# GitHub for saving and syncing your work

## Learning Objectives

Understand what a GitHub repository is

How to set up and clone a remote repository

Use GitHub desktop to sync work to the remote repository from your device

## What is GitHub and why are we using it in this course?

* Git is a source control management system used in software development, originally developed by the creator of Linux. Developers typically utilise Git to capture and store their code, collaborate with colleagues, and manage changes in software from small to large projects.
* Git is distributed – allowing work on content in a local copy of a repository, sync (push) that work to the remote, browser-based repository and have full version control and roll back options for documents, data sets and code files, across devices.
* GitHub, bought by Microsoft in 2018, is the biggest repository hosting service for Git. GitHub includes capabilities for continuous development, including GitHub Desktop which is an easy-to-use end user application for Git that is already installed on your lab environment.
* As you will be working on course materials and projects in a lab environment with a time limit, we will use GitHub and GitHub Desktop to facilitate the rapid sync of files between lab environments and your online repository. This will ensure you don’t lose any work along the way, will have a way to collaborate with your classmates and will have a long-term secure store of your work on this program.

Remote Repository (GitHub)

Local Repository - GIT (GIT)

Local Repository - GIT

Local

Repository - GIT

PUSH

PULL

PUSH

PULL

PUSH

PULL

Figure 4 - Distributed repository model

## Set up your GitHub account and online repository

* Go to GitHub and create your profile

In any browser, go to [github.com](http://www.github.com) to sign up for a free personal account- you will need to use an email address and create a password. You must validate your email to get beyond this stage.

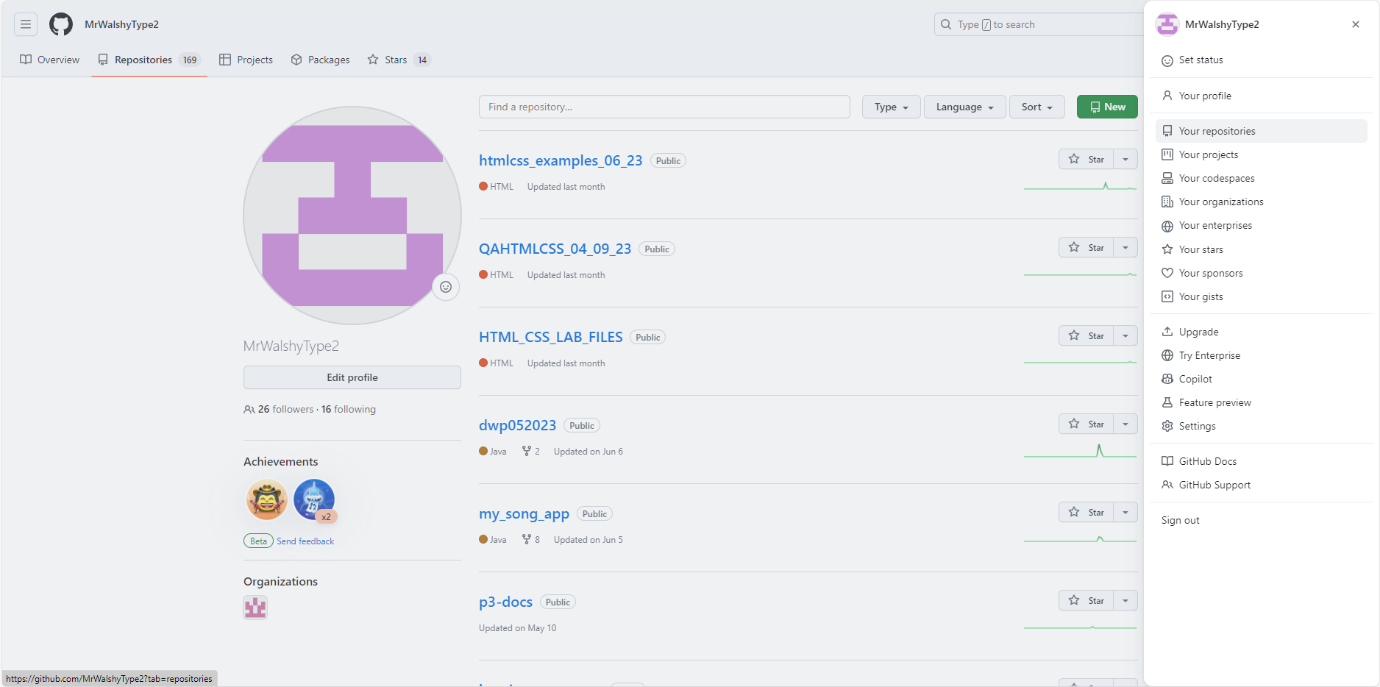
A screenshot of a computer

Description automatically generated

* Create a new repository

Once your github profile is activated, you can create your first repository. You first open the list of your repositories and then use the green **New** icon to create the repository.

