EXPERIMENT 2

Source code Version Controlling, Central vs Distributed Version Controlling, Git Workflow, Installing Git, and Important Git Commands

Prepared by

NEEL VAGHASIYA

Semester - 7

BTech in Information Technology

Batch year: 2021-2025

neelvaghasiya003@gmail.com

TABLE OF CONTENTS

01 INTRODUCTION TO VERSION CONTROL

01 INTRODUCTION TO VERSION CONTROL

What is Version Control?

- Also known as source control.
- It is the process of tracking and managing changes to file(s) or source code over time.
- Version Control is like a “Checkpoint” feature in video games, where we can save our progress at different points and return to any checkpoint if something goes wrong.

What is Version Control System?

- A Version Control System(VCS) is a software that helps tracking and managing changes to source code.
- It keeps track of changes made and takes a snapshot on each modification.

01 INTRODUCTION TO VERSION CONTROL

Why to use VCSs?

- To keep track of all the modifications made to the source code.
- Simplifies backup and recovery processes.
- Provides the functionalities for Merging & Branching.
- Helps to boost productivity.

Examples of some popular VCSs


Git

Subversion


Mercurial

CVS

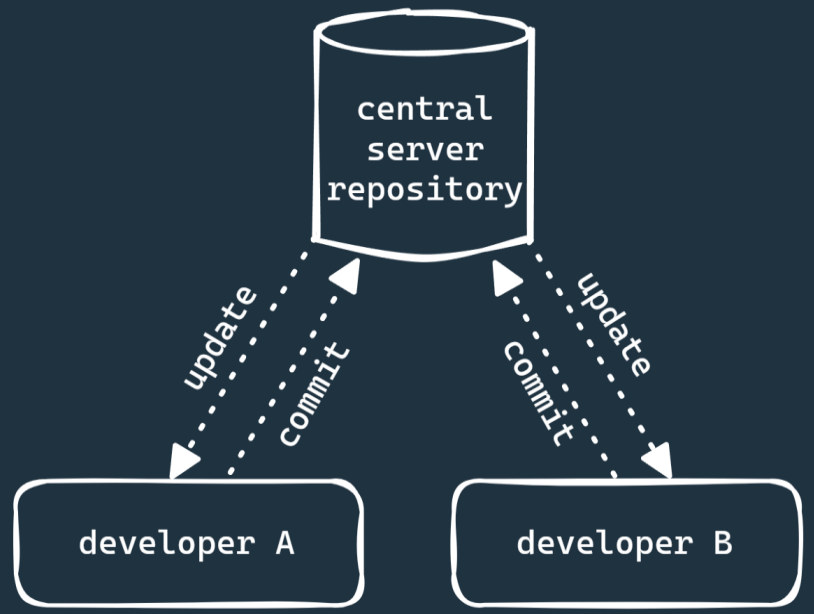

Helix Core

02 TYPES OF VERSION CONTROL SYSTEMS (VCSs)

02 TYPES OF VERSION CONTROL SYSTEMS

Centralized Version Control Systems (CVCS)

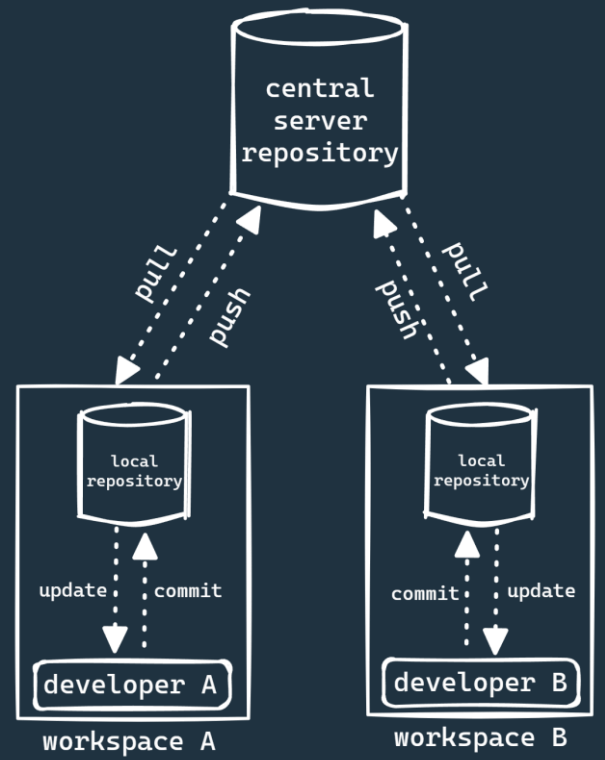
- Traditional VCS where the source code is stored in a central server. Developers check out the code they need to work on and make changes directly to that codebase.
- If server goes down, no one can access the remote repository.
- Slow response time, as every operations are performed on remote server.
- Limited branching and merging.
- Chances of Multi-developer conflicts (due to Race condition)
- Examples: Subversion(SVN), Helix core



02 TYPES OF VERSION CONTROL SYSTEMS

Distributed Version Control Systems (DVCS)

- In DVCS, each developer has a full copy of the remote repository on their local machine, with the entire history.
- If server goes down, developers can continue working on their local codebase.
- Fast response time, as majority operations are performed locally.
- Advanced branching and merging, allows efficient parallel development.
- Reduced chances of Multi-developer conflicts
- Examples: Git, Mercurial



03 INTRODUCTION TO GIT

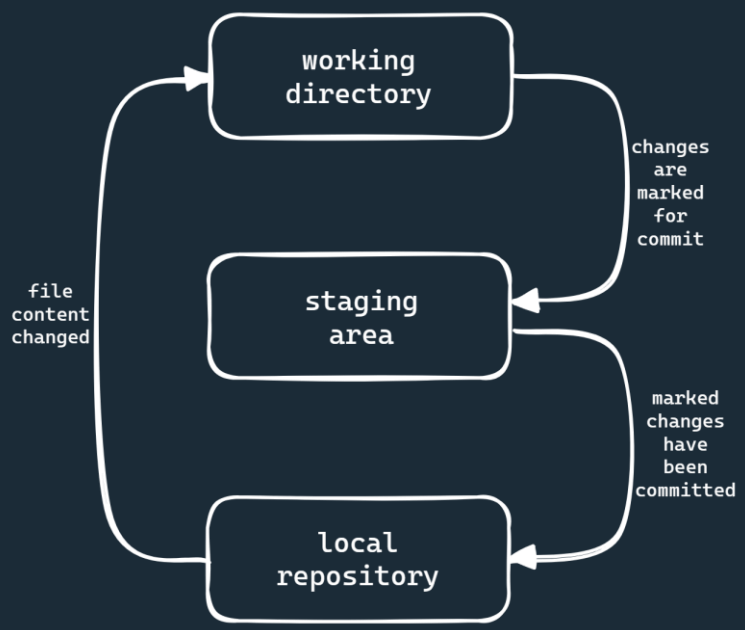
03 INTRODUCTION TO GIT

What is Git?

- Git is a free and open source, Distributed Version Control System (DVCS) designed for speed and efficiency.
- Git is very fast, because it works locally.
- It facilitates branching and merging functionality.
    - Branch: A separate version of the project. It allows us to work on different features of a project without impacting main codebase.
- Some other terminologies: Repository (Repo), Commit, Clone, Merge, Pull and Push (while working with remote repo)
- Important: Git and GitHub are **not same!!**

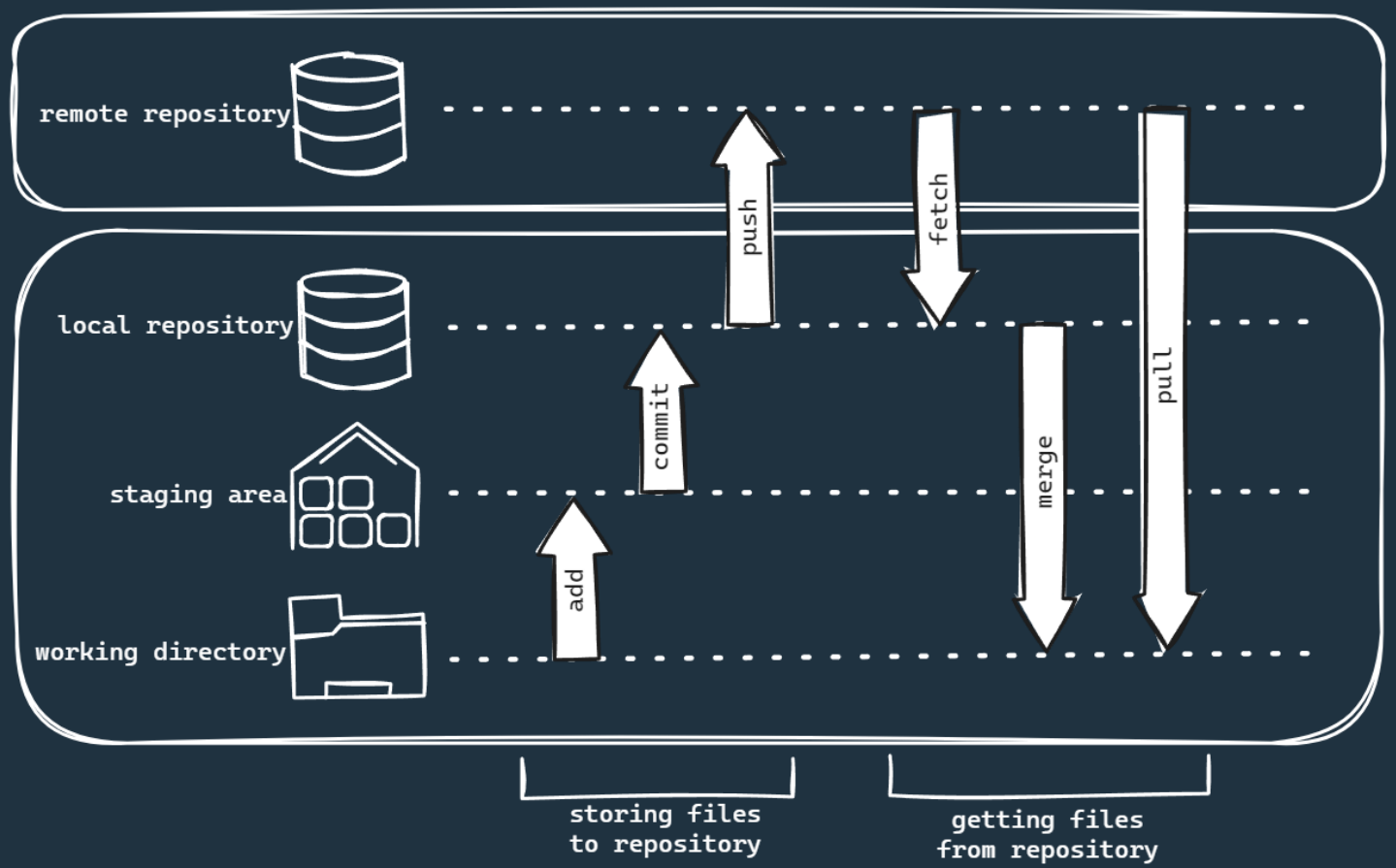
03 INTRODUCTION TO GIT

- Git will keep track of everything happening inside the folder.
- Git has three main states that files can reside in:
    1. Modified: Changes have been made but not yet staged.
    2. Staged: Ready to be committed.
    3. Committed: Changes have been stored in the local repository.
- Three main areas in Git:
    1. Working directory: Local file system, where we modify files.
    2. Staging area (index): Area where changes are reviewed before committing.
    3. Local repository: Area where the committed changes are stored, with history of previous commits.



04 GIT WORKFLOW AND BRANCHING

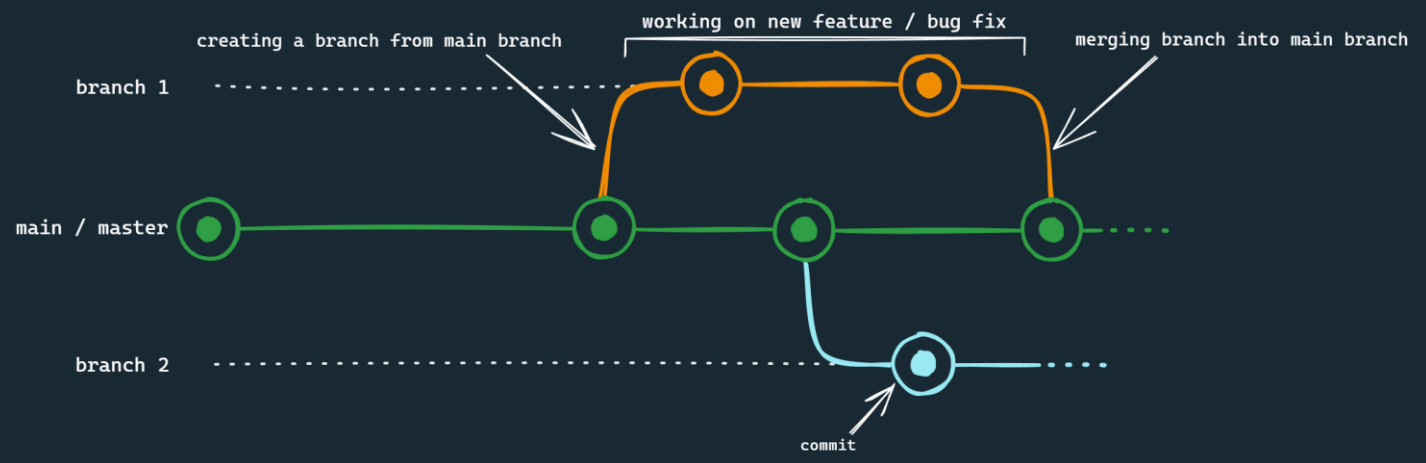
04.1 GIT WORKFLOW



04.2 BRANCHING

What is Branch?

- Branches are the isolated environment where developers can work on specific tasks (like adding new features, fixing a bug, or experiments) **without affecting main codebase**.
- Default branch: “**main**” or “**master**”



04.2 BRANCHING

- Main branch should contain fully tested and stable code.
- Do not directly modify the main branch. Instead:
  - Create a new branch
  - Work in isolation on the new branch (without affecting main branch)
  - Test and review changes within new branch.
  - Merge the branch back into the main branch.

Why Branching is essential?

- Parallel development: Allows multiple developers work on different tasks simultaneously without interfering with each other's code.
- Reduces risk: Main code remains stable while changes are reviewed in separate branches.
- Experimentation: Provides isolated space

04.2 BRANCHING

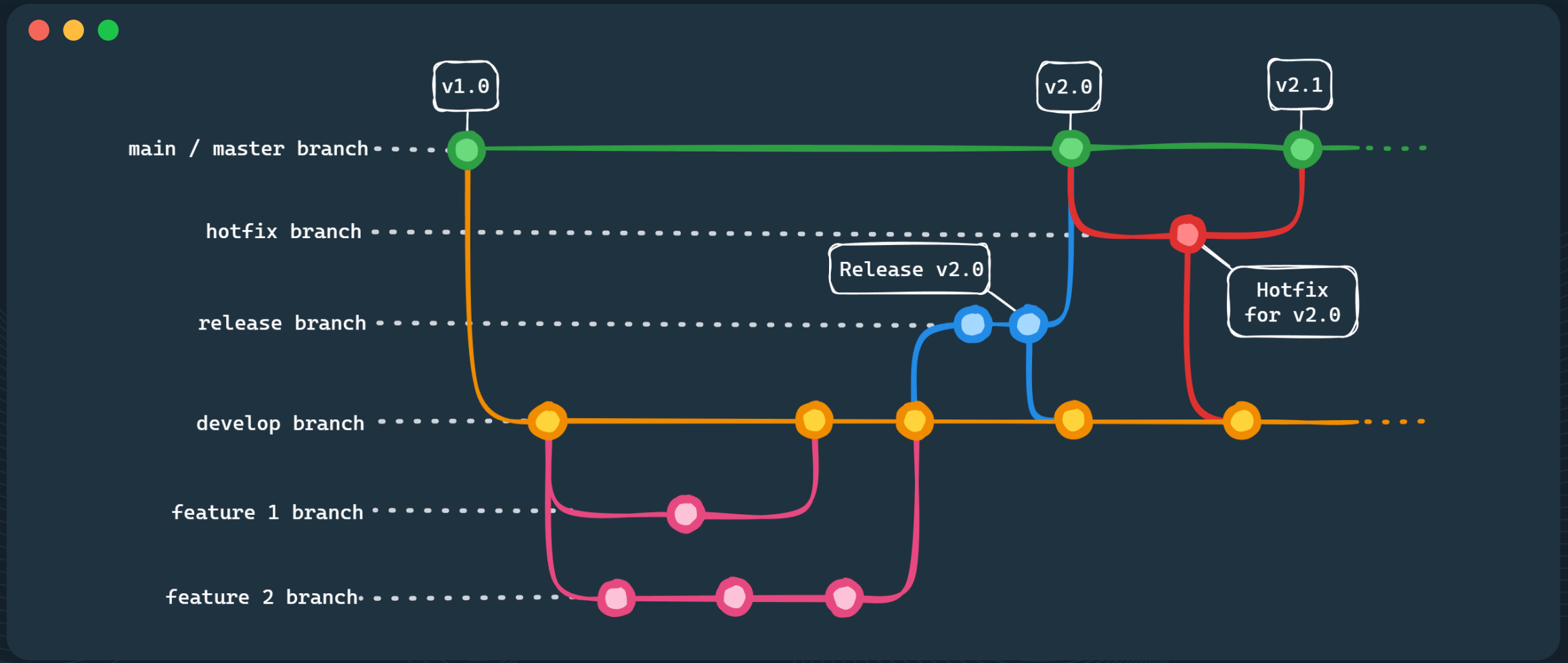
Branching Strategies:

- Gitflow: Structured branching model, involves long-lived and short-lived branches
- GitHub flow: Simplified branching model, involves short-lived branches

04.2.1 Gitflow

- Key branches in Gitflow:
  - Master: Long-lived branch that always contains the production-ready code
  - Develop: Long-lived branch that contains latest development changes
  - Feature: Short-lived branches, created from develop and merged back into develop once the feature is complete.
  - Release: Short-lived branches, created from develop to prepare a new production release and merged into both master and develop.
  - Hotfix: Short-lived branches, created from master to quickly fix issue in production environment, and merged into both master and develop.

04.2 BRANCHING

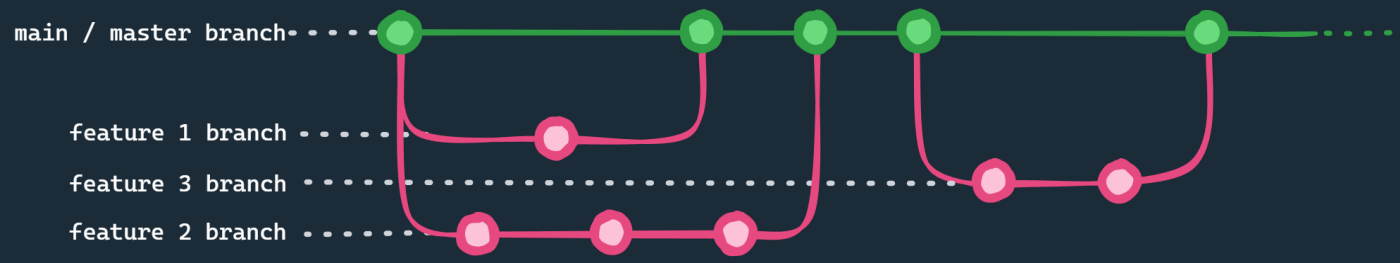


04.2 BRANCHING

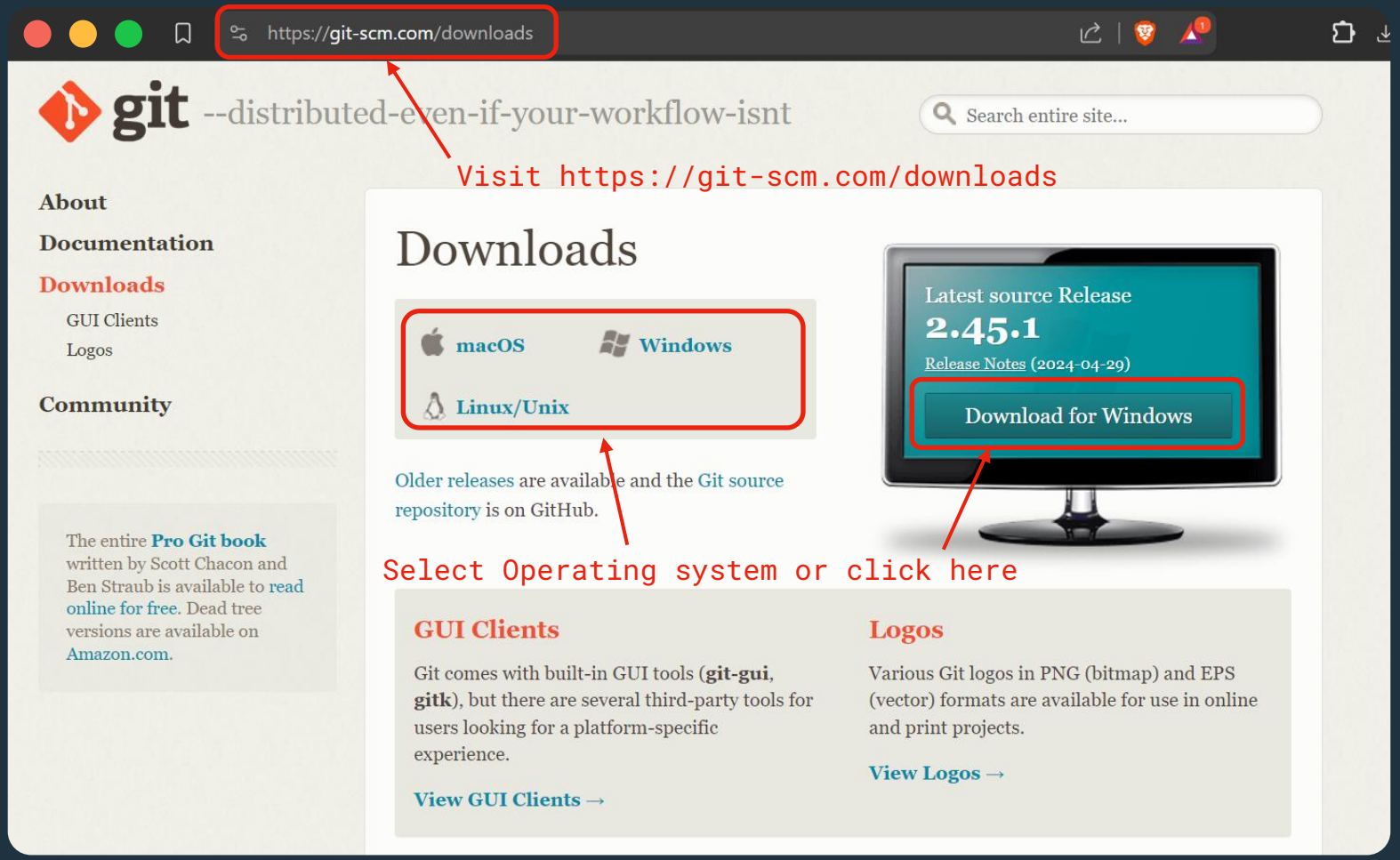
04.2.2 GitHub flow

- Key branches in GitHub flow:
  - Master: Long-lived branch that always contains the production-ready code
  - Feature: Short-lived branches, created from master and merged back into master once the feature is complete.

- **Workflow**
  - Create Feature branch from master
  - Work and commit
  - Open pull request
  - Review and merge back into master



05 INSTALLATION OF GIT.


https://git-scm.com/downloads
git --distributed-even-if-your-workflow-isnt
Search entire site...
Visit https://git-scm.com/downloads
About
Documentation
Downloads
GUI Clients
Logos
Community
The entire Pro Git book written by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.
Downloads
macOS
Windows
Linux/Unix
Older releases are available and the Git source repository is on GitHub.
Latest source Release
2.45.1
Release Notes (2024-04-29)
Download for Windows
Select Operating system or click here
GUI Clients
Git comes with built-in GUI tools (git-gui, gitk), but there are several third-party tools for users looking for a platform-specific experience.
View GUI Clients →
Logos
Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.
View Logos →

Download for Windows

Click here to download the latest (**2.45.1**) **64-bit** version of **Git for Windows**. This is the most recent maintained build. It was released **12 days ago**, on 2024-05-14.

