

Tutorial – Git Basics

This tutorial reviews how to initialize a Git repository and the regular “add, commit, and push” workflow that you should be performing regularly while working on your project.

Prerequisite: Git Clients and Git Repository Providers

Git Client (GUI or CLI)

Before starting this tutorial, ensure you have a Git Client installed on your computer.

If you are on an AIE classroom computer, one should already be installed for you. Ask your instructor for clarification on which client you should be using if one is designated as such.

If on a personal PC, you should pick from one of the following:

- [Git for Windows](https://gitforwindows.org/) (CLI-based) via <https://gitforwindows.org/>
- [GitKraken](#) (Graphical/GUI)
 - NOTE: Requires a paid/student subscription for use with private repositories.
 - Learners can claim a student subscription via GitHub’s [Student Dev Pack](#) program.
 - Trainers can claim a teacher subscription via GitHub’s [Teacher Toolbox](#) program.

These instructions are aimed to be tool agnostic and should be usable on a variety of different Git clients. For specifics, refer to your program’s documentation (e.g. [Git for Windows](#) and [GitKraken](#)).

Git Repository Provider

You will also need an account with a Git Repository provider. The de facto standard for individuals not hosting their own is [GitHub](#), which provides free public and private repositories.

Prerequisite: Configuring Your Identity

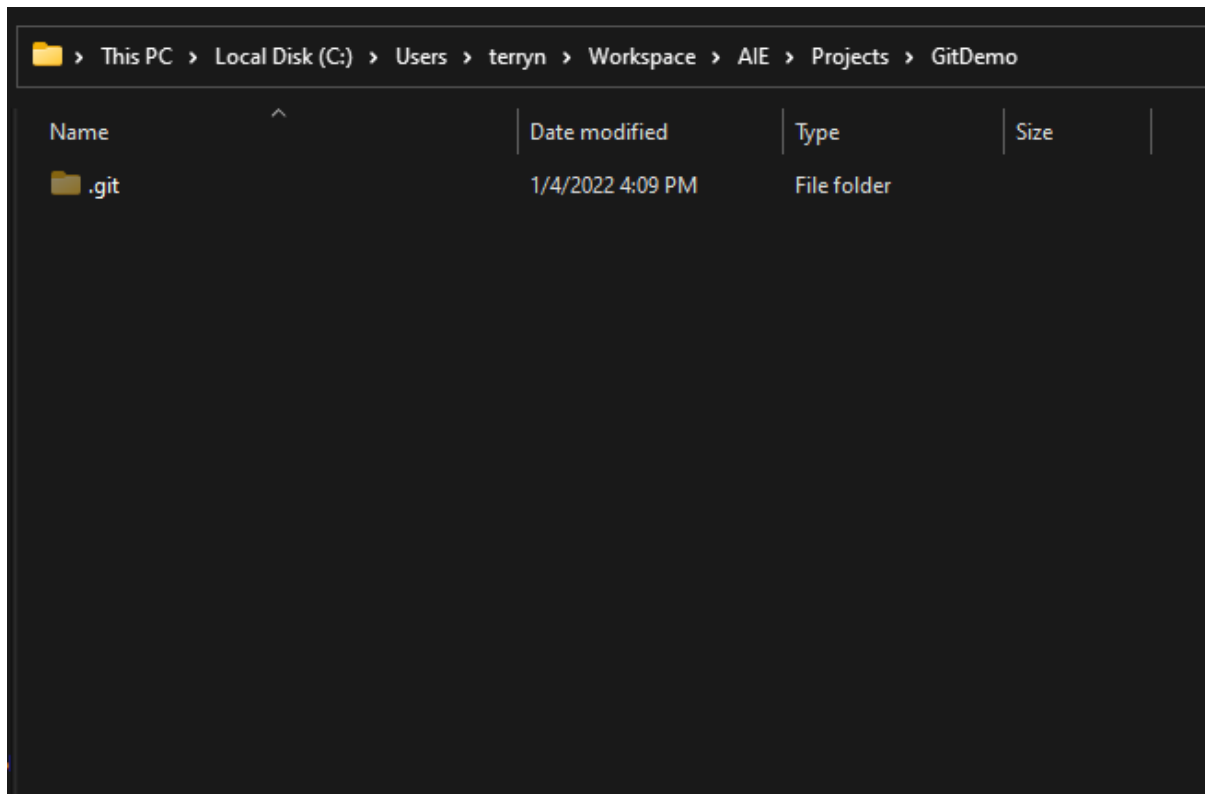
Git clients need to know who you are to attribute your work to you. Ensure that you have provided your name and your e-mail address (the same as your GitHub account) to your Git client before proceeding.

