

This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2019. <http://www.swfox.net>



# Migrating to Git from Mercurial

*Rick Borup  
Information Technology Associates, LLC  
701 Devonshire Dr., Suite 127  
Champaign, IL 61820  
[www.ita-software.com](http://www.ita-software.com)  
[rborup@ita-software.com](mailto:rborup@ita-software.com)*

*While Mercurial remains a perfectly satisfactory choice for distributed version control, it's no secret that Git has become the dominant player. The acquisition of GitHub by Microsoft has accelerated the movement toward Git, particularly if you want to share or contribute to open source projects. What are your options if you are currently using Mercurial? Should you consider migrating to Git? Can you convert your existing repositories or do you have to start over from scratch? Are there tools to help you work with both Git and Mercurial? This session explores the answers to these questions and more.*

### Introduction

Mercurial and Git are two of the most popular distributed version control systems (DVCS) in use today. Conceptually they are very similar, but there are enough differences—some superficial and some substantial—that having a working knowledge of one doesn't automatically enable you to use the other to best effect.

The intended audience for this paper is the developer who is familiar with and may be currently using Mercurial, but who is interested in learning Git with the intention to either use both or to transition to Git entirely. The descriptions and examples are for developers working on Microsoft® Windows. Most of the code is generic but some examples are based on Microsoft Visual FoxPro.

### Notes on the syntax of sample code

In some of the sample code for both Git and Mercurial commands, I have used the Visual FoxPro “&&” token as a familiar way to demark inline comments. Inline comments are not valid in either Git or Mercurial. Unless otherwise noted, anything following && in a line of sample code for Git or Mercurial in this paper should be interpreted as a comment, not as part of the command.

```
C:\>git init    && this is a comment
```

When a folder name is relevant to a code example, the command prompt is written with the appropriate drive/path/folder, as in *C:\myProject>*. If the folder name is not relevant, the command prompt is written merely as *C:\>*. This is done to keep the sample code lines as short as possible; it does not necessarily imply the command should be run from the root of the C: drive.

```
C:\myProject>git log    && this line of code should be run from C:\myProject
```

```
C:\>git log    && folder doesn't matter for this example
```

If a sample line of code is written without any command prompt, the command prompt is implied.

```
git log    && the C:\> command prompt is implied
```

In examples combining command prompt entries and output from those commands, the commands are always be preceded by a command prompt while the output lines are not.

```
C:\myProject>git init  
Initialized empty Git repository in C:/myProject    && output from git init
```

In some cases, blank lines in the actual output are removed to save space.

## Review of DVCS

### How version control systems work

Version control systems track changes to files over time. The fundamental concept of a version control system is the *repository*, the name for the location where the history of file changes over time is stored.

As developers make changes to source code in the course of work on a project, they periodically tell the version control system to save the current state of their work as a new entry in the repository. Over time this series of snapshots becomes the history of changes to the project's source code files. The repository makes it possible for the developer to trace the history of changes to each individual file, to recover earlier versions of a file, to share changes with other developers, and to merge changes made by other developers with their own version of the source code.

### Distributed vs centralized version control systems

A *centralized* version control system has only one single repository for each project. This repository is stored in some centrally accessible location shared by all the developers working on that project. When a developer wants to work on a file, she checks it out of the central repository. In many centralized systems, when a file is checked out by one developer it's marked as locked and no other developer can check out the same file until the first developer checks it back in. This means two or more developers cannot work on the same file at the same time. It also means nobody can get any work done if the central repository is inaccessible, for example due to a network communications failure.

*Distributed* version control systems (DVCS) were created to address some of the problems with centralized systems. In a distributed version control system, each developer has his or her own local copy of the project's repository. For example, if Alice and Bob are both working on the same project, Alice can check out a file from her local repository and make changes to it on her machine while Bob concurrently checks out the same file from his local repository and makes changes to it on his machine. As they work, each of them can commit their own changes to their own local repository without affecting the other. When it comes time to put their work together, the DVCS provides a mechanism to merge the two sets of changes together into a final version of the code.

It's common for teams using a DVCS to push to and pull changes from a central repository to share code with one another and facilitate collaboration. However, even in this scenario the individual local repositories continue to give each developer independence from the others until it's time to share.

Version control systems are sometimes referred to by other names such as source control or revision control. For our purposes, those terms all mean the same thing. The important distinction is that when we're talking about Git and Mercurial, we're talking about *distributed* version control systems.

### A bit about Git internals

Although your normal daily interactions with Git will be via the command line or a GUI, you'll find some knowledge of Git's internals helpful in understanding what Git is doing and why things happen the way they do.

Git stores objects. There are four types of objects: commits, trees, tags, and blobs (binary large objects). Each object is identified by a unique object ID, a SHA-1 hash derived from the contents of the object.

When a file is added or modified, Git creates a snapshot of its current contents and stores it in the repository it as a blob object. Blobs are stored in a compressed format, so although they're complete snapshots they are smaller than the actual file whose content they represent.

Git objects are stored in the `.git/objects` subfolder. This folder is initially empty save for two subfolders named *info* and *pack*, which are also initially empty. Objects are added to the objects folder as development work progresses and files are modified, new file are added, commits are made, branches and tags are created, and so on.

Over time the objects folder may contain thousands or even millions of objects. To add some structure to a folder with potentially that many individual files, Git creates subfolders using the first two characters of the objects' names. For example, an object with ID `021ab20d7b6d70eea17713ff77f1c123da669ca8` is stored in a subfolder named `objects\02` as file name `1ab20d7b6d70eea17713ff77f1c123da669ca8`.

One key difference between Git and Mercurial is that Git uses an intermediate structure called the *index* that sits between the working copy and the repository. You'll encounter the index in several places while working with Git, so it's important to gain some understanding of its role in Git's design and use. The index is introduced and discussed where appropriate in the sections that follow.

### Similarities and differences between Git and Mercurial

Git and Mercurial are conceptually similar, but their command syntax and nomenclature differ in some places. Following are some of the basic concepts, commands, and terminology with which you need to be familiar. The key differences between Git and Mercurial are pointed out where appropriate.

#### The repository

The *repository* (or *repo*, for short) is the place where the history of source code changes over time is stored. The *local repository* is the one on the developer's own machine. For any given project, any repository other than the developer's local one is a *remote repository*. If Alice and Bob are working on the same project and want to share changes with one another, Alice's local repo can serve as Bob's remote repository and vice versa, assuming the appropriate permissions have been set. More commonly, developers collaborate via a remote repository located on a shared resource such as a file server that's accessible to

everyone on the team. In situations where this isn't feasible, developers can share changes by creating patch files and exchanging them with one another by email.

In a DVCS, a project's root folder—the folder where the developer's local copy of the source code files is stored—is referred to as the *working directory* or *working tree*. This folder is the parent of the local repository subfolder. The files in the working directory are collectively referred to as the *working copy*.

In both Git and Mercurial, a new repository is created with the *init* command. This command creates the folder structure and files the DVCS uses to manage and store the local repository. In Git the local repository is stored in the *.git* subfolder, while in Mercurial it's in the *.hg* subfolder.

Shared remote repositories typically do not have a working directory. These repositories contain only the history of changes, not the source code files themselves. For this reason, they are called *bare repositories*. Bare repositories are usually located on a file server or other shared resource. They do not need a working copy because developers do not make changes directly to files in that location.



To create a bare repository in Git, you need to use the *--bare* option with the *init* command. Mercurial does not have, nor does it require, this option.

### Tracking changes to files over time

Version control systems track files, but not all files in a project need to be tracked. Files that have been placed under version control are referred to as *tracked files* or *versioned files*. Other files in the same project that are not placed under version control are called *untracked* or *non-versioned* files.

When you want to begin tracking a file, you *add* it to the repository.

Mercurial and Git both provide a way to *ignore* certain types of files, such as executables, that you never want to track. This is done by specifying the files, folders, and matching patterns (e.g., \*.exe) you want to be ignored. In Git this file is called *.gitignore* while in Mercurial it's called *.hgignore*.<sup>1</sup> The two systems use essentially the same syntax for these files, so if you're already using Mercurial you can make a copy of your hgignore file and save it in your Git repository as gitignore and it should be valid, at least as a starting point.

When you want to check the state of the repository, you ask for the *status*. Files can be in one of several states: new, modified, staged (Git only), untracked, or ignored.

---

<sup>1</sup> Files and folders whose names start with a dot—commonly referred to as "dot files"—are treated as hidden in Unix and Linux operating systems. This naming convention is carried over to Windows implementations of Mercurial and Git, although Windows does not hide them in the file system. In this paper, the leading dot is henceforth dropped when referring to dot files in text after the first time they're mentioned.

After you've modified a file or files in the working directory, you *commit* those modifications to the local repository. Commits are accompanied by a *commit message* describing the changes.



In Mercurial, you only need to *add* a file to the repository once. Thereafter, if a tracked file has been modified, Mercurial's default behavior is to include that file in the next commit.

In Git, if you modify a tracked file you must *add* it again before you can to commit it. This is called *staging* the file. Staging a file creates a new snapshot of the file's contents and places it in the *index*, a data structure Git uses to keep track of the status of tracked files. Git's *commit* command updates the repository from the index rather than directly from the working copy. Although staging adds another layer to the commit process, it enables you to commit the parts of your work that are ready to go while holding back other modifications until later.

With only a few exceptions, Mercurial's history of commits is immutable. This means anything you commit to a Mercurial repository always appears in its history. Git objects are also immutable but Git offers several ways to modify history. This is good if you want to be able to "clean up" a history of changes before sharing or making them public, but the trade-off is that some commits may disappear from Git's history. Purists would say this means you can't really know if a file got from state "A" to state "B" in one step or as the result of several intermediate steps.

If you want to temporarily set aside changes to a file before committing, you can *stash* those changes in Git or *shelve* them in Mercurial. Those changes can later be retrieved and re-applied, or they can be discarded.

Changes to different sections of code in the same file are referred to as *hunks*. There may be times when you want to commit some hunks while stashing others for later.

The set of changes to one or more files that comprise a single commit is called a *changeset* in Mercurial. Git refers to it simply as a *commit*. In both Git and Mercurial, commits are uniquely identified by a SHA-1 hash, a 40-character strings of alphanumeric characters. Git calls this a *commit ID* while Mercurial refers to it as a *changeset ID*.

Because full SHA-1 IDs are unwieldy to write out in full or to refer to in conversation, it's customary to abbreviate them and refer only to the first few characters, whatever number it takes to be unique within the repository. Git refers to this as a *short SHA-1*. A short SHA-1 must be at least four characters long, but eight characters is common. For example, you could refer to commit ID `2a6c5ecc062752464faef0e8e616cda4aa61721f` using the short SHA-1 `2a6c5ecc` and Git is smart enough to figure out which commit you're referring to.

The odds of two different objects ending up with the same SHA-1 hash, called a SHA-1 collision, are extremely remote. The following quote is from the online version of the book *ProGit* by Scott Chacon and Ben Straub:

“Generally, eight to ten characters are more than enough to be unique within a project. For example, as of February 2019, the Linux kernel (which is a fairly sizable project) has over 875,000 commits and almost seven million objects in its object database, with no two objects whose SHA-1s are identical in the first 12 characters.”

The authors whimsically conclude that the chance of a SHA-1 collision is “less likely than every member of your programming team being attacked and killed by wolves in unrelated incidents on the same night.” See A SHORT NOTE ABOUT SHA-1 at <https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection> for the full article with interesting details.

When you want to see the difference between the contents of a file in two different states, you ask for a *diff*. For example, when preparing to do a commit you might want to view the changes that are about to be committed. When researching how changes to a particular file evolved or where a certain change was made, you may want to compare the contents of that file in one commit to the contents of the same file in a different commit. When run from the command line, diffs are displayed in *unified diff* format, aka *combined diff* format. Both Git and Mercurial enable you to specify an external diff tool, such as KDiff3 or Beyond Compare, to display the differences in a side-by-side format that's visually easier to understand.

### Tags and other references

It's often useful to assign a human-readable name to a commit so it can be more easily identified later in the history of changes. This is done by applying a descriptive *tag*.



In Mercurial, adding a tag creates a new commit. In Git, it does not. The rules for valid tag names differ, too. For example, "Release 1.2.345" is a valid tag name in Mercurial but not in Git because, among other rules, Git does not allow spaces. Tags in Git come under the rules for creating reference names – see <https://git-scm.com/docs/git-check-ref-format>.

In addition to the globally unique SHA-1 changeset IDs, Mercurial uses locally scoped *revision numbers* to identify changesets within the local repository. Unlike changeset IDs, revision numbers are simply sequential integers whose value is unique only within the local repository. Many Mercurial commands can make use of Mercurial's *revsets* language to reference a single revision or range of revisions by their revision number. It's also easier to talk and write about revision numbers than SHA-1 strings. Git does not use revision numbers, so—ignoring tags and other types of references for the moment—Git commits are referred to only by their full or abbreviated commit ID.

When you want to sync your working copy with a particular commit in the repository you *checkout* that commit in Git or *update* to that commit in Mercurial. In both cases, the files in the working copy are modified to match their state in the designated commit. To me, *checkout* seems like a better choice of words than *update*, because update implies checking out something newer when in fact you can (and often will) checkout a commit that's older than the files in your working copy.

### Using a remote repository

When you are ready to share changes from your local repository, you *push* them to a remote repository.



Mercurial's default behavior is to push all branches containing commits newer than those on the remote. Git's default behavior is to push only the current branch. Mercurial pushes tags by default (because they are separate commits). Git does not push tags unless you add the *--tags* option to the *push* command.

When you need to get changes made by others, you *pull* them from a remote repository.



Both Git and Mercurial have a *pull* command but its behavior is different in important ways. In Mercurial, a *pull* updates the local repository with newer commits from the remote repository, but it does not update the working copy unless you use the *-u* option. In Mercurial, the developer typically follows the *pull* with a *merge* to update the working copy and then a *commit* to update the local repository with the merged changes.

In Git, the *pull* command does all of that in one step. If you want to pull down newer commits from a remote repository in Git without updating the working copy, you use the *fetch* command.

Mercurial does not have a native *fetch* command. It does have a *fetch* extension, but that behaves like Git's *pull* – confusing, to say the least.

If you want to create a complete copy of an existing repository in a new location, you *clone* it. The clone is an identical copy of the original.

### Branching and merging

Branches are divergent lines of development within a repository. There is always one main branch. In Git it's called the *master* branch while in Mercurial it's called the *default* branch. Developers can create other branches and give them names.

Some development teams use the master branch as the release branch—i.e., the branch from which the app is compiled and released—while other teams prefer to use a designated "stable" or "release" branch. How you choose to do it is a matter of individual preference or perhaps company policy. Among the few teams I have talked to the preference seems to be to avoid the complexity of a release branch in favor of simply releasing from the master branch.



In Mercurial, a branch is not actually created until the first commit is done on it. Until you commit, the new branch doesn't appear in Mercurial's list of branches. Thereafter, the branch persists permanently in history—it can be closed but not removed.

Git, on the other hand, uses what are often called light-weight branches. Git's *branch* command immediately creates a reference to the new branch, which you

can see if you list the branches. You must then *checkout* that branch before committing anything to it. When you're done with a branch in Git you can keep it or delete it, in the second case either after merging its contents into another branch or simply discarding it entirely. Because of this, one school of thought in Git is that *\*all\** modification should be made on a feature or development branch—in other words, development work should never be committed directly to the master branch. Branch names in Git can be re-used after being deleted, so you can do all your work in a *dev* branch, delete it after merging back into *master*, and then create a new *dev* branch when you start on the next round of changes.

In both Mercurial and in Git, a *head* is a changeset that has no children. The most recent commit in each branch is the head of that branch. In Mercurial, the *tip* is the most recent head in the entire repository. In Git, the terms *head* and *tip* are used interchangeably to refer to the most recent commit on any branch, as in "the head of the branch" or "the tip of the branch".

Git uses a symbolic reference named HEAD (capitalized) that always points to the most recent commit on the current branch. If you check out a different branch, the HEAD reference changes to point to the most recent revision on that branch. There are also other HEAD references in Git, such as ORIG\_HEAD, FETCH\_HEAD, and MERGE\_HEAD. You may encounter these references from time to time when working with Git.

In Mercurial, if you update to (check out) a revision that is not the head of its branch, Mercurial notifies you that it is *not a head revision*. In Git, you are notified you are in a *detached head* state.

It's often necessary to combine the changes made to a file by one developer with the changes made to the same file by another developer, or by the same developer in a different branch. This is called doing a *merge*.



In Mercurial, a merge creates a single new commit comprising all the changes being merged. Git, on the other hand, automatically performs a *fast-forward* merge whenever possible. A fast-forward merge moves the HEAD pointer of the target branch but does not create a new commit. If you use the *--no-ff* option to override this behavior, Git creates a merge commit and moves the HEAD pointer of the target branch to it. This most closely corresponds to a Mercurial merge.

If a merge is attempted after two different sets of changes have been made to the same lines of code—for example, by two different developers or by the same developer on two different branches—a *merge conflict* can occur. Merge conflicts must be *resolved* before the merge can be completed. When performing merges, the changes in the target branch are sometimes referred to as "ours" while the changes from the other branch are referred to as "theirs". Resolving a merge conflict involves accepting "ours", accepting "theirs", or editing the file(s) to create some combination of the two.

If you want to copy specific changes from other branches onto your local branch, you *graft* them in Mercurial or *cherry pick* them in Git.

## Comparison of basic commands

Table 1 presents some of the most frequently used commands in Git and Mercurial. Many of them share the same name in both systems, although sometimes they behave differently. Others have the same behavior but different names.

Table 1: Comparison of common Git and Mercurial commands.

Function	Git	Mercurial	Comments
<b>Create a repository</b>	init	init	The repository is created in subfolder <code>.git</code> for Git and in <code>.hg</code> for Mercurial.
<b>Start tracking a file</b>	add	add	Git requires <i>add</i> after each modification (called <i>staging</i> ). Mercurial does not.
<b>List the files being tracked</b>	ls-files	manifest	
<b>Commit a set of changes to the local repository</b>	commit	commit	Both require a commit message
<b>View the history of commits to the local repository</b>	log	log	
<b>Push changes to a remote repository</b>	push	push	By default, Mercurial pushes all branches. Git pushes only the current branch.
<b>Pull changes from a remote repository</b>	pull	pull	In Git, use <i>fetch</i> to pull without updating the working copy. In Mercurial, use the <code>-u</code> option to update the working copy.
<b>Get the status of the working directory</b>	status	status	Lists new and modified files. Git indicates if the files have been staged.
<b>Get a list of existing branches</b>	branch	branches	

<b>Create a branch</b>	branch	branch	
<b>Sync the working copy with a specific revision or branch</b>	checkout	update	
<b>Resolve a merge conflict</b>	-	resolve	Git puts conflict markers in the file(s), which must be edited to resolve the conflict.
<b>Create a copy of an existing repository in a new folder</b>	clone	clone	
<b>Show the differences between revisions for specified file(s)</b>	diff	diff	

## Getting help

The most direct way to get help in either Git or Mercurial is to run the *help* command from the command line.

### Getting help in Mercurial

In Mercurial, the help command displays the requested MAN (manual) page in the command window. Running the help command with no parameter returns a list of common commands for which help is available. To get help for a particular command, run the *help* command with the name of another command as the parameter.

```
hg help      && returns a list of commands
hg help log  && displays the MAN page for the log command
```

Most MAN pages require much more than a single screen to be displayed in the command window. If you want to capture the output in a more readable format and perhaps save it for future reference, direct the output to a text file.

```
hg help log > log.txt && captures the MAN page for the log command to log.txt
```

### Getting help in Git

As in Mercurial, running the help command with no parameter in Git displays a list of common commands. Unlike Mercurial, running the help command with the name of another command as the parameter open the MAN page for that command as an HTML page in your default browser. The MAN pages are installed on your local hard drive. An index page is found at <file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/>.

```
git help      && returns a list of commands
git help log  && opens C:/Program Files/Git/mingw64/share/doc/git-doc/git-log.html in
your default browser
```



Keep the local Git documentation handy by opening a browser tab to any help page. Leave it open and change the part of the html file name that follows “git-“ as needed. The rest of the URL remains unchanged. For example, change *log* to *push* in the listing above and refresh the page to get help on the *push* command.

Many Git command have a bewildering array of options. The good news is you don't have to be familiar with all of them, only with the ones most commonly used. Some options have both a long form, which begins with a double dash, and a short form that begins with a single dash. For example, *-n* is the short form of *--dry-run* and *-v* is the short form of *--verbose*. Capitalization is sometimes important, too. The short form of the *--all* option for the *add* command is *-A* (capitalized). The *git add --all* command can also be shortened to *git add .* (*git add* followed by a space and a period), which is much easier to type.

Due to the typically large number of options, the HTML MAN pages for Git commands tend to be quite lengthy. They are useful for reference but don't make for great reading. Fortunately, there are many other resources to help you learn Git. Several of them are listed in the References section at the end of this paper.

### Getting Started with Git on Windows

Git's home on the Web is <https://git-scm.com>. Git's roots are in Linux®, so it runs natively on Linux and Apple operating systems. For Windows you need to install a project called *Git for Windows*. You can download Git for Windows from the “Downloads for Windows” link on the Git homepage, or from its own home on the Web at <https://gitforwindows.org>.

Git for Windows enables you to run Git from the command line on a Windows machine. It also comes with a graphical user interface called Git GUI. This paper is based on Git for Windows v2.21.0 and 2.22.0, which were released in early 2019. A newer version is probably available by the time you read this. As with most software, it's generally advisable to use the latest release.

Git for Windows implements most but not all Git commands. After installing Git, check the release notes for details about the version you installed. If you installed Git to its default location on a 64-bit Windows machine, there is a local copy of the release notes in <file:///C:/Program Files/Git/ReleaseNotes.html>.

### Installing Git for Windows

The Git for Windows installer works like any other Windows application installer but be prepared to make a few decisions you might not expect. Following are the dialogs you'll encounter when installing Git for Windows 2.21.0 / 2.22.0. The available options tend to change a bit over time—for example, Visual Studio Code was recently added as a choice for the default editor—so if you're installing an earlier or a later version you might see some differences.

The first dialog presents the license agreement. The installer does not require you to accept the license, but of course you're implicitly agreeing to it if you continue.

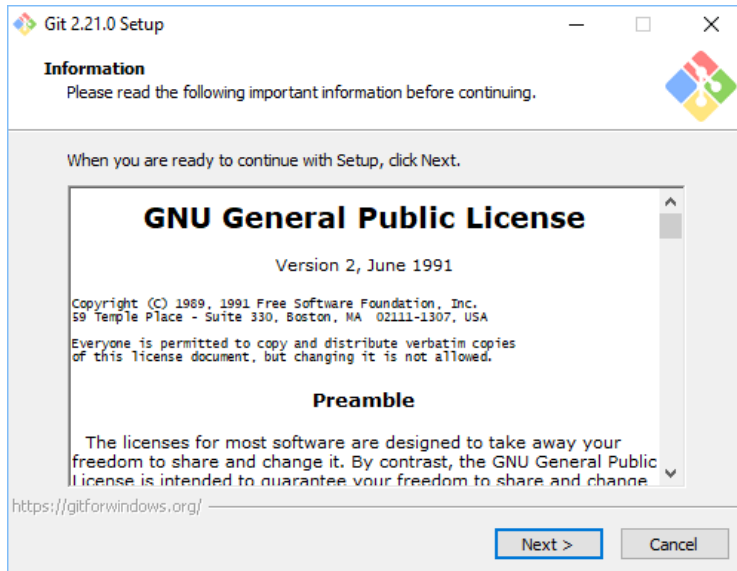


Figure 1: Read and understand the license agreement.

The second dialog is where you select the components you want to install. Figure 2 shows the default choices.

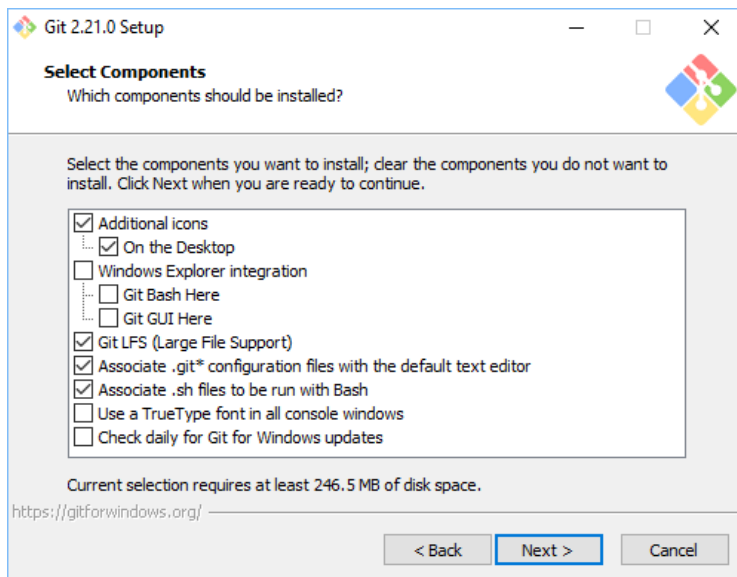


Figure 2: Choose the desired components - these are the defaults.

The third dialog enables you to choose the editor you want to use as the default for Git. The default is Vim but as a Windows developer you'll probably want to choose a different one.

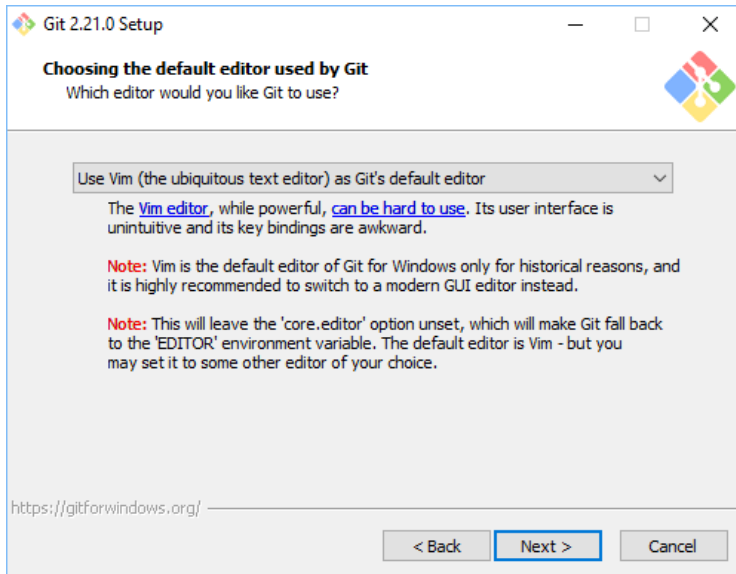


Figure 3: The default editor for Git is Vim, but you can choose another one.

The dropdown list of available editors now includes Visual Studio Code. If you don't like any of the suggested choices, you can select a different one.

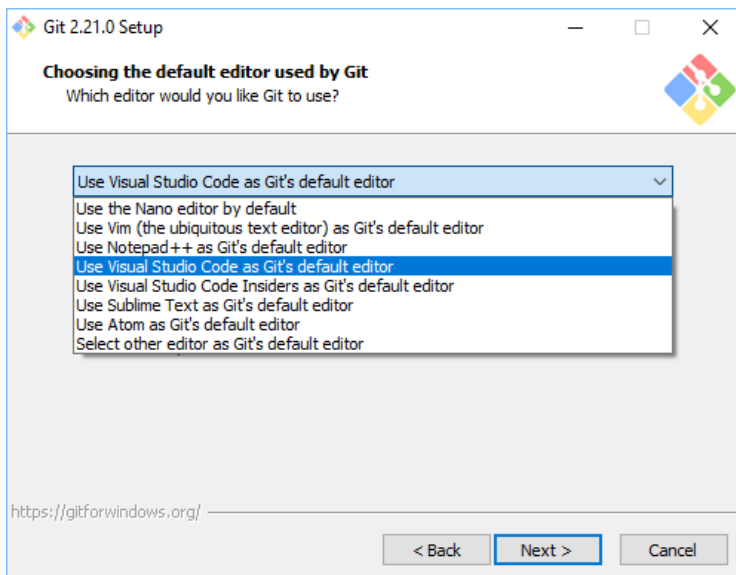


Figure 4: Visual Studio Code is now available as a default editor for Git.

If you choose Visual Studio Code you see the dialog in Figure 5. I believe the warning about this choice being installed “only for this user” is because Visual Studio Code itself is installed on a per-user basis.

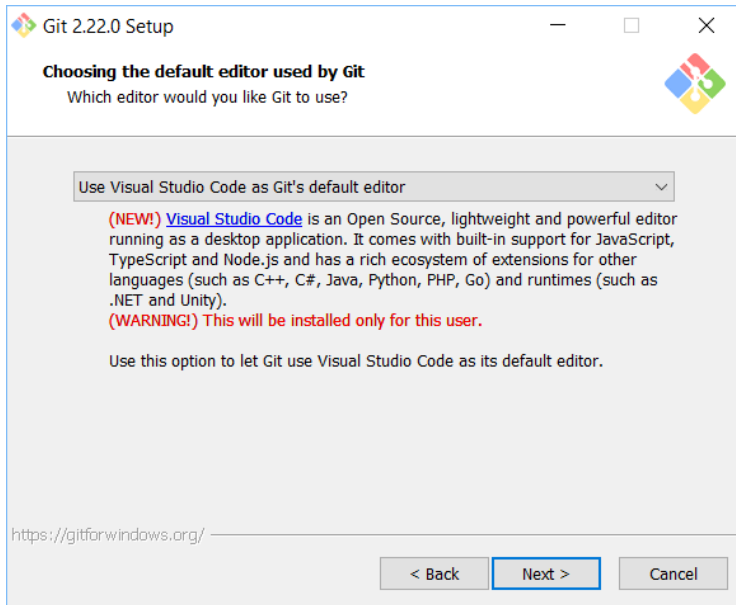


Figure 5: This dialog provides information if you chose Visual Studio Code as the default editor.

The next dialog asks you to choose if and how to modify your Windows path in order to use Git. If you want to be able to run Git commands from the Windows Command Prompt in any folder, select the second option as shown in Figure 6. This tells the installer to add `C:\Program Files\Git\cmd` to your Windows PATH.

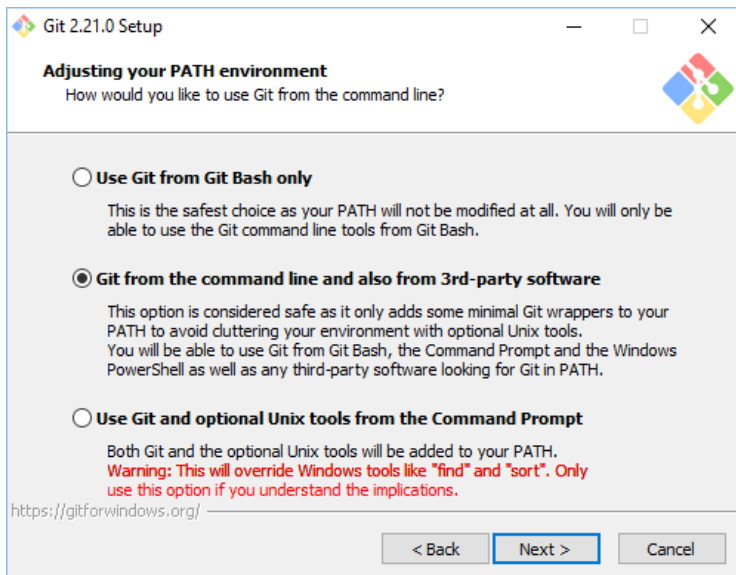


Figure 6: Choose the second option if you want to be able to run Git commands from the Windows Command Prompt in any folder.

The seventh dialog asks you choose the executable for SSH. The default is openSSH.

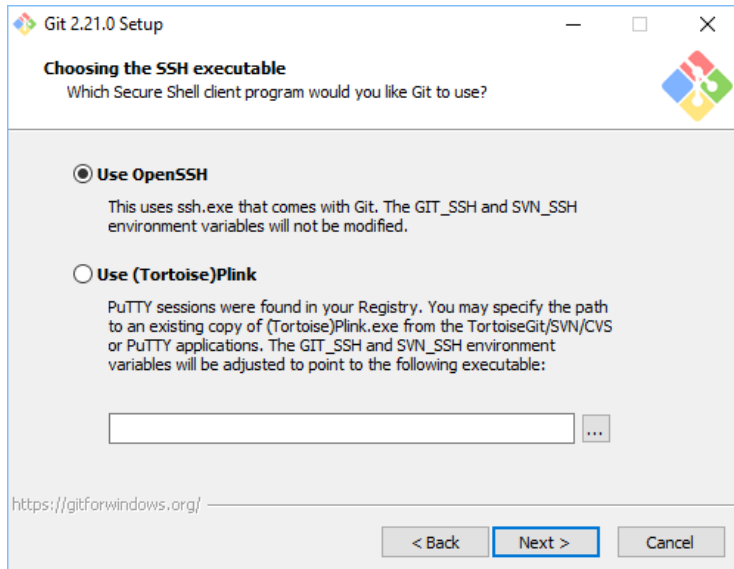


Figure 7: OpenSSH is the default.

OpenSSH is the default library for HTTPS.

