GitHub Tutorial

James McCarter

February 13, 2020

Contents

Raynier GitHub Organization **2**
0.1 Justification for Version Control . . . 2

Getting Connected to GitHub **3**
1.2 Create GitHub Account . . . 3
1.3 Invitation to Organization . . . 4
1.4 GitHub Overview . . . 5

Git Tutorial **11**
2.1 Git Quick Start . . . 11
2.1.1 Using Git GUI . . . 13
2.1.2 Using Git Bash . . . 16
2.1.3 Using TortoiseGit . . . 19
2.2 Intermediate Git . . . 25
2.2.1 Typical Daily Workflow . . . 25
2.2.2 File Differences . . . 29
2.2.3 Show Log . . . 32
2.2.4 Branching and Merging . . . 34
2.2.5 Tags . . . 41
2.2.6 Stashing . . . 43
2.2.7 Forking versus Branching . . . 43
2.2.8 Integrated Development Environment (IDE) Integration with Git . . . 43
2.2.9 General Guidance on Version Control . . . 50
2.3 Advanced Git . . . 51
2.3.1 Squash/Rebase Commits . . . 51
2.3.2 Merge vs Rebase . . . 51
2.4 Git Terminology . . . 52

GitHub Tutorial **54**
3.1 GitHub Termonology . . . 54
3.2 GitHub Desktop . . . 55
3.3 GitHub/Local Repository Synchronization . . . 56
3.3.1 Clone Repository from GitHub . . . 56
3.3.2 Publish Local Repository to GitHub . . . 57
3.3.3 Pull Repository from GitHub . . . 58
3.3.4 Push Repository to GitHub . . . 59
3.3.5 Staying in Sync Between GitHub and Local Repositories . . . 60
3.3.6 Moving Local Repositories . . . 60
3.3.7 Using GitHub R package repositories . . . 61
3.4 Migrate Repository from AWS CodeCommit to GitHub . . . 62
3.4.1 Reset Local Repository to GitHub from AWS CodeCommit . . . 66
3.5 Additional GitHub Concepts . . . 67
3.5.1 Please verify your device . . . 67

3.5.2 Setting up Repository for Access . . . 67
3.5.3 Teams, Members, and Collaborators . . . 68
3.5.4 Watch, Star, Fork . . . 68
3.5.5 GitHub Wiki . . . 69
3.5.6 Cloning Wiki . . . 69
3.5.7 GitHub Release . . . 69
3.5.8 Wiki Versus GitHub Pages . . . 69
3.6 GitHub Project Management . . . 70
3.6.1 Issues and Pull Requests . . . 70
3.6.2 Managing Issues . . . 70
3.6.3 Milestones . . . 76
3.6.4 Projects . . . 77

Rayonier GitHub Enterprise

Rayonier IT has created an Enterprise GitHub instance for the Biometrics Team and cooperators.

Git is a distributed version-control system for tracking source code changes during software development. It can successfully be used for traditional software development, web page development, package or module development (e.g R, Python) any scripting environment (e.g. automation of ArcMap, SQL database management, etc.), and any analysis process that is automated (e.g. R, SAS, etc.). Unlike some version control systems Git can also handle binary files (e.g. .xlsx, .docx, .mdb, .accdb, image files, etc.).

With Git every local working directory is a full-fledged repository with complete history and full version tracking abilities. As with most version control/tracking systems Git uses the concepts of *working copy* and *repository* for all files under version/revision control. The *local copy* (sometime called a *checkout*) is your personal copy of files in the project. When your edits are ready to save or share you *commit* your changes to the repository.

GitHub is a company that provides hosting for software development version control using Git. It offers all of the capabilities of Git but also adds features including: access control, bug tracking, feature requests, task management, and wikis for documentation of projects.

0.1 Justification for Version Control

Many business critical systems used by Rayonier have been developed or customized internally. Version control systems allow for the management of the development process. Source code file version control allows for the capture of the current working state and retrieval of previous states if recent edits cause unintended consequences in the system. But the real power of version control relies in the support for parallel and collaborative development.

In addition, version control software integrated with issue tracking and project management features allows organizations to capture issues and generate statistics about the development process. Combined with project/task management GitHub provides tools to measure progress in the development process. Questions such as "Are requirements met", "Are there fewer bugs today than yesterday", "Are we making progress" can now be answered by tracking and generating statistics on the development and issue fixing processes.

Git (version control software) combined with GitHub (web based hosting, issue tracking, and project management) are now being used at Rayonier to provide some assistance in our internal software development processes. Other source code tools (e.g. Team Foundation Server) are still being used, but the Biometrics team is migrating to GitHub (and logically Git) for the issue tracking and management features.

Getting Connected to GitHub

To use GitHub several steps are required. If you are using Git for local version control functions, several software packages (required and optional) need to be installed on your local computer.

Note: You can use GitHub for issue, task, and project management even if you are not using Git for local version control. In this case, no software is required, you just login to GitHub via a web browser.

1.2 Create GitHub Account

To get started using GitHub with Rayonier you need to first create a GitHub account by going to https://github.com and click the SIGNUP button.

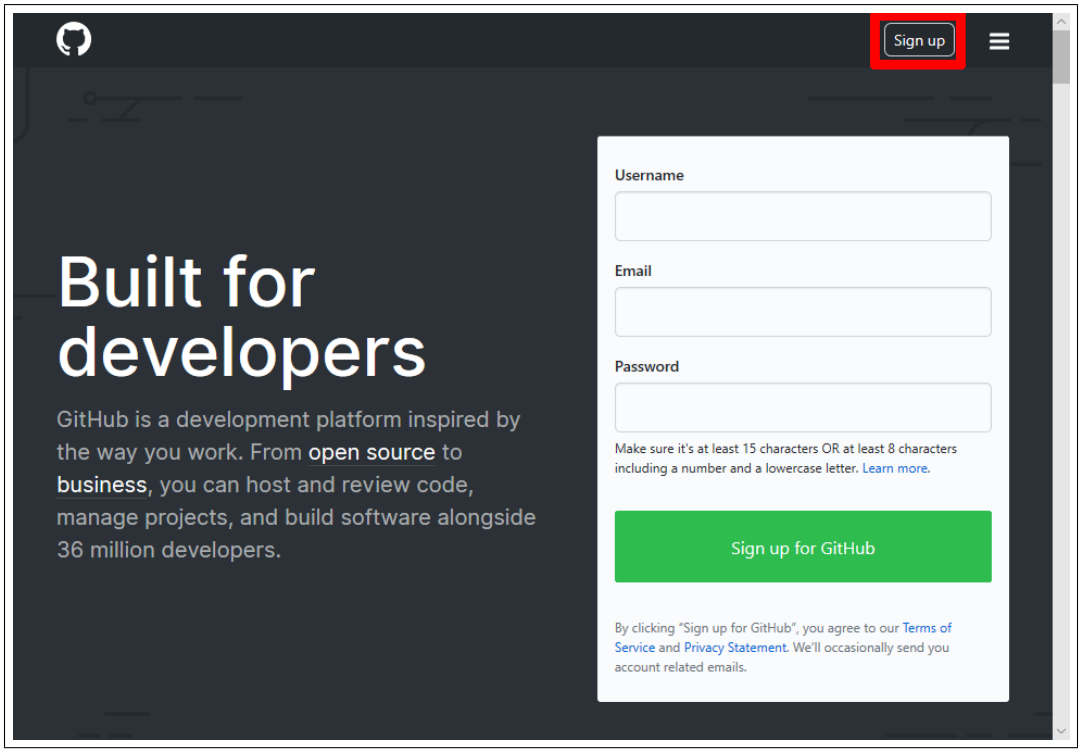


Figure 1.1: GitHub SIGNUP Screen

A GitHub username can contain alphanumeric characters, no spaces, and dash. I used JimMcCarter-Rayonier to distinguished my account associate with Rayonier from others. Passwords should be at least 15 characters long OR at least 8 characters including a number and lower case letter.

After creating your User account you can login using the Sign in to GitHub page.

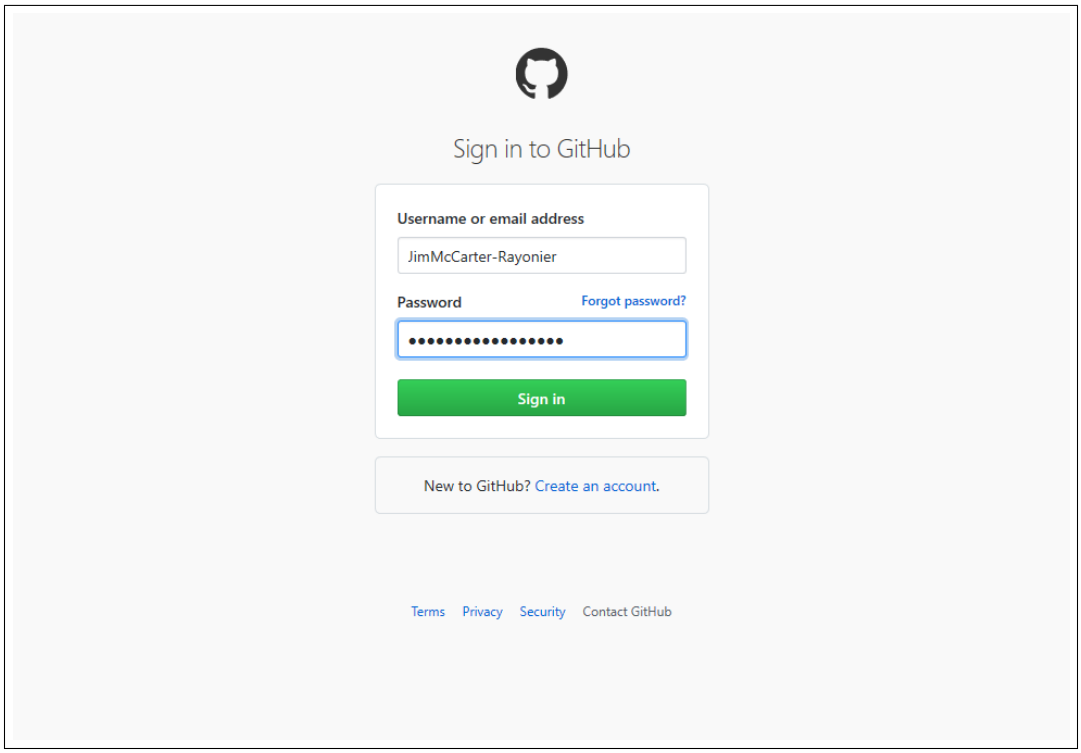


Figure 1.2: GitHub Login Screen

1.3 Invitation to Organization

An invitation to the Rayonier GitHub organization will be initiated by Rayonier IT (contact Jim McCarter or Lane Meadows) and sent to your Rayonier email address. Click on the link in the invitation email and login using your previously created credentials and you should see that you are associated with the Rayonier organization.

1.4 GitHub Overview

Once logged in you should be presented with a screen like the following showing your personal context, any Repositories you have and any Teams you are a member of. Your dashboard context can be changed by clicking on the down arrow and selecting from the options listed.

Note: the Rayonier Organization no longer looks identical to these screen captures because we are actively using the site and adding content.

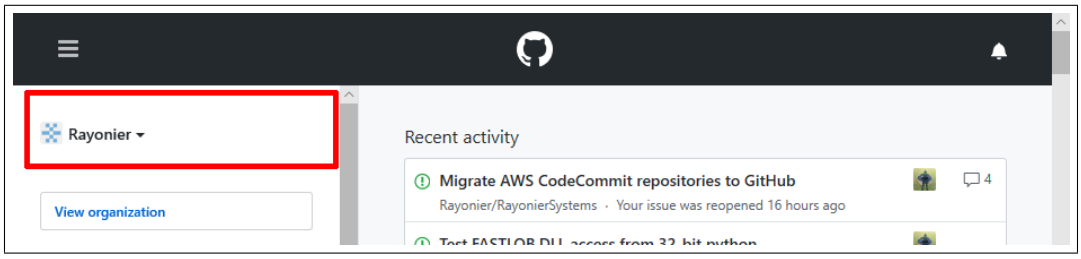


Figure 1.3: GitHub Screen after login

Depending on the width of your browser window you will see a different top section of the page. When narrow (Figure 1.4), additional functions are available by clicking the upper left button of the window (highlighted below).

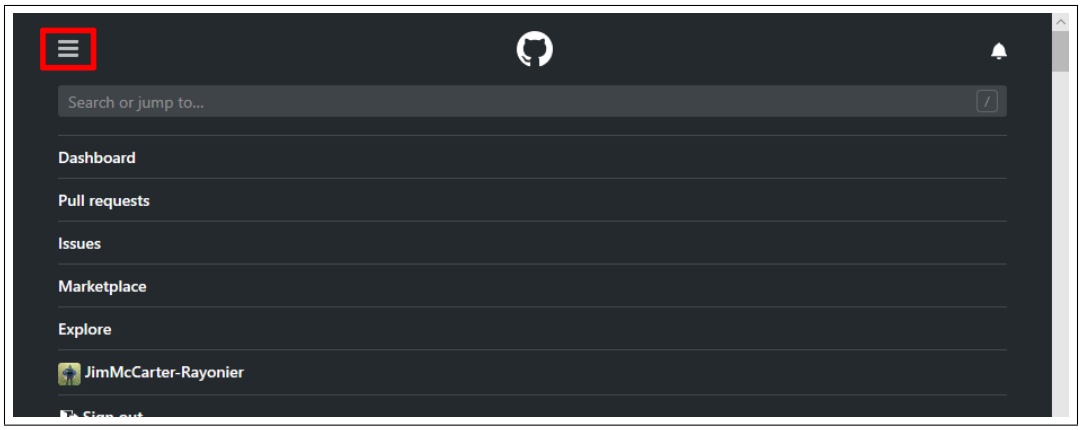


Figure 1.4: GitHub Dashboard With Narrow Browser Window

When the browser window is wider all the functionally is contained along the top of the windows (Figure 1.5).

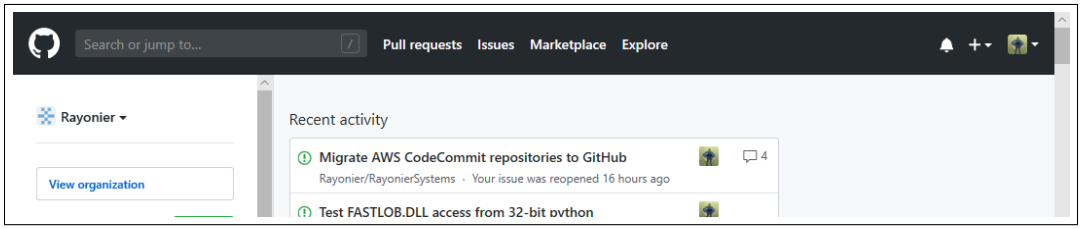


Figure 1.5: GitHub Screen after login

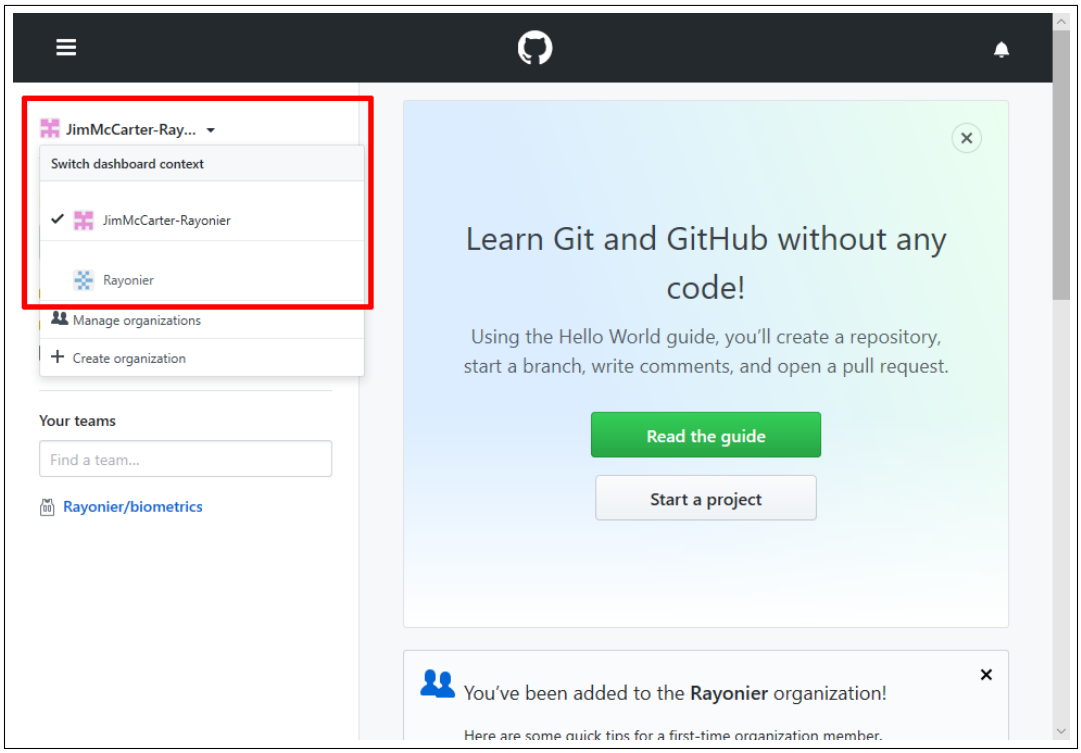


Figure 1.6: GitHub Switch Dashboard Screen

Using the down arrow and Switch dashboard context you can switch between the Rayonier organization, your personal context, and other organizations you are associated with. In this case select Rayonier. You will see recent activity associated with repositories and issues.

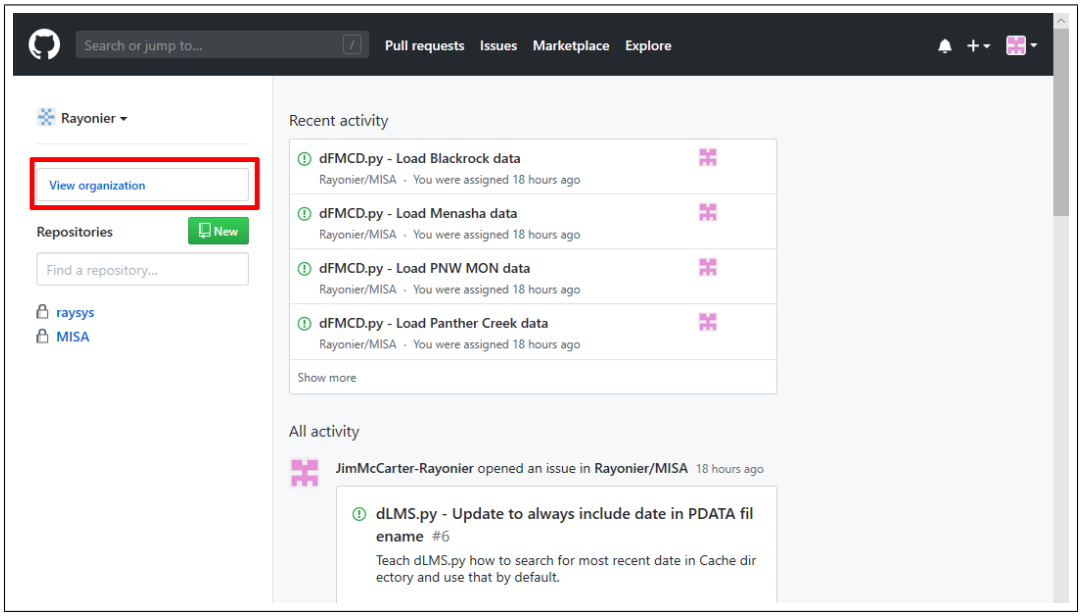


Figure 1.7: GitHub Rayonier Dashboard Repositories

Click on View organization to see a list of Repositories and tabs for Packages, People, Teams, and Projects. Repositories are listed most recently updated at the top.

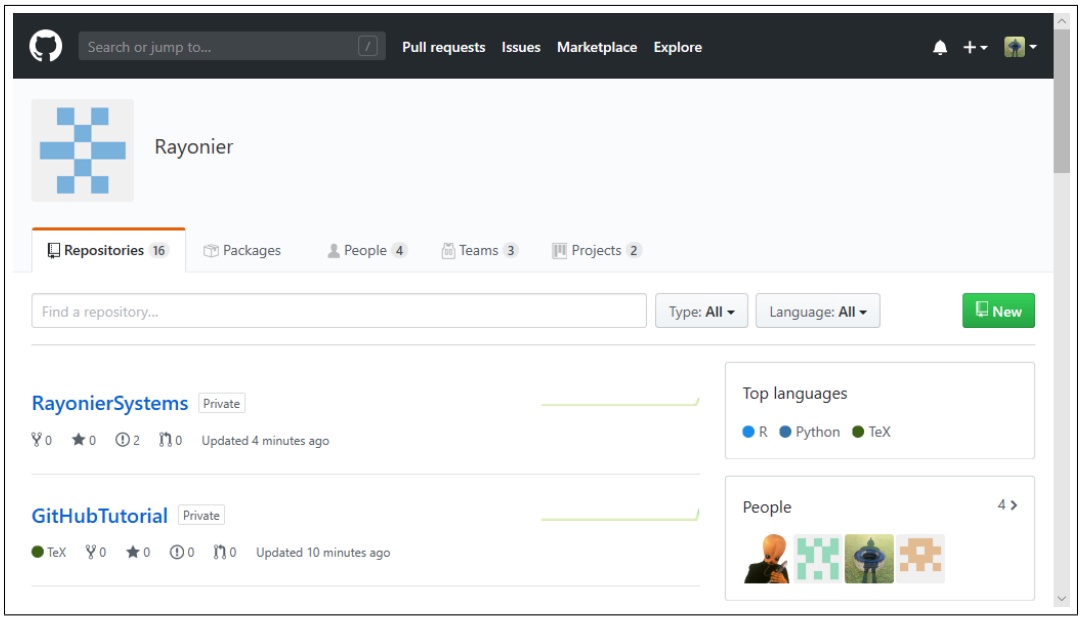


Figure 1.8: GitHub Rayonier Dashboard Repositories

You now see the Repositories that exist associated with the Rayonier organization.

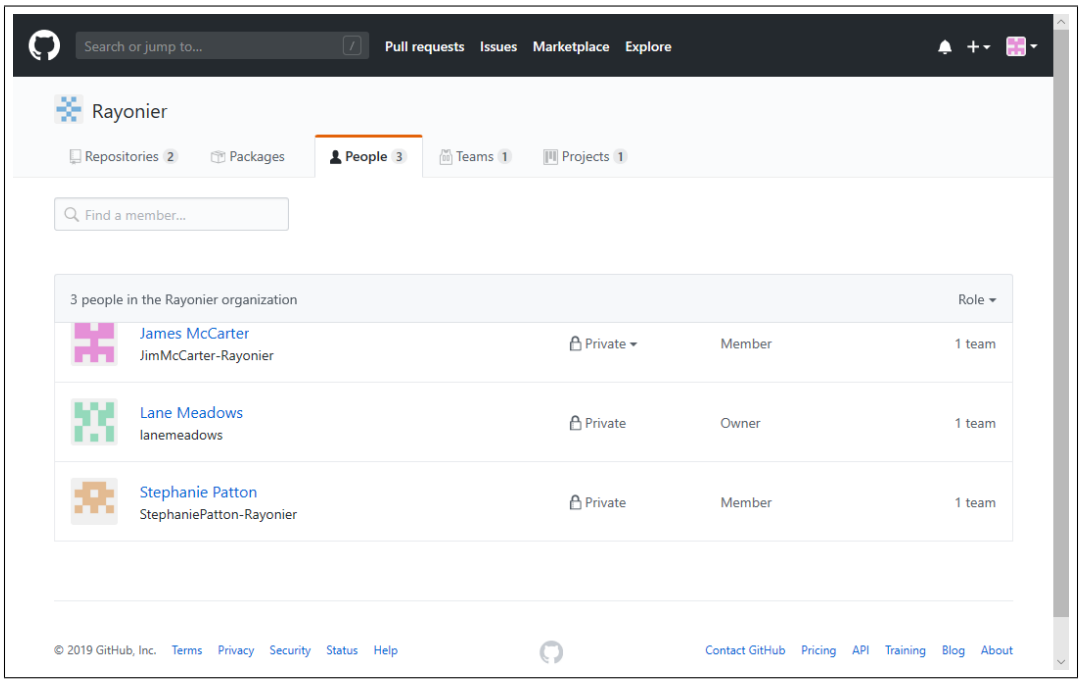


Figure 1.9: GitHub Rayonier Dashboard People

Click on People to see the list of folks that are currently associated with the organization.

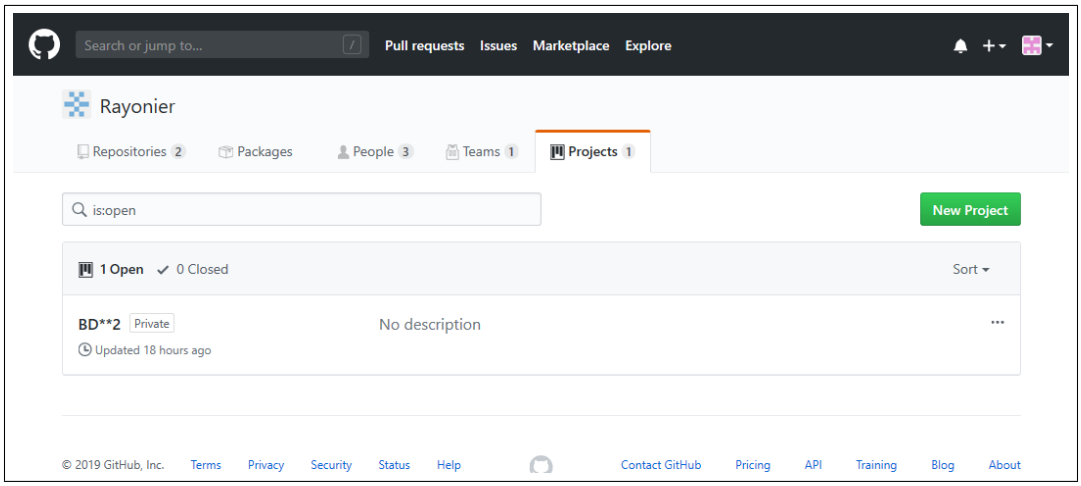
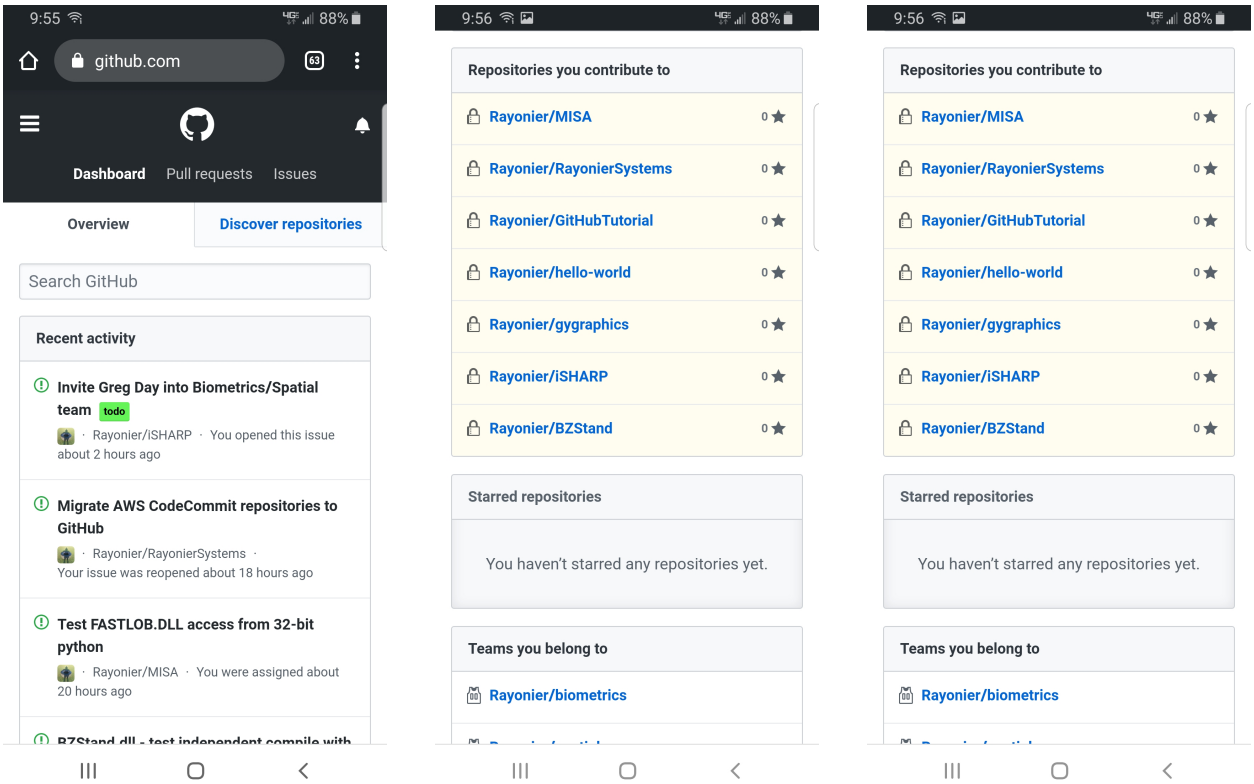


Figure 1.10: GitHub Rayonier Dashboard Projects

Click on Projects to see what projects are currently defined. Projects may provide enough support to replace Asana or other task/project management tools. See the Project Management section for more information.

Below shows what the interface to GitHub looks like from a mobile device, in this case using Chrome browser on an Android phone. Note: A new native app called Github for mobile is now in beta for iOS and is coming soon for Android devices.



(a) Page 1 (b) Page 2 (c) Page 3

Figure 1.11: GitHub interface using mobile browser.

Now we will show an existing repository that has been migrated to GitHub from CodeCommit (or local). This is my McCarter Inventory and Simulation Architecture (MISA) Python project. After clicking on the repository name, the first screen that appears is the Code tab. This will list the individual files contained in the repository and details about the file. Individual files can be viewed by clicking on the file name.

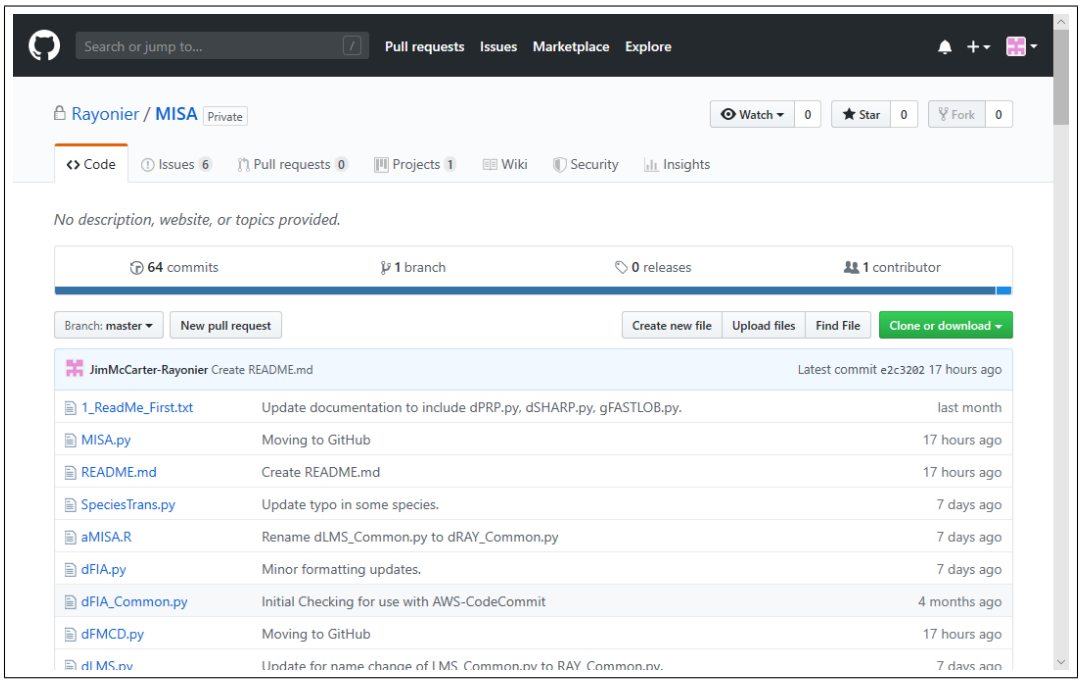


Figure 1.12: MISA repository - Code tab

Click the Issues tab to view current/open issues associated with the source code or project.

