Tracking Your Changes: Version Control Using Git And GitHub

September, 2021

Approximate Completion Time: 3 hours

This tutorial is designed to teach the basics of Git and GitHub. The two topics are presented separately: first a Git section teaches the use of Git alone by developing a simple application in a local repository; then a second GitHub section repeats the same development process, this time using GitHub as the primary repository, with a local Git repository as a working directory.

The tutorial models the development of a very simple PHP-based film ratings application.The actual coding is simulated: a **code-dev** folder is provided that contains all the required code, that can be copied into the Git repository as directed. This is not actually required: if preferred, you can use the tutorial with your own code. (Instructors might like to replace the application with something appropriate to their own course material).

Since the tutorial is designed for first year programming students, no previous coding or file management experience is assumed.

The tutorial is provided as-is, and can be modified as needed to suit your own purposes. You are asked to acknowledge the source in any publication that uses this material, whether modified or not.

Mike O’Kane, September, 2021

Table of Contents

Introduction

Version control is critical for all software development, allowing you to maintain a coherent record of ongoing modifications and release versions, develop and merge application components, collaborate with others, revert to previous versions, distribute your work, and much more. In this chapter you will learn about a widely used Version Control System (VCS) known as Git, and a related online repository, GitHub. Much of the chapter consists of two hands-on tutorials: the first demonstrates how to install Git and create a Git repository on your local computer to develop a small application (similar to the other applications that you developed for the exercises in this book); the second tutorial will more or less repeat the same steps as the first but this time taking advantage of the resources available in GitHub. Together these two tutorials will introduce key terminology and procedures, and provide you with a working knowledge that you can put to immediate use in your own work.

What is Version Control?

Perhaps the best way to introduce this topic is to consider how you might worked on code exercises without the use of version control software. If you save your files a number of times as you develop your code, you may have quickly learned to use a slightly different file name each time so you could retrieve a previous version if you got stuck. For example if an exercise required work on a file named **software-order.php**, you might have saved your first version as **software-order-1.php**, the second as **software-order-2.php**, and so on. You may have also learned to make a note in the comment section of each file to remind yourself of the changes that you made since the previous version. Once you completed the exercise the final version could then be saved or renamed as **software-order.php** and the other versions could be deleted (or saved if you wanted to keep a history of your work).

This is a basic version control system: saving multiple copies of your development files provides you with a history of changes to the application as you worked on it, and allows you to revert to a previous version if you need to. But this method can quickly become overwhelmingly complex as you lose track of which versions contain successul and unsuccessful code components, and what specific changes were made to each file. Even with a small application, you can quickly end up with a lagre number of files containing mostly dupplicted code. And if your application consists of multiple files, it will be almost impossible to develop a naming system that allows you to synchronize changes made too all files for the next version.

Even a simple real world application demands a more comprehensive Version Control System (VCS). As a developer, you will need to: track and synchronize changes made to multiple files (code, stylesheets, media, documents, data, etc); compare and retrieve different versions as needed; maintain a complete version history of the project; develop some requirements independently of others (merging them only when the work on each one has been successfully completed and tested); collaborate with others at each stage (design, coding, testing, review, maintenance, etc).

The good news is that a number of VCS's are available, providing standard tools and procedures that simplify your work flow and make it easy to collaborate with others. Currently the most popular is known as Git. Git is often used in conjunction with a cloud-based repository and development environment known as GitHub.

Git and GitHub

Git is a widely used, open source, and freely available VCS that runs on Windows, macOS, and Linux. Git was created by Linus Torvalds in 2005 when he was engaged in developing the Linux kernel. Git is designed to track changes across your entire application development by taking a "snapshot" of changes to your files (as well as added or deleted files) whenever directed to do so. Git can recreate any previous version by referring to this record of changes. Once Git has been installed, you can define any application work folder to function as a Git **repository**, or **repo**, which then allows you to apply Git commands to the files in the folder.

While Git functions very well as a standalone tool for local development, it is often used in conjunction with a cloud-based hosting and version control service called GitHub. GitHub is founded on Git tools and facilitates code sharing and distribution, project planning and management, version-based documentation, collaborative development, and remote storage. It is very common to use Git and GitHub together; a standard approach is to maintain your application's "home" repository in GitHub while downloading (**cloning**) a version to your local git repository which you will use to make changes (this ensures that your modifications are developed separately from the primary version that is maintained in GitHub). These changes can be uploaded (**push**ed) to GitHub at any stage, as often as needed, and, once in GitHub, may be shared, reviewed and tested by others before being merged to be incorporated into the primary version of the application. Similarly, you can download (**pull**) components, or complete versions, from GitHub as needed. While GitHub is an excellent tool for collaboration, it also serves very well for developers working alone. Additionally GitHub serves as a hosting service, allowing developers to share their work worldwide.

GitHub is a proprietary product (owned by Microsoft) and offers a number of priced plans, which are generally directed at larger scale development, however you can sign up for a free account that provides a comprehensive set of features at no cost.

There is an initial learning curve to become accustomed to the terminology and features of Git and GitHub but the effort is well worth it. Familiarity with these tools not only adds a key resource to your work process, but also increases your professional value.

Learning Git

In order to use Git, you will need to install the software and become familiar with Git procedures and terminology. These will be presented in the form of two hands-on tutorials, working with a small PHP-based application named **film-ratings** that allows users to view and submit film ratings, and add new films to the list of films that can be rated. The application is very simple, similar to the kind of exercises that you might have worked in a beginning programming course. You won't have to develop any code yourself since our only concern here is to demonstrate how an application can be developed within a Git repository. The actual coding is simulated: the **code-dev** folder that is provided with this tutorial contains all the required code and the tutorial tells you when to copy material to the repository, as if you were developing the code yourself (of course, if you prefer, you can use your own code with the tutorial).

This first tutorial will demonstrate how to use Git to develop **film-ratings** entirely locally, on your own computer. You will learn the Git commands needed to: track and manage modifications, document your changes, go back to earlier versions, develop components separately from the main version, and merge these components when you are ready to do so.

Once you are familiar with basic Git terms and operations, the second tutorial will repeat the same development, showing how Git and GitHub can be used together: this time **film-ratings** will be based and managed in GitHub, **cloned** to a local Git repository so you can develop the code in your own work environment, and then **push**ed back to GitHub for review, and for merging with the production version. You will also learn how to **pull** content down to your local repository from GitHub as needed.

Before beginning the first tutorial, let's review some basic Git procedures and terminology.

General Terminology

A Git repository, or **repo**, applies version control to all changes made to the files that are identified as a part of an application. These files may include code, documentation, images, data files, style sheets, notes and references, etc. All of the application files must be stored under a single project folder, or working directory, which may include any number of nested sub-folders. The Git **init** command must be used to initiate a folder as a Git repository.

The Git version control process is based on a series of "snapshots" of your application development: each snapshot records the changes you have made since the previous snapshot. As you work, you will be adding code, modifying existing code, deleting code, experimenting, often working on multiple files. When you wish to preserve your recent changes you will take a new snapshot. These snapshots are called **commits** in Git terminology, because each snapshot commits your recent changes to the repository.

