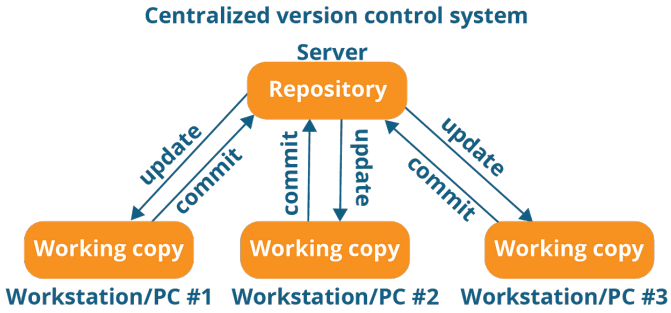
Introduction-Version Control

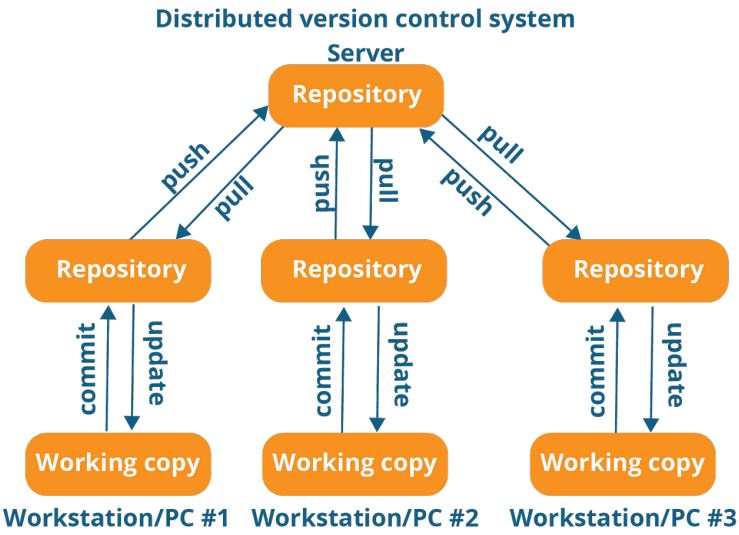
Version Control refers to the management of multiple versions of the project. When we say, managing, we mean keeping track of every modification (addition, edition, or deletion) made to project files such as programming code and documentation. Thus, it helps you navigate between changes, allowing us to compare previous versions of the code to help fix mistakes or quickly revert to a previous correct version, causing zero or minimal disruption to the flow of the project.

The key advantage of VC is enabling effective teamwork, as tracking and managing changes made by contributors can be tedious. With VC, multiple people can work on their own copy of the project (called branches) and only merge those changes to the main project when they are satisfied with the work. This enables tracking of individual changes and prevents conflicts of concurrent work.
There are primary three types of VC systems.

- **Local VC System:** This involves keeping track of modifications to project files in a single database stored locally. Thus, all changes are kept on a single computer, making it hard to collaborate and putting data at risk of loss.
- **Centralized VC System:** This system keeps track of changes to the project on a single remote server that the author can connect to. However, network connection or server errors can cause major bottlenecks. Once a file is being used by someone, it is locked, and other team members cannot work on it. Team members have a separate working copy(selective file pulled, not full copy), which means they must coordinate with each other simply to modify a single file. All commits are done to the central copy and it is visible to other co-workers when they update their copy.



- Distributed VCS: Each user/team has a copy or clone of a remote repository along with change history on its local machine. The committed changes are inside the local repository until it is pushed to the main central repository. Git, Mercurial, and Bazaar are the 3 most popular DVCS in the market.



Git is a powerful version control system that you can use to manage your code. Git allows you to track changes over time, revert changes, and collaborate with other developers on the same files.

What does Git do?

- Manage projects 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.

GitHub is a web-based platform used for version control with the help of Git.

- GIT focuses on version control and code sharing. GitHub focuses on hosting the centralized source code.
- GIT is primarily a command-line tool where GITHUB is administered through the web.
- GIT has no features for user management. GitHub contains built-in user management.

Popular terms in Version Control systems.

Repository: A centralized storage with the revision history of all related and specified files.

Workspace: Refers to the copies of the files in the local machine of the user. Also known as working copy.

Branches: Known as the lines of development. There are 5 main types of branches.

- Main Branch
- Feature Branch
- Release Branch
- Hotfix Branch
- Develop Branch

Tag: Represent a particular snapshot of a project at a given time.