Initializing and Working with a Repository

Creating a Repository

Before you can put a project “on Git”, you need to create a repository to put it into. This can occur both locally (on your computer) or remotely (on a service like GitHub). These instructions will walk you through on doing it locally.

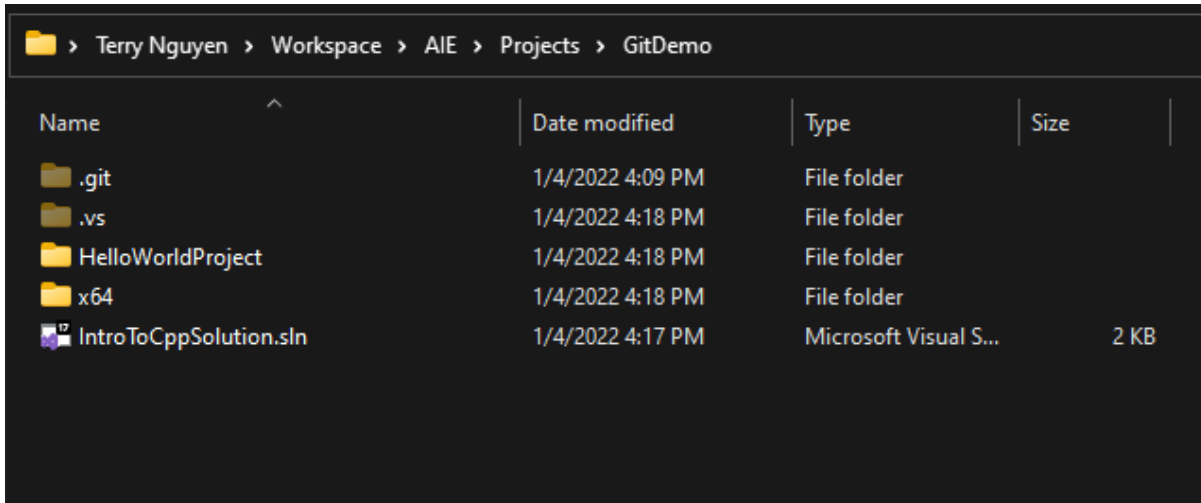
To start, **create an empty Git repository** using your Git client. It should create a new and almost empty folder of its own (with no more than a single “.git” inside of it, which is normally hidden).



A screenshot of the folder created by the repository, referred to as the “working directory”

Anything you want to add to your Git repository, you add to this folder (aka the repo’s “working directory”). Future Git commands will look through this folder for any changes such as new or modified files in your project.

If you have a Visual C++ or Unity project to “add to Git”, you would move it to the working directory and discontinue the use of the original file location. For tutorial purposes, let’s create and copy a C++ project into this folder.



| Terry Nguyen > Workspace > AIE > Projects > GitDemo | | | | |
|---|------------------|-----------------------|------|--|
| Name | Date modified | Type | Size | |
| .git | 1/4/2022 4:09 PM | File folder | | |
| .vs | 1/4/2022 4:18 PM | File folder | | |
| HelloWorldProject | 1/4/2022 4:18 PM | File folder | | |
| x64 | 1/4/2022 4:18 PM | File folder | | |
| IntroToCppSolution.sln | 1/4/2022 4:17 PM | Microsoft Visual S... | 2 KB | |

We now have a C++ project in the working directory of the project, but it is not yet part of the Git repository. To put the project into Git, we need to **add** the files and **commit** the work into the Git repository. Once that's done, we can **push** that commit somewhere like GitHub.

.gitignore

Before we do any of that though, we need to tell Git which it should ignore or exclude when determining which files should be added to the repository. This is done by adding a .gitignore file to the repository, often times at the root directory of the directory (at the top), which contains a set of rules on what files/folders to ignore.

! Important – Do Not Skip This Step

The **.gitignore** helps prevent you from accidentally commit sensitive files or autogenerated files that can cause issues with Git, other people's copies of your projects, or your own ability to work on your project.

If you are ever unsure where to find or how to write one, ask your instructor for help.

Generally, you want to ignore anything that fits that following the criteria:

- Can be regenerated from project files on a new machine, like builds
- Contains user-specific data, like user preferences
- Contains sensitive information, like passwords or API keys
- Contains data generated by tools that isn't meant to be shared like cache files

If we were to apply those rules to our Visual C++ project, we'd get something like this:

```
# User configuration files
*.user

# Tool generated files
.vs/

# Build artifacts
x64/

Debug/
Release
```

Comments are provided using lines prefixed with the pound (#) symbol.

