**Lesson 1**

**Git Introduction**

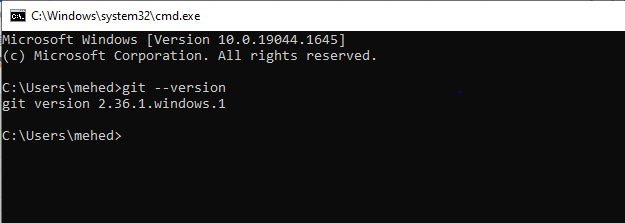
* Git is a version control system.
* Git use for
  + Tracking code changes
  + Tracking who made changes
  + Coding collaboration
* Git can
  + manage project with Repositories
  + Clone a project to work on a local copy
  + Control and track changes with **Staging** and **Committing**
  + **Branch** and **Merge** to allow for work on different parts and versions of a project.
  + **Pull** the latest version of the project to a local copy
  + **Push** local updates to the main project.
* Working with Git
  + Initialize Git on a folder, making it a Repository
  + Git now creates a hidden folder to keep track of changes in that folder.
  + When a file is changed, added or deleted, it is considered modified.
  + You select the modified files you want to **Stage**
  + The **Staged** files are **Committed,** which prompts Git to store a **permanent** snapshot of the files.
  + Git allows you to see the full history of every commit.
  + You can revert back to any previous commit.
  + Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!
* Why Git?
  + Over 70% of developers use Git!
  + Developers can work together from anywhere in the world.
  + Developers can see the full history of the project.
  + Developers can revert to earlier versions of a project.
* GitHub
  + Git is not the same as GitHub
  + GitHub makes tools that use Git.
  + GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.

**Git Install**

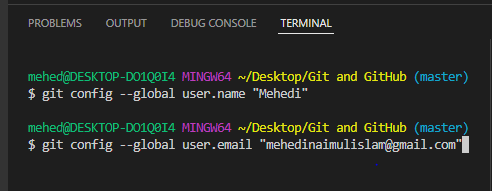
Download Git from Here (<https://git-scm.com/>)



**Using Git with Command Line**



**Configure Git**



Note: Use *global* to set the username and e-mail for every repository on your computer.

If you want to set the username/e-mail for just the current repo, you can remove *global.*

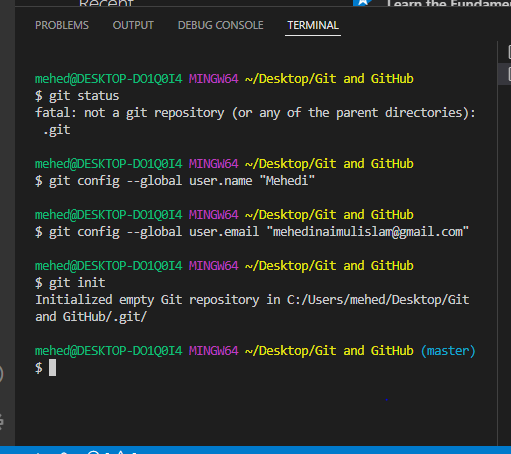
**Creating Git Folder**



* **mkdir** makes a new directory.
* **cd** changes the current working directory.

**Initialize Git**

Once you have navigated to the correct folder, you can initialize Git on that folder:

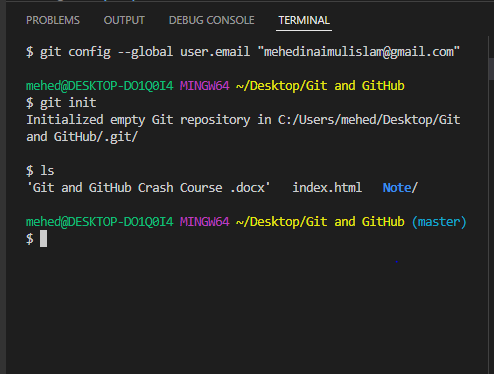


**Note:** Git now knows that it should watch the folder you initiated it on.

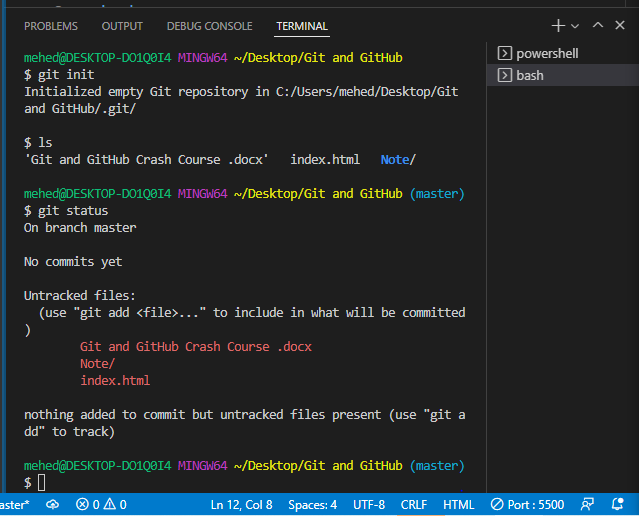
Git creates a hidden folder to keep track of changes.

**Git Adding New Files**

Create and save index.html file and let’s go back to the terminal and list the files in our current working directory:



Check git *status*



Files in your Git repository can be in one of 2 states:

* Tracked – files that GIt knows about, and are added to the repository
* Untracked – files that are in your working directory, but not added to the repository

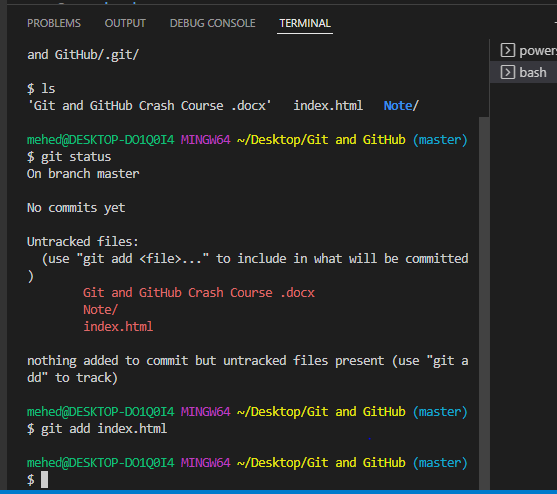
When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

**Git Staging Environment**

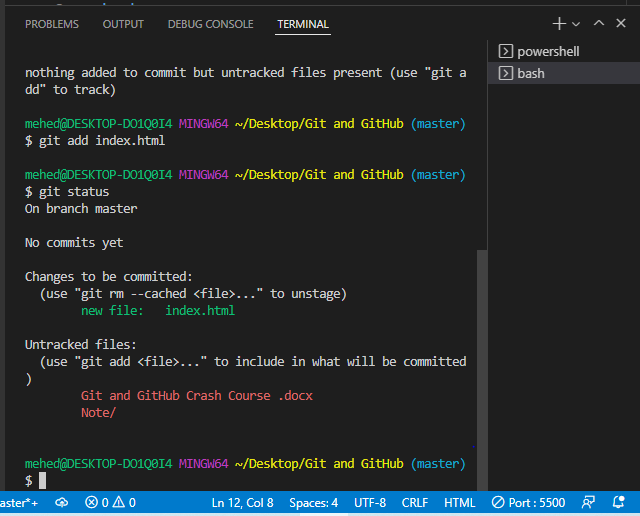
One of the core functions of Gits is the concepts fo the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

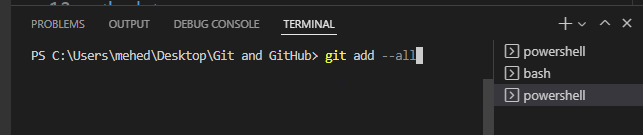
Staged files are files that are ready to be committed to the repository you are working on.



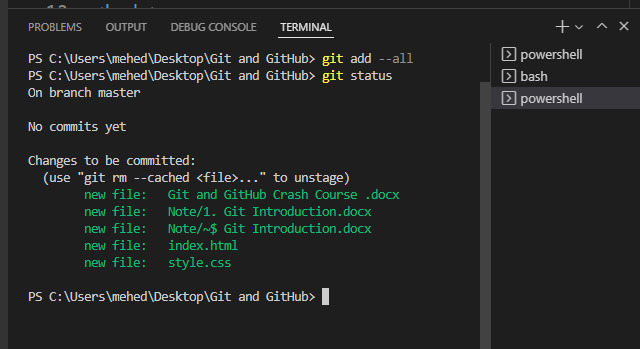
Lets’ check the status



**Git Add More than One File**



Using –all instead of individual filenames will *stage* all changes (new, modified, and deleted) files.