The repository requires a name (words and numbers, no spaces), optional description and you need to decide if this will be public or private. Private repos require you to explicitly give anyone else permission to access your work.



1

2

A screenshot of a computer

Description automatically generated

We recommend you create a readme file so that you can add a detailed description of the repo contents at a later date. Also, it worth using the python .gitignore template from now on as we are going to do lots of work in python.

A screenshot of a computer

Description automatically generated

## Using GitHub Desktop

* Clone your repository

Once your repo exists in GitHub it is ready for cloning – this means copying the directory structure to your local machine. This means when you save work locally you can push it neatly to the remote location.

Use the <>Code menu on the repository to open with GitHub Desktop which will require you to sign in with your github credentials.

A screenshot of a computer

Description automatically generated

Or open GitHub Desktop, sign into GitHub.com and Clone the repository from the File menu.

A screenshot of a computer

Description automatically generated

* Where is your local copy?

Once cloned you can see the location of the repo in your machine by hovering over the repository name in the top left corner of the GitHub Desktop interface. You should use this directory for all of the work you will do and copy any relevant class / project files into here (Username\Documents\GitHub)

A screenshot of a computer

Description automatically generated

* Pull before pushing

One note of caution – if you change anything in your remote (online repository) it wont automatically sync here and its easy to forget you have changed something online, then it change it locally also – getting things messy in the process. So we work on a principle for GitHub of ‘pull before you push’ and you can easily keep your local repo up to date with the remote version by using the Fetch origin icon – this works like a refresh, and makes sure any online changes are reflected locally. Get in the habit of hitting that fetch origin button whenever you open GitHub Desktop.

A black rectangular sign with white text

Description automatically generated

## Your daily GitHub Desktop workflow

1. **Save changes to the content (local)**
2. **Pull and capture any remote changes locally**
3. **Prepare to commit the changes**
4. **Push to sync the changes to the repository (remote)**

**1. Save changes to the content (local)**

You will developing new content and making changes to files in your local github repo directory on the C drive. Lets say you have added a text file into your directory called example.txt

A screenshot of a computer

Description automatically generated

When you have saved the file, with some text typed in, go to GitHub Desktop.

**2. Pull and capture any remote changes locally**

First, hit the fetch origin button to ensure your repo is synced with any changes made to the remote version

A black rectangular sign with white text

Description automatically generated

Now review the changes proposed for the text file – you should see 1 change proposed - the file is created and your text is visible- in green (means everything is good) – and ready to commit. You can choose to tick or untick changes here – so you don’t have to commit everything in one go.

A screenshot of a computer

Description automatically generated

**3. Prepare to commit the changes**

Before committing files and changes to the remote repository, it is important to understand how Git handles tracking these files. Git uses a snapshot model for tracking the different commits of files, each commit representing a specific snapshot (version) of the project’s files at some point in its history. What this means is that each commit has its own unique copy of the files which show the contents of the files at the time the commit was created, this enables any commit to be reverted to or accessed rapidly.

#rfge44fd

#fsfdf33r

#f34f4erf

Commit number

Figure 7 - The snapshot model

Each snapshot, a commit, will be assigned a hash identifier which identifies the specific changes made, when the changes were made and who created the changes. If you wanted to see the contents of a specific file at some point in its history, you can then reference the associated commit hash to view its state at the point in time the commit was created. Each commit will also have an associated message, this is used to briefly describe the changes made in the commit and should be meaningful in nature. The commit message is added with a description in the bottom right side of the GitHub Desktop window.

A screenshot of a chat

Description automatically generated

**4. Push to sync the changes to the repository (remote)**

Once you commit the changes to main (this means the only branch of your repository – we are keeping it simple and using just one branch for now!) you will see the window has updated to indicate you are ready to push the commit to the remote repository (known as origin, because that’s where we first created the repository). From here you can navigate in your browser to the GitHub site.

A screenshot of a computer

Description automatically generated

Once in the online browser you will see the new file and the commit history.

A screenshot of a computer

Description automatically generated

# Learning Check

How confident are you in answering the following questions?