We ignore files ending in .user because they are probably specific to that user.

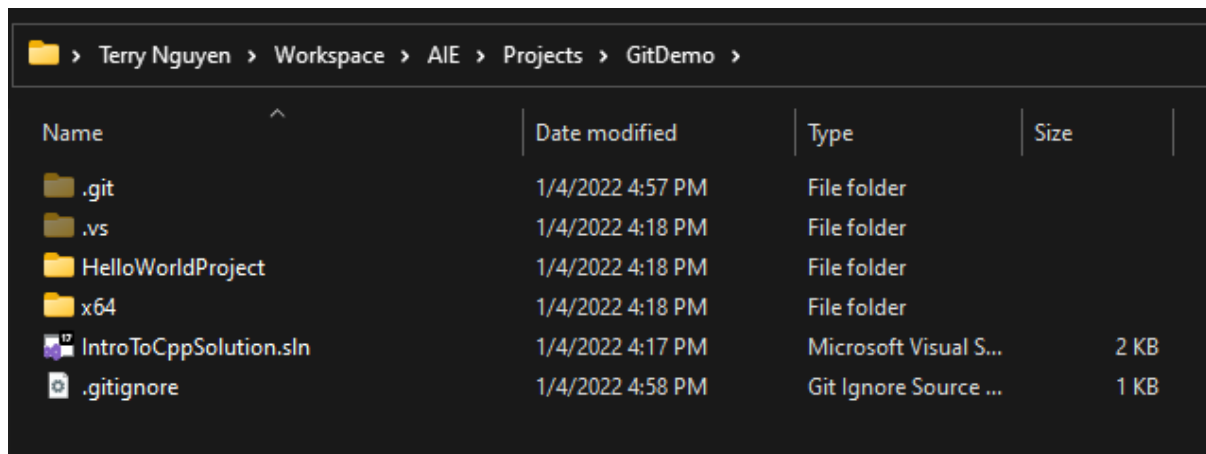
We ignore the .vs/ folder because it contains large cache files used to help VS understand your project. Although helpful, it's best to have VS automatically regenerate this when you or a teammate opens this project on another computer.

We ignore the build artifacts (your EXEs, your build OBJ files) because you can always regenerate these by building the project again. As such, they don't belong on version control.

Adding the .gitignore to the Repository

Create a file called ".gitignore" without the double quotes in the root working directory of your repository. Open it in a text editor like Notepad or Visual Studio and add the rules provided above.

When you look at your project directory, it should appear as follows:



| Name | Date modified | Type | Size |
|------------------------|------------------|-----------------------|------|
| .git | 1/4/2022 4:57 PM | File folder | |
| .vs | 1/4/2022 4:18 PM | File folder | |
| HelloWorldProject | 1/4/2022 4:18 PM | File folder | |
| x64 | 1/4/2022 4:18 PM | File folder | |
| IntroToCppSolution.sln | 1/4/2022 4:17 PM | Microsoft Visual S... | 2 KB |
| .gitignore | 1/4/2022 4:58 PM | Git Ignore Source ... | 1 KB |

Ignoring Files vs. Ignoring Folders

We can ignore files by name, extension, or some combination thereof.

For example, to ignore all “.junk” files, we would add the following rule:

```
*.junk
```

A wildcard “*” is used for the name of the file while the file extension is “.junk”. This means that files with *any name* will be ignored as long as they have the “.junk” extension.

To ignore a folder, we specify a forward slash “/” instead of an extension. For example, to ignore all “Debug” folders, we would add the following rule:

```
Debug/
```

The folder name is “Debug” which is followed by a forward slash, indicating that all “Debug” folders will be ignored.

Advanced .gitignore Rules

The .gitignore syntax is robust and supports more features like additional forms of wildcard matching, limited scoping, and even the ability to create exceptions to previously defined rules. This is out of scope for this tutorial, you can review this online in the Git Reference manual: [gitignore](https://git-scm.com/docs/gitignore).

github/gitignore

For future reference, you can consult [github/gitignore](https://github.com/github/gitignore), a public Git repository containing a large collection of community maintained .gitignore rulesets to use when starting a new project.

Tools like GitKraken integrate this into the repository creation process, allowing you to automatically download and place the correct one in your project.

