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. By keeping a detailed history of changes, they enable multiple developers to collaborate efficiently. VCS allows developers to revert to previous versions, track modifications, and manage parallel work on the same codebase.

B. Benefits of Utilizing Version Control for Software Development

- *Collaboration:* Enables multiple developers to work on the same project without conflicts.

- *History Tracking:* Keeps a record of every change made to the codebase.

- *Backup:* Acts as a backup by maintaining a copy of the code at different points in time.

- *Branching and Merging:* Facilitates experimentation and integration of new features.

- *Blame Assignment:* Identifies who made specific changes, which helps in debugging and accountability.

II. Core Concepts of Git

A. Repositories: Local and Remote

- *Local Repository:* A version of the project stored on the developer's local machine.

- *Remote Repository:* A version of the project hosted on a server, enabling collaboration among multiple developers.

B. Working Directory: Workspace for Project Files

The working directory is where the project files reside and where the developer performs their work. Changes made here are tracked by Git but not yet recorded.

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

The staging area is a preparatory area where changes are reviewed and selected before committing. It allows for organizing and grouping changes logically.

D. Commits: Capturing Project States with Descriptive Messages

Commits are snapshots of the project at a specific point in time. Each commit has a unique identifier and a descriptive message to explain the changes made.

E. Branches: Divergent Development Paths within a Repository

Branches allow for multiple lines of development within the same repository. They enable experimentation and development of new features without affecting the main codebase.

III. Essential Git Commands

A. Initialization: Creating a New Git Repository

git init

This command initializes a new Git repository in the current directory.

B. Tracking Changes: Identifying Modified Files

git status

Shows the status of changes in the working directory and staging area.

C. Staging and Committing: Preparing and Recording Changes

git add <file>

git commit -m "Descriptive commit message"

git add stages changes, and git commit records them with a message.

D. Branching: Creating and Switching Between Development Lines

git branch <branch-name>

git checkout <branch-name>

git branch creates a new branch, and git checkout switches to the specified branch.

E. Merging: Integrating Changes from Different Branches

git merge <branch-name>

Merges changes from the specified branch into the current branch.

F. Remote Repositories: Collaboration and Shared Workspaces

git remote add origin <repository-url>

git push -u origin <branch-name>

git pull origin <branch-name>

git remote add adds a remote repository, git push uploads local changes, and git pull fetches and merges changes from the remote repository.

IV. Mastering Git Workflows

A. Feature Branch Workflow: Streamlined Development and Integration

In this workflow, each new feature is developed in its own branch. Once the feature is complete, it is merged back into the main branch.

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

Gitflow is a branching model that defines a strict branching strategy by using different branches for different purposes (e.g., feature, develop, release, hotfix).

V. Advanced Git Techniques

A. Resolving Merge Conflicts: Handling Conflicting Changes

When changes from different branches conflict, Git requires manual intervention to resolve the conflicts.

git merge <branch-name>

Resolve conflicts in files

git add <resolved-file>

git commit -m "Resolved merge conflict"

B. Stashing Changes: Temporarily Shelving Uncommitted Work

git stash

git stash pop

git stash saves changes in a temporary area, and git stash pop reapplies them.

C. Using Tags: Annotating Specific Project Versions

git tag -a v1.0 -m "Version 1.0 release"

git push origin v1.0

II->>>>

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Todo App</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Todo List</h1>
  <input type="text" id="new-todo" placeholder="Add a new task">
  <button id="add-btn">Add</button>
  <ul id="todo-list"></ul>
  <script src="app.js"></script>
</body>
</html>
```

CSS->>

```
body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 20px;
  background-color: #f4f4f4;
}

h1 {
  color: #333;
}

input {
```

```
  padding: 10px;
  font-size: 16px;
}

button {
  padding: 10px;
  font-size: 16px;
  margin-left: 10px;
  cursor: pointer;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  padding: 10px;
  background: #fff;
  margin-bottom: 5px;
  border: 1px solid #ddd;
}

li.completed {
  text-decoration: line-through;
  color: #888;
}
```

JAVASCRIPT

```
document.getElementById('add-btn').addEventListener('click', function() {
  const newTodo = document.getElementById('new-todo').value;
  if (newTodo === '') return;
```

```
    const todoList = document.getElementById('todo-list');
    const todoItem = document.createElement('li');
    todoItem.textContent = newTodo;

    todoItem.addEventListener('click', function() {
        this.classList.toggle('completed');
    });

    todoList.appendChild(todoItem);
    document.getElementById('new-todo').value = '';
});
```

Version Control with Git

Initialize Git Repository

```
git init
```

Add Files to Staging Area

```
git add index.html style.css app.js
```

Commit Initial Version

```
git commit -m "Initial commit with basic Todo App structure"
```

Create a New Branch for Feature Development

```
git branch feature/add-complete-task
git checkout feature/add-complete-task
```

Implement Feature: Mark Task as Completed

```
# Add JavaScript code to mark tasks as completed
```

Stage and Commit Feature Changes

```
git add app.js
git commit -m "Add feature to mark tasks as completed"
```

Merge Feature Branch into Main Branch

```
git checkout main
git merge feature/add-complete-task
```

Push Changes to Remote Repository (if applicable)

git remote add origin <repository-url>

git push -u origin main