Since you are likely to be working on multiple files between commits, and since you may not necessarily want all changes to all files to be included in the next commit, Git provides a two-step process. Before issuing a commit, you first need to **add** selected files to a "staging area", to indicate that the modifications made to these files since the last commit are to be included in the next commit. Among other things this means that you can add individual files to the stage as soon as you finish working on them. Files that haven't been changed, or that contain changes that you don't wish to be included in the next commit, are not added to the stage. When you have added all the files to the stage that contain changes since the last commit that you wish to be recorded, you will issue a **git commit** command. Git will then record changes to files that are listed in the staging area, and will only record changes that were made to a file before it was added to the stage.

Staging ensures that you can design each commit to only track changes made for a specific purpose, such as completing a code section, adding a new feature, fixing a bug, or trying something out. Grouping your modifications in this way allows you to document each commit by purpose, providing a clear history of your development, and making it easy to review, or return to, a commit that contains a specific modification.

It is not unusual to add a file to the stage and then realize it needs some more work before it's ready for the next commit. That's not a problem: you can add the same file to the stage multiple times, in which case only the changes since the last commit that appear in the latest version will be included in the next commit.

A unique sequence of 40 hexadecimal characters is generated to identify each commit, and when you issue a commit you will also be invited to provide a summary to document its purpose.

Once a commit is executed the stage is cleared so you can start adding files for your next commit.

Imagine a sequence of these commits, forming a version history of changes to your application. This sequence is called a **branch**, and every repository includes a **main** (or **master**) branch that tracks changes that have been applied to your core application. A pointer called the **head** points to the latest commit on a branch. If you are not happy with your most recent commit(s) and want to revert to a previous commit in the branch you can use the **reset** command to re-point the head to the earlier commit, which then becomes the latest commit (as you will see the reset command includes options to determine whether or not the contents of actual files are rolled back when you do this).

A single repository can contain multiple branches. A **main** branch may be all you need for a small application but it's often a good idea to develop each component or stage of your application on a separate "side" branch. That way you can track a sequence of new commits on this side branch, and only merge these back into the main version when the component has been successfully completed.

A new branch is created with the Git **branch** command and, once created, you can switch to the branch using the **checkout** command. Remember that the **main** branch contains the history of changes (commits) that have been made to the main version of your application, and that the head points to the latest commit. When you create a new branch this branch initially points to the same head as the **main** branch. Once you checkout a new branch , your future commits will be added to the new branch, until you checkout to a different branch. You can checkout different branches as needed, just remember that your commits are always applied to the branch that you have currently checked out. This may sounds confusing; let's say you're working on the **film-ratings** application, and you've added a branch named **addFilms** to develop code for a new feature that allows users to add new films; you checkout the **addFilms** branch and make commits as you work on this feature; while you're working on this you notice a minor bug in the code that is unrelated to the **addFilms** feature; since you don't want this to be recorded as a commit in your **addFilms** branch you checkout the **main** branch so you can commit the bug fix there; then you checkout the **addFilms** branch again to continue working on your feature.

When (and if) you are satisfied with your modifications in any branch you can tell Git to merge the branch commits with the commits in the **main** branch. If you haven't added any commits to the **main** branch since you started work on the side branch, this is a simple merge operation: the branch commits are sequentially added to the **main** branch, and the latest of these becomes the new head of the **main** branch. If commits have been added to the **main** branch as well as to the side branch, then Git will need to test that there are no conflicts (for example that the same lines have not been changed in both branches). If any conflicts are found Git will report these to you so you can resolve them. Once the merge process has been successfully completed, you can delete the new branch since the branch commits are now included in your main branch development.

If you are working on a relatively small project you are likely to develop side branches sequentially as follows: create a side branch if you have a significant task to perform; issue commits on the side branch while you work through your task; and, once the task has been completed, merge the commits from the side branch into the main branch and delete the side branch. Repeat this process for each task until all your application requirements have been satisfied.

Often however you will want to work on multiple branches in parallel, in order to develop different parts of your application at the same time. For example you may be working on one side branch to develop a new feature, another to modify the CSS, and another to troubleshoot a bug of code of some kind. You will need to checkout different branches while you work so that your commits will record changes in the correct branches. Parallel work on multiple branches is a standard approach in cases where a team of developers is working together on an application: different team members can work on different branches at the same time. In more complex applications you can even create branches of branches.

Don't worry if this is a bit confusing when you first read it. Things will be clearer once you work through the tutorial.

Master or Main?

Until recently the primary branch in a Git repository was by default named **master**. Appropriate concern has been expressed about the use of this name given its past association with slavery. In 2020 GitHub changed its default branch name from **master** to **main**, but some Git versions still use **master**. The examples in this tutorial will all refer to **main**, simply substitute **master** if that name is used by your version.

Summary of Common Git Commands

Here is short summary of the Git commands mentioned above, and a few others that you will also use in the tutorial:

`git init` to establish your project’s working directory as a Git repository.

`git add` to add files to the Git staging area, in preparation for the next commit.

`git status` to see a list of the files that have already been added to the staging area, and also a list of files that have been modified but not yet added to the stage.

`git commit` to create a snapshot of all the changes made to the files that are listed in the staging area at the time that each file was added.

`git branch` to create a separate branch for your development work, allowing you to commit changes to the new branch instead of the main branch.

`git checkout` to switch to a different branch. When you do this, your future commits will be associated with this branch, until you checkout a different branch.

`git merge` to merge your commits on one branch into another branch (usually the main branch). If Git discovers a conflict (changes made to the same lines in both files) it will stop the merge and alert you of the conflicts so you can resolve them.

`git diff` command to compare differences of all kinds (for example, to view the changes you have made to a file since it was added to the stage, or to view differences between two different commits).

`git reset` to return to (restore) a previous version of your application. This command includes and option to remove all changes made in your application files since that previous version.

`git log` to view a history of your commits. The history includes the ID and description or each commit, which is useful since you may need this information when issuing other Git commands

Each of these commands have many options and arguments allowing a great deal of customization, and there are also a number of other Git commands available for your use.

A Hands-On Git Tutorial

NOTE: Your **code-dev** folder should include two folders named **git-files**, and **github-files**. You will use the files in the **git-files** folder to develop a simple film rating application. As you work through the Git tutorial you will be told to copy different files to your new **git-tutorial** repository to simulate developing the **film-ratings** application. Before you get started you should look through these files. (If you would like to run the initial version of the application to see what it does, just copy the **git-files** folder into the **htdocs** folder of any local or hosted Web server that includes a PHP processor, and open **film-ratings.html** in any Web browser, for example **http://localhost/git-files/film-ratings.html**. Feel free to rate some of the films).

Install Git

Before you can use Git you will need to install it on your computer, if it is not already installed. There are a number of third-party programs available that provide a graphical user interface to Git, but the best way to get started is to work directly from the command line. The latest installation instructions for all platforms can be found at **https://github.com/git-guides/install-git** and these instructions include a test just in case Git is already installed on your system.

