**Module 1: Git**

**What is Git?**



Git is a **distributed version control system** used to track changes in source code during software development. It allows multiple developers to collaborate on a project by managing various versions of files, and it provides tools to merge changes from different contributors efficiently.



**Why is Git Important?**

* **Collaboration**: It enables teams to work together seamlessly without overwriting each other’s work.



* **Code Integrity**: By tracking changes, Git ensures that code is safe from accidental or unintentional modifications.



* **Backup and Recovery**: Since every user has a full copy of the repository, losing a central server won’t result in data loss.



* **History and Accountability**: Git keeps a detailed history of who changed what and when, making it easier to track issues or undo mistakes.

Installing Git

<https://git-scm.com/downloads>

**Basic Commands for Versioning and Managing Changes**

A diagram of a software development

Description automatically generated



**1.** **git init**: **Initialize a New Git Repository**

This command creates a new Git repository in your project folder, setting up a .git directory to track changes.

**Git init**

**2.** **git add: Stage Changes for Commit**

git add moves changes from the working directory to the staging area, preparing them to be included in the next commit.



This gives you control over what gets included in each commit, allowing you to break up changes into logical commits rather than committing everything at once.



**Git add .**

**3.git status: Check the Status of Your Repository**

This command shows the current state of the working directory and staging area, including:

* Untracked files (new files not yet added)
* Changes to tracked files
* Files staged for the next commit

**Git status**

**4. git commit: Commit Staged Changes**

A commit records changes in the repository’s history. It’s like a snapshot of the project at a particular point

A commit also includes a message describing the changes

**git commit -m "Commit message describing changes"**

**Working with Repositories**

**1. git clone: Cloning Repositories**

* To copy an existing remote repository to your local machine, use git clone. This command creates a local copy of the repository and sets up a connection to the remote.

git clone <repository\_url>

**2. Working with Remote Repositories**

* A **remote repository** is a version of your project hosted on the internet or another network

To work with remote repositories, you need to know some key commands:

* **git fetch**: Fetch updates from the remote repository but don’t apply them to your local branch.
* **git pull**: Fetch and apply changes from the remote repository to your local branch
* **git push**: Send your local commits to the remote repository.

**Branching**

**1. Creating and Switching Branches**

* **Branches** allow you to work on features, fixes, or experiments separately from the main codebase.



Create a new branch using:



**git branch <branch\_name>**

Switch to another branch:

**git checkout <branch\_name>**

**Importance of Branching in Collaboration and Feature Development**

* Branches make it easy for multiple developers to work on different features or bug fixes without affecting the main project (usually on the main or master branch).



* Branching provides a safe environment to work on a new feature and test it independently.



* Once a branch is stable, you can merge it back into the main branch, ensuring that the feature is integrated without disrupting the project.



**Merging**

**1. Merging Branches Using git merge**

* After completing work on a feature branch, you can merge it back into another branch

git checkout main



**git merge <feature\_branch>**



**2. Fast-Forward vs. Non-Fast-Forward Merges**

* **Fast-forward merge**: This occurs when the target branch has not diverged from the source branch. Git simply advances the pointer of the target branch to the latest commit of the source branch.
  + git checkout main
  + git merge feature-branch
  + (No extra commit is made, Git just “moves” the branch pointer)
* **Non-fast-forward merge**: If the target branch has diverged from the source branch, Git creates a new "merge commit" to join the histories.
  + - git merge --no-ff feature-branch
  + (This creates a separate merge commit, showing that changes were integrated.)

A graph with colorful dots and lines

Description automatically generated

**Handling Conflicts**

**1. Resolving Conflicts During Merges**

* **Merge conflicts** occur when Git can’t automatically merge changes between branches, usually because the same lines of code were changed in both branches.
* When a conflict arises:
  1. Git pauses the merge and marks the conflict in the file(s).
  2. You must manually resolve the conflict, keeping the changes you want and discarding the others.



**2. Tools and Strategies for Conflict Resolution**

* **Manual Resolution**:
  + Open the conflicting file(s) and decide which changes to keep or how to combine them
  + After resolving the conflicts, stage the resolved files and commit them

**Git Tools for Conflict Resolution**:

* **Git’s Built-in Conflict Resolver**: Git itself marks conflicts and provides hints in the code.
* **GUI Tools**: Git GUIs (e.g., SourceTree, GitKraken) provide visual conflict resolution interfaces, making it easier to see differences and resolve conflicts.