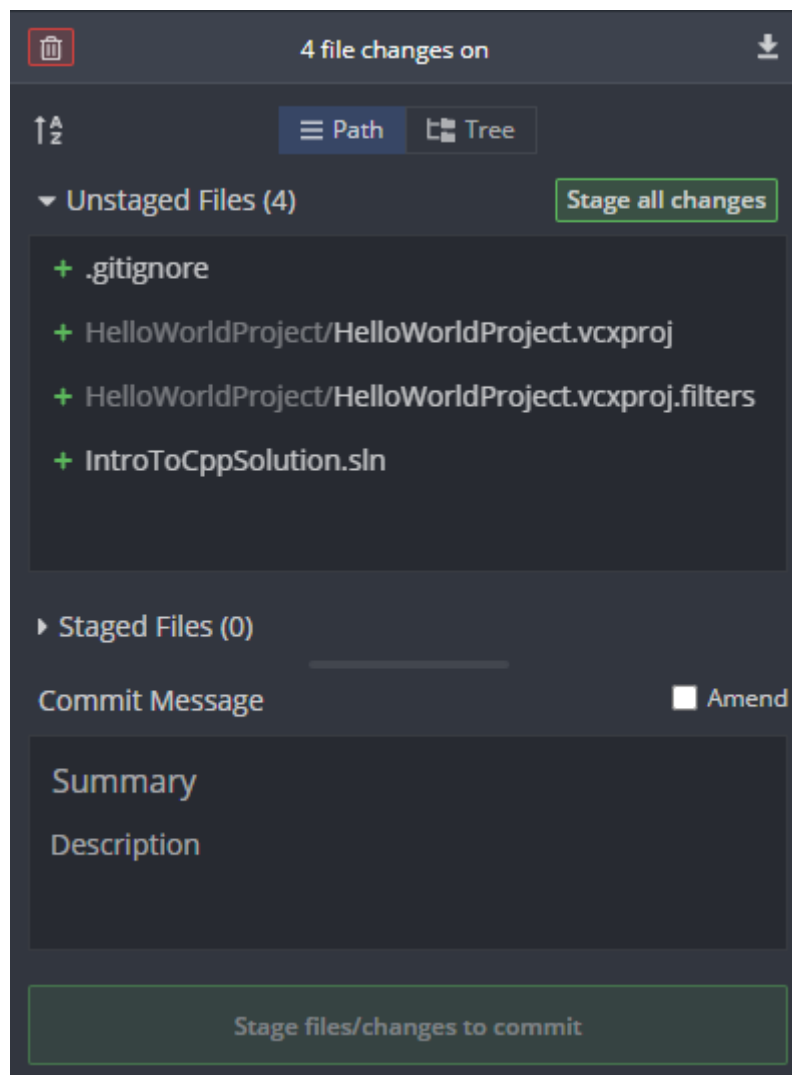
Add, Commit, Push

Once you've created your Git repository, added your project files, and added a gitignore file (do not skip the .gitignore file!), you're ready to **add**, **commit**, and **push** your work.

Adding Work

It's not enough to just drop the files into the working directory. To tell Git which files we want to include in the repository, we need to “**add**” or “stage” those files to be committed in the next commit.

Tools like GitKraken will display this in a panel, indicating which files are unstaged (will not be included in the next commit) and which files are staged (will be included in the next commit).



You can also see that the .gitignore we added earlier is working: folders like the x64/ folder are not being shown, as is the case for the .vs/ folder.

Go ahead and stage everything you want to include, which should be everything at this point in time.

Committing Work

When we push or pull to and from another repository like those found on GitHub, Git doesn't look at your files, but instead, your commits to determine what needs to be transferred.

By extension, this means that we need to commit our work to share it.

Once you've "added" or "staged" all your work, go ahead and provide the following:

- A very brief summary or message, describing the work you've done
 - This should be 50 characters or less.
- (Optional) A more in-depth description

These are visible to anyone who can see your Git repository (your team, potential employers you share it with), so make sure to write something descriptive and helpful.

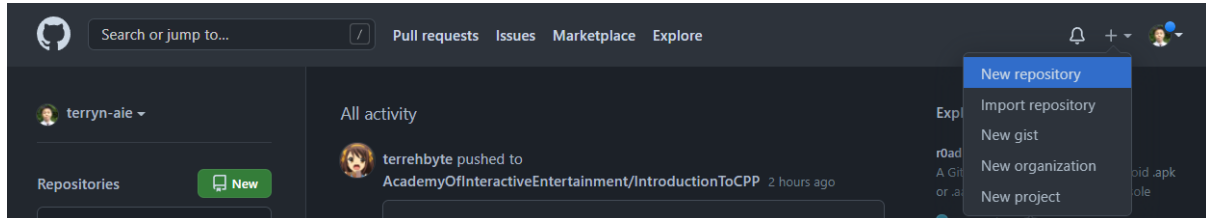
Commit messages are traditionally written in the imperative tone, like you would with a command:

- "Add initial project files"
- "Create a new repository"
- "Make the repo"

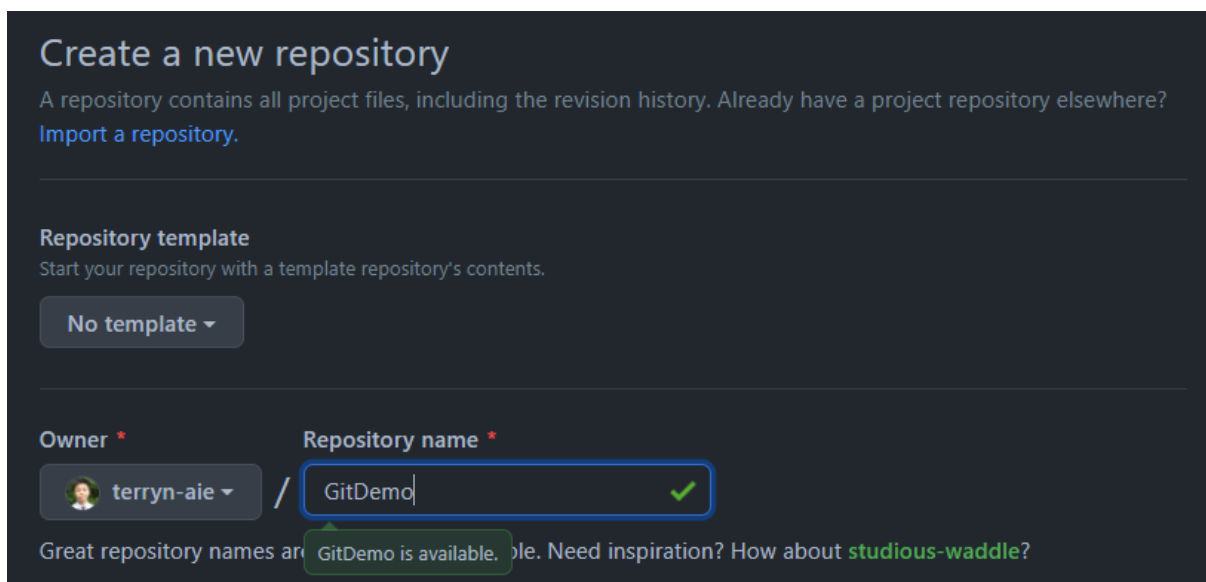
Pushing Your Work (First-Time)

The first time we push our work, we need to tell Git where to push it to. This is typically a shared repository on a host like GitHub, so let's visit their website to complete that part of the process.

Once signed-in, select “New repository” from the “New” menu located in top-right of the page:

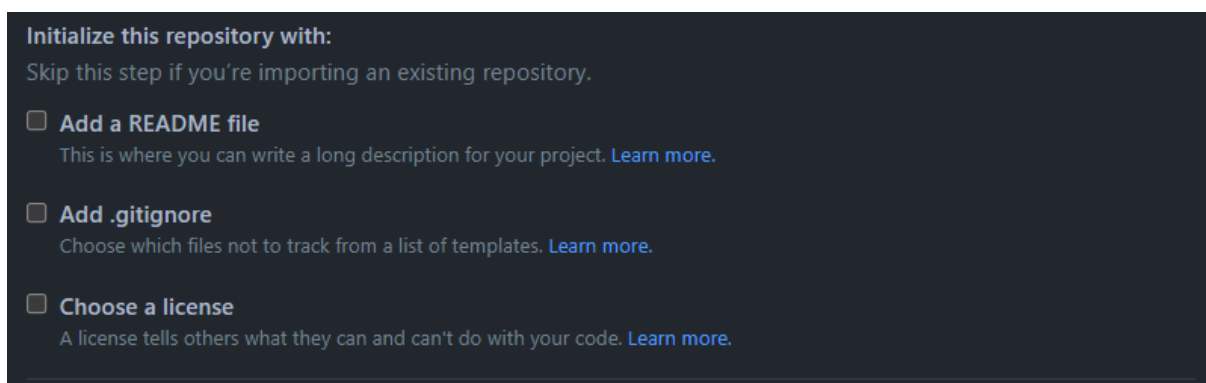


This page has a lot of options, but all you need to do for now is to give your repository a name. In this case, we'll call it **GitDemo**. This is what GitHub will use to refer to your repository in the future.