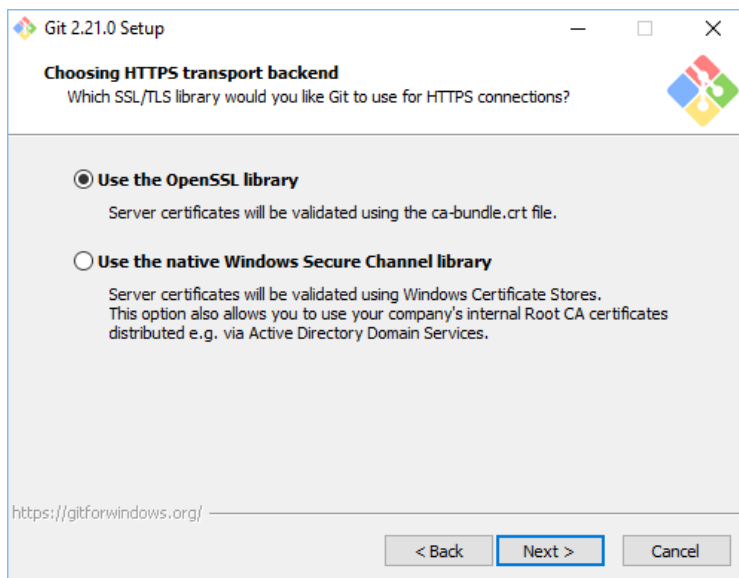


Figure 8: Choice of SSL library.

The next dialog asks you to choose a line ending option. This can make a difference in a cross-platform development environment because Windows, Linux / Unix, and Apple OS each use different line ending characters. The choices are explained in the dialog shown in Figure 9. The first option is the default for Git on Windows and is suitable for developers working in cross-platform development environments. The third option is OK for single-platform development environments.

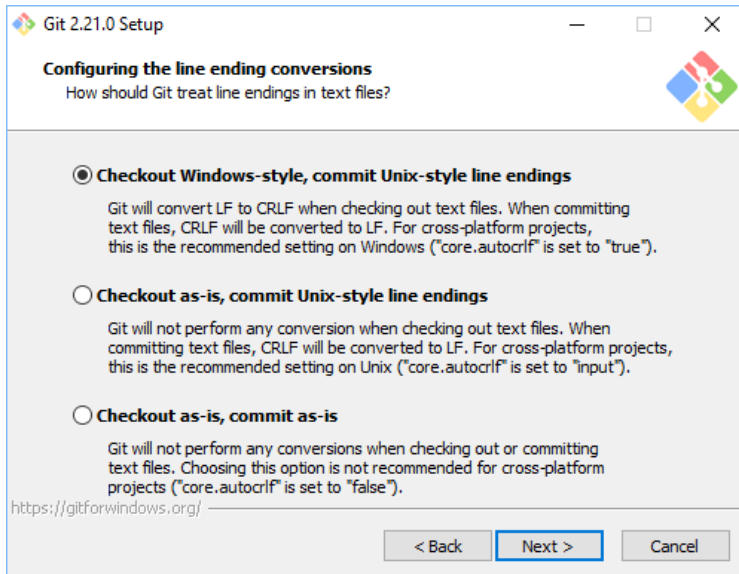


Figure 9: Choose a line-ending conversion setting.

The next dialog offers a choice between two options for terminal emulation – in other words, how you prefer to use Git from a command prompt. The first option, which is the default, installs an emulator for Git Bash. This offers what the installer describes as a more fully featured experience—better use of colors, more information on the command prompt, resizable window, etc.—but takes some getting used to if you’re not familiar with Linux. Choosing the first option does not prevent you from also using Git from the Windows command prompt, so I don’t see any downside to it. See Figure 10 for more information.

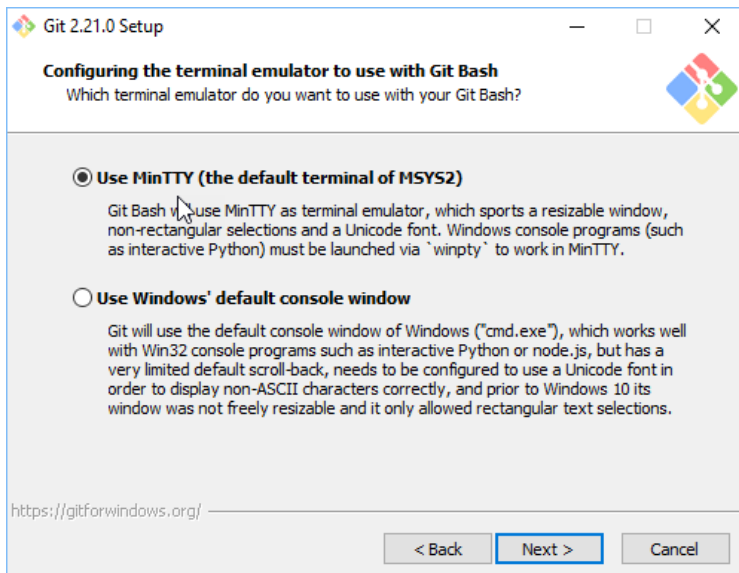


Figure 10: You get a more fully featured shell by selecting MinTTY as the terminal emulator.

The next set of options are all selected by default.

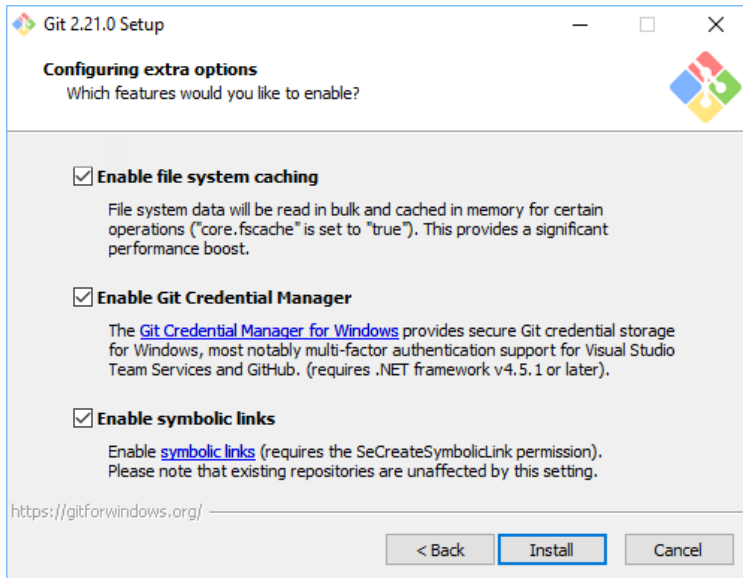


Figure 11: These extra options are selected by default.

The final dialog enables you to turn on experimental feature(s). These features vary depending on the version of Git you are installing. In Git 2.22.0 it's an experimental interactive 'add' command. I chose not to turn it on so I can't comment on how it works. Experimental features often become integrated features in future releases anyway.

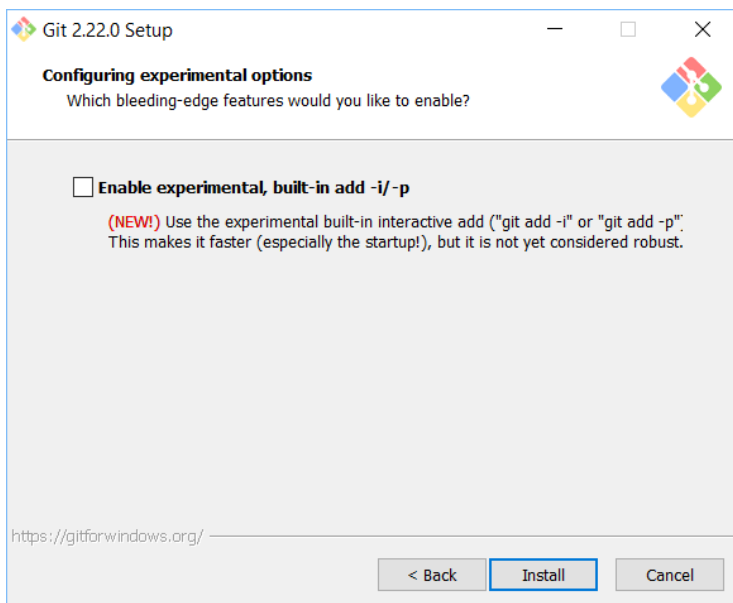


Figure 12: This dialog is for experimental features. These change with different versions of Git.

From there on the installation offers no more surprises and proceeds as expected. The following dialog is displayed when installation is complete.

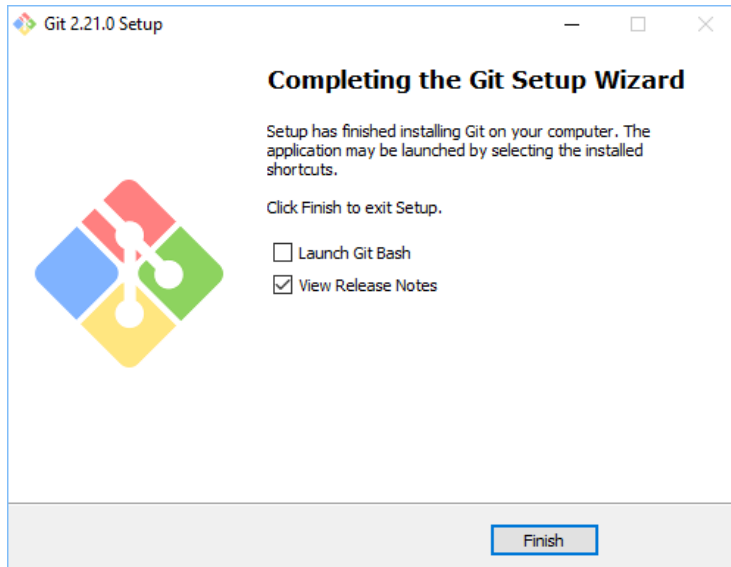


Figure 13: Setup is complete – you’re ready to begin using Git.

Once installed, you can verify Git is working as expected by opening a command window and asking Git to display its version number.

```
C:\>git --version
git version 2.22.0.windows.1
```

If Git responds with the version number, your installation is OK. If Windows responds with the message *'git' is not recognized as an internal or external command, operable program, or batch file*, you probably need to modify your PATH to include `C:\Program Files\Git\cmd`.

### Configuring Git

The first thing to do after installing Git on a new machine is to configure it with your name and email address. This is required because Git associates every change made to tracked files with the person who made it. If you set these values in your PC's global (per-user) configuration file, Git uses them for every project you work with on that PC. You can override the global defaults with per-repository settings for specific projects if that becomes necessary.

You can set the global defaults for your name and email address from the command line like this:

```
git config --global user.name "Rick Borup"
git config --global user.email rborup@ita-software.com
```

If you're working from a GUI interface, look for the appropriate menu item to edit these and other configuration settings. In Git GUI they're available from the Edit | Options menu as shown in Figure 14.

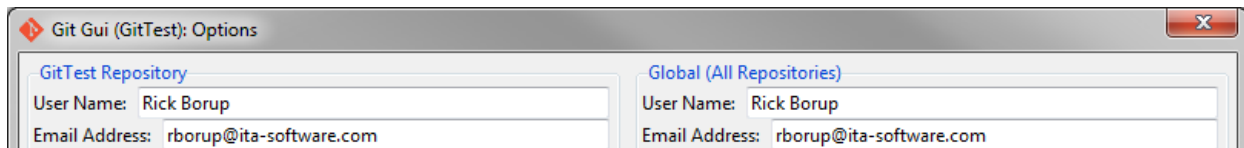


Figure 14: Git GUI provides a form to enter your local and global username and email address.

There are many other configuration settings you may want or need to change as you use Git, but for starters you only need those two. Git stores global configuration settings in a dot file named *.gitconfig* located in the user's profile folder. It's a standard INI file format you can open in any text editor. Listing 1 shows how a brand-new *gitconfig* file looks after setting the username and email address.

Listing 1: The user's global configuration settings are stored in `C:\Users\<username>\.gitconfig`.

```
[user]
  email = rborup@ita-software.com
  name = Rick Borup
```

## Preparing your project for Git

### Initializing a new repository

To use Git as the version control system for your project, begin by initializing a new Git repository in your project folder. From the Windows command prompt, navigate to your project's root folder and run Git's *init* command.

```
C:\myProject>git init
```

This creates the *.git* folder and its subfolders. By default, the *.git* folder is created as a hidden folder in Windows, so you won't see it in the list of folders and files in your project's root. This isn't a problem because it's usually not necessary to interact with the *.git* folder structure directly, but if you want to see it you can remove the hidden attribute so it shows up.

```
C:\myProject>attrib -h .git
```

### Telling Git which files and folders to ignore

The next step is to tell Git which files you want to place under version control—or, more accurately, to tell Git which files you do *not* want to place under version control. Git uses the *gitignore* file for this purpose. The *gitignore* file has essentially the same syntax as Mercurial's *hignore* file, so if your project is already under Mercurial version control you can save a copy of your *hignore* file as *gitignore* and use it with Git, at least as a starting point.

There are basically two ways to use a *gitignore* file. One is to tell Git to ignore everything (use *\*.\** in *gitignore*) and then to individually *add* the files you want to track. The advantage to this approach is that nothing you don't explicitly add will get tracked, so there's no risk of accidentally cluttering up your repository with files or folders you don't want. The

disadvantage is that for your initial commit you must individually add every file you want to track rather than being able to add all them all in one step.

The more common approach is to create a gitignore file identifying the files and folders you don't want to track. These can be specified either as individual files or folder name, or by using pattern matching. For example, placing *\*.bak* in the gitignore file tells Git to ignore all files with a .bak extension, while *Temp/* ignores all files in the Temp folder and its subfolders.

The initial commit to a new repository should include all the files and folders you want to track. You can use the *--all* option with Git's *add* command to include all of them in one step, a major convenience over having to add them individually. Without a gitignore file, the *add --all* command would add every file in your project folder and its subfolders. To avoid this, make sure the gitignore file is in place before the initial commit so *add --all* adds only those files and folders that are not being ignored.

Either way, it's a good idea to use the *--dry-run* option to preview what will get added before actually making the initial commit. If you discover files or folders in the list that you don't want to track, you can tweak your gitignore file and repeat the *--dry-run* until you get the results you want.

As an example, consider a minimalist VFP project called myProject. The project folder contains myProject.PJX/PJT, a main program file named main.prg, and a DLL named SuperCrypto.dll, and some temporary files in a Temp subfolder as shown in Listing 2.

Listing 2: Directory listing of files in a VFP project.

```
07/12/2019 04:40 PM <DIR>      .
07/12/2019 04:40 PM <DIR>      ..
07/12/2019 04:40 PM          321 .gitignore
07/12/2019 04:39 PM          7 main.prg
07/12/2019 04:39 PM          7 myProject.pjt
07/12/2019 04:39 PM          7 myProject.pjx
07/12/2019 04:39 PM          7 SuperCrypto.dll
07/12/2019 04:39 PM <DIR>      Temp
```

You decide you want to track the project files and main.prg because they will change over time, but you don't want to track the DLL file or any of the files in the Temp subfolder and you also don't want to track compiled FXP files. A possible gitignore file for this project is shown in Listing 3. Entries are case sensitive, so it's advisable to specify both possibilities. Comment lines, which begin with a # symbol, are optional.

Listing 3: This gitignore file tells Git to ignore compiled VFP program files, a specific DLL, and anything in the Temp folder.

```
#-----
# Files to ignore
#-----
*.fxp
*.FXP
```

```
SuperCrypto.dll
```

```
#-----  
# Folders to ignore  
#-----  
[Tt]emp/
```

Running `git add --all --dry-run` produces the following output.

```
add '.gitignore'  
add 'main.prg'  
add 'myProject.pjt'  
add 'myProject.pjx'
```

Note that the gitignore file itself is listed. That's because it's present in the project folder and is not being explicitly ignored. As with other files that may change over time, it's customary to include gitignore in the repo.

### VFP binary source code files

The essence of using source control is to enable diff'ing and merging of code from different lines of development. The problem for Visual FoxPro developers has always been that you can't diff or merge VFP's binary source code files. To enable diff'ing and merging, you need a way to create a text-equivalent version of the binary files and place those files in the repository.<sup>2</sup> Before every commit, including before the initial commit, the VFP developer must create a up-to-date text-equivalent version of the current binary code files and include them in the commit.

The question every VFP developer needs to answer is, should the binary source code files also be included in the repo? It's a trade-off of safety versus repository size. If you have complete confidence in your binary-to-text/text-to-binary tool and are willing to rely on it to recover the binary form if necessary, you can leave the binary files out of the repository. On the other hand, including the binary files in the repo guarantees they can be recovered directly if for any reason re-creating them from their text-equivalent file fails.

Appendix A is a suggested gitignore file for Visual FoxPro projects. It is configured to include the VFP binary files in the repository. To exclude them, remove the comment mark—the # symbol—from the lines for the types of files you wish to exclude.

### Adding files to the repository

Once your gitignore file in place you're ready to tell Git to start tracking the files you want to place in your repository. The `add` command tells Git which file or files to include. Because this is the first commit and nothing is yet being tracked, you'll want to add all the files that are not excluded by gitignore. You can do this in one step by using the `--all` option. Use the

---

<sup>2</sup> My tool of choice is the Fernando Bozzo's excellent FoxBin2Prg, which creates .xx2 files—for example, .sc2 from .scx—in .prg format from their binaries, and vice versa.

`--dry-run` option if you want to preview what will be added without actually adding anything. The file `myProject.pj2` is the text-equivalent of `myProject.pjx`.

```
C:\myProject\git add --all --dry-run
add '.gitignore'
add 'main.prg'
add 'myProject.pjt'
add 'myProject.pjx'
add 'myProject.pj2'
```

When you're ready to proceed, run the add command without the `--dry-run` option.

```
C:\myProject\git add --all
```

Git stages the files but there is no output from this command.

### Checking to see what got added

The `status` command shows the state of the repository by listing files whose state has changed since the last commit, for example by being modified and/or staged. When run from the Windows command line, modified files are shown in red while staged files are shown in green. The status command's default output also tells which branch you're on and includes information and instructions about how to discard your changes, how to stage files for commit, and whether there are modified files that have not yet been staged. Use the `--short` option for a more condensed list.

In the examples below, file `main.prg` has been modified but not yet staged. Listing 4 is the default output. Listing 5 is the short version of the same command.

Listing 4: The status command shows that the file `main.prg` has been modified but not yet staged for commit.

```
C:\myProject>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
       modified:   main.prg
no changes added to commit (use "git add" and/or "git commit -a")
```

Listing 5: The output from the status command with the `--short` option.

```
C:\myProject>git status --short
 M main.prg
```

After the file has been staged, the status command shows the file name in green.

```
C:\myProject>git add main.prg && stage the file to be committed
C:\myProject>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
       modified:   main.prg
```

In the short form, only the “M” (for “modified”) is green.

```
C:\myProject>git status -s
M main.prg
```

### Performing the initial commit

The `commit` command tells Git to record the staged changes to the repository. This is the final step in the process. Committing is quick because Git already created the objects to be committed when you ran the `add` command. These objects are stored in Git’s intermediate storage area, the index.

Like the `add` command, the `commit` command has a `--dry-run` option you can use to see what will be committed before actually committing them. Commits require a commit message, but you can leave it out if you're doing a dry run.

```
C:\myProject>git commit --dry-run
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
       modified:   main.prg
```

Use the `-m` option to include the commit message. If you run the `commit` command without the `-m` option, Git opens a temporary file in your default editor and prompts you to enter the message. Figure 15 shows this in Visual Studio Code.

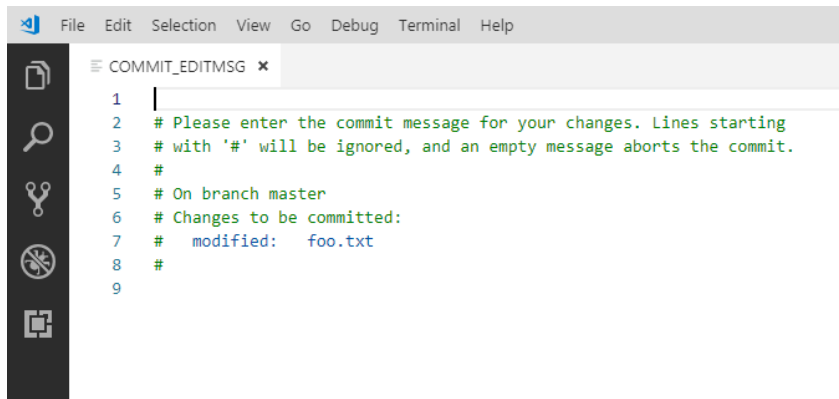


Figure 15: Git opens the default editor if you omit the `-m` option on a commit.

This is useful if you intend to write longer commit message, possibly with a summary line at the top and details below as is the convention used by some developers. If you take this approach, Git waits for you to close the editor before finishing the commit.

For short, single-line commit messages the `-m` option is quicker and easier.

```
C:\myProject>git commit -m "Remove blank last line from main.prg"
[master 098e080] Remove blank last line
 1 file changed, 1 deletion(-)
```

## Using Git in your daily development workflow

Once your project is under Git version control, you'll want to adopt a routine set of steps for interacting with it. These steps become your standard daily workflow. Some of the decisions that factor into creating a workflow that works for you involve strategies for committing, branching and merging, and determining how and when to interact with remote repositories.

### Start with fetch if using a remote repository

If there's a remote repository involved with your Git project, your first task before beginning work should be to issue a *fetch* command to see if the remote has anything newer than you have in your local repository. I can tell you from experience that you should strive to make this part of your workflow even if you're the only developer working on the project. If you sometimes push to the remote from two or more different machines, like a desktop PC in the office and a laptop while at home or travelling, it's easy to forget which machine has the newest commits.

If there are newer commits on the remote repository and you make changes to files on your local machine before fetching from the remote, you'll find you can't pull or push without first taking steps to merge your local work with what's on the remote, or to stash or abandon your local changes.

### When and how often to commit

The rule of thumb is to commit often and in small chunks. This gives you the most granular control over your changes. A good time to commit is whenever you get to a meaningful breakpoint—for example, after completing and testing a new method, or after a few hours of coding even if what you're working on is not finished yet. Commits made to your local repository are typically for your benefit only, at least at first. Later, if you need to share your changes, you can clean up history using techniques such as squash merging and rebasing before publishing them.

### Branching and merging

A branch is a divergent line of development. Branching provides a way to keep the changes involved in developing a new feature or working on some experimental code separate from the stable, main line source code.

Although you could commit all your changes on the master branch, it's considered a best practice is to use *master* as the branch where the most recent stable version of the code resides. If you adopt this approach, the master branch typically serves as the release branch. All changes are made on a development branch or branches, which is then merged back into the master branch when the code is ready for release.

Some workflows involve many several different branches, all in use at the same time. Others may involve only a single development branch and the master branch. There is no "right" way of doing it. The best workflow is the one that works best for you or your team. The best advice I can offer is to keep it as simple as possible.

### When to create a branch

Create a new branch whenever you embark on a new line of development.

One difference to keep in mind is that in Mercurial, the branch is created at the time of the commit whereas in Git you need to create the branch and check it out before you can commit to it. This may require altering your workflow if you're used to working in Mercurial.

### How to create a branch

In Git, you can create a new branch with the *branch* command or with the *checkout* command.

Without any parameters, the command returns a list of branches.

```
C:\>git branch
  dev
* master
```

To create a new branch, use *git branch* followed by the name of the branch you want to create. Branch names must conform to the rules for Git reference names.<sup>3</sup>

```
C:\>git branch newbranch
```

Git notifies you if the branch name you specified already exists, otherwise it creates the new branch. Use the *checkout* command to switch to the new branch.

```
C:\>git checkout newbranch
Switched to branch 'newbranch'
```

You can create and check out a new branch in one step using the *-b* option with the *checkout* command.

```
C:\>git checkout -b newbranch
Switched to new branch 'newbranch'
```

### When to delete a branch

One plus of Git's lightweight branching model is that you can delete a branch if you no longer need it. This means you can create a branch to work on a new feature or to experiment with a new idea knowing it will not become a permanent part of the history unless you want it to.

If you're working in a branch and decide to discard the changes entirely, simply delete the branch. If you want to keep the changes, you can merge them into another branch (like *master*) and then keep or delete the branch as you choose. This is another reason it's a good idea to do all your work on a branch.

---

<sup>3</sup> See <https://git-scm.com/docs/git-check-ref-format>

When a branch has been deleted, it no longer appears in the list of branches and cannot be checked out. However, you can create another new branch using the same name as one that has been deleted. For example, you could repeatedly use *dev* or *temp* as the name for your working branch, deleting it each time you're done with it.

### How to delete a branch

Use the *git branch* command with the *-d* option to delete a branch.

```
C:\>git branch -d newbranch
Deleted branch newbranch (was 952edce).
```

Git has a built-in safeguard to prevent deleting the current branch, i.e., the branch that's currently checked out.

```
C:\>git branch -d newbranch
error: Cannot delete branch 'newbranch' checked out at 'C:/'
```

### How to perform a merge

Merging is the process of integrating the contents of a file from one commit with the contents of the same file from a different commit. The most common need for a merge occurs when the changes on a development branch are brought back into another branch such as the master branch.

#### *Merge without a merge conflict*

As an example, consider a file *foo.txt* with the following contents that has been committed to the master branch.

```
C:\>type foo.txt
line 1
line 2
```

The developer creates a new development branch, adds a line to *foo.txt*, and commits the change.

```
C:\>git checkout -b dev
Switched to a new branch 'dev'
C:\>Notepad foo.txt && add line 3
C:\>type foo.txt
line 1
line 2
line 3
C:\>git add foo.txt && stage foo.txt
C:\>git commit -m "add line 3" && foo.txt is committed on branch 'dev'
```

The developer now merges the change made on the *dev* branch back into the master branch. In this example, Git is able to perform a “fast forward” merge, which simply moves the HEAD pointer without creating a merge commit. In Listing 6, note that the HEAD pointer for both the *master* branch and the *dev* branch is positioned on the same commit (line 16).

Listing 6: A merge with no merge conflicts. In this example, Git was able to perform a fast-forward merge.

```
1. C:\>git checkout master
2. Switched to branch 'master'
3. C:\>type foo.txt && foo.txt doesn't have line 3 on branch 'master'
4. line 1
5. line 2
6. C:\>git merge dev
7. Updating f2ae23b..101006b
8. Fast-forward
9. foo.txt | 1 +
10.1 file changed, 1 insertion(+)
11.C:\>type foo.txt && foo.txt has line 3 after the merge
12.line 1
13.line 2
14.line 3
15.C:\>git glog && my alias for a nicely formatted graphical log
16.* 101006b (HEAD -> master, dev) add line 3
17.* f2ae23b initial commit
```

There were no merge conflicts, so the merge is complete.

If you want Git to create a merge commit even when a fast-forward merge is possible, you can use the `--no-ff` option to override the default fast-forward behavior (see Listing 7). The merged content is the same in both cases but with a different history.

Listing 7: The same example as in Listing 6, but using the no-fast-forward option.

```
1. C:\>git checkout master
2. Switched to branch 'master'
3. C:\>type foo.txt && foo.txt doesn't have line 3 on branch 'master'
4. line 1
5. line 2
6. C:\>git merge --no-ff dev
7. git merge --no-ff dev
8. Merge made by the 'recursive' strategy.
9. foo.txt | 1 +
10.1 file changed, 1 insertion(+)
11.C:\>type foo.txt && foo.txt has line 3 after the merge
12.line 1
13.line 2
14.added line 3 on 'dev' branch
15.C:\>git glog && my alias for a nicely formatted graphical log
16.* c60f2b8 (HEAD -> master) Merge branch 'dev'
17.|\
18.| * 4b4126c (dev) add line 3
19.|\
20.* f2ae23b initial commit
```

Compare the history after a merge commit (lines 16-20 in Listing 7) to the history after a fast-forward merge in to the last two lines in Listing 6. After a merge commit, the tip of the *dev* branch is still at commit 4b4126c (line 18), while the tip of the *master* branch is at c60f2b8 (line 16).

### *Merge with a merge conflict*

Now consider an alternate scenario, starting from the point in the previous example where line 3 has been added and committed on the *dev* branch but not yet merged back into the *master* branch. In this scenario, a hotfix is made to *foo.txt* on the *master* branch, then an attempt is made to merge the *dev* branch back into *master*.

```
C:\>git checkout master
Switched to branch 'master'
C:\>type foo.txt && foo.txt doesn't have line 3 on branch 'master'
line 1
line 2
C:\>notepad foo.txt && add a hotfix on line 3
C:\>type foo.txt
line 1
line 2
add hotfix on line 3 in 'master' branch
C:\>git add foo.txt
C:\>git commit -m "add hotfix on 'master' branch"
C:\>git merge dev && attempt to merge the 'dev' branch - conflict detected!
[master f73f314] add hotfix on 'master' branch
 1 file changed, 1 insertion(+)
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git has inserted conflict markers into *foo.txt*, which now looks like this:

```
line 1
line 2
<<<<<<< HEAD
add hotfix on line 3 in 'master' branch
=====
added line 3 on 'dev' branch
>>>>>>> dev
```

This line(s) between <<<<<<< HEAD and ===== are from the local version (working copy) of the file. The line(s) between ===== and >>>>>>> are what was brought in by the merge.

At this point you need to open an editor and resolve the conflict(s) by editing the code. After making the desired changes, remove the conflict markers and save the file. In this example the developer elected to keep both lines.

```
C:\>notepad foo.txt && resolve the conflict and remove the conflict markers
C:\>type foo.txt && this is what it looks like after the conflict was resolved
line 1
line 2
add hotfix on line 3 in 'master' branch
kept line 3 from 'dev' branch as line 4
```

*foo.txt* is now a modified file in the *master* branch and must be staged and committed.

```
C:\>git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   foo.txt
no changes added to commit (use "git add" and/or "git commit -a")
C:\>git add foo.txt
C:\>git commit -m "resolved conflict caused by merge with 'dev' branch"
[master 4150dc1] resolved conflict caused by merge with 'dev' branch
```

The merge is now complete, and the graphical history looks like this:

```
C:\>git glog && see footnote4
* 4150dc1 resolved conflict caused by merge with <E2><80><98>dev<E2><80><99> branch
| \
| * a9ae527 (dev) add line 3
* | f73f314 add hotfix on 'master' branch
| /
* ba73a77 initial commit
```

### Modifying history

Git provides several ways to modify history. This is often useful to fix simple mistakes, such as correcting a commit message or including a file that was inadvertently omitted from a commit, to more complicated corrections such as condensing multiple commits into one or moving a series of commits from one branch to another.

A general rule of thumb applies to the use of all commands that modify history: use them only on local commits. Do not use them if your changes have already been shared with other developers.

#### Modifying a commit

The *commit* command has an *--amend* option that enables you to modify the most recent commit. Amending a commit replaces the tip of the current branch with a new commit. Even if all you do is change the commit message, the new commit has a different SHA-1 ID because the commit message is part of the commit and therefore part of the content from which the SHA-1 is derived.

Listing 8 is an example of amending a commit to change the commit message. The original commit message on line 3 indicated bar.txt was changed, when in fact foo.txt was changed.

Listing 8: Use the *commit* command with the *--amend* option to change the commit message.

1. C:\>notepad foo.txt && make a change
2. C:\>git add foo.txt
3. C:\>git commit -m "changed bar.txt" && commit message is wrong

---

<sup>4</sup> "glog" is not a native Git command. It's an alias I created for "git log --graph --abbrev-commit --oneline". Aliases are discussed in the Common Questions section later in this paper.

```
4. [master 3f9747e] changed bar.txt
5. 1 file changed, 1 insertion(+)
6. C:\> git commit --amend -m "changed foo.txt" && amend with corrected message
7. [master cc6ecb6] changed foo.txt
8. Date: Wed Aug 7 16:51:13 2019 -0500
9. 1 file changed, 1 insertion(+)
10.C:\>git log -n 2 --oneline --abbrev-commit
11.cc6ecb6 (HEAD -> master) changed foo.txt
```

Note the commit ID of the amended commit on line 7 is different than the ID of the original commit on line 4. The log shows only the amended commit with the corrected commit message (line 11).

One caveat here is that you should never amend a commit if there's a chance it has already been shared with another developer, for example if it's been pushed to a shared repository. Think of amending a commit as a local operation only.

### Removing a commit

If you make a commit and immediately realize you really didn't want to do so, you can remove that commit. The same caveat applies here as applies to amending a commit – do so only if there's no chance the commit you are going to remove has been shared with another developer.

Suppose you're working with a file named `foo.txt` that has two lines and has already been committed to the repository.

```
C:\>type foo.txt
line 1
line 2
```

You add a third line to `foo.txt`, stage it, and commit it.

```
C:\>notepad foo.txt
C:\>type foo.txt
line 1
line 2
line 3
C:\>git add foo.txt
C:\>git commit -m "add line 3 to foo.txt"
[master da09cdb] add line 3 to foo.txt
 1 file changed, 1 insertion(+)
C:\>git glog
* da09cdb (HEAD -> master) add line 3 to foo.txt
* 8ae78be initial commit
```

You immediately realize you did not want to commit yet and would like to undo it. Git's `reset` command gives you a couple of ways to do this. It has three modes: `soft`, `mixed`, and `hard`. Your working copy should be clean before using the `reset` command.

### *Soft reset*

A soft reset moves the HEAD pointer back to the previous commit without touching the index or the working copy. After a soft reset, the state of your work is what it was after staging the changes.

```
C:\>git reset --soft HEAD~1  && HEAD~1 is way of referencing the commit before HEAD
C:\>git glog
* 8ae78be (HEAD -> master) initial commit  && commit da09cbd is not longer present
C:\>git stat  && status shows changes to foo.txt are still staged
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   foo.txt
C:\>type foo.txt  && line 3 still exists in the working copy
line 1
line 2
line 3
```

### *Mixed reset*

A mixed mode reset moves the HEAD pointer back to the previous commit and resets the index, but does not change the working copy. After a mixed reset, the state of your work is what it was after you changed your working copy but before you staged those changes.

```
C:\>git reset --mixed HEAD~1
Unstaged changes after reset:
M   foo.txt
C:\>git glog
* 8ae78be (HEAD -> master) initial commit  && commit da09cbd is not longer present
C:\>git stat  && status shows changes to foo.txt are not yet staged
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   foo.txt
no changes added to commit (use "git add" and/or "git commit -a")
C:\>type foo.txt  && line 3 still exists in the working copy
line 1
line 2
line 3
```

### *Hard reset*

A hard reset moves the HEAD pointer back to the previous commit and resets both the index and the working copy. After a hard reset, the state of your work is what it was before you made any changes in your working copy. You can lose work with a hard reset, so be sure it's what you really want to do.