Note: The shorthand command for *git add --all* is *git add – A*

**Git Commit**

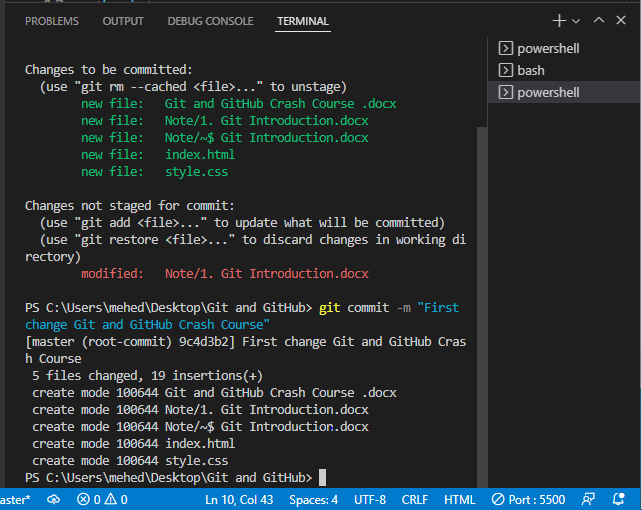
Since we have finished our work, we are ready move from *stage* to *commit* for our repo.

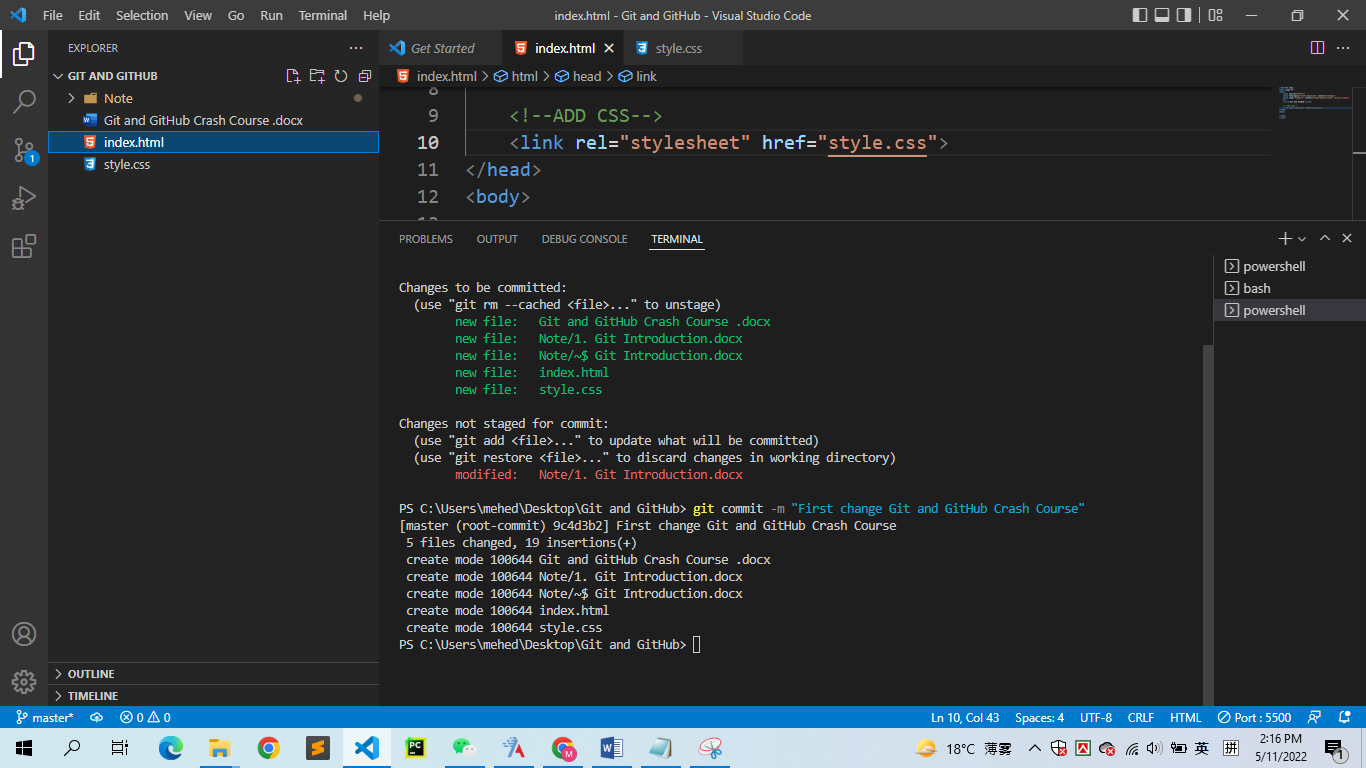
Adding commits keep track of our progress and changes as we work. Git considers each *commit* change point or “save point”. It is a point in the project you can go back to if you find a bug, or want to make a change.