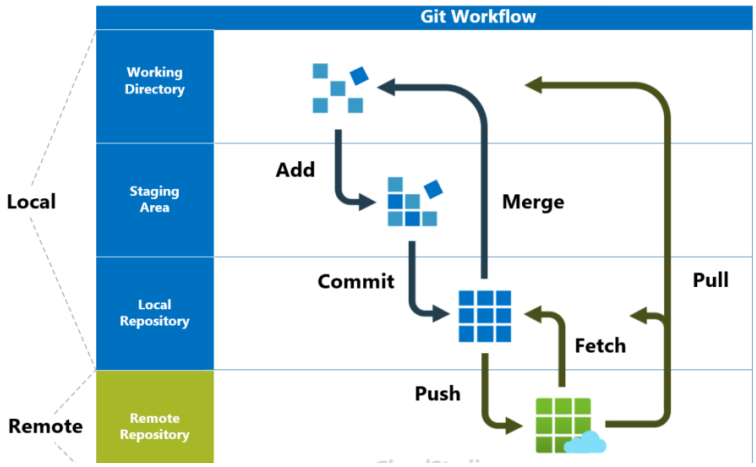
Pull / Update: Update the local working copy with the latest changes through the main branch or another local space.

Commit/Check-In: Store a change in the central Version Control storage.

Push: Used to send commits to the determined central repository.

Conflict: A situation where 2 developers try to commit changes in the same region of the same file.

Merge: Combining changes in different working copies to the same file in the repository.



Installation and Setup

Working with Git-Local Repository VC

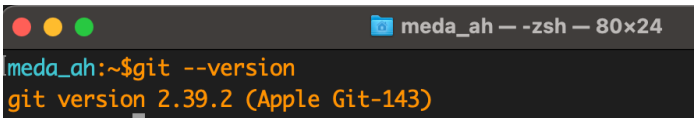
Three stages to creating a version history.

Before creating a version, we must Initialize Git on a folder, making it a Repository. Thus, Git now creates a hidden folder to keep track of changes in that folder.

1. **Modified**: Modify (add, edit, or delete) files on the working directory
2. **Staged:** Select and add the modified file/s to move to the next commit snapshot.
3. **Commit:** Once staged files are Committed, it prompts Git to store a permanent snapshot of the files.

Getting Started:

1. **Check if Git is properly installed.** If Git is installed, it shows something like git version 2.xx



2. **Configure Git**: if you haven't configured your username and email, use the following

```
meda_ah:~$git config --global user.name "Your username"
meda_ah:~$git config --global user.email "your email"
```

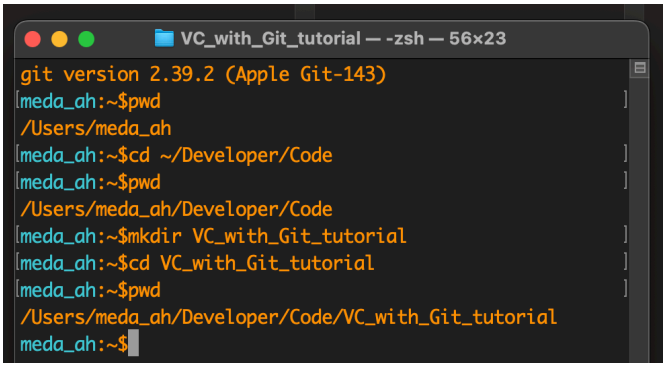
However, if you configured it before and want to check/confirm it.

```
meda_ah:~$git config --list
credential.helper=osxkeychain
init.defaultbranch=main
user.name=A-Meda23
user.email=a_meda@outlook.com
```

If you want to exlude all .DS_Store files from your future repositories:

```
echo .DS_Store >> ~/.gitignore_global
git config --global core.excludesfile ~/.gitignore_global
```

3. **Creating Git folder/Working directory:** Locate your pwd to where you want to create the git folder and create a folder.



4. **Initialize Git**: Once you have navigated to the correct folder, you can initialize Git on that folder: initializing an empty Git repository.

```
meda_ah:~$git init
Initialized empty Git repository in /Users/meda_ah/Developer/Code/VC_with_Git_tutorial/.git/
meda_ah:~$
```

Now, the hidden Git folder keeps track of changes to the files inside the working directory- VC_with_Git_tutorial.