```
C:\>git reset --hard HEAD~1
HEAD is now at 8ae78be initial commit
C:\>git glog
* 8ae78be (HEAD -> master) initial commit  && commit da09cbd is not longer present
C:\>git stat  && nothing has changed
On branch master
Nothing to commit, working tree clean
```

```
C:\>type foo.txt  && line 3 no longer exists in the working copy
line 1
line 2
```

### Squash merging

Suppose you're working on a new feature. You create a development branch named *dev* for it and make several commits along the way, each one reflecting the state of the code at successive points in its development.

At some point you're done with the new feature and are ready to merge it back into the master branch for release. In this example, file *foo.txt* starts out with two lines. In Listing 9, two more lines are added and committed on the *dev* branch, followed by a conventional merge back into the *master* branch.

Listing 9: Two commits are made on the *dev* branch, which is then merged back into *master*.

```
1. C:\>git checkout -b dev
2. C:\>notepad foo.txt  && add line 3
3. C:\>git add foo.txt
4. C:\>git commit -m "add line 3"
5. C:\>notepad foo.txt  && add line 4
6. C:\>git add foo.txt
7. C:\>git commit -m "add line 4"
8. C:\>git checkout master
9. C:\>git merge dev
10.C:\>git glog
11.44260bc (HEAD -> master, dev) add line 4
12.ab0aa57 add line 3
13.ba73a77 initial commit
```

Git performed a fast forward merge. Note that both commits made on the *dev* branch (lines 4 and 7) appear in the history of the *master* branch (lines 11 and 12).

What if you'd prefer the history not to show all the intermediate commits made on *dev*, but rather to merge all of them as one commit on *master*?

That's what a squash merge does.

Line 1 in Listing 10 corresponds to line 8 in Listing 9, where the changes have been made and committed on the *dev* branch and the *master* branch is being checked out in preparation for the merge. This time, the *--squash* option is used with the merge command. This results in *foo.txt* appearing as a modified file on the *master* branch and a commit is required to complete the operation. It's not necessary to stage *foo.txt* because the modified object is already stored in the index.

Listing 10: The history after a squash commit shows only one commit.

```
1. C:\>git checkout master
2. C:\>git merge --squash dev
3. C:\>git status
```

```
4. On branch master
5. Changes to be committed:
6. (use "git reset HEAD <file>..." to unstage)
7. modified:   foo.txt
8. C:\>git commit -m "merge with dev"
9. C:\>git glog
10.e9b8dfa (HEAD -> master) merge with dev
11.ba73a77 initial commit
```

The two commits made on the *dev* branch were squashed into one modification to *foo.txt*, which in turn was committed to the *master* branch and now appears as a single commit with a new commit message (line 10 in Listing 10).

### Rebasing

Rebasing is a way of replaying a series of commits onto a different base. The typical use case for rebasing is when subsequent changes have been made to the branch on which another branch was originally based.

For example, a developer might be in the process of making changes on a development branch. The base of the development branch was the HEAD of the *master* branch at the time the branch was created, but now the *master* branch has evolved with two more commits. The developer would like to continue work on the development branch but wants to change its base to be the new HEAD of the *master* branch, thereby incorporating the changes on *master* that didn't exist when the development branch was created.

To better understand the concept of rebasing, I created an exercise based on an example found in the Git *rebase* command documentation at [file:///C:/Program Files/Git/mingw64/share/doc/git-doc/git-rebase.html](file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-rebase.html). Figure 16, which illustrates the status of a sample repository before and after rebasing, is taken directly from that documentation.

Assume the following history exists and the current branch is "topic":

```
    A---B---C topic
    /
D---E---F---G master
```

From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
    A'--B'--C' topic
    /
D---E---F---G master
```

**NOTE:** The latter form is just a short-hand of `git checkout topic` followed by `git rebase master`. When rebase exits `topic` will remain the checked-out branch.

Figure 16: A “before” and “after” example of rebasing, taken from the Git documentation examples.

In the code that follows to illustrate this example, I’ve substituted the more familiar *dev* in place of *topic* as the name of the branch. For that matter, the branch names are irrelevant to the example—it’s not at all important that one of them is the *master* branch.

In the diagrams in Figure 16, read left-to-right to go from oldest to newest. On the *master* branch D is the parent of E, E is the parent of F, and F is the parent of G while on the *dev* branch E is the parent of A, A is the parent of B, and B is the parent of C. In the log outputs in the code below, reading from bottom to top to equates to reading from left to right in the diagrams in Figure 16.

### *What history looks like before rebasing*

At the starting point, there are two branches in this repository, a *master* branch and a *dev* branch. There is one source code file, `foo.txt`. On the *master* branch it contains only modifications D and E. Modifications A, B, and C have been made and committed on the *dev* branch. The *dev* branch is the current branch.

```
C:\>git branch
  master
* dev
```

Reading from the bottom up, the log shows the commit sequence D-E-A-B-C. Commit E is the head of the *master* branch and commit C is the head of the *dev* branch.

Listing 11. Commits A, B, and C in the *dev* branch are based on commit E in *master*.

```
C:\>git glog
```

```
* fdaf158 (HEAD -> dev) C
* 4835389 B
* 1304d5a A
* b28d798 E
* 172fc10 D
```

Modifications F and G are now made and committed on the *master* branch. The log as seen from the *master* branch now shows the commit sequence D-E-F-G, and commit G is the head of that branch.

Listing 12: The tip of the *master* branch is now at commit G.

```
C:\>git checkout master
C:\>glog
* 800bf00 (HEAD -> master) G
* 3429ac7 F
* b28d798 E
* 172fc10 D
```

The *dev* branch does not include modifications F and G, which at this point exist only on the *master* branch. The state of the repository as shown in Listing 11 and Listing 12 corresponds to the upper diagram in Figure 16.

The developer now learns that modifications F and G have been made and committed on the *master* branch. She needs to keep working on the development branch, but wants to incorporate modifications F and G into her work. It would be nice if there were a way to grab modifications A, B, and C, pick them up as a group, and move them on top of the new tip of the *master* branch at commit G, thereby incorporating modifications F and G into her work on the *dev* branch.

That's what rebase does.

### *Performing the rebase*

Rebasing begins by checking out the branch whose commits you want to move (rebase), which in this example is the *dev* branch.

```
C:\>git checkout dev
```

Next, run the rebase command and specify *master* as the branch onto whose tip you want to rebase the commits in the *dev* branch. The tip of the *master* branch is commit G, which will become the new base of the *dev* branch.

```
C:\>git rebase master
```

Because of the way the contents of *foo.txt* were created for this example, Git displays a list of scary-looking messages about failure due to a merge conflict with instructions to either resolve the conflict and continue, or to abort (see Listing 13).

Listing 13: If Git encounters problems during a rebase it tells you how to continue or abort.

```
First, rewinding head to replay your work on top of it...
Applying: A
Using index info to reconstruct a base tree...
M       foo.txt
Falling back to patching base and 3-way merge...
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
error: Failed to merge in the changes.
hint: Use 'git am --show-current-patch' to see the failed patch
Patch failed at 0001 A
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
```

As with any merge conflict, Git has place conflict markers in `foo.txt`. Edit the file and resolve the conflict, in this case by arbitrarily keeping the modifications from both *master* and *dev*. Then stage the modified `foo.txt` file and tell the rebase to continue.

```
C:\mtgcode\gitrebase2>notepad foo.txt  && resolve the merge conflict

C:\mtgcode\gitrebase2>git add foo.txt  && stage the modified foo.txt file

C:\mtgcode\gitrebase2>git rebase --continue  && tell Git to continue the rebase
Applying: A
Applying: B
Using index info to reconstruct a base tree...
M       foo.txt
Falling back to patching base and 3-way merge...
Auto-merging foo.txt
Applying: C
Using index info to reconstruct a base tree...
M       foo.txt
Falling back to patching base and 3-way merge...
Auto-merging foo.txt
```

### *What history looks like after rebasing*

Listing 14 shows the log as seen from the *dev* branch after rebasing, showing that commits A, B, and C have now been applied on top of the tip of the *master* branch at commit G.

Note that commits A, B, and C now have different SHA-1 than they did in Listing 11. This corresponds to commits A', B', and C' in the lower diagram in Figure 16. The state of the repository now corresponds to that diagram.

Listing 14: After rebasing, commits A, B, and C are now based on commit G.

```
C:\>git glog  && the current branch is still dev
* 025ad0c (HEAD -> dev) C
* 4af92ff B
* 94fceac A
* 800bf00 (master) G
```

- \* 3429ac7 F
- \* b28d798 E
- \* 172fc10 D

This rebase is now complete. The developer can continue working on her dev branch knowing that the modifications in commits F and G are now incorporated into her work.

If you're interested in more information about rebasing, I encourage you to read the entire discussion and examples in the local Git documentation at [file:///C:/Program Files/Git/mingw64/share/doc/git-doc/git-rebase.html](file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-rebase.html).

## Remote repositories

### What is a remote repository?

A remote repository is any repository for the same project other than the developer's local repository.

An individual developer might use one or more remote repositories as a backup and/or to share changes among different machines. A team of developers might use a central remote repository to share changes with one another. For any use that involves sharing, the remote repository must be located on a resource accessible by everyone who needs it. For a team working in the same office, this could be the local area network file server. For a team working remotely, the central remote repository could be on a shared resource like Jungle Disk or on a service like GitHub or Bitbucket.

### Bare repositories

If you're initializing a repository in a remote location as a place to share code changes with other developers, or even if you intend to use the remote as a backup location only for yourself, you need to create the remote as a *bare repository*. A bare repository is a repository that has no working copy of the files. It doesn't need one because nobody will modify files directly in the bare repository anyway.

Use the *init* command with the *--bare* option to create a bare repository in Git. For example, a team might decide to share changes with each other using a bare repository located in a shared folders on a file server accessible to all team members. To create a bare repository in F:\myProject, specify that path as the target parameter in the *init* command.

```
C:\>git init --bare F:\myProject
```

Alternatively, you can navigate to F:\myProject and then simply *init --bare*.

```
F:\myProject>git init --bare
```

### Adding a remote repository

Before you can push from your local repository to a remote, you need to tell the local repository about the remote one by giving it a name and telling Git where to find it. This is

done by running Git's *remote* with the *add* option. Leaving branches out of the picture for the moment, the general syntax of this command is:

```
C:>git remote add <name> <url>
```

The name can be anything you want it to be as long as it conforms to Git's rules for reference names. The URL can be either an online URL or a local drive\path reference. The name *origin* is commonly used for a shared remote repository.

```
C:\myProject>git remote add origin F:\myProject
```

The local repository now has a link to a remote named *origin* for both fetching and pushing. You can run the remote command with no options to see the names of the remotes the local repository is aware of. Use the *-v* option (short for *--verbose*) to also see the URLs.

```
C:\myProject>git remote  
origin
```

```
C:\myProject>git remote -v  
origin F:\myProject (fetch)  
origin F:\myProject (push)
```

When working with a new bare repository for the first time, you need to let Git know about the branch you want to push. This will typically be the *master* branch for starters. Use the following syntax to for the initial push of the *master* branch to the *origin* remote.

```
C:\myProject>git push --set-upstream origin master
```

### Pushing to a remote repository

Now that a bare repository has been created in *F:\myProject* and has been added to the local repository as *origin*, you can begin pushing change from the local to the remote. The workflow is (a) make changes to the source code, (b) commit the changes to your local repository, and (c) push them to the remote.

Use the *push* command to push changes from your local repository to a remote. Git compares the state of your local repository to the state of the remote, identifies any commits in your local that are not yet on the remote, and if any are found it attempts to push those.

```
C:\myProject>notepad foo.txt && make some changes  
C:\myProject>git add foo.txt && stage the changes  
C:\myProject>git commit -m "made some changes" && commit to the local repository  
C:\myProject>git push origin && push the change to the remote named origin  
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 326 bytes | 163.00 KiB/s, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To F:\myProject
```

```
e9b8dfa..4cf1a66 master -> master
```

Pushing to the remote not only makes your changes available to other developers but also preserves them as a backup in case something happens to your local repository. The importance of pushing to the remote is only half-jokingly illustrated in the sign in Figure X.



Figure 17: Keep your priorities straight in an emergency. (Photo credit unknown.)

### Fetching and pulling from a remote repository

It's important to fetch changes from the remote before you begin making changes in your local. If there are any newer commits on the remote, pull them into your local repository and see what's changed before you start making your own modifications. (The difference between fetching and pulling is explained earlier in this paper.)

Make *git fetch* a habit before you start coding. A push can fail if the remote has newer commits on the same branch that are not yet in your local repository. If you forget to fetch, make modifications, commit, and then discover your push fails, you must either discard your local change or stash them before pulling. It's a lot easier to pull from the remote into a clean working copy because Git can usually create a merge commit with no merge conflicts.

If there are no newer commits in the remote, the fetch command returns no results. The current branch is assumed if a branch name is not specified.

```
C:\myProject>git fetch origin
```

If there are newer commits in the remote, pull them into your repository. Remember that in Git, pulling updates your working copy so be sure your working copy is clean before pulling.

```
C:\myProject>git pull origin
```

### Remote tracking branches

A remote tracking branch is a branch in your local repository that is linked to a branch in a remote repository. In the example above, the `--set-upstream` option created a remote tracking branch for `master` in the local repository. Remote branches are displayed by running the `branch` command with the `--all` option.

```
C:\GitTest>git branch --all
dev
* master
newdev
remotes/origin/master
```

The last line is how Git stores the reference to the `master` branch on the `origin` remote. Remote tracking branches are required in Git because Git pushes changes by branch, unlike Mercurial which by default pushed changes to all branches.

### Choosing a Git GUI

You don't need anything other than the command line to use Git, but as a developer accustomed to working with visual development tools like Visual Studio and Visual FoxPro you are most likely going to prefer using Git with a graphical user interface.

The Git website has a page dedicated to Git GUIs that run on Windows at <https://git-scm.com/download/gui/windows>. There were 24 of them the last time I looked, so you have a lot of choices. Following is a brief introduction to the ones I've used, all of which are free. Screenshots of the revision history of the sample myApp project are included for comparison.

- Git GUI – installed with Git
- TortoiseGit – from TortoiseGit and contributors
- SourceTree – from Atlassian (makers of Bitbucket)
- Git Extensions - a community project on GitHub
- GitHub Desktop - specifically for interacting with repos hosted on GitHub

### Git GUI

Git for Windows comes with a built-in graphical user interface called *Git GUI*. After installing Git, you should see an entry for Git GUI in your start menu. Git GUI is functional but not necessarily the one you'll like best, particularly since there are other, more robust GUIs available for free.

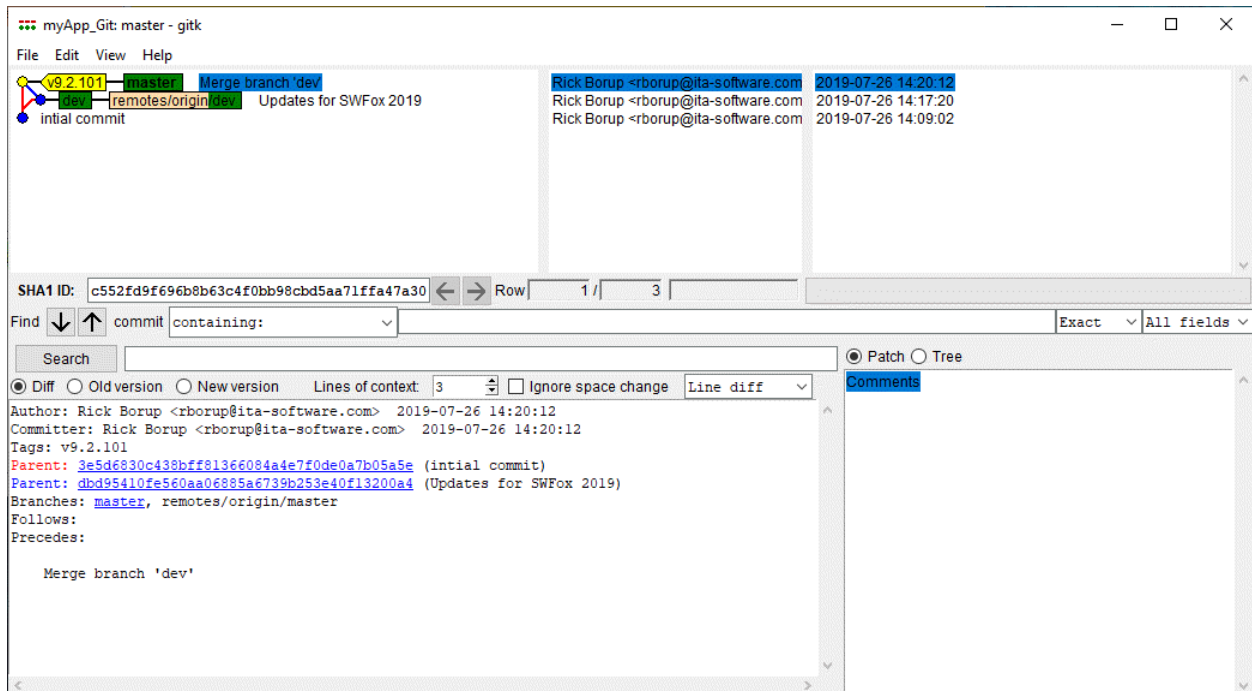


Figure 18: The revision history of myApp as shown in Git GUI.

## TortoiseGit

TortoiseGit is a free GUI for Git. It's similar to TortoiseHg for Mercurial, but if you're used to working with TortoiseHg be prepared for a big difference. TortoiseHg comes with the TortoiseHg Workbench, a container that presents the various individual components—the repository tree, the graphical history, the commit dialog, etc.—nicely arranged in a single window.

TortoiseGit does not have a workbench app. Instead, what you get are the individual pieces, each with its own window and each of which needs to be individually launched from the File Explorer context menu (see Figure 19).

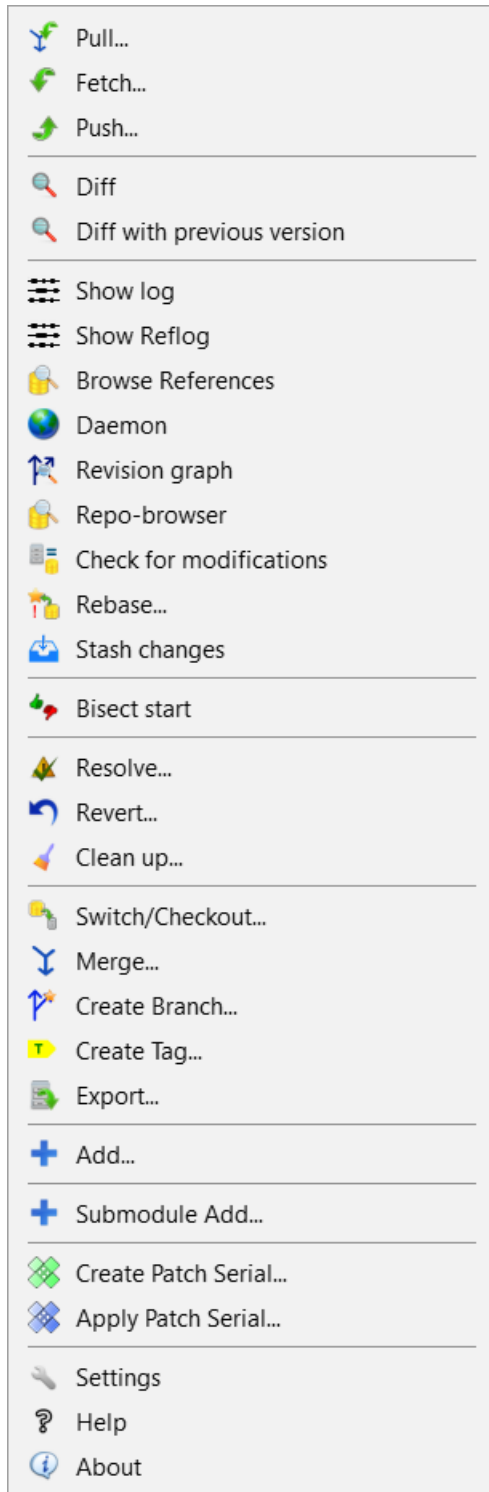


Figure 19: The TortoiseGit popup menu appears when you right-click on a file in a Git repository.

Selecting the *log* item on the TortoiseGit context menu brings up the revision history in graphical form along with other information, as shown in Figure 20.

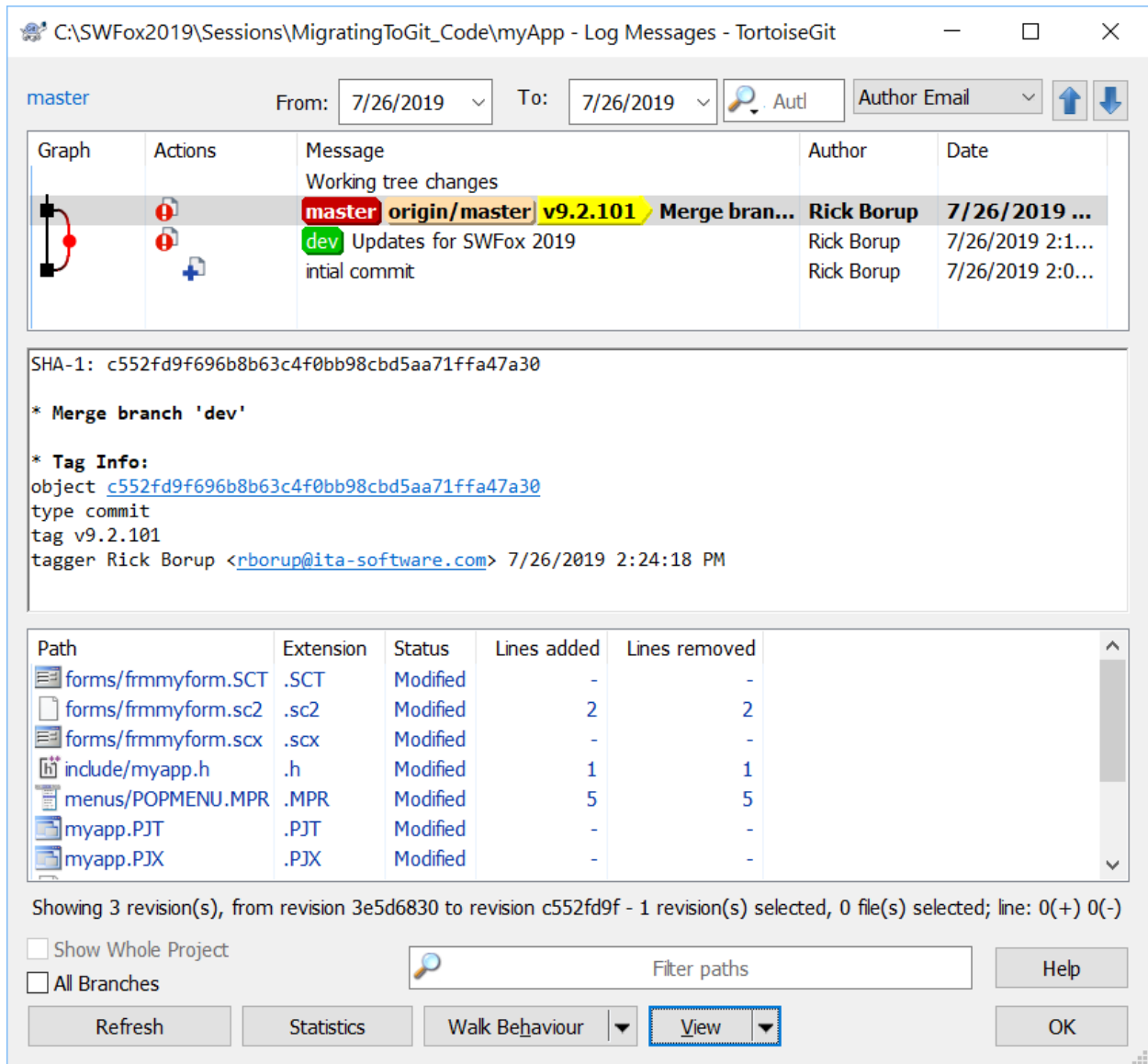


Figure 20: The revision history of myApp as shown in Tortoise Git.

TortoiseGit is available from <https://tortoisegit.org/>.

### Sourcetree

Sourcetree is a free GUI from Atlassian, makers of the Bitbucket cloud hosting service and many other tools including the acquisition of Trello in January of 2017. One big advantage of Sourcetree is that it works with both Git and Mercurial, making it the obvious GUI of choice if you use both and want to stick with a single interface.<sup>5</sup> Another advantage to Sourcetree is that it enables you to create custom actions. One custom action I find

<sup>5</sup> In August 2019 Atlassian announced they are dropping support for Mercurial repositories on the Bitbucket cloud hosting service effective June 1, 2020. It's unknown at this time whether that decision will also affect support for Mercurial in Sourcetree.

indispensable is the ability to run FoxBin2Prg on modified files directly from the context (popup) menu in the Sourcetree user interface.

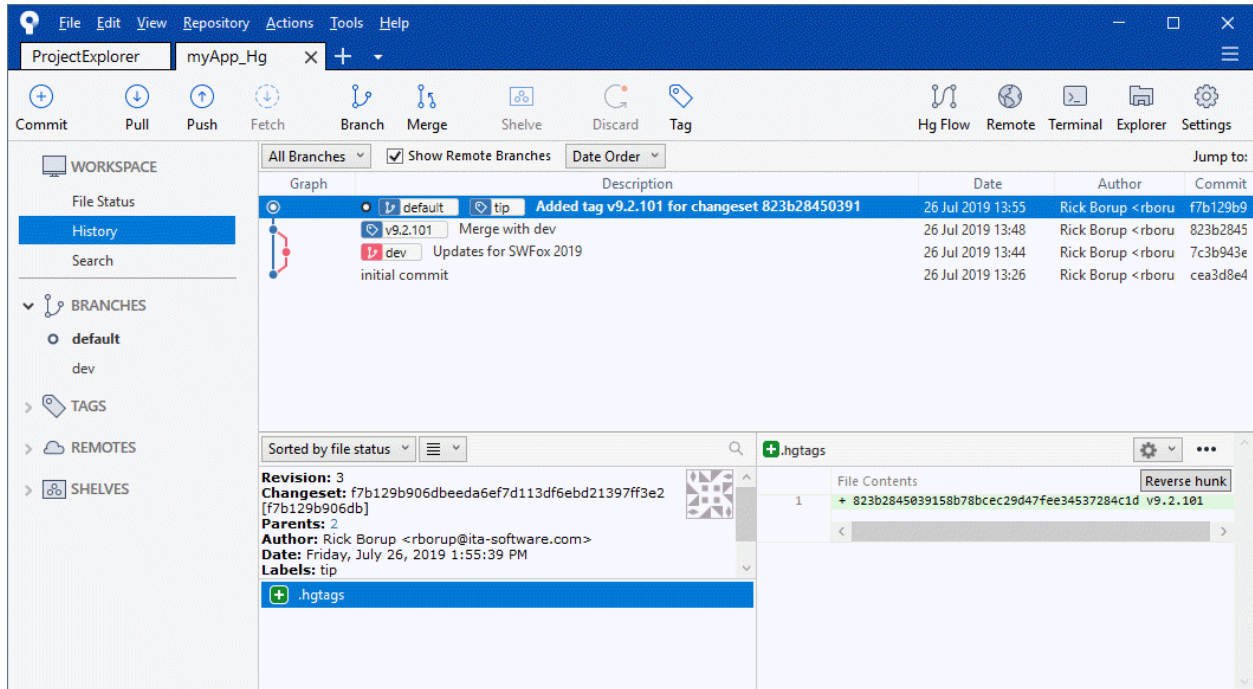


Figure 21: Sourcetree works with both Mercurial and Git repositories. The current tab is the revision history of the Mercurial repository for myApp. The Project Explorer tab is the Git repository for Doug Hennig's VFP Project Explorer repository on GitHub.

Sourcetree is available from <https://www.sourcetreeapp.com/>.

### Git Extensions

Git Extensions is a free GUI for Git. It's a community-based project originally on SourceForge and now hosted on GitHub. It has a nice, clean interface that I found enjoyable to use. Git Extensions is available as a standard MSI installer and also as a portable app. If you want to check it out without installing anything, you can just download and unzip the portable app to a thumb drive and run it from there.

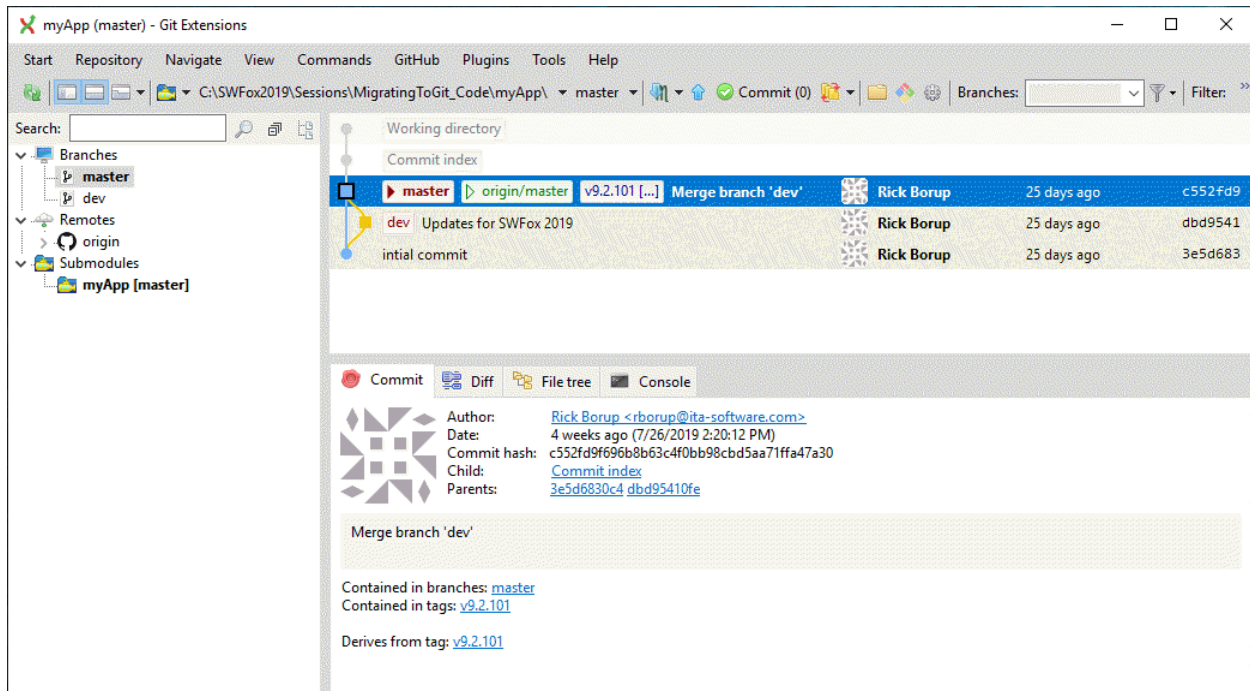


Figure 22: The revision history of myApp as shown in Git Extensions.

Git Extensions is available from the following locations:

<https://gitextensions.github.io/>

<https://github.com/gitextensions/gitextensions>

<https://sourceforge.net/projects/gitextensions> (still there, but points you to GitHub)

## GitHub Desktop

GitHub Desktop is a desktop app designed to simplify your interactions with Git and GitHub. It was first released in August of 2015, with version 2.0 following in June of 2019. GitHub Desktop is developed and maintained by the GitHub team. See the section entitled *The GitHub Desktop Tool* later in this paper for information on installing and using GitHub Desktop.

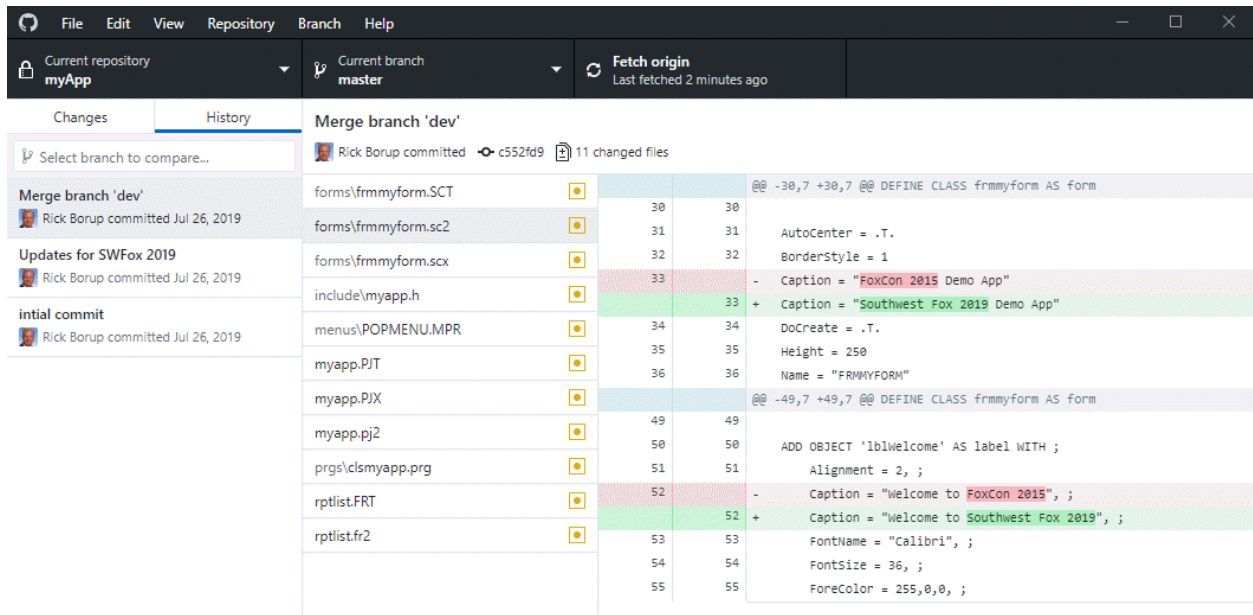


Figure 23: The revision history of myApp as shown in GitHub Desktop.

