GIT:
From Beginner to Fearless
Workbook

Brian Gorman, Author/Instructor/Trainer

Welcome to GIT: From Beginner to Fearless!

Here is a little bit of information about your instructor – Brian Gorman:

I've been working with software for over 18 years now and teaching online for over 10 years. I have many courses available in various topics ranging from HTML and Bootstrap to C#, ASP.Net MVC, Java, Java EE, and, of course, GIT! I also have MCSA: Web Applications, MCSD: App builder, and MCT certifications.

It is my hope that you will enjoy this in-depth professional training. The course is designed to bring you from entry level/beginner to a professional user of GIT. By the end of the course, I hope you'll find yourself in command of GIT and ready to learn more and take on even bigger challenge.

Let me start out by saying "GIT is NOT just for developers." Although GIT is a great resource and is definitely aimed at source control for developers, there are many other uses for which GIT can be an excellent solution.

For example, are you working with images? How many times have you been editing a photo and wished you could go back to a previous version? How many times have you feared "Losing your work" – either to a drive failure, a file corruption, or just plain losing track of where the file might be on your drive?

Are you working with spreadsheets or other files that store advanced data and/or calculations? Would these files benefit from your ability to reset them to a previous version and/or to start fresh from a default template at will? GIT can give you all of this and more.

Another common activity for anyone that is working with important files is to make multiple copies of the file and/or entire copies of folders. How many times have you lost track of these "versions" and wished you had a cleaner structure to work from on your machine? Again, GIT could potentially be a great solution for you!

For those of us who are in software development or website development, GIT is an invaluable resource. With options for GitHub, GitLab, BitBucket, and others, we can determine if we want our code to be public or private, and make many different repositories to manage each of our projects individually.

The course will start off with basic setup, then begin working with a single developer flow with no branching, and then move into branching and merging – which gets you ready to work as a team. Finally, the course will finish off with some advanced commands to show you that GIT is not scary, but rather, is quite powerful and effective. By the end of the course, you'll find yourself in command of some of the more advanced concepts with GIT, ready to lead the charge for your team, while feeling confident about your GIT skills instead of fearful of what is happening with your code and/or files.

I am greatly looking forward to working with you through this course, and look forward to hearing your feedback as well as seeing your continued success.

Introduction

Overview and Table of Contents

Section 01: Getting started

- Welcome and Introduction, What the course will and won't do for you
- Getting an account setup at GitHub and/or BitBucket
- Creating a new repository
- Getting setup with GIT on your local machine
- Configuration: **`git config`**
- Initialize a new repository: **`git init`**
- Cloning a remote repository on the local machine: **`git clone`**
- Pushing files to the remote repository: **`git push`**
- Pulling files from the remote repository to the local repository: **`git pull`**
- Forking an existing repository

Section 02: Let's talk about GIT

- Distributed, Centralized, and/or Local Repositories
- Information about terms: Working Directory, Staging Changes, Committing changes, Pushing/Pulling changes
- Remote tracking branches
- Things to stop being afraid of, a few fears we'll conquer

Section 03: Getting setup for the rest of the course

- How to get the most out of this course
- Getting an existing project into a repository
- Forking a personal project repository
- GitVis tool for visualizing our repository, history, and branches [an optional program that requires Visual Studio to compile]
- Visualizing GIT with D3 [https://onlywei.github.io/explain-git-with-d3/]

Section 04: Checking the state of the repo and ignoring files

- Git Status activity -- **`git status`**
- Git Ignore activity -- **`.gitignore`** file

Section 05: Basic GIT operations: A general flow for a single person

- General thoughts and information about a single-user flow [no branches – all work done on the master branch]
- Conquered fear: No more fear of losing work!
- Staging changes with: **`git add`**
- Committing changes: **`git commit`**
- Pushing changes: **`git push`**
- Getting changes from remote**`: git pull`**
- Reviewing your commit history with: **`git log`** and **`git show`**

Section 06: Tools to improve our ability to work with GIT in the BASH command line terminal

- Use VSCode [or another editor for editing files]
- Activity: Setting the default editor
- Setting and using a diff tool for viewing differences
- Turning off the prompting for the diff tool

Section 07: Branching and Merging

- Introduction to branching
- Creating a local development branch with: **`git checkout`**
- Creating branches on the remote repository
- Working on different branches
- Determine changes from the remote repository with: **`git fetch`**
- Retrieve changes and update your local repository with: **`git pull`**
- Make changes and push to your remote branch with: **`git commit`** and **`git push`**
- Merging code from branches into master both LOCAL and REMOTE [**git merge**]
- Pushing changes to GitHub
- Retrieving changes from GitHub
- Creating and Merging code via Pull Requests at GitHub
- Delete local branches with: **`git branch -d`**
- Force delete local branches with: **`git branch -D`**
- Clean up references with the [**`git prune`**] command

Section 08: A common/simple multi-developer flow with merge conflicts

- Working as an organization at GitHub
- Setting up teams and privileges at GitHub
- Transferring a repo from a user to an organization/team
- Two developers: Two Branches
- Multiple changes on each branch that conflict
- Resolving merge conflicts when multiple teammates are modifying code
- Create a pull request for team member 2 – Resolve any conflicts
- Merge the pull request for team member 2 after conflict resolution

Section 09: Advanced GIT operations: Interacting with the repository outside the bounds of simple workflows

- Overview & Activity: **`git amend`**
- Overview & Activity: **`git reflog`**
- Overview & Activity: **`git squash`**
- Overview & Activity: **`git alias`**
- Overview & Activity: **`git reset`** and **`git clean`**
- Overview & Activity: **`git revert`**

- Overview & Activity: **`git rebase`**
- Overview & Activity: **`git cherry-pick`**
- Overview & Activity: **`git stash`**

Section 10: Using Tags to manage releases

- Tagging your commits with: **git tag**

Section 11: Workflows

- Integration Manager Overview and Example
- Dictator and Lieutenants Overview and Example

Section 12: Working with GIT and GitHub from Visual Studio or Eclipse

- Branches and Team Operations
- Visual Studio: Sync
- Visual Studio: viewing and resolving conflicts
- Visual Studio: shortcomings

Section 13: Conclusion

- Final Thoughts/Next Steps
- Conclusion

Section 01 – Getting Started

Start by doing what's necessary; then do what's possible; and suddenly you are doing the impossible – Saint Francis of Assisi

Learning:

GIT is exciting and challenging. This will not be an easy journey. However, the rewards of a challenge are often great. In this first module, we'll get introduced to some basic GIT commands, as well as take our first steps to becoming familiar with GitHub.

Goals:

- Get setup with GIT
- Get an account at GitHub || BitBucket
- Create a new repository
- Clone a repository
- What are your personal goals for this course?

Tasks:

- Watch the videos for section one
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the end of this section, you'll be on your way to mastering Git. The first steps of the journey give us the opportunity to get setup and start interacting with our first repository. Not only will we be getting used to working with GitHub (or BitBucket), but we'll have GIT setup on our computer, ready to go for the next steps in our journey.

Videos:

- Welcome and Intro
- What this course is NOT going to do
- Setup an account at GitHub || BitBucket
- Create a new Repository at GitHub || BitBucket
- Setup GIT on your machine
- Create a local repository
- Clone a remote repository locally to your machine
- Setting up credentials – with Activity
- Add, Commit, and Push some files to the remote repository
- Pull some changes from the repository
- Fork an Existing repository

Notes

GIT: From Beginner to Fearless

Activity: Setting credentials

Brian Gorman, Author/Instructor/Trainer

Introduction

Setting your credentials is much more trivial now than it used to be. We are lucky that we have credential managers. Credential managers allow us to store our username and password combinations in one place and thereby forego entering our username/password combination on every critical action going forward.

In this activity, we'll take a look at some different ways that we can setup our credentials going forward. For the most part, however, we'll be relying on the credential managers. Additionally, not doing this activity just means you would need to enter your information more frequently – but that's only if you also don't have a credential manager in place by default.

Remember that there are three levels of config {System, Global, Local}. Since we are an individual user, the best place to store the credentials would be in the Global level [for the user account].

Let's get started!

GFBTF: GIT Credentials Activity [designed for Windows Users]

Step 1: Determine your credentials:

a) *First, see if you have any credentials set*
[**git config –list**]:

b) *If credentials are set, make a note of what they currently are.*

Step 2: See what it's like with no credentials:

This step shows us what it's like if credentials are not stored. We'll have to enter our credentials every time we want to do something that requires permission, even if the activity is not destructive.

a) *Remove your credentials*
[**git config --global --unset-all credential.helper**]
or [**git config --unset-all credential.helper**]

b) *Clone a private repo*
[**git clone <url> <newFolderRepoName>**]
requires permissions. If you don't have a private repo, create one at BitBucket [or GitHub --- but remember GitHub requires payment]

c) *Enter your credentials*

d) Remove the repo by deleting the files and folders
[**rm –rf <root_folder_name>]**

e) *Repeat steps b and c.*
You should be required to enter your credentials again.

Step 3: Set up credentials with a cache:

This step takes us through what it's like to work with the cahce to store credentials. Essentially, here we can setup our credentials to be temporarily stored in cache.

Notes

a) *Set credentials to timeout in 2 minutes [120 seconds]*
   [**git config --global credential.helper 'cache --timeout=120'**]
b) *Clone the repo And enter your credentials*
   [**git clone <url> <folder>**]
c) *Remove the repo from your file system*
   [**rm –rf <foldername>**]
d) *Clone the repo – this time you should not have to enter credentials*
   [**git clone <url> <folder>**]
e) *Remove the repo from your file system*
   [**rm –rf <foldername>**]
f) *Play solitaire, look at facebook, etc, for 2 minutes*
g) *Clone the repo – expect to reenter your credentials*
   [**git clone <url> <folder>**]

Step 4: Unset credentials:

a) *Unset existing credentials*
   [**git config --global --unset-all credential.helper**]
b) *Remove the credential section from your config*
   [**git config --global --remove-section credential**]

Step 5: Set credentials to permanent store:

a) *Set credentials to store*
   [**git config --global credential.helper store**]
b) *Clone the repo – enter your credentials*
   [**git clone <url> <folder>**]
c) *Remove the repo from your file system*
   [**rm –rf <foldername>**]
d) *Play solitaire, look at facebook, etc, for 2 minutes*
e) *Clone the repo – no timeout so credentials should not need to be re-entered*
   [**git clone <url> <folder>**]

Step 6: Resetting your credentials:

a) *Reset your credentials to use the credential manager [Windows]*
[**git config --global credential.helper manager**]

b) *See additional resources to set up OS X Keychain [MAC]*

c) *See additional resources to use a keystore in Ubuntu [Linux]*

Step 7: View your credentials:

a) *List your current credentials*
[**git config --global --get credential.helper**]

Closing Thoughts and additional resources

In this activity, we've seen how we can use various methods to store our credentials. Since there are various security implications for some of the storage mechanisms, the safest and best way is using a credential manager or keystore. If that is not possible, however, we now know that we can easily set our credentials for at least a period of time to avoid having to re-enter them each time a critical command is issued.

Additional Resources:

MAC:
http://tech.lds.org/wiki/Git_Credential_Caching_on_Mac_OS_X

Linux [Ubuntu]
https://askubuntu.com/questions/740183/store-git-credentials-permanently-and-encrypted-using-a-keystore-in-ubuntu

Notes

Section 02 – Let's talk about GIT

It's the little details that are vital. Little things make big things happen – John Wooden

Learning:

We've started to see the beginning of how GIT works. In this section we take a step back and look at some different types of source control systems, including centralized and decentralized. We'll examine some of the pros and cons of each type of source control. We'll then talk about a few of the key terms in GIT and how we can move files through different stages. We'll finish up with a discussion of things that should not cause us fear going forward.

Goals:

- Be able to discuss differences between centralized and decentralized source control systems
- Recognize the different statuses of changes and where changes are located in this process during normal flow
- Understand how to move files between local and remote repositories, as well as how local repositories keep track of their remote upstream

Tasks:

- Watch the videos for section two
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the end of this section, understand the different phases that changes track through during normal flow in a GIT repository. Additionally, be in command of the ideas of how to move files between repositories, and how branches keep track of their upstream source.

Videos:

- Centralized, Decentralized, Distributed, and/or Local only
- Working Directory, Staging changes, Committing Changes
- Pushing and Pulling Changes
- What is a remote tracking branch?
- Things that shouldn't scare you, now, and later.

Notes

Section 03 – Getting setup for the rest of the course

An investment in knowledge pays the best dividends – Benjamin Franklin

Learning:

In this section, we cover how to succeed in this course, getting an existing project into a repository, forking an existing repository, and take a look at a tool that helps us map our commits on local repositories.

Goals:

- Learn how to succeed in this course
- Get an existing project into a repo
- Fork our own repo

Tasks:

- Watch the videos for section three
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we'll have a few strategies to help us succeed in our learning for this course. We'll also have mastered the art of making repositories, including empty repositories and repositories for existing projects.

Videos:

- How to get the most out of this course
- Getting an existing project into a repository
- Forking our own repositories
- Using GitVis to map and see commit history on our local repository

Notes

Section 04 – Checking the state of the repo and ignoring files

It's really clear that the most precious resource we all have is time – Steve Jobs

Learning:

In this section, we take a deep dive into the git status command and setting up a .gitIgnore file. We'll need the status command a lot in the future to check the state of our repo. Once we have a valid .gitIgnore file in place, we can keep our repository limited to only the files we want to track.

Goals:

- Use GIT status to check the state of the local repo
- Get an existing project into a repo
- Fork our own repo

Tasks:

- Watch the videos for section four
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we'll have a few strategies to help us succeed in our learning for this course. We'll also have mastered the art of making repositories, including empty repositories and repositories for existing projects.

Activities:

- git status activity
- git ignore activity

Videos:

- How to get the most out of this course
- Getting an existing project into a repository
- Forking our own repositories
- Using GitVis to map and see commit history on our local repository

Notes

GIT: From Beginner to Fearless

GIT status, diff, log, and show Activity: What is the state of my repo?

Brian Gorman, Author/Instructor/Trainer

Introduction

There will be many times we will want to check on the status of our repository. By "status," we are actually looking to find out what has been changed, as well as what is ready to be committed. Don't worry if these terms are still new to you at this point, because as we spend more time working through this activity, they will become quite a bit more familiar.

We've already worked through a few flow operations where we've seen files move from untracked to tracked to staged to committed. Additionally, we've seen the "official" diagram regarding the general flow of files through different phases in GIT. We've learned about the working directory, the Index (or staging area) and the Head, which is essentially a pointer to the last commit on the checked-out branch.

In this activity, we'll go through some of these things in detail, and learn about ways we can use the status command to get us the information we need about the current state of our repo.

Let's gets started!

Step 1: Start with a working copy of the repository files:

a) Download the default web as a .zip file.
We're going to start fresh for practice and to see the status change for various objects

b) Alternatively, if you already have a copy and want to use what you have
You could just copy the existing directory
then delete the .git Folder from within it.

Step 2: Establish a repository:

a) Check your folder structure
Use [**ls-al**] to determine if a repo already exists. If one exists, delete it

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb
$ ls -al
total 52
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:23 ./
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:23 ../
-rw-r--r-- 1 Brian 197608 6946 Jun 30 00:15 About.html
-rw-r--r-- 1 Brian 197608 6392 Jun 30 00:15 ContactUs.html
drwxr-xr-x 1 Brian 197608    0 Jun 29 22:50 css/
-rw-r--r-- 1 Brian 197608 3213 Jun 30 00:18 details.html
drwxr-xr-x 1 Brian 197608    0 Jul 25  2016 fonts/
drwxr-xr-x 1 Brian 197608    0 Jun 30 00:00 images/
-rw-r--r-- 1 Brian 197608 5745 Jun 30 00:15 index.html
drwxr-xr-x 1 Brian 197608    0 Jun 29 22:45 js/
-rw-r--r-- 1 Brian 197608 6933 Jun 30 00:15 portfolio.html
```

Step 3: Check Status:

a) Create a new repository
[**git init**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb
$ git init
Initialized empty Git repository in G:/Data/GFBTF/DefaultWeb/.git/
```

Notes

[**ls -al**] to see all files, validate a .git folder exists

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb
$ ls -al
total 56
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:23 ./
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:23 ../
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:23 .git/
-rw-r--r-- 1 Brian 197608 6946 Jun 30 00:15 About.html
-rw-r--r-- 1 Brian 197608 6392 Jun 30 00:15 ContactUs.html
drwxr-xr-x 1 Brian 197608    0 Jun 29 22:50 css/
-rw-r--r-- 1 Brian 197608 3213 Jun 30 00:18 details.html
drwxr-xr-x 1 Brian 197608    0 Jul 25  2016 fonts/
drwxr-xr-x 1 Brian 197608    0 Jun 30 00:00 images/
-rw-r--r-- 1 Brian 197608 5745 Jun 30 00:15 index.html
drwxr-xr-x 1 Brian 197608    0 Jun 29 22:45 js/
-rw-r--r-- 1 Brian 197608 6933 Jun 30 00:15 portfolio.html
```

.git folder is present when a local repo exists

b) Take a look at the status of the repo right now:

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        About.html
        ContactUs.html
        css/
        details.html
        fonts/
        images/
        index.html
        js/
        portfolio.html

nothing added to commit but untracked files present (use "git add" to track)
```

Note that at this time, all the listed objects are untracked.

What do you think the red color means?_____

c) Review the state of the repo again

[**git status –s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
?? About.html
?? ContactUs.html
?? css/
?? details.html
?? fonts/
?? images/
?? index.html
?? js/
?? portfolio.html
```

What do you think the red double ??'s mean?:_____

Step 4: Adding and Removing a file from the Index ["Staging Area"]:

a) Add a single file to index

[**git add About.html**]

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add About.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   About.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ContactUs.html
        css/
        details.html
        fonts/
        images/
        index.html
        js/
        portfolio.html
```

Note the added file is Green and the rest of the files are red. Green indicates the file is "staged" or is in the "Index" ready to be committed. If we were to type 'git commit –m "…"' we would commit just the About.html file. However we're not going to do that just yet.

b) Review the short status

[**git status –s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
A  About.html
?? ContactUs.html
?? css/
?? details.html
?? fonts/
?? images/
?? index.html
?? js/
?? portfolio.html
```

Here it is very important to note something: look at how the first column is "A " ["A" and a BLANK]. What do you think this means?
The double ??'s indicate that the remaining files are both unstaged and untracked.

c) Remove a file from INDEX

[**git rm –cached About.html**]
[**git status**]
[**git status –s**]
*** IMPORTANT note. This ONLY works without consequences because we've never committed the file. If we had committed the file, this would be a terrible idea because it would remove it from the repo as well. If the file is already in the repo and we need to roll back to where it was, we'd use RESET. We'll show that later in the course.

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git rm --cached About.html
rm 'About.html'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
?? About.html
?? ContactUs.html
?? css/
?? details.html
?? fonts/
?? images/
?? index.html
?? js/
?? portfolio.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        About.html
        ContactUs.html
        css/
        details.html
        fonts/
        images/
        index.html
        js/
        portfolio.html

nothing added to commit but untracked files present (use "git add" to track)
```

The files have returned to “UnIndexed” and “UnTracked”

Step 5: Adding and Removing multiple files:

a) Add all files that are unstaged to the INDEX

[**git add .**]

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .
warning: LF will be replaced by CRLF in fonts/glyphicons-halflings-regular.svg.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
A  About.html
A  ContactUs.html
A  css/site.css
A  details.html
A  fonts/glyphicons-halflings-regular.eot
A  fonts/glyphicons-halflings-regular.svg
A  fonts/glyphicons-halflings-regular.ttf
A  fonts/glyphicons-halflings-regular.woff
A  fonts/glyphicons-halflings-regular.woff2
A  images/linkedin.png
A  images/twitter.png
A  index.html
A  portfolio.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$
```

b) Remove all of the files from the INDEX

[**git rm –cached -r**] **again, Don’t do this once files are being tracked.

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git rm --cached -r .
rm 'About.html'
rm 'ContactUs.html'
rm 'css/site.css'
rm 'details.html'
rm 'fonts/glyphicons-halflings-regular.eot'
rm 'fonts/glyphicons-halflings-regular.svg'
rm 'fonts/glyphicons-halflings-regular.ttf'
rm 'fonts/glyphicons-halflings-regular.woff'
rm 'fonts/glyphicons-halflings-regular.woff2'
rm 'images/linkedin.png'
rm 'images/twitter.png'
rm 'index.html'
rm 'portfolio.html'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
?? About.html
?? ContactUs.html
?? css/
?? details.html
?? fonts/
?? images/
?? index.html
?? portfolio.html
```

Now we know a way to add and remove all files. Please remember that using rm is something we should be careful with. Once files are being tracked, this actually removes them, so we'll learn a different command [reset] later

c) Committing our files

Let's add just one file – about.html – to the INDEX. What is the command again to add just one file? _____

After adding the about.html file to the INDEX, commit the file with a commit message:

[**git commit –m 'Added the About.html file'**]

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add About.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m 'Added the About.html file'
[master (root-commit) f69f692] Added the About.html file
 1 file changed, 97 insertions(+)
 create mode 100644 About.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
?? ContactUs.html
?? css/
?? details.html
?? fonts/
?? images/
?? index.html
?? portfolio.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$
```

Wait! Where is my About.html file? It's not listed!!!

Maybe if I use status I can see it?

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ContactUs.html
        css/
        details.html
        fonts/
        images/
        index.html
        portfolio.html

nothing added to commit but untracked files present (use "git add" to track)
```

Oh no! I lost my file! --- or not….It's simply not shown because there are no changes to the file and it's already committed as-is. Want to see it?

d) View files using the log and show commands

[**git log**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m "Added the h3 tag for upcoming changes"
[master cb5204a] Added the h3 tag for upcoming changes
 1 file changed, 1 insertion(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
 M details.html
```

Note "Added the About.html file" is the commit message I typed. If you typed something different, you'd see that there.

[**git show --summary**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git show --summary
commit f69f6921daa4a7f758e0bd849c98501ade961a2c (HEAD -> master)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 09:56:05 2017 -0500

    Added the About.html file

 create mode 100644 About.html
```

This tells me that the file "About.html" was "created"

We can also see this with other versions of the show command:

[**git show --stat**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git show --stat
commit f69f6921daa4a7f758e0bd849c98501ade961a2c (HEAD -> master)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 09:56:05 2017 -0500

    Added the About.html file

 About.html | 97 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 97 insertions(+)
```

[**git show --name-status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git show --name-status
commit f69f6921daa4a7f758e0bd849c98501ade961a2c (HEAD -> master)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 09:56:05 2017 -0500

    Added the About.html file

A       About.html
```

e) Add the remaining files

Right now, we have just the About.html file committed (in HEAD). And the rest of the files are Untracked, with nothing in Index (Staging).

Add the remaining files to the Index. What is the command again to add all the untracked or modified file changes to Index?

_____

[**git commit -m "Added the rest of the files"**]
[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m "Added the rest of the files"
[master 874d595] Added the rest of the files
 12 files changed, 711 insertions(+)
 create mode 100644 ContactUs.html
 create mode 100644 css/site.css
 create mode 100644 details.html
 create mode 100644 fonts/glyphicons-halflings-regular.eot
 create mode 100644 fonts/glyphicons-halflings-regular.svg
 create mode 100644 fonts/glyphicons-halflings-regular.ttf
 create mode 100644 fonts/glyphicons-halflings-regular.woff
 create mode 100644 fonts/glyphicons-halflings-regular.woff2
 create mode 100644 images/linkedin.png
 create mode 100644 images/twitter.png
 create mode 100644 index.html
 create mode 100644 portfolio.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$
```

So all of our files are added, and note that the short version of status shows nothing!

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
nothing to commit, working tree clean
```

[**git show --name-status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git show --name-status
commit 874d5950c848e6bee50db22c6bb0b62458f1ca0d (HEAD -> master)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 10:07:51 2017 -0500

    Added the rest of the files

A       ContactUs.html
A       css/site.css
A       details.html
A       fonts/glyphicons-halflings-regular.eot
A       fonts/glyphicons-halflings-regular.svg
A       fonts/glyphicons-halflings-regular.ttf
A       fonts/glyphicons-halflings-regular.woff
A       fonts/glyphicons-halflings-regular.woff2
A       images/linkedin.png
A       images/twitter.png
A       index.html
A       portfolio.html
```

Note: About.html is not shown. Why do you think that is?

_____

Is About.html sill in the repo?

_____

How can I know for sure?

_____

[**git log**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git log
commit 874d5950c848e6bee50db22c6bb0b62458f1ca0d (HEAD -> master)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 10:07:51 2017 -0500

    Added the rest of the files

commit f69f6921daa4a7f758e0bd849c98501ade961a2c
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Wed Jul 5 09:56:05 2017 -0500

    Added the About.html file
```

Step 6: More Show and Log examples:

a) Make some changes

[**vim Details.html**]

When the file comes up, make modifications.

[**i**] //hit 'i' to insert in VIM

Add an h2 tag under the <h1> that says 'details page with more content:

[**<h2>Git is awesome and I'm learning so much cool stuff!"</h2>**]

[**{esc}**] //hit the escape key to exit insert mode in VIM

[**:wq**] //to write and quit VIM

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ vim Details.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
 M details.html
```

Note the file has a red "M". The red "M" is in the second column. This means the file is "Modified" against the Index version of the file {which is yet another way to know we are tracking the file..}. Note the "M" is RED because the file is not yet added to index.

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ 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:   details.html

no changes added to commit (use "git add" and/or "git commit -a")
```

b) Add the changes

Add the details.html file to the index. What is the command for adding a single file to Index again?

_____

What if I had just typed 'git add .'? [all modified files would be staged – and since only one file was modified…it would be staged just the same!]

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
M  details.html
```

Note: Now the file has a Green "M" in the Index column. We now know the file is Modified and Added to Index [staged] for commit.

c) Commit the changes

Let's commit the changes:

[**git commit –m "Added an h2 tag to the details page"**]

[**git status -s**]

[**git log --oneline**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m 'Added an h2 tag to the details page'
[master 331b291] Added an h2 tag to the details page
 1 file changed, 2 insertions(+), 2 deletions(-)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
nothing to commit, working tree clean

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git log --oneline
331b291 (HEAD -> master) Added an h2 tag to the details page
874d595 Added the rest of the files
f69f692 Added the About.html file
```

Note the three commits, and note HEAD -> master. This means the current HEAD is at commit 331b291 and the branch we are on is master. Don't worry if that's a little unclear still. We'll be learning much more about this later.

Step 7: Modified files that are already tracked, only commit the things we want to commit:

a) Repeat the modification steps as above to make another change in details.html.

[**vim details.html**]

[**i**]
//under the <h2> tag we entered above, place the following line
[**<h3>Multiple changes are coming</h3>**]
[**{esc}**]
[**:wq**]
[**git add details.html**] //add the change for the <h3> tag to INDEX

b) Create another change

[**vim details.html**]
[**i**]
//under the <h3> tag we entered above, place the following line
[**<h3>Yet another change</h3>**]
[**{esc}**]
[**:wq**]
//DO NOT add this change to staging (INDEX).
[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
MM details.html
```

Whoa! Now we have one modification ready to add [Red, modified against HEAD but not in index] and one modification ready to commit [Green, modified against HEAD and Staged in INDEX]. That's cool. But what will I commit if I commit from this state? [we know it's just the h3 tag, but this is an easy activity. Think about hundreds of changes in a system with only some of them added to Index and some not added, or a practical situation where you want certain changes but don't want others…]

c) See what is added and ready to commit

[**git diff --cached details.html**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git diff --cached details.html
diff --git a/details.html b/details.html
index b92420b..19ce9f9 100644
--- a/details.html
+++ b/details.html
@@ -44,6 +44,7 @@

             <h1> Details Page with more content... </h1>
            <h2> Git is awesome and I'm learning so much cool stuff!</h2>
+           <h3> More changes are coming </h3>
             <hr />
             <footer>
                 <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com
">Major Guidance Solutions</a></p>
```

If we commit now, we'd see the green line added to the file in the new HEAD. Exactly what we expect.

To see what changes would NOT be committed [they would NOT be lost, just not committed to HEAD]:

[**git diff details.html**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git diff details.html
diff --git a/details.html b/details.html
index 19ce9f9..5812aae 100644
--- a/details.html
+++ b/details.html
@@ -45,7 +45,8 @@
             <h1> Details Page with more content... </h1>
            <h2> Git is awesome and I'm learning so much cool stuff!</h2>
            <h3> More changes are coming </h3>
-           <hr />
+           <h4> Yet another change </h4>
+           <hr />
             <footer>
                 <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com
">Major Guidance Solutions</a></p>
             </footer>
```

Here we can see that the changes for the <h4> tag are waiting to be staged.

It's important to remember that ANYTHING that is not added to INDEX [staged] at time of commit will be left out.

It is equally as important to remember that anything that is modified but not staged is NOT deleted by a commit action. It's just not included.

This is critical: Committing partial changes DOES NOT reset your file to its original state.

d) Commit the partial changes

Let's see this in action to lose any remaining fear we have:
[**git commit -m "Added the h3 tag for upcoming changes"**]
[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m "Added the h3 tag for upcoming changes"
[master cb5204a] Added the h3 tag for upcoming changes
 1 file changed, 1 insertion(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
 M details.html
```

Again we see that our h3 tag was added, and now we have ONLY the remaining modification for the h4 tag. Since nothing is added to INDEX for staging, '**git diff –cached details.html**' yields 0 results, but '**git diff details.html'** still shows our remaining changes!

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git diff --cached details.html

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git diff details.html
diff --git a/details.html b/details.html
index 19ce9f9..5812aae 100644
--- a/details.html
+++ b/details.html
@@ -45,7 +45,8 @@
             <h1> Details Page with more content... </h1>
            <h2> Git is awesome and I'm learning so much cool stuff!</h2>
            <h3> More changes are coming </h3>
-            <hr />
+            <h4> Yet another change </h4>
+            <hr />
             <footer>
                 <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com
">Major Guidance Solutions</a></p>
             </footer>
```

e) Final commit for the activity

Let's do one final commit to wrap up the activity:

[git commit -m "Added the h4 tag change"]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -am "Added the h4 tag change"
[master c8672c8] Added the h4 tag change
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Note: We can add and commit on the same line, in one command. Also note this only works on files that are already tracked, so using '-am' requires a modification to a previously committed file in order to work as expected.

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status -s
```

Running 'git log --oneline' reveals our final history:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git log --oneline
c8672c8 (HEAD -> master) Added the h4 tag change
cb5204a Added the h3 tag for upcoming changes
331b291 Added an h2 tag to the details page
874d595 Added the rest of the files
f69f692 Added the About.html file
```

Closing Thoughts

In this activity, we've worked through making changes and seeing the state of our repo with the status command. We saw that status has both a long version and a short version. This allows us flexibility in how we want to show the status.

Additionally, we learned about how we can use the log and show commands to be able to list out our commit history, see details about commits, and view the contents of a commit.

We also learned that not everything has to be committed at one time, and also how important it is to make sure that everything we want to commit is actually staged into the INDEX. We can always see what is going to be added or review what is not going to be added using the diff command. We'll also learn in the future that the diff command will be highly useful for comparing changes for entire commits, as well as how we've used it here to view changes in the working directory and index.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner to Fearless

GIT Ignore Activity:
Excluding files from our repository

Brian Gorman, Author/Instructor/Trainer

Introduction

In almost every repository, there will be many files we are wanting to track changes on. Additionally, there will be a few files that we never want to track for various reasons. Perhaps the files are auto-generated by a build process. Perhaps the files are unique to a user. Perhaps the files contain sensitive information. In any event, we are able to exclude files from our repository very quickly and easily using a .gitIgnore file.

In this activity, we are going to take a look at how we can add a .gitIgnore file, and then we'll look into how we can modify that file to make sure that specific files and folders are eliminated from tracking.

Let's gets started!

Step 1: Creating a global .gitIgnore file

c) Check to see if there is an excludes file directive in your global config.

First we will = tell our system where to find our global .gitIgnore [excludes] file if one doesn't already exist.

[**git config --global core.excludesfile**]

```
Brian@Prometheus MINGW64 /c/Data/GFBTF
$ git config --global core.excludesfile

Brian@Prometheus MINGW64 /c/Data/GFBTF
$
```

If nothing is set – nothing is shown. If you see a file listed, then you already have a global core excludes file, and you should be able to edit that file.

Here, I have a file set already:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global core.excludesfile
C:/Users/Brian/.gitIgnore
```

If no file is present, create it:

[**git config --global core.excludesfile ~/.gitIgnore**]

d) Open the .gitIgnore File and add exclusions

Make sure you have the path to your gitIgnore file handy [as created or listed in a above]

[**vim <path-to-your-global-ignore>]**

[**i**]

[***.dll**]

[***.class**]

[***.jar**]

[**{esc}**]

[**:wq**]

```
#This is a comment

#blank lines are also ignored

#exclude all files with the extension *.noinclude
*.dll
*.class
*.jar
~
~
~
```

Notes

Step 2: Check that the .gitIgnore file is working:

b) Get the resources; extract them; copy them; or just create new ones

Under the root of the project, create a new 'bin' folder

[**mkdir bin**]

[**ls -al**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ mkdir bin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ ls -al
total 56
drwxr-xr-x 1 Brian 197608    0 Jul  5 13:41 ./
drwxr-xr-x 1 Brian 197608    0 Jul  5 12:36 ../
drwxr-xr-x 1 Brian 197608    0 Jul  5 11:51 .git/
-rw-r--r-- 1 Brian 197608 6946 Jun 30 00:15 About.html
drwxr-xr-x 1 Brian 197608    0 Jul  5 13:41 bin/
-rw-r--r-- 1 Brian 197608 6392 Jun 30 00:15 ContactUs.html
drwxr-xr-x 1 Brian 197608    0 Jul  5 09:47 css/
-rw-r--r-- 1 Brian 197608 3351 Jul  5 11:31 details.html
drwxr-xr-x 1 Brian 197608    0 Jul 25  2016 fonts/
drwxr-xr-x 1 Brian 197608    0 Jun 30 00:00 images/
-rw-r--r-- 1 Brian 197608 5745 Jun 30 00:15 index.html
drwxr-xr-x 1 Brian 197608    0 Jun 29 22:45 js/
-rw-r--r-- 1 Brian 197608 6933 Jun 30 00:15 portfolio.html
```

Place the two files for exclusion into the bin folder. Not that you will need to move the files from the location they are in to the location we created. You don't have to do this via BASH if you don't want to.

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ cp -R /g/data/GFBTF/resources_for_ignore_act/excludes/ /g/Data/GFBTF/DefaultWe
b/bin/
```

Make sure the repo does not recognize that anything has changed

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Note that even though we've added files, they are excluded by the .gitIgnore

c) Prove the folder is monitored

To prove the folder is being monitored, let's add something that would be found:

[If necessary, cd to the directory with the excluded files]

[**touch info.txt**]

[**vim info.txt**]

[**i**]

[**Working with .gitIgnore, this file should be included**]

[**{esc}**]

[**:wq**]

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git status -s
?? ../

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ../

nothing added to commit but untracked files present (use "git add" to track)
```

d) Add and commit the info.txt file – further proving the others are ignored.

[**git add .**]
[**git status -s**]
[**git commit –am "Added info.txt during gitIgnore Activity"**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git add .
warning: LF will be replaced by CRLF in bin/excludes/info.txt.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git status -s
A  info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git commit -m "added info.txt during gitIgnore Activity"
[master 4f86487] added info.txt during gitIgnore Activity
 1 file changed, 3 insertions(+)
 create mode 100644 bin/excludes/info.txt
```

Step 3: Create a local .gitIgnore just for this repo:

d) Navigate to the root directory of your repo (the folder that has the .git folder in it).

