Getting started with git

git

Contents

- Introduction
    - What is version control
    - What is git
- Setting up git
    - Installation
    - Configuring git
- Git Basics
    - Common terminologies
    - Working with a git repository
- Conclusion
    - Recap

🔹 Introduction

Git is everywhere, literally. As a developer, understanding and using git is **not** optional, regardless of your specialization chances are you will definitely come across and need to use git. This article explains why git is so important and ubiquitous with some easy to follow tips on setting up and using git on your local machine.

What is version control?

Before we define git, let's look at the problem git solves, doing so will help us better understand and appreciate it. One of the core uses of git is version control, what is that? **Version control** is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Imagine this scenario, you've finished writing your thesis and saved it as final.docx but discovered you had to make a minor tweak and do so, then you save the new version as final_v2.docx then another and another until you get to final_v8. Now you have 8 files in your folder all just a variation of your initial file.

Wouldn't it be nice if you instead just had 1 file that contains all those versions that you could easily switch to whenever you please? That is very much possible with version control systems (VCS).

What is git?

The earlier scenario was really basic and as a developer in the real world you wouldn't only be working with word files. Instead you'd probably be working with tens and hundreds of files with other developers and need to keep track of each file version. That is where git comes in.

Git is a **distributed version control system (DVCS)** that allows multiple developers to work on a project simultaneously without interfering with each other's work. In a DVCS like Git, every contributor has a full copy of the entire codebase and its history on their local machine, enabling offline work and better collaboration.

Setting up git

Installation

To download and install Git on your local machine via git's website:

1. Visit git-scm.com
2. Download the version for your OS (Windows, Mac, or Linux)
3. Install it using the default settings
4. Open your terminal (Bash, Zshell or Powershell) and run git --version:
5. If you see a version number, it means the installation was successful.

```
$ git --version
git version 2.49.0
```

Alternatively, you can do the installation on your terminal via *Powershell* on Windows, *Zshell* on MacOS and *Bash* on Linux (Debian based) respectively by running:

```
$ winget install --id Git.Git -e --source winget
```

```
$ brew install git
```

```
$ sudo apt update
$ sudo apt install git
```

Git configurations

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer.

Git comes with a tool called **git config** that lets you get and set configuration variables that control all aspects of how Git looks and operates. This tool accepts 3 flags which determines the level and scope of your configurations. The flags are:

- --system: applies the configuration to all users of that computer
- --global: applies the configuration to the current user and persists in all git repositories created by that user
- --local: applies the configuration to a single repository.

Each level overrides the previous level with --local having the highest precedence.

Configuring your identity

The first thing you should do when you install Git is to set your username and email address. This is important because every Git commit uses this information to identify users that made the commit, and it's immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email "johndoe@example.com"
```

Configuring your editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message (e.g when committing with git commit). If not configured, Git uses your system's default editor.

I would recommend using nano for this. Nano is a simple command-line text editor for UNIX systems that is easy to set up and use and works well for this purpose.

It often comes pre-installed on Linux and MacOS. To install it on Windows yourself, use Powershell or cmd and run:

```
$ winget install GNU.Nano
```

Regardless of your operating system, you can confirm if you have nano installed on your machine by running: nano --version. You might want to watch a brief tutorial about using nano if you're not familiar with it.

With nano in your machine, you can now configure it as your git editor by running:

```
$ git config --global core.editor "nano"
```

Configuring your default branch name

By default Git will create a branch named master when you create a new git repository with git init. but you can set a different name for the initial branch. Usually developers go with main as the default branch name

To set main as the default branch name run:

```
$ git config --global init.defaultBranch main
```

Getting help

If you ever need help with any git command you can run: git <command> --help

```
$ git config --help
```

Git Basics

There are over 250 git commands and I don't think anyone, even the creator of git Linus Torvalds, knows all of them. Luckily for us, we don't have to know or use most of these commands. I think the pareto principle also works here. 80% of the work you do would require knowing just 20% of these commands. In this section, we will learn some of the common git terminologies and cover some of the frequently used commands.

Common git terminologies

Repository (repo): A storage location for your project's files and their revision history. This is the folder containing all your project files as well as a hidden .git folder that tracks changes in every file of that folder.

Commit: A snapshot of your project at a specific time. Think of it as a "save point" in a video game that you can easily go back to and load all your progress, collectibles and settings exactly how it was saved.

Branch: A branch is a parallel version of your repo where you can make changes and add commits without affecting the main project.

Merge: To combine changes from one branch with another. A way to bring the changes made from another branch to your main project.

Remote: A remote is a cloud copy of a git repository, usually hosted on services like GitHub, GitLab or Bitbucket. Think of it as the Google drive backup of your repo with additional collaborative functionalities. This is the central hub developers use to get and send changes on the same project.

Clone: To download a copy of a remote repo with all the files, commits, and branches to your local machine.

There are many more git terms but these are the ones you'll often see in the wild and understanding them helps to understand the others better.

Working with a git repository

In this section we will learn how to initialize a git repo, commit changes to the repo, view our commit history and undo/edit commits. Before all that, let's understand how git categorizes our files into three main sections namely;

- **working directory (working tree):** Your project folder where you actively edit files, git detects changes here but doesn't track them yet, until you stage and commit the files.

  If a file in the working tree has never been committed to git before, it is marked as *untracked,* but if it has been committed before, it is marked as *tracked.*

  If a tracked file has not changed since the last time it was committed, it is marked as *unmodified* but if it has changed since the last commit, it is marked as *modified*.

- **staging area (index):** The area where you select and add files that are ready to be committed when you run git add. Think of it as a waiting room for files to be stored/committed by git after you run git commit
- **repository (.git folder):** This is where git permanently stores the final snapshot of your project as of that point and assigns a 40 character SHA1 hash alongside other details such as the author email and username, date and other information.

Most of the git commands we carry out simply just move our files from one of these three sections to the other.

Creating/getting a git repo

There are basically two ways of obtaining a git repo

I. By navigating to a directory/folder that has not been initialized as a git repo and running git init via the terminal

```
$ mkdir ~/Desktop/my-project
$ cd ~/Desktop/my-project
$ git init
```

Doing this creates a hidden .git folder inside my-project directory and enables git's functionalities for the folder essentially turning it into a git repository.

II. By cloning a remote repo with git clone <remote-url>

