# **Section 1: Course Orientation**

# 1. Welcome to the Course

# 2. What the Course Covers

# 3. A Note On the Exercises

# 4. Accessing The Slides & Diagrams

# **Section 2: Introducing Git!**

# 5. What Really Matters In This Section

# 6. What Exactly is Git?

# 7. Visualizing Git

* Track Changes across multiple files
* Compare versions of project
* Time Travel back to old versions
* Revert to a previous version
* Collaborate and share changes

# 8. A Quick History of Git

# 9. Who uses Git?

# 10. Git vs Github: What’s the difference

**Git** is the Version Control software that runs locally on your machine. You don’t need to register for an account. You don’t need the internet to use it. You can use Git without ever touching Github.

**Github** is a service that host Git Repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It’s an online place to share work that is done using Git.

# **Section 3: Installation & Setup**

# 11. What Really Matters In This Section

# 12. Installing Git: Terminal Vs. GUIs

**Git is (primarily) A Terminal Tool:**

Tit was created a s command-line tool. To use it, we run various git commands in a Unix shell. This is not the most user friendly experience, but it’s at the very core of Git!

**The Rise of GUI’s:**

Over the last few years, companies have created graphical user interfaces for Git that allow people to use Tit without having to be a command-line expert.

**Popular Git GUI’s include:**

* Github Desktop
* Source Tree
* Tower
* GitKraken
* Ungit

# 13. Windows Git Installation

# 14. MAC Git Installation

# 15. Configuring Your Git Name & Email

> git config --global user.email “moin.educat@gmail.com”

> git config --global user.name “moin.educat”

# 16. Installing GitKraken (Our GUI)

# 17. Terminal Crash Course: Introduction

# 18. Terminal Crash Course: Navigation

> ***ls*** *// to list out the files and folders of current directory*

> ***ls*** ***-a*** *// to list out the files and folders of current directory including hidden files*

> ***ls pets***  *// to list out contents of pets directory (note: it’s relative path)*

> ***open .*** *// to open the folder window (start . for windows)*

> ***pwd***  *// print working directory (that means shows the exact path of working directory)*

> ***cd foldername/path***  *// cd command to navigate around*

> ***cd ..***  *// move one step back in folder tree*

> ***clear***

# 19. Terminal Crash Course: Creating File & Folders

> ***touch abc.txt*** *// to create a file called abc.txt*

> ***mkdir temp*** *// to create a folder with name temp*

# 20. Terminal Crash Course: Deleting Files & Folders

> ***rm abc.txt*** *// to delete abc.txt file*

> ***rm -rf temp*** *// to delete temp folder*

# **Section 4: The Very Basics of Git: Adding & Committing**

# 21. What Matters

# 22. What is a Git Repo

A Git “Repo” is a workspace which tracks and manages files within a folder.

Anytime we want to use Git with a project, app, etc we need to create a new git repository. We can have as many repos on our machine as needed, all with separate histories and contents

# 23. Our first commands: Git init and git status

> ***git init*** // to initialize the current folder as git folder (we can see .git folder have got created which is hidden)