To test that your installation was successful, open a Terminal window and type the following at the command prompt (a separate tutorial is available for help using the command line):

```
git version
```

You should see a message something like this:

```
git version 2.17.1
```

*NOTE: Some macOS users may receive an **xcrun** error message (this is because some macOS updates did not include an update to **Xcode Command-line Tools**):*

```
xcrun: error: invalid active developer path
(/Library/Developer/CommandLineTools), missing xcrun at:
/Library/Developer/CommandL occurring ineTools/usr/bin/xcrun
```

The following solution should prevent this error from occurring:

https://stackoverflow.com/questions/52522565/git-is-not-working-after-macos-update-xcrun-error-invalid-active-developer-pa

Create a Work Folder for your Application

Before you can create a Git repository you must create a folder where you will develop your application. In your File Manager navigate to the folder that you want to contain your repository and create a folder named **git-tutorial** (if you have a local Web server installed you might want to locate this under your **htdocs** folder).

Create a Git Repository

Now let's initialize your **git-tutorial** folder as a git repository. To do this you must first navigate to the **git-tutorial** folder at the command line, using your operating system's change directory (**cd**) command. Check the location of the **git-tutorial** folder in your file system and then use the **cd** command to change to that location. For example, on macOS, if you created the git-tutorial folder under **Applications** your **cd** command might look like this:

cd /Applications/git-tutorial

(you will need to tweak this if your **git-tutorial** folder is in a different location)

On Linux if you created the git-tutorial folder under the htdocs folder of a local Web server located under **/opt/lampp** you might use:

cd /opt/lampp/htdocs/git-tutorial

(you will need to tweak this if your git-tutorial folder is in a different location.)

On Windows, first change to the correct drive letter if necessary (for example if your **git-tutorial** folder is located on a USB drive on drive **E:**, at the command prompt type **E:** and then press **Enter**; you should see the command prompt change to **E:)** Once your prompt indicates the correct drive, use the **cd** command to change your location to the **git-tutorial** folder, for example (the tilde **~** indicates your home folder):

cd ~\Desktop\git-tutorial

(you will need to tweak this if your **git-tutorial** folder is in a different location on the drive.)

Your command prompt should now indicate that you are in the **git-tutorial** folder. To initialize this folder as a Git repository, type:

```
git init
```

This command will create the repository and you should see a message stating that the repository has been initialized.

Let's check the status of your repository:

```
git status
```

You will receive a response something like:

```
On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

As you can see, your **git-tutorial** repository is currently empty but note that Git has already created a main branch named **main** (or **master**), ready for your to begin work.

NOTE: if you get an error message when you issue a git command, you might have forgotten to include git when you typed the command, for example **git status** and not just **status**.

Now let’s simulate the process of developing the **film-ratings** application by making a series of “modifications” (you will just simulate each modification by transferring code from your **code-dev/git-files** folder). Each modification will be committed to the repository and you can check and manage your commits as your work progresses.

Code Development for First Commit

In this first step, five application files are to be created: **film-ratings-view.php** contains the code to view film ratings; **film-list.txt** contains the list of available film titles; **film-ratings-list.txt** file contains a list of ratings that have been submitted; **film-ratings-submit.php** contains the form to submit a new film rating; and **film-ratings-add.php** contains a first draft of the code to process the film-submit form. In real life you would develop these materials yourself but for this simulation you will simply copy these five files from your **code-dev/git-files** folder into your **git-tutorial** folder (don't copy any of the other files yet). You can use your File Manager to copy the files or use the **cp** command at the command prompt.

Copy these five files now (in alphabetical order these are: **film-list.txt**, **film-ratings-add.php**, **film-ratings-list.txt**, **film-ratings-submit.php**, **film-ratings-view.php**

Now let's check the status of your Git repository again (first check the command prompt to be sure that you are located in your **git-tutorial** folder and use the **cd** command to change to this folder if necessary):

```
git status
```

You will see something like this:

```
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
     film-list.txt
     film-ratings-add.php
     film-ratings-list.txt
     film-ratings-submit.php
     film-ratings-view.php
nothing added to commit but untracked files present (use "git add"
to track)
```

Git is now reporting that there are five files in the repository but indicating that none of these files are being tracked yet. These five files will make up the initial commit version of this repository, so let's add all of these files to the staging area in preparation of your first commit. We will use the **add** command to do this, and note that the command is followed by a space and then a period, which is shorthand to indicate that all the files in the repository are to be added (this would include any sub-folders and their contents if there were any):

```
git add .
```

These files should now be added to the stage; let's issue the **status** command again to check:

```
git status
```

You will see a list of the files that have been added to the stage and are ready to be committed. All five files are listed since we added all the files in the repo.

```
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
     new file:   film-list.txt
     new file:   film-ratings-add.php
     new file:   film-ratings-list.txt
     new file:   film-ratings-submit.php
     new file:   film-ratings-view.php
```

Now you are ready to issue our first commit.

Issue your First Commit

Use the **git commit** command to commit all of the files that have been added to the stage:

```
git commit
```

When you issue a **commit** command you are expected to include a message that describes the purpose of the commit. You can include this in the commit command (see below), but if you don’t Git will open a command line editor called **Less**, which runs in your Terminal window, so you can type your message there. These messages are usually quite short but the editor is useful if you want to write a longer message. For this first commit, when the editor opens, just type “Initial Commit” (it is important to type this as the very first line in the editor window, above the “# Please enter” prompt). To save your message press **Ctrl** and the **S** key, then to quit the editor and return to the command line, press **Ctrl** and **X**.

In future you can avoid using the editor by typing your message when you issue the git commit command: just include an **-m** switch, followed by the message, for example:

```
git commit -m "Initial Commit"
```

If you use **-m** to provide your message the editor will not appear.

When you quit the editor the commit will execute and you will see a short report that includes a hexadecimal ID (in this example the ID is **845fed6** but your ID will be different) , which identifies this commit, followed by the message (“Initial Commit”) that you provided.

```
[main (root-commit) 845fed6] Initial Commit
 5 files changed, 159 insertions(+)
 create mode 100644 film-list.txt
 create mode 100644 film-ratings-add.php
 create mode 100644 film-ratings-list.txt
 create mode 100644 film-ratings-submit.php
 create mode 100644 film-ratings-view.php
```

(The commit ID listed here is actually a short version of a longer 40-character ID. You will see the complete ID shortly when you view a history of your commits.)

Code Development for Second Commit

You have just committed the files that had been added to the stage, and the stage is now empty so you can prepare for your next commit. For this commit step you will complete work on **film-ratings-add.php**. To simulate this, use your text editor to first **open film-ratings-add.php** in your **git-tutorial** folder. Next open **film-ratings-add-step-2.txt** in your **code-dev/git-files** folder and follow the instructions in this file to copy the PHP code from **film-ratings-add-step-2.txt** into **film-ratings-add.php** in your **git-tutorial** folder. Let's assume that you have now completed the modifications to this file so save **film-ratings-add.php** with the changes, and add this file to the stage (first check the command prompt to be sure that you are located in your **git-tutorial** folder and use the **cd** command to change to this folder if necessary).

Since you are just adding a single file to the stage, you can specify the name of the file to be added:

```
     git add film-ratings-add.php
```

If you wish use **git status** to check that the file has been added to the stage. Now issue your next commit, this time with the message "Completed work on film-ratings-add.php" and including this message when you issue the commit command:

```
git commit -m "Completed work on film-ratings-add.php"