GitHub Desktop is available from <https://desktop.github.com/>.

### Others

Two others Git Gui's that seem to be popular, although not free, are GitKraken and SmartGit. I haven't used either of them so I can't comment on how they work or how they compare to the others mentioned here.

GitKraken is a Git client for Windows, Mac, and Linux. It's free for use with public repos hosted on GitHub.com, GitLab.com, or Bitbucket.org but it appears you need a paid license to use it with privately hosted repos. The features and prices for various plans are available at <https://www.gitkraken.com/pricing>. GitKraken is available from <https://www.gitkraken.com/>.

SmartGit is a product from the German company syntevo gmbH. It's free for non-commercial use, but a license costs \$79/user. SmartGit is available from <https://www.syntevo.com/smartgit/>.

## Migrating to Git from Mercurial

There are several reasons why you might want to consider migrating to Git from Mercurial.

- You like Mercurial and plan to continue using it, but feel you need to learn Git because a lot of other people are using it.
- You start a new project and decide to put it under Git while keeping your other projects under Mercurial.
- You inherit a project that's already under Git, or you join a team where Git is the standard.

- You decide Git is the future and want to migrate all your projects to it.
- You want to start using GitHub as a private remote repository for your projects.
- You want to contribute to an open source project hosted on GitHub.
- You want to publish your project on GitHub so others can see and contribute to it.

If you're starting a new project and want to use Git, you can simply initialize a Git repository in the project folder, set up your gitignore file, add your source code files, and you're off and running.

If you inherit a project or join a team that's working on a project already under Git version control, you would typically clone or otherwise acquire a copy of the repository for that project and begin using it as your local repo.

If you want to migrate an existing project from Mercurial to Git, or you want to collaborate on a Git project while still using Mercurial locally, your situation likely falls into one of the following three scenarios.

### Starting over from scratch with Git

Starting over from scratch with a new history in Git is the simplest although not necessarily the best choice.

If you can afford to cut ties with the Mercurial repository all together, abandoning all the history (although possibly retaining it for archival purposes), you can simply initialize a new Git repository and use Git for all future commits as if it were a brand-new project.

#### Initialize a Git repository in your project folder

Because they use different folder names for the repository—.git for Git and .hg for Mercurial—a Git repository and a Mercurial repository can peacefully co-exist in the same project folder. Although you certainly do not want to use both Git and Mercurial at the same time in the same project folder, you can leave the Mercurial repository in place while you make the switch to Git. Simply initialize a new Git repository in your existing project folder.

```
C:\myProject>git init
```

Do not use the *--bare* option. The source code files and other files already present in this folder comprise the working copy for the new Git repository, so it is not a bare repository.

#### Create the gitignore file

Create a gitignore file before adding files to the new Git repository for the initial commit. The easiest way is to simply copy the Mercurial hgignore file as gitignore and then add the entries in Listing 15 so Git will ignore the folder and files related to the Mercurial repository.

Listing 15: Add these entries to your gitignore file to ignore the folder and files related to Mercurial.

```
#-----  
# Ignore anything having to do with Mercurial  
#-----  
.hg/  
.hg*
```

If you don't want to keep the Mercurial repository in the project folder, you can move the .hg folder and its associated files such as .hgignore to some archive location. The archived Mercurial repository in effect becomes a bare remote repository for the project, preserving its state at the time you switched to Git. If necessary, you can still access the Mercurial repository for archival purposes even though it's no longer actively used.

### Add files and do the initial commit

Once the gitignore file is in place, you're ready to add files and do the initial commit. First run the *add* command with the *--dry-run* option.

```
C:\myProject\git add --all --dry-run
```

If the list of file to be added includes ones you don't want, tweak the gitignore file as necessary and repeat the dry run until the set of files to be added has been narrowed down to just the ones you want. Then stage the files with the *add* command and follow with the initial commit.

```
C:\myProject>git add --all  
C:\myProject>git commit -m "initial commit to Git"
```

### Using Mercurial to collaborate on a Git project

If you need to collaborate on a project that uses Git as the shared repository but you want to keep using Mercurial locally, the *Hg-Git* extension for Mercurial can help. This extension, or plugin, adds the ability for Mercurial to push to and pull from a Git repository.

### Installing the Hg-Git plugin for Mercurial

If you installed Mercurial by installing TortoiseHg, you already have the Hg-Git plugin. To enable it, mark the *hggit* check box in the Mercurial settings dialog as shown in Figure 24. This can be done either at the global level to make it available for all projects, or at the project (local repository) level to apply only to a specific project.

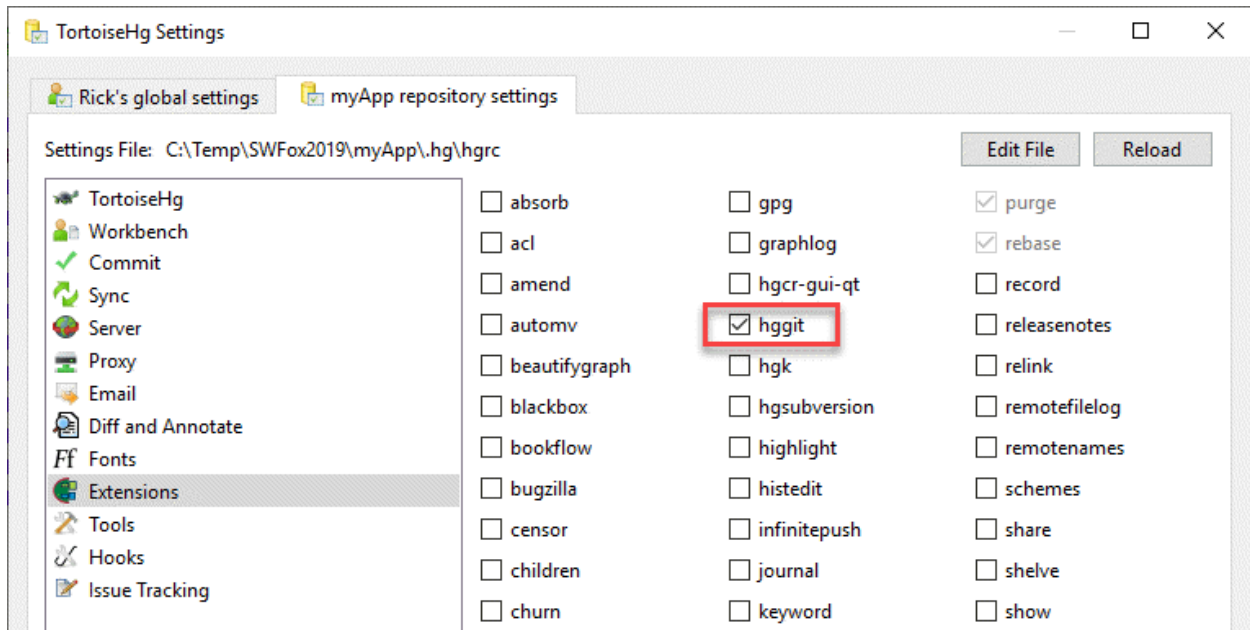


Figure 24: Enable the Hg-Git plugin by marking the hggit check box in TortoiseHg.

Enabling Hg-Git from TortoiseHg adds the entries in Listing 16 to your Mercurial configuration file. If you choose the global settings these entries are placed in your global configuration file at `%userprofile%\mercurial.ini`. For individual projects they are written to the `.hg\hgrc` file in the project folder.

Listing 16: Enable embedded Hg-Git by adding "hggit =" to the [extensions] section of your Mercurial configuration file.

```
[extensions]
hggit =
```

If you did not install TortoiseHg and still want to use Hg-Git, you can install it by following the instructions on the plugin's webpage at <http://hg-git-github.io> or you can clone it from its primary Git repository at <https://bitbucket.org/durin42/hg-git/src/default/>. The second choice is the easier of the two.

I have seen mentions online that the Hg-Git installed with TortoiseHg has not always worked. If you installed TortoiseHg and enabled the embedded Hg-Git plugin but find that it's not working, install Hg-Git using one of the two methods above and then enable it by modifying your global or project-level Mercurial configuration to point to the plugin's location on your local machine. For example, if you clone Hg-Git to `C:\Tools\Hg-Git`, use the configuration file entry is shown in Listing 17.

Listing 17: This tells Mercurial to look for the Hg-Git extension in `C:\Tools\Hg-Git\src\hggit`.

```
[extensions]
hggit = C:\Tools\Hg-Git\src\hggit
```

With the Hg-Git extension enabled, you can do any of the following:

- clone a Git repository into Mercurial
- push from a Mercurial repository to a Git repository
- pull from a Git repository into a Mercurial repository.

### Cloning a Git repository into Mercurial

The Hg-Git extension enables you to use the standard `hg clone` command to clone a Git repository and convert into a Mercurial repository. If the Git repository to be cloned is reachable with the `git://` protocol—for example, a public repository located on GitHub—you can clone it like this:

```
hg clone git://github.com/RickBorup/SWFox2019_myApp
```

Otherwise, if the Git repository to be cloned is located on a local resource accessible via the file system, you can simply reference it with the standard `drive\path\folder` syntax.

```
hg clone C:\SWFox2019_myApp
```

Either way, you end up with a clone whose history is now in Mercurial format.

### Using the clone source as a remote repository

The clone operation creates a project-level `hgrc` configuration file with a `[paths]` entry containing a backlink named `default` that points to the source of the clone.

```
[paths]
default = git://github.com/RickBorup/SWFox2019_myApp
```

or, if cloned from a local file system source

```
[paths]
default = C:\SWFox2019\Sessions\MigratingToGit_Code\Hg-Git\myApp_Git
```

The source of the clone is now a de-facto remote repository for your local copy of the project. This enable future interactions to specify `default` as the target for push or the source for pull commands.

```
hg push default
```

-or-

```
hg pull default
```

In my opinion, "default" is not a good name for this `[paths]` entry because `default` is also the name of the master branch in a Mercurial repository, so when you reference `default` in a Mercurial command it may not be obvious if it refers to the branch or to the remote repo. Changing the name in the `[paths]` entry to "origin" avoids this potential and is also consistent with Git's naming convention.

```
[paths]
origin = C:\SWFox2019\Sessions\MigratingToGit_Code\Hg-Git\myApp_Git
```

You can now work on the project using your local Mercurial repository and, when necessary, push to and pull from the Git repository.

One important caveat here: Git does not allow pushing to the same branch on a non-bare repository. If the Git repository from which you cloned is not a bare repository, then *hg push default* (or *hg push origin*, if you changed the [paths] name) fails with the message

```
abort: git remote error: refs/heads/master failed to update
```

Git disallows pushing to a non-bare repository in order to prevent overwriting other changes which may have occurred to files in the target's working copy. However, Git provides an option to override this behavior. Run the following command in the folder containing the non-bare repository to which you want to push.

```
git config --local receive.denyCurrentBranch updateInstead
```

This adds the following entry to the Git *config* file for the current project:

```
[receive]
  denyCurrentBranch = updateInstead
```

Shout-out to [Ciro Santilli](#) for posting this solution on [stackoverflow](#). Follow the link in the footnote<sup>6</sup> for a great example, discussion, and references to the Git documentation for this option.

The other, more common, solution to this problem is to create a new branch in your clone and push that branch upstream to the remote origin. To make it unique, the branch name would typically include the developer's name along with some identifying string, for example *rickborup\_bugfix\_123*. Pushing this branch is allowed because it creates a new branch on the remote origin and does not conflict with anything already there.

### Converting a Mercurial repository to Git

The Hg-Git extension for Mercurial can also be used to migrate a project from Mercurial to Git while preserving its history. If your Mercurial repository is on a local resource accessible from the file system, this is the way to do it.

If your Mercurial repository is accessible from an online resource like Bitbucket, you can use GitHub's import function to convert it to Git. An example of how to do that is included later in this paper, after GitHub has been introduced.

Using Hg-Git to convert a local repository from Mercurial to Git involves three steps.

---

<sup>6</sup> <https://stackoverflow.com/questions/1764380/how-to-push-to-a-non-bare-git-repository>

### Step 1: Create a bare Git repository outside the project folder

First, initialize a bare Git repository somewhere outside your project folder. This repository will be treated as a remote repository in step 2, so give it a name to help you remember its purpose. If you're working on a project called *myApp*, a good name for this folder might be *myApp\_GitRemote*.

```
C:\>git init --bare myApp_GitRemote
Initialized empty Git repository in C:/myApp_GitRemote/
```

### Step 2: Push the Mercurial history to the new Git repository

The next step is to push the history from your project's Mercurial repository to the new Git repository you created in step 1. There are a couple of things to do before pushing, though.

First, be sure the Mercurial working directory is clean and the *default* branch is up to date with any changes you want to be carried over to Git. Work in progress on branches other than the *default* branch does not get pushed to Git. If you have work in progress on a development branch, finish it and merge it back into the *default* branch before proceeding.

When you use Hg-git to push from Mercurial to Git, Mercurial bookmarks are converted to HEAD references in Git. You want Mercurial's *default* branch to become the *master* branch in Git, so the second step is to create a bookmark named *master* on the tip of the *default* branch before pushing. Similarly, create a bookmark on the head of any other Mercurial branches you want to be able to reference in Git. Assume the app is in C:\myApp\_Hg.

```
C:\>cd myApp_Hg
C:\ myApp_Hg>hg update default
C:\ myApp_Hg>hg bookmark "master" && bookmark the tip of default as master
C:\ myApp_Hg>hg bookmark --rev 1 "develop" && rev 1 is the tip of the dev branch
```

Listing 18 shows the Mercurial history after adding these bookmarks. Mercurial does not allow you to add a bookmark with the same name as an existing branch, so the bookmark on the *dev* branch is named *develop* instead.

Listing 18: The Mercurial history of the sample VFP application.

```
C:\myApp_Hg>hg log --graph
@ changeset: 3:b07836c6729c
| bookmark:  master
| tag:       tip
| user:     Rick Borup <rborup@ita-software.com>
| date:    Wed Aug 14 10:39:19 2019 -0500
| summary:  Added tag Release 9.2.101 for changeset 823b28450391
|
o  changeset: 2:823b28450391
|\ tag:       Release 9.2.101
| | parent:   0:cea3d8e4d494
| | parent:   1:7c3b943edc10
| | user:    Rick Borup <rborup@ita-software.com>
| | date:    Fri Jul 26 13:48:16 2019 -0500
| | summary: Merge with dev
```

```
| |
| o changeset: 1:7c3b943edc10
| / branch:    dev
|   bookmark:  develop
|   user:      Rick Borup <rborup@ita-software.com>
|   date:      Fri Jul 26 13:44:47 2019 -0500
|   summary:   Updates for SWFox 2019
|
o changeset: 0:cea3d8e4d494
  user:      Rick Borup <rborup@ita-software.com>
  date:      Fri Jul 26 13:26:11 2019 -0500
  summary:   initial commit
```

When ready, push the Mercurial history to the temporary Git remote repository.

```
C:\myApp_Hg>hg push ..\myApp_GitRemote && push to the bare Git repo created earlier
```

If errors are encountered, they will be noted in the output from the push command. Otherwise, the push was successful and the output will look something like this.:

```
pushing to ..\myApp_GitRemote
searching for changes
adding objects
added 4 commits with 12 trees and 31 blobs
```

Change to the myApp\_GitRemote folder and look at the Git log output to see what happened.

```
C:\myApp_GitRemote>git glog
* 7bbcccf (HEAD -> master) Added tag v9.2.101 for changeset 823b28450391
* 390ae60 (tag: v9.2.101) Merge with dev
| \
| * 8df56ff (develop) Updates for SWFox 2019
| /
* 697e6a9 initial commit
```

Note that the *master* bookmark from Mercurial became the HEAD reference on the *master* branch in Git, and the *develop* bookmark became a reference to the commit that was the head of the *dev* branch in Mercurial. There are two branch references in the new Git repository—*master* and *develop*.

```
C:\myApp_GitRemote>git branch
  develop
* master
```

Remember that this is a bare repository, so there's no working copy and you can't check out a branch at this point. The next step is to create a new local Git repository from this bare one so you can begin to work on the project in Git.

### Step 3: Create a new local Git repository from the remote one

The goal in the final step is to create a new folder, complete with history and a working copy, where you can continue working on the project but now using Git instead of Mercurial. There are a couple of ways to do this. One is to clone the remote created in step 2 into a new folder. The other is to manually create a new folder, initialize a Git repository in it, add a reference to the remote from step 2, and then pull. The repositories will differ a bit, but the working copy will be the same in both cases.

#### *Step 3 option 1: Clone the remote*

The basic syntax of the clone command is

```
git clone <source> <target>
```

The clone command creates the target folder, but the <target> parameter is optional. If omitted, the target folder is given the same name as the folder from which it is being cloned. Otherwise, it will have the name specified as the target.

To clone the temporary Git remote repository into a folder name `myAppGit`, run the clone command and specify `myAppGit` as the target folder.

```
C:\>git clone myApp_GitRemote myApp_Git
Cloning into 'myApp_Git'...
done.
```

When you clone any repository in Git, Git creates backlinks to the original repository. These can be seen by looking at the log output in the new cloned location.

Listing 19: The log output after cloning.

```
C:\myApp_Git>git log
* 7bbcccf (HEAD -> master, origin/master, origin/HEAD) Added tag v9.2.101 for
  changeset 823b28450391
* 390ae60 (tag: v9.2.101) Merge with dev
|\
| * 8df56ff (origin/develop) Updates for SWFox 2019
|/
* 697e6a9 initial commit
```

As noted in the documentation for Git's `clone` command, cloning creates and checks out an initial branch from the cloned repository's currently active branch while creating remote-tracking branches for the branches in the cloned repository. The currently active branch in `myApp_GitRemote` was the `master` branch when the clone was performed, so the clone has a `master` branch along with references to remote-tracking branches.

```
C:\myApp_Git>git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

The clone command also performs a checkout of the initial branch, so the clone contains the working copy of the files from the tip of the *master* branch. Figure 25 is a screenshot comparing the contents of the original Mercurial folder (on the left) with the new Git folder (on the right).

Name	Size	Modified	Name	Size	Modified
.hg	512,858	8/14/2019 11:19:52 AM	.git (h)		8/14/2019 11:39:56 AM
classes	0	7/26/2015 4:33:50 PM			
data	19,327	8/13/2019 5:11:29 PM			
files	158	8/13/2019 5:11:29 PM	files	158	8/14/2019 11:39:56 AM
forms	7,932	8/13/2019 5:11:26 PM	forms	7,932	8/14/2019 11:39:56 AM
icons	1,078	8/13/2019 5:11:26 PM	icons	1,078	8/14/2019 11:39:56 AM
include	218	8/13/2019 5:11:26 PM	include	218	8/14/2019 11:39:56 AM
menus	6,363	8/13/2019 5:11:26 PM	menus	6,363	8/14/2019 11:39:56 AM
prgs	12,510	8/13/2019 5:11:27 PM	prgs	12,510	8/14/2019 11:39:56 AM
.hgignore	3,594	7/26/2019 12:52:27 PM	.hgignore	3,594	8/14/2019 11:39:56 AM
.hgtags	50	7/26/2019 1:55:39 PM	.hgtags	51	8/14/2019 11:39:56 AM
history.txt	155	6/19/2019 11:51:08 AM	history.txt	155	8/14/2019 11:39:56 AM
myapp.EXE	20,347	7/26/2019 1:31:56 PM			
myapp.ini	26	7/26/2019 1:11:26 PM	myapp.ini	26	8/14/2019 11:39:56 AM
myapp.pj2	3,923	7/26/2019 1:16:29 PM	myapp.pj2	3,923	8/14/2019 11:39:56 AM
myapp.pjt	433,191	7/26/2019 1:48:05 PM	myapp.pjt	433,191	8/14/2019 11:39:56 AM
myapp.pjx	11,853	7/26/2019 1:48:05 PM	myapp.pjx	11,853	8/14/2019 11:39:56 AM
readme.txt	155	6/19/2019 11:51:54 AM	readme.txt	155	8/14/2019 11:39:56 AM
rptlist.fr2	24,473	7/26/2019 1:48:05 PM	rptlist.fr2	24,473	8/14/2019 11:39:56 AM
rptlist.frt	4,224	7/26/2019 1:48:05 PM	rptlist.frt	4,224	8/14/2019 11:39:56 AM
rptlist.frx	7,506	7/26/2019 1:30:55 PM	rptlist.frx	7,506	8/14/2019 11:39:56 AM

Figure 25: The left side is the original project folder with a Mercurial repository. The right side is the same project in a new folder with a Git repository.

There are some differences worth noting. The `hgignore` and `dot hgtags` files from the source folder also exist on the right side because they were tracked files in the Mercurial repository and were therefore re-created in the new Git project folder when the project was cloned. The executable file `myapp.EXE` doesn't exist on the right side because it was not tracked in the Mercurial repository. The `classes` and `data` folders in the Mercurial project folder do not exist in the cloned folder because the `classes` folder is empty and the contents of the `data` folder were not tracked in Mercurial. Finally, all the files on the right side have a newer datetime stamp, although their content is the same as their counterpart on the left.

One thing that doesn't exist yet in the new project folder is a `gitignore` file. You could create one from scratch, but since Git's `gitignore` file uses essentially the same syntax as Mercurial's `hgignore` you can simply rename `hgignore` to `gitignore` and then edit it if necessary. The `dot hgtags` file can be deleted.

The Git history can be compared to the Mercurial history to see how well it came across. Figure 26 is a screenshot comparing the revision graph for both histories as displayed in Sourcetree, which can handle both Mercurial and Git repositories. Notice that unlike TortoiseHg, Sourcetree does not display revision numbers for Mercurial repositories.

## Migrating to Git from Mercurial

---

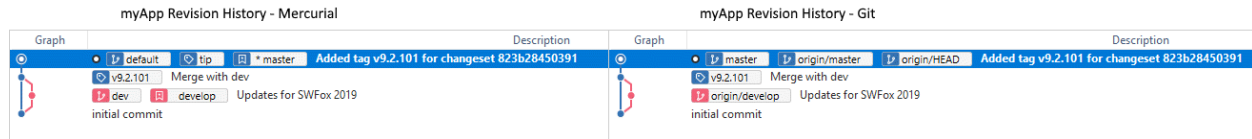


Figure 26: The revision history in Git, on the right, is identical to the original revision history in Mercurial.

Although not shown in the image, the commits in the Git repository have different commit IDs than their counterparts in the original Mercurial repository.

### *Step 3 option 2: Create a new folder and pull*

As an alternative to cloning, you can create a new folder, initialize a Git repository in it, add a remote reference to the repository created in step 2, and then pull from it.

Create a new folder and initialize a Git repository in it.

```
C:\>md myApp_Git
C:\>cd myApp_Git
C:\myApp_Git>git init
Initialized empty Git repository in C:/myAppGit/.git/
```

Add the remote as a tracked repository. The `--tags` option tells Git fetch (pull) to import tags from the remote repository.

```
C:\myApp_Git>git remote add --tags origin ../myApp_GitRemote
```

Verify the remote was added as expected.

```
C:\myApp_Git>git remote -v
origin C:\myApp_GitRemote (fetch)
origin C:\myApp_GitRemote (push)
```

Pull from the remote repository. A branch name is required—in this case, it's *master*.

```
C:\myApp_Git>git pull origin master
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 47 (delta 15), reused 0 (delta 0)
Unpacking objects: 100% (47/47), done.
From C:\myApp_GitRemote
 * branch          master       -> FETCH_HEAD
 * [new tag]       v9.2.101     -> v9.2.101
 * [new branch]    master       -> origin/master
```

Check the log output to see what was pulled.

Listing 20: The log output after pulling.

```
C:\myApp_Git>git log
* 7bbcccf (HEAD -> master, origin/master) Added tag v9.2.101 for changeset
  823b28450391
* 390ae60 (tag: v9.2.101) Merge with dev
```

```
| \  
| * 8df56ff (origin/develop) Updates for SWFox 2019  
| /  
* 697e6a9 initial commit
```

Compare the log output in Listing 19, which shows the results after cloning, to that in Listing 20, which shows the results after pulling. The working copy—i.e., the content of the source code and other tracked files—is the same in both cases.

### Wrapping up

Regardless of whether you used option 1 or option 2, you now have a choice about what to do with intermediate remote repository from step 2.

If you don't want or need it anymore you can simply delete it. If you do, also remove the references to the remote-tracking branches in your new copy. Git's *remote* command has a syntax for doing that.

```
C:\>git remote remove origin
```

Alternatively, you can continue to use it as a remote repository, even if only as a backup.

```
C:\>git push origin && push future changes you make to this project
```

If it's not already in a shareable location and you want to use it to collaborate with other developers, you can move it to a new location that's accessible to everyone who needs it. After moving it, change the URL of remote references in your own copy to point to the new location. Git's *remote* command has a syntax for doing that, too.

```
C:\>move myApp_GitRemote F:\Shared && move the folder to F:\Shared\myApp_GitRemote  
C:\>git remote set-url origin F:\Shared\myApp_GitRemote && change the URL for origin
```

## GitHub

GitHub is a commercial service for hosting and sharing Git repositories. It was founded circa 2007 and acquired by Microsoft in 2018. With tens of millions of users and upwards of a hundred million repositories, GitHub is arguably the largest source code hosting service in the world. One competitor is Bitbucket from Atlassian. Bitbucket began as a hosting service for Mercurial repositories. It added the ability to host Git repositories in 2011 and now supports both.<sup>7</sup>

---

<sup>7</sup> In August 2019 Atlassian announced it is ending support for Mercurial on the Bitbucket cloud effective June 1, 2020. At that time all Mercurial repositories on Bitbucket will be removed. See the full announcement at <https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket>.

### Getting started with GitHub

GitHub offers both free and paid accounts. Repositories hosted on GitHub can be either public or private. Private repositories used to be available only on paid accounts, but free plans now offer unlimited private repositories.

Public repositories are discoverable and accessible for cloning or forking by anyone. You make a repository public when you want to share your code with the world. Private repositories are not discoverable and are accessible only to those to whom permission has been granted by the repository's owner. You mark a repository private when you want to keep it to yourself or to share code with a team but not with the entire world. It's easy to switch a repository from private to public and vice versa, so you're not stuck with whatever you choose first.

You don't have to use GitHub to use Git, but you do have to use Git to use GitHub.

### Setting up a GitHub account

Setting up a GitHub account for yourself is easy and free. Go to <https://github.com>, enter a username, an email address, and a password and you're off and running. Repository names on GitHub are prefixed with the account owner's username, so choose one that you want the world to see. Capitalization is respected—for example, mine is *rickborup* because I didn't use camel case, while Doug Hennig's is *DougHennig* because he did. Usernames can be changed but doing so changes the repository's URL and may also have other unexpected consequences.<sup>8</sup>

### Using GitHub from a browser

Figure 27 is an annotated screenshot of the page you first see in your browser after signing into GitHub.

---

<sup>8</sup> See <https://help.github.com/en/articles/changing-your-github-username>

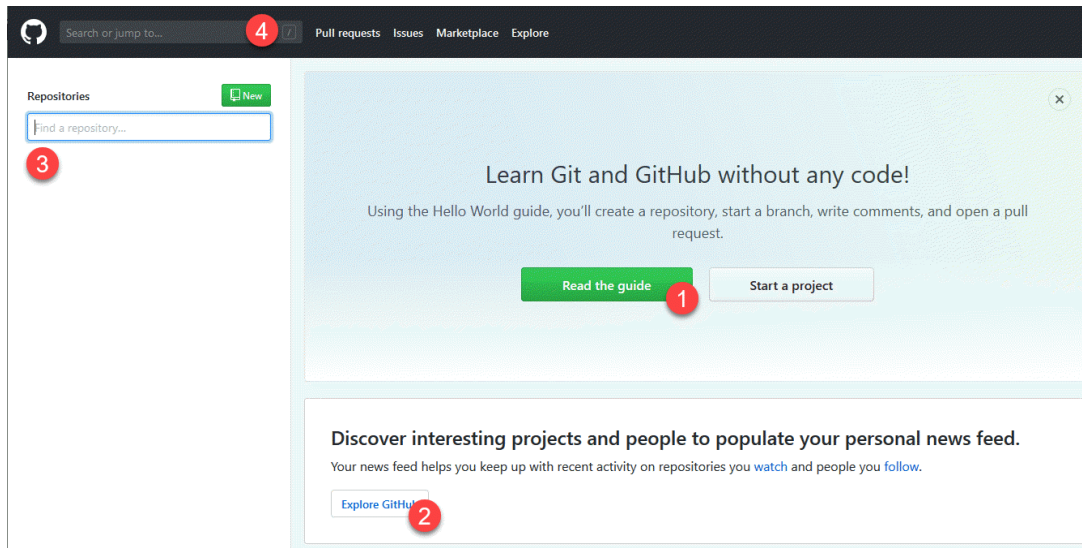


Figure 27: The GitHub home page.

The GitHub website has many resources to help you learn and explore. If you're unfamiliar with GitHub, take a few minutes to read the Guide—the big green button at ① in Figure 27—and work through the tutorial demonstrating GitHub's basic functionality. The Explore GitHub link at ② takes you to a page featuring trending repositories, popular apps, etc.

If you have already created any repositories in your GitHub account, they are listed in the left-hand column at ③ in Figure 27. That area is blank if you haven't created any repositories yet. The search field at the top of that area, under the *Repositories* label, enables you to search for existing repositories in your own account, which is convenient if you have a lot of them and don't want to scan through a long list.

The search area at ④ in Figure 27 is where you can search all the public repositories on GitHub. Try searching for "VFP", "VFPX", "FoxPro", or "Visual FoxPro" to see how it works.

### Creating a new repository on GitHub

The first step toward putting a project on GitHub is to create a new repository for it. Click the green *New* button at the upper left in Figure 27 and enter a name for your repository. Repositories on GitHub are prefixed with the account owner's username, which is filled in for you based on the account you're signed into. A description is optional. Choose whether you want this repository to be public or private. Skip the last part if you're going to import an existing repository, as we will in this example.

When you're ready, click the *Create repository* button at the bottom. Figure 28 shows the creation of a new private repository for the VFP demo app named *myApp* under the *SWFoxDemo* account.

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?  
[Import a repository.](#)

---

Owner Repository name \*

SWFoxDemo ▾ / myApp ✓

Great repository names are short and memorable. Need inspiration? How about [curly-spork?](#)

Description (optional)

A VFP demo app for Southwest Fox 2019

---

**Public**  
Anyone can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

---

Skip this step if you're importing an existing repository.

**Initialize this repository with a README**  
This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾ | Add a license: None ▾ ⓘ

---

**Create repository**

Figure 28: Fill in the information to create a new repository on GitHub.

GitHub encourages you to add a `readme.md` (Markdown) file and a license. These are important for public repositories but less so for private ones.

### Adding an existing repository to GitHub

Figure 29 shows the page that comes up after you create a new repository on GitHub, containing suggestions on how to proceed in one of several ways.

Although you can create and edit files directly through the GitHub interface, most of the time you're probably going to want to add an existing project that's already under Git version control on your local machine. This is done by adding a remote reference to your local Git repository that points to the new GitHub repository, and then pushing the local to the GitHub remote. Each new GitHub repository automatically gets a unique URL based on its account owner's username plus the repository name, which you can use to reference it as needed. In this example the URL is <https://github.com/SWFoxDemo/myApp.git>. Note the dot git extension on the URL.

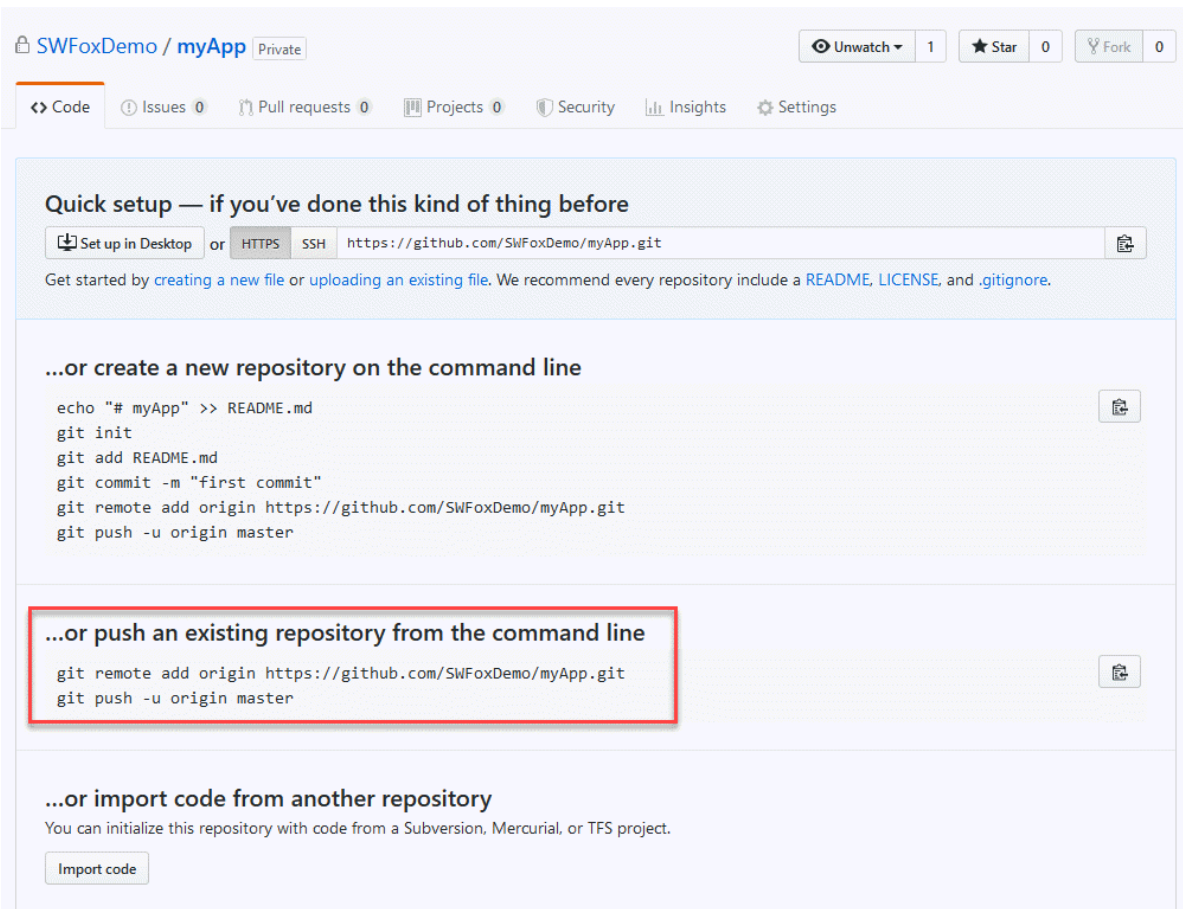


Figure 29: Push an existing repository from your local to the new GitHub repository.

Because we're starting with a local Git repository, this example uses the "push an existing repository" option. In Figure 29, you may have noticed the "import code" option below it which mentions Mercurial as one of the types GitHub can import. There's a full example of importing a Mercurial repository into GitHub later in this paper.

In a command window on your local machine, and working from the folder containing your project and its Git repository, run the two commands outlined in red in Figure 29. You don't have to use *origin* as the name for this remote, but it's customary to do so. Listing 21 shows the output from running these two commands for the sample *myApp* project.

Listing 21: Add a remote reference to the new GitHub repository and push your local repo to it.

```
C:\myApp>git remote add origin https://github.com/SWFoxDemo/myApp.git
C:\myApp>git push -u origin master
Enumerating objects: 47, done.
Counting objects: 100% (47/47), done.
Delta compression using up to 4 threads
Compressing objects: 100% (42/42), done.
Writing objects: 100% (47/47), 164.13 KiB | 3.35 MiB/s, done.
Total 47 (delta 15), reused 0 (delta 0)
remote: Resolving deltas: 100% (15/15), done.
```

```
To https://github.com/SWFoxDemo/myApp.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

The `-u` option on the push command is the short form of `--set-upstream`. It tells Git to add an upstream tracking reference for the branch being pushed, which in this case is the *master* branch. The remote tracking branch is now included in the list of all branches in the local repository.

```
C:\myApp>git branch --all
dev
* master
remotes/origin/master
```

Displaying the remotes for the local repository shows that the links to GitHub are now stored for fetch and push.

```
C:\myApp>git remote -v
origin https://github.com/SWFoxDemo/myApp.git (fetch)
origin https://github.com/SWFoxDemo/myApp.git (push)
```

Figure 30 shows the GitHub Code page for this repository after pushing. GitHub is now set up to be a remote repository for this project. As with any other remote repository, future modifications committed to the local repository can be pushed to GitHub and future modifications committed by others (or by yourself from a different machine) can be pulled.

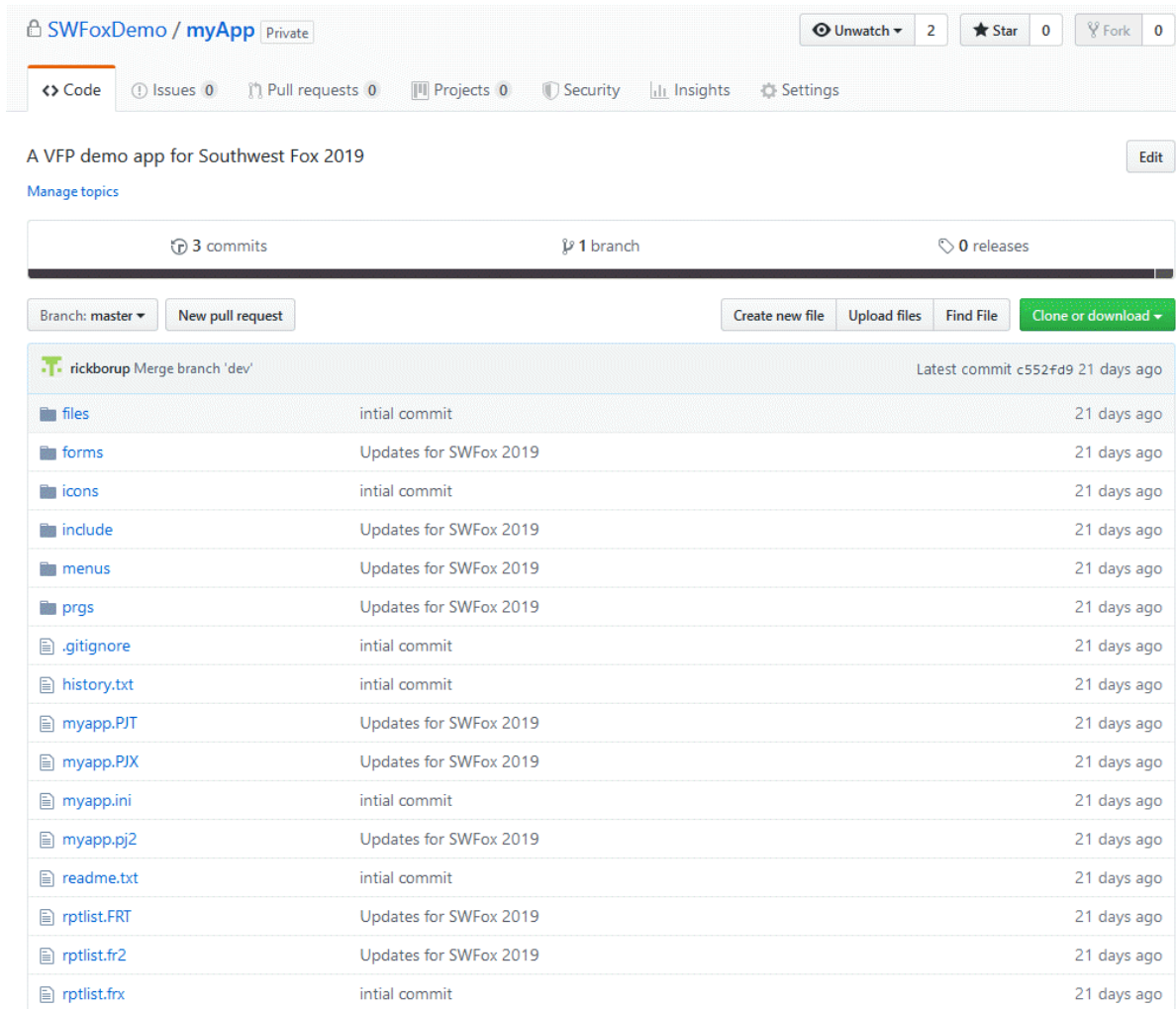


Figure 30: The source code files and commit history are shown on the Code page in GitHub.

## Including a readme file

It's customary to include a readme file for your project on GitHub. The purpose of the readme file is to provide up-front information about your project to other people. It can be as simple as one or two lines describing the project or it can be a lengthy document that includes instructions for using and/or contributing to the project.

Put the readme file in the project's root folder. GitHub recognizes both plain text files (readme.txt) and markdown files (readme.md). if a readme file is present, GitHub automatically displays its contents at the bottom of the project's main page. A readme.md file takes precedence over a readme.txt file if both are present.

## Settings

The Settings page for a GitHub project provides access to options where you can do things like renaming the project, adding branches and setting branch protection rules, adding collaborators, and changing the visibility of a project from private to public or vice versa.

Every project has a row of tabs that appear at the top of the project's home page, under the project name. The Settings tab is the one at the far right. If you don't see a Settings tab it means you don't have the right permissions to edit them for that project. Typically, only the owner of the account can access the project's settings.

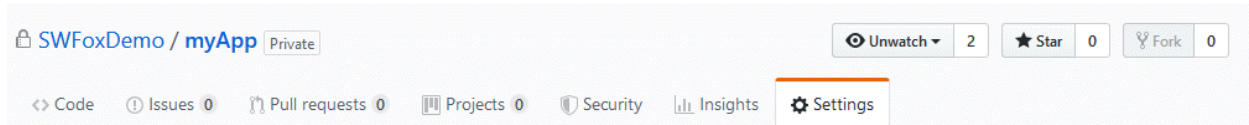


Figure 31: The Settings tab is present if you have the right permissions to access them.

### *Changing the visibility of a project*

Careful thought should be given to changing the visibility of a project because it can cause problems if other people are involved. In most cases you'll want a private repository to remain private and a public repository to remain public. However, as is the case with the sample projects for this conference session, there may be times when you want to change its visibility. Here's how to do it.

Click the Settings tab and choose Options from the list on the left. Scroll down to the bottom of the page until you see the Danger Zone. The options in this area are labelled as dangerous because they are potentially destructive.

## Danger Zone

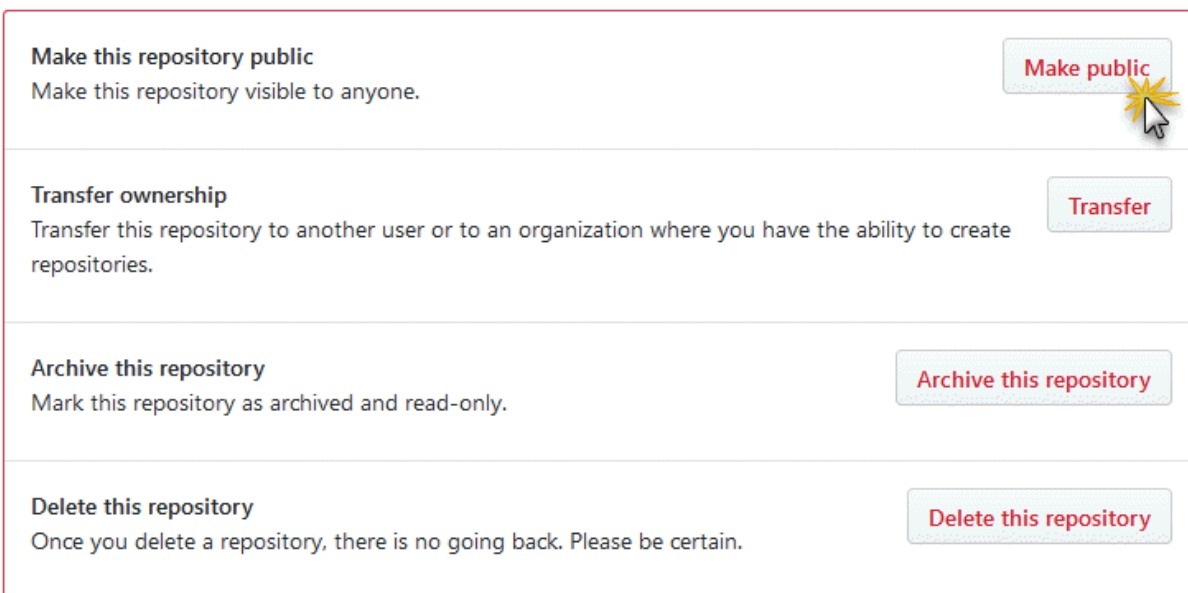


Figure 32: You can change the visibility of a project from private to public or vice versa.

The first choice in the Danger Zone enables you to change the visibility of the project. If the repository is currently private the choice is to make it public. If it's already public, the choice is to make it private. Click the *Make public* button to initiate the change.

After clicking the button, GitHub presents a screen of information to be sure you understand the implications of the change you are about to make and prompts you to confirm the action.



Figure 33: GitHub warns you about the implications of what you are about to do and prompts you to confirm the action.

The “I understand” button at the bottom in Figure 33 becomes enabled after you type in the name of the project in the text box above it. Click that button to complete the action.

Follow the same process to change the project back from public to private.

### GitHub for the solo developer

Although designed primarily for teams, GitHub is useful for the solo developer, too. Pushing your commits to GitHub creates an offsite, online backup from which you can recover your work if necessary—an important part of an effective disaster recovery plan. GitHub is typically set up as the *origin* remote; pushing changes to `origin/master` (or `origin/<another branch name>`) after committing is part of the standard Git workflow.

Even as a solo developer you may work on the same project from multiple machines. In that case you can also use GitHub as the intermediary to synchronize your work among your various computers. For example, I work on a desktop machine while I’m at the office and on a laptop while I’m at home or on the road. My daily workflow is to push to GitHub from the office and the end of the day and then pull from GitHub when I fire up the laptop in the evening. If I do any work on the laptop, I push those commits to GitHub and then pull them onto my office machine the next morning.

### GitHub for teams

GitHub is designed to facilitate collaboration. It's used by teams who are collaborating on the same project and for open source projects where there can potentially be an unlimited number of contributors.

There are a couple of ways teams can use GitHub to work together on a project, even using a private repository. One way is to add people as collaborators. The other is to use pull requests.

### Adding collaborators

If you want to enable other people to work on your project by directly pushing to and pulling from GitHub, you can add them as collaborators. You must be the owner of the repository or otherwise have the appropriate permissions to add collaborators to a project.

Choose *Collaborators* from the Settings page, enter then GitHub username of the person you want to invite, and click *Add collaborator*. This sends an email invitation to the potential collaborator, which that person then accepts to complete the process. Figure 34 shows that GitHub user *rickborup* (me) has been added as a collaborator on the myApp project by the owner of the SWFoxDemo/myApp repository (also me).

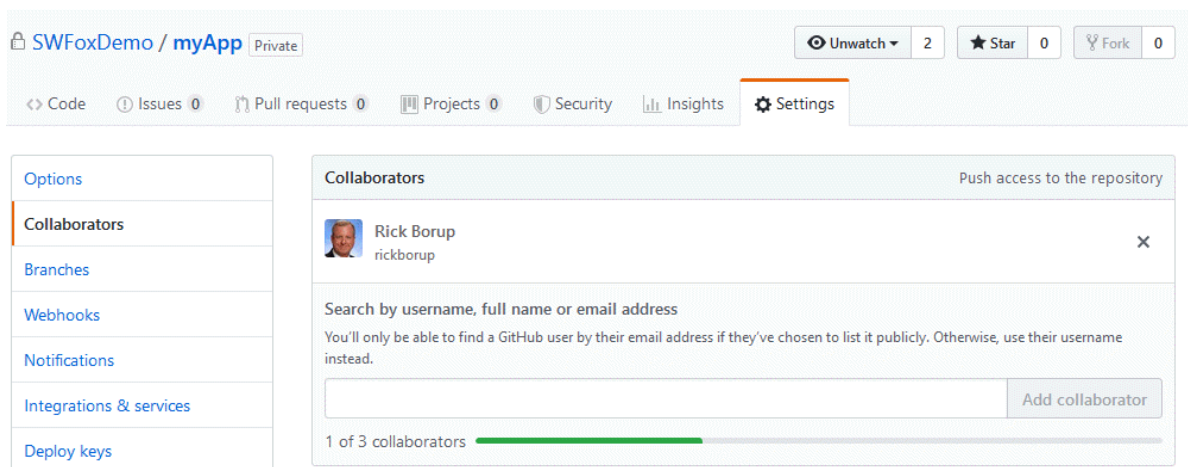


Figure 34: Rick Borup has been added as a collaborator on the myApp project under the SWFoxDemo GitHub account.

Collaborators can directly push commits to the GitHub repo. This may or may not be the ideal workflow for any given team or project. Naturally some coordination is necessary to avoid stepping on one another's work in the course of development. One approach would be for each person to create their own branch and to commit their work to that branch. Someone then needs to be put in charge of reviewing and approving changes before merging them into *master*.

Collaborators have other permissions as well. See <https://help.github.com/en/articles/permission-levels-for-a-user-account-repository> for complete information.

An alternative to adding collaborators who can push commits directly to GitHub is to use pull requests, but first a quick look at cloning vs forking.

### **Cloning vs forking**

Cloning is an integral feature of Git. It creates a local copy of another repository. To create a clone, all you need is Git installed on your computer and access to the repository you want to clone. You can create a clone of any repository you have access to—it does not have to be on GitHub.

Forking, on the other hand, is a feature of GitHub. Forking creates a copy of someone else's GitHub repository under your own GitHub account. To create a fork, you need to have a GitHub account of your own plus access to the GitHub repository you want to fork. The forked repository contains a link back to the repository from which it was forked, which is called the *upstream* remote

After forking a project onto your account on GitHub, you clone the fork into some folder on your local machine so you can work on the project. At this point it's like any other project—you can edit files, create new files, create a new branch, commit changes to your local repository, and periodically push them to *origin*, which is your forked repository on GitHub. When you're ready to share your changes, the links in your forked repository on GitHub enable you to create a pull request, which is a notice to the owner of the upstream repository requesting a review and hopefully acceptance of your work.

The important thing to remember is that you cannot create a pull request from a clone. You can only create a pull request from a fork.

### **Using pull requests**

Pull requests are a way for developers working on a forked project to ask the owner or maintainer of the upstream project to incorporate their changes. In this workflow, commits are not pushed directly to GitHub. Instead, once a modification has been made and committed to a developer's local repository, that developer uses GitHub to initiate a pull request. The owner or maintainer of the GitHub repository receives notification that a pull request has been created and can review the request, exchange comments and possibly ask for more changes or improvements, and ultimately accept or reject the request. If the request is accepted, the repository owner pulls it into the repository. In a team environment, pull requests can be assigned to another person for review and comment.

If the GitHub repository is public, anyone can clone or fork it and create a pull request. If the repository is private, only collaborators can do so.

### *Forking a repository*

To fork a repository, sign in to your GitHub account, browse to the home page of the repository you want to fork, and click the *Fork* button.

As an example, looking at the list of files for the SWFoxDemo/myApp project in Figure 30, I notice it does not have a readme.md file so I set out to create one. While signed in to GitHub

on my own account, I browse to the home page of the myApp project in the SWFoxDemo account and create a fork in my own account (see Figure 35).

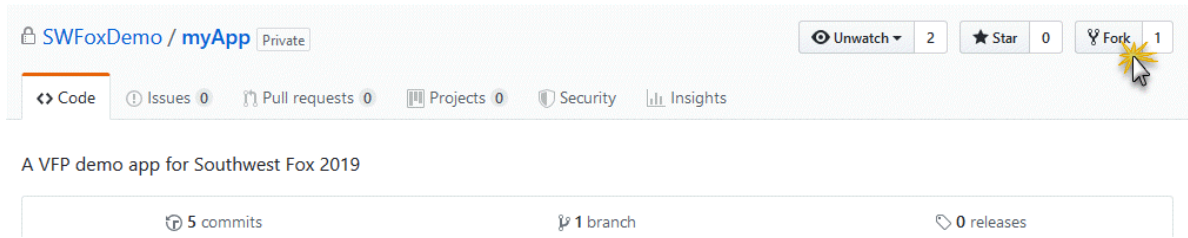


Figure 35: Click the *Fork* button on the home page of the GitHub repository you want to fork.

My GitHub account now shows the fork as a repository named rickborup/myApp with the notation that it is a fork of SWFoxDemo/myApp (see Figure 36).

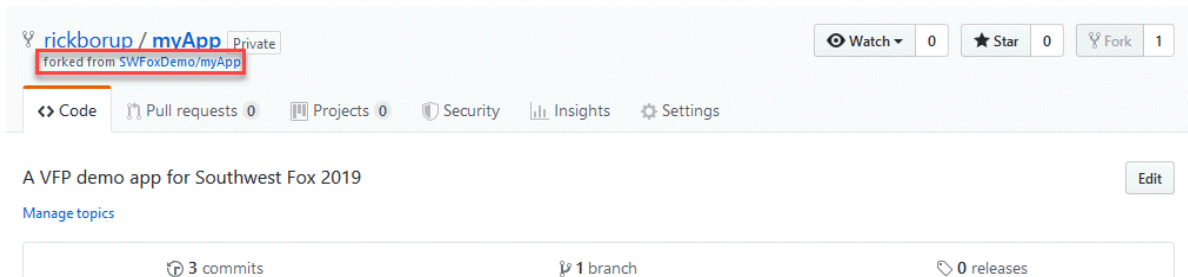


Figure 36: The myApp project now appears in my rickborup GitHub account as a fork of the SWFoxDemo/myApp project.

### *Making changes on the fork*

If I were doing to do any real work on this project the next step would be to clone the forked repository into a folder on my local machine. The green *Clone or download* button has the URL for cloning, as shown in Figure 37.

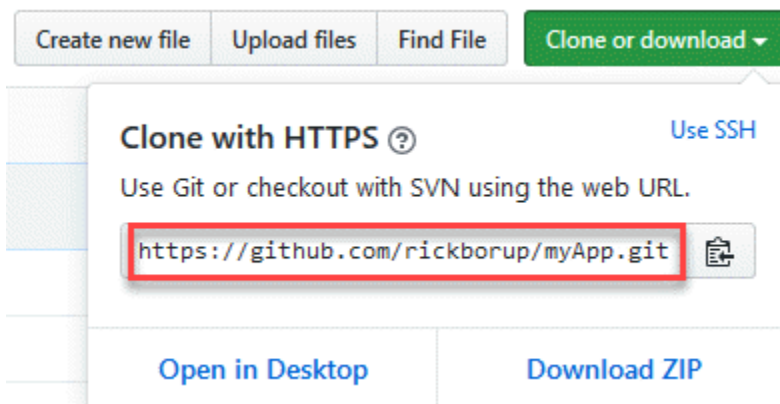


Figure 37: The Clone or download dropdown displays the URL for cloning.

In this case, since all I want to do is add a new file, I can skip the cloning step and simply use the *Create new file* button. Enter a name for the file in the space provided and enter the desired contents for the file (Figure 38).

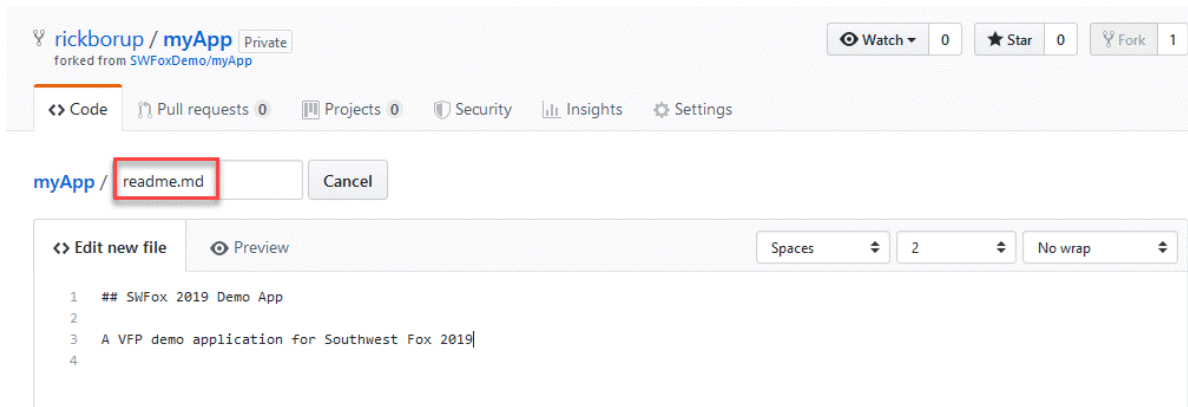


Figure 38: Create the readme.md file directly in GitHub.

Scroll down to the bottom on the page to see the commit area. While it's customary to create a new branch for every change, this is a simple case so I'll commit directly to *master*. Remember this commit is on the forked copy on my GitHub account, not on the upstream account. Click the green *Commit new file* button as shown in Figure 39.

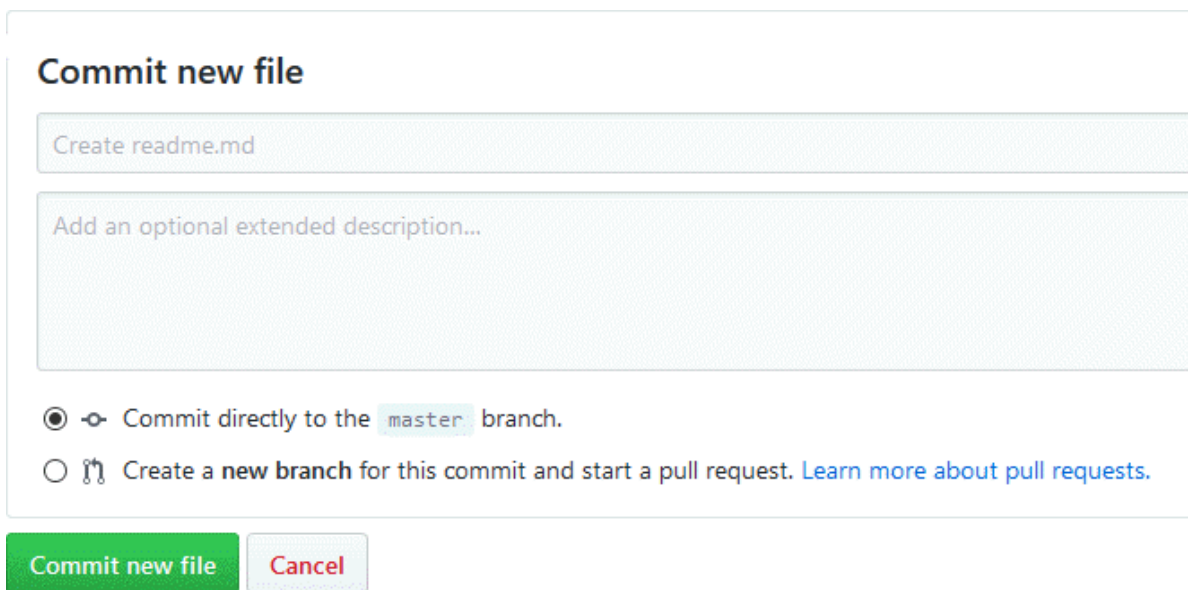


Figure 39: In this case the new file is committed directly to the *master* branch.

### *Creating a pull request*

The readme.md file is now part of the forked repository on my GitHub account. The next step is to create a pull request as a notice to the owner of the upstream repository (SWFoxDemo/myApp) that someone has suggested changes to be pulled. To initiate the pull request, click the *Pull request* button as shown in Figure 40.

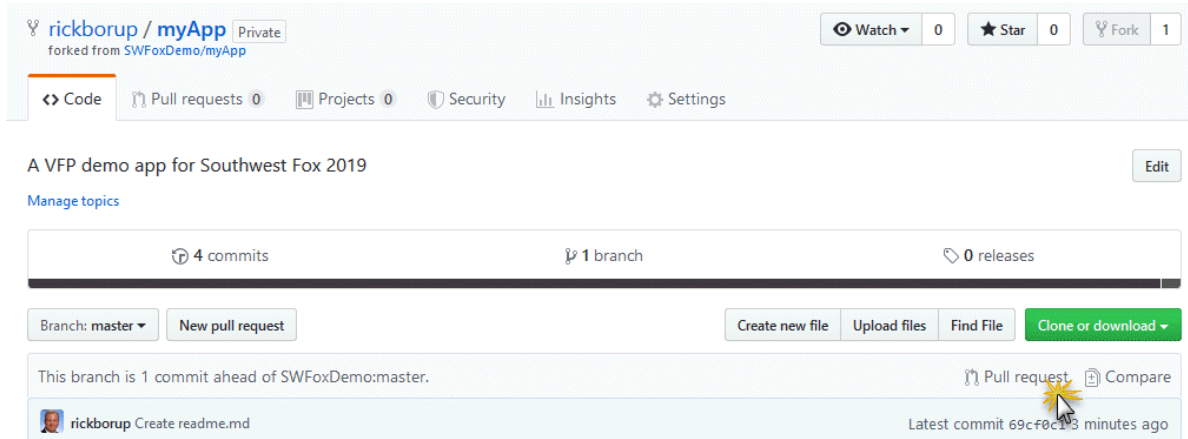


Figure 40: Click the *Pull request* button to start a pull request.

GitHub compares the status of the forked repository (rickborup/myApp) to the upstream repository (SWFoxDemo/myApp) and displays a page where you can take further action. In this case, the changes can be merged. Click the green *Create pull request* button shown in Figure 41.

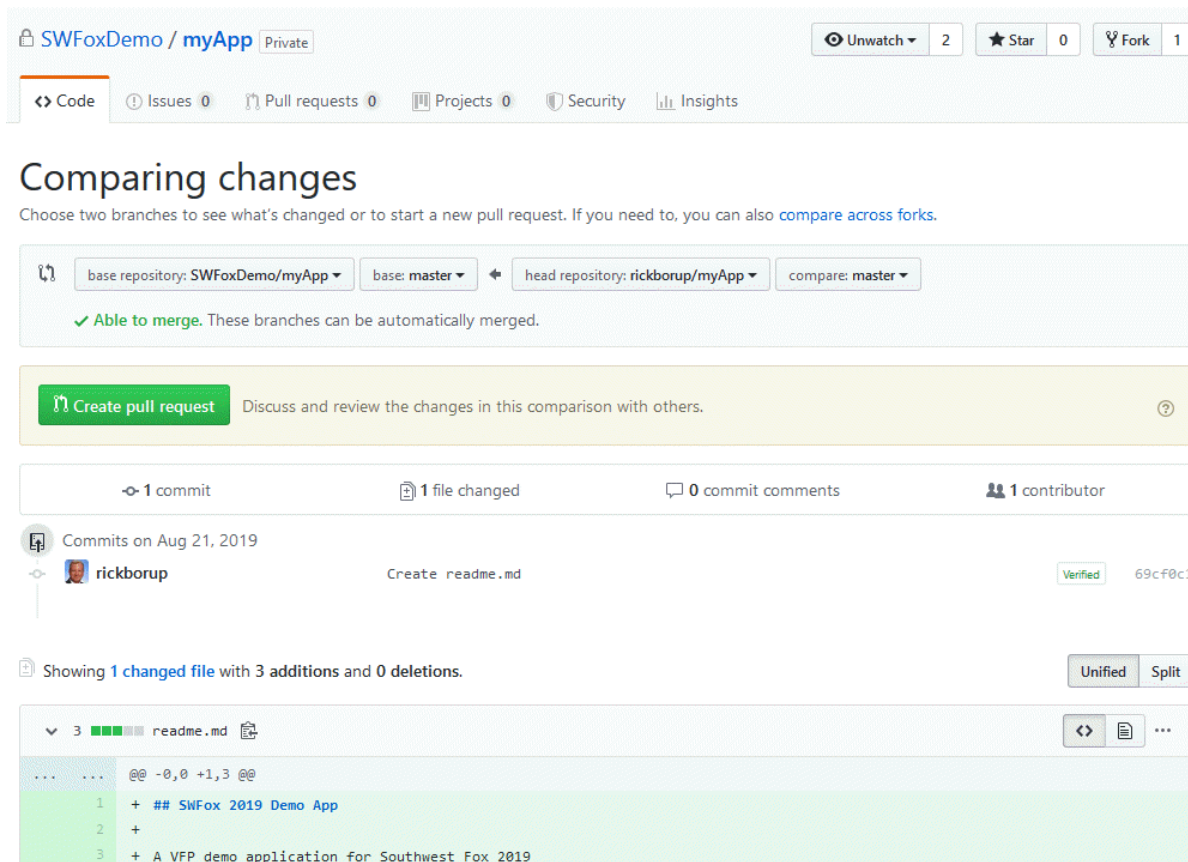


Figure 41: Click the green *Create pull request* button.

The last step is to enter any optional comments you want to provide to the maintainer of the upstream repository. This is a good place to explain what you did and why. Click the green *Create pull request* button to complete the process.

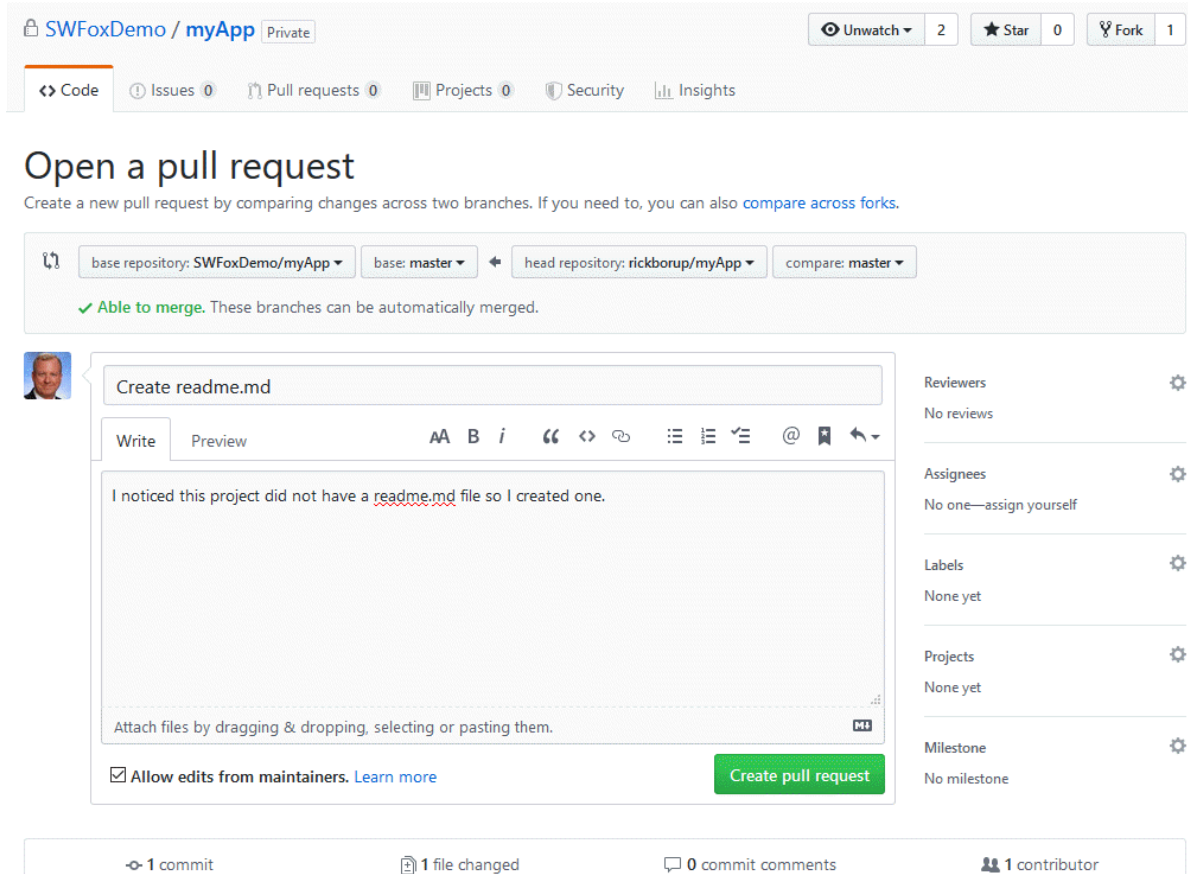


Figure 42: The final step in creating a pull request is to include comments to explain your work to the maintainer of the upstream repository.

