***Git Basics Handbook***

## **I. Introduction to Version Control**

### **A. Definition and Significance of Version Control Systems**

Version Control Systems (VCS) are tools that help manage changes to source code over time. They track every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

### **B. Benefits of Utilizing Version Control for Software Development**

1. **Collaboration:** Multiple developers can work on the same project simultaneously without interfering with each other’s work.
2. **Backup and Restore:** Keeps a complete history of the project, making it easy to recover from any mistakes.
3. **Branching and Merging:** Facilitates the creation of branches to work on different features or fixes and merge them back into the main codebase.
4. **Tracking Changes:** Every change is documented with a message from the developer, making it easier to understand the history and reason for changes.
5. **Version History:** Enables developers to explore the entire history of the project, making it easier to understand the evolution of the code.

## **II. Core Concepts of Git**

### **A. Repositories: Local and Remote**

* **Local Repository:** A repository on your local machine where you can make changes, commit them, and explore the history of the project.
* **Remote Repository:** A repository hosted on a remote server (like GitHub, GitLab, or Bitbucket) used to collaborate with others and back up your local repository.

# Create a new local repository

**git init**

# Clone an existing remote repository

**git clone https://github.com/username/repo.git**

### **B. Working Directory: Workspace for Project Files**

The working directory is where you modify, add, or delete project files. It’s the place where you actively develop.

### **C. Staging Area (Index): Selecting Changes for Commits**

The staging area is a file (also called the index) that contains information about what will go into your next commit. It allows you to decide which changes you want to include in a commit.

# Add a file to the staging area

**git add filename**

# Add all changed files to the staging area

**git add .**

### **D. Commits: Capturing Project States with Descriptive Messages**

A commit is a snapshot of your repository at a specific point in time. Each commit has a unique ID and a descriptive message that explains why the changes were made.

# Commit changes with a message

**git commit -m "Descriptive commit message"**

### **E. Branches: Divergent Development Paths within a Repository**

Branches are separate lines of development within a repository. They allow you to work on different features or fixes without affecting the main project until you are ready to merge your changes back in.

# Create a new branch

**git branch feature-branch**

# Switch to the new branch

**git checkout feature-branch**

# Create and switch to a new branch in one command

**git checkout -b feature-branch**

## **III. Essential Git Commands**

### **A. Initialization: Creating a New Git Repository**

To create a new Git repository, navigate to your project’s directory and run:

# Initialize a new Git repository

**git init**

### **B. Tracking Changes: Identifying Modified Files**

To see which files have been modified, added, or deleted, use:

# Show the status of your working directory and staging area

**git status**

### **C. Staging and Committing: Preparing and Recording Changes**

**Staging Changes:**# Add changes to the staging area  
git add <file>

Add all changes:  
git add .

**Committing Changes:**

# Commit the staged changes  
git commit -m "Descriptive message about the changes"

### **D. Branching: Creating and Switching Between Development Lines**

**Creating a Branch:**# Create a new branch  
git branch <branch-name>

**Switching Branches:**# Switch to another branch  
git checkout <branch-name>

### **E. Merging: Integrating Changes from Different Branches**

**Merging a Branch into Current Branch:  
bash**

**Copy code**

**#** Merge a branch into the current branch

**git merge feature-branch**

### 

### **F. Remote Repositories: Collaboration and Shared Workspaces (Future Section)**

To collaborate with others and push your changes to a remote repository:

**Adding a Remote Repository:  
git remote add origin https://github.com/username/repo.git**

**Pushing Changes:**  
# Push changes to the remote repository

**git push origin main**

# Fetch changes from the remote repository

**git fetch origin**

# Pull changes from the remote repository and merge them

**git pull origin main**

## **IV. Mastering Git Workflows**

### **A. Feature Branch Workflow: Streamlined Development and Integration**

**Create a New Feature Branch:**  
# Create and switch to a feature branch

**git checkout -b feature-branch**

**Develop Your Feature and Commit Changes:**# After making changes, add and commit them

**git add .**

**git commit -m "Feature implementation"**

**Merge the Feature Branch into Main:**  
**git checkout main**

**git merge <feature-branch>**

### **B. Gitflow Workflow: Structured Approach for Large-Scale Projects**

Gitflow is a branching model designed for collaboration and release management.

1. **Feature Branches:** Develop new features.
2. **Develop Branch:** Integrate features and prepare for release.
3. **Release Branches:** Finalize release details.
4. **Master Branch:** Production-ready state of the code.

# Install git-flow (if needed)

brew install git-flow

# Initialize git-flow in your repository

**git flow init**

# Start a new feature

**git flow feature start feature-name**

# Finish the feature

**git flow feature finish feature-name**

# Start a release

**git flow release start 1.0.0**

# Finish the release

**git flow release finish 1.0.0**

# Start a hotfix

**git flow hotfix start hotfix-name**

# Finish the hotfix

**git flow hotfix finish hotfix-name**

## **V. Advanced Git Techniques**

### **A. Resolving Merge Conflicts: Handling Conflicting Changes**

When a conflict occurs during a merge, Git will notify you. Open the conflicted files and manually resolve the conflicts. After resolving:

**git add <resolved-file>**

**git commit -m "Resolved merge conflict"**

### **B. Stashing Changes: Temporarily Shelving Uncommitted Work**

To temporarily save changes without committing them:

git stash

To apply stashed changes later:

git stash apply

### **C. Using Tags: Annotating Specific Project Versions**

Tags are used to mark specific points in the repository’s history.

**Creating a Tag:**

# Create a lightweight tag

**git tag v1.0.0**

# Create an annotated tag

**git tag -a v1.0.0 -m "Version 1.0.0"**

# List tags

**git tag**

# Push tags to the remote repository

**git push origin --tags**