[main 5097b06] Completed work on film-ratings-add.php
 1 file changed, 12 insertions(+), 1 deletion(-)
```

Note that this commit is identified with a new ID and the message that you provided. Note also that the commit has been added to the main branch, so this is now the latest commit on this branch. You now have a history of two commits.

Code Development for Third Commit

Next you will "develop" two more files: **film-ratings.html** which provides a front-end that allows the user to choose what they want to do, and also **film-ratings.css** which will set the styles for the application. To simulate this work, copy these two files from your **code-dev/git-files** folder into your **git-tutorial** repository. Now let's add these two files to the stage and issue a new commit with the message "Added home page and style sheet" (first check the command prompt to be sure that you are located in your git-tutorial folder and use the cd command to change to this folder if necessary):

```
git add film-ratings.html
git add film-ratings.css
git commit -m "Added home page and style sheet"
```

Code Development for Fourth Commit

Now let’s modify a file so that we can compare different versions of the same file that were recorded in different commits. Use any text editor to open **film-list.txt** and change one of the existing titles in the list to a film of your choice. Now add another film title to the end of the list (be sure to add this title on a new line and don’t leave blanks lines in the list of titles). Save and close the file. Now add this file to the stage and issue another commit with the message “Added new film titles”

```
git add film-list.txt
git commit -m "Added new film titles"
```

You now have four commits registered on your main branch and these reflect a series of changes that have been made to your application at different stages. Note that these changes include modifications to existing files as well as addition of new files. If you had been developing these yourself, you might have issued a larger number of commits in order to take more snapshots of your modifications as you worked but these four will give you the general idea. Let’s look at a history of your commits and view the differences between two of these commits.

Viewing the Log of Your Commits

You can use the **git log** command to view a history of your four commits on the **main** branch. Note that the log output displays these with the latest commit listed first. For each commit the log lists the complete (40 character) commit ID, the author, date, and your description. You will also see the message “(HEAD - > main)” at the end of the first line of the latest commit, indicating that the head is currently pointing at this commit and branch. Note also that, although the log displays the entire 40-character ID hash for each commit; you can refer to any commit using a shortened ID, consisting of just the first 7 characters (for example the shortened ID of the most recent commit is **ca0a442**, and the ID of the third commit is **caf03ac**).

```
git log

commit ca0a442b5a5c892765b0219810fb3edece39aa59 (HEAD -> main)
Author: Mike O'kane <mike@mikeokane.com>
Date:   Thu Mar 18 13:01:31 2021 +0000
    Added new film titles

commit caf03acae0974ff25f6a2c82585c6dacfc975006
Author: Mike O'kane <mike@mikeokane.com>
```

```
Date:   Thu Mar 18 12:58:24 2021 +0000
    Added home page and style sheet

commit 5097b06bcb068930188042860839 8f18f15244ac
Author: Mike O'kane <mike@mikeokane.com>
Date:   Thu Mar 18 12:25:33 2021 +0000
    Completed work on film-ratings-add.php

commit 845fed647d4a78af0cf4de627400149d7e872c18
Author: Mike O'kane <mike@mikeokane.com>
Date:   Thu Mar 18 12:12:45 2021 +0000
    Initial Commit
```

(You may need to press **q** to return to the command prompt.)

This list demonstrates how Git maintains a history of your changes along a branch of commits. Each commit documents the changes made since the previous commit, and you can refer to the log when you want to compare versions or return to a previous version.

Comparing the Differences Between Two Commits

The **git diff** command can be used in a number of ways, depending on the arguments that you provide. Let’s use it here just to see how you can view the changes between the last two commits (the only differences were the changes that you made to **film-list.txt**). Provide the shortened ID’s of these two commits, with the earlier version listed first. Be sure to use the ID’s of your own commits: refer to your git log results to obtain the IDs and just use the first 7 characters of each ID.

```
git diff caf03ac ca0a442
```

The results are displayed using the **Less** editor. You will see that the listing shows a comparison of two versions of the same file, which is why film-list.txt is indicated twice in the heading (diff allows many other comparisons). If changes had been made to more than one file between these two commits, these changes would be listed one file at a time.

```
diff --git a/film-list.txt b/film-list.txt
index 42a4bb4..0e2d2aa 100644
--- a/film-list.txt
+++ b/film-list.txt
@@ -1,4 +1,5 @@
 Avengers Endgame
 The Lion King
-Toy Story 4
+The Martian
 Joker
+The Dig
```

This comparison of the two commits displays “Toy Story 4” in red, preceded by a minus sign, indicating this text has been removed, while “The Martian” and “The Dig” are both displayed in green preceded by a plus sign, indicating that these have both been added. Note the order that the two commits are listed in the **git diff** command: the ID of the earlier commit is listed first, and the later commit is listed second: if you had listed the commits in the opposite order, “Toy Story 4” would have been shown to have been added and “The Martian” and “The Dig” to have been deleted. You can try this, it demonstrates that the **diff** command simply shows differences based on the order that you provide.

You can compare any two commits in your version history.

Restoring a Previous Version

Next let’s commit some more changes, but this time decide not to keep them. This example shows a very simple change in two files, just to explain how you can reset to a previous commit.

Using any text editor, open the style sheet, **film-ratings.css**, and change the **font-size** to **12** and the **color** to **blue** in the following line:

```
     p { font-size:10pt; color:black; }
```

Save **film-ratings.css**. Now open **film-ratings.html** and change the text in the **<h1>** heading from "<h1>Film Ratings</h1>" to "<h1>Movie Ratings</h1>". Save **film-ratings.html**. Now add these two files to the stage and issue a commit with the message "Minor Style Changes"

If you run **git log** again you will now see a history of five commits. Let’s say you get some feedback regarding these latest changes and decide that you don’t want them. You can effectively remove this commit by using the **git reset** command to reset the branch head to points to an earlier commit. If you add the **--hard** argument with the **reset** command, Git will perform a hard reset, which not only resets the head to an earlier commit but also removes all changes made to the actual files since the earlier commit. Since that’s what we want in this case let’s restore the second commit as the head commit and make this a hard reset:

```
git reset --hard ca0a442

     HEAD is now at ca0a442 Added new film titles
```

If you open **film-ratings.css** and **film-ratings.html** you will see that these files are now back in the state they were in at the time of the second commit, and the changes that you made before the fifth commit have been removed. If you want to remove the commit but **keep** the actual changes in the files, you can issue a soft reset:

```
git reset --soft ca0a442

Unstaged changes after reset:
M    film-ratings.css
M    film-submit.php
```

Since the two files still contain the changes made since what is now the new head commit, Git alerts you that these files have changes that have not been added to the stage.

(In case you're wondering why you might want to issue a soft reset, a common reason is that, after issuing a commit on the **main** branch, you decide that you would prefer your changes to be committed to a new branch (see next topic). A soft reset allows you to keep the changes to the files, remove the commit that was added to the main branch, create and checkout a new branch, and then commit the files again with the changes intact, this time to the new branch.)

Creating a New Branch

Let's consider a more significant modification to your application, Suppose you decide to add a new feature, allowing the user to submit new films to the current list of films that can be rated. This new feature requires three tasks: create a new file named **film-submit.php** that provides a form for the user to submit a new film; create a new file named **film-add.php** that receives the name of the film from the form and adds it to the file of films; and add a new link in **film-ratings.html** that allows the user to connect to **film-submit.php**.

It makes sense to do this work on a separate branch from the main branch. That way you can make changes and issue commits as you develop this feature without making changes to your main application on the main branch; once the feature has been successfully completed and tested you can merge the new branch with the main branch.

Use the **git branch** command to create a new branch, and let's call this branch **addFilms**:

```
git branch addFilms
```

Note that if you were to issue a **git log** command at this point, you see that the first line of the latest commit now indicates (HEAD - > main, addFilms). That's because the head is still pointing at the latest commit on the **main** branch, and this is also the latest commit on the **addFilms** branch at this time.

The addFilms branch will be created but note that we have not told Git that we now want to work in that branch. We do this using the git checkout command:

```
git checkout addFilms
```

You will see a message that you have been switched to the **addFilms** branch, so your future staging and commits will be associated with this branch. Note that you can combine these last two commands by adding the **-b** switch to the **checkout** command:

```
git checkout -b addFilms
```

This is the equivalent of **git branch addFilms** followed by **git checkout addFilms**.

Making Changes to Your New Branch

Use **git status** just to be sure you are working in the **addFilms** branch. Now let’s add the changes that will deliver a film submission feature. You can pretend you have done this work since the two files are already provided in your **code-dev/git-files** folder: just copy **film-submit.php** and **film-add.php** from this folder to your **git-tutorial** folder. You also need to modify **film-ratings.html** to include a link to **film-submit.php**. Actually the code for this link is already included in **film-ratings.html** but it doesn't display in a Web browser since it is hidden as a comment. To make the link visible, open **film-ratings.html** in your **git-tutorial** in a text editor and remove the leading **<!--** and the trailing **-->** from the following line:

```
<!-- <p><a href="film-submit.php">Submit a New Film</a></p> -->
```

The line should now look this:

```
<p><a href="film-submit.php">Submit a New Film</a></p>
```

Now save **film-ratings.html**.

You have now taken all the steps to add your new feature, and it's time to add these changes to the stage (first check the command prompt to be sure that you are located in your **git-tutorial** folder and use the **cd** command to change to this folder if necessary).

```
git add film-ratings.html
git add film-add.php
git add film-submit.php
```

Now commit these changes, using the **-m** argument to provide a message:

```
git commit -m "Feature to Add New Film Titles"
```

```
[addFilms af6d7dc] Feature to Add New Film Titles
 3 files changed, 78 insertions(+), 2 deletions(-)
 create mode 100644 film-add.php
 create mode 100644 film-submit.php
```

Note that the first line of Git’s response indicates the **addFilms** branch and not the **main** branch. If you issue a **git log** command now, you will see that the latest commit that is listed is the commit that you just added to the **addFilms** branch, and the head is listed as (HEAD -> addFilms). The previous commits in the list were made on the main branch. If you now issue a **git checkout main** command to switch to the **main** branch, and then issue **git log** again, you will no longer see this most recent commit but only the commits made to the **main** branch. This shows how Git keeps tracks of commits made to different branches.

Switching Between Branches

To save time, you have completed the work in **addFilms** by copying two files and making minor changes to another file; these changes were recorded in a single commit. In a real world situation, your work would probably be more extensive and the branch would likely consist of a number of commits. While working on a branch you might find that you need to also make some changes on your **main** branch, for example if someone notifies you of a bug, not related to your **addFilms** feature, that you want to fix immediately. You can easily switch between branches: in this case just **git checkout main**, make and commit the necessary changes on the **main** branch, then **git checkout addFilms** to continue working on the **addFilms** branch. You may also create other branches to work on different features at the same time; you can simply **checkout** each branch as needed. Be sure to check out the right branch when you do this, and be careful not to make changes on one branch that might conflict with changes made on another branch. Given the potential for conflicts it's usually best to work on the fewest number of branches at the same time if you can (more on this below).

Merging and Deleting a Branch

So you have now used a separate branch (**addFilms**) to develop a new feature and made a commit with your changes. Let's assume you are satisfied that your feature is ready to be merged into your **main** branch. To perform a merge, you must first checkout to the branch that you wish to merge into, in this case the **main** branch:

```
git checkout main
```

Now use **git status** just to be sure the head is now pointing to the main branch.

Now use **git merge** to merge the **addFilms** branch with the **main** branch:

```
git merge addFilms
```

Git will now attempt to merge the changes that were tracked in the **addFilms** branch with the current branch (**main**). There are three possible outcomes: (1) if the **main** branch has not had any new commits since the **addFilms** branch was created, the merge is very simple (known as a **Fast-forward**) since the commits in the **addFilms** branch are just added to the commits of the **main** branch; (2) if one or more commits have been made in the **main** branch since the **addFilms** branch was created Git will need to ensure there are no conflicts between the commits made in either branch before adding the **addFilms** commits to the **main** branch; and (3) if there are conflicts (for example commits in both branches include changes to the same lines in a specific file), then Git will suspend the merge operation and alert you of the conflicts so you can handle them.

In this case there should be no problems and in Git's response message that the merge was completed as a fast-forward operation. While you are still in the **main** branch, issue the **git log** command again. You will see that the commit that you issued while in the

addFilms branch is now included in the **main** branch. If you had made multiple commits in the **addFilms** branch these would all now be included in the **main** branch.

Deleting a Branch

Once the commits from a branch have been successfully merged into the **main** branch (or any other branch) the branch that has been merged is no longer needed and can be deleted using the **git branch** command again, but this time including the **-d** switch with the name of the branch to be deleted (be sure to checkout the main branch first, you can't delete a branch while you are in it):

```
git branch -d addFilms
```

Note that the **-d** switch specifies that the branch should only be deleted if it does not include any commits that have not yet been merged with another branch. Since you already merged the commits in **addFilms** the deletion will take effect. In some case you may want to force a deletion even if the commits have not been merged, for example you may have concluded that the changes made in the branch are not wanted and should not be merged. In that case you can use the **-D** switch (upper-case **D** instead of lower-case) to force a deletion, and lose the commits that were made in the branch:

```
get branch -D addFilms
```

Experiment with Git

You have now completed the Git tutorial. By now you should have a general idea of how Git works, and understand that your Git repository maintains a complete record of changes to your application in the form of a history or commits, providing a complete documentation of your development work. You should also have an idea how this can help with your own code development. There are lots of resources and tutorials available online to learn more about Git. For a much more complete explanation of Git commands try **https://git-scm.com/docs/**

Now take a breath, and when you're ready let's move on to the second tutorial and learn how to integrate a local Git repository with GitHub.

Introducing GitHub

You may have already used GitHub to download software or other materials. GitHub is a Web-based hosting platform designed to provide a global resource to support code development, version control, collaboration, code-sharing, and project management. Whereas Git is open source and freely available as a local resource, GitHub is a cloud-based Microsoft product and service. At the time of writing the basic GitHub service, including storage, is available at no cost, with additional services available at different fee levels. Anyone can create a GitHub account, and GitHub works very well in combination with local repositories. There are other online Git-based platforms, but we will focus on GitHub since it is currently the most widely used.

Once you have a GitHub account you can create, maintain, update, and manage your repositories online, while continuing to modify and update the content in a local Git repo, working with a downloaded (cloned) copy. This work model is very common: the stable version (often production version) of an application evolves on the main branch of the GitHub repository, and whenever work is to be undertaken, a branch is created for that purpose. The branch is usually cloned to a local repository for development, and when the work is complete it is pushed back up the GitHub branch, and “pull requests” are issued, inviting review, testing, and modifications, until the work on the branch is approved for merging with the main branch. This model works just as well for solo development as for larger projects involving teams, even projects that are open to anyone who wishes to become involved. Access to a GitHub repo can be kept private, shared with selected users, or made fully public. Collaboration is at the heart of modern application development and GitHub provides a range of tools.

Note that it is a great idea to list any involvement in collaborative efforts on GitHub on your professional resume: apart from the skill set this also indicates your willingness to engage and learn as a professional.

GitHub Tutorial

There is far more that can be done with GitHub than can covered here, but this tutorial will walk you through a simple development, based on the model described above. The tutorial will repeat the same development that you completed previously using Git alone, but this time will skip many of the Git steps since you are already familiar with them. As you begin the tutorial assume you are once again starting from scratch with the film-ratings application. Let’s start with a brief overview.

You will begin by signing up for a GitHub account and creating a new GitHub repository that will serve as your primary repository for the film-ratings application. You will then copy (**clone**) this GitHub repository to a local repository, and use the local repo to work on the files, just as you did before. This time you will upload (**push**) your local commits to the GitHub repository whenever you want to update your primary version and synchronize the two repositories. When you want to start a new branch you will create the branch in GitHub and then download (**pull**) it to your local repository. Once you have made some commits on this branch, you will push these back to the same branch in

GitHub, and when the work has been completed you will work in GitHub to: review and document the changes; merge the branch with the main branch; and delete the branch (you will also delete the local branch). Finally you will pull the **main** branch down from GitHub to your local **main** branch so that the two repos are again fully synchronized and ready for further development. This should give you a good basic idea of how GitHub can work with a local repository, and provide you with a useful template that you can apply to other development work. As the tutorial will hopefully show, using GitHub to host your primary application and Git for your actual work makes it easy to work with others, obtain feedback, distribute your application, and much more.

Three new **git** commands were just mentioned: the **git clone** command copies a repo to another location and also tracks the relationship between the two repos, which simplifies work between them; the **git push** command is used to upload commits to a remote system, in this case your GitHub repo; similarly the **git pull** command is used to download commits, to update the local repo with changes made on the remote repo.

Before you begin this tutorial you must have installed Git on your local computer and feel comfortable with the Git commands that were covered in the first tutorial. Since you will have a number of important pieces of account information to keep track of, it's a very good idea to create a text file with a suitable name (for example **GitHub-settings.txt**) and keep this file open while working through the tutorial so you can make notes and record account information.

Setting Up Your GitHub account

To create a GitHub account, go to **https://github.com**, and choose "Sign Up for GitHub". You will be asked to provide a username, email address, and password. Make a note of your user name and password. You will then be asked a few questions about your primary purpose and interests, and will need to confirm your email; be sure to do this, once your email has been confirmed your account will be created. You may now be asked what you want to do next, if so, select the option to create a new repository and then follow the instructions below. (NOTE: if your account creation process is different and you don't see an option to create a repository, once your account creation has been completed, just modify the URL in your browser's address window to **https://github.com/xxx**, where **xxx** is your chosen username. That will bring you to your home page. Click the "Repositories" button near the top of the page, and select "New" to arrive at the page to create a new repository. Now follow the instructions below.

Creating a New Repository

Before anything else, copy the URL of your GitHub account into your **git-settings.txt** file for your record (this will be **https://github.com/xxx**, where **xxx** is your chosen username).

Now do the following on the "Create a New Repository" page:

1. Choose a name for your new repository. Use **film-ratings** unless you would like to do this exercise using some code of your own.

2. Provide a short description, for example: “Exercise to learn about GitHub”
3. Select “Public” or “Private”: Choose “Private” for the moment (you can change this later if you prefer).
4. Check the “Add a README” file box.
5. Leave the other boxes unchecked.
6. Click “Create Repository”

The **film-ratings** repository will be created and you will see your repository’s home page. Note the URL in your browser address window, something like

https://github.com/xxx/film-ratings

where **xxx** is your chosen username. Copy this URL into your **git-settings.txt** file for future reference. You can use this URL to get back to this page if you ever get lost in GitHub.

Below the headings on this page you will see a window showing the contents of a **README.md** file, which is intended to introduce the purpose of this repository. Right now the file only contains the name of the repo. Click the pencil icon on the right side of the window to edit this file. Type the following description, or use your own description:

“film-ratings is a simple application to view or submit film ratings and to add new films to be rated. It is intended only as an exercise to learn to use git and GitHub.”

When you are done, scroll down and click the “Commit Changes” button. Your changes will be committed to become the first commit on the **main** branch of the **film-ratings** repository.

Creating a Personal Access Token (PAT)

Next you will need to obtain a Personal Access Token (PAT). This is a 40 character string that GitHub will generate to use (instead of your password) when you connect to your account from the command line. You only need one PAT for use with any number of repos. As a security precaution, GitHub automatically removes personal access tokens that haven't been used in a year.

To obtain a PAT, click the icon located in the upper-right corner of any page (this will be your profile photo if you added one), then click “Settings” in the user bar.

1. In the left sidebar that appears, scroll down and click “Developer settings”.
2. In Developer settings, in the left sidebar, click “Personal access tokens”.
3. In the Personal access tokens window click “Generate new token”.

To generate a new token:

1. Provide a note to indicate the token’s purpose, for example “command-line access”.
2. In the “Select the scopes, or permissions, you'd like to grant this token section”, just check “repo” so all of the repo actions are checked. You don’t need to check any other boxes.

3. Scroll down and click “Generate token”.

You will see a 40 character token. Be sure to copy and paste this entire string of characters into your **git-settings.txt** file and save the file; once you navigate off this page, you will not be able to obtain this token again.

Keep a secure backup of your **git-settings.txt** file so you don’t lose your PAT. As you will see when you complete the exercise, you will use this token instead of your GitHub user password every time you perform an operation at the command line of your local computer to interact with any repositories in your GitHub account.

Clone Your Application to a Local Repository

You now have a GitHub account and personal access token. Your **film-ratings** repository has been created on GitHub, and currently consists of just one file: **README.md**. Maintaining the primary (production) version of an application in GitHub confers many benefits, for example: a secure, high quality development environment; a standard distribution platform if you wish to make your application publicly available; well defined, standard procedures for collaboration with other developers and work groups; a range of tools to facilitate project design, work flow, documentation, etc.

Under this model you will continue to work on the content of your application locally, so your next step is to copy the initial version of the **film-ratings** repository to your own computer. The process of copying a repository to another location is called “cloning”. In order to complete this operation you will need your GitHub account user name (or email), your PAT, and the URL to your GitHub repo (**https://github.com/xxx/film-ratings**, where **xxx** is your GitHub user name).

Decide where you want to locate your new **film-ratings** repository. You can locate it in the same folder that contains your **git-tutorial** folder but this isn’t necessary since this is an entirely separate repository. Important: do not create a new folder for the **film-ratings** repository, the clone process will take care of that. Just identify the folder where you want the **film-ratings** repo to be cloned to.

Be sure that Git is installed and running on your computer. Open a Terminal window and type **git version** if you are not sure. At the command prompt use the **cd** command to change directory to the folder that you want to contain your repo folder. Be sure that the command prompt now shows that you are in the right folder.

Now, still working from your computer’s command line, issue the **git clone** command listed below. Note that **xxx** in the URL should be replaced with your own GitHub username.