Consider your answers and then discuss with your trainer.

1. What is source control used for ?
2. What are the steps to clone a repo from GitHub to your virtual machine?
3. When you have updated work on a virtual machine, how do you sync that back to the remote repository?
4. Why does your commit message need to be meaningful ?
5. Whats first in the GitHub Desktop workflow, push or pull?

---------------------------------------------new document -------------------------------------------------

# Additional useful GitHub concepts

## Cloning versus forking

### Cloning

In Git, a clone of a repository is a full copy of nearly all data contained on the server hosting the repository; this has particular benefit when it comes to a critical system failure, the clones on a local machine can be used to (nearly always) provide a whole backup of the original repository should the server hosting the original repository fail. You can get the URL to use for the clone from the GitHub repository:

A screenshot of a computer

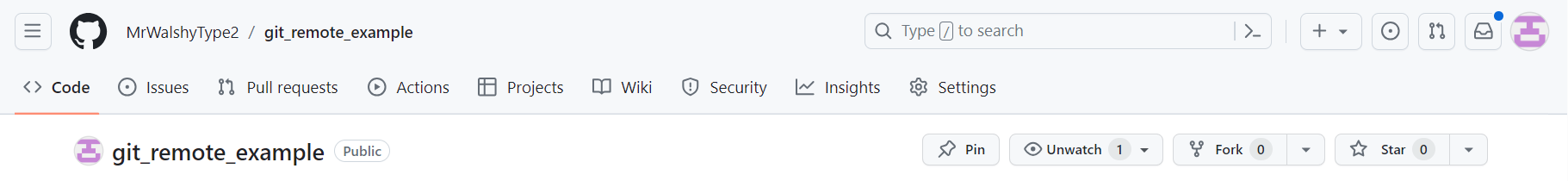
Description automatically generated

You should also pay attention to which URL you use for cloning; public repositories can be cloned using the HTTPS or SSH URLs; a private repository will require you to use the URL which matches the authentication method you choose. If you use personal access tokens (PATs), the HTTPs URL is needed for private repositories. If you use SSH public-private key pairs, the SSH URL is required instead for private repositories.

### Forking

A fork is a GitHub specific concept where a new repository is created in your account which shares the files, branches, and visibility settings of the original upstream repository. This allows for making changes to a project without affecting the original and is popular in open-source development. Once a fork has been created, you can then fetch updates from the upstream repository or propose changes from the fork to the upstream repository using a pull request.

Creating a fork of a repository is simple, all it requires is that you have the permission to do so. All public repositories enable forking, keep in mind that this does not remove the licence assigned to the project and you must still abide by any legal obligations indicated by said licence agreement. After navigating to a project, there will be a Fork button which you can press to start the process of forking:



The next screen will open a form for creating the fork, you will supply the account to make the fork in, the repository name, and a description. You can also specify whether to copy all the branches from the upstream repository or just the main branch. Click the Create fork button once ready and GitHub will make a copy of the repository for you in your account which you can then clone to your local machine to work on. You will have full ownership of this copy, within the legal boundaries outlined by the licence agreement.

A screenshot of a computer

Description automatically generated

## The gitignore file

The .gitignore file is a special file within a Git repository which can be used to specify files to ignore when committing files from the local drive. To ignore files, specify a root relative path to the file or a directory – root relative meaning from the root of the repository. If you had a repository with the following structure:

Repository

| file1.txt

| file2.txt

| -- somedir

| file3.txt

| file4.txt

You can ignore file1.txt and the somedir directory using a .gitignore file (name the file .gitignore) with the following contents:

file1.txt

somedir/

Now, only file2.txt can be added to the repository.

This is a useful technique when working with different coding technologies such as jupyter and python which involve the inadvertent creation of installation, cache and checkpoint files locally. Fortunately, GitHub can create a gitignore file for us which highlights the common python files and directories to ignore. This can be edited at any point to add files that you don’t want to push to your repo for whatever reason, e.g. they contain sensitive information.

## Branching

Branching is a concept common to many version control systems (VCS), it is just the divergence of work away from the main line of code which represents the production system. This is to allow working in isolation from mission-critical code, lowering the chance for failure as the sideline code can be queued for testing before its integration into the main branch/line.

Git’s implementation of branching is more powerful, efficient, and faster than most other version control systems. Other systems often use a tree of differences being tracked between commits, Subversion (SVN) does this for example, whereas Git uses the snapshot model discussed in an earlier section. This is beneficial when branching as the files referenced by a branch are available near immediately instead of having to be built up from differences. In Git, a branch is merely a pointer to one of these snapshots (commits).