5. **Add files to the Git folder:** We just created the first local Git Repo, but it's empty. Add project files you want to track or create one. Let's create an HTML file with VS code(you can use your favorite text editor).
   - After the files are saved go back to the terminal and **list out the file in the working directory**.

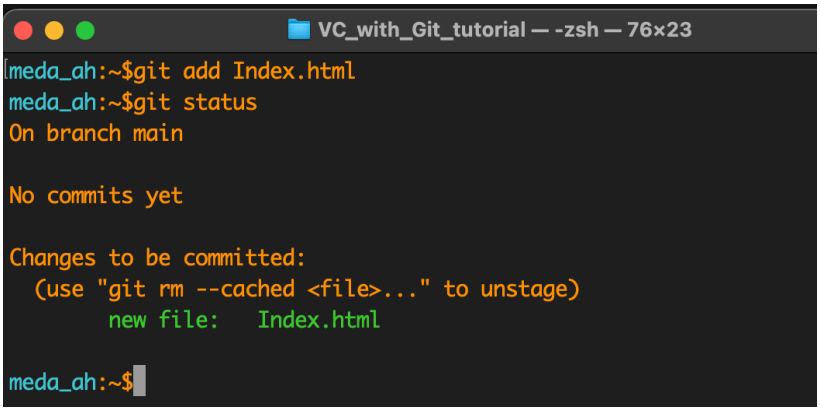
```
meda_ah:~$ls
Index.html
```

   - Then we **check the Git status** and see if it is a part of our repo:

```
meda_ah:~$git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
meda_ah:~$
```

   Now Git is aware of the file but has not added it to our repository! Files in your Git repository folder 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.

6. **Staging:** 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.

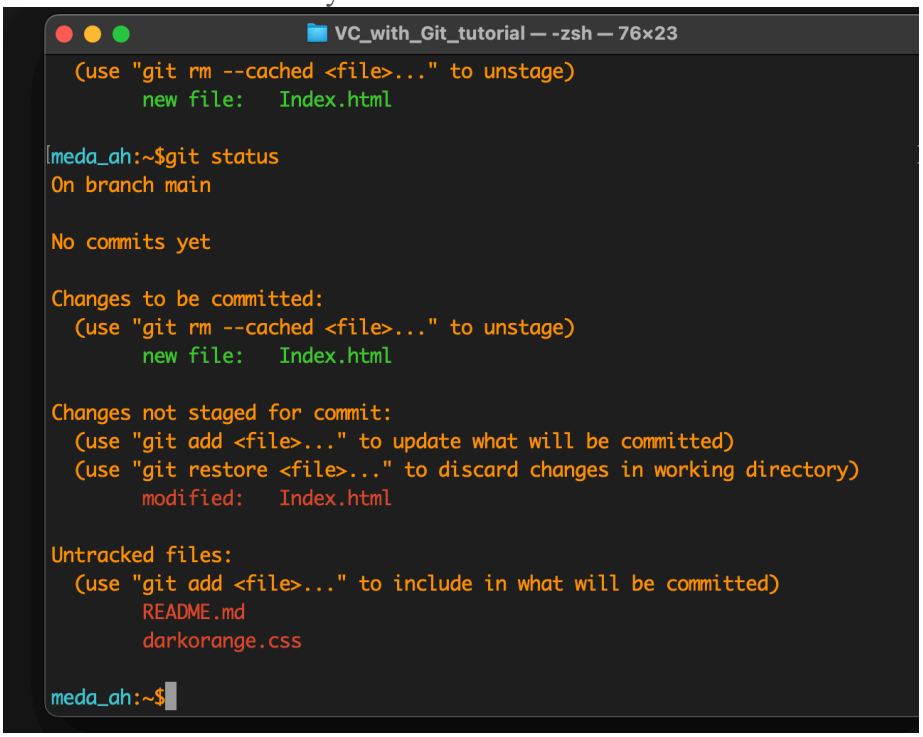


Git status indicates new the file has been added to the staging environment and is ready for commit.

`git add file`	Pick individual file
`git add folder/`	Pick all files inside a folder (and subfolders)
`git add .`	Pick all files (in folder command line is running in)

Let us stage more files for convenience and follow the same procedure. I created README.md, a file that describes the repository (recommended for all repositories), a basic CSS file, and updated the Index.html file.

