the remote location, such as migrating to a different provider.

4. Remove a Remote Repository

```
git remote remove origin
```

- This removes the connection to the specified remote repository.

Pushing Changes to the Remote Repository (`git push`)

Once you've made and committed changes locally, you need to push them to the remote repository.

1. Push Changes to the Main Branch

```
git push origin main
```

- Sends the local `main` branch updates to the remote repository.

2. Push a Specific Branch to the Remote Repository

```
git push origin dev
```

- Replace `dev` with the name of the branch you want to push.

3. Push all Local Tags to the Remote Repository

```
git push --tags
```

- Sends all locally created tags to the remote repository.

Getting Changes from the Remote Repository (`git pull` and `git fetch`)

To stay up to date with remote changes, you can pull or fetch updates.

1. Update your Local Repository with the Latest Changes and Merge Automatically

```
git pull main
```

- Equivalent to `git fetch` followed by `git merge`, it downloads and integrates remote changes.

2. Fetch Changes from the Remote Repository Without Merging

```
git fetch
```

- Downloads updates but does not apply them automatically.

3. Merge Fetched Changes Manually

```
git merge
```

- Merges the fetched updates into your local branch.

Removing Local and Remote Repositories

1. Removing a Repository Locally

To delete a Git repository from your local machine, you can remove the directory where it is stored. Be cautious, as this will remove all files and commit history within the repository.

```
# Using Bash (Linux/macOS)
rm -rf /path/to/your/repository

# Using PowerShell (Windows)
Remove-Item -Path "C:\Projects\portfolio\tests_git" -Recurse -Force

git commit -m "Remove tests_git directory"
git push
```
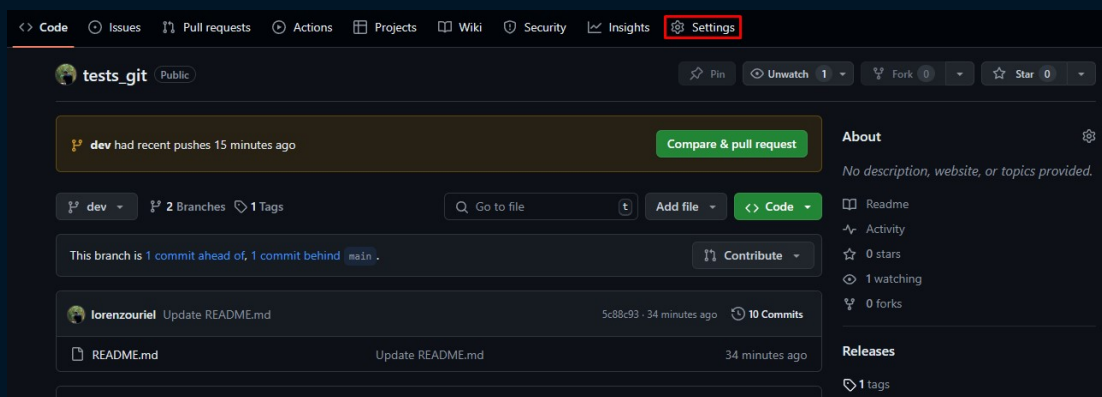
2. Deleting Remote Repositories on GitHub

To delete a remote repository on GitHub, follow these steps:

1. Navigate to the repository page on GitHub.
2. Go to the Settings section of the repository.

3. Scroll down to find the Delete this repository option.

4. Confirm the deletion by entering your password or using GitHub Mobile.

Complete Flow Example

This example demonstrates a full workflow for contributing via a fork and pull request:

1. Clone the Repository

```
git clone https://github.com/lorenzouriel/ebook-git-github.git
git remote add upstream
https://github.com/lorenzouriel/ebook-git-github.git

cd ebook-git-github/
```

2. Create a New Branch, Make Changes, and Commit

```
git checkout -b my-chapter
# Modify files

git add .
git commit -m "Added a new chapter"
```

3. Push Changes to Your Fork

```
git push
```

4. Create a Pull Request via GitHub

- Navigate to the original repository.
- Click "New Pull Request".
- Select your branch (`my-chapter`) and the main branch of the original repository.
- Add a description and submit the pull request.

5. Sync with the Original Repository After a Merge

Update your local repository with the latest changes:

```
git fetch upstream
git checkout main
git merge upstream/main
```

Push the synced main branch back to your fork:

```
git push
```

6. Delete the Merged Branch

```
git branch -d my-chapter
```

- Optionally, delete the remote branch as well:

```
git push origin --delete my-chapter
```

This workflow ensures synchronization and collaboration while contributing to open-source projects.