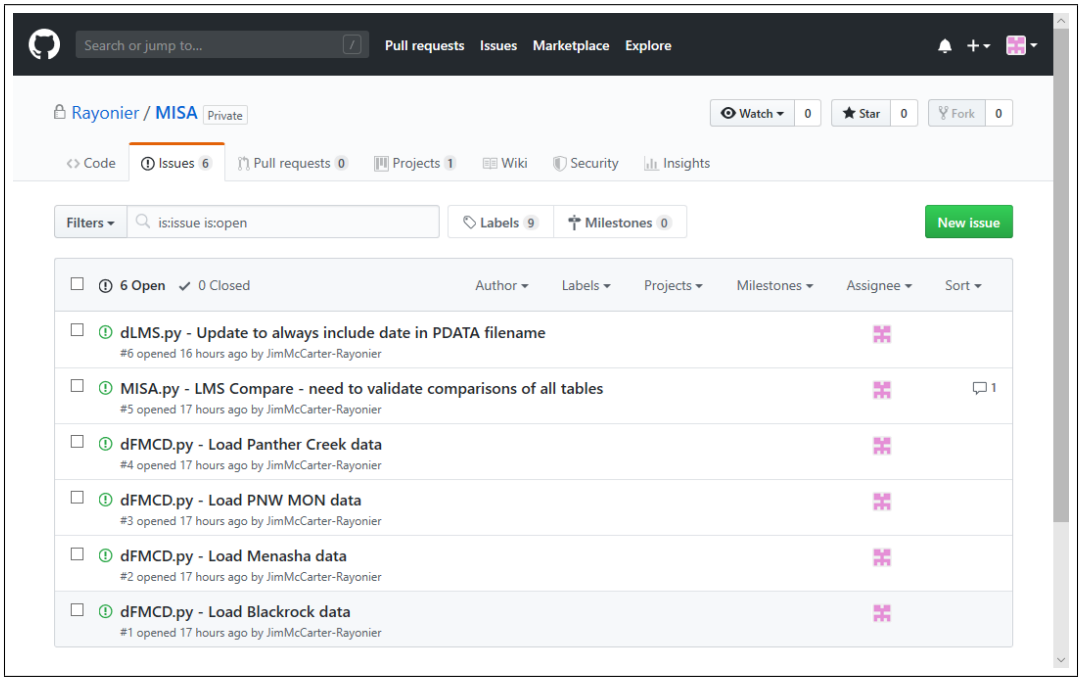


Figure 1.13: MISA repository - Issues tab

Click on the Projects tab to see what project are defined for the repository.

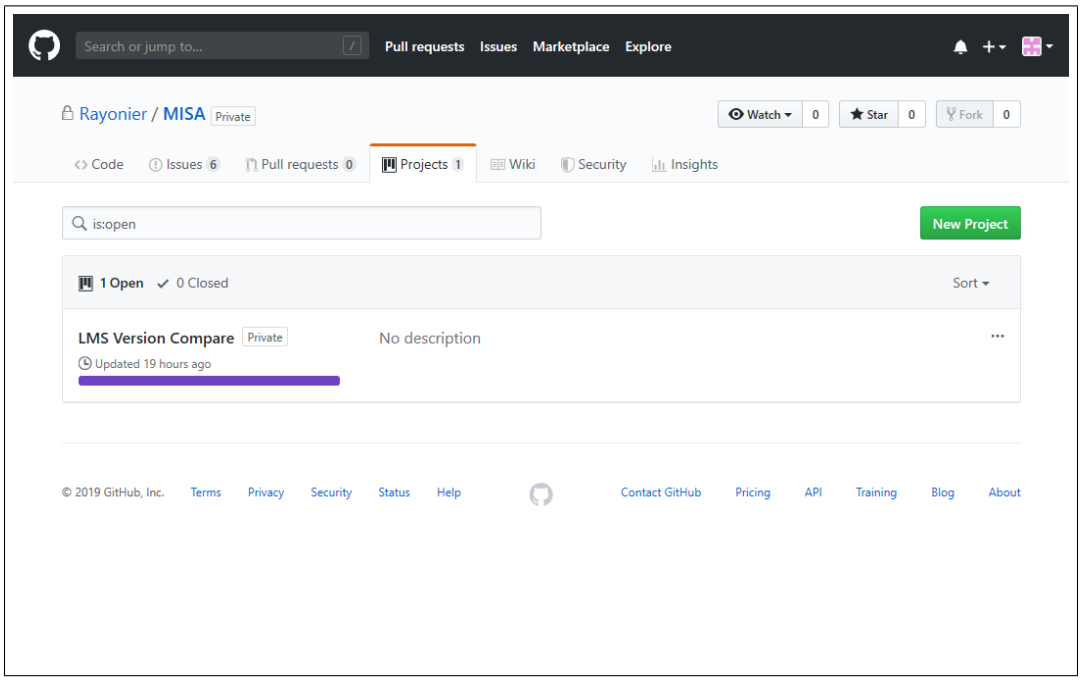


Figure 1.14: MISA repository - Projects tab

Pull Requests are use to propose collaboration on changes to a repository.

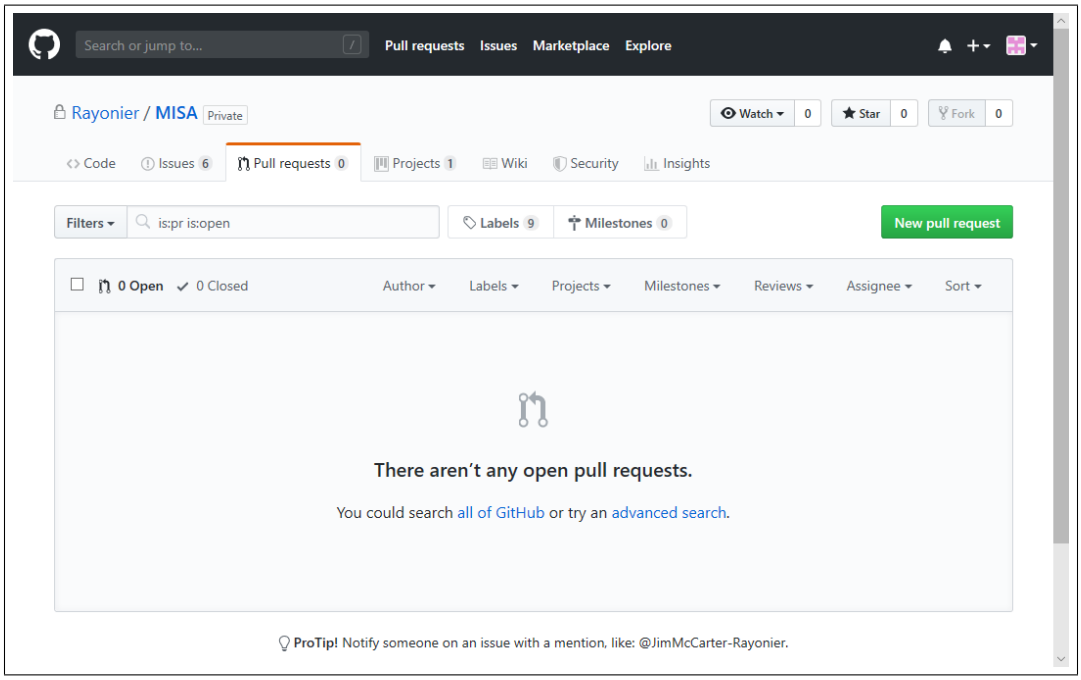


Figure 1.15: MISA repository - Pull Requests tab

Git Tutorial

Git is version control software that can be used locally for file versioning but also in collaboration with other developers through sharing of repositories across networks.

2.1 Git Quick Start

Getting Started with Git.

- Download and install Git software (Git-2.22.0-64-bit.exe when this tutorial was developed) to local computer from https://git-scm.com/. Install will all default options.
- Optionally download and install TortoiseGit (TotoiseGit-2.8.0.0-64bit.msi when this tutorial was developed) from https://tortoisegit.org/. Install with all default options after Git has been installed.

You can use Git GUI, Git Bash, TortoiseGit, or some combination as your interface to the Git system. If you are starting a new project, use Windows Explorer to create a new empty folder, then navigate into that empty folder. If you have an existing project and want to create a repository, navigate to the folder with Windows Explorer. In the file pane area, but not highlighting any specific file, right click to bring up the Windows Context Menu (Figure 2.2). Our example below is the folder for this GitHub Tutorial:

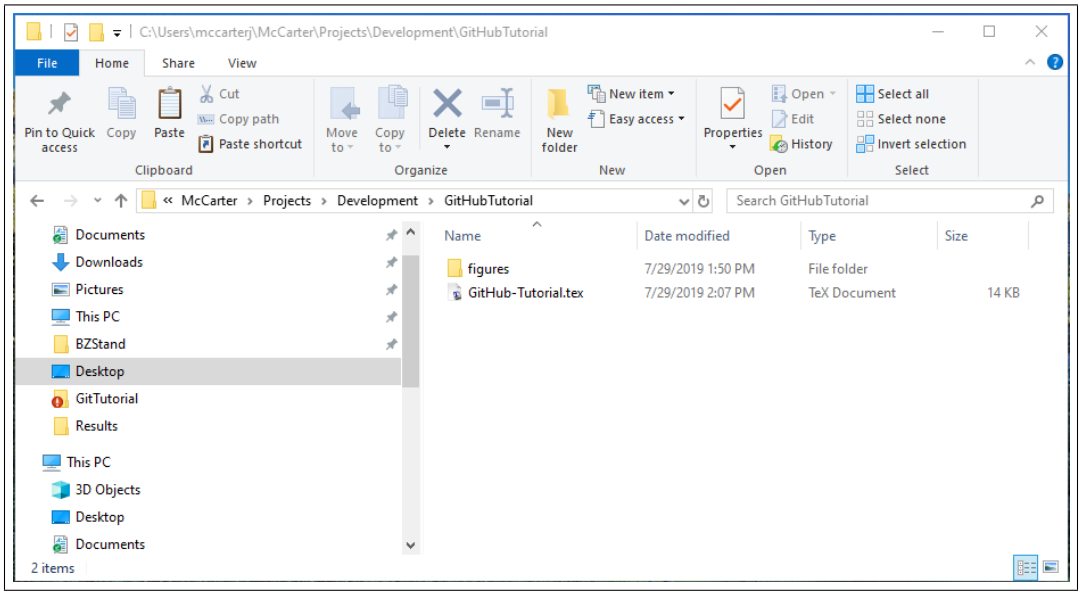


Figure 2.1: GitHub Tutorial Folder

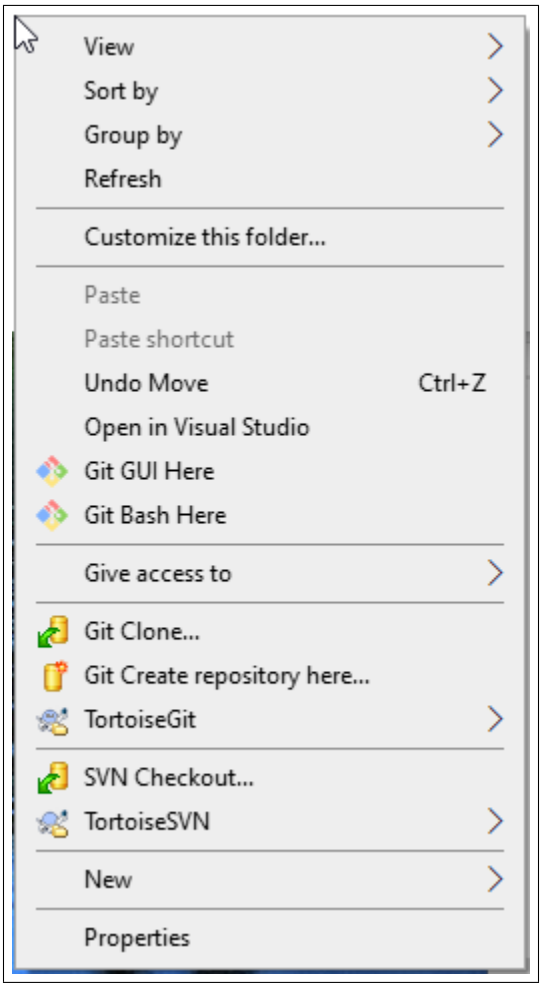


Figure 2.2: Windows Explorer Context Menu

Several options are available depending on what software you have installed (shown above). If you only have Git installed you will see the **Git GUI Here** and **Git Bash Here** menu commands. If you have also installed TortoiseGit you will also see the **Git Clone...**, **Git Create repository here...**, and **TortoiseGit** menu commands. My computer also has SVN (another version control system) installed so also includes the **SVN Checkout...** and **TortoiseSVN** menu commands.

2.1.1 Using Git GUI

Creating a local repository

After browsing to the desired directory with Windows Explorer, right click in the directory and select "Git GUI Here" from the Windows Explorer context menu. If there is no repository in the folder the Git GUI initial dialog will open. Select Create New Repository from the dialog (Figure 2.3), browse to/confirm the directory.

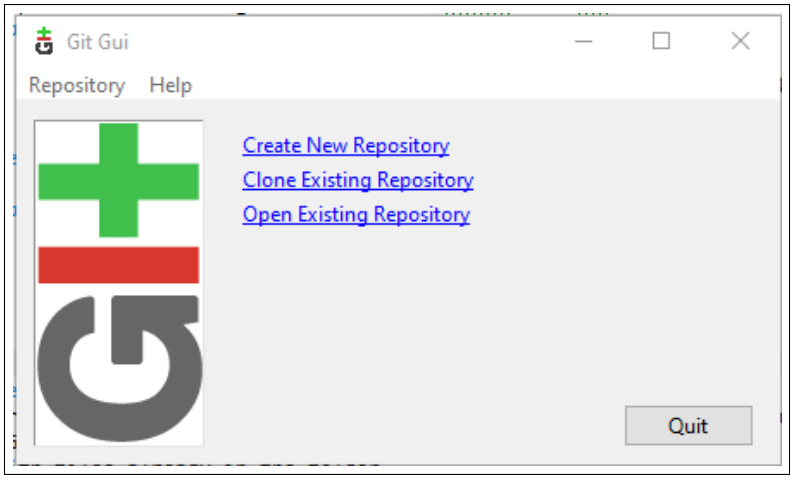


Figure 2.3: Git GUI Dialog

The Git GUI should now open on the folder (Figure 2.5).

You now should have a directory that is initialize for Git as indicated by the presence of the .git folder. This folder is a hidden system folder and may not show up depending on your Windows Explorer options. If you are not able to see the .git folder in Windows Explorer go the View menu, select Options, select the View tab, and make sure **Show hidden files, folders, and drives** is selected.

Adding files to repository

After we format this tutorial there are several more files in the folder. Some of these we want to added to the repository (.tex) and some we want to ignore (.aux, .log, .pdf, .gz, .toc).

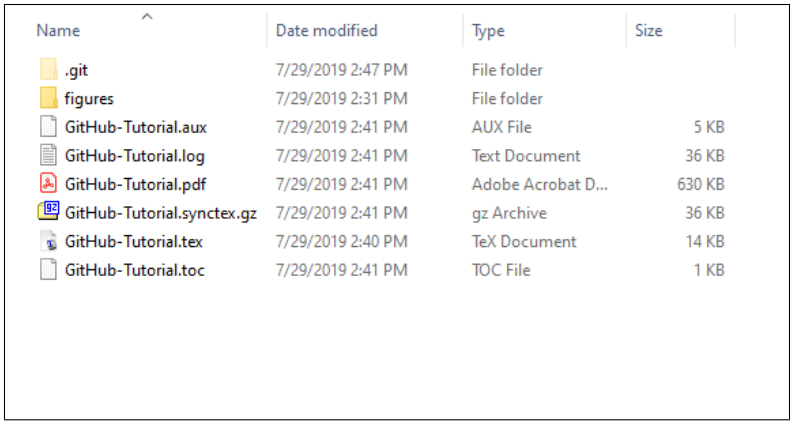


Figure 2.4: GitHub Tutorial Folder after repository created

The Git GUI dialog has Menus and 4 areas in the dialog: **Unstaged Changes**, **Staged Changes**, **File/Change Viewer**, and **Commit Message**. Files listed in Unstaged can be move to Staged by clicking the icon left of filename. This will add the file to the repository if is not already included and stage the file for committing changes to the repository.

To add a new file to a repository click the icon to the left of the file in the **Unstaged Changes** area, to "stage" it. The file will show up in the **Staged Changes** area. If you accidentally add a file, click the icon next to the file in the **Staged Changes** area to move it back to the **Unstaged Changes** area.

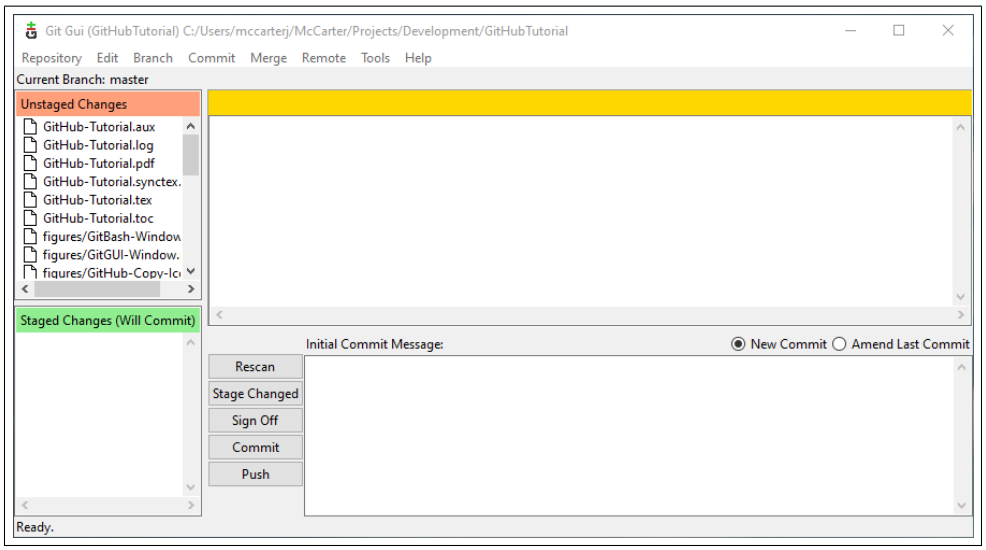


Figure 2.5: Git GUI Dialog

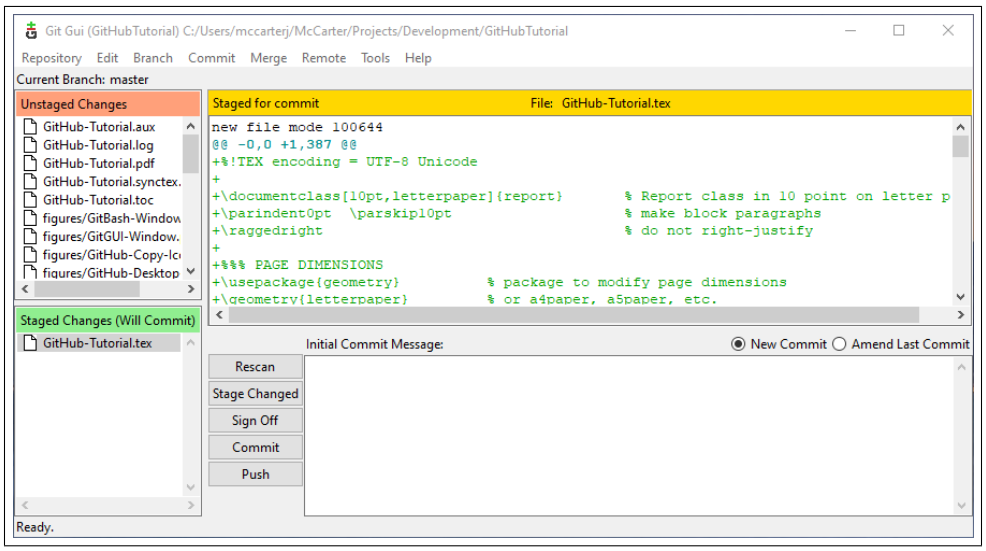


Figure 2.6: Git GUI Dialog File Staged

Ignoring files in working directory

Not possible in Git GUI without adding a tool to the interface. See How to Ignore files using Git GUI for the workflow of adding **Add to .gitignore** tool.

Committing files in repository

The next step in using Git is to commit your file changes to the repository. This basically stores a copy of the current state of the file(s) at the time of the commit. This allows you to make changes to files, add those changes with a future commit, or roll back the changes to a version from a previous commit.

When you have files listed in the **Staged Changes** area you can enter a Commit Message and click the Commit button.

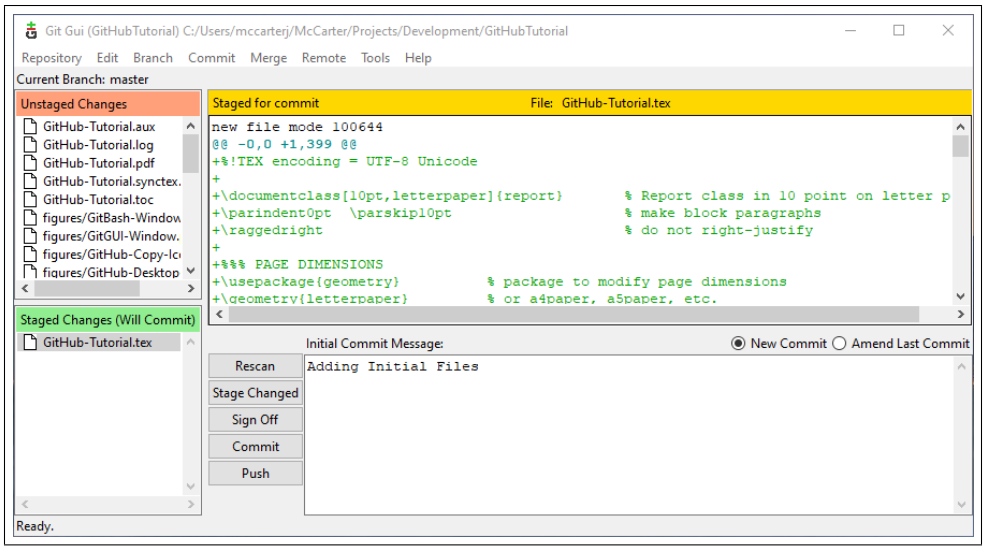


Figure 2.7: Git GUI Commit

The file will disappear from the dialog until there are changes to the file. Use Rescan after editing any files in the repository for them to show. If you want to stage the change, click the icon to the left of the filename to stage the file, add or edit the commit message, and select the commit button.

2.1.2 Using Git Bash

Creating a local repository

After browsing to the desired directory with Windows Explorer, right click in the directory and select "Git Bash Here" from the Windows Explorer context menu. This will open a *bash* command shell window where you can enter *git init* from the bash shell (Figure 2.8).

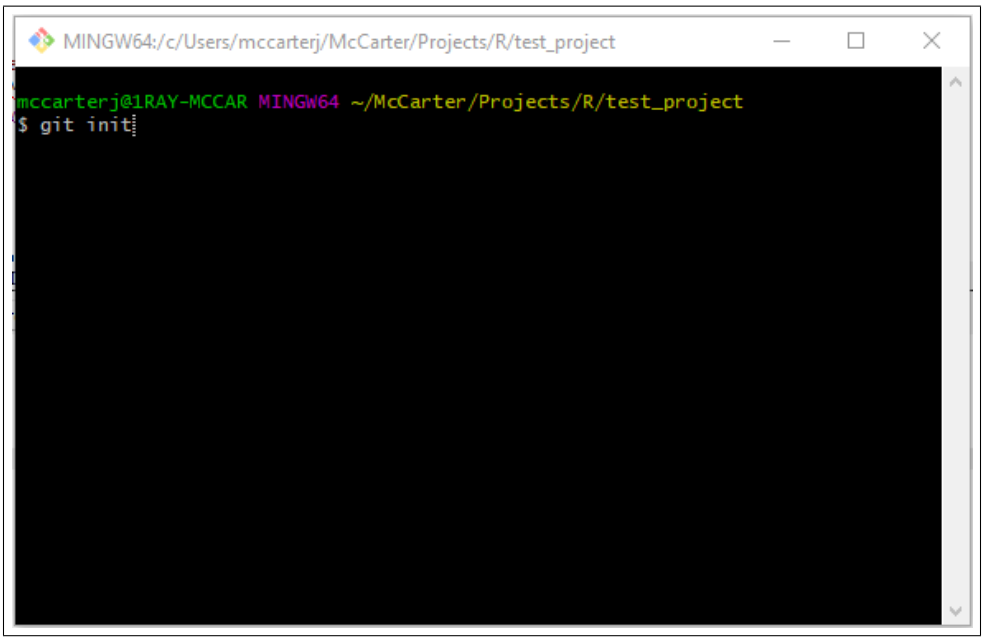


Figure 2.8: Git Bash Window

Adding files to repository

After we format this tutorial there are several more files in the folder. Some of these we want to add to the repository and some we want to ignore.

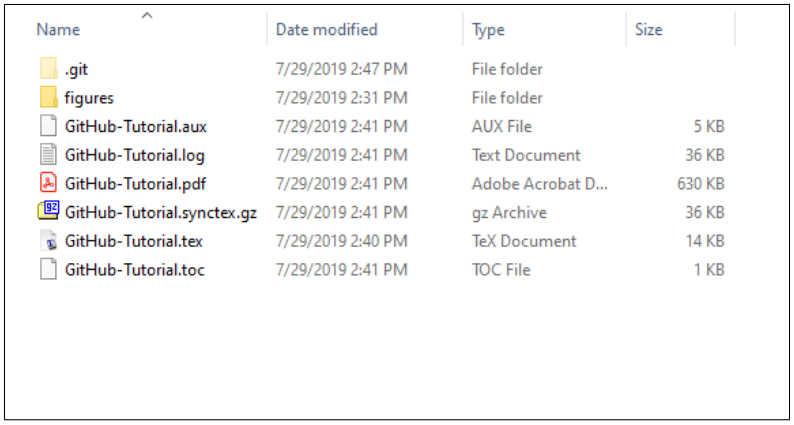


Figure 2.9: GitHub Tutorial Folder after repository created

Use the **git add FilenameToAdd** command to add desired files to the repository.

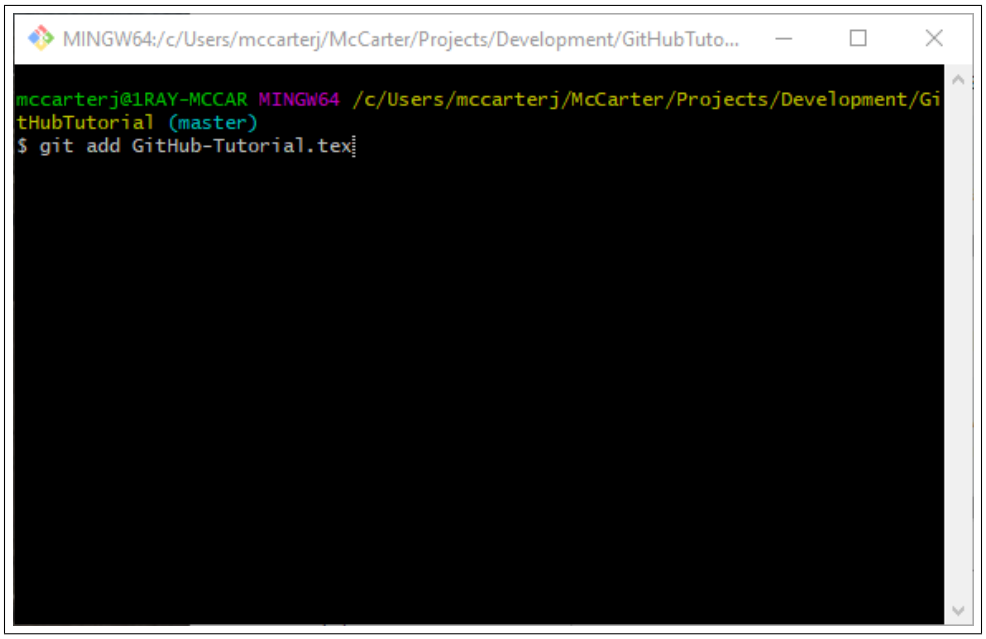


Figure 2.10: Git Bash Add File

Use **git status** at any time in the Git Bash shell to examine file status of the repository.

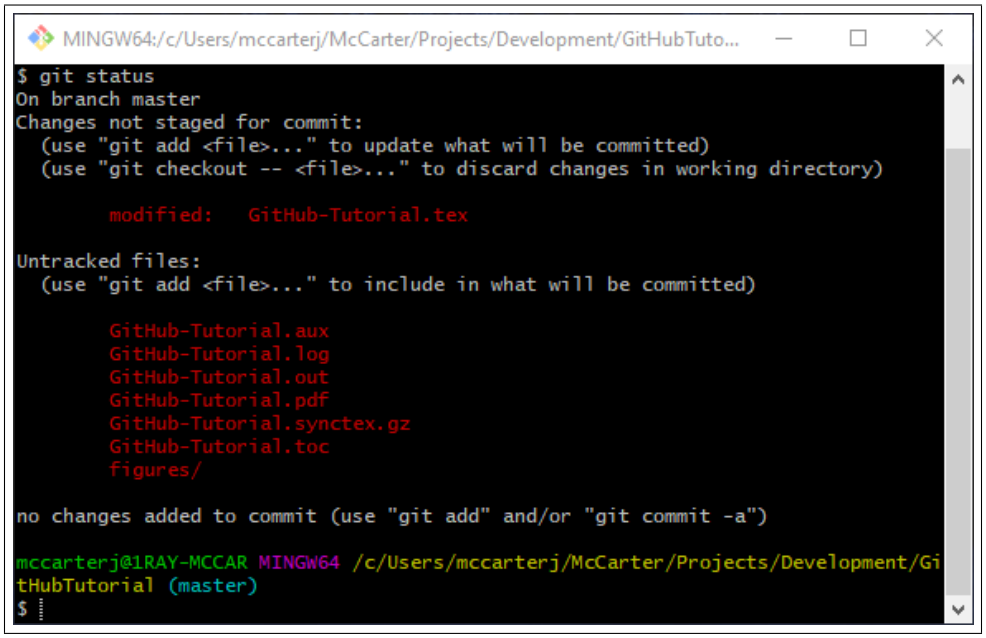


Figure 2.11: Git Bash Status

Ignoring files in working directory

If the file has not been added to repository then create or edit .gitignore file and add individual filenames or filenames with wildcards to this file.

If the file has already been added to the repository they you use **git reset NameOfFile** to unstage a file

and **git rm –cached NameOfFile** if the file has already been pushed.

You now should have a directory that is initialize for Git as indicated by the .git folder. This is a hidden system folder and may not show up depending on your Windows Explorer options. If you are not able to see the .git folder in Windows Explorer go the View menu, select Options, on the View tab, and make sure "Show hidden files, folders, and drives" is selected.

Committing files in repository

The next step in using Git is to commit your file changes to the repository. This basically stores a copy of the current state of the file(s) at the time of the commit. This allows you to make changes to files, add those changes with a future commit, or roll back the changes to a version from a previous commit.

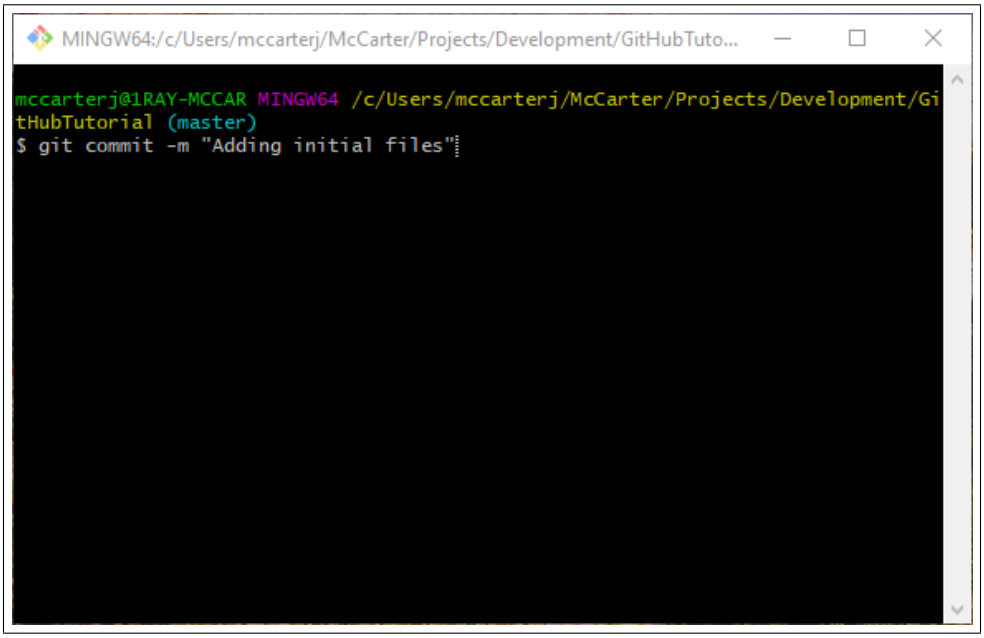


Figure 2.12: Git Bash Commit

2.1.3 Using TortoiseGit

On the surface TortoiseGit may appear to have a more complicated workflow. It is designed to expose many of the features of Git in Windows Explorer context sensitive menus instead of having to know all the command line options. Depending on the status of folder and/or file selected when you right click, the menu changes presenting the available options at that stage.