> ***ls -a*** // to view the files and folders including hidden things

> ***git status***

# 24. The Mysterious .git Folder

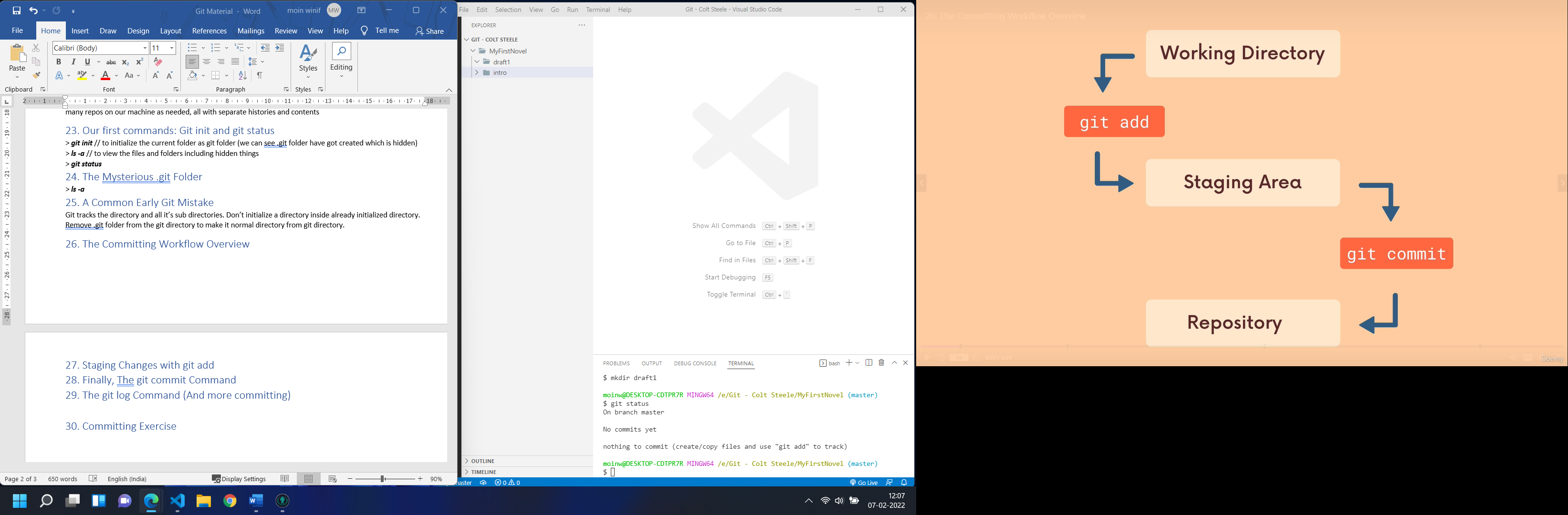
> ***ls -a***

# 25. A Common Early Git Mistake

Git tracks the directory and all it’s sub directories. Don’t initialize a directory inside already initialized directory.

Remove .git folder from the git directory to make it normal directory from git directory.

# 26. The Committing Workflow Overview



What is committing?

A snapshot (check point) of the project will be saved in git repo.

(creation of new files, modified files, and file deleting are tracked always)

# 27. Staging Changes with git add

> ***git add <filename>*** ***<filename>*** *// to add things to staging area*

> ***git add .***  *// to stage all*

***To Remove file from staging area:***

*If you don’t want staged copy of file, delete it with git rm. If you want a staged copy, commit it.*

> ***git rm --cached <filename>*** *// (to remove file from staging area)*

> ***git restore --staged <filename>*** *// (to unstage)*

***Transaction between Working Directory and Staging Area:***

*When a working directory file has got updated after staged, you may want to follow one of the two cases*

# *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)*

# 28. Finally, The git commit Command

> ***git commit -m “first commit”***  *// to commit the staged changes*

*Writing message with the help of default editor:*

> ***git commit*** *// don’t write -m flag, then default editor opens for message to be typed.*

# 29. The git log Command (And more committing)

> ***git log --oneline*** *// to view the commit history*

***(or)***

> ***git log*** *// to view the commit history*

# 30. Committing Exercise

# **Section 5: Commits In Detail(And Related Topics)**

# 31. What Matters

# 32. Navigating the Git Documentation

# 33. Keeping Your Commits Atomic

When possible, a commit should encompass a single feature, change, or fix. In other words, try to keep each commit focused on a single thing.

This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.

# 34. Commit Messages Present or Past Tense?

Present Tense imperative is recommended by official documentation

# 35. Escaping VIM & Configuring Git’s Default Editor

Running git commit will commit all staged changes. It also opens up a text editor and prompts you for a commit message.

This can be overwhelming when you’re starting out, so instead you can use -m flag with the command.

The -m flag allows us to pass an inline commit message, rather than launching a text editor.

To make VS Code the corresponding editor

***git config –global core.editor “code - -wait"***

# 36. A Closer Look At The Git Log Command

***> git log***

***> git log --oneline***

***> git log --pretty***

# 37. Committing with A GUI

# 38. Fixing Mistakes with Amend

Suppose you just mad a commit and then realized you forgot to include a file! Or, maybe you mad a type in the commit message that you want to correct.