To properly understand branching, you also need to understand how commits and branches are managed under the hood. As previously mentioned, there is a hidden .git directory in a Git repository which holds a database of Git objects and data surrounding the project – this directory is the repository in reality. When you commit some work to the repository, Git will store a commit object containing a pointer to the snapshot of work that was previously staged (known as a tree). The commit object also contains:

- The author’s name and email address.

- The commit’s message.

- A pointer to the prior commit(s), if any. These are known as the parent(s). There will be no parent commit on the initial commit, one parent for a commit after this, and multiple parents when a commit results from merging multiple branches.

A visualisation of this structure can be seen below ; take note of how each commit points to its parent (the prior commit) and a tree which represents the different blobs of data stored reflecting a specific snapshot.

The Feature Branch Workflow, often confused with the Gitflow Workflow, is focused on encapsulating change away from the main branch of code in so-called feature branches and extends the Centralised Workflow.

a43sd

tree: 908ca

author: Bob

committer: Bob

parent:

bd32d

tree: af83a

author: Bob

committer: Bob

parent: a43sd

af3s5

tree: 43jo3

author: Bob

committer: Bob

parent: bd32d

**Oldest**

**Newest**

Figure 11 - Commits visualisation

A feature branch represents some work on the main code base isolated away from the main branch so as not to introduce and minimise the chance of errors getting into the project’s main codebase. When creating a feature branch to work on, it should have a descriptive name indicating its purpose. Feature branches should also be pushed up to GitHub, this sets them up as candidates for pull requests which are safer than locally merging changes and pushing them up to the main branch on the remote repository. Figure 3 from an earlier section visualises this feature branch model.

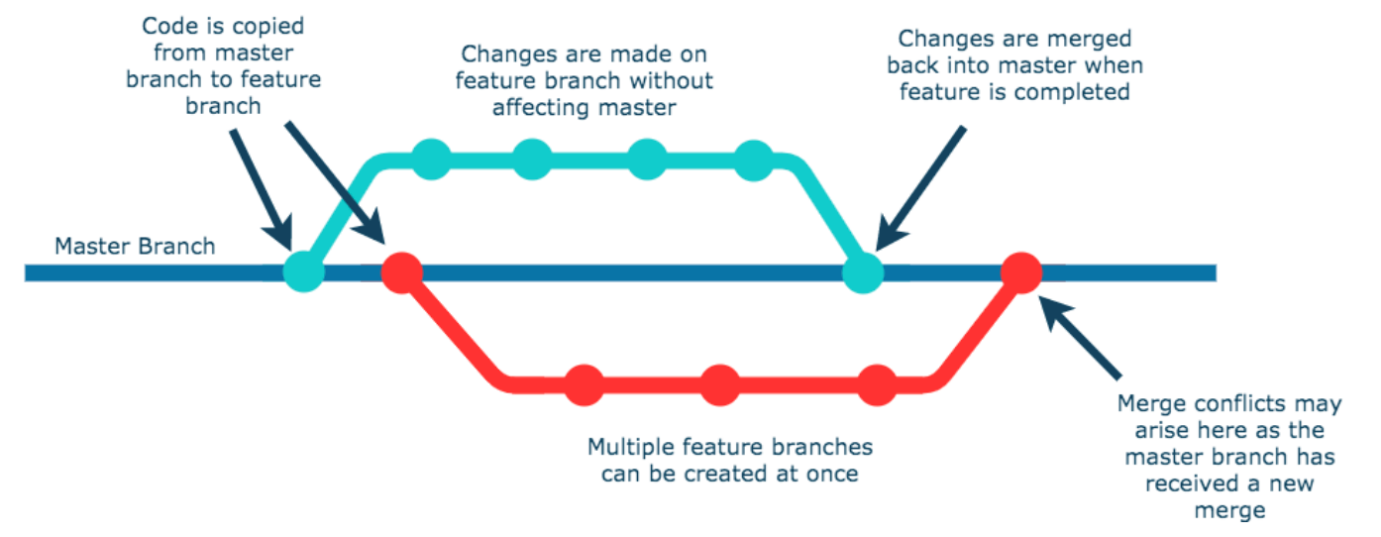


Figure 16 - Using branching to manage change

## How branching fits into the Gitflow Workflow

The Gitflow Workflow is an extension of the Feature Branch Workflow, it doesn’t introduce any new concepts related to Git and GitHub themselves but does enforce the usage of certain branches to help protect and organise the code base more than just feature branches alone can. These branches are focused around:

- Separating the main production code base from the development code base

- Using specific branches for releases, hotfixes, etc…

As indicated, the Gitflow Workflow operates using two main branches – a main branch holding the production-ready code and a develop branch which holds the currently in-development code, the feature branches are merged into this development branch instead of the main branch. This layer of abstraction aids in protecting the main branch as well as in aiding the DevOps philosophy of continuous integration and deployment.

