GITHUB

Github

Environment Setup

In this lesson, you will install **VSCode** and **Git** to use them during the course. **Git** is the most popular version control system. **VSCode** is a free powerful code editor. We will use it for some examples.

Please note: the setup is different for Windows and macOS, so please proceed to the appropriate section.

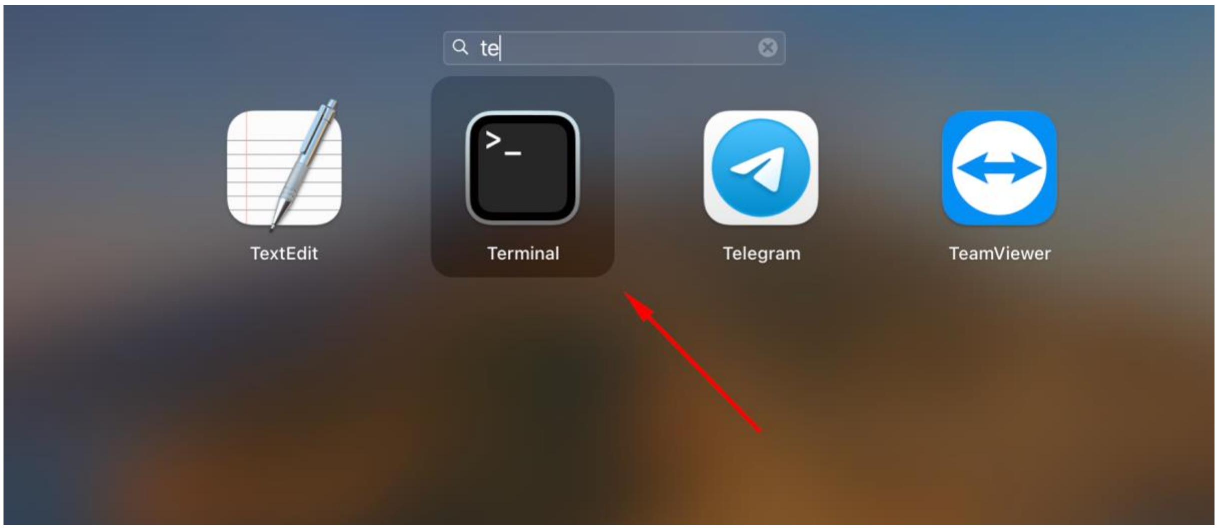
macOS

VSCode Installation (macOS)

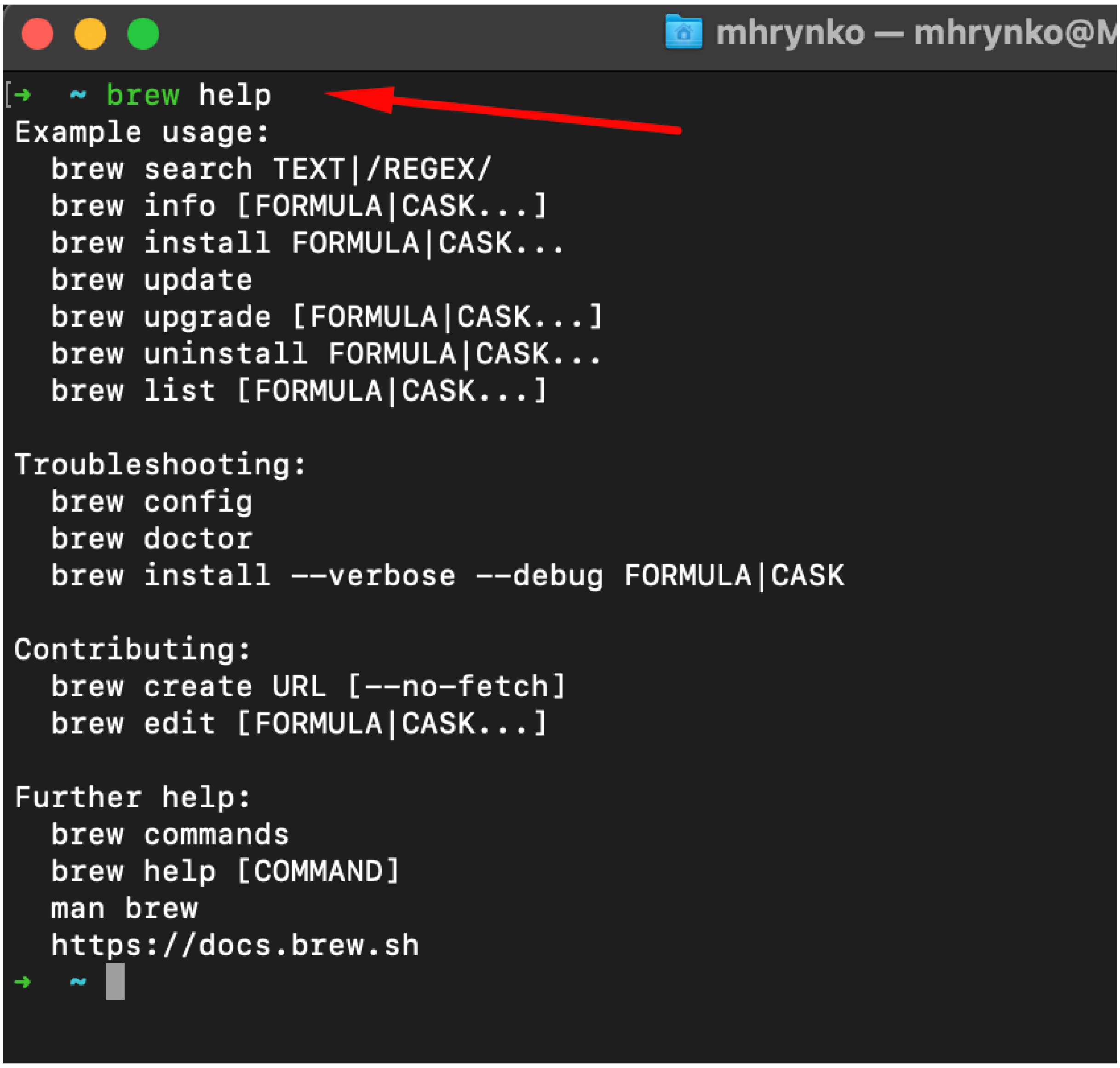
If you use macOS, download **VSCode**, **extract** it and **run**. Then you need to install **Git**.