Check Gilt status: Identify tracked and untracked files



Here, you can notice we have, two untracked new files and a modified tracked file yet to be staged. Next, stage all files. Use either of `git add --all` is `git add -A`, or `git add .`

```
meda_ah:~$git add --all
meda_ah:~$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:   README.md
        new file:   darkorange.css

meda_ah:~$
```

7. **Commit:** 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. The commit command performs a commit, and the -m "message" adds a message.

```
meda_ah:~$git commit -m "First_Version of VC_with_Git_tutorial"
[main (root-commit) 071fdd4] First_Version of VC_with_Git_tutorial
 3 files changed, 20 insertions(+)
 create mode 100644 Index.html
 create mode 100644 README.md
 create mode 100644 darkorange.css
meda_ah:~$
```

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(not recommended though as you might include unwanted change". The -a option will automatically stage every changed, already tracked file.
Let's add a small update to index.html: add a new line in the file. And then use status –short to see changes in a more compact way. **Note: Short status flags are:**

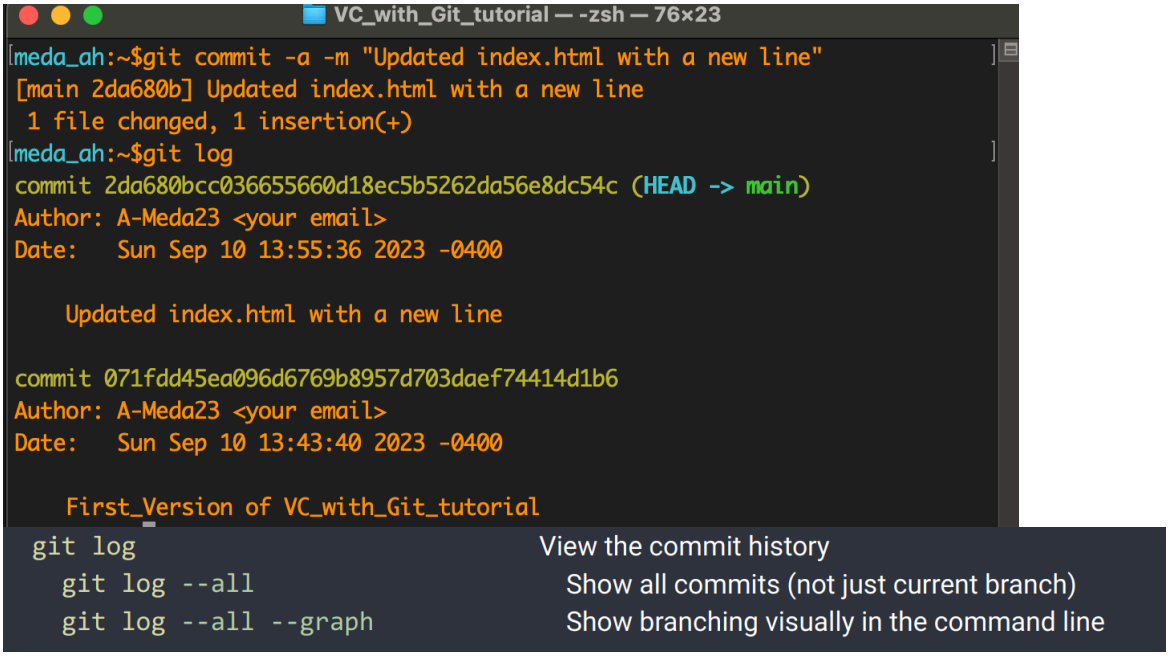
- ?? - Untracked files
- A - Files added to stage
- M - Modified files
- D - Deleted files

```
meda_ah:~$git status --short
 M Index.html
meda_ah:~$
```

Next commit:

```
VC_with_Git_tutorial — -zsh — 76×23
meda_ah:~$git commit -a -m "Updated index.html with a new line"
[main 2da680b] Updated index.html with a new line
 1 file changed, 1 insertion(+)
meda_ah:~$
```

8. **Gilt commit log:** use the log command to view the history of your commit.



Note: Use git help If you are having trouble remembering commands or options for commands. Such as git help –all, git commit -help, etc

Working with GitHub

GitHub is a powerful platform for version control, collaboration, and project management. This detailed guide will walk you through the essential steps to work effectively with GitHub.