```
$ git clone https://github.com/rowleks/my-project
```

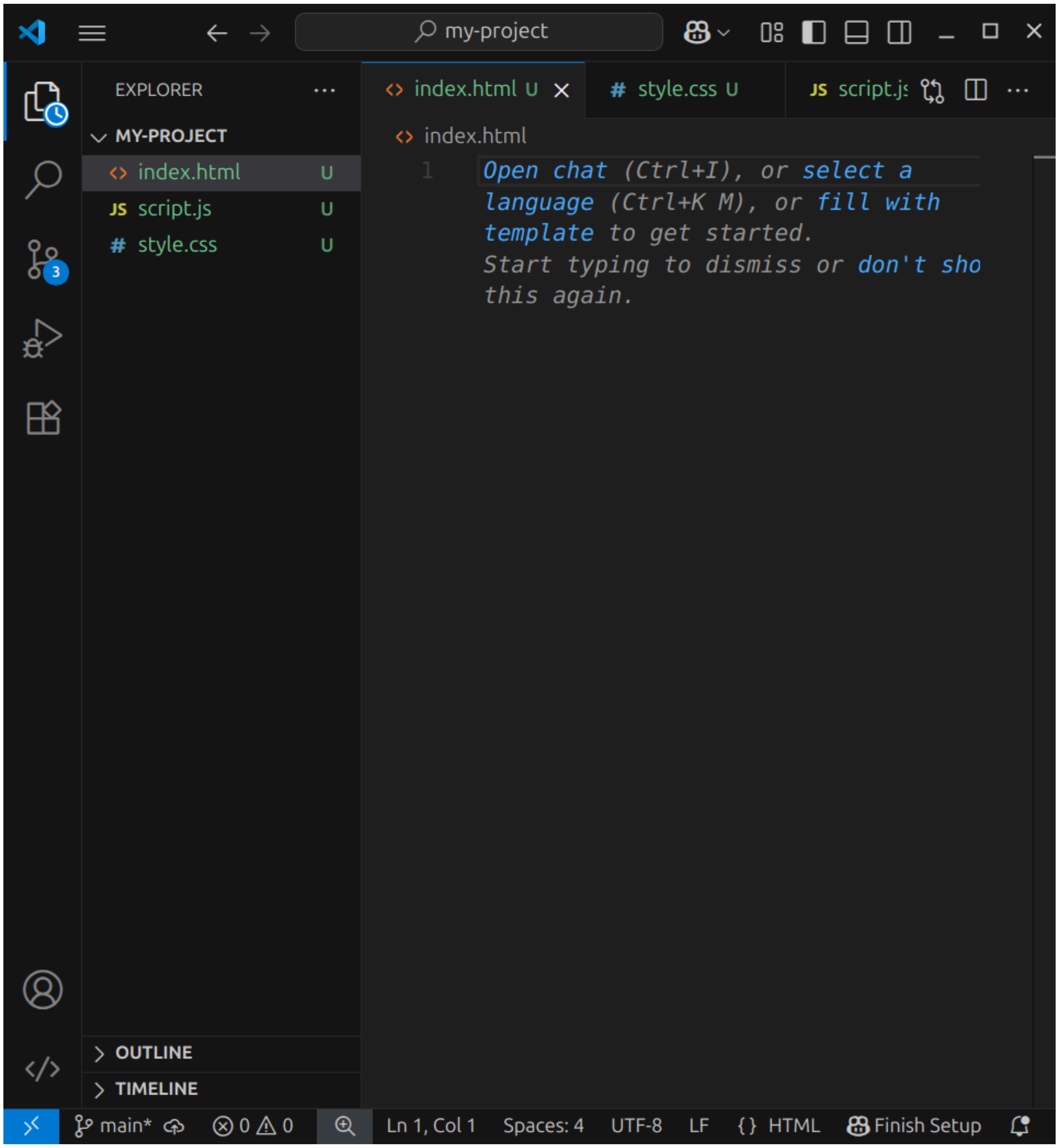
Doing this would download the remote repo my-project from GitHub into the directory you were currently on before running the git clone command.

Committing changes to the repo

Basically, the git workflow goes like this:

1. You create/modify files in your working directory
2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

To illustrate these steps better, let's open our project folder in VS code and create 3 files; *index.html, style.css* and *script.js*

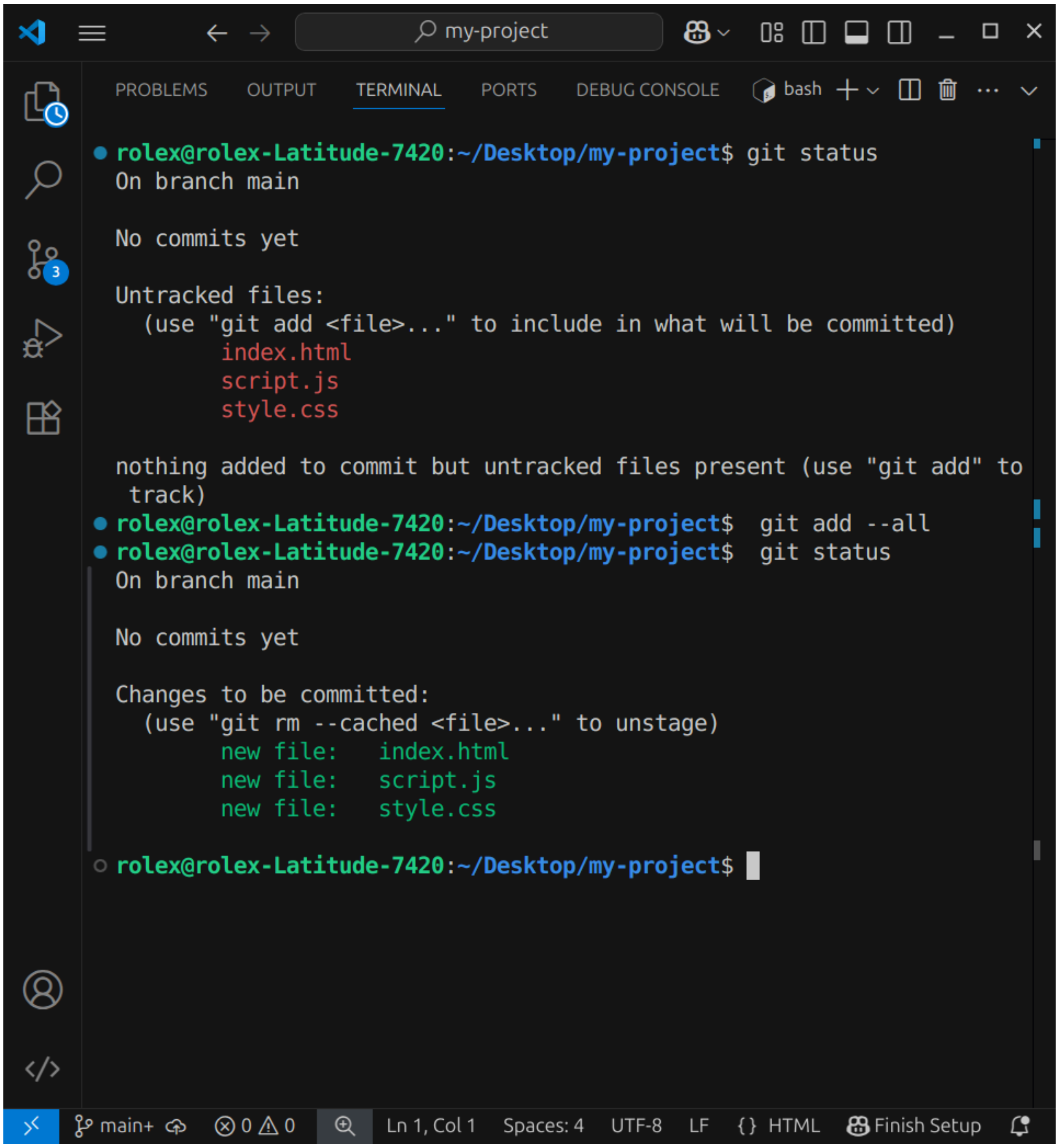


