Web Development

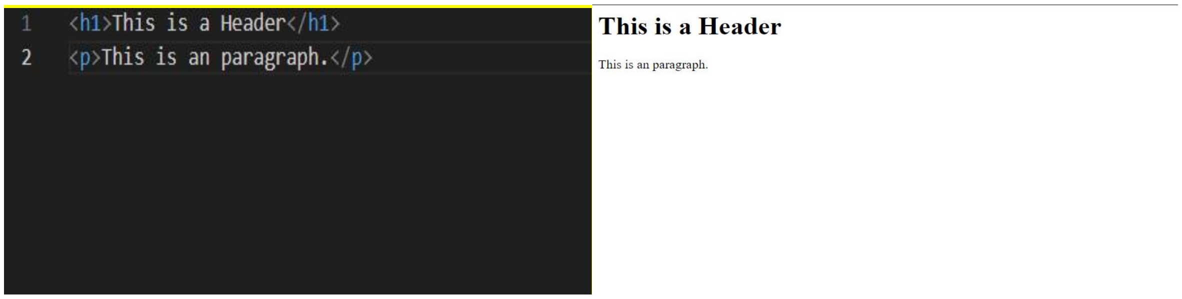
Front-End	**Back-End**
Front-end is everything that users interact with directly on a website or web application. Includes UI/UX	Backend development deals with the server-side of web applications
Basically , **HTML**, **CSS** and **JAVASCRIPT** are the languages that are used to create an interactive front-end of an web application.	Backend developers work on the server, database, and application logic

We will discuss about Front-end part only.

1. HTML (Hypertext Markup Language):

HTML is used to create and design web pages. It provides the structure and content of a web page by defining elements and their attributes. i.g. <h1></h>, <p></p> this are the elements of HTML.

Let's see an example of how HTML works:



```html
<h1>This is a Header</h1>
<p>This is an paragraph.</p>
```

```
This is a Header

This is an paragraph.
```

This example of html defining a header and a line of paragraph in the web page.

We, can see HTML provides the basic structure of a page. Here's a list of some common things you can do with HTML on a web page:

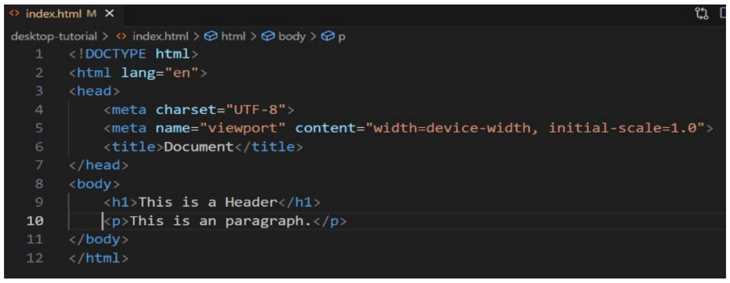
- Text Contents.
- Images and Media.
- Forms and Inputs.
- Tables

- Semantic Elements.
- Structural Elements.
- Links
- Embedded Contents etc.

Next would be design part of the page by CSS.

But before diving into CSS let's put previous HTML code in a correct syntax.

Here is the demonstration:



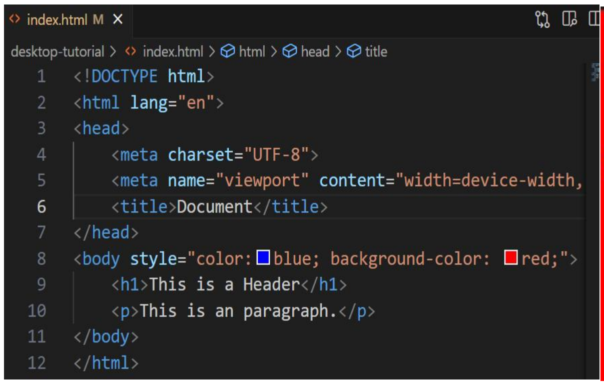
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>This is a Header</h1>
    <p>This is an paragraph.</p>
</body>
</html>
```

For, now let's avoid everything except the head and body tag.

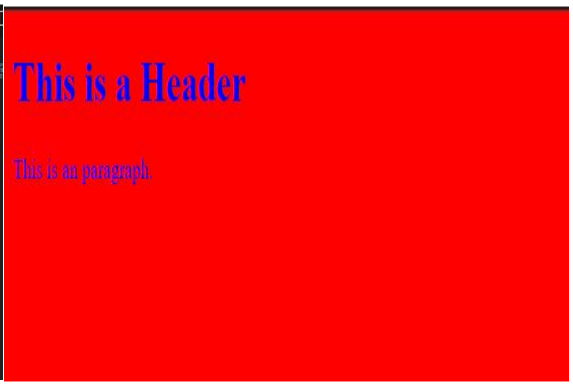
Basically, contents that are not going to show in page are in head tag and all the page contents are in the body tag.

2. CSS (Cascading Style Sheet):

Now let's see how to style this page. This part will be done with CSS. let's see how.



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
</head>
<body style="color: blue; background-color: red;">
    <h1>This is a Header</h1>
    <p>This is an paragraph.</p>
</body>
</html>
```



Although, CSS can be implemented in 3 different ways. In above example we used the CSS style attribute method to design the texts to be blue and background of the body red.