Working with Tags and Releases

1. Create a Tag

Tags are used to mark specific points in the commit history, such as releases.

```
git tag v1
```

2. Create an Annotated Tag

Annotated tags include additional metadata, such as the tag message, making them ideal for releases.

```
git tag -a v1 -m "Version 01"
```

3. Push a Tag to the Remote Repository

After creating a new tag, push it to the remote repository:

```
git push origin v1
```

4. Push All Local Tags to the Remote Repository

To push all tags at once:

```
git push --tags
```

5. List All Tags

To list all tags in your local repository:

```
git tag
```

6. Delete a Tag Locally

To delete a tag from your local repository:

```
git tag -d v1
```

7. Delete a Tag in the Remote Repository

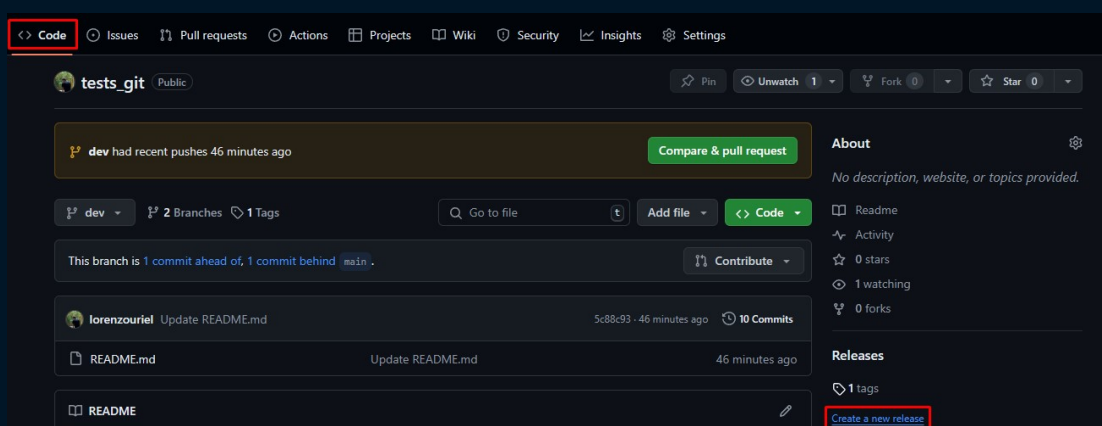To delete a tag from the remote repository:

```
git push origin --delete v1
```

8. Create a Release on GitHub

To create a release on GitHub:

1. Navigate to the Releases section of the repository.
2. Click **Create a New Release**

3. Fill in the release information, associate a tag, and click **Publish a New Release**:

4. Your releases will be organized in the releases section under the **Code** tab:

Working with Forks and Pull Requests

Forking a repository allows you to contribute to a project without modifying the original repository directly. It's a way to contribute with Open-Source projects.

1. Fork a Repository

On GitHub use the web interface to fork the desired repository.

-

2. Clone the Forked Repository

```
git clone
https://github.com/lorenzouriel/opentelemetry-python-contrib.git
```

-   This creates a local copy of your forked repository.

3. Add an Upstream Repository to Sync With the Original Project

```
git remote add upstream
https://github.com/open-telemetry/opentelemetry-python-contrib.git
```

-   The **upstream** repository refers to the original repository from which you forked.

4. Sync your Fork With the Original Repository

Fetch changes from the upstream repository:

```
git fetch upstream
```

-   Merge updates into your main branch:

```
git checkout main
git merge upstream/main
```

68

• Push the updates to your fork:

```
git push origin main
```

5. Create a Pull Request

- On GitHub, **go to the original repository and click "New Pull Request".**
- Compare changes between your fork and the original repository.
- Click **Create Pull Request** and add a descriptive message.
- Once reviewed and approved, the maintainer can merge your contribution.

More About the `git remote` Command

The git remote command is used to manage remote repositories in Git. It allows you to view, add, and remove remotes in your repository.

1. View the Current Remotes

To list all the remote repositories linked to your local repository, use the following command:

```
git remote -v
```

- This will display the URLs of the remote repositories for fetching and pushing.