```
git clone https://github.com/xxx/film-ratings.git
```

By default your new local repository folder will be given the same name as the GitHub repo (**film-ratings**) but it is not required to have the same name. If you want your local repository to have a different name just add a space at the end of the command, followed by your preferred name.

When you run the **git clone** command Git will ask for your user name and password (you can use either your GitHub username or your email for the user name). IMPORTANT: when asked for a password use your PAT token, not your GitHub account password. The token is far too long to type: open your git-settings.txt file, select the complete PAT, right-click and choose “Copy” from the drop down menu. Now return to the command prompt that is asking for a password: click the location where you must type the password so the password will be pasted in the right place, right-click and choose **Paste** from the menu, then press **Enter** (note that the password will not appear when you paste it).

You should now see a message something like this:

```
Cloning into 'film-ratings'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.29 KiB | 661.00 KiB/s, done.
```

Congratulations! You have just copied your repo down to your local computer. The clone process created a **film-ratings** folder, initialized this as a git repository, and copied the files and commits from the GitHub repo. Use **cd film-ratings** to change to the repo folder, and then use the **ls** command, to view the contents of the newly created **film-ratings** folder. You should see the **README.md** file that you created in GitHub.

Working in your Local Repo

Now that you have a working copy of your repo, it's time to start work and develop the application. Since you have already completed a tutorial showing how to use Git to develop the files for **film-ratings** on your local computer, let’s just assume you have already done all this work. Your **code-dev** folder contains a folder named **github-files** folder, which contains a duplicate set of the files in the **git-files** folder (which you worked on in the Git tutorial). To simulate development, copy all of the files from the **github-files** folder, except **film-submit.php** and **film-add.php**, into your **film-ratings** folder. There should be seven files in the folder.

In the Terminal window, make sure you are in the **film-ratings** folder (use the **cd** command if you need to). Now add all of these to the stage:

```
    git add .
```

And now issue a commit, with the name “Initial Development”:

```
git commit -m "Initial Development"
```

That was very rapid development! In reality of course this work would have taken some time, and may have required a number of commits as the work evolved. Now you have added these files to your local repository, your local repo and your GitHub are “out of synch” with each other. Usually as you work on your local files and commit changes it will be your decision when to upload (**push**) these local commits to GitHub in order to re-synchronize the two repos. You may want to push each commit as soon as it has been created locally, or you may want to push groups of commits all together, when you have completed specific development stages. Since our first local commit actually contains a lot of initial work on the **film-ratings** application, let’s push this to GitHub, remembering that in real life this push might actually contain multiple commits.

Pushing your Local Repo Commits to the GitHub Repo

Since the local **film-ratings** repository was cloned, Git keeps track of the source of the repo (in this case the **film-ratings** repo in GitHub). That means you don’t need to tell Git where to **push** these local commits and files. Instead you can just type **git push** at the command line, although you will still need to provide your username and PAT token. (If you wanted to push the files to a different repo you would need to provide the URL and login information to connect to that location).

Type **git push** now. Assuming you correctly enter your user name and password (PAT token), you should get a response similar to the following:

```
git push
Username:
Password:

Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 2.95 KiB | 1.48 MiB/s, done.
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/mickokane/film-ratings.git
   b34cb75..027e596  main -> main
```

Note that the last line of the response from Git indicates that the push operation moved the head of the main branch from the commit that was previously the head (in this case **b34cb75**), to the latest commit that was pushed (in this example **027e596**). Note also that these commit IDs are also synchronized, and are the same in both repos.

If you now go back to your GitHub site and look at the **film-ratings** repo you will see that it now contains all the **film-ratings** files.

Adding a Branch to your GitHub Repo

Now you are ready to add a new feature to your **film-rating** application. As you learned in the Git tutorial it is usually best to develop new material on a separate branch and only merge with the main branch when the work has been completed and tested. There are two ways to approach this. You can just create a branch locally (let’s call it **addFilms**), checkout **addFilms** and make your changes, then when you are ready merge your commits from **addFilms** with the local **main** branch, and delete the branch **addFilms**. This is the same procedure that you followed in the Git tutorial except that now you **push** the local **main** branch back to your GitHub **main** branch. All the branch work is done locally and your GitHub repo is simply updated when this work is completed. In this case GitHub is never "aware" of the **addFilms** branch.

The alternative is to create branch **addFilms** locally but then **push** the branch to GitHub so the branch is present in both repos. This way you can push commits from your local **addFilms** branch to the GitHub **addFilms** branch as you work. When you have fulfilled the requirements for this branch, and the GitHub branch is synchronized with the local branch, you can merge the GitHub **addFilms** branch into the GitHub main **branch**, then delete the **addFilms** from both GitHub and your local repo. This second approach has significant advantages: your branch development is fully documented in GitHub; you can "save as you go" (using the GitHub branch as a backup while you work); and, crucially, you can follow GitHub's approval process for merges, which helps to ensure that an appropriate review process is always undertaken before the main branch is changed. The merge review process is especially important when working with others, allowing your collaborators to view, test, provide feedback, make changes, and finally approve your work as needed before it is merged into the main branch.

Let's take this second approach, starting by creating a local branch and pushing this to GitHub.

Creating and Pushing a New Branch

In your Terminal Windows, create and checkout a new branch named **addFilms**:

You can do this using two separate commands:

```
git branch addFilms
git checkout addFilms
```

or with a single command:

```
git checkout -b addFilms
```

Here is the **push** command to push **addFilms** to your GitHub repo:

```
git push -u origin addFilms
```

The **-u** switch tells Git to set up tracking between the local and the remote **addFilms** branches (you will receive a confirmation of this in Git's response to your push operation). The origin argument tells Git to push the branch to the GitHub repository that the local repo was cloned from.

To test that the **push** operation was successful, go to your **film-ratings** page in GitHub. On the left of the window, below the headings and above the list of files, you will see a button indicating that you are currently viewing the **main** branch. Click this button and you should see that the **addFilms** branch is also listed. Select the **addFilms** branch, and the button will change to show that you are now viewing this branch (just as if you issued git checkout **addFilms** at the command line to switch to your local **addFilms** repo). You will see a note the **addFilms** branch is “even with main”, that’s because at this point you have not issued any new commits on the **addFilms** branch. Note that you can switch between (checkout) branches in GitHub just by clicking the branch button and choosing a different branch.

If you need to rename or delete a branch, click the “2 branches” indicator next to the branch button: you will see a list of your branches; each listing includes a pen icon (rename) and bin icon (delete), and also a “New pull request” button, which will be discussed shortly.

Now that you have pushed (with tracking) your **addFilms** branch to GitHub let’s start working on the local branch.

Working in your Local Branch

To start working in your local **addFilms** branch, first checkout the branch:

```
git checkout addFilms

Switched to branch 'addFilms'
Your branch is up-to-date with 'origin/addFilms'.
```

You will now make the same modifications on this branch that you made when working through the Git tutorial. This requires two steps:

Step 1: use your File Manager (or the **cp** command) to copy **film-submit.php** and **film-add.php** from your **coursework/chapter17/GitHub-files** folder to the **film-ratings** folder.

Step 2: in your **film-ratings** folder, open **film-ratings.html** in a text editor. Just as you did when working through the Git tutorial, remove the leading **<!--** and the trailing **-->** from the line:

```
<!-- <p><a href="film-submit.php">Submit a New Film</a></p> -->
```

The line should now look this:

```
<p><a href="film-submit.php">Submit a New Film</a></p>
```

Now save **film-ratings.html**.

For this tutorial let’s assume you have now completed your work on this branch and it's time to add and commit them (first check the command prompt to be sure that you are located in your **film-ratings** folder and use the **cd** command to change to this folder if necessary):

```
git add film-ratings.html
git add film-add.php
git add film-submit.php

git commit -m "New Feature: Add a Film Title"
```

Your changes will be recorded, and you will see a message something like this:

```
[addFilms aa68f78] New Feature: Add a Film Title
 3 files changed, 78 insertions(+), 1 deletion(-)
 create mode 100644 film-add.php
 create mode 100644 film-submit.php
```

This commit contains all of the required modifications for the **addFilms** branch, and it's time to push the commit to the **addFilms** branch in GitHub. Keep in mind that this is just a simple example to familiarize you with the process: a **push** command will push all commits made on a branch since the previous push, and commits may be pushed multiple times during the branch development.

Pushing Branch Commits

To push the commits in your **addFilms** branch (use **git branch** to check that you are working in the **addFilms** branch; if you are in the **main** branch, first issue **git checkout addFilms**).

```
     git push
```

Since we already checked out the **addFilms** branch Git will push the commits from this branch and not from the **main** branch. Also since the branch is being tracked, Git knows that the commits should be pushed to the **addFilms** branch in your GitHub film-ratings repo.

Go back to GitHub and compare the **main** and the **addFilms** branches; you will see that the **addFilms** branch now reflects the latest commit (named "New Feature: Add a Film Title"): **film-add.php** and **film-submit.php** have been added, and the changes to **film-ratings.html** have been recorded.

Submitting a Pull Request to Merge a Branch in GitHub

You have now completed the work on the local **addFilms** branch, and have uploaded (pushed) the commits that record your work up to the **addFilms** branch in GitHub. The next step is to merge these changes into your **main** GitHub branch so that they become part of the production (often public) version of your application. Since this is a serious step GitHub treats a merge as a proposal to **pull** the changes from one branch into another, known as a **pull request**. Once a pull request is submitted, you, along with project collaborators or others that you invite, have the opportunity to review and discuss the changes, test, suggest or even commit additional modifications, and approve the merge. This tutorial will introduce this process but you will want to explore this further as you develop your own applications.

On the **film-ratings** page in GitHub, **checkout** the **addFilms** branch by clicking the "branch" button and selecting "addFilms". Now click the "Compare and Pull Request" green button, located above and to the right of the file listings.

In the "Open a Pull Request" window, you will see that you are opening a pull request to compare the **addFilms** branch with the **main** branch. You should also see an "Able to merge" message, indicating that GitHub doesn't see any difficulty merging the two branches automatically. A comment box is provided so you can add an initial comment to document the purpose and scope of the merge. If this request is intended to be sent to others to review you can also clarify the kind of review you are looking for, and any specific concerns that you would like to be addressed. In this case you might simply write something like:

"The addFilms branch provides a new feature for film-ratings that allows users to submit a new film to the list of films available for review. Two files film-submit.php and film-add.php were added,and film-ratings.html was modified to include a link to film-submit.php. This pull request is to merge the addFilms branch into the main branch."

Scroll down and click "Create Pull Request". The pull request is now created and the pull request page includes four tabs that support your review process as follows:

The "Conversation" tab (currently open) invites and displays comments as you and other reviewers discuss the changes, and decide whether or not to confirm the merge. Note that this conversation can be useful even if you are the only reviewer, allowing you to note the results of tests, or user feedback, note additional modifications, etc. This tab is also where you will find the "Merge pull request" button, when (and if) you are ready to go ahead with the merge.

The "Commits" tab allows you to consider the work completed in specific commits. The "Review changes" button allows you to add comments regarding the commit, and to approve or request changes.

The "Checks" tab is targeted at collaborative development, where various conditions must be fulfilled before certain actions can be taken (for example the number of approvals that are required before a merge can take place). This is a more advanced topic and is not covered in this introductory tutorial.

The "Files Changed" tab allows you to view and comment on changes to different files associated with a specific commit, all commits, or since your last review.

Take some time to navigate these tabs just to get a feel for the tools that are available.

Assuming you are ready to complete the merge, go to the "Conversation" tab and click the "Merge pull request" button and then the "Confirm merge" button. You should see a message "Pull request successfully merged and closed", and a button that gives you the option to "Delete the branch" now that the commits have been merged with the **main** branch.

Go ahead and delete the branch. If you return to the **film-ratings** home page you will see that the repo now contains only the **main** branch, and the **main** branch now includes the commits from the deleted **addFilms** branch.

Deleting the Local Branch

You have deleted the **addFilms** branch in GitHub but not in your local repository. Go back to your local Terminal window. At the command line first **checkout main** so you are no longer in the **addFilms** branch, and then delete the **addFilms** branch:

```
git checkout main
git branch -d addFilms

warning: deleting branch 'addFilms' that has been merged to
         'refs/remotes/origin/addFilms', but not yet merged to
HEAD.
Deleted branch addFilms (was 34f11ea).
```

Git deletes the branch but also issues a warning message. Usually Git requires a forced deletion if the branch to be deleted has not yet been merged with the **main** branch, but in this case Git will allow the deletion since the branch commits have already been merged to the remote **addFilms** branch in GitHub. The warning serves as a useful reminder that, since the commits were not merged locally, your local **main** branch is behind the remote **main** branch.

(Note that you could have avoided this warning if you had first merged your local **addFilms** branch with the local **main** branch before deleting the **addFilms** branch, but it's often better practice to pull the latest version of the **main** branch from GitHub in order to update your local **main** branch, just to be sure that the two **main** branches are fully synchronized.)

Issue the **git pull** command now:

```
git pull

remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 624 bytes | 624.00 KiB/s, done.
From https://github.com/xxx/film-ratings
   cf68b8d..b690e83  main       -> origin/main
Updating cf68b8d..b690e83
Fast-forward
 film-add.php      | 50 ++++++++++++++++++++++++++++++++++++++++++
 film-ratings.html |  2 +-
 film-submit.php   | 27 +++++++++++++++++++++++++++
 3 files changed, 78 insertions(+), 1 deletion(-)
 create mode 100644 film-add.php
 create mode 100644 film-submit.php
```

In Conclusion

You have now completed your **addFilms** branch development. This may seem a lengthy and complicated process but it soon becomes routine. This second tutorial provides a brief introduction to GitHub and a useful step-by-step procedure for basic development, but you will want to explore GitHub tools and features (including project management) in more detail to take full advantage of its functionality.

For a full list of Git commands and their switches and arguments, see the excellent **https://git-scm.com/docs/** and note that a number of Git GUI clients are available, removing the need to work at the command line, and providing additional development support.