Thus, defining of structure and designing is complete for the page.

3. JAVASCRIPT:

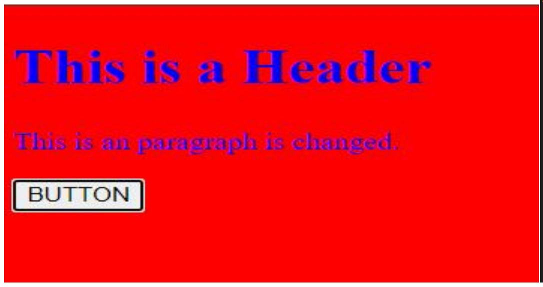
The role of javascript language is to add interactivity, dynamic behavior, and functionality to web pages.In short, javascript to program the behavior of web pages.

We can implement javascript in two ways:

1. Developers can include JavaScript code directly within the HTML document using <script> tags

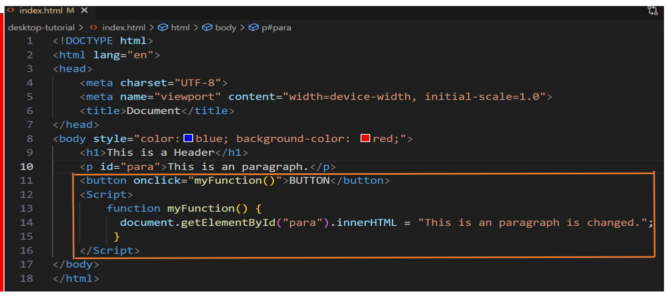
2. Or, they can just link to external JavaScript files.

Let's see an example of implementation in way 1:



This is a Header

This is an paragraph is changed.



