Starting with Git & GitHub: Course Material

This comprehensive course material is designed to teach Git and GitHub with practical examples, covering all requested modules and aligning with the specified learning outcomes. Each module includes explanations, commands, and hands-on exercises to ensure students can apply the concepts effectively.

Module 1: Introduction to Git

Overview

Git is a distributed version control system that tracks changes in source code during software development. It enables collaboration, version tracking, and rollback capabilities.

Key Concepts

- **Version Control**: Tracks changes to files, allowing multiple versions to coexist and enabling recovery of previous versions.
- **Benefits of Git**:
  - Distributed: Every developer has a full copy of the repository.
  - Fast: Most operations are local, reducing dependency on network access.
  - Flexible: Supports branching, merging, and collaborative workflows.

Practical Example

1. **Install Git**:
   - On Windows/Mac/Linux, download and install Git from git-scm.com.
   - Verify installation:
   - `git --version`
2. **Configure Git**:
   Set up your name and email for commit messages:
   - `git config --global user.name "Your Name"`
   - `git config --global user.email "your.email@example.com"`

Exercise

- Install Git on your system.
- Configure your Git username and email.
- Run `git --version` to confirm installation.

Module 2: What are Repositories?

Overview

A repository (repo) is a storage space where your project files and their history are stored. Git repositories can be local (on your machine) or remote (e.g., on GitHub).

Key Concepts

- **Local Repository**: A directory on your computer where Git tracks changes.
- **Remote Repository**: A hosted repository (e.g., on GitHub) for collaboration.
- **Structure**: Contains `.git` folder for version history, working directory for files, and staging area for changes.

Practical Example

1. **Initialize a Repository**:
   Create a new directory and initialize a Git repository:
   ```
   1. mkdir my-project
   2. cd my-project
   3. git init
   ```
2. **Clone a Repository**:
   Copy an existing repository from GitHub:
   ```
   1. git clone https://github.com/username/repository.git
   ```

Exercise

- Create a new folder called `test-repo` and initialize it as a Git repository.
- Clone a public repository from GitHub (e.g., `https://github.com/git-guides/git-test`).

Module 3: Basic Snapshotting

Overview

Snapshotting involves saving the state of your project at a specific point in time through commits. Git uses a three-stage process: working directory, staging area, and repository.

Key Concepts

- **Working Directory**: Your current project files.
- **Staging Area**: A temporary area to prepare changes for a commit.
- **Commit**: A snapshot of staged changes stored in the repository.

Practical Example

1. **Create and Track Files**:
   ```
   1. echo "# My Project" > README.md
   2. git add README.md
   ```
2. **Commit Changes**:
   ```
   1. git commit -m "Initial commit with README"
   ```
3. **Check Status**:
   View the state of your working directory and staging area:
   ```
   1. git status
   ```

Exercise

- Create a file `index.html` with basic HTML content.
- Stage and commit the file with a meaningful commit message.
- Check the status after committing.

Module 4: Branching and Merging

Overview

Branching allows you to work on different versions of a project simultaneously. Merging integrates changes from one branch into another.

Key Concepts

- **Branch**: A parallel version of the repository.
- **Merge**: Combines changes from multiple branches.
- **Merge Conflict**: Occurs when Git cannot automatically resolve differences.

Practical Example

1. **Create a Branch**:
   ```
   1. git branch feature-branch
   2. git checkout feature-branch
   ```

   Or combine into one command:

   ```
   git checkout -b feature-branch
   ```

2. **Make Changes and Commit**:
   ```
   1. echo "New feature" >> feature.txt
   2. git add feature.txt
   3. git commit -m "Add new feature"
   ```

3. **Merge the Branch**:
   Switch back to the main branch and merge:
   ```
   1. git checkout main
   2. git merge feature-branch
   ```
4. **Resolve a Merge Conflict**:
   If a conflict occurs, Git will pause the merge. Edit the conflicting files, mark them as resolved, and complete the merge:
   ```
   1. git add <conflicted-file>
   2. git commit
   ```

Exercise

- Create a branch called `add-footer`.
- Add a footer to `index.html` and commit the change.
- Merge `add-footer` into `main`.
- Intentionally create a merge conflict by editing the same line in `main` and `add-footer`, then resolve it.

Module 5: Sharing and Updating Projects

Overview

Git and GitHub enable collaboration by sharing repositories and syncing changes between local and remote repositories.

Key Concepts

- **Push**: Send local commits to a remote repository.
- **Pull**: Fetch and merge changes from a remote repository.
- **Fetch**: Download changes without merging.

Practical Example

1. **Create a Remote Repository**:
   - On GitHub, create a new repository.
   - Link your local repository:
   - `git remote add origin https://github.com/username/repository.git`
2. **Push Changes**:
   - `git push -u origin main`
3. **Pull Changes**:
   - `git pull origin main`

Exercise

- Create a GitHub repository and link it to your local `test-repo`.
- Push your local changes to GitHub.
- Make changes directly on GitHub, then pull them to your local repository.