[**touch .gitIgnore**]
[**vim .gitIgnore**]
[**i**]
[**bin/**]
[**{esc}**]
[**:wq**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ touch .gitIgnore

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ vim .gitIgnore
```

```
bin/
~
~
~
```

Note that files we are already tracking (info.txt) will still be tracked. However, new files created in bin/ will be ignored.

e) Add another file to prove local .gitIgnore is working

Note that this points out the fact that our global .gitIgnore is NOT being tracked…

[navigate back to the bin folder or use full paths]

[**touch anotherFile.txt**]

[**vim anotherFile.txt**]

[**i**]

[**this is another file and it should be ignored**]

[**{esc}:wq**]

[**git status -s**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ../.gitIgnore

nothing added to commit but untracked files present (use "git add" to track)
```

[we're able to add the new .gitIgnore but no sign of anotherFile.txt]

Add and commit the gitIgnore file [must be at the correct level or reference the path]

[**git add ../.gitIgnore**]

[**git commit –m "added the local gitIgnoreFile"**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin (master)
$ git add ../.gitIgnore
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin (master)
$ git commit -m "added the local gitIgnore file"
[master 8134e71] added the local gitIgnore file
 1 file changed, 1 insertion(+)
 create mode 100644 .gitIgnore
```

Step 4: More changes to show tracked/untracked behaviors

d) Make changes to the info.txt file to prove it is still tracked even though it is in a folder that is ignored:

[**vim info.txt**]

[**i**]

[**Still being tracked…**]

[**{esc}:wq**]

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin (master)
$ cd excludes/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ vim info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ 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:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

So now we know the file is modified and tracked even though the folder is excluded.

e) Create another file in the subfolder of bin/excludes to prove the ignore is recursive for any subfolder under root of bin/:

[navigate to …/bin/excludes]
[**touch yetAnotherFile.txt**]
[**vim yetAnotherFile.txt**]
[**i**]
[**This is yet another file that will now be ignored**]
[**{esc}:wq**]
[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ touch yetAnotherFile.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ vim yetAnotherFile.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ 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:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

So this proves the folder and subfolders are ignored now, but previously tracked files are still tracked.

Add and commit the changes
[**git commit -am "changes to tracked file in ignored folder are still tracked"**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git commit -am "changes to tracked file in ignored folder are still tracked"
warning: LF will be replaced by CRLF in bin/excludes/info.txt.
The file will have its original line endings in your working directory.
[master 43239f4] changes to tracked file in ignored folder are still tracked
 1 file changed, 1 insertion(+), 1 deletion(-)
```

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/bin/excludes (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Step 5: Adding Local resources that would ordinarily be ignored:

f) Sometimes we have files that we need to keep in the repo that would ordinarily be excluded

In practical situations, sometimes we need to keep a valuable resource around in order to have a valid version of it, or a reference to it in our project. Here we are going to simulate this practical situation
[navigate back to root]
[**mkdir Resources**]
[copy the resource files into that folder]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources (master)
$ cp -R /g/data/GFBTF/resources_for_ignore_act/includes/ .
```

[navigate to the includes folder]

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources (master)
$ cd includes

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ ls
MySql_ADONETConnector_v123.4.dll  MySql_JDBCConnector_v123.4.jar

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Note that the files are initially ignored, as expected for these types of files.

g) Include the resources

Next we need to re-include the files that are globally ignored. This will prove that our local .gitIgnore is able to supercede the global .gitIgnore

[navigate back to the root where the local .gitIgnore file is]

[**vim .gitIgnore**]

[**i**]

[**!resources/includes/*.dll**]

```
bin/

!resources/includes/*.dll
~
~
~
```

[**{esc}:wq**]

We are now including the includes folder dll [we could have included files directly by name, too, but this gets ALL dlls in the includes folder]

[**git add .**] //add the included dlls

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitIgnore
        new file:   Resources/includes/MySql_ADONETConnector_v123.4.dll
```

h) Include the whole folder, rather than just one file type

[**vim .gitIgnore**]

[**i**]

[**!resources/includes/***]

```
bin/

#include the resources folder
!resources/includes/*
~
~
```

[**{esc}:wq**]

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ vim .gitIgnore

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitIgnore
        new file:   Resources/includes/MySql_ADONETConnector_v123.4.dll

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:   .gitIgnore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Resources/includes/MySql_JDBCConnector_v123.4.jar
```

Now all the files are found, we need to add and commit to track and keep them.

[**git add .**]

[**git commit -m "added the important resource files"**]

[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitIgnore
        new file:   Resources/includes/MySql_ADONETConnector_v123.4.dll
        new file:   Resources/includes/MySql_JDBCConnector_v123.4.jar


Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m 'Added the important resource files'
[master 0a2a8de] Added the important resource files
 3 files changed, 3 insertions(+)
 create mode 100644 Resources/includes/MySql_ADONETConnector_v123.4.dll
 create mode 100644 Resources/includes/MySql_JDBCConnector_v123.4.jar

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
nothing to commit, working tree clean
```

i) Exclude a file after including the whole folder

[**vim .gitIgnore**]

[**i**]

[**!resources/includes/***]

```
bin/

#include the resources folder
!resources/includes/*

#make sure to ignore "notes"
resources/includes/notes.txt
~
~
~
```

[**{esc}:wq**]
[navigate to the includes directory]
[**vim notes.txt**]
[**i**]
[**these are important notes for myself that we are not tracking**]
[**{esc}:wq**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ cd Resources/includes/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ vim notes.txt
```

[**git add .**]
[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git add .

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ 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:   ../../.gitIgnore

no changes added to commit (use "git add" and/or "git commit -a")
```

j) Make sure can add another text file that would be tracked

[**vim developer_notes.txt**]
[**i**]
[**these are important developer notes that everyone needs**]
[**{esc}:wq**]
[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ 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:   ../../.gitIgnore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        developer_notes.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

[**git add .**]
[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   developer_notes.txt

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:   ../../.gitIgnore
```

Need to add the .gitIgnore as well:
[**git add ../../.gitIgnore**]
[**git status**]
[**git commit -m "Continuing with the .gitIgnore Activity…"**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git add ../../.gitIgnore
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   ../../.gitIgnore
        new file:   developer_notes.txt


Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Resources/includes (master)
$ git commit -m "Continuing with the .gitIgnore activity"
[master 524f0f7] Continuing with the .gitIgnore activity
 2 files changed, 5 insertions(+)
 create mode 100644 Resources/includes/developer_notes.txt
```

So now we have seen that we can include entire folders, exclude entire folders, include specific files within excluded folders and exclude specific files using different statements in the .gitIgnore.

Also, due to the fact that the local .gitIgnore .dll works when named, we can see the hierarchy – that a local .gitIgnore can override a global gitIgnore.

Step 6: Using patterns in the .gitIgnore file:

d) Make some changes

[navigate to the root folder]
[**vim .gitIngore**]
[**i**]
[change bin/ to **[Bb]in**]

```
[Bb]in/

#include the resources folder
!resources/includes/*

#make sure to ignore "notes"
resources/includes/notes.txt
~
~
~
```

[**{esc}:wq**]
[**git status**]

[Rename the bin folder to Bin]
[**mv bin Bin**]
[**ls**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ mv bin Bin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ ls
About.html      css/          images/     portfolio.html
Bin/            details.html  index.html  Resources/
ContactUs.html  fonts/        js/
```

e) Commit the changes

[**git status**]

[**git add .**]
[**git commit -m "added pattern for gitIgnore on [Bb]in/****]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ 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:   .gitIgnore

no changes added to commit (use "git add" and/or "git commit -a")

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m "added pattern for gitIgnore on [Bb]in/**"
[master f737642] added pattern for gitIgnore on [Bb]in/**
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note that even though we changed the directory from bin/ to [Bb]in/, excluded files are still excluded as would be expected.

f) Make sure subfolders are ignored as expected

Make directories 'debug', 'release', and 'program' under 'bin'
Add a simple text file to each.
Validate the files are ignored due to our new settings:
[**navigate to Bin**]
[**mkdir debug**]
[**mkdir release**]
[**mkdir program**]
[**touch debug/info.txt**]
[**touch release/info.txt**]
[**touch program/info.txt**]
[**git status**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ cd Bin/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ mkdir debug

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ mkdir release

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ mkdir program

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ touch debug/info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ touch release/info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ touch program/info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb/Bin (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Notice that none of the files in the subfolders are showing up as untracked, they are simply ignored.

g) Track all files in a subfolder of an excluded folder (tricky)

[navigate to the root folder]
[**vim .gitIngore**]
[**i**]
[change the file to look like this:]

```
![Bb]in/
[Bb]in/*
![Bb]in/release*
![Bb]in/release*/**

#include the resources folder
!resources/includes/*

#make sure to ignore "notes"
resources/includes/notes.txt
```

[**{esc}:wq**]

The first line says 'don't exclude the bin folder at the top level – if it's excluded, no subfolders can show

The second line hides everything in the bin folder by default

The third line unhides the release folder
The fourth line unhides all the files in release folder.

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ 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:   .gitIgnore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Bin/release/

no changes added to commit (use "git add" and/or "git commit -a")

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git add .
warning: LF will be replaced by CRLF in .gitIgnore.
The file will have its original line endings in your working directory.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitIgnore
        new file:   Bin/release/info.txt
        new file:   Bin/release/myprogram.dll
```

h) Commit and review the log

[**git add .**]
[**git status**]
[**git commit -m 'Added the files from the release folder'**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git commit -m "Added the files from the release folder"
[master f2746c6] Added the files from the release folder
 3 files changed, 5 insertions(+), 1 deletion(-)
 create mode 100644 Bin/release/info.txt
 create mode 100644 Bin/release/myprogram.dll

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git status
On branch master
nothing to commit, working tree clean
```

[**git log –oneline**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git log --oneline
f2746c6 (HEAD -> master) Added the files from the release folder
f737642 added pattern for gitIgnore on [Bb]in/**
524f0f7 Continuing with the .gitIgnore activity
0a2a8de Added the important resource files
43239f4 changes to tracked file in ignored folder are still tracked
8134e71 added the local gitIgnore file
4f86487 added info.txt during gitIgnore Activity
c8672c8 Added the h4 tag change
cb5204a Added the h3 tag for upcoming changes
331b291 Added an h2 tag to the details page
874d595 Added the rest of the files
f69f692 Added the About.html file
```

Should look something like this after working through both the status and the .gitIgnore activities.

Closing Thoughts

In this activity, we looked at creating both a global and a local .gitIgnore file. We were able to prove that the local .gitIgnore supercedes the global .gitIgnore field.

In our examination, we saw that it is very easy to exclude a folder and its subfolders, as well as include/exclude files by name. We also saw that it is possible to set patterns in the .gitIgnore that allow for pattern matching to apply the rules that are appropriate for the repo or user.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

Section 05 – Basic GIT operations: A general flow for a single person

I may not have gone where I intended to go, but I think I have ended up where I needed to be – Douglas Adams

Learning:

In this section we work through what it takes to work with GIT as a single developer. This means we'll be working with the basic commands of adding [staging] changes, committing changes locally and pushing to remote, and pulling from remote back to our local branch/repository.

Goals:

- Use GIT add and commit to stage and commit changes
- Use GIT push to push our local changes to the remote repository
- Use GIT pull to get changes from the remote repository

Tasks:

- Watch the videos for section five
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we should have no issues working with our own local and remote repository. This flow is great for a single developer that doesn't have to worry about sharing code with anyone or worry about others making changes to the code.

Activities:

- No official activities, but can work through making changes and pushing to remote, then make a change at remote and pull to local

Videos:

- No more fear of losing work
- Staging changes with git add
- Committing changes with git commit
- Pushing changes with git push
- Getting changes with git pull
- Viewing history with git log and/or git show

Notes

Section 06 – Tools to improve our ability to work with GIT in the BASH command line terminal

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency – Bill Gates

Learning:

In this section we'll cover setting up VSCode to be our default editor and difftool, so that we can move away from working with VIM in the console to edit files and view changes

Goals:

- Have VSCode or another editor setup as the default editor
- Have VSCode or another editor setup as the default difftool

Tasks:

- Watch the videos for section six
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we'll have VSCode (or another editor of choice) setup to perform edits and review changes. Using a program like VSCode will make sure we don't have to use VIM to do these things any longer.

Activities:

- Set VSCode as our default editor
- Set VSCode as our default difftool

Videos:

- Get a text editor for use when editing files
- Setting the default editor
- Setting and using a difftool to view differences
- Turning off the difftool prompt

Notes

GIT: From Beginner to Fearless

GIT Default Editor Activity:
Setting our default editor to VSCode

Brian Gorman, Author/Instructor/Trainer

Introduction

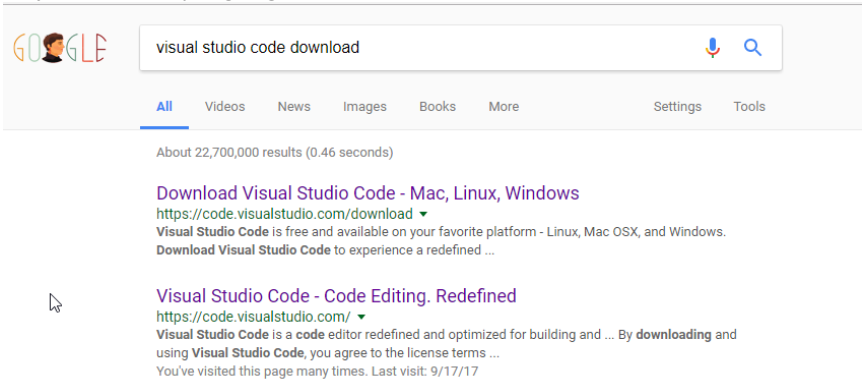
Although BASH allows us edit files via a call to VIM as the default editor, sometimes we might want to take a different approach. I remember the first time VIM opened up for a message and all I could think was "How the heck do I get out of this thing!" Long story short, unless VIM is your thing, you probably would also like another, more common/modern option. Please don't get me wrong here, I'm not "Bash"ing VIM, as it is quite a capable tool. Most of us these days are just used to a little more user-friendly.

Let's gets started!

Step 1: Get Visual Studio Code Setup [if you don't already have it]

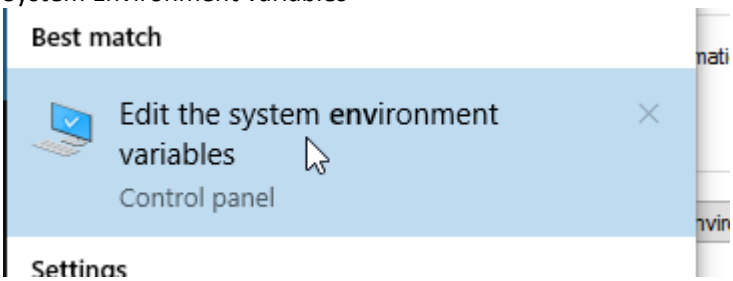
e) Download and install Visual Studio Code onto our machine.
Go To: https://code.visualstudio.com/download

Or just do a simple google search for Visual Studio Code:

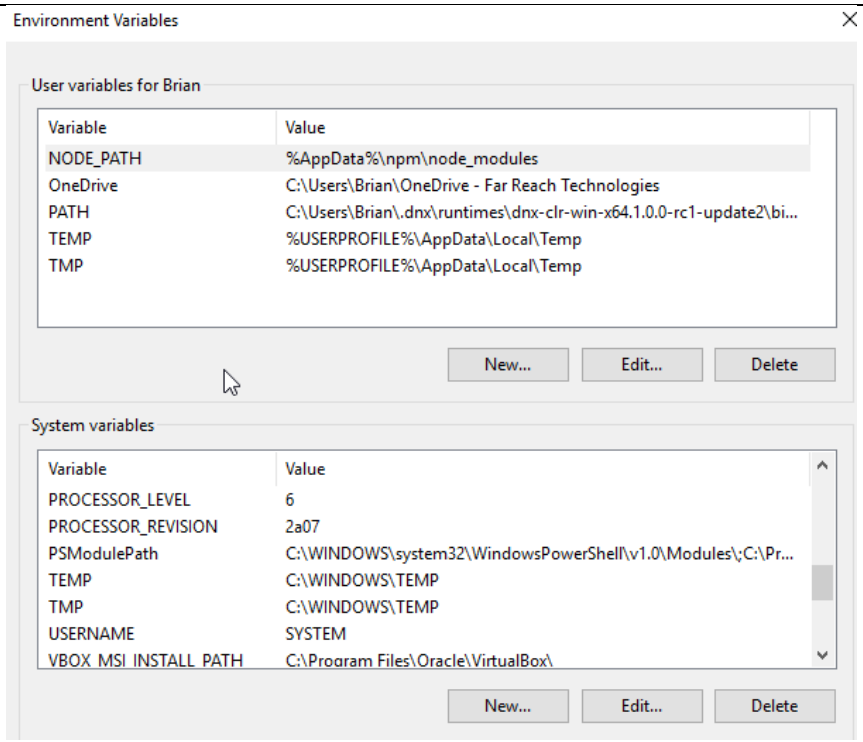


Install the application, start it up and make sure it works.
More information about the application can be found here:
https://code.visualstudio.com/docs

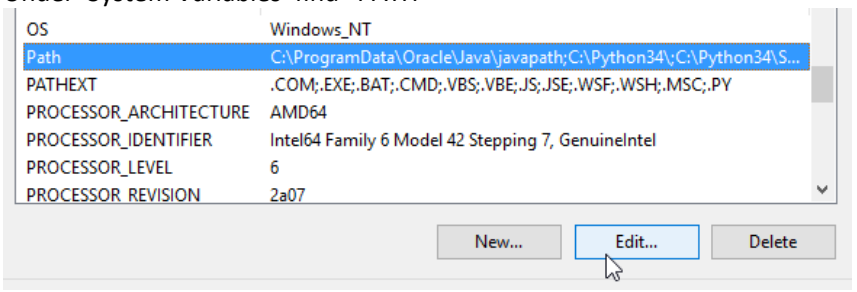
f) If on a windows machine, make sure code is in your PATH variable.
We don't have to do this, but it will make things a lot easier. We'll be able to reference the executable directly, rather than having to code the entire path to the executable.
Go to the start menu and type "Environment Variables" Then select "Edit System Environment variables"



Notes



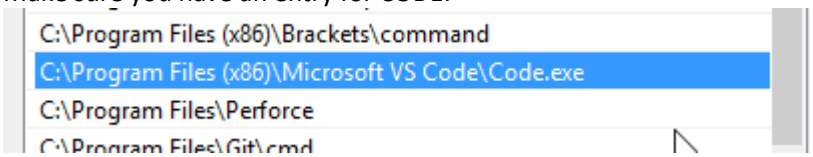
Under 'System Variables' find "PATH"



Select Edit.

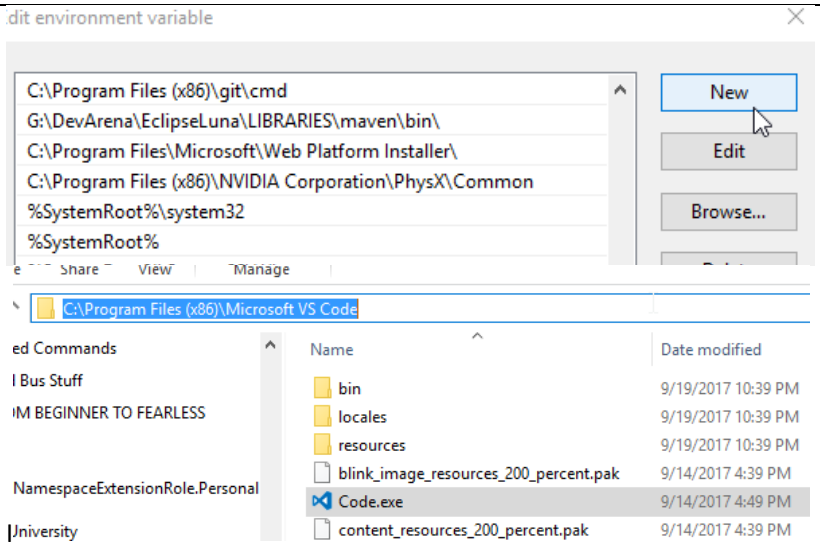
If you are on an older version of windows, you may need to parse the string in a text editor. It's much easier now in Windows 10:

Make sure you have an entry for CODE:



If you do not, then you will want to add one using the "NEW…" button:

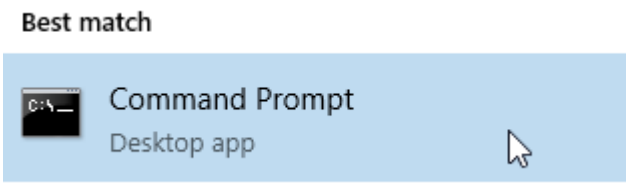
First find the path to your executable:



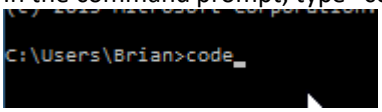
Then place that path into your Environment variables.

g) Test that it works:

Go to the start menu and type 'cmd'
Then select "Command Prompt"



In the command prompt, type "code":



If VS Code launches, you are all set. If not, you can either try again or just use the full path from above when setting values in the git config for difftool.

Step 2: Go to any repository

a) Make sure you are on any working repository. It doesn't even have to be up to date. We are not going to be affecting anything.

Our goal is to get our global config to have one entry for a default editor:

```
core.editor='C:\Program Files (x86)\Microsoft VS Code\code.exe' -w
```

For code, the critical item is the "-w" flag. Without it, BASH won't wait until we are done editing to complete our action. This would cause a big problem for some items later on. Also, it might be sufficient to just enter "code -w" here, when code is in our environment variables. However, I'm going to leave the full path so that I have it referenced and so that we have one that shows what it takes to setup the value with a full path in case environment variables are not working.

b) Unset a global config value.

Since my global config already has an entry for the core.editor, now is a great time to review how to unset a global value. You won't need to run this command now, but even if you did it wouldn't hurt anything:
[git config --global --unset <value>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (master)
$ git config --global --unset core.editor

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (master)
$ git config --global --get core.editor
```

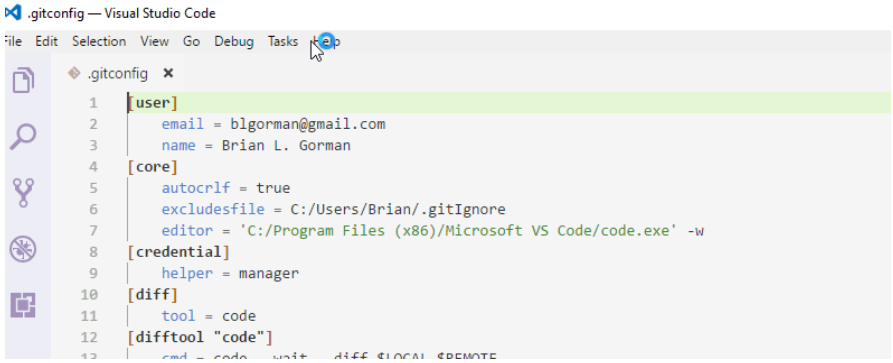
c) Set the core editor to be VSCode

To make this work, it is easiest to just modify the file in VIM.
[git config --global -e]
[hit <i> to insert]
[enter the value core.editor]
[core]
editor='C:/Program Files (x86)/Microsoft VS Code/code.exe' –w

Step 3: Verify editor is working

a) To see that the editor is working, all we need to do is just try to edit the global config file:

[git config --global -e]
//we know it's working if the editor pops up in code



This concludes our GIT Default Editor Activity.

Closing Thoughts

Setting VSCode [or another tool] to use for the default editor gives us a nice way to start working with a more user-friendly option [rather than VIM]. Again, VIM is perfectly fine if you like that tool, and so using a tool like VSCode is entirely optional.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner to Fearless

GIT Default MergeTool Activity: Setting our default Merge Tool to VSCode

Brian Gorman, Author/Instructor/Trainer

Introduction

Although BASH allows us edit files via a call to VIM as the default merge tool for resolving conflicts, using it is a very tricky operation at best. I remember the first time VIM opened up for a message and all I could think was “How the heck do I get out of this thing!”

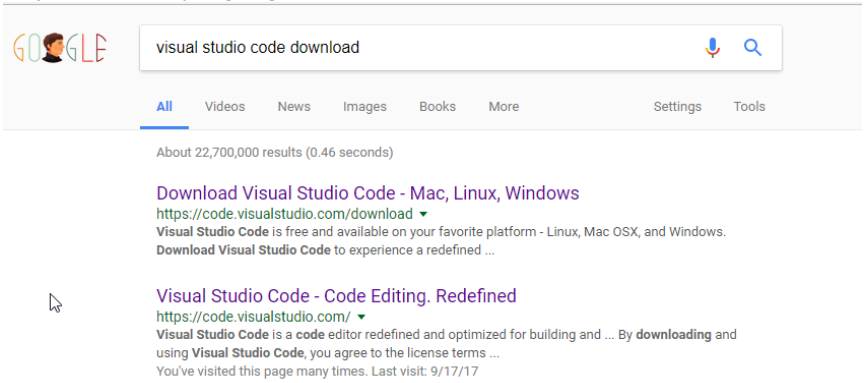
VSCode gives us an incredibly powerful mergetool that easily lets us select the source, target, or both changes, and also easily edit the changes during the merge operation. We definitely want a nice tool like this in place to make our lives easier.

Let’s gets started!

Step 1: Get Visual Studio Code Setup [if you don't already have it]

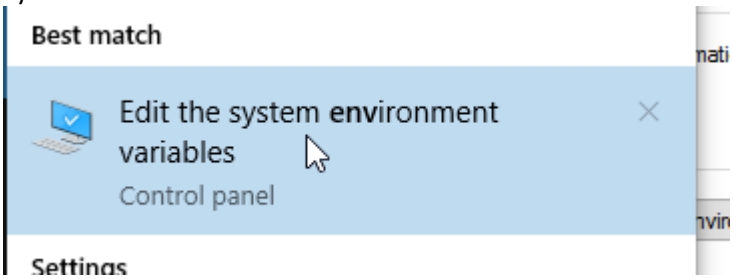
h) Download and install Visual Studio Code onto our machine.
Go To: https://code.visualstudio.com/download

Or just do a simple google search for Visual Studio Code:

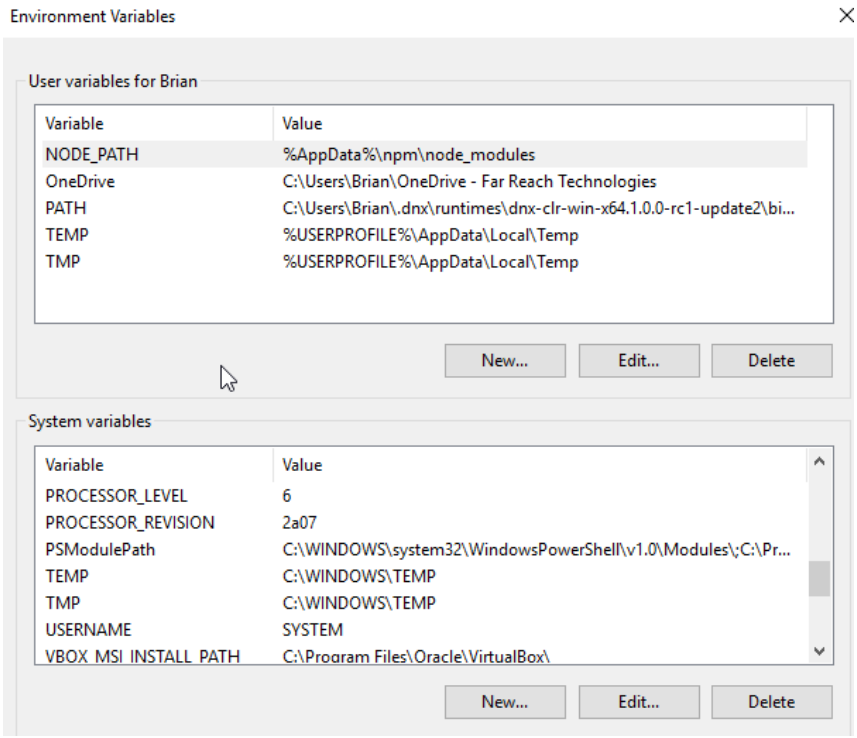


Install the application, start it up and make sure it works.
More information about the application can be found here:
https://code.visualstudio.com/docs

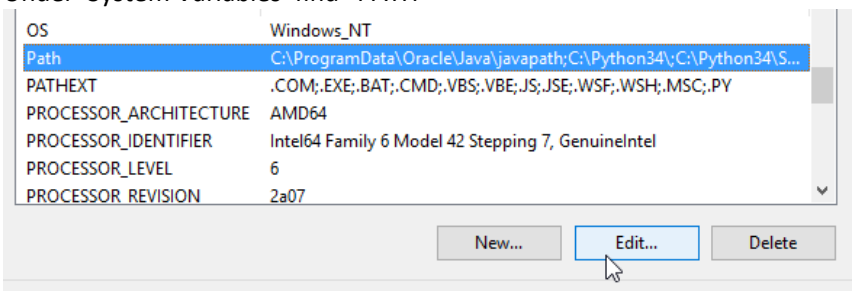
i) If on a windows machine, make sure code is in your PATH variable.
We don't have to do this, but it will make things a lot easier. We'll be able to reference the executable directly, rather than having to code the entire path to the executable.
Go to the start menu and type "Environment Variables" Then select "Edit System Environment variables"



Notes



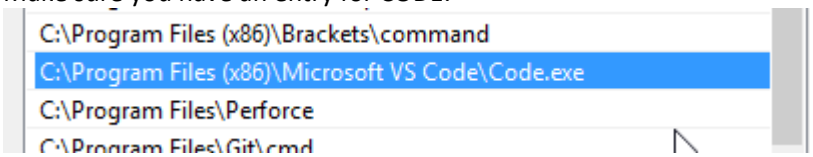
Under 'System Variables' find "PATH"



Select Edit.

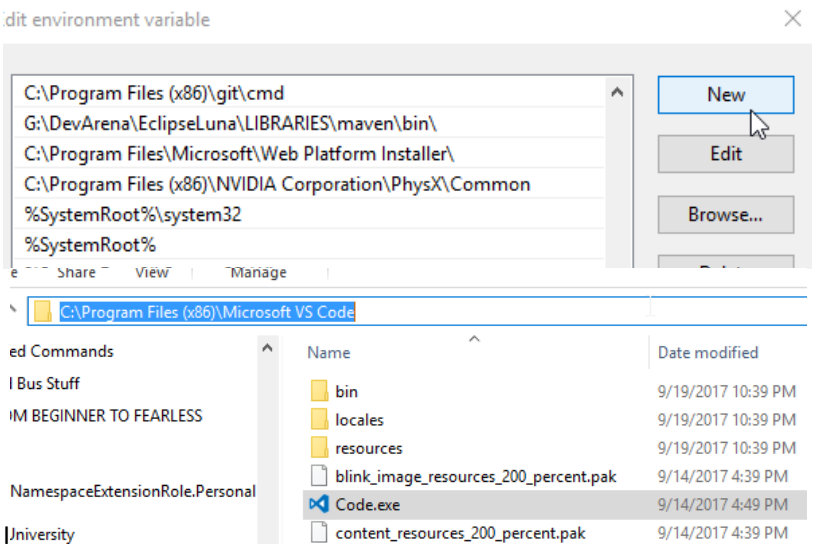
If you are on an older version of windows, you may need to parse the string in a text editor. It's much easier now in Windows 10:

Make sure you have an entry for CODE:



If you do not, then you will want to add one using the "NEW…" button:

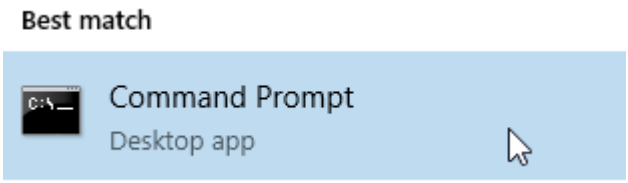
First find the path to your executable:



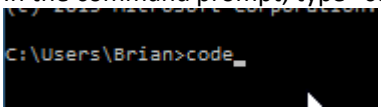
Then place that path into your Environment variables.

j) Test that it works:

Go to the start menu and type 'cmd'
Then select "Command Prompt"



In the command prompt, type "code":



If VS Code launches, you are all set. If not, you can either try again or just use the full path from above when setting values in the git config for difftool.

Step 2: Go to any repository

d) Make sure you are on any working repository. It doesn't even have to be up to date. We are not going to be affecting anything.

e) Take a quick look at config file settings:

[git config --global --list]

```
$ git config --global --list
user.name=Brian Gorman
user.email=brian@majorguidancesolutions.com
core.excludesfile=C:/Users/Brian/.gitIgnore
core.editor=code
diff.tool=default-difftool
difftool.default-difftool.cmd=code --wait --diff $LOCAL $REMOTE
difftool.prompt=false

Brian@Prometheus MINGW64 /c/Data/GFBTF/ultimate-default-web (mgs-45678-feature
```

No settings are in place for the merge tool at this point.

f) Add the entry for the merge tool

[git config --global merge.tool code]
[git config –global mergetool.code.cmd “code --wait $MERGED”]

```
Brian@Prometheus MINGW64 /c/Data/GFBTF/ultimate-default-web (mgs-456
)
$ git config --global merge.tool code

Brian@Prometheus MINGW64 /c/Data/GFBTF/ultimate-default-web (mgs-456
)
$ git config --global mergetool.code.cmd "code --wait $MERGED"
```

g) Add settings to avoid prompting and keeping backups

[git config –global mergetool.prompt false]
[git config --global mergetool.keepbackup false]

```
Brian@Prometheus MINGW64 /c/Data/GFBTF/ultimate-default-web (mgs-45678
)
$ git config --global mergetool.prompt false

Brian@Prometheus MINGW64 /c/Data/GFBTF/ultimate-default-web (mgs-45678
)
$ git config --global mergetool.keepbackup false
```

h) Review the config settings:

[git config --global --list]

```
$ git config --global --list
user.name=Brian Gorman
user.email=brian@majorguidancesolutions.com
core.excludesfile=C:/Users/Brian/.gitIgnore
core.editor=code
diff.tool=default-difftool
difftool.default-difftool.cmd=code --wait --diff $LOCAL $REMOTE
difftool.prompt=false
merge.tool=code
mergetool.code.cmd=code --wait
mergetool.prompt=false
mergetool.keepbackup=false
```

--looks like the code command didn’t work as expected

i) Enter the editor and setup the command for mergetool as expected manually

[git config --global -e]

```
Brian@Prometheus MINGW64 /c/D
)
$ git config --global -e
```

Enter "$MERGED" into the mergetool.code.cmd line

```
[merge]
    tool = code
[mergetool "code"]
    cmd = "code --wait $MERGED"
[mergetool]
    prompt = false
    keepbackup = false
```

Step 3: Verify mergetool is setup

b) To see that the mergetool is working, we'll need a conflict to resolve. When we get to that, if something doesn't work, come back to this activity and make sure that everything is setup as expected. For now, let's just take a quick look:
[git config --global -e]

Make sure have the entries as above or as shown here [same values]

```
[merge]
    tool = code
[mergetool "code"]
    cmd = "code --wait $MERGED"
[mergetool]
    prompt = false
    keepBackup = false
```

This concludes our GIT Default MergeTool Activity.

Closing Thoughts

Setting VSCode [or another tool] to use for the default merge tool gives us a nice way to start working with a more user-friendly option [rather than VIM]. Again, VIM is perfectly fine if you like that tool, and so using a tool like VSCode is entirely optional, but highly recommended, especially for merge operations.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

Section 07 – Branching and Merging

Two roads diverged in a wood and I – I took the one less traveled by, and that has made all the difference – Robert Frost

Learning:

In this section we'll see what it takes to work with branches. Branches are great because they give use the ability to work on a feature without any fear of breaking the "master" code branch. When we're ready to commit, we can push our changes and then merge them into master. Alternatively, we could merge locally and then push to remote – but this is a very bad idea if more than one developer is involved.

Goals:

- Understand how to work with branches
- Be able to create and close a pull request

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we should have no trouble creating branches and making changes locally to the branches, then pushing to remote. We'll also understand how to create a pull-request and merge our changes to the remote repository.

Activities:

- Branching
- Merging locally
- Basic branching and merging with a pull request

Videos:

- Introduction to branching
- Make sure to have latest when starting
- Create a local branch, make changes, commit, push
- Merge locally
- Merge with a pull request
- Deleting branches
- Pruning references

Notes

GIT: From Beginner To Fearless

GIT Branching Activity:
A Single developer branching exercise

Brian Gorman, Author/Instructor/Trainer

Introduction

Branches are one of the key functions of working with GIT. While a single developer could work on a repository without ever branching, using branches is a critical way to make sure that it's very easy to protect your work and have the flexibility to make changes without worrying about messing up your repository.

In this activity, we're going to learn about using branches. By creating and using a branch, we have the opportunity to start working, save our changes with a commit, and easily switch back to the previous version if need be. Additionally, after we have our changes completed, we can then merge our changes into our main master branch.

This activity will NOT complete the circuit with a merge (we'll look at that in the next activity to finish this up). However, we'll take the time to learn about creating and working with branches, which will set us up for the next steps.

Let's get started!

Step 1: Creating a new branch

k) Make sure you are in any repository, and that you are on master and up-to-date. For this activity, we'll be using our default web, but you can use any repository.

Always remember to run the following commands from the master branch before starting new work:
[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$
```

l) Create a new feature branch

You might think the command to create a new branch would include the word "branch" in it, however, that would be incorrect. Instead, if we want to create a new branch, we use the checkout command with a flag '-b'.

[git checkout -b My-Feature-Branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout -b My-Feature-Branch
Switched to a new branch 'My-Feature-Branch'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$
```

Note that not only did I create the new branch, but I also checked it out, and 'switched' to the new branch.

Step 2: Working on the branches

a) Work on your feature branch

Create some changes on your branch using VSCode or VIM in the details.html file:
[code details.html]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ code details.html
```

Notes

Here I'm simply adding an h2 to the page:

```
43          <div class="container body-content">
44
45              <h1> Details Page with more content... </h1>
46
47              <h2> I'm making some changes! </h2>
48              <hr />
49              <footer>
50                  <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidan
51              </footer>
52          </div>
```

Save it and close your editor.
[git status -s]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Ac
$ git status -s
 M details.html
```

So I have changes. I want to add them. Switching branches with changes will keep the changes as we move to the new branch, so beware of this!
[git commit -am 'changes to details']

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git commit -am 'changes to details'
[My-Feature-Branch 7b6a932] changes to details
 1 file changed, 1 insertion(+)
```

View our history:
[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (M
$ git log --oneline
7b6a932 (HEAD -> My-Feature-Branch) changes to details
b7edba6 (origin/master, master) initial commit
```

b) Switch branches

Next we're going to switch back to the master branch. By doing this, we'll see our change go away, as well as our commit:
[git checkout master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

[git status -s]
[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git status -s

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
b7edba6 (HEAD -> master, origin/master) initial commit
```

NOTE: the commit 7b6a932 is GONE!
This is EXPECTED. Our branch is "1 commit ahead of master" and that commit contains our change to details.

Open the file to see the change is missing:
[code details.html]

```
        </div>
        <div class="container body-content">

            <h1> Details Page with more content... </h1>

            <hr />
            <footer>
                <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidance Solut
            </footer>
        </div>

        <!-- Scripts -->
        <!--<script src="js/lib/jquery-3.2.1.min.js" type="text/javascript"></script>-->
```

[don't make any changes, and don't close it. Or close it and then re-open after switching].

So working on a branch allows us to keep our "master" protected in case some of the changes we are making go horribly wrong!

c) Switch back to feature branch

[git checkout My-Feature-Branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout My-Feature-Branch
Switched to branch 'My-Feature-Branch'
```

IF you left code open, look at it now!
If you didn't leave it open, type [code details.html]

```
    </div>
    <div class="container body-content">

        <h1> Details Page with more content... </h1>

        <h2> I'm making some changes! </h2>
        <hr />
        <footer>
            <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidance Solut
        </footer>
    </div>
```

Our changes are back!

Also, check the log:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git log --oneline
7b6a932 (HEAD -> My-Feature-Branch) changes to details
b7edba6 (origin/master, master) initial commit
```

Our commit is back, just like we expected.