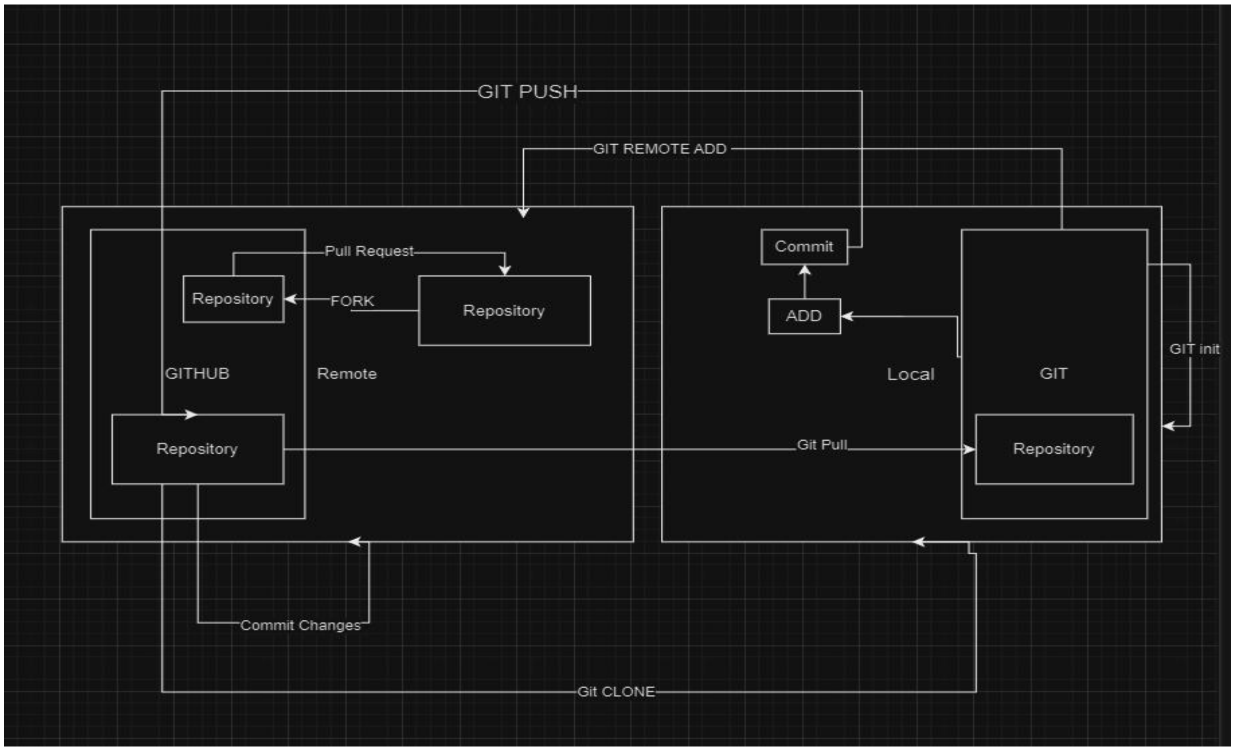
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body style="color: blue; background-color:  red;">
    <h1>This is a Header</h1>
    <p id="para">This is an paragraph.</p>
    <button onclick="myFunction()">BUTTON</button>
    <Script>
        function myFunction() {
          document.getElementById("para").innerHTML = "This is an paragraph is changed.";
          }
    </Script>
</body>
</html>
```

Here, clicking the button changed the contents of the line of paragraph.When the button is called the myfunction function is called where the contents of paragraph which have id=para is changed by reassigning the paragraph with new text.

Using javascript functions we made the page interactive for the user.

We will additionally discuss about the topic **Git and Github.It is very crucial in order to manage a project, control version and to Backup-recovery of the project. . It allows multiple developers to collaborate on projects by providing mechanisms to manage, track, and merge changes efficiently.

Let’s start with a Diagram:

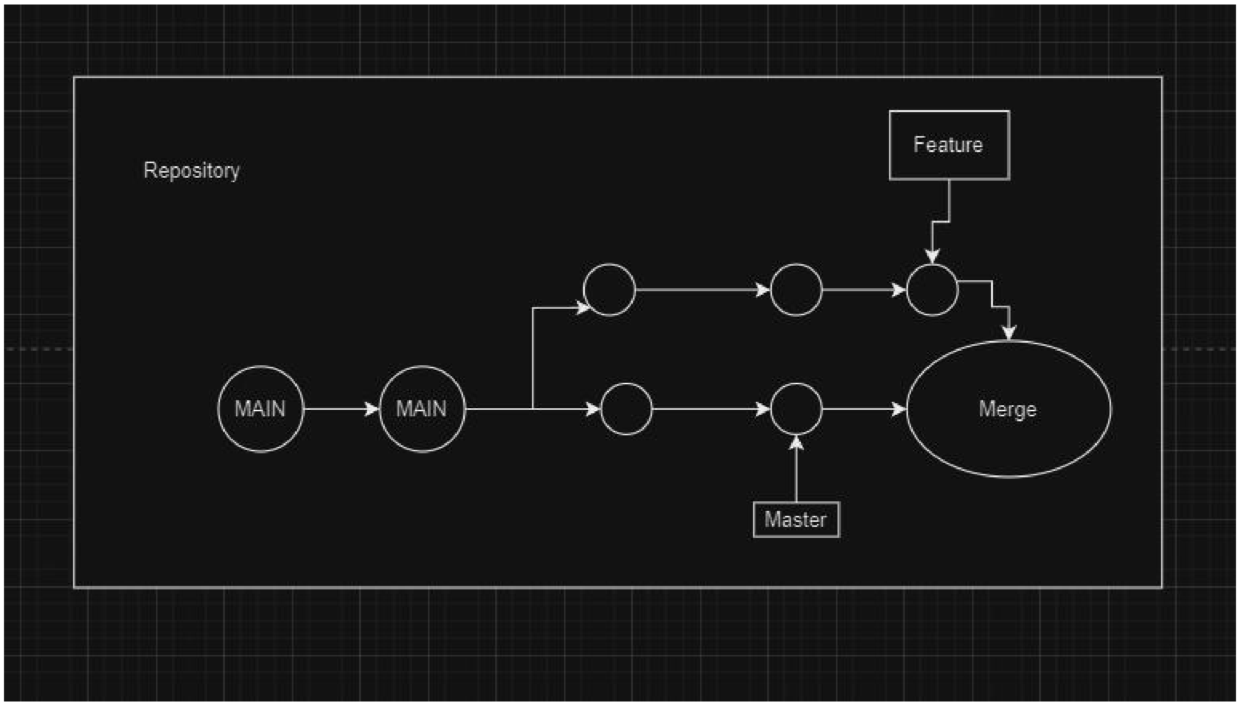


This diagram represents a basic Git workflow with GitHub:

- Developers clone the remote repository from GitHub to their local machines.
- Each developer works in their own local repository, creating branches for new features or bug fixes.
- After making changes, developers commit their changes to their local repository and push their branches to the remote repository on GitHub.
- Developers can then open pull requests on GitHub to propose their changes for review and integration into the main branch of the remote repository.
- Other team members can review the changes, provide feedback, and merge the pull requests as needed.

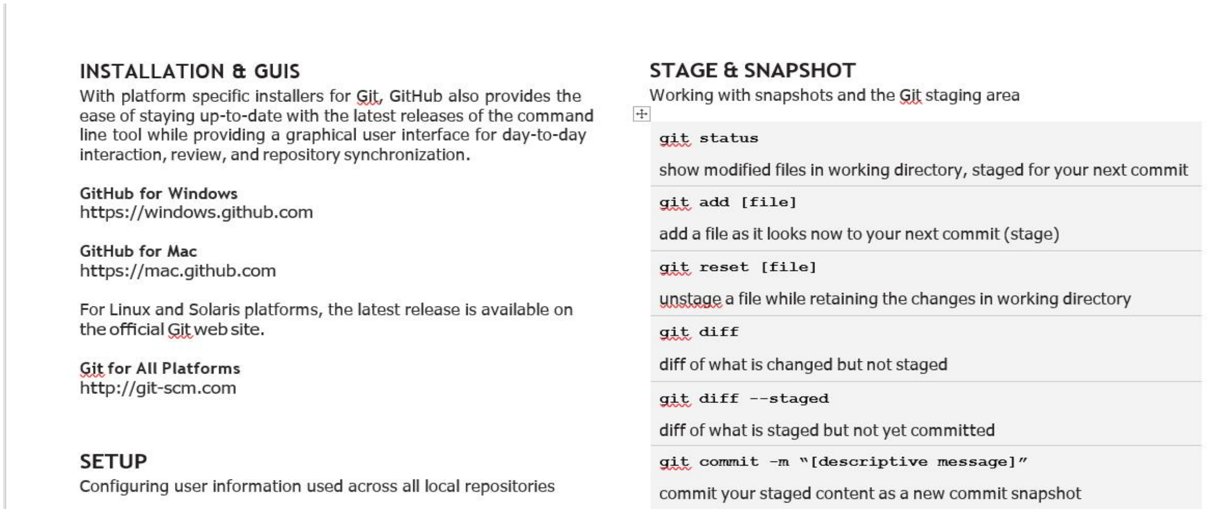
In a Repository there can be several branches. Means teams are working in different part of the project or features.

Here, simply putting a diagram of a repository with several branches:



This is how branches are work.

To work with the GIT we need certain codes to run on the git terminal. Here's a copy of necessary codes to work with.



INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status` show modified files in working directory, staged for your next commit
`git add [file]` add a file as it looks now to your next commit (stage)
`git reset [file]` unstage a file while retaining the changes in working directory
`git diff` diff of what is changed but not staged
`git diff --staged` diff of what is staged but not yet committed