Creating a local repository

After browsing to the desired directory with Windows Explorer, right click in the directory and select "Git Create repository here..." from the context menu. This will open a conformation dialog (Figure 2.13). Leave "Make it Bare" unchecked, and click OK.

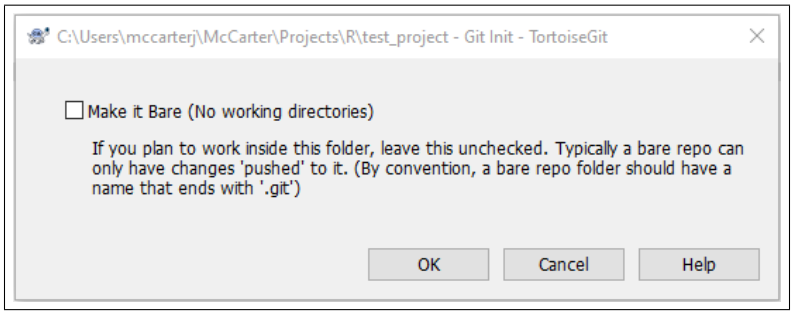


Figure 2.13: Tortious Git Create Repository

You now should have a directory that is initialize for Git as indicated by the .git folder. This folder is a hidden system folder and may not show up depending on Windows Explorer options. If you are not able to see the .git folder in Windows Explorer go the View menu, select Options, on the View tab, and make sure "Show hidden files, folders, and drives" is selected.

Adding files to repository

After we format this tutorial there are several more files in the folder. Some of these we want to add to the repository and some we want to ignore.

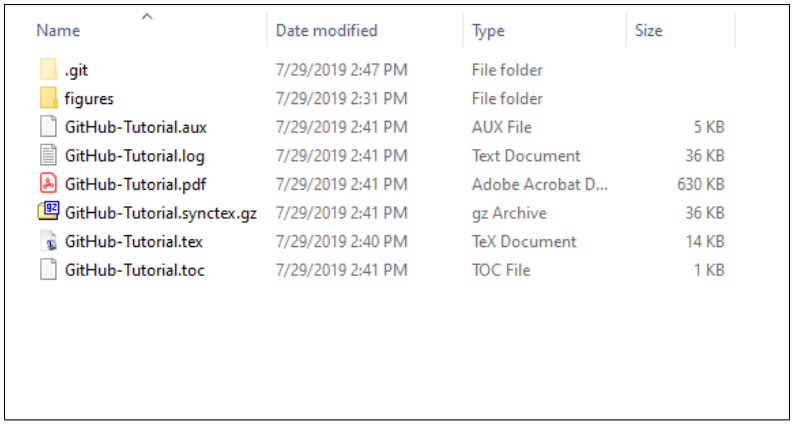


Figure 2.14: GitHub Tutorial Folder after repository created

Right click on the file you want to add and from the TortoiseGit menu select Add... (Figure 2.15)

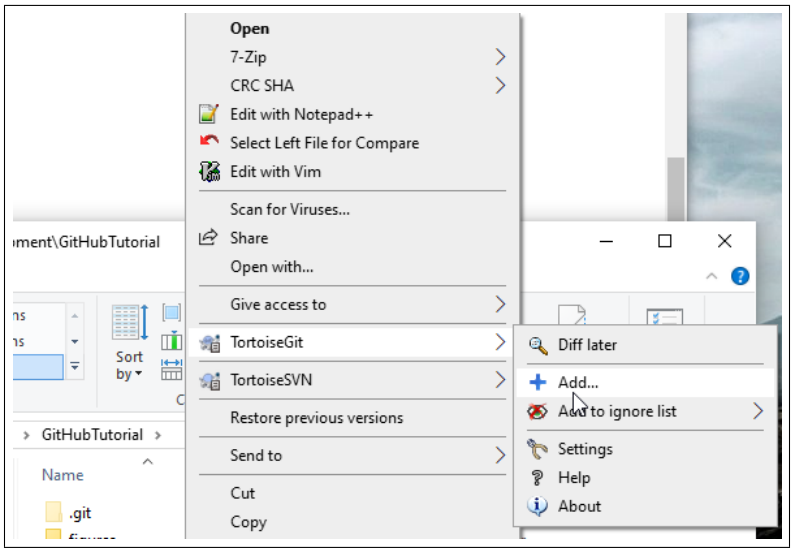


Figure 2.15: Tortious Git - Add File

Confirm that the file has been added to the repository (Figure 2.16).

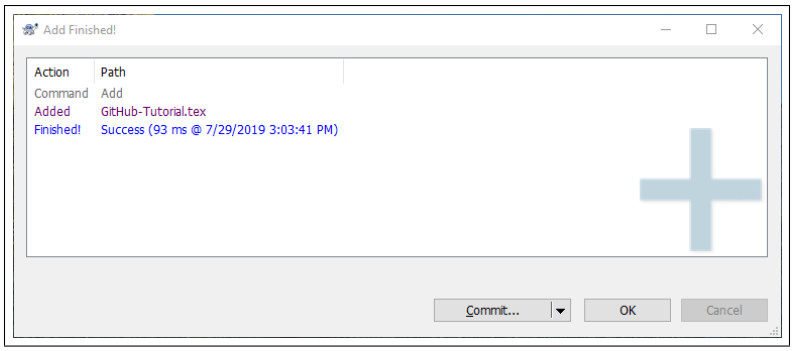


Figure 2.16: Tortious Git - Add File Dialog

Note that after the file is added the icon now has a + icon (Figure 2.17) next to the filename indicating the file has been added, but is not up to date in the repository.

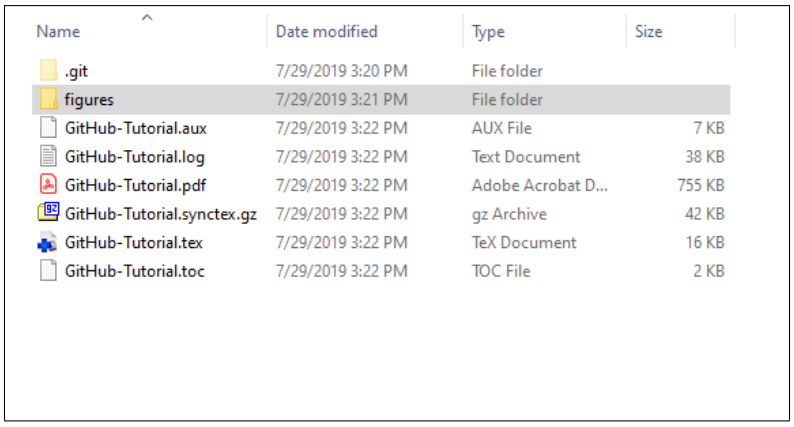


Figure 2.17: Tortious Git - File Added

Ignoring files in working directory

The ignore file interface is where TortoiseGit begins to show is utility. Right click on a specific file or in the directory and select the TortoiseGit menu (Figure 2.18), select Add to ignore list and then choose to ignore a specific file or all files of that type. This can also be done for directories (e.g. a temporary directory where your code works but of which the files should not be stored in a repository).

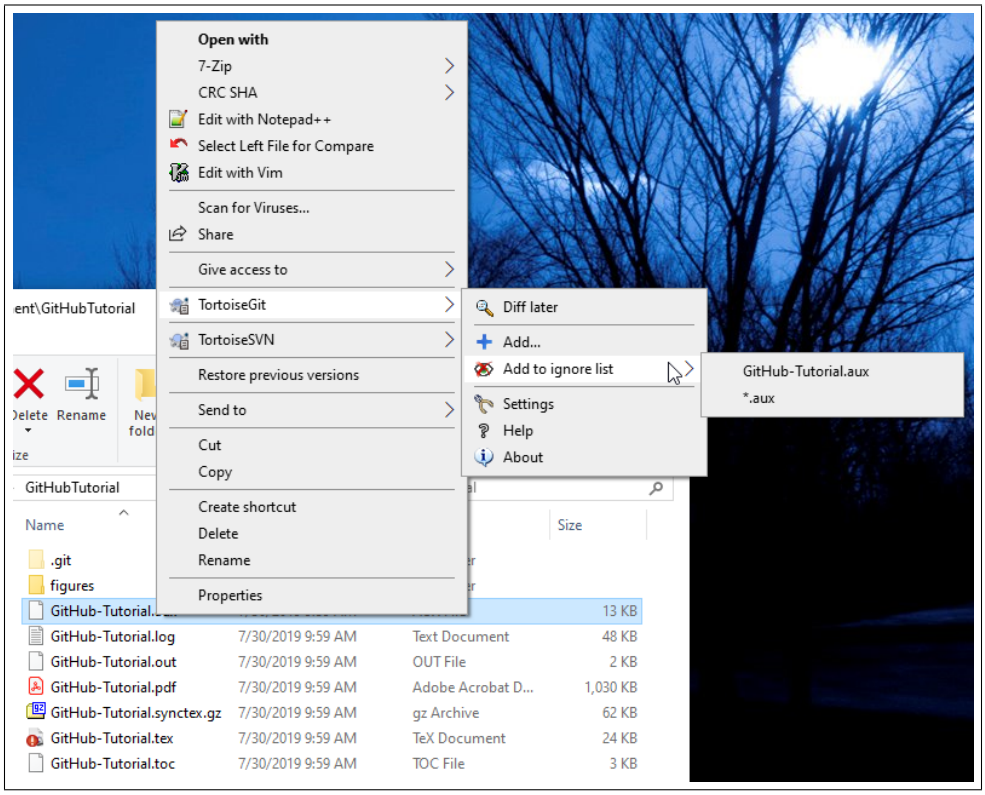


Figure 2.18: Tortious Git - Ignore Files

Committing files in repository

The next step in using Git is to commit your file changes to the repository. This basically stores a copy of the current state of the file(s) at the time of the commit. This allows you to make changes to files, add those changes with a future commit, or roll back the changes to a version from a previous commit.

I like using a combination of selecting individual files and groups of files for my repository commits depending on whether I want comments added to specific files or the same comment added to several files. If you want to add individual file comments on the commits, highlight the file with Windows Explorer and right click on the file to bring up the Windows Explorer context menu, select the TortoiseGit menu, and then select **Check for Modifications** (Figure 2.19)

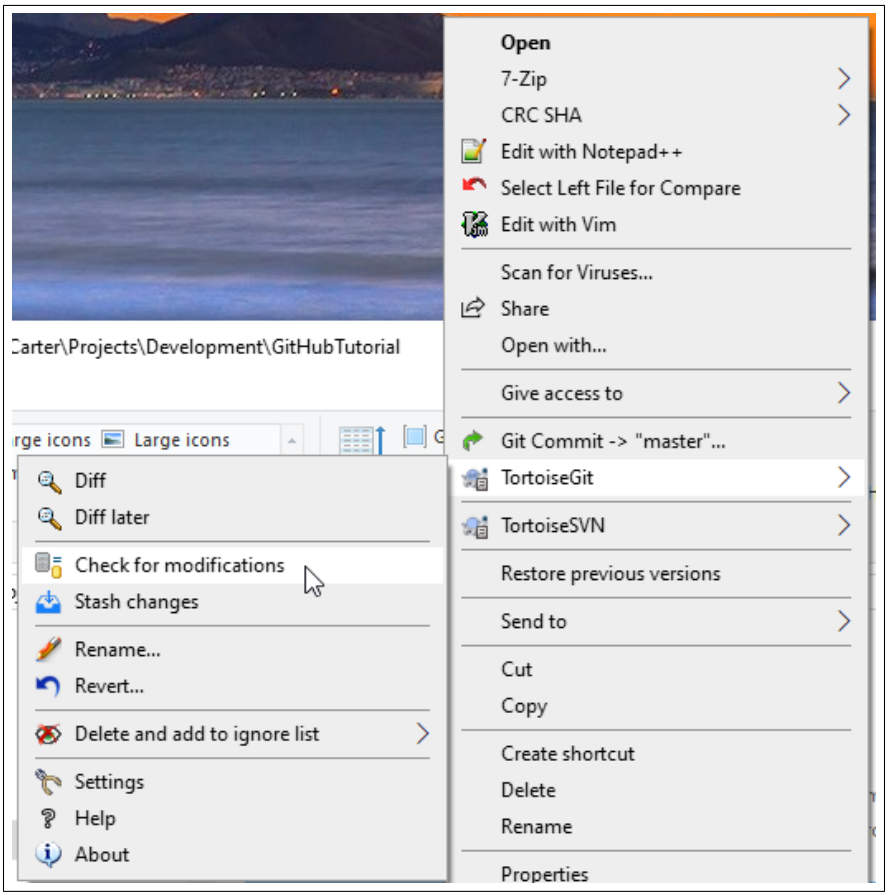


Figure 2.19: Tortious Git - Check For Modifications

This will bring up a Working Tree dialog showing Status of the selected file (Figure 2.20) or group of files in the folder (Figure 2.21). Note that the Working Tree dialog also allows you to perform other functions like adding files, checking for differences between the working copy and previously committed versions.

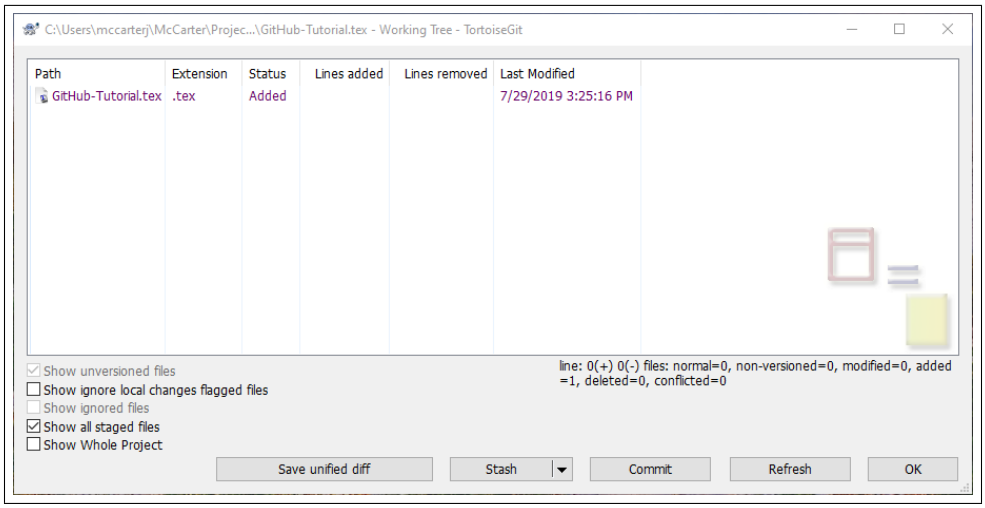


Figure 2.20: Tortious Git - Check For Modifications Dialog

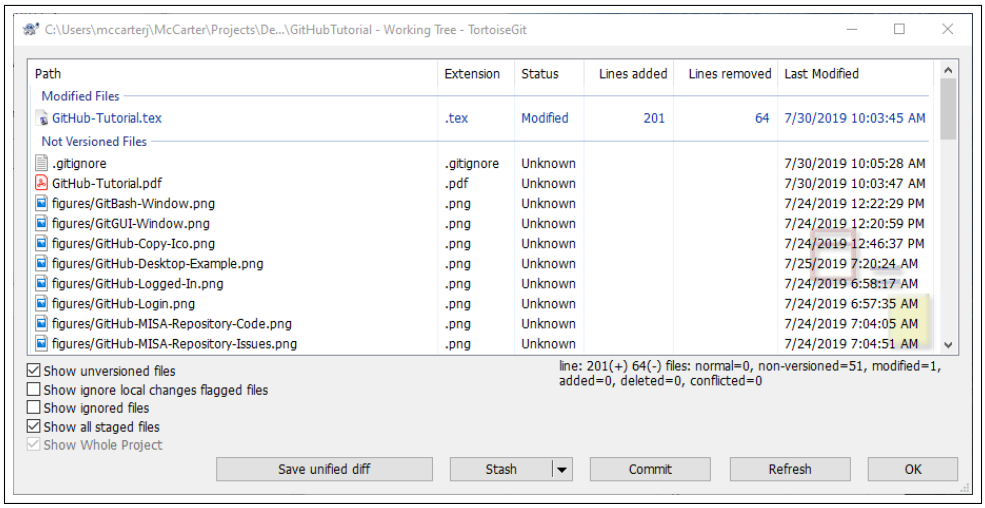


Figure 2.21: Tortious Git - Check For Modifications Dialog on Folder

Click the Commit button and you will see a Commit Dialog (Figure 2.22). Enter a message for the file or group of files and click the Commit button. You will see a confirmation dialog that will detail any issues encountered most often indicating the process was successful. Click Close to dismiss the dialog.

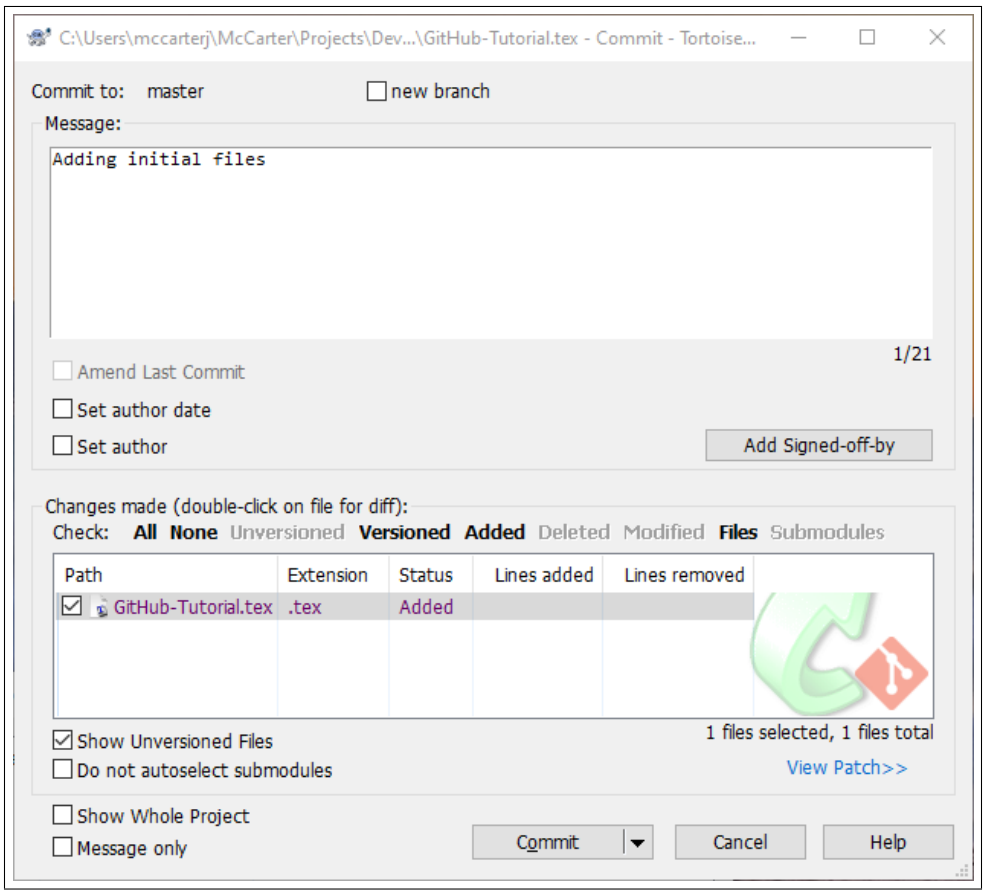


Figure 2.22: Tortious Git - Commit Dialog

Looking back at the folder in Windows Explorer we see that our file is now highlighted with a green check icon (Figure 2.23), indicating it is checked into the repository and "up to date" – no changes have been made to your local working copy.

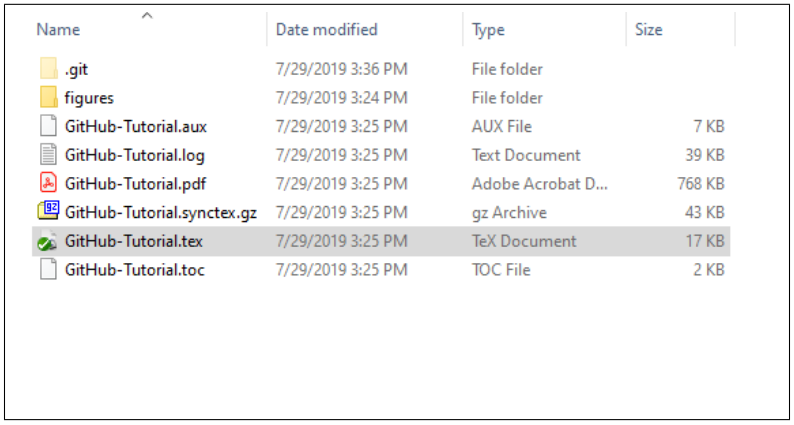


Figure 2.23: Tortious Git - Committed

2.2 Intermediate Git

The following section provides some more advanced Git topics and presents a typical workflow for using Git tools.

2.2.1 Typical Daily Workflow

The typical daily workflow consists of editing and/or creating files, deciding to add files to repository, deciding to commit updates to repository, and synchronizing the local working copy with the GitHub repository. For additional information see the online TortoiseGit Daily Use Guide.

1. Make edits to existing files. In this example, I've modified GitHub-Tutorial.tex and GitHub-Tutorial.pdf as indicated by the red exclamation icon next to those files.

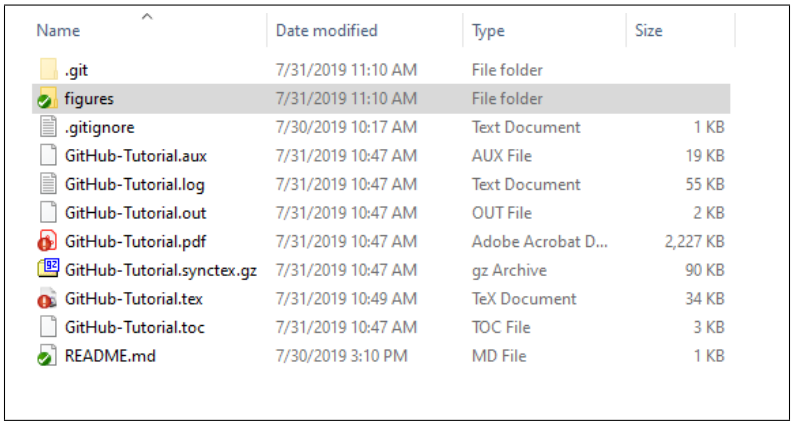


Figure 2.24: Windows Explorer showing modified files with TortoiseGit icons

2. Create new files as needed, add to repository

3. Check status of files and determine any additional files to add to the repository.

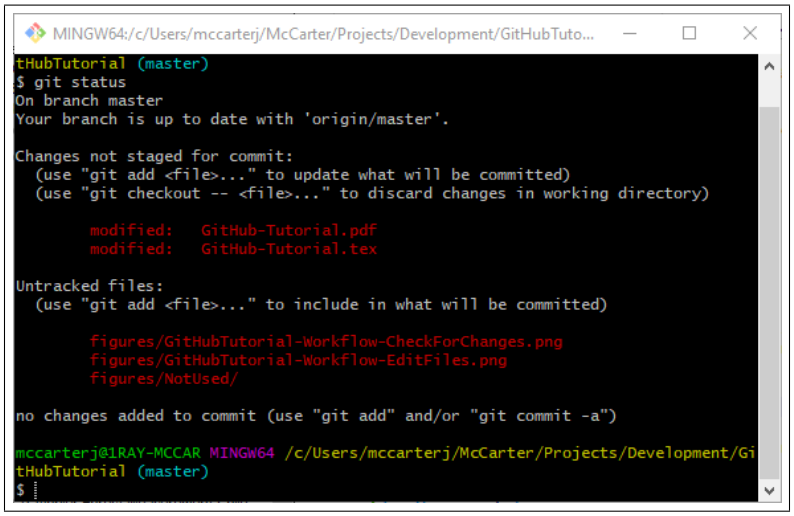


Figure 2.25: Git Status using Git Bash

You can use **Git Bash** using the *git status* command also shows the two files modified in addition to several files in the figures folder that have been added to the folder, but not added to the repository yet. Git Bash can be used to add files to the repository with the *git add filename* command.

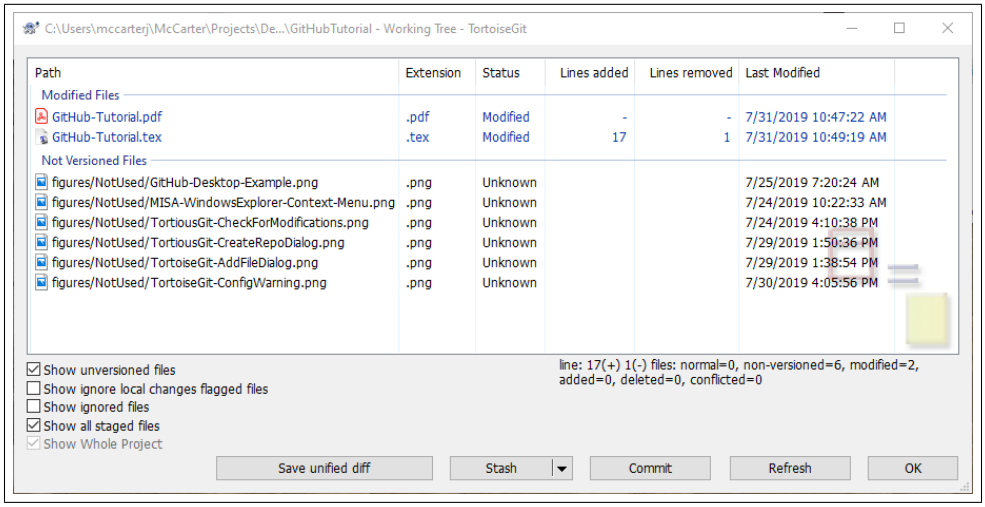


Figure 2.26: Checking for modifications with TortoiseGit

Alternatively we can use the TortoiseGit - **Check for modifications** menu command resulting in the dialog above that shows files that have been modified and files that are not currently in the repository under "Not Versioned Files". You can easily add files to the repository by right clicking on a file and selecting Add (Figure 2.27).

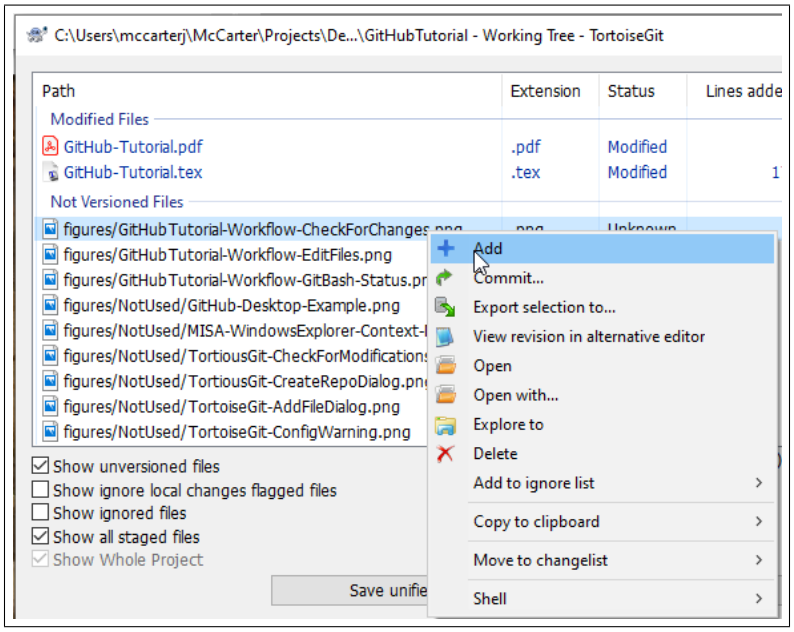


Figure 2.27: Add file to repository from TortoiseGit Working Tree dialog

4. Commit changes to local repository - From the TortoiseGit Working Tree dialog you can perform a commit for the group of files highlighted. Enter your commit message and click the Commit button.

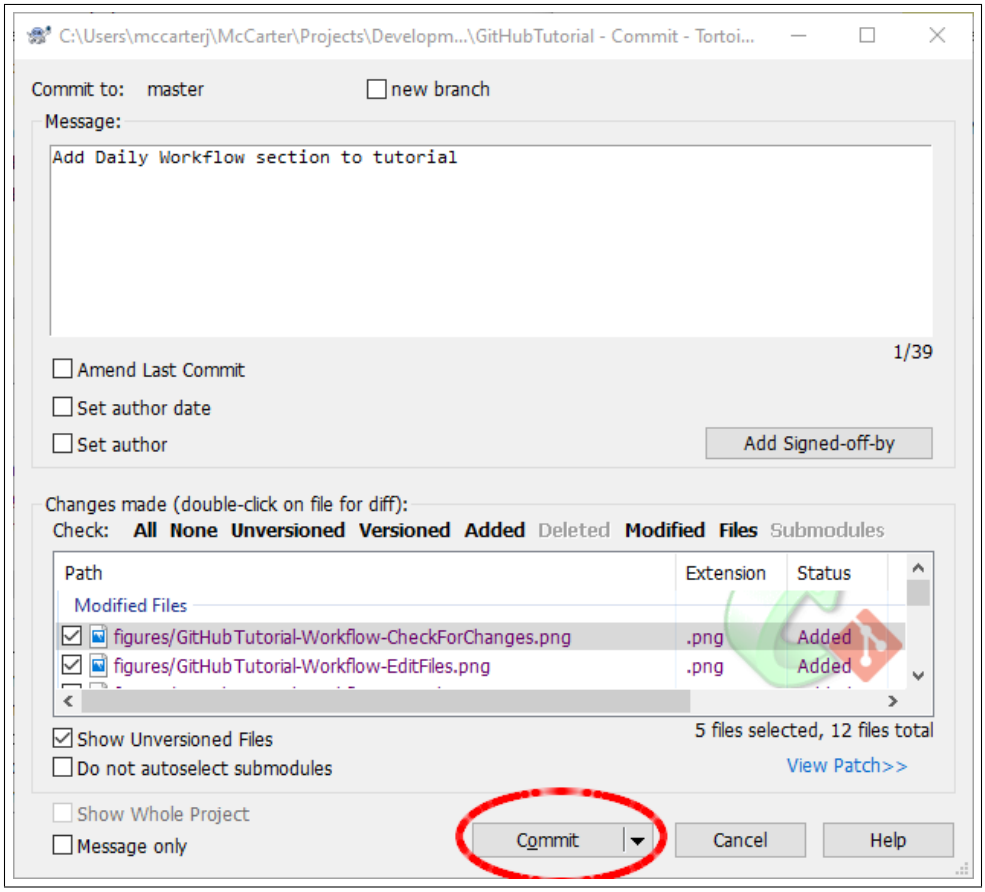


Figure 2.28: Commit changes to local repository with TortoiseGit

The commit will be processed and status of the process will be displayed in the confirmation dialog.

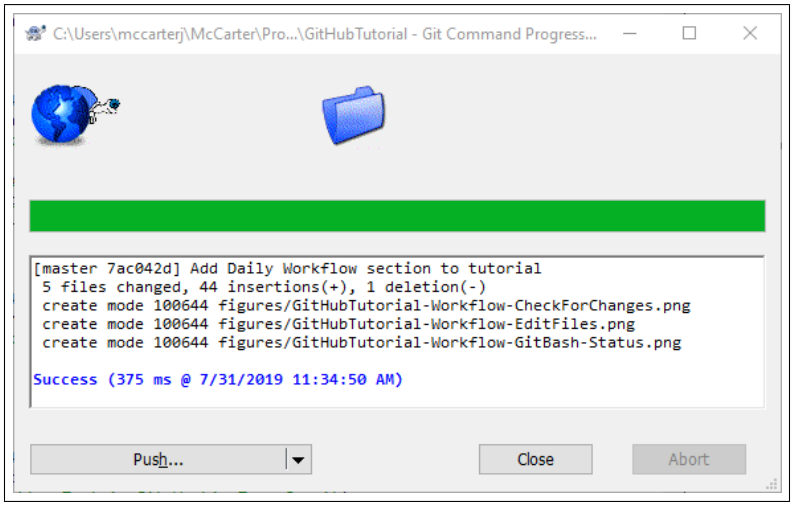


Figure 2.29: TortoiseGit Commit Confirmation dialog

5. Push to GitHub

   Local repository changes can be pushed to GitHub.com and shared with the team using **GitHub Desktop**. Figure 2.30 displaying the history of commits and shows that there are two file modifications and three new files to be pushed to GitHub.com. Click the **Push origin** button to upload the changes.