Other Git for Windows downloads

Standalone Installer

32-bit Git for Windows Setup.

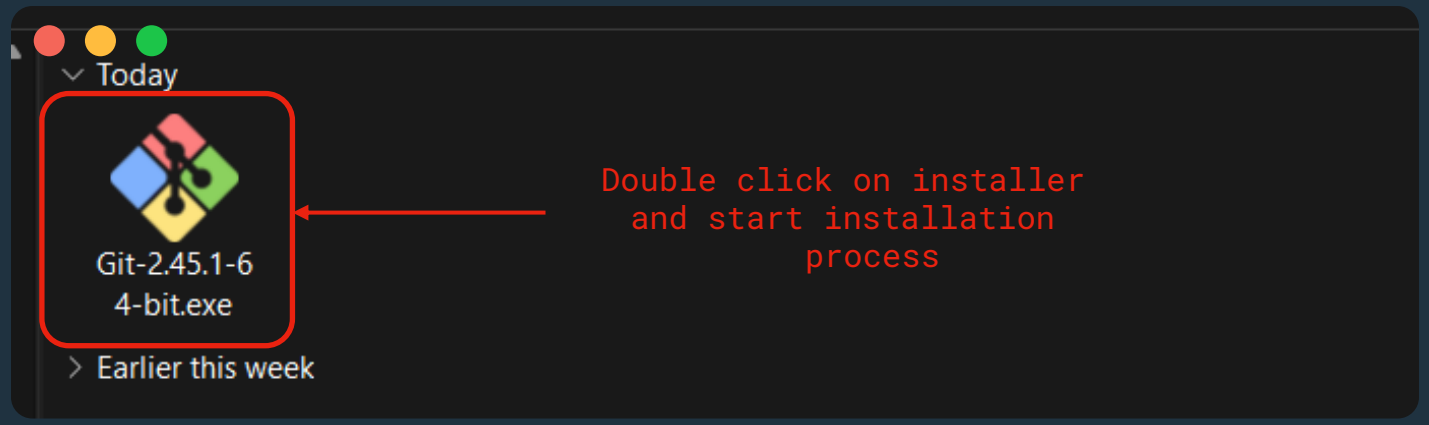
64-bit Git for Windows Setup.

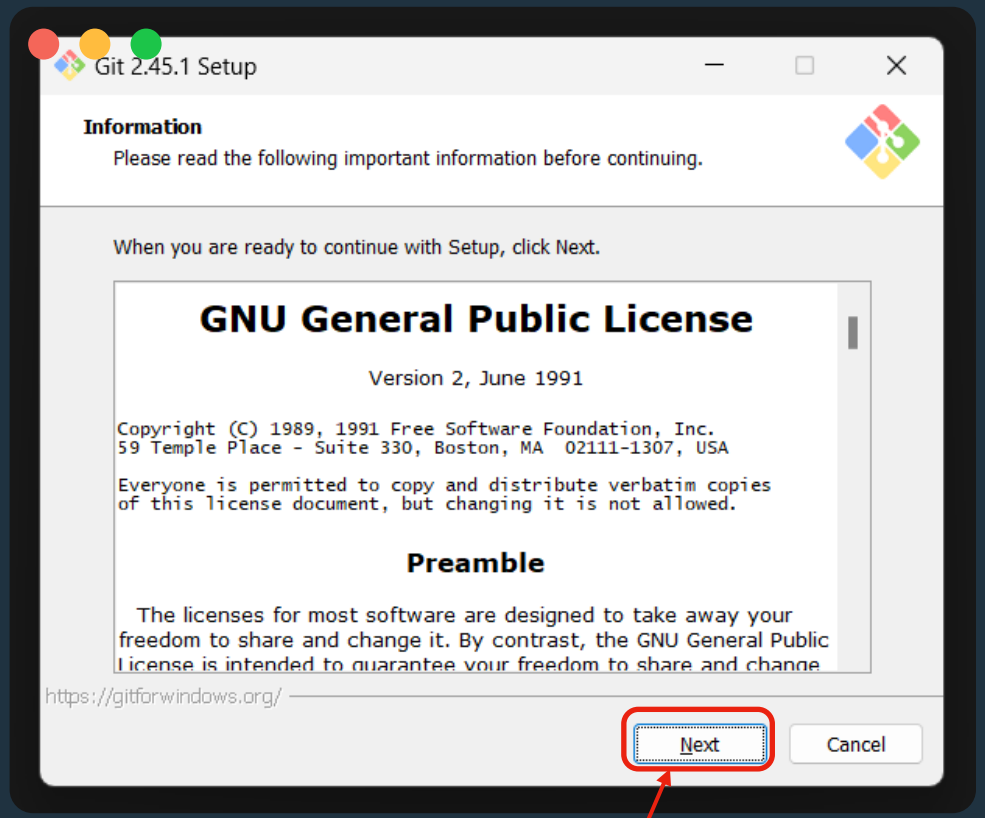
Select system type and download installer file

Portable ("thumbdrive edition")

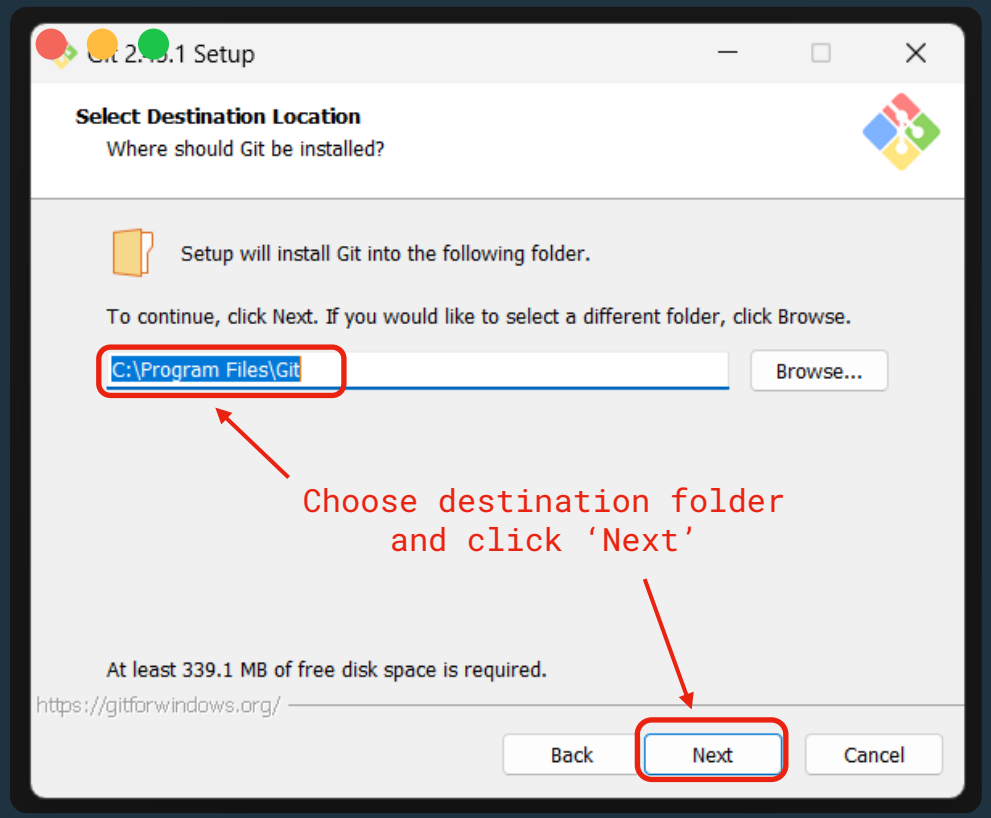
32-bit Git for Windows Portable.

64-bit Git for Windows Portable.

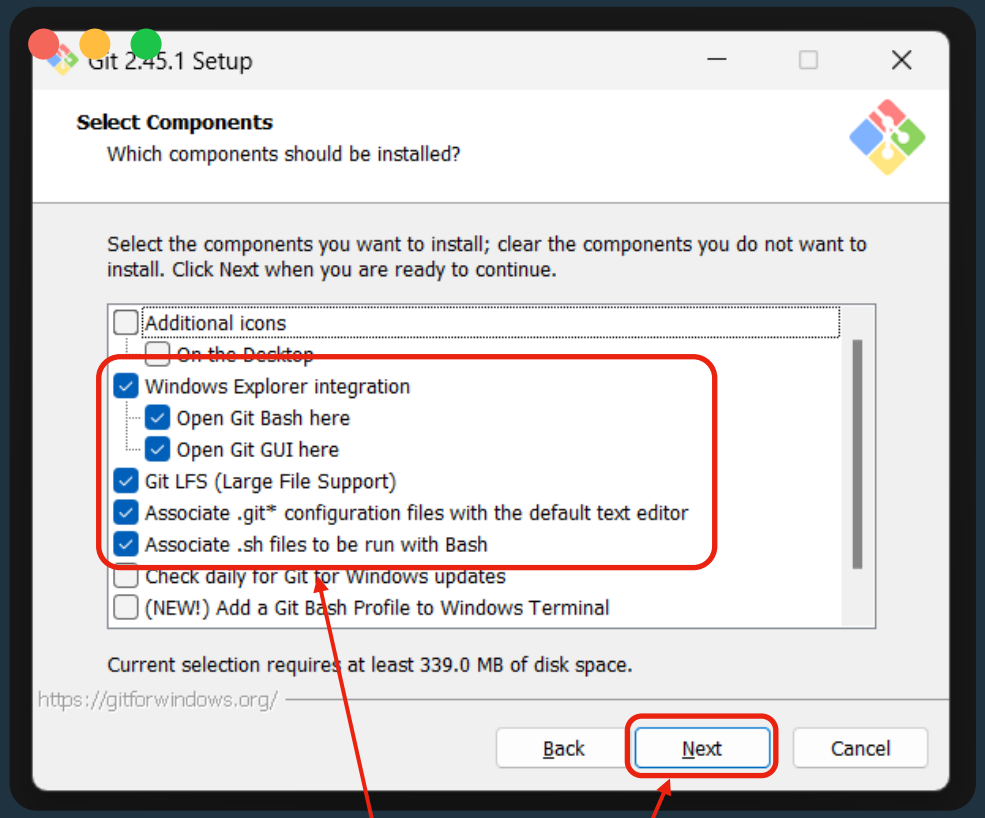




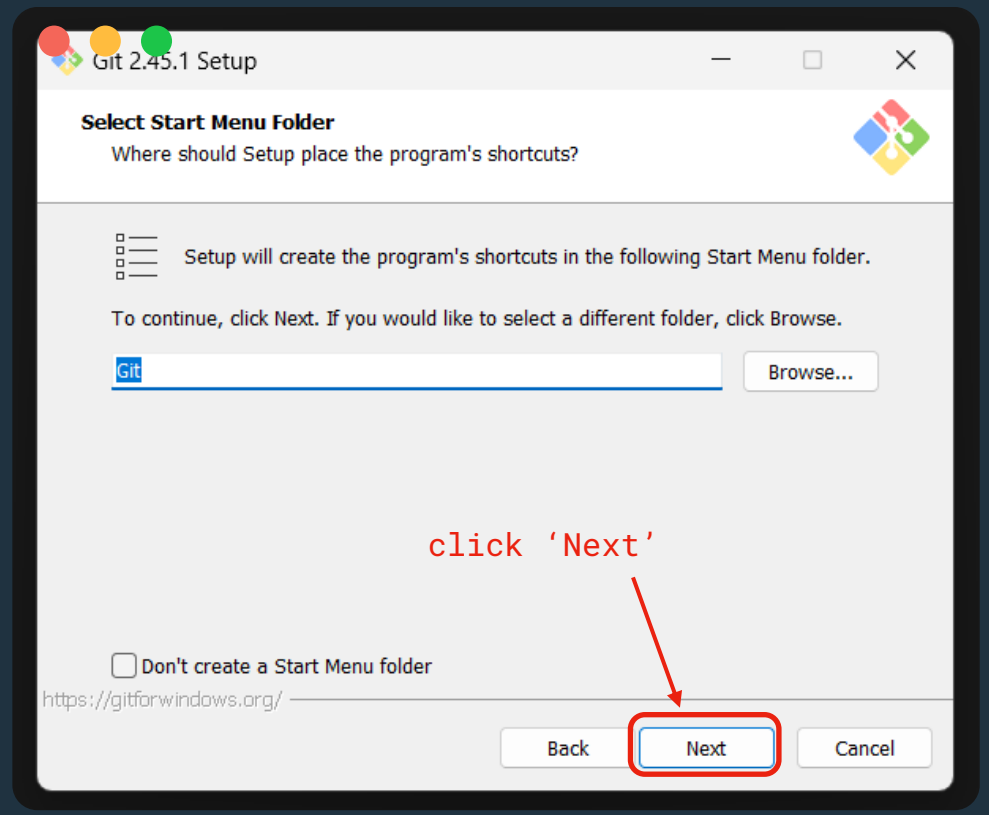
Read terms & conditions
and click 'Next'



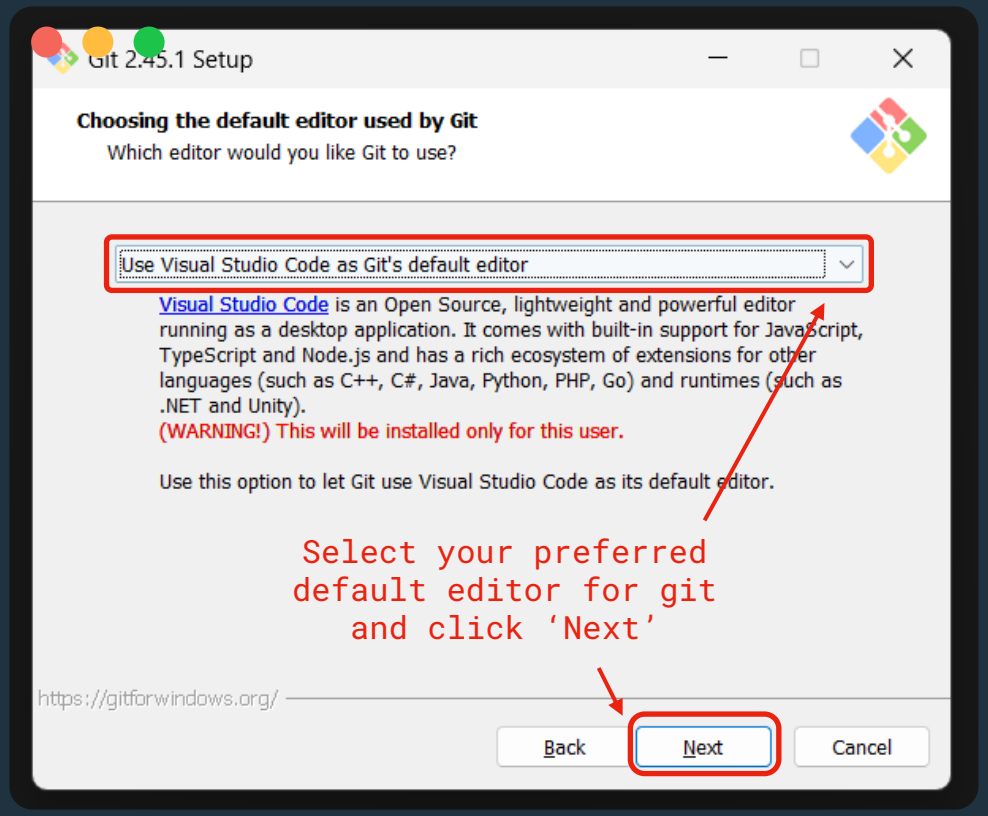
Choose destination folder
and click 'Next'

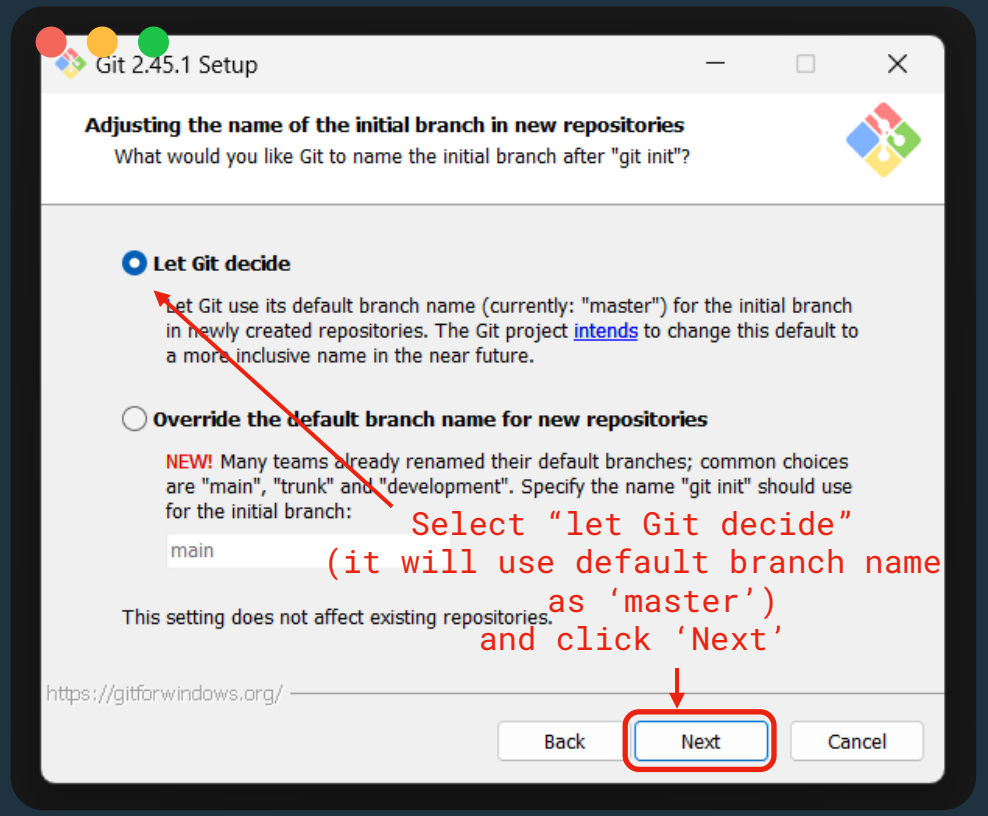


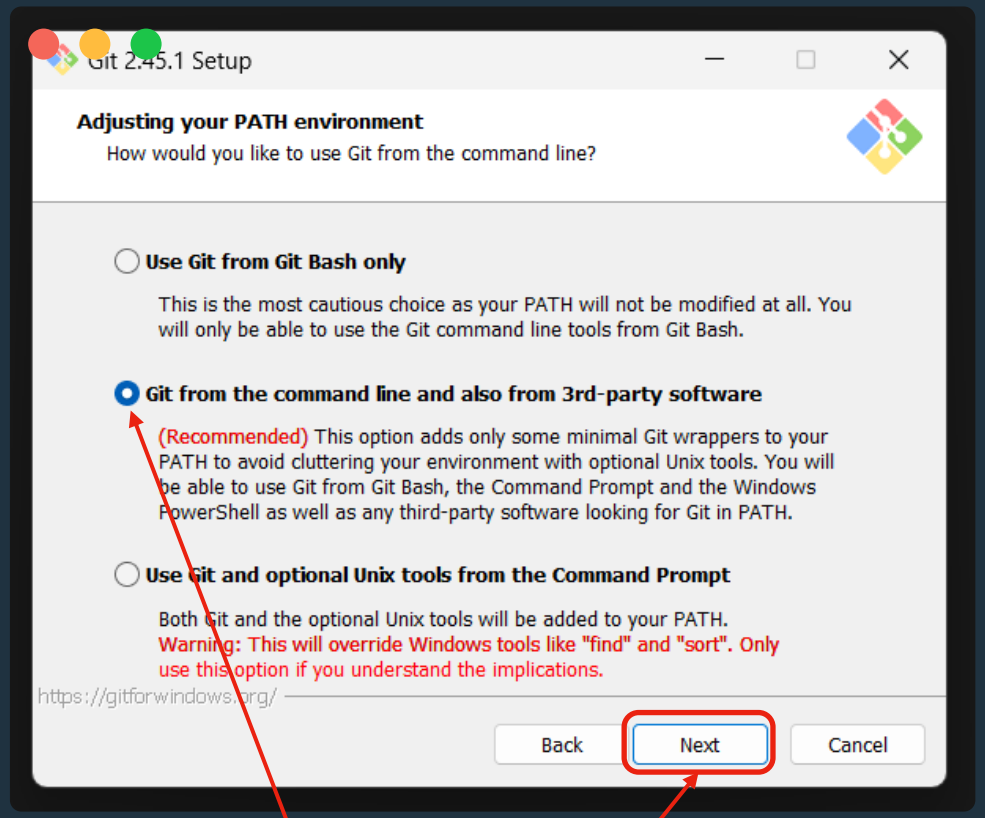
Keep these options selected and click 'Next'

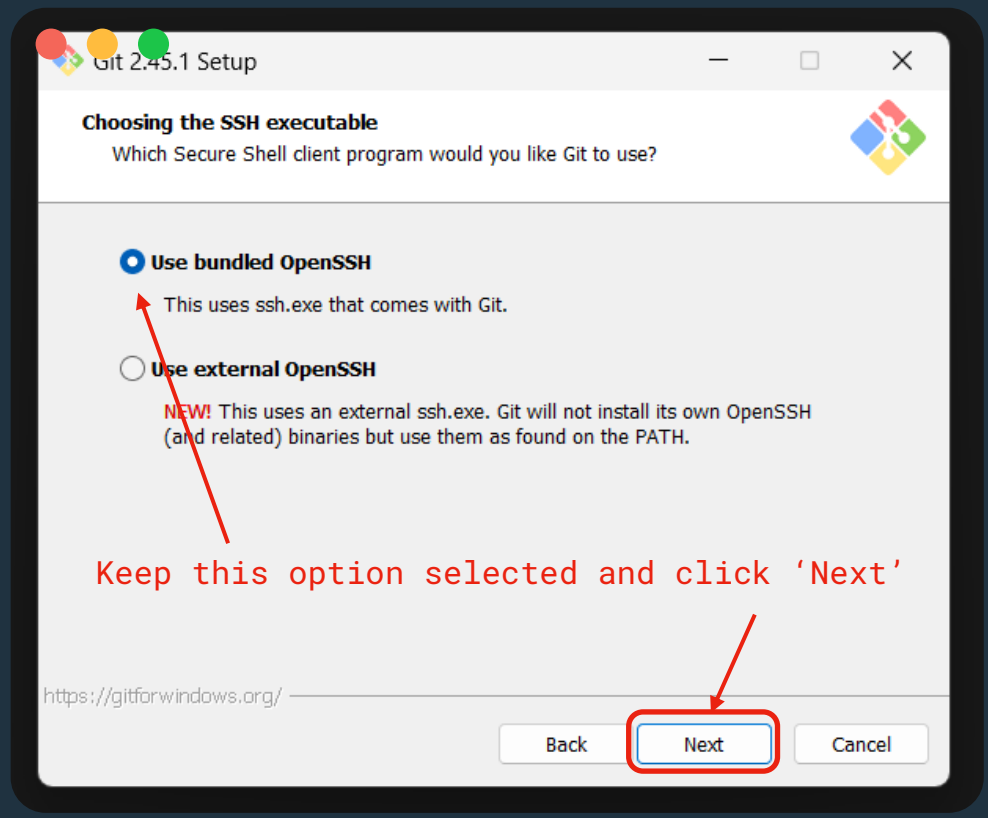


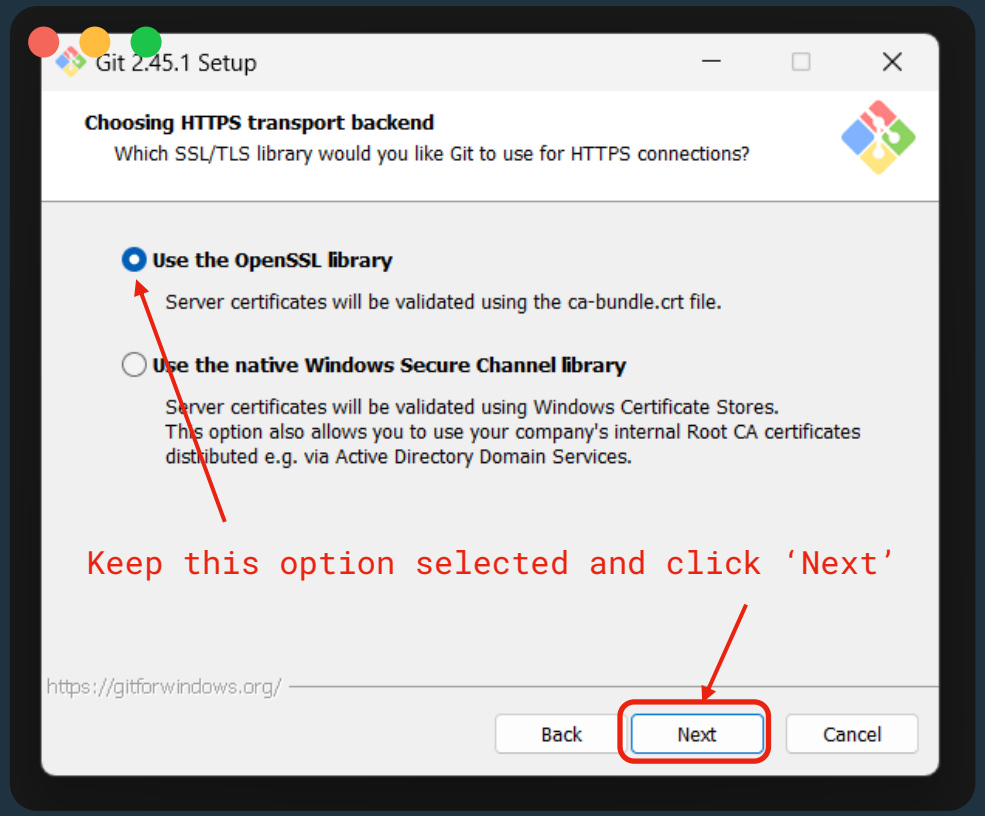
click 'Next'