```
Open chat (Ctrl+I), or select a
language (Ctrl+K M), or fill with
template to get started.
Start typing to dismiss or don't sho
this again.
```

Because we have run git init on the folder earlier, git was able to identify that these 3 new files are untracked hence the U we see next to the filenames.

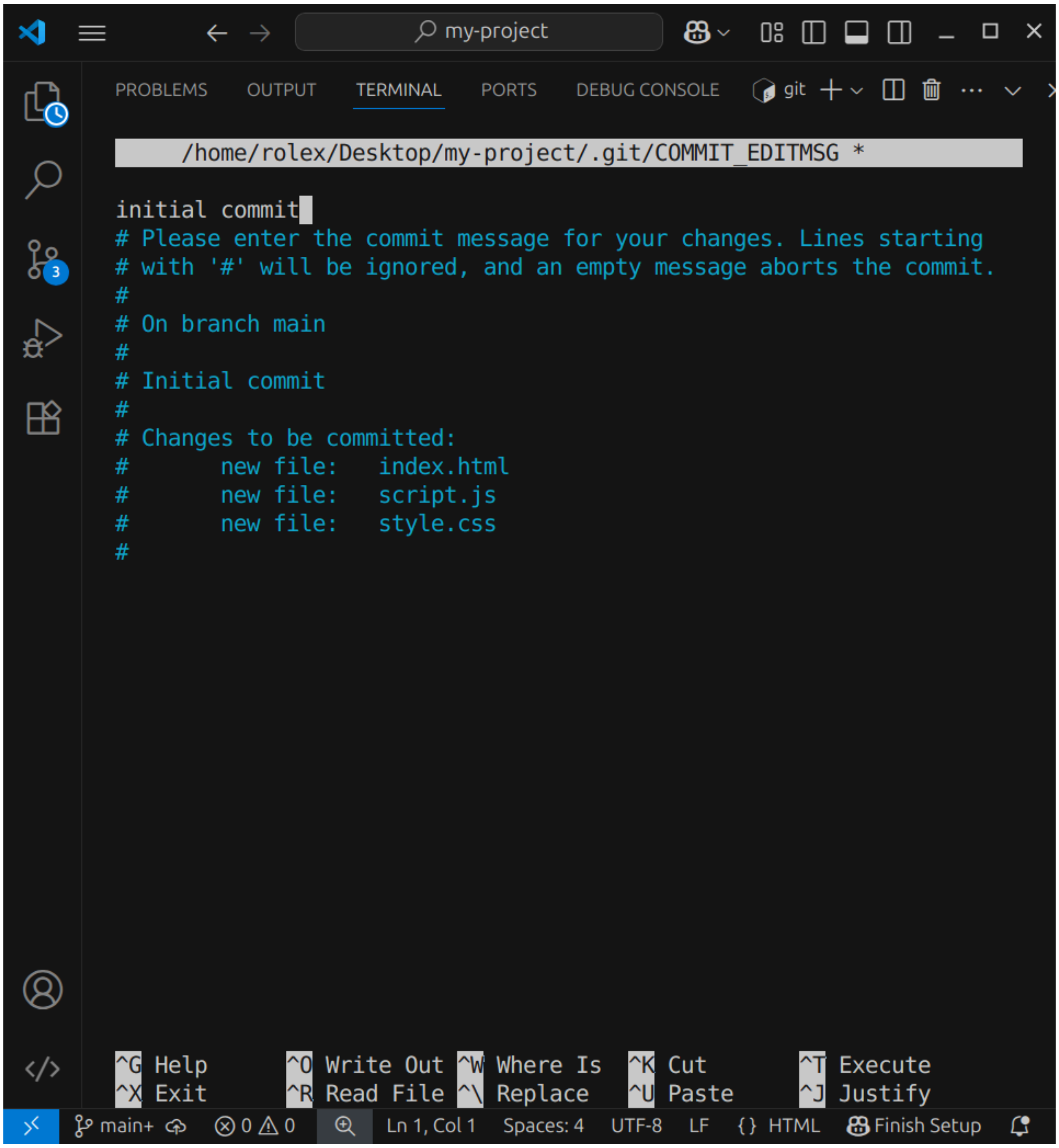
We can use git status to have git show us a summary of untracked, modified and staged files in our working tree

Next we will let git track these files by staging and committing them using git add --all and git commit. We can do these using the built-in VS code terminal which works the same as using other terminals like the native Bash, Zsh or Powershell.



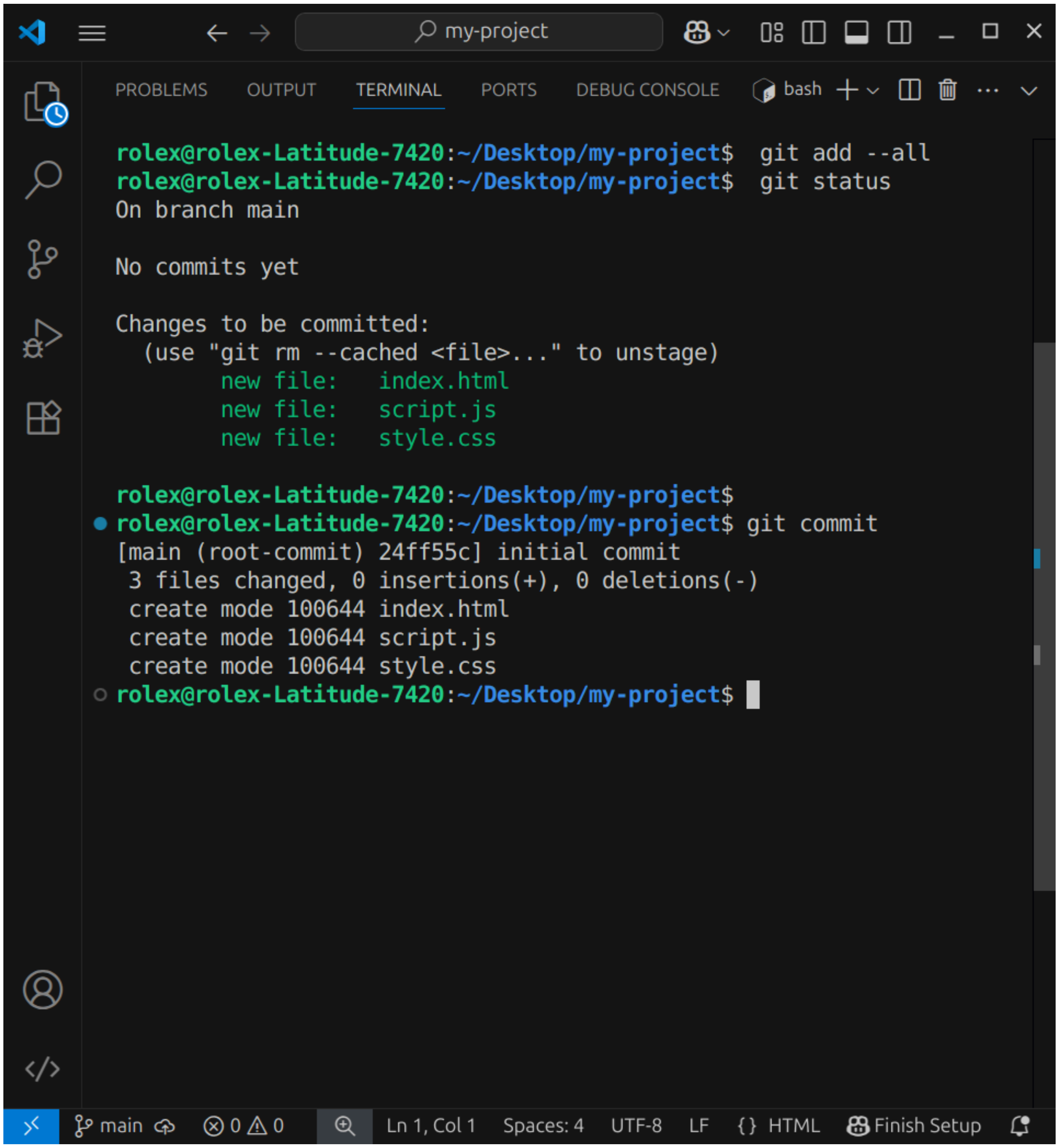
```
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html
        script.js
        style.css

nothing added to commit but untracked files present (use "git add" to
 track)
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git add --all
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
        new file:   script.js
        new file:   style.css

rolex@rolex-Latitude-7420:~/Desktop/my-project$ 
```

As shown in the terminal above, when I first ran git status, it listed the three untracked files and after running git add --all, it added all the files to the staging area, now running git status again showed us the three staged files waiting to be committed.



```
     /home/rolex/Desktop/my-project/.git/COMMIT_EDITMSG *

initial commit
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
#       new file:   index.html
#       new file:   script.js
#       new file:   style.css
#

^G Help       ^O Write Out ^W Where Is  ^K Cut       ^T Execute
^X Exit       ^R Read File ^\ Replace   ^U Paste     ^J Justify
```

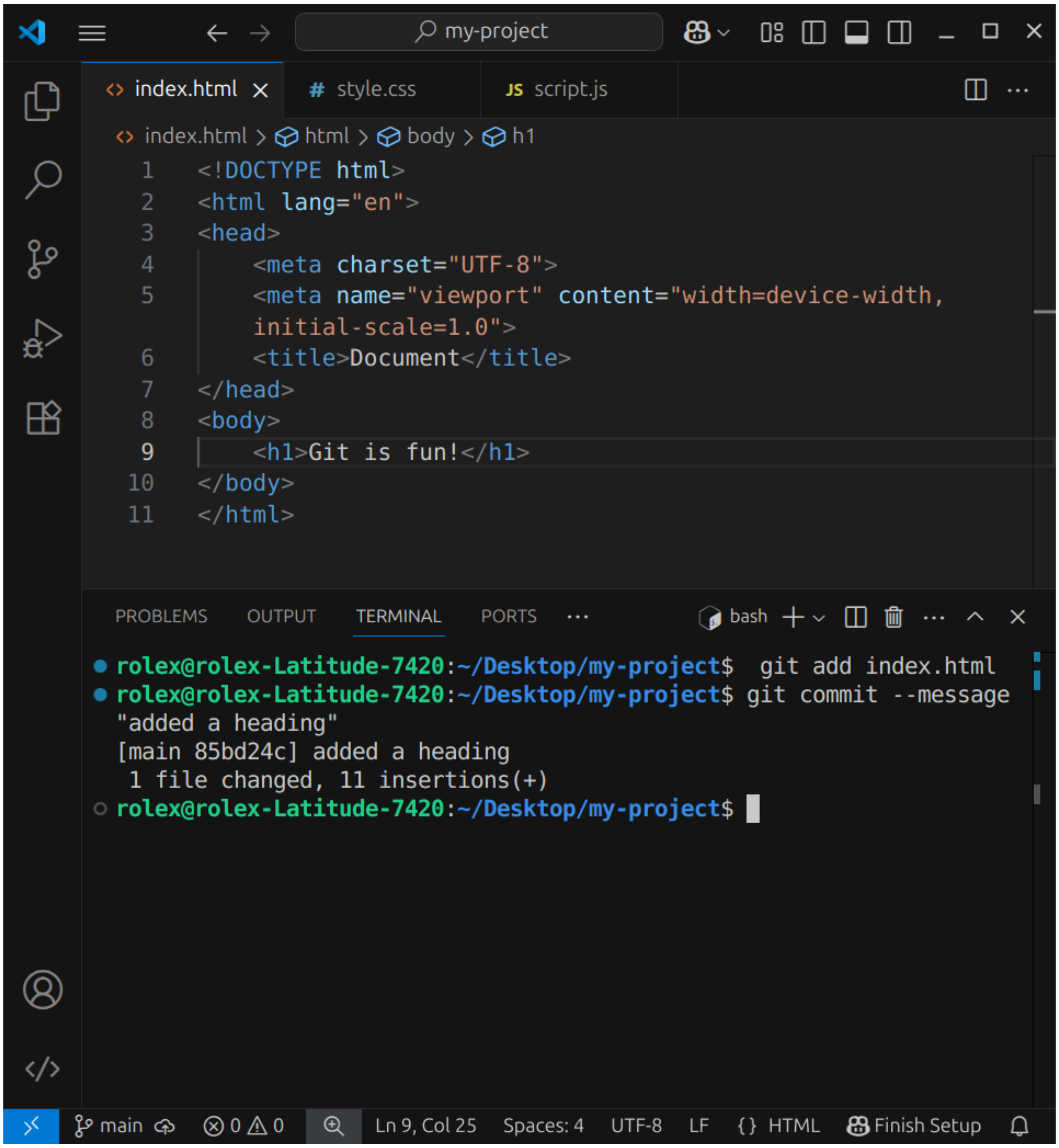
In the screenshot above, we run git commit and git will open our editor nano, the one we configured earlier (with git config --global core.editor "nano"). Finally we ignore all the commented out text that begins with # and type our commit message "initial commit" and press **CTRL + S** to save and **CTRL + X** (Not ^ X) to exit the editor.



