## **QUICK SYLLABUS GLANCE:**

# **Notes - Git & GitHub Bootcamp**

## **Day 1: Introduction to Git**

* **Introduction to Version Control**
  + Importance of version control in software development.
  + Brief comparison of Git with other version control systems.
* **Setting Up and Configuring Git**
  + Installing Git: **git --version** to check if Git is already installed.
  + Setting user name: **git config --global user.name "Your Name"**.
  + Setting user email: **git config --global user.email "your\_email@example.com"**.
* **Basic Git Operations**
  + Initializing a new Git repository: **git init**.
  + Cloning an existing repository: **git clone <repository-url>**.
  + Checking status: **git status**.
  + Adding changes to staging area: **git add <file-name>** or **git add .** for all changes.
  + Committing changes: **git commit -m "Commit message"**.
  + Viewing commit history: **git log**.
* **Branching and Merging**
  + Creating a new branch: **git branch <branch-name>**.
  + Switching branches: **git checkout <branch-name>**.
  + Merging branches: **git merge <branch-name>**.
  + Resolving merge conflicts: Manual process, then **git add <resolved-file>**, and finally **git commit**.

## **Day 2: Deep Dive into Git**

* **Advanced Branching Techniques**
  + Renaming branches: **git branch -m <old-name> <new-name>**.
  + Deleting branches: **git branch -d <branch-name>**.
* **Undoing Changes**
  + Undoing working directory changes: **git checkout -- <file-name>**.
  + Reverting commits: **git revert <commit-hash>**.
  + Resetting to a previous commit: **git reset --hard <commit-hash>**.
* **Stashing and Rebasing**
  + Stashing changes: **git stash** and **git stash apply**.
  + Rebasing: **git rebase <base-branch>**.
* **Remote Repositories**
  + Viewing remote repositories: **git remote -v**.
  + Adding a remote repository: **git remote add <name> <url>**.
  + Fetching from a remote repository: **git fetch <remote-name>**.
  + Pulling from a remote repository: **git pull <remote-name> <branch-name>**.
  + Pushing to a remote repository: **git push <remote-name> <branch-name>**.

## **Day 3: Introduction to GitHub**

* **Exploring GitHub**
  + Creating and setting up a new GitHub account.
  + Creating new repositories on GitHub.
  + Understanding the GitHub interface and features.
* **Collaborating on GitHub**
  + Forking repositories: GitHub UI operation.
  + Cloning a forked repository: **git clone <fork-url>**.
  + Configuring a remote for a fork: **git remote add upstream <original-repo-url>**.
  + Syncing a fork: **git fetch upstream** and then **git merge upstream/<branch-name>**.
  + Making pull requests: GitHub UI operation.
  + Reviewing and managing pull requests: GitHub UI operation.
  + Opening, labelling, and closing issues: GitHub UI operation.
* **GitHub Best Practices and Workflows**
  + Writing good commit messages.
  + Using branches for feature development.
  + Pull request based workflows.
  + Introduction to GitHub Actions for CI/CD.

## **NOTES:**

## **What is Version Control?**

Version control is a system that allows us to manage and track changes made to files and documents over time. It provides a structured way to store, organize, and retrieve different versions of files, enabling developers to collaborate seamlessly and keep track of project history.

## **Importance of Version Control in Software Development**

*1. Collaboration:* Version control systems facilitate collaboration among developers working on the same project. It allows multiple team members to work on the code simultaneously, keeping track of who made what changes and when.

*2. History and Accountability:* Version control maintains a detailed history of changes, allowing developers to view, compare, and revert to previous versions if needed. This accountability ensures that no work is lost or overwritten accidentally.

*3. Branching and Experimentation:* Version control systems like Git enable developers to create branches, which are independent lines of development. This feature allows for experimentation and the isolation of new features or bug fixes before integrating them into the main codebase.

*4. Conflict Resolution:* With multiple developers working on the same code base, conflicts may arise when merging changes. Version control systems provide mechanisms to identify and resolve conflicts effectively, ensuring a smooth integration process.

## **Benefits and Use Cases**

The benefits of version control systems extend beyond software development and can be leveraged in various scenarios, including:

*1. Source Code Management:* Version control systems allow teams to manage and track changes to their source code, ensuring efficient collaboration, code integrity, and seamless integration of new features.