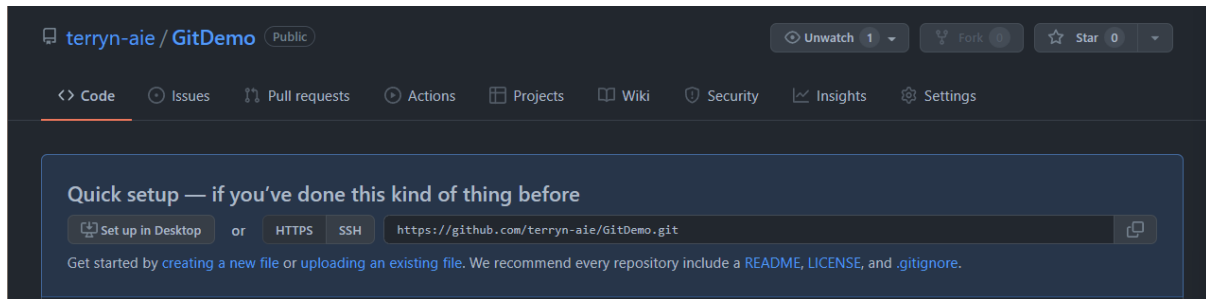
Repository names are only unique to the user, so no need to worry about name conflicts.

When prompted to initialize the repository with anything, leave all options blank and unchecked (we're initializing our repository locally, not on GitHub).



GitHub can initialize your repository for you, but we'll be skipping that this time.

Once you create the GitHub repository, you should be presented with setup instructions for working with an empty repository. It largely assumes you're working on the command-line, but has some information useful for GUIs as well.



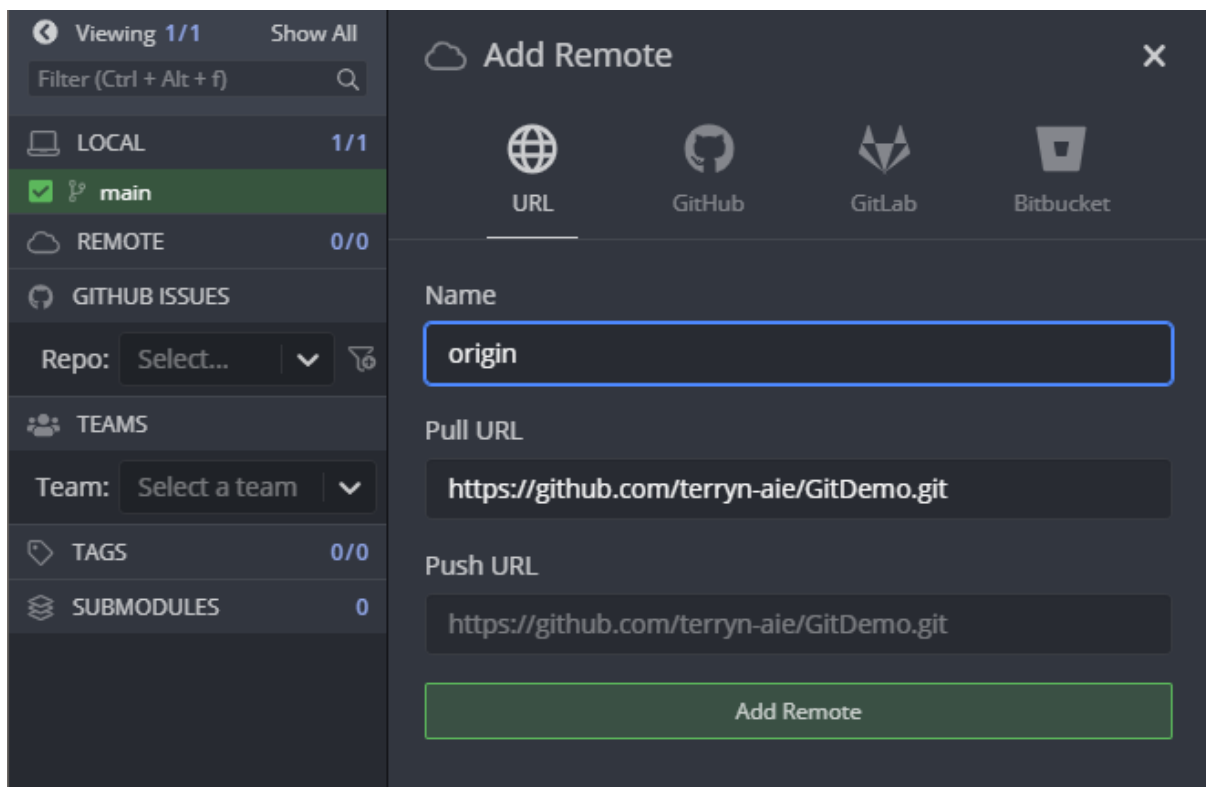
The onboarding instructions provided by GitHub.

