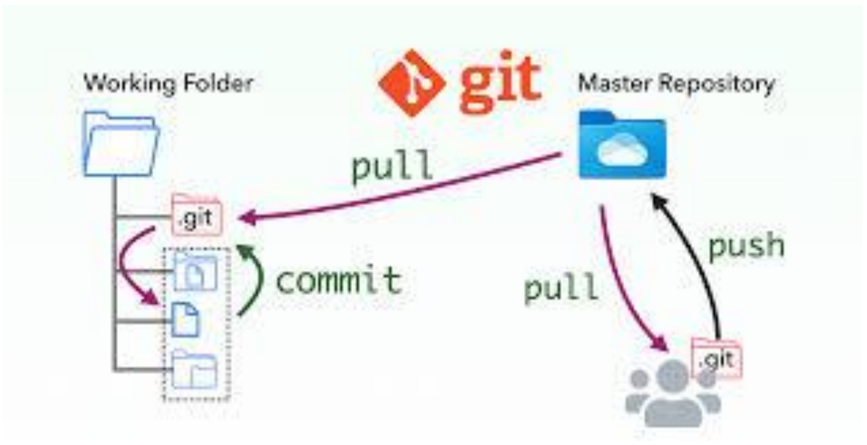
Git Basics

Introduction to Git

Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on the same project without overwriting each other's changes and keeps a record of all changes made to the project.

In this session, we will cover the fundamentals of Git, its basic workflow, and how to use it effectively with GitHub or GitLab.



Git Fundamentals

What is Git?

Git is a free and open-source version control system used for tracking changes in source code during software development. It enables collaboration between multiple developers and maintains a history of every modification. Version control is essential because it helps manage source code, resolve conflicts between developers, and provide a historical record of what has changed and why.

Some key features of Git:

- **Distributed**: Git stores the entire repository on every user's computer, making it easy to work offline.
- **Fast**: Git is designed to handle projects of all sizes with speed and efficiency.
- **Reliable**: It ensures the integrity of your code and keeps track of all changes.

Why Use Git?

- **Collaboration**: Multiple developers can work on the same project simultaneously.
- **Version History**: Git tracks all the changes in your project, allowing you to go back to any point in time.
- **Branching**: Git supports branching, enabling multiple features to be developed simultaneously without affecting the main project.

- **Backup**: Every clone of the repository is a full backup.

Git Basic Commands

1. Initializing a Repository (git init)

The first step in using Git is to initialize a Git repository. This command sets up the necessary structure for version control and tracking in your project.

```
git init
```

When you run this command, it creates a hidden .git directory that contains all of the configuration files and tracking information.

- **When to use it**: Run git init when starting a new project that you want to track using Git.

2. Staging Changes (git add)

Once you have made changes to your files, you need to stage them before committing. Staging allows you to prepare a snapshot of your changes before committing them.

```
git add <file-name>
```

You can stage all changes at once with:

```
git add
```

This command adds all the modified and new files to the staging area.

- **When to use it**: Run git add after modifying files and before committing them. This marks the changes that you want to include in the next commit.

3. Committing Changes (git commit)

After staging changes, the next step is to commit them. Committing in Git means recording the changes you've made to the repository's history.

```
git commit -m "Your commit message"
```

The -m flag allows you to write a short, descriptive message about the changes made.

- **When to use it**: Run git commit after staging files to create a snapshot of the changes. Each commit should have a clear and concise message explaining what the changes are.

Basic Git Workflow

Git has a simple workflow that revolves around modifying files, staging changes, and committing them. Here's a breakdown of the process:

1. **Modify**: You work on your project by adding, modifying, or deleting files.

2. **Stage**: After making changes, you stage the changes by using the git add command.
3. **Commit**: Once changes are staged, you commit them using the git commit command with a message that describes the changes.

Example Workflow

1. **Create a new file:**

```
echo "Hello, Git!" > example.txt
```

2. Stage the file for commit:

```
git add example.txt
```

3. Commit the changes:

```
git commit -m "Add example.txt with initial content"
```

The file is now tracked by Git, and the changes have been committed.

Viewing the History (git log)

To view the history of commits, you can use the following command:

```
git log
```

This command shows a list of commits, including commit messages, authors, and timestamps.