E-BOOK GIT & GITHUB DIGITAL

THE BASICS TO GET STARTED

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 degree 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: Mohamed Said. - 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.

Getting Started in the Git & GitHub Universe

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.

Main features of Git:

- **Distributed:** Each developer has a complete copy of the repository's history.
- **Performance:** Designed to be fast and efficient, even with large amounts of data.
- **Security:** Ensures the integrity of the code and the changes made.
- **Flexibility:** Supports a variety of workflows and development processes.

Git Installation and Initial Configuration

Step 1: Git Installation

To install Git, follow the steps below according to your operating system:

Windows:

- Download the Git installer from the official website: https://git-scm.com/.
- Run the installer and follow the on-screen instructions, keeping the default settings.

macOS:

- Open Terminal.
- Run the command: `brew install git`. (Requires Homebrew, which can be installed from https://brew.sh/).

Linux:

- Open Terminal.
- Run the appropriate command for your distribution:
- Debian/Ubuntu: `sudo apt-get install git`.

Step 2: Git Initial Configuration

After installing Git, you need to configure it. Use the following commands in the terminal to configure your username and email, which will be used in your commits:

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

You can check your settings with the command:

```
git config --list
```

GitHub Installation and Initial Configuration

GitHub is a source code hosting platform that uses Git for version control. It allows you to collaborate on projects and share code with other developers.

Step 1: Creating a GitHub account

- Go to https://github.com/.
- Click "Sign up" and follow the steps to create a new account.

Step 2: Setting up Git to use GitHub (Windows)

- Launch Git Bash and generate an SSH key for secure authentication:
- In the terminal, run: `ssh-keygen -t rsa -C "youremail@example.com"`.
- Press Enter to accept the default file location.
- Enter a secure password (or leave it blank).
- Add the SSH key to your GitHub:
- Copy the public key that was generated.
- Go to GitHub Settings > "SSH and GPG keys".
- Click "New SSH key", paste the key and save.

The steps are the same for other operating systems, you will need to understand which commands and where they save the SSH keys that were generated.

Step 3: Connecting to GitHub To clone a repository from GitHub, use the command:

```
git clone git@github.com:username/repository-name.git
```

Fundamental Concepts

Introduction to Repositories and Concepts

A repository is the place where the project, its changes and files are stored. It can be local or remote, each with its own specific characteristics:

- **Remote:** Hosted on platforms such as GitHub, other version control systems (CVS) or dedicated servers.
- **Local:** Stored directly on the developer's machine.

Main Concepts

Branches

Branches (or branches) are parallel versions of the project. The main branch is usually called **main**, but it is possible to create new branches to develop specific changes without affecting the main branch.

- **Example:** Create a branch called **dev-task-13**, in this branch you will focus on finishing task 13 of your backlog, when you finish you **commit** and do a **merge** with the main branch. This way, you do not affect your main branch with changes that could break the code.

Commit

A commit is an operation that captures the current state of the project. It's like taking a snapshot of the files at that moment. Each commit has a unique identifier (hash) and a descriptive message.

Merge

A merge is the operation of combining changes from different branches. Typically, you perform a merge when you want to integrate changes from a feature or bug fix branch back into the main branch.

Issues

In the context of Git, an issue refers to a way to track tasks, enhancements, bugs, or discussions related to a specific project.

Repository Structure

HEAD

The HEAD is a pointer that points to the **most recent commit in the current branch**.

When you make a new commit, the HEAD moves to that new commit. It is also used to determine the current state of the Working Tree and the Index.

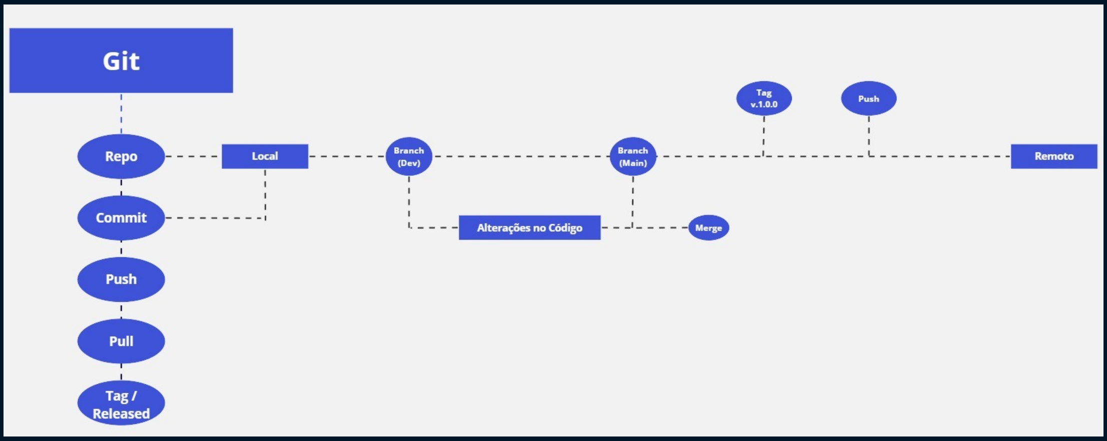
Working Tree

The Working Tree is where the files are actually stored and where you make your changes.

Index

The index is where Git stores what will be committed. It acts as an intermediate area between the Working Tree and the Git repository. To add files to the index, we use the command: `git add file.txt`

Git Architecture and Concepts

- 

Main Commands

`Commit`

This happens when we want to save the latest updates that were made.

Processes related to Commit:

- Check the status of the modified files:

```
git status
```

- Add the files to the commit:

```
git add file1.txt file2.js

git add .
```

- Perform the commit:

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

The `-m` is used to add a message to the commit, followed by the text `"First Commit"`.

`Push`

This happens when we send the changes to the repository - it "pushes" the modifications to the remote repository

Processes related to Push:

- Perform the push to the remote repository:

```
git push

# or

git push origin branch-name
```

- If this is the first connection and first push:

```
git push -u origin branch-name
```

You need to configure the relationship between the local and remote branches using the command above.

The `-u` establishes a tracking relationship, facilitating future pushes and pulls.

Pull

Updates your local repository with the changes in the remote repository - "pulls" the changes from the remote repository.

It brings all the changes from your remote repository to your local repository.

Pull-related processes:

- Perform a pull to get the latest changes:

```
git pull origin branch-name
```

- If you have already set up the tracking relationship during git push `-u`, you can just use:

```
git pull
```

Tag

A tag in Git is a specific reference to a point in the history of your repository. It is commonly used to mark stable or important versions of your project.

Tags are useful for creating fixed reference points that do not move as new commits are made.

Tag-related processes:

- List existing tags:

```
git tag
```

- Create a new tag:

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

- This command creates an annotated tag named `"v1.0"` with a descriptive message `"Version 1.0"`.