Gitflow takes it further though, branches will also exist for hotfixes and releases for example. These branches act as a sort of staging area for you to perform testing and other validation techniques before committing to merging the changes into the main branch. Figure 17 visualises how this might look; the lines represent the branches and the dots on the lines represent the commits.

main

release

feature

develop

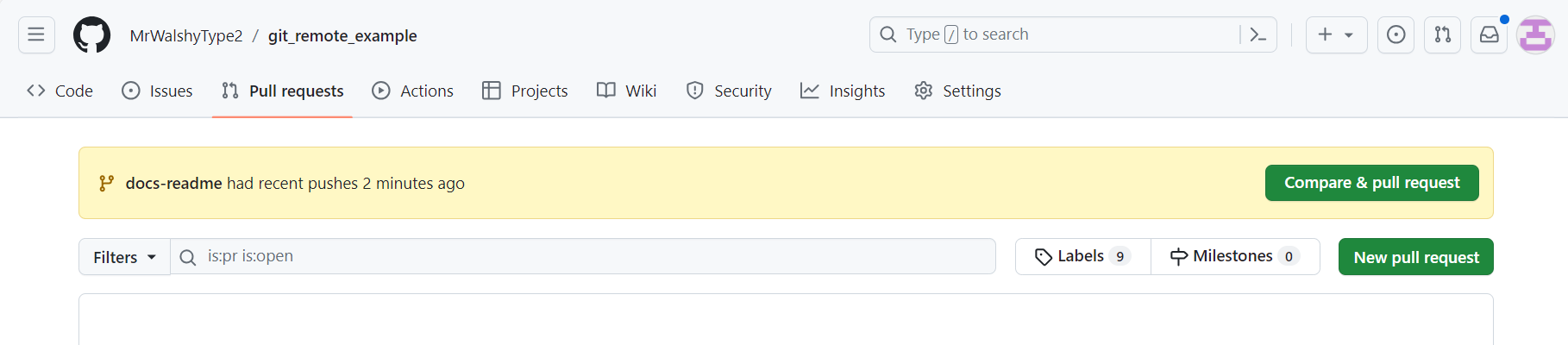
Figure 17 - The Gitflow Workflow visualised