Show the changes [note, you'll need to replace the commit id to what you see in your log report]:
[git show <commitid>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git show 7b6a932
commit 7b6a9326e08d970880b868136da7a0dcf8c5820e (HEAD -> My-Feature-Branch)
Author: Brian L. Gorman <brian@majorguidancesolutions.com>
Date:   Sat Oct 14 19:15:37 2017 -0500

    changes to details

diff --git a/details.html b/details.html
index 8767a1e..254f664 100644
--- a/details.html
+++ b/details.html
@@ -44,6 +44,7 @@

             <h1> Details Page with more content... </h1>

+            <h2> I'm making some changes! </h2>
             <hr />
             <footer>
                 <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com
">Major Guidance Solutions</a></p>
```

Leave this branch in tact. We're going to use this in the next activity (merging).

Step 3: Other [git branch] commands of note

a) Create a branch locally to delete later.

Switch to master
[git checkout master]

Make sure you left the other branch alone. Switch to a new branch
[git checkout -b <some-branch-name-here>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout -b another-branch
Switched to a new branch 'another-branch'
```

As an FYI – if you forget the –b, you'll get a 'pathspec' error:
[git checkout no-such-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (another-branch)
$ git checkout no-such-branch
error: pathspec 'no-such-branch' did not match any file(s) known to git.
```

b) Listing branches

Now switch back to master
[git checkout master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (another-branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

List your branches
[git branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch
  My-Feature-Branch
  another-branch
* master
```

List all the branches. Use the -a flag to list all branches, including branches at remote that we know about.
[git branch -a]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -a
  My-Feature-Branch
  another-branch
* master
  remotes/origin/master
```

c) Delete the local branch

NOTE: you need to NOT be on the branch you want to delete, and it needs to not have changes at this point. Note: DO NOT DELETE My-Feature-Branch!

[git branch -d another-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -d another-branch
Deleted branch another-branch (was b7edba6).
```

[git branch]

```
Brian@SENTINEL MINGW64 /g/Data
$ git branch
  My-Feature-Branch
* master
```

d) Force delete a local branch

Switch back to your feature branch that is one commit ahead of master
[git checkout My-Feature-Branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/Defau
$ git checkout My-Feature-Branch
Switched to branch 'My-Feature-Branch'
```

Now create a new branch from here [it will be like My-Feature-Branch, with a commit and changes – so anytime you need to experiment from any branch, you can always just branch off of the branch!]
[git checkout -b my-feature-branch-experiment]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git checkout -b my-feature-branch-experiment
Switched to a new branch 'my-feature-branch-experiment'
```

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (
riment)
$ git status
On branch my-feature-branch-experiment
nothing to commit, working tree clean
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (my-feature-branch-expe
riment)
$ git log --oneline
7b6a932 (HEAD -> my-feature-branch-experiment, My-Feature-Branch) changes to det
ails
b7edba6 (origin/master, master) initial commit
```

Note: the branch is on the same commit as the parent it was created from. If you wanted, you could do another commit and log to see the current feature branch move ahead another commit.

Now comes the fun.

Switch back to master:
[git checkout master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activit
riment)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

And delete the experiment branch:
[git branch -d my-feature-branch-experiment]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -d my-feature-branch-experiment
error: The branch 'my-feature-branch-experiment' is not fully merged.
If you are sure you want to delete it, run 'git branch -D my-feature-branch-expe
riment'.
```

What? Why did that happen?
Because the commit history shows that this branch is one ahead of master, just like the other branch would, and GIT gives you a protective layer – an "are you sure you want to do this" warning!

Now, we know we do want to do this, so we'll force the issue by changing the -d to -D. Yes, seriously, all it takes to force the issue is making a small d into a capital D.
[git branch –D my-feature-branch-experiment]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -D my-feature-branch-experiment
Deleted branch my-feature-branch-experiment (was 7b6a932).
```

[git branch -a]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -a
  My-Feature-Branch
* master
  remotes/origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
```

This concludes our git branching activity.

Closing Thoughts

In this activity, we have briefly seen how to work with branches. We noticed that branches are lightweight and easy to checkout, and our changes definitely live within the branches we commit to.

We got to see what it takes to create local branches, change some things, make a commit, and then switch branches to see the commit go away, followed by switching back to see it come back.

We also learned about deleting branches with no changes, and then we finished up by seeing how to delete a branch that had unmerged changes. During that last part, we also saw how the location branch we are in when creating a branch is critical. For this reason, most of the time you'll want to create branches from master. However, if you want to experiment, you can always create a branch from any location, and nest them infinitely, if you so desire.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Merging Activity:
A simple multiple-developer merging exercise

Brian Gorman, Author/Instructor/Trainer

Introduction

We've created a feature branch, and now we want to make those changes permanent. How do we do this? With a MERGE!.

Merging is important because this allows us to get our repository master history updated with the changes that we want to keep which we've created on a feature branch. Eventually, we'll learn that with multiple people working on the repository, conflicts will arise that we'll need to resolve. However, once we have merging under our belt, conflict resolution is just another easy step towards mastering our fears and working with GIT.

In this activity, we rely on changes from the branching activity. If you haven't done that activity, please complete it now. We're going to pick up where we left off on that activity, and merge our changes to our master locally, and then push them out to GitHub.

Let's gets started!

Step 1: Make sure we have changes to merge

m) Check to see that we have changes to merge.

Open our GIT BASH terminal, and review branches. We should have at least two, master, and MY-Feature-Branch. If we don't have these, make sure to complete the branching activity before continuing:
[git checkout master]
[git fetch origin]
[git pull origin master]
[git branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch
  My-Feature-Branch
* master
```

n) Switch to the branch that has changes

[git checkout My-Feature-Branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout My-Feature-Branch
Switched to branch 'My-Feature-Branch'
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git log --oneline
7b6a932 (HEAD -> My-Feature-Branch) changes to details
b7edba6 (origin/master, master) initial commit
```

We see that we have at least one commit to merge.

Step 2: Merge the changes

a) Switch to master.

[git checkout master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (My-Feature-Branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Notes

Now, before we do this, I want to say something. Ordinarily you don't want to do things this way. We should ***~~never~~ almost never*** modify master directly – instead, use feature branches, push to a remote like GitHub or BitBucket, and then utilize pull requests to merge code.

However, for this demo, we're going to change master locally, and push it to directly to GitHub.

Again, if you are working on a team, this could be very dangerous, especially if someone else merged changes into master first. If you are by yourself, there is less risk involved in working with this workflow.

b) Merge the changes locally

Make sure you are on master.
[git checkout master]
Then merge your feature branch
[git merge My-Feature-Branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git merge My-Feature-Branch
Updating b7edba6..7b6a932
Fast-forward
 details.html | 1 +
 1 file changed, 1 insertion(+)
```

So now my local master is one commit ahead of origin, has the same commit as the feature branch did:
[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
7b6a932 (HEAD -> master, My-Feature-Branch) changes to details
b7edba6 (origin/master) initial commit
```

Note how HEAD -> Master is one ahead of origin/master and is even with My-Feature-Branch

[we can now safely delete My-Feature-Branch locally]

Step 3: Push your changes to GitHub

a) Pushing the changes

Now that we have our changes committed locally, we need to get the changes out to GitHub. This will make sure that our REMOTE repository has the changes and we will never have to fear losing them.

To push master, we don't need to use the -u flag, because the branch is already present at REMOTE. The -u flag is only needed when a branch is untracked.

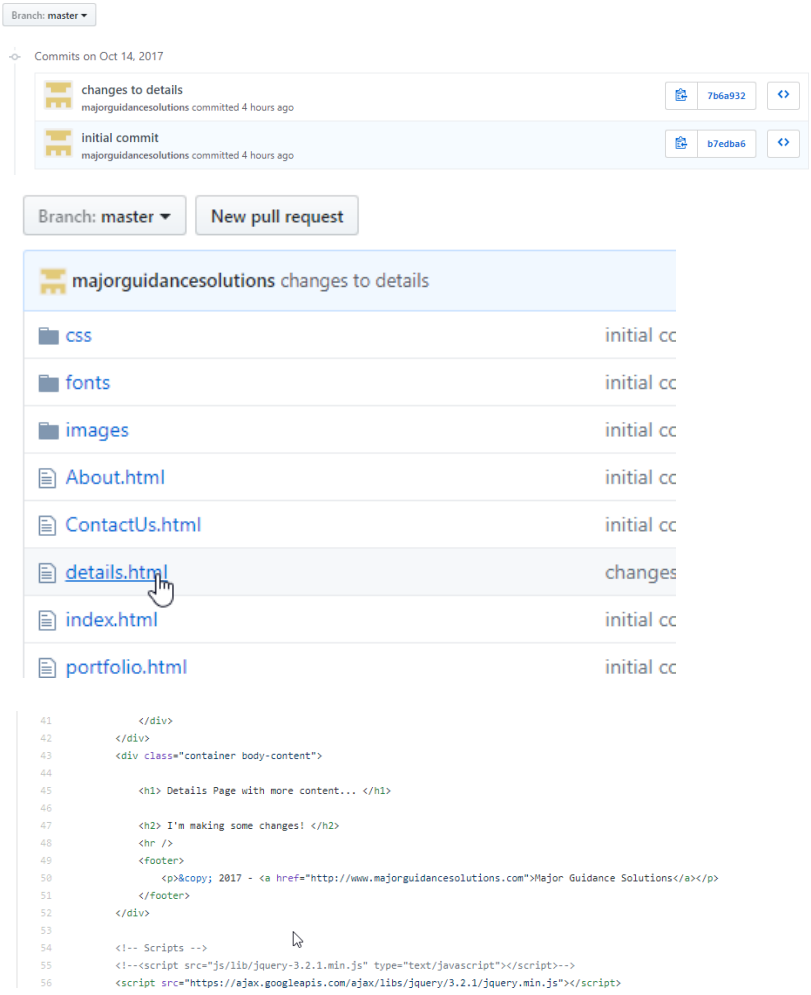
Again: please don't do this on a team project:
[git push origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 342 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/defaultweb_activity.git
   b7edba6..7b6a932  master -> master
```

b) Verify changes at GitHub

Browse to the repo and validate that the file we modified has been updated at the REMOTE repository.

Check the commit history -> the most recent commit should have the message that we entered on our last commit and have the same commit id that we saw a few times in the git log oneline




```
            </div>
        </div>
        <div class="container body-content">

            <h1> Details Page with more content... </h1>

            <h2> I'm making some changes! </h2>
            <hr />
            <footer>
                <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidance Solutions</a></p>
            </footer>
        </div>

        <!-- Scripts -->
        <!--<script src="js/lib/jquery-3.2.1.min.js" type="text/javascript"></script>-->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
```

This concludes our merging activity.

Closing Thoughts

In this initial merging activity, we see how we can move our changes from our local feature branch to the local master branch. After we merged the changes to master, we then pushed our changes out to the remote repository at GitHub.

Since the changes were already in master, no merging was needed at GitHub, and therefore no new commits were generated -> the changes were just updated. This is ok when we are the only person modifying the repository, but if there were multiple people interacting with the repository, this would actually be very dangerous. As we continue to learn through this course, we'll see better ways to go about getting our changes out to the remote repository.

Also, for almost every single case, we should try to avoid directly merging to the master like we did in this activity. Again, this is ok if we are the only person interacting with the repo, but for a team of people, history conflicts could become a very difficult problem to solve.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

Basic Branching and merging with a pull request

Brian Gorman, Author/Instructor/Trainer

Introduction

We've made some strides to this point, learning about branches and seeing how we can then merge the changes from one local branch to another. However, as mentioned in the basic merging activity, it's dangerous to merge directly to master on the local repository and then push to the remote master branch. If there are more than one users working against the repository, each of the users will be dependent on a common history tree for master. Once someone changes this history tree, the other users would have to resolve the conflicts that could arise from changes to that history. Generally simple merging won't cause too many problems, but other operations could.

To get around this issue, GitHub has the ability to create 'pull requests,' which allows for the changes to be merged at GitHub from one branch to another, including master. This is a much safer way to make sure that changes are not lost and history is not corrupted for other users of the public master branch.

In this activity, we'll take a look at making local changes and pushing the branch to GitHub, then resolving with a pull request. After the pull request is completed, a merge commit will be created at master. In order to keep our local repository in sync, we'll finish the activity by performing the fetch and pull operations to update our local repository from the remote repository.

Let's get started!

Step 1: Make sure your local repository is up to date

o) Get the latest version of the remote master

Perform the operations that follow to make sure our current local repository is in sync.
[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

p) Create a local branch and push the branch to GitHub.

[git checkout -b feature-branch-one]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout -b feature-branch-one
Switched to a new branch 'feature-branch-one'
```

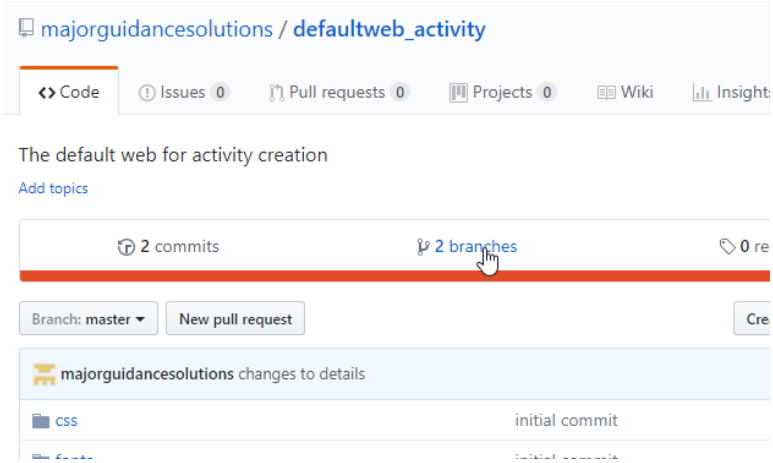
Now push the branch using the -u flag since it is unpublished:

[git push origin -u feature-branch-one]

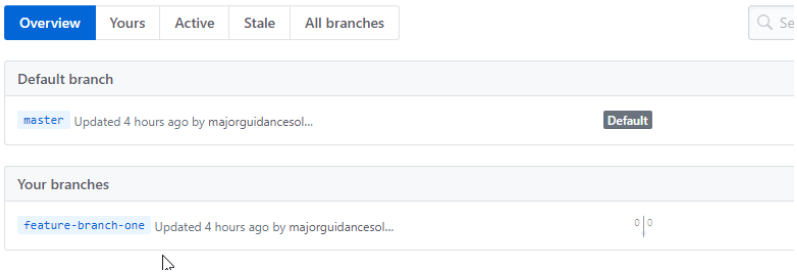
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-branch-one)
$ git push origin -u feature-branch-one
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/majorguidancesolutions/defaultweb_activity.git
 * [new branch]      feature-branch-one -> feature-branch-one
Branch feature-branch-one set up to track remote branch feature-branch-one from
origin.
```

Notes

Validate the branch was pushed. At GitHub, select the 'branches' tab:



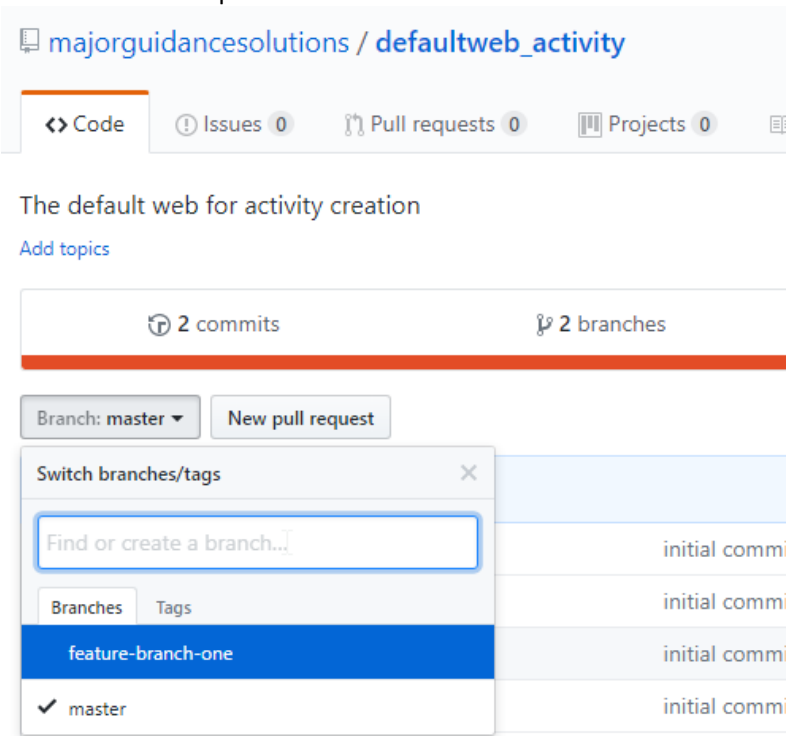
Validate the new branch is there:



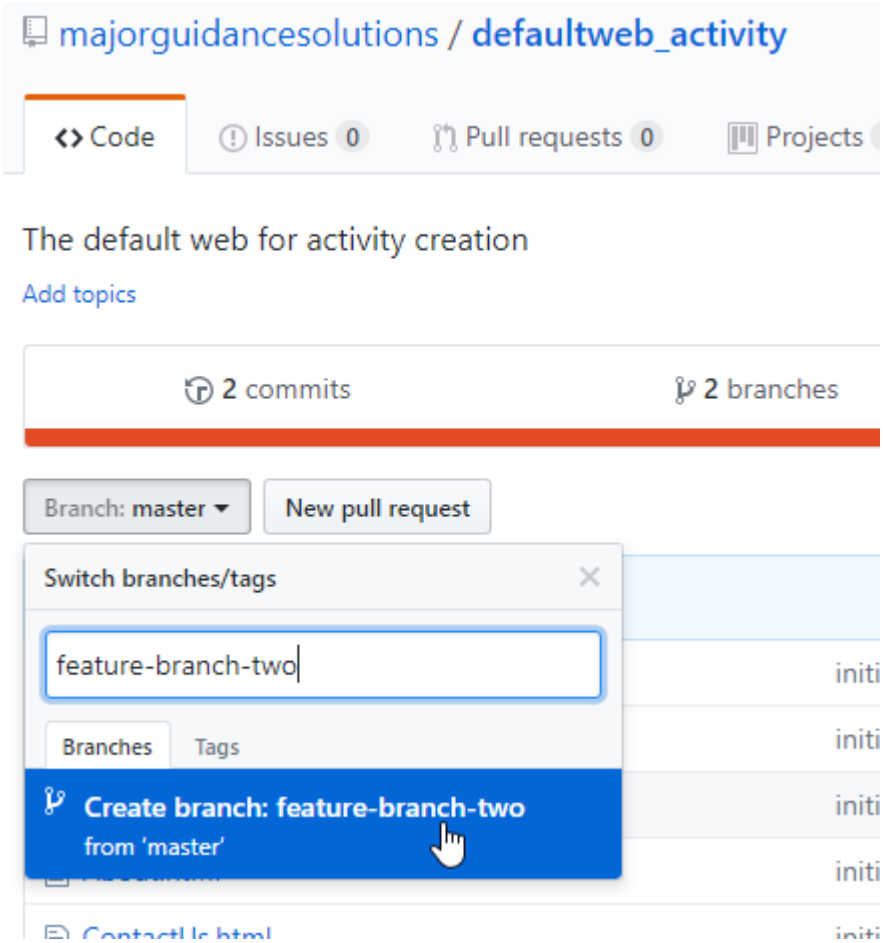
Step 2: Create a feature branch at GitHub

a) Log in to GitHub and browse to your repository

After getting to the correct repository, locate the "Branch:" dropdown (which should be pointing to master. Select this dropdown and enter a new branch name in the empty box. Also note, we could switch to our other branch(es) that are listed in the dropdown if we would like:

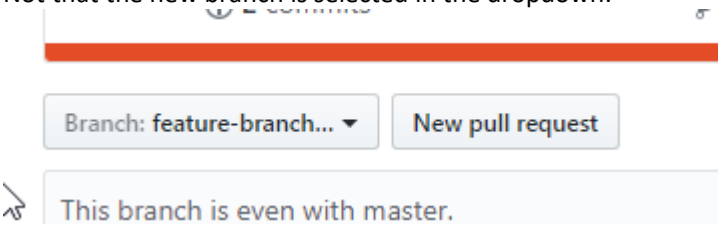


b) Enter the branch name into the dropdown

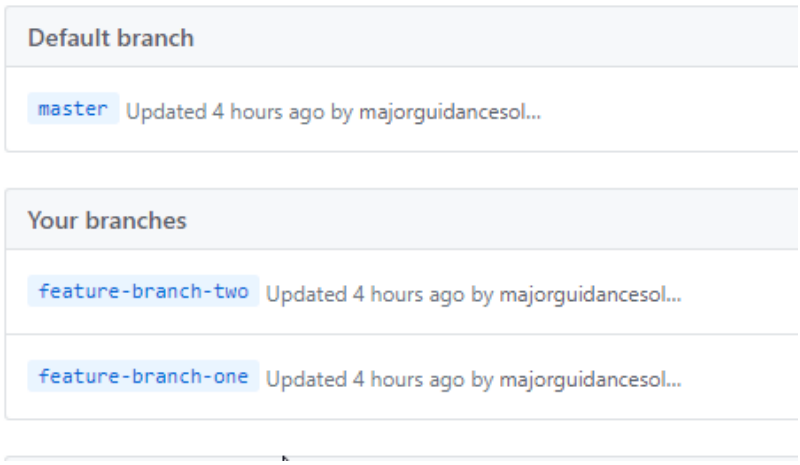


Then select the “Create Branch” button

Not that the new branch is selected in the dropdown:



And validate the branch is listed:



Step 3: Bring the latest changes back to our local repo

a) Now that we created a new branch at GitHub, we need to get it locally

[git checkout master]
[git fetch origin]
[git pull origin]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-bran
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin
From https://github.com/majorguidancesolutions/defaultweb_activity
 * [new branch]      feature-branch-two -> origin/feature-branch-two

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin
Already up-to-date.
```

[git branch -a]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -a
  My-Feature-Branch
  feature-branch-one
* master
  remotes/origin/feature-branch-one
  remotes/origin/feature-branch-two
  remotes/origin/master
```

b) Checkout the feature branch to create a local reference to it

[git checkout feature-branch-two]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout feature-branch-two
Switched to a new branch 'feature-branch-two'
Branch feature-branch-two set up to track remote branch feature-branch-two from
origin.
```

Note that we didn't need to use the -b flag to create the branch since a branch ref to remote already existed. The new branch is already setup and tracks to remote as expected.

Step 4: Make some changes on either feature branch, but only use one of them [for now], then push and merge at GitHub

a) Choose a branch to work on. For example, feature-branch-one

Make some small changes in the details.html file.
[git checkout feature-branch-one]
[code details.html]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-branch-two)
$ git checkout feature-branch-one
Switched to branch 'feature-branch-one'
Your branch is up-to-date with 'origin/feature-branch-one'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-branch-one)
$ code details.html
```

[you can make any changes you would like]

```
    <h2> I'm making some changes! </h2>
    <h3> Changes made on feature-branch-one for merge at GitHub</h3>
    <hr />
    <footer>
```

Add and commit your changes

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-branch-one)
$ git commit -am 'Changes on feature-branch-one'
[feature-branch-one 7c4f044] Changes on feature-branch-one
 1 file changed, 1 insertion(+)
```

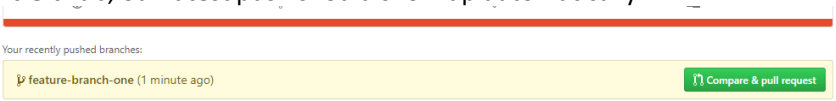
b) Push the changes out to GitHub [do not merge to the local master!]

[git push origin feature-branch-one]

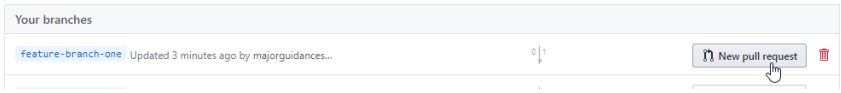
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (feature-branch-one)
$ git push origin feature-branch-one
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 378 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/defaultweb_activity.git
   7b6a932..7c4f044  feature-branch-one -> feature-branch-one
```

c) Create a pull request at GitHub

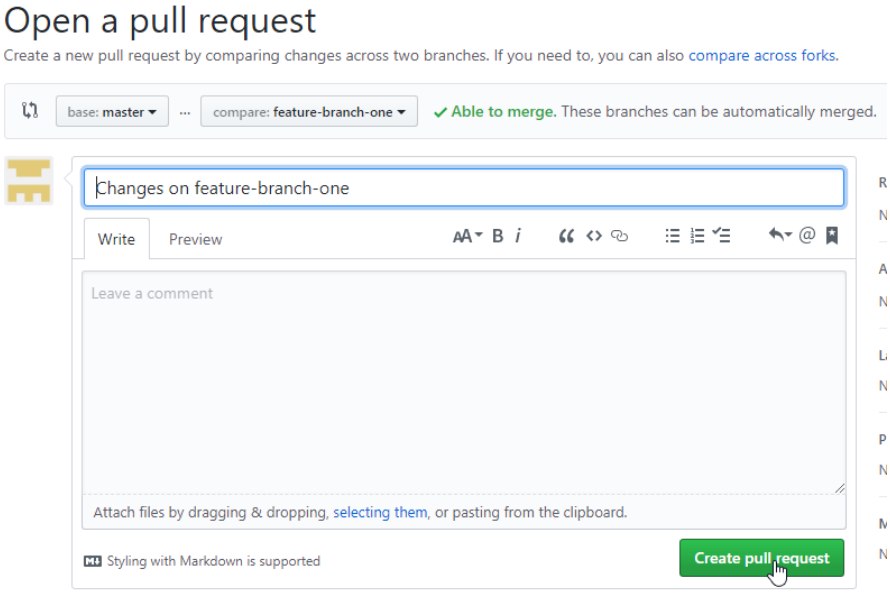
At GitHub, our latest push should show up automatically:



We can click on 'compare & pull request' here. We can also browse to the branch and select 'new pull request'. Also note that on the branch view, we can see that this branch is 1 commit ahead of master and 0 commits behind

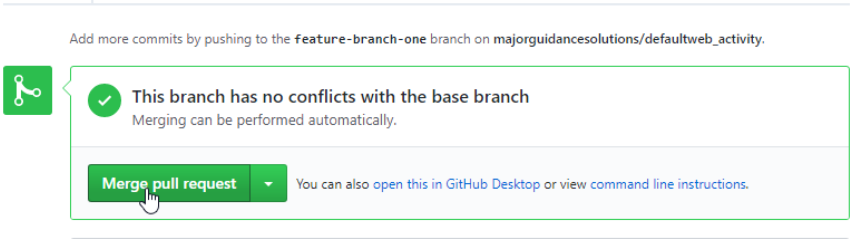


Click on either of the two buttons, and then click “Create Pull Request”



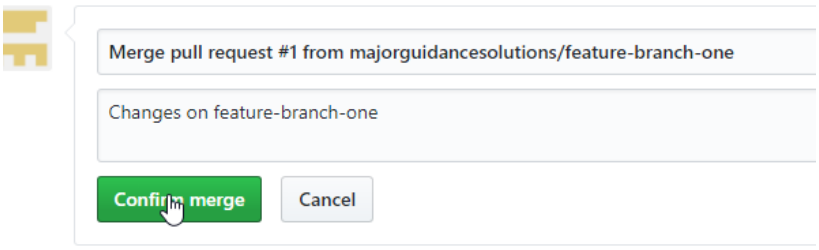
d) Merge the pull request

Click ‘Merge pull request’



And “Confirm Merge”

Also note, you could put details in here and change the commit message. We are creating a new commit in history for the actual merge, so that message will show up in the history (i.e. Merge pull request #1…)



Unless there are specific reasons, it's generally a good idea to delete the branch once it's been merged.

Step 5: Sync the changes locally to make sure our history is up to date

[git checkout master]
[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity
$ git log --oneline
7b6a932 (HEAD -> master, origin/master, origin/feature-b
h-two, My-Feature-Branch) changes to details
b7edba6 initial commit
```

Even though it says we are up to day, we are not!

[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/majorguidancesolutions/defaultweb_activity
   7b6a932..69f03ea  master     -> origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Updating 7b6a932..69f03ea
Fast-forward
 details.html | 1 +
 1 file changed, 1 insertion(+)
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
69f03ea (HEAD -> master, origin/master) Merge pull request #1 from majorguidance
solutions/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

Now we are up to date!

Step 6: Cleanup

a) Optional cleanup

At this point, we could do some cleanup. For example, we could delete the feature branch locally and at GitHub. Or, if we wanted, we could keep working on the feature branch, but the best step would be then to merge or rebase origin/master into the feature branch first so as to keep the history up to date.

Feel free to practice cleaning up the branches that have been merged. We'll learn other strategies to clean up our remote and local repositories later in the course.

Closing Thoughts

During this activity, we have seen how we can create our changes locally on a feature branch, and then push those changes at the branch level to GitHub. We then saw how we can perform a basic merge with a pull request at GitHub. Once that was done, we needed to pull the changes back to our local to make sure we are in sync going forward.

Ordinarily, there would be some cleanup after this, such as deleting the branches at both remote and local. If you would like to cleanup the feature-branch-one now, please feel free to do so. We know it is incredibly easy to create branches locally, so we can always create a new version of the branch.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

Section 08 – A common/simple multi-developer flow with merge conflicts

Two roads diverged in a wood and I – I took the one less traveled by, and that has made all the difference – Robert Frost

Learning:

In this section we'll begin learning about working as a team, creating branches on our team project, and then merging the code with pull requests.

Goals:

- Understand how to work with branches
- Be able to create and close a pull request

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we should have no trouble creating branches and making changes locally to the branches, then pushing to remote. We'll also understand how to create a pull-request and merge our changes to the remote repository.

Activities:

- Team Flow with Merge Conflict

Videos:

- Introduction to branching
- Make sure to have latest when starting
- Create a local branch, make changes, commit, push
- Merge locally
- Merge with a pull request
- Deleting branches
- Pruning references

Notes

GIT: From Beginner To Fearless

Team Branching and merging with a pull request and conflict resolution

Brian Gorman, Author/Instructor/Trainer

Introduction

If you've followed along to this point, we are in a really good place with understanding how to get changes out to the repository at GitHub (or other REMOTE). However, we've only taken the happy path so far – where only one user is pushing and there have been no conflicts.

In the real world, for 90% of the people who will be using GIT, conflicts will be a daily occurrence and resolving them while merging will become part of the necessary routine.

In this activity, we're going to simulate what might happen in a team environment. To keep this very simple, we'll be doing all of the stuff ourselves, so we'll have to pretend some of the changes come from another developer. If you really want to take it to the next level, you could go to the level of setting up an organization, creating a second account and using a team repository for your two accounts. However, to simulate the team for this activity, we aren't going to go that deep.

The general flow of what we'll do is to get the current version of the repository to our local machine. We'll then make some changes on a feature branch on our machine, and push them up to GitHub. In the meantime, we'll create a branch at GitHub and make a change there which will conflict with our changes. We'll merge those changes in, and then see what it would take to resolve the conflict on our local machine.

Let's get started!

Step 1: Make sure your local repository is up to date

q) Get the latest version of the remote master

Perform the operations that follow to make sure our current local repository is in sync.
[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

r) Create a local branch and push the branch to GitHub.

[git checkout -b developer-one-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout -b developer-one-branch
Switched to a new branch 'developer-one-branch'
```

Now push the branch using the -u flag since it is unpublished:

[git push origin -u developer-one-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git push origin -u developer-one-branch
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/majorguidancesolutions/defaultweb_activity.git
 * [new branch]      developer-one-branch -> developer-one-branch
Branch developer-one-branch set up to track remote branch developer-one-branch f
rom origin.
```

Notes

Step 2: Make some changes on your local branch and push

a) Create a simple change

[code details.html]

Enter a change similar to the following:

```
<div class="container body-content">

    <h1> Details Page with more content... </h1>

    <h2> I'm making some changes! </h2>
    <h3> Changes made on feature-branch-one for merge at GitHub</h3>

    <h1>These are developer one's changes, created during team simulation</h1>

    <hr />
    <footer>
        <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidance Solut
    </footer>
</div>
```

b) Make sure you have saved and have changes to commit:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git status -s
 M details.html
```

c) Add, Commit, Push

[git commit –am 'Developer one critical changes']

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git commit -am 'Developer one critical changes'
[developer-one-branch 8d80062] Developer one critical changes
 1 file changed, 3 insertions(+)
```

[git push origin developer-one-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git push origin developer-one-branch
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 399 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/defaultweb_activity.git
   69f03ea..8d80062  developer-one-branch -> developer-one-branch
```

d) Review current commit history

[git log --oneline]

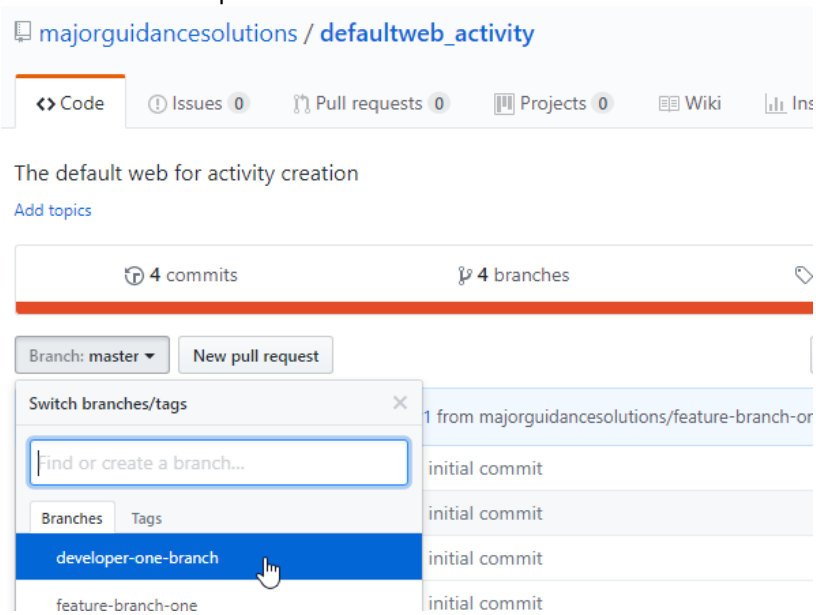
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git log --oneline
8d80062 (HEAD -> developer-one-branch, origin/developer-one-branch) Developer on
e critical changes
69f03ea (origin/master, master) Merge pull request #1 from majorguidancesolution
s/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

Note your recent commit id [mine is 8d80062].

Step 3: Create a feature branch at GitHub then Merge a change from it directly at GitHub

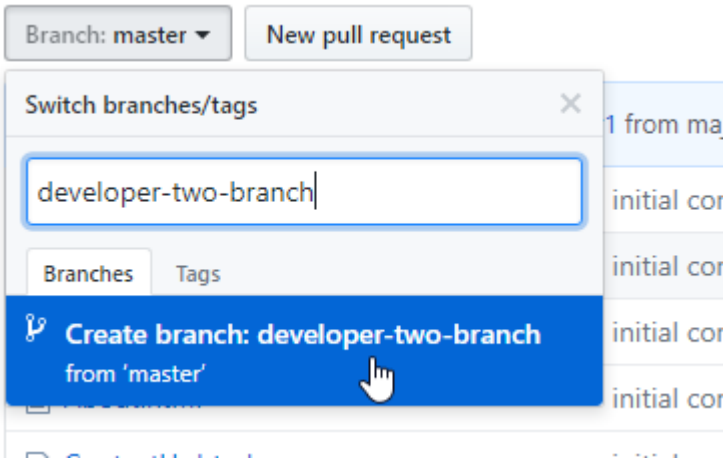
c) Log in to GitHub and browse to your repository

After getting to the correct repository, locate the "Branch:" dropdown (which should be pointing to master. Select this dropdown and enter a new branch name in the empty box. Also note, we could switch to our other branch(es) that are listed in the dropdown if we would like:



d) Enter the branch name into the dropdown

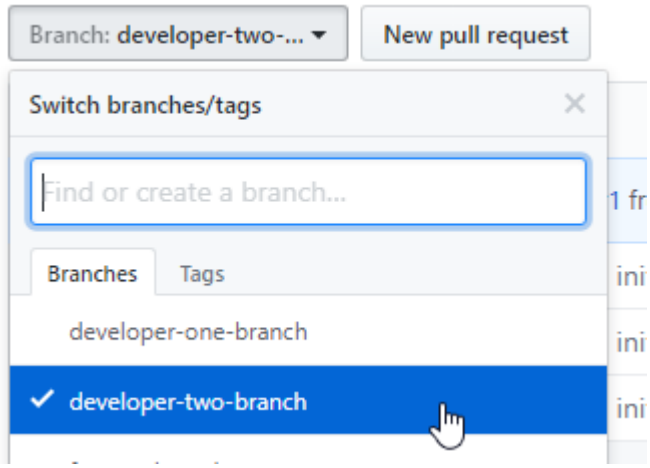
[developer-two-branch]



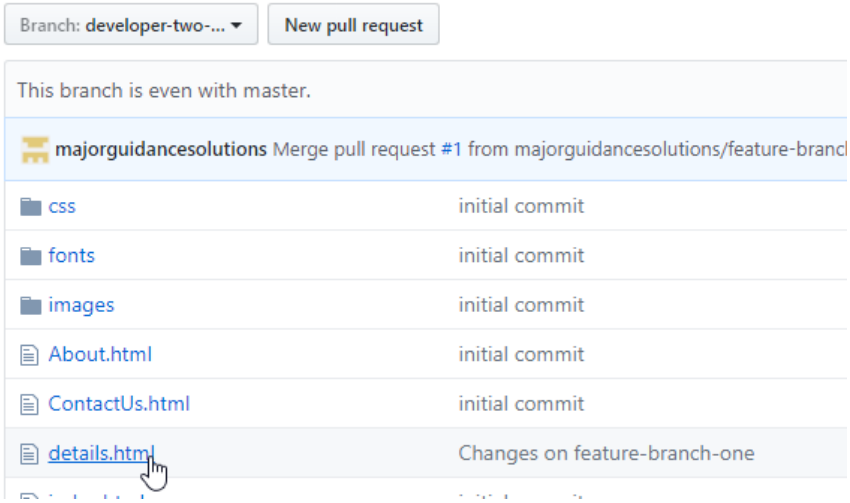
Then select the "Create Branch" button

e) Make sure you are on the developer two branch and make a change

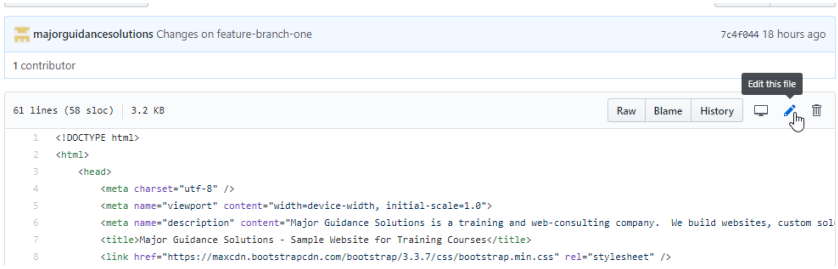
Make sure you have your dev 2 branch selected in the dropdown



On the main file listing, select the 'details.html' file:



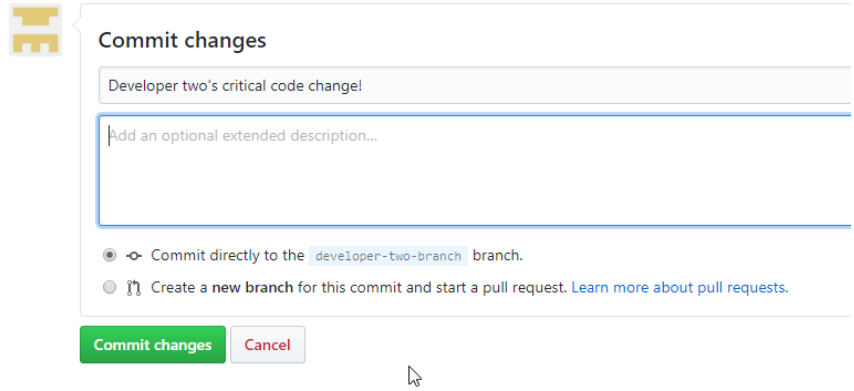
When this opens in GitHub, select the "Edit" pencil on the top-right corner of the file:



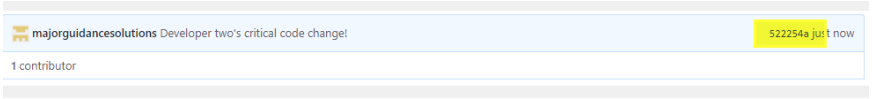
Change anything about the file in generally the same area we made the currently unmerged change for dev 1:

```
46
47          <h2> I'm making some changes! </h2>
48          <h3> Changes made on feature-branch-one for merge at GitHub</h3>
49
50          <h2> I'm developer two, creating a conflict with my critical code change!</h2>
51          <hr />
52          <footer>
53              <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidance Solutions</a></p>
54          </footer>
```

Add a commit message and create a commit directly on this branch:



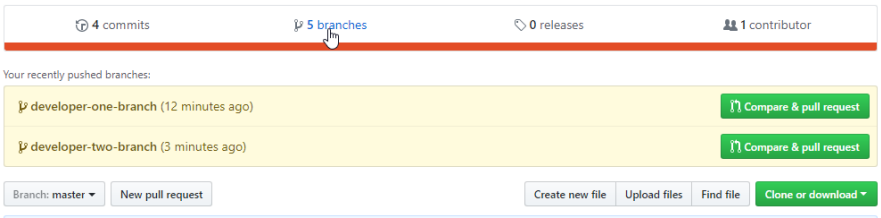
By pressing the "Commit Changes" button. Note the commit id:



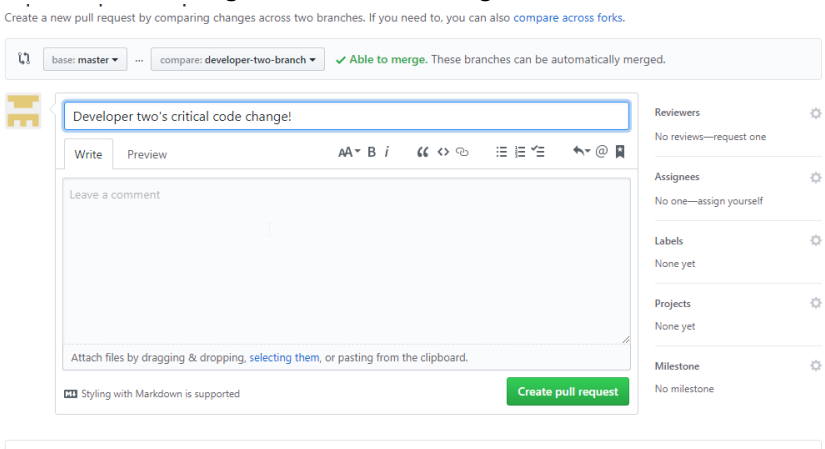
f) Create and merge a pull request

As we are pretending to be developer two, imagine that another developer had created a local feature branch and had made this change, pushed it up, and is now asking for a pull request. You know this will eventually conflict with your changes, but their changes are ready and there is nothing wrong with the code, so you are going to go ahead and merge theirs, then you'll resolve on your branch eventually.

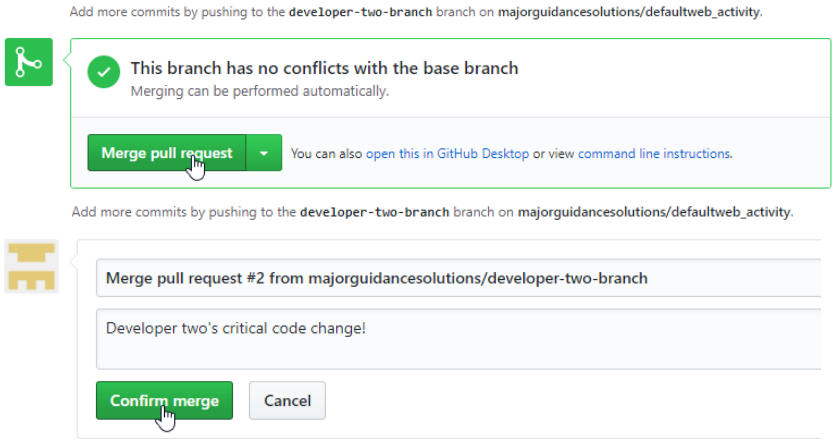
Browse to the branch under the branches tab and select "New Pull Request" Alternatively, just click on the developer-two-branch "Compare & Pull Request"



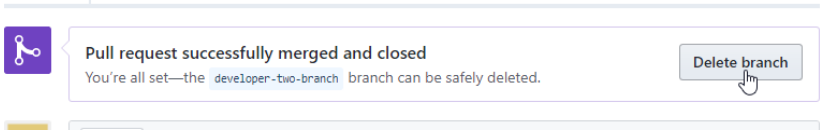
Note that their changes are able to be merged:



Which is indicated by the green "Able to merge" next to the checkmark. Create and Merge the pull request, pretending to be the code reviewer that allowed the request for dev 2's changes to be merged:



After the merge, go ahead and delete the branch:

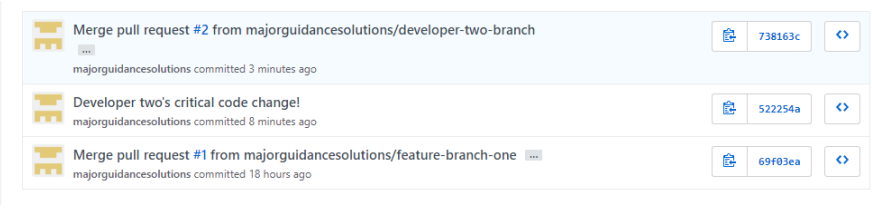


Note the merge commit ID:

Also note that GitHub has an 'auto-revert' button. If you needed to rollback, you could click Revert. You can also restore the branch if you want. We don't need to do either right now, but note your options.

The merge commit id for me was 738163c.

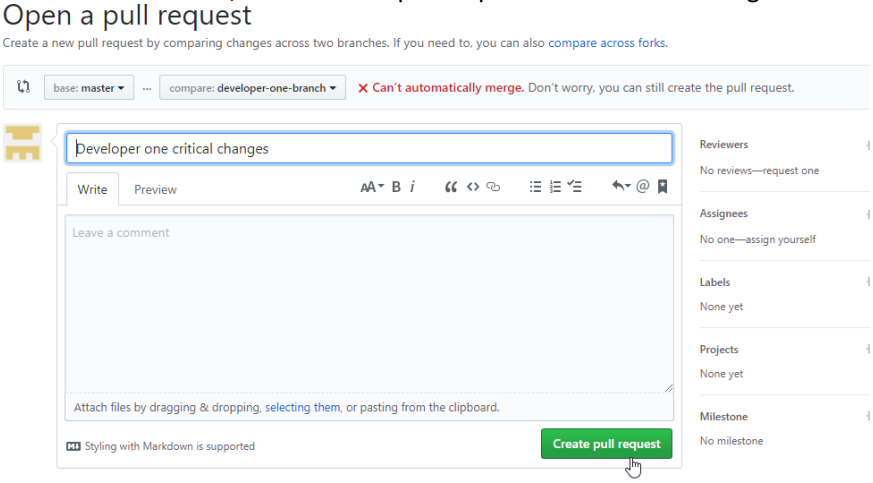
So here is my current history at GitHub (click on 'Commits' from the main screen to see yours):



Step 4: Create the pull request for developer one

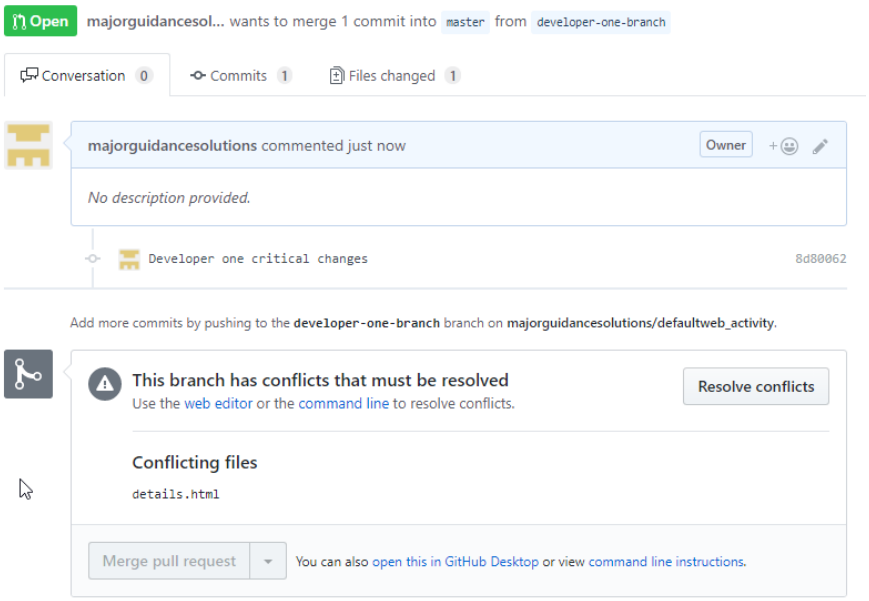
a) Create the Pull Request

While still at GitHub, create a new pull request for the dev 1 changes:



Note the message: "Can't automatically merge"
This is expected. We knew there would be a conflict. Good news: We can still create the PR. Even better news, for something this simple we could easily resolve right at GitHub (if you want to do that, you can, and then you could re-simulate another conflict and continue this again to resolve at Local first, which is the recommended way to do this to keep from making mistakes.

Go ahead and create the pull request, even with the conflict still not resolved:



We are going to resolve at local. This is the way I would recommend doing it, because you won't have any syntax help at GitHub. It would be very easy to create a "fix" at GitHub for this, but what if I mistype something? I may not know. Therefore, just get the changes locally and fix the conflict with our mergetool, and then we'll be easily able to merge.

Step 5: Bring the latest changes back to our local repo, merge and resolve the conflict, then push

c) Now that we have a couple of commits in the chain on master and a conflict to resolve, we first want to just get everything up to date on master locally:

[git checkout master]

[git fetch origin]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/majorguidancesolutions/defaultweb_activity
   69f03ea..738163c  master     -> origin/master
```

[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Updating 69f03ea..738163c
Fast-forward
 details.html | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
738163c (HEAD -> master, origin/master) Merge pull request #2 from majorguidance
solutions/developer-two-branch
522254a Developer two's critical code change!
69f03ea Merge pull request #1 from majorguidancesolutions/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

As we expected, we have the latest now for master, including dev two's merged changes

d) Stat the merge of the changes into our local feature branch

Our goal here is to make sure that we can put our own changes into the repository. What we need to do is get the changes that are in conflict and already at GitHub, and then we merge to our feature branch where we are working. In this way, we don't lose the changes the other developer did. We may need to modify their change slightly if it affects us, or we might be able to just manually merge our change with their changes. We will be keeping all of their changes as well as adding ours.

Switch to our dev branch:

[git checkout developer-one-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git checkout developer-one-branch
Switched to branch 'developer-one-branch'
Your branch is up-to-date with 'origin/developer-one-branch'.
```

Merge their changes from master into ours:

[git merge master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git merge master
Auto-merging details.html
CONFLICT (content): Merge conflict in details.html
Automatic merge failed; fix conflicts and then commit the result.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch|M
ERGING)
$
```

Note, we are now in conflict and the branch is appended with | MERGING on it. IF we didn't want to continue, we could just abort [git merge --abort]. We are going to continue, however.

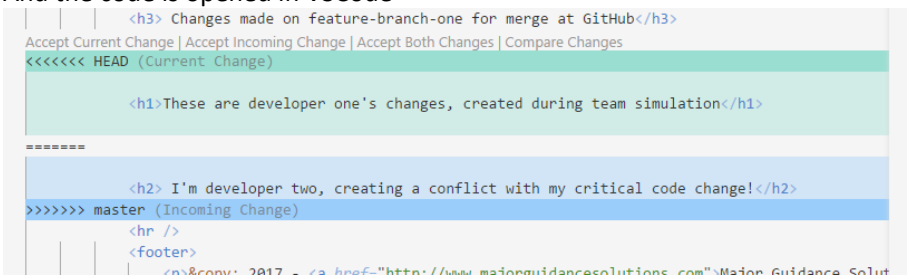
e) Resolve the conflict with our mergetool

NOTE: If you haven't setup a mergetool, you will be taken back to 1985 in the VIM merge editor. Therefore, make sure you have first setup your default mergetool – which is an activity earlier in the course]

[git mergetool]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch|M
ERGING)
$ git mergetool
Merging:
details.html

Normal merge conflict for 'details.html':
  {local}: modified file
  {remote}: modified file
```

And the code is opened in VSCode



We're going to click on the item that says "Accept Both Changes" which then gives us:

```
<h2> I'm making some changes! </h2>
<h3> Changes made on feature-branch-one for merge at GitHub</h3>

<h1>These are developer one's changes, created during team simulation</h1>


<h2> I'm developer two, creating a conflict with my critical code change!</h2>
<hr />
<footer>
    <p>&copy; 2017 - <a href="http://www.majorguidancesolutions.com">Major Guidanc
```

We could clean this up further if we would wish. It will be your job as the developer to review the conflict and make sure that your changes and the other changes play nice and functionality is still in tact after the merge resolution. They will rarely be this easy to resolve ☺.

Save the changes and close VSCode to continue the merge operation

Once you save and close you get taken back to the terminal:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch|M
ERGING)
$
```

Now you have two choices. Commit or Continue. If you want to commit, just type the command [git commit -m 'merge resolved']. I'm going to go the continue route here, and if you have multiple conflicts you would want to do this as well if you weren't already automatically taken to the next conflict

Continue the merge with --continue

[git merge --continue]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch|M
ERGING)
$ git merge --continue
```

Which brings up the editor to add a commit message, which is already set if I want to keep it:

```
COMMIT_EDITMSG
1   Merge branch 'master' into developer-one-branch
2
3   # Conflicts:
4   #   details.html
```

I'm going to keep that, save it, and close. This will then create the commit in the terminal:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch|M
ERGING)
$ git merge --continue
[developer-one-branch 42ed51b] Merge branch 'master' into developer-one-branch
```

Now look at our history to make sure we look good:

[git log --oneline]

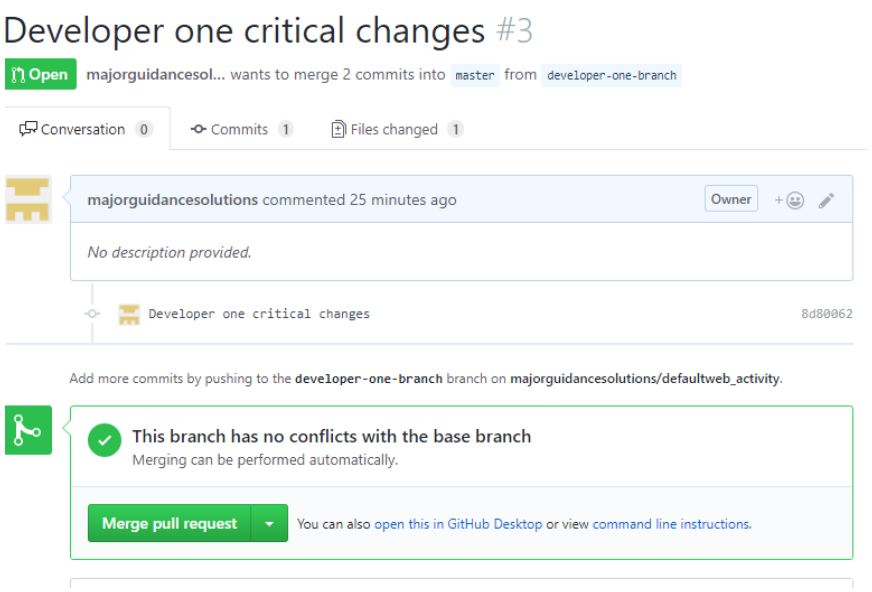
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git log --oneline
42ed51b (HEAD -> developer-one-branch) Merge branch 'master' into developer-one-
branch
738163c (origin/master, master) Merge pull request #2 from majorguidancesolution
s/developer-two-branch
522254a Developer two's critical code change!
8d80062 (origin/developer-one-branch) Developer one critical changes
69f03ea Merge pull request #1 from majorguidancesolutions/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

And that shows us being in place, including the commits from both developers in our history, so we know we are now safe to push and merge at GitHub:

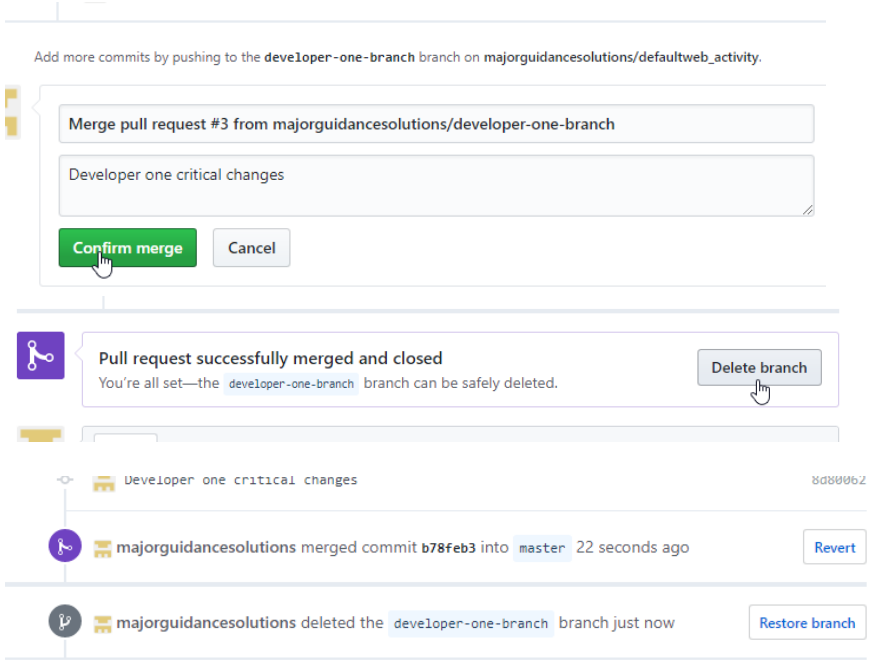
[git push origin developer-one-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git push origin developer-one-branch
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 437 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/defaultweb_activity.git
   8d80062..42ed51b  developer-one-branch -> developer-one-branch
```

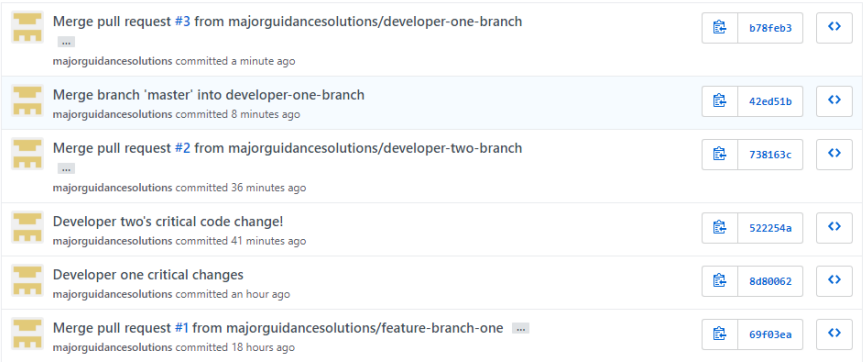
f) Now we can merge the pull request!



Go ahead and merge and confirm. This will get everything up to date. Then delete the branch at REMOTE.



And review the commits on the repository:



Does that lineup with what we have locally on master?
[git checkout master]
[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (developer-one-branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
738163c (HEAD -> master, origin/master) Merge pull request #2 from majorguidance
solutions/developer-two-branch
522254a Developer two's critical code change!
69f03ea Merge pull request #1 from majorguidancesolutions/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

NO! we are missing the final merge commit!

Step 6: Cleanup

b) Optional cleanup

We already see that we don't have the merge commit – so let's get that:
[git fetch origin --prune]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git fetch origin --prune
From https://github.com/majorguidancesolutions/defaultweb_activity
 - [deleted]         (none)     -> origin/developer-one-branch
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
   738163c..b78feb3  master     -> origin/master
```

Wait! What's prune? Oh yeah, that will delete any references [not the actual branch] on my local that no longer exist on remote. So we should no longer see 'origin/developer-*-branch' in our branch list after this
[git branch -a]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -a
  My-Feature-Branch
  developer-one-branch
  feature-branch-one
  feature-branch-two
* master
  remotes/origin/feature-branch-one
  remotes/origin/feature-branch-two
  remotes/origin/master
```

Hey, it's still there locally! [yes, we have to delete that locally to make it go away. This is for our protection]

[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/defaultweb_activity
 * branch            master     -> FETCH_HEAD
Updating 738163c..b78feb3
Fast-forward
 details.html | 3 +++
 1 file changed, 3 insertions(+)
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git log --oneline
b78feb3 (HEAD -> master, origin/master) Merge pull request #3 from majorguidance
solutions/developer-one-branch
42ed51b (developer-one-branch) Merge branch 'master' into developer-one-branch
738163c Merge pull request #2 from majorguidancesolutions/developer-two-branch
522254a Developer two's critical code change!
8d80062 Developer one critical changes
69f03ea Merge pull request #1 from majorguidancesolutions/feature-branch-one
7c4f044 (origin/feature-branch-one, feature-branch-one) Changes on feature-branc
h-one
7b6a932 (origin/feature-branch-two, feature-branch-two, My-Feature-Branch) chang
es to details
b7edba6 initial commit
```

And we are up-to-date!

Let's delete the local copy of the dev 1 branch, and any other branches we no longer want around:
[git branch -d <branch-to-delete>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb_Activity (master)
$ git branch -d developer-one-branch
Deleted branch developer-one-branch (was 42ed51b).
```

This concludes our team merge conflict resolution activity.

Closing Thoughts

This was a fun activity, and this is something that we want to "master"- ha! To be certain, working with a team is a critical part of working with GIT and surely you will run into many conflicts.

What we saw here was the fact that one developer made a change and got it into master while we were working on a conflicting change. While we could have fixed this at GitHub, it is much safer and more realistic to fix on our local branch and then be easily able to merge at GitHub because we'll have already solved all of the conflicts.

I would recommend practicing this activity a few times to get the flow and understanding down. Once you're confident that you have this down, you should be in fairly good control of the ability to work with GIT as a team at a very functional level.

Notes

Section 09 – Advanced GIT operations: Interacting with the repository outside the bounds of simple workflows

One of the greatest discoveries a man makes, one of his greatest surprises, is to find he can do what he was afraid he couldn't do.

Learning:

In this section we'll tackle some of the more advanced commands that we'll need when working with GIT. We're already good with the day-to-day things that can happen in GIT, but what do we do when something goes wrong, or we need to do something beyond the norm? GIT gives us all kinds of powerful tools. Knowing how to use these tools can make you the champion of your team.

Goals:

- Be familiar with many of the more advanced GIT commands
- Conquer our fear about history rewriting
- Understand how to reset and/or revert our code
- Finding lost commits.
- Cleaning up our working directory
- Get commits from one branch into another
- Saving changes without committing

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we will be confident with many of the more advanced git commands. This will give us the ability to know that we do not need to fear making complex changes on our repository. With the advanced commands, we can overcome most of the problems that would arise during the development cycle when working with GIT.

Activities:

- Changing the commit message
- Adding a file/change to the previous commit
- Examining the reflog
- Squash and merge
- Aliasing our commands
- Always prune on fetch
- Soft Reset
- Hard Reset

Notes

- Reverting one or more commits
- Rebasing (3 activities)
- Cherry Picking
- Stashing Changes

Videos:

- Amending Commits parts 1 & 2
- Git Reflog parts 1 & 2
- Squash and merge at GitHub (3 videos)
- Using Aliases
- Soft Reset
- Hard Reset
- Reverting
- Rebasing
- Cherry-picking
- Stashing

GIT: From Beginner To Fearless

GIT Commit with Amend Activity:
A simple exercise using git commit with amend

Brian Gorman, Author/Instructor/Trainer

Introduction

Git makes it possible for us to “change” a commit. Sometimes we want to change the commit message. Other times we might want to actually make a small change, perhaps to fix a bug or remove some commented code that we don’t want in the repo.

In the case where we want to fix something and rewrite the commit history, we can do so with the git amend command.

Let’s gets started!

Step 1: Amend a commit to change the commit message.

s) Make sure you have any repository up to date and are working on any branch.

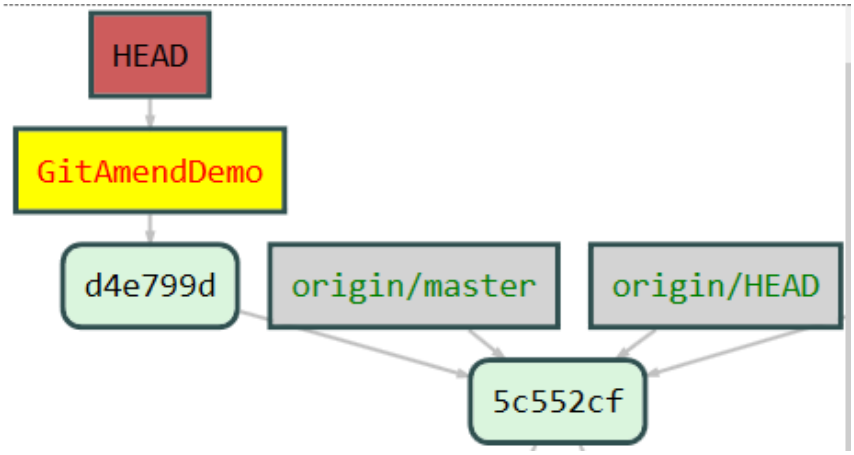
[clone repo]
Or
[git checkout master]
[git fetch origin]
[git pull origin master]
[git checkout -b GitAmendDemo]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFold
$ git checkout -b GitAmendDemo
Switched to a new branch 'GitAmendDemo'
```

Make a small change, add and commit.
[code info.txt]
[git status –s]
[git commit -am “This is my commit messag”]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo
$ git status -s
 M info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo
$ git commit -am "This is my commit messag"
[GitAmendDemo d4e799d] This is my commit messag
 1 file changed, 2 insertions(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo
```

Notes

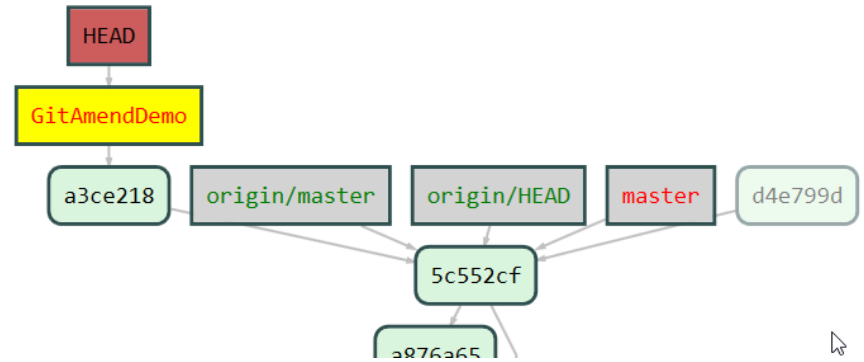


Note the commit id [d4e799d]

t) The message had a typo, let's fix it.

[git commit --amend -m "This is an important change to the code"]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git commit --amend -m "This is an important change to the code"
[GitAmendDemo a3ce218] This is an important change to the code
 Date: Sat Sep 23 21:45:21 2017 -0500
 1 file changed, 2 insertions(+)
```



Note that the commit id is different, so we would need to be careful on a public branch if we are using an amend. We can also see the old commit is still showing as unreachable. Therefore we should do a quick cleanup.

Step 2: Clean up the unreachable commits.

a) Use reflog to expire unreachable and then cleanup with the garbage collector

To clean up the commits we just need to make sure we have the reflog set to expire our commits and then run the garbage collector. I have these commands aliased, but in case you don't and you want to run these [or want the commands for later reference], here they are:

[**git reflog expire –expire-unreachable=now –all**]

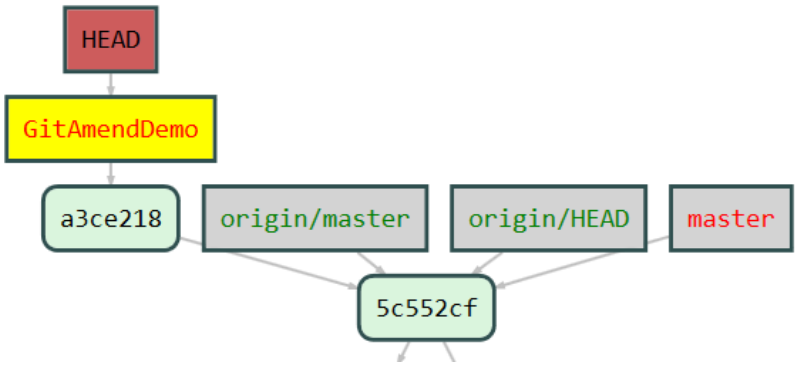
And

[**git gc –prune=now**]

And here are my aliases in my global config [check out the aliasing activity for more info about aliasing]:

```
alias.expireunreachablenow=reflog expire --expire-unreachable=now --all
alias.gcunreachablenow=gc --prune=now
fetch.prune true
```

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git expireunreachablenow

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git gcunreachablenow
Counting objects: 25, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (25/25), done.
Total 25 (delta 13), reused 20 (delta 11)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$
```

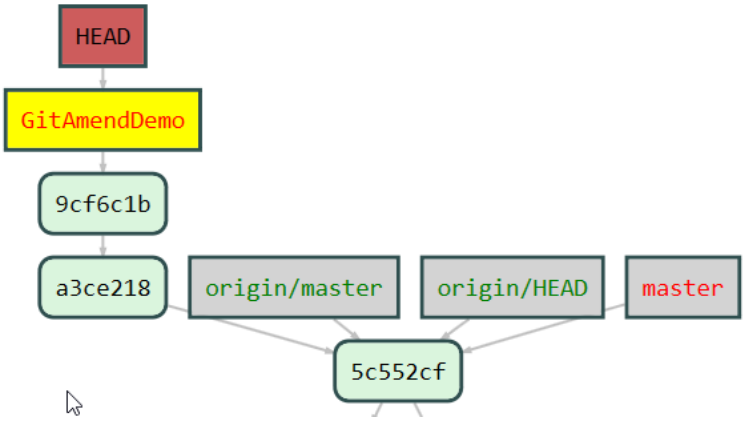


Step 3: Amend with some actual changes.

a) Create some changes to info.txt and commit

[code info.txt]
[git status –s]
[git commit -am “Some changes for the next release”]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (Git
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (Git
$ git status -s
 M info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (Git
$ git commit -am "Some changes for the next release"
[GitAmendDemo 9cf6c1b] Some changes for the next release
 1 file changed, 2 insertions(+), 1 deletion(-)
```



Make note of the commit id [9cf6c1b]

b) Add a new file to the repo

[touch readme.txt]

[code readme.txt]

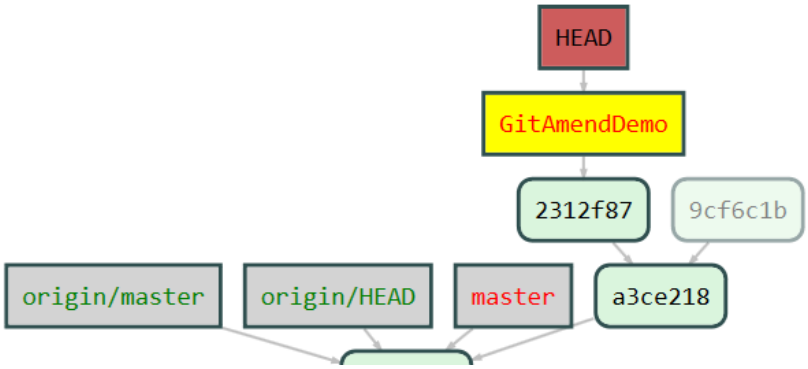
[git status -s]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/De
$ touch readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/De
$ code readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/De
$ git status -s
?? readme.txt
```

c) Commit with amend to put the new file into the same commit as the changes for info.txt

[git commit --amend -m “Changed info.txt and added readme.txt”

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git commit --amend -m "Changed info.txt and added readme.txt"
[GitAmendDemo 2312f87] Changed info.txt and added readme.txt
 Date: Sat Sep 23 22:33:01 2017 -0500
 1 file changed, 2 insertions(+), 1 deletion(-)
```

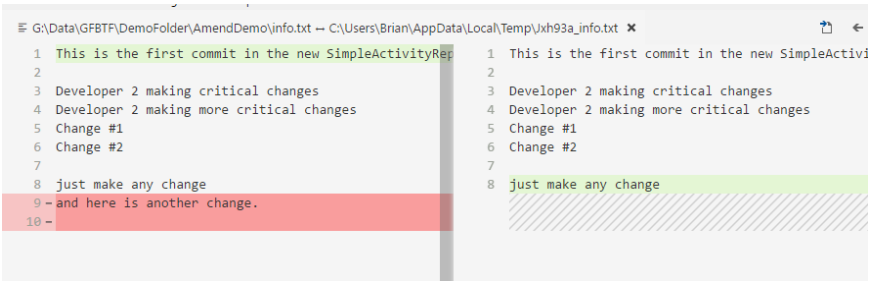


[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git log --oneline
2312f87 (HEAD -> GitAmendDemo) Changed info.txt and added readme.txt
a3ce218 This is an important change to the code
5c552cf (origin/master, origin/HEAD, master) Merge pull request #2 from m
dancesolutions/rebasing-demo-1
a876a65 more changes on my local branch
```

[git difftool 2312f87 a3c3218]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFol
$ git difftool 2312f87 a3ce218
```



Using the difftool I noticed I forgot to add the new file!!!
[git status –s]

```
Brian@SENTINEL MINGW64 /g/Dat
$ git status -s
?? readme.txt
```

d) First add, then commit with amend

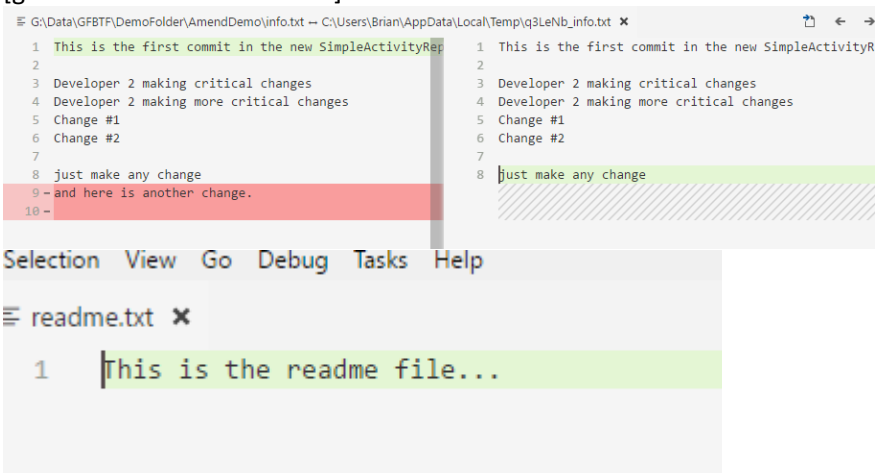
[git add .]
[git commit --amend -m “Changed info.txt and added readme.txt”]
[git status –s]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (G
$ git add .