*2. Documentation Management:* Version control can also be applied to manage documentation, making it easier to track revisions, collaborate on technical documents, and maintain a reliable documentation history.

*3. Content Management:* Version control is useful for managing content such as website pages, blog posts, or marketing materials. It enables content creators to track changes, revert to previous versions, and collaborate on content development.

*4. Project Management:* Version control systems can serve as a centralized repository for project assets, including design files, project plans, and configuration files, making it easier to manage and track changes across various project components.

# **Introduction to Git**

## **Git’s Distributed Nature**

Git is a distributed version control system, which means that every developer working on a project has a complete copy of the repository, including its entire history. Unlike centralized systems, where a single server stores the repository and users fetch and submit changes to that central server, Git allows developers to work independently, commit changes locally, and synchronize those changes with others when needed. This distributed nature of Git enables offline work, faster operations, and increased resilience against server failures.

## **Comparison to Other Version Control Systems**

*1. Decentralized Workflow:* Unlike centralized systems like SVN, Git does not rely on a central server for every operation. Developers have the freedom to work independently and commit changes locally, providing greater flexibility and enabling faster development.

*2. Branching and Merging:* Git makes branching and merging easy and efficient. Creating branches allows developers to work on separate features or experiments without interfering with the main codebase. Merging branches combine changes seamlessly, preserving project history and facilitating collaboration.

*3. Speed and Performance:* Git is designed to handle large projects and perform well even with extensive histories. Operations such as committing changes, switching branches, and comparing versions are incredibly fast, enhancing developer productivity.

*4. Data Integrity:* Git uses a cryptographic hash algorithm (SHA-1) to ensure data integrity. Every change made to files or directories is tracked by a unique hash, making it virtually impossible for data to be lost or tampered with unnoticed.

## **Git’s Key Features**

*1. Commit History:* Git keeps a detailed and chronological record of all commits, allowing developers to track changes, view commit messages, and understand the evolution of the codebase.

*2. Branching and Merging:* Git’s branching model is lightweight and flexible, enabling the creation of branches for different features or bug fixes. Merging branches allows for the seamless integration of changes.

*3. Staging Area:* Git has a staging area (also known as the index) where developers can selectively choose which changes to include in the next commit. This feature provides fine-grained control over committing changes.

*4. Distributed Architecture*: Git’s distributed nature allows developers to work offline, commit changes locally, and synchronize with others later. It ensures that each developer has a complete copy of the repository.

*5. Conflict Resolution:* Git provides tools for resolving merge conflicts that may occur when integrating changes from different branches. Developers can manually resolve conflicts, ensuring code integrity during the merge process.

# **Setting up Git**

## **How to Install Git**

***1. Windows:***

- Visit the official Git website at <https://git-scm.com/> and download the Git for Windows installer.

- Run the downloaded installer and follow the prompts.

- Select the desired installation options, such as the installation location and components to install.

- Choose the default options unless we have specific requirements.

- Complete the installation process by clicking “Next” and then “Finish.”

- Open the command prompt or Git Bash to verify the installation by running the command:

git --version

***2. macOS:***

- The easiest way to install Git on macOS is by using Homebrew, a popular package manager. Open the Terminal application.

- Install Homebrew by running the following command:

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

- Once Homebrew is installed, install Git by running:

brew install git

- Verify the installation by typing:

git --version

***3. Linux:***

- For Debian/Ubuntu-based distributions, open the Terminal and run the command:

sudo apt-get update

sudo apt-get install git

- For Fedora-based distributions, use the command:

sudo dnf install git

- For other Linux distributions, refer to their specific package manager or documentation to install Git.

- Verify the installation by running:

git --version

## **Configuration of Git Settings:**

After installing Git, it is essential to configure our personal information:

*1. Set the Username:*

- Open the Terminal or command prompt.

- Run the following command, replacing “Your Name” with the actual name:

git config --global user.name "Your Name"

*2. Set the Email:*

- Run the following command, replacing “your.email@example.com” with the email address:

git config --global user.email "your.email@example.com"

*3. Set Default Editor (Optional):*

- By default, Git uses the system’s default text editor. To set a specific editor, run the command:

git config --global core.editor "editor-name"

Replace “editor-name” with the preferred editor, such as “vim” or “nano.”