### *Reviewing and accepting the pull request*

The owner of the upstream SWFoxDemo/myApp repository receives an email saying a pull request has been created. The email contains information about the request including its commit message and the comments included by the requestor. The home page of that repo also shows there is one pull request pending (see Figure 43).

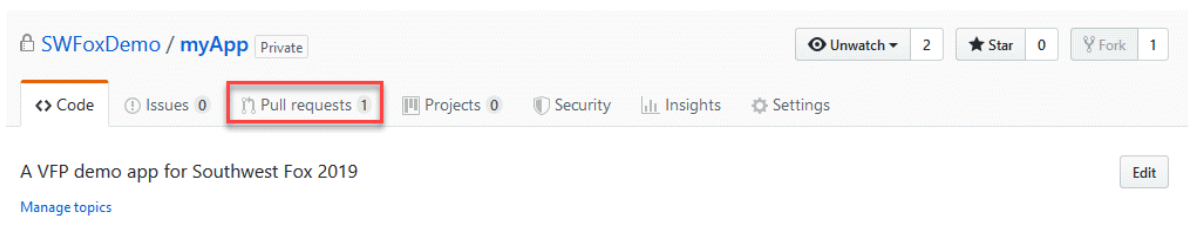


Figure 43: One pull request is pending.

Clicking the Pull requests button opens a page listing the open pull requests. Hovering the mouse over the title of the pull request turns it into a link and displays a popup box with details (see Figure 44). Click the title of the pull request to proceed.

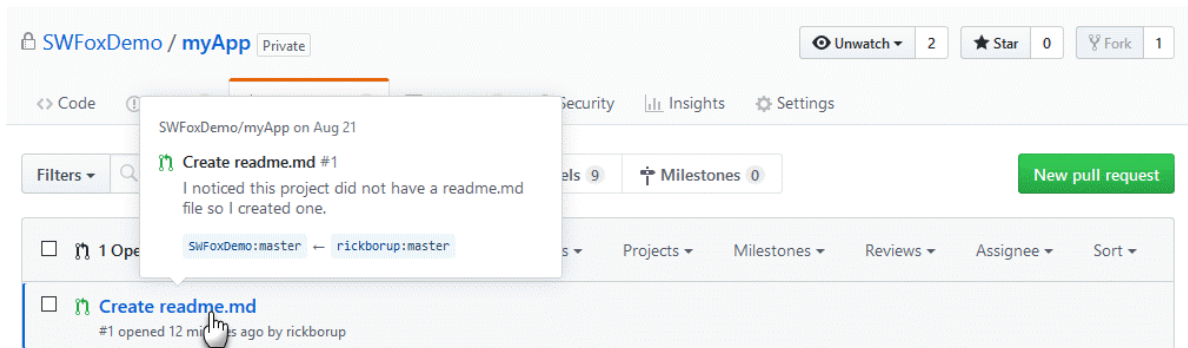


Figure 44: GitHub displays a list of the open pull requests.

The page for the pull request is shown in Figure 45. The person responsible for reviewing the pull request can either accept it as is or send comments back to the requestor.

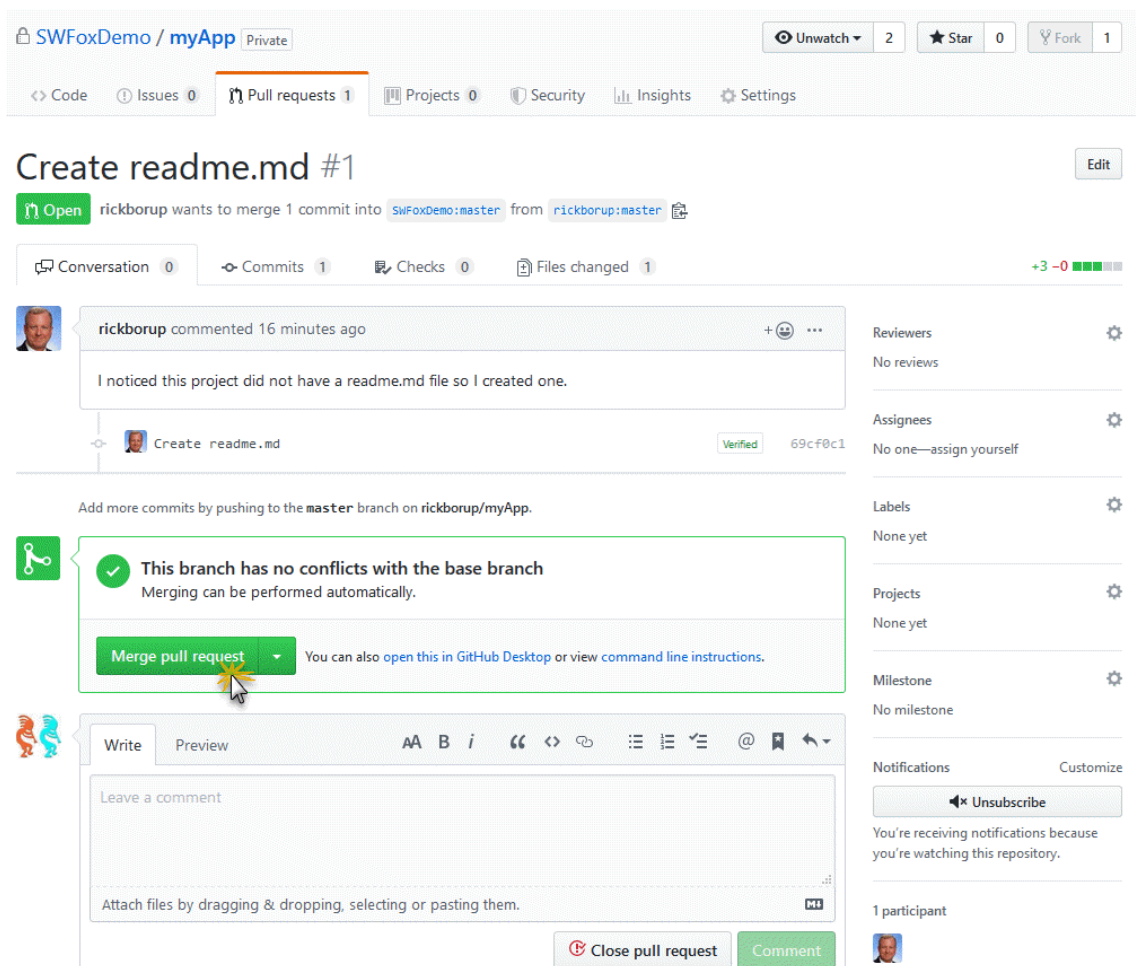


Figure 45: The reviewer / maintainer can accept the pull request and merge it into the repository, or write comments back to the requestor and close the request without accepting.

In this example the reviewer determines the pull request is OK and no comments or revisions are necessary. There is only one commit in this pull request, so GitHub determines that a merge commit is in order. The dropdown arrow on the green *Merge pull request* button shows the other available choices for the merge, as shown in Figure 46.

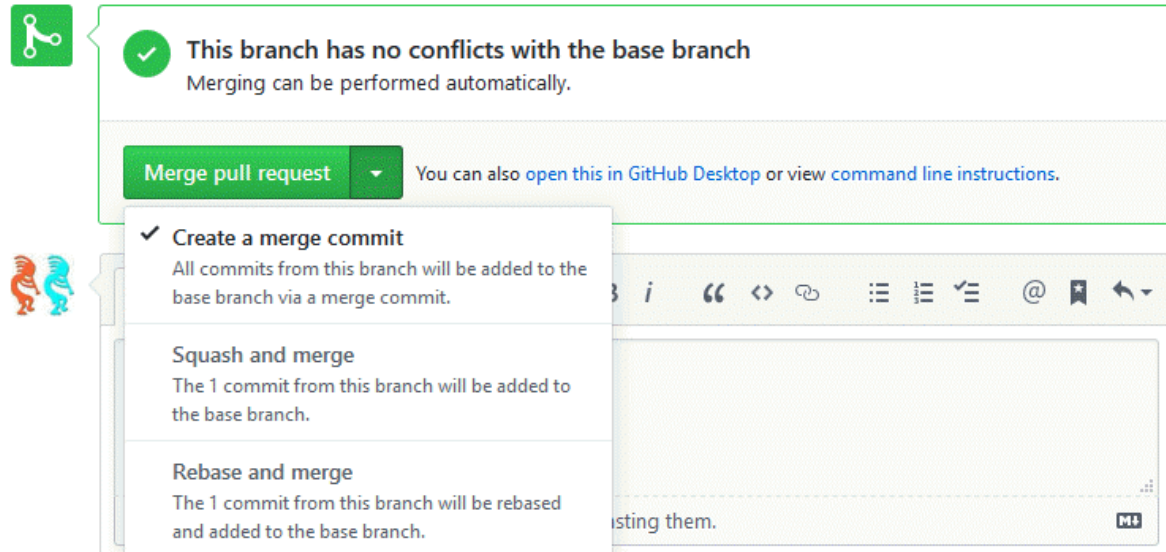


Figure 46: A pull request can be accepted into the project as a merge commit, a squash merge, or a rebase merge.

Click the green *Merge pull request* button to continue. There is a confirmation step (not shown), after which the merge is performed. The list of files on the Code tab of the home page for the project now includes the `readme.md` file. Because it's now the `readme.md` file for the project, GitHub displays its contents at the bottom of the page.

The screenshot shows a GitHub pull request interface. At the top, it indicates 5 commits, 1 branch, and 0 releases. Below this, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find File', and 'Clone or download'. The main content area shows a merge pull request #1 from rickborup/master. A table of files is displayed, with 'readme.md' highlighted in red. The table lists files with their commit messages and timestamps. Below the table, the content of 'readme.md' is shown, including the title 'SWFox 2019 Demo App' and the description 'A VFP demo application for Southwest Fox 2019'.

File	Commit Message	Timestamp
files	initial commit	26 days ago
myapp.pj2	Updates for SWFox 2019	26 days ago
readme.md	Create readme.md	42 minutes ago
readme.txt	initial commit	26 days ago
rptlist.FRT	Updates for SWFox 2019	26 days ago
rptlist.fr2	Updates for SWFox 2019	26 days ago
rptlist.frx	initial commit	26 days ago

**readme.md**

### SWFox 2019 Demo App

A VFP demo application for Southwest Fox 2019

Figure 47: The readme.md file is now part of the upstream project and GitHub displays its contents at the bottom of the page. (Portions of the list above have been cut out to save space.)

### *Wrapping up the pull request*

The pull request process is now complete. You can either delete the forked repository from your account or keep it active for potential future collaboration on the project. If you're actively contributing to the project, keep the fork but remember to check for changes in the upstream repository and pull them into your forked repository before doing any more work. On the other hand, there's no real downside to deleting it because a new fork can be created later if needed.


### **Exercise: creating a pull request using only your own GitHub account**

To make it as realistic as possible, the example above involved two GitHub accounts—my own personal account and the SWFoxDemo account—but you don't need two accounts to experiment with pull requests. Here's an exercise you can do.

Sign in to your GitHub account in a browser and create a new private repository.

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner:  rickborup / Repository name:  ✓

Great repository names are short and memorable. Need inspiration? How about [musical-eureka?](#)

Description (optional):

Public  
Anyone can see this repository. You choose who can commit.

Private  
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer.

Add .gitignore:  | Add a license:  ⓘ


[Create repository](#) 

Figure 48: Create a new private repository.

Click the *creating a new file* link in the *Getting started* sentence.

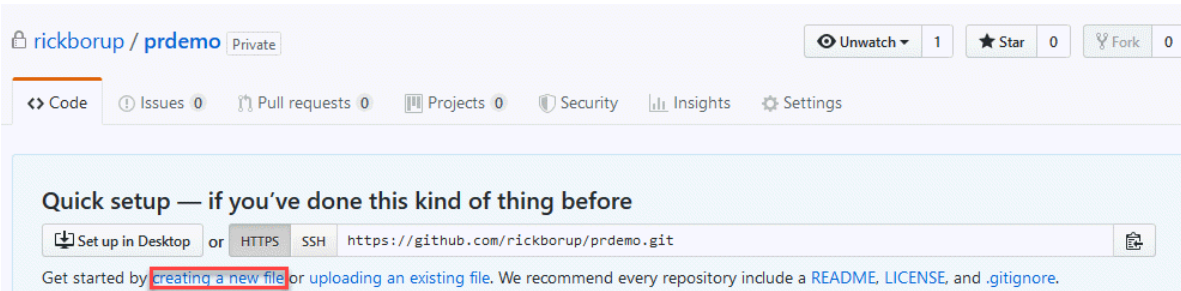


Figure 49: Create a new file.

Let's pretend this is a VFP project and create a file named `main.prg` to display "GitHub rocks!" on the VFP main window. Click the green *Commit new file* button at the bottom of the page (not shown in Figure 50).

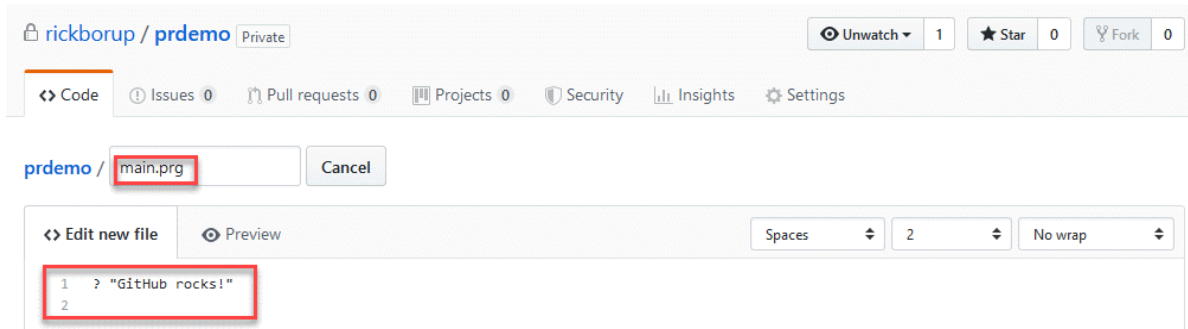


Figure 50: Create a new file named `main.prg`.

GitHub enables you to modify files directly from the browser. The name of each file listed on the Code page is a link to that file. Click the *main.prg* file name.

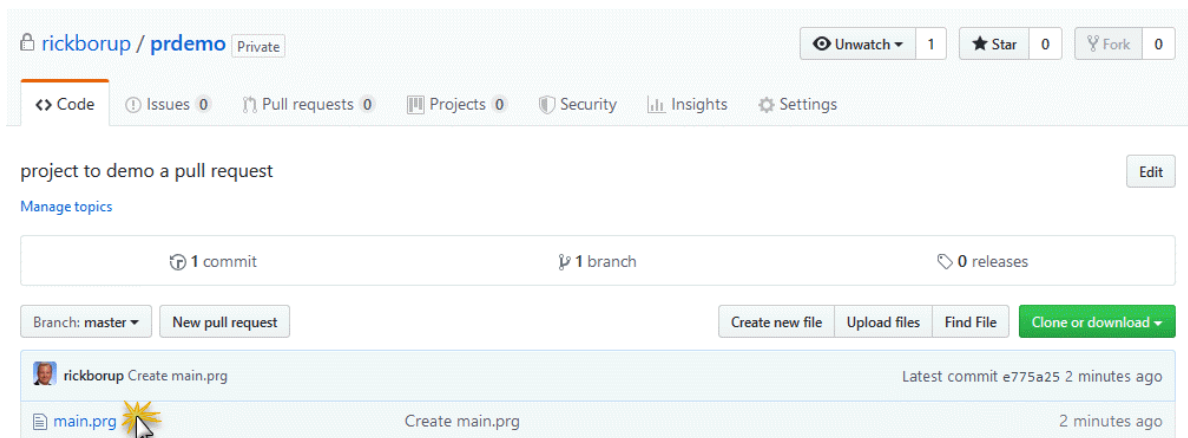


Figure 51: Click the file name.

Click the pencil icon to edit the file.

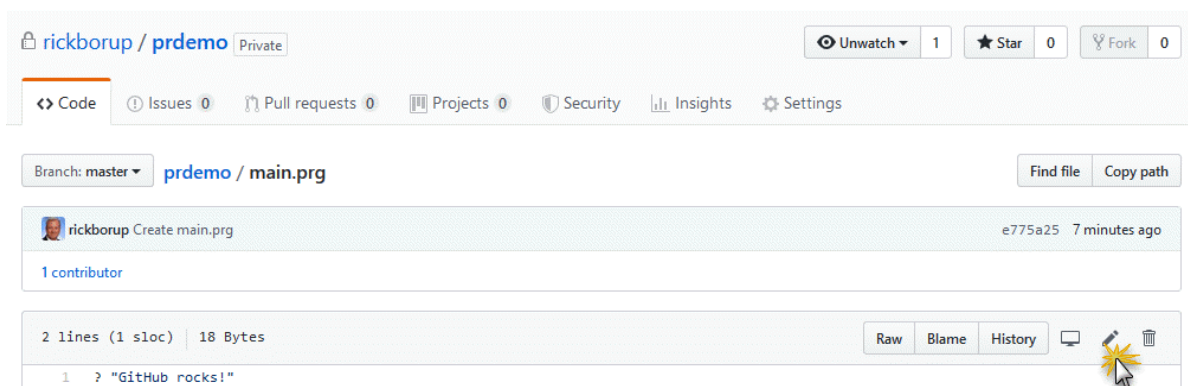


Figure 52: Click the pencil icon to edit the file.

Modify the contents of the file.

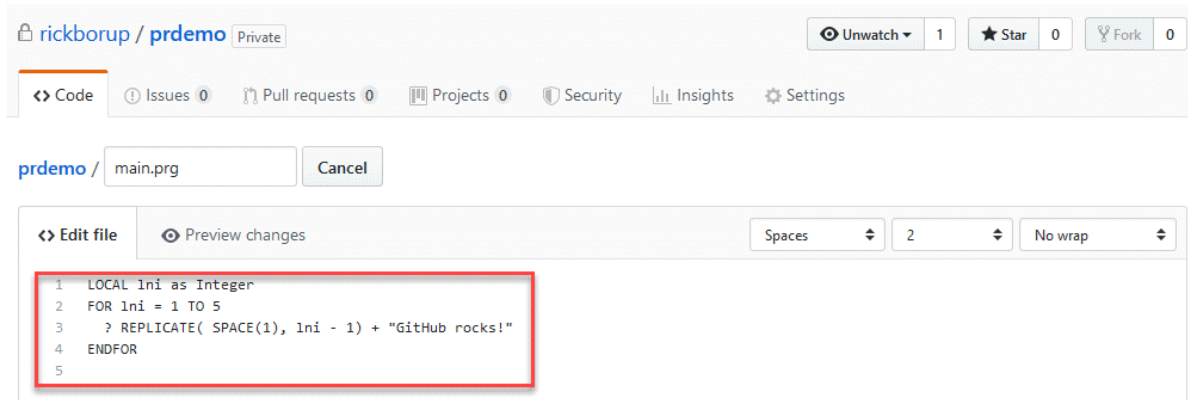


Figure 53: Modify the contents of the file.

Scroll down to the bottom of the page to the *Commit changes* section. Mark the option to create a new branch for this commit. GitHub generates a branch name automatically. You can use the generated branch name or replace it with another if you prefer. Click the green *Propose file change* button.

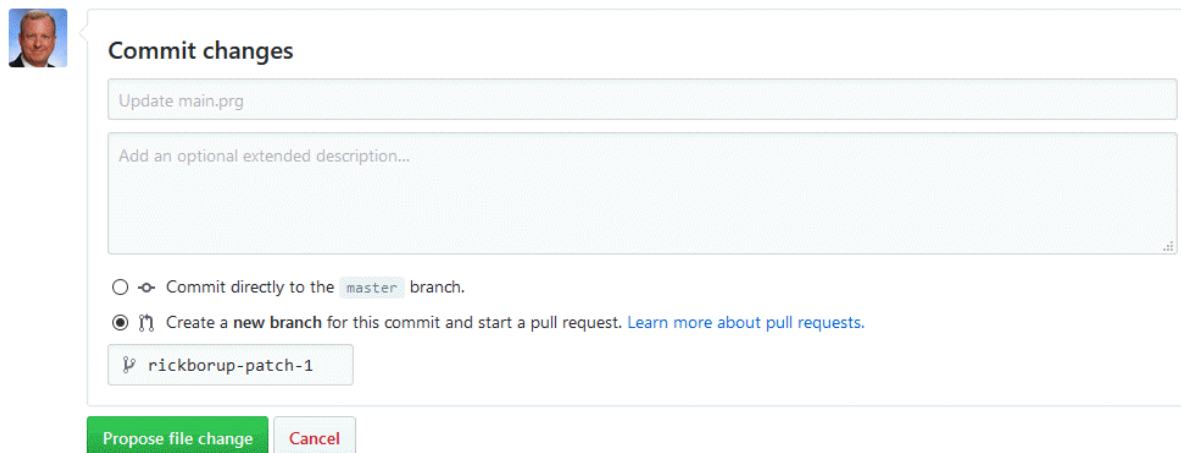


Figure 54: Create a new branch for this commit.

Write a description of the changes you're requesting in this pull request and click the green *Create pull request* button.

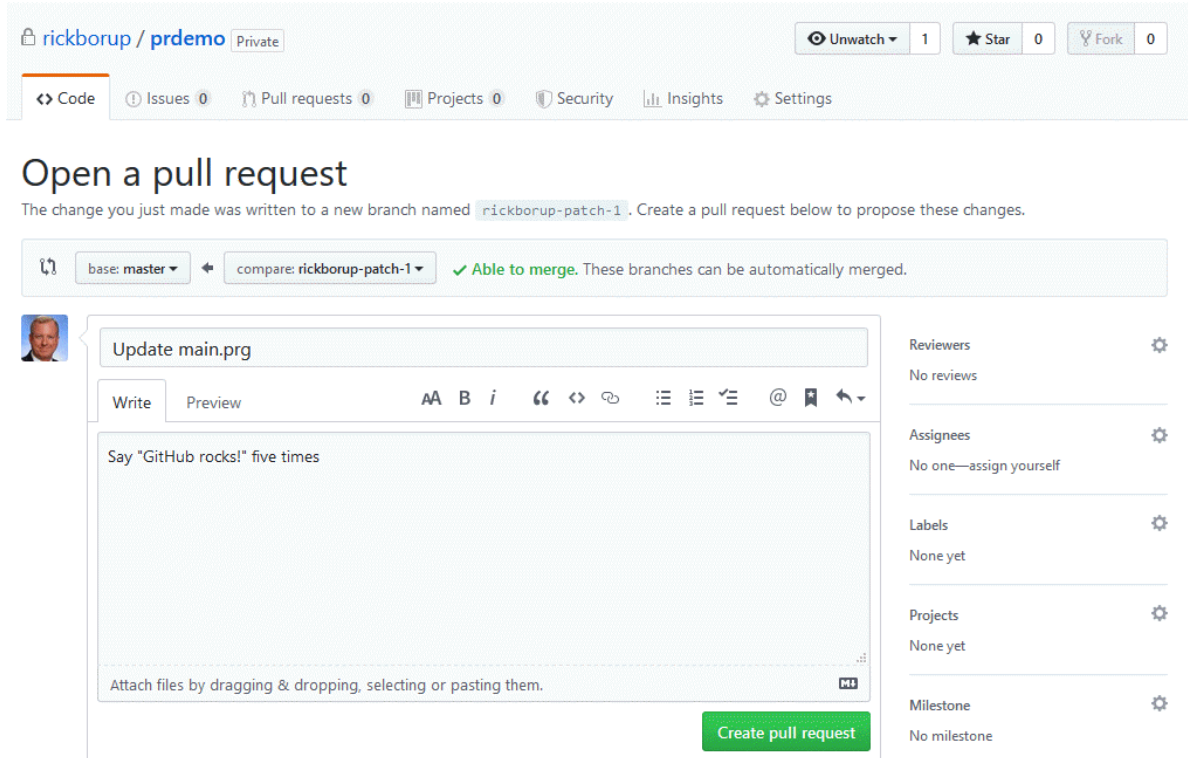


Figure 55: Enter a description and create the pull request.

GitHub selects the *Pull requests* tab on the home page, where the new pull request can be seen.

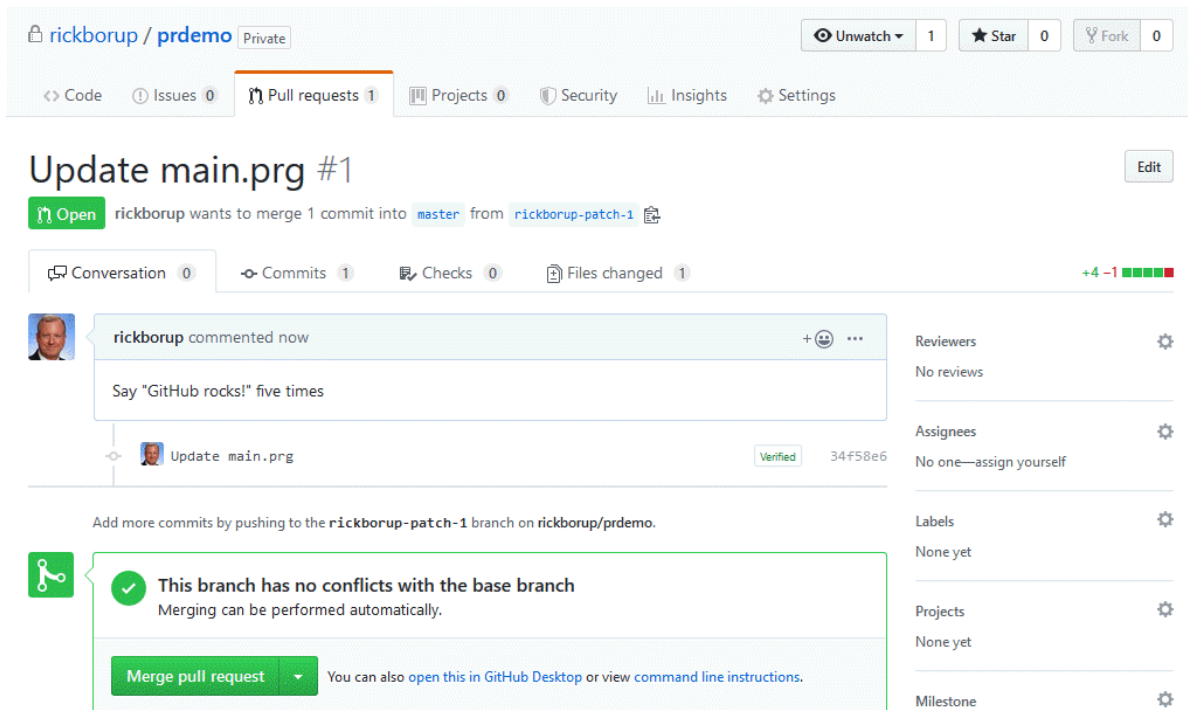


Figure 56: The new pull request is displayed on the *Pull requests* tab.

GitHub has determined there are no conflicts so this pull request can be merged. Click the green *Merge pull request* button, then click *Confirm merge*.

The screenshot shows a GitHub pull request for the repository 'rickborup / prdemo'. The pull request is titled 'Update main.prg #1' and is from the 'rickborup-patch-1' branch to the 'master' branch. A comment from 'rickborup' says 'Say "GitHub rocks!" five times'. The pull request is ready to be merged, and a merge confirmation dialog is open. The dialog shows the merge commit message: 'Merge pull request #1 from rickborup/rickborup-patch-1' and the file changes: 'Update main.prg'. The 'Confirm merge' button is highlighted in green.

Figure 57: Confirm the merge.

GitHub confirms the merge was successful. If desired, you can delete the branch with the button in the lower right corner.

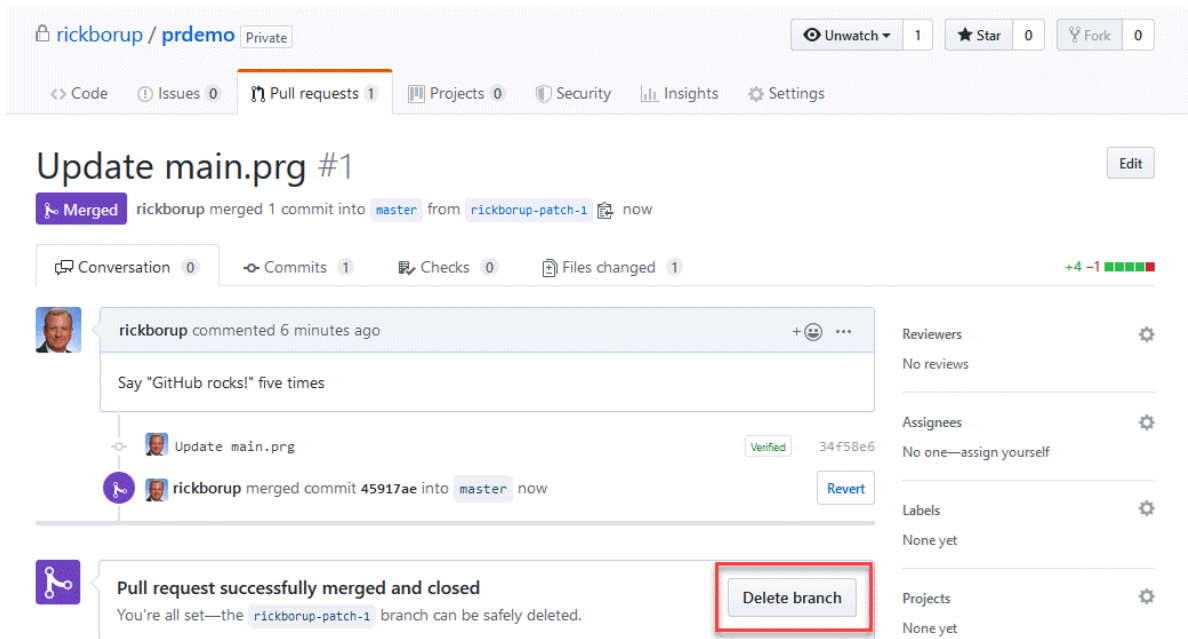


Figure 58: You can delete the branch after merging the pull request.

The revision history for this repository now shows three commits and no open pull requests.

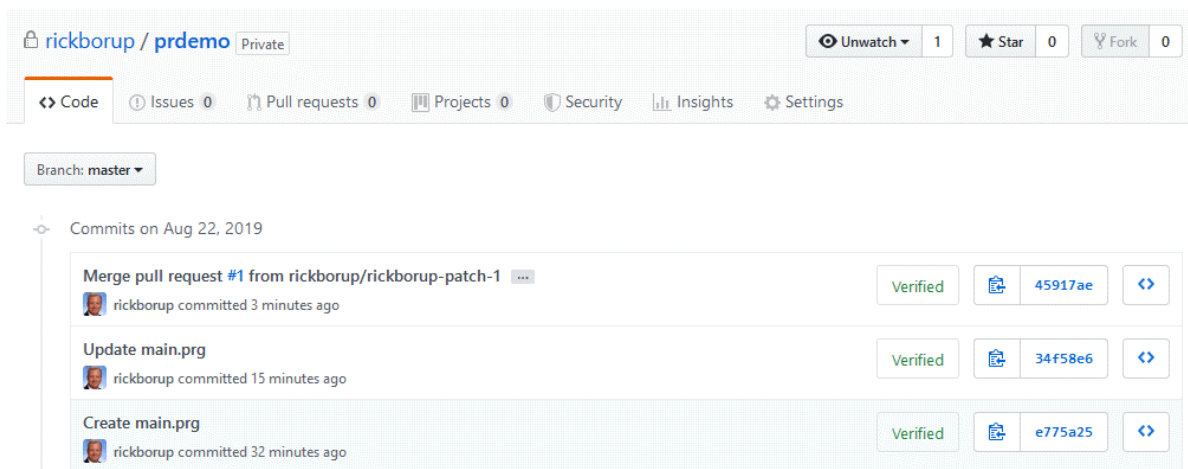


Figure 59: There are now three commits on this repository and zero pull requests.

Like file names on the Code page, commit messages in the list of commits become links when you hover the mouse over them. Click the link to see details of the commit. Figure X shows the information GitHub displays after clicking the link for the *Merge pull request #1* commit.