Rathaer than making a brand new separate commit, you can “redo” the previous commit using the --amend flag

(only the recent commit, not further down can be amended)

Amending Commits:

> git commit -m “some commit”

> git add forgotten\_file

> git commit –amend

After a commit stage some other modification and >git commit --amend

# 39. Ignoring Files w/ .gitignore

Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files & folders to ignore:

.DS\_Store will ignore files named .DS\_Store

folderName/ will ignore an entire directory

\*.log will ignore any files with the .log extension

# **Section 6: Working with Branches**

# 40. What Matters

# 41. Introduction to Branches

# 42. The master branch (Or is it Main?)

**Master:**

Many people designate the master branch as their “source of truth” or the “official branch” for their codebase, but that is left to you to decide.

From Git’s perspective the master branch is just like any other branch. I does not have to hold the “master copy” of your project.

**Master or Main?**

In 2020, Github renamed the default branch from master to main. The default Git branch name is still master, thought the Git team is exploring a potential change. We will circle back to this shortly.

# 43. What on Earth is HEAD?

**HEAD:**

We’ll often come across the term HEAD in Git.

HEAD is simply a pointer that refers to the current “location” in your repository. It points to a particular branch reference.

So far, HEAD always points to the latest commit you made on the master branch, but soon we’ll see that we can move around and HEAD will change!

# 44. Viewing All branches With git branch

> ***git branch*** *// to view (list out) all branches available (like below)*

*$git branch*

*\* master*

*b1*

*b2*

# 45. Creating & Switching Branches

**Creating Branches:**

Use git branch <branch-name> to make a new branch based upon the current HEAD

This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

> ***git branch b1*** *// it creates new branch named b1*

**Switching Branches:**

> ***git switch b1*** *// it activates b1 branch*

*(or)*

> ***git checkout b1*** *// it activates b1 branch (checkout used to switch branch or to any commit (by providing hash) like >git checkout b30c125 ) (>git checkout -b b5 //to create and activate a branch at same time)*

# 46. More Practice with Branches

> git commit -a -m “blah blah” // to stage and commit

# 47. Another Option: Git Checkout Vs. Git Switch

**Another way of switching??**

Historically, we used git checkout <branch-name> to switch branches. This still works.

The checkout command does a million additional things, so the decision was mad to add a standalone switch command which is much simpler.

You will see older tutorials and docs using checkout rather than switch. Both work now.

> ***git checkout b1*** *// it activates b1 branch (checkout used to switch branch or to any commit (by providing hash) like >git checkout b30c125 ) (>git checkout -b b5 //to create and activate a branch at same time)*

**Creating & Switching:**

> ***git switch -c b10*** *// it activates b10 and switch to it as well*

# 48. Switching Branches with Unstaged Changes

**If try to switch to other branch with unstaged chanes, the following thing happens:**

error: Your local changes to the following files would be overwritten by checkout:

outlineee.txt

Please commit your changes or stash them before you switch branches.

Aborting

# 49. Deleting & Renaming Branches

> ***git branch -d b1*** *// it deletes b1 branch*

*(or)*

> ***git branch --delete b1*** *// it deletes b1 branch*

If you committed the changes to the branch and not merged yet, then you’ll see the following:

***$ git branch -d b1***

***error: The branch 'b1' is not fully merged.***

***If you are sure you want to delete it, run 'git branch -D b1'.***

**Rename:**

1) first switch to the branch you want to rename

2) > ***git branch -m newbranch***

# 50. How Git Stores HEAD & Branches

# **Section 7: Merging Branches**

# 52. What Matters

# 53. An Introduction to Merging

Merging Made Easy:

To merge, follow these basic steps:

1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)
2. Use the git merge command to merge changes from a specific branch into the current branch.