Module 6: Inspecting and Comparing

Overview

Git provides tools to inspect the history of changes and compare different versions of files.

Key Concepts

- **Log**: View commit history.
- **Diff**: Compare changes between commits, branches, or files.
- **Show**: Display details of a specific commit.

Practical Example

1. **View Commit History**:
   ```
   1. git log --oneline
   ```
2. **Compare Changes**:
   Compare the working directory with the staging area:
   ```
   1. git diff
   ```

   Compare two branches:

   ```
   git diff main feature-branch
   ```

3. **Inspect a Commit**:
   ```
   1. git show <commit-hash>
   ```

Exercise

- Use `git log` to view the commit history of `test-repo`.
- Compare changes between two commits using `git diff <commit1> <commit2>`.
- Use `git show` to inspect the details of your latest commit.

Module 7: Stashing and Cleaning

Overview

Stashing temporarily saves changes, allowing you to switch contexts. Cleaning removes untracked files from the working directory.

Key Concepts

- **Stash**: Save uncommitted changes for later use.
- **Clean**: Remove untracked files and directories.

Practical Example

1. **Stash Changes**:
   Make changes to a file but don't commit:
   ```
   1. echo "Temporary change" >> temp.txt
   2. git stash
   ```

   Apply the stashed changes later:

   ```
   git stash apply
   ```

2. **Clean Untracked Files**:
   Remove untracked files:
   ```
   1. git clean -f
   ```

Exercise

- Make changes to a file in `test-repo` and stash them.
- Switch to another branch, then apply the stashed changes.
- Create an untracked file and use `git clean` to remove it.

Module 8: Advanced Tools

Overview

Advanced Git tools like rebase and cherry-pick provide powerful ways to manage history and troubleshoot issues.

Key Concepts

- **Rebase**: Reapply commits on a different base, creating a linear history.
- **Cherry-pick**: Apply specific commits from one branch to another.
- **Bisect**: Find the commit that introduced a bug.

Practical Example

1. **Rebase a Branch**:
   ```
   1. git checkout feature-branch
   2. git rebase main
   ```
2. **Cherry-pick a Commit**:
   ```
   1. git cherry-pick <commit-hash>
   ```
3. **Use Bisect**:
   Start bisecting:
   ```
   1. git bisect start
   2. git bisect bad
   3. git bisect good <commit-hash>
   ```

   End bisecting:

   ```
   git bisect reset
   ```

Exercise

- Create a branch with multiple commits, then rebase it onto `main`.
- Cherry-pick a commit from one branch to another.
- Simulate a bug and use `git bisect` to find the problematic commit.

Module 9: Tagging

Overview

Tags mark specific points in a repository's history, often used for releases.

Key Concepts

- **Lightweight Tags**: Simple pointers to a commit.
- **Annotated Tags**: Include metadata like tagger name and message.
- **Tagging Releases**: Common for marking version numbers (e.g., `v1.0.0`).

Practical Example

1. **Create an Annotated Tag**:
   ```
   1. git tag -a v1.0.0 -m "Version 1.0.0 release"
   ```
2. **List Tags**:
   ```
   1. git tag
   ```
3. **Push Tags to Remote**:
   ```
   1. git push origin v1.0.0
   ```
4. **Delete a Tag**:
   ```
   1. git tag -d v1.0.0
   2. git push origin --delete v1.0.0
   ```

Exercise

- Create an annotated tag `v0.1.0` for a commit in `test-repo`.
- Push the tag to GitHub.
- Delete the tag locally and remotely.

Module 10: Course Conclusion

Overview

This module ties together all concepts through a practical project and reflection on the importance of version control.

Practical Example: Collaborative Project

1. **Set Up a Collaborative Project**:
    - Create a GitHub repository and invite a collaborator.
    - Initialize a local repository and push initial files.
    - Each collaborator creates a branch, makes changes, and submits a pull request.
    - Review and merge pull requests on GitHub.
    - Resolve any merge conflicts collaboratively.
2. **Reflect**:
    - Discuss how Git enables version control and collaboration.
    - Highlight the importance of clear commit messages, branching strategies, and tagging for releases.

Exercise

- Work in pairs to create a simple project (e.g., a markdown documentation site).
- Each person creates a branch, adds content, and submits a pull request.
- Merge the pull requests and tag the final version as `v1.0.0`.
- Write a short reflection on how Git and GitHub improved your workflow.

Learning Outcomes Recap

By completing this course, students will:

1. Understand the importance and benefits of version control with Git.
2. Navigate and manage Git repositories locally and remotely.
3. Perform snapshotting with meaningful commits.
4. Utilize branching and merging to manage parallel development.
5. Share and update projects via GitHub.
6. Inspect and compare changes using Git's history tools.
7. Manage changes with stashing and cleaning.
8. Employ advanced tools like rebase and bisect.

9. Implement tagging for release management.
10. Apply Git and GitHub in real-world collaborative scenarios.

This course equips students with the skills to use Git and GitHub effectively in software development and collaborative projects.