Git and GitHub: A Practical Guide

This document provides an explanation of Git and GitHub, along with practical code examples for common operations.

What is Git?

Git is a free and open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on the same project simultaneously without overwriting each other's changes.

Key Concepts of Git

- **Repository (Repo):** A directory containing all the files of your project, along with the entire history of changes.
- **Commit:** A snapshot of your repository at a specific point in time. Each commit has a unique ID and a message describing the changes.
- **Branch:** A parallel version of your repository that allows you to work on new features or bug fixes independently without affecting the main codebase.
- **Merge:** The process of combining changes from one branch into another.
- **Clone:** Creating a local copy of a remote repository.
- **Pull:** Fetching changes from a remote repository and merging them into your current local branch.
- **Push:** Uploading your local commits to a remote repository.

What is GitHub?

GitHub is a web-based platform that uses Git for version control. It provides a centralized location for developers to store, manage, and collaborate on their

code. GitHub offers a user-friendly interface, issue tracking, project management tools, and more.

Practical Git Code Examples

Here are some fundamental Git commands you'll use regularly.

1. Initializing a new repository

To start tracking changes in a new project, navigate to your project directory in the terminal and run:

```
None
git init
```

This command creates a hidden `.git` directory, which is where Git stores all its tracking information.

2. Checking the status of your repository

To see which files have been modified, staged, or are untracked:

```
None
git status
```

3. Adding files to the staging area

Before committing changes, you need to "stage" them. This tells Git which changes you want to include in your next commit.

To stage a specific file:

```
None
git add <filename>
```

To stage all changes in the current directory:

```
None
git add .
```

4. Committing your changes

Once files are staged, you can commit them with a descriptive message:

```
None
git commit -m "Your descriptive commit message here"
```

For a more detailed commit message, omit `-m`:

```
None
git commit
```

This will open your default text editor to write the commit message.

5. Viewing commit history

To see a log of all commits in the current branch:

```
None
git log
```

To view a more concise log:

```
None
git log --oneline
```

6. Creating a new branch

To create a new branch named `feature-x`:

```
None
git branch feature-x
```

7. Switching between branches

To switch to the `feature-x` branch:

```
None
git checkout feature-x
```

You can also create and switch to a new branch in one command:

```
None
git checkout -b new-feature-branch
```

8. Merging branches

To merge changes from `feature-x` into your current branch (e.g., `main`):

First, switch to the `main` branch:

```
None
git checkout main
```

Then, merge `feature-x`:

```
None
git merge feature-x
```

9. Cloning a remote repository

To get a local copy of a repository from GitHub:

```
None
git clone <repository-url>
```

For example:

```
None
git clone https://github.com/user/repository.git
```

10. Pushing changes to a remote repository

After committing changes locally, push them to GitHub:

```
None
git push origin <branch-name>
```

For the first push of a new branch, you might use:

```
None
git push -u origin <branch-name>
```

This sets the upstream branch, so future `git push` commands don't require the branch name.

11. Pulling changes from a remote repository

To get the latest changes from the remote repository and merge them into your local branch:

```
None
git pull origin <branch-name>
```

Often, when on your main branch, you can simply use:

```
None
git pull
```

GitHub Implementation Workflow

Here's a typical workflow when working with GitHub:

1. **Create a new repository on GitHub:**
    - Go to GitHub.com and click "New repository."
    - Give it a name, description, and choose public/private.
    - You can optionally add a README, .gitignore, and license.
2. **Clone the repository to your local machine:**

```
None
git clone <repository-url>
```

3. **Make changes to your code:**
    - Edit existing files, add new ones, etc.
4. **Stage your changes:**

```
None
git add .
```

5. **Commit your changes:**

```
None
git commit -m "Meaningful commit message"
```

6. **Push your changes to GitHub:**

```
None
git push origin main # Or your current branch name
```

7. **Create a new branch for new features or bug fixes (optional but recommended):**

```
None
git checkout -b feature/new-feature
```

    - Make your changes, commit, and push this new branch.
8. **Create a Pull Request (PR) on GitHub:**
    - After pushing a new branch, GitHub will often prompt you to create a PR.
    - A PR is a request to merge your changes from your branch into another (e.g., `main`).
    - Others can review your code, provide feedback, and suggest changes.
9. **Review and Merge the Pull Request:**
    - Once the code is reviewed and approved, the PR can be merged into the target branch (e.g., `main`).
10. **Pull the latest changes to your local main branch:**
    - After merging, switch back to your `main` branch locally and pull the changes:

```
None
git checkout main
git pull origin main
```

Conclusion

Mastering Git and GitHub is essential for any software developer. Consistent practice is key to becoming proficient.

How Software Developers Write and Practice Code with Git and GitHub

Software developers utilize Git and GitHub in a structured manner to ensure code quality, collaboration, and efficient project management.

1. **Understand the Feature/Bug:** Before writing any code, a developer thoroughly understands the new feature to implement or the bug to fix. This often involves reviewing requirements, design documents, or bug reports.

2. **Branching Out:** Instead of directly modifying the `main` (or `master`) branch, developers create a new, dedicated branch for each new feature or bug fix. This isolates their changes and prevents conflicts with other developers' work.

```
None
git checkout -b feature/my-new-feature
```

3. **Iterative Development and Local Commits:** As they write code, developers make small, logical changes and commit them frequently to their local branch. Each commit represents a single, cohesive change and includes a clear, descriptive message.

```
None
git add .
git commit -m "Implement user authentication logic"
```

4. **Regularly Pulling from Main:** To stay updated with the latest changes from the main codebase and avoid significant merge conflicts later, developers regularly pull changes from the `main` branch into their local development branch.

```
None
git pull origin main
```

5. **Testing Locally:** Before pushing changes, developers thoroughly test their code locally to ensure it works as expected and doesn't introduce new bugs.
6. **Pushing to Remote:** Once a set of changes is complete and tested, the developer pushes their local branch to the remote GitHub repository.

```
None
git push origin feature/my-new-feature
```

7. **Creating a Pull Request (PR):** On GitHub, the developer then creates a Pull Request (PR) from their feature branch to the `main` branch. The PR serves as a discussion forum for the changes.
    - The PR description should clearly explain what the changes do, why they were made, and any relevant context (e.g., linked issues).
    - Code reviewers (other developers) examine the code, provide feedback, suggest improvements, and ensure it adheres to coding standards.
8. **Addressing Feedback and Iterating on PR:** Developers actively respond to comments and suggestions on their PR. They might make additional commits to their feature branch based on the feedback and push those changes.
9. **Merging the PR:** Once the PR is approved by the required number of reviewers, it can be merged into the `main` branch. This integrates the new feature or bug fix into the primary codebase.
10. **Cleaning Up:** After the PR is merged, the developer typically deletes the feature branch both locally and on the remote to keep the repository clean.

```
None
git branch -d feature/my-new-feature
git push origin --delete feature/my-new-feature
```

11. **Pulling Merged Changes to Local Main:** Finally, the developer switches back to their local `main` branch and pulls the newly merged changes to ensure their local `main` branch is up-to-date.

```
None
git checkout main
git pull origin main
```

How to Practice Git and GitHub

Consistent practice is vital for building muscle memory and understanding the nuances of Git and GitHub.

1. **Start Small and Simple:**
    - Create a new local directory.

    - Initialize a Git repository (`git init`).
    - Create a few simple text files (`file1.txt`, `file2.txt`).
    - Practice `git add`, `git commit` with various messages.
    - Experiment with `git status` and `git log`.
2. **Practice Branching and Merging:**
    - Create multiple branches (`git branch`, `git checkout`).
    - Make changes on different branches.
    - Practice merging them back into `main` (`git merge`).
    - Intentionally create a merge conflict and learn how to resolve it.
3. **Work with a Remote Repository (GitHub):**
    - Create a new public or private repository on GitHub.
    - Clone it to your local machine (`git clone`).
    - Make local changes, commit them, and `git push` them to GitHub.
    - Make a change directly on GitHub (e.g., edit a README file) and then `git pull` those changes to your local machine.
4. **Collaborate (Even with Yourself):**
    - If you don't have a team, simulate collaboration by working on the same repository from two different local directories (or by creating a second GitHub account and having it contribute).
    - Practice the PR workflow: create a branch, push it, create a PR, review it, and merge it.
5. **Explore Advanced Commands:**
    - `git revert`: Undoing a commit.
    - `git reset`: Moving the HEAD pointer.
    - `git stash`: Temporarily saving changes.
    - `git rebase`: Rewriting commit history (use with caution, especially on shared branches).
    - `git remote`: Managing remote repositories.
6. **Use Real-World Scenarios:**
    - Start a small personal project and manage its version control entirely with Git and GitHub.
    - Contribute to an open-source project (start with simple bug fixes or documentation improvements).
7. **Read the Documentation:**
    - The official Git documentation is comprehensive. Refer to it when you encounter unfamiliar commands or concepts.
    - Many online tutorials and courses offer practical exercises.
8. **Solve Problems:**

    - When you run into a Git issue, don't just copy-paste solutions. Try to understand *why* the issue occurred and *how* the solution fixes it. This builds a deeper understanding.

Git and GitHub Learning Roadmap for Software Freshers

This roadmap outlines a structured approach for software freshers to learn and master Git and GitHub, progressing from fundamental concepts to practical application and advanced topics.

Phase 1: Fundamentals of Version Control (Weeks 1-2)

Week 1: Understanding Git Basics

- **Content:**
    - What is Version Control? (Centralized vs. Distributed)
    - Why Git? (Speed, efficiency, branching model)
    - Installing Git
    - Configuring Git (username, email)
    - Key Git Concepts: Repository, Commit, Branch, Merge, Clone, Pull, Push
- **Practical Exercises:**
    - Initialize a new local repository (`git init`).
    - Create and modify text files.
    - Practice `git status`, `git add`, `git commit -m`.
    - View commit history (`git log`, `git log --oneline`).
    - Create, switch, and delete local branches (`git branch`, `git checkout`).
    - Perform basic merges of local branches.

Week 2: Introduction to GitHub and Remote Repositories

- **Content:**
    - What is GitHub? (Platform for collaboration, hosting)
    - Creating a GitHub account and new repositories.
    - Cloning remote repositories (`git clone`).
    - Connecting local repositories to remote ones (`git remote add origin`).

    - Pushing local commits to GitHub (`git push`).
    - Pulling changes from GitHub (`git pull`).
    - Understanding `origin` and `main`/`master` branches.
- **Practical Exercises:**
    - Create a new repository on GitHub.
    - Clone it to your local machine.
    - Make changes, commit, and push them to GitHub.
    - Make a small change directly on GitHub (e.g., edit README.md) and pull it locally.
    - Simulate a simple collaboration: make changes on two different local clones of the same repository and push/pull to resolve.

Phase 2: Collaborative Workflow and Best Practices (Weeks 3-4)

Week 3: Branching Strategies and Pull Requests

- **Content:**
    - Feature Branch Workflow (common industry practice).
    - The importance of small, focused branches.
    - Creating effective commit messages.
    - Understanding Pull Requests (PRs): purpose, lifecycle, review process.
    - Resolving merge conflicts manually.
- **Practical Exercises:**
    - Implement a small feature using a new branch.
    - Push the feature branch to GitHub.
    - Create a Pull Request on GitHub.
    - Practice resolving merge conflicts that arise from concurrent work on different branches.
    - Add comments to a simulated PR.

Week 4: Advanced Git Commands and Collaboration Etiquette

- **Content:**
    - Undoing changes: `git revert`, `git reset` (soft, mixed, hard - **caution advised**).
    - Stashing changes: `git stash`.
    - Comparing changes: `git diff`.
    - Ignoring files: `.gitignore`.
    - Forking and contributing to open-source projects.

    - GitHub Issues and Project Boards for task management.
- **Practical Exercises:**
    - Experiment with `git revert` to undo a previous commit.
    - Use `git stash` to temporarily save changes.
    - Create and use a `.gitignore` file.
    - Fork a simple open-source project on GitHub and attempt to create a small PR (e.g., fixing a typo in documentation).
    - Explore GitHub Issues and project boards for a sample repository.

Phase 3: Integration and Continuous Learning (Ongoing)

Week 5 onwards: Continuous Practice and Real-World Application

- **Content:**
    - Advanced branching models (GitFlow - understand, but don't overcomplicate initially).
    - Rebasing: `git rebase` (use with extreme caution, especially on shared branches).
    - Squashing commits.
    - Git hooks (optional, for automation).
    - Understanding CI/CD integration with GitHub Actions (basic concepts).
- **Practical Exercises:**
    - **Personal Project:** Start a personal coding project and use Git/GitHub from day one to manage all its versions.
    - **Open Source Contributions:** Actively look for simple open-source issues (e.g., "good first issue" tags) to contribute to.
    - **Team Projects:** If possible, participate in a team project where Git/GitHub is used extensively.
    - **Mock Interviews:** Practice explaining Git concepts and workflows.
    - **Regularly Review:** Revisit challenging Git concepts and commands.
    - **Read Documentation:** Refer to official Git and GitHub documentation for deeper understanding.

Resources for Learning:

- **Official Git Documentation:** https://git-scm.com/doc
- **GitHub Guides:** https://guides.github.com/
- **Pro Git Book:** https://git-scm.com/book/en/v2 (Free online book)
- **Online Tutorials/Courses:** (e.g., freeCodeCamp, Udemy, Coursera, YouTube tutorials)

- **Interactive Git Learning Tools:** (e.g., `learngitbranching.js.org`)

Key Takeaways for Freshers:

- **Practice is Paramount:** The only way to truly learn Git and GitHub is by consistently using them.
- **Understand, Don't Just Memorize:** Focus on understanding *why* you use certain commands and *what* they do.
- **Start Simple:** Don't try to master everything at once. Build a strong foundation first.
- **Don't Be Afraid to Make Mistakes:** Git is designed to help you recover from errors. Use them as learning opportunities.
- **Collaborate:** Git and GitHub shine in collaborative environments. Work with others or simulate it to learn the workflow.

Sure, here is the training material for software freshers, along with examples:

Git and GitHub for Software Freshers: Training Material with Examples

This training material is designed to provide software freshers with a hands-on introduction to Git and GitHub. It combines theoretical explanations with practical, executable code examples, building upon the concepts outlined in the "Git and GitHub Learning Roadmap."

Module 1: Git Fundamentals - Your Local Version Control

Objective: Understand basic Git concepts and commands for managing local project history.

Duration: Approximately 3-4 hours of active practice.

1. Setting Up Git

Before you begin, ensure Git is installed on your system.

- **Installation:**
  - **Windows:** Download from git-scm.com and follow the installer.
  - **macOS:** Install with Homebrew (`brew install git`) or Xcode Command Line Tools (`xcode-select --install`).
  - **Linux (Debian/Ubuntu):** `sudo apt-get install git`
- **Configuration:** After installation, configure your user name and email. This information will be attached to your commits.

```
None
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Self-Check: To verify your configuration, run:

```
None
git config --list
```

2. Creating and Initializing a Repository

A Git repository (repo) is where Git stores all the project files and their history.

- **Creating a Project Directory:**

```
None
mkdir my-first-git-project
cd my-first-git-project
```

- **Initializing Git:** This command turns your directory into a Git repository by creating a hidden `.git` folder.

```
None
git init
```

Output: `Initialized empty Git repository in /path/to/my-first-git-project/.git/`

3. The Git Workflow: Working Directory, Staging Area, and Repository

Understand the three states of files in Git:

- **Working Directory:** Your current files, where you make changes.
- **Staging Area (Index):** A temporary area where you prepare changes for your next commit.

- **Git Repository:** Where Git permanently stores committed snapshots.

4. Tracking Changes: `git status`, `git add`, `git commit`

- **`git status`:** Shows the state of your working directory and staging area.

```
None
git status
```

Example:

a. Create a file: `touch index.html`
b. Run `git status`
Output: `Untracked files: index.html` (Git sees it but isn't tracking it yet)

- **`git add <filename>` or `git add .`:** Adds changes from the working directory to the staging area.

Example:

a. `git add index.html`
b. Run `git status`
Output: `Changes to be committed: new file: index.html` (The file is now staged)

Example (add all changes):

a. Create `style.css`
b. Run `git add .` (stages `style.css` and `index.html` if `index.html` was unstaged)
c. Run `git status`

- **`git commit -m "Your descriptive message"`:** Takes a snapshot of the staged changes and saves them permanently in the repository. The message is crucial for understanding the commit's purpose.

Example:

a. `git commit -m "Initial commit: Added basic HTML structure"`
b. Run `git status`
   Output: `nothing to commit, working tree clean`

Practice:

a. Modify `index.html` (e.g., add `<h1>Hello Git!</h1>`).
b. Run `git status` (shows `modified: index.html`).
c. `git add index.html`
d. `git commit -m "Add H1 heading to index.html"`

5. Viewing History: `git log`

- **`git log`:** Displays a list of all commits in the current branch, showing author, date, and commit message.

```
None
git log
```

- **`git log --oneline`:** Provides a more concise view, showing a shortened commit ID and the commit message on one line.

```
None
git log --oneline
```

6. Understanding Branches: `git branch`, `git checkout`

Branches allow you to work on new features or bug fixes without affecting the main codebase.

- **`git branch`:** Lists existing branches. An asterisk indicates your current branch.

```
None
git branch
```

Output: `* main` (or `master` if your Git version is older)

- **`git branch <branch-name>`:** Creates a new branch.

```
None
git branch feature/contact-form
```

Self-Check: Run `git branch` again. You'll see `feature/contact-form` listed, but you're still on `main`.

- **`git checkout <branch-name>`:** Switches to an existing branch.

```
None
git checkout feature/contact-form
```

Output: `Switched to branch 'feature/contact-form'`

- **`git checkout -b <new-branch-name>`:** Creates a new branch AND switches to it in one command.

```
None
git checkout -b feature/about-page
```

Output: `Switched to a new branch 'feature/about-page'`

7. Merging Branches: `git merge`

Merging combines changes from one branch into another.

- **Scenario:** You developed a new feature on `feature/contact-form` and now want to integrate it into `main`.

Example:

a. Switch to `feature/contact-form` (if not already there): `git checkout feature/contact-form`
b. Create a new file: `touch contact.html`
c. Add content to `contact.html` (e.g., `<h2>Contact Us</h2>`).
d. Stage and commit:

```
None
git add contact.html
git commit -m "Add contact page structure"
```

e. Switch back to `main`: `git checkout main`
Observe: `contact.html` is no longer in your directory, as it only exists on the `feature/contact-form` branch.
f. Merge `feature/contact-form` into `main`:

```
None
git merge feature/contact-form
```

Output: Git will show a "Fast-forward" merge if `main` hasn't diverged, or it will create a merge commit.

g. *Self-Check:* `contact.html` should now be present in your `main` branch.
h. Clean up: Delete the merged branch (optional, but good practice):

```
None
git branch -d feature/contact-form
```

Output: `Deleted branch feature/contact-form (was <commit-hash>).`

Module 1 Practice Challenge:

1. Create a new directory called `my-website`.
2. Initialize a Git repository inside `my-website`.

3. Create `index.html` with a basic HTML structure and commit it (`"Initial website structure"`).
4. Create a new branch called `dev/add-styling`.
5. Switch to `dev/add-styling`.
6. Create `style.css` and add some CSS rules (e.g., `body { font-family: sans-serif; }`).
7. Commit `style.css` (`"Add basic styling for font"`).
8. Switch back to `main`.
9. Create another new branch called `dev/add-script`.
10. Switch to `dev/add-script`.
11. Create `script.js` and add a simple `console.log("Hello from script!");`.
12. Commit `script.js` (`"Add simple script file"`).
13. Merge `dev/add-styling` into `main`. (You'll need to go to `main` first, then merge).
14. Merge `dev/add-script` into `main`.
15. Use `git log --oneline` to view the commit history.
16. Delete the `dev/add-styling` and `dev/add-script` branches.

Module 2: Introduction to GitHub - Remote Collaboration

Objective: Understand how to use GitHub to store your repositories remotely and collaborate.

Duration: Approximately 2-3 hours of active practice.

**1. What is GitHub?GitHub is a web-based platform that uses Git for version control. It provides a centralized location for developers to store, manage, and collaborate on their codebases. Think of it as a social network for code, allowing you to:

- **Host Repositories:** Store your Git repositories online, making them accessible from anywhere.
- **Collaborate:** Work seamlessly with others, track changes, and manage contributions.
- **Manage Projects:** Utilize features like issue tracking, project boards, and wikis.
- **Showcase Work:** Share your projects with the world and build a portfolio.

2. Creating a GitHub Repository

Before you can push your local Git repository to GitHub, you need a place for it to live on the remote server.

- **Steps:**
    a. Go to github.com and sign in to your account.
    b. Click the + icon in the top right corner and select "New repository."
    c. **Repository Name:** Choose a descriptive name (e.g., `my-first-git-project`).
    d. **Description (Optional):** Briefly explain what your project is about.
    e. **Public or Private:**
        - **Public:** Anyone can see this repository. Ideal for open-source projects or portfolios.
        - **Private:** Only you and people you explicitly share with can see it.
    f. **Initialize this repository with a README:** *For existing local projects, leave this unchecked.* If you're starting a new project directly on GitHub, checking this will create a README.md file for you.
    g. Click "Create repository."

After creation, GitHub will provide you with instructions, including the repository's URL. It will look something like `https://github.com/your-username/my-first-git-project.git`.

3. Connecting Your Local Repository to GitHub

Now you need to tell your local Git repository where its remote counterpart lives.

- **Adding a Remote:**
  Navigate to your local `my-first-git-project` directory in your terminal.

```
None
git remote add origin
https://github.com/your-username/my-first-git-project.git
```

    - `git remote add`: Adds a new remote repository.

    - `origin`: This is the default name given to the primary remote repository. You can choose any name, but `origin` is standard.
    - `https://github.com/your-username/my-first-git-project.git`: This is the URL of the repository you just created on GitHub. Replace `your-username` with your actual GitHub username.
- *Self-Check:* To verify that the remote has been added correctly:

```
None
git remote -v
```

Output:

```
None
origin  https://github.com/your-username/my-first-git-project.git (fetch)
origin  https://github.com/your-username/my-first-git-project.git (push)
```

4. Pushing Your Local Commits to GitHub: `git push`

Once your local repository knows about its remote, you can push your local commits to GitHub.

- **Basic Push:**

```
None
git push origin main
```

    - `git push`: Sends your committed changes to the remote repository.
    - `origin`: Specifies the remote repository (the one named 'origin').
    - `main`: Specifies the local branch you want to push (e.g., `main` or `master`).
- **First Push for a New Branch (`-u` or `--set-upstream`):**
  When you push a new local branch for the first time, you'll need to set its upstream tracking branch. This links your local branch to a remote branch, so future `git push` and `git pull` commands are simpler.

```
None
git push -u origin main
```

    - `-u` (or `--set-upstream`): Sets the upstream reference. After this, you can often just type `git push` for that branch.
- *Example:*
    a. Ensure you have some commits on your local `main` branch.
    b. Create a new repository on GitHub (e.g., `my-remote-repo`).
    c. Connect your local `my-first-git-project` to it:

```
None
cd my-first-git-project
git remote add origin https://github.com/your-username/my-remote-repo.git
```

    d. Push your `main` branch:

```
None
git push -u origin main
```

    You may be prompted for your GitHub username and Personal Access Token (PAT).

Self-Check: Go to your GitHub repository in your web browser. You should now see your files and commit history.

5. Pulling Changes from GitHub: `git pull`

`git pull` is used to fetch the latest changes from the remote repository and integrate them into your current local branch. It's essentially a `git fetch` followed by a `git merge`.

- **Scenario:** A colleague (or you, directly on GitHub) makes a change to the remote repository, and you want to get that change on your local machine.
- *Example:*
    a. Go to your `my-remote-repo` on GitHub.
    b. Click on `index.html` and then the "Edit this file" (pencil) icon.

c. Add a new line, e.g., `<p>This line was added on GitHub.</p>`.
d. Commit the changes directly on GitHub.
e. Back in your local terminal (make sure you are on the `main` branch in `my-first-git-project`):

```
None
git pull origin main
```

Or, if you set the upstream using `git push -u`:

```
None
git pull
```

Output: Git will show the changes that were pulled and merged.
Self-Check: Open `index.html` in your local directory. You should see the new line.

6. Cloning an Existing Remote Repository: `git clone`

If a project already exists on GitHub, `git clone` is how you get a complete local copy of it, including all its files and commit history.

- **Command:**

```
None
git clone <repository-url>
```

  - `<repository-url>`: This is the HTTPS or SSH URL of the GitHub repository. You can find this by clicking the "Code" button on the GitHub repository page.
- *Example:*
  a. Go to your `my-remote-repo` on GitHub.
  b. Click the green "Code" button and copy the HTTPS URL.

c. In your terminal, navigate to a directory *outside* `my-first-git-project` (e.g., your Desktop or a `projects` folder).
d.

```
None
git clone https://github.com/your-username/my-remote-repo.git
```

Output: Git will download the repository into a new folder named `my-remote-repo` (or whatever the repository name is).
Self-Check: `cd my-remote-repo` and then `ls` (or `dir` on Windows) to see the files. Also, `git log` will show the history.

Module 2 Practice Challenge:

1. Create a new, empty repository on GitHub called `my-online-portfolio`. *Do NOT initialize with a README.*
2. Go back to your `my-website` local directory from Module 1.
3. Add the new GitHub repository as a remote named `origin`.
4. Push your `main` branch from `my-website` to `my-online-portfolio` on GitHub, setting the upstream.
5. On GitHub, manually add a file named `LICENSE` to `my-online-portfolio` (e.g., choose an MIT license).
6. Back in your local `my-website` directory, pull the changes from GitHub.
7. Verify that the `LICENSE` file is now present locally.
8. Clone `my-online-portfolio` to a *new* directory on your computer (e.g., `my-portfolio-copy`).
9. Make a small change in `my-portfolio-copy` (e.g., edit `index.html`), commit it, and push it.
10. Go back to your original `my-website` directory and pull the latest changes. Observe how changes made from a different clone are now synchronized.