4. View Configuration:

- To view the Git configuration, including username and email, run:

git config --list

By installing Git and configuring the relevant settings, we are ready to start using Git for version control in our projects.

# **Git Basics**

*1. Repository:*

- A repository (or repo) in Git is a directory that contains all the files, directories and the complete history of a project.

- It is the central place where Git stores the entire history of commits, branches, tags, and other related data.

*2. Working Directory:*

- The working directory is the directory on our local machine where we have our files and directories for a particular project.

- It represents the current state of the project and serves as a sandbox for making changes.

*3. Staging Area (Index):*

- The staging area (also called the index) is an intermediate area between the working directory and the repository.

- It acts as a holding area where we can selectively choose which changes to include in the next commit.

- Files in the staging area are considered ready to be committed and will be part of the next commit.

## **Common Git Commands:**

*1. Initializing a Repository:*

- To initialize a Git repository in our project directory, open the command prompt or terminal and navigate to the project directory.

- Run the command:

git init

- This creates a new Git repository, and Git will start tracking changes in our project.

*2. Staging and Committing Changes:*

- Add files to the staging area using the command:

git add "filename"

(Replace “filename” with the actual file name or use `git add .` to add all files).

- Commit the changes to the repository by running:

git commit -m "Commit message"

(Replace “Commit message” with a descriptive message summarizing the changes made in this commit.)

*3. Viewing the Repository Status:*

- Check the status of our repository to see which files are modified, staged, or untracked by running:

git status

- It provides an overview of the current state of our repository and any pending changes.

*4. Viewing Commit History:*

- To view the commit history, including commit messages, authors, and timestamps, use the command:

git log

- It displays a chronological list of commits, starting with the most recent.

*5. Working with Branches:*

- Create a new branch using:

git branch "branch-name"

(Replace “branch-name” with the desired branch name.)

- Switch to a branch with:

git checkout "branch-name"

(Replace “branch-name” with the name of the branch.)

- List all branches in the repository:

git branch -a

- Deleting a Branch:

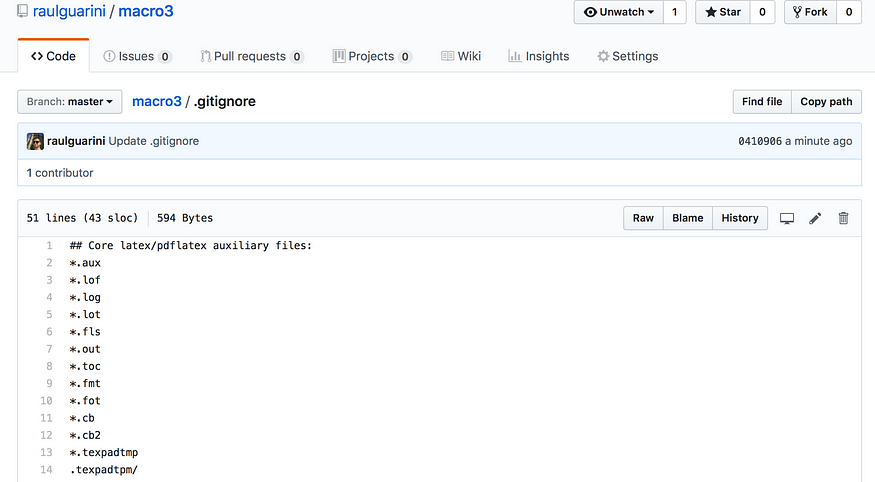
git branch -d "branch-name"

(To delete a branch, we need to make sure that we are on a different branch and run the command: (replace “branch-name” with the name of the branch that we want to delete.) Git will prevent us from deleting a branch that hasn’t been merged. To force deletion, use the command: `git branch -D “branch-name”`.)

*6. Ignoring Files:*

- Create a file named `.gitignore` in the root of the repository.

- Add filenames, patterns, or directories to this file to specify files that Git should ignore, such as build artifacts or sensitive information.