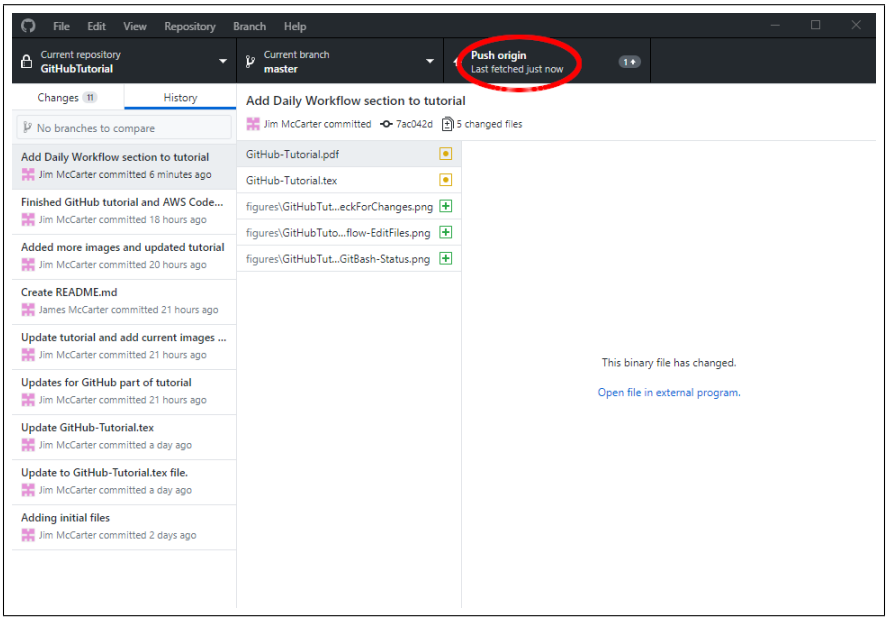


Figure 2.30: GitHub Desktop, ready to Push changes to GitHub.com

After the files are synced the button changes to **Fetch origin**.

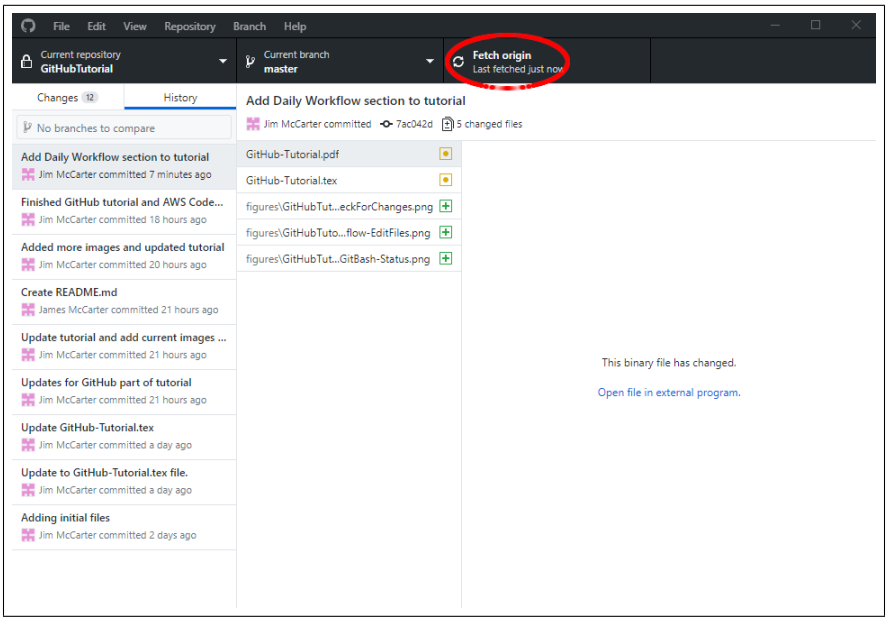


Figure 2.31: GitHub Desktop after Push

We can also look on GitHub.com and see that the files were recently committed.

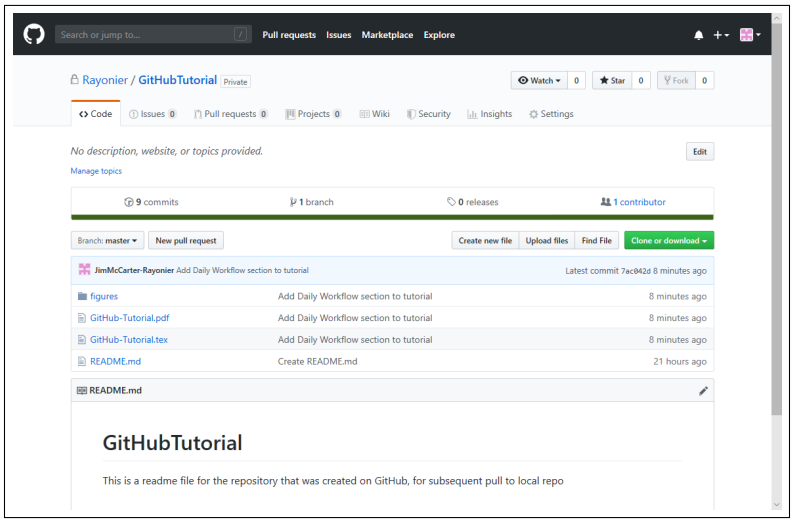


Figure 2.32: GitHub.com after push

2.2.2 File Differences

While editing code it is sometimes useful to compare the current working copy to a previous version to see where and what changes have being made to a file. Having version control allows you to be more aggressive in removing old code, because you know previously committed copies of files are available. The current working copy can easily be compared to any previous version.

File differences can be shown in Git GUI:

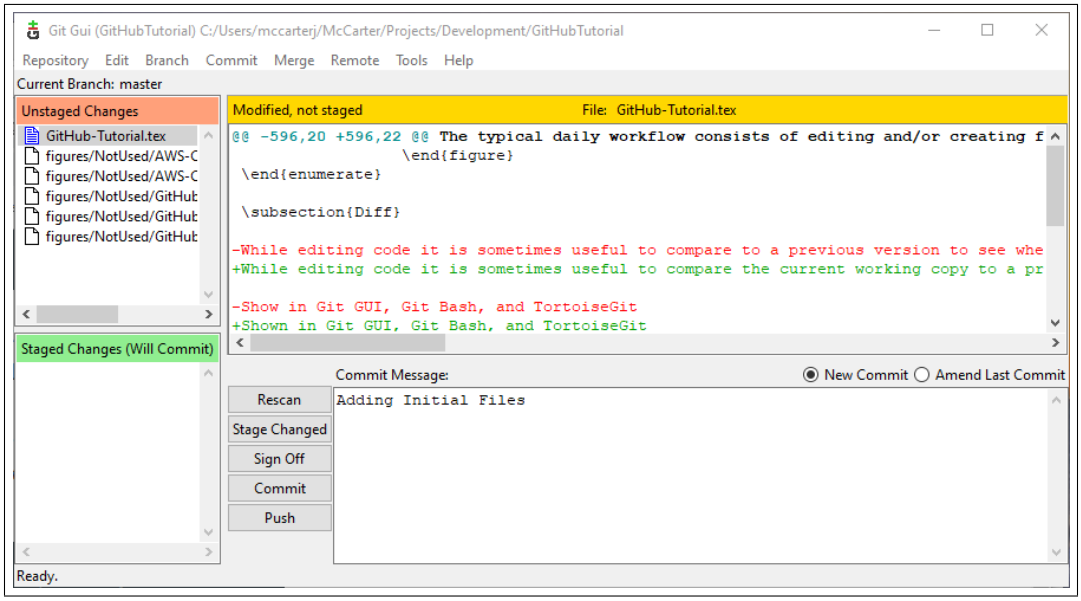


Figure 2.33: GitGUI showing differences in GitHub-Tutorial.tex file

Differences can also be shown in Git Bash using the "git diff HEAD GitHub-Tutorial.tex" command to compare the current working file to the most recent copy in the archive.

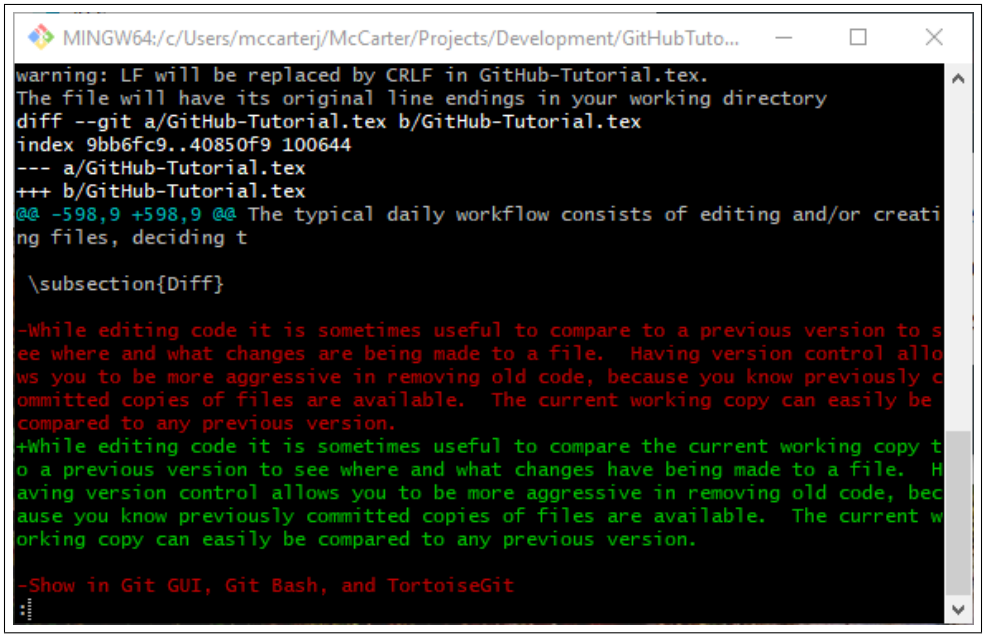


Figure 2.34: GitBash showing differences in GitHub-Tutorial.tex file

With TortoiseGit you can "Check for modifications..." to display the Working Tree dialog:

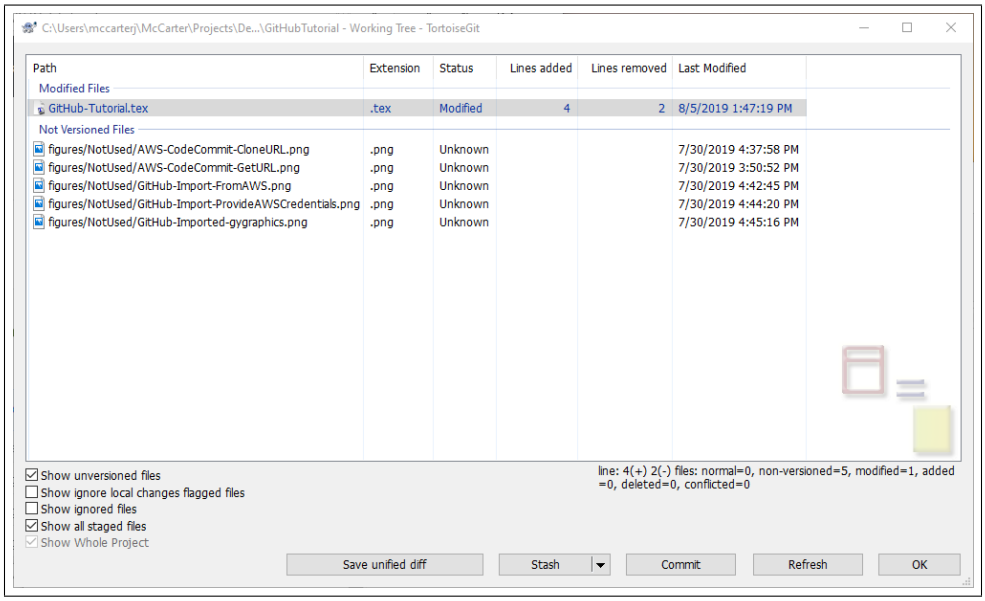


Figure 2.35: TortoiseGit showing Working Tree and which files have differences

From the Working Tree dialog you have several options on how you want to compare the current working file.

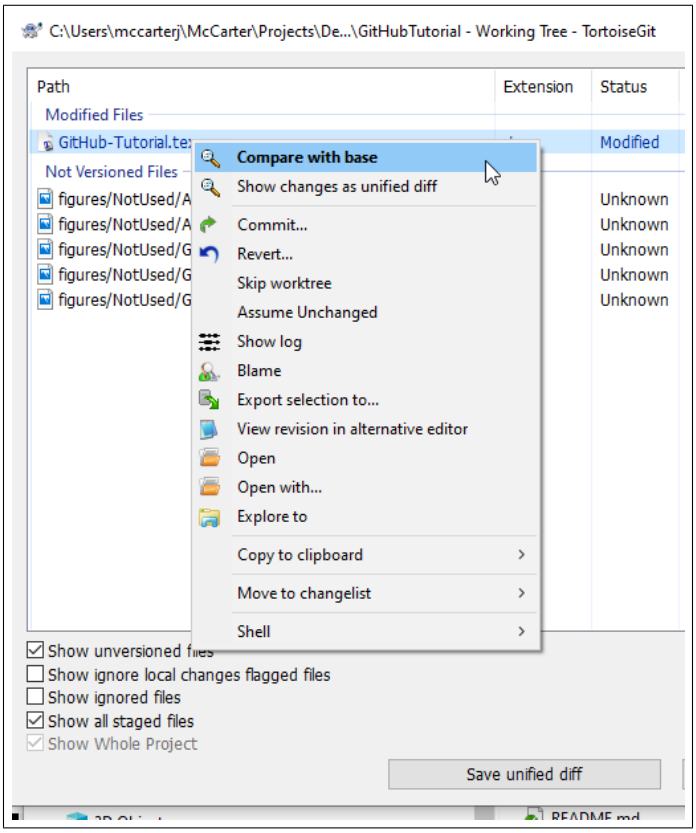


Figure 2.36: TortoiseGit showing options for comparing GitHub-Tutorial.tex file

TortoiseGit shows a nice side-by-side view of the changes:

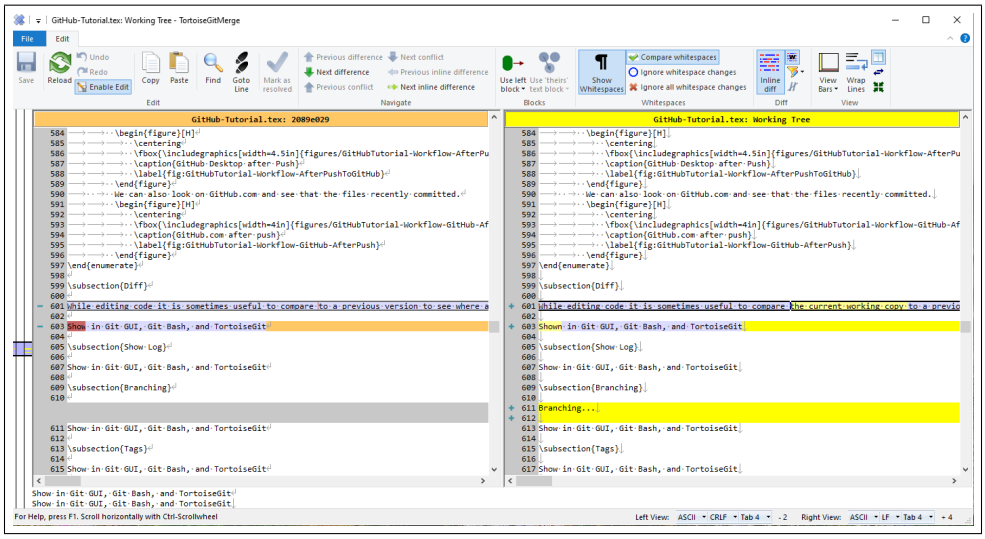


Figure 2.37: TortoiseGit showing differences in GitHub-Tutorial.tex file

2.2.3 Show Log

Git GUI - Repository/Visualize master's or All Branch History

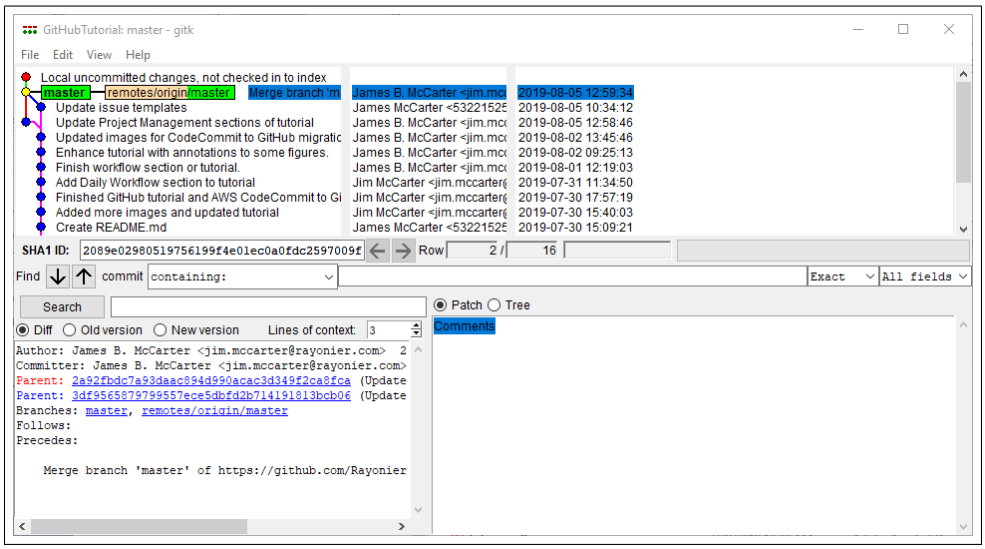
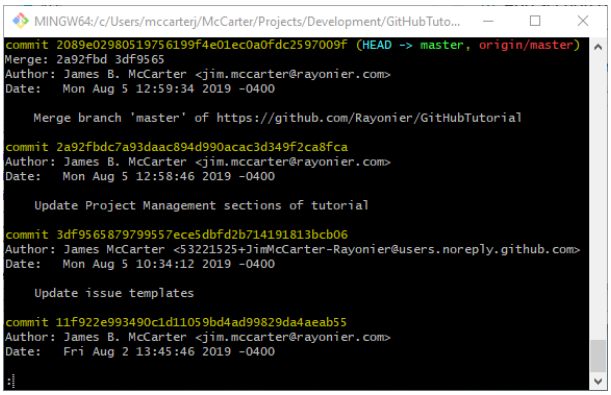
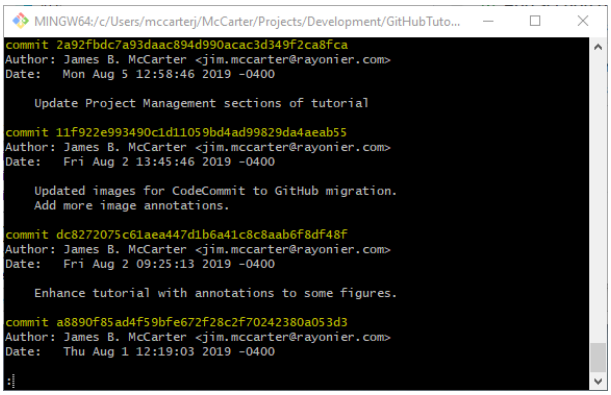


Figure 2.38: Git GUI Repository/Visualize master's History or Visualize All Branch History

Git Bash - "git log" or "git log GitHub-Tutorial.tex".



(a) git log



(b) git log "path"

Figure 2.39: Git Bash Show Log.

TortoiseGit - Show log or Show Reflog

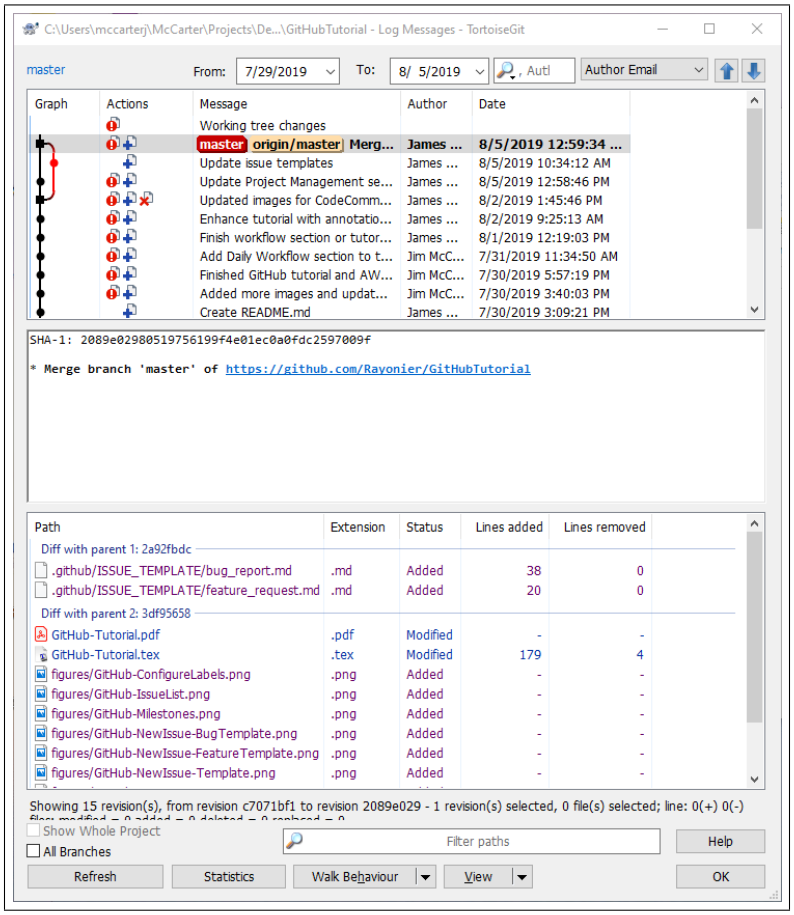


Figure 2.40: TortoiseGit View Log

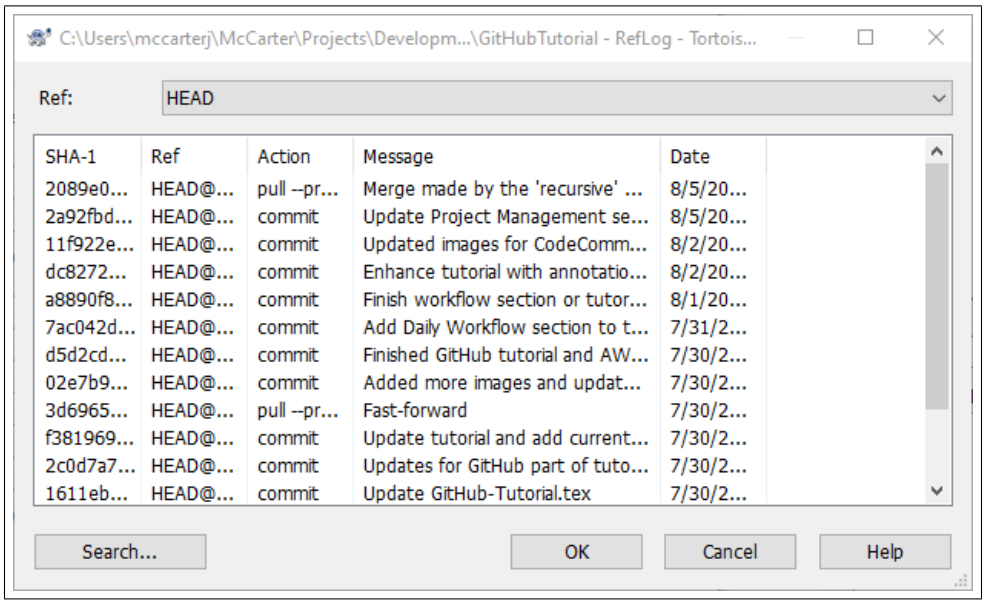


Figure 2.41: TortoiseGit View Reflog

2.2.4 Branching and Merging

Typical code development follows a somewhat linear path, and often our source code repositories only contain a linear history. Larger projects or efforts with more than one developer may require simultaneous or parallel edits to the source code base. Also, multiple releases to support different user requirements or operating systems platforms may need to be maintained. Branches (or forks) in the context of version control software supports flexible management of this environment.

But when more than one developer is working on a project they may need to make simultaneous or parallel edits. This process is aided with version control software with support for branching or forking the development path, often referred to as a tree.

Additional information: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Branch and Merge with GitGUI