`git commit -m "[descriptive message]"` commit your staged content as a new commit snapshot

`git config --global user.name "[firstname lastname]"`

set a name that is identifiable for credit when review version history

`git config --global user.email "[valid-email]"`

set an email address that will be associated with each history marker

`git config --global color.ui auto`

set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`

initialize an existing directory as a Git repository

`git clone [url]`

retrieve an entire repository from a hosted location via URL

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`

list your branches. a * will appear next to the currently active branch

`git branch [branch-name]`

create a new branch at the current commit

`git checkout`

switch to another branch and check it out into your working directory

`git merge [branch]`

merge the specified branch's history into the current one

`git log`

show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`

show the commit history for the currently active branch

`git log branchB..branchA`

show the commits on branchA that are not on branchB

`git log --follow [file]`

show the commits that changed file, even across renames

`git diff branchB...branchA`

show the diff of what is in branchA that is not in branchB

`git show [SHA]`

show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`

add a git URL as an alias

`git fetch [alias]`

fetch down all the branches from that Git remote

`git merge [alias]/[branch]`

merge a remote branch into your current branch to bring it up to date

`git push [alias] [branch]`

Transmit local branch commits to the remote repository branch

`git pull`

fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`

delete the file from project and stage the removal for commit

`git mv [existing-path] [new-path]`

change an existing file path and stage the move

`git log --stat -M`

show all commit logs with indication of any paths that moved

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`

system wide ignore pattern for all local repositories

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`

apply any commits of current branch ahead of specified one

`git reset --hard [commit]`

clear staging area, rewrite working tree from specified commit

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`

Save modified and staged changes

`git stash list`

list stack-order of stashed file changes

`git stash pop`

write working from top of stash stack

`git stash drop`

discard the changes from top of stash stack