Git Installation (macOS)

1. Open the **Terminal** app.



2. Type the `brew help` command in the **Terminal** and press **Enter** to check if you have `brew` installed.

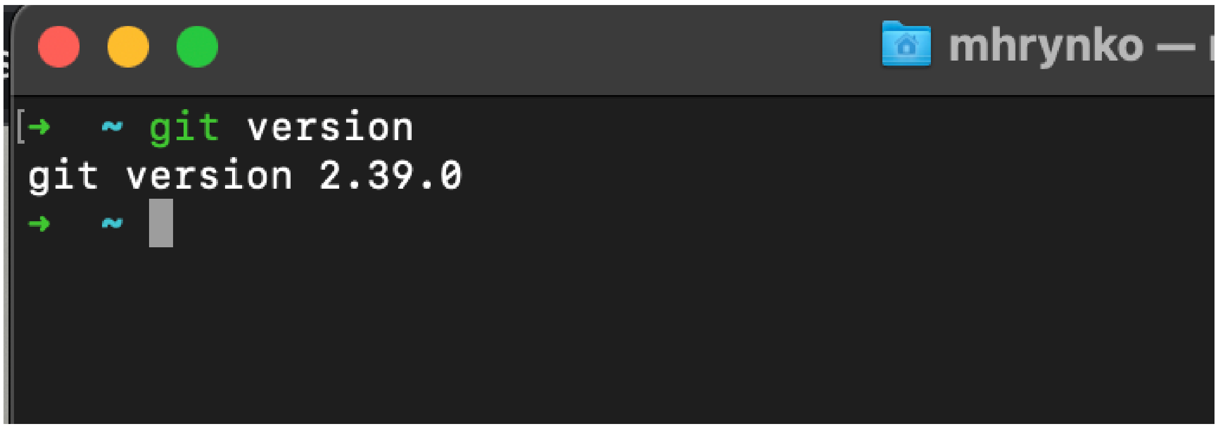


```
[➜  ~ brew help
Example usage:
  brew search TEXT|/REGEX/
  brew info [FORMULA|CASK...]
  brew install FORMULA|CASK...
  brew update
  brew upgrade [FORMULA|CASK...]
  brew uninstall FORMULA|CASK...
  brew list [FORMULA|CASK...]

Troubleshooting:
  brew config
  brew doctor
  brew install --verbose --debug FORMULA|CASK

Contributing:
  brew create URL [--no-fetch]
  brew edit [FORMULA|CASK...]

Further help:
  brew commands
  brew help [COMMAND]
  man brew
  https://docs.brew.sh
➜  ~ 
```

3. If you don't see the `brew` help, install it using the command from **the brew docs**.

4. Now run the `brew install git` command in the **Terminal** to install Git.

5. Close the **Terminal**, open it again, and run the `git --version` command to check if Git works.



```
[➜  ~ git version
git version 2.39.0
➜  ~ 
```

6. Run the following commands one by one with your name and email (change the text inside "" before pressing **Enter**):

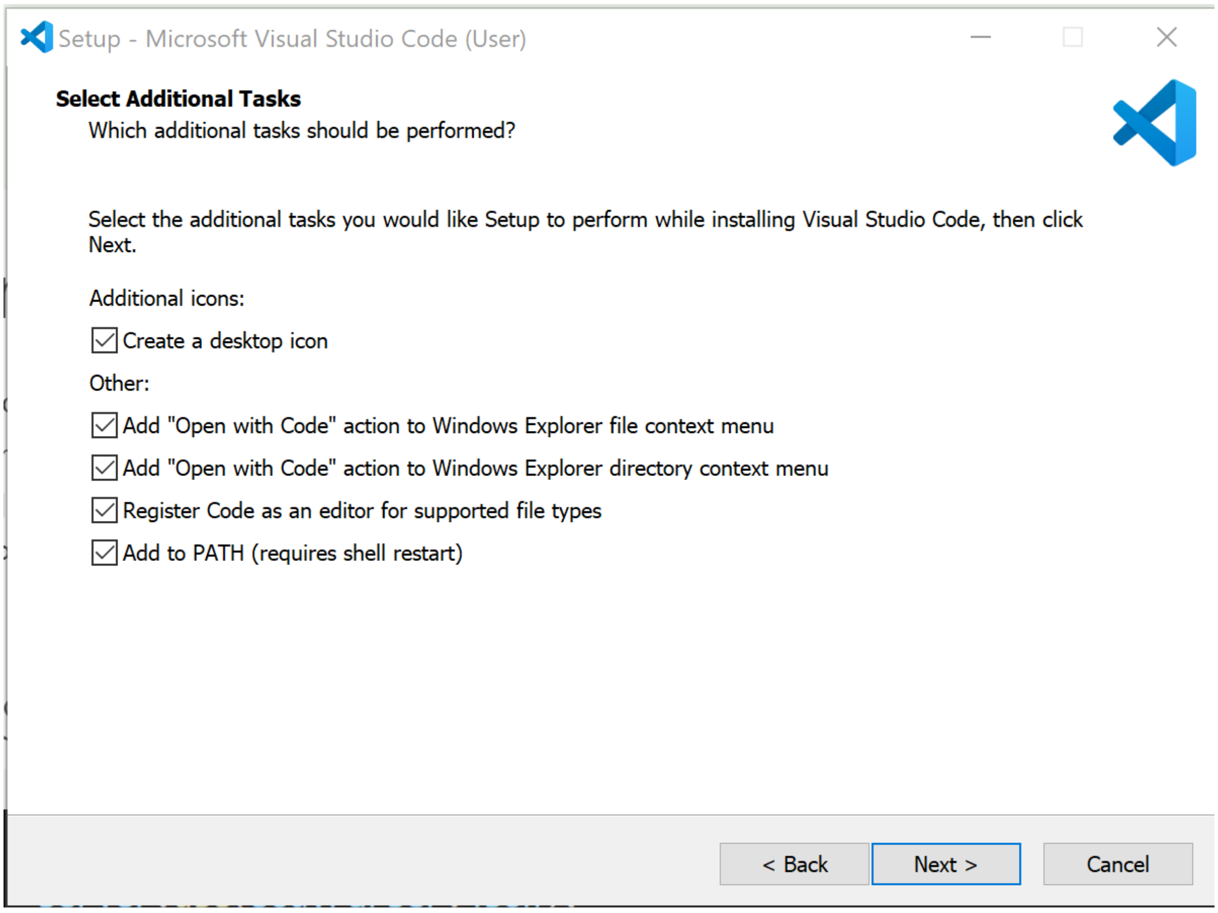
7. `git config --global user.name "Your Name"`

8. `git config --global user.email "your@mail.com"`

Windows

VSCode Installation (Windows)

1. Download **VSCode**.
2. Run the installation file as usually (**not** as an Administrator).
3. Proceed to the **Select Additional Tasks** screen.
4. Choose all the checkboxes there:
   - **Add "Open with Code" action ...** allows you to open files and folders from a right-click menu;
   - **Register Code ...** opens supported files in VSCode when you run them;
   - **Add to PATH** allows you to run VSCode from the **Terminal** using the `code` command.