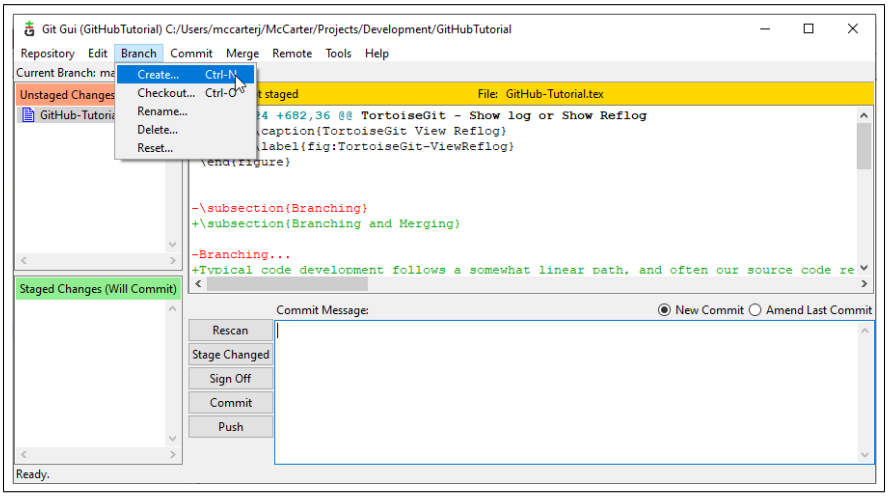


Figure 2.42: Git GUI - Branching

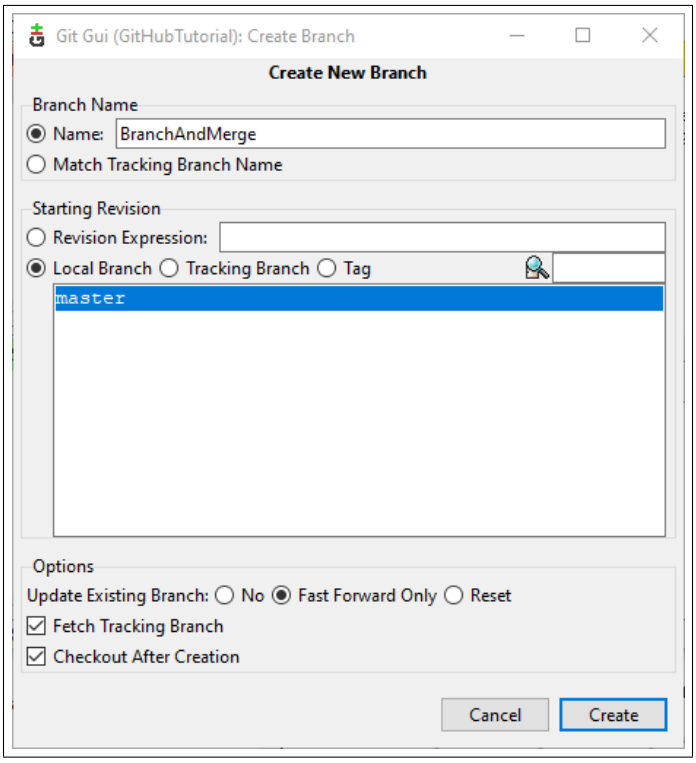


Figure 2.43: Git GUI - Branch and Merge

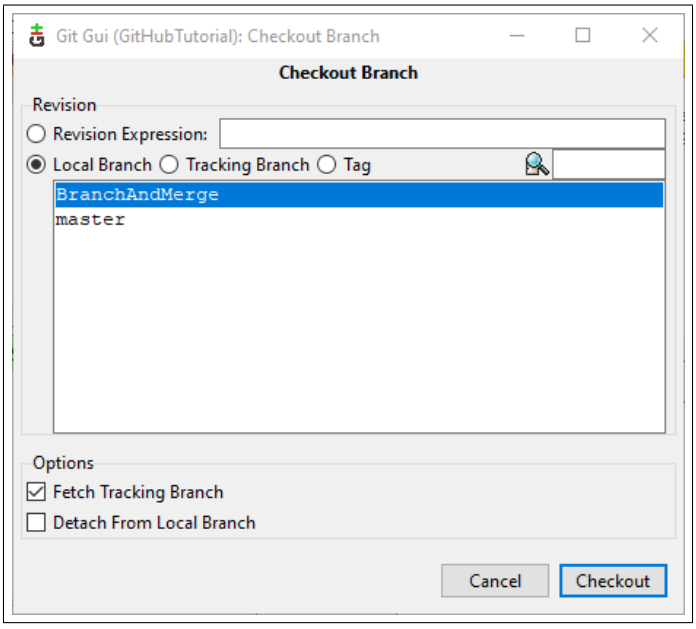


Figure 2.44: Git GUI - Branch Checkout

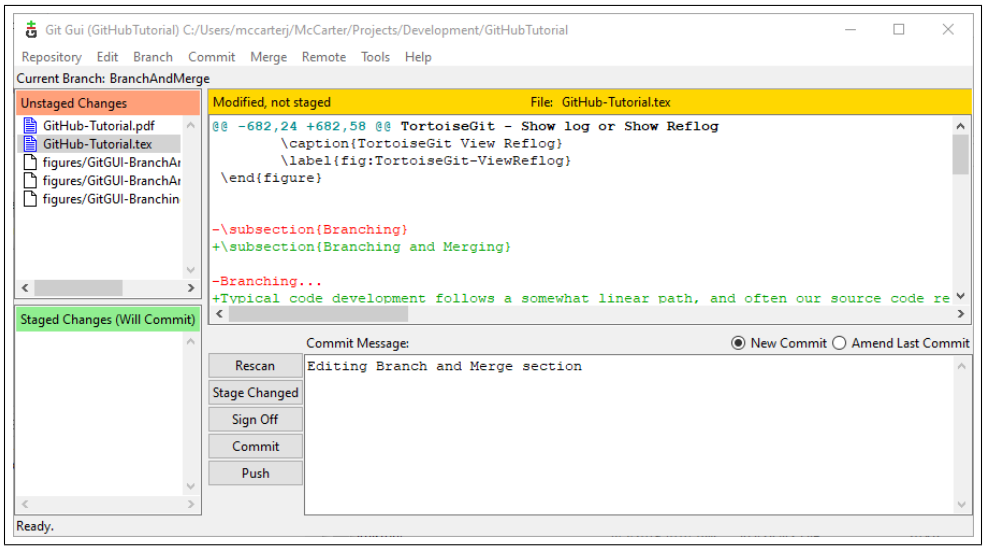


Figure 2.45: Git GUI - Working on Branch

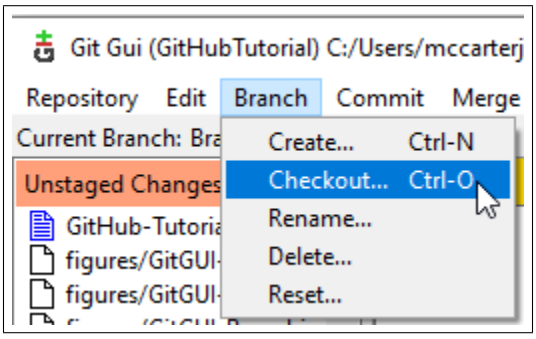


Figure 2.46: Git GUI - Working on Branch

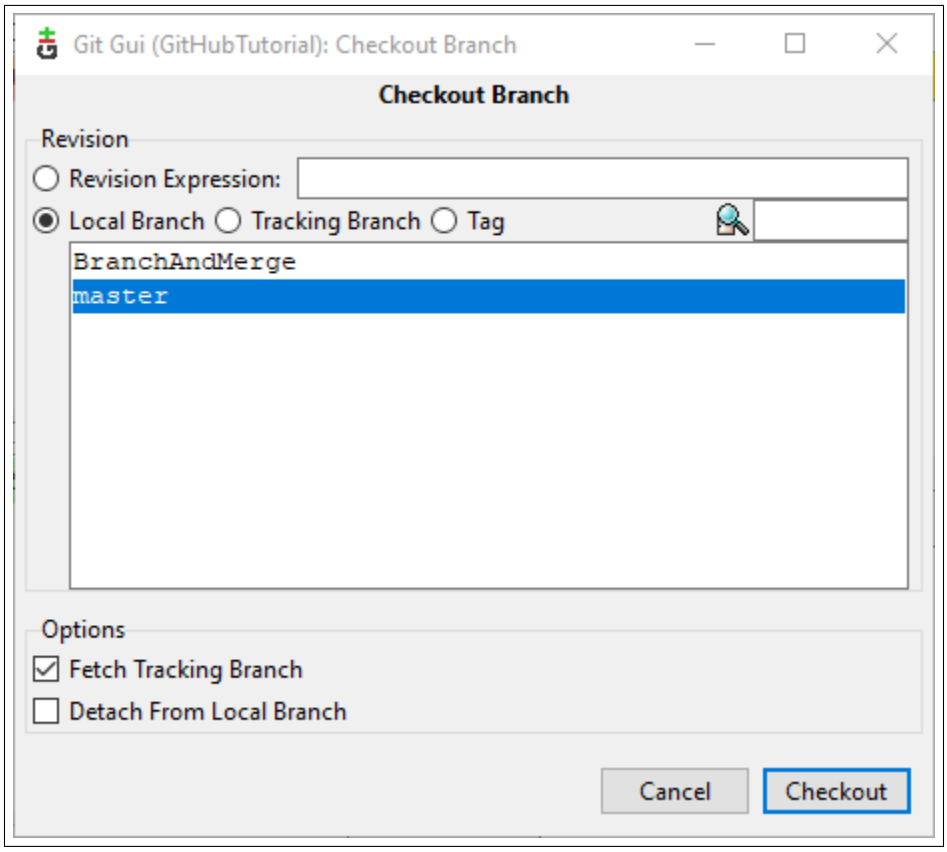


Figure 2.47: Git GUI - Working on Branch

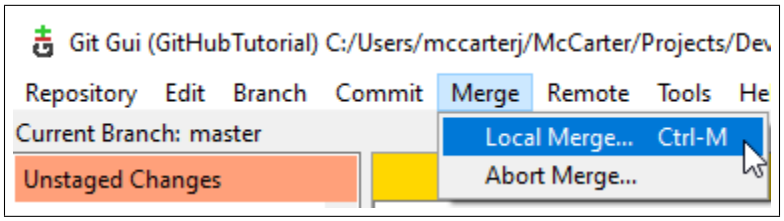


Figure 2.48: Git GUI - Working on Branch

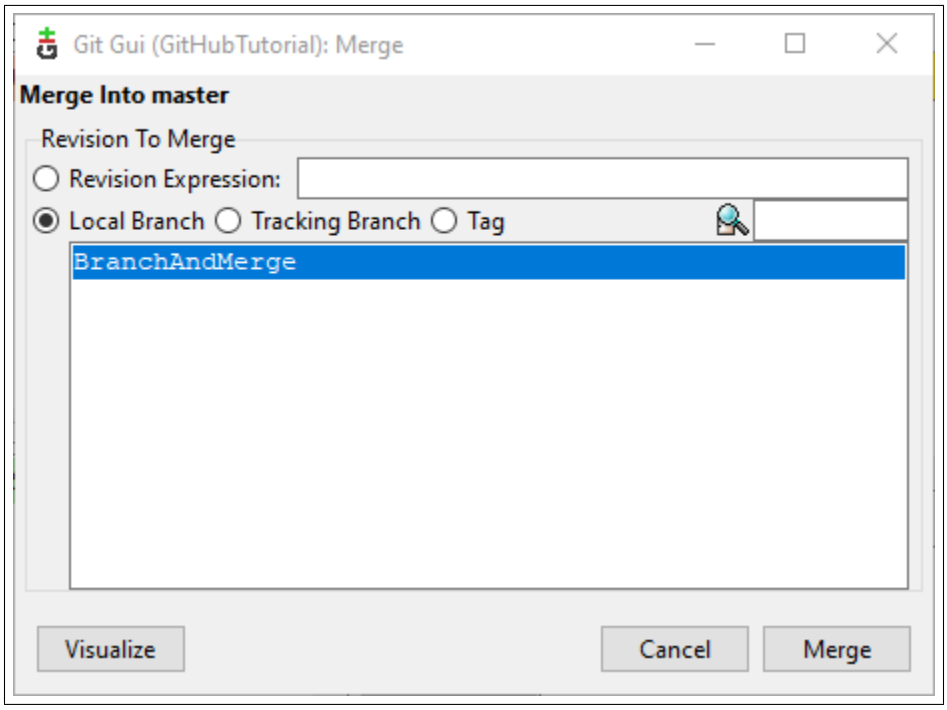


Figure 2.49: Git GUI - Working on Branch

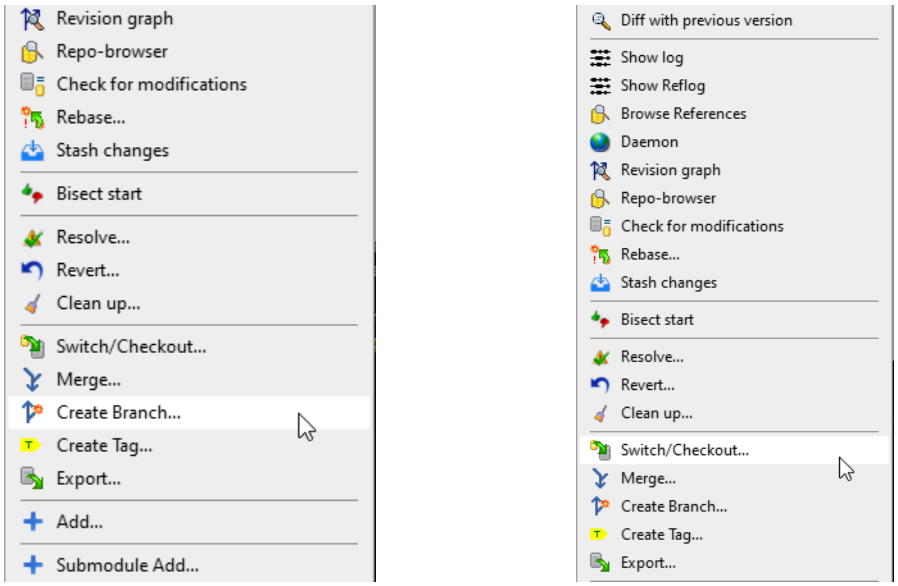
Branch and Merge with Git Bash

In this example we will demonstrate the workflow of creating a branch to fix a given issue (#53 in this example). In this workflow we first create the branch, make edits to fix the issue, commit the source code changes, checkout the "master" branch, merge "issue53" edits, then delete the issue53 branch.

- git checkout -b issue53
  - git branch issue53
  - git checkout issue53
- make edits
- git commit -a -m "made required edits to fix issue 53"
- git checkout master
- git merge issue53
- git branch -d issue53

Branch and Merge with TortoiseGit

The process using TortoiseGit is similar, but used Windows Explorer context menus.



(a) Create Branch (b) Switch/Checkout Branch

Figure 2.50: Tortoise Git - Create and Checkout Branch

You can switch back and forth between branches.

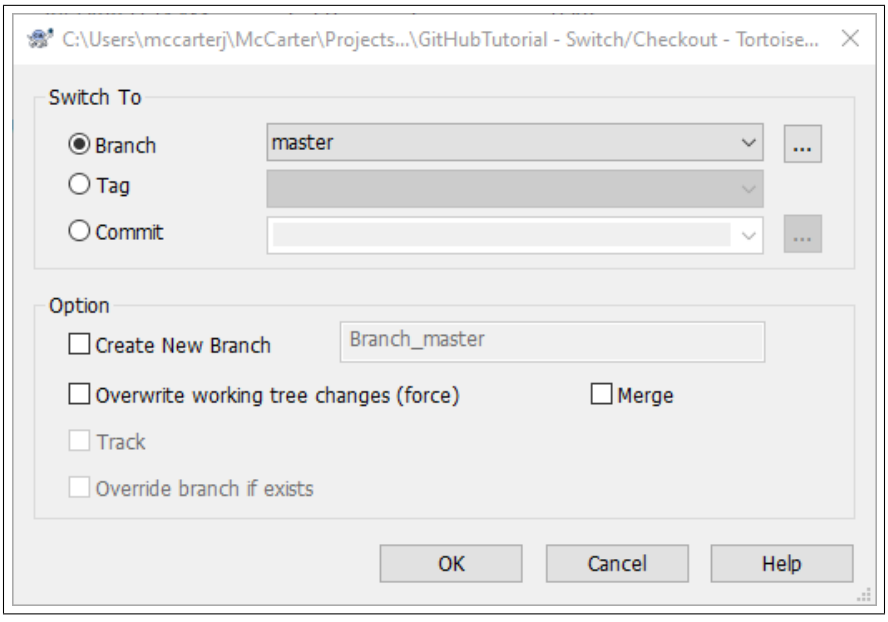


Figure 2.51: TortoiseGit - Switch/Checkout Dialog

When edits are done, you can merge branches.

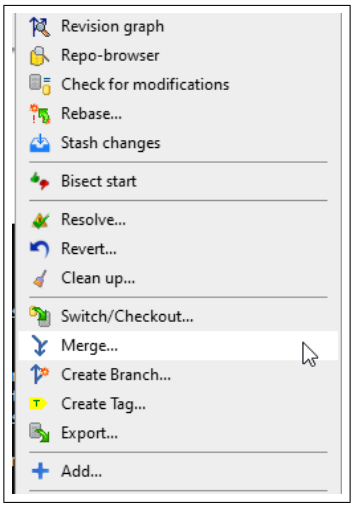


Figure 2.52: TortoiseGit - Menu - Merge Branch

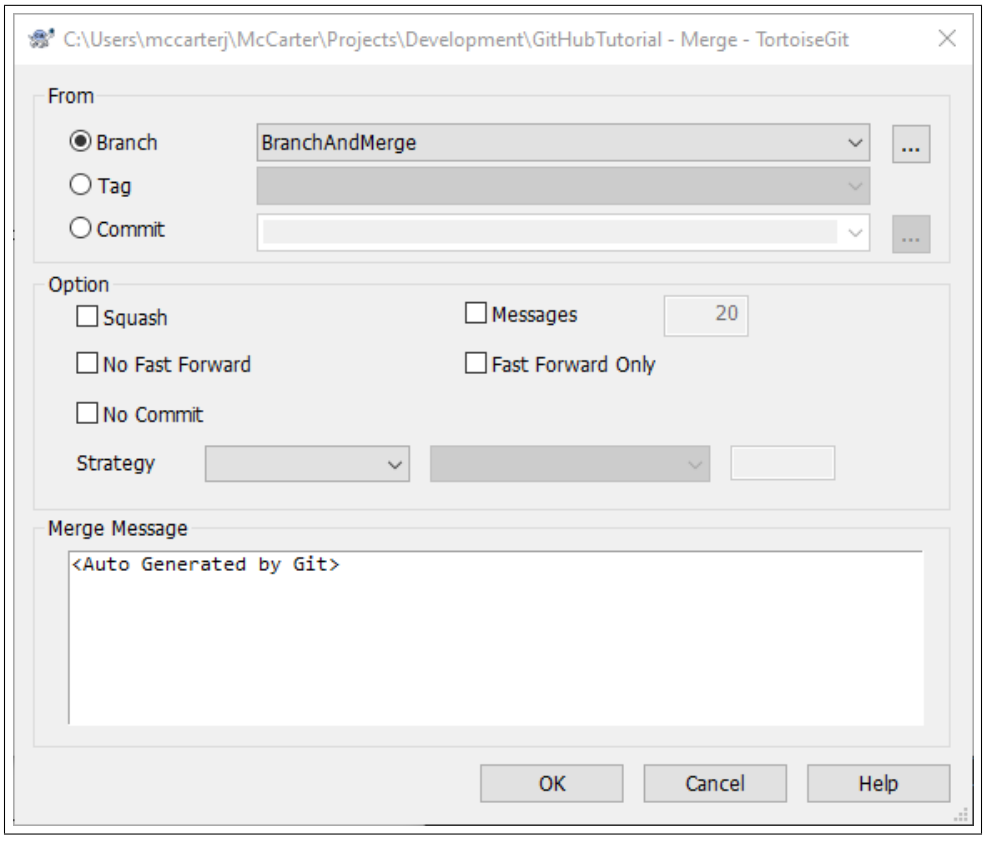


Figure 2.53: TortoiseGit - Merge Branch Dialog

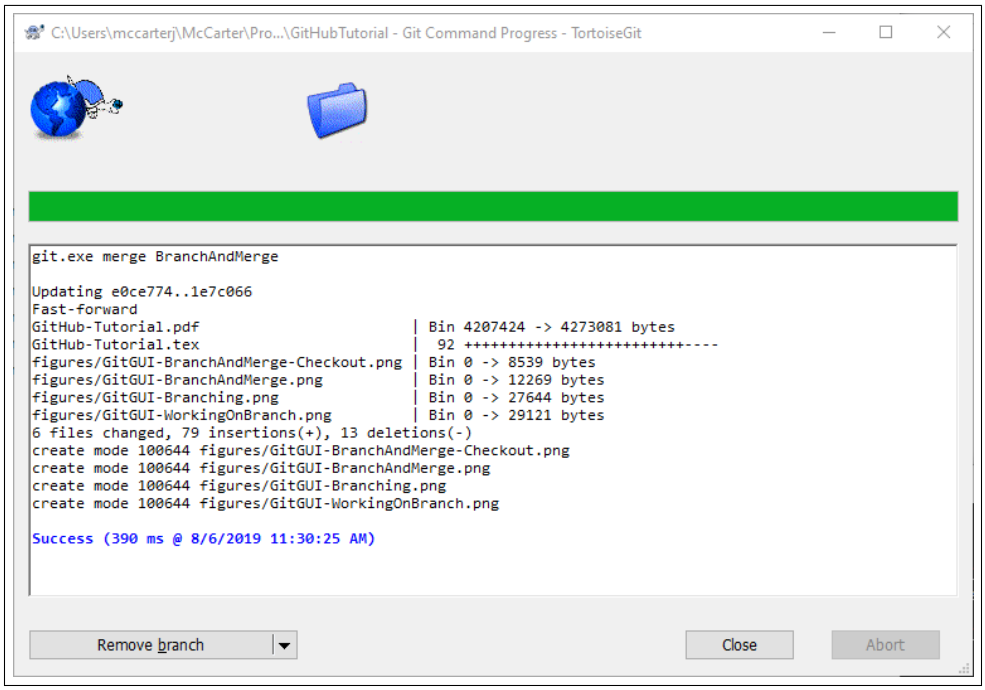


Figure 2.54: TortoiseGit - Merge Branch Confirmation

Branching and Merging Best Practices

Best practice is **branch out**, **merge often**, and **keep in sync**.

Branch when:

- You are about to implement major or disruptive change
- You are about to make changes that might not be used
- You want to experiment on something that might not work
- You are told to branch, because others are making updates to master

See Git branching and tagging best practices on sofwareenginering.stackexchange.com for additional insights.

2.2.5 Tags

A **tag** is a way to mark a point in time of a repository. You can use a tag to mark any point in the development process, but they are typically use to mark release points of software.

Each commit to the repository has a unique identifier (e.g. GitHubTutorial commit id f24c65506abe071abad946ccab00769327242dac and shortened SHA-1 id f23c655). Tags provide an annotated pointer to the commit

You are tagging commits to the repository. Think of a tag as an annotated pointer to a commit.

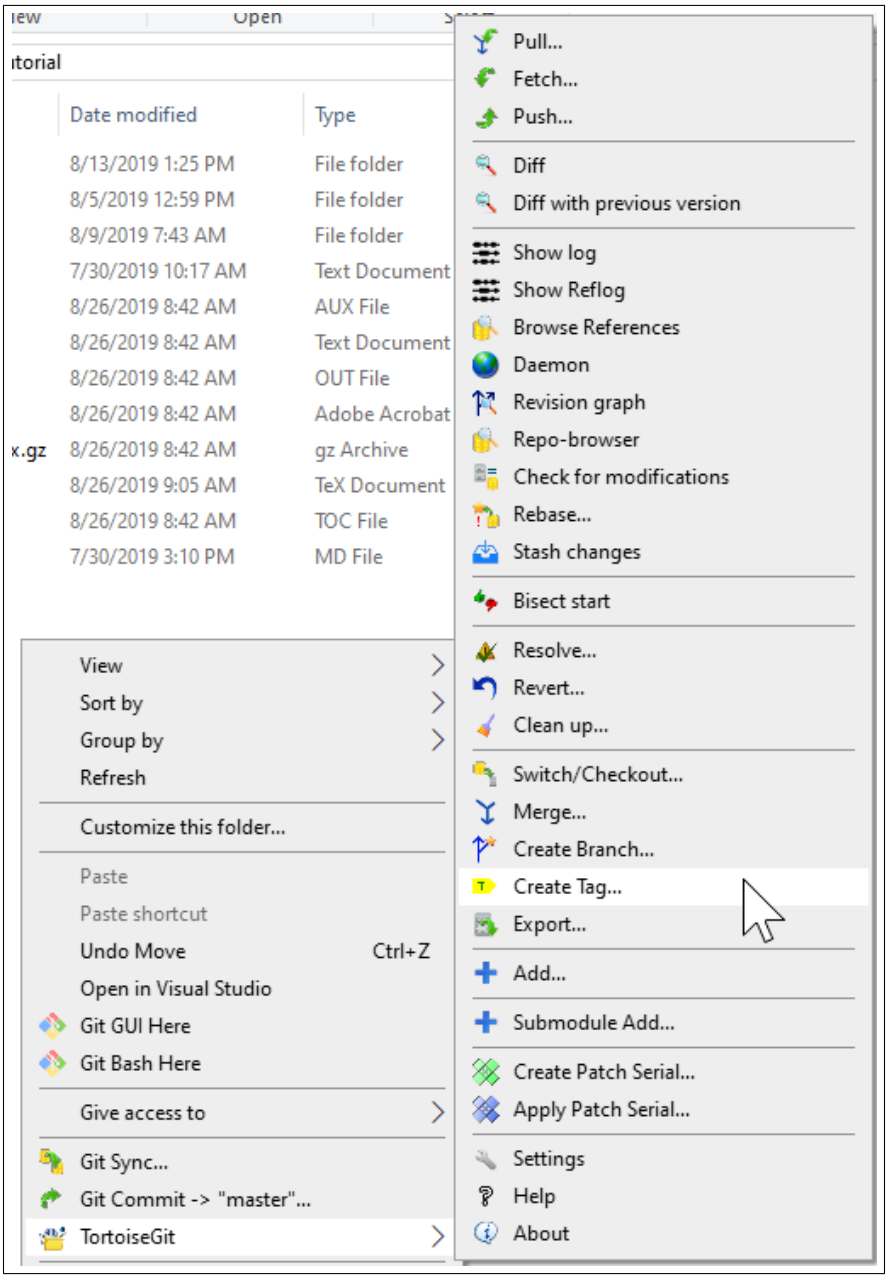


Figure 2.55: TortoiseGit - Menu - Create Tag

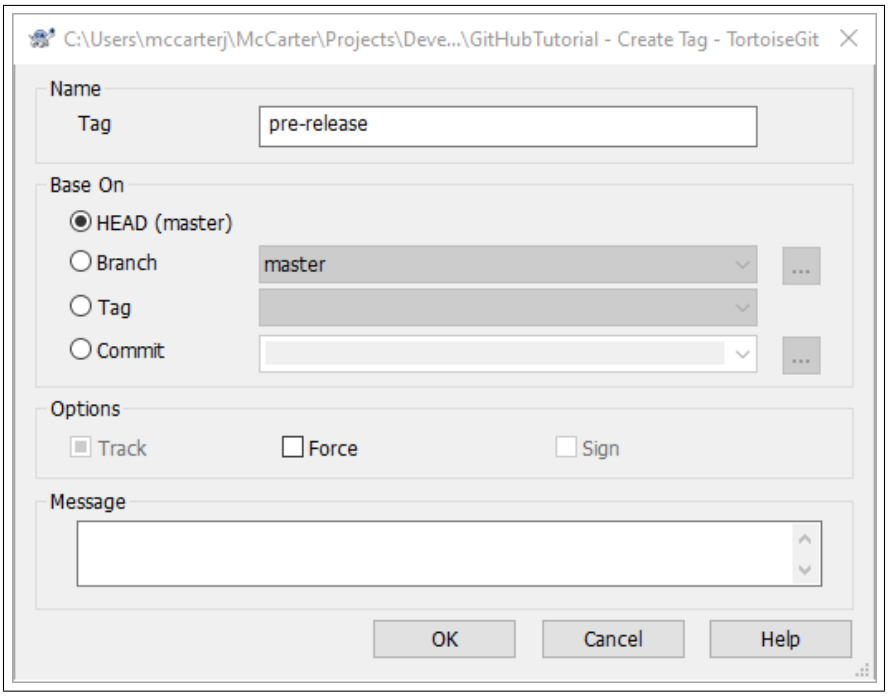


Figure 2.56: TortoiseGit - Create Tag Dialog

2.2.6 Stashing

Stashing takes the working directory in its *dirty* state, with edits of tracked files, and reverts the directory to match the HEAD commit. This allows you to store uncommitted modifications into a buffer area called the **stash**.

https://git-scm.com/book/en/v1/Git-Tools-Stashing

https://tortoisegit.org/docs/tortoisegit/tgit-dug-stash.html

2.2.7 Forking versus Branching

https://stackoverflow.com/questions/3611256/forking-vs-branching-in-github

2.2.8 Integrated Development Environment (IDE) Integration with Git

Several of the common Integrated Development Environment (IDE) tools already in use at Rayonier have source control integration that supports Git.

RStudio

RStudio Version Control using Git and Subversion (SVN). RStudio supports local Git and GitHub hosted repositories. To start, make sure RStudio is configured to use Git by selecting Tools / Global Options and selecting the Git/SVN tab. Confirm or check "Enable version control interface for RStudio project" and confirm the path to git.exe.

https://support.rstudio.com/hc/en-us/articles/200532077?version=1.2.1335&mode=desktop

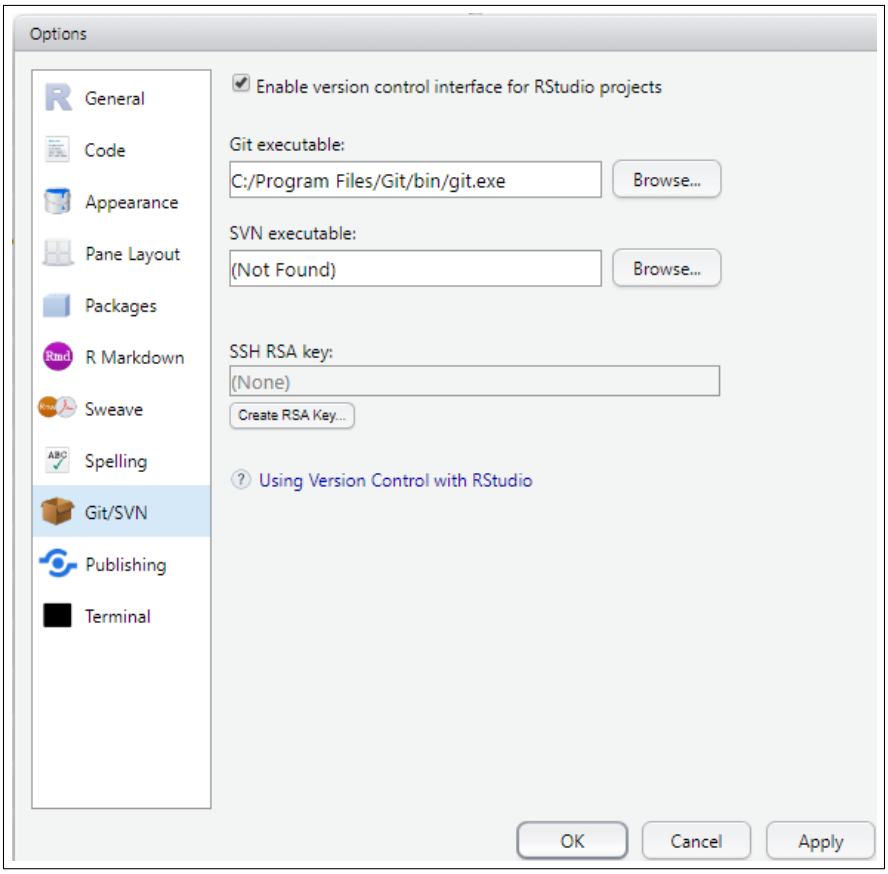


Figure 2.57: RStudio - Global Options

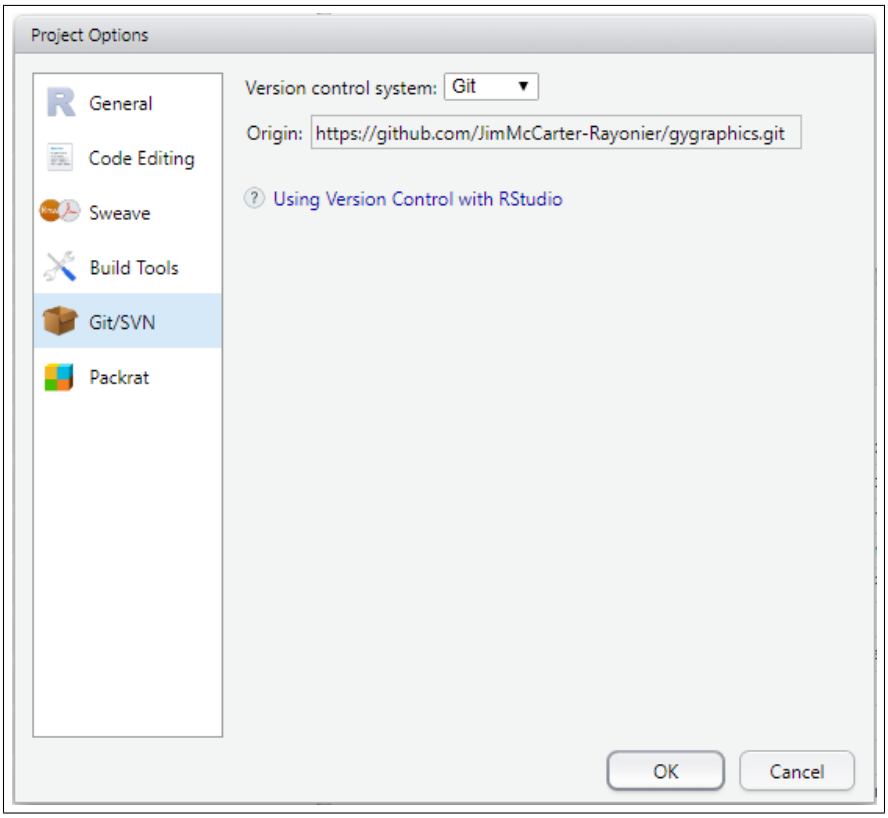


Figure 2.58: RStudio - Project Options

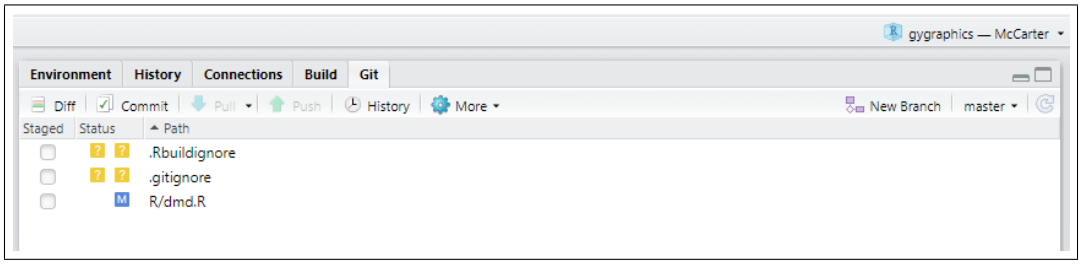


Figure 2.59: RStudio - Git Tab

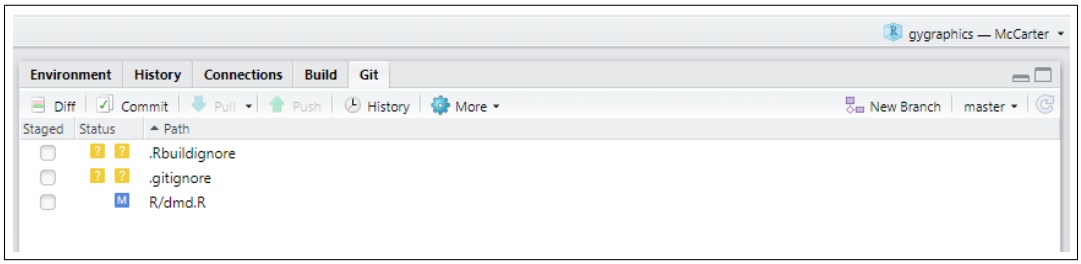


Figure 2.60: RStudio - Git Menu when editing

SlickEdit Pro

SlickEdit Pro is a powerful project level editor with compiler, version control, project management, and other features. It can integrate with many version control systems, including Git. It does not integrate directly with GitHub, instead focusing on the local repository and then relying on other tools (e.g. GitHub Desktop) to provide local to hosted synchronization.

Git is configured in SlickEdit by selecting the Tools / Options menu command and then navigating to Tools / Version Control / Version Control Setup to select Git.

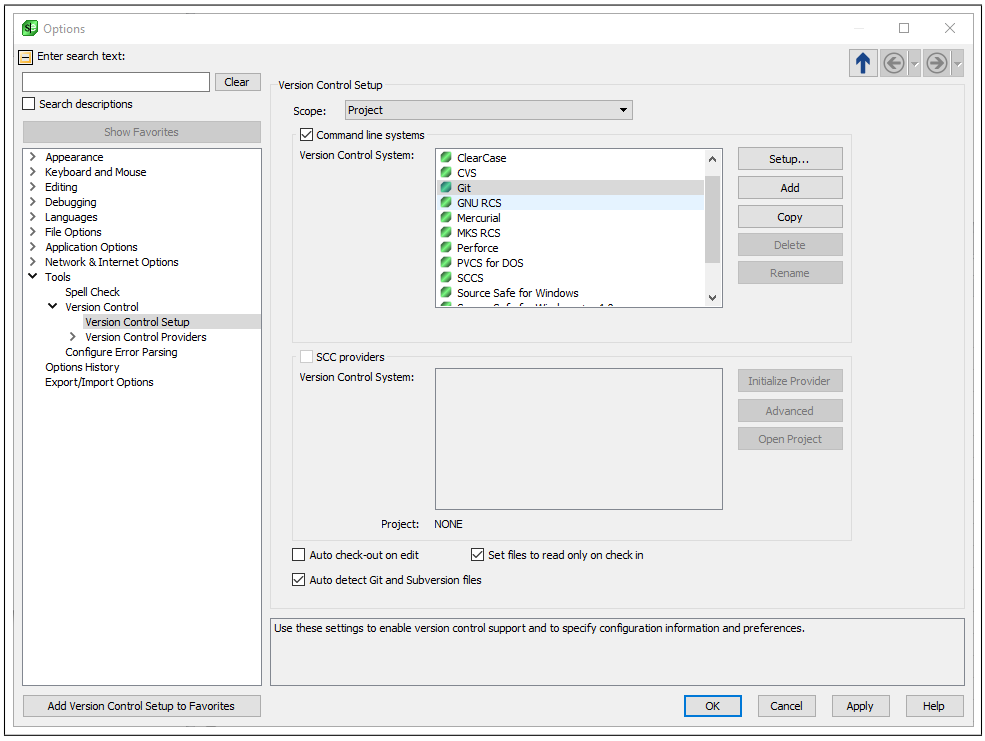


Figure 2.61: SlickEdit - Options / Version Control Setup

Specific provides for service can also be selected, so confirm the correct Git.exe is selected under Version Control Providers.

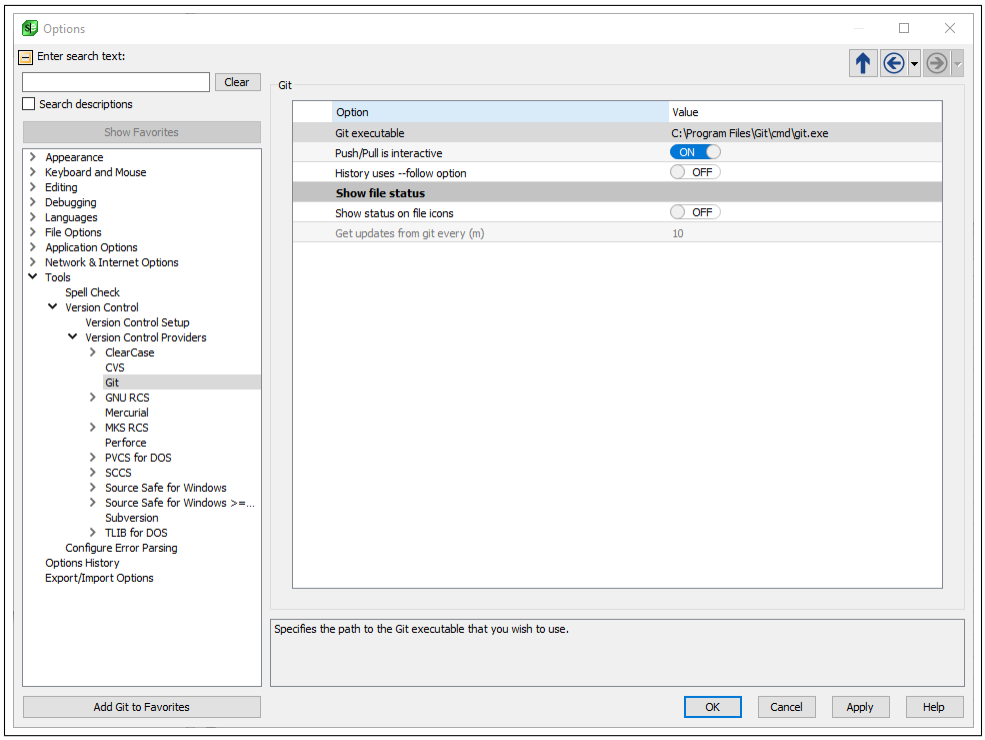


Figure 2.62: Visual SlickEdit - Options / Version Control Provider Setup

Once Version control is configured in the application all project will have access to the features using context menus. The example blow shows the context menu displayed by right clicking on a file in the Project Pane and then selecting the Version Control sub-menu. Many typical Git functions are available through this interface.