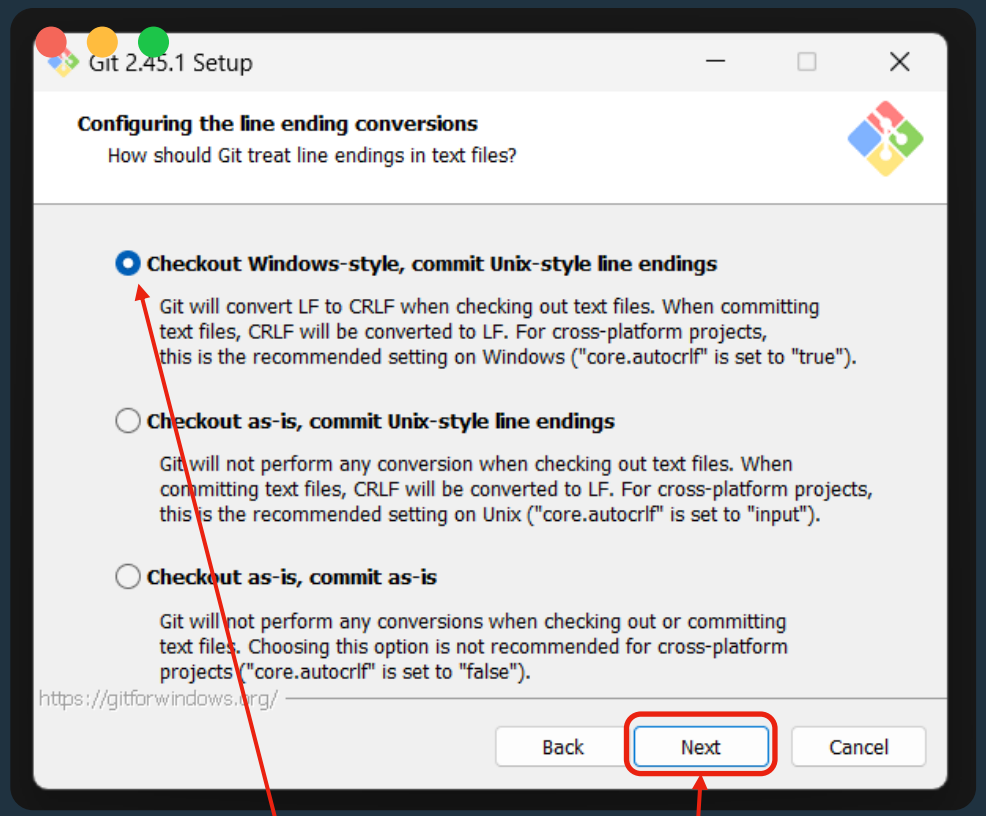

Git 2.45.1 Setup
Choosing the default editor used by Git
Which editor would you like Git to use?
Use Visual Studio Code as Git's default editor
Visual Studio Code is an Open Source, lightweight and powerful editor running as a desktop application. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity).
(WARNING!) This will be installed only for this user.
Use this option to let Git use Visual Studio Code as its default editor.
Select your preferred default editor for git and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel

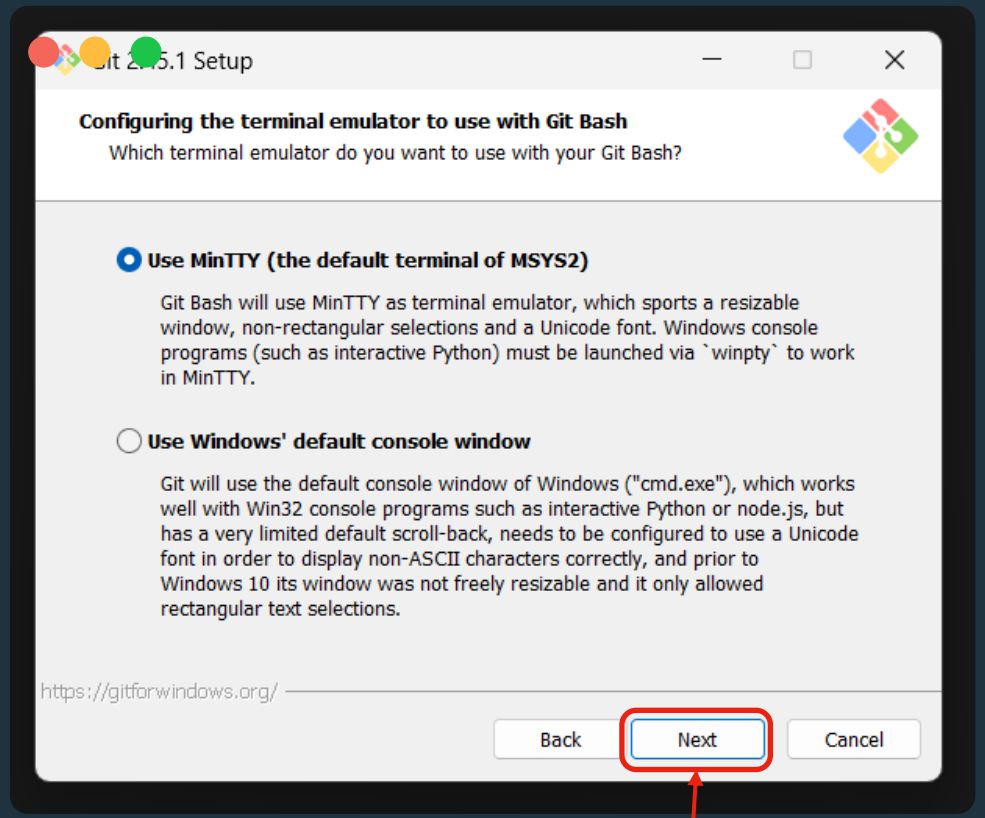

Git 2.45.1 Setup
Adjusting the name of the initial branch in new repositories
What would you like Git to name the initial branch after "git init"?
Let Git decide
Let Git use its default branch name (currently: "master") for the initial branch in newly created repositories. The Git project intends to change this default to a more inclusive name in the near future.
Override the default branch name for new repositories
NEW! Many teams already renamed their default branches; common choices are "main", "trunk" and "development". Specify the name "git init" should use for the initial branch:
main
This setting does not affect existing repositories.
Select "let Git decide" (it will use default branch name as 'master') and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel


Git 2.45.1 Setup
Adjusting your PATH environment
How would you like to use Git from the command line?
Use Git from Git Bash only
This is the most cautious choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.
Git from the command line and also from 3rd-party software
(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.
Use Git and optional Unix tools from the Command Prompt
Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.
https://gitforwindows.org/
Back
Next
Cancel
Keep this option selected and click 'Next'

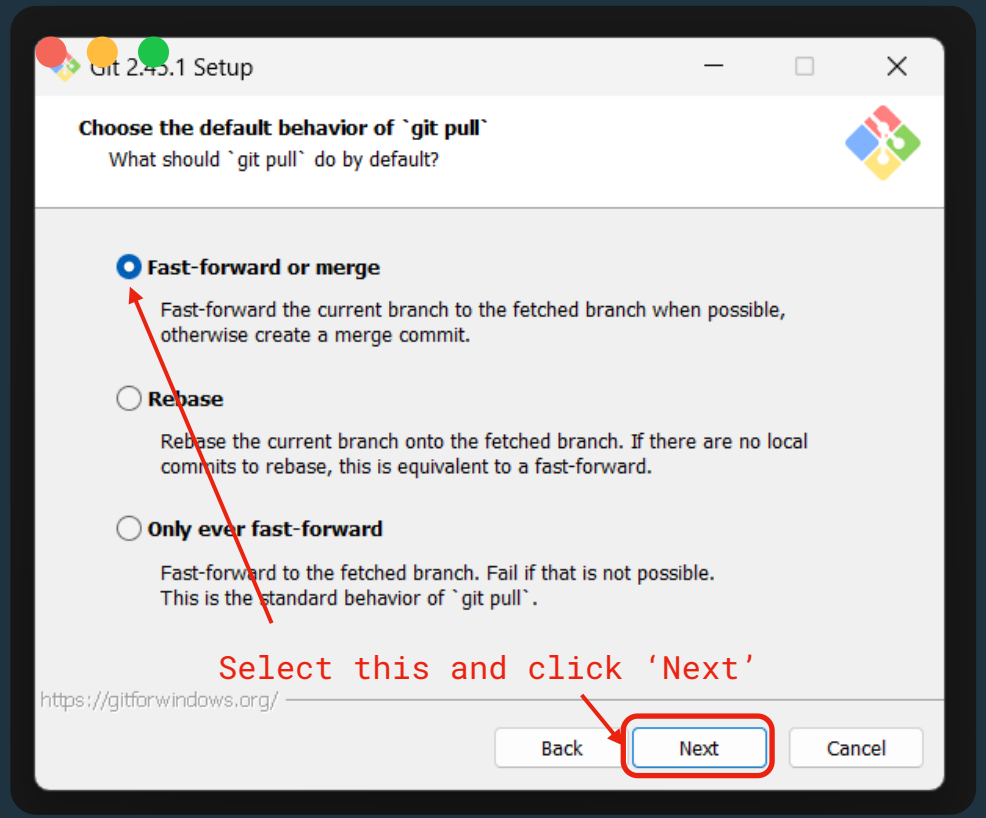

Git 2.45.1 Setup
Choosing the SSH executable
Which Secure Shell client program would you like Git to use?
Use bundled OpenSSH
This uses ssh.exe that comes with Git.
Use external OpenSSH
NEW! This uses an external ssh.exe. Git will not install its own OpenSSH (and related) binaries but use them as found on the PATH.
Keep this option selected and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel


Git 2.45.1 Setup
Choosing HTTPS transport backend
Which SSL/TLS library would you like Git to use for HTTPS connections?
Use the OpenSSL library
Server certificates will be validated using the ca-bundle.crt file.
Use the native Windows Secure Channel library
Server certificates will be validated using Windows Certificate Stores.
This option also allows you to use your company's internal Root CA certificates
distributed e.g. via Active Directory Domain Services.
Keep this option selected and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel

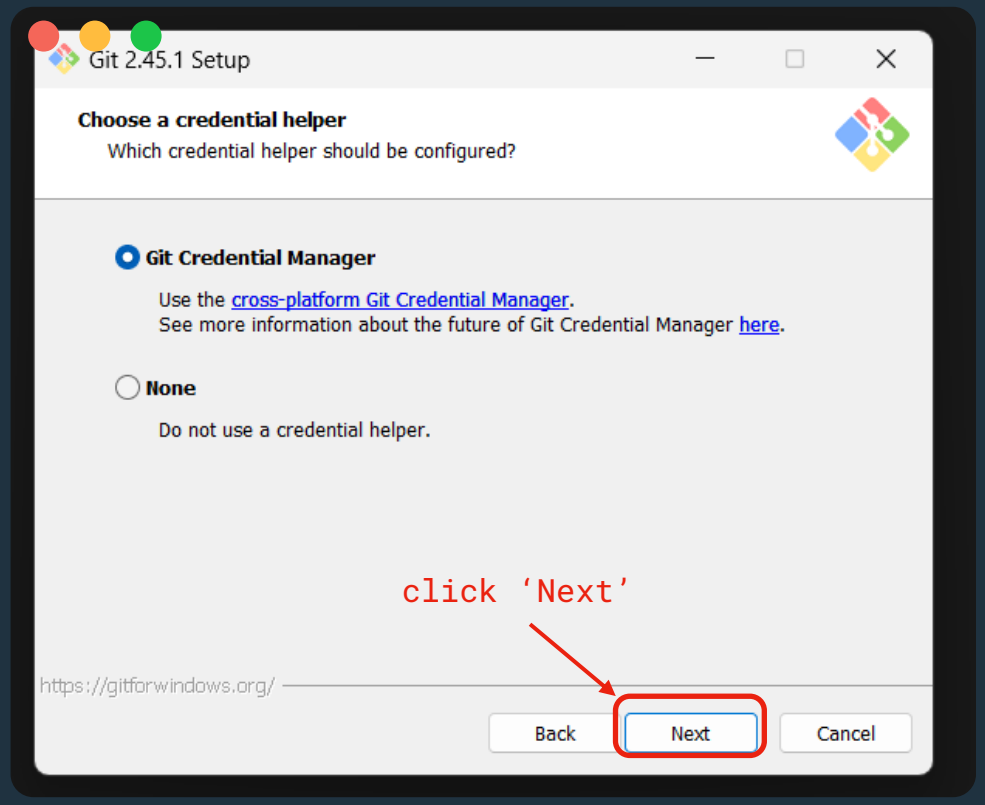

Git 2.45.1 Setup
Configuring the line ending conversions
How should Git treat line endings in text files?
Checkout Windows-style, commit Unix-style line endings
Git will convert LF to CRLF when checking out text files. When committing
text files, CRLF will be converted to LF. For cross-platform projects,
this is the recommended setting on Windows ("core.autocrlf" is set to "true").
Checkout as-is, commit Unix-style line endings
Git will not perform any conversion when checking out text files. When
committing text files, CRLF will be converted to LF. For cross-platform projects,
this is the recommended setting on Unix ("core.autocrlf" is set to "input").
Checkout as-is, commit as-is
Git will not perform any conversions when checking out or committing
text files. Choosing this option is not recommended for cross-platform
projects ("core.autocrlf" is set to "false").
Select this and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel

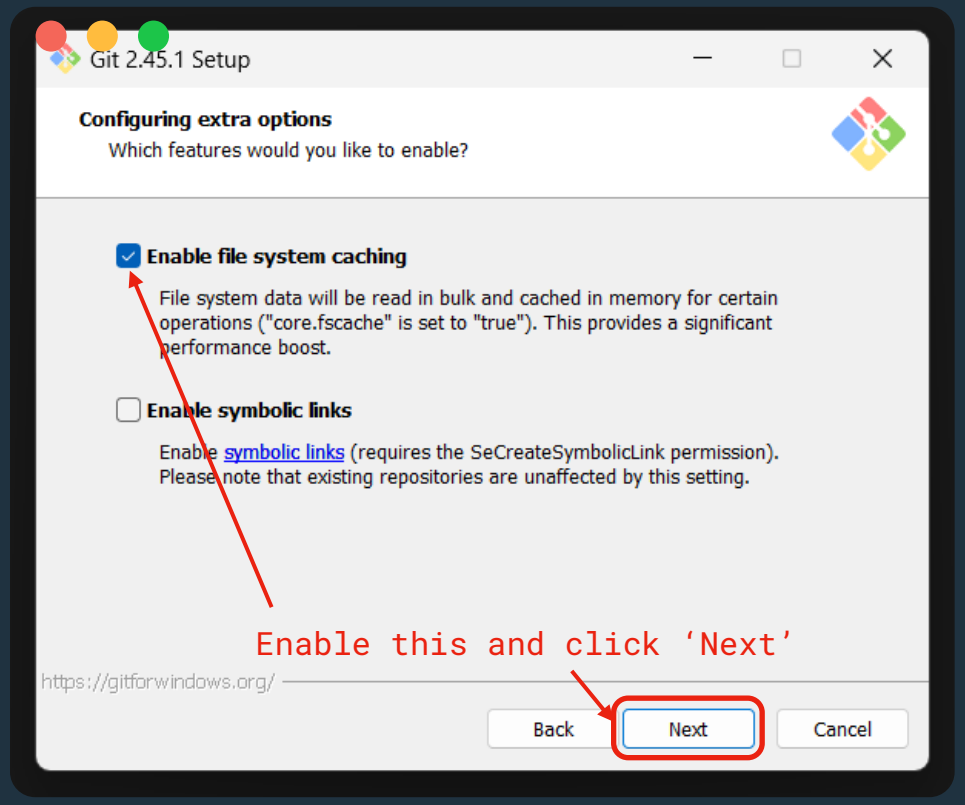


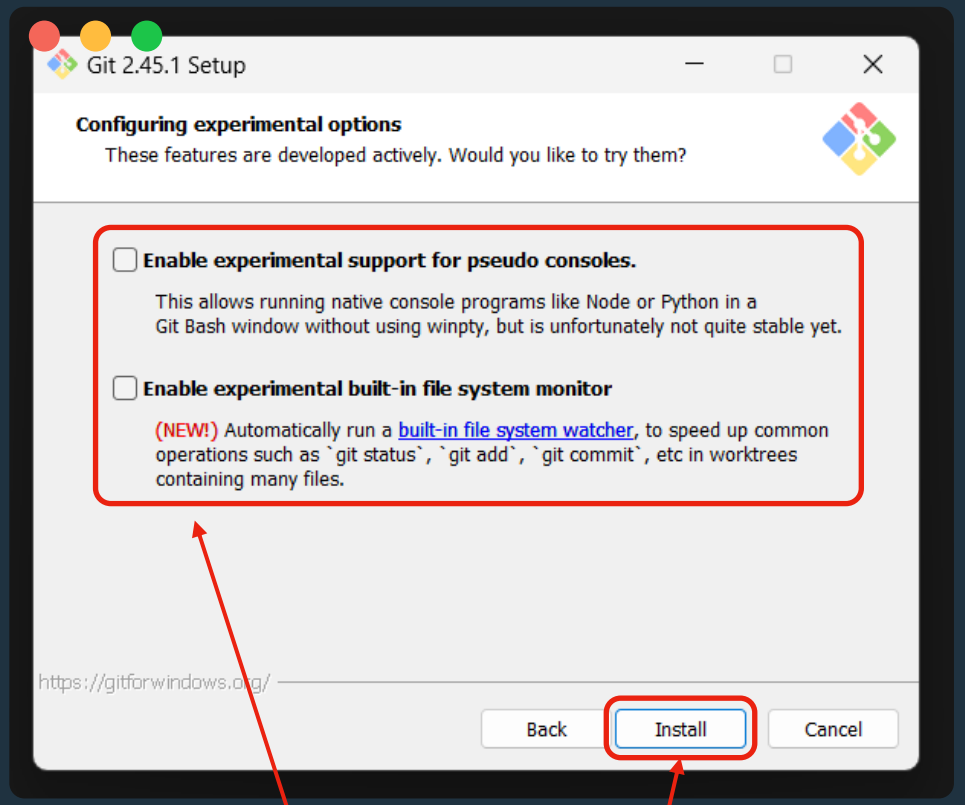
Choose terminal of your choice and click ‘Next’



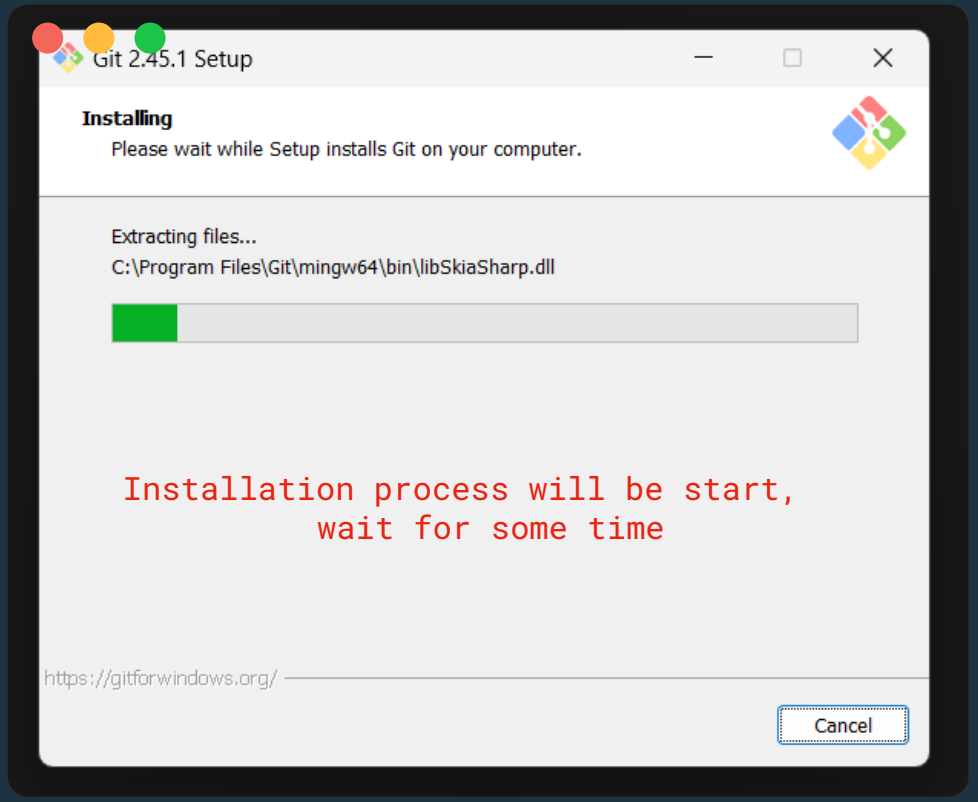
Select this and click ‘Next’


Git 2.45.1 Setup
Choose a credential helper
Which credential helper should be configured?
Git Credential Manager
Use the cross-platform Git Credential Manager.
See more information about the future of Git Credential Manager here.
None
Do not use a credential helper.
click 'Next'
https://gitforwindows.org/
Back
Next
Cancel