**Best Practices for Avoiding Conflicts**:

1. **Pull Often**: Regularly pull updates from the remote repository to keep your branch in sync and minimize divergence.
2. **Small, Frequent Commits**: Committing small, incremental changes reduces the risk of conflicts.



1. **Rebase**: Instead of merging, consider using git rebase, which replays your branch’s commits on top of another branch’s commits. This creates a linear history and reduces merge conflicts.



**Lab : Version control with git**

**Git Init**

**Git Add.**

**Git commit**

**Git branch**

**Module 2 &3**



**Tags in Git**

Tags in Git are used to mark specific points in your repository’s history. They are often used to mark important commits, such as releases or significant milestones.



**Types of Tags**:



**Lightweight Tags**



* **Description**: A simple pointer to a commit, like a branch that doesn’t change. It’s essentially a named reference to a commit hash.



* **Use case**: Used for temporary or less significant milestones.



How to create:



**git tag <tag\_name>**



**Annotated Tags**

* **Description**: A tag that stores additional metadata, such as the tagger’s name, email, date, and a message. Annotated tags are stored as full objects in the Git database.



* **Use case**: Used for marking official releases or important milestones, as they contain descriptive information.



How to create

**git tag -a <tag\_name> -m "Tag message"**

Example

**git tag -a v1.0 -m "Initial release version 1.0"**

Git tag

**Viewing Tags**

* To see all the tags in your repository

Git tag

**Pushing Tags to a Remote Repository**: By default, tags are not pushed to remote repositories. To push all tags:

**git push --tags**

**What are Submodules?**



A **submodule** in Git allows you to keep a Git repository as a subdirectory of another Git repository. This is useful when you want to include an external repository (or another project) inside your own project but maintain separate histories.



**Use Cases:**

* Submodules are particularly useful in projects that require integrating third party libraries, tools, or dependencies, while allowing you to track and control which version of the submodule is used.



**Adding a Submodule**

To include an external repository as a submodule:

1. **Navigate to Your Repository:**

cd your\_project\_directory

1. **Add a Submodule:**

* Run the following command to add the submodule

**git submodule add <repository\_url> <submodule\_directory>**

For example

**git submodule add https://github.com/user/repo.git libs/repo**

1. **Commit the Changes:**

* Git will create a .gitmodules file that tracks the submodule's URL and the submodule directory. Commit these changes:

git commit -m "Added submodule repo"

**Cherry-picking a Specific Commit**



**Cherry-picking** is the process of applying a specific commit from one branch to another. It’s particularly useful when you want to apply only a certain change from a feature branch or when you need a bug fix but don't want to merge the entire branch.



**Steps:**

1. **Find the Commit You Want to Cherry-Pick:**
   * First, use git log to find the commit hash (SHA-1) of the commit you want to cherry-pick
   * Copy the commit hash

**git log**

1. Switch to the Branch You Want to Apply the Commit To:

**git checkout <target\_branch>**

1. **git cherry-pick <commit\_hash>**

4. **Resolve Conflicts (if any):**

* If there are conflicts, Git will pause the cherry-pick and mark the conflicting files.
* After resolving conflicts, complete the cherry-pick by staging and committing the changes.

**Finalizing the Cherry-Pick:**

* After resolving any conflicts, continue the cherry-pick process by committing the changes

git commit

**Use Cases for Cherry-Picking in Real-World Scenarios**

* **Bug Fixes**: When a bug fix is made on one branch, and you want to apply it to another branch (e.g., from a feature branch to main).



* **Selective Feature Transfer**: Transfer individual feature commits between branches without merging the entire feature branch.



**What are Git Hooks?**

Git hooks are scripts that Git automatically executes before or after certain Git events, such as commits, pushes, or merges. These hooks allow you to automate tasks, enforce coding standards, or integrate with other tools.



Types of Git Hooks

**Pre-commit Hook:**

* This hook runs before a commit is finalized.



* **Use cases**:
  + Linting or code style checks before committing.



* + Ensuring commit messages follow a specific format.



* + Running tests to ensure they pass before committing.



**Pre-push Hook:**



* This hook runs before changes are pushed to the remote repository.
* **Use cases**:
  + Running tests or static analysis to ensure code quality before pushing.



* + Preventing sensitive information from being pushed



**Post-merge Hook:**



* This hook runs after a git merge operation completes.
* **Use cases**:
  + Automatically running tests after merging.



* + Checking for changes in dependencies and installing them.



**Lab Assignment: Cherry-Pick a Specific Commit**

* Create Two Branches
* Make some changes, commit them, and switch back to the main branch
* Cherry-Pick the Commit