# **Branching in Git**

## **Branching Concepts and Importance**

Branching is a powerful feature in Git that allows for parallel lines of development within a repository. It enables developers to work on different features or experiments simultaneously without affecting the main codebase. Branches provide isolation, enabling changes to be made without disrupting the stability of the main branch. Branching is important for several reasons:

*1. Parallel Development:* Branches enable developers to work on multiple features or bug fixes simultaneously, speeding up development and improving productivity.

*2. Isolation and Experimentation:* Branches provide a safe space to experiment with new ideas or test different approaches. Developers can work independently without affecting the stability of the main codebase.

*3. Collaboration:* Branches facilitate collaborative development. Team members can work on separate branches and later merge their changes back into the main branch, ensuring smooth integration of features.

## **Popular Branching Strategies:**

*1. GitFlow:*

- GitFlow is a branching model that emphasizes a structured approach to branch management.

- It involves two main branches: “master” and “develop.” The “master” branch represents production-ready code, and the “develop” branch contains ongoing development.

- Feature branches are created from the “develop” branch for implementing new features or bug fixes.

- Once a feature is completed, it is merged back into the “develop” branch, and releases are made from the “develop” branch to the “master” branch.

*2. Feature Branching:*

- Feature branching is a simpler branching strategy where each new feature is developed in a dedicated branch.

- Developers create a new branch for each feature or task they are working on.

- Once a feature is completed, it is merged back into the main branch.

- This strategy allows for more flexibility and agility in development.

These branching strategies provide frameworks for managing parallel development and ensuring smooth collaboration among team members. Choosing the right branching strategy depends on the specific needs and requirements of the project.

# **Collaborative Development with Git**

## **Remote Repositories and Their Role in Collaboration**

*1. Centralized Code Storage:* Remote repositories act as centralized locations for storing the codebase. They provide a common ground for collaboration, allowing multiple developers to access and contribute to the project.

*2. Sharing and Synchronization:* Developers can push their local changes to a remote repository to share their work with others. They can also pull changes from the remote repository to synchronize their local copy with the latest updates made by other team members.

## **Cloning a Repository from a Remote Source**

*1. Obtain the Repository URL:* Obtain the URL of the remote repository that we want to clone. This can typically be found on the hosting platform (e.g., GitHub, GitLab).

*2. Open the Command Prompt or Terminal:* Open the command prompt or terminal on our local machine.

*3. Navigate to the Desired Directory:* Use the `cd` command to navigate to the directory where we want to clone the repository.

*4. Clone the Repository:* Run the following command, replacing “repository-url” with the actual URL of the remote repository:

git clone "repository-url"

5. Verify the Cloning: Git will download the repository files and create a local copy. Verify the cloning process by checking if the repository directory has been created in the specified location.

## **Demonstrating Push and Pull Commands for Synchronizing Changes**

Once we have cloned a repository and made changes locally, we can use the following commands to synchronize our changes with the remote repository and collaborate effectively:

*1. Pushing Changes:*

- To send our local commits to the remote repository, use the command:

git push origin "branch-name"

(Replace “branch-name” with the branch name that we want to push.)

*2. Pulling Changes:*

- To retrieve the latest changes from the remote repository, use the command:

git pull origin "branch-name"

- This command fetches the latest commits from the remote repository and merges them into our current branch.

# **Resolving Conflicts**

## **Merge Conflicts and How They Occur**

Merge conflicts occur when Git encounters conflicting changes in different branches that cannot be automatically merged. Conflicts typically happen when two or more developers make conflicting modifications to the same file or code section. It’s important to understand the following about merge conflicts:

*1. Conflict Occurrence:* Conflicts arise when Git cannot determine which changes to keep during a merge or a pull operation.

*2. Conflicting Changes:* Conflicts can arise when multiple developers modify the same line(s) of code or when changes overlap in a way that Git cannot automatically reconcile.

