#

Building and Using a Basic Version Control System with Git and GitHub

Instruction Manual by Anthony Silva

## **Background: What Is Git?**

Git is one of the most widely used software development tools in the world. It is a specific kind of tool known as a **version control system (VCS)**. The purpose of a VCS is to record changes to a set of files over time. What makes Git stand out among other VCSs is how it stores the version history of a project. Each collection recorded in this timeline is called a **commit**. When using Git, there are three steps a programmer takes to create a commit:

1. **Editing**: the user initializes new files and/or makes edits to them where necessary.
2. **Staging**: the user marks the file(s) as ready for the next commit after all the necessary changes have been made
3. **Committing**: a collection of changes to the project is made and stored in the local computer or specified server

## **Purpose and Structure**

While there are multiple features that Git has to offer, this guide is intended to introduce experienced programmers to the basics of Git and the commands used. The guide is primarily for individual usage, including but not limited to assisting with personal organization of code and portfolios. While this guide is split up into 10 primary steps of setting up a Git repository and how to use it, certain groups of steps are grouped together so completing a certain task can be completed in a certain order. More specifically, there are five different sections to this guide:

1. [**Steps 1-2**](#_Step_1:_Configure): Configuring Git to the Machine and Initializing the Git Directory
2. [**Steps 3-5**](#_Step_3:_Set): Performing the Committing Process in the Software Development Environment
3. [**Steps 6-7**](#_Step_6:_Create): Creating a GitHub Repo
4. [**Steps 8-10**](#_Step_8:_Set): Connecting the Git Repository to GitHub
5. [**Bonus**](#_Bonus_Step_1:): Branching, Pull Requests, and Code Review

## **SECURITY WARNING! A white triangle with a exclamation mark  Description automatically generated**

Part of following this guide involves creating and using sensitive information. Always keep any keys and passwords secret, and only store information on paper or on the local device. Failure to access this information may result in being locked out permanently. In addition, never share sensitive information with others or post your data on the Internet, even if it is on a website that requires a login like Google Docs. A hacker may find this data and use it to log into accounts, and they might likely destroy the program or even the device!

## **Software Required**

Before you can use Git, a **Git client** must be installed on the device. Some recommended places to install Git based on the operating system are provided:

* **Windows**: Visit gitforwindows.org, click the blue “Download” button in the top header, and follow all instructions to download Git to the computer.
* **Linux/Mac**: It may be easiest to install Git using Homebrew. In the command terminal, just type *brew install git*.

A **software development environment**, where code is written and Git commands are run through, is required to use Git. All visuals in this guide will use Visual Studio Code, though any other environment may be used to use Git. All steps will require the use of a command terminal, so set up a terminal in the project directory before starting. Verify the command line is set to “Git Bash” so that each command works properly.

## **SECURITY WARNING! A white triangle with a exclamation mark  Description automatically generated**

Please refer to the security warning about passwords on page 2 before proceeding.

The last item required for this project is a **GitHub account**. GitHub is a popular platform for saving code in the cloud, as well as store information on Git commits. If needed, follow these steps to create a GitHub account:

1. Go to github.com
2. Click the “Sign Up” button towards the top-right corner of the main page
3. Follow the prompts required to set up an account

## **Git Command Summary**

Table 1 below is designed to be used to look up basic commands as Git projects are completed. Each command is explained in more detail throughout this guide. Each row in the table includes the command, what it does, and which step the command is found in.