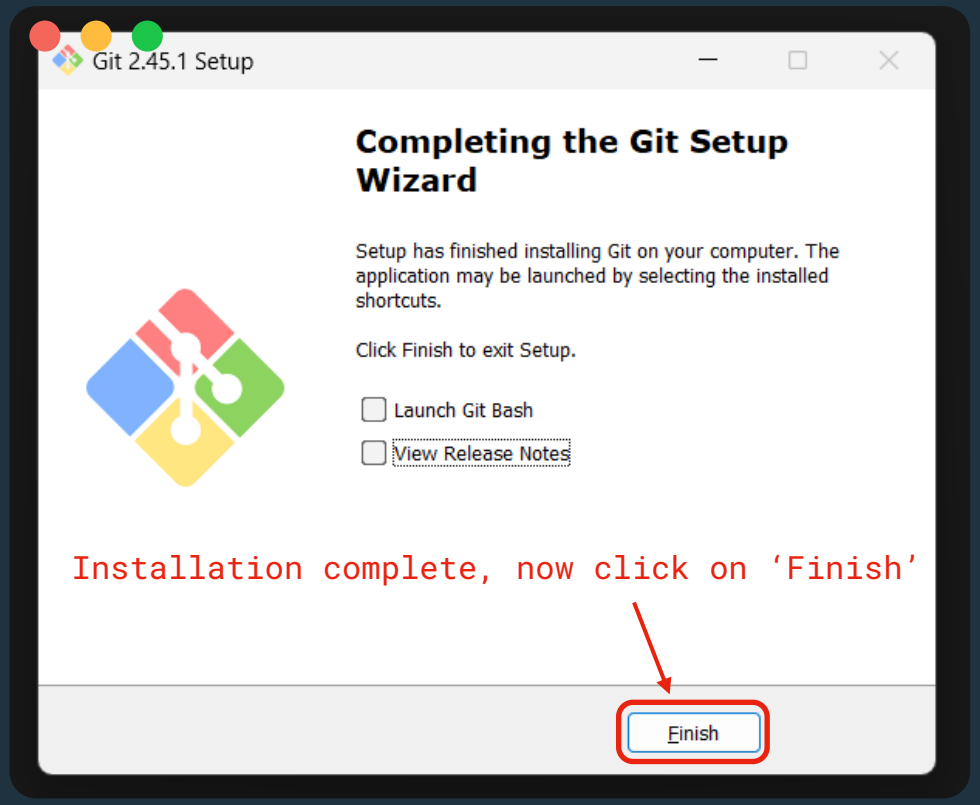

Git 2.45.1 Setup
Configuring extra options
Which features would you like to enable?
Enable file system caching
File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.
Enable symbolic links
Enable symbolic links (requires the SeCreateSymbolicLink permission).
Please note that existing repositories are unaffected by this setting.
Enable this and click 'Next'
https://gitforwindows.org/
Back
Next
Cancel

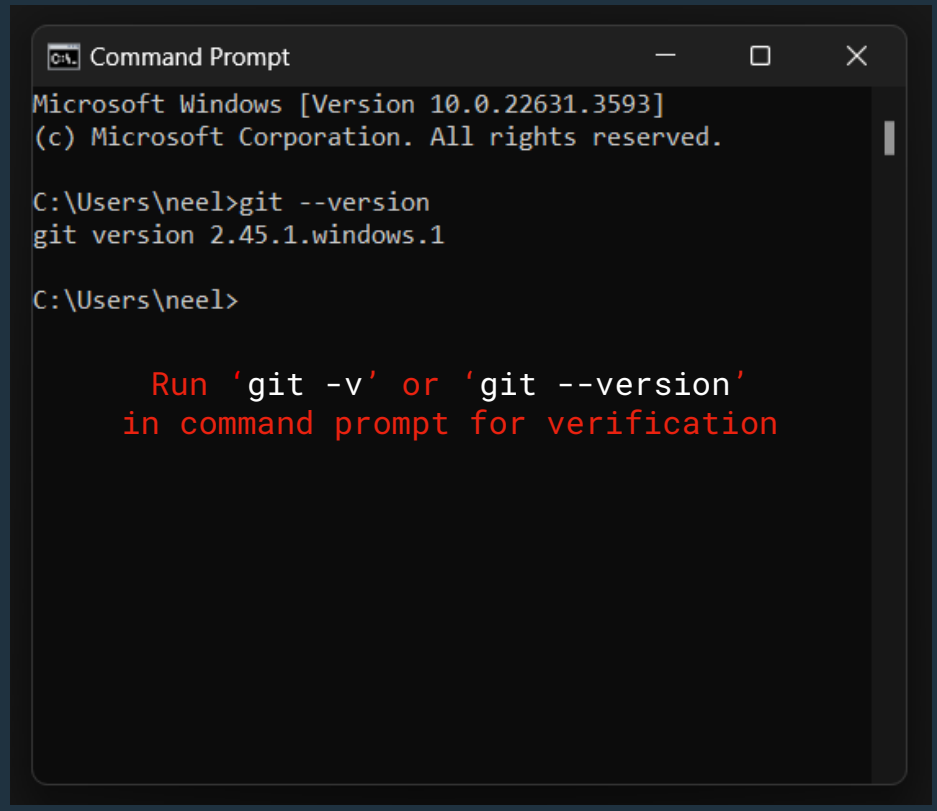


Enable optional feature
If you want and then click 'Install'



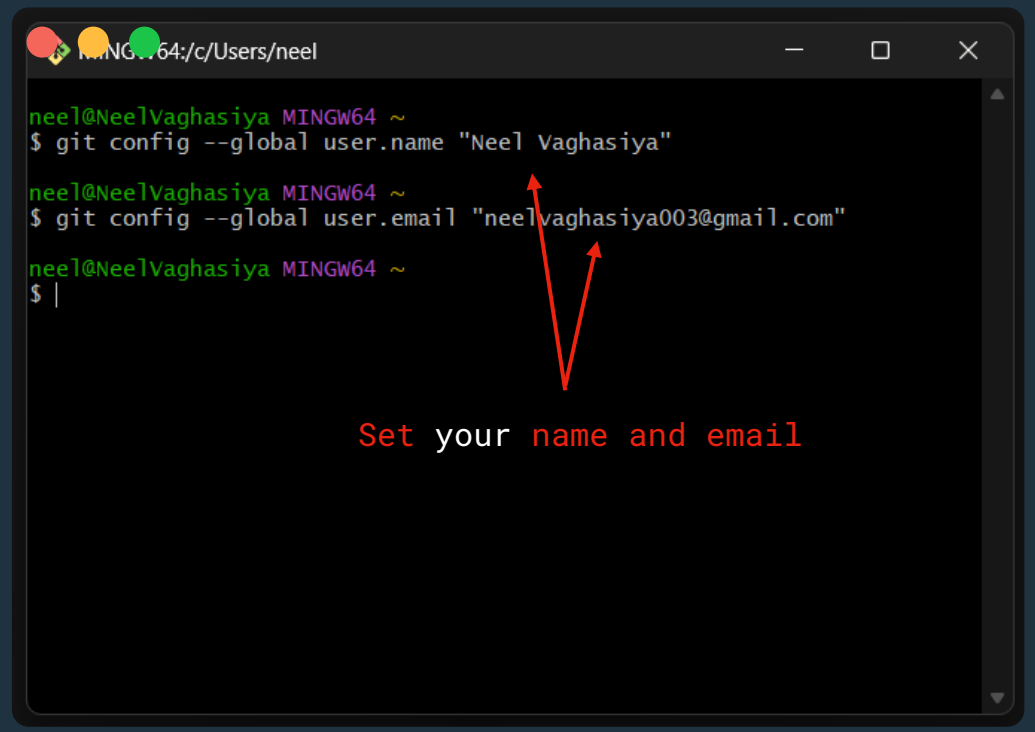
Installation process will be start,
wait for some time


Git 2.45.1 Setup
Completing the Git Setup Wizard
Setup has finished installing Git on your computer. The application may be launched by selecting the installed shortcuts.
Click Finish to exit Setup.
Launch Git Bash
View Release Notes
Installation complete, now click on 'Finish'
Finish

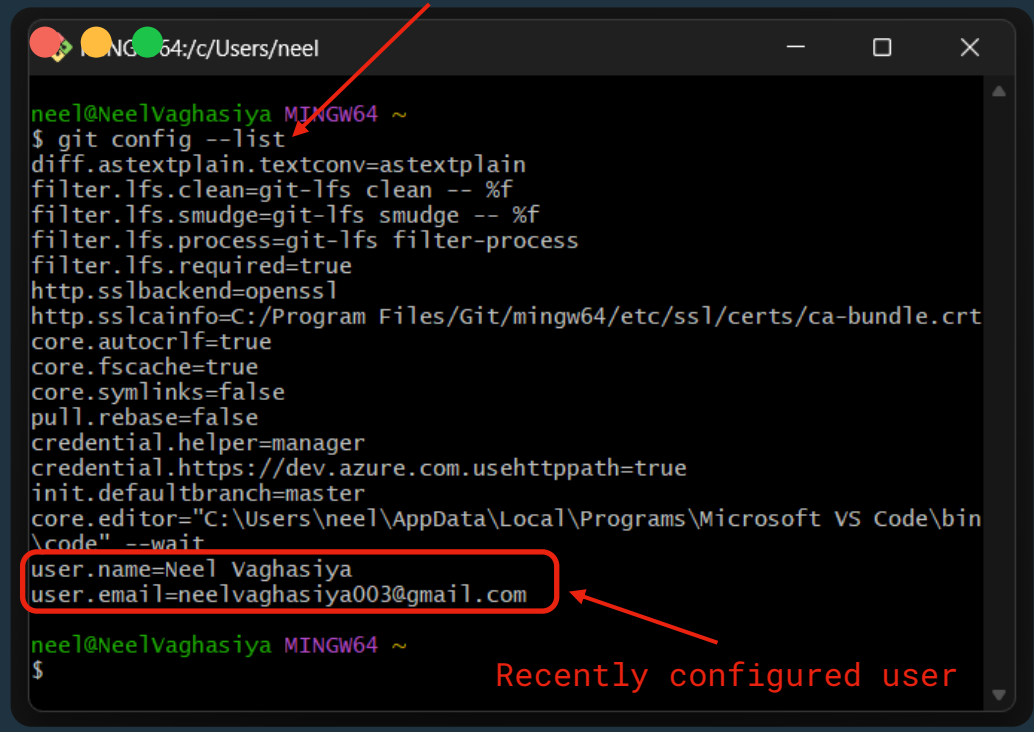

Command Prompt
Microsoft Windows [Version 10.0.22631.3593]
(c) Microsoft Corporation. All rights reserved.
C:\Users\neel>git --version
git version 2.45.1.windows.1
C:\Users\neel>
Run 'git -v' or 'git --version'
in command prompt for verification

Next step is to configuring the user information.

- Open start menu and search for “Git Bash”, and open it.
- Run following commands to configure user information.



Run ‘git config --list’ in git bash to list global git configurations



Setup complete – let’s Git started!!

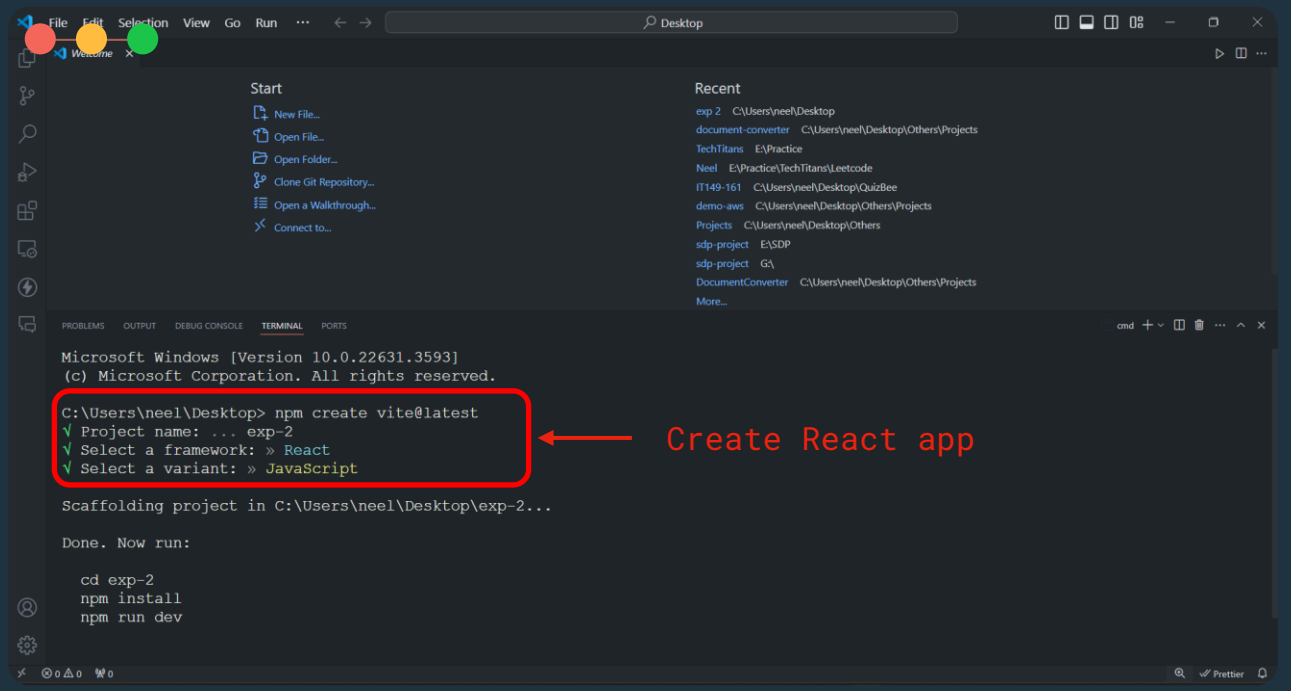
06 IMPORTANT GIT COMMANDS

06 IMPORTANT GIT COMMANDS

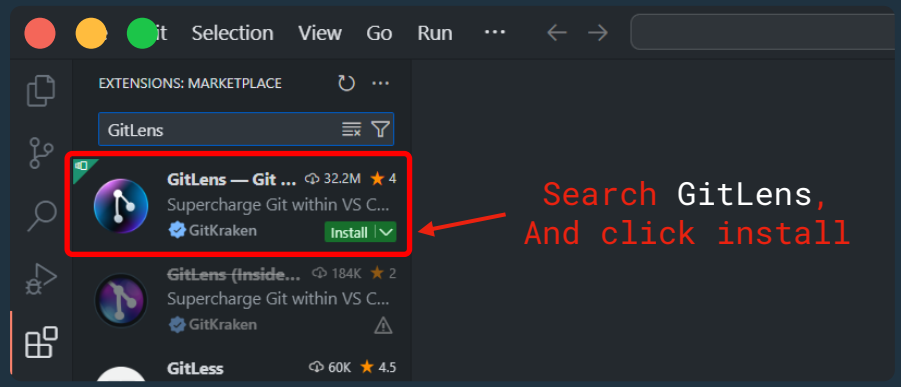
- List of some important commands:
  - init
  - status
  - add
  - commit
  - branch
  - checkout
  - merge
  - rebase
  - stash
  - stash pop
  - stash apply
  - stash drop
  - log
  - blame
  - remote
  - clone
  - push
  - pull

- Let's understand each Git command while working on a simple React app

First step is to create react app using vite.



Then install GitLens extension in VS code



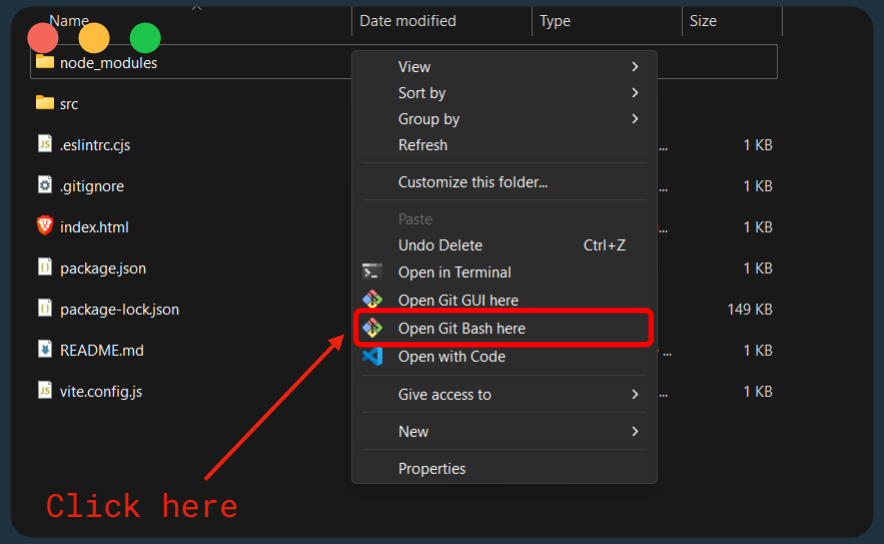
We will create ‘components’ folder under ‘src’ folder and modify the App.jsx

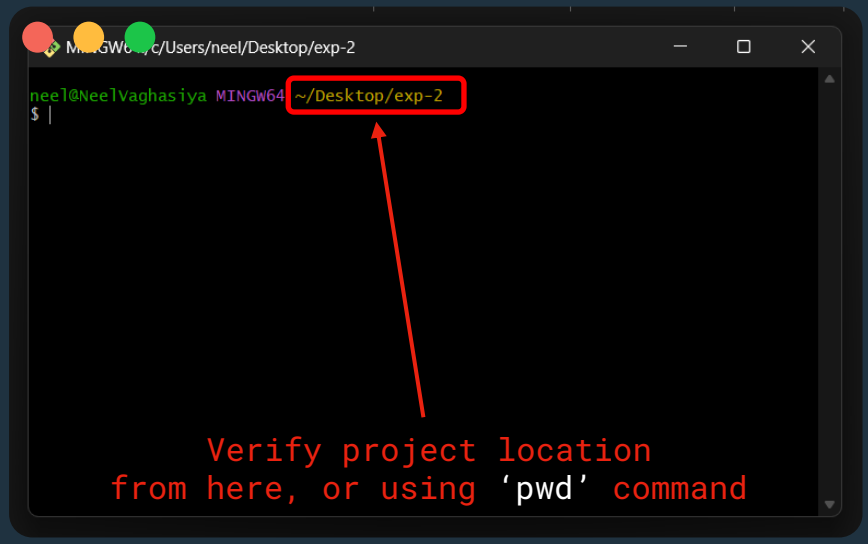
```jsx
import React from 'react'

const App = () => {
  return (
    <div>
      Hello world..!
    </div>
  )
}

export default App
```

We have just updated our project without versioning, so we won’t be able to recover previous files

Now, go to the project folder, right click and select “Open Git bash here”.

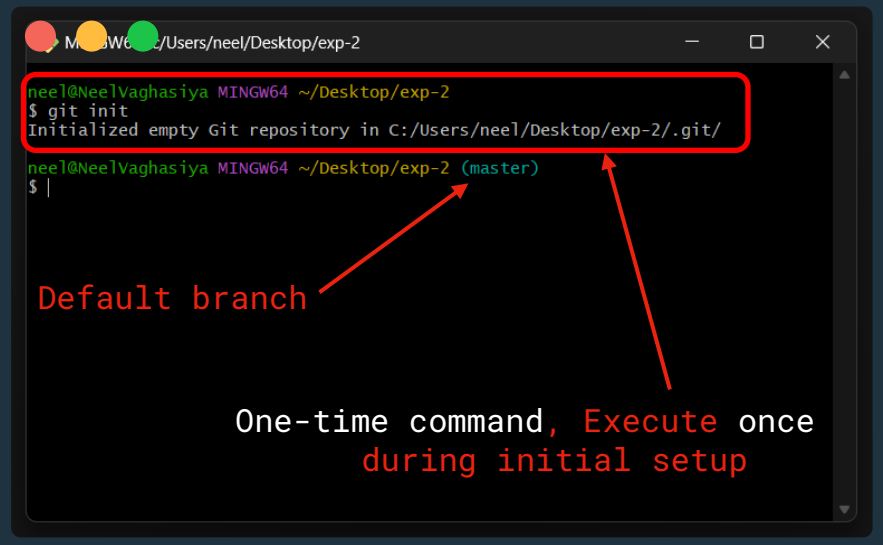
Git bash will open.

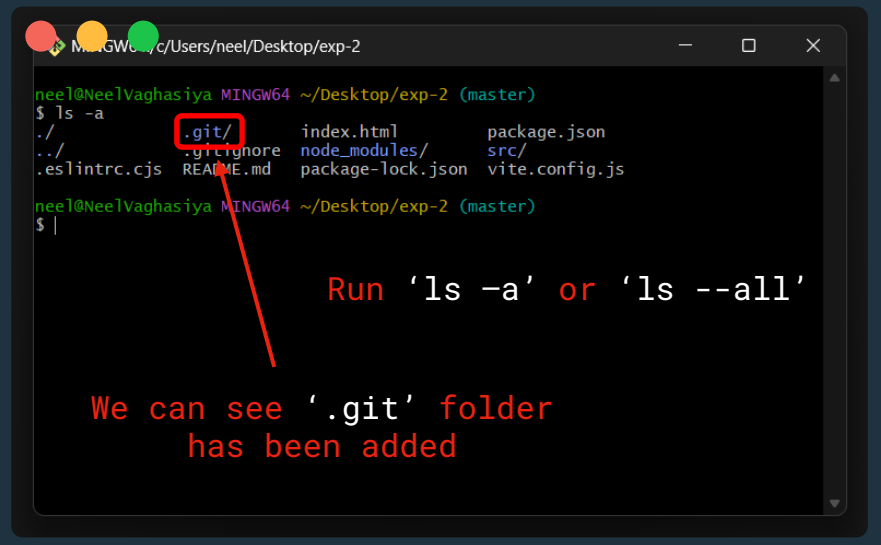




1. git init : First step towards Versioning

- Initializes empty git repository
- Creates ".git" folder (stores details of all changes made into the folder in which git is initialized)

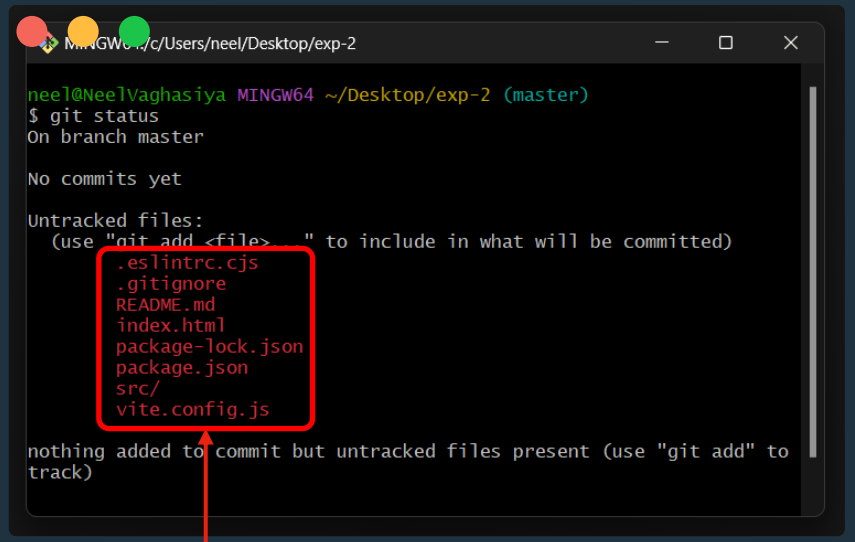




2. git status: View the current status of working directory and staging area

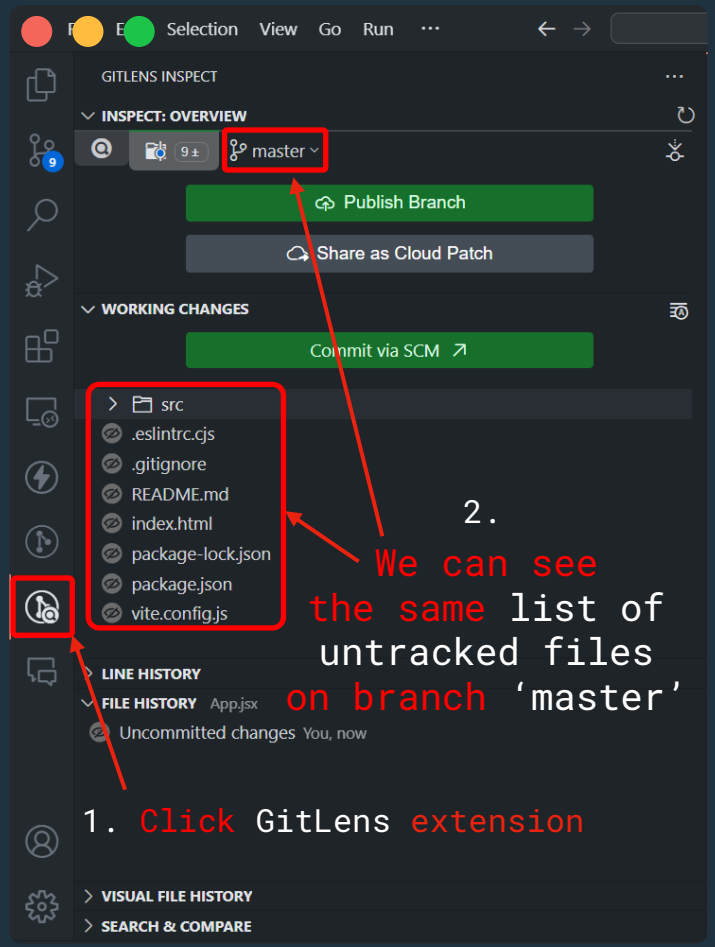
- Syntax: **git status <options>**
- Options: '-s' or '--short': gives output in short format,
  '-b' or '--branch': shows branch and tracking info, etc.
  '--long': long format default output