- **1. a: Creates an annotated tag.**

- **2. v1.0: Tag name.**

- **3. -m "Version 1.0": Descriptive message associated with the tag.**

- Share the tag to the remote repository:

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

`init`

Initializes a new Git repository in the current directory.

```
git init
```

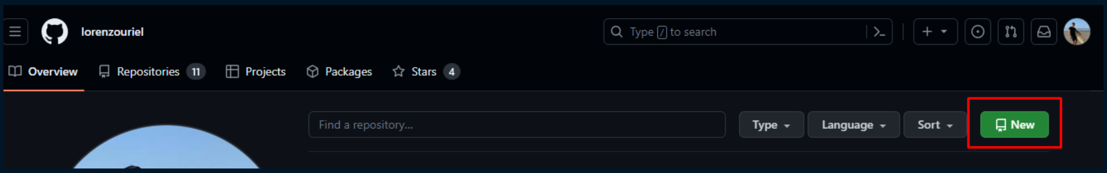
`clone`

Clones a remote repository to your computer.

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

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—use the cd p command to enter the desired directory.

```
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
```

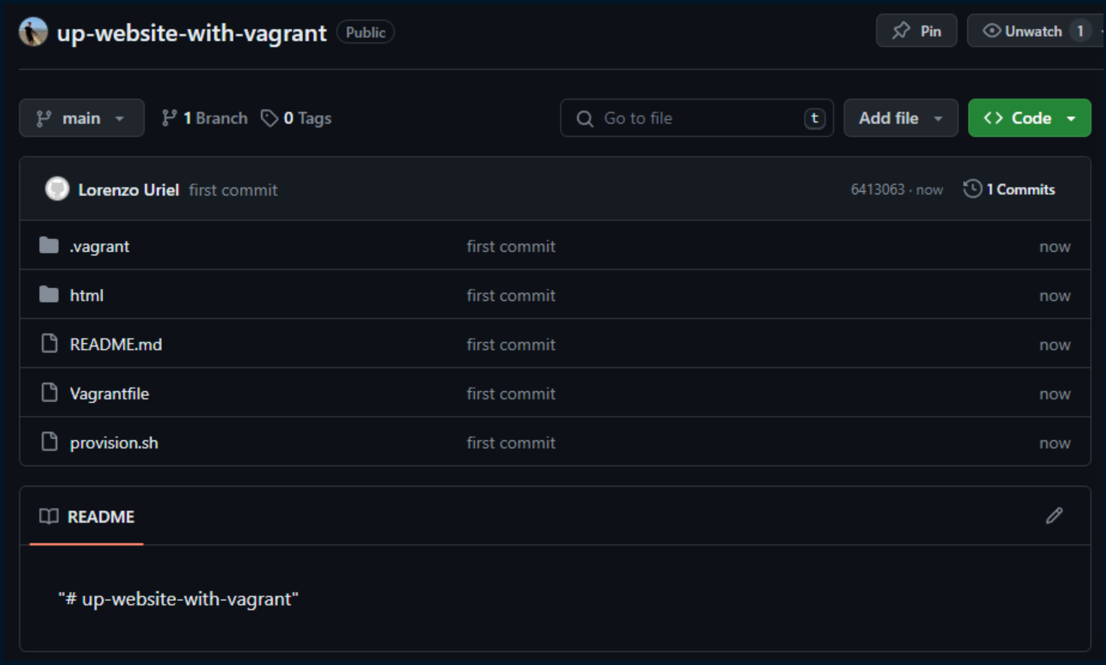
9. Adds a remote repository named "origin". The term "origin" is a standard term used to refer to the main remote repository. The URL is the repository's address on GitHub.

```
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:

- 

up-website-with-vagrant Public

main · 1 Branch · 0 Tags

Lorenzo Uriel first commit · 6413063 · now · 1 Commits

.vagrant	first commit	now
html	first commit	now
README.md	first commit	now
Vagrantfile	first commit	now
provision.sh	first commit	now

README

"# up-website-with-vagrant"

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.

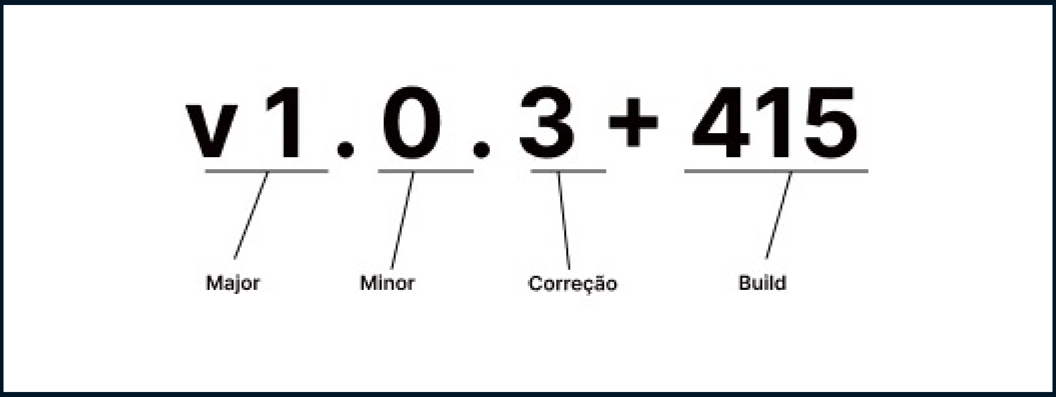
It is a way to organize and document your work.

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:

- 
v1.0.3+415
Major
Minor
Correção
Build

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

1. First item
2. Second item
3. Subitem 2.1
4. Subitem 2.2

~~~
#### Links:
- To create links, use square brackets for the link text and parentheses
for the URL.
```md
[Link to my website](https://www.example.com)
~~~

Images:

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

~~~
#### Emphasis (bold and italics):
- To create emphasis in text, you can use asterisks or underscores.
```md
This is an **amazing project** that uses _modern technologies_.
~~~

Quotes:

- To create a quote, use the > symbol.

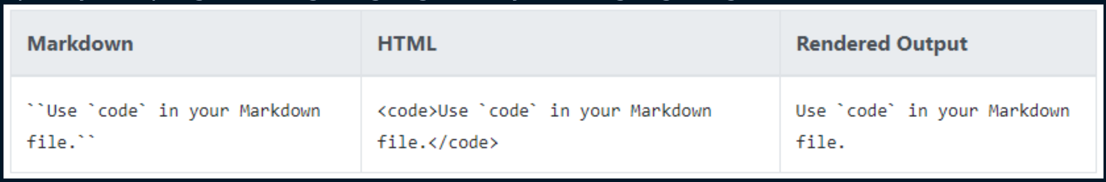
```
> "Be the best version of yourself." - Einstein
```

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.
- 

Markdown	HTML	Rendered Output
``Use `code` in your Markdown file.``	<code>Use `code` in your Markdown file.</code>	Use `code` in your Markdown file.

Working with Branches

Creating and Managing Local Branches

1. Create a new branch:

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

2. Switch to a branch:

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

3. Create and switch to a new branch:

```
git checkout -b <branch-name>
```

4. List all branches:

```
git branch
```

5. Rename a branch:

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

6. Delete a branch:

```
git branch -d <branch-name>
```

Merging Branches

1. Merge a branch into the current branch:

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

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)
>>>>>>> <branch-name>
```

After resolving the conflicts, mark the files as resolved:

```
git add .
```

Finish the merge:

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

Remote Branches and Branch Tracking

1. List remote branches:

```
git branch -r
```

2. Create a branch tracking a remote branch:

```
git checkout --track origin/<remote-branch-name>
```

- You only need to track if the remote branch does not exist in your local repository.