What we're interested in is the "HTTPS" address that is formatted as follows:

`https://github.com/YOUR-USERNAME/YOUR-REPOSITORY-NAME.git`

This URL refers to the location of the remote Git repository located on GitHub's servers, which is the repository we want to start pushing commits to.

This is recorded in your local Git repository as a "remote", typically referred to as "origin".



Adding a remote called "origin" pointing to the newly created repository in GitKraken.

Once this is done, you can **push** your work to GitHub.

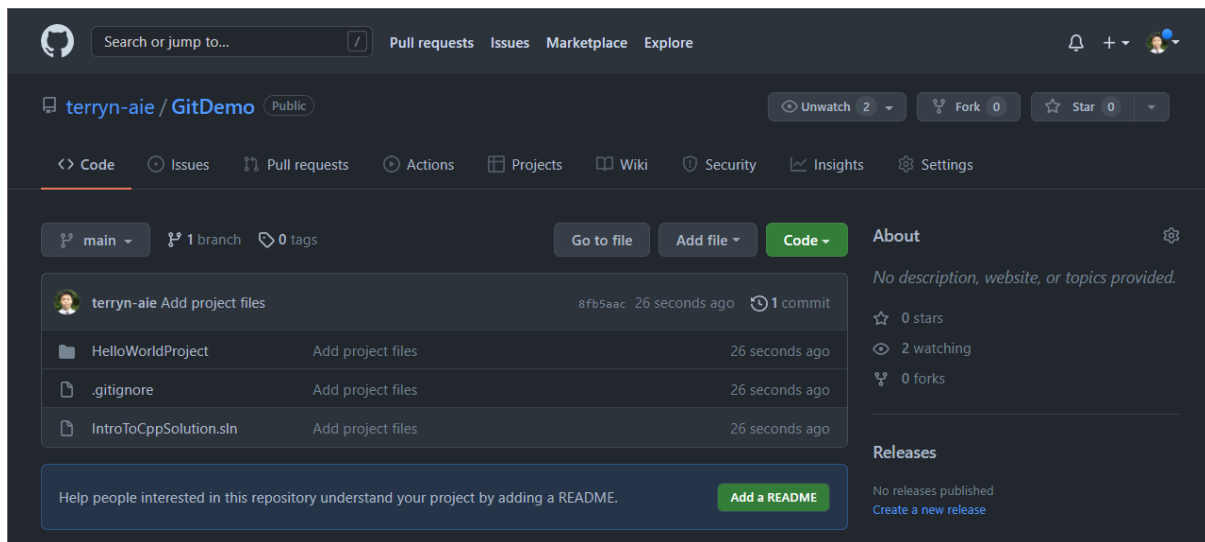
Since this is the first time you are pushing this branch, you will need to confirm with Git that you will want to associate your local “main” branch with the remote “main” branch. This might seem obvious, but Git is flexible (almost to a fault) so you’ll need to confirm this the first time.



A dialog box from GitKraken asking 'What remote/branch should "main" push to and pull from?'. It has a dropdown menu set to 'origin' and a text input field containing 'main'. To the right are 'Submit' and 'Cancel' buttons.

Graphical clients like GitKraken may visually prompt you to confirm this.

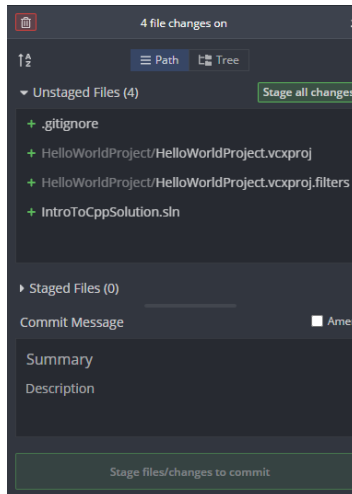
You should now be able to refresh your repository on GitHub and review your work.