- It shows the changes to be committed, untracked files, etc.
- Run ‘git status’ in the Git bash, and see output:



Untracked files:

- Files that have not yet been added to Git
- Git is not aware of these files
- Those are not part of Version Control

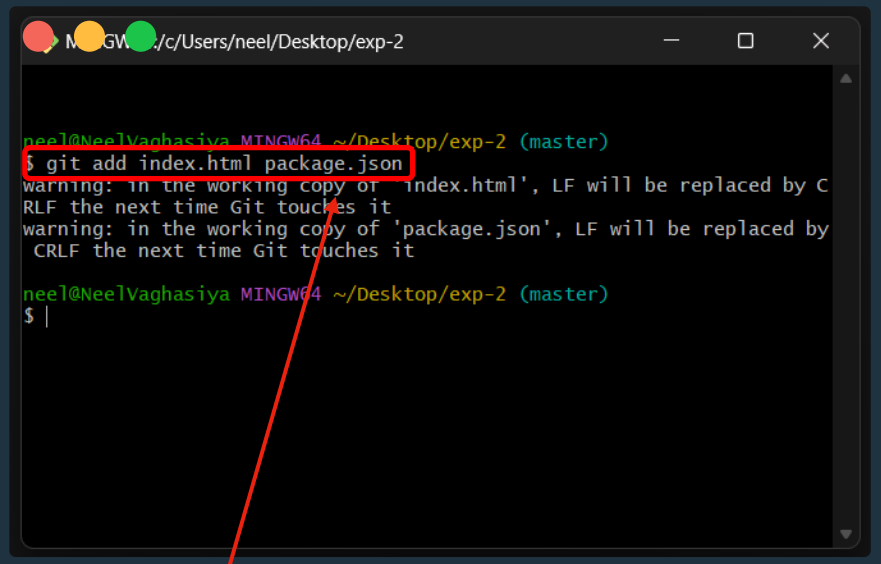


3. git add : Stages the changes

- Syntax: **git add <file(s)>**
- eg: git add file1.txt

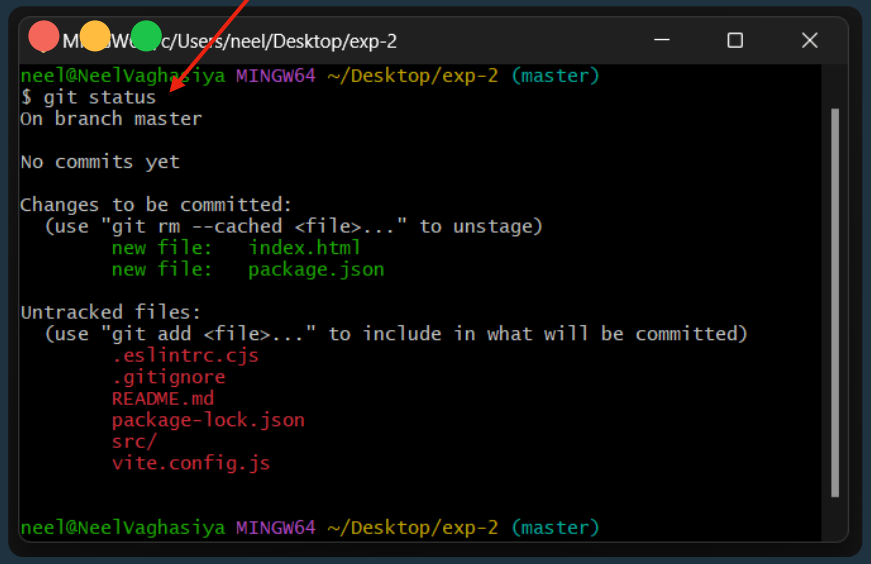
  git add file2.txt file3.txt file4.txt

  git add .
- moves files from untracked or modify state to staging area
- Prepares files for the next commit

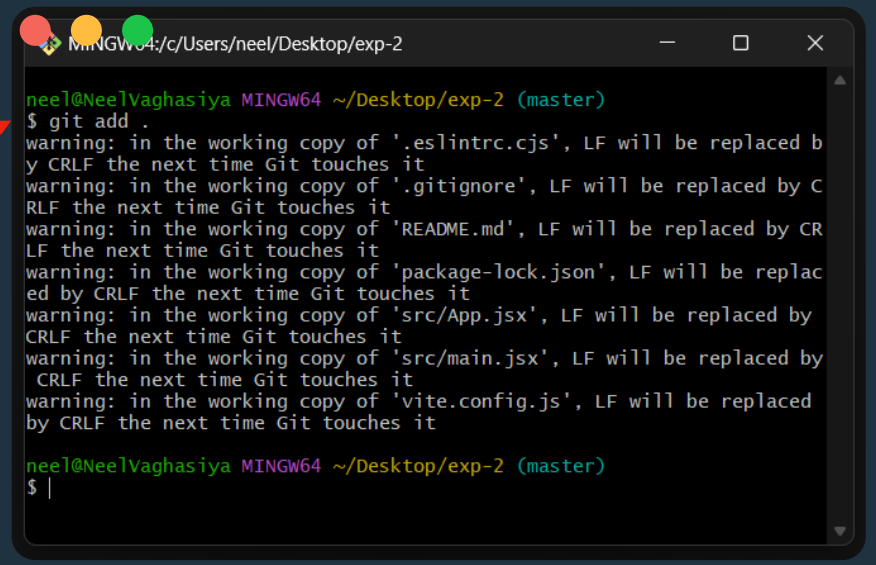


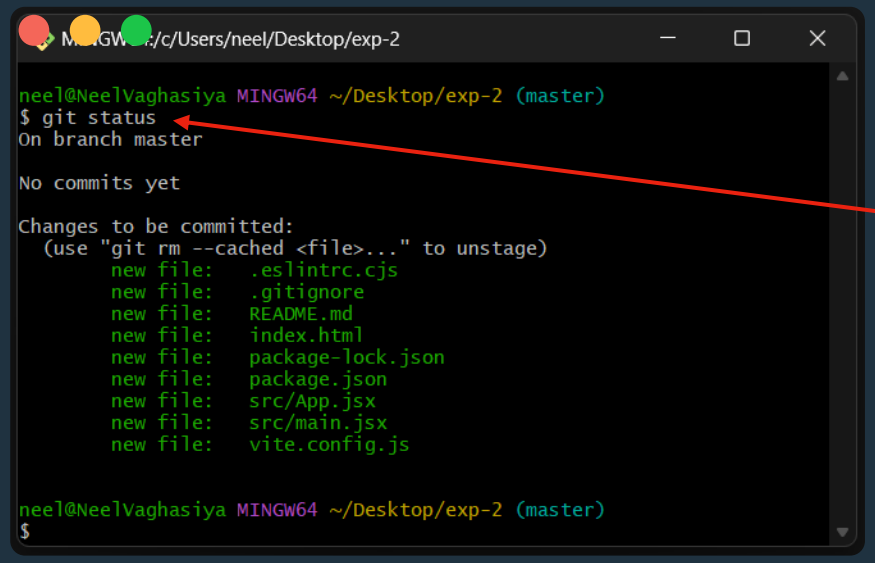
Added index.html and package.json

Now run 'git status'



Run
‘git add .’

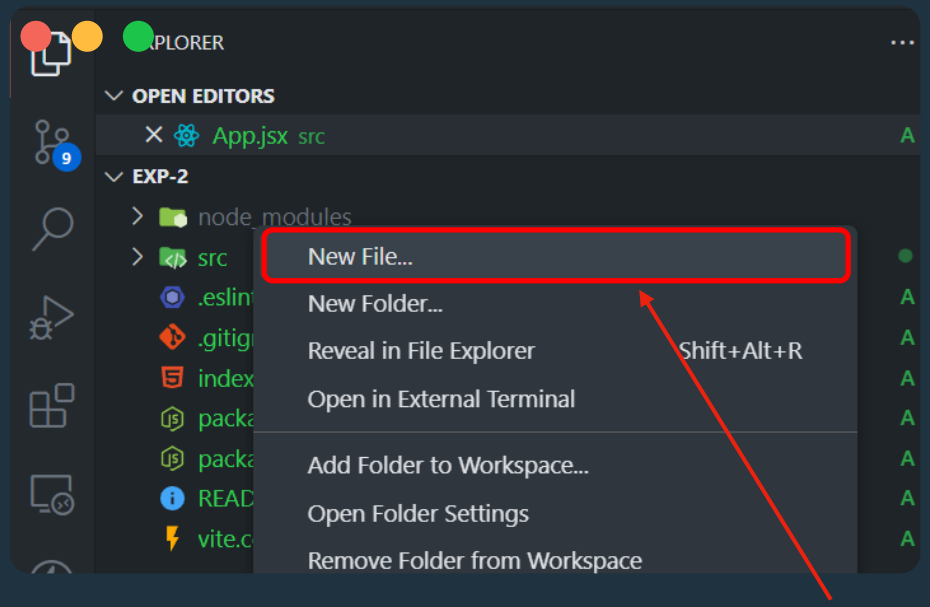


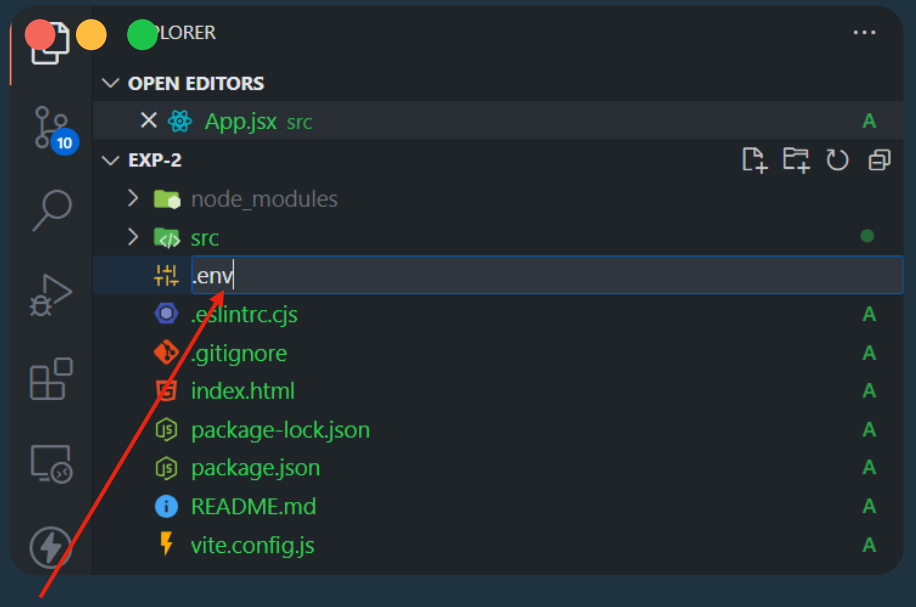


Run
‘git status’

What if we want Git to keep certain files untracked?

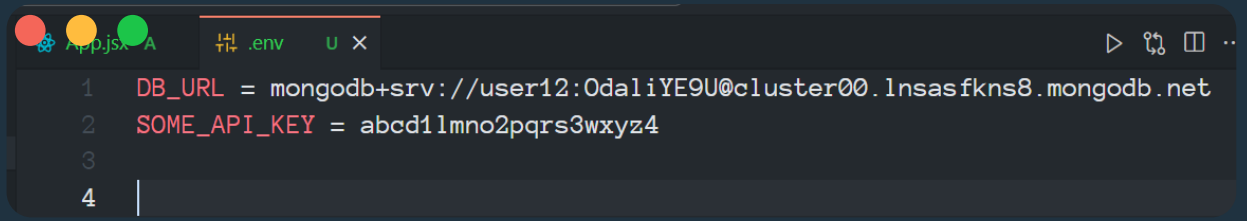
Let’s understand by an example

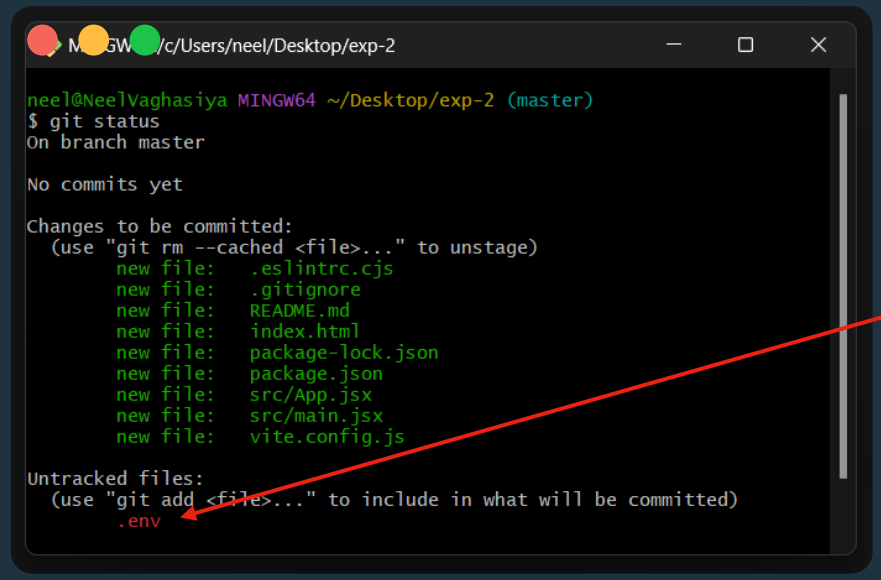




Right click on project tree, and click on 'New File...',
and save it with '.env' name

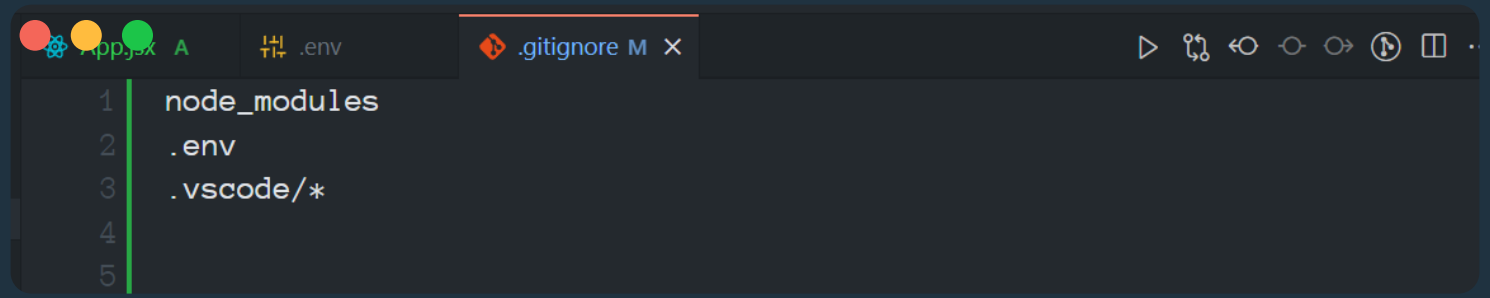
- '.env' file contains sensitive information such as API keys, or database credentials.
- We should not expose these information.



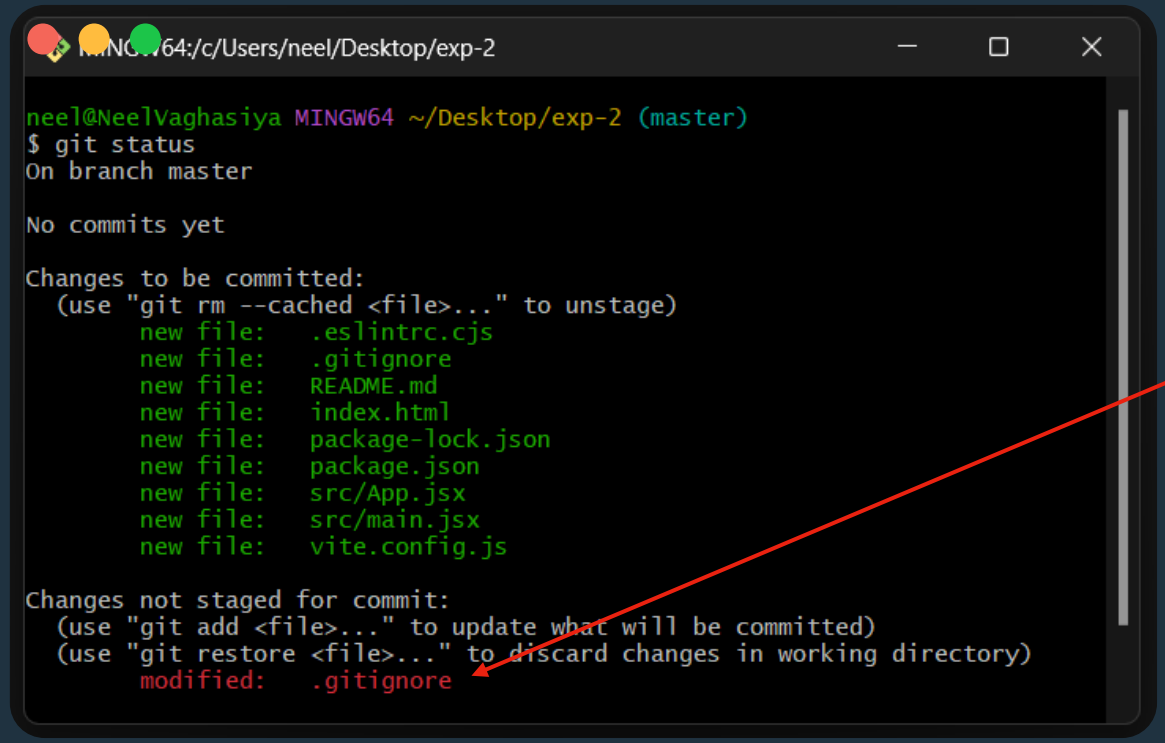


If we run 'git status' we can see that Git suggests adding the '.env' file to the staging area

- For this, create file with name '.gitignore', where git repo is initialized.
- Add the name of the files, or folders into '.gitignore', and save it.



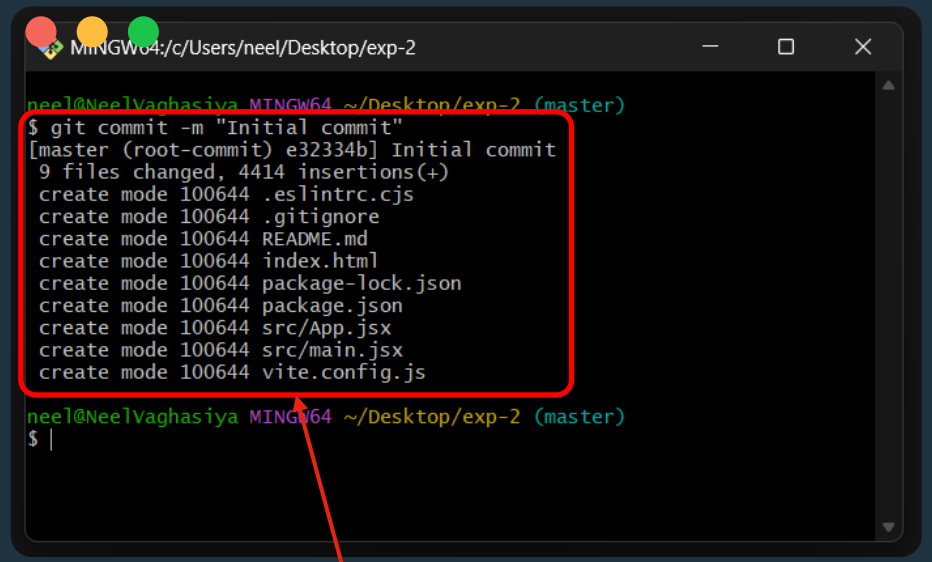
- Again run 'git status' command, and observe output:



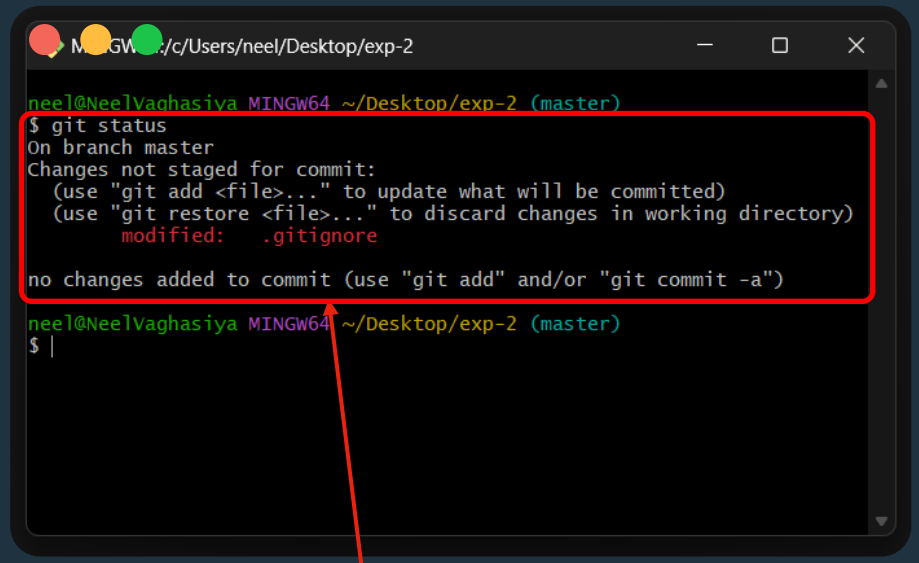
We modified '.gitignore' file, and we can see '.env' has been ignored