3. Update remote branches:

```
git fetch
```

4. Merge changes from a remote branch into the current branch:

```
git pull origin <remote-branch-name>

# Or just

git pull
```

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

1. Feature Branch:

- Create a new branch for the feature:

```
git checkout -b feature/<feature-name>
```

- Work on the feature and make commits.

- Merge the feature branch into the main branch (usually `main` or `dev`):

```
git checkout main
git merge feature/<feature-name>
```

2. Hotfix Branch:

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

```
git checkout -b hotfix/<hotfix-name> main
```

- Work on the hotfix and commit.

- Merge the hotfix branch into the main branch:

```
git checkout main
git merge hotfix/<hotfix-name>
```

- Merge the hotfix branch into the `dev` branch (if any):

```
git checkout dev
git merge hotfix/<hotfix-name>
```

3. Release Branch:

- Create a new branch for the release from the development branch:

```
git checkout -b release/<version> <branch-name>

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

- Test and prepare the release, committing as needed.

- Merge the release branch into the main branch and tag the release:

```
git checkout main
git merge release/<version>
git tag -a v<version> -m "Release <version>"

# Code
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:

```
git checkout dev
git merge release/<version>
```

--

Sync and Collaboration

Setting Up Remote Repositories

1. Add a remote repository:

```
git remote add origin <remote-repository-url>
```

- Replace `<remote-repository-url>` with the URL of your remote repository (e.g., on GitHub, GitLab, or Bitbucket).

2. Check the configured remote repositories:

```
git remote -v
```

3. Change the URL of a remote repository:

```
git remote set-url origin <new-remote-repository-url>
```

4. Remove a remote repository:

```
git remote remove origin
```

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

1. Push changes to the main branch:

```
git push origin main
```

2. Push a specific branch to the remote repository:

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

3. Push all local tags to the remote repository:

```
git push --tags
```

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

1. Update the local repository with changes from the remote repository and merge automatically:

```
git pull origin main
```

- This is equivalent to `git fetch` followed by `git merge`.

2. Get changes from the remote repository without merging:

```
git fetch origin
```

3. Merge changes pulled from the remote repository:

```
git merge origin/main
```

Working with Forks and Pull Requests

1. Fork a repository:

- On GitHub, GitLab, or Bitbucket, use the web interface to fork the desired repository.

2. Clone the forked repository:

```
git clone <url-of-forked-repository>
```

3. Add an upstream repository to sync with the original repository:

```
git remote add upstream <url-of-original-repository>
```

- The term "**upstream" is used to refer to the remote repository that is the official source for a project.** This is the main repository from which you typically want to sync updates.

4. Sync your fork with the original repository:

- Fetch changes from the original repository:

```
git fetch upstream
```

- Merge changes into the main branch:

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

- Push changes to your fork on GitHub:

```
git push origin main
```

5. Create a pull request:

- On GitHub, GitLab, or Bitbucket, **use the web interface to create a pull request from your fork to the original repository.** Describe the changes you made and request review.

Complete Flow Example

1. Add Remote Repositories:

```
git remote add origin <your-fork-url>
git remote add upstream <original-repository-url>
```

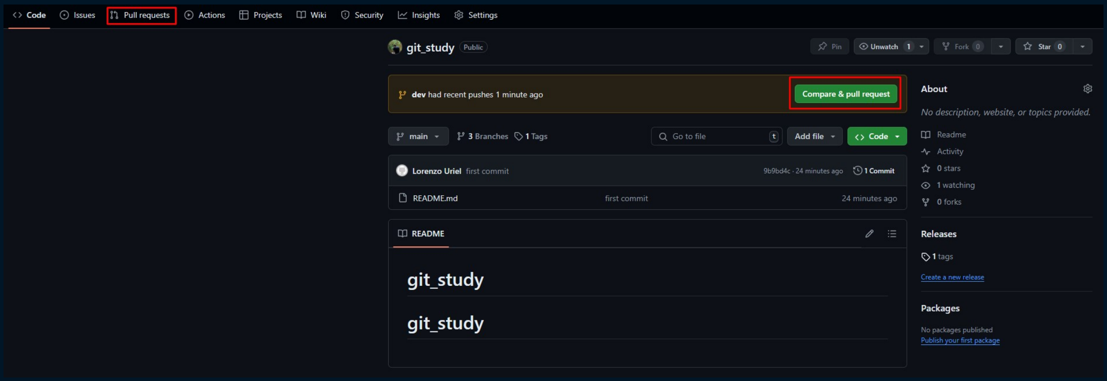
2. Make Changes and Commit:

```
git add .
git commit -m "My Contribution"
```

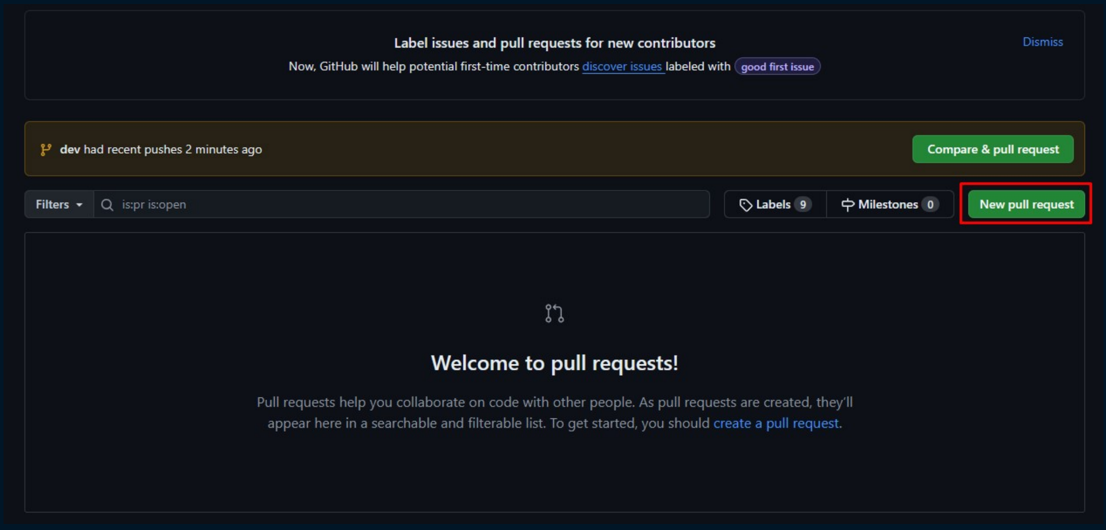
3. Push Changes to the Fork:

```
git push origin main
```