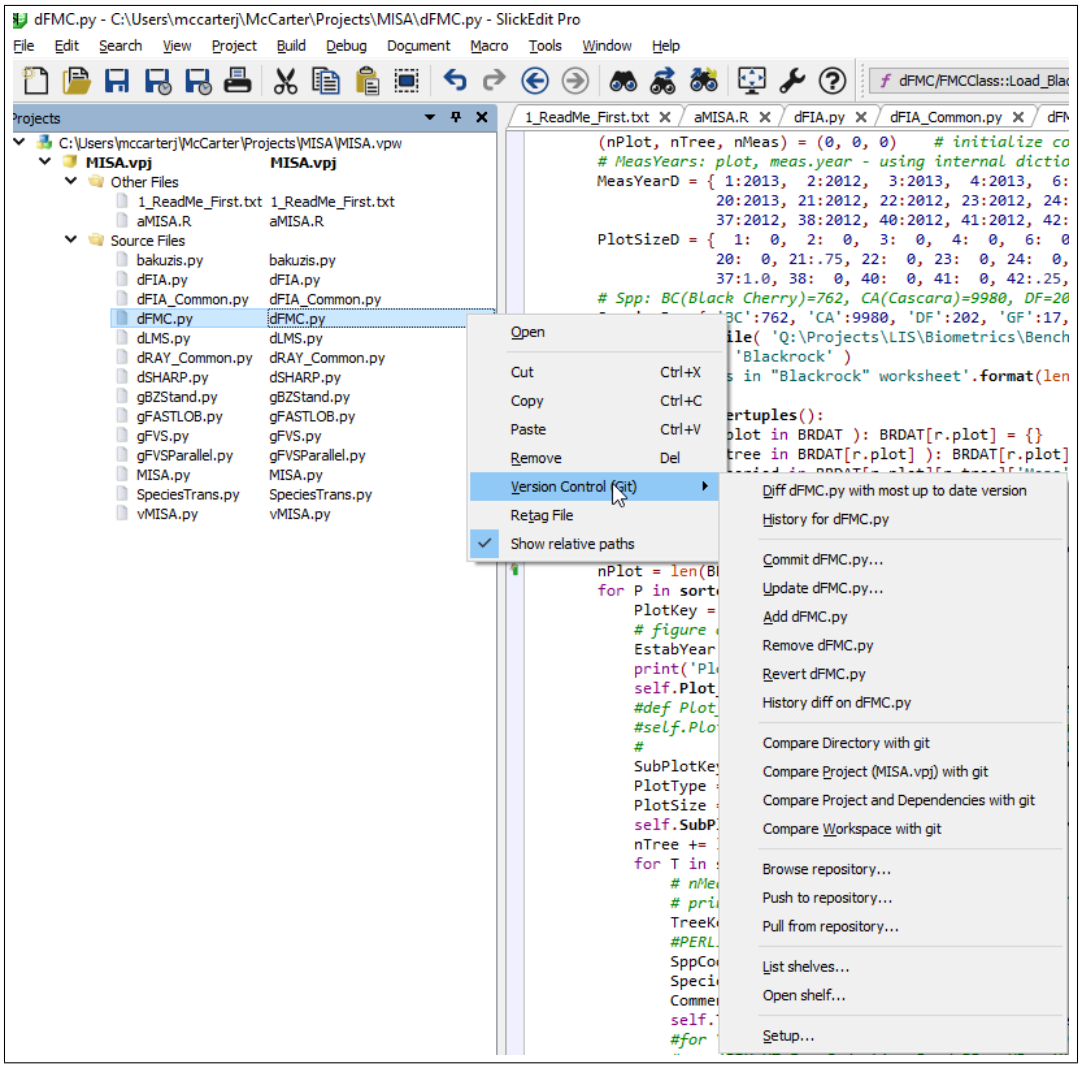


Figure 2.63: SlickEdit

Visual Studio

Microsoft Visual Studio can also be configured to work with Git. Select Tools/Options/Source Control/Plug-in Selection and select Git from the drop-down menu.

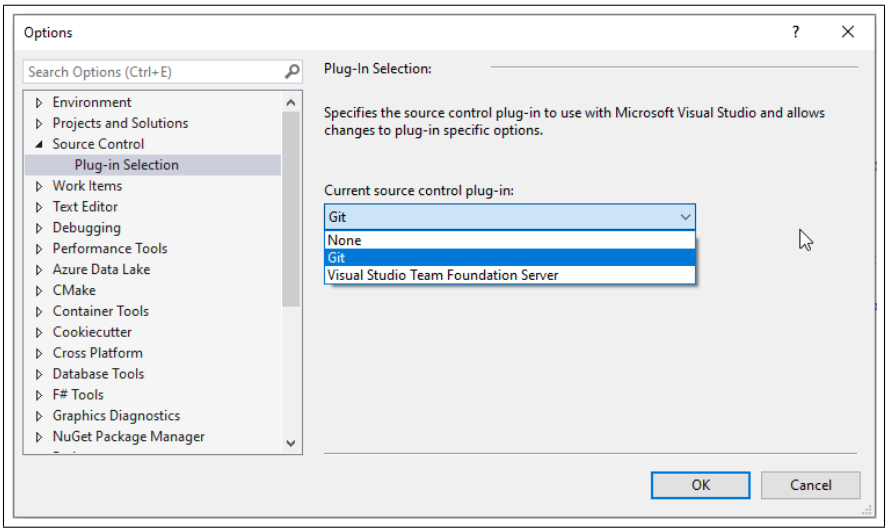
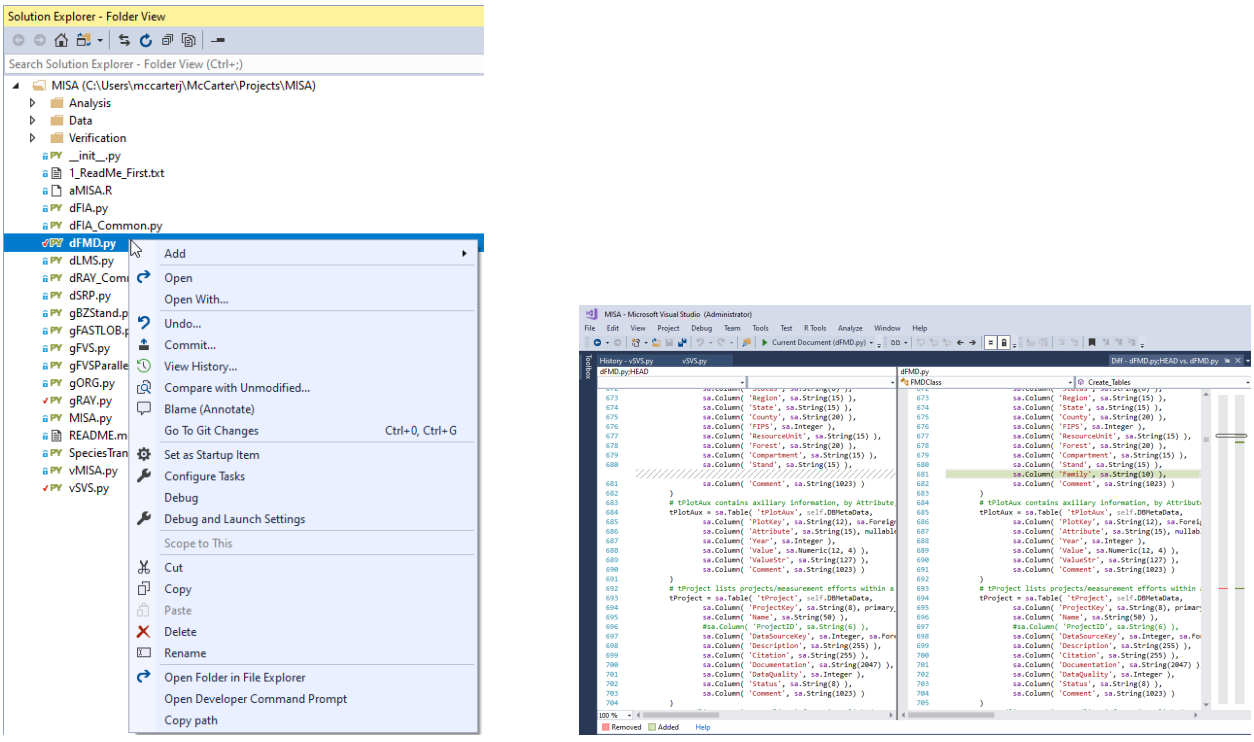


Figure 2.64: Visual Studio - Source Control configuration for Git



(a) Solution Explorer/Context Menu

(b) Compare with unmodified

Figure 2.65: Visual Studio/Gut Integration

2.2.9 General Guidance on Version Control

- Use a descriptive commit message
- Make each commit a logical unit
- Avoid indiscriminate commits - know which files you are committing and provide a specific log message that applies to that group of changes.
- Don't commit generated files

https://alvinalexander.com/git/git-cheat-sheet-git-reference-commands

https://homes.cs.washington.edu/ mernst/advice/version-control.html

2.3 Advanced Git

2.3.1 Squash/Rebase Commits

When actively modifying source code and frequent edit/commit/test cycle is desirable when growing or refactoring code. This may result in a large number of commits that we don't want in a published repository and that make it more difficult for review of system changes.

Git and TortoiseGit support squashing of adjacent commits into a single commit while preserving all the comments associated with the multiple commits.

https://tortoisegit.org/docs/tortoisegit/tgit-dug-rebase.html

2.3.2 Merge vs Rebase

When multiple people are working on a project it is desirable to branch/fork the repository as new features are added. Depending on the development cycle one of the branches may become more useful as features are added. Merge is used to bring the source code on the master up to date from the branch.

Rebase is used to bring changes in the mater branch into the branch, making it the new line of development. See

https://stackoverflow.com/questions/12528854/how-to-perform-rebase-squash-using-tortoisegit

https://hackernoon.com/git-merge-vs-rebase-whats-the-diff-76413c117333

https://blog.carbonfive.com/2017/08/28/always-squash-and-rebase-your-git-commits/

2.4 Git Terminology

- **branch** - A "branch" is an active line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.

- **checkout** - The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.

- **commit** - As a noun: A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.

  As a verb: The action of storing a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.

- **fetch** - Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too. See also git-fetch[1].

- **head** - A named reference to the commit at the tip of a branch. Heads are stored in a file in $GIT_DIR/refs/heads/ directory, except when using packed refs. (See git-pack-refs[1].)

- **HEAD** - The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it directly references an arbitrary commit.

- **master** - The default development branch. Whenever you create a Git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.

- **merge** - As a verb: To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.

  As a noun: unless it is a fast-forward, a successful merge results in the creation of a new commit representing the result of the merge, and having as parents the tips of the merged branches. This commit is referred to as a "merge commit", or sometimes just a "merge".

- **pull** - Pulling a branch means to fetch it and merge it. See also git-pull[1].

- **push** - Pushing a branch means to get the branch's head ref from a remote repository, find out if it is an ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.

- **rebase** - **Rebase** is one of two **Git** utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge . Merge is always a forward moving change record. Alternatively, **rebase** has powerful history rewriting features.

- **repository** - A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.
- **revision** - Synonym for commit (the noun).
- **squash** -
- **tag** - A ref under refs/tags/ namespace that points to an object of an arbitrary type (typically a tag points to either a tag or a commit object). In contrast to a head, a tag is not updated by the commit command. A Git tag has nothing to do with a Lisp tag (which would be called an object type in Git's context). A tag is most typically used to mark a particular point in the commit ancestry chain.
- **working tree** - The tree of actual checked out files. The working tree normally contains the contents of the HEAD commit's tree, plus any local changes that you have made but not yet committed.

GitHub Tutorial

GitHub offers plans for free, professional, and enterprise accounts. Repositories can be hosted on GitHub.com or with enterprise accounts on privately run hardware behind a corporate firewall. Because of security concerns with enterprise accounts, repositories hosted in an organization are not treated the same. Organization accounts don't behave the same as User logins since you don't login to Organization accounts nor can you do Git operations against the Organization account.

3.1 GitHub Termonology

- **Assignee** - The person assigned to address the issue or feature request.
- **Collaborators** - A *collaborator* is needs access to a repository in a different way then the default. This can be people from outside the organization, or used to assign higher level privileges (e.g. Admin) to an individual.
- **Issue** - An *Issue* can be a bug report, feature request, or simply point of discussion and/or clarification in the development cycle. It allows for the capture of information and workflow associated with it, allows for assignment, progress tracking, and helps provide statistics on progress in the development process.
- **Mention** - A person or team can be *@mentioned* using their User account login name to be automatically notified that their involvement in an issue is requested.
- **Milestone** - A *milestone* is a way to provide a date goal for resolving a collection of issues.
- **Release** - A *release* can be used to package software, notes, and binary files for other people to use.
- **Repository** - the main organizational level associated with GitHub. Repositories are typically a collection of file associated together, but can also just be a landing place for issues and projects.
- **Organization** - enterprise level GitHub membership/license
- **People** - individual users who are members of the organization
- **Permission level** - Controls the level of access to repositories and associated resources. Each user in the organization has default Read. Permission levels are: *Read*, *Triage*, *Write*, *Maintain*, and *Admin*.
- **Project** - A way to organize and track workflows at the Organization, Team, or Repository level.
- **Pull request** - A *pull request* let you tell others about changes you've pushed to a branch in a repository on GitHub. It is used to discuss and review potential changes with collaborators along with follow-up commits before the changes are merged back into the master/base branch.
- **Tag** - A *tag* is a label for a given commit in a repository. It can provide a more descriptive label than the default commit id. It is also the basis for a GitHub release.
- **Team** - groups of people belong together. The Rayonier organization is starting with the Biometrics, LIS, and GIS teams. People can be members to multiple teams.
- **User** - individual accounts that can be licensed directly or joined to organizations

3.2 GitHub Desktop

Fortunately, GitHub Desktop is our friend and can simplify the workflow of managing and synchronizing repositories. Download a copy of GitHub Desktop from https://desktop.github.com/. After download run GitHubDesktopSetup.exe to install the software locally.

Warming: Never *git add*, *commit*, or *push* sensitive information to a remote repository. Sensitive information can include but is not limited to: Passwords, SSH keys, AWS access keys, credit card numbers, PIN numbers, etc.

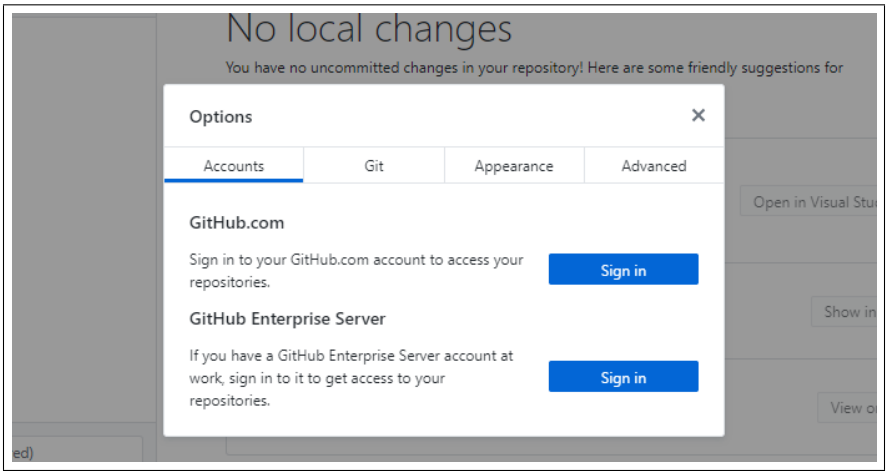


Figure 3.1: GitHub Desktop SignIn Screen

Login using your previously create credentials for GitHub by clicking the Sign in button next to GitHub.com (Figure 3.1). Once you login you should see a list of repositories in your associated organizations and any personal/user repositories you have created (Figure 3.2). In addition you see three options to "Clone a repository from the Internet...", "Create a New Repository on your hard drive", and "Add an Existing Repository from your hard drive...".

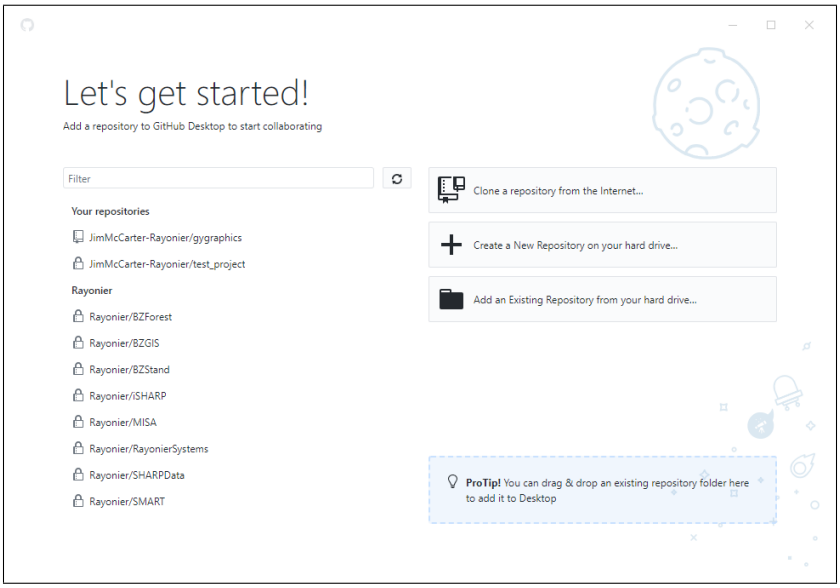


Figure 3.2: GitHub Desktop Getting Started

3.3 GitHub/Local Repository Synchronization

Git operates on local repositories. GitHub hosts shareable repositories. This requires that the local and hosted copies be synchronized to periodically. The processes around this activity include:

- making a local copy of a hosted repository (clone),
- moving a local copy to hosted (Publish),
- updating a local repository with hosted changes (pull),
- and sharing local changes back to a hosted copy (push).

3.3.1 Clone Repository from GitHub

You can clone an existing repository from GitHub by simply selecting the repository from the list and providing a local directory. In GitHub Desktop, if you already have local repositories you can access the repository list by using **File/Clone repository...**.

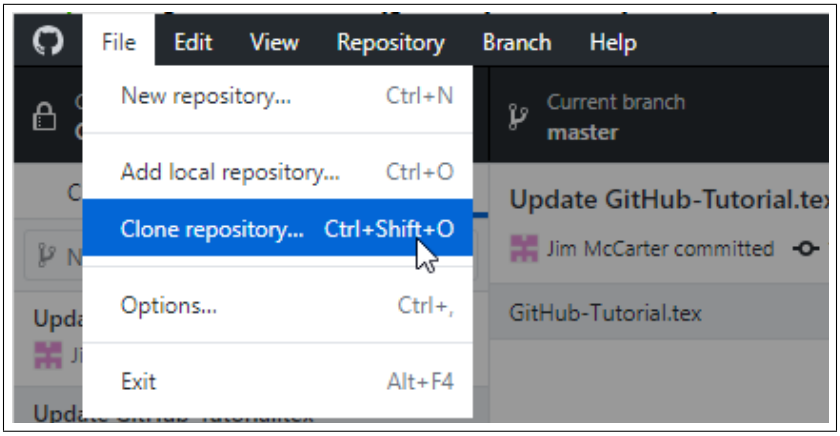


Figure 3.3: GitHub Desktop Clone Repo Menu

The example below we have selected the BZStand repository from the Rayonier organization and are making a working copy in the default local directory, in my case my local G:/My Drive/Documents/GitHub/BZStand folder.

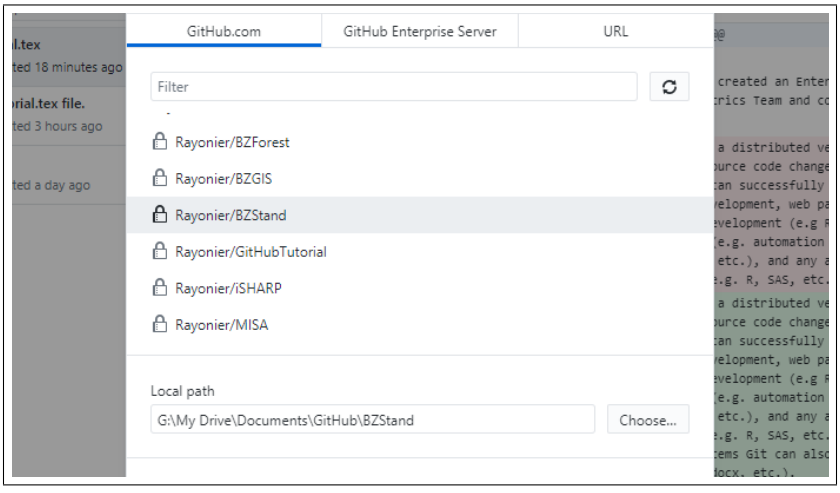


Figure 3.4: GitHub Desktop Clone From GitHub

3.3.2 Publish Local Repository to GitHub

Publishing a repository to GitHub assumes the repository does not already exist on your personal/user account or in the organization. This process will create the repository where you designate it. Figure 3.5 shows the GitHub Tutorial repository ready to publish. We have some of the files added and locally committed. Click the Publish Repository button with the cloud image.

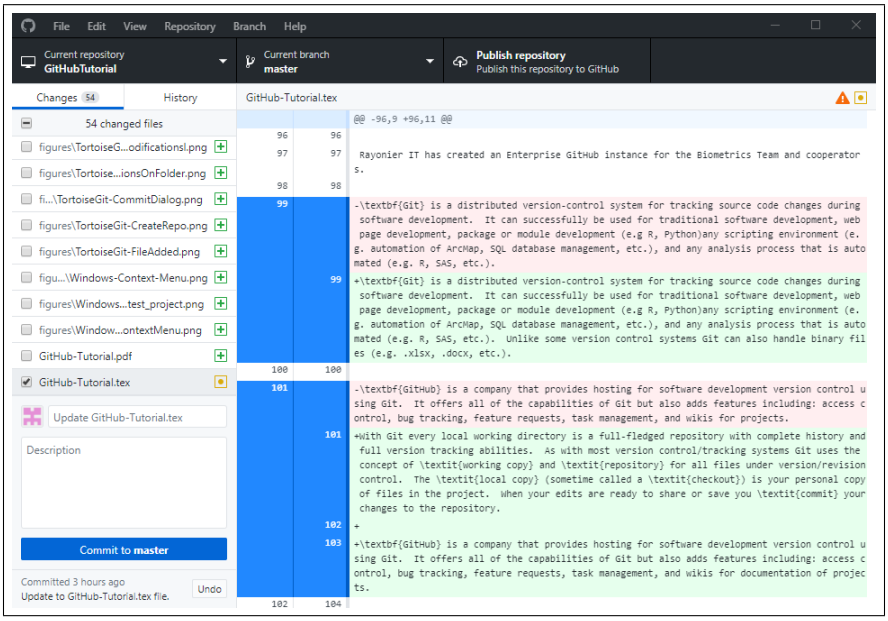


Figure 3.5: GitHub Desktop Ready to Publish GitHub Tutorial

From the Publish Repository dialog make sure you are on the GitHub.com tab, confirm the "Keep this code private" is checked, and select the Rayonier Organization. Click the Publish Repository to create the repository under the Rayonier organization online.

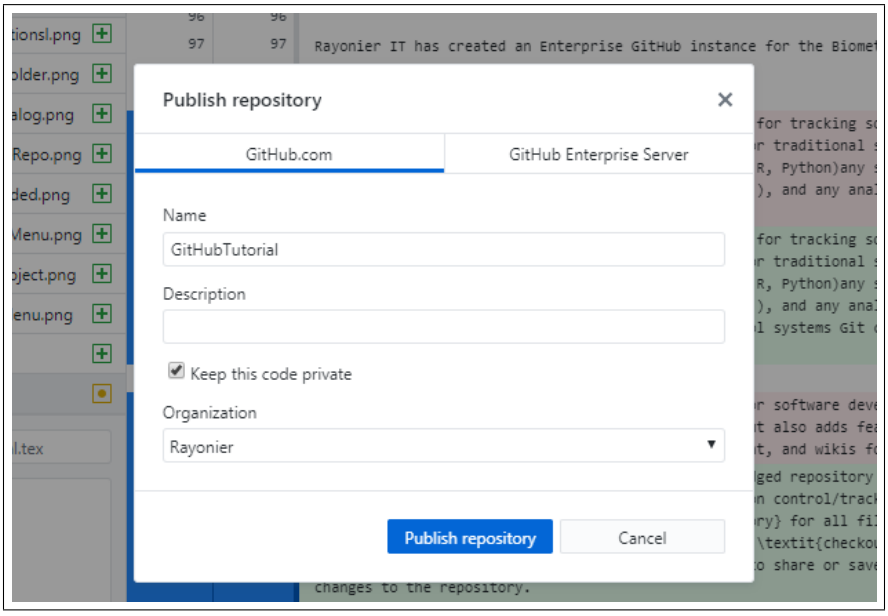


Figure 3.6: GitHub Desktop Publish GitHub Tutorial

3.3.3 Pull Repository from GitHub

If changes have been made to a GitHub repository that you want to have available in your local working copy you can pull the repository from GitHub to your local working directory.

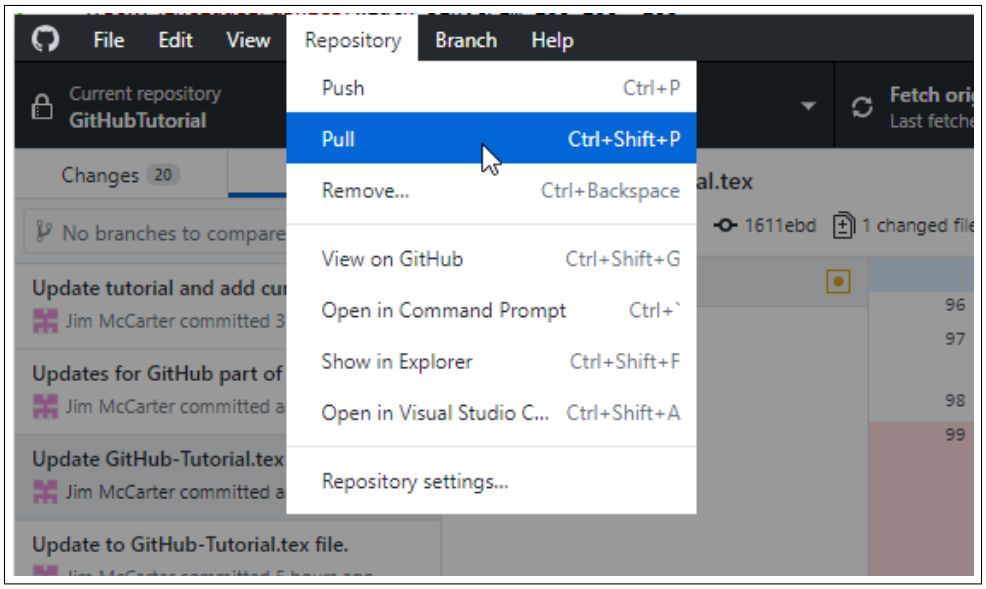


Figure 3.7: GitHub Desktop Pull Repository from GitHub

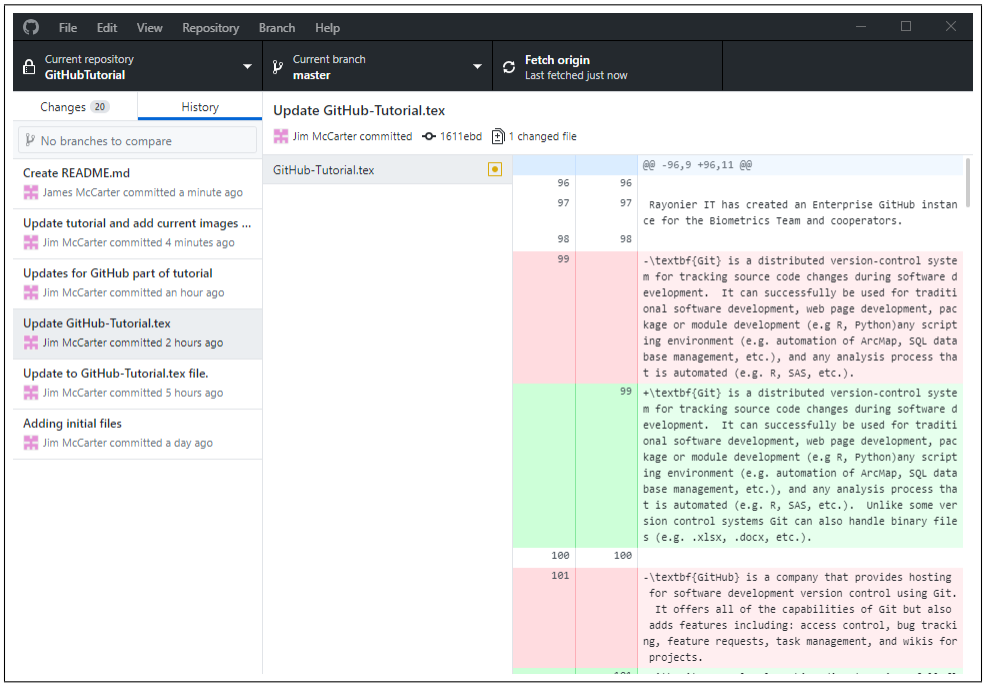


Figure 3.8: GitHub Desktop Repository After Pull showing Readme.md downloaded

3.3.4 Push Repository to GitHub

After a number of modification to the tutorial we have added some screen captures and updated the .tex and .pdf file. We can use TortoiseGit to check for modifications (Figure 3.9).

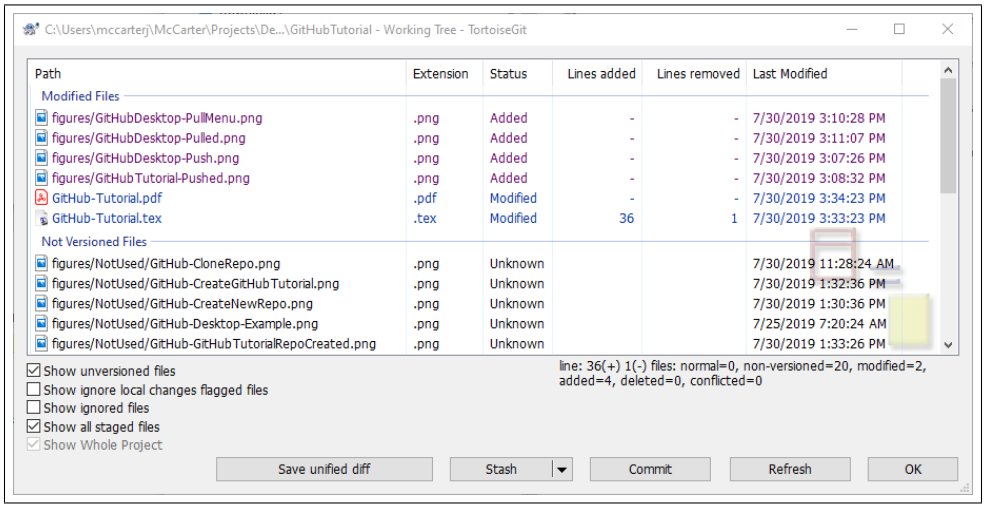


Figure 3.9: TortoiseGit Check for Modifications

We see we have added four screen capture images, updated the .tex file and have created a new .pdf copy of the tutorial. We can commit these files and then Push the changes to GitHub using the Push button.