**> git switch master**

**> git merge bugfix**

# 54. Performing a Fast Forward merge

When the receiving and received branches have no conflicts, git performs a fast forward merge

**> git switch master**

**> git merge bugfix**

# 55. Visualizing Merges

# 56. Generating Merge Commits

# 57. Oh No! Merge Conflicts

**Master Branch**

***Index.html:***

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Heading</h1>

</body>

</html>

**Now create b1 branch:**

***> git branch b1***

Now master and b1 branches are identical

**Now modify index.html at both branches:**

**Master Branch**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Heading</h1>

    <p>a para from master</p>

</body>

</html>

**Now commit**

git commit -a -m "a para from master"

***> git switch b1***

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Heading</h1>

    <p>a para from b1</p>

</body>

</html>

***git commit -a -m "a para from b1"***

now switch to master and merge:

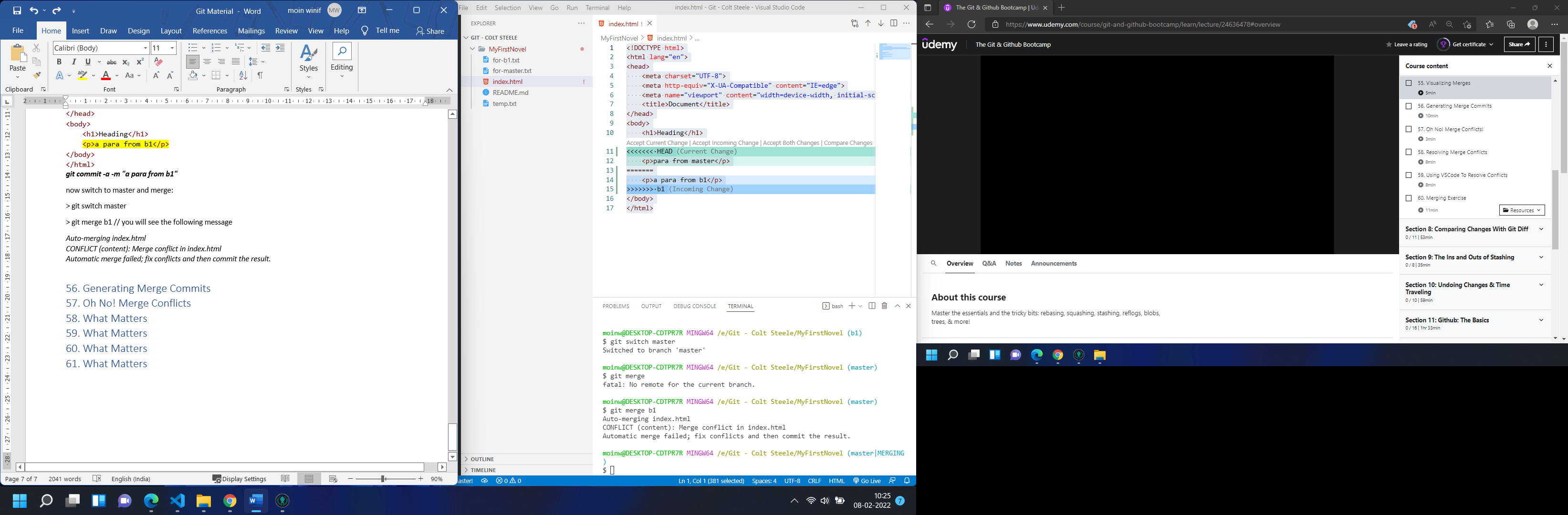
***> git switch master***

***> git merge b1*** // you will see the following message

*Auto-merging index.html*

*CONFLICT (content): Merge conflict in index.html*

*Automatic merge failed; fix conflicts and then commit the result.*



Now click Accept Both Changes from screen appeared like above (If the file not opened automatically, you have to open it manually)

**Now Save and commit, to finish the merging process:**

***> git commit -a -m "merger done"***

# 58. Resolving Merge Conflicts

1) Open up the file(s) with merge conflicts

2) Edit the file(s) to remove the conflicts. Declare which branch’s content you want to keep in each conflict. Or keep the content from both

3) Remove the conflict markers in the document

4) Ad your changes and make a commit

# 59. Using VS Code to Resolve Conflicts

# 60. Merge Exercise

# **Section 8:**