2. Add a New Remote

To add a new remote repository, use the following command:

```
git remote add repo-name repo-url
```

For example:

```
git remote add new-origin
https://github.com/lorenzouriel/ebook-git-github.git
```

3. Remove a Remote

To remove an existing remote, use the following command:

```
git remote remove repo-name
```

For example, to remove the origin remote:

```
git remote remove new-origin
```

4. Rename a Remote

If you want to rename a remote repository, use:

```
git remote rename old-repo-name new-repo-name
```

For example:

```
git remote rename new-origin upstream
```

5. Change the URL of a Remote

To change the URL of an existing remote:

```
git remote set-url repo-name repo-new-url
```

For example:

```
git remote set-url origin
https://github.com/lorenzouriel/ebook-git-github.git
```

6. Show Information About a Remote

To view detailed information about a remote repository, use:

```
git remote show repo-name
```

For example:

```
git remote show origin
```

This will display detailed information, including the fetch and push URLs, tracking branches, and more.

7. Fetch from a Remote

To fetch updates from a remote repository:

```
git fetch repo-name
```

For example:

```
git fetch origin
```

This retrieves changes from the remote but does not merge them into your current branch.

The idea here was to show some examples and flows that you will repeat while developing your projects.

A Guide to Troubleshooting

If something goes wrong, you need to know how to respond, Git isn't just about the happy path:

- You work
- You make a commit
- Merge the branch
- Deploy

We love the happy path, that's why is happy!

But more often than not, things don't go as planned and we need to know how to react in that cases.

How do you recover lost changes? How do you revert mistakes?

This isn't just about Git, it applies to every project.

That's exactly what I'll cover in this chapter.

Undoing Commits (git revert, git reset, git checkout)

When working with Git, you may need to undo commits for some reasons, such as fixing errors, reverting unexpected changes, or cleaning up history.

Git provides three main commands for undoing commits: `git revert`, `git reset` and `git checkout`. Each has a different behavior and should be used accordingly.

git revert

`git revert` creates a new commit that undoes the changes introduced by a specific commit without modifying the repository history.

This is useful when you want to keep the history clean and not remove commits.

Usage:

```
git revert commit-hash
```

Example:

1. Check the commit you want to revert:

```
git log --oneline

# Return
PS C:\ebooks\git-github-ebook-test> git log --oneline
56c9845 revert
3d87987 herge tag 'v1.0' into dev Showw show
d207431 (tag: v1.0, main) Merge branch 'release/v1.0'
28b146b Yes
0a91ea2 Updated commit message
ca382f6 (origin/main) filename3
db14708 new file
5a26f39 (tag: v1.0.0) first commit
PS C:\ebooks\git-github-ebook-test>
```

2. Copy the hash and run `git revert`:

```
git revert 56c9845

# Return
PS C:\ebooks\git-github-ebook-test> git log --oneline
4134ad2 (HEAD -> dev) Revert "revert"
56c9845 revert
3d87987 herge tag 'v1.0' into dev Showw show
d207431 (tag: v1.0, main) Merge branch 'release/v1.0'
28b146b Yes
0a91ea2 Updated commit message
ca382f6 (origin/main) filename3
db14708 new file
5a26f39 (tag: v1.0.0) first commit
PS C:\ebooks\git-github-ebook-test>
```

- This creates a new commit that reverts the changes from commit 56c9845.

`git reset`

`git reset` moves the HEAD pointer to a previous commit, this is used for modifying commit history. It has three main modes:

Modes of `git reset`:

- **Soft (`--soft`)**: Resets the HEAD to a previous commit but keeps the changes in the staging area, allowing you to modify or amend the commit.
- **Mixed (`--mixed`)**: Resets the HEAD to a previous commit and removes the changes from the staging area, but leaves the changes in the working directory.
- **Hard (`--hard`)**: Completely removes the commit and all changes, both from the staging area and the working directory.

`--soft` Example:

The `--soft` option undoes the last commit, but keeps the changes in the staging area, so you can easily modify the commit or change the commit message. This is useful if you want to redo a commit without losing the work you've done.

You go back to the `git add .` step.

```
git reset --soft HEAD~1
```

- In this example, HEAD~1 moves the HEAD back by one commit, all the changes remain staged, so you can re-commit them after making adjustments.

`--mixed` Example:

The `--mixed` option resets HEAD to the specified commit, removes the changes from the staging area, but leaves them in the working directory. This is the default behavior of `git reset` if no mode is specified.

It is similar to `--soft`, the main difference is that with `--mixed` you go back before `git add ..`

```
git reset --mixed HEAD~1
```

- This command undoes the last commit and unstages the changes, but the changes remain in your working directory, so you can still modify them.

`--hard` Example:

The `--hard` option resets both the staging area and the working directory to match the specified commit. This means all changes are lost and cannot be recovered unless you have backups or use `git reflog`.

Destructive action, caution here!

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

- This will remove the last commit and all the changes associated with it, both in the staging area and the working directory.

`git checkout`

`git checkout` can be used to undo changes in individual files or switch branches. We've talked about him only for branches since now.