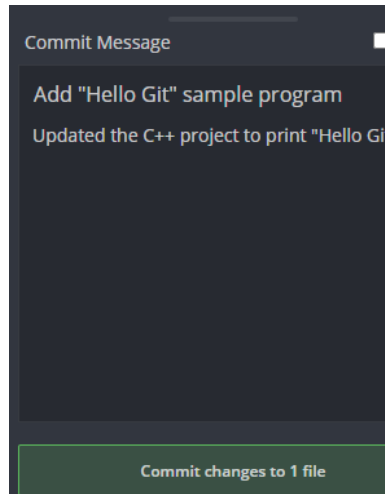
The GitHub repository now presents the repository as of its latest commit.

Git Workflow

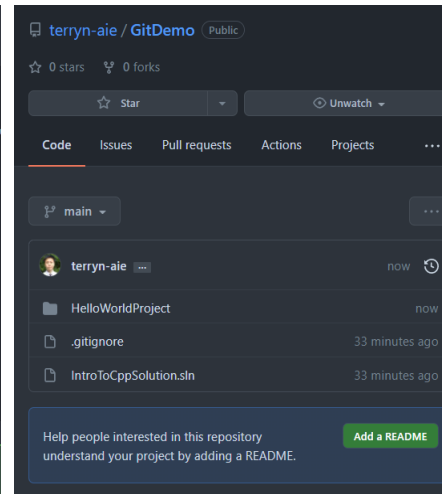
From this point on, you should be able to repeat the process of adding, committing, and pushing your work to a GitHub.



Add Your Work



Commit Your Work



Push it to GitHub

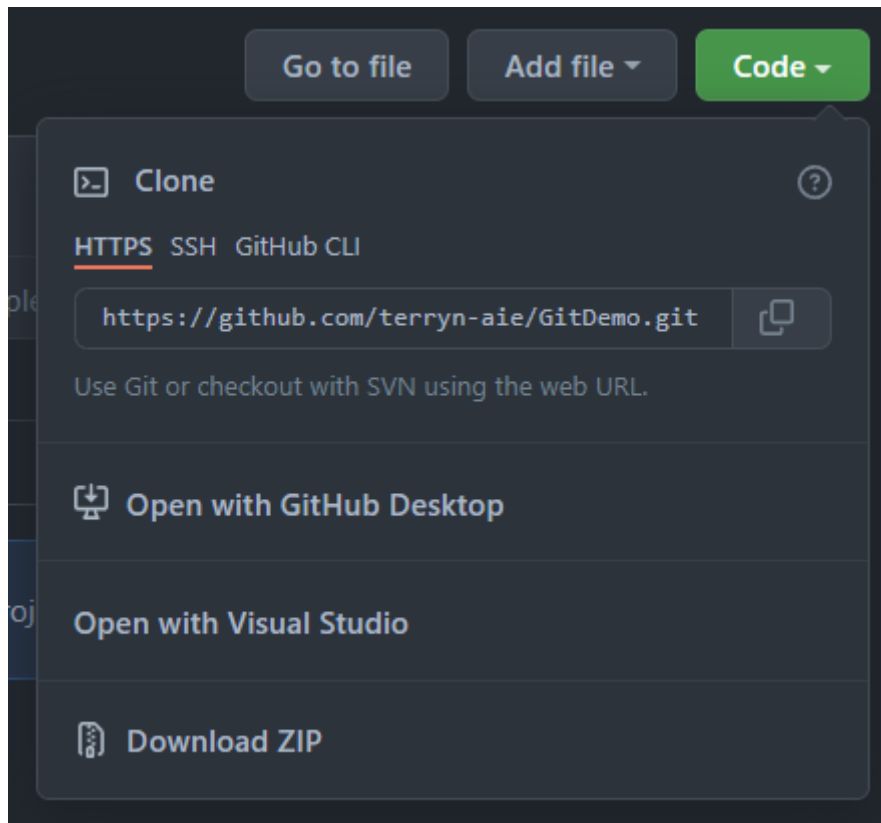
You should do this whenever you have a significant amount of work that you've completed, such as fixing a bug or building a new part of a feature you're working on. It's not unusual for work to be split up across multiple commits.

The process is the same as demonstrated above, only you don't need to specify the remote again. Git will remember the next time you push.

Cloning an Existing Repository

To clone an existing repository from GitHub (whether from you or a friend), you will need the URL to the Git repository (same as the one you'd use to push to it).

To find this on GitHub, visit a repository and select the green “Code” button, which reveals to you a “Clone” panel.



The “Code” panel also provides integrations with other tools, like GitHub Desktop or Visual Studio.

Copy the HTTPS clone URL (or the SSH if you know what you're doing) and provide it to the clone action in your Git client. It should automatically create a folder for your repository which will serve as the working directory for that repository.

You should then be able to add, commit, and push with it as though you had originally created that repository yourself.