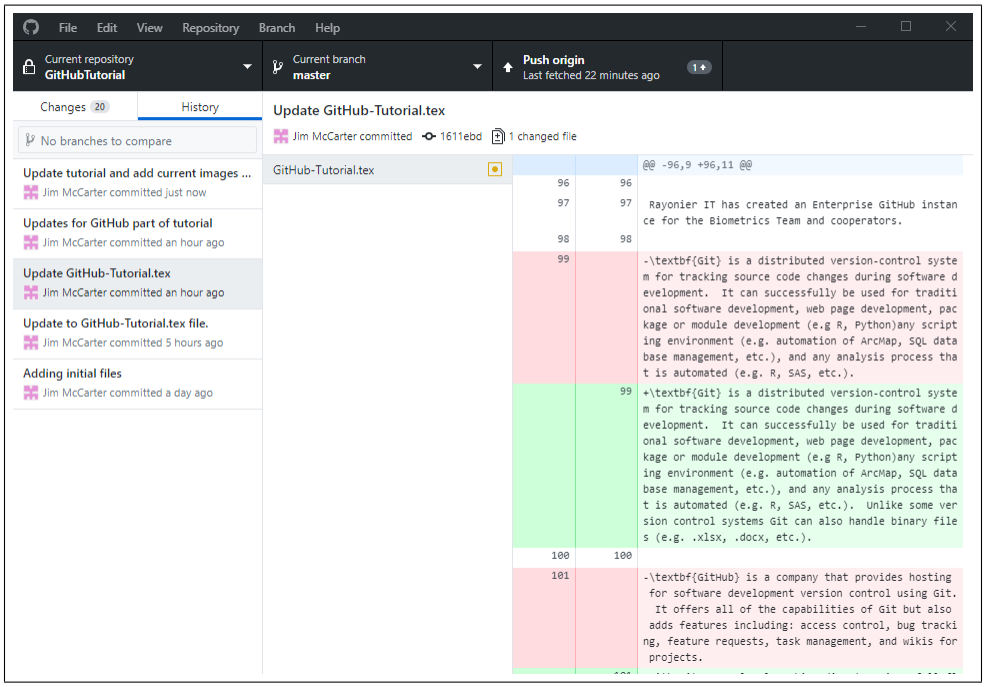


Figure 3.10: GitHub Desktop Push Repository

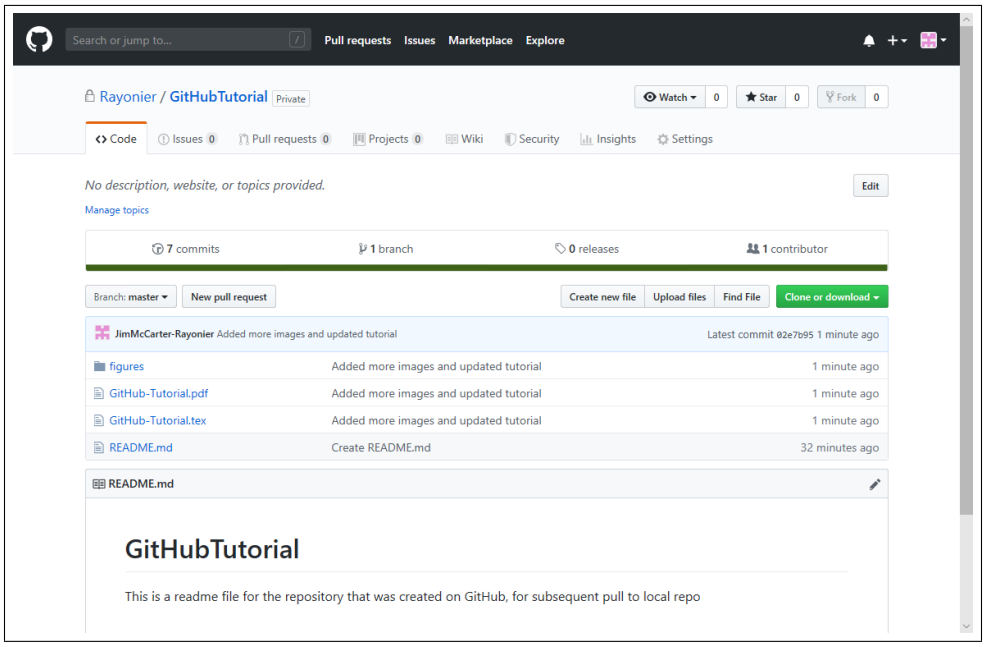


Figure 3.11: GitHub Tutorial Pushed

3.3.5 Staying in Sync Between GitHub and Local Repositories

It is relatively easy to stay in sync between GitHub and local repositories using GitHub Desktop.

If Fetch Origin says it was last fetched days ago, then you check against GitHub by clicking on "Fetch origin". This should update the status for the repository and you will see if there are any changes that need to be pulled. Then you can use "Pull Origin" to update the local repository.

3.3.6 Moving Local Repositories

You can easily move local repositories since all information about the repository is stored in the .git folder (normall hidden). After moving, GitHub Desktop will notice the file is no longer where it used to be, and prompt you to Locate, Clone Again, or Remove the local repository.

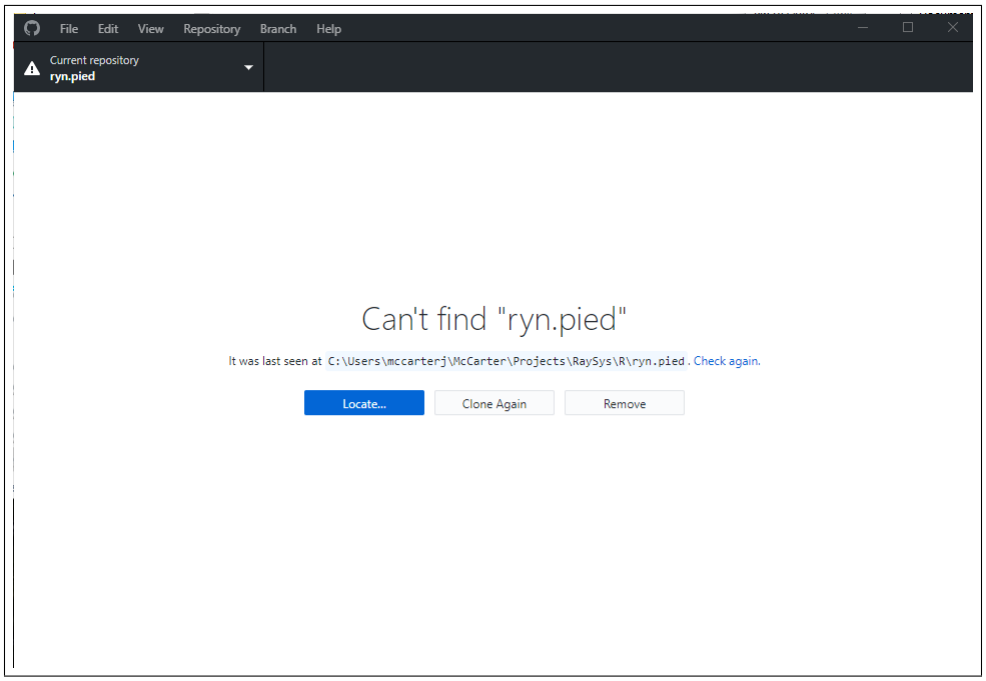


Figure 3.12: GitHub Desktop - Locate Moved Repository

3.3.7 Using GitHub R package repositories

- Confirm you have build tools. If not:
  - Download and install Rtools
  - From RStuido run install.packages('devtools')
- Download or clone repository
- Create R Studio project in repository folder
- Build Binary or Source package
- detach("package:name", unload=TRUE)
- Install package in R/RStudio

can also install R packages directly from GitHub using devtools

or from .zip file stored on github if included in the repo

Settings/Developer settings/Personal access tokens/Generate new token

3.3.8 R Package Development and GitHub

3.4 Migrate Repository from AWS CodeCommit to GitHub

A number of Git repositories had already been shared online using AWS CodeCommit. When the change was made to use GitHub, several workflows were investigated to migrate repositories from CodeCommit to GitHub. All workflows tested to migrate from CodeCommit to GitHub through local files didn't work without major local repository hacking. This is because Organizational repositories are different beasts than personal/User repositories. But there is a simple solution, GitHub can clone directly from AWS CodeCommit. Note: All AWS hosted repositories for the Biometrics team have been migrated. This workflow is documented here in case other repositories need to be migrated in the future:

1. Make sure your current local repository is synced with CodeCommit
2. From GitHub Rayonier organization begin by starting to create a new repository

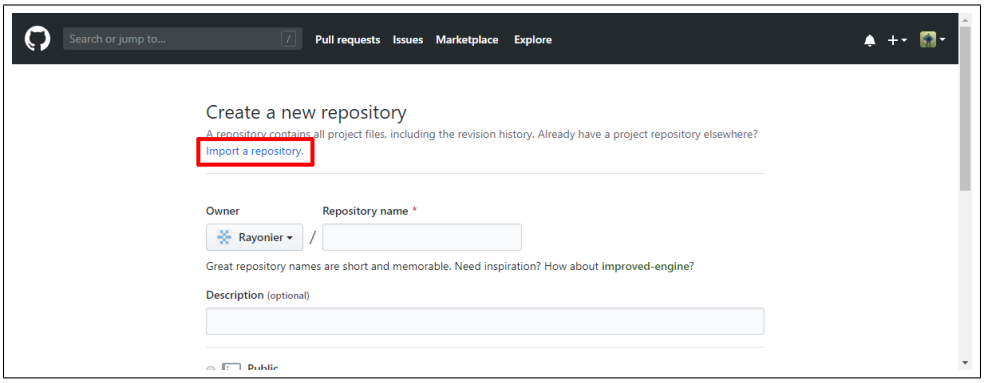


Figure 3.13: GitHub Create a new repository screen.

3. Click on import a repository to begin migration from AWS CodeCommit. Now we need to fill in the URL and provide credentials.

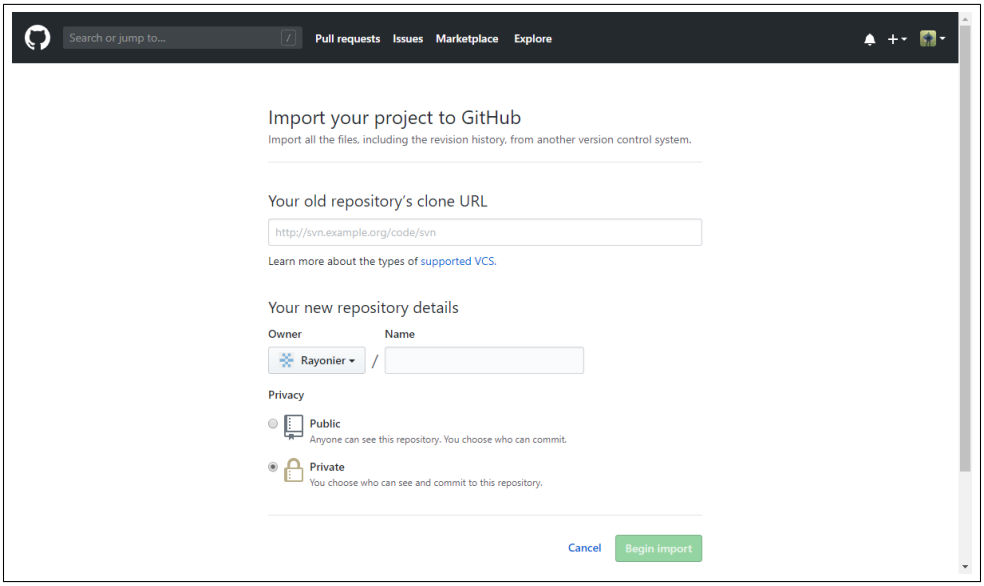


Figure 3.14: GitHub import project to GitHub screen.

4. Login to AWS CodeCommit and navigate to the desired repository. Click the **Clone URL** button and select **Clone HTTPS**.

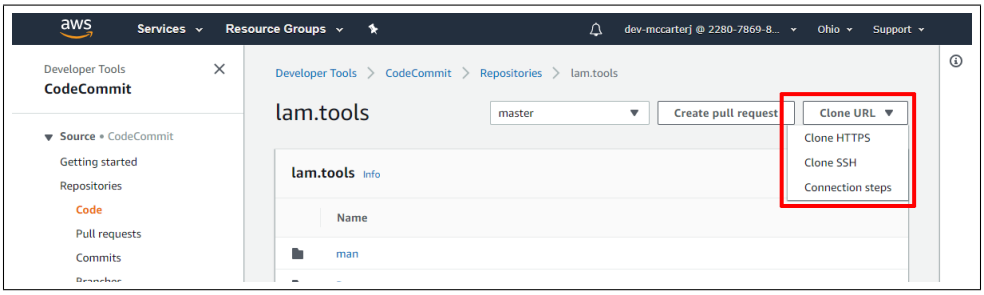


Figure 3.15: AWS Code Commit Clone URL

5. After clicking **Clone HTTPS** the URL should be copied to the clipboard, or highlight and copy from the web page.

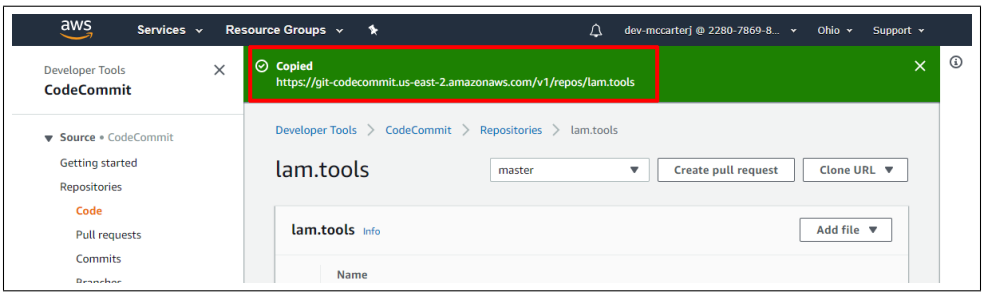


Figure 3.16: AWS Code Commit Clone URL

6. Paste the URL for the AWS CodeCommit project in the "Your old repository's clone URL", confirm we are migrating to the Rayonier organization or your personal account, provide the repository name (same as previous), then click **Begin import** button.

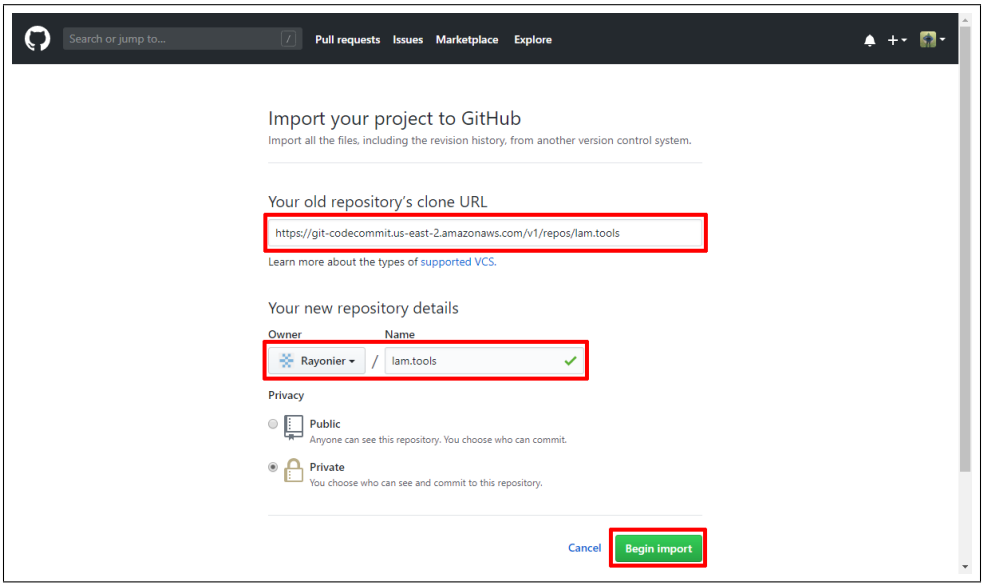


Figure 3.17: GitHub Import Repository from AWS Code Commit

7. On the Preparing your new repository click on the repository name.

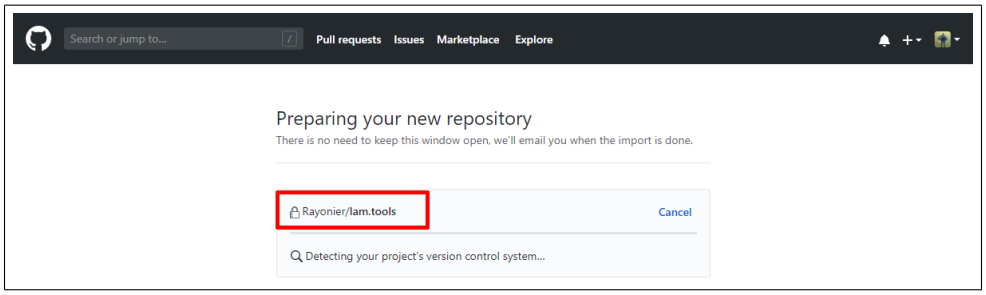


Figure 3.18: GitHub Import Click on Repo to provide credentials

8. Provide your AWS Code Commit credentials and click "Submit":

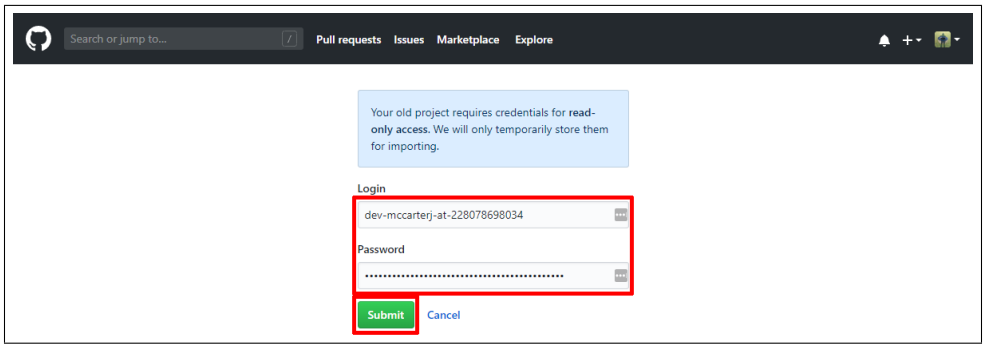


Figure 3.19: GitHub Import - Provide AWS credentials

9. Depending on the size of the project it may take a few minutes to migrate

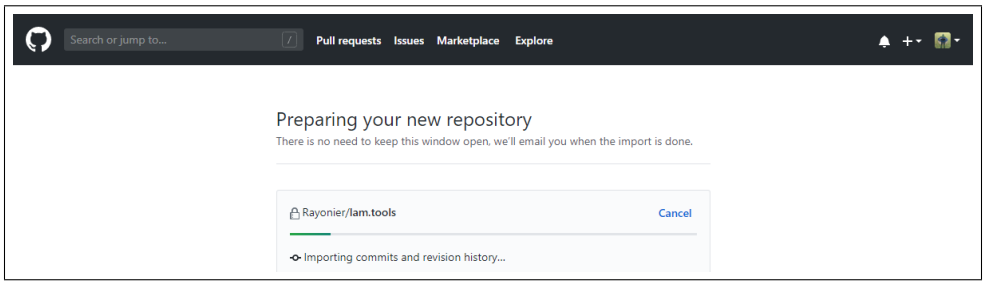


Figure 3.20: GitHub Import - project importing

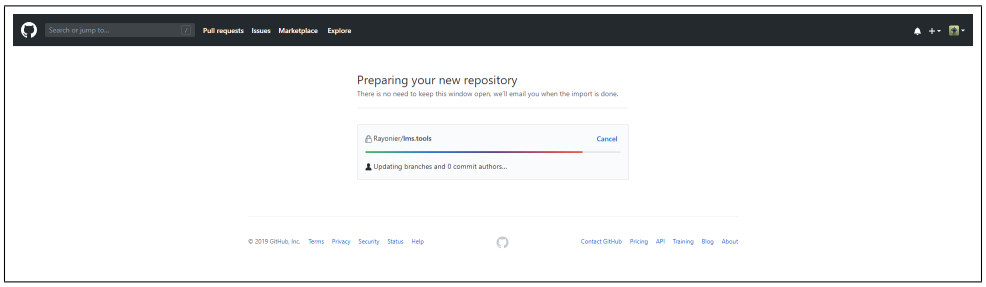


Figure 3.21: GitHub Import - updating branches

The repository is now migrated to GitHub with all history and revisions in tact.

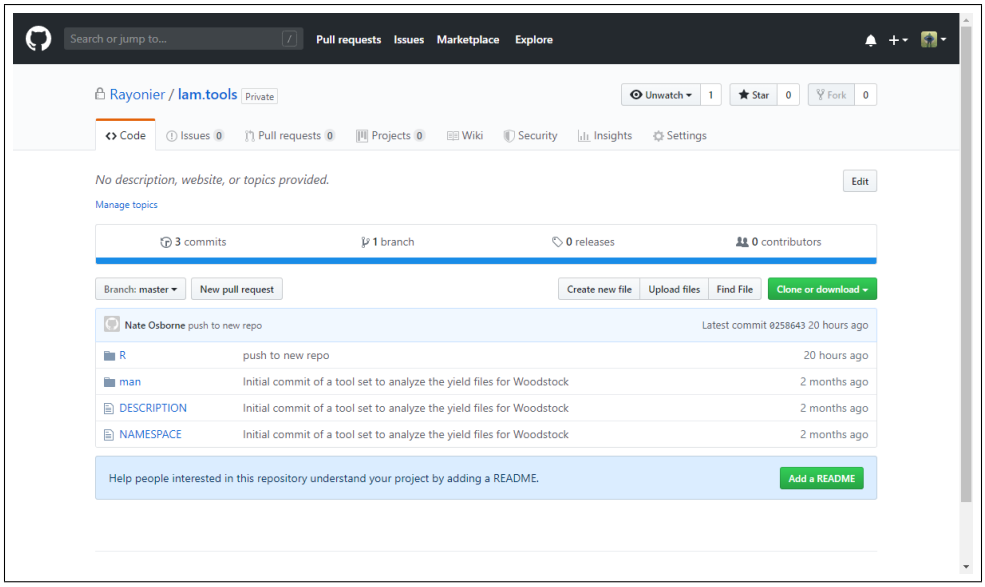


Figure 3.22: GitHub Import - Imported gygraphics repository

3.4.1 Reset Local Repository to GitHub from AWS CodeCommit

- If this is a repository you downloaded and have not made any changes to
    1. You can simply delete the local directory and use GitHub Desktop to close a fresh copy from GitHub
- If this is a repository you developed and have other files associated with, but not included in the repository
    1. Be paranoid about losing files, so make a complete backup of the repository folder
    2. Rename your project directory (e.g. lam.tools renamed to save.lam.tools)
    3. Clone the GitHub repository using GitHub Desktop software, selecting your working folder where the repository is going to live.
    4. Copy any other project related files from your renamed folder into the new local repository.

3.5 Additional GitHub Concepts

This section includes some additional concepts associated with GitHub that users may find useful as they get used to the platform.

3.5.1 Please verify your device

When you access GitHub from a new computer or device you should receive and email to your Rayonier email account asking you to confirm access. This message should look similar to:

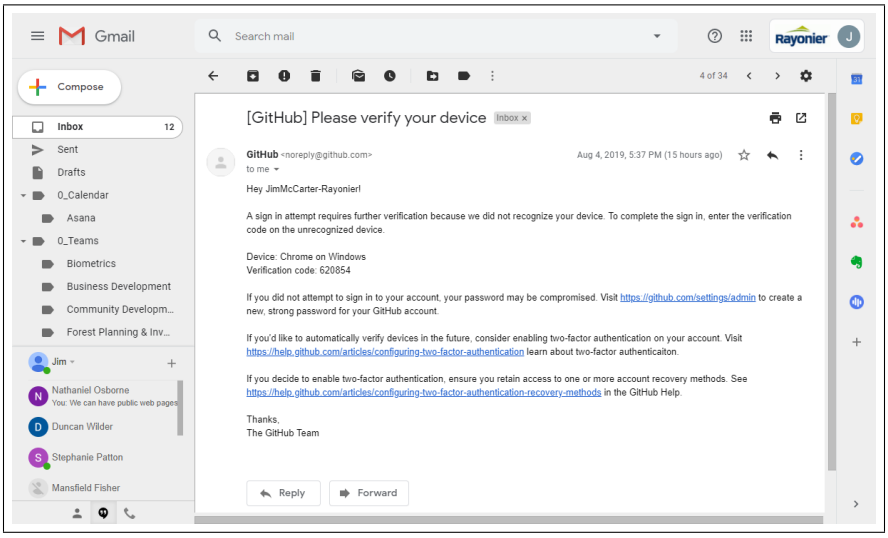


Figure 3.23: GitHub Verify Device Email

This email provides a Verification code that you should enter on that device to authenticate for access to GitHub.com. There is also advice on creating a new strong password on your account if you are not trying to access GitHub from a new device along with links to read about enabling two-factor authentication for your GitHub account.

3.5.2 Setting up Repository for Access

When creating a new GitHub repository in the organization is all members are giving *Read* access. This allows users to read the files and information, clone a local copy of the repository, allows them to create issues and pull requests, but does not allow them to manage issue and pull requests or push updates to the repository.

Table 3.1: GitHub Repository Permission Levels

Level	Description
Read	Read, clone, and open/comment on issues/pull requests
Triage (beta)	Read level, and can manage issues/pull requests
Write	Read, clone, push, and manage issues/pull requests
Maintain (beta)	Write level, and update some repository settings
Admin	Read, clone, push, manage issues/pull requests, update repository settings, and manage collaborators

An easy way to provide additional access is to use the team concept in GitHub. Go to your repository Settings and the Collaboratros & teams tab. Add each team that needs upgraded permissions and select the level desired. In

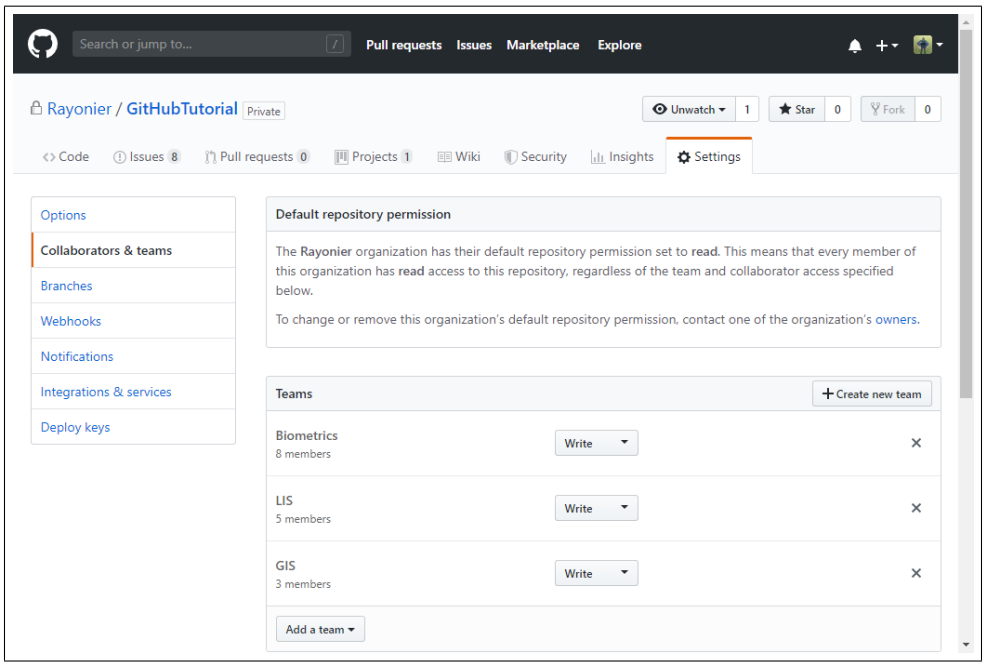


Figure 3.24: GitHub Respository Collaborators

3.5.3 Teams, Members, and Collaborators

The Rayonier GitHub Enterprise account establishes the Rayonier organization on GitHub. The organization hosts Repositories, Packages, People, Teams, and Projects. **Repositories** were originally designed for hosting source code, but their use in the context of GitHub has been expanded and can also be used for issue tracking and project management without source code. **Packages** allow releases of source code projects to be released in a number of ways (not currently applicable for our use). **People** are the individual users of the platform who can be organized into **Teams**. **Projects** help organize and prioritize work. **Collaborators** are members assigned to specific repositories or projects.

The Rayonier organization is starting with the *Biometrics*, *LIS*, and *Spatial* teams at the organization level and *Monitoring* as a sub-team under Biometrics. Teams allow users to be grouped together and control default access to repositories that are associated with the team.

The default is **Read** access for all team members, with those members that are collaborators (co-developers) granted **Write** access as a collaborator. The creator of a repository has **Admin** access.

Teams that have parent teams inherit repository access and from their parent teams. Access can be different at the Team and Sub-Team level granting the individual member the highest access. For example, a repository that is associated with the Monitoring team will have Read access by default from the Biometrics team, but may have Write access because the Monitoring team grants them Write access.

3.5.4 Watch, Star, Fork

Use **Watch** to keep track of projects/repositories of interest to be notified of Pull Requests and Commits to the repository. Use **Not watching** to be notified if you are participating or @mentioned.

Use **Star** if you want to bookmark the project and make it show up on your "home page".

3.5.5 GitHub Wiki

A *Wiki* is a website where users collaborate to modify content and structure directly from a web interface. GitHub Wiki's support a simplified markup language. See the RayonierSystems Wiki MarkdownDemo for examples.

Depending on the level of the Wiki, this would be a convenient place to post Team norms and other documents associated with our strategic plan.

3.5.6 Cloning Wiki

There are several reasons you might want to clone your GitHub Wiki pages: 1) to have a backup just in case, 2) to be able to edit offline, 3) to add images to your wiki to jazz up the pages.

To clone a repository wiki pages just get the repository URL and add ".wiki" to the URL.

Example:

git clone https://github.com/Username/RepoName.wiki.git

Once cloned you can create an "images" folder, add PNG or GIF files there, then reference those from your Wiki pages using [[images/ImageName.PNG]]

3.5.7 GitHub Release

You can "package" software into a release, along with release notes, links to binary file, for use by other people. Releases are based on Git tags, which mark a specific point in your repository history. Releases are ordered by the date they were created on GitHub.

3.5.8 Wiki Versus GitHub Pages

GitHub repositories can have Wiki and or GitHub Pages documents associated with them. Wiki pages are internal documentation at the repository level. GitHub pages are websites that are public facing but have the content version control managed in association with the repository.

GitHub repositories can have their own web pages hosted directly in the repository and hosted on GitHub. See https://pages.github.com for more information until this tutorial is completed.

3.6 GitHub Project Management

GitHub has an impressive suite of tools for managing issues associated with repositories and the workflow with tracking progress.

3.6.1 Issues and Pull Requests

Issues are for when we have a question, see something that might be wrong, but we don't know how to address the issue or need additional interaction to figure out how to fix it. These *issues* can be bugs, enhancements, or project requirements. In GitHub they are define loosely so they can track the workload associated with the item. **Pull Requests** are when there is a problem and we know how to fix it or have already fixed the source code.

3.6.2 Managing Issues

Issues can be entered a number of ways, depending on how the repository is configured. In all cases a number of options are available when filling out the New Issue dialog.

Creating Issues

Give a short descriptive title for the issue and optionally a detailed description in the comment section. Decide if you should designate an Assignee, use any defined Labels, assign to an existing Project, and if the issue belongs to an existing Milestone. The Comment section for all issues support Markdown – you can format your comments using highlighting, font sizes, provide lists, task lists with checkboxes, and attached images/screen captures as needed. You can add images using drag and drop. You can also notify others using @mentions with their GitHub usernames.

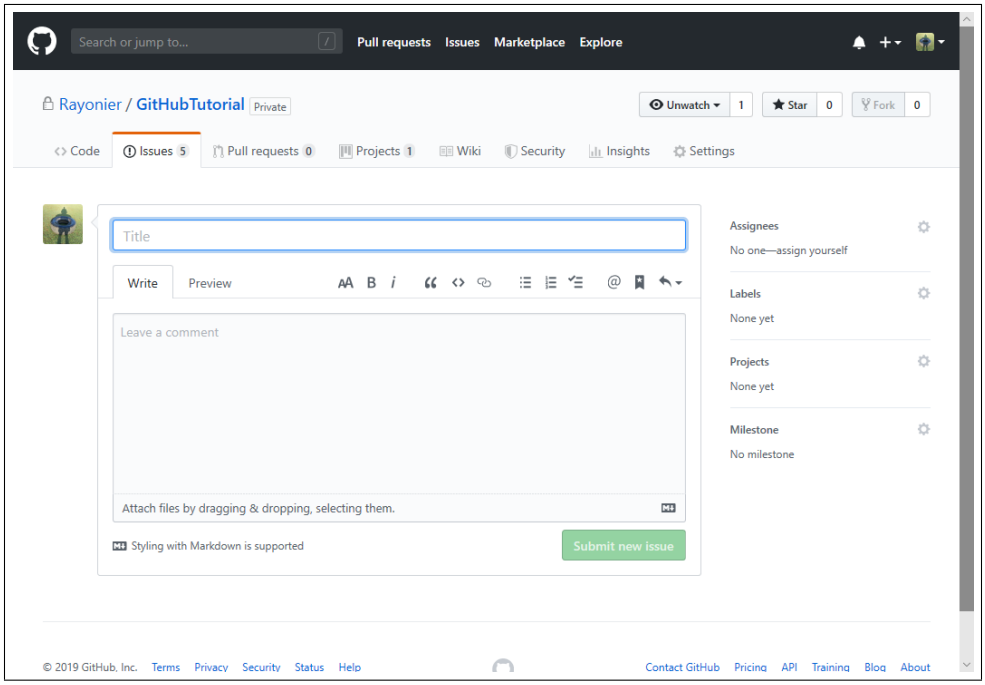


Figure 3.25: Entering New Issue

When a repository is setup with Issue templates instead of going strait to the Open regular issue dialog you are presented with a page to select your desired template. The example below shows the default templates for a Bug report and Feature request. Custom templates can be developed, prompting the use to enter required information to help describe issues.