`git checkout` is also used for undoing commits, but now `git switch` and `git restore` are recommended for these tasks.

Usage to restore specific files:

`git checkout` can be used to undo changes in individual files and restore them to the last committed state.

```
git checkout -- filename.txt
```

- This command will discard any uncommitted changes in the `filename.txt` file and restore it to its last committed state.

The `--` is used to tell Git that you're referring to a file (not a branch) and you want to discard changes in the working directory, restoring the file to the state of the last commit.

`git restore` Example:

`git restore` is the recommended command for restoring files, and it's more intuitive than using `git checkout` for this purpose.

```
git restore filename.txt
```

- This command will discard any uncommitted changes in the `filename.txt` file and restore it to the version from the last commit, just like `git checkout -- filename.txt`.

Recovering Lost Commits

If you accidentally lose a commit in Git, you can recover it using commands like `git reflog` and `git fsck`. These tools help you identify and restore commits that seem to have been lost.

With sure, it's a type of command that you didn't use a lot, but you need to know that exists.

Recovering a Lost Commit with `git reflog`

`git reflog` records updates to the tip of branches (HEAD), allowing you to see the history of commits and other Git actions, even if the commit is no longer part of any branch or reference.

1. View the recent HEAD changes:

```
git reflog
```

- This command will display a list of recent changes to the HEAD, including commits, merges, rebases, and resets.

2. Identify the commit hash of the lost commit.

```
PS C:\ebooks\git-github-ebook-test> git reflog
4134ad2 (HEAD -> dev) HEAD@{0}: reset: moving to HEAD~1
73ac997 HEAD@{1}: commit: git reset example
4134ad2 (HEAD -> dev) HEAD@{2}: reset: moving to HEAD~1
bed2cc3 HEAD@{3}: commit: git reset example
4134ad2 (HEAD -> dev) HEAD@{4}: revert: Revert "revert"
56c9845 HEAD@{5}: commit: revert
3d87987 HEAD@{6}: merge vv1.0: Merge made by the 'ort' strategy.
28b146b HEAD@{7}: checkout: moving from main to dev
d207431 (tag: vv1.0, main) HEAD@{8}: checkout: moving from main to main
d207431 (tag: vv1.0, main) HEAD@{9}: merge release/v1.0: Merge made by
the 'ort' strategy.
ca382f6 (origin/main) HEAD@{10}: checkout: moving from release/v1.0 to
main
28b146b HEAD@{11}: checkout: moving from dev to release/v1.0
28b146b HEAD@{12}: checkout: moving from feature/feature-article to dev
28b146b HEAD@{13}: checkout: moving from dev to feature/feature-article
28b146b HEAD@{14}: reset: moving to HEAD
```

- Look through the output to find the commit you need to recover. Each entry is associated with a reference, such as HEAD@{2}, and a commit hash.

3. Checkout or reset to the lost commit:

Once you've identified the commit hash, you can either checkout or reset to that commit.

- **Using `git checkout`:** This will move your HEAD to the specific commit, but it doesn't modify your working directory or staging area.

```
git checkout commit-hash
```

- **Using `git reset --hard`:** This will reset your HEAD and working directory to the specific commit, discarding any uncommitted changes.

```
git reset --hard commit-hash
```

Example:

View the reflog to find the lost commit:

```
git reflog
```

Output example:

```
8b3dac5 (HEAD, dev) HEAD@{2}: commit: git checkout
4134ad2 HEAD@{3}: reset: moving to HEAD~1
73ac997 HEAD@{4}: commit: git reset example
4134ad2 HEAD@{5}: reset: moving to HEAD~1
bed2cc3 HEAD@{6}: commit: git reset example
4134ad2 HEAD@{7}: revert: Revert "revert"
56c9845 HEAD@{8}: commit: revert
3d87987 HEAD@{9}: merge vv1.0: Merge made by the 'ort' strategy.
```

Recover the commit by checking it out:

```
git checkout 73ac997
```

Or, if you want to reset your branch to this commit, use:

```
git reset --hard 56c9845
```

If you run `git reflog` again you will see that all the changes are made as a new commit:

```
56c9845 (HEAD) HEAD@{0}: reset: moving to 56c9845
8b3dac5 (dev) HEAD@{1}: reset: moving to 8b3dac5
73ac997 HEAD@{2}: checkout: moving from dev to 73ac997
8b3dac5 (dev) HEAD@{3}: commit: git checkout
4134ad2 HEAD@{4}: reset: moving to HEAD~1
73ac997 HEAD@{5}: commit: git reset example
4134ad2 HEAD@{6}: reset: moving to HEAD~1
bed2cc3 HEAD@{7}: commit: git reset example
4134ad2 HEAD@{8}: revert: Revert "revert"
56c9845 (HEAD) HEAD@{9}: commit: revert
```

Finding Dangling Commits with `git fsck`

If you can't find the commit in the reflog, it might be "dangling" meaning it's not referenced by any branch or tag but still exists in Git's database. You can search for these dangling commits using `git fsck`.