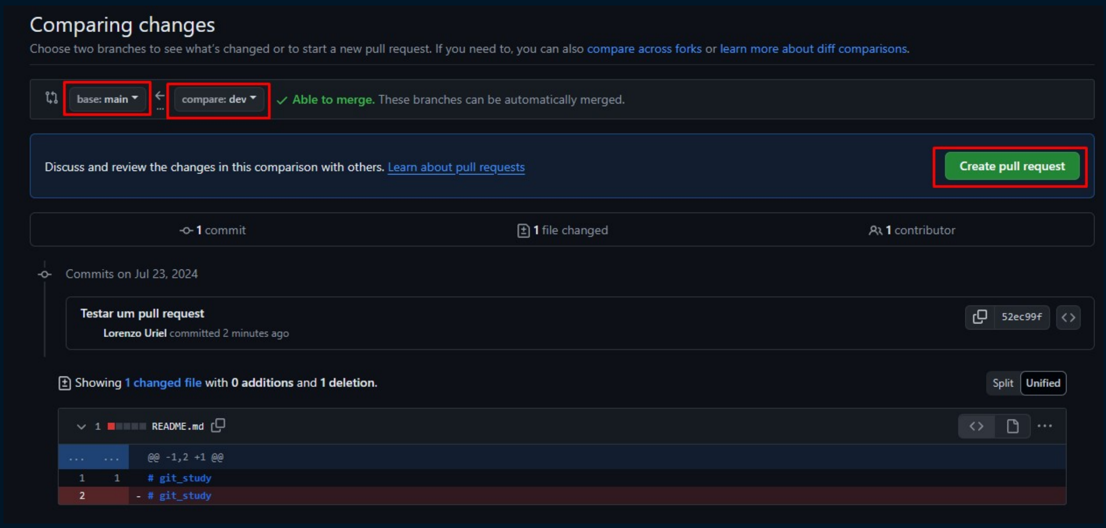
4. Create a Pull Request:

- On GitHub, **go to the original repository and click "New Pull Request".**
- 

- In the **Pull Request** tab, click **New Pull Request**

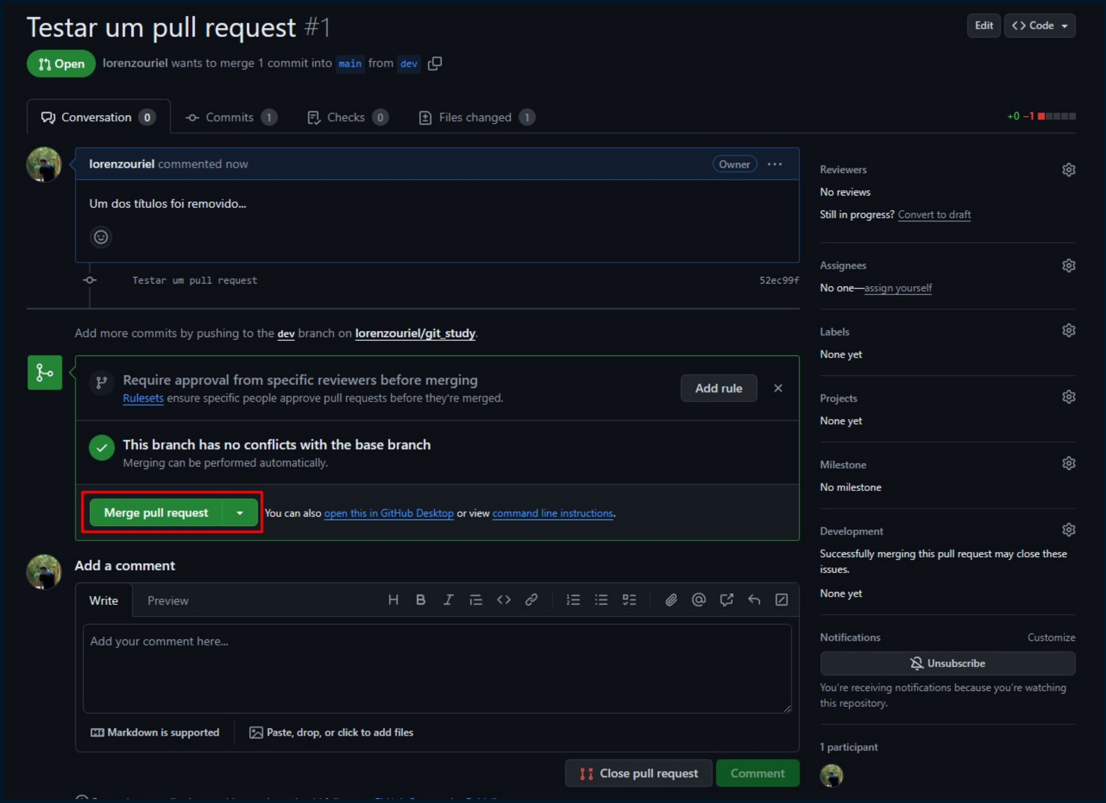
- 
Label issues and pull requests for new contributors
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue
Dismiss
dev had recent pushes 2 minutes ago
Compare & pull request
Filters is:pr is:open
Labels 9 Milestones 0 New pull request
Welcome to pull requests!
Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a searchable and filterable list. To get started, you should create a pull request.

- Select the branch that has changes (`compare:dev`) and the one you want to merge (`base:main`). Click on **Create Pull Request**

- 
Comparing changes
Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff comparisons.
base: main ← compare: dev ✓ Able to merge. These branches can be automatically merged.
Discuss and review the changes in this comparison with others. Learn about pull requests
Create pull request
1 commit 1 file changed 1 contributor
Commits on Jul 23, 2024
Testar um pull request
Lorenzo Uriel committed 2 minutes ago
52ec99f
Showing 1 changed file with 0 additions and 1 deletion.
Split Unified

```
1 README.md
@@ -1,2 +1 @@
  # git_study
- # git_study
```

- GitHub will navigate to this tab and test again if the changes have no conflicts, after checking click on **Merge pull request**

- 
  Testar um pull request #1

  Open lorenzouriel wants to merge 1 commit into `main` from `dev`

  Conversation 0 · Commits 1 · Checks 0 · Files changed 1

  lorenzouriel commented now

  Um dos títulos foi removido...