|  |  |  |
| --- | --- | --- |
| **Git Command** | **Usage** | **Step Found (with Link)** |
| **git config --global user.name "Your Name"** | Configures the device with the git user’s name | [Step 1](#_Step_1:_Configure_1) |
| **git config --global user.email "Your Email"** | Configures the device with the git user’s email | [Step 1](#_Step_1:_Configure_1) |
| **git init** | Initializes a Git repository | [Step 2](#_Step_2:_Initialize) |
| **git status** | Displays the status of changed files | [Step 3](#_Step_3:_Set_1) |
| **git add “file name 1” “file name 2” …** | Sets specified files to the staging area | [Step 4](#_Step_4:_Add) |
| **git rm –cached “file name 1” “file name 2” …** | Removes specified files from the staging area | [Step 4](#_Step_4:_Add) |
| **git commit -m “Details about the commit”** | Records a commit into the Git repository with a message stating what was changed | [Step 5](#_Step_5:_Commit) |
| **git log** | Displays a timeline of all commits | [Step 5](#_Step_5:_Commit) |
| **git remote add origin “HTTPS URL”** | Creates a new remote (named “origin”) from the working directory to a Git repository on the cloud | [Step 8](#_Step_8:_Set) |
| **git remote -v** | Displays a list of remotes created | [Step 8](#_Step_8:_Set) |
| **git branch -M main** | Renames a Git branch as “main” | [Step 9](#_Step_9:_Push) |
| **git push** | Pushes all commits to the repository set by the remote | [Step 9](#_Step_9:_Push) |
| **git push -u origin main** | Finalizes the connection set by the remote | [Step 9](#_Step_9:_Push) |
| **git pull** | Collects the entire commit history from the repository | [Step 10](#_Step_10:_Pull) |
| **git stash** | Stores changes made in a stack | [Step 10](#_Fixing_Issues_with) |
| **git stash pop** | Pulls the last set of changes from the stash | [Step 10](#_Fixing_Issues_with) |
| **git branch** | Lists out all branches in the working directory | [Bonus Step 1](#_Bonus_Step_1:) |
| **git branch “branch-name”** | Creates a new branch in the working directory | [Bonus Step 1](#_Bonus_Step_1:) |
| **git checkout “branch-name”** | Switches work over to the specified branch name | [Bonus Step 1](#_Bonus_Step_1:) |
| **git push –set-upstream “remote-name” “branch-name”** | Uses the remote from the working directory to the destination of the remote and pushes the specified branch into the destination, as well as all changes in that branch | [Bonus Step 1](#_Bonus_Step_1:) |

**Table 1:** List of Git commands shown in this guide and their usage. Source: Primary

## **Step 1: Configure Git to The Device**

When using Git on some device for the first time, Git must be configured to the device. Git uses this information to display the author of each commit made in a repository [1]. To begin:

1. Open a software development platform
2. Create a new command line terminal
3. Type in the following line in the terminal, replacing “Your Name” with your actual name.

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

1. Type in the following line in the terminal, replacing “Your Email” with the email you are using for Git.

**git config --global user.email "Your Email"**

## **Step 2: Initialize A Directory with Git**

**DATA WARNING! **

Upon completion of this step, a new directory will be created that will be hidden from the file explorer. Do not attempt to modify anything in this directory. Manually modifying data in this directory could corrupt the Git repository.

Every git repository associates with a directory on the device. For this guide, a blank git repository will be created before any files can be added for the project. Upon initializing a directory with Git, a new Git directory will be set up where information about the Git repository will be stored. With the software development platform open, follow these steps:

1. Based on your operating system, create a new directory either from the file explorer or the software development platform.

**Note**: if using the bash command terminal, it is easiest to create a new directory using the “mkdir” command, which automatically creates a new directory in the current folder. This is done with the following line, replacing “dir name” with the name of the directory:

**mkdir “dir name”**

1. In the bash command terminal, use the “cd” command to navigate into this directory:

**cd “dir name”**

1. Run this line to initialize this new directory as a Git repository:

**git init**

1. Verify that the initialization was successful by running this line. It is typically shown as “.git/”.

**ls -a**

Figure 1 below shoes the results of creating a basic git repository and what the list results may show.

****

**Figure 1:** Typing in “git init” into the terminal and displaying the results of creating the git repository. Source: Primary

## **Step 3: Set Up or Edit Files for the Git Repository (Editing)**

Once a git repository is set up, the committing process can begin. Setting up and editing the files in this repository is the first stage of the committing process. This may be completed with either one or both of the following processes:

* Set up new files to be recorded in the Git repository. The fastest way to do this is with the bash command terminal using the “touch” command, adding as many files as necessary, replacing the quotes with the appropriate file names:

**touch “file name 1” “file name 2” …**

* Make any necessary changes to the files already in the directory in the text editor of the software development environment. Save these changes to the device with Ctrl+S (Windows) or Cmd+S (Mac) when finished.

Figure 2 below shows the example code used in the development of the project, which is an object-oriented program in C++ that prints “HELLO WORLD!” Note the names of the two files, “helloWorld.h” and “helloWorld.cpp”. These will be used within the next few steps.