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (G
$ git commit --amend -m "Changed info.txt and added readme.t
[GitAmendDemo 0cac76b] Changed info.txt and added readme.txt
 Date: Sat Sep 23 22:33:01 2017 -0500
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (G
$ git status -s
```

[git log –oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/A
$ git log --oneline
0cac76b (HEAD -> GitAmendDemo) Changed info.txt a
a3ce218 This is an important change to the code
5c552cf (origin/master, origin/HEAD, master) Merg
dancesolutions/rebasing-demo-1
```

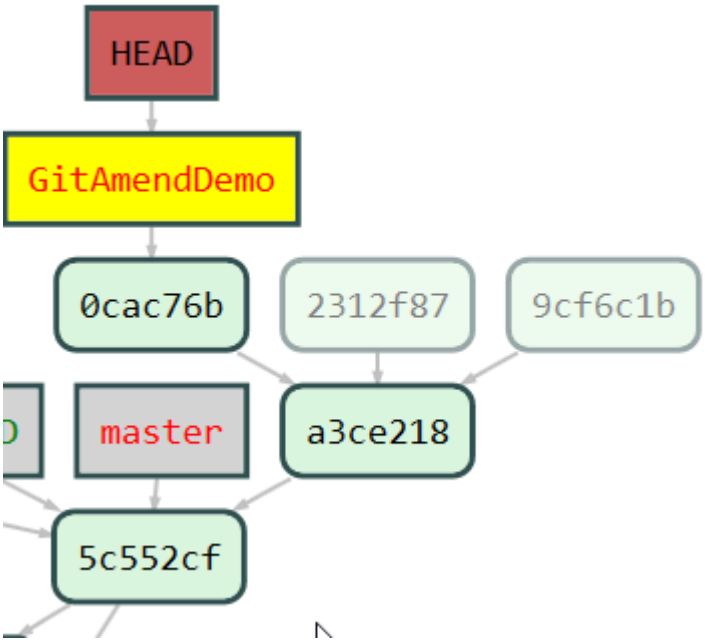
[git difftool 0cac76b a3ce218]



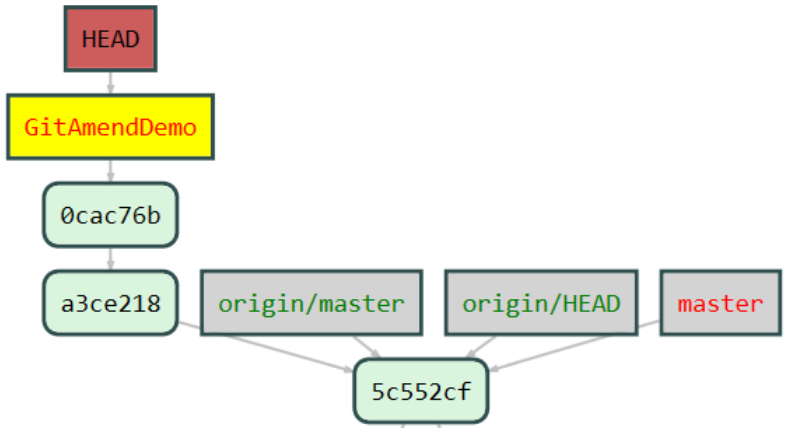
Now we are on track.

Step 3: Cleanup the local unreachables, push to GitHub and merge changes into master.

a) Repeat Step 2 for cleanup.



[commands for cleanup as in step 2]

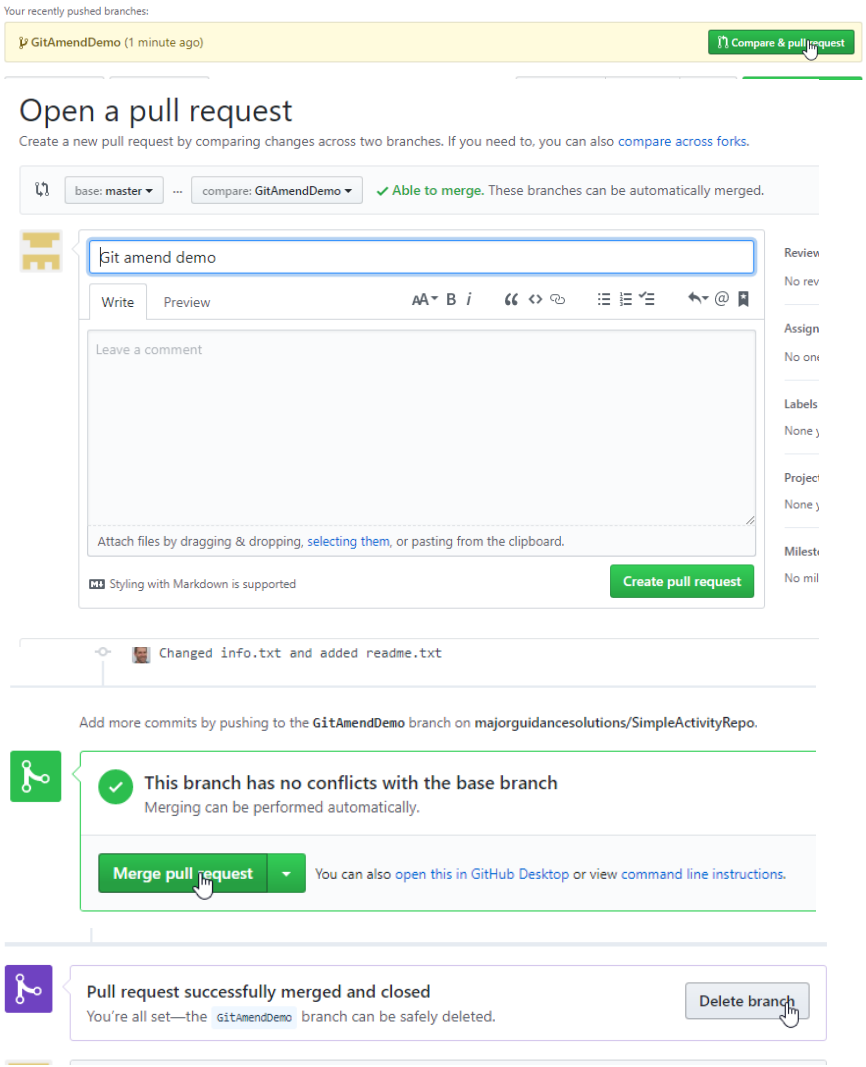


b) Push to GitHub

[git push -u origin <branchname>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (GitAmendDemo)
$ git push -u origin GitAmendDemo
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 662 bytes | 0 bytes/s, done.
Total 7 (delta 4), reused 5 (delta 2)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
 * [new branch]      GitAmendDemo -> GitAmendDemo
Branch GitAmendDemo set up to track remote branch GitAmendDemo from origin.
```

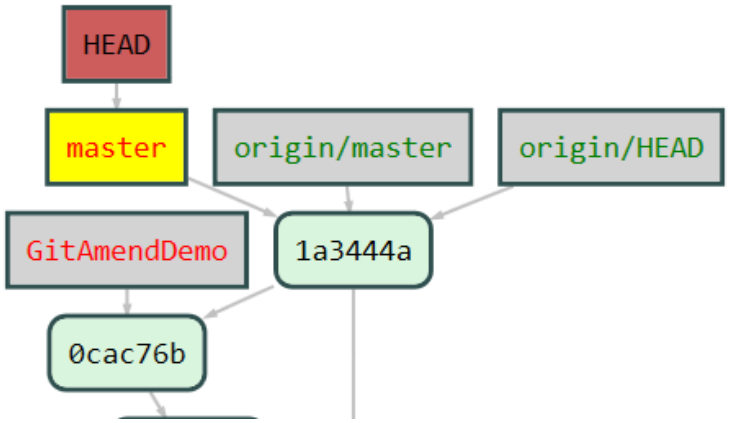
Create a Pull Request and Merge at GitHub



c) Get the latest from github into the local repo

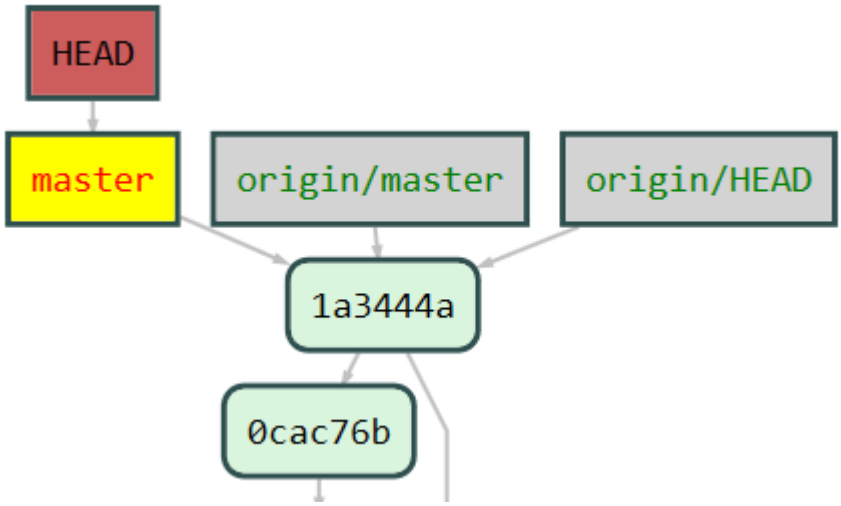
[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (
$ git fetch origin
From https://github.com/majorguidancesolutions/SimpleActivi
 - [deleted]         (none)     -> origin/GitAmendDemo
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 1 (delta 0), pack-reused
Unpacking objects: 100% (1/1), done.
   5c552cf..1a3444a  master     -> origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/AmendDemo (
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivi
 * branch            master     -> FETCH_HEAD
Updating 5c552cf..1a3444a
Fast-forward
 info.txt   | 3 +++
 readme.txt | 1 +
 2 files changed, 4 insertions(+)
 create mode 100644 readme.txt
```



[git branch –d GitAmendDemo]
[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Ame
$ git branch -d GitAmendDemo
Deleted branch GitAmendDemo (was 0cac76b).

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Ame
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```



This concludes our git commit with amend activity.

Closing Thoughts

In this activity we saw a couple of scenarios where we were able to change the commit history and amend changes into the most recent commit.

The first amend was a simple change to the commit message. The second change was a full commit with added files amended into the commit.

With the amend, we are able to change the most recent commit, but again we would want to keep the history intact if this was a public branch. Since this is a private branch, we were able to do what we wanted.

The other nice thing about the amend is that all our changes only made one combined commit into the repository.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Reflog Activity:
A simple exercise using git reflog

Brian Gorman, Author/Instructor/Trainer

Introduction

In life, things go wrong. In GIT, if you do something that somehow messed up your repo (which is not that easy to do), along comes REFLOG to save the day.

If you've seen any of the videos for the course where I use GitViz, or have worked through other activities where a branch was deleted, commits were reset, amended, or otherwise became 'unreachable.' When we look at regular log in GIT, an unreachable commit is not listed. However, reflog shows us everything that we have in cache for our current repository. And, to answer your question -> Yes, GitHub has a reflog as well. I believe that using the GitHub reflog would require using the GitHub API, and that is outside the scope of this course.

For this activity, we're going to take a look at the reflog and see how we can glean information from it, as well as how that is useful to us when things are not quite going the way we'd have liked them to.

Let's gets started!

Step 1: Taking a look at the reflog

u) In order to do this activity, you should be on an active local repo that has a chain of commits.

If you don't have an active repo with a few commits, then take a moment right now to create a local repo that has 5-10 commits. Make sure to also do a few things like switch your branch a couple of times. If you want to get even more ambitious, do some revert and/or reset operations.

v) Reviewing the reflog

To take our first look at the reflog, simply enter the command:
[git reflog] //note: Your reflog will undoubtedly be different but similar to this:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog
f2746c6 (HEAD -> master) HEAD@{0}: commit: Added the files from the release
er
f737642 HEAD@{1}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{2}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{3}: commit: Added the important resource files
43239f4 HEAD@{4}: commit: changes to tracked file in ignored folder are stil
acked
8134e71 HEAD@{5}: commit: added the local gitIgnore file
4f86487 HEAD@{6}: commit: added info.txt during gitIgnore Activity
c8672c8 HEAD@{7}: reset: moving to head
c8672c8 HEAD@{8}: commit: Added the h4 tag change
cb5204a HEAD@{9}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{10}: commit: Added an h2 tag to the details page
874d595 HEAD@{11}: reset: moving to head
874d595 HEAD@{12}: reset: moving to head
874d595 HEAD@{13}: commit: Added the rest of the files
f69f692 HEAD@{14}: commit (initial): Added the About.html file
```

Here, I have some 15 objects in my reflog, and these are mostly commits and resets. Had I switched branches, that would show here as well.

Note that each commit has the commit message, which can be useful. Additionally, the commit SHA1 that I was on is listed on the left. For example, I added the rest of the files to commit 874d595, then did stuff and reset back to it two more times. Pretty cool.

Note that each entry has "HEAD@{n}". This means we can start the list from any place (for example if you had 100 you could start at 50). Something similar to this:
[git reflog HEAD@{9}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{9}
cb5204a HEAD@{9}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{10}: commit: Added an h2 tag to the details page
874d595 HEAD@{11}: reset: moving to head
874d595 HEAD@{12}: reset: moving to head
874d595 HEAD@{13}: commit: Added the rest of the files
f69f692 HEAD@{14}: commit (initial): Added the About.html file
```

Notes

w) Using time entries to review the reflog

The reflog is powerful in ways that we can check the state of the repo at specific commits as well as specific times. For example, suppose you want to see the reflog for some time periods. You know you had a branch 2 days ago:
[git reflog HEAD@{2.days.ago}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{2.days.ago}
f2746c6 (HEAD -> master) HEAD@{Wed Jul 5 16:29:13 2017 -0500}: commit: Added the
 files from the release folder
f737642 HEAD@{Wed Jul 5 15:48:33 2017 -0500}: commit: added pattern for gitIgnor
e on [Bb]in/**
524f0f7 HEAD@{Wed Jul 5 15:41:36 2017 -0500}: commit: Continuing with the .gitIg
nore activity
0a2a8de HEAD@{Wed Jul 5 15:22:56 2017 -0500}: commit: Added the important resour
ce files
43239f4 HEAD@{Wed Jul 5 14:38:31 2017 -0500}: commit: changes to tracked file in
 ignored folder are still tracked
8134e71 HEAD@{Wed Jul 5 14:31:11 2017 -0500}: commit: added the local gitIgnore
file
```

Here you can see that this repo is actually quite a bit older. If your repo is newer, then it becomes more useful. Here are some of the different time constraints we can use:
{1.minute.ago}…{2.minutes.ago}…{253.minutes.ago}…{<n>.minutes.ago}
{1.hour.ago}…{2.hours.ago}…{n.hours.ago}
{1.day.ago}…{2.days.ago}…
{yesterday}
{1.week.ago}…{2.weeks.ago}…{n.weeks.ago}
{n.month(s).ago}
{n.year(s).ago}
And specific date {yyyy-mm-dd.hh:mm:ss}
[git reflog HEAD@{2017-07-05.11:51:38}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{2017-07-05.11:51:38}
c8672c8 HEAD@{Wed Jul 5 11:51:38 2017 -0500}: commit: Added the h4 tag change
cb5204a HEAD@{Wed Jul 5 11:46:24 2017 -0500}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{Wed Jul 5 11:18:31 2017 -0500}: commit: Added an h2 tag to the details page
874d595 HEAD@{Wed Jul 5 11:14:29 2017 -0500}: reset: moving to head
874d595 HEAD@{Wed Jul 5 11:03:57 2017 -0500}: reset: moving to head
874d595 HEAD@{Wed Jul 5 10:07:51 2017 -0500}: commit: Added the rest of the files
f69f692 HEAD@{Wed Jul 5 09:56:05 2017 -0500}: commit (initial): Added the About.html file
```

Note, if you try a date prior to the repo, GIT will yell at you and tell you that there are no such entries
[git reflog HEAD@{4.years.ago}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{4.years.ago}
warning: Log for 'HEAD' only goes back to Wed, 5 Jul 2017 09:56:05 -0500.
```

Step 2: Show differences between two reflog entries

a) Find a couple of entries in your reflog to compare by index

If you don't have a lot, then you will want to create some.

[git diff HEAD@{9} HEAD@{3}]
[git difftool HEAD@{9} HEAD@{3}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git difftool HEAD@{9} HEAD@{3}
```

//shows a bunch of changes so we can see the differences between the two commits

b) Compare the differences in a repo over a timespan

[git diff HEAD@{1.day.6.hours.ago} HEAD@{14}]
[git difftool HEAD@{1.day.6.hours.ago} HEAD@{14}]
//more differences
[git diff HEAD@{14.days.22.hours.ago} HEAD@{1.minute.ago}]
[git difftool HEAD@{84.days.ago} HEAD@{now}]
//etc. You can keep playing with this as you would like.

c) Checkout a reflog

[git reflog]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog
f2746c6 (HEAD -> master) HEAD@{0}: checkout: moving from c8672c878f2569139976c19eb58a5e8d05487960 t
r
c8672c8 HEAD@{1}: checkout: moving from master to HEAD@{7}
f2746c6 (HEAD -> master) HEAD@{2}: commit: Added the files from the release folder
f737642 HEAD@{3}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{4}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{5}: commit: Added the important resource files
43239f4 HEAD@{6}: commit: changes to tracked file in ignored folder are still tracked
8134e71 HEAD@{7}: commit: added the local gitIgnore file
4f86487 HEAD@{8}: commit: added info.txt during gitIgnore Activity
c8672c8 HEAD@{9}: reset: moving to head
c8672c8 HEAD@{10}: commit: Added the h4 tag change
cb5204a HEAD@{11}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{12}: commit: Added an h2 tag to the details page
874d595 HEAD@{13}: reset: moving to head
874d595 HEAD@{14}: reset: moving to head
874d595 HEAD@{15}: commit: Added the rest of the files
f69f692 HEAD@{16}: commit (initial): Added the About.html file
```

Assume for some reason you need to go back to HEAD@{10}
[git checkout HEAD@{10}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git checkout HEAD@{10}
Note: checking out 'HEAD@{10}'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at c8672c8... Added the h4 tag change

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb ((c8672c8...))
$
```

If we wanted to do anything, we could checkout a branch from here to create a new commit, etc.

[git reflog]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb ((c8672c8...))
$ git reflog
c8672c8 (HEAD) HEAD@{0}: checkout: moving from master to HEAD@{10}
f2746c6 (master) HEAD@{1}: checkout: moving from c8672c878f2569139976c19eb58a5e8d05487960 to master
c8672c8 (HEAD) HEAD@{2}: checkout: moving from master to HEAD@{7}
f2746c6 (master) HEAD@{3}: commit: Added the files from the release folder
f737642 HEAD@{4}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{5}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{6}: commit: Added the important resource files
43239f4 HEAD@{7}: commit: changes to tracked file in ignored folder are still tracked
```

Note we can see the movement.
Go back to master
[git checkout master]

Step 3: Set expire and use garbage collection to cleanup unreachable commits.

There are many times when we rewrite history, drop branches, or perform other various operations in GIT which end up "orphaning" a commit. Essentially, the commit is in a state that is referred to as "unreachable." Keeping these commits around is not always a bad idea (as long as they are around we can checkout the commit and work with it). However, there are other times when you just want to clean up or perhaps the unreachable commits are getting very stale. In these cases we want to cleanup the unreachables.

a) Cleanup anything older than 14 days

[git reflog expire --expire-unreachable=14.days.ago –all]
[git gc --prune=14.days.ago]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog expire --expire-unreachable=14.days.ago --all

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git gc --prune=14.days.ago
Counting objects: 62, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (50/50), done.
Writing objects: 100% (62/62), done.
Total 62 (delta 22), reused 0 (delta 0)
```

b) Clean up all the loose objects and expired/unreachable commits as of now

[git reflog expire --expire-unreachable=now --all]
[git gc –prune=now]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog expire --expire-unreachable=now --all

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git gc --prune=now
Counting objects: 62, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (28/28), done.
Writing objects: 100% (62/62), done.
Total 62 (delta 22), reused 62 (delta 22)
```

This concludes our git reflog activity.

Closing Thoughts

In this activity, we learned about looking into the reflog in order to see the history of our repo as it has been interacted with at the local level. The reflog is a powerful tool when you need to find the general commits around a timeframe or within a few commits.

Once we pull up the reflog, we can easily start comparing the repository on reflog indexes as well as via timespan queries.

We also saw how to potentially checkout at a reflog entry, and then know we could checkout a branch based on the state of the repo at a particular moment as shown in the reflog if we wanted to make further changes from that point in history.

Finally, we saw how we can use the reflog to set unreachable objects to expired and then run the garbage collector to clean up the expired unreachable objects.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

Squash And Merge Activity:
Squashing commits during merge at GitHub

Brian Gorman, Author/Instructor/Trainer

Introduction

Many times we have a pull request that contains a feature that was developed in a local repository. The feature has been completed and through all of the rigor required to be implemented. During the development of the feature, multiple commits were recorded and the history at master would reflect these commits if the pull request is merged with a regular merge.

Depending on the rules of your repository, the size of the commit chain, and a number of personal factors, one option that can be done is to merge the pull request using a "Squash and Merge" operation. If this option is selected, the multiple commit history will be compacted into one commit at merge, with a regular merge message and details that automatically contain the commit messages.

A word about squash and merge, however, before everyone jumps on the 'this is incredibly awesome' bandwagon. First of all, if a squash and merge is completed, then it is very critical that the feature branch is deleted. Failing to do this will result in a commit history mismatch. Depending on the order you've worked through some of the activities, you might have heard me talk about never changing history on a publicly available branch. This is the same thing in reverse, with the caveat that it is entirely possible to continue working on the feature branch at local, and the problem will mostly surface during merge when it looks like many commits need to be merged even though they should already be in master.

In this activity, we'll walk through doing the pull request with a squash and merge, and see what it looks like when we don't delete the branch, and then do it again while also deleting the branch.

Let's gets started!

GFBTF: Git Squash And Merge Activity

Step 1: Make sure you have a working repository that is up to date.

x) Start with any repo, make sure you have the latest in master, and create a feature branch.

First clone the repo if it doesn't exist:

[git clone <link> <folder>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/SimpleActivityRepo.git Git
SquashAndMergeActivity
Cloning into 'GitSquashAndMergeActivity'...
remote: Counting objects: 30, done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 30 (delta 15), reused 23 (delta 8), pack-reused 0
Unpacking objects: 100% (30/30), done.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
```

If you didn't clone, make sure master is up to date

[**git checkout master**]

[**git fetch origin**]

[**git pull origin master**]

[git checkout –b SquashAndMergeFeature]

```
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git fetch origin
git
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git checkout -b SquashAndMergeFeature
Switched to a new branch 'SquashAndMergeFeature'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Squa
hAndMergeFeature)
```

Step 2: Make four commits, push, squash and merge.

a) For this activity, we need to do 3-4 commits on our branch.

[code info.txt] //leave it open after saving

[git commit -am "Squash and Merge commit #1"]

[make another change in info.txt]

[git commit -am "Squash and Merge commit #2"]

[make another change in info.txt]

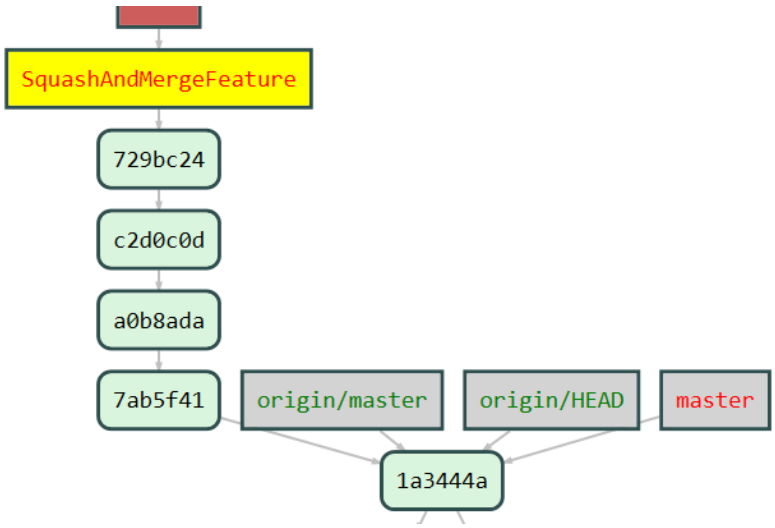
[git commit -am "Squash and Merge commit #3"]

[make another change in info.txt]

Notes

[git commit -am “Squash and Merge commit #4”]

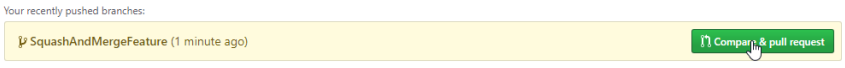
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (S
hAndMergeFeature)
$ git commit -am "Squash And Merge Commit#1"
[SquashAndMergeFeature 7ab5f41] Squash And Merge Commit#1
 1 file changed, 2 insertions(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (S
hAndMergeFeature)
$ git commit -am "Squash And Merge Commit#2"
[SquashAndMergeFeature a0b8ada] Squash And Merge Commit#2
 1 file changed, 1 insertion(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (S
hAndMergeFeature)
$ git commit -am "Squash And Merge Commit#3"
[SquashAndMergeFeature c2d0c0d] Squash And Merge Commit#3
 1 file changed, 2 insertions(+), 1 deletion(-)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (S
hAndMergeFeature)
$ git commit -am "Squash And Merge Commit#4"
[SquashAndMergeFeature 729bc24] Squash And Merge Commit#4
 1 file changed, 2 insertions(+), 1 deletion(-)
```

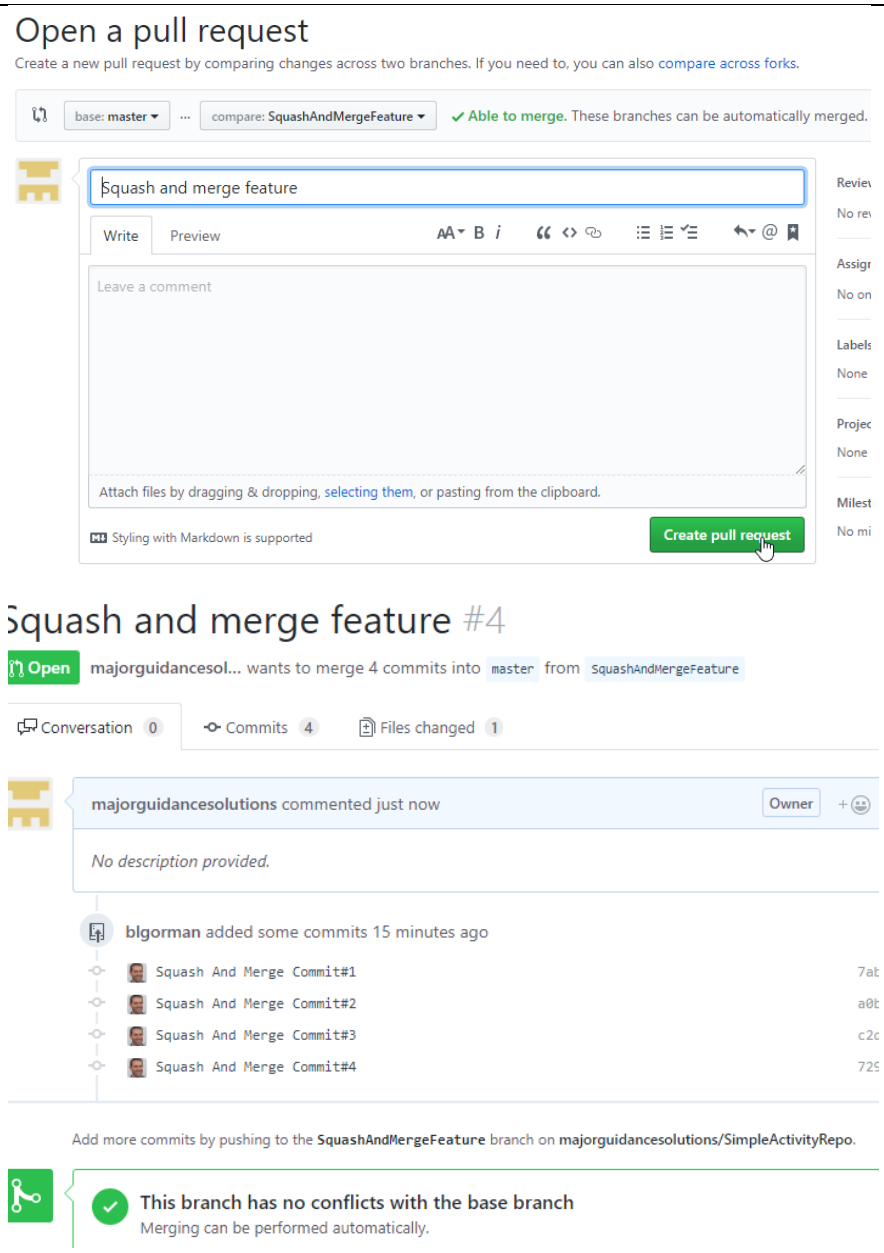


b) Push to GitHub, Create a Pull Request

[git push -u origin SquashAndMergeFeature]

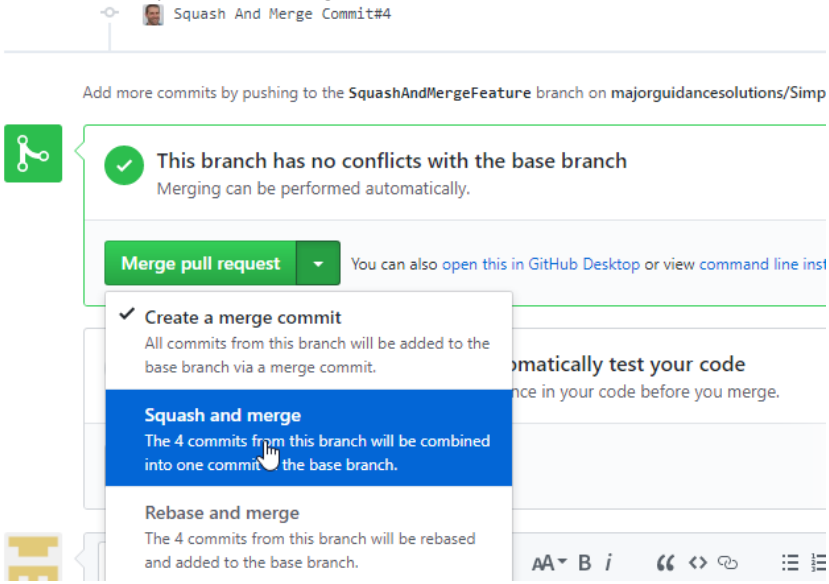
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Squas
hAndMergeFeature)
$ git push -u origin SquashAndMergeFeature
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 1.06 KiB | 0 bytes/s, done.
Total 12 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
 * [new branch]      SquashAndMergeFeature -> SquashAndMergeFeature
Branch SquashAndMergeFeature set up to track remote branch SquashAndMergeFeature
 from origin.
```



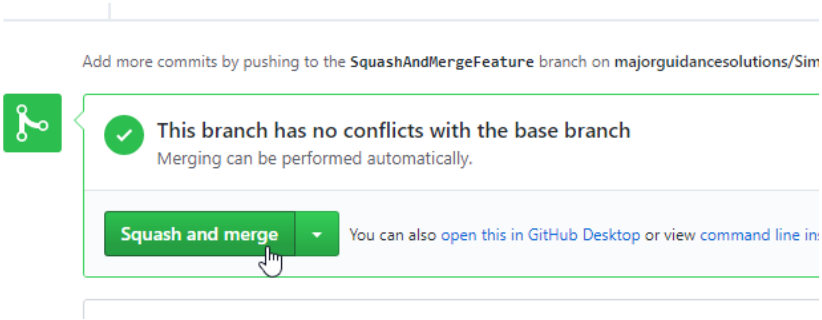

Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.
base: master
compare: SquashAndMergeFeature
Able to merge. These branches can be automatically merged.
Squash and merge feature
Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting them, or pasting from the clipboard.
Styling with Markdown is supported
Create pull request
Squash and merge feature #4
Open
majorguidancesol... wants to merge 4 commits into master from SquashAndMergeFeature
Conversation 0
Commits 4
Files changed 1
majorguidancesolutions commented just now
Owner
No description provided.
blgorman added some commits 15 minutes ago
Squash And Merge Commit#1
Squash And Merge Commit#2
Squash And Merge Commit#3
Squash And Merge Commit#4
Add more commits by pushing to the SquashAndMergeFeature branch on majorguidancesolutions/SimpleActivityRepo.
This branch has no conflicts with the base branch
Merging can be performed automatically.

c) Squash and merge the request – DO NOT delete branch

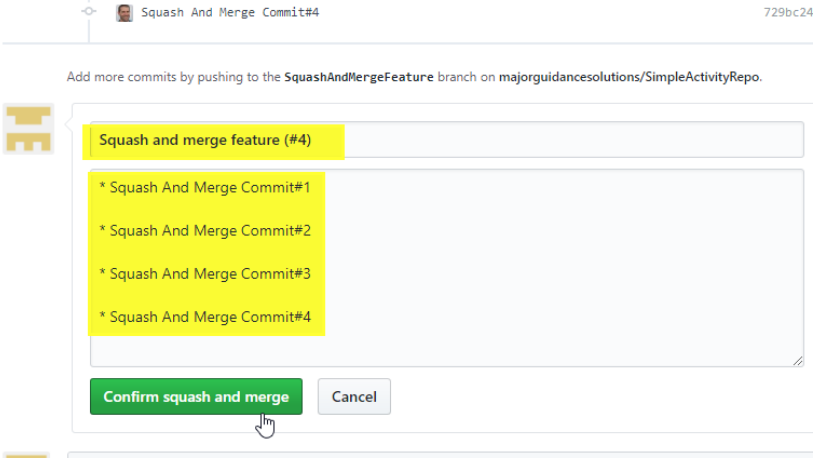
First, select “Squash And Merge”



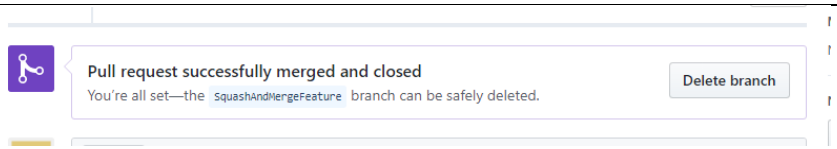
Next, hit the ‘Squash and Merge’ button



Then note the commit messages are put into the details. You can change the commit message if you would like



Confirm the squash and merge [reminder, do not delete the branch]

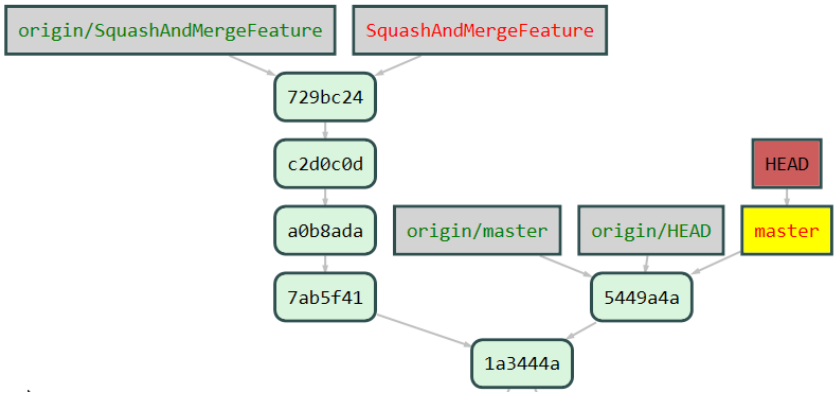


Step 3: Go back to local and get master up to date, then compare with the feature branch.

a) Get LOCAL master up to date

[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity
hAndMergeFeature)
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwar
  (use "git pull" to update your local branch)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity
r)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity
r)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating 1a3444a..5449a4a
Fast-forward
 info.txt | 5 +++++
 1 file changed, 5 insertions(+)
```



Here we can see that our feature four commits do not line up with the master commit history – this is to be expected, but it poses a problem. If we were to try to do a pull request we end up looking like we have multiple commits.

b) Merge master into feature

[git checkout <feature>]

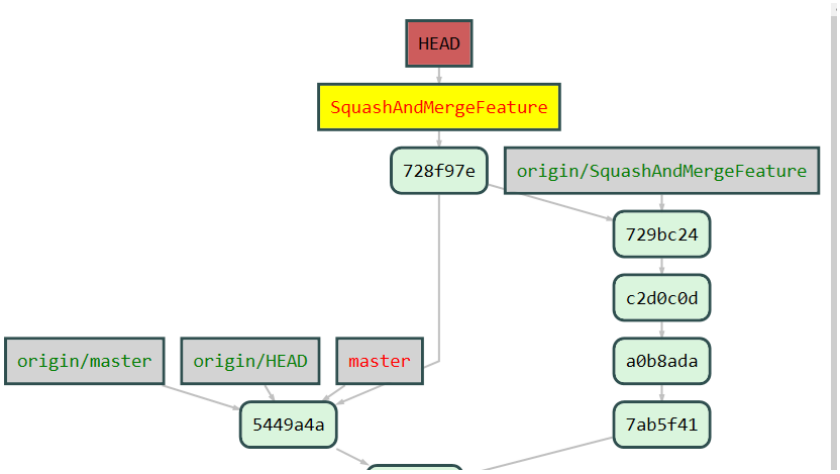
[git merge master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMerg
r)
$ git checkout SquashAndMergeFeature
Switched to branch 'SquashAndMergeFeature'
Your branch is up-to-date with 'origin/SquashAndMergeFeature'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMerg
hAndMergeFeature)
$ git merge master
```

MERGE_MSG

```
Merge branch 'master' into SquashAndMergeFeature

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFold
hAndMergeFeature)
$ git merge master

Merge made by the 'recursive' strategy.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFold
hAndMergeFeature)
$
```



So now our local master has nothing in it that feature doesn't. Also, the original four commits are in master as one commit. Let's add one quick change to the feature.