  Testar um pull request 52ec99f

  Add more commits by pushing to the dev branch on lorenzouriel/git_study.

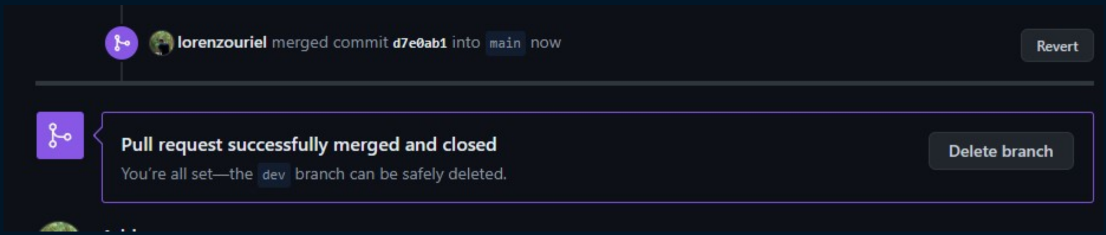
  Require approval from specific reviewers before merging
  Rulesets ensure specific people approve pull requests before they're merged.

  This branch has no conflicts with the base branch
  Merging can be performed automatically.

  Merge pull request — You can also open this in GitHub Desktop or view command line instructions.

  Reviewers: No reviews. Still in progress? Convert to draft
  Assignees: No one—assign yourself
  Labels: None yet
  Projects: None yet
  Milestone: No milestone
  Development: Successfully merging this pull request may close these issues. None yet
  Notifications: Unsubscribe. You're receiving notifications because you're watching this repository.
  1 participant

- If you have this culture, you can delete the branch directly in the **Pull Request** step

- 
  lorenzouriel merged commit `d7e0ab1` into `main` now

  Pull request successfully merged and closed

  You're all set—the `dev` branch can be safely deleted.

5. Sync with the Original Repository:

- After applying the changes, you need to push to your local repository:

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

- Then, you can delete the branch from the local repository:

```
git checkout -d dev
```

Repository Management

Removing Local Repositories

1. Removing a repository locally:

- To delete a Git repository locally, you can simply delete the directory where it is located. For example:

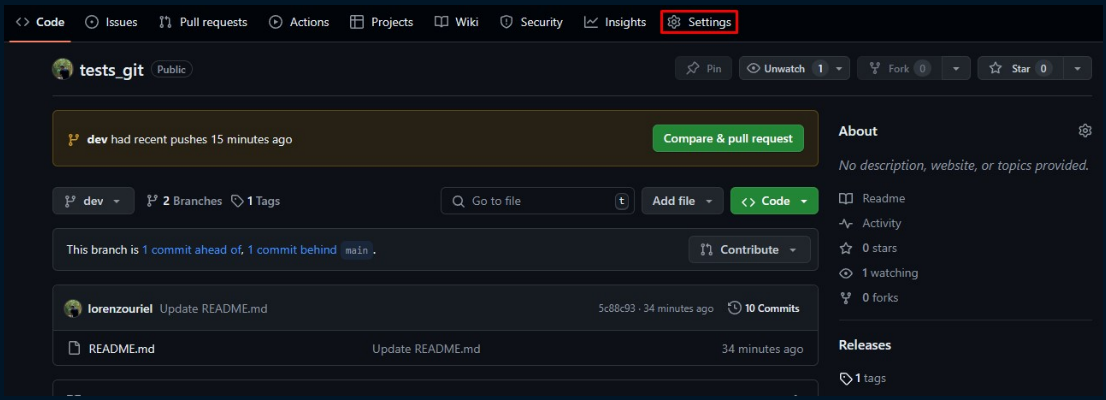
```
# bash
rm -rf /path/to/your/repository
git commit -m "Remove tests_git directory"
git push

# power shell
git rm -r --cached /projects/portfolio/tests_git
Remove-Item -Path "C:\Projects\portfolio\tests_git" -Recurse -Force
git commit -m "Remove tests_git directory"
git push
```

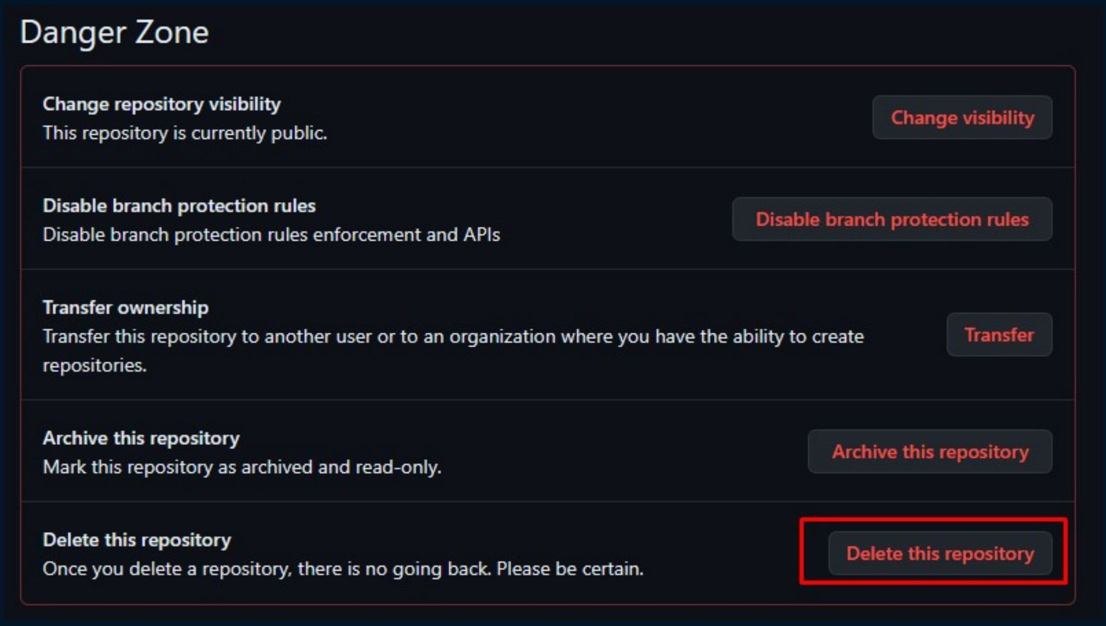
- Use this command with caution, as it will permanently delete all files and commit history in the repository.

Deleting Remote Repositories on GitHub

Go to the repository page on your hosting service.

In the repository settings, look for the option to delete the repository.

- 

  dev had recent pushes 15 minutes ago
  This branch is 1 commit ahead of, 1 commit behind main .
  lorenzouriel Update README.md 5c88c93 · 34 minutes ago
  README.md Update README.md 34 minutes ago
  No description, website, or topics provided.
- 
  Danger Zone
  Change repository visibility
  This repository is currently public.
  Disable branch protection rules
  Disable branch protection rules enforcement and APIs
  Transfer ownership
  Transfer this repository to another user or to an organization where you have the ability to create repositories.
  Archive this repository
  Mark this repository as archived and read-only.
  Delete this repository
  Once you delete a repository, there is no going back. Please be certain.