**Figure 2:** Code for two programs used for the example project. Source: Primary

To check the status of the changes made in the Git repository, run this line in the command terminal:

**git status**

Upon running this line, a message will display on the terminal briefly explaining what files contain **changes to be committed** and what files are **untracked**. At this stage, any files created or changed will be marked as untracked, meaning they are in the project directory, but are not recognized under version control. This is illustrated in Figure 3 below.



**Figure 3**: results of running “git status” when changes were made to the code, but no work has been done to prepare to create a commit. Source: Primary

## **Step 4: Add Files to the Staging Area (Staging)**

Before a commit can be made in the Git repository, the files and their changes must be placed in the staging area. This allows for the ability to only add certain files to commit when more have been edited earlier. To add files to the staging area, run this line in the command terminal, replacing the quotes with the names of the files that were changed, and adding as many files as necessary:

**git add “file name 1” “file name 2” …**

Running the “git status” command again at this point shows the files moved from “untracked” to “changes to be committed”. Verify you have all the files needed to be recorded in the commit.

Figure 4 below shows the status of our files when only one file has been moved to the staging area. If the next step was completed at this point, only the changes made to the staged file will be recorded. Figure 5 on the other hand displays the status of both files when they are both in the staging area. Then the changes to both files will be recorded in the next step.



**Figure 4:** The status of the two files after only “helloWorld.cpp” has been staged. Source: Primary



**Figure 5:** The status of the two files after both “helloWorld.cpp” and “helloWorld.h” have been staged. Source: Primary

## **Step 5: Commit Changes (Committing)**

Now that files are in the staging area, they are ready to be committed. Add a commit to the Git repository using this line in the command terminal, replacing the quotes with a message describing the changes made to the files:

**git commit -m “Details about the commit”**

Now a commit is recorded in the hidden Git directory. Running the line “git status” in the command terminal displays nothing if all file changes were committed. A timeline of all commits created in the Git repository can be viewed by running this line in the command terminal:

**git log**

This command will display a list of entries representing each commit in the Git repository. Each entry will display these four items in order:

1. A commit hash, a long hexadecimal number representing the identifier of the commit in the Git repository.
2. Information about the author of the commit, including name and email, which were provided when configuring Git to the device.
3. A timestamp indicating when the commit was made to the repository.
4. The commit message provided by the Git user.

Figure 5 below shows what the terminal should look like when a commit is successfully created, in addition to the git log that displays the one commit created in the example project.



**Figure 5:** The results of successfully creating a commit for the project. Source: Primary

**Note**: repeat steps 3 through 5 as necessary throughout the development of the project. Make sure to follow these steps in that order so changes and commits can be recorded successfully.

## **Step 6: Create a GitHub Repository**

The history of the development of the project is now ready to be stored in the cloud on GitHub. However, a GitHub repository must be created before storing the information in each file of the project. To create a repository on GitHub, follow these steps:

1. Login to your GitHub account with your credentials.
2. Once in the main dashboard, navigate to the top header and click on the “+” sign.
3. A dropdown menu will appear. In this menu, click “New Repository”.
4. Fill in all the necessary details of the repository:
	1. Give a name for the repository.
	2. If needed, add a short description of what the project in the repository does.
	3. Set who can see your repository. If anyone can view it, click “public”. If you only want yourself and others to view it, click “private”.
	4. Click “Add a README file” if you want to provide a longer description of the project.
	5. Click “Add .gitignore” if you want to hide certain files from Git, and select the appropriate template you want to use.
	6. If needed, add an appropriate license to control what others can do with the code.
5. Click “Create repository” to finalize the repository.

## **Step 7: Create a Personal Access Token to Connect to GitHub via HTTPS**

## **SECURITY WARNING! A white triangle with a exclamation mark  Description automatically generated**

Please refer to the security warning about passwords on page 2 before proceeding.

Before pushing the commits from the Git repository into GitHub, a communication token must be created. There are two options to create this: creating an HTTPS token or setting up SSH keys. For this guide, an HTTPS token will be created, and it will be the classic version. Here is how to create one:

1. Verify your GitHub email if you have not already done so.
2. Click on your profile photo on the upper-right hand corner of any page.
3. A drop-down menu will appear. In this menu, click “Settings”.
4. Scroll down to the bottom of the left sidebar and click “Developer Settings”.
5. On the left sidebar of the next page, click “Personal Access Tokens”, then click “Tokens (classic)”.
6. On the next page, click “Generate new token”, then “Generate new token (classic)”.
7. If directed, enter the password for your GitHub account.
8. On the form, input the following information:
	1. In the “Note” field, enter a short description on what the token is for.
	2. Click on “Expiration” and select a timeframe for how long it will take for the token to expire.
	3. Select the checkboxes under “Select scopes” to determine who can access personal tokens.
9. Click “Generate Token” on the bottom of the page.