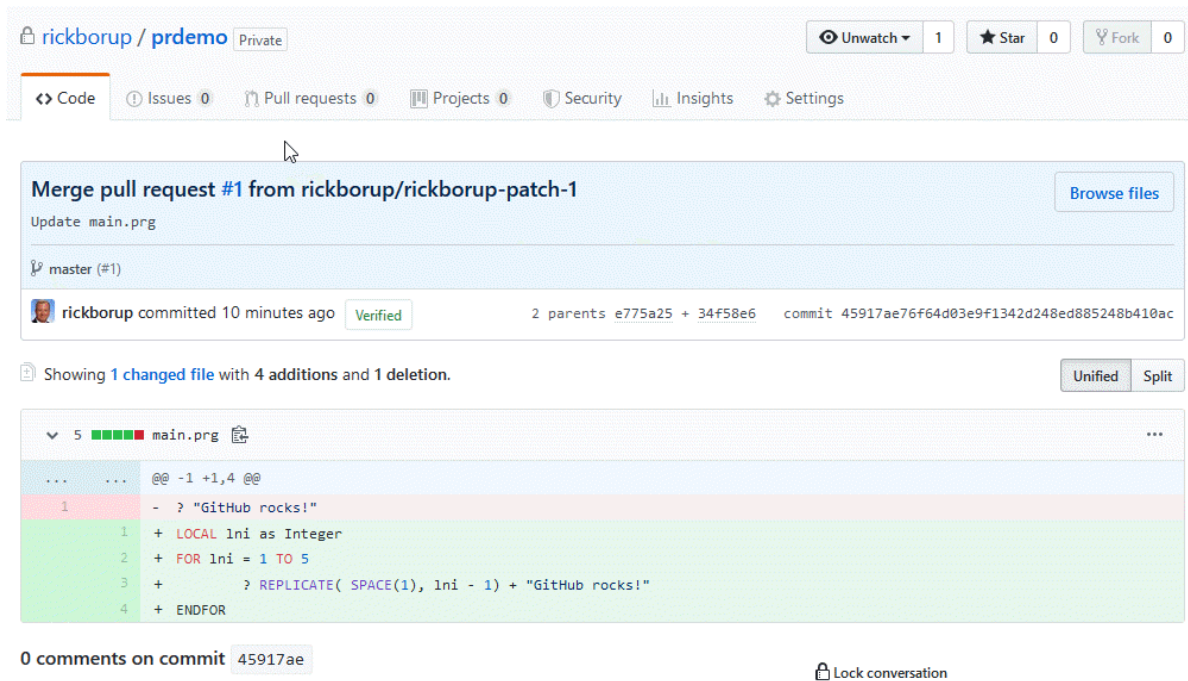


Figure 60: GitHub displays the detail of changes made in a commit.

That's the end of this exercise. You can now delete this repository if you don't want to keep it.

### GitHub for open source projects

Using GitHub for open source projects is much like using it for a team except that (a) the repository must of course be public and (b) you're essentially inviting anyone in the world to look at it, clone it, fork it, and possibly propose changes.

When a team uses GitHub to collaborate on a project, it's likely the team members all know one another—either in person or at least via email, social media, or some other form of one-to-one contact—and that they communicate with each other outside of GitHub. With an open source project, on the other hand, it's very likely the repository owner does not know the people who may want to contribute to the project and has no contact with them outside of GitHub, so effective communication via GitHub becomes more important.

#### Open source from the contributor's point of view

##### *Writing pull requests*

If you're a developer who wants to contribute to an open source project, you need to make an extra effort to communicate clearly and fully when submitting a pull request. A pull request described only as "Fix problem with user sign in" might be acceptable in a team situation where everybody on the team already knows what the problem is and that you've been assigned to fix it. On the other hand, If you're submitting a pull request to an open source project where you don't know anybody on the development team and they don't know you, you need to compose a much more complete and descriptive message to accompany your pull request if you want it to receive the attention you feel it deserves.

GitHub supports adding images, including animated GIFs, to pull requests, which can be very helpful in explaining the change(s) being requested.

You can find some guidelines for writing good pull requests on the GitHub blog titled “How to write the perfect pull request” at <https://github.blog/2015-01-21-how-to-write-the-perfect-pull-request/>,

One thing to remember when submitting a pull request to an open source project is that the project’s maintainers may have limited resources, may also be busy working on other projects, and are probably dealing with dozens or hundreds of other pull requests. You can’t reasonably expect yours to receive immediate attention or even any attention at all. Not all pull requests will be reviewed, much less accepted and merged into the project.

### *Submitting an issue*

If you’re exploring a project and discover a bug, unexpected behavior, or just want to suggest a change but not submit a pull request, you can instead submit an issue. Issues are not restricted to open source projects. They can be equally useful for private development teams and even for solo developers. This just seemed like a good place to introduce them.

Submitting an issue is just like submitting a pull request except that you’re not proposing and code changes to be pulled and merged. Other than that, the same care should be taken in composing the issue as would be taken for a pull request.

Let’s say I’m looking at the SWFoxDemo / myApp project and decide the color of the “Welcome” label on the main form would look better in Southwest Fox Kokopelli orange instead of red. While signed in to GitHub with my own account, I go to the home page for that project and create a new issue as shown in Figure 61.

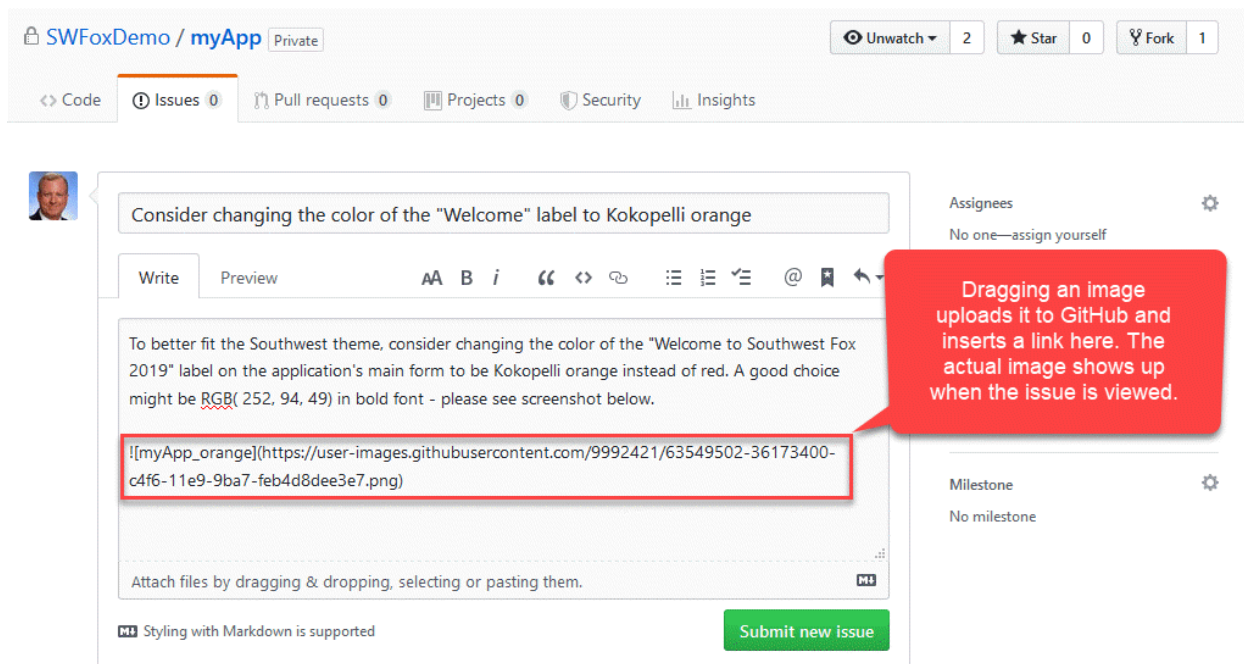


Figure 61: Anyone can create and submit an issue on a public repository.

I first described the issue as *The "Welcome" label on the main form should be Kokopelli orange, not red*. Before submitting it, I realized I was expressing an opinion—not everyone would agree that the color should be changed—so I reworded it to say *Consider changing the color of the "Welcome" label to Kokopelli orange*, which I hoped would be more favorably received.

To help the reviewer see the effect of the requested change, I attached an image to the issue by dragging a screenshot onto the page. Doing this uploads the image to GitHub. In *Write* mode the image shows as a link (see Figure 61). Click the *Preview* tab to see how the issue will look to the maintainer when they receive it. Clicking the green *Submit new issue* button completes the process and creates the issue on the project owner's GitHub account.

### Open source from the maintainer's point of view

If you're the owner or maintainer of an open source project on GitHub, you can expect to receive notifications of issues and pull requests from people who look at your code. Some may be snarky but hopefully most will be submitted in good faith with the intent to help fix bugs and suggest new features.

Anyone can create an issue for a public project on GitHub. An issue is a notice that someone has discovered a bug or other unexpected behavior and wants to make you aware of it without submitting any proposed changes to address it. As the maintainer of the project it's a good idea to pay attention to each issue as soon as you're notified of it. It might be a bad bug you want to fix right away!

### Addressing an issue

GitHub keeps track of the issues associated with a project. You can see the list on the *Issues* tab of the project's home page. An issue is either open or closed. Figure 62 shows the Issues tab of the SWFoxDemo / myApp project the way that project's owner sees it after the issue described above was submitted.

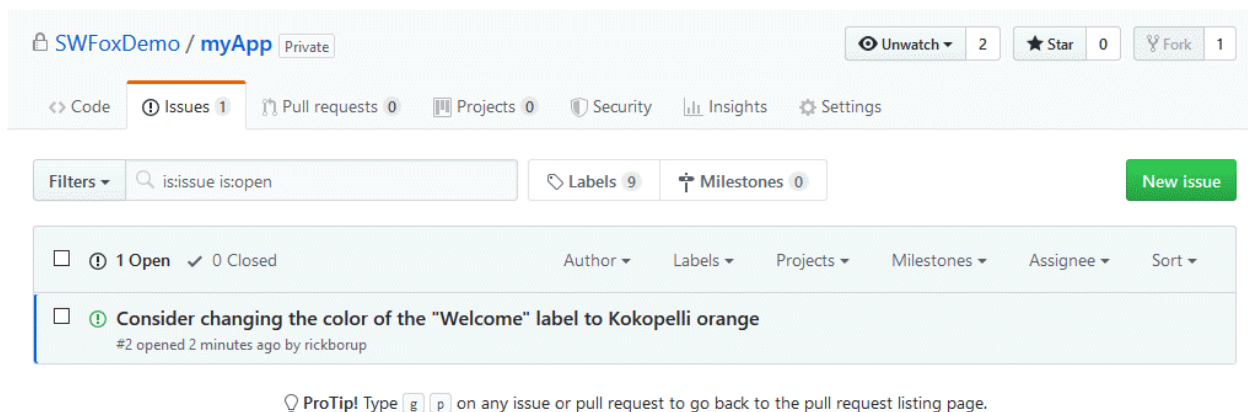


Figure 62: The project owner sees the list of issues that have been created for this project.

Open issues can be categorized with labels to indicate their type or status. To apply a label to an issue, select the issue by marking the check box on the left, click the *Label* dropdown list, and choose the desired label from the list.

Labelling issues helps the developer(s) to prioritize what needs to be worked on first. This issue is marked as an enhancement (see Figure 63).

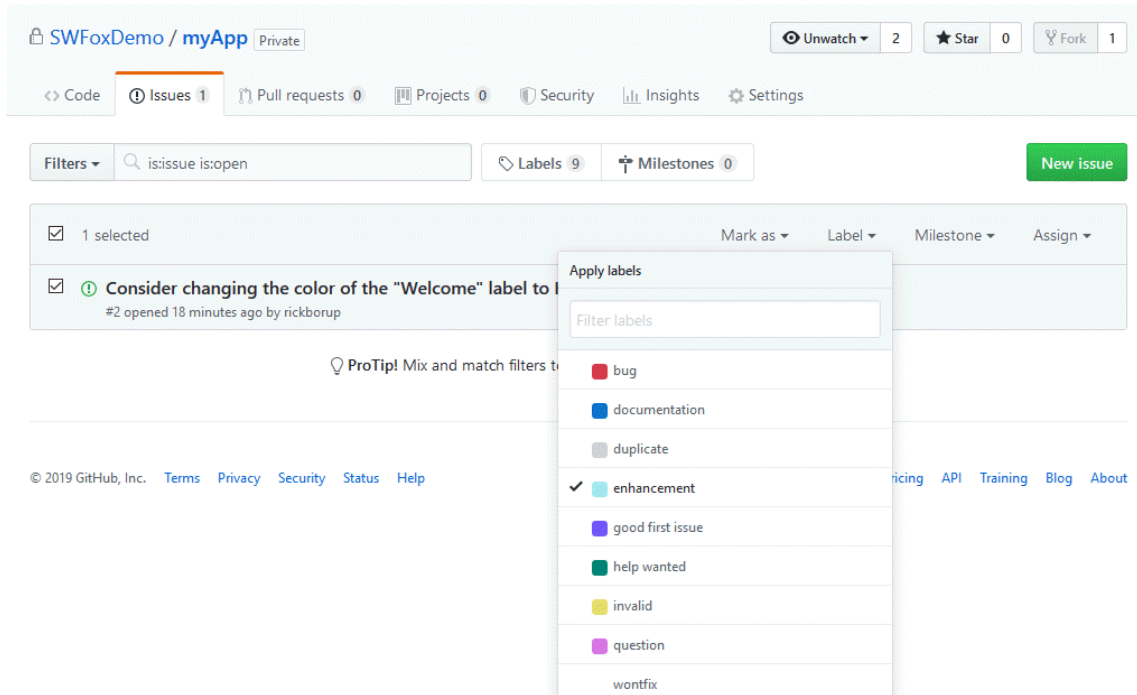


Figure 63: Issues can be marked with labels.

GitHub comes with the nine labels shown in the dropdown list in Figure 63, but you can edit them and/or add new ones if you want to. On the *Issues* tab, click the *Labels* button in the area to the left of the green *New issue* button, as shown in Figure 64.

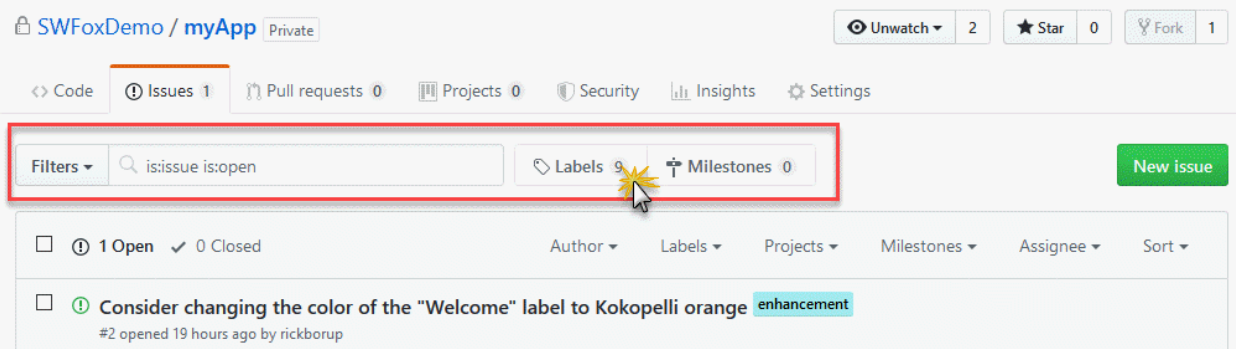


Figure 64: Click the *Labels* button to open the page where you can edit issue labels.

This brings up a page where you can add, edit, or delete issue labels for this repository.

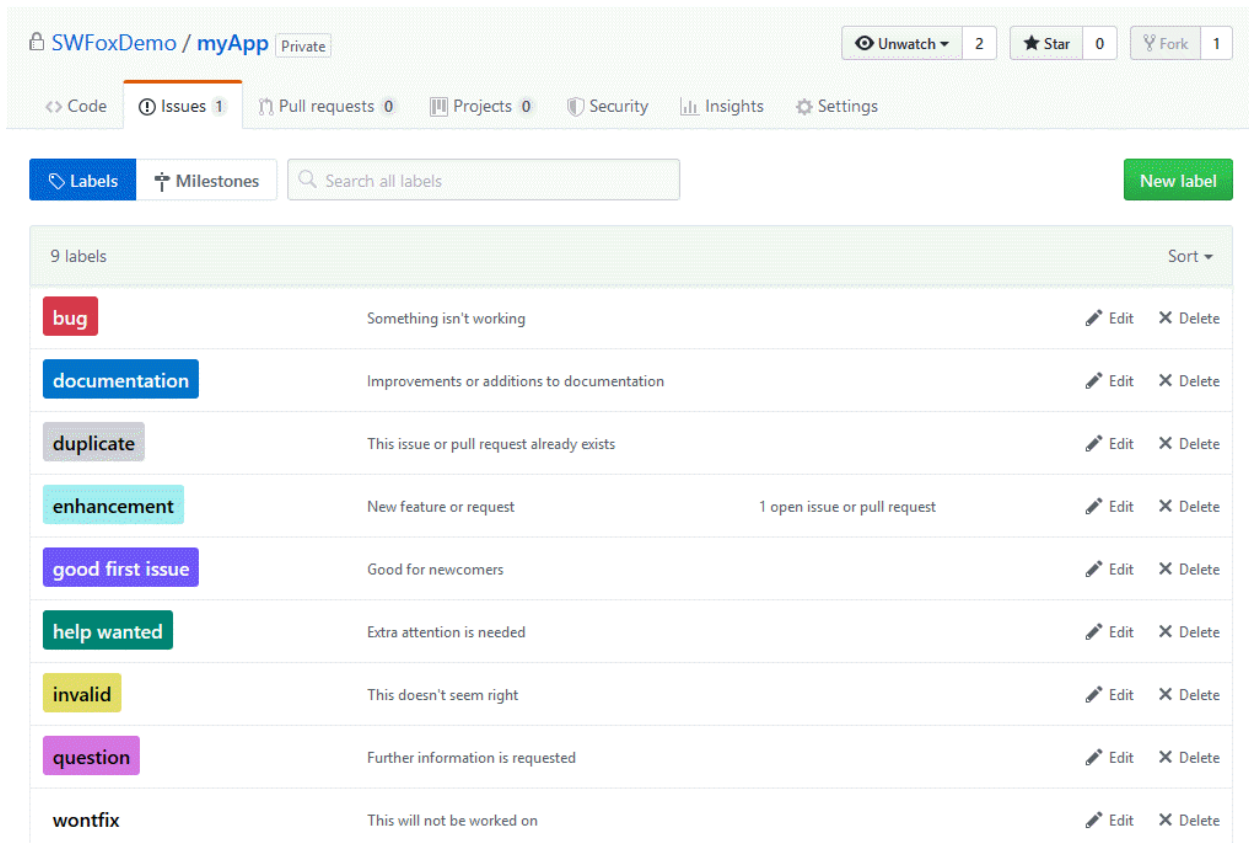


Figure 65: On this page you can add, edit, and delete issue labels for this repository.

GitHub also provides a mechanism for commenting on issues, providing an opportunity for a conversation between the maintainers of the repository and the people who are following it. There is an area at the bottom of each issue's detail page where you can write a comment. Figure 66 shows a comment from the maintainer of this project. The *Close and comment* button, which appears only for the project's owner / maintainers, provides a way to both comment on and close the issue on one step.

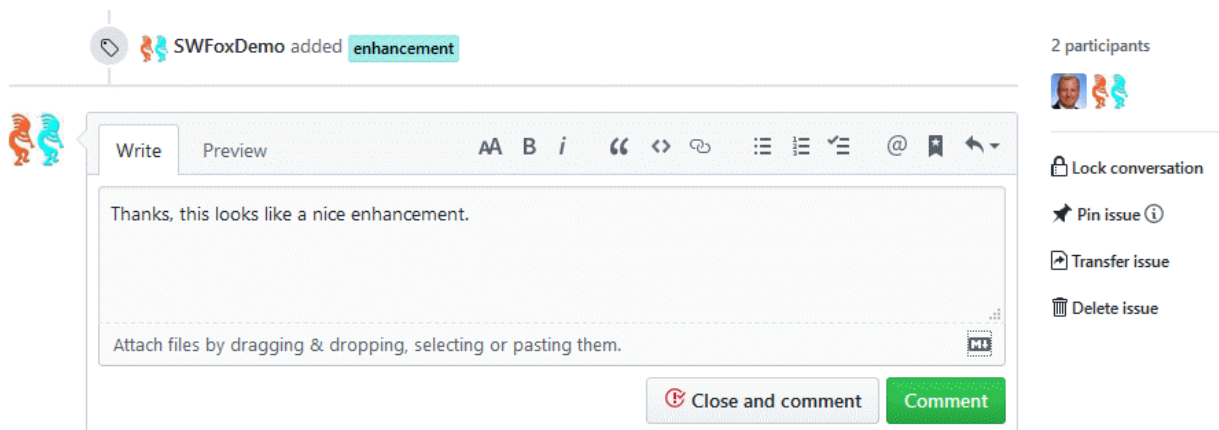


Figure 66: Comments are a way to conduct a conversation about an issue.

If an issue has been commented on, GitHub displays an icon showing how many comments there are.

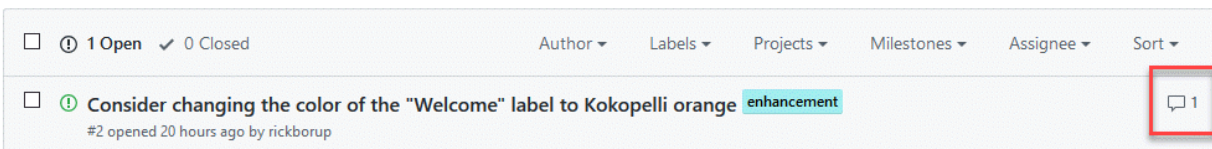


Figure 67: An icon shows the number of comments on an issue.

An issue can remain open for as long as the maintainers of the project want it to be. Once the issue has been addressed it can be marked as closed. Closing an issue removes it from the list of open issues.

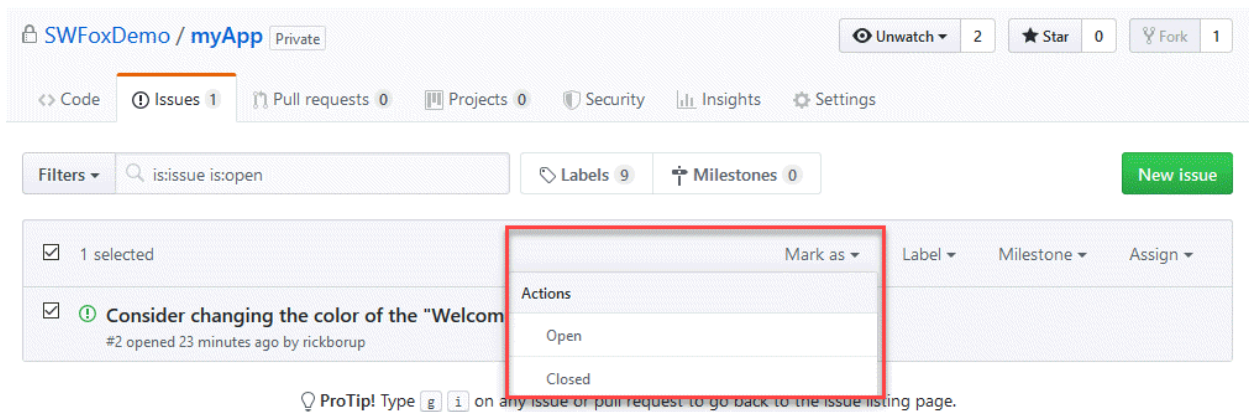


Figure 68: An issue can be closed after it's been addressed.

Issue tracking is an incredibly useful feature of GitHub. It can augment or even replace the need to keep track of issues using another tool.

### *Responding to pull requests*

The mechanics of dealing with pull requests on open source projects is identical to that for teams as discussed above. The main difference is that there will likely be more back and forth conversation with the submitter, since you don't know them and they don't know you.

## GitHub Projects

In addition to tracking issues, GitHub enables you to create projects. A project is a way to keep track of tasks associated with a repository. A repository can have more than one projects. To work with projects, start by clicking the Projects tab on the repository's home page (see Figure 69).

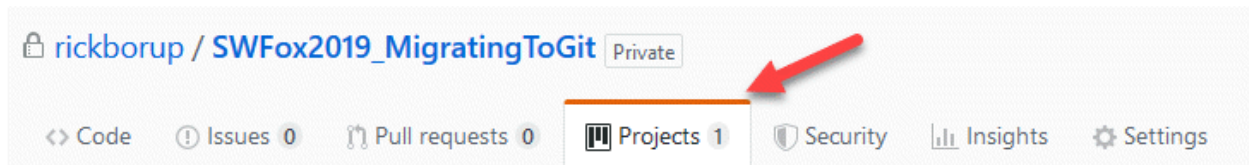


Figure 69: Click the Projects tab to create or work with GitHub projects.

Click the New Project button to create a new project. Give it a name and, optionally, choose a template. GitHub comes with a small number of templates including a basic Kanban-style “to do” list. You can start with blank slate but using a template saves time.

As an example, I put the development of this white paper under Git version control and created a repository for it on GitHub. I then created a basic Kanban to keep track of what’s done and what still needs to be done. Notes are easily added to a column by clicking the + icon, and can be moved from one column to another by clicking and dragging. Figure 70 shows the Kanban for this white paper at the point where I was working on this section.

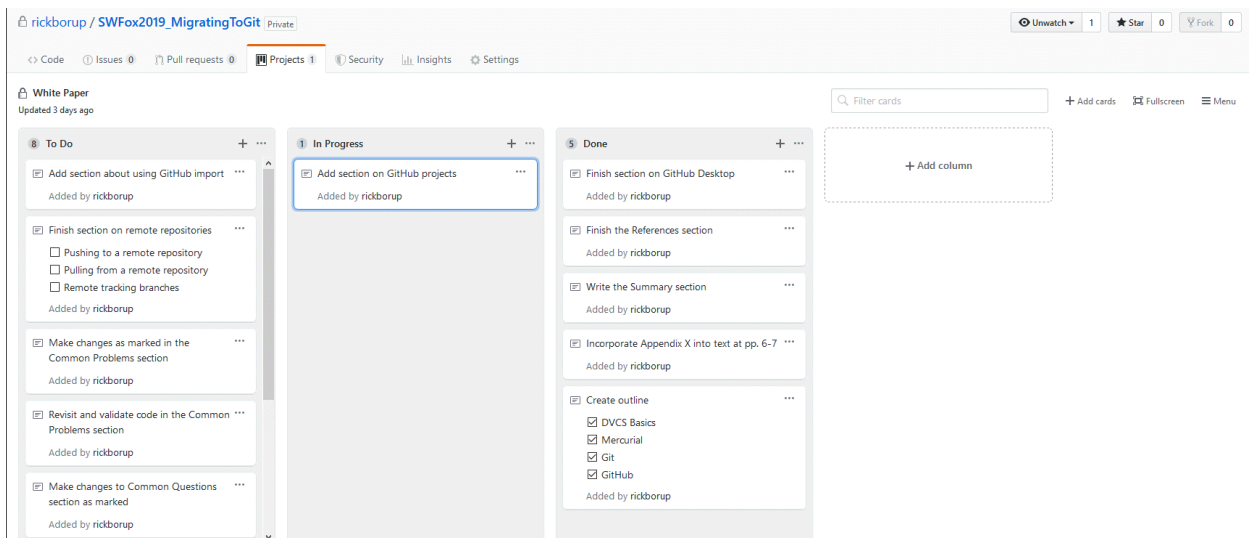


Figure 70: One type of project is a Kanban-style “to do” list.

While there are of course other ways to keep track of the “to do” list for a project, I find that having it integrated into the GitHub repository is tremendously useful.

### Using GitHub import to convert a Mercurial repository to Git

Now that you’re familiar with GitHub, it’s time to look at using GitHub’s *import* feature to convert a repository from Mercurial to Git. The primary limitation here is that the Mercurial repository must be accessible from a URL. GitHub’s import feature cannot access a repository from a local file system reference even if using the **Error! Hyperlink reference not valid.** type of URL. One option is to push your local Mercurial repository to Bitbucket or another hosting service from which GitHub can access it.

You can get started importing a Mercurial repository into GitHub in one of two ways. One is to create a new GitHub repository and then click the *Import code* button at the bottom (see

Figure X). The other is to click the + icon at the top of the GitHub home page (next to your profile icon) and choose *Import repository* from the popup menu as shown in Figure X.

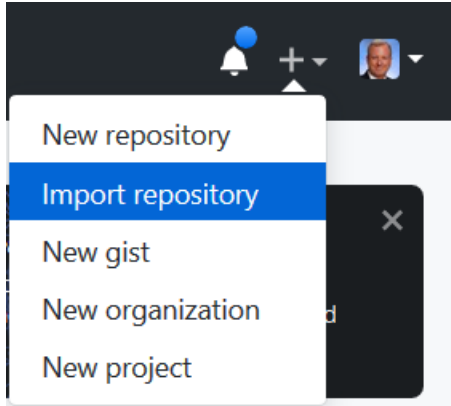


Figure 71: You can create a new GitHub repository and import another repository in one step.

If you do it that way, the next step is to enter the URL of the Mercurial repository to be imported along with a name for the new GitHub repository to be created.

### Import your project to GitHub


Import all the files, including the revision history, from another version control system.

#### Your old repository's clone URL

Learn more about the types of [supported VCS](#).


#### Your new repository details


Owner

 rickborup ▾

Name

Privacy

 **Public**  
Anyone can see this repository. You choose who can commit.

 **Private**  
You choose who can see and commit to this repository.

[Cancel](#)

[Begin import](#)

Figure 72: Enter the URL of the old repository along with a name for the new GitHub one.

Click the *Begin import button* to continue. GitHub begins the import process by detecting the type of version control system being imported, and starts the process. It may take a while to complete the process, so GitHub will email you when it's done.

## Preparing your new repository

There is no need to keep this window open, we'll email you when the import is done.

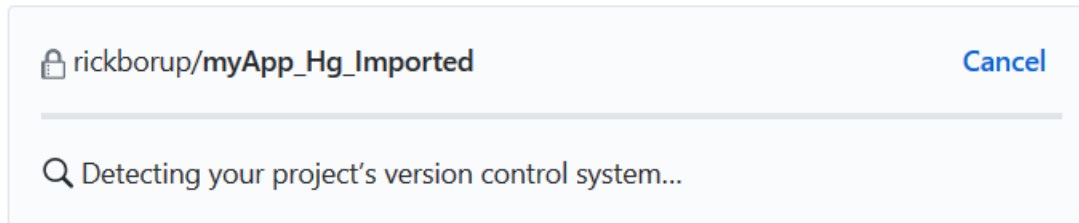


Figure 73: GitHub automatically detects the type of version control system being imported.

There may be one additional step, which is for you to provide the credentials GitHub needs to access the repository being imported. If requested, enter username and password when requested.

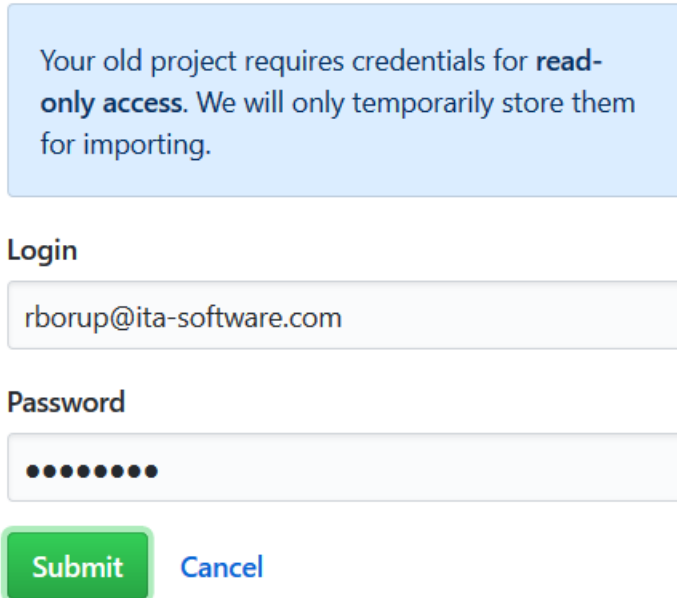
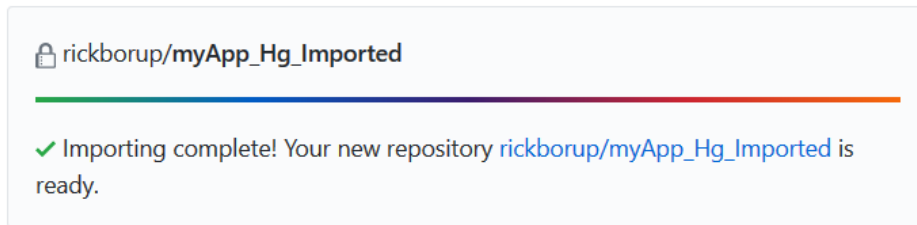


Figure 74: GitHub may need you to enter the username and password to access the repository being imported.

GitHub displays the page shown in Figure 75 when the import is complete. The part about matching authors means GitHub needs to know how to identify the people who have committed to this repository.

## Preparing your new repository

There is no need to keep this window open, we'll email you when the import is done.



### What's next?

We found **1 commit author** while importing that we need help identifying.

[Match authors](#)

Figure 75: The import is complete.

In this example I am the only committer (author) on the repository and both my Bitbucket and GitHub accounts are under the same email address, so GitHub make the connection. Click the *Connect* button to complete the process.

## Update commit authors

Change author email addresses or map to existing GitHub users.