When we *commit,* we should always include a message.

When we *commit,* we should always include a message.

By adding clear messages to each *commit,* it is easy for yourself (and others) to see what has changed and when.



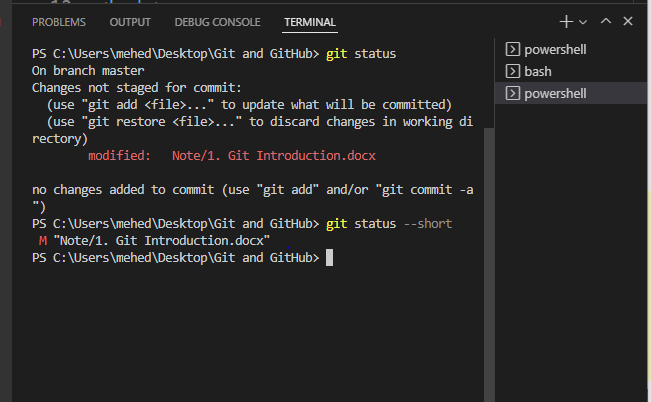


The *commit* command performs a commit, and the –m “message” adds a message

**Git Commit without Stage**

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The –a option will automatically stage every changed, already tracked file.

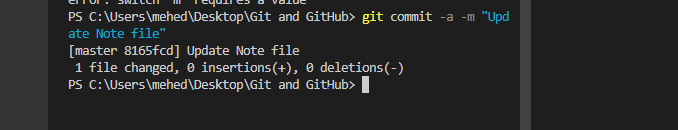
We, will use the --short option to see the changes in a more compact way:



**Note:** Short status flags are:

* ?? – Untracked files
* A – Files added to stage
* M – Modified files
* D – Deleted files

We see the file we expected is modified. So let’s commit it directly:

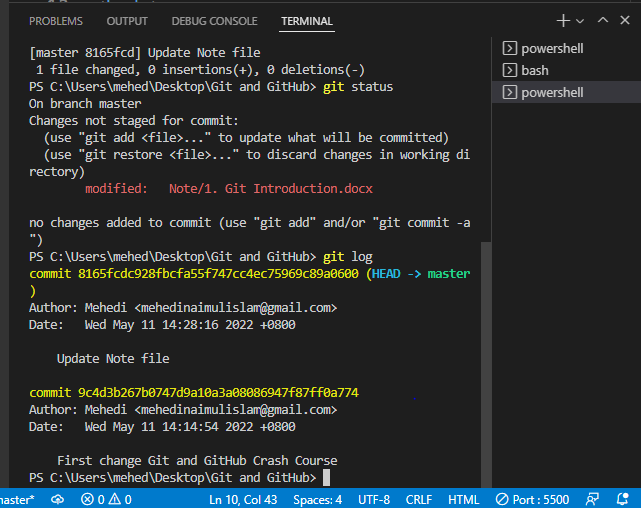


Warning: Skipping the Staging Environment is not generally recommended.

Skipping the stage step can sometimes make you include unwanted changes.

**Git Commit Log**

To view the history of commits for a repository, you can use the *log* command:

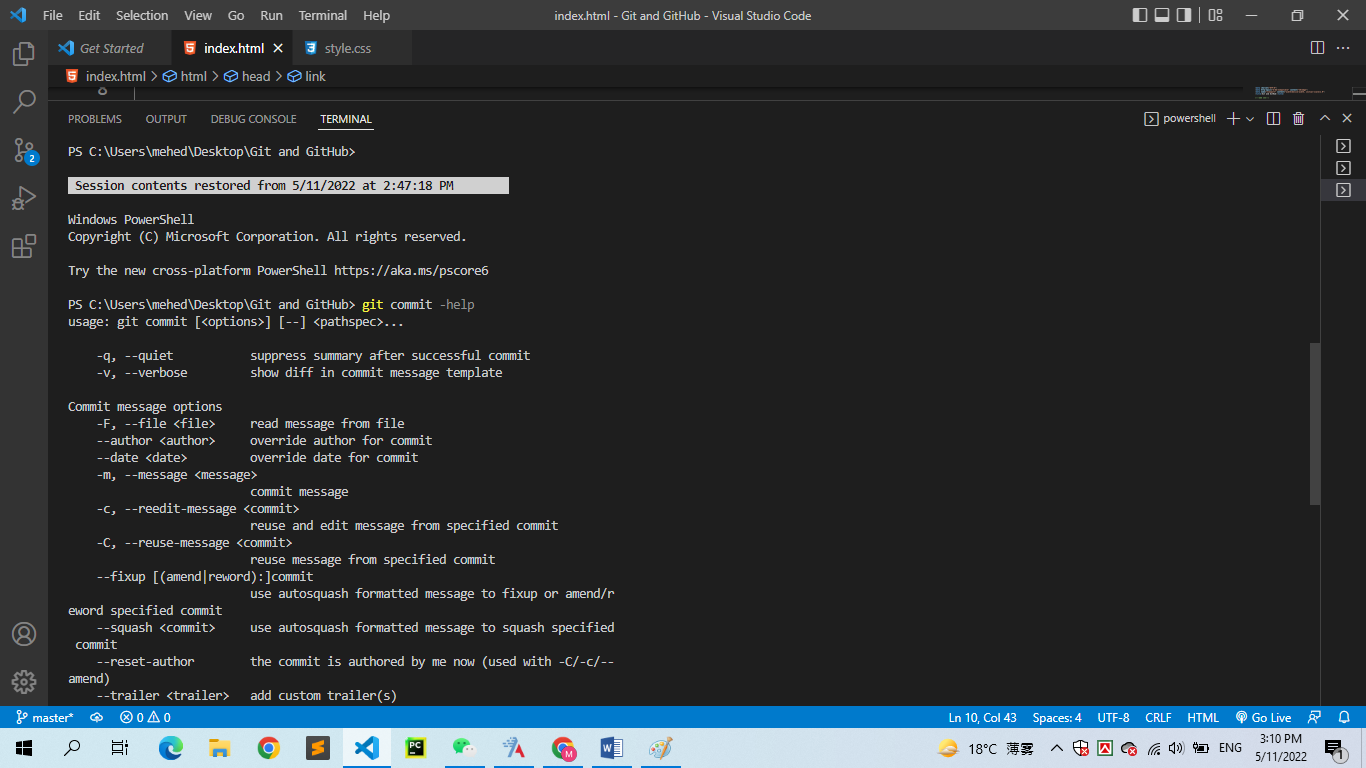


**Git Help**

If you are having trouble remembering commands or options for commands, you can use Git *help.*

There are a couple of different ways you can use the *help* command in command line:

* *git command –help* – See all the available options for the specific command.
* *Git help --all* – See all possible commands