**ACCOUNT WARNING! **

Before leaving, be sure to copy the new token upon competition before leaving and keep it somewhere secure. The token can only be viewed once. Failure to copy the new token immediately will result in the inability to push and pull commits between the device and GitHub.

## **Step 8: Set Up GitHub Repo as a Remote for the Local Repo**

Now that a working directory and a GitHub repository have been created, a connection between the two can now be created. To start, the GitHub repository must be registered as a **remote** within the local repository. Here is how to create one:

1. On GitHub, navigate to the new repository.
2. In the blue box, verify that the “HTTPS” button has been pressed.
3. Copy the repository URL to the right of the “HTTPS” button to the device clipboard.
4. Navigate to the working directory in the software development platform and open a new terminal.
5. Type in the following line, replacing the quotes with the HTTPS URL copied earlier:

**git remote add origin “HTTPS URL”**

A remote should be set up to connect the working directory with the specified GitHub repository. It will be named “origin”, indicating it is the default remote for the repository. To verify one was created, run this line in the terminal:

**git remote -v**

Figure 6 below shows the results of successfully creating a remote from the example project directory to the GitHub repository used. Notice the name of the remote is at the beginning of each of the two output lines, while the URLs to the GitHub repository follow.



**Figure 6:** The results of successfully creating a remote between the example project directory and the GitHub repository. Source: Primary

## **Step 9: Push All Changes to the GitHub Repo**

With a remote set up, Git commits can now be pushed into the GitHub repository. Before continuing, verify the local repository is using the correct branch name (main) with this line in the terminal:

**git branch -M main**

Type in this line into the terminal to push the current commit history into the Git repository:

**git push -u origin main**

The “git push” command is responsible for pushing the commit history into GitHub. By adding the “-u” flag into the above line, a permanent connection between the local and GitHub repositories is established. Hence afterwards, when “git push” is typed into the terminal, the commit history on GitHub will be automatically updated.

Figure 7 shows the output lines in the terminal after the above line is run.



**Figure 7:** result of running the first “git push” command into the GitHub repository. Source: Primary

To verify the commits were pushed to GitHub successfully, navigate to the GitHub repository, refreshing the page if necessary, click on the button on the upper right hand corner of the list of files, and view the entire list of commits from there. An example list is shown in Figure 8.



**Figure 8:** example list of commits on GitHub, which shows only one. Source: Primary

## **Step 10: Pull the Most Recent Commits From GitHub**

Git is often used when a project needs to be completed from different devices. Extra care must be taken to make sure the most up to date code is being changed. When working with multiple devices or people, always verify you are pulling all the changes on GitHub. To do this, run the following command in the terminal:

**git pull**

This will pull all commits from the GitHub repository and into the working directory.

### Fixing Issues with Pulling Commits

If changes are made to code before pulling from GitHub, an error will be returned. The easiest way to fix this issue is to stash the changes made to the files. To stash any changes, run this line into the command terminal:

**git stash**

This will revert any changes made since the last time commit history was updated in the working directory. These changes will be stored on the computer for future use.

Once the above command is run, run the “git pull” command in the terminal to update the code. To add in the changes from the stash, run the following command in the terminal:

**git stash pop**

**Note:** the “pop” argument added in this command is used to update the code with the last set of stashed code. Therefore, if you run “git stash” twice, run “git stash pop” twice to access the changes from the first set of stashed code.

## **Bonus Step 1: Branches**

Up until this point, we have been pushing commits onto GitHub through the main **branch**. When working with teams on GitHub, we may have to create our own branches before pushing all commits into the main branch. This is frequently used in an environment where code review is implemented. Information about how this workflow works will be covered in the rest of this manual.

To start, open the command terminal and type in the following command:

**git branch**

This will display a list of the current branches in the repository. It will also mention what branch is currently being worked on in the working directory (marked with a \*).