```
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git add --all
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
        new file:   script.js
        new file:   style.css

rolex@rolex-Latitude-7420:~/Desktop/my-project$
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git commit
[main (root-commit) 24ff55c] initial commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
 create mode 100644 script.js
 create mode 100644 style.css
rolex@rolex-Latitude-7420:~/Desktop/my-project$ 
```

Git brings us back to our terminal, shows us an abbreviation of the commit hash (24ff55c) with the message and that's it. Now git will always track those 3 files from this point forward.

Committing modified changes to the repo



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Git is fun!</h1>
</body>
</html>
```

```
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git add index.html
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git commit --message
"added a heading"
[main 85bd24c] added a heading
 1 file changed, 11 insertions(+)
rolex@rolex-Latitude-7420:~/Desktop/my-project$ 
```

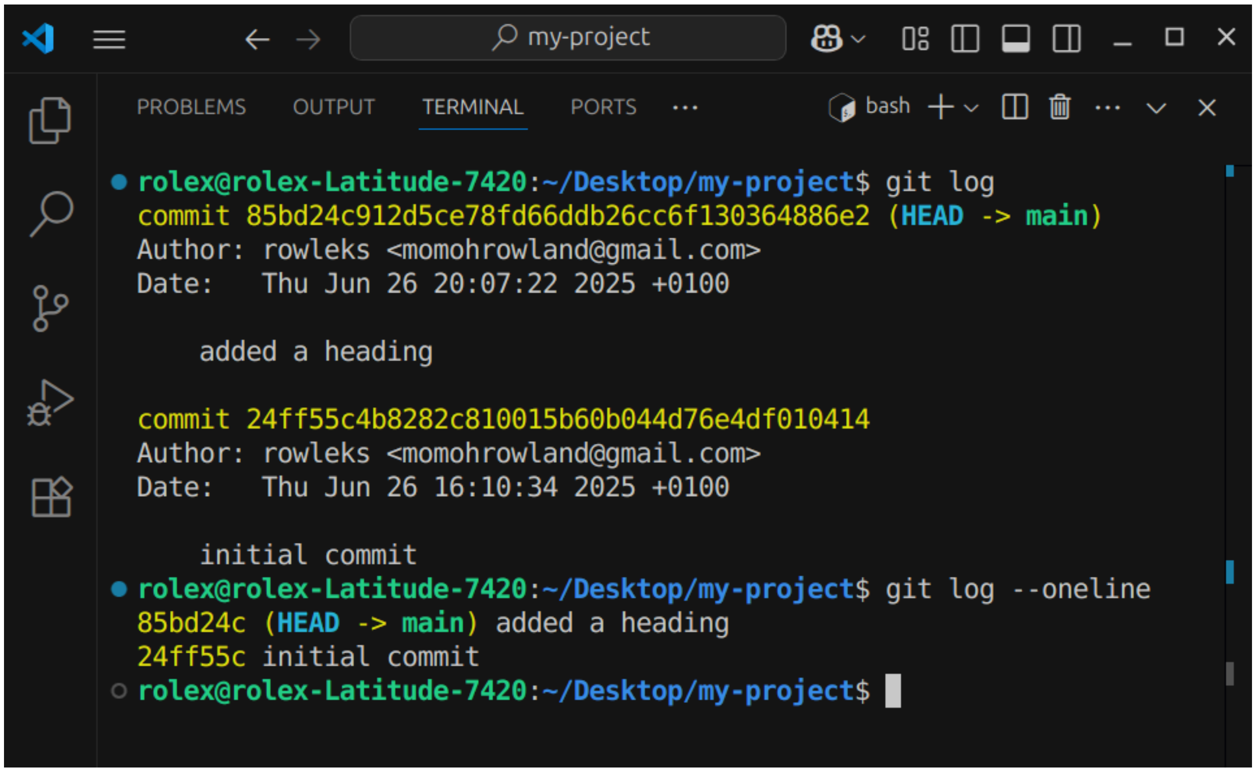
In the screenshot above, we modified only the *index.html* file and populated it with a basic html structure with a heading that says “Git is fun!”.

In the terminal we ran two git commands:

- **git add index.html:** To stage the changes for committing. We could have still run git add --all and git would have been smart enough to omit the unmodified files but specifying the file we want gives us more control over the files we are staging.
- **git commit --message “added a heading”:** This is a shortcut to bypass using the nano editor to type our commit message instead of --message you can also just use -m to achieve the same result.

Viewing commit log/history

So far we have made two commits, the first being the initial commit for enabling git track our files. The second is a commit containing the modified index.html. We use the git log command to view our commit history



```
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git log
commit 85bd24c912d5ce78fd66ddb26cc6f130364886e2 (HEAD -> main)
Author: rowleks <momohrowland@gmail.com>
Date:   Thu Jun 26 20:07:22 2025 +0100

    added a heading

commit 24ff55c4b8282c810015b60b044d76e4df010414
Author: rowleks <momohrowland@gmail.com>
Date:   Thu Jun 26 16:10:34 2025 +0100

    initial commit
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git log --oneline
85bd24c (HEAD -> main) added a heading
24ff55c initial commit
rolex@rolex-Latitude-7420:~/Desktop/my-project$
```

From the above screenshot, we use git log to view detailed information about our commits which includes the 40 character commit hash, the author, date and the commit message.