4. git commit : Takes snapshot at a specific time

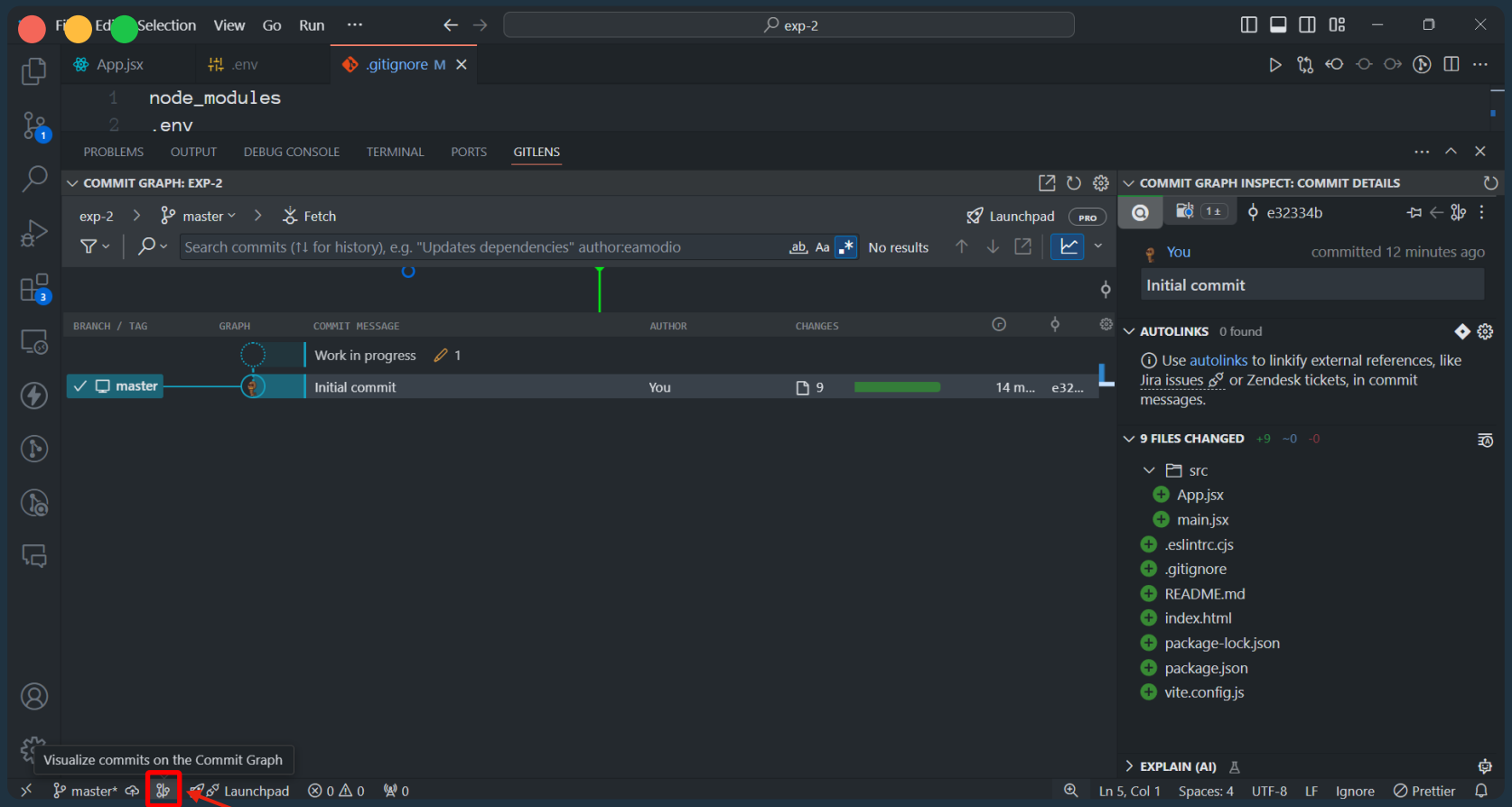
- Syntax: **git commit <options>**
- eg: git commit ← Opens default text editor to write the commit message
  - git commit –m “message here” ← Commit staged changes with a message
  - git commit –am “message here” ← Add changes and commit in one step (for tracked and modified files)
- Commit message should be clear and concise
- Let’s make our first commit



Committed changes on master branch



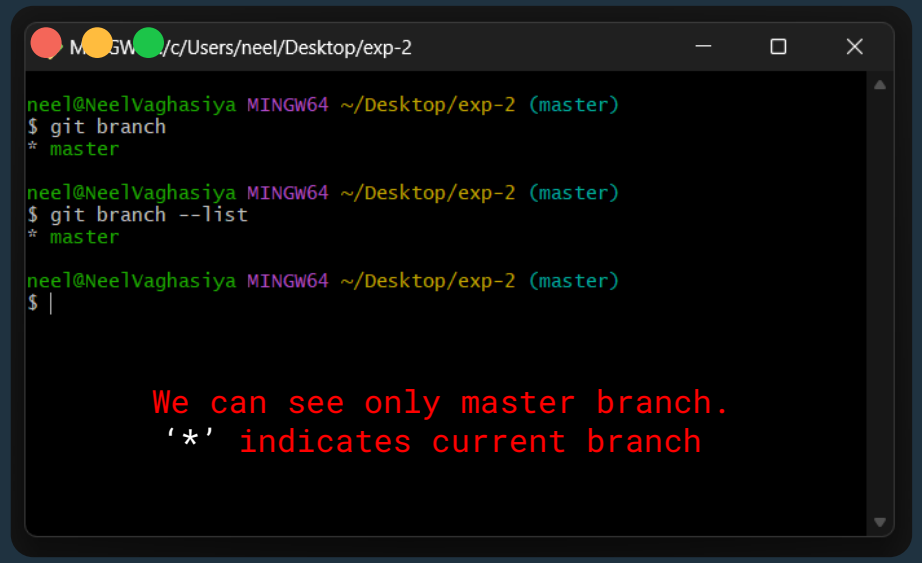
We can see staging area is empty

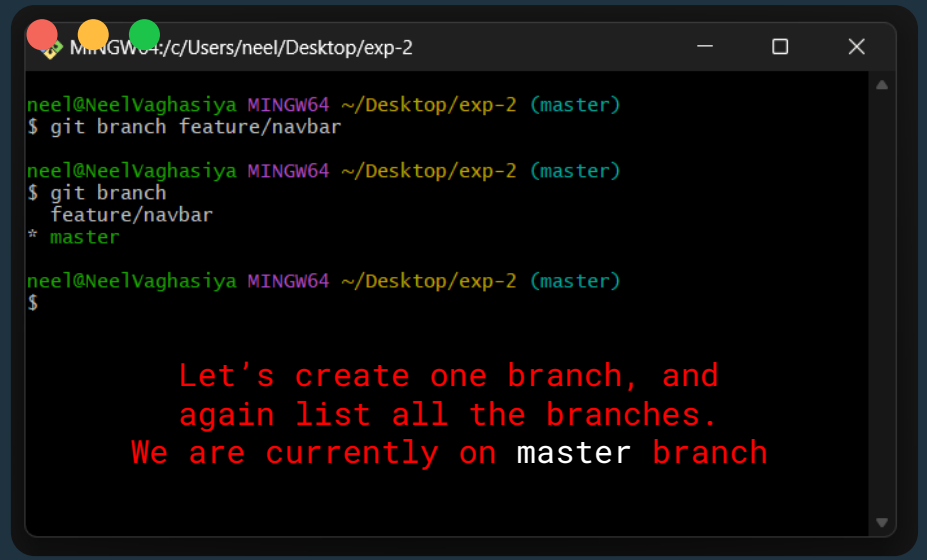


We can see the detailed commit graph,
by clicking this button in VS code

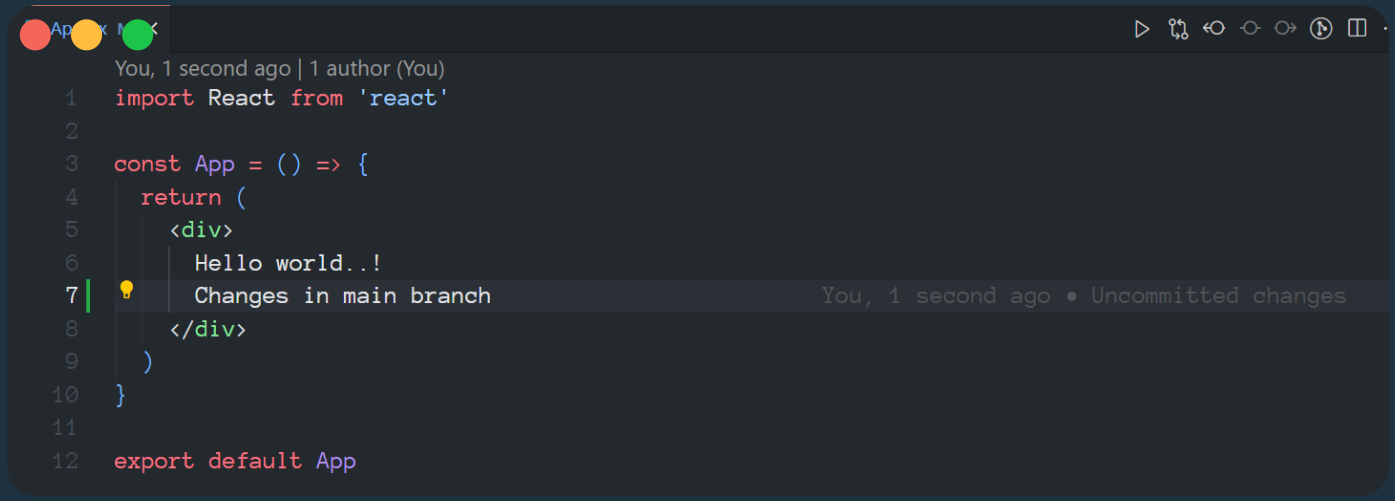
5. git branch : To create, list, rename, and delete branches

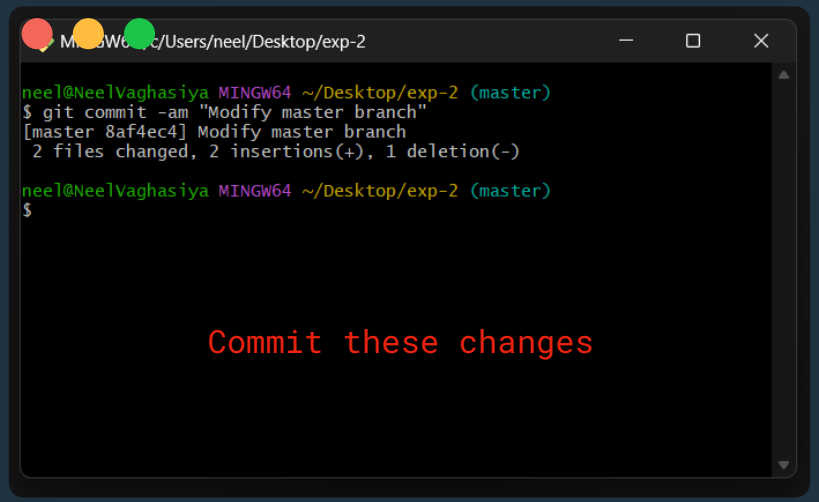
- Syntax: **git branch <options>**
- eg: git branch or git branch --list ← Lists all the branches
  - git branch <branch> ← Creates a new branch named <branch>
  - git branch –d <branch> ← Deletes the <branch> (if it has been merged)
  - git branch –D <branch> ← Force deletes the <branch> (even if it has not been merged)
  - git branch –m <new branch> ← Renames the current branch to <new branch>
- Let’s create one branch

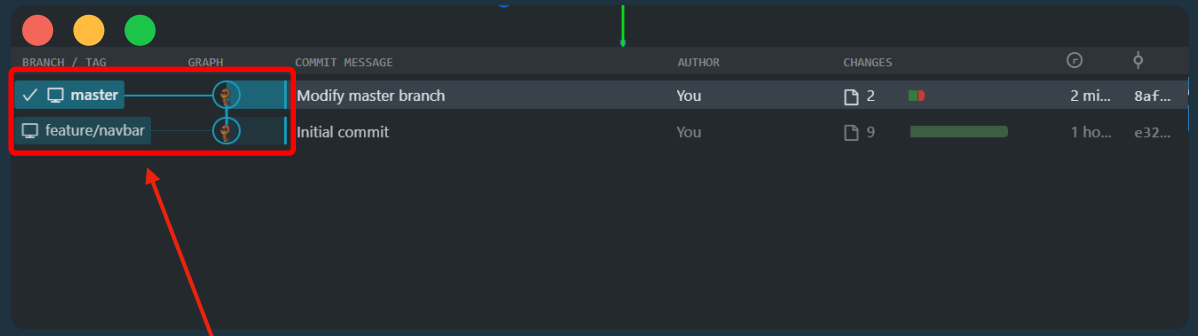




- Now let’s modify master branch and make commit.





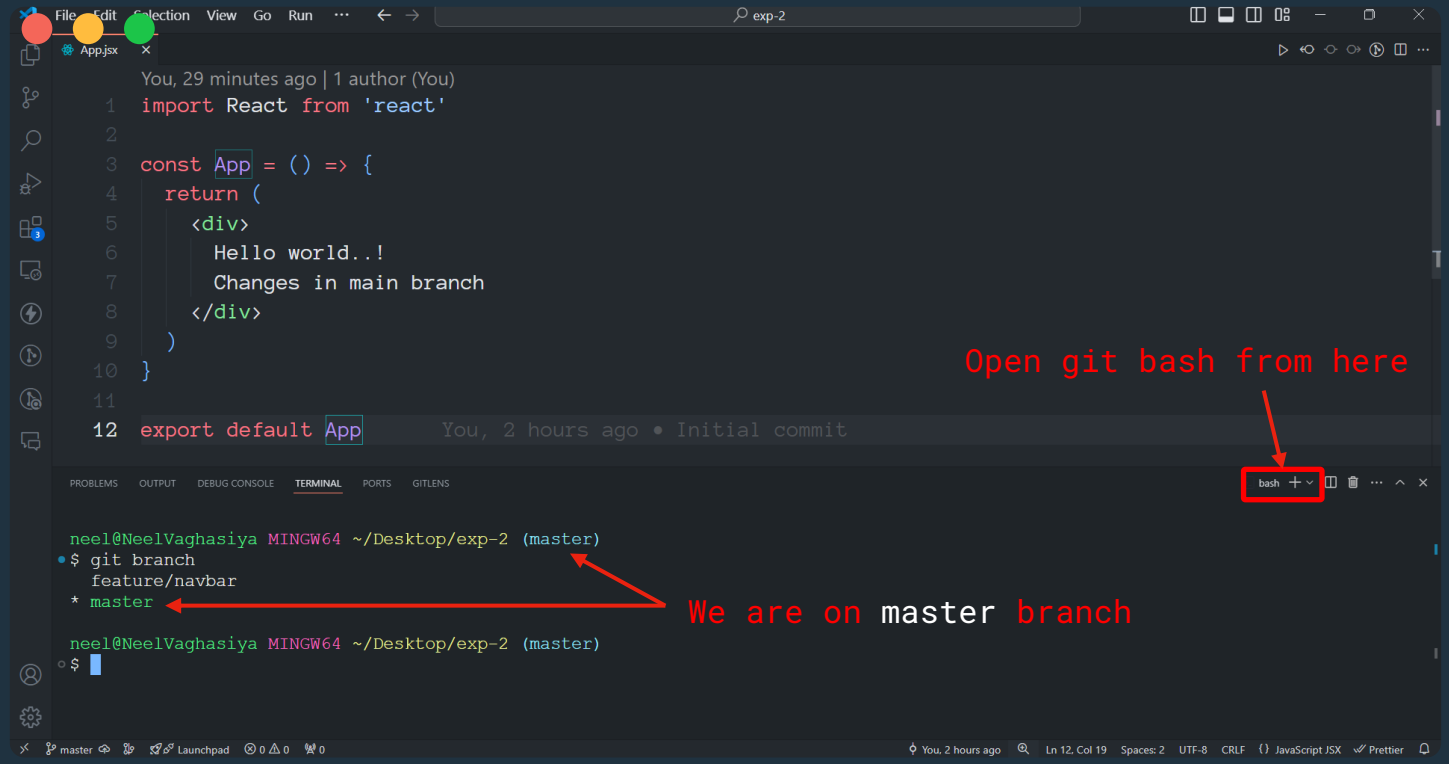


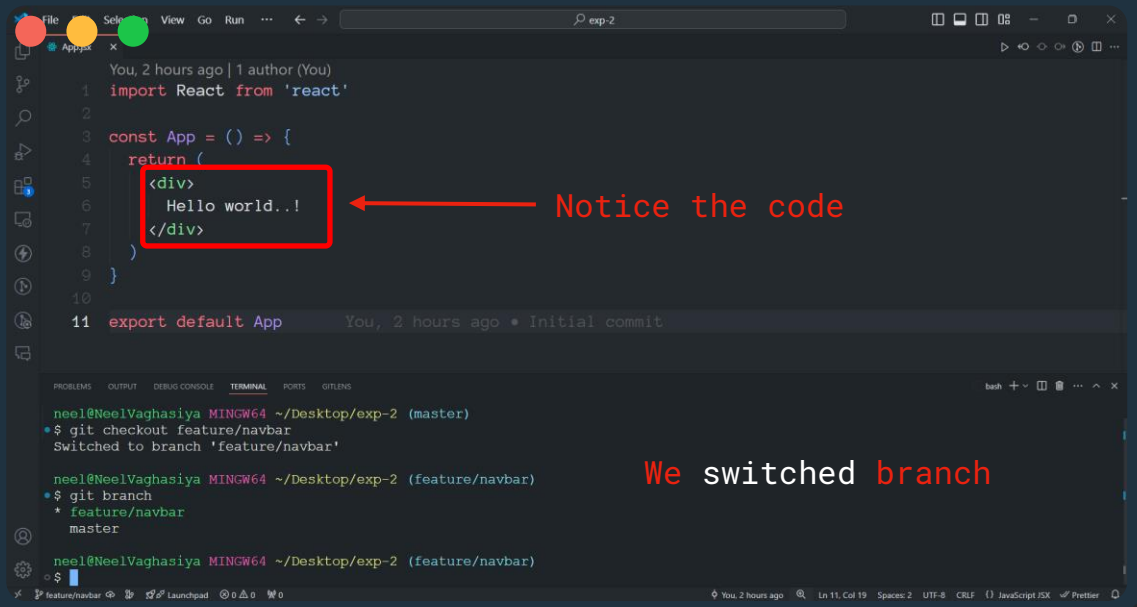
We can see, master branch is one commit ahead of navbar, and there are no commits on navbar branch

6. git checkout : Switching between branches

- Syntax: **git checkout <options>**
- eg: git checkout <branch> ← Switch to an existing branch

  git checkout –b <branch> ← Creates a new branch named <branch> and switch to it
- Let’s see an example, first open Git bash from VS code

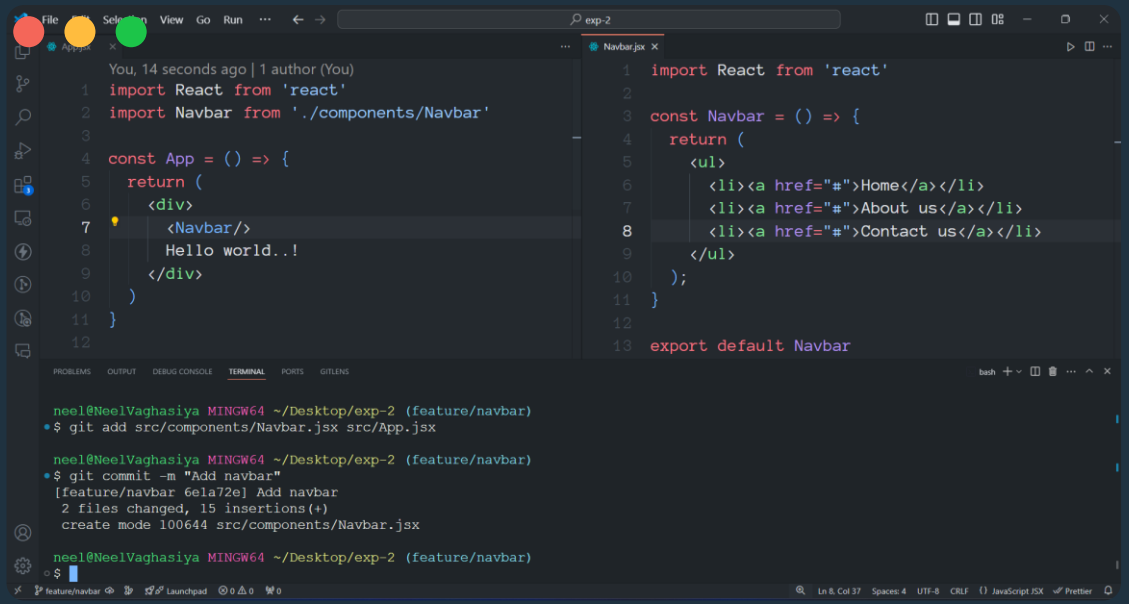


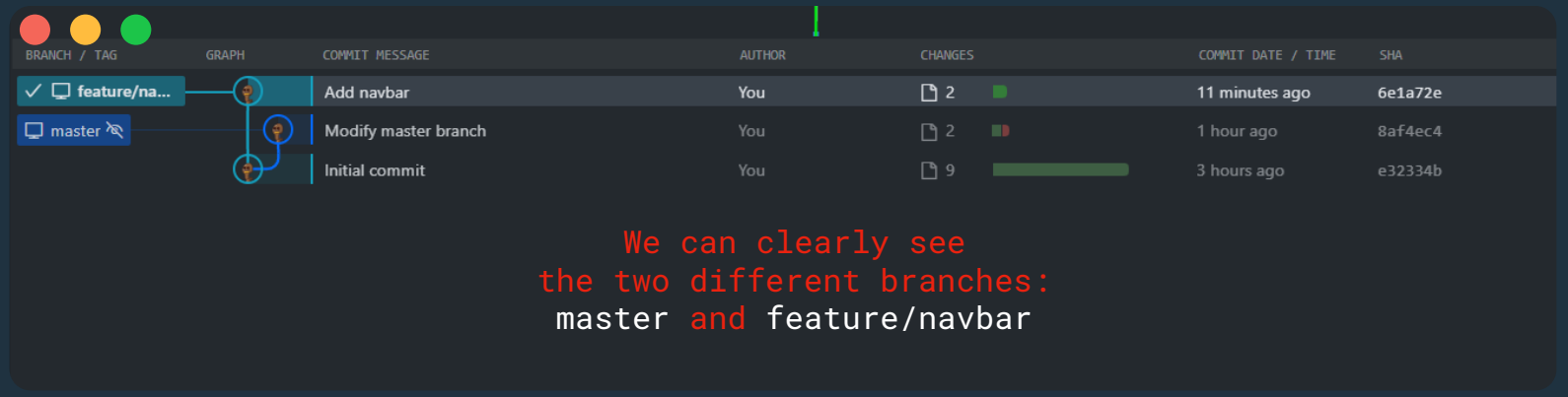


We switched branch

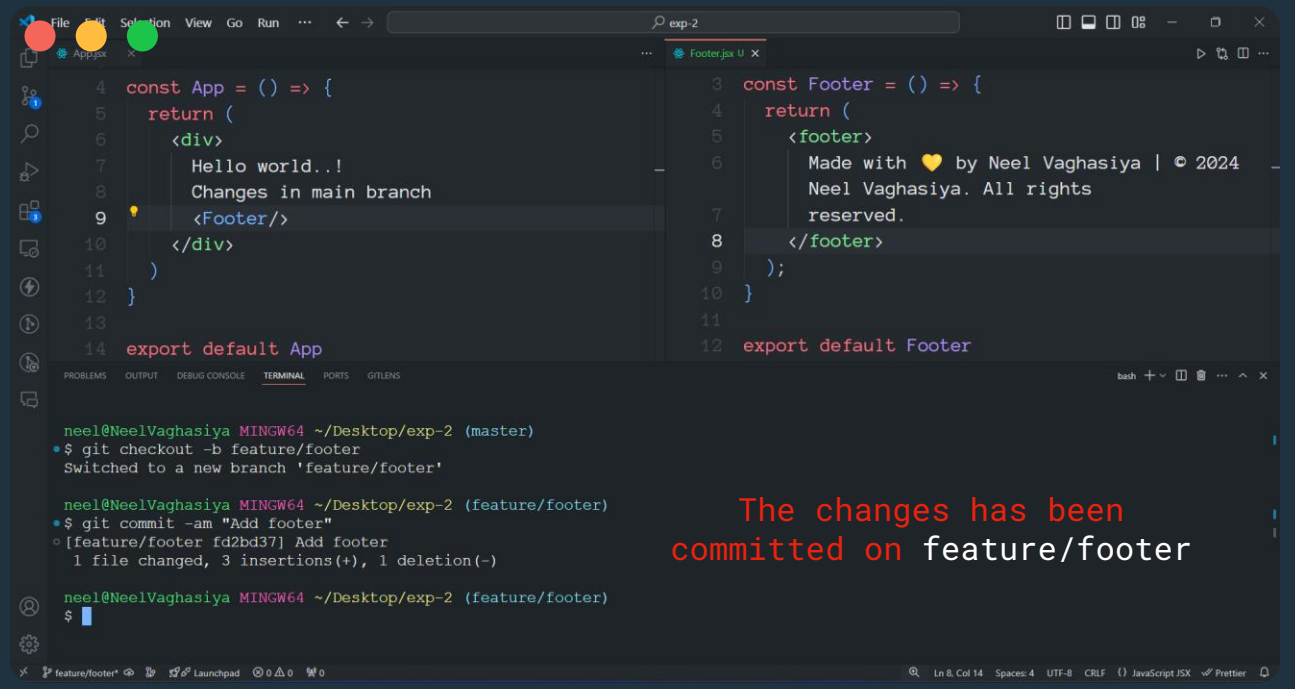
- Let's modify this branch.
- Create Navbar.jsx component, and use it inside App.jsx.
- Add and commit the changes.