To create a new branch in your working directory, run the following command, replacing the quotes with the name of the branch you want to create:

**git branch “branch-name”**

Now if you run “git branch” again, you should see the name of your new branch. However, you will notice you are still on the branch you were in before you ran the above command. Should you wish to switch to this new branch, run the following command:

**git checkout “branch-name”**

You will also want to push this branch into GitHub, but you will need to set it up with the remote first. If you need to create a new remote from your working directory to the GitHub repository, create a new one (see [step 8](#_Step_8:_Set)). Once a new remote is created, run the following command:

**git push –set-upstream “remote-name” “branch-name”**

This will push the new branch and its files to the GitHub repository. Should you remain on this branch, make new changes to your files, and run “git push”, the changes will be pushed into this branch.

Note there are a couple more processes to discuss where we can merge these new branches with the main branch. These will be covered in the final two sections below.

## **Bonus Step 2: Pull Requests**

**Pull requests** are a feature in GitHub that allows contributors to a project to see what others are contributing before merging their changes into the main branch of the project. When a pull request is made by someone, someone else can examine what changes are being made, and the reviewer can either accept or reject these changes, while also providing verbal feedback on the code.

To create a pull request, follow these steps below:

1. Push all changes to your branch on GitHub.
2. Navigate to your project repository on GitHub.
3. In the top tab bar of the website, click on “Pull requests”.
4. Click on the green “New pull request” button.
5. In the top box, click on the “base:” box, and a dropdown menu will appear. Select the branch you want to push changes to.
6. In the same box, click on the “compare:” box, and another dropdown menu will appear. Select the branch you are pushing changes from.
7. Click on the green “Create pull request” button.
8. Add a title for the pull request and a description detailing what changes are being made.
9. Click on the green “Create pull request” button.

Once the above steps are completed, you will be taken to the conversation page for your new pull request. This summarizes the pull request, lists the commits being added, and provides a forum for the general discussion of the pull request. This is where we will start doing code review, which is discussed in the final section below.

## **Bonus Step 3: Code Review**

After a pull request is created, a reviewer can examine the changes being made and provide feedback before merging the individual’s branch into the main branch. This helps to make sure high-quality code is implemented, in addition to everyone understanding the functionality of the code itself.

**Note:** In order for code review to work, you must make your GitHub repository public to implement code review for free.

To make code review mandatory for the project, you can implement a **branch protection rule**. To create one specifically for code review:

1. Navigate to your project repository on GitHub.
2. In the top tab bar of the website, click on “Settings”.
3. On the left-hand menu, click on “Branches”.
4. Click “Add classic branch protection rule”.
5. Enter a name for your branch protection rule.
6. Scroll down through the menu, and check off the following options:
	1. “Require a pull request before merging”
	2. “Require approvals”
	3. “Require status checks to pass before merging”
7. Click “Create” at the bottom of the page.

Now code review is enforced on the project, and all contributors must create a pull request on their own branch and have it be approved by someone else before it can be merged into the main branch.

Now say a pull request has been created. To perform code review on someone else’s work:

1. Navigate to your project repository on GitHub.
2. In the top tab bar of the website, click on “Pull requests”.
3. Click on the name of the pull request that needs to be addressed.
4. Click on the “Files changed” tab.
5. Click on the green “Review changes” button.
6. Leave comments to provide feedback in the text box given.
7. Choose from the following options:
	1. “Comment”: submit general feedback without approving the pull request.
	2. “Approve”: submit feedback and approve the changes from the pull request to be merged into the main branch.
	3. “Request Changes”: submit feedback to suggest appropriate changes, thus not approving the pull request.”
8. Click “Submit Review”.
9. If you approved the pull request and there are no merge conflicts with the main branch, click on “Merge pull request”.

By following these steps successfully and approving the pull request, you will merge all changes from the feature branch into the main branch on GitHub. If you requested changes to be made, the pull request will still be open. It is important to note that commits updated to the pull request every time the publisher of the pull request pushes new changes into the feature branch.

**IMPORTANT!**

When new changes are pushed into the main branch, it is important to pull all changes into your working directory! See [step 10](#_Step_10:_Pull) for details.

## **Reference:**

[1] R. Hess. CS 362. Class lecture, topic: “Git and GitHub: An Individual’s Perspective.” College of Engineering, Oregon State University, Apr. 29, 2024.

[2] R. Hess. CS 362. Class lecture, topic: “A Team-Based Workflow for Using Git and GitHub.” College of Engineering, Oregon State University, May 13, 2024.