Confirm the deletion with the password or by accessing GitHub Mobile.

History Cleanup (`Rebase`, `Squash`)

1. Rebase:

Rebase is used to rewrite the commit history.

Git Rebase moves one branch on top of another, rewriting the history in the process. Instead of performing a merge, it rewrites the history.

For example, to rebase your `dev` branch onto the `main` branch:

```
git checkout dev
git rebase main
```

The idea of rebasing is to create a more linear commit history, making future code reviews easier.

2. Interactive Rebase:

To merge or reorder commits, use interactive rebase:

```
git rebase -i HEAD~n
```

- Where `n` is the number of commits you want to review. You will see a list of commits that you can edit, merge (squash), or reorder.

When you run rebase, your terminal will return like this:

```
pick f7f3f6d Meu commit 01
edit 310154e Meu commit 02
squash a5f4a0d Meu commit 03
drop a5f4a0d Meu commit 04

# Rebase 9b9bd4c..9b9bd4c onto 9b9bd4c (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.
#
~
```

Each command specifies exactly what you can do with the commits.

Your commits will appear above the commands, just select the line of the desired commit and click just the initial letter of each command. The command will automatically add it to the beginning of the commit.

What does the full script do?

- Apply commit `f7f3f6d` as is.
- Apply commit `310154e` and stop to allow editing.
- Combines commit `a5f4a0d` with the previous commit `310154e`.
- Removes commit `a5f4a0d` from the history.

This results in:

- The first commit (`f7f3f6d`) being applied normally.
- The second commit (`310154e`) being applied and edited as needed.
- The third commit (`a5f4a0d`) being combined (`squashed`) with the second commit.
- The fourth commit (`a5f4a0d`) being removed from the history.

Additional Notes

- The `squash` command will prompt you to edit the resulting commit message.
- `squash` combines multiple commits into a single commit.
- The `edit` command will stop the rebase to allow modifications to the commit.
- The `drop` command completely removes the commit from the history.

Tags and Releases

1. Create a Tag:

- Tags are used to mark specific points in the commit history, often used to mark release versions.

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

2. Create an Annotated Tag: Annotated tags are recommended for releases because they contain additional information.

```
git tag -a <tag-name> -m "Tag Message"
```

3. Push a Tag to the Remote Repository: Whenever you create a new tag you need to push it to the remote repository.

```
git push origin <tag-name>
```

4. Push All Local Tags to the Remote Repository:

```
git push --tags
```

5. List All Tags:

```
git tag
```

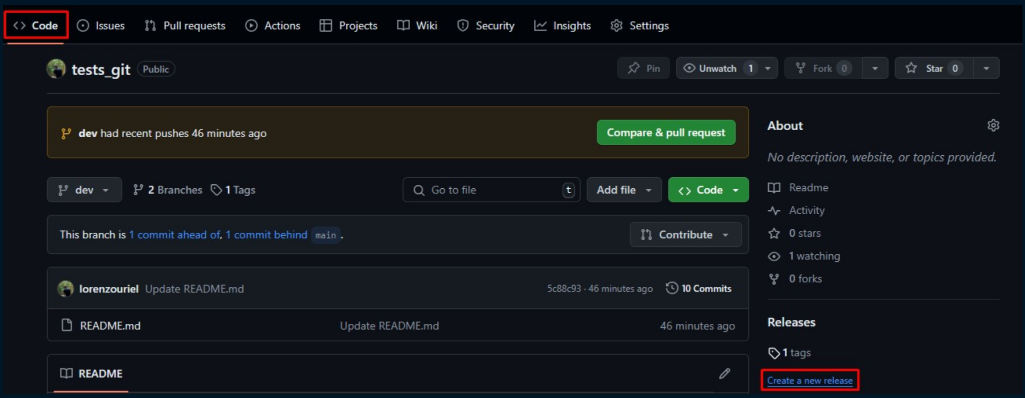
6. Delete a Tag Locally:

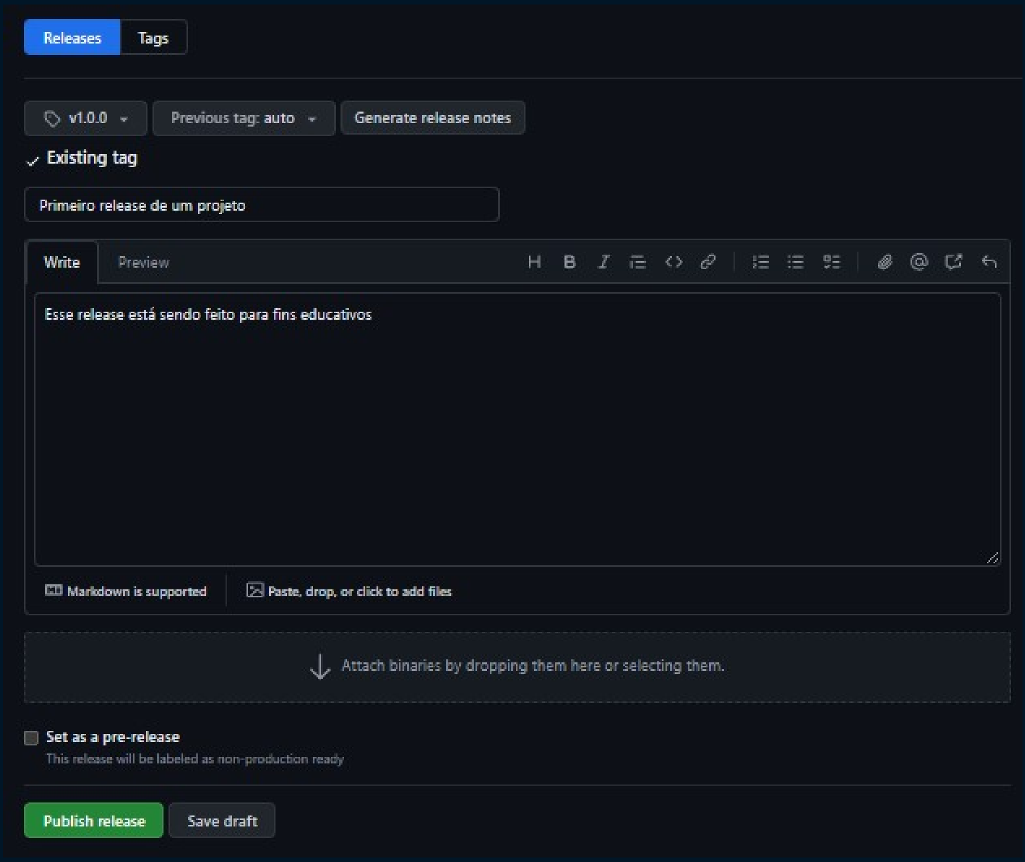
```
git tag -d <tag-name>
```

7. Delete a Tag in the Remote Repository:

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

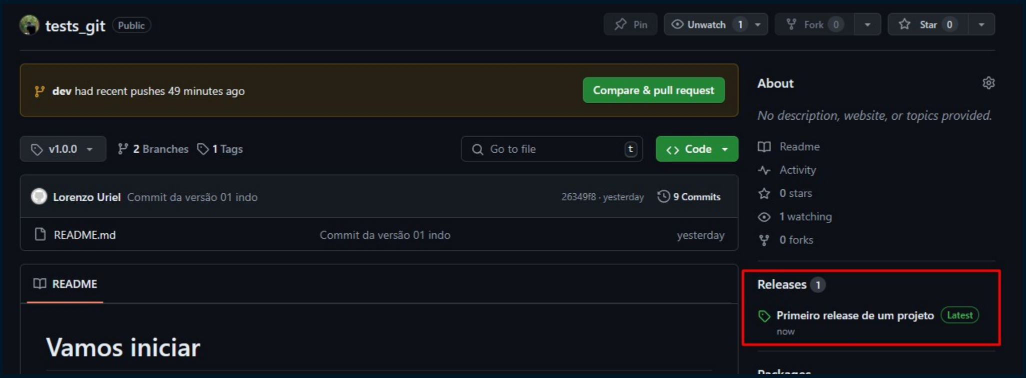
8. Create a Release on GitHub:

- On GitHub, go to the repository page.
- Go to the "Code" > "Create a New Release" section.
    - 

- Fill in the information, add the tag and click "Publish a New Release":
    - 

      Primeiro release de um projeto

      Esse release está sendo feito para fins educativos

- Your releases are organized in the "Code" section
  - 

dev had recent pushes 49 minutes ago

Lorenzo Uriel Commit da versão 01 indo 26349f8 · yesterday

README.md Commit da versão 01 indo yesterday

Vamos iniciar

About

No description, website, or topics provided.

Releases 1

Primeiro release de um projeto Latest
now

Best Practices

Clear and Frequent Commits

Frequent Commits:

- Make small, frequent commits to make it easier to identify problems and revert specific changes.
- Each commit should represent a single, logical change to the code.
- Every change should have a commit.

Clear Commit Messages:

- Use descriptive, meaningful commit messages.
- The line should be a short, concise summary (50 characters or less), and you can add a more detailed description if necessary.

```
git commit -m "Fix bug in login function"
git commit -m "Add unit tests for authentication module"
```

Using `.gitignore`

Creating and Using the `.gitignore` File:

- The `.gitignore` file is used to specify which files and directories should be ignored by Git.
- Common examples include local configuration files, build directories, and temporary files.
- You can use the toptal website to find complete and populated `.gitignore` files for your language.

Example of a `.gitignore` file:

```
# Logs
*.log

# Build directories
/build/
/dist/

# Configuration files
.env
.vscode/

# Temporary files
*.tmp
```

- To create or edit the `.gitignore` file, add it to the root of your repository.

Branch Repositioning

Rebase instead of Merge:

Use `git rebase` to apply commits from one branch on top of another, creating a more linear history.

```
git checkout feature-branch
git rebase main
```

- The `git rebase` command is used to merge changes from one branch into another, but in a different way than the `git merge` command. While `merge` creates a new `merge` commit, `rebase` reapplies the commits from one branch over the tip of another branch, resulting in a linear history.

Resolve conflicts if necessary and continue with:

```
git rebase --continue
```

Merge Branches:

Use `git merge` to combine changes from one branch into another, preserving the commit history.

```
git checkout main
git merge feature-branch
```

Maintaining a Clean and Comprehensible History

Interactive Rebase to Clean Up History:

Use `git rebase -i` to reorder, edit, squash, or discard commits.

```
git rebase -i HEAD~n
```

- In the editor that opens, change `pick` to `squash` to combine commits.

Avoid unnecessary merge commits by using `git pull --rebase` instead of `git pull`.

Automating Tasks with Git Hooks

Configure Git Hooks:

Git hooks are scripts that run automatically on certain events, such as before a commit or after a push.

Examples of hooks include `pre-commit`, `pre-push`, `post-commit`, and others.

Example of a `pre-commit` hook to check code style:

```
@echo off
# Pre-commit hook to run pytest

echo Running pytest
pytest

# if pytest fails, don't commit
if errorlevel 1 (
echo Tests failed
exit /b 1
)
```

Place the hook scripts in the `.git/hooks/` directory of your repository and make the scripts executable:

```
chmod +x .git/hooks/pre-commit
```

Summary

- Clean and frequent commits make collaboration and code management easier.
- Using .gitignore keeps the repository clean of unnecessary files.
- Repositioning branches with proper rebasing and merging keeps the history readable and linear.
- Maintaining a clean history with tools like interactive rebasing avoids unnecessary complexity.
- Task automation with Git hooks ensures code quality and repository consistency.

Troubleshooting

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

`git revert`

Used to **create a new commit that undoes the changes of a specific commit, preserving history.**

```
git revert <commit-hash>
```

This creates a new commit that undoes the changes of the specified commit.

This is what your log looks like, after `revert`:

```
loren@ MINGW64 /c/Projects/portfolio/git_study (dev)
$ git revert 0d3bd1def0f408e07b1249d7c599759b46380640
hint: Waiting for your editor to close the file...
[No write since last change]

Press ENTER or type command to continue
[No write since last change]

Press ENTER or type command to continue
[dev 68feb15] Revert "Commit para reverter"
 1 file changed, 1 insertion(+), 3 deletions(-)

loren@ MINGW64 /c/Projects/portfolio/git_study (dev)
$ git log
commit 68feb159e81c636e290febc9bbe0d94ac8a15ccb (HEAD -> dev)
Author: Lorenzo Uriel <lorenzo.uriel@.com.br>
Date:   Thu Jul 25 22:05:30 2024 -0300

    Revert "Commit para reverter"

    This reverts commit 0d3bd1def0f408e07b1249d7c599759b46380640.

commit 0d3bd1def0f408e07b1249d7c599759b46380640
Author: Lorenzo Uriel <lorenzo.uriel@.com.br>
Date:   Thu Jul 25 22:05:02 2024 -0300

    Commit para reverter
```

I made an initial `commit` with the message `Commit to revert`, then I got the commit hash and did a `git revert 0d3bd1def0f408e07b1249d7c599759b46380640` with the message `Revert "Commit to revert"`.

`git reset`

Used to **reset the state of the repository to a previous commit.** The `reset` command can change history.