[code info.txt]

[git commit –am “A single new commit on feature”]

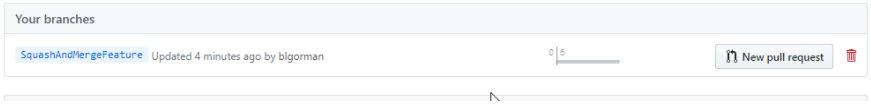
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMerg
hAndMergeFeature)
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMerg
hAndMergeFeature)
$ git commit -am "A single new commit on feature"
[SquashAndMergeFeature 3741e4f] A single new commit on feature
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Step 3: Push to GitHub and merge.

a) Now let’s do another push and create a pull request at GitHub to see what this looks like…

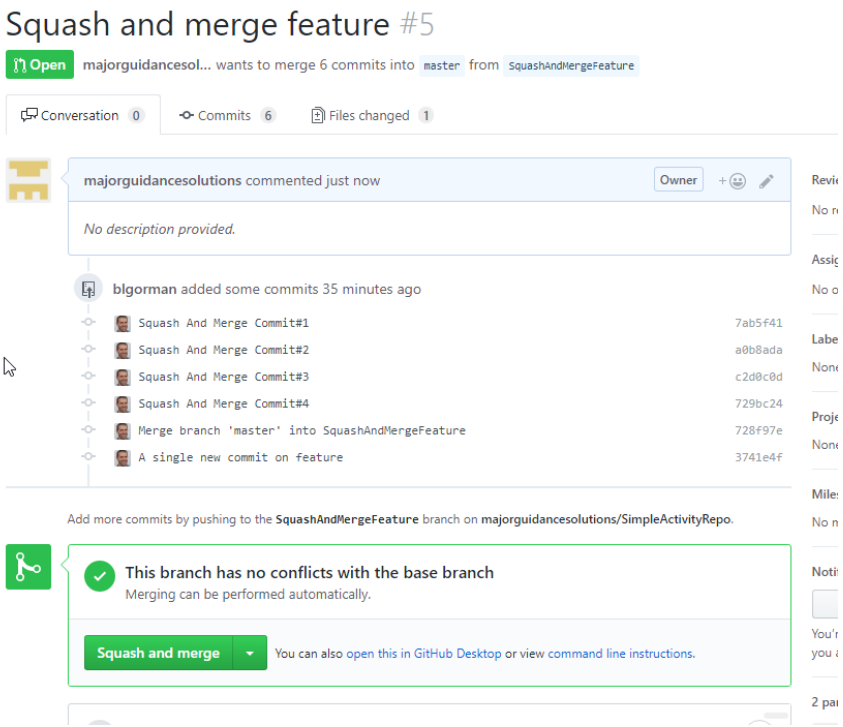
[git push –u origin <featurebranch>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Squas
hAndMergeFeature)
$ git push -u origin SquashAndMergeFeature
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 565 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
   729bc24..3741e4f  SquashAndMergeFeature -> SquashAndMergeFeature
Branch SquashAndMergeFeature set up to track remote branch SquashAndMergeFeature
 from origin.
```



6 commits ahead. Obviously there are only the two we would want (merge master and the new change). This shows why not deleting the branch is an issue. What if this was a major change? Would you ‘trust’ that your original changes were in master?

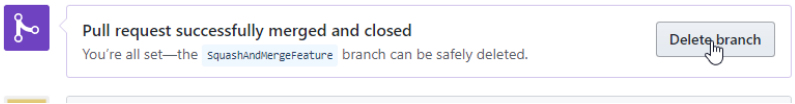
Create the pull request. Before merging, look:



There are my first four commits again, the merge commit, and the new change. 6 commits to get up to date for a simple change.
Notice also that the squash and merge option is still selected. Make sure to change that back if you don't want to squash and merge every time you finish a code review.

Luckily, even with the bad commits showing, the file is still only showing the simple changes that were made:

Go ahead and do a regular merge or a squash and merge if you want just one more commit. This time, delete the branch on completion:



Step 4: Repeat all of the operations from step 2. This time, delete the feature branch after merge.

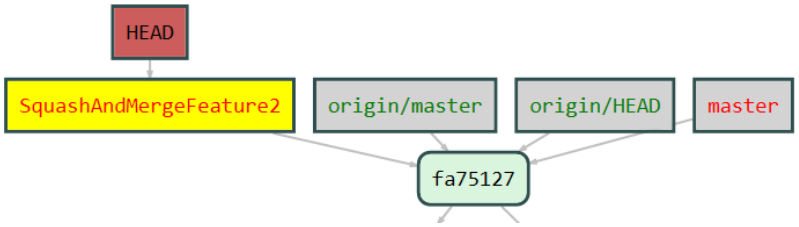
a) Get our repo up to date

[git checkout master]
[git fetch origin] //includes a prune if set. If not [git fetch origin --prune]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Squa
hAndMergeFeature)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mas
r)
$ git fetch origin
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 1 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/majorguidancesolutions/SimpleActivityRepo
   5449a4a..fa75127  master     -> origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mas
r)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating 5449a4a..fa75127
Fast-forward
 info.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

[git branch -d SquashAndMergeFeature]
[git checkout –b SquashAndMergeFeature2]

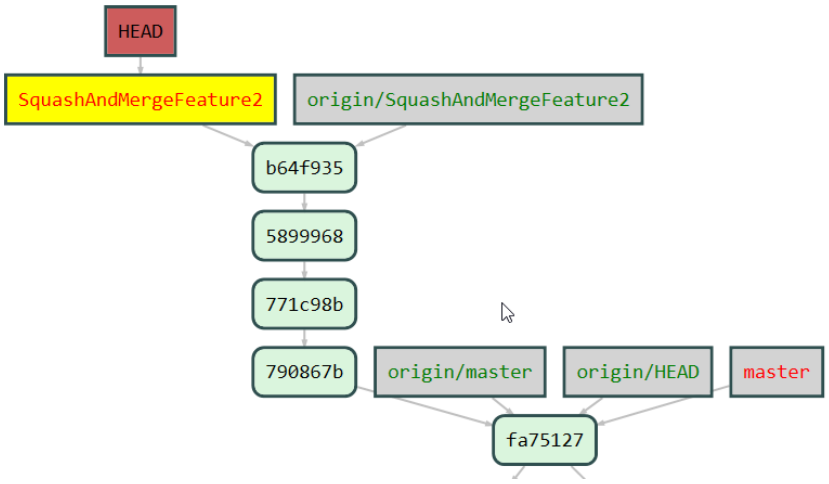
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashA
r)
$ git branch -d SquashAndMergeFeature
Deleted branch SquashAndMergeFeature (was 3741e4f).

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashA
r)
$ git checkout -b SquashAndMergeFeature2
Switched to a new branch 'SquashAndMergeFeature2'
```



b) Perform four commits on the feature, push to GitHub, create PR, squash and merge it.

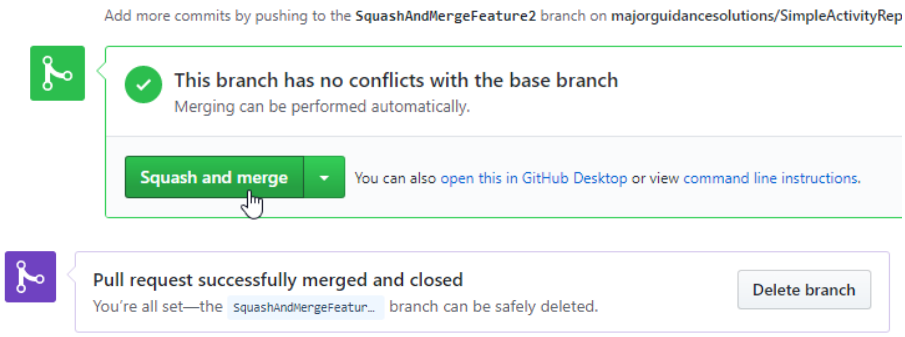
[code info.txt] //leave it open
[git commit –am “Squash and Merge #6]
[make changes]
[git commit –am “Squash and Merge #7]
[make changes]
[git commit –am “Squash and Merge #8]
[make changes]

[git commit –am “Squash and Merge #9]




```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Sq
hAndMergeFeature2)
$ git push -u origin SquashAndMergeFeature2
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 1.03 KiB | 0 bytes/s, done.
Total 12 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
 * [new branch]      SquashAndMergeFeature2 -> SquashAndMergeFeature2
Branch SquashAndMergeFeature2 set up to track remote branch SquashAndMergeFea
e2 from origin.
```

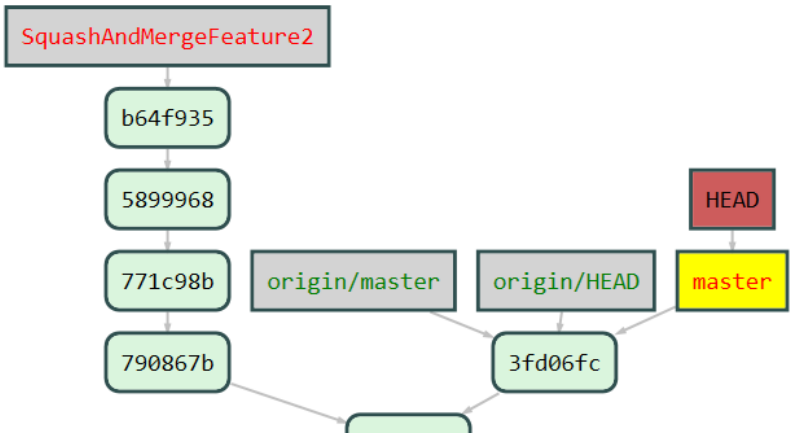
c) Get the pull request going, commit with squash and merge, delete the branch



Step 5: Clean up the local repo, get master up to date.

a) Get master up to date

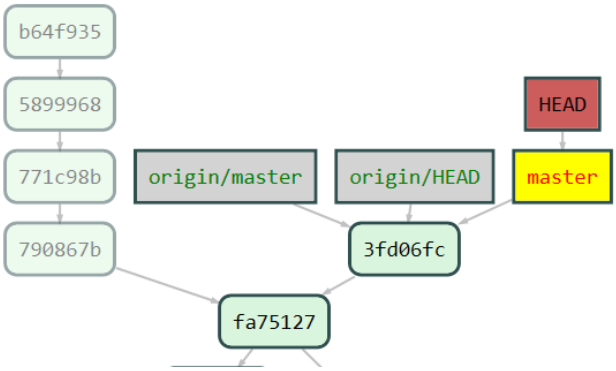
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (Squa
hAndMergeFeature2)
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git pull origin mater
fatal: Couldn't find remote ref mater

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (mast
r)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating fa75127..3fd06fc
Fast-forward
 info.txt | 7 ++++++-
 1 file changed, 6 insertions(+), 1 deletion(-)
```



b) Delete the feature branch

We'll have to force the delete because once again we have four commits that don't line up with the history in master. If we don't use the –D option, git will warn us about unmerged commits:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMergeActivity (maste
r)
$ git branch -d SquashAndMergeFeature2
error: The branch 'SquashAndMergeFeature2' is not fully merged.
If you are sure you want to delete it, run 'git branch -D SquashAndMergeFeature2
'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquash
r)
$ git branch -D SquashAndMergeFeature2
Deleted branch SquashAndMergeFeature2 (was b64f935).
```

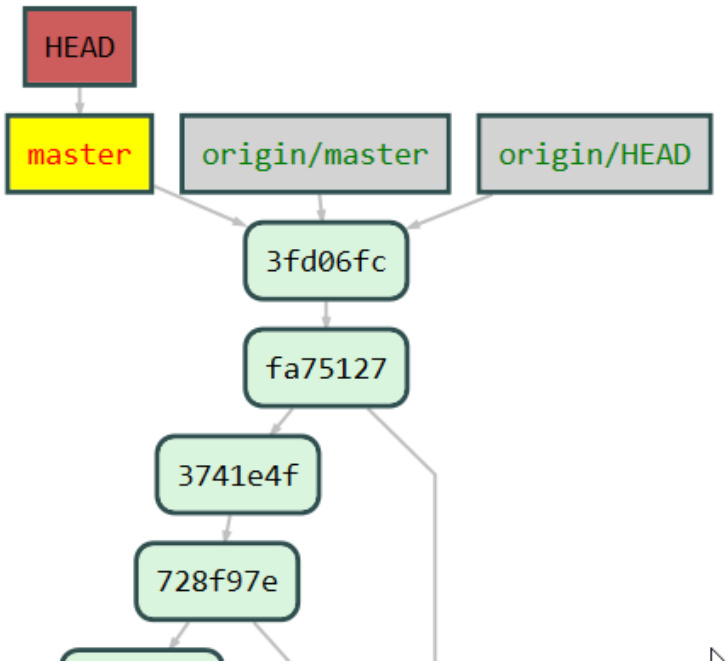


Note that we have four unreachable commits. We need to clean these up.

c) Expire unreachables and garbage collect.

[git reflog expire –expire-unreachable=now –all]
[git gc --prune=now]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMe
r)
$ git reflog expire --expire-unreachable=now --all

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitSquashAndMe
r)
$ git gc --prune=now
Counting objects: 51, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (49/49), done.
Writing objects: 100% (51/51), done.
Total 51 (delta 27), reused 0 (delta 0)
```



This completes our SquashAndMerge at GitHub activity.

Closing Thoughts

Squashing and merging is an easy way to get a number of commits down to just one for storage into the repo history. Often, this would take place at the completion of a feature branch.

Since the squash does change history, it's pretty important to delete your local branch after declaring a squash and merge, simply because your repo at local won't line up with the commit history in master.

In the end we see that when done properly a squash and merge is nice, but the potential ramifications give us reason to make sure we don't keep a squashed branch around.

This activity squashed at remote. It is also possible to squash at local, although it is much more involved. To do this requires an interactive rebasing operation. For now, we're going to hold off on that.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Aliasing Activity:
Using Aliases to simplify commands

Brian Gorman, Author/Instructor/Trainer

Introduction

Git has a lot of flexibility and a lot of commands. However, some commands are just plain verbose, while others are hard to remember, and still others are just tedious to type over and over again. For this reason GIT provides a way for us to setup aliases to use in our day-to-day operations.

In this activity, we'll take a quick look at creation and use of aliases in GIT.

Let's gets started!

Step 1: Working with aliases

Before creating any new aliases, it would be a great idea to make sure they aren't already there. If they are and you added it again, it would just overwrite anyway, but it's nice to know if you already have them or be able to list them at any time.

y) Determine current aliases.

[git config –global –list]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --list
user.email=blgorman@gmail.com
user.name=Brian L. Gorman
core.autocrlf=true
core.editor='C:\Program Files (x86)\Microsoft VS Code\code.exe' -w
core.excludesfile=C:/Users/Brian/.gitIgnore
credential.helper=manager
diff.tool=code
difftool.code.cmd=code --wait --diff $LOCAL $REMOTE
difftool.prompt=false
merge.tool=code
mergetool.code.cmd=code --wait $MERGED
mergetool.prompt=false
mergetool.keepbackup=false
alias.onelinegraph=log --oneline --graph --decorate
alias.expireunreachablenow=reflog expire --expire-unreachable=now --all
alias.gcunreachablenow=gc --prune=now
fetch.prune=true
push.followtags=true
```

I have three aliases set at this time. If I want to see just my aliases, I can do one of two commands:

[git config --global --list | grep alias]

or

[git config –get-regexp alias]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --list | grep alias
alias.onelinegraph=log --oneline --graph --decorate
alias.expireunreachablenow=reflog expire --expire-unreachable=now --all
alias.gcunreachablenow=gc --prune=now

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --get-regexp alias
alias.onelinegraph log --oneline --graph --decorate
alias.expireunreachablenow reflog expire --expire-unreachable=now --all
alias.gcunreachablenow gc --prune=now
```

z) Add an alias.

Before we add these, remember that if you set aliases in the global config they are going to be machine specific. For this reason, you may wish to backup your global config somewhere, as using an alias you rely upon on another machine won't be possible until it's added. Additionally, not typing the command over and over again might cause you to forget the actual command you like to run.

Notes

The syntax to add an alias is fairly straight forward [also remember that if the alias is already there it will get overwritten].

Add an entry to the global config for
alias.<your-alias-name>
followed by the command without 'git' in it and wrapped in quotes.

For example, adding an alias for git checkout master called 'chkmstr' would look like this:
[git config --global alias.chkmstr 'checkout master']

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global alias.chkmstr 'checkout master'
```

[git config --global --get alias.chkmstr] //to see it

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --get alias.chkmstr
checkout master
```

Then just run the command with the keyword git followed by your alias:
[git chkmstr]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git chkmstr
Already on 'master'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git checkout -b anotherBranch
Switched to a new branch 'anotherBranch'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (anotherBranch)
$ git chkmstr
Switched to branch 'master'
```

aa) Add any alias you would like

Here I want to challenge you to pick a couple of commands you would like to alias. For this exercise, keep it to a single command. I'm going to show you how to set up a multiple command alias in just a bit.
//here are my three aliases listed as above:

[git config --global alias.onelinegraph 'log --oneline --graph --decorate']
[git config --global --get alias.onelinegraph]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global alias.onelinegraph 'log --oneline --graph --decorate'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --get alias.onelinegraph
log --oneline --graph --decorate
```

[git config --global alias.expireunreachablenow 'reflog expire --expire-unreachable=now --all']
[git config --global --get alias.expireunreachablenow]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global alias.expireunreachablenow 'reflog expire --expire-
unreachable=now --all'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --get alias.expireunreachablenow
reflog expire --expire-unreachable=now --all
```

[git config --global alias.gcunreachablenow 'gc --prune=now']
[git config --global --get alias.gcunreachablenow]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global alias.gcunreachablenow 'gc --prune=now'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git config --global --get alias.gcunreachablenow
gc --prune=now
```

Step 2: Working with aliases that take variables

As you might already be thinking, perhaps I want to set a variable and use an alias with a variable, such as 'git checkout -b <some-new-branch>. Git has made this possible.

a) Add an alias that allows checking out a variable branch name

The syntax for the alias creation is exactly the same. The only difference is that now we need to add a $n variable where n is the number of the parameter. To add an alias that lets you checkout a new branch at any time, we need to invoke the shell, and we MUST pass the 'git' keyword along with the command now:
[git config --global alias.chknewbr '!sh -c "git checkout -b $1"']
[git config --global --get alias.chknewbr]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git config --global alias.chknewbr '!sh -c "git checkout -b $1"'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git config --global --get alias.chk
alias.chkmstr    alias.chknewbr

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git config --global --get alias.chknewbr
!sh -c "git checkout -b $1"
```

now invoke it:
[git chknewbr 'brian-testing']

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git chknewbr 'brian-testing'
Switched to a new branch 'brian-testing'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (brian-testing)
$
```

And now think of another command you might want to run with a variable and see if you can get that one to work.

Step 3: Aliases that take variables and multiple commands.

To finish up our activity, let's see what happens if we need to run multiple commands and also take a variable

a) Create an alias for checking out master, fetching origin, pulling master to local, then switching to a variable branch and merging master!
[git config --global alias.setuplocalbr '!sh -c "git checkout master && git fetch origin && git pull origin master && git checkout $1 && git merge master"']
[git config --global –get alias.setuploclbr]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (brian-testing)
$ git config --global alias.setuplocalbr '!sh -c "git checkout master && git fet
ch origin && git pull origin master && git checkout $1 && git merge master"'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (brian-testing)
$ git config --global --get alias.setuplocalbr
!sh -c "git checkout master && git fetch origin && git pull origin master && git
 checkout $1 && git merge master"
```

Run it:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (brian-testing)
$ git setuplocalbr brian-testing
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.
Switched to branch 'brian-testing'
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (brian-testing)
$
```

This concludes our look at working with aliases in GIT

Closing Thoughts

Git aliasing is powerful and effective. By setting up a few simple aliases we can easily make our day-to-day command line operations easier to manage.

In this activity, we set up a few simple aliases, and then also saw how we can create aliases that run multiple commands as well as take positional parameters for execution.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Reset and Clean Activity:
Reset changes and clean up your repo/working directory

Brian Gorman, Author/Instructor/Trainer

Introduction

The command [git reset] is a command that can be used to reset your current repository to the state it was in at any particular commit. The commit you may be targeting is the last commit [throwing away your current changes, for example]. The commit you may be targeting could also be a few commits down the chain in history. Going back in history can be somewhat dangerous however, so it is critical to use caution when resetting back to a previous commit that is deep in history. Additionally, if the commit chain is public, and other developers rely on this commit history, then you should probably try to find another way to "reset" your code, unless it is simply unavoidable.

In this activity, we're going to look at different types of resets and some scenarios where we would want to use a soft reset [fairly harmless] to a hard reset [red flag: can be very dangerous]. Reset is one of the few commands that gives us the ability to really wreck our repository, but it should still not induce panic and fear. Always remember, if you are scared to do something, you can fork a repo and try it there, with no risk of your changes causing the main repo to become corrupt.

The [git clean] command gives us a lot of power, and can be used to recursively wipe out files and folders. For that reason you may want to do a dry run or practice your command on a secondary copy of the repository in order to avoid problems. The clean command gives a lot of options, so doing research and running dry runs before performing the actual clean may be your best friends when it comes time to do some cleanup.

Let's gets started!

Step 1: Resetting your current branch to the most recent [HEAD] commit:

bb) Start with any repo, make sure you have the latest in master, and create a feature branch.

First clone the repo if it doesn't exist:

[git clone <link> <folder>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (master)
$ git clone https://www.github.com/majorguidancesolutions/SimpleActivityRepo.git
 ResetAndCleanActivity
```

If you didn't clone, make sure master is up to date

[**git checkout master**]

[**git fetch origin**]

[**git pull origin master**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (master)
$ git fetch origin
warning: redirecting to https://github.com/majorguidancesolutions/SimpleActivity
Repo.git/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (master)
$ git pull origin master
warning: redirecting to https://github.com/majorguidancesolutions/SimpleActivity
Repo.git/
From https://www.github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

[git checkout –b reset-and-clean]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (master)
$ git checkout -b reset-and-clean
Switched to a new branch 'reset-and-clean'
```

cc) Make some changes and then perform a soft reset

[code info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (reset-
-clean)
$ git status
On branch reset-and-clean
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:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Notes

Now reset the state of the repo:

[git reset]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (
-clean)
$ git reset
Unstaged changes after reset:
M       info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (
-clean)
$
```

Nothing happens. So [reset] didn't actually remove the changes. That is good to know. So what does reset do in this case? Nothing. If the changes are staged for commit, then something would have happened.

dd) Stage a change, make another change, perform a soft reset.

[git add info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanA
-clean)
$ git add info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanA
-clean)
$ git status
On branch reset-and-clean
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   info.txt
```

[code info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActi
-clean)
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActi
-clean)
$ git status
On branch reset-and-clean
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   info.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working

        modified:   info.txt
```

Now perform the reset

[git reset]

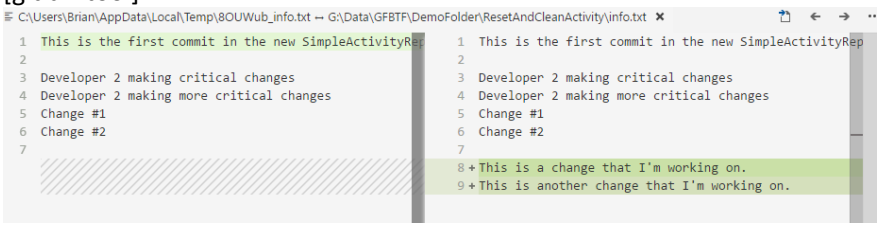
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoF
-clean)
$ git reset
Unstaged changes after reset:
M       info.txt
```

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity
-clean)
$ git status
On branch reset-and-clean
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working dire

        modified:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Check the file. Changes are still there, but now there is nothing that is staged for commit.

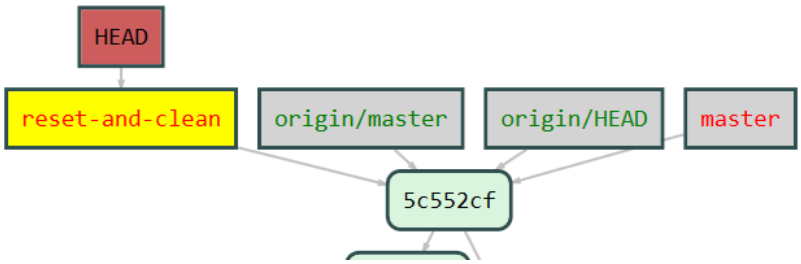
[git difftool]



Step 2: Resetting to a previous commit:

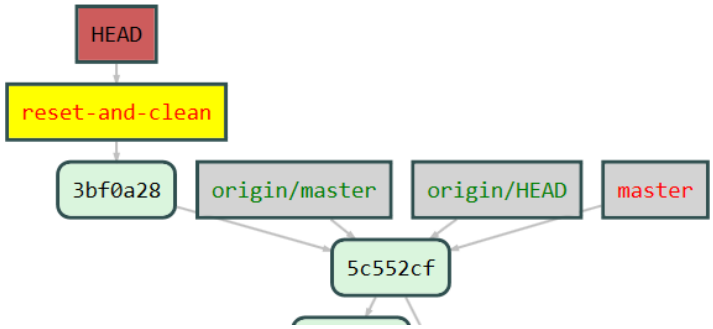
Sometimes we commit our changes and then decide we don't want the commit anymore. Let's take a look at how we might be able to do that:

a) Commit the previous changes



[git commit -am "I think I want to commit these changes"]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ git commit -am "I think I want to commit these changes"
[reset-and-clean 3bf0a28] I think I want to commit these changes
 1 file changed, 3 insertions(+)
```



b) Reset back to the previous commit

Now we decide that we don't want to have that commit after all. What do we do?

We can reset back to the previous commit [5c552cf in this case]

Using git log we can see the commit history as well [if not using GitVis]

[git log --oneline]

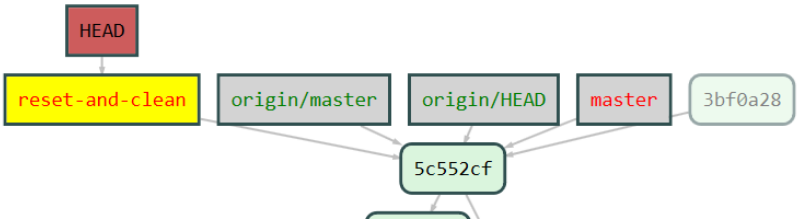
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetA
-clean)
$ git log --oneline
3bf0a28 (HEAD -> reset-and-clean) I think I want to cc
5c552cf (origin/master, origin/HEAD, master) Merge pul
dancesolutions/rebasing-demo-1
a876a65 more changes on my local branch
22e2dd3 changes on my local before rebase
342e3be Merge pull request #1 from majorguidancesoluti
6391b90 Update info.txt
b8a8570 Update info.txt
82b3d34 Create info.txt
7413714 Initial commit
```

[git reset 5c552cf]

```
Brian@SENTINEL MINGW64 /g/Data/GFBT
-clean)
$ git reset 5c552cf
Unstaged changes after reset:
M       info.txt
```

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ git status
On branch reset-and-clean
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working c

        modified:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```



c) Cleanup the unreachable commit

To clean up the commits we just need to make sure we have the reflog set to expire our commits and then run the garbage collector. I have these commands aliased, but in case you don't and you want to run these [or want the commands for later reference], here they are:

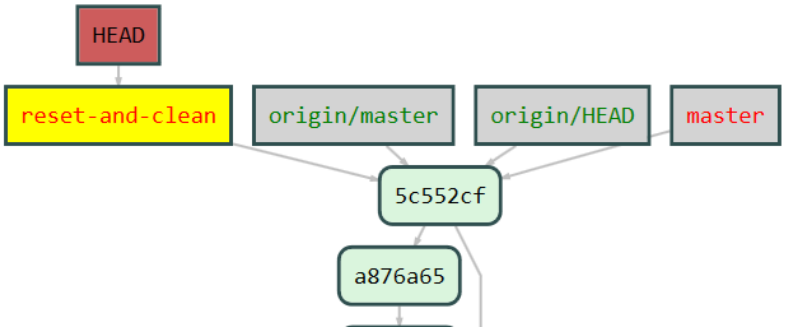
[**git reflog expire –expire-unreachable=now –all**]

And

[**git gc –prune=now**]

And here are my aliases in my global config [check out the aliasing activity for more info about aliasing]:

```
alias.expireunreachablenow=reflog expire --expire-unreachable=now --all
alias.gcunreachablenow=gc --prune=now
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanAc
-clean)
$ git expireunreachablenow

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanAc
-clean)
$ git gcunreachablenow
Counting objects: 22, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (22/22), done.
Total 22 (delta 11), reused 0 (delta 0)
```



Even though we moved back to a previous commit, we didn't lose the changes that were in the commit that we were on

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivi
-clean)
$ git status
On branch reset-and-clean
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working di

        modified:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Step 3: Performing a hard reset:

a) Reset back to HEAD with a hard reset. This will remove any changes, so we only want to do this when we are sure that we don't mind losing changes.

[git reset --hard]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetA
-clean)
$ git reset --hard
HEAD is now at 5c552cf Merge pull request #2 from majo
g-demo-1
```

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/De
-clean)
$ git status
On branch reset-and-clean
nothing to commit, working tree clean
```

Step 4: Performing a hard reset when untracked files are present:

When all of the files are tracked, performing a hard reset will not remove the untracked files. Sometimes we want that to happen as well.

a) Make some changes to info.txt

Before you begin, if you are on a repo that has any important files, you might want to do a quick backup. Our clean operation we are about to run is going to wipe the slate for untracked files

[code info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ git status
On branch reset-and-clean
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working di

        modified:   info.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

a) Add a second text file that is going to remain untracked

[touch readme.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ touch readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActiv
-clean)
$ git status
On branch reset-and-clean
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working d

        modified:   info.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

b) Perform a hard reset back to HEAD

[git reset –hard]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (
-clean)
$ git reset --hard
HEAD is now at 5c552cf Merge pull request #2 from majorguidancesolution
g-demo-1

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity (
-clean)
$ git status
On branch reset-and-clean
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        readme.txt

nothing added to commit but untracked files present (use "git add" to t
```

The readme.txt file is still there. We need to run a clean to get rid of it.

Step 5: Cleaning our repo with [git clean]:

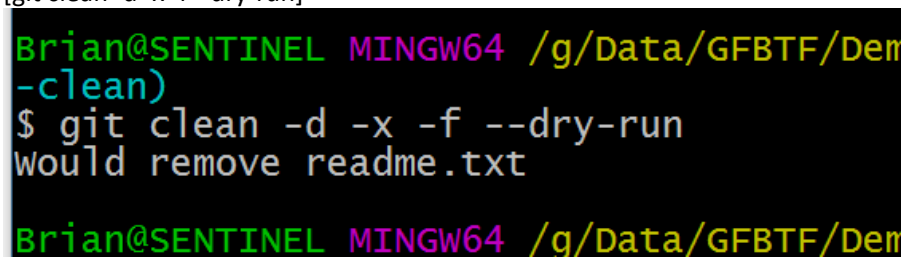
Sometimes there are files that end up in our repo that we don't want anymore. Other times we create a file and don't want it anymore. In either case, we need an easy way to clean up our directory to get our working directory to line up with what the repo says it should have at the latest commit.

a) Perform a full clean

one last warning. Issuing this command will wipe out all files that aren't in the repo from this folder and in it's subfolders.

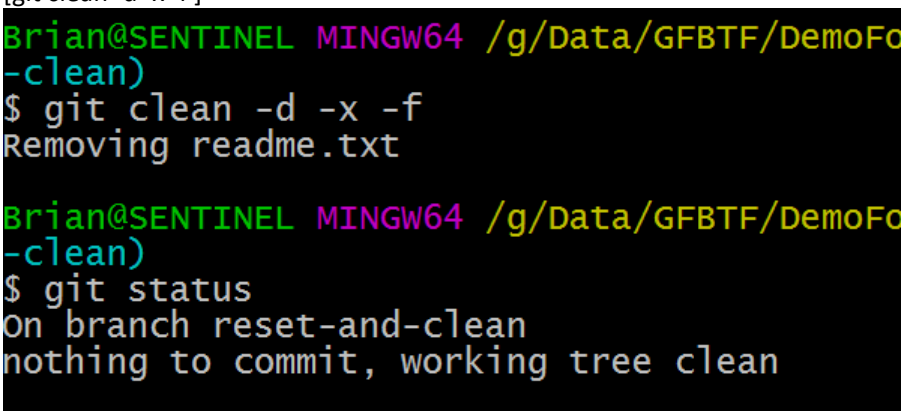
Since the clean command is destructive, let's do a dry-run to make sure it won't hurt us too badly:

[git clean -d -x -f --dry-run]



That looks ok to me. Let's do it:

[git clean -d -x -f]



So this points out that we can use clean anytime (not just after a reset) to just get our files and folders cleaned up on our local repository.

b) Perform an interactive clean

We've seen that the clean command can be destructive. What if the dry-run command above had listed one file that we wanted to keep? In that case we couldn't have used the -f option. For this reason, there is an interative option.

[mkdir resources]
[cd resources]
[touch important.dll]
[touch notimportant.dll]
[touch readme.txt]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/
-clean)
$ mkdir resources

Brian@SENTINEL MINGW64 /g/Data/GFBTF/
-clean)
$ cd resources

Brian@SENTINEL MINGW64 /g/Data/
(reset-and-clean)
$ touch important.dll

Brian@SENTINEL MINGW64 /g/Data/
(reset-and-clean)
$ touch notimportant.dll

Brian@SENTINEL MINGW64 /g/Data/
(reset-and-clean)
$ touch readme.txt
```

[ls]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity/resourc
(reset-and-clean)
$ ls
important.dll  notimportant.dll  readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity/resourc
(reset-and-clean)
$ git status
On branch reset-and-clean
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ./

nothing added to commit but untracked files present (use "git add" to track)
```

[git clean -d -x -i] // i => interactive x => cleans even ignored files

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/ResetAndCleanActivity/reso
(reset-and-clean)
$ git clean -d -x -i
Would remove the following items:
  important.dll     notimportant.dll  readme.txt
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers
    4: ask each             5: quit                 6: help
What now> |
```

[choose option 4] //you can play with the others if you want

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoF
(reset-and-clean)
$ git clean -d -x -i
Would remove the following items:
  important.dll     notimportant.dll  read
*** Commands ***
    1: clean                2: filter by p
    4: ask each             5: quit
What now> 4
Remove important.dll [y/N]?
```

[keep important.dll]
[remove the rest]

```
What now> 4
Remove important.dll [y/N]? n
Remove notimportant.dll [y/N]? y
Remove readme.txt [y/N]? y
Removing notimportant.dll
Removing readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/
(reset-and-clean)
$ git status
On branch reset-and-clean
nothing to commit, working tree clean

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/
(reset-and-clean)
$
```

This concludes our reset and clean activity.

Closing Thoughts

In this activity, we have seen how we can use the fairly safe “soft reset” in order to reset our repository back to the state it was in at the last commit without losing any changes.

We’ve also seen how we can reset back to any other commit in our history. It’s very important to remember, however, that anytime there is a history re-write capability, we should be very careful not to do this against a commit history that is public.

We wrapped up the activity with a look at using the [git clean] command to clear out our working directory of files that are untracked, which can happen on a hard reset when we’ve added files, or a build added files, etc.

It is equally important to remember that a git clean operation is destructive to our working directory, so we examined the --dry-run capability, as well as doing an interactive clean with the –i flag.

Take a few minutes to make some notes about the various commands we’ve learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Revert Activity:
Undoing a public commit using revert

Brian Gorman, Author/Instructor/Trainer

Introduction

Sometimes we have made a commit and we need to undo the commit. In many cases, when the commit is private, we can simply 'reset' back to where we want to go. However, there are times when a commit has been made public. In these cases, one of the safest ways to undo the commit is to use the revert command.

In essence, a revert command simply reverses the original commit(s) and gets the repository back to the state where it was before the reverted commit(s). The system then records a new commit that manages the "undo" operations, and allows us to publish a public commit that will allow all other dependent users to be able to easily get the latest changes, which include the revert commit with changes reverted. The public commit allows for history to remain in tact. As with any merge, reset, rebase, and/or pull operation, a revert requires any conflicts to be resolved for the operation to complete successfully.

Let's gets started!

Step 1: Make sure you have a valid working repository, where a pull request was just closed on master.

ee) Clone the repo with a commit to revert or get the latest on your existing repo.

[**git clone <repo> <folder>**]

[cd <folder>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/SimpleActivityRepo.git Gi
RevertDemo
Cloning into 'GitRevertDemo'...
remote: Counting objects: 30, done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 30 (delta 15), reused 23 (delta 8), pack-reused 0
Unpacking objects: 100% (30/30), done.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ cd GitRevertDemo/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$
```

--if getting latest:

[git checkout master]

[git fetch origin]

[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (master)
$
```

ff) Create a new branch for changes

[git checkout -b GitRevertDemo]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDem
$ git checkout -b GitRevertDemo
Switched to a new branch 'GitRevertDemo'
```

Notes

Step 2: Create a simple commit chain, then revert one commit:

a) Make a small change

[code info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (GitRevertDemo)
$ code info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo (GitRevertDemo)
$ git status
On branch GitRevertDemo
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:   info.txt
```

b) Add and commit the change

[git commit -am "I want to revert this change"]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevert
(GitRevertDemo)
$ git commit -am "I want to revert this change"
[GitRevertDemo 2b482c4] I want to revert this change
 1 file changed, 2 insertions(+)
```

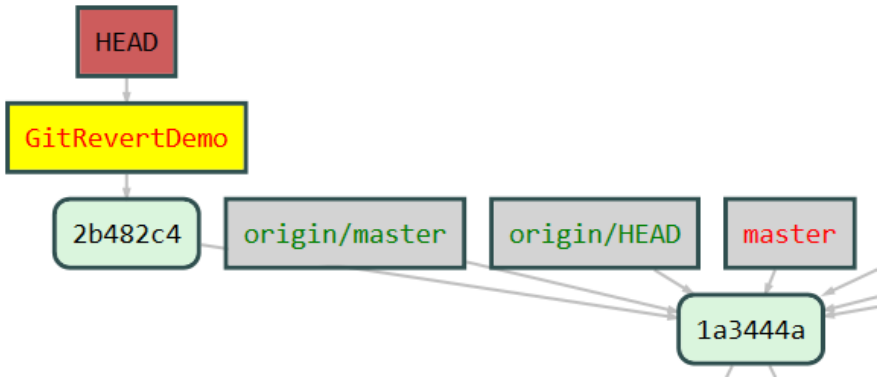
c) View the log – find the commit to revert to, revert it.

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRevertDemo
(GitRevertDemo)
$ git log --oneline
2b482c4 (HEAD -> GitRevertDemo) I want to revert this change
1a3444a (origin/master, origin/HEAD, master) Merge pull reque
t #3 from majorguidancesolutions/GitAmendDemo
0cac76b Changed info.txt and added readme.txt
```

d) Revert the commit.