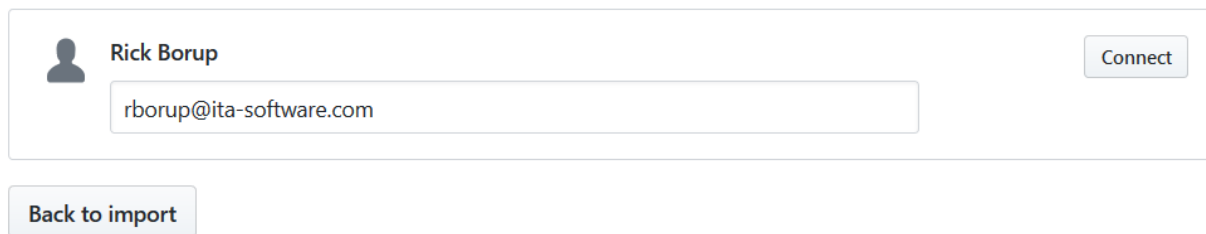


Figure 76: GitHub uses email address to map the author(s) of the imported repository to existing GitHub user(s).

I can now clone the new repository from my GitHub account into some folder on my local machine. From there I can work with it as before, except now it's under Git version control instead of Mercurial.

In this simple test the conversion was 100% successful. The new GitHub repository contains all the file from the old Mercurial one. The Mercurial *default* branch became the *master* branch, as expected, and the *dev* branch was recognized and converted as well. I was able to open the project in VFP and successfully build the EXE.

I don't know how well the import process might work with larger and more complex Mercurial repositories, but this test is encouraging. If you can make your Mercurial repository accessible to GitHub via a URL, using GitHub import is certainly easier than using the Hg-Git extension.

### The GitHub Desktop tool

GitHub Desktop is a tool for working with GitHub repositories without using a browser. The original version was released in 2015 as a replacement for the older GitHub for Mac and Windows. GitHub Desktop version 2.0, which was rewritten using Electron, was released in June 2019.

#### Installing GitHub Desktop

GitHub Desktop is free and open source. You can download and install it from <https://desktop.github.com>. Installation is straightforward but requires a 64-bit version of Microsoft Windows 7 or later.

One step of the installation process asks if you want to share usage stats with GitHub. If you opt in, be aware that the information sent to GitHub includes your GitHub account ID, as shown in Listing 22. See <https://desktop.github.com/usage-data/> for more information

Listing 22: Your GitHub account ID is included if you elect to send usage stats from GitHub Desktop.

```
  "unattributedCommits": 0,
  "unguidedConflictMergeCompletionCount": 1,
  "updateFromDefaultBranchMenuCount": 11,
  "user_id": <your GitHub account ID appears here>,
  "version": "0.8.1-beta3",
  "welcomeWizardSignInMethod": "basic"
}
```

GitHub Desktop documentation is available at <https://help.github.com/en/desktop>. The *Getting Started with GitHub Desktop* topic is a good place to start learning about it.

#### Using GitHub Desktop

To me, GitHub Desktop represents a layer of abstraction on top of GitHub, which, similar to other GUIs, is itself a layer of abstraction on top of Git. Instead of interacting with GitHub from your browser, GitHub Desktop has its own interface with menu items to perform tasks such as pushing, pulling, creating a new branch, merging, and creating pull requests. Other menu items enable you to do things like jumping to the home page for the current repository on GitHub in your browser, opening a command prompt at the local repository folder, or launching an instance of File Explorer at the local repository folder. Although the interface is different, the underlying mechanics of working with Git are the same—another reason it's important to understand Git from the command line level.

The GitHub Desktop main window has two tabs, Changes, and History, at the upper left. The Changes tab, which is selected in Figure 77, shows that in this example the working copy is clean—i.e., there are no changes pending in the local repository.

# Migrating to Git from Mercurial

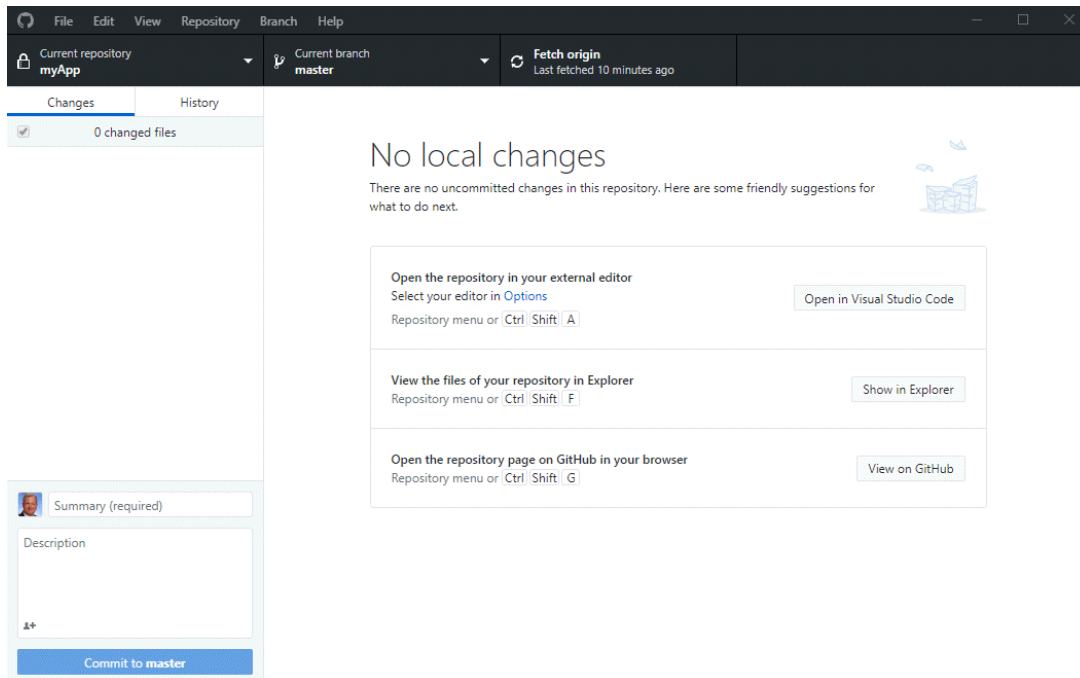


Figure 77: The *Changes* tab of the main GitHub Desktop window, with no local changes pending.

The *History* tab presents an interactive list of the revisions to the project. Clicking on a revision opens a list of the files that were modified in that commit. By selecting a file from the list, you can view the changes made to that file in that commit. In Figure 78, the `readme.md` file in the most recent commit is selected, showing the changes made to that file in that commit.

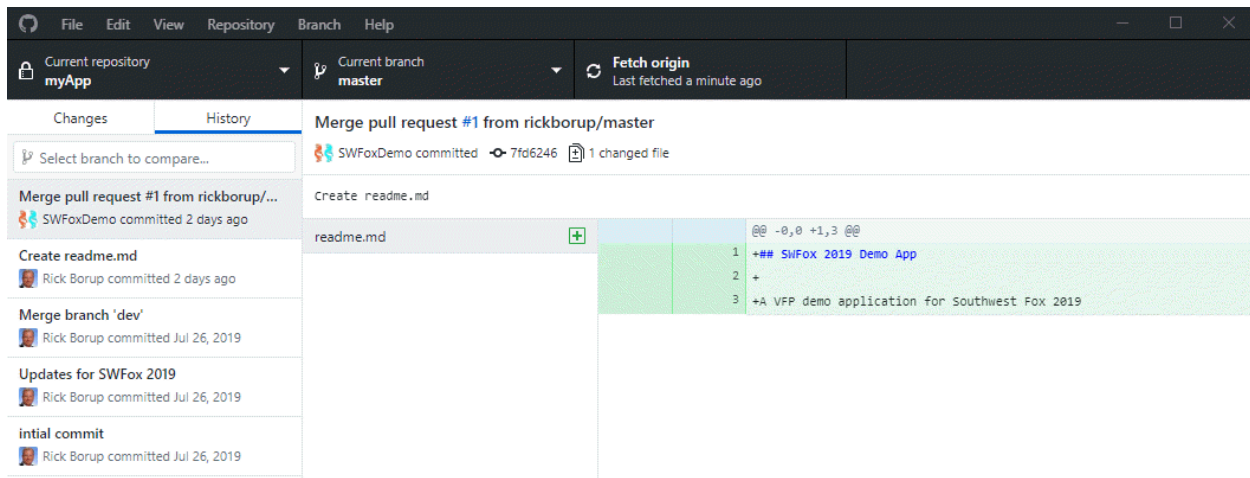


Figure 78: GitHub's *History* tab enables you to browse through the project's revision history.

You can probably do everything you need to do with GitHub from GitHub Desktop, but here may be times when you prefer to work on your project from your browser or even from the command prompt. GitHub Desktop's menu items make it easy to switch to those interfaces when you want to.

## Common Questions

This section lists some of the questions that often come up when using Git. It's intended as a quick reference. Some of these are discussed in more detail elsewhere in this paper.

### How do I find out which files are being tracked?

Use the *ls-files* command with the *--stage* option.

```
git ls-files --stage && add --abbrev for short SHA-1's
```

### How do I find out which files are not being tracked?

Use the *ls-files* command with the *--others* option.

```
git ls-files --others
```

### How do I know if my working directory is clean?

Use Git's *status* command.

```
git status
```

### How do I view the history of changes to my repository?

Use the *log* command. To view the log in text format:

```
git log && multiple lines of output per commit, or
```

```
git log --oneline && one line per commit with short SHA-1's
```

Note that what you see in the log depends on which branch you're on.

To view the log as a graph:

```
git log --graph && displays full SHA-1's, or
```

```
git log --graph --abbrev-commit && displays short SHA-1's, or
```

```
git log --graph --abbrev-commit --oneline && one line per commit with short SHA-1's
```

That's a lot to type every time if you use it frequently. The next section shows how to create an alias.

### Can I create shortcuts for long Git commands I use frequently?

Yes, you can. It's called creating an alias, and it's done using the *git config* command. The preceding command is a good example because it's relatively long and could be difficult to remember and tedious to type in every time you want to use it. It would be convenient to use something shorter to get the same result.

Listing 23 shows how to create a global alias to do that. Note that double quotes are required when working from the Windows command prompt. Use single quotes if you're working in Git Bash.

Listing 23: Create an alias as a shortcut to long Git commands.

```
git config --global alias.glog "log --graph --abbrev-commit --oneline"
```

This tells Git to add the following entry to the global `.gitconfig` file:

```
[alias]
  glog = log --graph --abbrev-commit --oneline
```

Now you can type

```
git glog
```

and Git runs the long version of the command for you. Nice!

To modify an alias, run *git config* as before but specify a different command for the alias. For example, remove the `--abbrev-commit` option if you prefer to see the full commit IDs.

```
git config --global alias.glog "log --graph --oneline"
```

To delete the alias, use

```
git config --global --unset alias.glog
```

or edit the global `.gitconfig` file and remove the entry manually, either by opening the file from a text editor or launching Git's default text editor with

```
git config --global --edit
```

Be careful with the second choice. Editing the configuration file directly carries the risk of making a mistake and screwing something up.

Coming from Mercurial, I'm accustomed to using the abbreviation `stat` for `status` when working from the command line. Git doesn't understand `stat`, but it's easy to create an alias for it.

```
git config --global alias.stat "status"
```

### How do I discard a change to a file I haven't staged or committed?

There is more than one way to discard changes that have not yet been staged or committed.

```
git reset --hard HEAD  && discard all changes to all files in the working copy
```

```
git checkout foo.txt  && discard changes to foo.txt
```

```
git stash  && stash changes to all modified files
```

```
git stash -- foo.txt && stash changes to foo.txt
```

### How do I know if any files are staged to be committed?

Use Git's *status* command.

```
git status
```

### How do I list the branches in my repository?

Use the *branch* command.

```
git branch && local branches
```

```
git branch --all && include remote branches
```

The current branch is marked with an asterisk.

### How do I create a new branch?

Use the *branch* command followed by the *checkout* command to switch to it.

```
git branch dev && create the 'dev' branch  
git checkout dev && switch to the 'dev' branch
```

Or do both in one step.

```
git checkout -b dev && create the 'dev' branch and switch to it
```

### How do I delete a branch?

Use the *-d* option with the *branch* command.

```
git branch -d <branch name>
```

Git does not allow you delete the currently checked out branch.

You can delete a branch on a remote repository without navigating to the remote by pushing an empty branch with the same name. You must of course have push permissions on the remote.

```
git push <remote name> :<branch name> && note the colon
```

For example

```
git push origin :dev && delete the dev branch on origin
```

### How do I know if there are any changes to be pulled from upstream?

Use the *fetch* command. The general syntax is

```
git fetch [<options>] [<repository>]
```

The *fetch* command downloads objects and references from another repository without updating your working copy. If the *repository* parameter is omitted, the default is *origin*.

Use the `--dry-run` option to show what would be fetched without downloading anything.

```
git fetch --dry-run <repository>
```

If there is nothing to be fetched, there is no output from this command.

To fetch from the *origin*, simply run

```
git fetch
```

If changes were fetched, the status command tells you how far behind your branch is.

```
C:\>git status
On branch master
Your branch is behind 'origin/master' by 4 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working tree clean
```

Follow with *git pull* to incorporate the downloaded changes into your working copy. *Fast forwarding* is possible if the branch can be updated by simply moving the HEAD without creating a merge commit.

### How can I find all the commits where a file was changed?

Use the *log* command with the `--all` option and pass the file name as a parameter.

```
git log --all -- foo.txt
```

Note the space after the final double dash. The name of the file, `foo.txt`, is a parameter, not an option.

The output from this generates several lines per commit in the command window, so it is not the easiest thing to work with because you need to scroll through several screens to follow it. Personally, I prefer to view repository history in graph format with abbreviated commit IDs and one line per commit. The following is a modification to accomplish that.

```
git log --all --abbrev-commit --graph --oneline -- foo.txt
```

You can append additional options and parameters to aliased commands. If you're using the *glog* alias that was introduced earlier, you can use it in this situation too.

```
git glog --all -- foo.txt
```

The output looks like this:

```
* 952edce (HEAD -> master, ThumbDrive_I/master, NASDrive_T/master) merged dev after
fix conflict in foo.txt
| * 2c4bf4d (ThumbDrive_I/dev, NASDrive_T/dev, dev) add a line to foo.txt
```

```
* | 098e080 Remove blank last line
* | 070ff20 (tag: v1.0.4) changed foo.txt
|/
* e46e6fd Changed file in branch 'dev'
* f6887dd modified foo.txt
* 7de1e5a (tag: v1.0.0) initial commit
```

This format makes it easy to see the commits that include foo.txt. Tools like Sourcetree typically provide a way to get this information from the GUI, too.

### What is the difference between fetch and pull?

Fetch downloads references to newer commits found in the remote repository but doesn't merge them into the local repository. The references are stored in `FETCH_HEAD`. See <file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-fetch.html>.

Pull downloads changes from the remote repository and merges them into the local repository. Pull is equivalent to `git fetch` followed by `git merge FETCH_HEAD`. See <file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-pull.html>.

### How do I find out what remotes are associated with my local repository?

Use Git's `remote` command. With no options it simply lists the remotes.

```
C:\>git remote
origin
```

With the `--verbose` option (or `-v` for short) the `remote` command also displays the path.

```
C:\>git remote --verbose
origin https://github.com/rickborup/myProject (fetch)
origin https://github.com/rickborup/myProject (push)
```

### How do I find out which remote branches are associated with my repository?

Use the `--all` option with the `branch` command.

```
git branch --all
  dev
* master
  remotes/origin/master
```

### How do I find out the location of a remote repository?

Use the `get-url` option with the `remote` command.

```
C:\>git remote get-url origin
https://github.com/rickborup/myProject
```

### How do I change the location of a remote repository?

Use the `set-url` option with the `remote` command.

```
git remote set-url <name> <new URL>
```

### How do I find out if a tag name is valid in Git?

Git provides a *check-ref-format* command that checks if a reference name is valid. The rules for valid reference names—for example, names you might want to use for tags—are spelled out in the Git manual page for the *check-ref-format* command in the local documentation at <file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-check-ref-format.html>.

The command exits with a 0 value if the name is valid or with a non-zero value if it is not. It does not display any output in the command window, but a trick found on Stackoverflow<sup>9</sup> provides a way to capture and display the result.

The following code uses Git's *check-ref-format* command to check if the tag "Release 9.2.101" is valid. In this case the "&&" is part of the command, not a comment.

```
C:\myProject>git check-ref-format "tags/Release 9.2.101" && echo OK || echo not OK
not OK
```

## Common Problems

### Help! I typed 'git log' and now the command window is stuck

If the log overflows the screen, Git displays a colon on the bottom line and waits for you to press a key to continue. Press the ENTER key to advance one line or the spacebar to advance to the next page. When the output gets to the end it displays (END) on the bottom line but at that point there's no obvious way to exit out of it—none of the expected keys like ESC, ENTER, Ctrl+C, or F6 work. You need to type the letter 'q' to quit and return to the Windows command prompt.

Listing 24: The colon indicates there is more. Press the ENTER key to continue.

```
commit 359808acce6c8694affadb497ef8b1aad00be778
Author: Rick Borup <rborup@ita-software.com>
Date:   Fri Jul 12 12:40:03 2019 -0500
    add .gitignore
commit 443a81831c48a13c87fa05f2501a30d89533da2d
Author: Rick Borup <rborup@ita-software.com>
Date:   Mon Jul 8 17:30:28 2019 -0500
:
```

Continue to press the ENTER key or the spacebar until the (END) marker appears.

Listing 25: The (END) marker identifies the end of the output. Press "q" to quit.

```
commit a9ae527419ca19d9d327f1e0bcdf36536b7c9d68 (dev)
Author: Rick Borup <rborup@ita-software.com>
Date:   Thu Aug 1 17:47:06 2019 -0500
```

---

<sup>9</sup> The 'echo' trick is from a post by Willem Van Onsem and @NikosAlexadris on Stackoverflow at <https://stackoverflow.com/questions/26382234/what-names-are-valid-git-tags>

```
add line 3
```

```
commit ba73a777f31ef6375e625276944017721cb04d69
Author: Rick Borup <rborup@ita-software.com>
Date: Thu Aug 1 17:46:06 2019 -0500
```

```
initial commit
(END)
```

Press 'q' to return to the command prompt.

### Help! I got a merge conflict after pulling. Now what do I do?

This situation occurs when Git attempts to do a merge after pulling from a remote repository. A merge conflict can result if the file that was modified in your local repository was modified in a different way in the commit being pulled.

```
C:\GitTest>git pull origin
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The file foo.txt now looks like this:

```
C:\GitTest>type foo.txt
line 1
line 2
<<<<<< HEAD
this line 3 was added and committed in the local repo && this is "ours"
=====
this line 3 was added in a clone and then pushed to origin && this is "theirs"
>>>>>> 0b57fe905449c8014d7d98c5cd0e413dc444a368
```

Resolve the conflict by editing the file and then finishing the merge. Look for the conflict markers in the file, make the necessary changes, save the file, add it to the index, and tell the merge to continue.

```
C:\>notepad foo.txt && fix the conflict and save the file
C:\>git add foo.txt
C:\>git merge --continue
```

### Help! I pulled before committing. Now what do I do?

This situation occurs if you pull from a remote while your working copy is "dirty", meaning it contains uncommitted changes to a file.

```
C:\>notepad foo.txt && modify foo.txt but do not add it yet
C:\>git pull origin && origin contains changes to foo.txt you do not have yet
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
```

```
Unpacking objects: 100% (3/3), done.
From C:\Temp\myProject
 22afe54..c09c2c3 master    -> origin/master
error: Your local changes to the following files would be overwritten by merge:
  foo.txt
Please commit your changes or stash them before you merge.
Aborting
Updating 22afe54..c09c2c3
```

Remember that a *pull* updates the working copy. Git alerts you that completing the pull would overwrite your local changes, aborts the merge, and prompts you to commit or stash you changes before continuing.

In this example, you'll wind up with a merge conflict whether you commit or stash. In both cases the conflict needs to be resolved—the difference is simply in the sequence of steps.

If you commit your changes and then pull again, you're in the situation described in the previous example: you get a merge conflict, which you need to resolve before finishing.

If you stash your changes and then pull again, the pull succeeds. You can then reapply your changes from the stash after the pull is complete. Reapplying the stashed changes results in a merge conflict, which you then resolve as in the previous example.

```
C:\>git stash && stash the current state of your working copy
Saved working directory and index state WIP on master: 22afe54 resolve merge conflict
with pulled commit
```

```
C:\>git stash list && (optional) list the contents of the stash
stash@{0}: WIP on master: 22afe54 resolve merge conflict with pulled commit
```

```
C:\>git stat && (optional) confirm the working copy is now clean
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)
nothing to commit, working tree clean
```

```
C:\>git pull origin && pull again - this time it succeeds
Updating 22afe54..c09c2c3
Fast-forward
 foo.txt | 1 +
 1 file changed, 1 insertion(+)
```

```
C:\>git stash pop && apply the changes from the top of the stash stack
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
The stash entry is kept in case you need it again.
```

### Help! My source code file disappeared!

You know for certain that your project has a source code file named *file2.txt*, but when you look at your project folder in File Explorer, *file2.txt* is not there! Where did it go?

The most likely explanation is that file2.txt is a tracked file and you've checked out a branch that does not include it.

This example begins with a single file named file1.txt in the master branch. Here's the workflow:

```
1. C:\>git checkout master && check out the master branch
2. C:\>dir && file1.txt is the only file
3. 08/28/2019 02:44 PM          7 file1.txt
4.          1 File(s)          7 bytes
5. C:\>git checkout -b dev && create and checkout a dev branch
6. C:\>git checkout master && check out the master branch
7. C:\>notepad file2.txt && create file2.txt, add, and commit it
8. C:\>git add file2.txt
9. C:\>git commit -m "add file2.txt"
10. C:\>dir && confirm that both file1.txt and file2.txt exist
11. 08/28/2019 02:44 PM          7 file1.txt
12. 08/28/2019 02:45 PM          7 file2.txt
13.          2 File(s)         14 bytes
14. C:\>git checkout dev && go back to work on the dev branch
15. C:\>dir && what happened to file2.txt??
16. 08/28/2019 02:44 PM          7 file1.txt
          1 File(s)          7 bytes
```

Remember that checking out a branch updates the working copy to match the state of the branch head. When the dev branch was checked out at line 14, the working copy was updated to the state it was in as of the branch head, which does not include file2.txt

Solution: check out the *master* branch and file2.txt is present again.

In the example above, you already knew file2.txt exists in the master branch. But what if you didn't know which branch or branches it exists in? How would you find out?

The first step is to use Git's log command with the *--all* option followed by the file name as a parameter. Note the space between the *--* and the file name. This identifies the commit(s) that include that file.

Listing 26: The log command accepts a file name as a parameter.

```
git log --all -- file2.txt
commit 31fe77d51968277d2a052c0e237ac9b5c389e5b8 (master)
Author: Rick Borup <rborup@ita-software.com>
Date:   Wed Aug 28 14:49:14 2019 -0500
    add file2.txt
```

The next step is to use Git's branch command to find out which branch contains that commit. In this example, it's the master branch.

```
git branch --contains 31fe77d
*master
```

Now that you know `file2.txt` is in the *master* branch, you have a choice depending on what you want to do next. If you want to leave the *dev* branch, you can stash the changes (if any) you've made in the *dev* branch and then check out the *master* branch. On the other hand, if you want to stay on the *dev* branch and get to the file, you can check out that specific file from that specific commit.

```
git checkout dev
git checkout 31fe77d -- file2.txt
```

At this point, Git sees `file2.txt` as a new file, because it's not previously known to the *dev* branch. Treat it like any other new file by staging it and then committing it on the *dev* branch.

```
git add file2.txt
git commit -m "added file2.txt in dev branch"
```

To see a recap of what's been done, run the `log` command in Listing 26 again and note that `file2.txt` now shows up in both the *master* and the *dev* branches.

```
git log --all -- file2.txt
commit be99da88de21d8694e8143fc79ce6439bbc9ecff (HEAD -> dev)
Author: Rick Borup <rborup@ita-software.com>
Date:   Wed Aug 28 15:02:11 2019 -0500
    add file2.txt in dev branch
commit 31fe77d51968277d2a052c0e237ac9b5c389e5b8 (master)
Author: Rick Borup <rborup@ita-software.com>
Date:   Wed Aug 28 14:49:14 2019 -0500
    add file2.txt
```

The goal of making `file2.txt` accessible while working in the *dev* branch was probably so you could make changes to it.

```
[make some change to file2.txt - still on the dev branch]
git add file2.txt
git commit -m "modified file2.txt in dev branch"
```

Assuming you're done with changes in the *dev* branch, it's now time to merge it back into the *master* branch. This will most likely result in a merge conflict because `file2.txt` is different in *master* than it is in *dev*.

```
git checkout master
git merge dev
CONFLICT (add/add): Merge conflict in file2.txt
Auto-merging file2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Open the file in an editor, find the conflict markers, and make the appropriate changes to resolve the conflict. The unresolved file looks like this:

```
file2
<<<<<<< HEAD
```

```
=====  
line 2 added in dev branch  
>>>>>> dev
```

Assuming you want to retain the new line that was added in the dev branch, remove the conflict markers, save the file, then stage and commit it.

```
[still on the master branch]  
git add file2.txt  
git commit -m "resolved conflict in file2.txt"
```

### Help! I can't push to a remote repository on the local file system

You want to create a remote repository in some folder accessible via the local file system—for example, on a file server or a thumb drive—so you can push changes to it. You create a new folder for the remote and initialize a Git repository. You then try to push to the new remote from your local repository but you get the following error:

```
1. C:\myProject>cd F:  && a file server on F:  
2. F:\>md myRemote  && create a folder named myRemote  
3. F:\>cd myRemote  
4. F:\myRemote>git init  && initialize a Git repository  
5. Initialized empty Git repository in F:/myRemote/.git/  
6. F:\>cd C:\myProject  && return to your local project folder  
7. C:\myProject>git remote add myRemote F:\myRemote  && add the remote to your repo  
8. C:\myProject>git push -u myRemote master  && attempt to push to F:\myRemo  
9. Fatal: 'myRemote' does not appear to be a git repository
```

Explanation: The push failed because the Git repository in F:\myRemote was not initialized as a bare repository in line 4, above.

The clue is on line 5, which shows the Git repository was initialized in F:/myRemote/.git/ instead of F:/myRemote/. Bare repositories do not have a .git folder.

You might be tempted to redefine myRemote as F:/myRemote/.git/ and push to that. It won't work—you'll get an interesting set of messages about refusing to update the checked-out branch in a non-bare repository.

The solution is to erase the .git folder from F:\myRemote and start over from the *init* command, this time using the *--bare* option.

```
F:\myRemote>git init  && initialize a bare Git repository  
Initialized empty Git repository in F:/myRemote/  && notice there's no /.git/
```

The push now succeeds.

### Help! I need to modify a commit I just made. What do I do?

The rule of thumb is, don't change history if it's already been shared or made public. In other words, don't modify or remove a commit if it has already been pushed to a public or

shared remote, or if there's a chance another developer already has a copy of it in some other way.

If you keep that in mind, you can easily amend the most recent commit in your local repository. To amend only the commit message

```
git commit --amend -m "new message"
```

To amend the contents of the commit (one or more modified files), make changes, stage if necessary, then

```
git commit --amend [-m "new message"]
```

If you want to delete the commit instead of amending it, please see *Removing a commit* earlier in this paper.

## Summary

While Mercurial remains a great choice for distributed version control, it's clear Git is far more widely used. Git's popularity continues to grow, due in part to its adoption as a standard by Microsoft and the corresponding popularity of GitHub, which was recently acquired by Microsoft. Today, knowing how to use Git can be considered an essential skill for software developers.

If you're currently using Mercurial and want to stick with it, fine. Mercurial is still under active development and is currently being ported from Python 2 to Python 3, a good sign for its future. But it's increasingly likely you'll run into Git at some point in your career, whether by choice or by circumstance, so it's well worth your while to learn how to use and become comfortable with it.

## Resources

### Papers

*Multi-track Development Strategies in DVCS*, Rick Borup, Southwest Fox 2013

<http://bit.ly/NHgonB>

*Version Control Face-off – Git vs Mercurial*, Rick Borup, Southwest Fox 2015

<http://bit.ly/1VspWTJ>

### Online

Pro Git (online version of the book by Scott Chacon and Ben Straub)

<https://git-scm.com/book/en/v2>

Git Documentation

<https://git-scm.com/docs>

Git Manual (MAN) pages – local hard drive

<file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/>

Atlassian Git Tutorials

<https://www.atlassian.com/git/tutorials>

Git Branching – Branches in a Nutshell

<https://git-scm.com/book/en/v2/Git-Branching-Branched-in-a-Nutshell>

The Hg-Git extension for Mercurial

<http://hg-git.github.io/>

Open Source Guides

<https://opensource.guide/>

Git Do's and Don'ts

<https://blog.axosoft.com/git-dos-donts/>

Hub – the command line wrapper for GitHub

<https://hub.github.com/>

### Books

*Version Control with Git, Second Edition* by Jon Loeliger and Matthew McCullough (O'Reilly), Copyright 2012 Jon Loeliger, ISBN 978-1-449-31638-9

*Git Pocket Guide* by Richard E. Silverman (O'Reilly), Copyright 2013 Richard Silverman, ISBN 978-1-449-32586-2

*GitHub for dummies* by Sarah Guthals, PhD and Phil Haack, Copyright 2019 John Wiley & Sons, Inc., ISBN 978-1-119-57267-1

*Jump Start GIT* by Shaumik Daityari, Copyright 2015 SitePoint Pty. Ltd., ISBN 978-0-9943469-2-6 (ebook)

## Biography

*Rick Borup is owner and president of Information Technology Associates, LLC, a professional software development, computer services, and information systems consulting firm he founded in 1993. Rick earned BS and MBA degrees from the University of Illinois and spent several years developing software applications for mainframe computers before turning to PC database development tools in the late 1980s. He began working with FoxPro in 1991, and has worked full time in FoxPro and Visual FoxPro since 1993. He is a co-author of the books *Deploying Visual FoxPro Solutions* and *Visual FoxPro Best Practices For The Next Ten Years*. He has published articles in *FoxTalk*, *FoxPro Advisor*, and *FoxRockX* and is a frequent speaker at Visual FoxPro conferences and user groups. Rick is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in Visual FoxPro.*

*Copyright © 2019 Rick Borup. Windows® is a registered trademark of Microsoft Corporation in the United States and other countries. Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries. Linux® is the registered trademark of Linus Torvalds in the United States and other countries. All other trademarks are the property of their respective owners.*

## Appendix A

This sample file represents the folders and file types you may typically want version control systems to ignore for Visual FoxPro projects. It can be used either as gitignore for projects under Git version control or as hgignore for projects under Mercurial.

Lines beginning with a hash symbol (the # character) are comments. The lines for the VFP Project Files and VFP binary source code files are included in this sample file for reference but are commented out. If you use this file as is, those file types will not be ignored. If you want to exclude those file types, remove the comment symbol on the appropriate lines.

The entries are case sensitive, which is why most appear in both lower case and upper case.

```
## Ignore selected files in a Visual FoxPro project.

#-----
# VFP Project Files
#-----
#
# Comment out or remove this section if you want
# these files to be included in the repository.
#
#*.pjt
#*.PJT
#*.pjx
#*.PjX

#-----
# VFP labels, menus, reports, screens, and class libraries
#-----
#
# Comment out or remove this section if you want these
# files to be included in the repository.
#
#*.lbt
#*.LBT
#*.lbx
#*.LBX
#*.frt
#*.FRT
#*.frx
#*.FRX
#*.mnt
#*.MNT
#*.mnx
#*.MNX
#*.mpr
#*.MPR
#*.sct
#*.SCT
#*.scx
#*.SCX
#*.spr
```

```
##*.SPR
##*.vct
##*.VCT
##*.vcx
##*.VCX

#-----
# Common VFP project subfolders
#-----
[Bb]mps/
[Dd]ata/
[Ii]cons/
[Ii]mages/
[Tt]emp/
[Tt]est/
[Ww]izards/

#-----
# VFP database files
#-----
*.dbc
*.DBC
*.dct
*.DCT
*.dcx
*.DCX
*.dbf
*.DBF
*.cdx
*.CDX
*.fpt
*.FPT
*.idx
*.IDX

#-----
# VFP compiled files and other binary code files
#-----
*.app
*.APP
*.dll
*.DLL
*.exe
*.EXE
*.fll
*.FLL
*.fxp
*.FXP
*.mpx
*.MPX
*.ocx
*.OCX

#-----
# VFP backup and other misc files
```

```
#-----  
*.err  
*.ERR  
*.log  
*.LOG  
*.mem  
*.MEM  
*.tbk  
*.TBK  
*.vue  
*.VUE
```

```
#-----  
# WLC Project Builder backup files  
#-----  
*.ftk  
*.FTK  
*.fxk  
*.FXK  
*.ltk  
*.LTK  
*.lxk  
*.LXK  
*.mtk  
*.MTK  
*.mxk  
*.MXK  
*.stk  
*.STK  
*.sxk  
*.SXX  
*.vtk  
*.VTK  
*.vxk  
*.VXK
```

```
#-----  
# Common graphics files  
#-----  
*.bmp  
*.BMP  
*.cur  
*.CUR  
*.gif  
*.GIF  
*.ico  
*.ICO  
*.msk  
*.MSK  
*.png  
*.PNG  
*.jpg  
*.JPG  
*.tif  
*.TIF
```

```
#-----  
# Common documentation and Microsoft Office files  
#-----  
*.chm  
*.CHM  
*.doc  
*.DOC  
*.dot  
*.DOT  
*.hlp  
*.HLP  
*.rtf  
*.RTF  
*.pdf  
*.PDF  
*.xls  
*.XLS  
*.xlt  
*.XLT  
  
#-----  
# Other misc files  
#-----  
*.bak  
*.BAK  
*.bat  
*.BAT  
*.dat  
*.DAT  
*.old  
*.OLD  
*.orig  
*.ORIG  
*.sav*  
*.SAV*  
*.tmp  
*.TMP  
*.wav  
*.WAV  
*.zip  
*.ZIP
```