Main methods of `git reset`:

- `--soft`: Keeps files and indexing as they are, only resets the HEAD pointer

```
git reset --soft <commit-hash>
```

- `--mixed`: Keeps files in the working directory, but undoes index changes.

```
git reset --mixed <commit-hash>
```

- `--hard`: Resets the HEAD pointer, index, and working directory. ``bash git reset --hard

```
### **`git checkout`**
Used to **change branches or restore files in the working directory.**
`Checkout` does not affect history. ```bash
git checkout <branch-name>
git checkout <commit-hash> -- <file-path>
```

Common Conflict Resolution

Merge Conflicts

When a merge conflict occurs, Git marks the conflicting files so you can resolve them manually.

Example:

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

Resolve conflicts by editing the conflicting files. Example message:

```
loren@ MINGW64 /c/Projects/portfolio/git_study (main)
$ git merge dev
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

- Mark conflicts as resolved:

```
git add <file-path>
```

- Complete the merge:

```
git commit -m "Fixed bugs, ready to go"
```

Rebase Conflicts

During rebasing, conflicts are resolved in a similar way to merge conflicts.

Example:

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

- Resolve conflicts by editing the conflicting files.
- Continue the rebase:

```
git rebase --continue
```

- To abort the rebase:

```
git rebase --abort
```

Recovering Lost Commits

`git reflog`

Keeps a record of all the **HEADs** that have been changed in the local repository.

It is often used for lost commits.

Example:

```
git reflog
```

- Identify the commit you want to recover and use `git checkout` or `git reset` to restore it.

Example of return in the terminal:

```
$ git reflog
76e8ecf (HEAD -> main) HEAD@{0}: commit (merge): Erros ok
3fd367d HEAD@{1}: commit: Vou fazer um merge
7ce5614 (origin/main) HEAD@{2}: checkout: moving from dev to main
1dbfda0 (dev) HEAD@{3}: checkout: moving from dev to dev
1dbfda0 (dev) HEAD@{4}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{5}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{6}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{7}: commit: Commit para dar um reset
68feb15 HEAD@{8}: revert: Revert "Commit para reverter"
0d3bd1d HEAD@{9}: commit: Commit para reverter
7ce5614 (origin/main) HEAD@{10}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{11}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{12}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{13}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{14}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{15}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{16}: rebase (finish): returning to
```

```
refs/heads/dev
7ce5614 (origin/main) HEAD@{17}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{18}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{19}: rebase (start): checkout main
39f9d27 HEAD@{20}: checkout: moving from main to dev
7ce5614 (origin/main) HEAD@{21}: rebase (finish): returning to
refs/heads/main
7ce5614 (origin/main) HEAD@{22}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{23}: commit (merge): Commit 04
39f9d27 HEAD@{24}: rebase (finish): returning to refs/heads/main
39f9d27 HEAD@{25}: rebase (start): checkout HEAD
39f9d27 HEAD@{26}: checkout: moving from dev to main
39f9d27 HEAD@{27}: checkout: moving from main to dev
39f9d27 HEAD@{28}: checkout: moving from dev to main
39f9d27 HEAD@{29}: rebase (finish): returning to refs/heads/dev
39f9d27 HEAD@{30}: rebase (start): checkout main
52ec99f (origin/dev) HEAD@{31}: rebase (finish): returning to
refs/heads/dev
52ec99f (origin/dev) HEAD@{32}: rebase (start): checkout HEAD
52ec99f (origin/dev) HEAD@{33}: checkout: moving from main to dev
```

`git cherry-pick`

Used to **apply a specific commit from one branch to another.**

Example:

```
git cherry-pick <commit-hash>
```

Troubleshooting Tips

Problem Diagnosis

Use commands like `git status`, `git log`, and `git diff` to understand the current state of your repository and identify issues.

- `git status`: Provides an overview of the current state of your repository. It shows which files have been modified, which are ready to be committed, and which are untracked. This helps you quickly understand what has changed since your last commit.

- `git log`: Used to view the history of commits. This is useful for identifying changes made over time and understanding how the repository has evolved. You can add options like `--oneline` for a more concise view or `--graph` for a graphical representation of the commits.

- `git diff`: Compares changes between commits, between the repository and the working directory, or between different branches. It is useful for reviewing specific changes to files and understanding how the code was modified.

Ignore Files Locally

If you need to ignore changes to files that are tracked, use

```
git update-index --assume-unchanged <file-path>
```

To undo:

```
git update-index --no-assume-unchanged <file-path>
```

Discard Local Changes

To discard changes to modified files:

```
git checkout -- <file-path>
```

To discard all local changes:

```
git reset --hard
```

Troubleshoot Push/Pull Issues

If you encounter problems when pushing or pulling, such as push rejections, first fetch to synchronize your repository:

```
git fetch
```

- Resolve any conflicts, then try again

Tools and Integrations

Git Integration with Popular IDEs

1. VSCode:

- VSCode has native integration with Git. You can initialize repositories, commit, pull, push, view commit histories, and resolve conflicts directly from the editor.
- There are several extensions, such as **GitLens**, that enhance the Git experience in VSCode, offering features such as viewing line authorship, comparing branches, and more.

2. IntelliJ:

- IntelliJ offers robust integration with Git, allowing you to manage repositories, commits, branches, merges, and more, directly from the IDE.
- Graphical tools for file comparison and merge conflict resolution are built in and quite intuitive.

Using Graphical Tools for Git

1. GitKraken:

- Provides a clear visual interface for managing Git repositories, with features such as viewing commits, creating and managing branches, and resolving conflicts.
- Supports integrations with several repository hosting platforms such as GitHub, GitLab, Bitbucket, among others.

2. SourceTree:

- Allows you to clone, create, and manage local and remote repositories with an intuitive graphical interface.
- Provides a detailed view of the commit and branch history, making it easier to track changes.

Integration with CI/CD

1 Jenkins:

- Jenkins can be configured to automatically start builds on Git events such as commits and pull requests.
- There are specific plugins for integrating with Git, which facilitate the monitoring of repositories and execution of pipelines.

2. GitHub Actions:

- Allows the creation of automated workflows that are triggered by Git events (push, pull request, etc.).
- Workflows are configured using YAML files, offering great flexibility and control over CI/CD processes.

3. GitLab CI:

- Defines CI/CD pipelines directly in the repository with `.gitlab-ci.yml` files.
- Deep native integration with the GitLab repository, offering advanced CI/CD features such as automatic build, test, and deploy.

GitFlow

GitFlow is a branching model that defines an organized and efficient development process.

Main branches:

- `main`: Contains the production code.
- `develop`: Contains the code for the next version that will be released.

Support branches:

- `feature/*`: Used for developing new features.
- `release/*`: Preparation of a new production version.
- `hotfix/*`: Urgent fixes in production.