Sometimes those other information aren't necessary and we just want to see a concise version of our commit history. We include the --oneline flag to the git log command to shorten the commit hash to the first 7 digits and display just the commit message.

Editing/Amending a commit

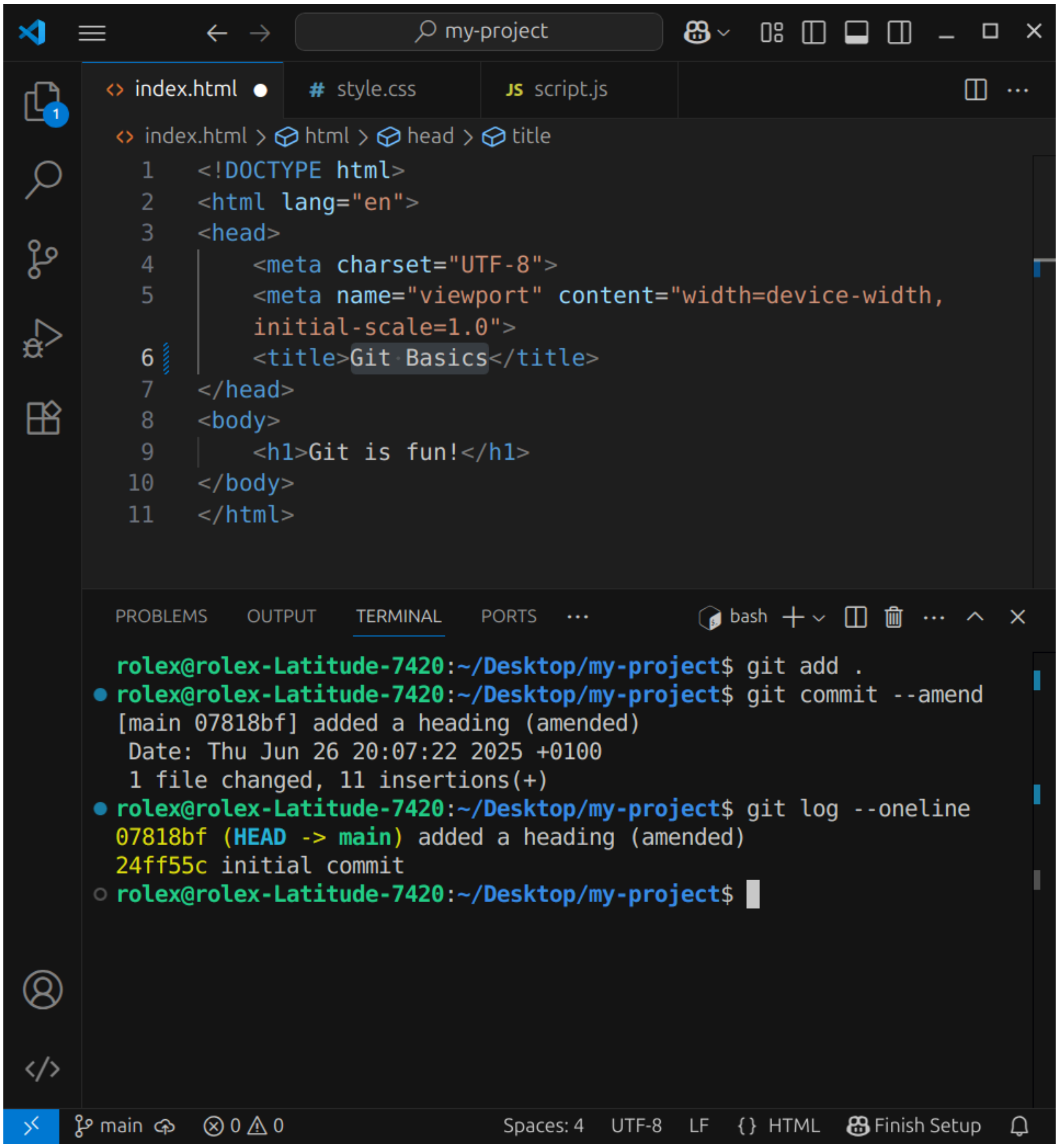
Sometimes we commit too early and realise we made a mistake or may want to add more files/contents to our last commit. Git provides the git commit --amend command to help us edit the last commit.

In our example, our last commit was "added a heading" with commit hash 85bd24c Assuming we wanted to change just the title tag from "Document" to "Git Basics", we could do that and add a new commit with just this change but that's not really efficient is it? Creating a new commit just for a single change that could have been added to the last commit.

This is a perfect scenario for using the git commit --amend command, here's how we use it:

- Make the changes we wish and stage them using git add --all
- Include the --amend flag to git commit to let git know that we are replacing the last commit and not creating a new one.
- Git fires up the editor with the last commit's message, we can choose to use this message or write another one.
- Save and exit the editor.

Amending a commit changes the commit hash of our most recent commit. Even if we chose to use the same commit message, the commit hash will still change.