Push local repo to GitHub:

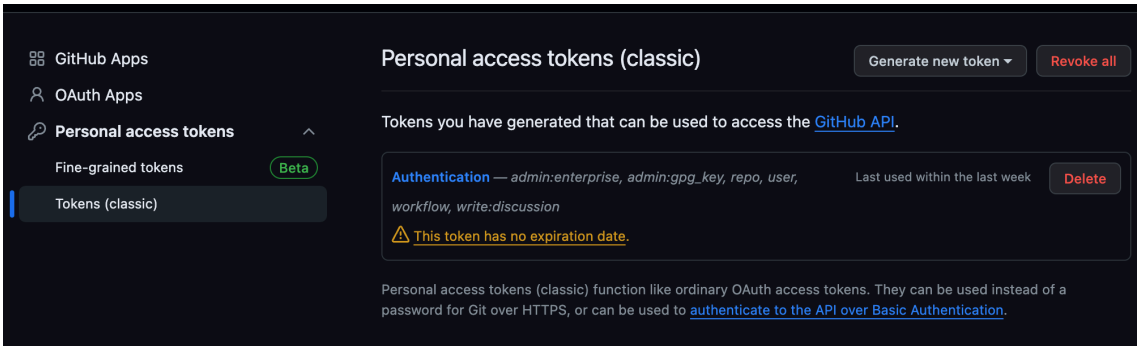
Since we have already set up a local Git repo, the next step is to push it to the GitHub repository.

1. Create a new repository on GitHub that we push local files to: On GitHub, click the '+' button on the top right and select 'New repository'. Give your repository a name, add a description (optional), choose to make the repository public or private, and click 'Create repository'.
2. Configure your GitHub username so you can get access to your GitHub repository. $git config --global credential.username <username>
3. Upload local repos content to a remote repository:

   git push <remote_name> <branch>

   i.e git push origin main

you may encounter a fatal error message if you haven't authenticated permission to push to your GitHub account. However, if run the step 2 command, you will be prompted to enter your authentication token, only once. To get the token–go to settings on GitHub user →Developer settings → Personal access tokens →Generate new token– set the expiration time to no expiration → pick repo(must) the rest as needed →Generate. Use that token for one-time authentication.



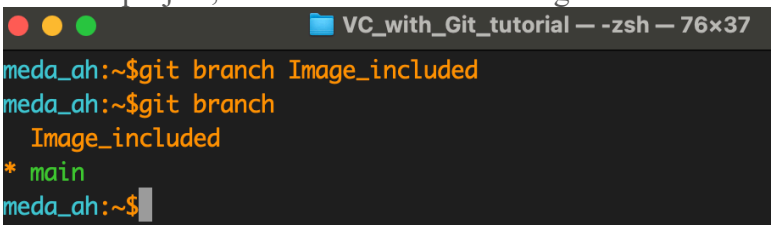
4.

Contribution management with Branching and Merging

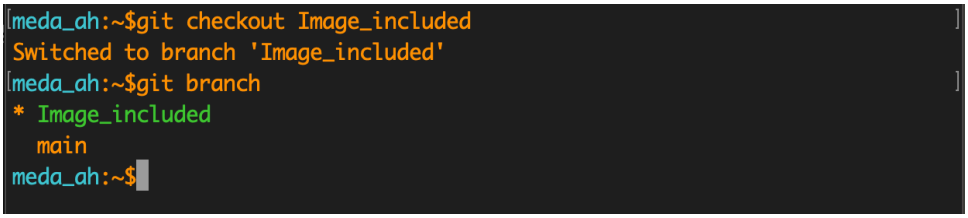
Git Branch:

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

5. Create a branch: Let's add some features to our index.html file. Since we don't want to disturb the main project, we create a branch. Let's give it a name 'image included."



Now we have created the branch, but we are not working on it. The * indicates that. We use the checkout command to move the branch.



Now we have moved our workspace from the main to the new branch. Let's make a change to the index.html file. Let us add an image to it. So, add an image(git_workflow.png) to the folder and add the code to the index.html file.

Let us check the status of the new branch

```
meda_ah:~$git status
On branch Image_included
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Index.html

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

no changes added to commit (use "git add" and/or "git commit -a")
meda_ah:~$
```

So let's go through what happens here:

- There are changes to our index.html, but the file is not staged for commit.
- img_hello_world.jpg is not tracked.