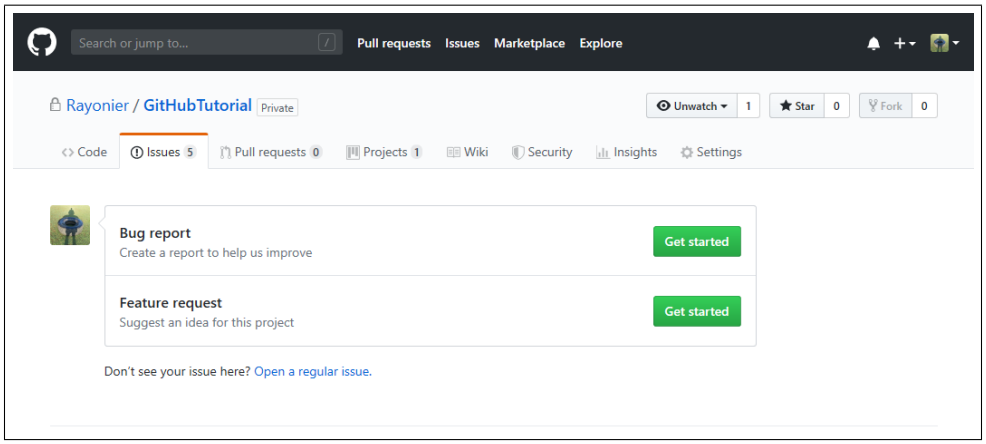


Figure 3.26: Entering New Issue with Issue Templates

The Feature request template prompts the user provide additional information about the request and how it might relate to other existing issues. It also asks for a description of what solution is desired, what alternatives have been considered, and additional context.

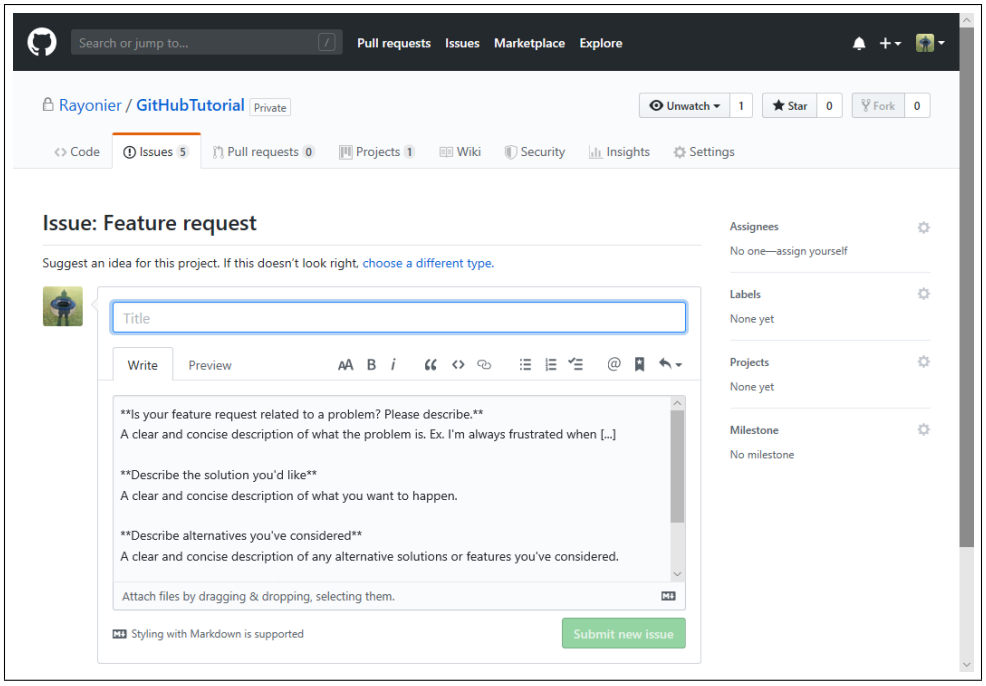


Figure 3.27: Entering New Issue using Feature Template

The Bug report template prompts the user to provide information that is useful in locating and fixing issues. It asks for a concise description, steps to produce, what the expected behavior was supposed to be, screen shots, what computing environment (OS, Browser, version, etc.), if Mobile (Device, OS, Browser, version, etc.), and any additional context that might be helpful.

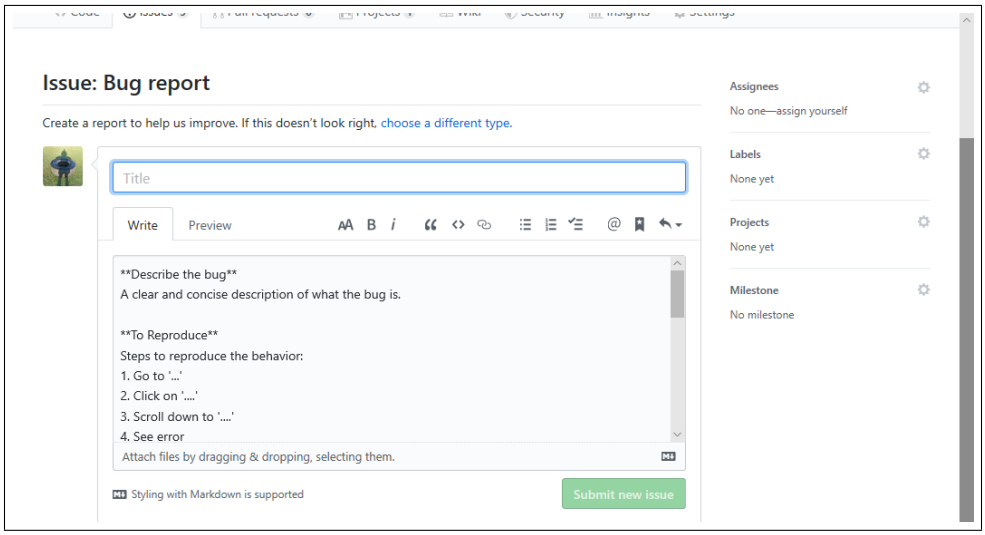


Figure 3.28: Entering New Issue using Bug report template

You can use labels to tag issues with a category or context. These can be used to track progress and filter by label later. Depending on how labels are defined (e.g. see Figure 3.30), you can track priority, status, etc. You can also manage issues in bulk by selecting multiple issues and then selecting the label to assign. You can also assign a series of issues to a different user by selecting multiple issues and then selecting the appropriate user.

Issues can also be sorted by Newest, Oldest, Recent, Comment level, and reactions (emoji).

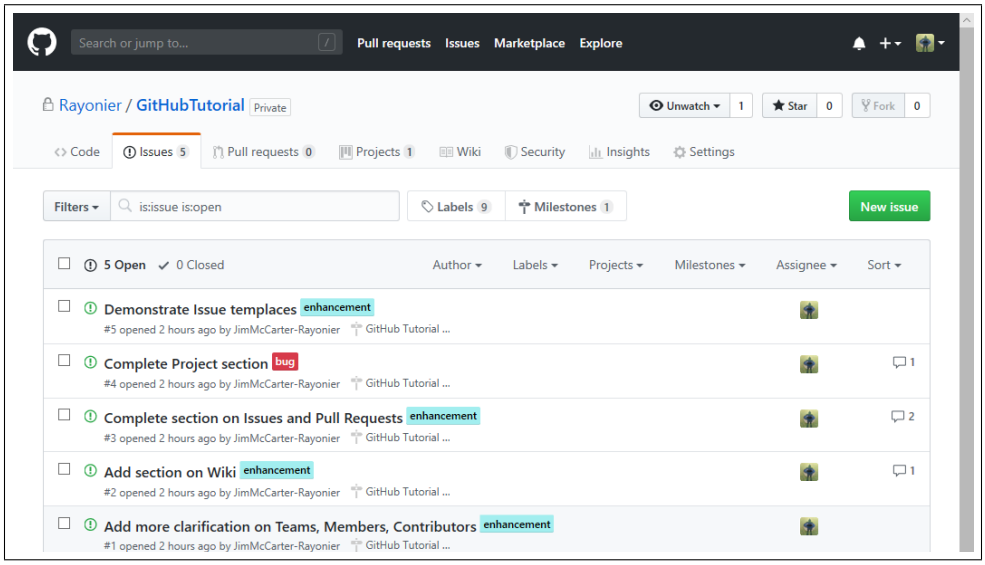


Figure 3.29: Issue List

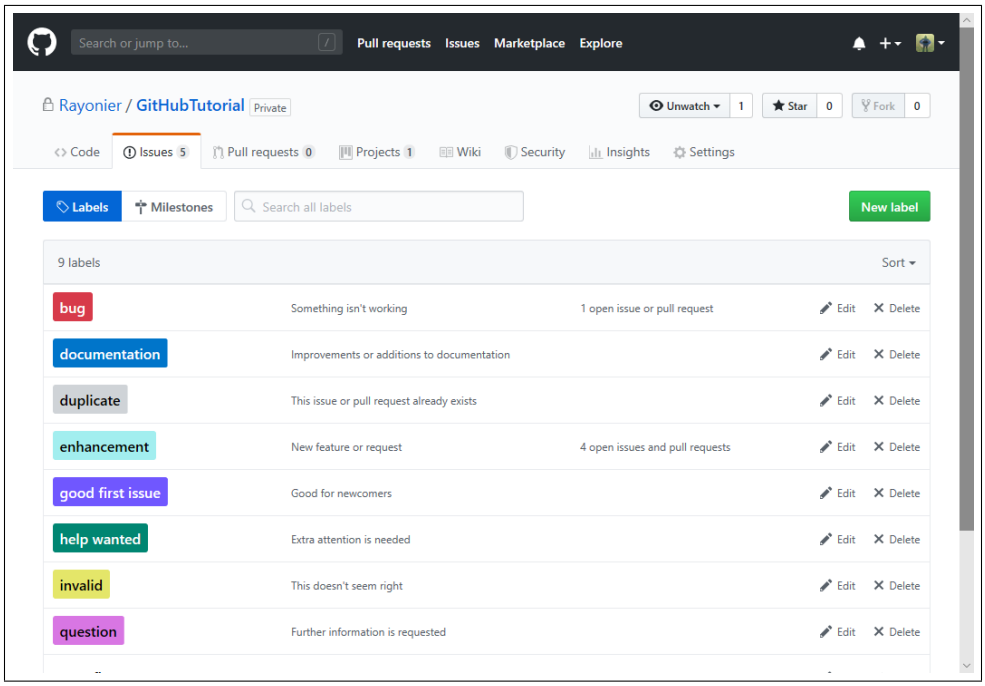


Figure 3.30: Configure Labels

Comment on Issues

Users with interest in a specific repository or who have been assigned or mentioned on an issue can make and edit comments associated with the issue.

As an example we will examine the RayonierSystems repository "Issue #10 - Migrate AWS CodeCommt repositories to GitHub". This issue was created by Jim McCarter, implemented a task list in markdown and then mentioned (@biometrica) Nate Osborne requesting feedback.

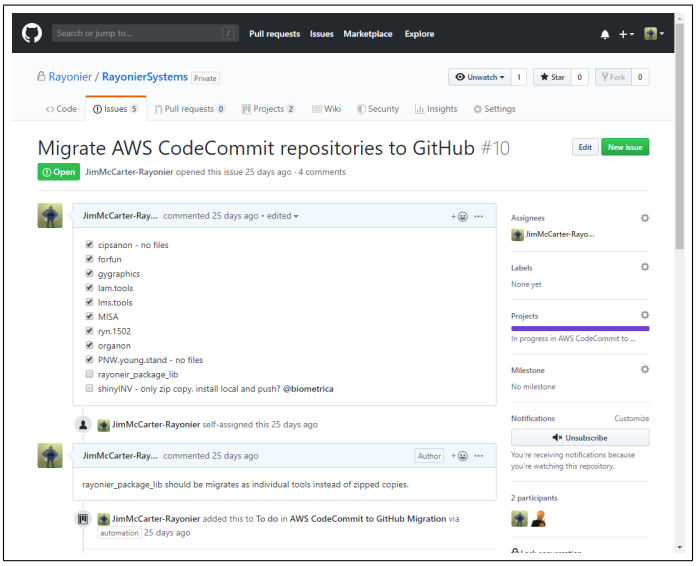


Figure 3.31: GitHub - Issue with task list and @mention

Nate Osborne commented and closed the issue asking for one specific repository to be migrated for use the next day. Jim McCarter performed the migration, but re-opened the issue because several low priority repositories had not been migrated yet. Also notice in the web page that integration with the associated project was automatically updated (**In Progress** to **Done** and then **Done** to **In Progress**) as the status of the issue was changed.

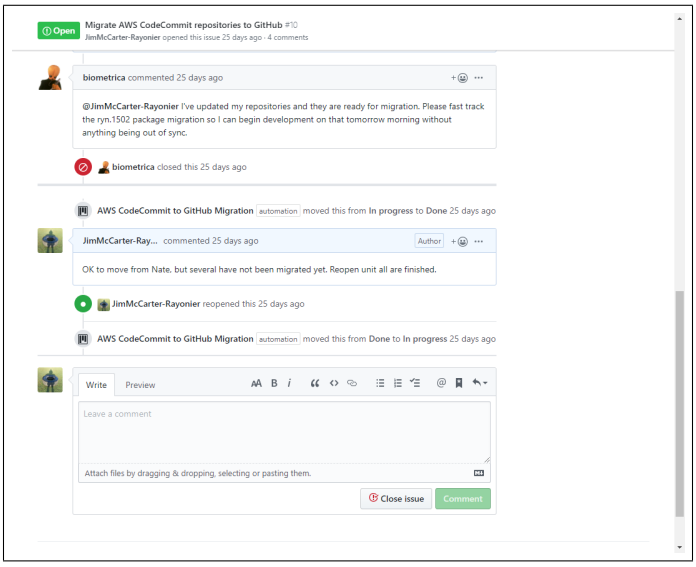


Figure 3.32: GitHub - Issue Comments, Close, Reopen

Closing Issues

As an example of closing issues, the GitHubTutorial repository has "Issue #7 - Add Stashing section to Intermediate Git". This issue was initially created on Aug 6th, was labeled as an "Enhancement", and assigned to "JimMcCarter-Rayonier". The issue was also added to the "Release Tutorial" project, "To do" tab, and added to the "GitHub Tutorial Rollout" milestone on Aug 6th. The issue was updated, but changing the name, Aug 9th. The issue was finally closed Aug 26.

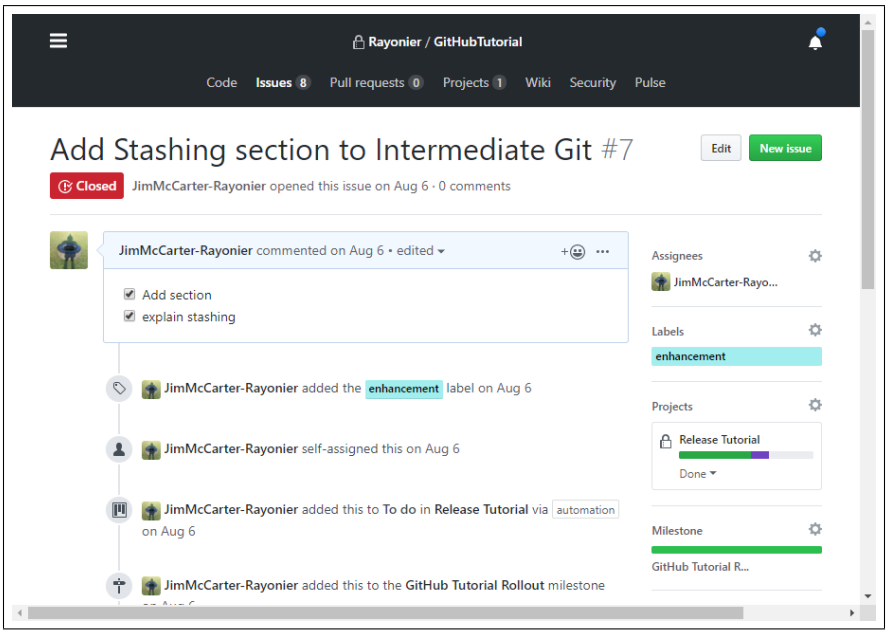


Figure 3.33: GitHub - Issue Workflow showing attributes and progress

Additional comments can be added, or if it's not properly resolved it can be Reopened.

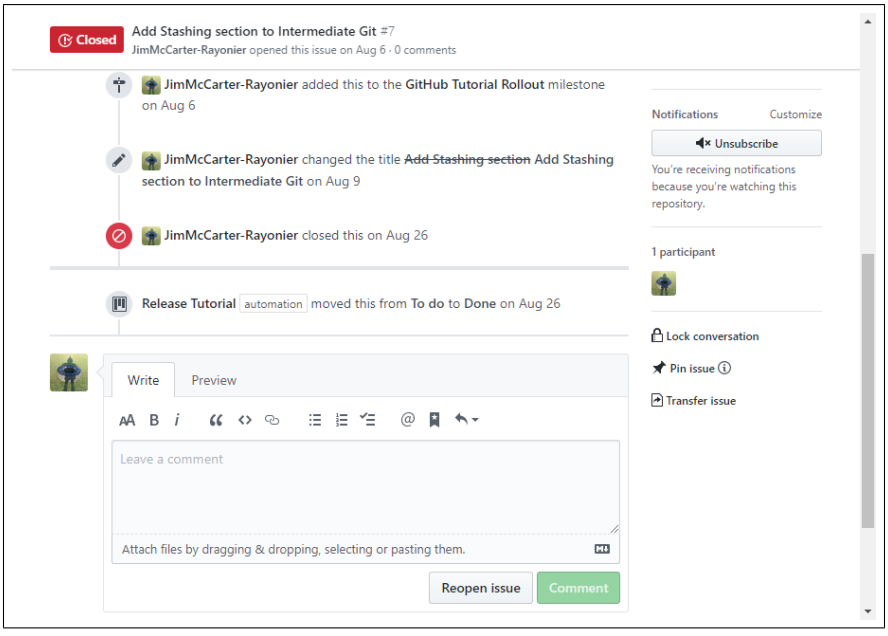


Figure 3.34: GitHub - Issue Closed

3.6.3 Milestones

Milestones let you organize issues into groups that satisfy objective with a particular time frame.

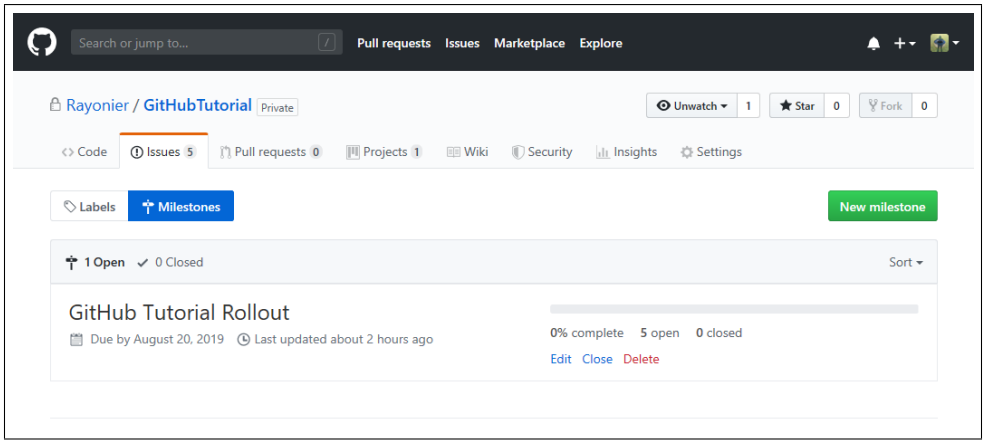


Figure 3.35: Milestones

Issues can be assigned to milestones and progress toward the milestone can be tracked.

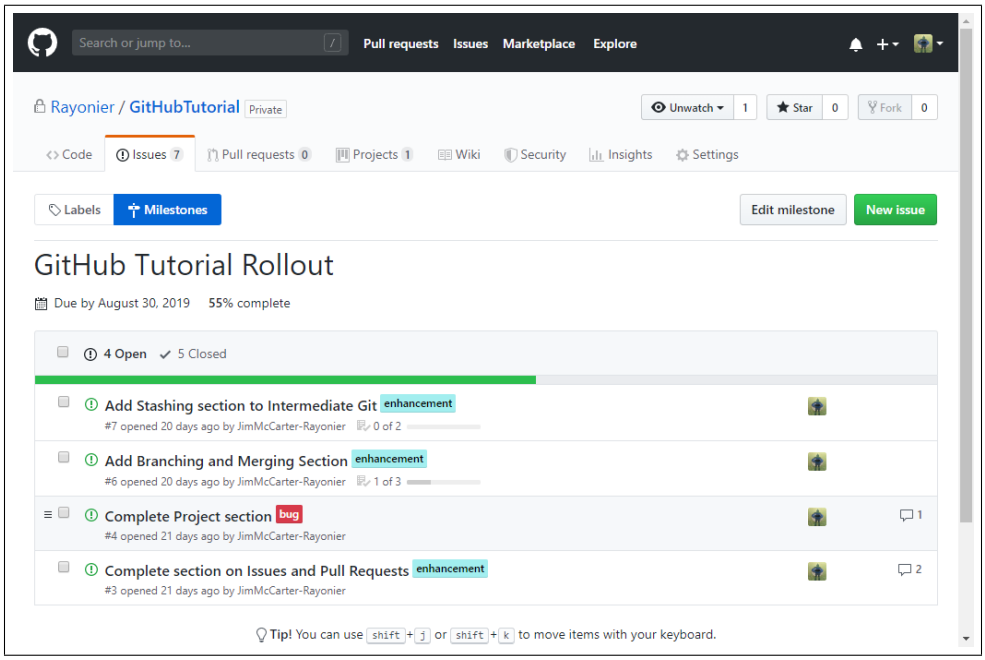


Figure 3.36: Milestones

3.6.4 Projects

Projects can be at the Organization, Team, or Repository level. A repository level project logically only includes things associated with the repository. Team and Organization level projects can include efforts that are not limited to a single repository, therefor are broader in scope.

Additional Resources:

- GitHub Project Management Tutorial - How to use GitHub Projects & Automation

When creating a new project you will be prompted to provide a **Project board name**, optionally provide a **Description**, and optionally provide a **Project template**. The available project templates are shown in Table 3.2.

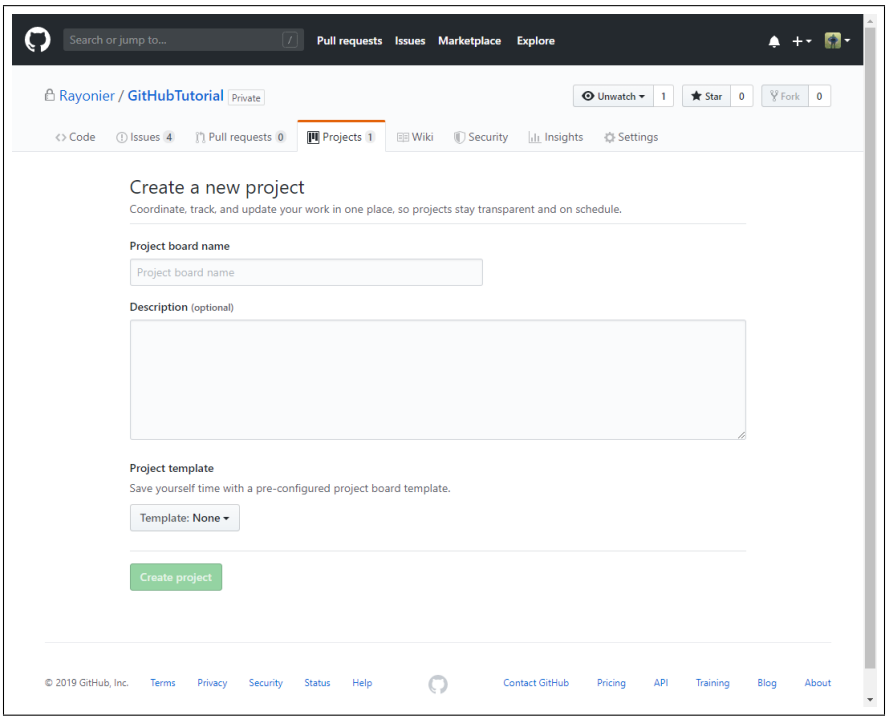


Figure 3.37: GitHub - New Project

Table 3.2: GitHub Project Templates

Template	Description
Basic kanban	Track tasks with To Do, In Progress, and Done columns
Automated kanban	Cards automatically move between To Do, In Progress, and Done columns
Automated kanban with review	Cards automatically move between To Do, In Progress, and Done columns, with additional triggers for pull requests and review status
Bug triage	Triage and prioritize bugs with Do Do, High Priority, Low Priority, and Closed columns

After creating a Project with no Template you can add columns providing names you desire. In the example below (Figure 3.38) we have created columns **Planned** and **In Progress**.

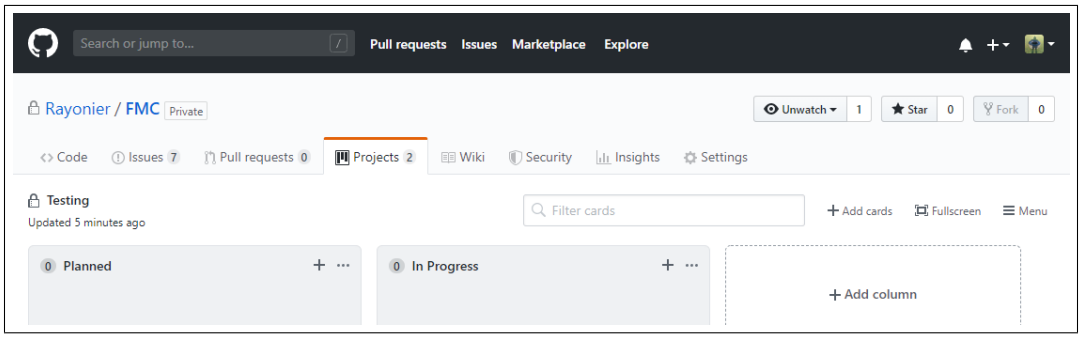


Figure 3.38: GitHub - Project - No Template

We can add **Notes** and **cards** to the project boards. Notes are basically comments, which can be converted to issues, and cards can be selected from the list of issues that exist in the repository.

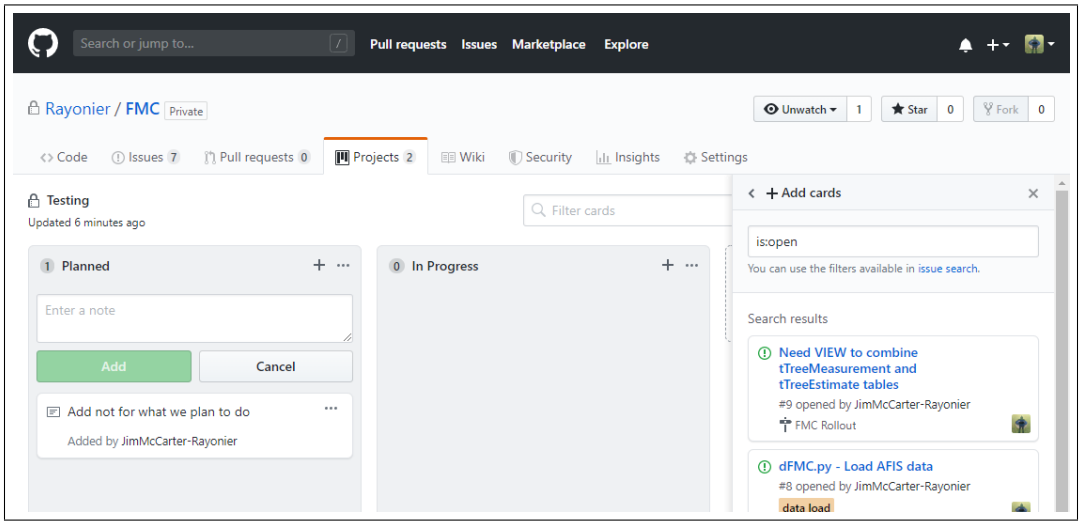


Figure 3.39: GitHub - Project - No Template - Add Notes and Cards

The FMC repository **Forest Measurements Collective Database** project is an example of a project that uses the "kanban" template. It come with pre-created columns of **To do**, **In progress**, and **Done**. Items can be moved between columns simply by dragging them between the columns.

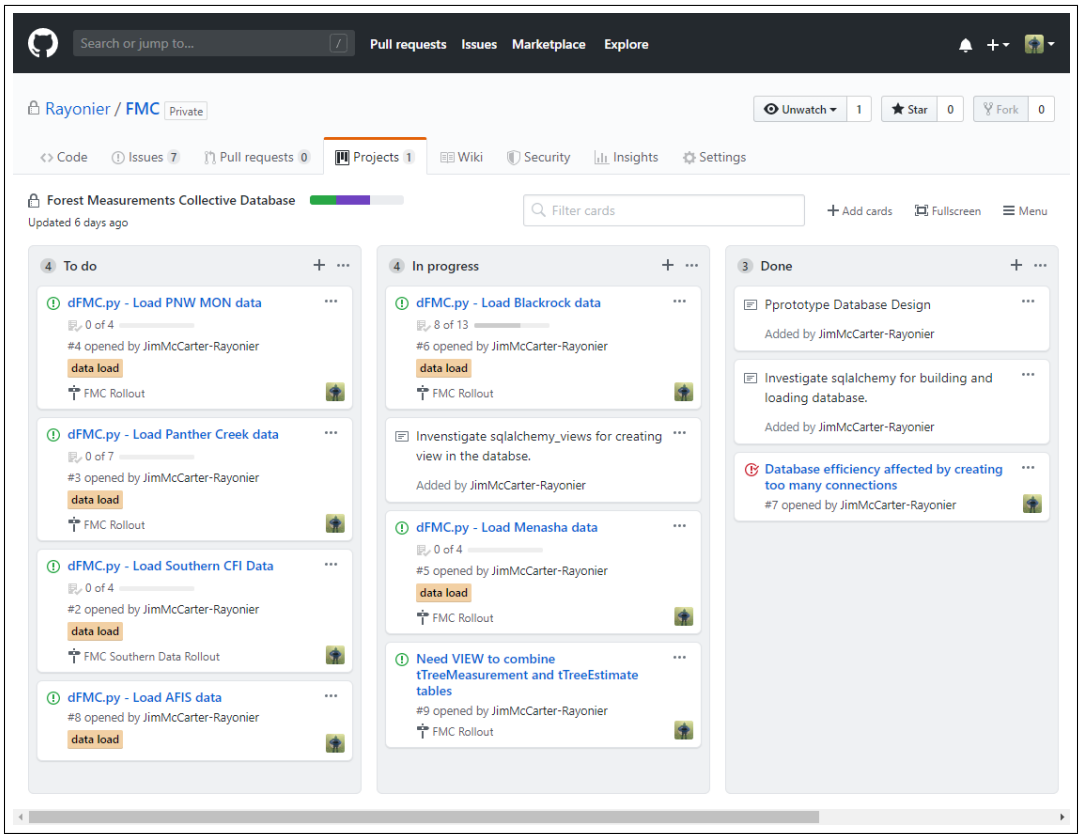


Figure 3.40: GitHub - Project - kanban

The GitHubTutorial **Release Tutorial** project is an example of a "kanban with automation and review" project template. Issues are automatically moved between In progress and Done based on their open/closed status. They can be moved to Review in progress and Reviewer approved manually. Pull requests will automatically be moved between the Review stages using the Reviewer assignments options (examples will be added later).

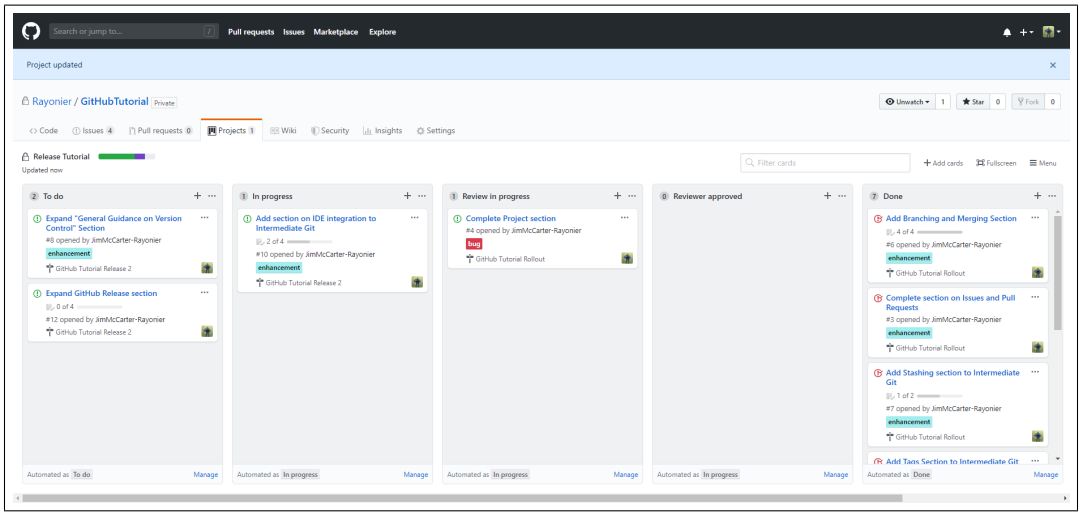


Figure 3.41: GitHub - Project - kanban with review