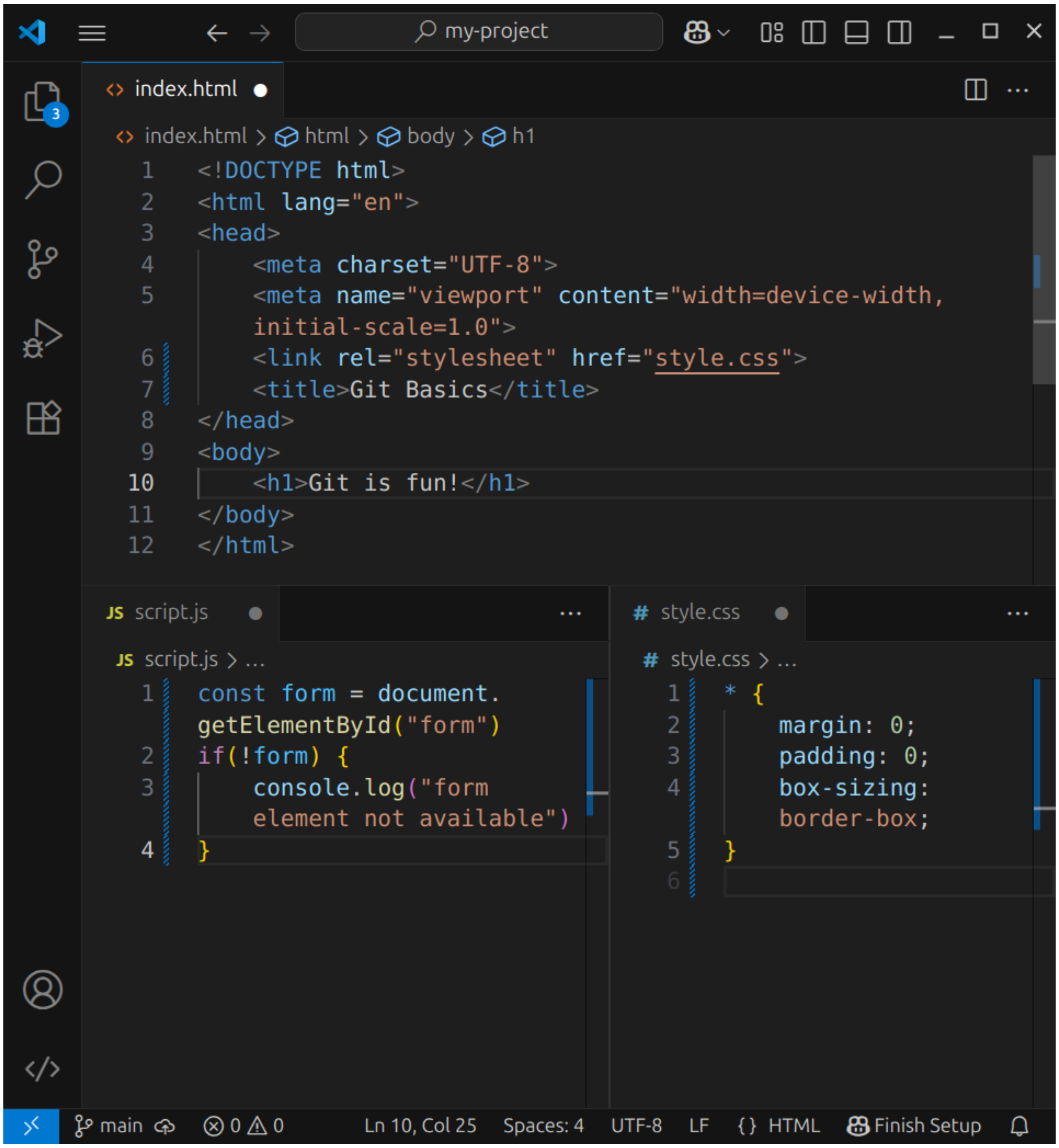
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Git Basics</title>
</head>
<body>
    <h1>Git is fun!</h1>
</body>
</html>
```

```
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git add .
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git commit --amend
[main 07818bf] added a heading (amended)
 Date: Thu Jun 26 20:07:22 2025 +0100
 1 file changed, 11 insertions(+)
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git log --oneline
07818bf (HEAD -> main) added a heading (amended)
24ff55c initial commit
rolex@rolex-Latitude-7420:~/Desktop/my-project$
```

From the above screenshot, we edited the title tag, staged the modification with git add . (same as git add --all) and used git commit --amend. In the nano editor I simply wrote **"added a heading (amended)"** as the commit message. In the git log output, you will notice both the commit message and the commit hash of the last commit changed which means we have successfully replaced the last commit with the corrected version.

Unstaging a file

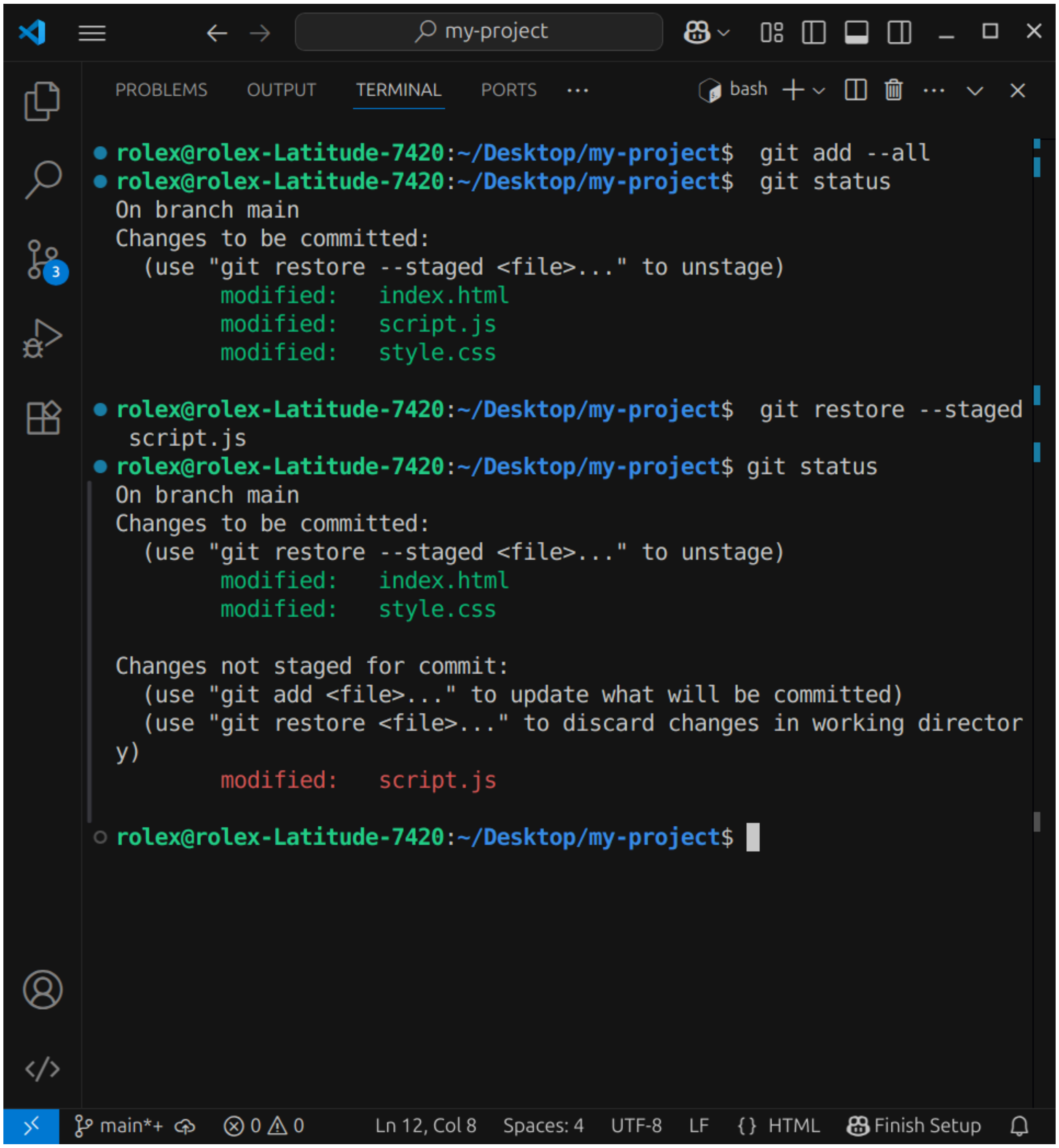
Suppose you mistakenly stage a file you don't want to commit probably because you used git add --all and git just included all modified and untracked files to the staging area, you can remove some of the unwanted staged files with the git restore --staged <file> command



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <link rel="stylesheet" href="style.css">
    <title>Git Basics</title>
</head>
<body>
    <h1>Git is fun!</h1>
</body>
</html>
```

```javascript
const form = document.
getElementById("form")
if(!form) {
    console.log("form
    element not available")
}
```

```css
* {
    margin: 0;
    padding: 0;
    box-sizing:
    border-box;
}
```



```
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git add --all
rolex@rolex-Latitude-7420:~/Desktop/my-project$  git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
        modified:   script.js
        modified:   style.css

rolex@rolex-Latitude-7420:~/Desktop/my-project$  git restore --staged
 script.js
rolex@rolex-Latitude-7420:~/Desktop/my-project$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
        modified:   style.css

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working director
y)
        modified:   script.js

rolex@rolex-Latitude-7420:~/Desktop/my-project$
```