NOTE: it is important to NOT enter the commit you want to revert TO, but only the commit you want to revert. IF there are other commits, they would be reverted. For example, on this chain:



If I stated [git revert 1a3444a], I would actually be trying to revert BOTH 2b482c4 and 1a3444a from where I am at, and that would be a mess. Instead, I want to revert 2b482c4 -> which essentially should get my repo to be the same as 1a3444a, just with both commits.

[git revert 2b482c4]

COMMIT_EDITMSG

```
Revert "I want to revert this change"

This reverts commit 2b482c4db28ea811a6109ad7f6d883f3c123a48e.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch GitRevertDemo
# Changes to be committed:
#   modified:   info.txt
#

```

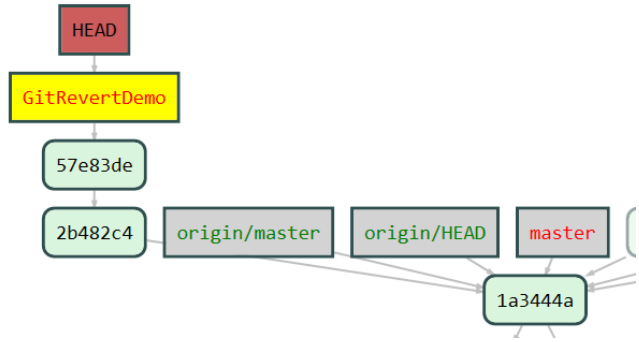
I just need to enter a message to perform the revert. I changed it to 'reverted a bad change'

COMMIT_EDITMSG

```
Reverted a bad change

This reverts commit 2b482c4db28ea811a6109ad7f6d883f3c123a48e.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit
```

Save and Exit:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRev
(GitRevertDemo)
$ git revert 2b482c4

[GitRevertDemo 57e83de] Reverted a bad change
 1 file changed, 2 deletions(-)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRev
(GitRevertDemo)
$
```



[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitRever
(GitRevertDemo)
$ git log --oneline
57e83de (HEAD -> GitRevertDemo) Reverted a bad change
2b482c4 I want to revert this change
1a3444a (origin/master, origin/HEAD, master) Merge pull
t #3 from majorguidancesolutions/GitAmendDemo
```

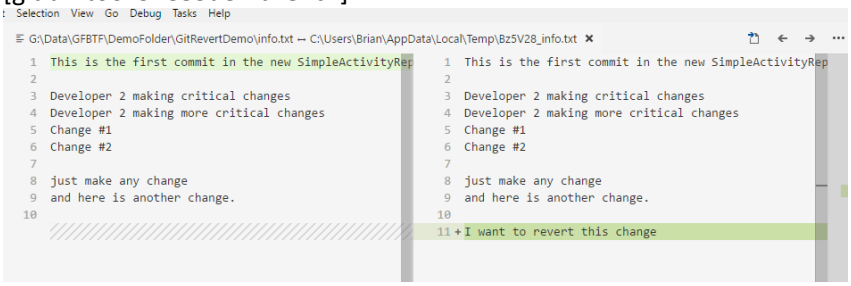
e) See the differences

[git difftool 57e83de 1a3444a]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/
(GitRevertDemo)
$ git difftool 57e83de 1a3444a

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/
(GitRevertDemo)
$
```

//as expected, no difference, difftool not launched

[git difftool 57e83de 2b482c4]



[git difftool 2b482c4 1a3444a]

f) If you want to go further

If you want a bigger challenge, make a couple of commits and revert the first commit in the chain (you'll need to revert everything back that you did)

If you want an even bigger challenge, revert a merge commit. That will require you being able to determine parents. Here are a couple of hints.

1) You can always see merge commit parents by using the command:
   [git show --pretty=raw <commit>]
2) You can dive into the commit parent differences with the following commands
   [git show <commit>^1] //parent 1
   [git show <commit>^2] //parent 2

g) Delete your branch that essentially has no changes [no need to merge]

[git branch –D <branchname>] //need to force it since has unmerged commits

This concludes our GIT Revert Activity.

Closing Thoughts

Using Git Revert is one way we can 'undo' a commit while keeping the commit history in tact. For this demo, we took a quick look into the command and hit a pretty easy revert scenario. Much more difficult revert scenarios exist, however this is beyond the detail I wanted to spend on our first encounter with the command.

The nice thing about a revert operation is that as long as there were no conflicts to resolve [such as when reverting a merge commit], the system will auto-revert for us and we just need to enter a message.

The main thing to remember is that we don't want to enter the commit id of the commit we want to get back to, but rather, we want to enter the commit id of the first commit (and any of its descendants) that we want to undo.

The last part gives a couple of challenges if you want to go deeper with the revert command.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Rebasing Activity: Moving commits in history

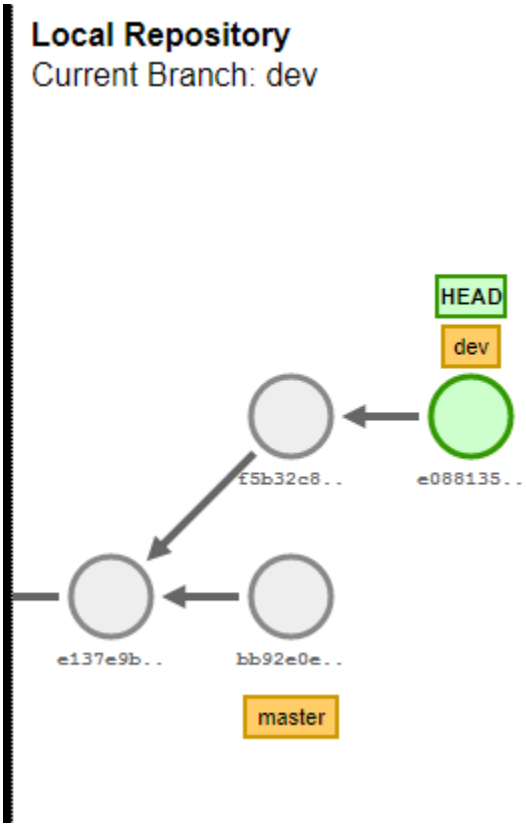
Brian Gorman, Author/Instructor/Trainer

Introduction

Rebasing is one of the most interesting commands we can do when working with GIT. To rebase or not to rebase – that is the question. Much like the 'tabs vs. spaces' or 'coke vs. pepsi' debates, there are strong camps on both sides of the pulling with and pulling without rebase camps. Just do a quick google search and you'll find many passionate pleas to 'always rebase when you pull' or 'never rebase your public branch.'

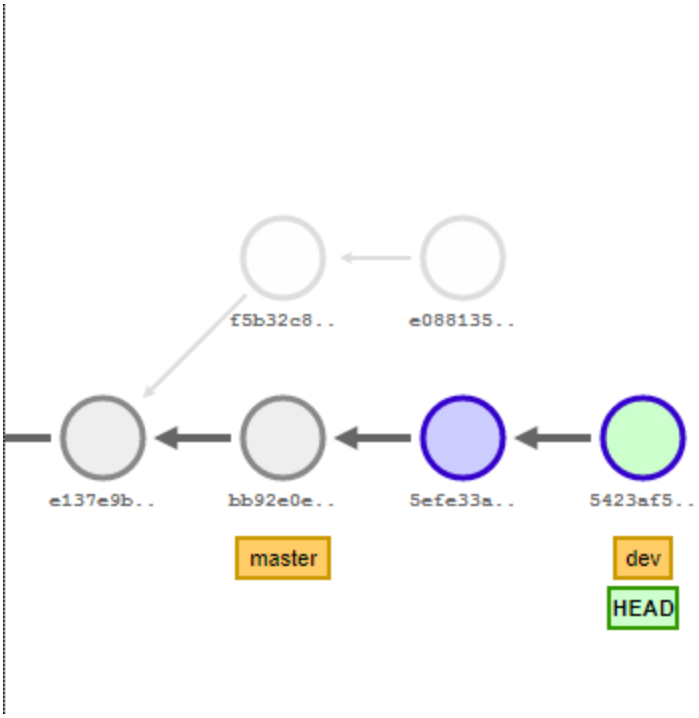
So what does a rebase really do, and why is it something we would want to use? To put it quite simply, rebasing is nothing more than changing the parent commit of another commit. To actually describe it would sound something more like 'moving the base commit of a chain of commits so that it appears to have been created in a linear timeline from the most recent commit on the public branch.

A quick look at a rebase shows one common rebasing scenario [from http://onlywei.github.io/explain-git-with-d3/#rebase]



Note that commit f5b32c8 currently has parent e137e9b. When we do a simple rebase, the parent changes to bb92e0e – but we also get a new commit id [this is why rebasing is somewhat dangerous – but nothing to fear – more on that later].

[git rebase master]



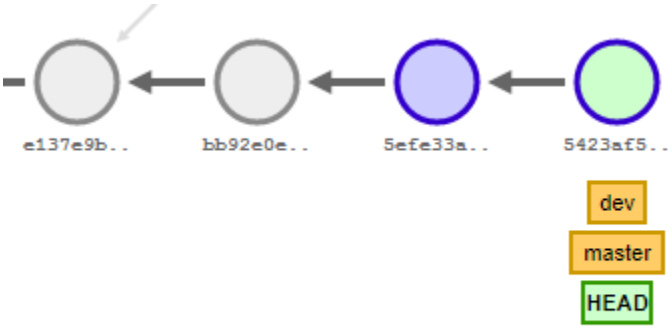
After rebasing, we have a linear commit chain, and it appears that commit 5efe33a started AFTER bb92e0e. In fact, the commit started after e137e9b, but we've changed the history timeline.

We have to be careful -> If other developers are relying on our history to show the commit chain as it was, committing this rebase to public would be a disaster.

Once we have the rebase done, however, we can commit the change into master as a regular merge

[git checkout master]

[git merge dev]



And now everyone can be happy with a history that is linear and public. In this activity, we're going to take a deeper dive into rebasing so we can master the idea of rebasing a commit or commit chain.

Let's gets started!

Step 1: Start with any public repository

a) Create a public branch, get it local, make a couple of changes

After creating the public branch, pull it to local, make a couple of changes, and commit to LOCAL HEAD, but don't push to REMOTE

[fork & **clone a simple repo – or create your own with a simple text file]**

[git clone <new_repo_url>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/SimpleActivityRepo.git Reb
asingActivity1
Cloning into 'RebasingActivity1'...
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 8 (delta 1), reused 8 (delta 1), pack-reused 0
Unpacking objects: 100% (8/8), done.
```

[**git fetch origin**]

[**git pull origin master**] //always make sure to be up-to-date

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

[git checkout -b <branchname>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Reba
$ git checkout -b rebasing-demo-1
Switched to a new branch 'rebasing-demo-1'
```

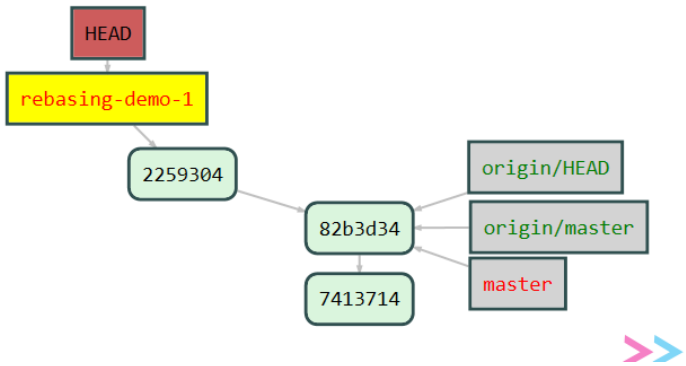
…make some changes…

[code info.txt]

info.txt

```
1   This is the first commit in the new SimpleActivityRepo
2
3   Change #1
```

[git commit –am "changes on my local before rebase"]



Notes

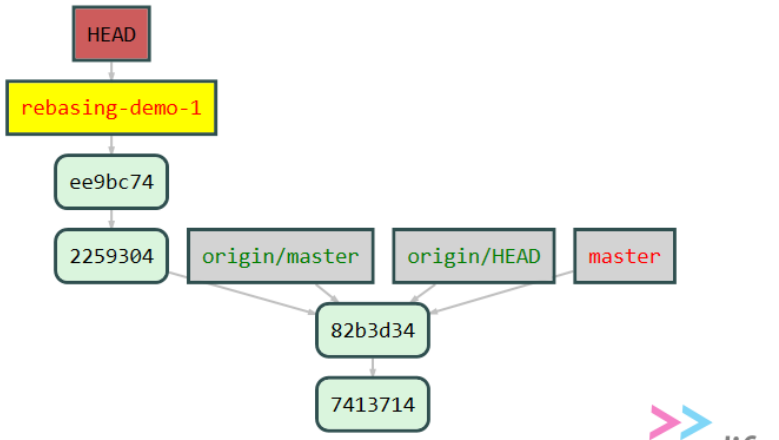
…make more changes…

info.txt

```
1   This is the first commit in the new SimpleActivityRepo
2
3   Change #1
4   Change #2
```

[git commit –am "more changes on my local branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (r
-1)
$ git commit -am "more changes on my local branch"
[rebasing-demo-1 ee9bc74] more changes on my local branch
 1 file changed, 2 insertions(+), 1 deletion(-)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (r
```

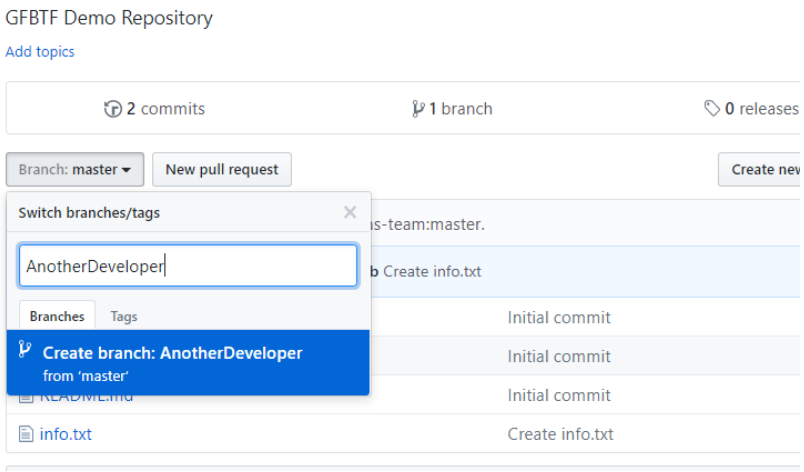


readifv

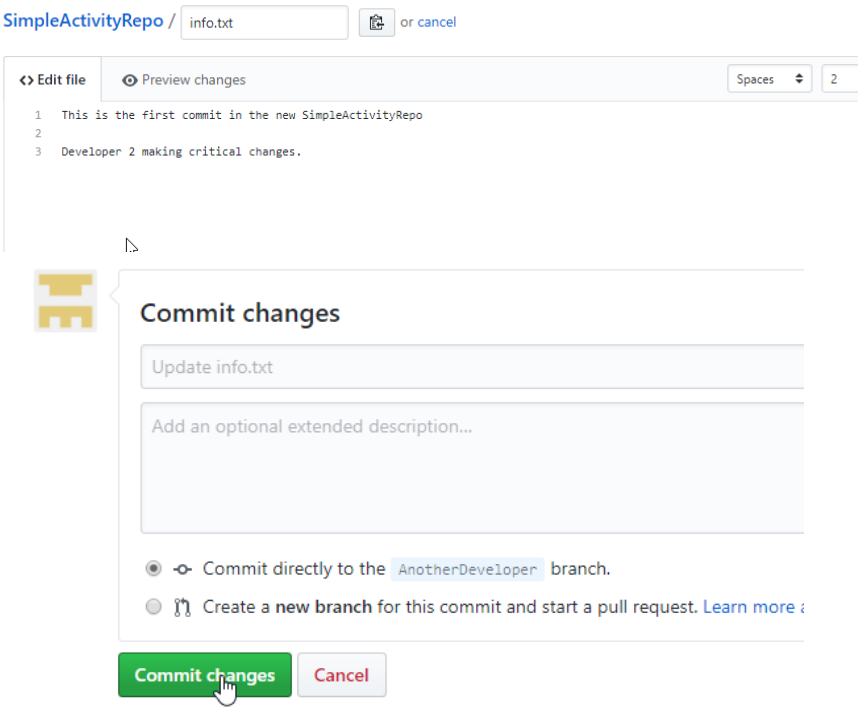
b) Simulate changes by another developer at the repo

Put in a couple of changes on a branch, the merge it into master

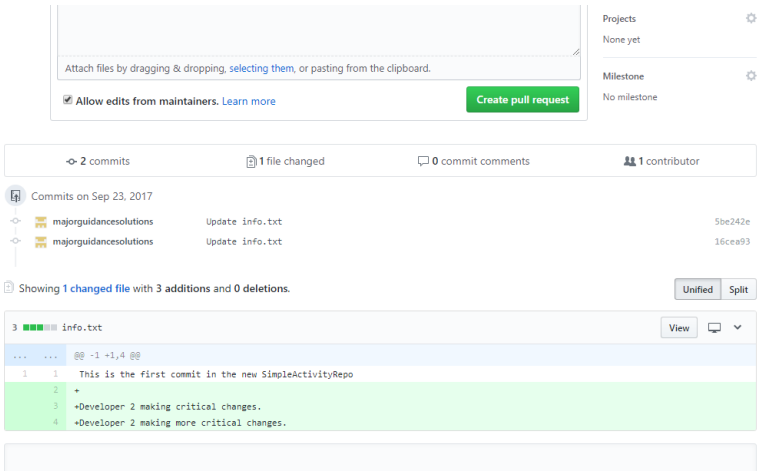
Create the branch



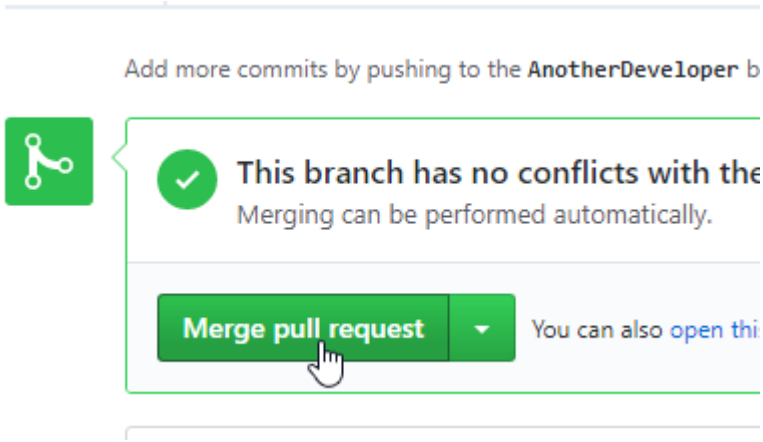
Modify the info.txt file [we are not avoiding conflict]



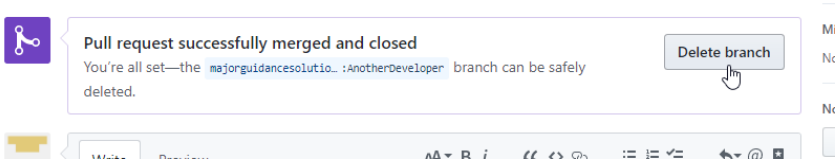
Repeat to create a second commit, then create pull request with the two commits:



Merge.



Delete branch

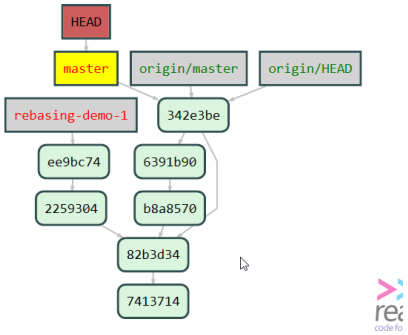


c) Get the latest locally, then rebase locally. Solve the merge conflict on rebase.

First, we need to switch back to master, fetch and pull:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git fetch origin
gremote: Counting objects: 7, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 5), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (7/7), done.
From https://github.com/majorguidancesolutions/SimpleActivityRepo
   82b3d34..342e3be  master     -> origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating 82b3d34..342e3be
Fast-forward
 info.txt | 3 +++
 1 file changed, 3 insertions(+)
```

Here we see that the origin master has moved ahead three commits – the two for the 'another developer branch' and the one for the merge of the pull request.



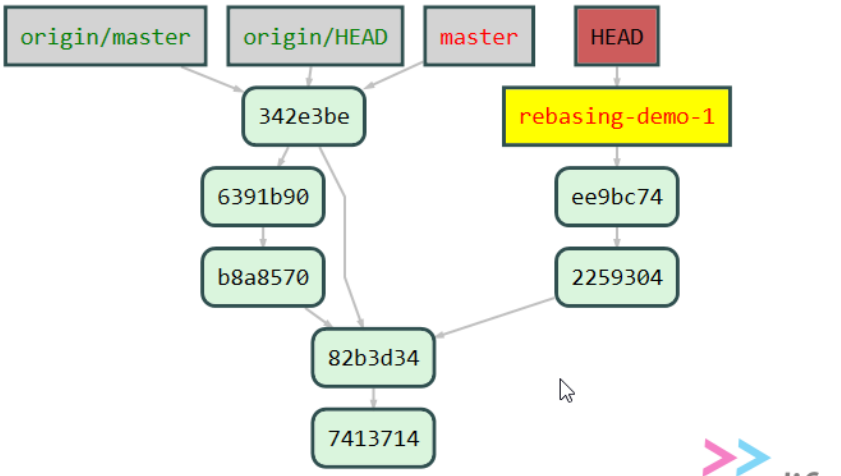
d) Rebase the changes from our branch onto the master

Switch back to our target branch, and rebase master. We'll need to resolve the conflicts with our merge tool as well:

First, make note of our local commit IDs [ee9bc74 and 2259304]

[git checkout rebasing-demo-1]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git checkout rebasing-demo-1
Switched to branch 'rebasing-demo-1'
```

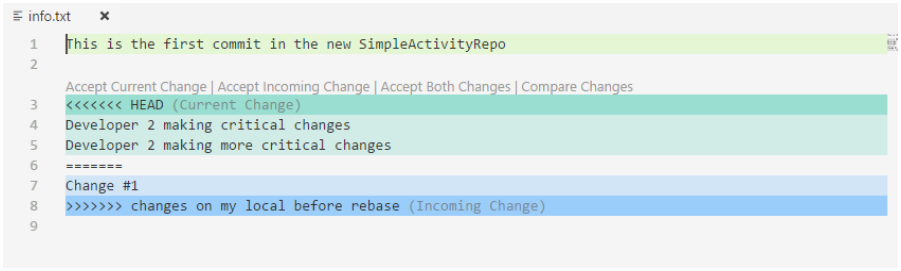


[git rebase master]

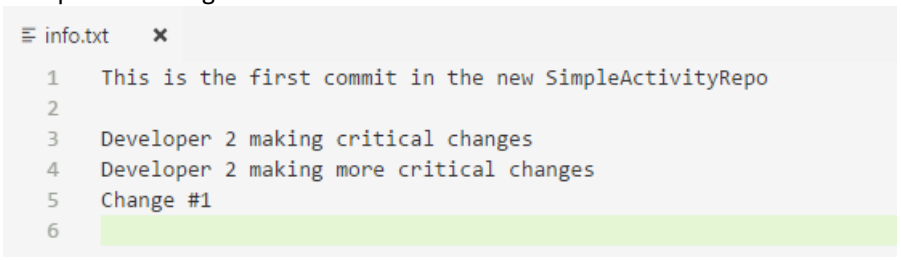
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: changes on my local before rebase
error: Failed to merge in the changes.
Using index info to reconstruct a base tree...
M       info.txt
Falling back to patching base and 3-way merge...
Auto-merging info.txt
CONFLICT (content): Merge conflict in info.txt
Patch failed at 0001 changes on my local before rebase
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".


Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1|REBASE 1/2)
$
```

[**git mergetool**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (reba
-1|REBASE 1/2)
$ git mergetool
Merging:
info.txt

Normal merge conflict for 'info.txt':
  {local}: modified file
  {remote}: modified file
```



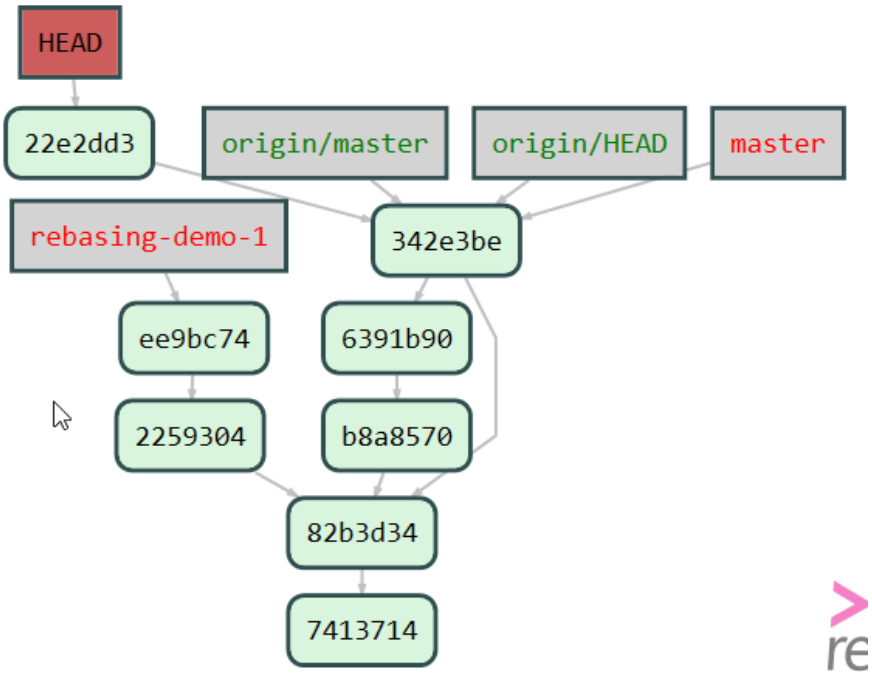
Accept both changes…



Note in the command line we have to rebase and resolve both commits. So this means we'll see the resolution one more time. [you can see REBASE 1/2 in the command text:

[**git rebase --continue**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1|REBASE 1/2)
$ git rebase --continue
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1|REBASE 1/2)
$ git rebase --continue
Applying: changes on my local before rebase
Applying: more changes on my local branch
error: Failed to merge in the changes.
Using index info to reconstruct a base tree...
M       info.txt
Falling back to patching base and 3-way merge...
Auto-merging info.txt
CONFLICT (content): Merge conflict in info.txt
Patch failed at 0002 more changes on my local branch
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".


Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1|REBASE 2/2)
$
```

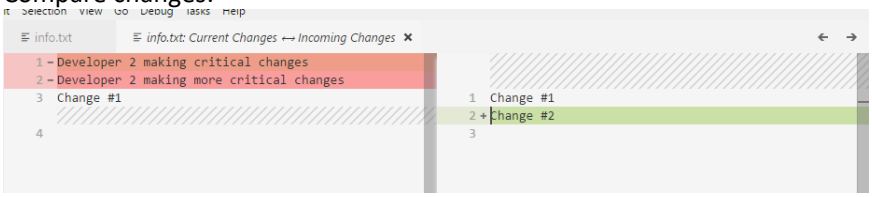


Make a note. We now have a new commit id that is the commit which resolved that first conflict (1 of 2) in the rebase activity. This is going to be our "new" history. This is why it is so critical to not rebase on a public branch. So far no one is dependent on our two commits [ee9… and 22593…] What do you think will happen on the next rebase merge resolution?

[**git mergetool**] //for our second commit.

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Reb
-1|REBASE 2/2)
$ git mergetool
Merging:
info.txt

Normal merge conflict for 'info.txt':
  {local}: modified file
  {remote}: modified file
```

Compare changes:

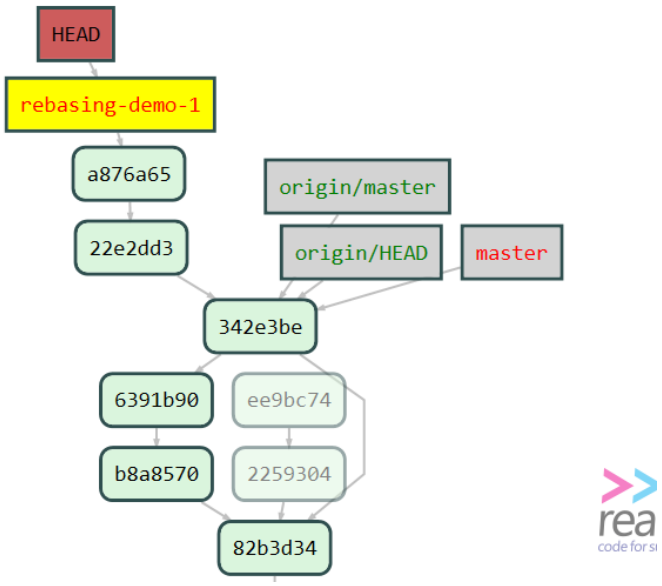


We need to keep the two lines that are getting removed and we'd be ok, so Accept both changes, and delete the duplicated Change #1 line:

info.txt

```
1   This is the first commit in the new SimpleActivityRepo
2
3   Developer 2 making critical changes
4   Developer 2 making more critical changes
5   Change #1
6   Change #2
7
```

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1
-1|REBASE 2/2)
$ git rebase --continue
Applying: more changes on my local branch
```



Note the commit id's have changed! Now that we've resolved both, we have two new commits. What happened to ee9bc74 and 2259304? They are kind of grayed out – because they are now in an 'unreachable' state. And that's ok.

Our current changes are in two new commits [22e2dd3 and a876a65]. So now we just need to clean up the repository and push to master.

e) Clean up the unreachable commits

To clean up the commits we just need to make sure we have the reflog set to expire our commits and then run the garbage collector. I have these commands aliased, but in case you don't and you want to run these [or want the commands for later reference], here they are:

[**git reflog expire –expire-unreachable=now –all**]

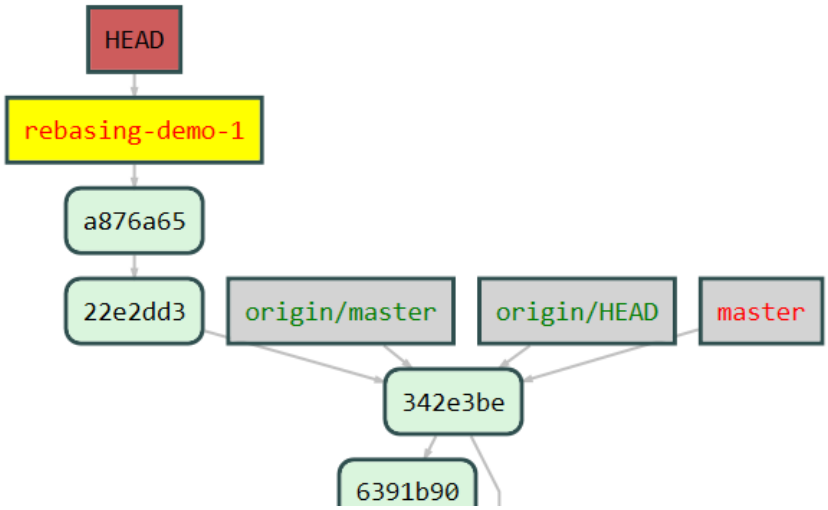
And

[**git gc –prune=now**]

And here are my aliases in my global config [check out the aliasing activity for more info about aliasing]:

```
alias.expireunreachablenow=reflog expire --expire-unreachable=now --all
alias.gcunreachablenow=gc --prune=now
```

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Rebas
-1)
$ git expireunreachablenow

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/Rebas
-1)
$ git gcunreachablenow
Counting objects: 21, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 11), reused 12 (delta 6)
```

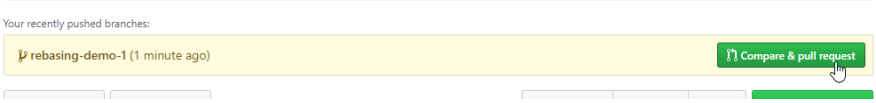


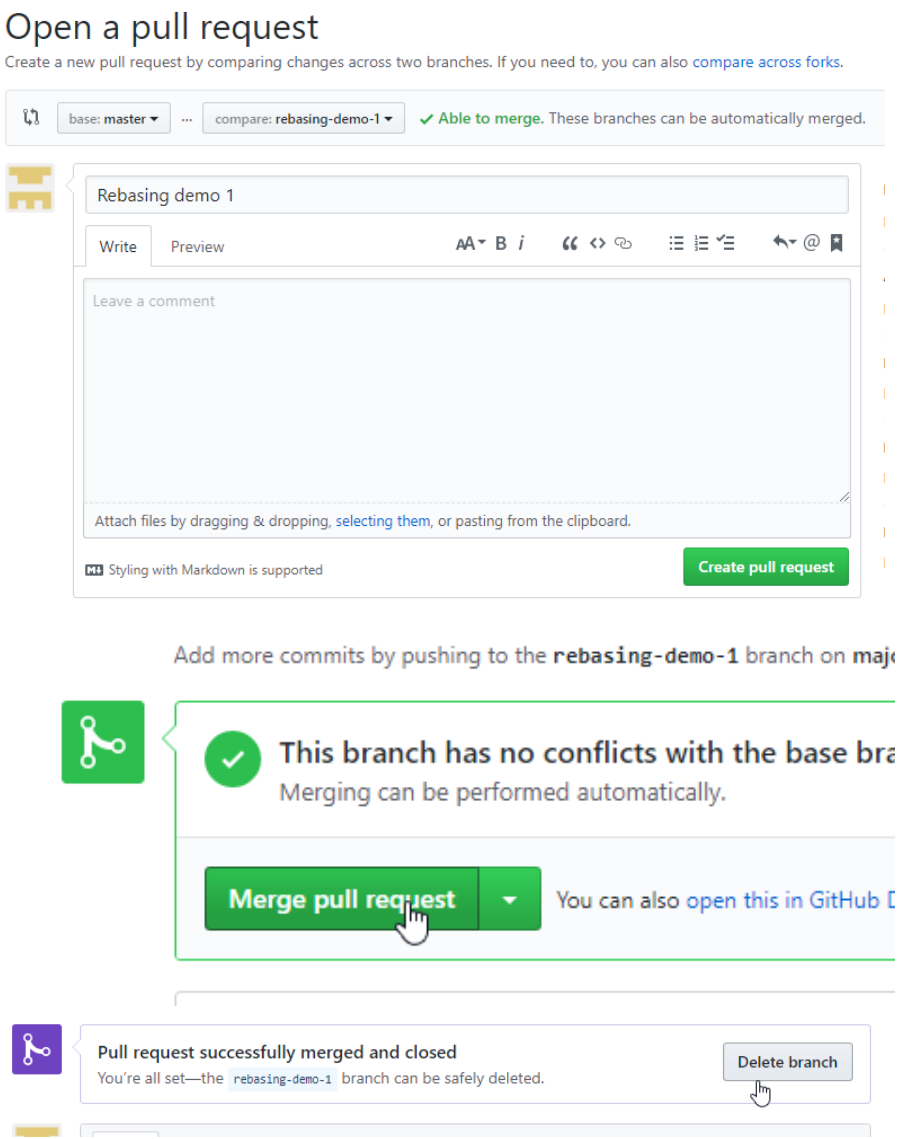
f) Push our changes, pull request, and merge to master