Imagine that exists a lost commit in the repository, you can find with the command `git fsck`. The lost commits could happen when we forgot to merge branches and delete the work.

1. Run `git fsck` to find dangling commits:

```
git fsck --lost-found
```

- This will list objects that are not part of any references, including dangling commits.

2. Inspect the commits:

Look for entries labeled "dangling commit" in the output, and use `git show` to inspect them.

```
git show commit-hash
```

Example:

Find dangling commits with `git fsck`:

```
git fsck --lost-found
```

Example output:

```
dangling commit 047885dbe5a76825fa1780cecb741f76eeb9ec35
dangling commit 18baee9d1b898ee1d82504174672b80d63aafa9a
dangling commit 227a6260d4d24c359aa5d4bcf667326c5cf10724
dangling tree 36e529c29f83cad089c8ddc57b7a1fa329564e87
dangling commit 413e484a74bef6b89674ecc6108f5f4364b9395f
dangling commit 726c4249600cb31b5bc8b52238faf3c3e31b4f68
dangling commit 73ac997c040ffda224e4a387614e2f860a8e0905
dangling commit 74a4b8ab890536d2ab8dacf076eaa64a5b24e486
dangling tree 96fc3be44051489ebc3303a66c07108bbcb563da
dangling commit a2d081673eb8bdd9f8e3159d787050c77116e901
dangling commit bed2cc3d74faff05c84ba21f84b29ab37357af63
dangling commit c2df1cab9005791812dd7efb690f9ce5f799bf70
dangling commit c8a104a4a311e3ee8cc2a696aeae5e4dc46cbeea
dangling commit fcde575ac34212c1187cee90604d35f0f6b00202
```

Inspect the dangling commit:

```
git show 047885dbe5a76825fa1780cecb741f76eeb9ec35
```

The return:

```
commit 047885dbe5a76825fa1780cecb741f76eeb9ec35
Author: Lorenzo Uriel <lorenzouriel394@gmail.com>
Date:   Sun Feb 16 21:01:01 2025 -0300

    Updated commit message

diff --git a/filename.txt b/filename.txt
index 8d1c8b6..91b27f5 100644
--- a/filename.txt
+++ b/filename.txt
@@ -1 +1,2 @@

+Lorenzo Uriel
\ No newline at end of file
```

- Display all the details of the commit, allowing you to decide if it's the commit you
  want to recover.

More About `git status`, `git log`, `git show` and `git diff`

Yes... we already use them above.

Understanding the commands `git status`, `git log`, `git show`, and `git diff` can help you track changes and managing your Git repository in a more smart way.

With these commands you can check the current state of your repository, help you explore commit histories, and allow you to see the differences between various states of your files.

It's like a bunch of query statements.

`git status`

`git status` provides an overview of the current state of the working directory and the staging area. It helps you track which changes have been staged, which are not, and which files are untracked.

```
git status
```

Example output:

```
HEAD detached from 73ac997
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
        modified:   filename.txt
        modified:   filename2.txt
        modified:   filename3.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        filename3 copy.txt

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

- Changes to be committed: These files have been staged and are ready to be committed.
- Untracked files: These files are not being tracked by Git yet.

git log

`git log` shows the commit history of the repository, allowing you to explore previous commits, their messages, authors, and timestamps. You can also use various options to filter and format the log output.

```
git log
```

Example output:

```
commit 71333c330b93ffe20e02e6bac2b9a57ce15e355c (HEAD)
Author: Lorenzo Uriel <lorenzouriel394@gmail.com>
Date:   Mon Mar 10 21:56:23 2025 -0300

    oh yeah

commit 56c9845e9e0ff4298def3b693456f8857d82824b
Author: Lorenzo Uriel <lorenzouriel394@gmail.com>
Date:   Sun Mar 9 21:33:00 2025 -0300

    revert

commit 3d87987f33e66c859aa25f6a950995815c8caf50
Merge: 28b146b d207431
Author: Lorenzo Uriel <lorenzouriel394@gmail.com>
Date:   Wed Mar 5 23:14:00 2025 -0300

    show
```

- Commit hash: The unique identifier for the commit.
- Author: Who made the commit.
- Date: When the commit was made.
- Commit message: A brief description of the changes made in that commit.

You can also use `git log --oneline` for more resumed details:

```
git log --oneline
```

Example output:

```
5b07142 (HEAD) html example for git show
71333c3 oh yeah
56c9845 revert
3d87987 herge tag 'vv1.0' into dev Showw show
d207431 (tag: vv1.0, main) Merge branch 'release/v1.0'
28b146b Yes
0a91ea2 Updated commit message
ca382f6 (origin/main) filename3
db14708 new file
5a26f39 (tag: v1.0.0) first commit
```

git show

`git show` allows you to view detailed information about a specific commit, including the commit message, author, date, and the changes made.

It was the command we use in the topic above.

```
git show 5b07142
```

Example output:

```
commit 5b071427e8369b85a56a5ee6b5388a2e2586306c (HEAD)
Author: Lorenzo Uriel <lorenzouriel394@gmail.com>
Date:   Mon Mar 10 22:02:09 2025 -0300

    html example for git show

diff --git a/README.md b/README.md
index 49d8b13..a71ae21 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,3 @@
  "# git-github-ebook-test"

-git revert test
+git revert testfcfdefd
diff --git a/filename.txt b/filename.txt
index 8d1c8b6..8ebae4c 100644
--- a/filename.txt
+++ b/filename.txt
@@ -1 +1,2 @@

+sdfdfdfadf
\ No newline at end of file
diff --git a/filename2.txt b/filename2.txt
```

- This will show the changes made in the commit (the diff) along with other metadata.

git diff

git diff shows the differences between various states of the repository, such as changes between the working directory and the last commit, between two commits, or between branches.

Compare working directory with the last commit:

```
git diff
```

Example output:

```
diff --git a/index.html b/index.html
index 64b907a..a24d290 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@
 <head>
     <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width, initial-
scale=1.0">
-    <title>My First HTML Page</title>
+    <title>My Second HTML Page</title>
 </head>
 <body>
     <h1>Welcome to My First HTML Page</h1>
```

- This shows the changes between the working directory and the last commit. The +
  sign indicates a line added, and the - sign indicates a line removed.

Compare two commits:

You can also compare the changes between two commits by specifying their hashes:

```
git diff 5b07142 e0f5e31
```

- This shows the differences between the two specified commits.

Example output:

```
diff --git a/index.html b/index.html
index a24d290..685af61 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@
 <head>
     <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width, initial-
scale=1.0">
-    <title>My Second HTML Page</title>
+    <title>My Third Example HTML Page</title>
 </head>
 <body>
     <h1>Welcome to My First HTML Page</h1>
```

Compare branches:

To see differences between branches, use:

```
git diff main dev
```

- This shows the differences between the two branches, helping you see what changes are in one branch but not the other.

Example output:

```
diff --git a/README.md b/README.md
index f3e1357..f70444d 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,3 @@
 "# git-github-ebook-test"
+
+git reset example
\ No newline at end of file
diff --git a/filename3.txt b/filename3.txt
index 8d1c8b6..3ad7dfd 100644
--- a/filename3.txt
+++ b/filename3.txt
@@ -1 +1,2 @@

+Changes on files
\ No newline at end of file
```

More About `git commit --amend`

One day you made a big, beautiful commit message and 5 minutes later you find a code change and need to commit again?

Why not just commit to the same last commit if it's just a small change? `--amend` can help with this.

The `git commit --amend` command is used to modify the most recent commit in Git. It allows you to correct the commit message, add or remove changes from the commit, or even update both the changes and the message in one go.

This command is particularly useful when:

- You realize that the commit message needs to be improved.
- You forgot to include some changes in the commit.

Usage:

```
git commit --amend
```

By default, running this command will open the commit editor to allow you to modify the commit message. You can also pass the `-m` option to directly update the commit message.

Example 1: Modify the Commit Message

If you want to update the message of the last commit, you can use:

```
git commit --amend -m "Updated commit message"
```

- This will replace the message of the last commit with the new one.

Example 2: Add Missed Changes

If you realize that you forgot to stage some changes for the last commit, you can:

1. Stage the changes you missed:

```
git add .
```

2. Amend the commit with the staged changes:

```
git commit --amend
```

- This will open the commit editor again, where you can modify the commit message if desired. If you don't want to change the message, simply save and close the editor.

The amended commit will now include both the original changes and the newly staged changes.

Example 3: Completely Replace the Commit (Message + Changes)

If you want to both change the commit message and the changes, stage the changes first and then run the `--amend` command:

```
git add .
git commit --amend -m "New commit message with updated changes"
```

- This will replace the last commit with the new changes and the new commit message.

`git commit --amend` *rewrites the last commit, so it changes its commit hash. This can cause issues if the commit has already been pushed to a shared repository. It's recommended to only amend commits that haven't been pushed yet, or to use it cautiously if you have already shared your changes.*

A Guide to Best Practices

You need to learn some best practices before making your next commit.

What is the best type of message? Or the best name for the branches?

We always seek to apply best practices in our projects. If you don't already do this, it's time to start.

Even more so when we work as a team, share repositories and have code reviews.

You know that commit that broke all the code and the message only said: "Now go!"? Well, that's what I'm talking about.

So what to do?

What are the best practices when it comes to Git & GitHub?

Clear and Frequent Commits

With sure the most important and the first of the list: Write in a manner that people can understand!

Make small and frequent commits to maintain a clear change history.

Use descriptive and standardized commit messages explaining what was changed and why.

Example:

```
git commit -m "fix: login issue by correcting password validation"
```

Avoid generic commits like only "fix" or "update." Prefer something with more details.

Make small commits, where each one addresses a single change. Stop making commits and writing a whole text explaining what changed, this is terrible when we need to revert.

Tips for Commits Messages

What to write in your commit message?

I know you've had this question, especially when we make several changes and need to specify everything in a single commit.

I follow some practices that I consider good in my projects. They are:

- feat: added support for automatic backup
- fix: adjusted data normalization
- revert: reverted change to the bank schema
- delete: removed table [old_logs]
- update: updated database structure
- config: adjusted database connection in .env
- ci: added migration step in the pipeline
- docs: documented table structure
- test: added tests for ETL functions

The idea is to put at the beginning a description of what the commit is about, followed by a clear detail of what was accomplished.

Frequent commits are always better! Don't wait to do everything at once.

Branch Management

Use branches to manage feature development, deploys, releases and other tasks. Adopt a workflow that is good for your team, such as:

- **Git Flow:** The most common and most popular way to work, divided by principal branches (`dev`, `main`) and secondary (`feature`, `release`, `hotfix`).
- **Trunk-Based Development:** The project is centralized in only one unique branch for all team, the `main` and we have the concept of `feature-branchs` for changes.

Tips for Branches Names

And we always find ourselves with the same question, which title to put in this branch?

Main Branches

The default I see in the industry for main branches is:

- `dev` - development
- `qa` - test
- `main`

Change or Feature Branches

For change and secondary branches always specify the objective and add a descriptive message:

- `feature/add-backup-automation`
- `fix/fix-null-values-in-report`
- `hotfix/fix-production-database-issue`
- `refactor/normalize-user-table`
- `release/1.0.0`

A good tip is relate the task id and the task title, if your backlog supports:

- `feature/task544-add-backup-automation`
- `fix/task17-fix-null-values-in-report`

Rebase

Another good tip when working with branches is the `git rebase` command. You can reorder or edit commits before merging. This helps maintain a linear and readable history:

```
git rebase -i HEAD~3  # Reorder or edit the last three commits
interactively
```

Avoid rebasing shared branches like main or dev to prevent conflicts and confusion for others.

Delete

Regularly delete old or outdated branches to keep the repository clean:

```
git branch -d feature/task544-add-backup-automation  # Delete a merged
branch
git branch -D feature/task544-add-backup-automation  # Force delete an
unmerged branch
```

Use .gitignore

Always configure an appropriate `.gitignore` file for your project to avoid versioning unnecessary files.

Exclude build files, dependencies, credentials and every possible environment configurations.

I recommend you to use the toptal site, he generates the most complete `.gitignore` files for every language. This is a python example generated with toptal.

You can also check the ready-made templates made by github in github/gitignore repository.

Git Hooks

With Git Hooks you can automate rules and standards that need to be applied at various stages of your workflow.

Pre-commit Hooks

Use pre-commit hooks to run automated checks before changes are committed. These hooks can enforce code your pre-defined quality standards, some examples:

- Running linters to catch syntax errors or style.
- Executing unit tests to ensure changes don't break the code.
- Validating commit messages to follow a consistent format.

Here you can detect errors early and prevent the developer from submitting any code that could affect the main project.

Post-receive Hooks

Use post-receive hooks to automate tasks after changes are pushed to a remote repository. These hooks are ideal for:

- Updating staging or production environments automatically.
- Triggering CI/CD pipelines to deploy or test changes.
- Sending notifications to your team about new updates.

Of course, minimizing errors and creating an automatic initial deployment.

A Guide to Workflow Types

What is a workflow? And why should you use one?

That's what I want to explain today. I'm sure you'll start using it today.

A workflow defines how you and your team can collaborate using Git and GitHub. Choosing the right workflow will depend on how your team is structured, the complexity of the project, and your strategies for releasing a new version.

I'll explore the most common workflows and use cases.

Trunk-Based Development

The most used, with sure. You probably just used this and didn't know that has a name.

But how it works?

Developers work directly on the `main` branch or create **short-lived feature branches.** All changes are merged into `main` multiple times a day.

It happens frequent integration that minimizes merge conflicts. With this we can have fast delivery, real-time collaboration and fewer merge conflicts.

Workflow Steps

1. Start work:

```
git checkout -b feature-article
```

2. Make changes and push frequently:

```
git commit -m "Finish Article"
```

3. Merge the branch

```
git checkout main
git merge feature-article
git push origin main
```

Diagram

GitFlow

Instead of having just one principal branch, we always have two, the `main` and the `dev`.

Based on `dev` we can always create another branchs for different purposes, like:

- `feature branches`: Created from `dev`, merged back when complete.
- `release branches`: Created from `dev` when preparing for a release.
- `hotfix branches`: Created from `main` to fix urgent issues and merged into both `main` and `dev`.

The role of the `main` always stores production code and `dev` integrates all feature branches. We develop into `dev` branch to stop using production code.

Gitflow is the best option for CI/CD purposes.

Workflow Steps

1. Initialize GitFlow:

```
git flow init
```

When initialize you will answers a small questionary to start the project:

```
Which branch should be used for bringing forth production releases?
    - dev
    - main
Branch name for production releases: [main]

Which branch should be used for integration of the "next release"?
    - dev
Branch name for "next release" development: [dev]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [v]
Hooks and filters directory? [C:/ebooks/git-github-ebook-
test/.git/hooks]
```

The `git flow init` itself will create all the structure.

2. Start a new feature:

```
git flow feature start feature-article
```

- This action creates a new feature branch based on dev and switches to it

3. Finish the feature:

```
git flow feature finish feature-article
```

- Merges `feature-article` into dev
- Removes the feature branch
- Switches back to dev branch

4. Publish a feature:

```
git flow feature publish feature-article
```

- This command publishes the resource itself to the remote repository for other users.

5. Create a release branch:

```
git flow release start v1.0
```

- It creates a release branch created from the `dev` branch.

6. Finish & deploy the release:

```
git flow release finish 'v1.0'
```

- Merge the `release` branch into `main`
- Tag the `release` with the name.
- Merge the `release` branch into `dev` again
- Then, removes the `release` branch

The prompt message are as follows:

```
C:\ebooks\git-github-ebook-test>git flow release finish 'v1.0'
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Merge made by the 'ort' strategy.
Already on 'main'
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)
Switched to branch 'dev'
Merge made by the 'ort' strategy.
Deleted branch release/v1.0 (was 28b146b).

Summary of actions:
- Release branch 'release/v1.0' has been merged into 'main'
- The release was tagged 'vv1.0'
- Release tag 'vv1.0' has been back-merged into 'dev'
- Release branch 'release/v1.0' has been locally deleted
- You are now on branch 'dev'
```

If you want to know more about the commands, I recommend this Gitflow cheatsheet.

Diagram

Feature Branch

The core is for teams working on multiple features simultaneously, here each feature is developed in a separate branch and feature branches merge back into `main` after review.

The unique process here is create a `feature` branch and the merge to the `main` after a review.

Workflow Steps

1. Create a feature branch:

```
git checkout -b feature-article
```

2. Work and commit changes:

```
git add .
git commit -m "Added Feature Branch Topic"
```

3. Push the branch and open a Pull Request:

```
git push origin feature-article
```

4. After approval, merge into `main` on the remote and local repository.

Diagram

Forking Workflow

Developers fork a repository instead of working directly on it. Recommended and most used on Open-Source projects.

They clone, create a branch, and push their changes to their own fork. After all this, the changes are submitted as a **Pull Request (PR).**

Workflow Steps

1. Fork the repository on GitHub.

2. Clone the forked repository:

```
git clone https://github.com/lorenzouriel/ebook-git-github.git
```

3. Create a branch and make changes:

```
git checkout -b feature-article
git add .
git commit -m "Added feature article"
git push origin feature-article
```

4. Submit a Pull Request to the original repository.

Diagram

By learning this, you will be able to decide which workflow best suits you and your team, remembering that in many cases we already use one without even realizing it.