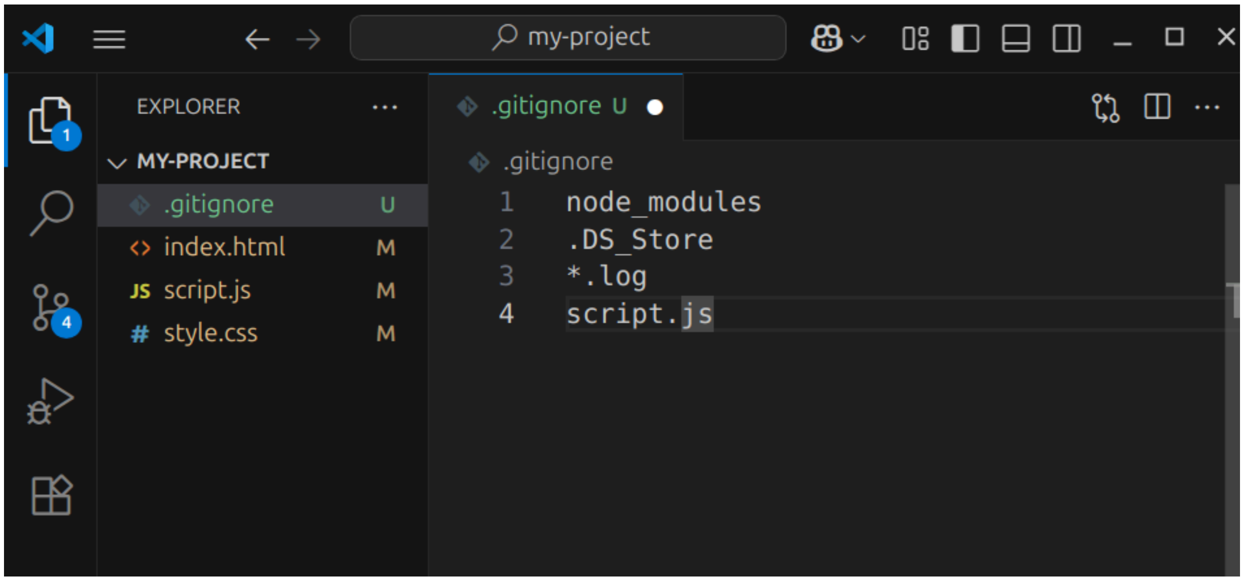
In the above screenshots, I have modified all three files, but I want to commit just the index.html and style.css file. I used git add --all instead of staging the files individually now all three files are in the staging area. To unstage only the script.js I ran git restore --staged script.js. We confirm that the file is no longer staged for committing with the git status command.

Ignoring certain files

Sometimes you don't want git to ever track or commit a particular file or folder in your repo because they are large files like node modules or log files that are not essential to your codebase.

To prevent git from tracking or staging certain files even when you run git add --all, create a .gitignore file in the root of your repo and write out the names of the folders/files you want git to always ignore

Although git will track the .gitignore file itself, it won't track anything written inside the file.



```
node_modules
.DS_Store
*.log
script.js
```

From the screenshot above, upon committing the .gitignore file, git will no longer track:

- node_modules folder
- DS_Store folder (In MacOS)
- All files ending with .log
- And script.js (So we don't accidentally stage it another time).

If you want git to resume tracking the script.js file, simply erase it from the .gitignore file.

Although you can use git to track and commit any kind of file no matter the size, it is best to only use it for tracking source code only. Other kinds of files like pictures and videos should be hosted on the cloud using platforms like cloudinary instead.

🔹 Conclusion

Git may seem daunting at first, but once you understand the basics, it becomes an indispensable tool in your development workflow. You've seen how to install Git, configure it to reflect your identity and preferences and set up a project to start tracking changes locally.

By now, you've learned how Git categorizes your files, how to stage, commit, and even amend changes when needed. You've also seen how to avoid unnecessary clutter in your repository using `.gitignore` a simple but powerful tool for keeping things tidy.

But this is just the beginning. Git's real strength shines when you start working with branches and remotes, exploring new features or fixing bugs without affecting your main codebase. Understanding how to manage branches effectively is key to unlocking collaboration and advanced workflows.

Up next— we'll dive into branching, why it matters, how to use it, and how it supercharges git.

🔹 Connect with Me

If you found this helpful and want to learn more about JavaScript, web development, and other tech topics, feel free to connect with me on:

- **X (formerly Twitter)**: https://x.com/rolex_devv
- **GitHub**: https://github.com/rowleks
- **LinkedIn**: https://www.linkedin.com/in/rowland-momoh-b32a7a22a/
- **Personal Portfolio**: https://rowland-momoh.netlify.app/

Let’s keep learning and building together!