## **Step-by-Step Guide on Resolving Conflicts Manually**

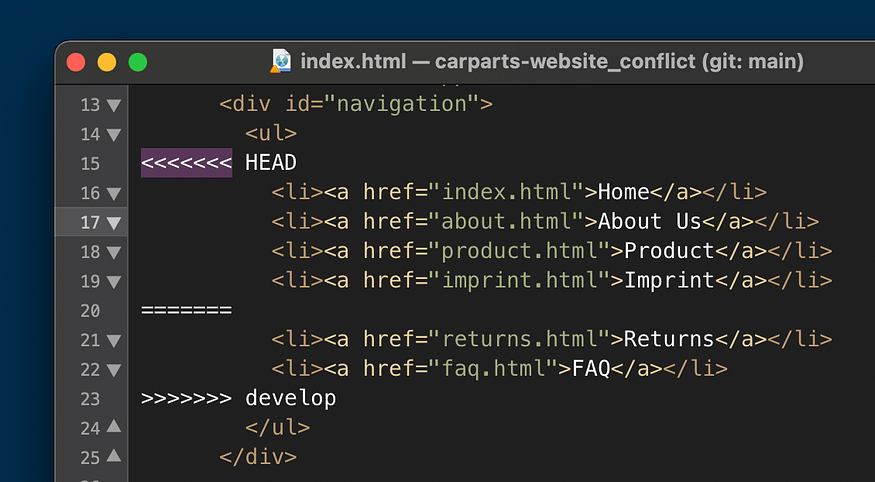
*1. Identify Conflicted Files:* Git will notify us of conflicts during a merge or pull operation. Identify the files marked as “conflicted” or “unmerged.”

*2. Open Conflicted Files:* Open the conflicted files in a text editor. Git will indicate the conflict with special markers, like

*<<<<<<< HEAD*

and

*>>>>>>> branch-name*

**

Merge Conflicts

*3. Review Conflicting Changes:* Within the conflicted file, locate the conflicting sections marked by the markers. Review the changes made in both the current branch (HEAD) and the incoming branch (the other branch).

*4. Choose Desired Changes:* Decide which changes to keep or modify. Edit the conflicted sections manually to resolve the conflict based on our intentions.

*5. Remove Conflict Markers:* Delete the conflict markers (e.g., “<<<<<<< HEAD”, “=======”, “>>>>>>> branch-name”) once we have resolved the conflict. Ensure that the resulting code is syntactically correct and aligned with the intended changes.

*6. Save and Commit Changes:* Save the modified files, then stage and commit the changes using `git add` and `git commit`. Git will detect that the conflict has been resolved.

## **Tips and Best Practices for Avoiding Conflicts**

While conflicts are a normal part of collaborative development, following these tips can help minimize their occurrence:

*1. Pull Before Pushing:* Always update the local branch with the latest changes from the remote repository before pushing our own changes to minimize conflicts.

*2. Communicate and Coordinate:* Communicate with the team to ensure that multiple developers are not working on the same code sections simultaneously. Coordinate work to reduce the chances of conflicting changes.

*3. Small and Frequent Commits:* Make small, focused commits instead of large, sweeping changes. This minimizes the likelihood of conflicts and makes it easier to understand and resolve them if they do occur.

*4. Use Feature Branches:* Create separate feature branches for each developer or feature. This way, changes are isolated, reducing the chances of conflicts with other branches.

*5. Regular Testing and Review:* Regularly test and review code changes to identify conflicts or issues early in the development process. This allows for prompt resolution and smoother collaboration.

# **Git Best Practices**

## **Frequent Commits and Their Benefits**

Frequent commits, where code changes are divided into smaller, focused units, offer several benefits:

*1. Granular History:* Frequent commits provide a detailed history of changes, making it easier to track and understand the evolution of the codebase over time.

*2. Easier Troubleshooting:* Smaller commits make it simpler to pinpoint the introduction of bugs or issues, aiding in efficient troubleshooting and debugging.

*3. Collaboration and Code Reviews:* By committing changes frequently, team members can collaborate effectively and conduct code reviews more efficiently, as changes are manageable and can be reviewed incrementally.