Setup - Microsoft Visual Studio Code (User)

Select Additional Tasks

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

- [✓] Create a desktop icon

Other:

- [✓] Add "Open with Code" action to Windows Explorer file context menu
- [✓] Add "Open with Code" action to Windows Explorer directory context menu
- [✓] Register Code as an editor for supported file types
- [✓] Add to PATH (requires shell restart)

Introduction to Command Line Basics

We'll now move to something all operating systems come with: the terminal. But why cover the topic, if there's nothing to install? Glad you asked, and the answer comes down to commands. Terminals have no UI, so we must learn how to interact with them through text input.

Without further ado…

💡 If you're a Windows user, use Git Bash for an optimal experience.

Navigating the File System

Overview of Basic Navigation Commands

Here's a table of essential commands we'll use to navigate through the command line:

Command	Description
`pwd`	Displays the current directory's full path
`clear`	Clears the terminal screen
`ls path/to/folder`	Lists the contents of a specified folder
`ls -l path`	Displays the folder contents in detailed list format
`ls -a path`	Reveals hidden files and folders
`ls -t path`	Sorts the folder contents by modification date
`cd path/to/folder`	Changes the current directory

💡 In this text, terms `catalog`, `folder`, and `directory` are used interchangeably.

Understanding Our Location with `pwd`

Like navigating through Finder (macOS) or Explorer (Windows), you're always situated within a directory while using the terminal. The `pwd` command reveals your current directory, simplifying file system navigation.

Clearing the View with `clear`

Use the clear command to refresh your terminal's display, keeping it uncluttered without needing to restart it.

Listing Directory Contents with `ls`

The `ls` command is the go-to for viewing the contents of our current or a specified directory. Adding various options like `-l` for a detailed list, `-a` for including hidden items, and `-t` for sorting by date enhances its utility. Combining these options, such as `ls -la` or `ls -lat`, tailors the output to our needs.

Moreover, specifying a path allows you to explore the contents of any directory, with relative paths (using `./` or `../`) or absolute paths (starting with `/` for root or ~ for our home directory).

Changing Directories with `cd`

The `cd` command is crucial for moving across directories, allowing you to specify a path in the same format as with the ls command.

File and Directory Management

Next, we'll cover how to create, view, modify, and remove files and directories.

Key Commands for File and Directory Operations

Here's a concise reference of commands for managing files and directories:

Command	Description
touch file.name	Creates a new, empty file
cat file.name	Displays a file's contents
cat > file.name	Creates or overwrites a file, allowing input of content
mkdir dirname	Creates a new directory
mkdir -p dir1/dir2/dir3	Creates nested directories in one command
mv path/to/item path/to/destination	Moves or renames files and directories
cp path/to/source path/to/destination	Copies files or directories
rm path/to/item	Removes files or directories, with options for forceful or recursive deletion

Creating and Viewing Files

- The `touch` command quickly generates an empty file, useful for starting new projects.
- The `cat` command not only displays file contents but also allows for creating or appending to files with redirected input (`cat` > `file.name`).

Directory Creation and Structure

Use `mkdir` to craft new directories. The `-p` option is particularly powerful for establishing a hierarchy of nested directories efficiently.

Moving and Renaming Essentials

The versatile `mv` command serves dual purposes: relocating items and renaming them, based on the nature of the target path provided.

Copying Considerations

The `cp` command facilitates duplication of files and directories. The `-r` option is necessary for copying directories to include all subdirectories and files therein.

Safe Deletion Practices

Deletion commands, especially `rm` with options `-r` for recursive deletion and `-f` for forceful execution, should be used with caution to prevent unintended data loss.

Command History

Command	Description
`Arrow UP`	Show the previous command
`Arrow DOWN`	Show the next command
`history`	Show command history
`history -c`	Clear command history
`!!`	Execute the last command from history
`!12`	Execute the 12th command from history
`!some-text`	Execute the last command from history that started with the given text
`CTRL + R`	Search in command history

Git Basics

Computer programs consist of dozens, hundreds or even thousands of files with instructions. Hence, it's impossible to remember what and when we change, especially while working with many other developers. This becomes an issue when we have to restore the files — the program — to its former state, for instance, due to the instability caused by latest changes.

Version control systems serve this exact purpose: They ley us control and save the history of file changes, and **switch quickly between versions.** The most popular (and free!) such system is Git, which we'll discuss today.

How Does Git Work

When we save a change, Git create a **commit** with the so-called **delta** (the difference: all added and removed code) for the modified file. Each change is signed with a checksum hash, SHA-1, calculated based on the content, message, author, and time. In other words, it's impossible to change a file or directory without Git noticing.

💡 Git stores all changes locally, that is, only on the user's device.

Initialize a Repository and Prepare Files for Saving

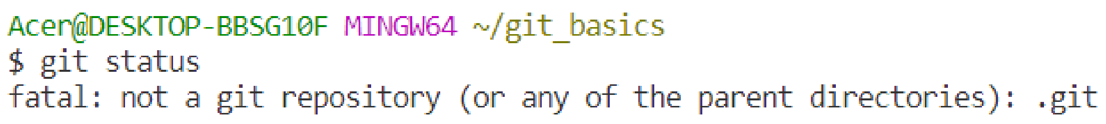
Activating Git

For starters, we need to activate Git. We do so by adding a hidden `.git` folder to a directory (thereafter known as a repository) and running below command in the terminal:

```
git init
```

All changes in the repository can be tracked using the `git status` command. This command shows the following:

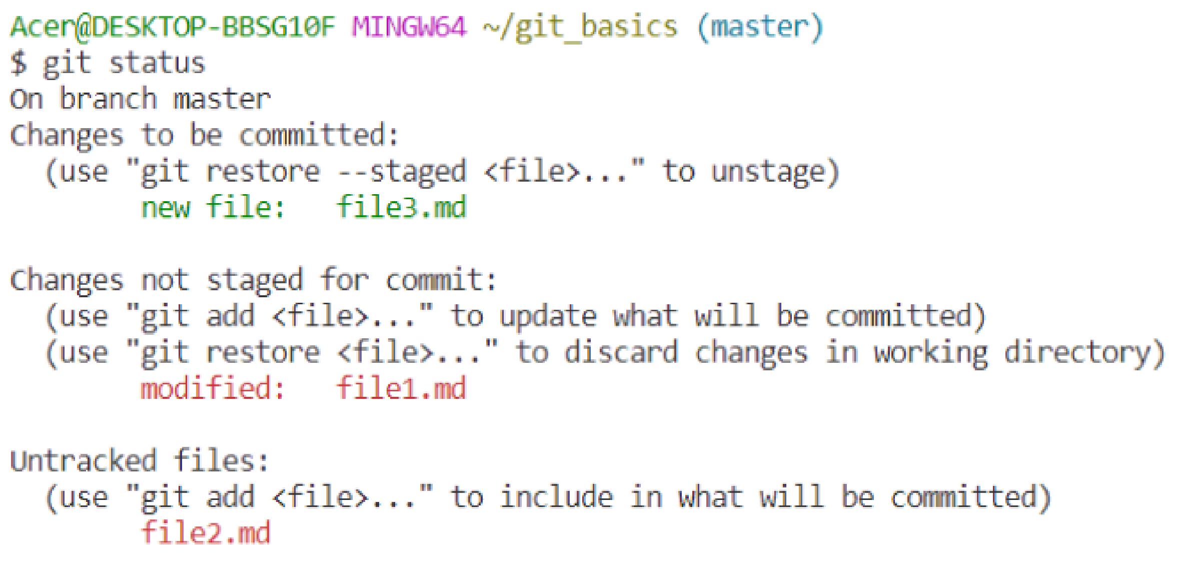
1. Whether the repository has been activated in the current directory.



```
Acer@DESKTOP-BBSG10F MINGW64 ~/git_basics
$ git status
fatal: not a git repository (or any of the parent directories): .git
```

2. Whether the repository has any *untracked files,* or whether changes have been made to the files tracked with Git (*modified*).