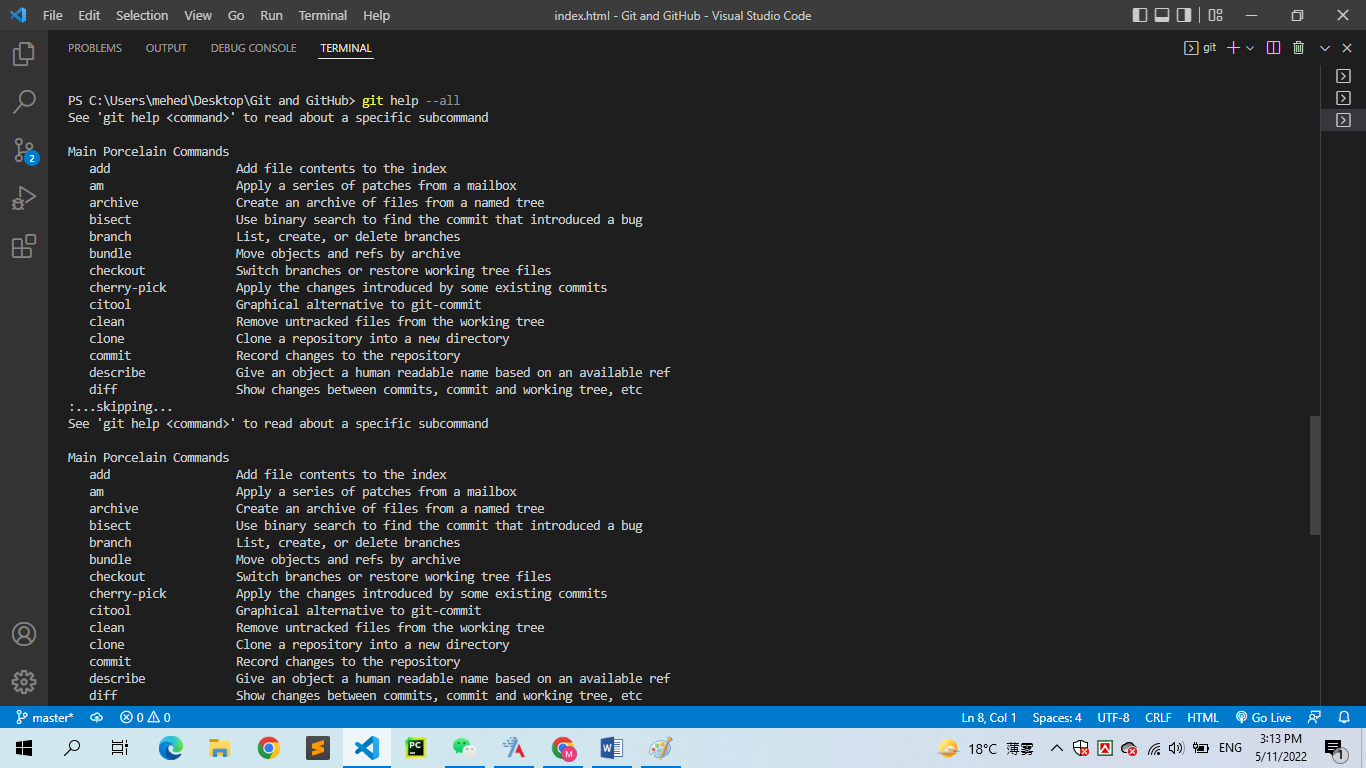


You can also use –help instead of –help to open the relevant Git manual page.

**Git help –all Sell All Possible Commands**

To list all possible commands, use the *help –all* command:

Warning: This will display a very long list of commands



Note: If you find yourself stuck in the list view, SHIFT + G to jump the end of the list then **q** to exit the view.

**Working with Git Branches**

In Git, a *branch* is a new/separate version of the main repository.

Let’s say you have a large project, and you need to update the design on it.

How would that work without and with Git:

* Make copies of all the relevant files to avoid impacting the live version.
* Start working with the design and find that code depend on code in other files, that also need to be changed!
* Make copies of the dependant files as well. Making sure that every file dependency references the correct file name.
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Save all your files, making a note of the names of the copies you were working on.
* Work on the unrelated error and update the code to fix it.
* Go back to the design, and finish the work there
* Copy the code or rename the files, so the updated design is on the live version.

With Git:

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERVENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix main branch
* You go back to the new-design branch, and finish the work there.
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

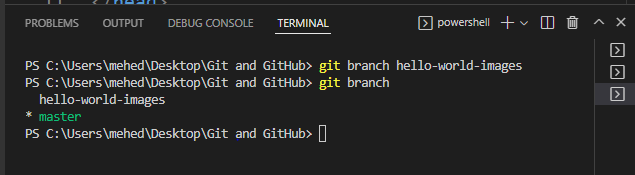
Branching in Git is very lightweight and fast!

**New Git Branch**

Let add some new features to our index.html page.

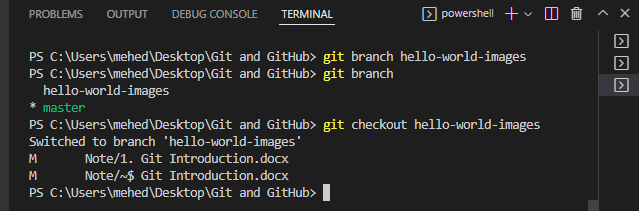
We are working in our local repository, and we do not want to disturb or possible wreck the main project.

So we create a new *branch:*



We can see the new branch with the name “hello-world-images”, but the \* beside *master* specifies that we are currently on that *branch.*

Checkout is the command used to check out a branch. Moving us *from* the current branch, to the one specified at the end of the command.



Now we have moved our current workspace from the master branch, to the new *branch*

Now check the status of the current branch:

