**Learning outcome 1: Setup repository**

**Indicative content 1.1: Definition of general key terms**

1. **Version control**

Version control, also known as source control, is **the practice of tracking and managing changes to software code.**

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter.

Before we dive into setting up version control, let me explain some common terms and commands used in version control so you understand what everything means!

* **Repo**, short for **Repository**: it’s a storage space where you project or code resides in, it’s like a folder for your code! There are two types of repos, local and remote.
  + **Local Repo**: it is going to be on the machine (laptop or computer) where your code is stored at. Locally storing it means only you have access to the code and if your computer crashes or you lose that code, then it would be pretty hard to get it back.
  + **Remote Repo**: it is going to be hosted on a website, like Github, for example, and your code here is accessible to not just you, but your whole team! If you lose your code on your computer, your code is still safe on here! This is akin to iCloud storage.
* **Committing**code: saving code locally on your computer, to your local repo.
* **Pushing** code: saving your committed code onto some hosted platform like Github, to your remote repo.
* **Pulling** code: downloading updated code from your remote repo to your local repo (machine).
* **Branch**: it’s a like folder inside your repo, where you can organize what features are in which folder. Say, you and your teammate are working on an app together and you are working on feature 1 and they are working on feature 2. You can just have two different branches, feature1 and feature2 to keep your code more organized and prevent bugs!
* **Merge**: when you use multiple branches, sometimes you want to combine the code in both branches, that is called a merge.

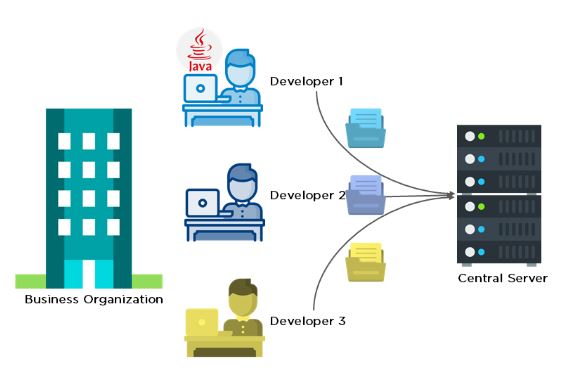
1. **Git**

**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.

[Git](https://www.simplilearn.com/tutorials/git-tutorial/git-tutorial-for-beginner) is a [DevOps tool](https://www.simplilearn.com/tutorials/devops-tutorial/devops-tools) used for source code management. It is a free and open-source version control system used to handle small to very large projects efficiently. Git is used to tracking changes in the source code, enabling multiple developers to work together on non-linear development.

Before diving deep, let’s explain a scenario before Git:

* Developers used to submit their codes to the central server without having copies of their own
* Any changes made to the source code were unknown to the other developers
* There was no communication between any of the developers



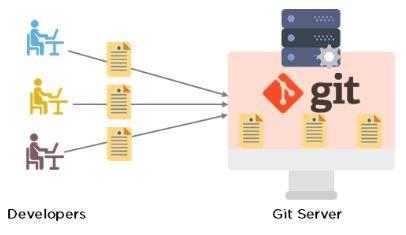
Now let’s look at the scenario after Git:

* Every developer has an entire copy of the code on their local systems
* Any changes made to the source code can be tracked by others
* There is regular communication between the developers



**Features of Git**

* Tracks history
* Free and open source
* Supports non-linear development
* Creates backups
* Scalable
* Supports collaboration
* Branching is easier
* Distributed development



1. **GitHub**

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

At a high level, GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code. To understand exactly what GitHub is, you need to know two connected principles:

* Version control
* Git

1. **Terminal**

This is very much like the Terminal. It is a text-based interface that exists in every operating system as a program and is usually integrated in coding programs as well. By typing commands in the Terminal, you can access files, install programs, modify programs, get information about various things (for example, check things like package loss and speed of your internet) and do more or less everything else as well.

The quickest way to install, remove or update applications is through the Terminal. The Terminal is also referred to as the shell, command line, prompt, or command prompt.

**Why the terminal?**

The Terminal is a great tool for communication - and by that, I don’t mean for chatting with your best friend, but rather to access the internal CPU and other networks and computers. In coding, it is used primarily for:

* Installing specific packages, updates, export and import your work to platforms like GitHub
* Code testing
* Error display

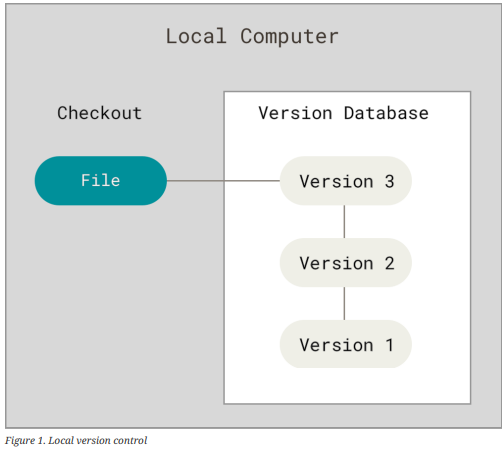
In short, it is a perfect way to be able to find information, manage installations and communicate with external services.

**1.2. Introduction to version control**

**1. Types of version control**

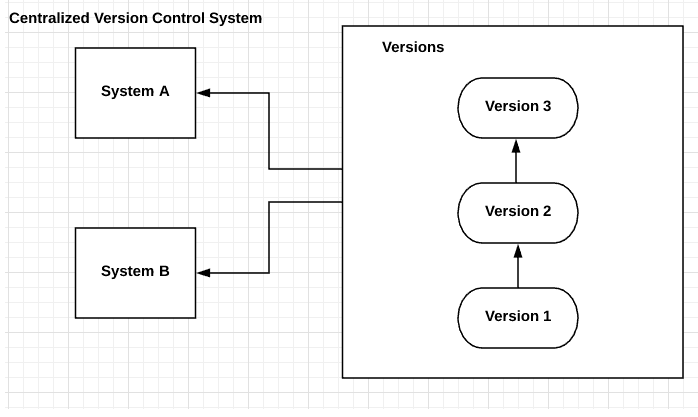
The various types of the version control systems are:  
1. Local Version Control System  
2. Centralized Version Control System  
3. Distributed Version Control System

### **Local Version Control System:**



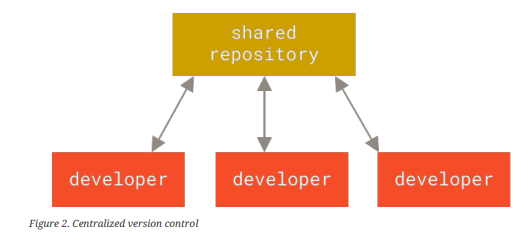
Local version control system maintains track of files within the local system. This approach is very common and simple. This type is also error prone(disposed to) which means the chances of accidentally writing to the wrong file is higher.

### **2. Centralized Version Control Systems**



In this approach, all the changes in the files are tracked under the centralized server. The centralized server includes all the information of versioned files, and list of clients that check out files from that central place.  
**Example: Tortoise SVN**

### **Distributed Version Control System:**

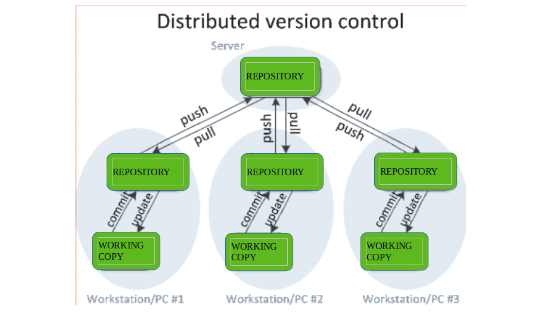


Distributed version control systems come into picture to overcome the drawback of centralized version control system. The clients completely clone the repository including its full history. If any server dies, any of the client repositories can be copied on to the server which help restore the server.

Every clone is considered as a full backup of all the data.

**Example: Git**

The **benefit** of CVCS (Centralized Version Control Systems) makes collaboration amongst developers along with providing an insight to a certain extent on what everyone else is doing on the project. It allows administrators to fine-grained control over who can do what.



**Purpose of Version Control:**

* Multiple people can work simultaneously on a single project. Everyone works on and edits their own copy of the files and it is up to them when they wish to share the changes made by them with the rest of the team.
* It also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
* It integrates the work that is done simultaneously by different members of the team. In some rare cases, when conflicting edits are made by two people to the same line of a file, then human assistance is requested by the version control system in deciding what should be done.
* Version control provides access to the historical versions of a project. This is insurance against computer crashes or data loss. If any mistake is made, you can easily roll back to a previous version. It is also possible to undo specific edits that too without losing the work done in the meanwhile. It can be easily known when, why, and by whom any part of a file was edited.
  + **Well known version control version system**

## What are the main version control systems?

The three most well-known version control tools are Git, Subversion, and Mercurial.

### **Git**

Git is the most popular option and has become synonymous with "source code management." Git is an open source distributed system that is used for software projects of any size, making it a popular option for startups, enterprise, and everything in between.

### **Subversion**

SVN is a widely adopted centralized VCS. This system keeps all of a project's files on a single codeline making it impossible to branch, so it's easy to scale for large projects. It's simple to learn and features folder security measures, so access to subfolders can be restricted.

### **Mercurial**

Mercurial is a distributed VCS that offers simple branching and merging capabilities. The system enables rapid scaling and collaborative development, with an intuitive interface. The flexible command line interface enables users to begin using the system immediately.

# Concurrent Versions System (CVS)

**CVS also is an abbreviation for *Computer Vision Syndrome*.**

Concurrent Versions System (CVS) is a program that lets a code developer save and retrieve different development versions of [source code](https://www.techtarget.com/searchapparchitecture/definition/source-code). It also lets a team of developers share control of different versions of files in a common repository of files.

## Benefits of version control

### **Quality**

Teams can review, comment, and improve each other’s code and assets.

### **Acceleration**

Branch code, make changes, and merge commits faster.

### **Visibility**

Understand and spark team collaboration to foster greater release build and release patterns.

**✔ Application of version control**

Version control systems (VCS) like Git have a wide range of applications across various fields and types of work. Here are some key applications of version control:

### 1. **Software Development**

* **Code Management**: Track changes to source code, allowing multiple developers to collaborate without conflicts.
* **Branching and Merging**: Enable feature development, bug fixes, and experimentation without affecting the main codebase.
* **Release Management**: Tag versions for releases, ensuring that specific code states can be referenced later.

### 2. **Document Collaboration**

* **Changes Tracking**: Keep track of revisions made to documents, allowing users to see who made changes and when.
* **Version History**: Restore previous versions of documents if needed, aiding in recovery from mistakes or unwanted changes.

### 3. **Web Development**

* **Asset Management**: Manage HTML, CSS, JavaScript, and other assets, allowing teams to collaborate on web projects seamlessly.
* **Deployment**: Version control is often integrated with deployment processes to ensure that the correct version of the code is deployed to production.

### 4. **Content Management**

* **Website Content**: Systems like Jekyll or Hugo use version control to manage website content, ensuring that changes to content can be tracked and rolled back if necessary.
* **Blogging Platforms**: Many blogging platforms utilize Git to manage posts and revisions.

### 5. **Research and Academia**

* **Data Management**: Researchers can track changes in datasets, code, and documentation, facilitating collaboration and reproducibility.
* **Collaborative Writing**: Academic papers can be collaboratively written and revised using version control, allowing multiple authors to contribute.

### 6. **Configuration Management**

* **Infrastructure as Code**: Systems like Ansible and Terraform use version control to manage configurations for servers and cloud infrastructure, enabling easy rollbacks and versioning of infrastructure setups.

### 7. **Game Development**

* **Asset Versioning**: Manage changes to game assets (like graphics, audio, and scripts) among teams, ensuring that all members work with the latest versions.
* **Collaboration**: Multiple developers can work on different features simultaneously without overwriting each other’s work.

### 8. **Education**

* **Learning Tools**: Version control systems are used in educational settings to teach programming and collaborative skills.
* **Project Tracking**: Students can manage their projects efficiently, learning best practices for version control and collaboration.

**1.3: Description of git**

✔ **Git Basic concept**

Here’s an overview of the basic concepts of Git, which is a distributed version control system commonly used for tracking changes in source code during software development:

### 1. **Repository**

* A repository (or repo) is a storage space where your project files and their revision history are stored. It can be local (on your machine) or remote (on a server like GitHub).

### 2. **Commit**

* A commit is a snapshot of your project at a specific point in time. Each commit has a unique ID and includes metadata such as the author, timestamp, and a commit message that describes the changes made.

### 3. **Branch**

* A branch is a lightweight movable pointer to a commit. The default branch in Git is usually called main (or master). Branches allow you to work on different features or fixes simultaneously without affecting the main codebase.

### 4. **Merging**

* Merging is the process of combining changes from different branches. When you merge, Git integrates the changes from one branch into another, often resolving conflicts if the same lines were modified.

### 5. **Staging Area (Index)**

* The staging area is a temporary space where you can prepare changes (staged files) before committing them to the repository. You can choose which changes to include in the next commit.

### 6. **Working Directory**

* The working directory is the current state of your files as you see them on your local machine. It contains files that you can edit, add, or delete.

### 7. **Remote Repository**

* A remote repository is a version of your project hosted on the internet or another network. It allows multiple people to collaborate on the same project and share changes. Common remote hosts include GitHub, GitLab, and Bitbucket.

### 8. **Clone**

* Cloning is the process of creating a local copy of a remote repository. This allows you to work on the project locally and push changes back to the remote repository.

### 9. **Pull and Push**

* **Pull**: Fetches changes from a remote repository and merges them into your local branch.
* **Push**: Sends your committed changes from your local repository to a remote repository.

### 10. **Tags**

* Tags are used to mark specific points in history as important, usually used for release versions. Tags can be lightweight (just a pointer to a commit) or annotated (with additional metadata).

**Initialisation**

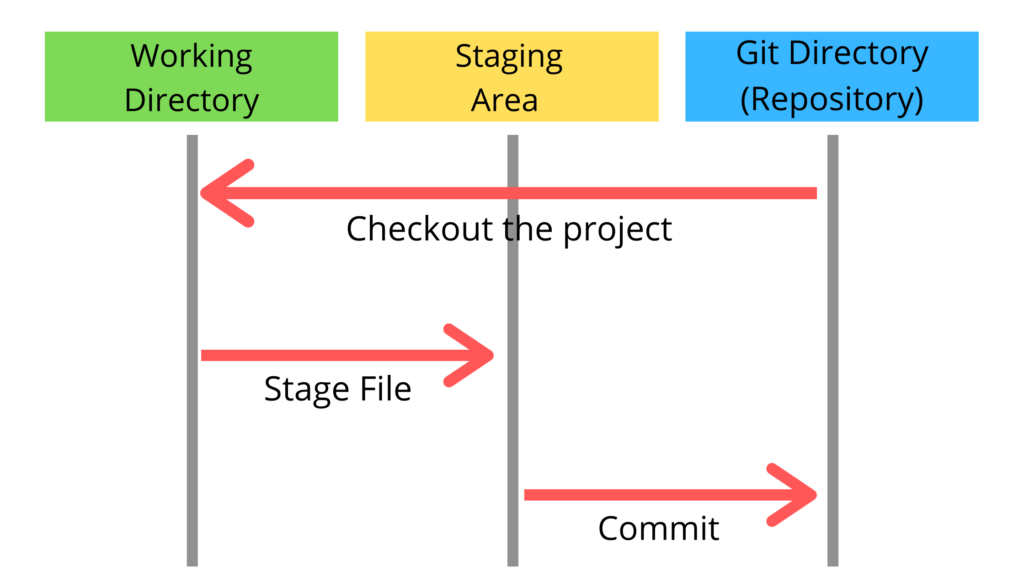
The git init command **creates a new Git repository**. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

The git init command is the first command that you will run on Git. The git init command is used to create a new blank repository.

**✔ Git architecture**

Many VCS’s use a two-tier architecture i.e a repository and a working copy.

Git uses three-tier architecture i.e a working directory, staging area and local repository. The three stages of GIT can store different (or the same) states of the same code in each stage.



**Git – Three Stage Architecture**

As we can see in above diagram there are three stages in Git. When we give any file to the git that file goes from each stage at least once. The three stages of Git can store different (or the same) states of the same code in each stage.

In above diagram there are three stages those are Working directory, Staging area and Git directory(Repository).

Working directory specifies the file explorer’s folder where your files are stored, Staging area is area where your those files are present which you want to send to commit(to create snapshot of files), After commit is fired, files which are in staging area will move to Git Repository.

Now if you made any changes in the files which are in Git repository, those files(with changes) will be in Unstaging area. You again have to add them into Staging area and Commit.

**✔ Git workflow**

The Git workflow is divided into three states:

* Working directory - Modify files in your working directory
* Staging area (Index) - Stage the files and add snapshots of them to your staging area
* Git directory (Repository) - Perform a commit that stores the snapshots permanently to your Git directory. Checkout any existing version, make changes, stage them and commit.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

**Some acronyms**

VCS: Version Control System

GIT: Global Information Tracker

CMD: Command or Command Prompt

BASH: Bourne Again Shell

**There are many version control systems apart from Git and GitHub version control systems**

1. CVS (Concurrent Version System)
2. Mercurial
3. SVN (subversion)
4. GitLab
5. AWS(Amazon Web Service) Code Commit
6. Perforce
7. Beanstalk
8. Team Foundation Server
9. Bitbucket

Here's a clear breakdown of the differences between the **working directory**, **staging area**, and **local repository** in version control systems like Git:

### **Working Directory**

* **Definition**: The local directory on your machine where project files are checked out.
* **Purpose**: It is where you edit, create, or delete files in your project.
* **State**:
  + Files can be untracked, modified, or unmodified.
  + Changes made here are not yet recorded in version control.

### **Staging Area (Index)**

* **Definition**: A temporary area that holds changes you want to commit.
* **Purpose**: Allows you to prepare and select specific changes to include in the next commit.
* **State**:
  + Contains staged changes ready to be committed.
  + Acts as a buffer between the working directory and the local repository.

### **Local Repository**

* **Definition**: A hidden directory in your project that stores the entire history of your commits.
* **Purpose**: It keeps track of all the changes made to the project, allowing you to revert to previous versions or review the commit history.
* **State**:

Contains committed changes, along with metadata about the project.

Includes the entire project history, branches, and tags.

### **Summary**

* **Working Directory**: Active space for development; where you make changes.
* **Staging Area**: Prepares changes for the commit; a selective buffer.
* **Local Repository**: Stores the complete history of commits; the authoritative source of your project’s version history.

a **snapshot** refers to a saved state of your project at a specific point in time

**✔ Initialisation of Git**

# **Terminal** Basic Git commands

Here is a list of some basic Git commands to get you going with Git.

|  |  |  |
| --- | --- | --- |
| Git task | Notes | Git commands |
| [**Tell Git who you are**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |
| [**Create a new local repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) | This is the first step in creating a repository. After running git init, adding and committing files/directories is possible | git init |
| [**Check out a repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) | Create a working copy of a local repository:  Git will create a directory locally with all files and repository history | git clone /path/to/repository |
| For a remote server, use: | git clone username@host:/path/to/repository |
| [**Add files**](https://www.atlassian.com/git/tutorials/saving-changes#git-add) | Add one or more files to staging (index): | git add <filename>  git add \* |
| [**Commit**](https://www.atlassian.com/git/tutorials/saving-changes#git-commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
| Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| [**Push**](https://www.atlassian.com/git/tutorials/syncing#git-push) | Send changes to the master branch of your remote repository: | git push origin master |
| [**Status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository#git-status) | List the files you've changed and those you still need to add or commit: | git status |
| [**Connect to a remote repository**](https://www.atlassian.com/git/tutorials/syncing#git-remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| List all currently configured remote repositories: | git remote -v |
| [**Branches**](https://www.atlassian.com/git/tutorials/using-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |
| [**Update from the remote repository**](https://www.atlassian.com/git/tutorials/syncing) | Fetch and merge changes on the remote server to your working directory: | git pull |
| To merge a different branch into your active branch: | git merge <branchname> |
| View all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging: | git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch> |
| After you have manually resolved any conflicts, you mark the changed file: | git add <filename> |
| **Tags** | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| Push all tags to remote repository: | git push --tags origin |
| [**Undo local changes**](https://www.atlassian.com/git/tutorials/undoing-changes) | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |
| Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git fetch origin  git reset --hard origin/master |
| **Search** | Search the working directory for foo(): | git grep "foo()" |

* **Installation of Git Setup**

Here’s a detailed step-by-step guide to installing Git on Windows:

### Step-by-Step Guide to Install Git on Windows

#### 1. ****Download Git for Windows****

* Go to the official Git website: [git-scm.com](https://git-scm.com/).
* Click on the **"Download"** button for Windows. The website should automatically detect your operating system and provide the appropriate installer.

#### 2. ****Run the Installer****

* Locate the downloaded .exe file (usually in your Downloads folder) and double-click it to start the installation.

#### 3. ****Follow the Installation Wizard****

* **Welcome Screen**: Click **"Next"** to proceed.
* **License Agreement**: Read the license agreement and select **"I Agree"**.
* **Select Destination Location**: Choose the installation path or leave it as the default. Click **"Next"**.
* **Select Components**: You can choose the additional components to install:

**Git Bash**: A command-line interface for Git.

**Git GUI**: A graphical interface for Git.

**Windows Explorer Integration**: Options to use Git from the context menu.

**Associate .git**: Recommended to associate .git files with Git Bash.

**Credential Manager**: Recommended for managing credentials.

After selecting your options, click **"Next"**.

* **Adjusting Your PATH Environment**: Choose how you want to use Git from the command line:
  + **Git from the command line and also from 3rd-party software** (recommended).

Click **"Next"**.

#### 4. ****Complete the Installation****

* Review your selections and click **"Install"** to begin the installation process.
* Once the installation is complete, you can choose to launch **Git Bash** or **Git GUI** directly from the installer.

#### 5. ****Verify Installation****

* Open **Git Bash** (you can find it in your Start Menu).
* Run the following command to verify that Git was installed correctly:

git --version

* This command should return the installed version of Git, confirming that the installation was successful.

The **Git Bash** looks like as



* **Configure Git**

Configuring Git on Windows involves a few straightforward steps. Here’s how to do it:

### Step-by-Step Guide to Configure Git on Windows

#### 1. ****Install Git for Windows****

* Download the Git installer from [git-scm.com](https://git-scm.com/).
* Run the installer and follow the prompts. Make sure to select options that suit your preferences, such as the default editor and line ending conversions.

#### 2. ****Open Git Bash or Command Prompt****

* After installation, you can use **Git Bash** (installed with Git) or the regular **Command Prompt**:
  + **Git Bash**: A terminal emulator for Git that provides a Unix-like command-line experience.
  + **Command Prompt**: The standard command line interface for Windows.

#### 3. ****Set Up User Information****

Configure your username and email address, which will be associated with your commits.

* **Open Git Bash or Command Prompt** and run:

git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

#### 4. ****Configure the Default Text Editor****

You can set your preferred text editor for Git commit messages. For example, to set **Notepad++** or **Visual Studio Code** as your default editor:

* For **Notepad++**:

**git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"**

* For **Visual Studio Code**:

**git config --global core.editor "code --wait"**

#### 5. ****Set Up Line Endings****

To manage line endings correctly across different platforms, configure Git’s handling of line endings:

* For Windows:

**git config --global core.autocrlf true**

#### 6. ****Configure Default Branch Name (Optional)****

If you want to set a default branch name (like main instead of master), run:

**git config --global init.defaultBranch main**

#### 7. ****Check Your Configuration****

To view your current Git configuration settings, run:

**git config --list**

This will display all the settings you have configured.

* **Git init command**

The git init command is used to create a new Git repository. Here’s a detailed explanation of how to use it and what it does:

### Overview of git init

* **Purpose**: Initializes a new, empty Git repository or reinitializes an existing one.
* **Usage**: It sets up the necessary file structure and initializes the .git directory, where all the metadata and version history will be stored.

### **How to Use git init**

#### 1. ****Open a Terminal or Command Prompt****

* On Windows, you can use either **Git Bash** or **Command Prompt**.
* On macOS or Linux, open your terminal.

#### 2. ****Navigate to Your Project Directory****

* Use the cd command to change to the directory where you want to create your repository. For example:

cd path/to/your/project

#### 3. ****Run the****git init****Command****

* Execute the following command to initialize a new Git repository:

**git init**

* After running this command, you will see a message indicating that an empty Git repository has been created.

**● Git – version command**

The git --version command is a simple yet useful command that displays the currently installed version of Git on your system. Here’s a detailed overview of how to use it:

### Overview of git --version

* **Purpose**: To check the version of Git installed on your computer.
* **Usage**: This command is often used to verify that Git is installed correctly and to determine if you need to update to a newer version.

### How to Use git --version

1. **Open a Terminal or Command Prompt**:
   * On Windows, you can use **Git Bash** or **Command Prompt**.
   * On macOS or Linux, open your terminal.
2. **Run the Command**:
   * Type the following command and press **Enter**:

git --version

1. **View the Output**:
   * The output will display the version of Git that is currently installed.

For example:

git version 2.34.1

**✔ Configure .git ignore file**

The **.gitignore file** is used to specify intentionally untracked files that Git should ignore in a project. This is particularly useful for excluding files that are not necessary to share with others, such as temporary files, build artifacts, and configuration files.

### How to Configure a **.gitignore** File

#### 1. ****Create a****.gitignore****File****

* In your project directory, create a new file named **.gitignore**

#### 2. ****Add Patterns to the****.gitignore****File****

Open the .gitignore file in your text editor and add file patterns that you want to ignore.

#### 3. ****Common****.gitignore****Entries by Environment****

Here are some common entries you might want to use, depending on your development environment:

#### 4. ****Save the****.gitignore****File****

After adding the necessary patterns, save the .gitignore file.

#### 5. ****Check the Status****

To see if your .gitignore is working, you can run:

**git status**

Files that match the patterns in .gitignore should not appear in the list of untracked files.

### **Summary**

The **.gitignore** file is a powerful tool for managing which files Git tracks. By configuring it correctly, you can keep your repository clean and free of unnecessary files, making collaboration easier and improving project organization.

**1.4. Use of GitHub repository**

**1. Description of GitHub**

GitHub is a web-based platform that uses Git for version control and collaborative software development. It provides a range of features to facilitate project management, code sharing, and collaboration among developers.

Here’s a detailed description of GitHub:

### **Key Features of GitHub**

1. **Version Control**:
   * Built on Git, GitHub allows users to track changes in their code over time, revert to previous versions, and collaborate on projects.
2. **Repositories**:
   * Users can create repositories to host their projects, which can be public (open to everyone) or private (restricted access).
3. **Collaboration Tools**:
   * **Pull Requests**: A mechanism for proposing changes to a project. Team members can review, discuss, and merge changes from different branches or forks.
   * **Issues**: A way to track bugs, feature requests, and other tasks related to a project, allowing teams to manage their workflow effectively.
4. **Branching and Merging**:
   * Supports the creation of branches to develop features or fix bugs independently. Changes can be merged back into the main codebase after review.
5. **Continuous Integration/Continuous Deployment (CI/CD)**:
   * GitHub Actions allows users to automate workflows, including testing and deployment processes, enhancing project efficiency.
6. **Documentation**:
   * Users can create and host project documentation directly from their repositories, often using Markdown files like README.md.
7. **Community and Social Features**:
   * Users can follow each other, star repositories, and fork projects, fostering community engagement and collaboration.
8. **Integration with Other Tools**:
   * GitHub integrates with numerous third-party tools and services, allowing for enhanced functionality, such as project management, code quality checks, and more.
9. **GitHub Pages**:
   * A feature that allows users to host static websites directly from their GitHub repositories, ideal for project documentation or personal portfolios.
10. **Security Features**:
    * GitHub provides several security features, including vulnerability alerts, dependency scanning, and required reviews for pull requests to enhance code security.

### **Use Cases**

* **Open Source Projects**: Many open source projects are hosted on GitHub, allowing contributors from around the world to collaborate on code.
* **Personal Projects**: Developers can use GitHub to showcase their work, manage personal projects, and maintain a portfolio.
* **Team Collaboration**: Organizations use GitHub to streamline their development process, manage team contributions, and maintain code quality.

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

**Some of its significant features are as follows.**

* Collaboration
* Integrated issue and bug tracking
* Graphical representation of branches
* Git repositories hosting
* Project management
* Team management
* Code hosting
* Track and assign tasks
* Conversations

**Benefits of GitHub**

GitHub offers a wide range of benefits for developers, teams, and organizations. Here’s a comprehensive overview of the key advantages of using GitHub:

### 1. **Collaboration**

* **Teamwork**: GitHub facilitates collaboration among developers through features like pull requests, code reviews, and comments, enabling teams to work together seamlessly.
* **Forking**: Users can fork repositories to create their own copies, allowing independent development without affecting the original project.

### 2. **Version Control**

* **Track Changes**: GitHub tracks every change made to a repository, providing a detailed history of modifications, which can be invaluable for debugging and understanding project evolution.
* **Branching**: Developers can create branches to work on new features or bug fixes without disrupting the main codebase. This encourages experimentation and safe development.

### 3. **Open Source Community**

* **Accessibility**: GitHub hosts millions of open-source projects, making it easy for developers to contribute to existing projects or showcase their own work.
* **Networking**: Being part of the GitHub community allows developers to connect with others, share knowledge, and collaborate on projects.

### 4. **Integration and Automation**

* **GitHub Actions**: This feature allows users to automate workflows, including testing, deployment, and code quality checks, streamlining development processes.
* **Third-Party Integrations**: GitHub integrates with numerous tools, such as CI/CD services, project management software, and code quality analyzers, enhancing overall productivity.( **Continuous Integration** is the practice of automatically integrating code changes from multiple contributors into a shared repository several times a day)

**Continuous Deployment** (and Continuous Delivery) refers to the practice of automatically deploying code changes to production after they pass the testing phase.

### 5. **Project Management**

* **Issues and Milestones**: GitHub’s issue tracking system helps teams manage tasks, bugs, and feature requests effectively. Milestones can be set to track progress towards project goals.
* **Kanban Boards**: GitHub Projects provides visual project management tools, allowing teams to organize and prioritize work efficiently.

### 6. **Documentation and Communication**

* **README Files**: Each repository can include a README file, providing essential information about the project, setup instructions, and usage guidelines.
* **Wiki**: GitHub allows for project documentation through wikis, making it easy to maintain comprehensive documentation alongside the code.

### 7. **Security Features**

* **Vulnerability Alerts**: GitHub automatically scans repositories for known vulnerabilities in dependencies and alerts maintainers.
* **Access Control**: Organizations can manage access permissions for team members, ensuring that only authorized users can make changes.

### 8. **GitHub Pages**

* **Hosting Websites**: Users can create and host websites directly from their GitHub repositories, perfect for project documentation, personal portfolios, or blogs.

### 9. **Free and Paid Options**

* **Cost-Effective**: GitHub offers free public repositories and various pricing plans for private repositories, making it accessible for individuals and teams of all sizes.

### 10. **Education and Learning**

* **Learning Resources**: GitHub provides a wealth of resources, including guides and tutorials, making it easier for new developers to learn version control and collaborative coding practices.

✔ **Create account on GitHub**

Creating an account on GitHub is a straightforward process. Here’s a step-by-step guide to help you set up your GitHub account:

### **Step-by-Step Guide to Create a GitHub Account**

#### 1. ****Visit the GitHub Website****

* Open your web browser and go to the GitHub homepage:  
  [GitHub](https://github.com/)

#### 2. ****Sign Up****

* Click on the **"Sign up"** button, usually located in the upper right corner of the page.

#### 3. ****Enter Your Details****

* **Username**: Choose a unique username. This will be your identity on GitHub.
* **Email Address**: Provide a valid email address. You’ll need to verify this later.
* **Password**: Create a strong password for your account.

#### 4. ****Verify Your Account****

* GitHub may ask you to complete a CAPTCHA to verify that you are not a robot.

#### 5. ****Choose Your Plan****

* You’ll be presented with options for GitHub plans. You can choose the **free plan**, which allows you to create unlimited public and private repositories.

#### 6. ****Customize Your Experience (Optional)****

* GitHub may offer options to customize your experience (such as subscribing to updates, etc.). You can select these based on your preferences.

#### 7. ****Complete the Setup****

* After filling out the necessary information, click the **"Create account"** button.

#### 8. ****Email Verification****

* Check your email inbox for a verification email from GitHub. Click on the link in the email to verify your email address.

#### 9. ****Set Up Your Profile (Optional)****

* After verification, you can set up your GitHub profile by adding extra details like your bio, website, and profile picture.

### **Summary**

Once you’ve completed these steps, you will have a GitHub account ready for use. You can start creating repositories, collaborating on projects, and exploring the vast GitHub community.

**✔ Create new remote repository**

Creating a new remote repository on GitHub is a straightforward process. Here’s a step-by-step guide to help you set it up:

### Step-by-Step Guide to Create a New Remote Repository on GitHub

#### 1. ****Log In to Your GitHub Account****

* Open your web browser and go to [GitHub](https://github.com/).
* Click the **"Sign in"** button in the upper right corner and enter your credentials.

#### 2. ****Create a New Repository****

* After logging in, click the **"+" icon** in the upper right corner of the page.
* Select **"New repository"** from the dropdown menu.

#### 3. ****Fill Out Repository Details****

* **Repository Name**: Enter a name for your repository. This should be descriptive of the project.
* **Description** (optional): Provide a brief description of your repository's purpose.
* **Public/Private**: Choose whether you want your repository to be public (accessible to everyone) or private (only accessible to you and people you invite).
* **Initialize this repository with** (optional):
  + **README file**: This is useful for providing information about your project.
  + **.gitignore**: Select a template to ignore certain files based on your project's language (optional).
  + **License**: Choose a license for your project if you want to specify how others can use your code.

#### 4. ****Create Repository****

* Click the **"Create repository"** button at the bottom of the page.

#### 5. ****Repository Created****

* After you click the button, you will be taken to your new repository page. Here you can start adding files, collaborating, and managing your project.

### **Summary**

You have now successfully created a new remote repository on GitHub. You can start adding files, collaborating, and managing your project directly from the GitHub interface. Additionally, you can link this repository to your local Git project to streamline your workflow.

**✔ Apply git commands related to repository**

Here are some essential Git commands related to managing repositories, along with brief explanations and examples of how to use them:

### 1. **Create a New Repository**

* **Initialize a Local Repository**:

**git init**

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

### 2. **Clone an Existing Repository**

* **Clone a Remote Repository**:

git clone https://github.com/username/repository-name.git

This command creates a local copy of the specified remote repository.

### 3. **Check Repository Status**

* **View the Status of the Repository**:

**git status**

This shows the current state of the working directory and staging area, including changes that are staged, unstaged, or untracked.

### 4. **Add Files to Staging Area**

* **Stage Changes**:

git add file.txt

This command stages a specific file for the next commit. To stage all changes, use:

git add .

### 5. **Commit Changes**

* **Commit Staged Changes**:

git commit -m "Your commit message"

This command saves your changes to the repository with a descriptive message.

### 6. **View Commit History**

* **Show Commit History**:

git log

This command displays a list of commits in the current branch, showing commit IDs, authors, dates, and messages.

### 7. **Create a New Branch**

* **Create a New Branch**:

git branch new-branch

This command creates a new branch named new-branch.

### 8. **Switch Branches**

* **Switch to Another Branch**:

git checkout new-branch

This command switches to the specified branch.

### 9. **Merge Branches**

* **Merge Changes from Another Branch**:

git merge new-branch

This command merges changes from new-branch into the current branch.

### 10. **Delete a Branch**

* **Delete a Branch**:

git branch -d new-branch

This command deletes the specified branch after it has been merged. Use -D to force delete if necessary.

### 11. **Add a Remote Repository**

* **Add a Remote**:

git remote add origin https://github.com/username/repository-name.git

This command adds a remote repository named origin.

### 12. **Push Changes to Remote Repository**

* **Push Commits to Remote**:

git push origin main

This command pushes your local commits to the main branch of the remote repository.

### 13. **Pull Changes from Remote Repository**

* **Fetch and Merge Changes**:

git pull origin main

This command fetches and merges changes from the remote main branch into your current branch.

### 14. **Remove Files from the Repository**

* **Remove a File**:

git rm file.txt

This command removes a file from the working directory and stages the removal for the next commit.

### **Summary**

These Git commands provide a foundational understanding of how to manage repositories, including creating, cloning, and modifying them. Mastering these commands will significantly enhance your workflow in version control and collaboration.

The Git commands related to **cloning** a repository and managing **remote** repositories:

### **1. Git Clone**

The git clone command is used to create a local copy of a remote repository. This allows you to work on the project offline and then sync your changes back to the remote repository when you're ready.

**git clone <repository-url>**

#### Example

To clone a repository from GitHub, you would do the following:

1. **Find the Repository URL**: Go to the repository page on GitHub and click the green **"Code"** button. Copy the URL provided (HTTPS or SSH).
2. **Run the Command**:

**git clone https://github.com/username/repository-name.git**

This creates a directory named repository-name in your current working directory, containing all the files and commit history from the remote repository.

#### Optional Parameters

* **Clone to a Specific Directory**:

**git clone <repository-url> <directory-name>**

This clones the repository into a directory with the specified name.

### 2. **Git Remote**

The git remote command is used to manage the set of repositories (remotes) that your local repository is connected to. This includes adding, removing, and viewing remote connections.

#### Common Commands

* **View Remote Repositories**:

**git remote -v**

This command lists all remote repositories associated with your local repository, showing their names and URLs.

* **Add a Remote Repository**:

**git remote add <name> <repository-url>**

This command adds a new remote repository. By convention, the default remote is named origin.

#### Example

**git remote add origin https://github.com/username/repository-name.git**

* **Remove a Remote**:

**git remote remove <name>**

This command removes a specified remote repository from your local configuration.

#### Example

**git remote remove origin**

* **Change URL of a Remote**:

**git remote set-url <name> <new-url>**

This command changes the URL of an existing remote.

#### Example

**git remote set-url origin https://github.com/username/new-repository-name.git**

### Summary

* **git clone**: This command is essential for duplicating remote repositories to your local machine, allowing you to work on projects offline.
* **git remote**: This command helps manage the connections to remote repositories, enabling you to add, remove, or modify remote links as needed.

**Learning outcome 2: Manipulate files**

**● Definition of general key terms**

#### Status

The git status command **displays the state of the working directory and the staging area**. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git.

#### Branch

The git branch command **lets you create, list, rename, and delete branches**. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

#### Commit

The "commit" command is **used to save your changes to the local repository**. Note that you have to explicitly tell Git which changes you want to include in a commit before running the "git commit" command.

**● Add file change to git staging area**

**✔ Operation on git status command**

##### View new untracked file

When working with new folders, git does not show the contents of those folders by default when you run git status.

To view untracked files run **git status**

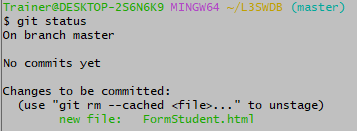


Figure 1 Display untracked files by using Git Status

From that image before, there is no untracked files

Let us create a file called **“studentregistration.html”** inside our repository by using **touch Flile name**

Then after run git status and see what happen

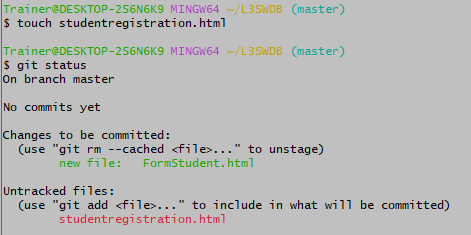


Figure 2 Create a file and check that is untracked by using Git Status

For that case it displays studentregistration.html is under untracked becouse it have been created but not staged means that it is under working stage.

##### View modified file

All modified files or changed files are viewed by using **Git status**

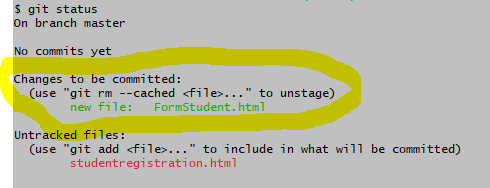
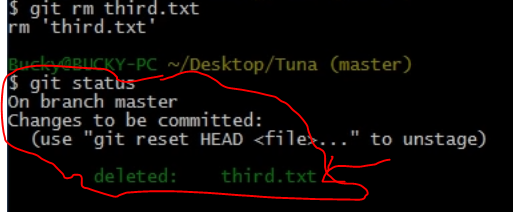


Figure 3 View modified files

##### View deleted file

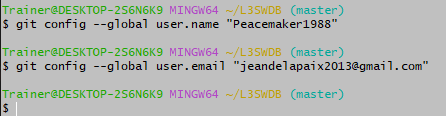
Remember that to delete a file you use **git rm <filename>** once you have deleted a file in git you can view them by using **Git status.**

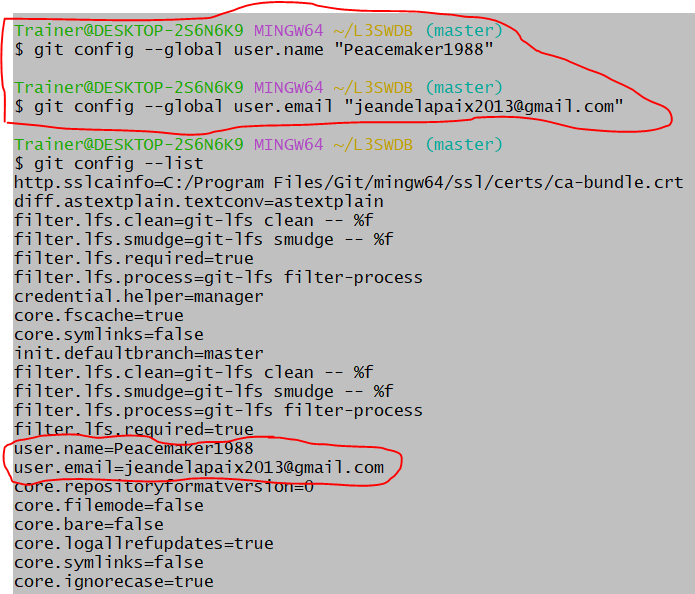


**✔ Operation on git add command**

##### Stage all files

Make sure that your git is configured to your username and email for stagging a file





Once You run git **config –list** it displays all configuration of git.

To stage all files you have to run (**git add .)** in other to add all files to the stagging area as mentioned in git workflow in **outcome 1.**

**N.B** :Don’t forget to put a dot at the end of git add

##### Stage a file

Before staging a file you have to check all existing files and unstagged files for existing files you use **ls command**.

For stagging a single file use git add filename.extention

For file extention use .html , .doc, .js, and others.

**Example** : git add index.html that command will add the index file to stagging area as shown on that image.

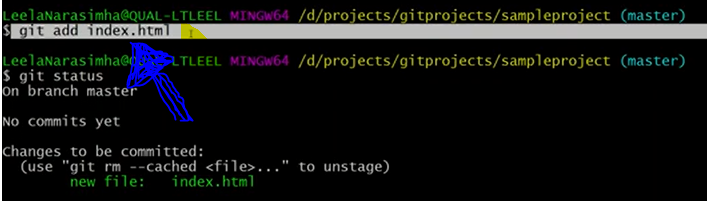


Figure 4 Stage a single file

Once you want to stage all unstaged files at the sametime you write **git add . or git add A**

##### Stage folder

???????

**✔ Operation on git reset command**

● Unstage a file

● Deleting and staging file/folder

## Unstage a File in Git

In Git, unstaging a file can be done in two ways.

1) git rm –cached <file-name>  
2) git reset Head <file-name>

### 1. Unstage Files using git `rm` command

One of the methods to unstage git files is using the ‘rm’ command. It can be used in two ways:

1) On the brand new file which is not on Github.  
2) On the existing file which exists on Github.

#### Case 1: rm –cached on new file which is not committed.

**rm –cached <brand-new-file-name>** is useful to remove only the file(s) from the staging area where this file is not available on GitHub ever. After executing this command, the file rests in the local machine, it just unstaged from the staging area.

**Example:**

$ git add filetwo.txt

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: filetwo.txt

$ git rm --cached filetwo.txt

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

filetwo.txt

nothing added to commit but untracked files present (use "git add" to track)

#### Case 2: rm –cached on existing file.

If **‘rm –cached <existing-file-name>.’**command is utilized on the existing file on git then this file will be considered for delete and endures as untracked on the machine. If we make a commit after this command then the file on Github will be deleted forever. We should be very careful while using this command. So, this case is not advised for unstaging a file.

### 2. Unstage Files using git reset

The most effortless way to unstage files on Git is by using the “git reset” command and specify the file you want to unstage.

**git reset <commit> -- <path>**

By default, the commit parameter is optional: if you don’t specify it, it will be referring to **HEAD**.

#### What does the git reset command do?

This command will **reset the index entries** (the ones you added to your staging area) to their state at the specified commit (or HEAD if you didn’t specify any commits).

Also, we use the double dashes as argument disambiguation meaning that the argument that you are specifying may be related to two distinct objects: branches and directories for example.

As a quick example, let’s pretend that you added a file named “README” to your staging area, but now you want to unstage this file.

**$ git status**

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

In order **to unstage the README file**, you would execute the following command

$ git reset -- README

You can now check the status of your working directory again with “git status”

$ git status

On branch master

Your branch is up to date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

### Unstage all files on Git

Previously, we have seen how you can unstage a file by specifying a path or a file to be reset.

In some cases, you may want **to unstage all your files** from your index.

To unstage all files also you can use the **“git reset”** command without specifying any files or paths.

**$ git reset**

Again, let’s pretend that you have created two files and one directory and that you added them to your staging area.

**$ git status**

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

new file: directory/file

In order to unstage all files and directories, execute “git reset” and they will be removed from the staging area back to your working directory.

$ git reset

$ git status

On branch master

Your branch is up to date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

directory/

nothing added to commit but untracked files present (use "git add" to track)

### Remove unstaged changes on Git

In some cases, after unstaging files from your staging area, you may want to remove them completely.

**In order to remove unstaged changes, use the “git checkout” command and specify the paths to be removed.**

**$ git checkout -- <path>**

Again, let’s say that you have one file that is currently unstaged in your working directory.

**$ git status**

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README

no changes added to commit (use "git add" and/or "git commit -a")

n order to discard changes done to this unstaged file, execute the “git checkout” command and specify the filename.

**$ git checkout -- README**

**$ git status**

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

Alternatively, if you want to discard your entire working directory, head back to the root of your project and execute the following command.

$ git checkout -- .

$ git status

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

### Unstage Committed Files on Git

In some cases, you actually **committed files** to your git directory (or repository) but you want to unstage them in order to make some modifications to your commit.

Luckily for you, there’s also a command for that.

### Unstage Commits Soft

**To unstage commits on Git, use the “git reset” command with the “–soft” option and specify the commit hash.**

$ git reset --soft <commit>

Alternatively, if you want to unstage your last commit, you can the “HEAD” notation in order to revert it easily.

$ git reset --soft HEAD~1

Using the “–soft” argument, **changes are kept in your working directory and index.**

As a consequence, your modifications are kept, they are just not in the Git repository anymore.

Inspecting your repository after a soft reset would give you the following output, given that you unstaged the last commit.

$ git status

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

What happens if you were to hard reset your commit?

In this case, all changes would be discarded and you would lose your changes.

### Unstage Commits Hard

To unstage commits on Git and discard all changes, use the “git reset” command with the “–hard” argument.

$ git reset --hard <commit>

**Note**: Be careful when using the reset hard command, you will lose all your changes when hard resetting.

## Commit File changes to git local repository

**✔ Best practice of creating a commit message**

When writing a git commit message, it's important to follow best practices so that your messages are clear and concise. Here are 7 of the best practices to follow.

#### 1. Separate subject from body with a blank line

The subject line should be used to describe the high-level change being made, while the body can be used for a more detailed description. By separating the two with a blank line, it’s easier to see at a glance what the commit is about.

It’s also important to keep the subject line under 50 characters, so that it can be easily read in most git tools. If you need to go over 50 characters, that’s fine, but anything over 72 characters should be wrapped to the next line.

#### 2. Limit the subject line to 50 characters

The subject line is the first thing people see when they view your commit, and it should be able to stand on its own as a brief summary of what the commit contains. If the subject line is too long, it will be truncated when viewed in tools like GitHub, making it difficult to understand what the commit is about at a glance.

By keeping the subject line short and sweet, you can ensure that your commit messages are easy to read and understand, which will make it easier for people to review and merge your changes.

#### 3. Capitalize the subject line

When git generates a log of commits, it uses the subject line to generate a summary of each commit. If the subject line is not capitalized, the summary will be difficult to read. For example, consider the following two commit messages:

Subject: Add new feature

Subject: add new feature

The first message is much easier to read than the second. Therefore, it’s important to always capitalize the subject line of your git commit messages.

#### 4. Do not end the subject line with a period

When the subject line of a git commit message is ended with a period, it can often be interpreted as an abbreviation. For example, “Fixed bug.” would be interpreted as “Fixed bug with no further explanation.” This can often lead to confusion for other developers who are trying to understand what was changed in that particular commit.

It’s much better to write git commit messages like this:

“Fixed bug where X was not happening.”

This makes it clear that the bug has been fixed and provides some context about the issue that was being faced.

#### 5. Use the imperative mood in the subject line

When you’re writing a git commit message, you’re essentially writing a command that tells the code to do something. For example, “fix bug” or “add feature.”

Using the imperative mood in your subject line makes it clear that you’re giving a command, which helps keep your messages consistent and easy to understand.

#### 6. Wrap the body at 72 characters

When git generates a patch, it uses the first line of the commit message as the subject and the rest of the message as the body. The generated email will have a limited width, so if the body isn’t wrapped, it will be very difficult to read.

By wrapping the body at 72 characters, you ensure that the generated email will be easy to read. This is especially important when multiple people are working on the same codebase, as they need to be able to quickly understand each other’s commits.

It’s also worth noting that some git clients will wrap the body for you automatically. However, it’s still good practice to wrap it manually, as this ensures that the generated email will be readable even if the client doesn’t wrap it correctly.

#### 7. Use the body to explain what and why vs. how

When reading a commit message, I want to know two things: what changed and why. The body of the commit message should explain those two things. The subject line should be a short summary (50 characters is a good rule of thumb) that tells me what changed. The body should tell me why this change was made.

The reason for this is simple: the what and the why are more important than the how. The how can be easily gleaned from the diff; what’s more important is understanding the motivation behind a change.

**✔ Operation on git commit command**

**●** Commit a file

**git commit:** This command will record the modifications done to the files to a local repository. For simple reference, each commit has a unique ID.

***Usage:***

# Adding a commit with message

$ git commit -m "Commit message in quotes"

***In Practice:***

$ git commit -m "My first commit message"

[SecretTesting 0254c3d] My first commit message

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 homepage/index.html

**●** Edit commit message

# **How to amend git commit message | change git commit message after push**

## The Git Commit Amend Command

This command will allow you to change files in your last commit or your commit message. Your old commit is replaced with a new commit that has its own ID.

**The following syntax is for the amend command:**

**git commit --amend**

Amending a commit does not simply change a commit. It substitutes it with a new commit which will have its own ID.

The new commit and message will seem on GitHub the next time you push.

### How to Amend the latest Git Commit Message?

Are you looking for the process of amending the latest Git commit message?  This section will explain you clearly. In case the message to be amended is for the latest commit to the repository, then the following commands are to be performed:

git commit --amend -m "New message"

git push --force repository-name branch-name

Remember that using –force is not supported, as this changes the history of your repository. If you force push, people who have already cloned your repository will have to manually fix their local history.

Execute the “git commit” command and make sure to specify the “–amend” option.

$ git commit --amend

**✔ Operation on git log command**

The git log command gives the order of the commit history for a repository. The command aids in getting the state of the current branch by showing the commits that lead to this state.

**● To see simplified list of commit**

*Usage:*

# Show git log with date pameters

$ git log --<after/before/since/until>=<date>

# Show git log based on commit author

$ git log --<author>="Author Name"

**● To see a list of commits with more detail**

# Show entire git log

$ git log

## Manage branches

**✔ Operations on branches**

**git branch:** To discover what branch the local repository is on, add a new branch, or delete a branch.

**Create branch**

# Create a new branch

$ git branch <branch\_name>

### **List branch**

# List all remote or local branches

$ git branch -a

### **Delete local and remote branch**

# Delete a branch

$ git branch -d <branch\_name>

*In Practice:*

# Create a new branch

$ git branch new\_feature

# List branches

$ git branch -a

\* SecretTesting

new\_feature

remotes/origin/stable

remotes/origin/staging

remotes/origin/master -> origin/SecretTesting

# Delete a branch

$ git branch -d new\_feature

Deleted branch new\_feature (was 0254c3d).

### **Switch branch**

**git checkout:** By using git checkout, you can easily switch branches, whenever the work is to be started on a different branch. The command works on three separate entities: files, commits, and branches.

***Usage:***

**# Checkout an existing branch**

**$ git checkout <branch\_name>**

**# Checkout and create a new branch with that name**

**$ git checkout -b <new\_branch>**

***In Practice:***

**# Switching to branch 'new\_feature'**

**$ git checkout new\_feature**

**Switched to branch 'new\_feature'**

**# Creating and switching to branch 'staging'**

**$ git checkout -b staging**

**Switched to a new branch 'staging'**

### **Rename branch**

You can rename a local or remote Git branch by using the **-m command**. While this is not a problem for the local branch, for the remote branch you must first **delete the outdated version** and replace it with the new one.

There may be times when you need to rename a Git branch.

This is because if the naming is wrong and other developers continue to work with it, you may run into problems. Fortunately, despite the tight integration and various forks, if you want to rename a Git branch, it’s not a big issue. To do this, use the **-m command**. The corresponding syntax always follows the same structure:

“**git branch -m <old-name> <new-name>**”.

However, there are differences between branches that you edit locally and those that are already remote. Below we explain the steps for both cases.

### Rename a local Git branch

A local Git branch exists only on your computer. You make changes and tests here without other developers noticing. Renaming it can therefore be done quickly.

1. In the command line, select the Git branch you want to rename. The command for this is “**git checkout old-name**”.
2. You will get a confirmation that you have selected the correct branch. This will read **“Switched to branch 'old-name'”**.
3. Now perform the actual rename for the local Git branch. The appropriate command for this is: **“git branch -m new-name”.**

Alternatively, you have the option to rename the Git branch **via the master**. To do this, use the following steps:

1. Switch to the master via the command **“git checkout master”**.
2. Now enter the following command if you want to rename a Git branch: “**git branch -m old-name new-name**”.
3. To ensure that the rename was successful, retrieve the current status of the branch using the “**git branch -a**” command.

### Renaming a remote Git branch

In a remote repository, you cannot simply rename a Git branch, as this would lead to complications. Instead, you need to delete the old name and then add the branch with the new name. Fortunately, this is not too hard either and can be done with a few simple commands. As with the local branch, you have **two options**.

1. First, make sure the local branch has the correct, new name. The appropriate command is “**git branch -a**”.
2. Now delete the branch with the old, incorrect name from the remote repository. To do this, use the following command: “**git push origin --delete old-name**”.
3. Verify that the old branch has been deleted properly.
4. Now add the branch with the correct name. For this, use the command “**git push origin -u new-name**”.
5. Lastly, perform a **reset of the upstream branch** to ensure that the changes are effective.

However, if you want to rename the remote Git Branch with just one command, you also have the following option.

1. Enter the following command: “**git push origin: old-name new-name**”.
2. Then also perform a **reset of the upstream branch** as described above.

**Learning outcome 3: Ship codes**

* **● Definition of general key terms**

**● pull**

**● fetch**

**● push**

**● pull request**

**● merge**

**● Fetch file from GitHub repository**

**✔ Operation on git fetch command**

**● Fetch the remote repository**

**● Fetch the specific branch**

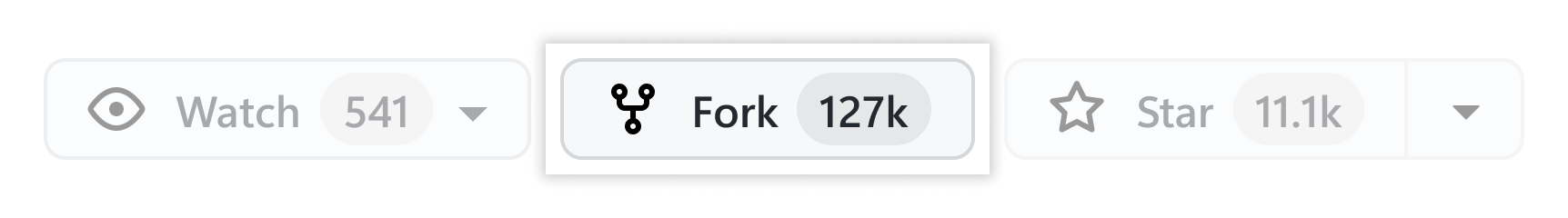
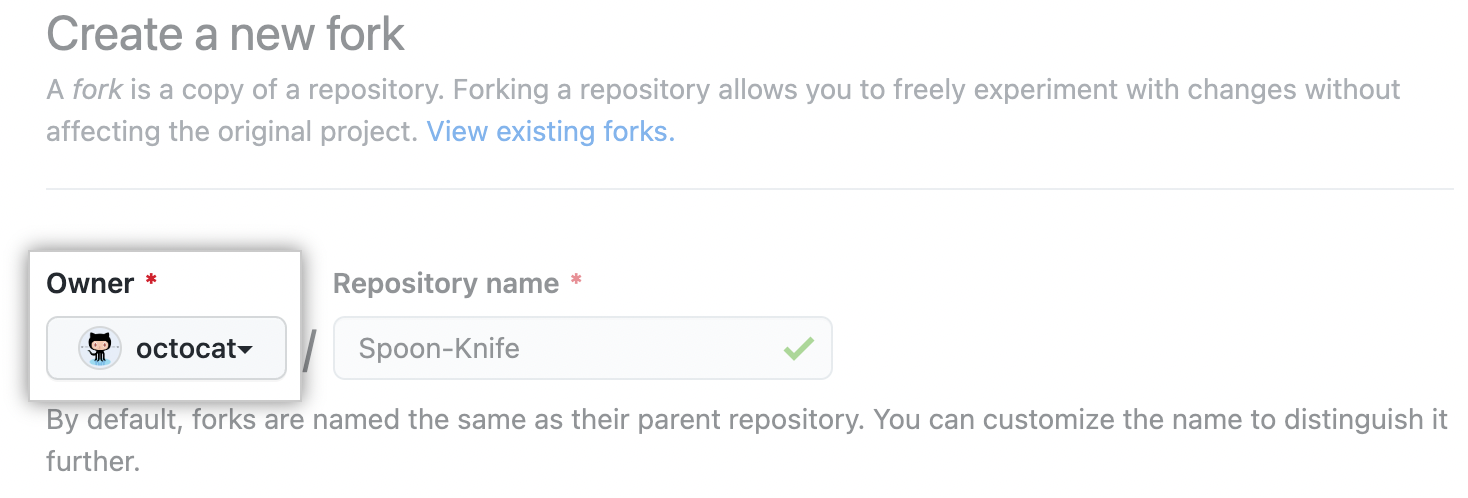
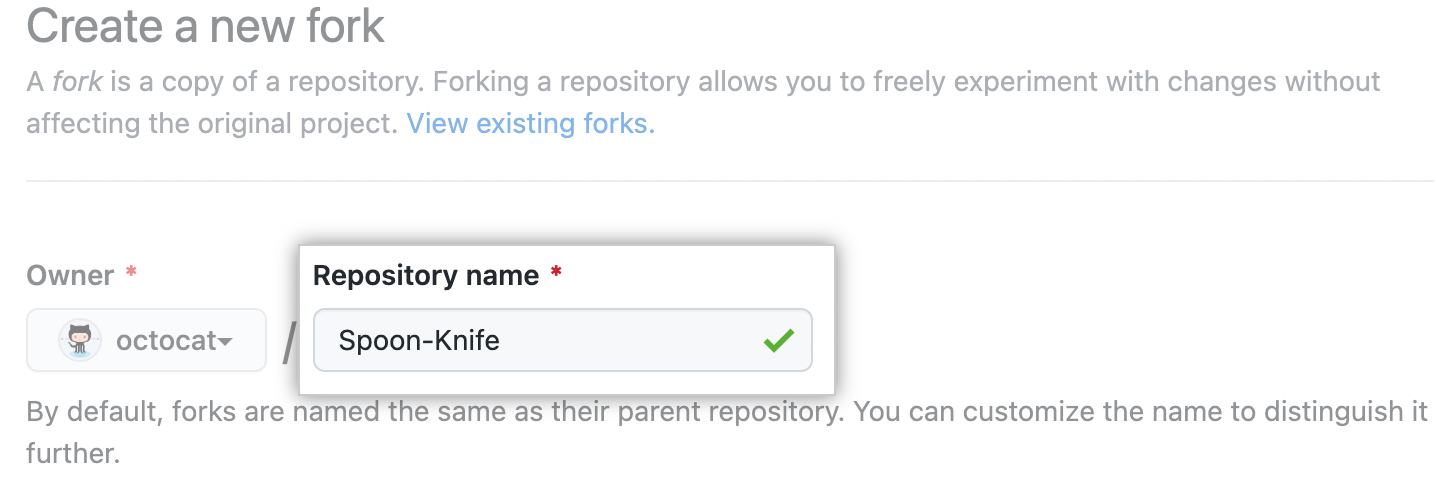
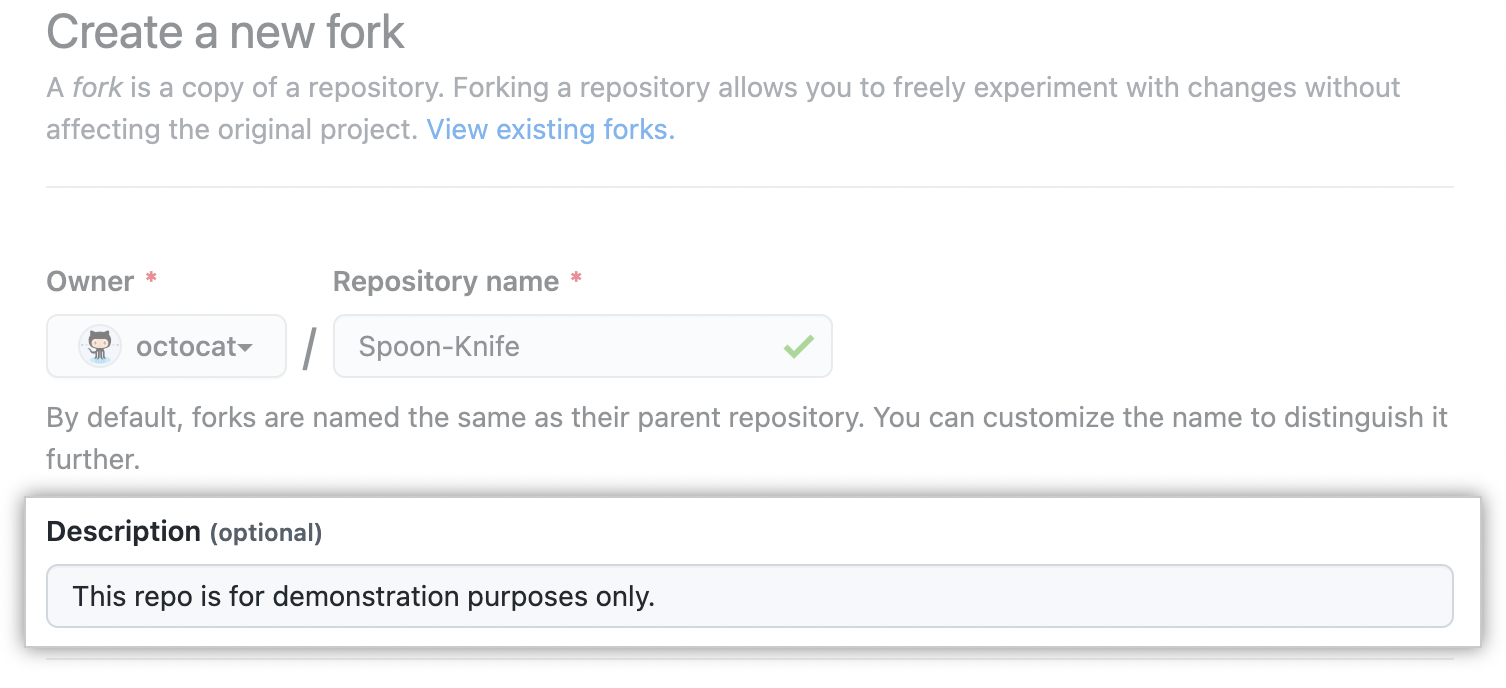
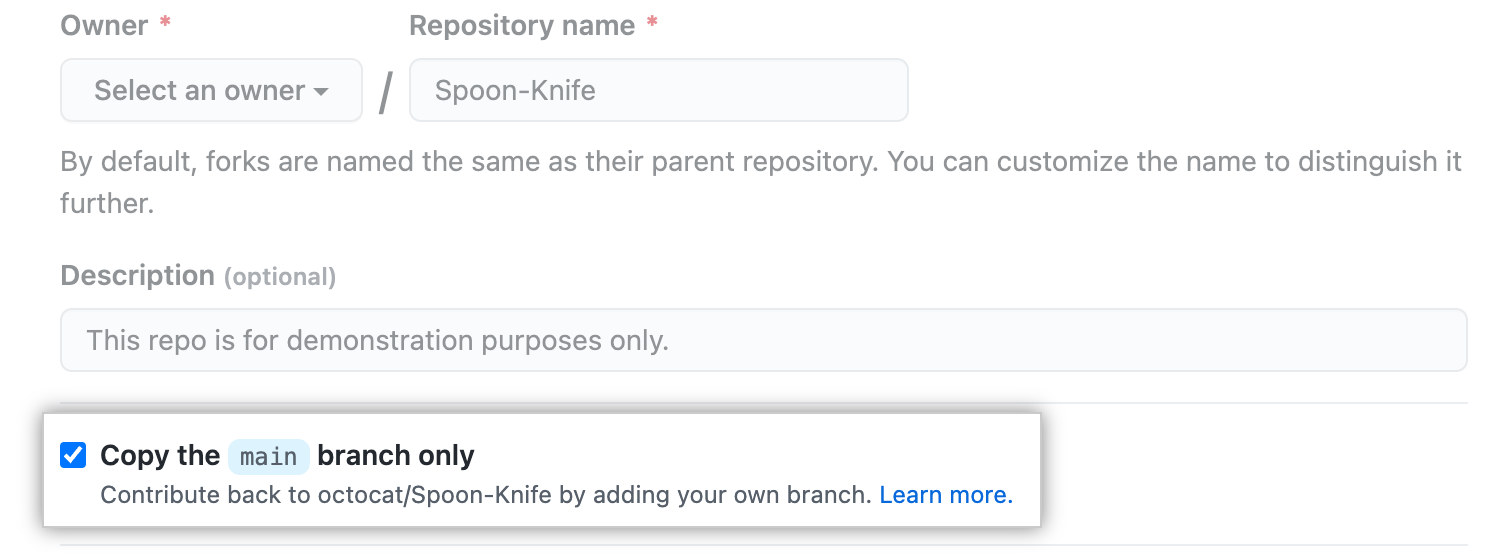
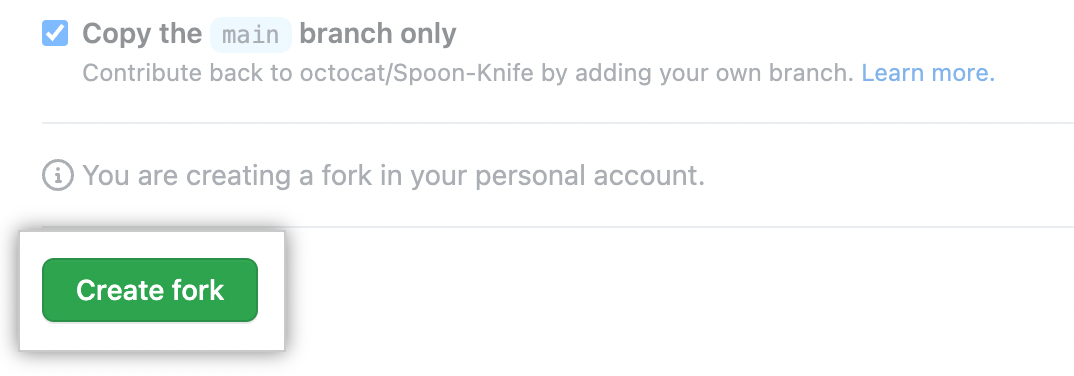
**● Fetch all the branch simultaneously**