3. Whether any files are prepared for saving — added to the index (*changes to be committed*).



```
Acer@DESKTOP-BBSG10F MINGW64 ~/git_basics (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file3.md

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:   file1.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.md
```

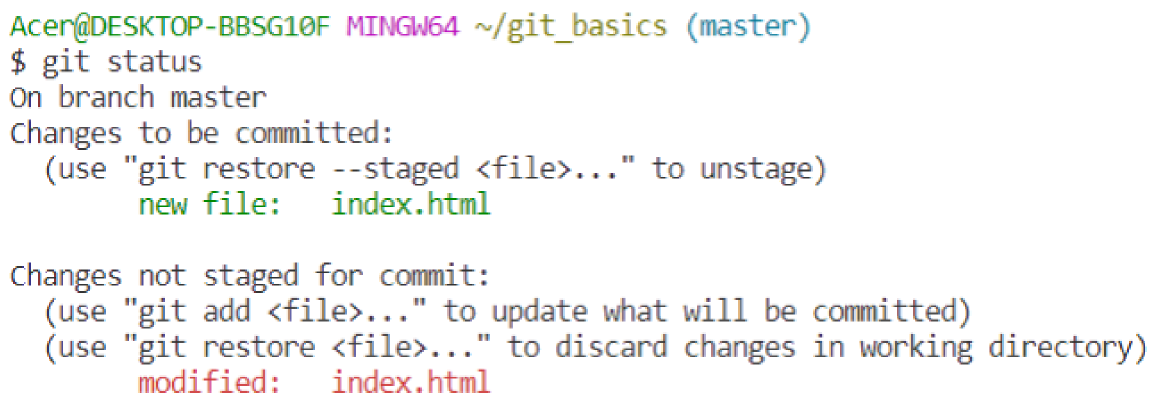
Deleting a repository is as simple as removing the hidden folder.

Index

Index is a file, usually located in a Git directory, containing information on what will be saved in the next commit. There are a few things we can add to it:

- One file at a time — `git add ./path/to/file`.
- Several files at a time, separating their names with spaces — `git add ./path/to/file ./path/to/file`.
- All files located in a specified directory — `git add path/to/folder`.
- All changed files in the current directory — `git add .`.

Git captures the state of a file's contents at a specific point in time, similar to taking a "snapshot." This happens when we add the file to the staging area (index), i.e., any changes made to the file after it's added to the index **aren't included automatically.** We'll need to add the file to the index again to capture the latest changes.



```
Acer@DESKTOP-BBSG10F MINGW64 ~/git_basics (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.html

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:   index.html
```

As shown in the image, Git interacts with us, offering suggestions on what commands to run next. To discard changes in a file that hasn't been added to the staging area yet, use:

```
git restore file_name
```

If the file was mistakenly added to the index, you can use the following command:

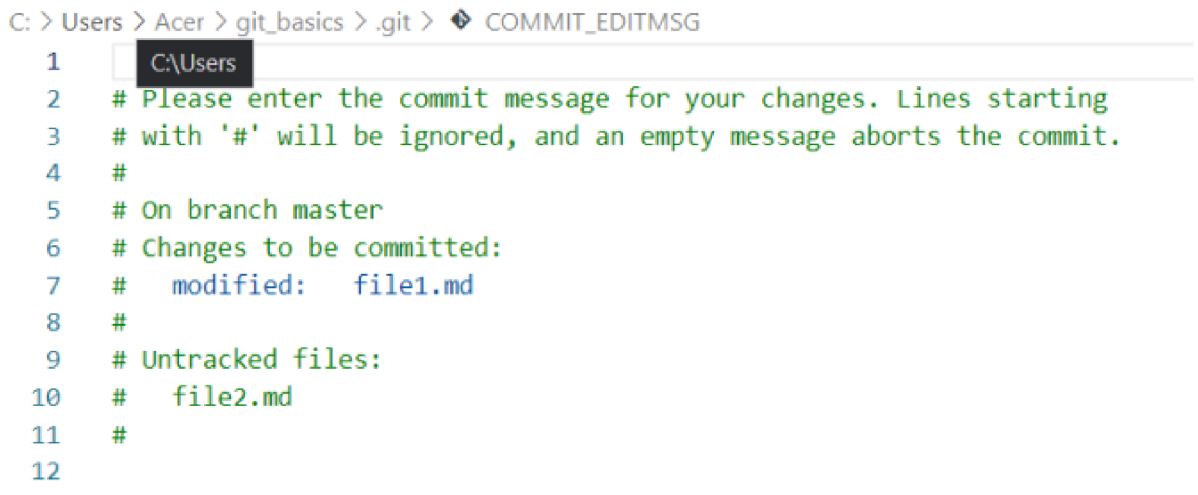
```
git restore -staged file_name
```

`.gitignore`

We can stop Git from tracking certain files and avoid changes to already tracked files. To do so, add a `.gitignore` file in the repository's root directory — where the `.git` folder is located — and list inside the names of the files you want Git to ignore.

