Git Basics Handbook:

Key concepts, Commands and Workflows

Table of Contents

I. Introduction to Version Control

A. Definition and Significance of Version Control Systems

Version Control Systems (VCS) are tools that manage changes to a project's files over time. They record modifications in a repository, allowing multiple developers to collaborate on a project efficiently.

Significance:

- **History Tracking**: Maintains a comprehensive history of changes, facilitating the understanding of a project's evolution.
- **Collaboration**: Enables multiple developers to work on the same project concurrently without overwriting each other's work.
- **Branching and Merging**: Supports parallel development through branches, which can be merged back into the main codebase.
- **Backup and Restore**: Acts as a safeguard, allowing previous versions of the project to be restored if needed.
- **Blame Assignment**: Identifies the author of changes, aiding in accountability and debugging.

B. Benefits of Utilizing Version Control for Software Development

- **Improved Collaboration**: Facilitates teamwork by providing a central repository for sharing changes and updates.

- **Code Quality**: Supports code reviews and continuous integration, enhancing the overall quality of the codebase.
- **Experimentation**: Allows safe experimentation with new features or ideas using branches without affecting the main project.
- **Risk Mitigation**: Reduces the risk of data loss and errors through robust version tracking and recovery mechanisms.
- **Audit Trail**: Keeps a detailed record of changes, useful for compliance and tracking down issues.

II. Core Concepts of Git

A. **Repositories:** Local and Remote

- **Local Repository**: A directory on your computer that contains the project files and a `.git` directory with all the version control information.
- **Remote Repository**: A version of your project hosted on the internet or network, allowing collaboration with others. Examples include repositories on GitHub, GitLab, and Bitbucket.

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

The working directory is the area where files are checked out from the repository. It contains the current version of the project files you are working on.

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

The staging area, or index, is an intermediary area where you can group changes before committing them to the repository. It allows you to fine-tune what will be included in your next commit.

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

A commit is a snapshot of the project at a specific point in time. Each commit includes a message that describes the changes made, helping to track the project's history and evolution.

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

Branches allow you to diverge from the main line of development and continue to work without affecting that main line. The default branch is usually called `main` or `master`. Branches enable multiple development lines to exist simultaneously.

III. Essential Git Commands

A. Initialization: Creating a New Git Repository

- Command: `git init`
  - Initializes a new Git repository in the current directory.

B. Tracking Changes: Identifying Modified Files

- Command: `git status`
  - Shows the status of changes as untracked, modified, or staged.

C. Staging and Committing: Preparing and Recording Changes

- **Staging**:
  - Command: `git add [file]`
    - Adds a file to the staging area.
  - Command: `git add .`
    - Adds all modified files to the staging area.
- **Committing**:
  - Command: `git commit -m "[message]"`
    - Records the staged changes with a descriptive message.

D. Branching: Creating and Switching Between Development Lines

- **Creating a Branch**:
  - Command: `git branch [branch-name]`
    - Creates a new branch.
- **Switching Branches**:
  - Command: `git checkout [branch-name]`
    - Switches to the specified branch.
- **Creating and Switching**:
  - Command: `git checkout -b [branch-name]`
    - Creates and switches to a new branch.

E. Merging: Integrating Changes from Different Branches

- Command: `git merge [branch-name]`
  - Merges the specified branch into the current branch.

F. Remote Repositories: Collaboration and Shared Workspaces

- **Adding a Remote**:
  - Command: `git remote add [name] [url]`
    - Adds a new remote repository.
- **Fetching Changes**:
  - Command: `git fetch [remote]`
    - Retrieves updates from the remote repository.

- **Pushing Changes**:
  - Command: `git push [remote] [branch]`
    - Pushes local changes to the remote repository.
- **Pulling Changes**:
  - Command: `git pull [remote] [branch]`
    - Fetches and merges changes from the remote repository to the local branch.

IV. Mastering Git Workflows

A. Feature Branch Workflow: Streamlined Development and Integration

The feature branch workflow involves creating a new branch for each feature or bug fix. This keeps the `main` branch clean and allows for focused development on individual features.

1. **Create a feature branch**: `git checkout -b feature-branch`
2. **Work on the feature**: Make changes and commit them to the feature branch.
3. **Merge the feature**: Once the feature is complete, switch to the main branch and merge the feature branch.

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

The Gitflow workflow uses two main branches:

- `main`: Stores the official release history.
- `develop`: Serves as an integration branch for features.

Additional branches include:

- Feature branches: For new features.
- Release branches: For preparing a new production release.
- Hotfix branches: For quick patches to the production version.

Steps:

1. Developing a feature: `git checkout -b feature-branch develop`
2. Finishing a feature: `git checkout develop && git merge feature-branch`
3. Creating a release: `git checkout -b release-branch develop`
4. Hotfixing: `git checkout -b hotfix-branch main`

V. Advanced Git Techniques

A. Resolving Merge Conflicts: Handling Conflicting Changes

When merging branches, conflicts may arise if changes overlap. Git highlights conflicts, and you can resolve them manually.

- Command: `git merge [branch-name]`
- Command: `git status` (to see conflicts)
- **Resolve conflicts** and then:
  - Command: `git add [file]`
  - Command: `git commit -m "Resolved merge conflicts"`

B. Stashing Changes: Temporarily Shelving Uncommitted Work

Stashing allows you to save changes temporarily without committing them, making it easier to switch branches.

- **Stashing**:
  - Command: `git stash`
- **Applying Stash**:
  - Command: `git stash apply`

C. Using Tags: Annotating Specific Project Versions

Tags mark specific points in the repository history, often used for releases.

- **Creating a Tag**:
  - Command: `git tag [tag-name]`
- **Listing Tags**:
  - Command: `git tag`
- **Pushing Tags**:
  - Command: `git push [remote] [tag-name]`