So we need to add both files to the Staging Environment for this branch:

```
meda_ah:~$git add --all
meda_ah:~$git status
On branch Image_included
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Index.html
        new file:   git_workflow.png

meda_ah:~$
```

We are happy with our changes. So we will commit them to the branch:

6. **Switching between Branches**: Now let's see just how quick and easy it is to work with different branches, and how well it works. We are currently on the branch Image_included. We added an image to this branch, so let's list the files in the current directory:

```
meda_ah:~$ls
Index.html              darkorange.css
README.md               git_workflow.png
meda_ah:~$
```

We can see the new file git_workflow.png is included. Now, let's see what happens when we change branch to main.

```
meda_ah:~$git checkout main
Switched to branch 'main'
meda_ah:~$ls
Index.html      README.md       darkorange.css
meda_ah:~$
```

The new image is not part of this branch. if we open the html file, we can see the code reverted to what it was before the alteration. See how easy it is to work with branches? And how this allow you to work on different things?

7. Emergency branch: Now imagine we are not done yet with the Image_included branch, but we need to fix the error on master. To not mess up both files, we created a new branch to deal with the emergency.

```
meda_ah:~$git checkout -b emergency-fix
Switched to a new branch 'emergency-fix'
meda_ah:~$
```

Now we have created a new branch from master, and changed to it. We can safely fix the error without disturbing the other branches. Let's fix our imaginary error:

```
meda_ah:~$git add Index.html
meda_ah:~$git commit -m "updated Index.html with emergency fix"
[emergency-fix fffe226] updated Index.html with emergency fix
 1 file changed, 1 insertion(+), 1 deletion(-)
meda_ah:~$
```

Merge Branches:

We have the emergency fix ready, and so let's merge the master and emergency-fix branches.

1. First, we need to change to the master branch:

```
meda_ah:~$git checkout main
Switched to branch 'main'
meda_ah:~$
```

2. Now we merge the current branch (main) with emergency-fix:

```
meda_ah:~$git merge emergency-fix
Updating 2da680b..fffe226
Fast-forward
 Index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
meda_ah:~$
```

Since the emergency-fix branch came directly from master, and no other changes had been made to master while we were working, Git sees this as a continuation of master. So it can "Fast-forward", just pointing both master and emergency-fix to the same commit.

As master and emergency-fix are essentially the same now, we can delete emergency-fix, as it is no longer needed:

```
meda_ah:~$git branch -d emergency-fix
Deleted branch emergency-fix (was fffe226).
meda_ah:~$
```

Merge Conflict:

1. let us move back ot Image_included branch and keep working. Let add another image (git_workflow2.jpeg) and modify index.html accordingly. Then after we stage and commit to the branch.

```
meda_ah:~$git checkout Image_included
Switched to branch 'Image_included'
meda_ah:~$git add --all
meda_ah:~$git commit -m "added new image"
[Image_included 7436a54] added new image
 2 files changed, 1 insertion(+)
 create mode 100644 git_workflow2.jpeg
meda_ah:~$
```

2.
3. We see that index.html has been changed in both branches. Now we are ready to merge hello-world-images into master. But what will happen to the changes we recently made in master?

```
meda_ah:~$git checkout main
Switched to branch 'main'
meda_ah:~$git merge Image_included
Auto-merging Index.html
CONFLICT (content): Merge conflict in Index.html
Automatic merge failed; fix conflicts and then commit the result.
meda_ah:~$
```

4. The merge failed, as there is conflict between the versions for index.html. Let us check the status:

```
meda_ah:~$git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   git_workflow.png
        new file:   git_workflow2.jpeg

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   Index.html

meda_ah:~$
```

5. This confirms there is a conflict in index.html, but the image files are ready and staged to be committed. So we need to fix that conflict. Open the file in our editor snd then stage the file.

```
meda_ah:~$git add Index.html
meda_ah:~$git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   Index.html
        new file:   git_workflow.png
        new file:   git_workflow2.jpeg

meda_ah:~$
```

6. The conflict has been fixed, and we can use commit to conclude the merge.

```
meda_ah:~$git commit -m "merged with image_included after conflit fix"
[main 6fb6f03] merged with image_included after conflit fix
```

7. Now we can delate the Image included branch, since it is merged,

```
meda_ah:~$git branch -d Image_included
Deleted branch Image_included (was 7436a54).
```

8.
```
meda_ah:~$
```