Commits

Files prepared for saving can be saved with the `git commit` command. It opens a text editor, where we need to enter an explanation for the commit (`commit message').



```

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#	modified:   file1.md
#
# Untracked files:
#	file2.md
#

```

With `-m`, we can add a message right away:

```
git commit -m "do something"
```

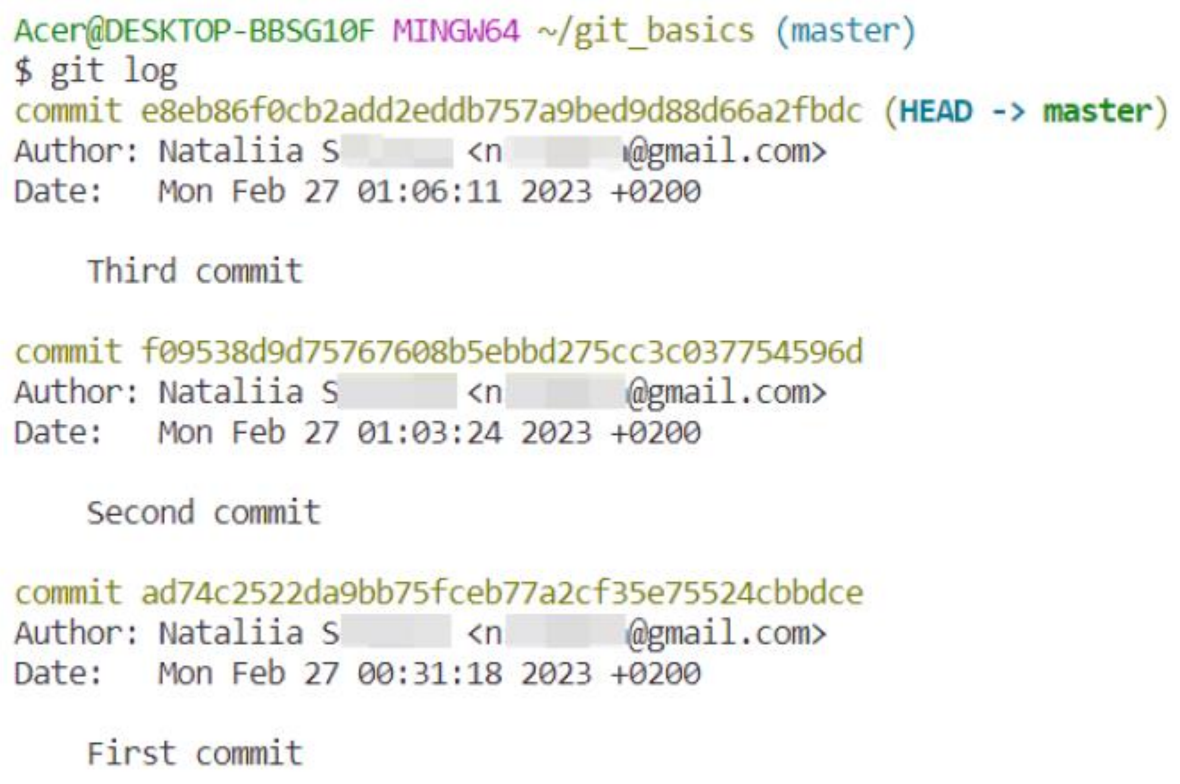
Thus, `git commit` or `git commit -m "message"` lets us save all files that have been added to the staging area. We can also skip the staging step and **commit all modified files directly** using `commit -a`:

```
git commit -am "do something"
```

...but it comes with the risk of commiting some files unintentionally.

History of Commits

Commits are stored in the history of commits, which we can see with the `git log` command:



```
Acer@DESKTOP-BBSG10F MINGW64 ~/git_basics (master)
$ git log
commit e8eb86f0cb2add2eddb757a9bed9d88d66a2fbdc (HEAD -> master)
Author: Nataliia S[redacted] <n[redacted]@gmail.com>
Date:   Mon Feb 27 01:06:11 2023 +0200

    Third commit

commit f09538d9d75767608b5ebbd275cc3c037754596d
Author: Nataliia S[redacted] <n[redacted]@gmail.com>
Date:   Mon Feb 27 01:03:24 2023 +0200

    Second commit

commit ad74c2522da9bb75fceb77a2cf35e75524cbbdce
Author: Nataliia S[redacted] <n[redacted]@gmail.com>
Date:   Mon Feb 27 00:31:18 2023 +0200

    First commit
```

This particular command shows hash, the author's name and email, the date of saving, and a commit message. If you want more info — **view this doc**.

We can switch between commits using branches or a commit hash with the `git checkout hash-or-branch` command. Usually, the first seven characters of hash are enough to switch between commits.

Aliases

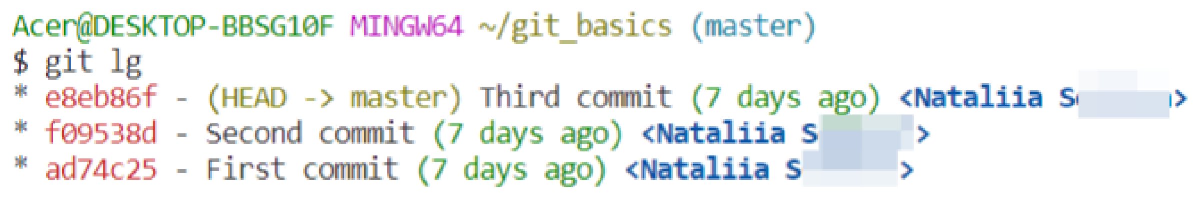
Every Git command can be "renamed" or "abbreviated" using aliases. All aliases are stored in the hidden `~/.gitconfig` file. To add an alias, use the following command:

```
git config --global alias.alias_name command_name
```

It's possible to modify the `.gitconfig` not only through the command line, but also by opening this file in any text editor. We suggest making the following changes:

```
[user]

  name = Your_Name Your_Surname

  email = your@email

[alias]

  ci = commit

  co = checkout

  br = branch

  st = status
  lg = log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --branches

[core]

  pager =

  editor = nano
```

As we can see, the file contains an alias for the `git log` command. **It makes the commit history more readable.**



```
Acer@DESKTOP-BBSG10F MINGW64 ~/git_basics (master)
$ git lg
* e8eb86f - (HEAD -> master) Third commit (7 days ago) <Nataliia S[redacted]>
* f09538d - Second commit (7 days ago) <Nataliia S[redacted]>
* ad74c25 - First commit (7 days ago) <Nataliia S[redacted]>
```

Basic Commands

Command	Description
`git config --global ...`	Sets global Git parameters
`git init`	Initializes new Git repo in current folder
`git status`	Shows info about the repo
`git add ./path/to/files`	Prepares file changes for saving
`git restore --staged file_name`	Removes changes in `file_name` from prepared for commit
`git commit`	Saves prepared changes and open the text editor to enter a commit message
`git commit -m "do something"`	Saves prepared changes with a given message
`git commit -am "do something"`	Prepares all modified files and save with a given message
`git log`	Shows the commits history
`git lg`	Custom alias we added to the `log`

Command	Description
`git branch`	Shows all branches
`git branch branch_name`	Creates a new `branch_name` branch
`git branch -D branch_name`	Deletes the `branch_name` branch
`git checkout hash-or-branch`	

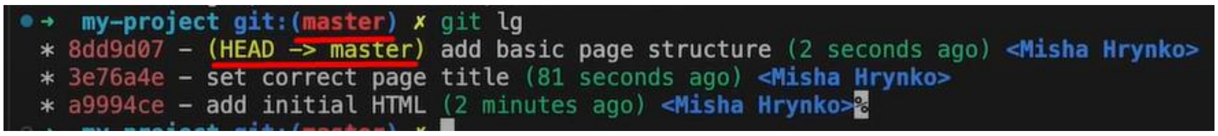
Working With Branches

Once we initialize a repository, the terminal displays the currently active branch next to the directory name. Said branch serves as a convenient label for a commit, meaning: we don't have to remember its hash. It facilitates working on multiple tasks simultaneously and helps in avoiding conflicts until changes are merged.

Typically, the default branch is named `main` (or `master` in older repositories), but other branches can be used as well. The `main` branch usually contains the stable version of the product, such as a website or app, that's ready for release.

Branch Management

Say we've just activated the repository, and created three commits. The active branch — in which all the development happens — is marked with the `HEAD` pointer in the history. Like so:



```
● → my-project git:(master) ✗ git lg
  * 8dd9d07 - (HEAD -> master) add basic page structure (2 seconds ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (81 seconds ago) <Misha Hrynko>
  * a9994ce - add initial HTML (2 minutes ago) <Misha Hrynko>%
```

We can show all branches in the terminal:

```
git branch
```

Create a new branch on the current commit:

```
git branch branch_name
```

Switch to another branch:

```
git checkout branch_name
```

```
git switch branch_name
```

Create a branch and switch to it right away:

```
git checkout -b branch_name
git switch -c branch_name
```

Rename the currently active branch:

```
git branch -m new_name
```

And delete an inactive branch with the `-d` flag:

```
git branch -d branch_name
```

If there are several branches on one commit, then when saving, the active branch is moved to a new commit, while the inactive ones remain on the previous one. To add changes from the selected branch to the active one, run:

```
git merge branch_name
```

When we're merging branches:

- If the active branch is behind the specified branch in the commit history, it will just move forward to the commit where the specified branch currently is.
- Otherwise, a new commit is formed incorporating all the changes from both the active and the specified branches, and the active branch advances to this new commit.
- However, if both branches have made changes to the same lines of code, a conflict occurs. We'll then need to manually decide which version of the changes to include in the final merge.

Reverting Changes

To revert the changes made by a specific commit and create a new commit with the opposite changes, use `revert hash`. In case you're wondering, `hash` represents the unique identifier of the commit we wish to revert:

```
git revert hash
```

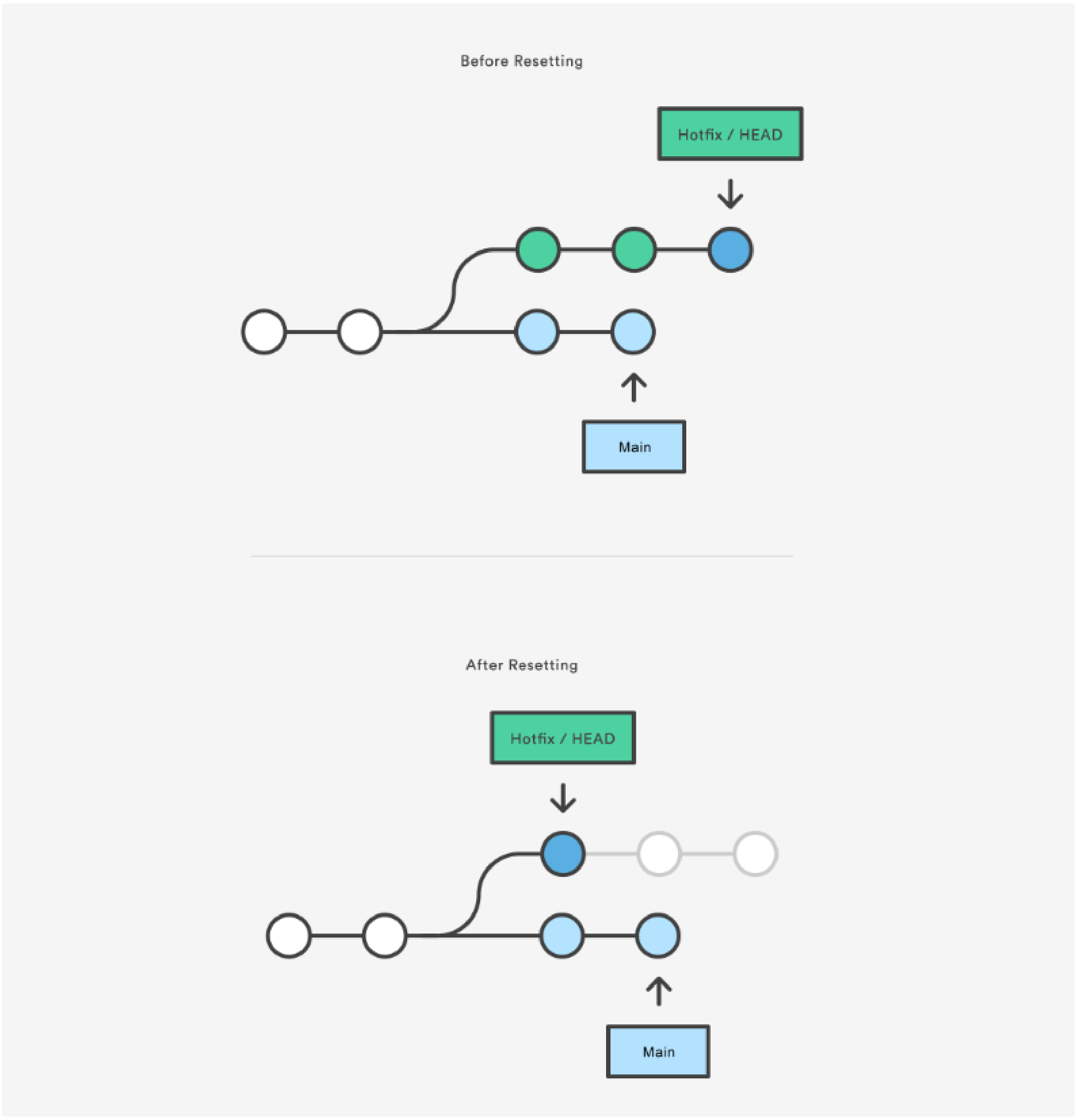
We can also move a branch to a particular commit with `reset hash`. Though, since we alter the commit history that way, better use it on non-public branches to prevent merge conflicts:

```
git reset hash
```

Finally, to discard all changes applied up to a certain point, including file modifications, we can apply the `--hard` option. Below command rolls the active branch back by two commits and removes all connected changes:

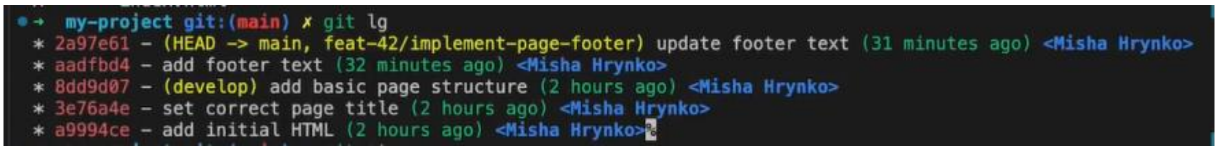
```
git reset HEAD^^ --hard
```

Alternatively, we can replace `HEAD^^` with the specific commit hash we'd like to revert to.



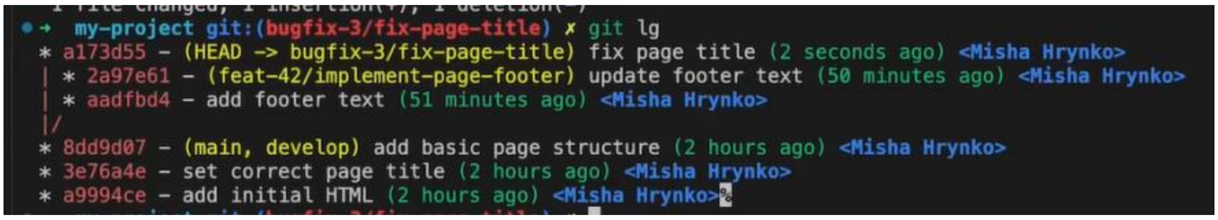
Linear and Non-Linear History

Sometimes it's convenient to keep the commit history linear (wihtout branches). Here's an example where all saves are lined up one after another:



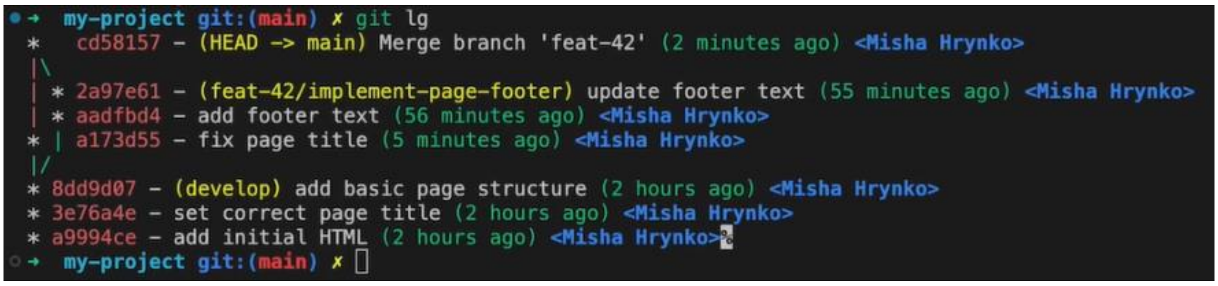
```
● → my-project git:(main) ✗ git lg
  * 2a97e61 - (HEAD -> main, feat-42/implement-page-footer) update footer text (31 minutes ago) <Misha Hrynko>
  * aadfbd4 - add footer text (32 minutes ago) <Misha Hrynko>
  * 8dd9d07 - (develop) add basic page structure (2 hours ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (2 hours ago) <Misha Hrynko>
  * a9994ce - add initial HTML (2 hours ago) <Misha Hrynko>%
```

We get a branched history by making two saves based on one commit:



```
● ➜  my-project git:(bugfix-3/fix-page-title) ✗ git lg
  * a173d55 - (HEAD -> bugfix-3/fix-page-title) fix page title (2 seconds ago) <Misha Hrynko>
  | * 2a97e61 - (feat-42/implement-page-footer) update footer text (50 minutes ago) <Misha Hrynko>
  | * aadfbd4 - add footer text (51 minutes ago) <Misha Hrynko>
  |/
  * 8dd9d07 - (main, develop) add basic page structure (2 hours ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (2 hours ago) <Misha Hrynko>
  * a9994ce - add initial HTML (2 hours ago) <Misha Hrynko>%
```

Merging branches breaks the linearity, too:

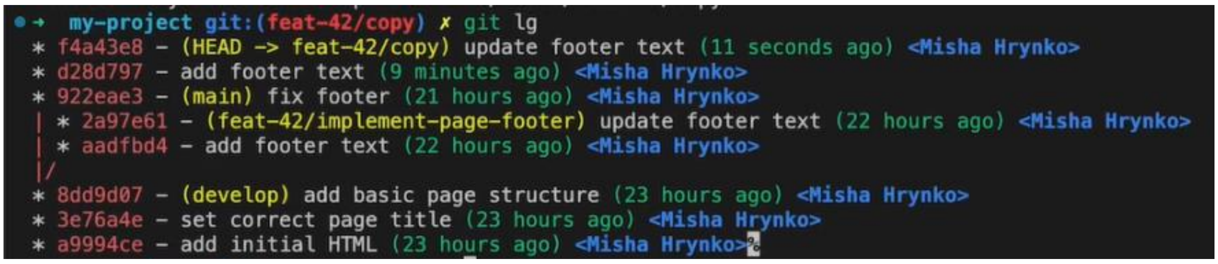


```
● ➜  my-project git:(main) ✗ git lg
  *   cd58157 - (HEAD -> main) Merge branch 'feat-42' (2 minutes ago) <Misha Hrynko>
  |\
  | * 2a97e61 - (feat-42/implement-page-footer) update footer text (55 minutes ago) <Misha Hrynko>
  | * aadfbd4 - add footer text (56 minutes ago) <Misha Hrynko>
  * | a173d55 - fix page title (5 minutes ago) <Misha Hrynko>
  |/
  * 8dd9d07 - (develop) add basic page structure (2 hours ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (2 hours ago) <Misha Hrynko>
  * a9994ce - add initial HTML (2 hours ago) <Misha Hrynko>%
○ ➜  my-project git:(main) ✗ 
```

To keep the history linear, we can rebase the branch before the merge:

```
git rebase branch_name
```

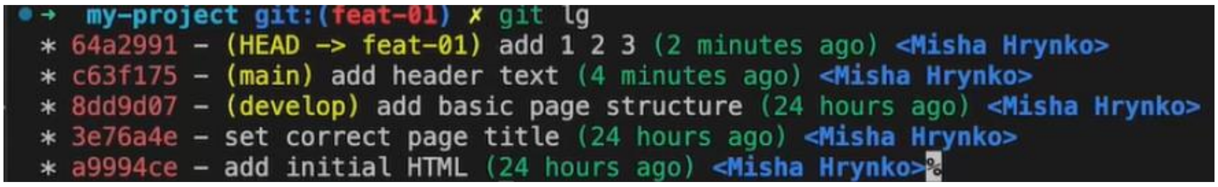
This command rebases the active branch starting from the one specified, duplicating any missing commits up to the most recent one. For example, using `git rebase main` would copy commits `aadfbd4` and `2a97e61` after commit `922eae3`, where the main branch currently is, and then move the active `feat-42/copy` branch to the latest commit.



```
● ➜  my-project git:(feat-42/copy) ✗ git lg
  * f4a43e8 - (HEAD -> feat-42/copy) update footer text (11 seconds ago) <Misha Hrynko>
  * d28d797 - add footer text (9 minutes ago) <Misha Hrynko>
  * 922eae3 - (main) fix footer (21 hours ago) <Misha Hrynko>
  | * 2a97e61 - (feat-42/implement-page-footer) update footer text (22 hours ago) <Misha Hrynko>
  | * aadfbd4 - add footer text (22 hours ago) <Misha Hrynko>
  |/
  * 8dd9d07 - (develop) add basic page structure (23 hours ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (23 hours ago) <Misha Hrynko>
  * a9994ce - add initial HTML (23 hours ago) <Misha Hrynko>%
```

If we want to merge portable commits, we can `rebase` with the `-i` (interactive) option:

```
git rebase -i branch_name
```



```
● ➜  my-project git:(feat-01) ✗ git lg
  * 64a2991 - (HEAD -> feat-01) add 1 2 3 (2 minutes ago) <Misha Hrynko>
  * c63f175 - (main) add header text (4 minutes ago) <Misha Hrynko>
  * 8dd9d07 - (develop) add basic page structure (24 hours ago) <Misha Hrynko>
  * 3e76a4e - set correct page title (24 hours ago) <Misha Hrynko>
  * a9994ce - add initial HTML (24 hours ago) <Misha Hrynko>%
```

Remember that all changes in history should be made before history becomes public to avoid conflicts.

Command Overview

Below are all commands we've used in this lesson:

Command	Description
`git branch`	Shows all branches
`git branch new_branch`	Creates a new branch with the `new_branch` name
`git branch -m new_name`	Renames current branch with the `new_name`
`git branch -d branch_name`	Delete a not active branch
`git checkout branch_name`	Switch to the `branch_name` from the current branch
`git switch branch_name`	Switch to the `branch_name` from the current branch
`git checkout -b new_branch`	Create and switch to the `new_branch` right away
`git switch -c new_branch`	Create and switch to the `new_branch` right away
`git merge branch_name`	Merge the `branch_name` to the current branch
`git revert hash_of_commit`	Create a new commit with reverted changes to commit with the `hash_of_commit`
`git reset hash_of_commit`	Rollback the current branch to the commit with the `hash_of_commit`
`git reset HEAD^^ --hard`	Rollback the current branch into 2 commits and delete all changes in files
`git rebase branch_name`	Rebase changes from the current branch under the `branch_name`. All commits of the current branch will be rebased under the `branch_name`
`git rebase -i branch_name`	Rebase changes from the current branch under the `branch_name`. Only chosen commits of the current branch will be rebased under the `branch_name`

Command	Description
`git restore file_name`	Delete not prepared changes to the commit in the `file_name`
`git restore --stage file_name`	Delete prepared changes to the commit in the `file_name`
`git restore -SW`	Delete all changes in the files (prepared to commit and not-prepared to commit)
`git commit --amend`	Replace the current commit with the other commit

Working with Remote Repositories on GitHub

Remote repositories are hosted on servers that can be accessed via the internet, acting as central points for storing, sharing, and managing code. This setup allows developers to work together from various locations, ensuring code backup, progress tracking, and support for continuous integration and delivery processes.

GitHub is a popular platform for hosting these repositories, providing a range of tools and features to facilitate version control and project management, which we will utilize for certain tasks. Other notable platforms include **GitLab** and **Bitbucket**, although we won't cover the two.

Registering on GitHub

To start using GitHub, you have to create an account:

1. Visit **GitHub's website**.
2. Click on "Sign up," provide your details, and follow the on-screen instructions to finish the registration.

We also recommend opening `Your profile` page, pressing `edit profile`, filling in your name, and adding a photo.

Creating an Empty Repository on GitHub

To create a new repo on GitHub:

1. Log in and click `New repository` on the dashboard.
2. Name your repository, add a description (optional), and choose public or private.
3. If you want the repo to have an initial commit with a `readme.md`, select the checkbox `Add a README file`.

4. Press the `Create repository` button.

Cloning the Repository

To clone this repository to your local machine, run the next command in the parent folder:

```
git clone https://github.com/username/repository.git
```

Replace `username/repository.git` with your repository's path. The folder will have the same name as your repo.

Connecting a Local Repository to GitHub

If you have an existing local repository you wish to connect to a new GitHub repository:

1. Create an empty GitHub repository (without a README file).
2. Connect the remote repo under some alias (here it is `origin`) by running in your local repository directory:

   ```
   git remote add origin https://github.com/username/repository.git
   ```

3. Push the `main` branch to the remote repo. This requires you to authorize on GitHub via a command line (or credentials manager), which we'll cover in the next section:

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

Once done, you'll achieve the same result as if you cloned a repo from GitHub. The flag `-u` syncs the specified branch with the same branch of the remote repo.

Generating a Personal Access Token

To authorize the `git push` command, you need to generate a **Personal Access Token** (PAT):

1. Log in to GitHub, click on your profile icon, and go to "Settings."
2. Navigate to "Developer settings" > "Personal access tokens."
3. Switch to `Tokens (classic)` in the left sidebar.
4. Click "Generate a personal access token".
5. Input some hint for the token.
6. Choose `No expiration` in the `Expiration` dropdown.
7. Select all checkboxes in the `repo` section (or the other scopes or permissions you need).
8. Press the `Generate Token` button.
9. Copy the token to use it as a password for authentication on GitHub from your terminal. (❕ If you don't copy it now, you'll need to delete the token and create a new one).

For a detailed guide, refer to the **official GitHub documentation on creating a PAT**.

Typical Feature Development Flow

A common workflow for adding a new feature might look like this:

1. Update your local main branch: `git checkout main && git pull origin main`.
2. Create a new branch for the feature: `git checkout -b feature/name`.
3. Make changes, commit them: `git commit -am "Add feature"`.
4. Push the feature branch to GitHub: `git push -u origin feature/name`.
5. Create a pull request on GitHub from `feature/name` into `main`.
6. After review, merge the pull request on GitHub.

Branch Naming Conventions

Adopting naming conventions for branches helps organize work and indicates the branch's purpose:

- **Feature branches** (`feature/name`) for new features or additions.
- **Bugfix branches** (`bugfix/name`) for fixing bugs in existing features.
- **Hotfix branches** (`hotfix/name`) for urgent fixes to production issues.

These conventions streamline collaboration and help quickly identify the role and status of various tasks within a project.

Useful GIT Commands to Work with Remote Repos

Command	Description
`git clone repo_url`	Clones an existing repo
`git remote`	Shows a list of remote servers
`git remote -v`	Shows a list of remote servers with URLs
`git remote show server_alias`	Shows detailed info about a server
`git remote add server_alias repo_url`	Adds a new alias for a server with given `repo_url`
`git remote remove server_alias`	Disconnects from a server

Command	Description
`git push server_alias branch_name`	Sends the `branch_name` to `server_alias`
`git fetch server_alias branch_name`	Downloads the changes without merging them to the local branch
`git pull server_alias branch_name`	Downloads and merges changes