**● Synchronize the local repository**

**✔ Operation on git pull**

## Forking a repository

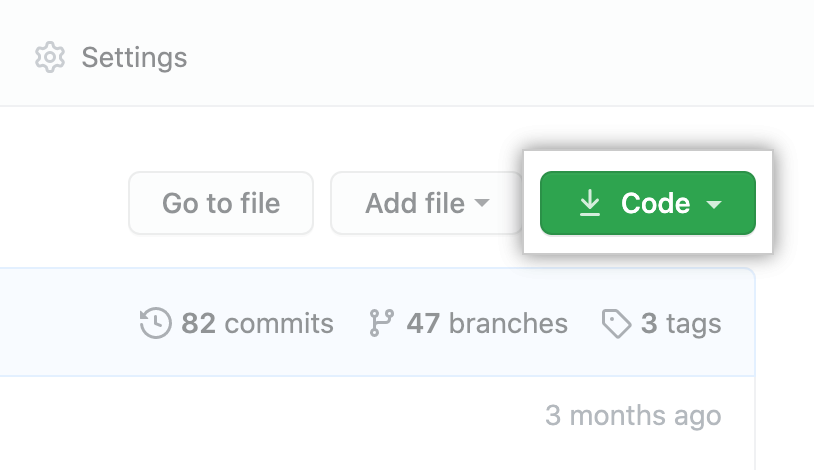
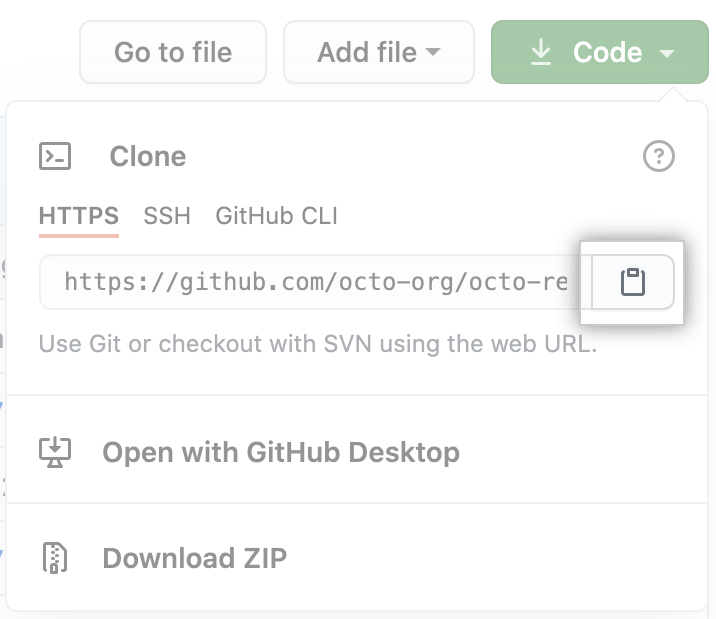
You might fork a project to propose changes to the upstream, or original, repository. In this case, it's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository you just forked.

1. On GitHub.com, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
2. In the top-right corner of the page, click **Fork**.
3. Select an owner for the forked repository.
4. By default, forks are named the same as their parent repositories. You can change the name of the fork to distinguish it further.
5. Optionally, add a description of your fork.
6. Choose whether to copy only the default branch or all branches to the new fork. For many forking scenarios, such as contributing to open-source projects, you only need to copy the default branch. By default, only the default branch is copied.
7. Click **Create fork**.

**Note:** If you want to copy additional branches from the parent repository, you can do so from the **Branches** page. For more information, see "[Creating and deleting branches within your repository](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-and-deleting-branches-within-your-repository)."

## Cloning your forked repository

Right now, you have a fork of the Spoon-Knife repository, but you do not have the files in that repository locally on your computer.

1. On GitHub.com, navigate to **your fork** of the Spoon-Knife repository.
2. Above the list of files, click  **Code**.
3. Copy the URL for the repository.
   * To clone the repository using HTTPS, under "HTTPS", click .
   * To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **SSH**, then click .
   * To clone a repository using GitHub CLI, click **GitHub CLI**, then click .
4. Open Git Bash.
5. Change the current working directory to the location where you want the cloned directory.
6. Type git clone, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:

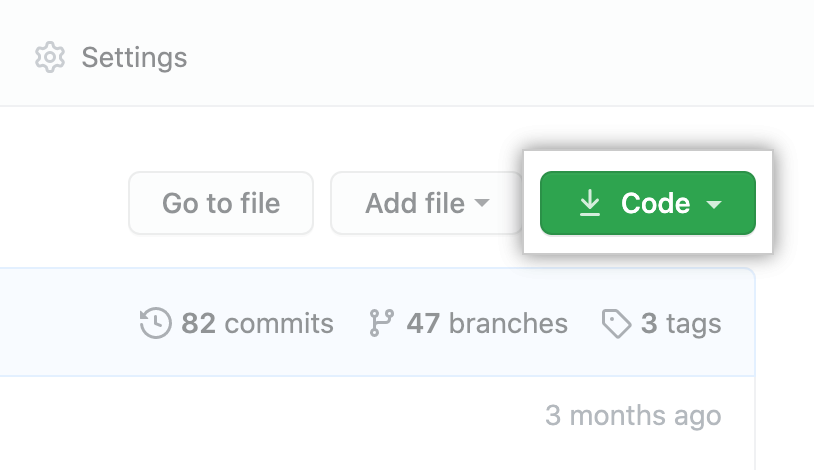
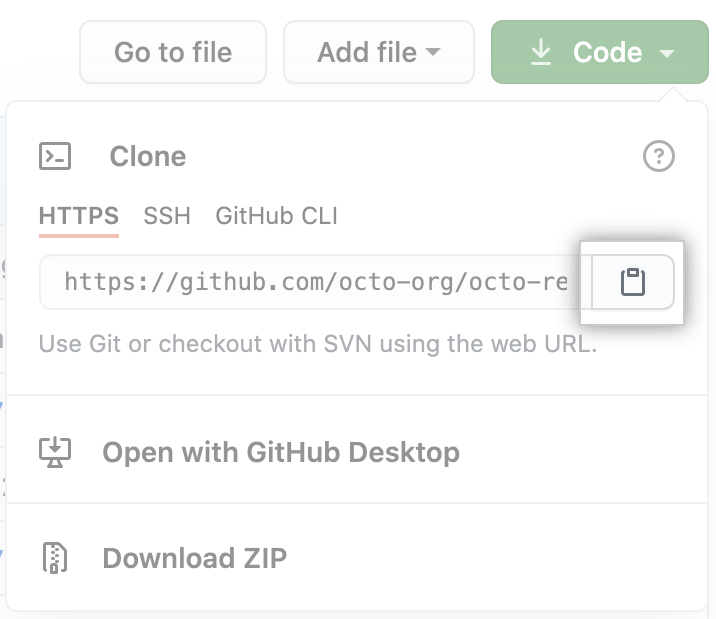
$ git clone https://github.com/YOUR-USERNAME/Spoon-Knife

1. Press **Enter**. Your local clone will be created.
2. $ git clone https://github.com/YOUR-USERNAME/Spoon-Knife
3. > Cloning into `Spoon-Knife`...
4. > remote: Counting objects: 10, done.
5. > remote: Compressing objects: 100% (8/8), done.
6. > remote: Total 10 (delta 1), reused 10 (delta 1)

> Unpacking objects: 100% (10/10), done.

## Configuring Git to sync your fork with the original repository

When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

1. On GitHub.com, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
2. Above the list of files, click  **Code**.
3. Copy the URL for the repository.
   * To clone the repository using HTTPS, under "HTTPS", click .
   * To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **SSH**, then click .
   * To clone a repository using GitHub CLI, click **GitHub CLI**, then click .
4. Open Git Bash.
5. Change directories to the location of the fork you cloned.
   * To go to your home directory, type just cd with no other text.
   * To list the files and folders in your current directory, type ls.
   * To go into one of your listed directories, type cd your\_listed\_directory.
   * To go up one directory, type cd ...
6. Type git remote -v and press **Enter**. You will see the current configured remote repository for your fork.
7. $ git remote -v
8. > origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (fetch)

> origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (push)

1. Type git remote add upstream, and then paste the URL you copied in Step 3 and press **Enter**. It will look like this:

$ git remote add upstream https://github.com/ORIGINAL\_OWNER/Spoon-Knife.git

1. To verify the new upstream repository you have specified for your fork, type git remote -v again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.
2. $ git remote -v
3. > origin https://github.com/YOUR\_USERNAME/ YOUR\_FORK.git (fetch)
4. > origin https://github.com/YOUR\_USERNAME/YOUR\_FORK.git (push)
5. > upstream https://github.com/ORIGINAL\_OWNER/ ORIGINAL\_REPOSITORY.git (fetch)

> upstream https://github.com/ORIGINAL\_OWNER/ ORIGINAL\_REPOSITORY.git (push)

Now, you can keep your fork synced with the upstream repository with a few Git commands. For more information, see "[Syncing a fork](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/syncing-a-fork)."

**● Default git pull**

**● Git pull remote branch**

**● Git force pull**

**● Git pull origin master**

**● Push files to remote branch**

**✔ Tags used on git push command**

**✔ operation on git push**

**● push on origin master**

**● git push force**

**● git push verbose**

**● delete a remote branch**

**● Merge branches on remote repository**

**✔ operation on git rebase command**

What Is Git Rebase: is git command used to integrate changes from one branch into another in streamlined way.

It allows you to move commits from one branch to another,more linear project history.

Here are some configuration options you can take advantage of. Note that these options will alter the Git rebase output’s feel and look.

* **rebase.stat-** This is a boolean set to “false” by default. This option toggles the display of visual diffstat content showing what has changed since the last rebase.
* **rebase.autoSquash-** This boolean value toggles the --autosquash behavior.
* **rebase.instructionFormat-** This is a Git log format string used for formatting an interactive rebase display.
* **rebase.missingCommitsCheck-** This option can be set to multiple values, changing rebase behavior around missing commits.

The values are:

* + **Warn.** This prints warning output in interactive mode and warns you of removed contents.
  + **Error.** This stops the rebase and prints any removed commit warning messages.
  + **Ignore.** This value is set by default and ignores missing commit warnings.

Now, about those rebase commands.

## Rebase Commands

Here’s a summary of the different commands associated with Git rebase.

|  |  |
| --- | --- |
| git rebase <base> | Performs the standard rebase |
| git rebase – interactive <base> | Performs the interactive rebase |
| git rebase – d | The commit gets discarded from the final combined commit block during playback. |
| git rebase – p | This leaves the commit alone, not modifying the content or message, and keeping it as an individual commit in the branches’ history. |
| git rebase – x | This executes a command line shell script for each marked commit during playback. |
| git status | Checks the rebase status. |
| git rebase – continue | Continue with the changes that you made. |
| git rebase --skip | Skips the changes |
| git add <project file> | Adds your branch to the repository |
| git commit -m "new commit for <branch name>." | Commits the changes. |

## About Rebase Branch

Sometimes, a developer will have many commits in different branches and want to combine them all into a single branch. There are two options: merge it or rebase it, and the latter is the best choice.

First, you need to switch to the branch in question:

git checkout <branch name>

Then, just rebase to the master.

git rebase master

## Git Advanced Rebase Application

The  - - onto­ command activates a more powerful rebase type that lets you pass specific refs to become the tips of a rebase. For example, you can achieve this through this command:

 git rebase --onto <newbase> <oldbase>

## The Dangers of Rebase

Git rebasing comes with risks, an understandable situation considering how it rewrites history. However, whenever users have the means to change a file that multiple users can access, it increases the risk of problems.

If your long-lived branch has strayed too far from the main, you may experience merge conflicts. In this case, you need to rebase against the main eventually, but the situation may have escalated because there are so many new commits that your branch changes will [conflict](https://www.simplilearn.com/tutorials/git-tutorial/merge-conflicts-in-git) with. You can avoid this problem by frequently rebasing your branch against the main and making more frequent commits.

When dealing with conflicts, you can reset or advance the rebase by passing the --continue and --abort command line arguments to the Git rebase.

Another rebase danger involves losing commits from interactive history writing. If you run an interactive mode rebase and execute subcommands such as drop or squash, you could remove commits from your branch’s immediate log. Use git reflog to restore the commits and undo the rebase.

Don't let these dangers deter you from rebasing in the final analysis. The only serious issue arises when you execute a history rewriting interactive rebase and end up force pushing the results onto a remote branch that other users share. Unfortunately, this could overwrite other users’ work when performing a pull.

**✔ create pull request**

**Making a pull request**

At last, you're ready to propose changes into the main project! This is the final step in producing a fork of someone else's project, and arguably the most important. If you've made a change that you feel would benefit the community as a whole, you should definitely consider contributing back.

To do so, head on over to the repository on GitHub where your project lives. For this example, it would be at https://www.github.com/<your\_username>/Spoon-Knife. You'll see a banner indicating that your branch is one commit ahead of octocat:main. Click **Contribute** and then **Open a pull request**.

GitHub will bring you to a page that shows the differences between your fork and the octocat/Spoon-Knife repository. Click **Create pull request**.

GitHub will bring you to a page where you can enter a title and a description of your changes. It's important to provide as much useful information and a rationale for why you're making this pull request in the first place. The project owner needs to be able to determine whether your change is as useful to everyone as you think it is. Finally, click **Create pull request**.

* **Opening pull requests:** If you are hoping to contribute back to the original repository, you can send a request to the original author to pull your fork into their repository by submitting a [pull request](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests).

### **Create a pull request**

Create a pull request to ask collaborators for feedback on your changes. Pull request review is so valuable that some repositories require an approving review before pull requests can be merged. If you want early feedback or advice before you complete your changes, you can mark your pull request as a draft. For more information, see "[Creating a pull request](https://docs.github.com/en/articles/creating-a-pull-request)."

When you create a pull request, include a summary of the changes and what problem they solve. You can include images, links, and tables to help convey this information. If your pull request addresses an issue, link the issue so that issue stakeholders are aware of the pull request and vice versa. If you link with a keyword, the issue will close automatically when the pull request merges. For more information, see "[Basic writing and formatting syntax](https://docs.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax)" and "[Linking a pull request to an issue](https://docs.github.com/en/github/managing-your-work-on-github/linking-a-pull-request-to-an-issue)."

## Pull Requests

Pull Requests are the heart of GitHub **collaboration**.

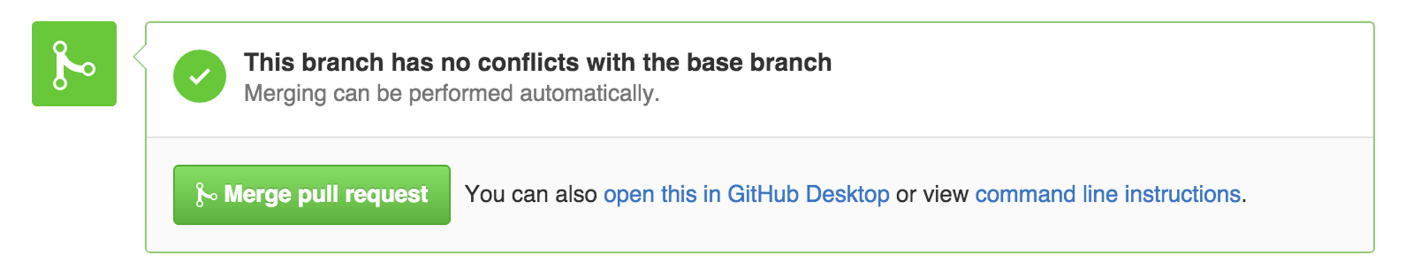
With a pull request you are **proposing** that your changes should be **merged** (pulled in) with the master.

Pull requests show content **differences**, changes, additions, and subtractions in **colors** (green and red).

As soon as you have a commit, you can open a pull request and start a discussion, even before the code is finished.

A a great way to learn GitHub, before working on larger projects, is to open pull requests in your own repository and merge them yourself.

You merge any changes into the master by clicking a "Merge pull request" button.



After merging you can delete the branch by clicking a "Delete branch button".

**✔ operation on git merge**

**● merge the specified commit to current active branch**

**● merge commits into the master branch**

**● git merge branch**