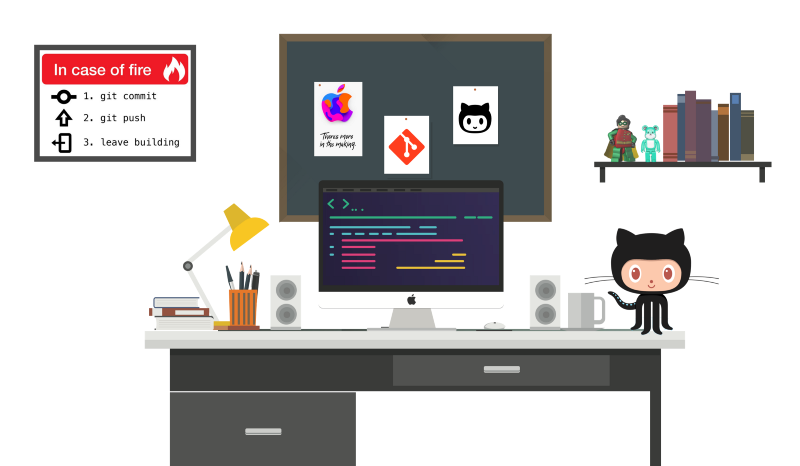
## **Importance of Writing Clear and Concise Commit Messages**

Clear and concise commit messages are crucial for effective collaboration and maintaining a well-documented codebase. Key points to consider include:

*1. Descriptive Summary:* Provide a succinct summary of the changes made in the commit. It should convey the purpose and impact of the changes.

*2. Additional Details:* If necessary, provide additional context or details about the commit, such as the rationale behind the changes or references to related issues or tasks.

*3. Avoid Ambiguity:* Ensure that the commit message is clear and unambiguous, allowing other developers to understand the intent of the changes without having to delve into the code.



In conclusion, Git is a powerful version control system that enables efficient collaboration and effective management of code changes. By mastering Git’s basics, utilizing its features, and following best practices, we can streamline our development workflow, enhance collaboration, and maintain a well-documented codebase.

**GIT FORK VS CLONE :**

**The key difference between Git clone and fork comes down to how much control and independence you want over the codebase once you've copied it.**

**Any public Git repository can be forked or cloned. A fork creates a completely independent copy of Git repository. In contrast to a fork, a Git clone creates a linked copy that will continue to synchronize with the target repository.**

## **Git clone vs. fork**

**When a Git repository is cloned, the target repository remains shared amongst all of the developers who had previously contributed to it. Other developers who had previously contributed to that codebase will continue to push their changes and pull updates from the cloned repository. Any developer who clones a repository can synchronise their copy of the codebase with any updates made by fellow developers.**

**In contrast to a clone, a Git fork operation will create a completely new copy of the target repository. The developer who performs the fork will have complete control over the newly copied codebase. Developers who contributed to the Git repository that was forked will have no knowledge of the newly forked repo. Previous contributors will have no means with which they can contribute to or synchronize with the Git fork unless the developer who performed the fork operation provides access to them.**

## **How to choose between Git fork and clone**

**A programmer who joins a software development team and plans to contribute back to the codebase will typically clone the target repository. When changes or updates are made, either by the developer or by other members of the team, any clone can be easily synchronised with a** [**git push**](https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/How-to-git-push-GitLab-commits-to-origin) **or a git pull.**

**A developer who wants to set up a new, separate and isolated project that is based on a publicly accessible Git repo should perform a fork.**

**Some famous examples of Git forks include:**

* [**Jenkins**](https://www.theserverside.com/quiz/Ten-Jenkins-quiz-questions-to-test-your-install-and-config-IQ)**, which is a** [**fork of**](https://medium.com/swlh/hudson-is-retiring-the-end-of-a-jenkins-story-9cee866135ab) **Hudson**
* **Fire OS for Kindle Fire, which is a fork of Android**
* **LibreOffice, which is a fork of OpenOffice**
* **Ubuntu, which is a fork of Debian**
* **MariaDB, from MySQL**

## **The Git fork and clone workflow**

**When a repository is forked, developers who plan to work with the new codebase will still need to perform a git clone operation on the forked repository. You'll still need to run push and pull operations to synchronise local changes with the forked repo, as shown in the diagram below. However, changes and updates to the forked repository will be isolated to the fork and will not be reflected in the original repo.**

**GITHUB - CI / CD:**

[**https://mindmajix.com/github-ci-cd-tutorial**](https://mindmajix.com/github-ci-cd-tutorial)

**ADDITIONAL LINKS:**[**Git Rebase - GeeksforGeeks**](https://www.geeksforgeeks.org/rebasing-of-branches-in-git/)

[**Git Branch Merge (w3schools.com)**](https://www.w3schools.com/git/git_branch_merge.asp)