- The changes has been committed.
- Now see the commit graph.

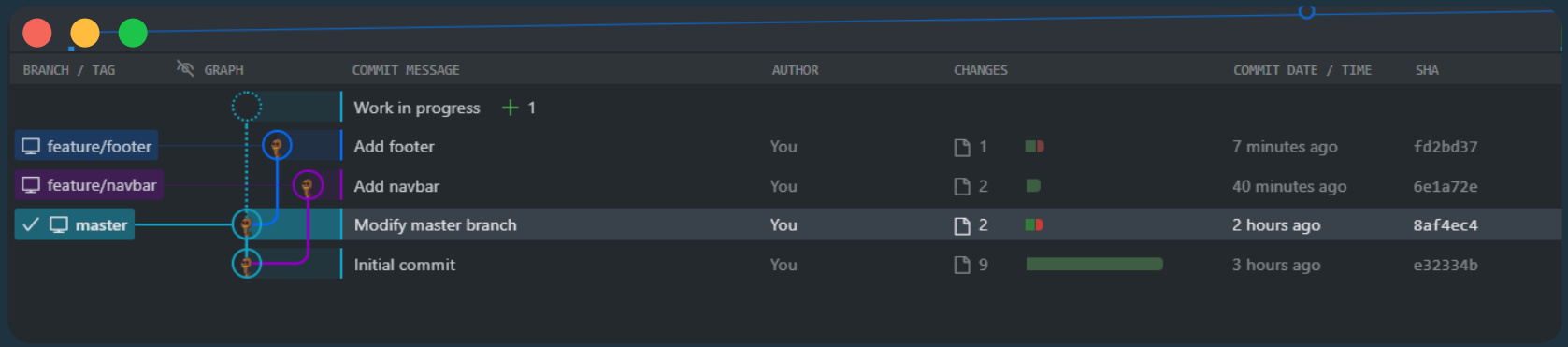




- Now create a new branch named feature/footer from master and switch to it.
- Add footer component and commit changes.

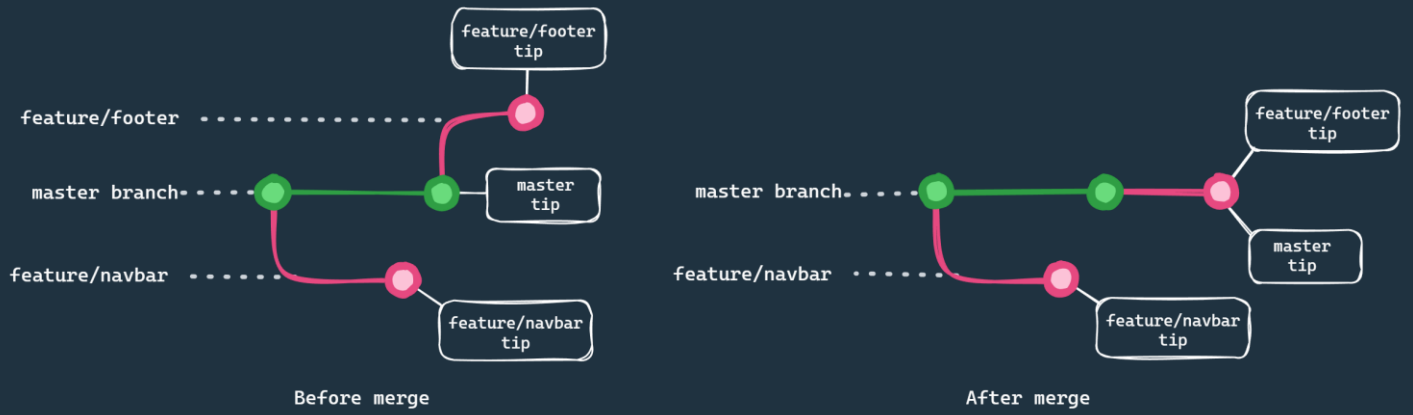


- Checkout to master and see commit graph for better understanding.



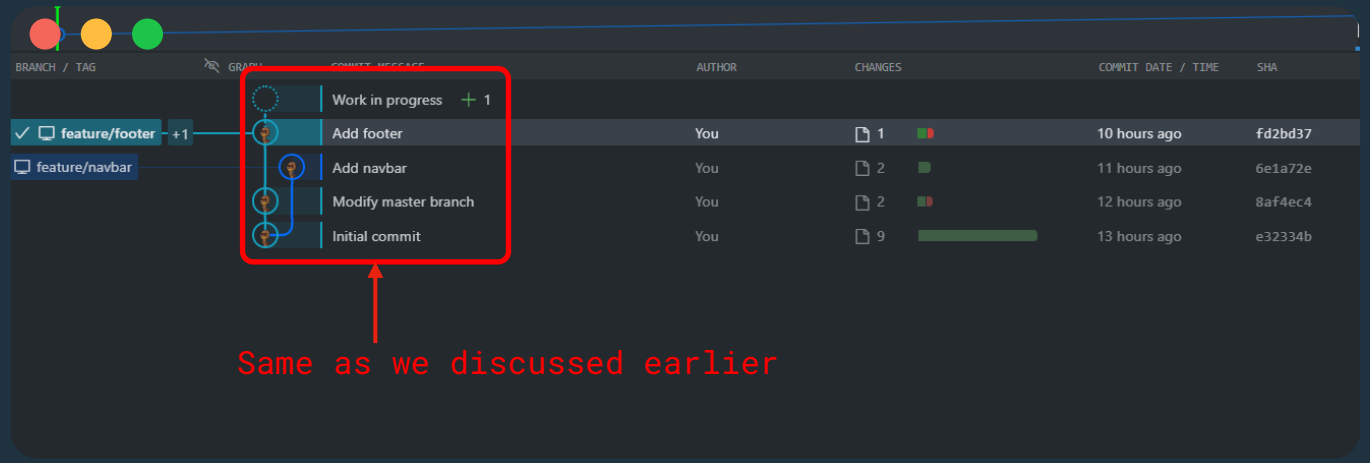
7. `git merge`: Combining changes from different branches

- Syntax: **`git merge <branch>`** ← Merge <branch> to the current branch
- **Fast Forward Merge:** If the current branch's tip is an ancestor of the other branch's tip

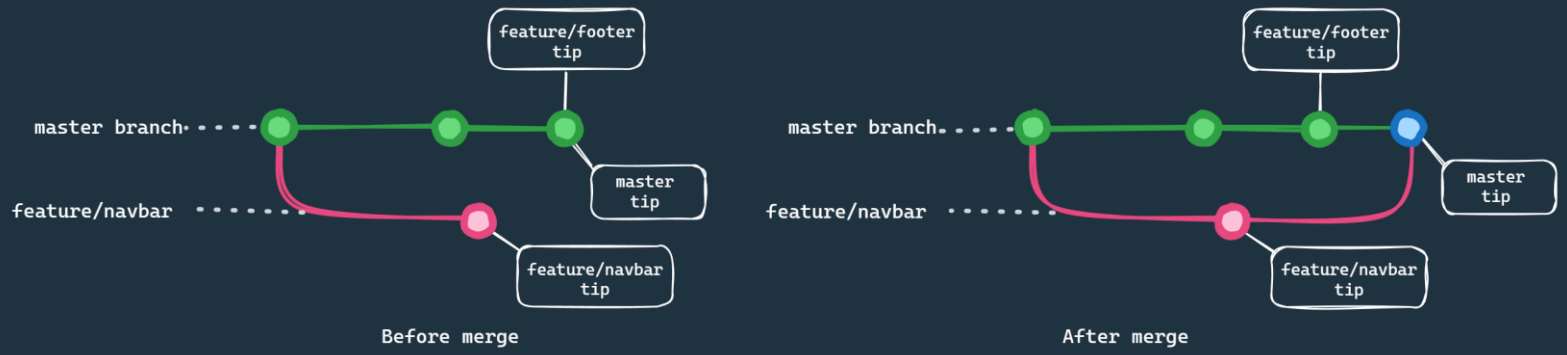


- Now let’s merge our feature/footer branch to master branch.
- First checkout to master branch, then run merge command

- Let’s check the commit graph.



- **Three-Way Merge:** If both branches have new commits since they diverged.



- Now let's try to merge our feature/navbar to the master branch

```
OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git branch
  feature/footer
  feature/navbar
* master

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git merge feature/navbar
Auto-merging src/App.jsx
CONFLICT (content): Merge conflict in src/App.jsx
Automatic merge failed; fix conflicts and then commit the result.

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$
```

Conflict!!
But why??

Let's see..!

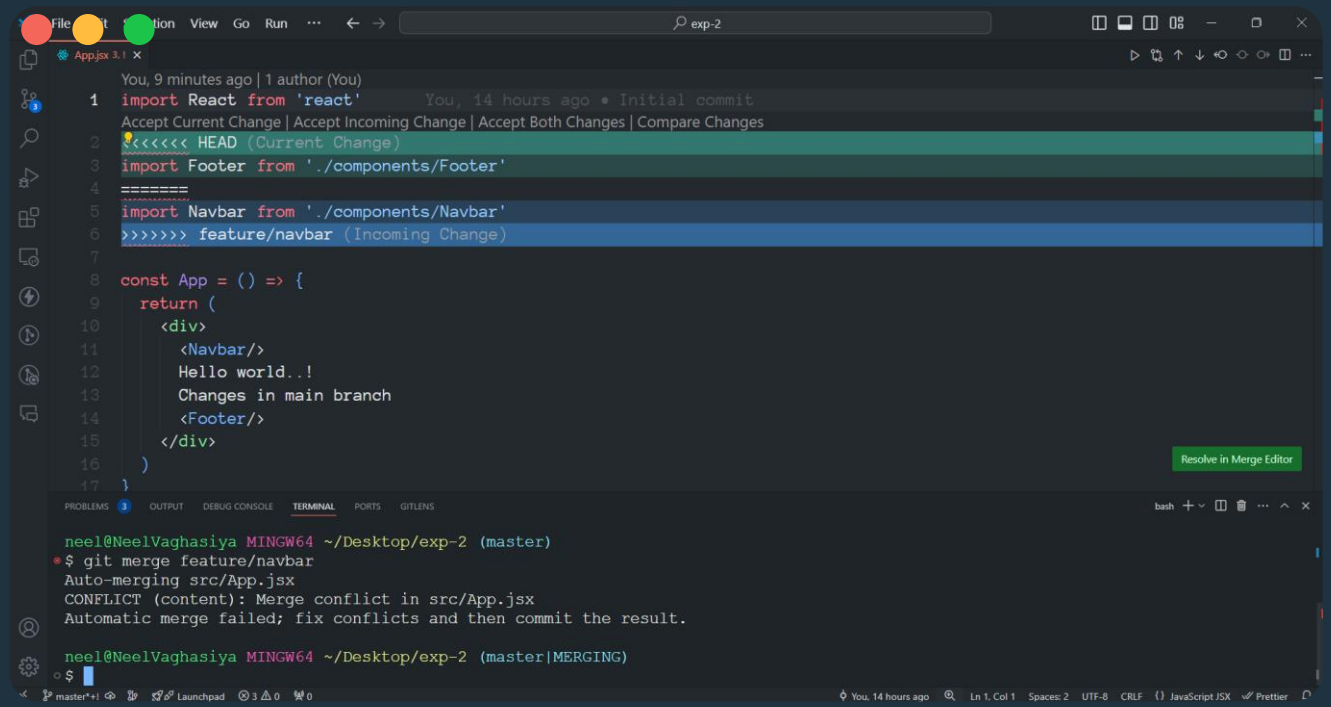
```
You, 11 hours ago | 1 author (You)
1  import React from 'react'        You, 14 hours ago • Initial commit
2  import Footer from './components/Footer'
3
4  const App = () => {
5    return (
6      <div>
7        Hello world..!
8        Changes in main branch
9        <Footer/>
10     </div>
11   )
12 }
13
14 export default App
```

Master branch

```
You, 11 hours ago | 1 author (You)
1  import React from 'react'        You, 14 hours ago • Initial commit
2  import Navbar from './components/Navbar'
3
4  const App = () => {
5    return (
6      <div>
7        <Navbar/>
8        Hello world..!
9      </div>
10   )
11 }
12
13 export default App
```

feature/navbar branch

Resolve this conflict manually Inside editor



<<<<<<< HEAD

Current branch content

=======

Incoming branch content

>>>>>>> BRANCH NAME

```
You, 11 hours ago | 1 author (You)
1  import React from 'react'
2  import Footer from './components/Footer'
3  import Navbar from './components/Navbar'
4
5  const App = () => {
6    return (
7      <div>
8        <Navbar/>      You, 12 hours ago • Add navbar
9        Hello world..!
10       Changes in main branch
11       <Footer/>
12     </div>
13   )
14 }
15
16 export default App
```

Resolved conflict

- We have resolved the conflict. Now check status, and add and commit changes.

```
DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   src/components/Navbar.jsx

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   src/App.jsx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        src/components/Footer.jsx


neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git add src/App.jsx src/components/Footer.jsx

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git commit
```

```
COMMIT_EDITMSG
1  Merge branch 'feature/navbar'
2
3  # Conflicts:
4  #   src/App.jsx
5  #
6  # It looks like you may be committing a merge.
7  # If this is not correct, please run
8  #   git update-ref -d MERGE_HEAD
9  # and try again.
10
11
12 # Please enter the commit message for your changes. Lines starting
13 # with '#' will be ignored, and an empty message aborts the commit.
14 #
15 # On branch master
16 # All conflicts fixed but you are still merging.
17 #
18 # Changes to be committed:
19 #   modified:   src/App.jsx

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git add src/App.jsx src/components/Footer.jsx

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git commit
hint: Waiting for your editor to close the file...
```

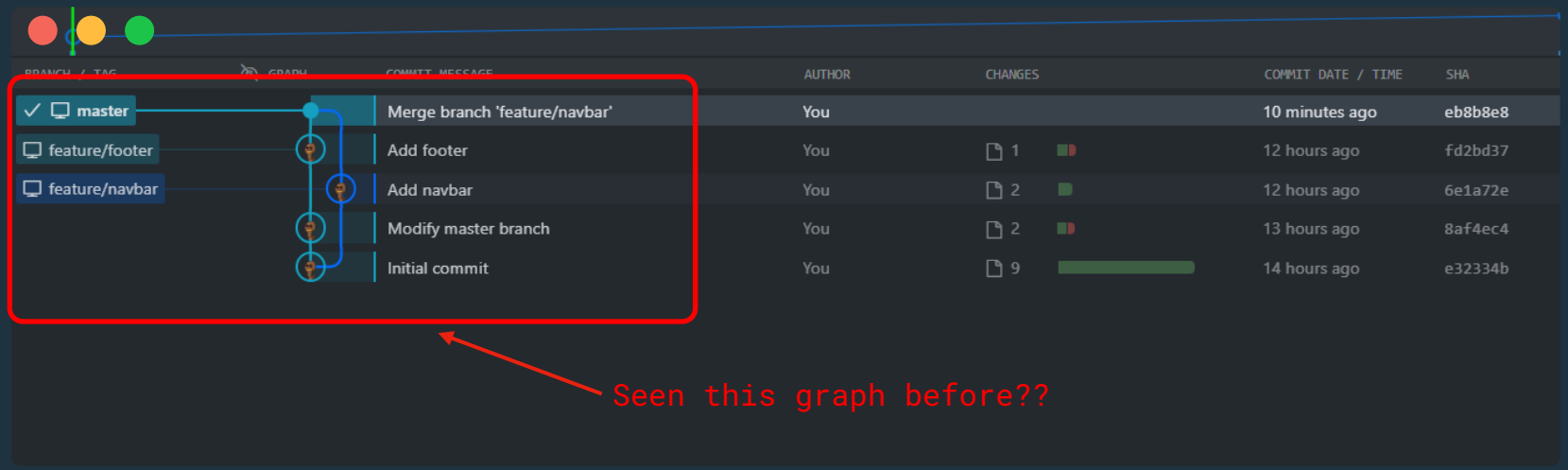
Enter commit message here then save and close file

Since we ran 'git commit', so git will open Default editor and it will wait for commit message

```jsx
import React from 'react'
import Footer from './components/Footer'
import Navbar from './components/Navbar'

const App = () => {
  return (
    <div>
      <Navbar/>      You, 12 hours ago • Add navbar
      Hello world..!
      Changes in main branch
      <Footer/>
    </div>
  )
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master|MERGING)
$ git commit
[master eb8b8e8] Merge branch 'feature/navbar'

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git status
On branch master
nothing to commit, working tree clean

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

- We successfully merged branches, now let's take a look at commit graph



8. git rebase: moves sequence of commits on top of another branch tip

- Allows to change base of the branch
- Syntax: **git rebase <branch>** ← Moves base of the current branch on tip of <branch>
- **!! NEVER USE REBASE COMMAND WHILE WORKING ON MASTER BRANCH !!**

Why rebasing?

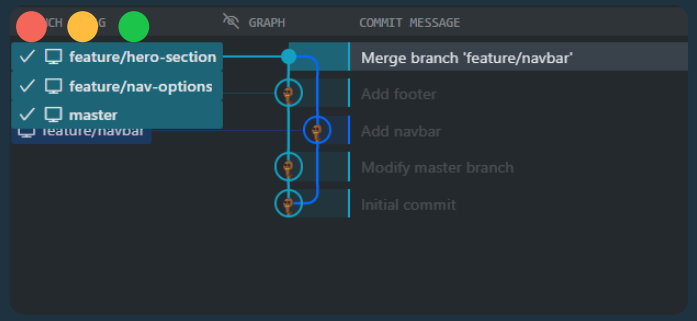
- Consider two developers working on different branches.
- One developer has finished his work and merged it into master.
- Now the other developer wants to integrate those changes, so he will rebase his branch onto the latest commit in master.

Key Benefits:

- Integrate latest changes
- Maintain clean commit history
- Simplify merging, without extra merge commit

- Let’s implement this by example:
- First we will create two different branches, from master branch

```
DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git branch feature/nav-options

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git branch feature/hero-section

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```



- Now, we will develop the hero-section, in feature/hero-section branch.

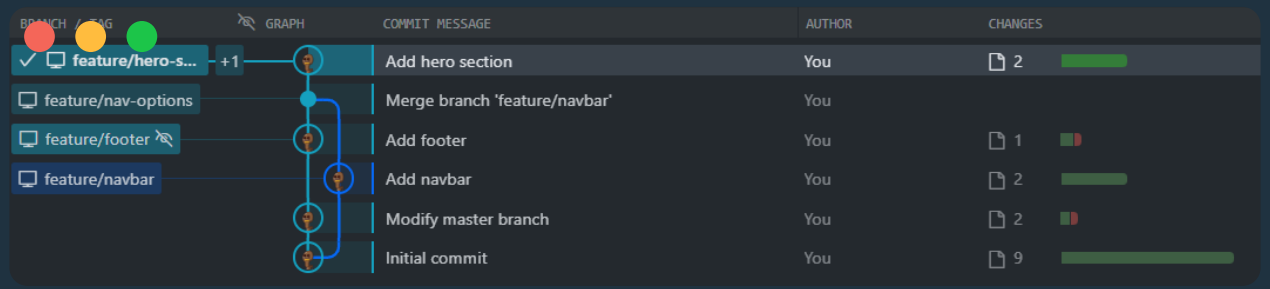
```jsx
import Navbar from './components/Navbar'
import HeroSection from './components/HeroSection'

const App = () => {
  return (
    <div>
      <Navbar/>
      Hello world..!
      Changes in main branch
      <HeroSection/>
      <Footer/>
    </div>
  )
}

export default App
```

HeroSection.jsx U

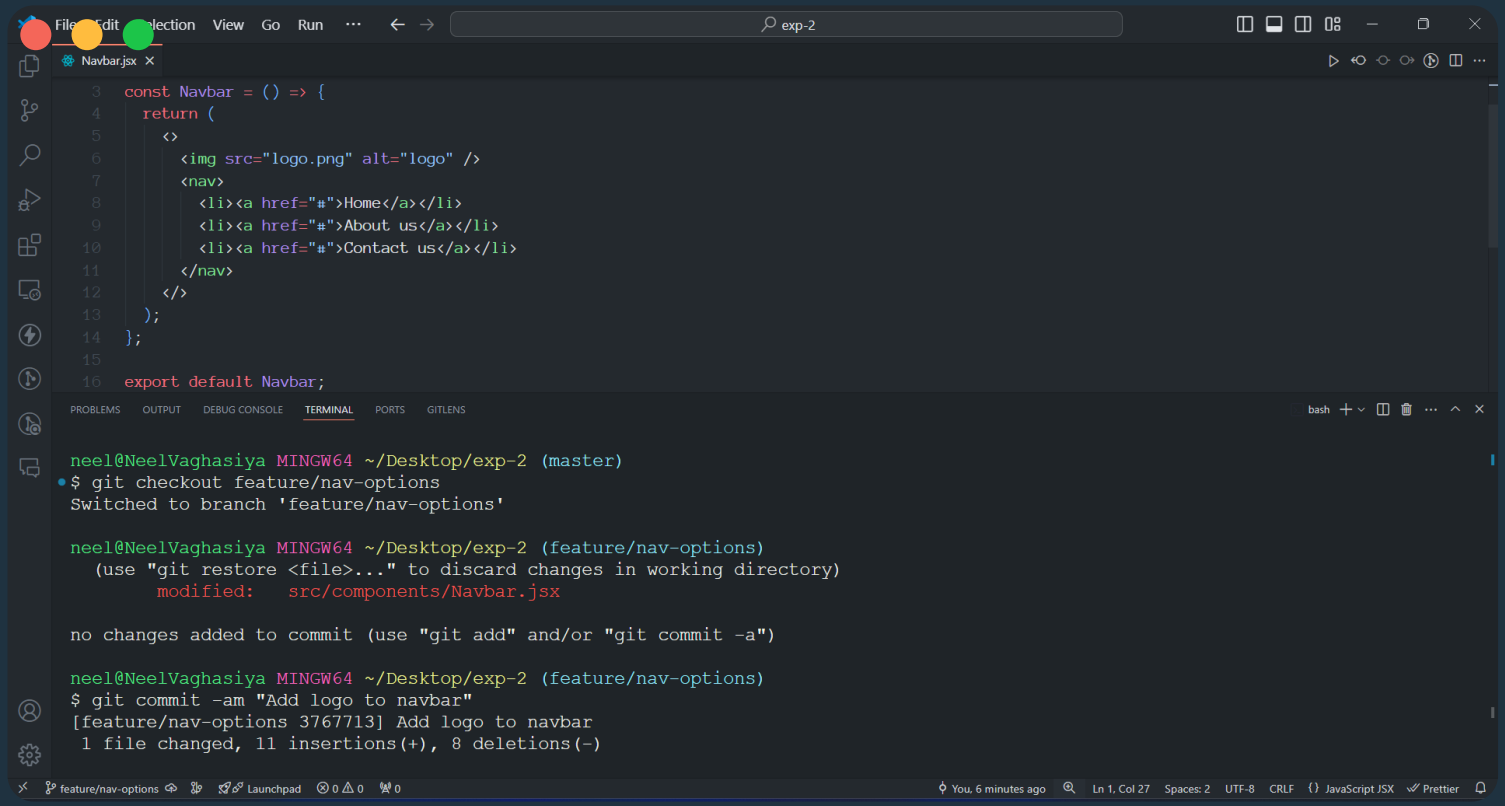
```jsx
import React from "react";

const HeroSection = () => {
  return (
    <>
      <section id="home-section">This is home section.</section>
      <section id="about-us-section">This is about us section.</section>
      <section id="contact-us-section">This is contact us section.</section>
    </>
  );
};

export default HeroSection;
```