## What is a pull request?

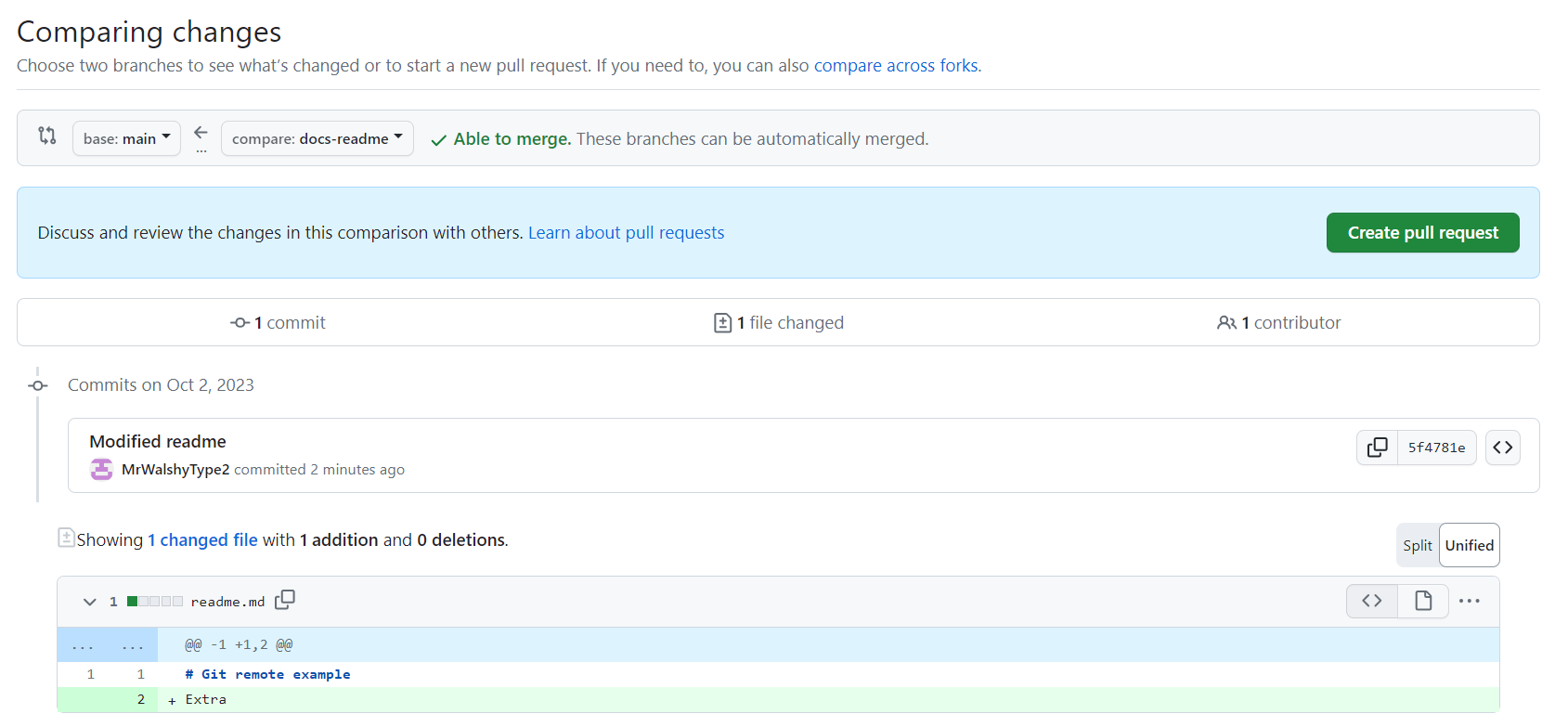
A pull request is a GitHub specific concept related to the merging of two branches, it allows for the history of two branches to be combined into one and thus bringing the two branches’ trees in line. A benefit of using pull requests is that GitHub offers a way to protect your main branches from unintended changes, when a pull request is created the developers can discuss and review the pull request. Until the pull request (PR) is merged into the desired branch, further commits can also be made on the branch being merged from.

### Using pull requests in GitHub to merge changes between branches

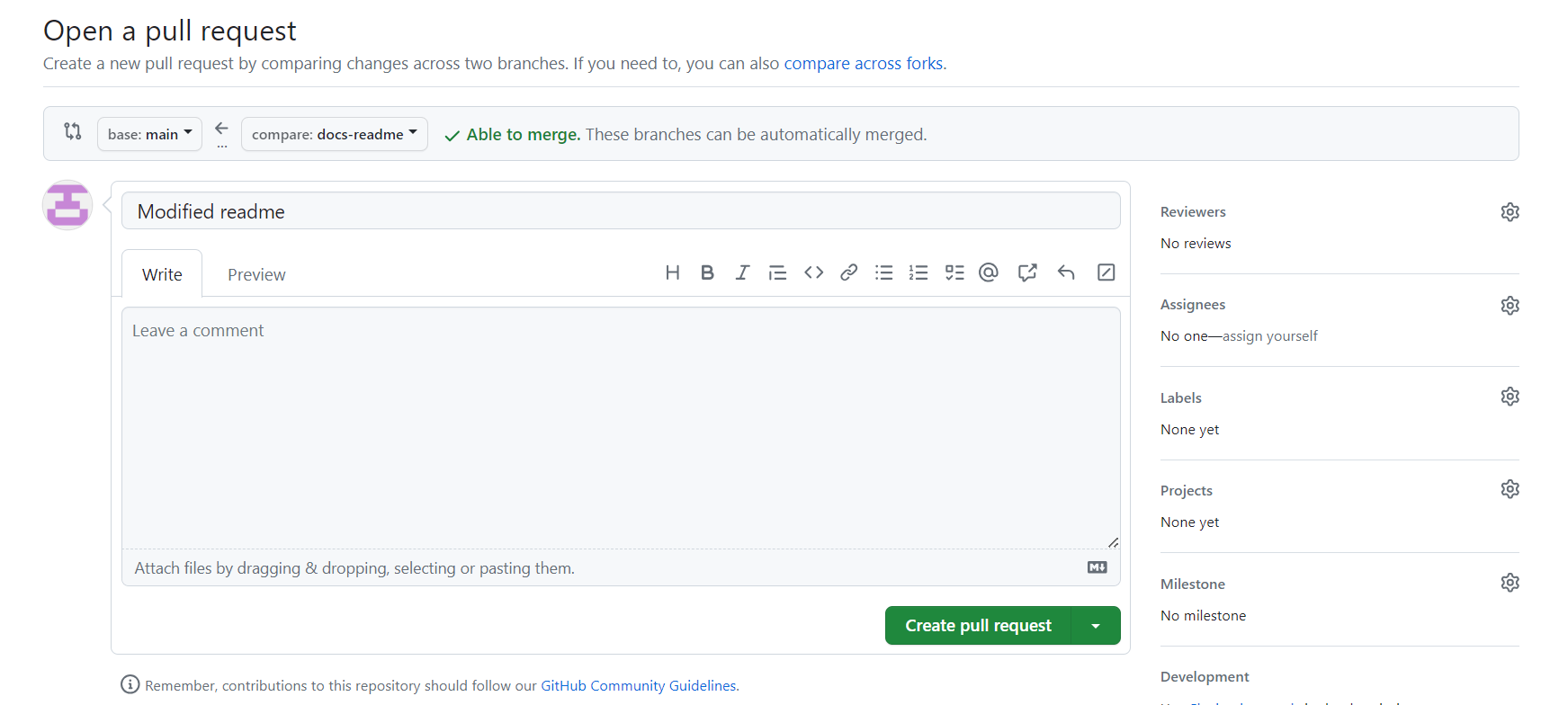
To create a pull request, your remote repository will require at least two branches; the target branch to merge into (the base branch) and the branch to merge from (the compare branch). You can create a new pull request by selecting the New pull request button on the Pull requests section of a repository:



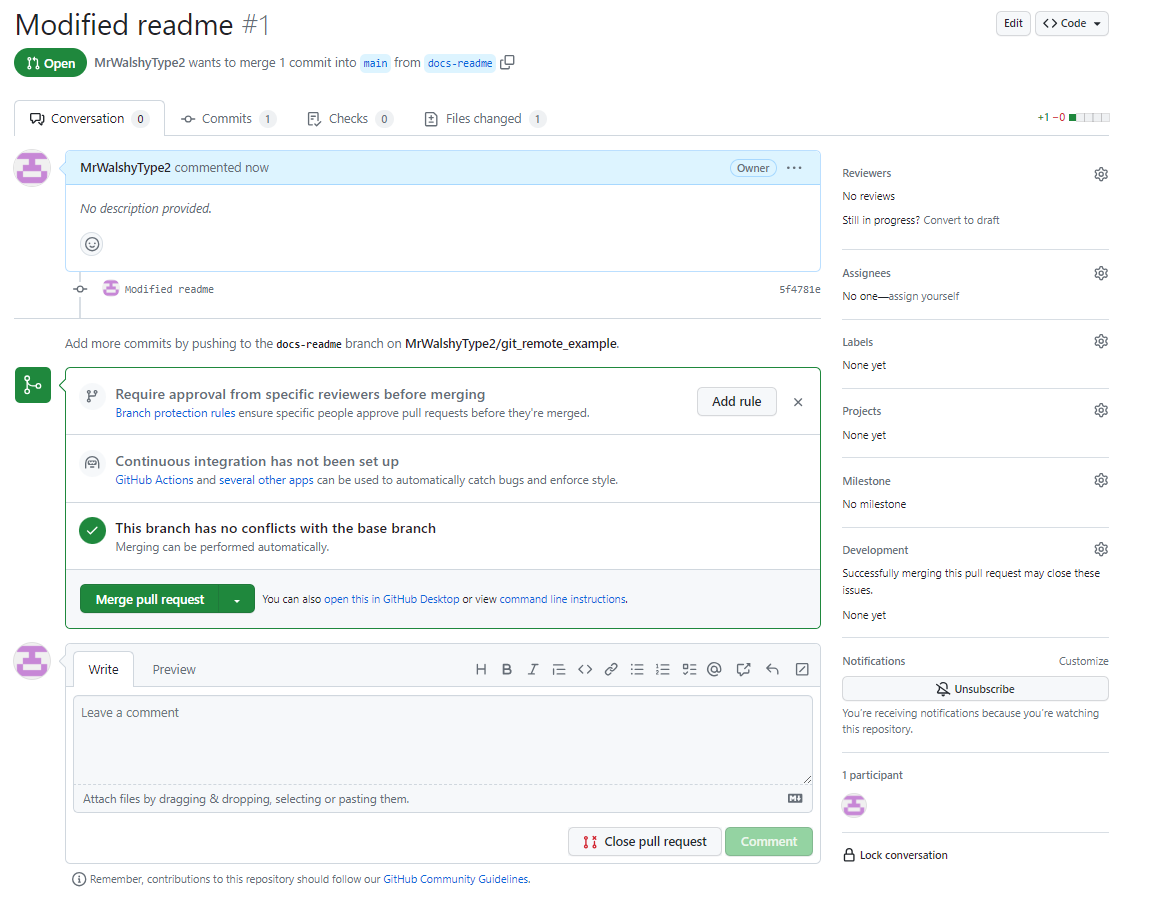
On the screen presented next, you will need to set the base branch as the one to merge into and the compare branch as the one to merge the changes from:



After setting the base and compare branches, select the Create pull request button to proceed to the next part of the form. In this section, you can add comments to the pull request, set reviewers, add labels, and more… Take your time in exploring this section, and then select the Create pull request button to actually create the request:



On the next screen, if you are happy to proceed and merge the changes, select the Merge pull request button to merge the compare branch into the main branch and then select Confirm merge when it appears:

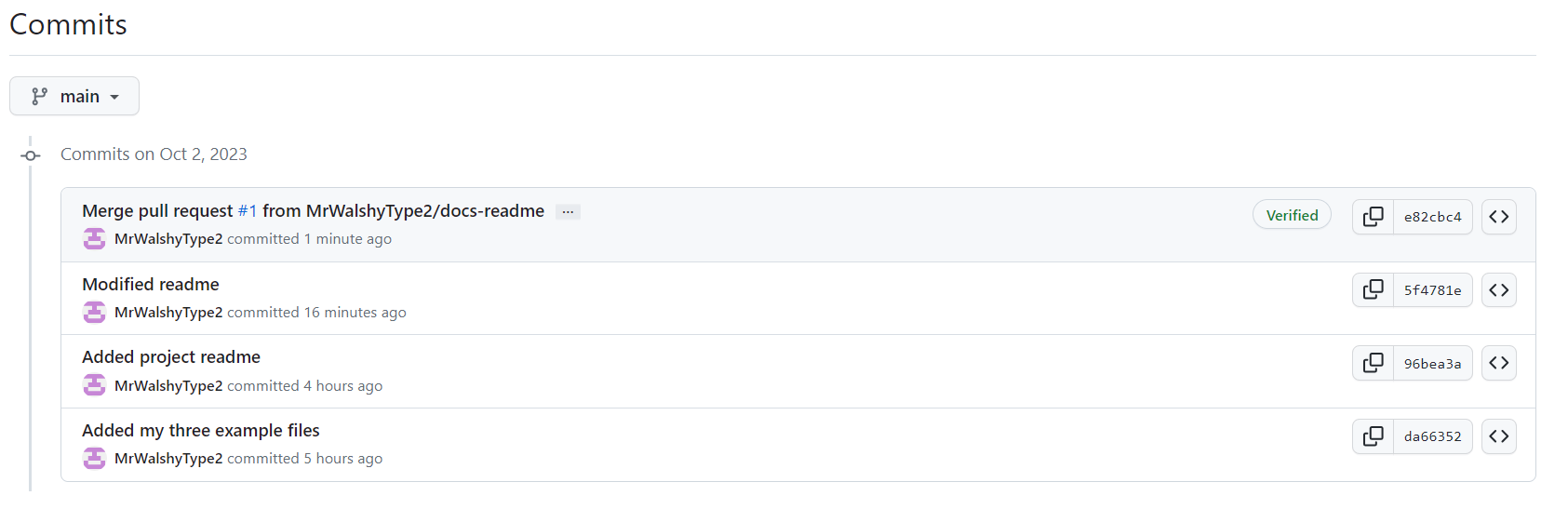


After completing the pull request, you will be prompted to delete the compare branch (optional):

A screenshot of a chat

Description automatically generated

If you look at the commit history of a repository after merging a pull request, you will see a merge commit indicating that the compare branch was merged into the base branch:



## Handling merge conflicts

A merge conflict occurs when two branches are being merged but both have modified the same file, this halts the merge process. Merge conflicts can sometimes be fixed in GitHub Desktop but normally require intervention using CLI to revert and resolve. Talk to your instructor if you run into any merge conflicts.