- Check status, add and commit changes, and merge the branch

```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (feature/hero-section)
$ git status
On branch feature/hero-section
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:   src/App.jsx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        src/components/HeroSection.jsx

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

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (feature/hero-section)
$ git add src/App.jsx  src/components/HeroSection.jsx

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (feature/hero-section)
$ git commit -m "Add hero section"
[feature/hero-section 4fb32a4] Add hero section
 2 files changed, 15 insertions(+)
 create mode 100644 src/components/HeroSection.jsx

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (feature/hero-section)
$ git checkout master
Switched to branch 'master'

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git merge feature/hero-section
Updating eb8b8e8..4fb32a4
Fast-forward
 src/App.jsx                 |  2 ++
 src/components/HeroSection.jsx | 13 +++++++++++++
 2 files changed, 15 insertions(+)
 create mode 100644 src/components/HeroSection.jsx
```

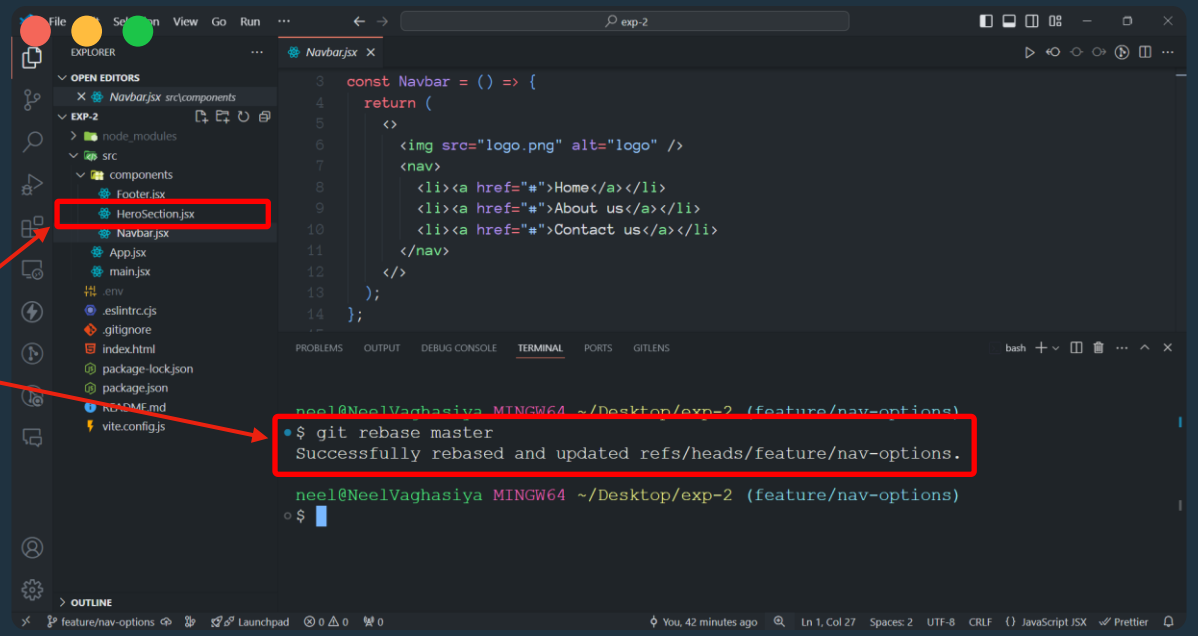


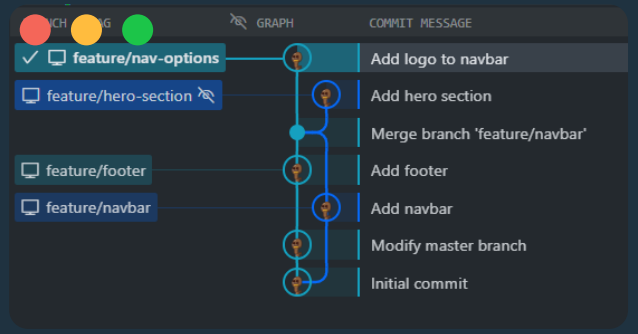
- Now we will checkout to feature/nav-options branch, and will add logo to the navbar



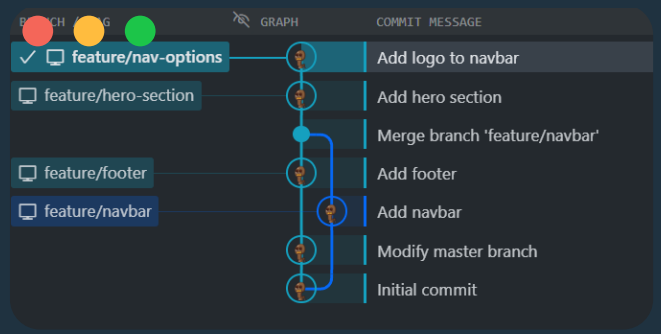
- In this branch we do not have hero-section.
- If we want to integrate those changes into our branch, we have to rebase our branch.
- Make sure, the current branch **must not be master** while performing rebasing.

HeroSection.jsx has been added after rebasing

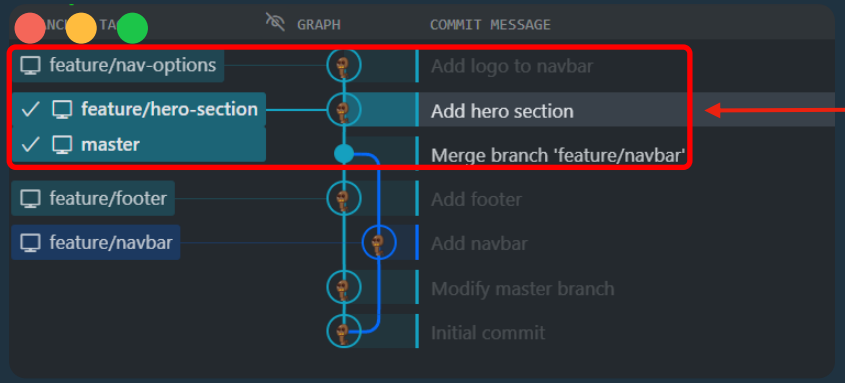




Before rebase



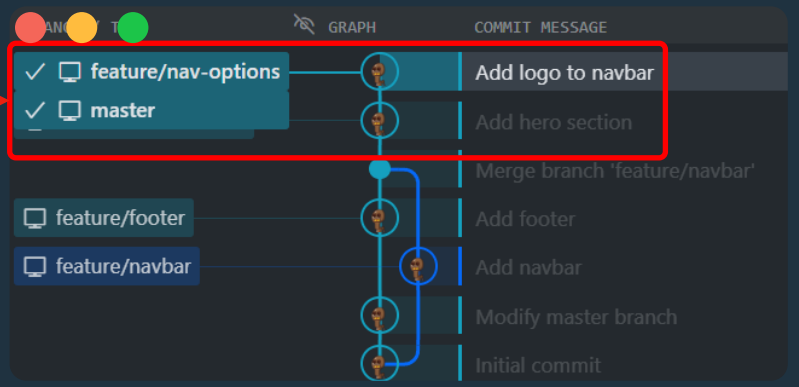
After rebase



- We can see, that rebased branch is ahead of master, so we need to merge it with master branch.
- Switch to master, and merge this branch.
- This will not create new merge commit, as we are doing Fast-Forward merge.

```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (feature/nav-options)
$ git checkout master
Switched to branch 'master'

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git merge feature/nav-options
Updating 4fb32a4..a039c2e
Fast-forward
 src/components/Navbar.jsx | 19 ++++++++++++--------
 1 file changed, 11 insertions(+), 8 deletions(-)

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

Merged without any extra merge commit



9. git stash: temporarily stores unfinished changes

- Syntax: **git stash save "message"** ← Stash the changes with description

  git stash list ← Shows a list of stashed changes
- **It saves the staged and unstaged changes, and resets the working directory.**

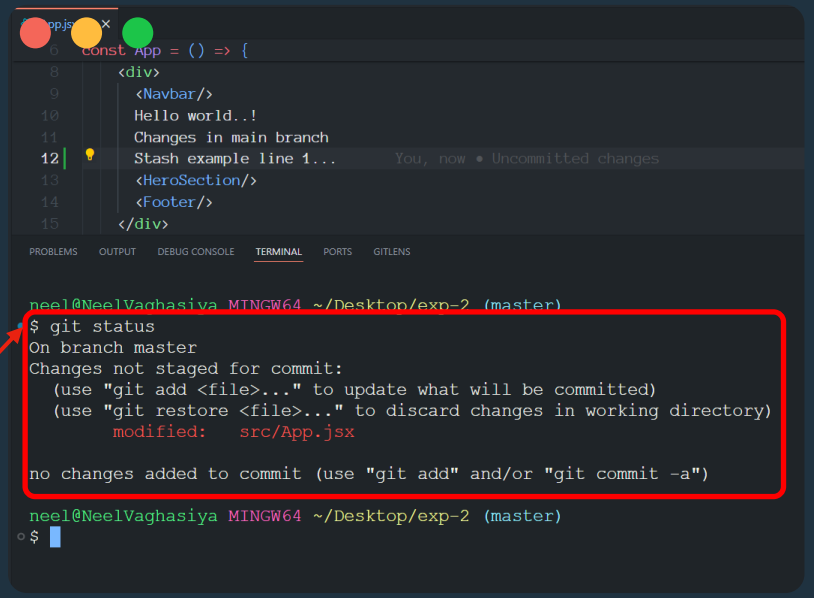
Why stashing?

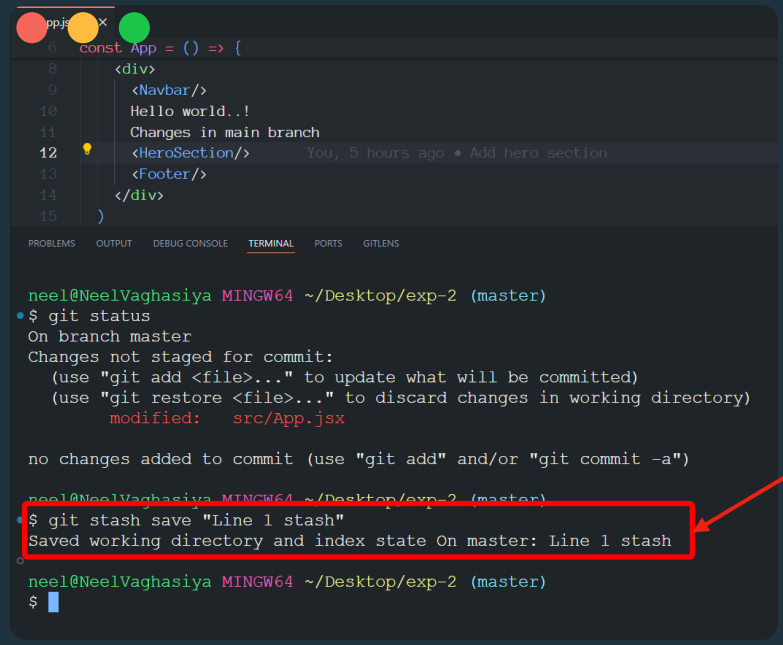
- Consider a developer working on a his feature branch.
- He needs to switch to a different branch to work on an urgent bug fix.
- To save the current work changes without committing, he can use git stash.
- After finishing the bug fix, the developer switches back to the feature branch and can reapply the stashed changes.

Key Benefits:

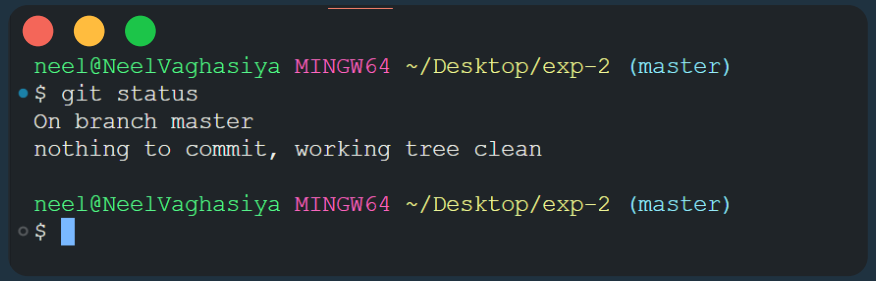
- No incomplete commits
- Quick context switch

- Let’s understand stashing by example:

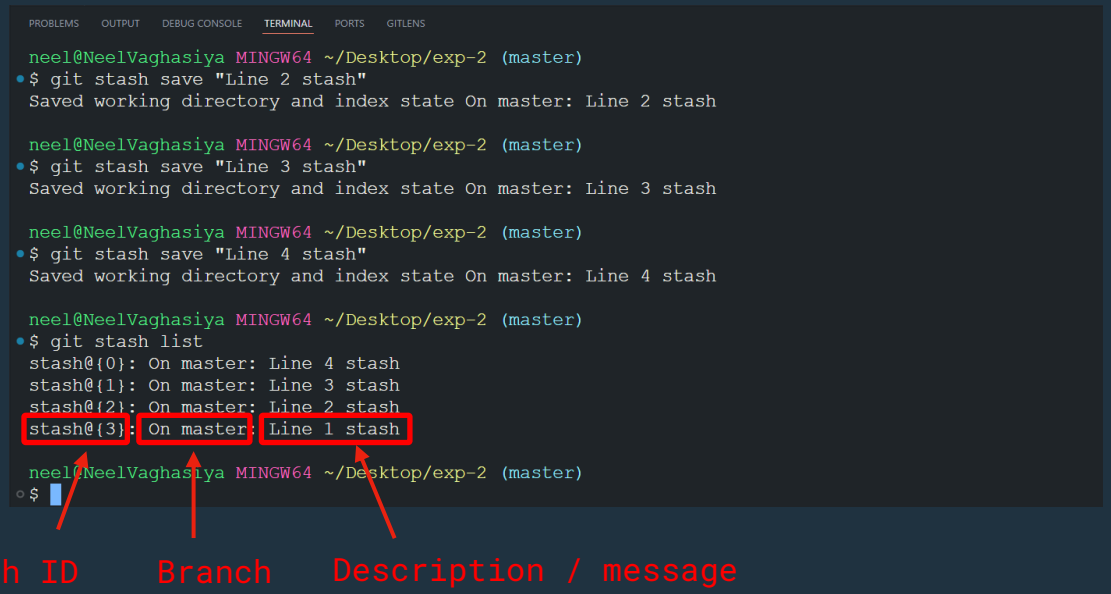




- If we check status again, we can observe, working directory has been reset



- Similarly, stash more changes by repeatedly modifying the file and running ‘git stash’.
- Then run ‘git stash list’ to see a list of stashed changes



10. git stash pop: reapplies the stash and removes it from the stash

- Syntax: **git stash pop**

  git stash pop <Stash ID>

- Let’s look at an example:

Stash pop output

```
const App = () => {
  return (
    <div>
      <Navbar/>
      Hello world..!
      Changes in main branch
      Stash example line 4...
      <HeroSection/>      You, 6 hours ago • Add hero section
      <Footer/>

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git stash pop
On branch master
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:   src/App.jsx

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (5b1884877fccb0f065cc99845bbe2cb2fa792991)

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

Stash pop <ID> output

```
const App = () => {
  return (
    <div>
      <Navbar/>
      Hello world..!
      Changes in main branch
      Stash example line 2...
      <HeroSection/>
      <Footer/>

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git stash pop stash@{2}
On branch master
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:   src/App.jsx

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{2} (596601aef057589aa193d8d394106f19afc97d61)

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

- In this case, commit or remove changes after the first pop, because Git will encounter conflicts if the working directory has changes that overlap with the next stash.
- Now let’s check the stash list:

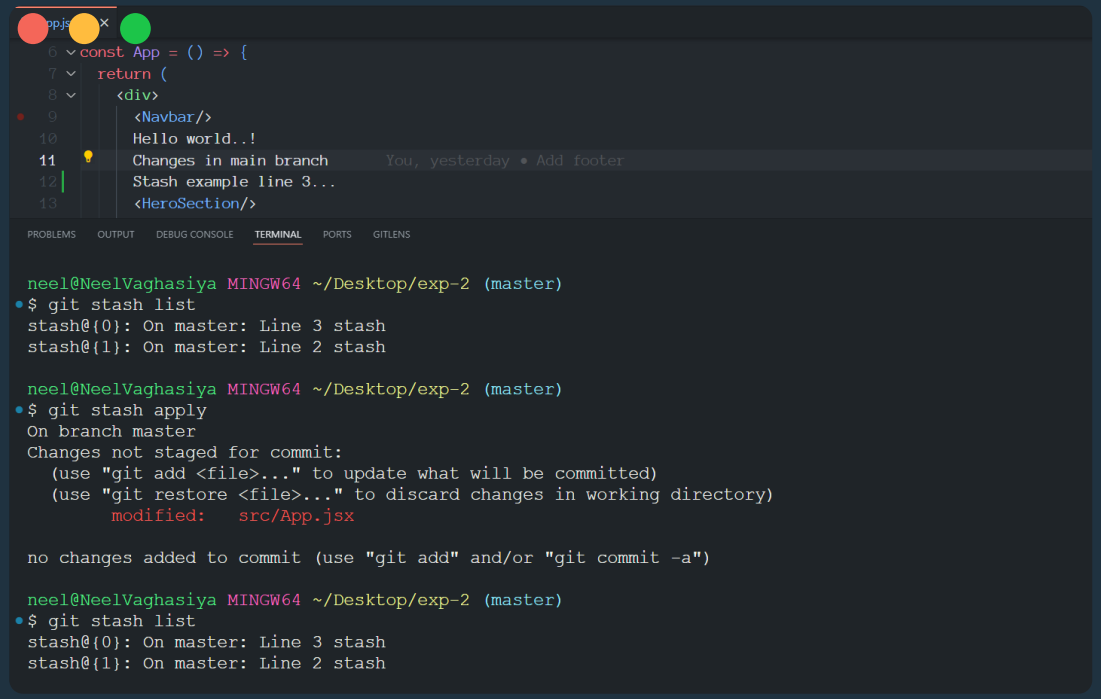
```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git stash list
stash@{0}: On master: Line 3 stash
stash@{1}: On master: Line 2 stash

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

11. git stash apply: reapplies the stash

- Syntax: **git stash apply**

  git stash apply <Stash ID>

- Example:



- Same as the ‘git stash pop’, but it will not remove stash from the list

12. `git stash drop`: deletes the stash

- Syntax: **`git stash drop`**

  `git stash drop <Stash ID>`

- Example:

13. git log: displays commit history

- Syntax: **git log**

  git log --oneline

  git shortlog

- Let's run each command:

```
l@ LVa siya MINGW64 ~/Desktop/exp-2 (master)
$ git log
commit a039c2e6322b2da63dc5f12f116ff720db01be83 (HEAD -> master, feature/nav-options, bugfix/hero-section)
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sun Jun 2 19:34:17 2024 +0530

    Add logo to navbar

commit 4fb32a41cb34cc0dea57238102e37faa829dccad (feature/hero-section)
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sun Jun 2 17:44:05 2024 +0530

    Add hero section

commit eb8b8e839d762a5d51bc6f2f5372c6997401222b
Merge: fd2bd37 6e1a72e
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sun Jun 2 11:39:46 2024 +0530

    Merge branch 'feature/navbar'

commit fd2bd37a163dd2da9de68739804a83411ff37b8b (feature/footer)
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sun Jun 2 00:19:37 2024 +0530

    Add footer

commit 6e1a72e216b9e337750b6a0258562515c226d8c3 (feature/navbar)
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sat Jun 1 23:46:57 2024 +0530

    Add navbar

commit 8af4ec43d933f1c1df49d607d6252d166ebe1136
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sat Jun 1 22:48:22 2024 +0530

    Modify master branch

commit e32334bccd3477f4b6f836c31181db878c38abbb
Author: Neel Vaghasiya <neelvaghasiya003@gmail.com>
Date:   Sat Jun 1 21:23:34 2024 +0530

    Initial commit
```

Run 'git log' for detailed log history

Run 'git log --online', For single line info. Contains commit ID & commit message

```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git log --oneline
a039c2e (HEAD -> master, feature/nav-options, bugfix/hero-section) Add logo to navbar
4fb32a4 (feature/hero-section) Add hero section
eb8b8e8 Merge branch 'feature/navbar'
fd2bd37 (feature/footer) Add footer
6e1a72e (feature/navbar) Add navbar
8af4ec4 Modify master branch
e32334b Initial commit

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

Run 'git shortlog', it groups each commit by author

```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git shortlog
Neel Vaghasiya (7):
      Initial commit
      Modify master branch
      Add navbar
      Add footer
      Merge branch 'feature/navbar'
      Add hero section
      Add logo to navbar


neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

14. `git blame`: displays the last author that committed a line in a file

- Syntax: **git blame <file>**
- Let's see in our App.jsx file:

```
neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$ git blame src/App.jsx
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530  1) import React from 'react'
fd2bd37a (Neel Vaghasiya 2024-06-02 00:19:37 +0530  2) import Footer from './components/Footer'
6e1a72e2 (Neel Vaghasiya 2024-06-01 23:46:57 +0530  3) import Navbar from './components/Navbar'
4fb32a41 (Neel Vaghasiya 2024-06-02 17:44:05 +0530  4) import HeroSection from './components/HeroSection'
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530  5)
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530  6) const App = () => {
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530  7)   return (
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530  8)     <div>
6e1a72e2 (Neel Vaghasiya 2024-06-01 23:46:57 +0530  9)       <Navbar/>
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 10)       Hello world..!
fd2bd37a (Neel Vaghasiya 2024-06-02 00:19:37 +0530 11)       Changes in main branch
4fb32a41 (Neel Vaghasiya 2024-06-02 17:44:05 +0530 12)       <HeroSection/>
fd2bd37a (Neel Vaghasiya 2024-06-02 00:19:37 +0530 13)       <Footer/>
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 14)     </div>
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 15)   )
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 16) }
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 17)
^e32334b (Neel Vaghasiya 2024-06-01 21:23:34 +0530 18) export default App

neel@NeelVaghasiya MINGW64 ~/Desktop/exp-2 (master)
$
```

USEFUL COMMANDS WHILE WORKING WITH REMOTE REPOSITORIES

- **Remote repositories**:
  - Git repositories hosted on the internet or another network.
  - Allows collaboration with others.
  - Key commands for managing remote repositories: remote, clone, pull, push etc.

1. git remote: for managing remote connections

- **git remote add <name> <url>** :
  - Create new connection to a remote repository
  - <name> is a convenient shortcut / alias for <url>.
  - Usage:

```
git remote add <name> <url>

git remote add origin http://dev.example.com/demo.git
```

- **`git remote remove <name>` :**
  - Remove an existing connection to a remote repository
  - Usage:

```
git remote remove <name>

git remote remove origin
```

- **`git remote -v` :**
  - Lists all the remotes
  - Usage:

```
git remote -v
```

2. git clone: to get a local copy of remote repository

- **git clone <url> :**
  - Usage:

```
git clone <url>

git clone http://dev.example.com/demo.git
```

3. git pull: for fetching and merging remote changes

- **git pull <name> <branch> :**
  - <name> : name of remote connection
  - <branch> : in which we want to merge fetched changes

```
git pull <name> <branch>

=

git fetch <name> + git merge <branch>
```

- Usage:

```
git pull <name> <branch>

git pull origin main
```

4. `git push`: uploads local repository changes to remote repository

- **`git push <name> <branch>`** :
  - Push changes to the `<branch>` of the remote repository
  - Usage:

```
git push <name> <branch>

git push origin main
```

WE WILL SEE ALL THESE COMMANDS IN THE NEXT EXPERIMENT

THANK YOU!

nee1-03