[git push –u origin rebasing-demo-1]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-demo
-1)
$ git push -u origin rebasing-demo-1
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 552 bytes | 0 bytes/s, done.
Total 6 (delta 4), reused 4 (delta 2)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
 * [new branch]      rebasing-demo-1 -> rebasing-demo-1
Branch rebasing-demo-1 set up to track remote branch rebasing-demo-1 from origin
.
```

Create a pull request and merge our changes at GitHub



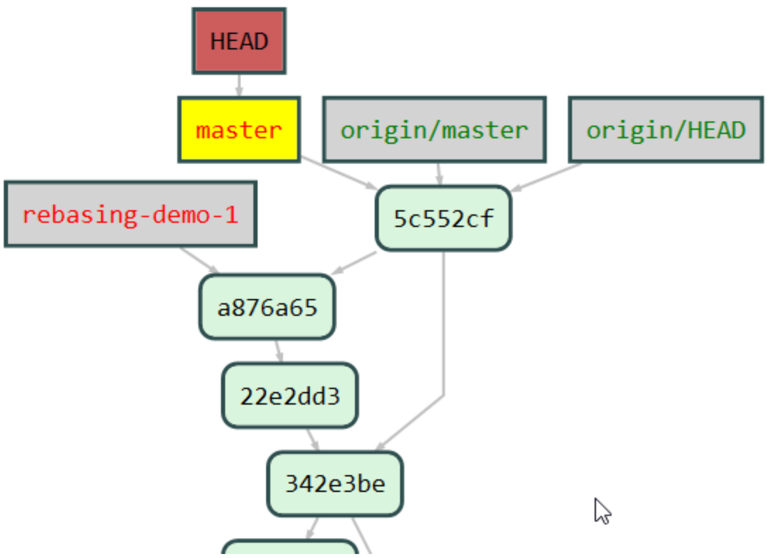


Now our master has everything and our feature branch is deleted so we need to git local up-to-date and cleaned up

g) Get up to date on local master branch and delete our feature branch

[**git checkout master**]
[**git fetch origin**]
[**git pull origin master**]

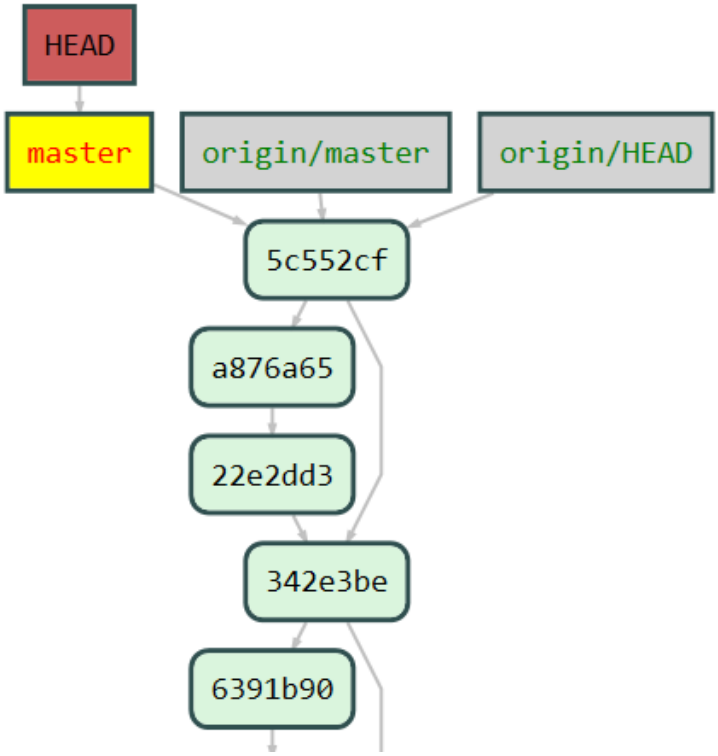
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (rebasing-d
-1)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git fetch origin
From https://github.com/majorguidancesolutions/SimpleActivityRepo

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (mas
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating 342e3be..5c552cf
Fast-forward
 info.txt | 2 ++
 1 file changed, 2 insertions(+)
```



[note: I have my local set to prune on every fetch, so my local has pruned origin/rebasing-demo-1. If you are doing this and see that branch, run [git fetch origin –prune] and it should go away if you have deleted the branch at REMOTE]

Now that master is up to date, the last thing I need to do is get rid of my feature branch.
[git branch –d rebasing-demo-1]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/RebasingActivity1 (master)
$ git branch -d rebasing-demo-1
Deleted branch rebasing-demo-1 (was a876a65).
```



And that is how we do a rebasing operation with conflict resolutions to move our feature branch commits to have a new parent from an updated history after other developers have made changes.

Closing Thoughts

In this activity we worked through a common rebasing scenario, where another developer had made changes on the repository while we were "in progress." The ability to easily rebase makes GIT fairly flexible as to how you want to create merge resolutions. Unlike the traditional route, using the rebase allows us to "change" the order of commits in history. So what had started out as being a couple of commits behind the actual history appears to happen directly after the commits.

In the end, you may never actually need to rebase your work, depending on whether or not you care if your work appears as a straight line with no branching or if you don't mind a few branches with reconnects.

Other scenarios for rebasing do exist. For example, I once had to port a Visual Studio Team System history into GitHub. If I didn't want to keep history, it wouldn't have mattered, of course. However, in order to preserve history, I actually was able to create the repo and then rebase master on top of the original history (I know, I said never to rebase master…to somewhat quote a line from one of my favorite movies "this is where you find out how often [Gorman] does things he says not to do").

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Cherry Picking Activity:
Get a specific commit into your history from a chain of commits

Brian Gorman, Author/Instructor/Trainer

Introduction

Imagine you are developing away on a feature branch when you discover a simple bug in some existing code. One strategy you might take is to commit your changes, go back to the master, create a new branch, and then checkout the new branch and make the fix, merge, pull down to local, merge into feature branch and then continue. This is a great way to solve the problem, but creates a few merge commits and also is a lot of extra jumping around (albeit very safe!).

Another scenario exists where you might be developing and you get something completed, and then you keep developing, and you then want to merge part of what you had done in order to get it tested, or just to get the code out the door. As above, there are many ways you can go about working with your codebase to do this.

Yet another scenario is that developer A and B accidentally were working on the same feature. Developer B's solution is farther along and ready to move forward, but there is one part of Developer A's branch that really makes sense to merge into the source for Developer B to use (and just throw away the rest of developer A's work – sorry dev A). Once again, there are many ways to accomplish this.

However, in all of these scenarios what we really want is just part of a feature branch's commit chain to be merged into master. This is where cherry-picking can be a great tool to use. Be advised that cherry-picking is one of the more complicated activities in GIT, but with more complication generally comes more power, right?

Let's dive in and find out!

Step 1: Make sure we have a working repository that is up to date

gg) Start with any repo, make sure you have the latest in master, and create a feature branch.

First clone the repo if it doesn't exist:

[git clone <link> <folder>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/SimpleActivityRepo.git Che
rryPickingActivity
Cloning into 'CherryPickingActivity'...
remote: Counting objects: 51, done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 51 (delta 28), reused 39 (delta 16), pack-reused 0
Unpacking objects: 100% (51/51), done.
```

If you didn't clone, make sure master is up to date

[**git checkout master**]

[**git fetch origin**]

[**git pull origin master**]

[git checkout –b feature-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git checkout -b feature-branch
Switched to a new branch 'feature-branch'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-b
ranch)
$
```

hh) Make 3 commits that are easily identified. Understand that if we take commit 2, commit 1 will be included by default and only commit 3 would be left out, so make your changes accordingly, and clear.

[code info.txt] //leave it open

[git commit -am "CherryPickingActivity – commit #1"]

[make changes in info.txt]

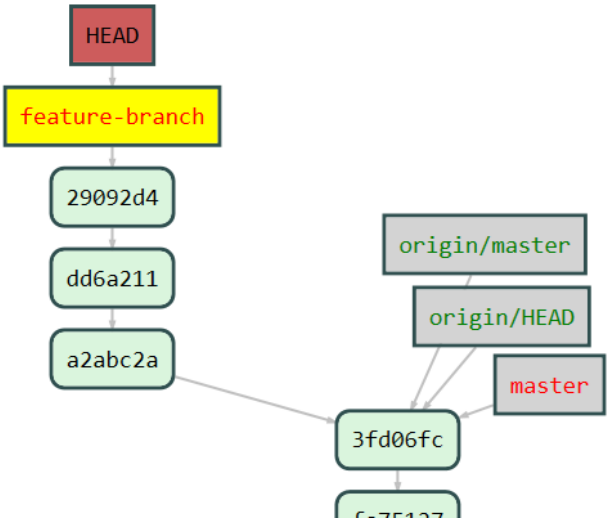
[git commit -am "CherryPickingActivity – commit #2"]

[make changes in info.txt]

[git commit -am "CherryPickingActivity – commit #3"]

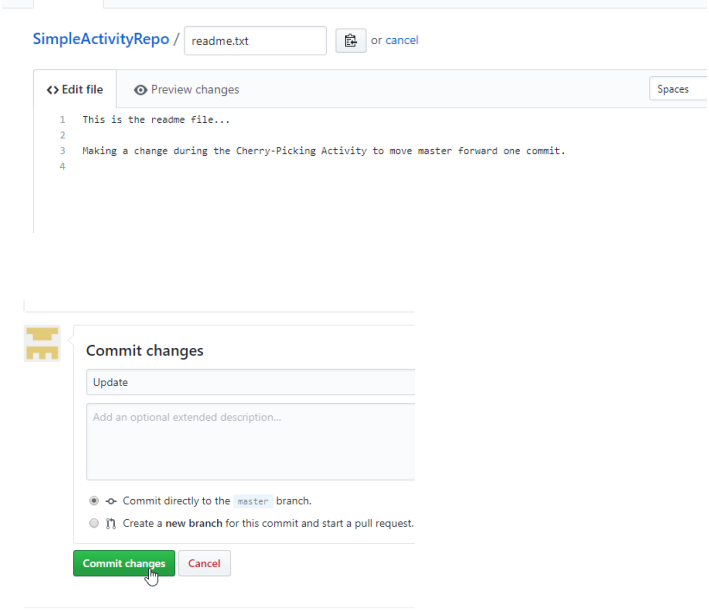
Notes

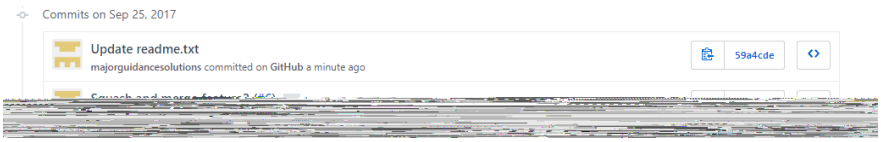
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (featur
ranch)
$ git log --oneline
29092d4 (HEAD -> feature-branch) CherryPickingActivity - commit #3
dd6a211 CherryPickingActivity - commit #2
a2abc2a CherryPickingActivity - commit #1
3fd06fc (origin/master, origin/HEAD, master) Squash and merge feature2 (#6)
```



Step 2: Make one or more commits on the master at GitHub

a) Go out to GitHub and create a new commit to move the master forward one commit. [Don't create a conflict unless you want to have practice resolving conflicts] You can make the commit directly on master to save some time.





Step 3: Get master up to date, create a branch to pick to

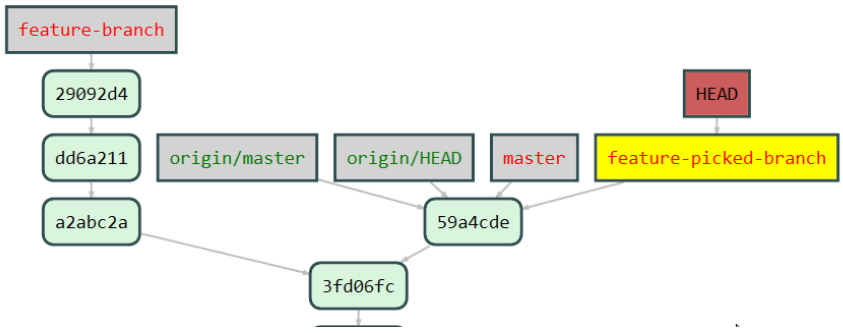
a) Get the latest changes into master

[git checkout master]
[git fetch origin]
[git pull origin master]

b) Checkout a new branch for cherry-picking into:

[git checkout <feature-branch>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git checkout -b feature-picked-branch
Switched to a new branch 'feature-picked-branch'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch)
```



Step 4: Perform the cherry-pick operation

Before we begin, I'd like to point out that if we want all of the commits we could easily do a simple rebase with [rebase master], which would bring all three commits over. If the final commit is easy to undo, it might be easier to rebase, and then revert the last commit. However, this activity is only a simple change, whereas in real life the changes probably wouldn't be that easy to mess with, and the cherry-pick is probably the safer and more trustworthy approach.

a) Begin the cherry-pick with a call to the commit (and by default its parents if any) that we want to pick. Here we will take the second commit, id dd6a211 in our history

[git cherry-pick dd6a211]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch)
$ git cherry-pick dd6a21
error: could not apply dd6a211... CherryPickingActivity - commit #2
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch|CHERRY-PICKING)
$
```

As expected, we have to resolve some merge conflicts. Here is where we'll need to tell GIT what to keep on the cherry-pick. Note that the terminal tells us that we are 'CHERRY-PICKING'. If for some reason we want out, we could simply abort. Do that now:

[git cherry-pick --abort]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch|CHERRY-PICKING)
$ git cherry-pick --abort

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
```

Nothing is done, and we go back to where we were. Of course we actually want to pick, so hit the up arrow a couple of times and re-run the command to cherry-pick dd6a21. Also, before doing that, if you want to validate, you could run a [git log --oneline] to see no extra commits have been added.

b) Resolve the merge conflicts

```
Squash and Merge commit #9
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
=======

Cherry Pick merge #1
Cherry Pick merge #2
>>>>>>> dd6a211... CherryPickingActivity - commit #2 (Incoming Change)
```

Here you can see simple commits I had done for the first two commits. The commit #3 is not shown because I'm not picking it [commit 29092d4]. This is exactly what I want. Of course in real life it will be more complex to hit all the changes, but that is ok as well, because we'll know what we should keep and what we should not keep.

I'll hit [Accept Incoming Change].

Then save and exit the mergetool

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (f
icked-branch|CHERRY-PICKING)
$ git status
On branch feature-picked-branch
You are currently cherry-picking commit dd6a211.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes to be committed:

        modified:   info.txt
```

Good thing I read that! I was about to commit! Instead, I'll run 'git cherry-pick –continue' as the terminal suggests:

[git cherry-pick --continue]

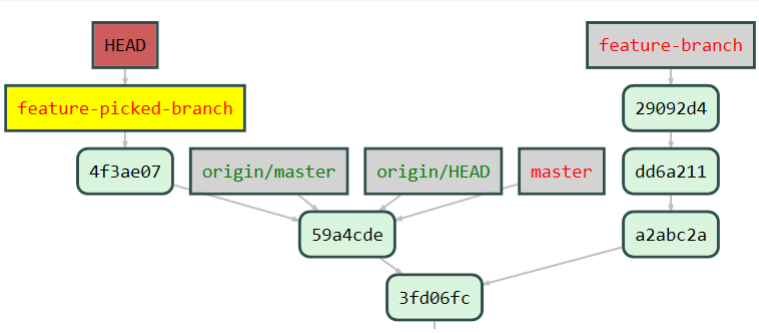
//which just gives me a chance to edit the commit message ☺

```
Selection  View  Go  Debug  Tasks  Help
COMMIT_EDITMSG
1   CherryPickingActivity - commit #2
2
3   # Conflicts:
4   #   info.txt
5   #
6   # It looks like you may be committing a cherry-pick.
7   # If this is not correct, please remove the file
8   #   .git/CHERRY_PICK_HEAD
9   # and try again.
10
11
12  # Please enter the commit message for your changes. Lines starting
13  # with '#' will be ignored, and an empty message aborts the commit.
14  #
15  # Date:      Mon Sep 25 18:03:35 2017 -0500
16  #
17  # On branch feature-picked-branch
18  # You are currently cherry-picking commit dd6a211.
19  #
20  # Changes to be committed:
21  #   modified:   info.txt
22  #
23
```

So I'll take this message, save, and close the editor

And the history:
[git log --oneline]

Sweet, it squashed my two commits into one. That's awesome. But now I need to clear out that other branch, so I need to be SURE that I don't want what's in commit 3 before doing that. I should also make sure I merge my branch to master and get that all squared away so that I have my expected changes before cleaning up.

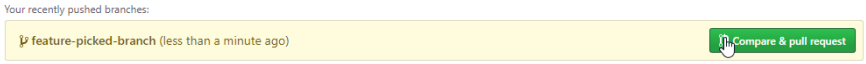


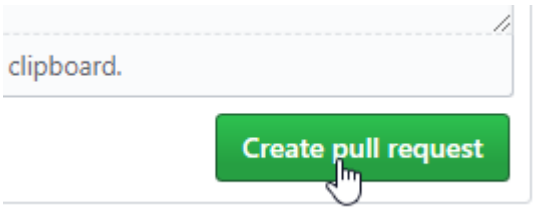
c) Push the feature picked branch to GitHub, do a pull request, merge to master, then update master locally with the history from GitHub

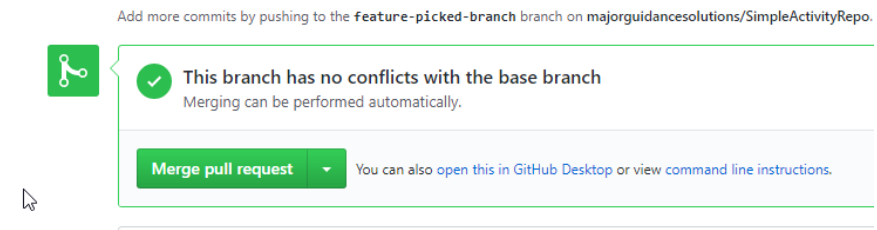
[git push -u origin <your-branch-name>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch)
$ git push -u origin feature-picked-branch
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 339 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo.git
 * [new branch]      feature-picked-branch -> feature-picked-branch
Branch feature-picked-branch set up to track remote branch feature-picked-branch
 from origin.
```

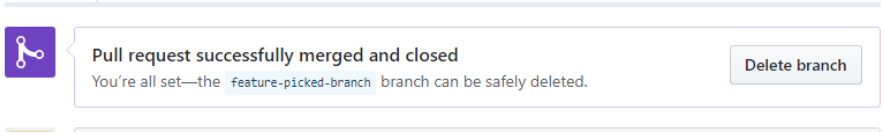
Then at GitHub:







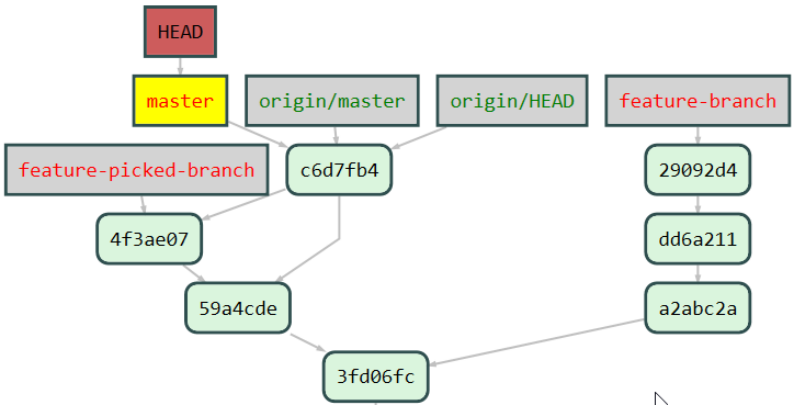
You can either do a regular merge, a squash and merge, or a merge with rebase here, it doesn't matter – as long as we delete our branch after for all but the regular merge commit.
I'm going to do a regular merge pull request as shown.



I'm also going to delete the branch.

Back at local, get master up to date
[git checkout master]
[git fetch origin]
[git pull origin master]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (feature-p
icked-branch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git fetch origin
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 - [deleted]         (none)     -> origin/feature-picked-branch
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 1 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
   59a4cde..c6d7fb4  master     -> origin/master

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Updating 59a4cde..c6d7fb4
Fast-forward
 info.txt | 3 +++
 1 file changed, 3 insertions(+)
```



d) Make sure changes we want are on master from feature branch

//if not using gitvis, would need to look into the reflog
[git reflog]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git reflog
c6d7fb4 (HEAD -> master, origin/master, origin/HEAD) HEAD@{0}: pull origin maste
r: Fast-forward
59a4cde HEAD@{1}: checkout: moving from feature-picked-branch to master
4f3ae07 (feature-picked-branch) HEAD@{2}: commit (cherry-pick): CherryPickingAct
ivity - commit #2
59a4cde HEAD@{3}: reset: moving to 59a4cdef45c63f7b5c3a0679f6035a793f3f8ac9
59a4cde HEAD@{4}: checkout: moving from master to feature-picked-branch
59a4cde HEAD@{5}: pull origin master: Fast-forward
3fd06fc HEAD@{6}: checkout: moving from feature-branch to master
29092d4 (feature-branch) HEAD@{7}: commit: CherryPickingActivity - commit #3
dd6a211 HEAD@{8}: commit: CherryPickingActivity - commit #2
a2abc2a HEAD@{9}: commit: CherryPickingActivity - commit #1
3fd06fc HEAD@{10}: checkout: moving from master to feature-branch
3fd06fc HEAD@{11}: checkout: moving from master to master
3fd06fc HEAD@{12}: clone: from https://github.com/majorguidancesolutions/SimpleA
ctivityRepo.git
```

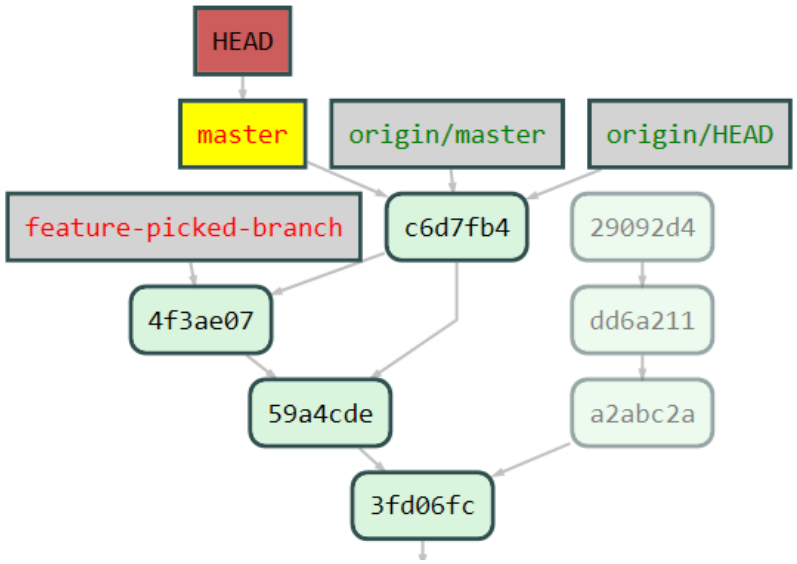
Note the three original commit ids at HEAD@{9}, HEAD@{8}, and HEAD@{7}. Since I'm using GitVis, it's easy to see them on the overall diagram from the previous page [same ids as shown on the reflog].

If the cherry-pick and merge was successful, master, commit c6d7fb4, will have everything from commit a2abc2a and dd6a211, but not 29092d4. Of course if I changed anything during the pick it may have a few differences [or whitespace differences]. That being said, we should be pretty solid for this activity.

Also note-even if we had deleted the branch and had these commit ids, we could still do the comparison:

In fact, let's do it! NO FEAR! [we can get them back if something goes wrong!] [git branch –D <your-branch-name>] //have to use –D to force it! One commit lost forever [right?]!

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity
$ git branch -D feature-branch
Deleted branch feature-branch (was 29092d4).
```



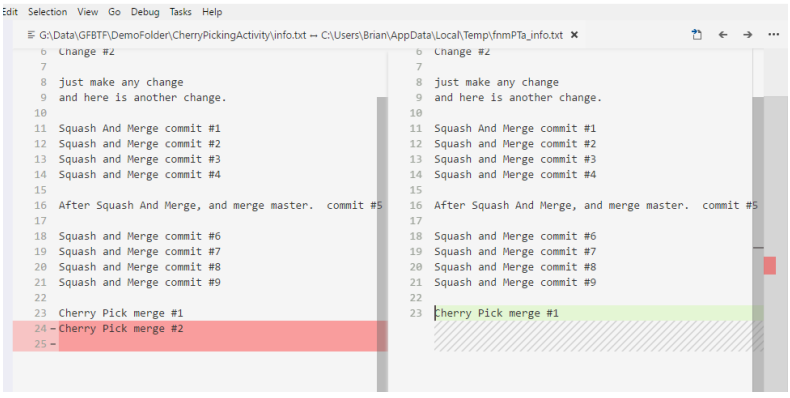
They are still there in cache!

[git reflog]

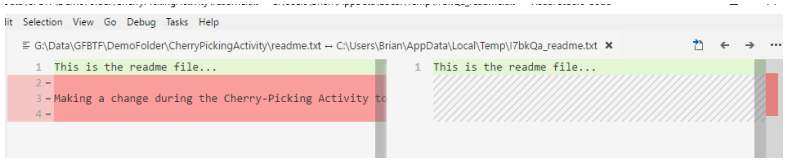
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/CherryPickingActivity (master)
$ git reflog
c6d7fb4 (HEAD -> master, origin/master, origin/HEAD) HEAD@{0}: pull origin maste
r: Fast-forward
59a4cde HEAD@{1}: checkout: moving from feature-picked-branch to master
4f3ae07 (feature-picked-branch) HEAD@{2}: commit (cherry-pick): CherryPickingAct
ivity - commit #2
59a4cde HEAD@{3}: reset: moving to 59a4cdef45c63f7b5c3a0679f6035a793f3f8ac9
59a4cde HEAD@{4}: checkout: moving from master to feature-picked-branch
59a4cde HEAD@{5}: pull origin master: Fast-forward
3fd06fc HEAD@{6}: checkout: moving from feature-branch to master
29092d4 HEAD@{7}: commit: CherryPickingActivity - commit #3
dd6a211 HEAD@{8}: commit: CherryPickingActivity - commit #2
a2abc2a HEAD@{9}: commit: CherryPickingActivity - commit #1
3fd06fc HEAD@{10}: checkout: moving from master to feature-branch
3fd06fc HEAD@{11}: checkout: moving from master to master
3fd06fc HEAD@{12}: clone: from https://github.com/majorguidancesolutions/SimpleA
ctivityRepo.git
```

And in reflog too! If I need 29092d4 back, I can just check it out, create a branch and go with it!

[git difftool c6d7fb4 a2abc2a]

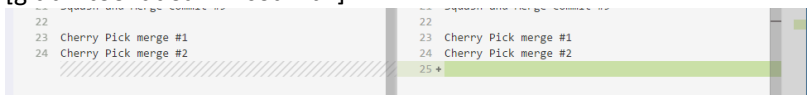


Wait, what? How come there is a difference? Because this was the first of TWO commits, and so it looks like there is a difference from this perspective [so be careful!].

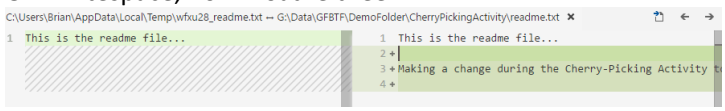


Ah crap – I forgot I made a change at master too. Of course that is different. Let's check the second commit – which should only have the change at master, and let's flip the order. Green is better than red, right?

[git difftool dd6a211 c6d7fb4]

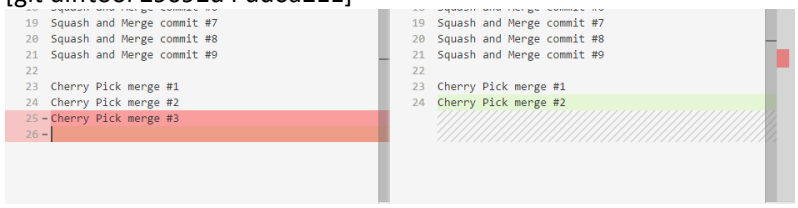


Oh whitespace, how I loathe thee.



Looks good.

Let's validate that commit 3 is not in there..
[git difftool 29092d4 dd6a211]



And it's not. That's great!

e) Clean up the unreachables.

Now that we know the commits that are unreachable don't matter to us anymore, let's use the reflog expire and garbage collector to clean them up.
[git reflog expire --expire-unreachable=now --all]
[git gc --prune=now]

f) Clean up the branch that is behind master now and not at origin anymore, then show the final state of the repo after this activity

[git branch -d feature-picked-branch]

Closing Thoughts

Cherry picking can be scary, so hopefully this activity has removed a lot (if not all) of that fear for you. What we've seen is that we can get the changes from a chain of commits and leave part of the changes out by not including one or more of the commits.

Cherry picking does give us an all-or-nothing operation on the commit, so if you're looking to get just part of a commit, you'd have to use something else, or do the pick and be careful during merge resolution as to what is included.

Because of the nature of the pick, we also got to see a few more things about the reflog and how we can get to 'unreachable' commits even after they don't have a direct reference. We also used the reflog to expire the unreachables and the garbage collector to get rid of those commits.

In the end, this powerful tool allowed us to easily merge the parts of our changes that we wanted while leaving the others behind.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

GIT: From Beginner To Fearless

GIT Stash Activity:
Sometimes you just don't want to commit and you need to get the changes back

Brian Gorman, Author/Instructor/Trainer

Introduction

I have a background that includes too many years in .Net to count. My first version control system was Visual Source Safe 6, which, if any of you have used or encountered, left more than a little bit to be desired.

Fast forward to 2012-2015 when I was using Team Foundation Server and Visual Studio Team System (now Services) for professional development (Git/BitBucket for personal). Anyway, one of my favorite features to use in this centralized source-control system was a feature called "Shelving." Shelving was great because I could put my changes "on the shelf" and basically reset the repository back to the last commit. Anytime I wanted to, I could then pull my changes "off the shelf" and start working with them again.

As such, I was super excited to learn about a feature in Git called "Stashing." My first reaction was "I can't wait to learn about that, because I'm going to use it all the time. It will be the most important feature that Git has to offer, I have no doubt.

Ok, so guess what. It's not. It's great, but it's mostly unnecessary if you don't want to use it. The simple fact of the matter is that creating a branch in Git is so inexpensive and easy to do, that it is easier to just check out and commit my changes to a new branch than to put them "on the shelf" with a stash command.

Add into that – using stash out of the box is a bit counter-intuitive, and without proper planning can quickly become pretty useless, confusing, and un-memorable.

There is one feature that I have found, however, that is perfect for stashing. The feature is when I want to repeatedly do something that I'm not going to check in and I want to be able to do that on any branch, at any time. Also, it's a change, not a command, obviously.

In this activity, we'll go over that one scenario while getting familiar with how the stash command works.

Let's gets to stashing!

Step 1: Start with a copy of my StashActivity repository [has the web.config file, etc]

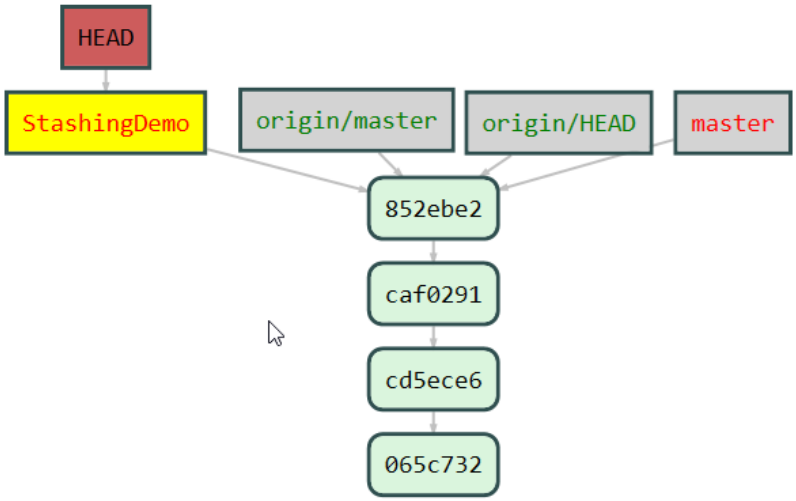
ii) Start with a copy of the repo and create a feature branch.

First fork [then clone local] my repo [it has the coveted web.config file in it]
If you don't want to fork go to the link and download the files and use them to create your own repo:
[git clone https://github.com/majorguidancesolutions/GFBTFStashingActivity.git <folder>]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/GFBTFStashingActivity.git
GitStashingActivity
Cloning into 'GitStashingActivity'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 14 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (14/14), done.
```

[git checkout –b feature-branch]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (master)
$ git checkout -b StashingDemo
Switched to a new branch 'StashingDemo'
```



Nothing spectacular here. The main thing is we have that web.config to play with, and some other files as well.

Notes

jj) Modify the web.config file to use your local db settings.

In case you are not a .net developer, the web.config file is traditionally a simple configuration file that sets up the project and configurable variables [like connection string info] for .Net projects. We're going to simulate this file [not an actual one, and no real db connection, etc]. Our main goal will be to stash our local connection string info while only have the test version of the file in the repository.

[code web.config]
[change the connection string settings from test to local

```
<add name="AppContext"
         connectionString="Server=local_server;Database=dbApp_Local;
                                User
Id=local_user;Password=local_user_pwd;"
         providerName="System.Data.SqlClient"/>
    <add name="AuthContext"

connectionString="Server=local_auth_server;Database=dbAuth_Local;
                                User
Id=local_auth_user;Password=local_auth_user_pwd;"
         providerName="System.Data.SqlClient"/>
```

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (Stashin
o)
$ git status
On branch StashingDemo
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:   web.config

no changes added to commit (use "git add" and/or "git commit -a")
```

Instead of committing, we are going to stash these changes. Doing this will add it to the stash and reset the repo.

Step 2: Simple Stashing

Now that we have the changes we want to use multiple times in our repository across branches, we stash the change:

a) Add the change to stash

[git stash]

[git status]

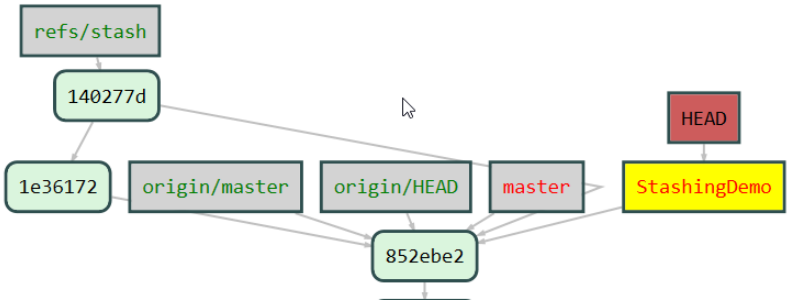
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash
Saved working directory and index state WIP on StashingDemo: 852ebe2 Create read
me.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git status
On branch StashingDemo
nothing to commit, working tree clean
```

b) List the stash

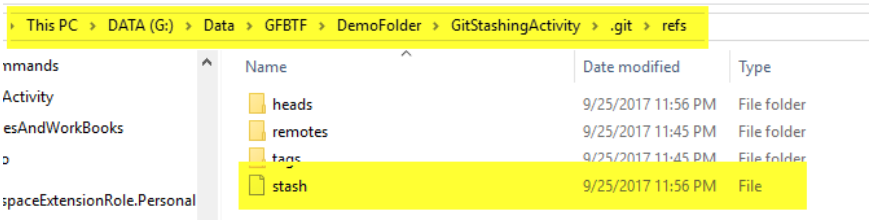
[git stash list]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list
stash@{0}: WIP on StashingDemo: 852ebe2 Create readme.txt
```

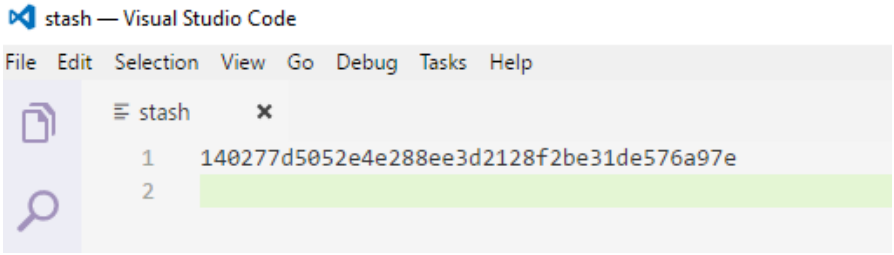
So that isn't very useful. Also, look at the state of our repo



There is some kind of weird thing for refs/stash. What is that?



Open in code:



So there is a commit id in that file. It essentially gives us a pointer to the changes we made on that stash.

Note back up to the list, that the stash points to the main commit.

c) Get changes back and remove from stash

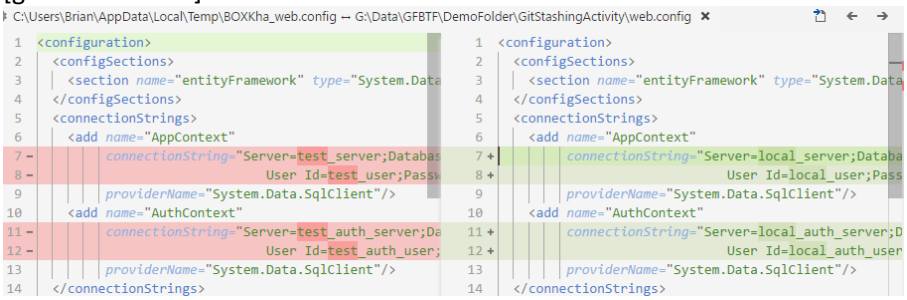
[git stash pop]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash pop
On branch StashingDemo
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:   web.config

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (140277d5052e4e288ee3d2128f2be31de576a97e)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
```

So popping removes from the stash and applies to local (would have to resolve any conflicts of course).

Compare to head to see that changes are as expected:

[git difftool head]



Perfect.

Re-add the stash:

[git stash save web-config-changes]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash save web-config-changes
Saved working directory and index state On StashingDemo: web-config-changes

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list
stash@{0}: On StashingDemo: web-config-changes

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
```

So naming is possible – we can then know which one to get later when we want to put the changes back in place.

d) Get Changes back and leave in stash

[git stash apply]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash apply
On branch StashingDemo
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:   web.config

no changes added to commit (use "git add" and/or "git commit -a")

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list
stash@{0}: On StashingDemo: web-config-changes
```

So getting our config changes back and leaving the changes in stash is possible.

Step 3: Advanced stash operations

We've seen one simple stash, and know how to pop and apply, as well as list. But what happens when there are multiple stash entries and we want to put just one in place? What about untracked files?

a) Stash all changes, including untracked files.

[git reset –hard head]

[touch aNewFile.txt]

[code info.txt]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDe
o)
$ git status
On branch StashingDemo
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:   info.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        aNewFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

[git stash save -u "A new file and modified info.txt"]

[git status]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash save -u "A new file and modified info.txt"
Saved working directory and index state On StashingDemo: A new file and modified
 info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git status
On branch StashingDemo
nothing to commit, working tree clean
```

[git stash list]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list
stash@{0}: On StashingDemo: A new file and modified info.txt
stash@{1}: On StashingDemo: web-config-changes
```

I wish I could just apply from the names I gave, but it's not that easy. Instead, I have to use the refs.

[git stash apply stash@{1}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash apply stash@{1}
On branch StashingDemo
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:   web.config

no changes added to commit (use "git add" and/or "git commit -a")
```

I could re-add to stash and get another entry that is the same as stash@{1}, but that would be pointless. Also note that this is like a stack – Last in = first out. So stash@{0} is always the top of the 'stack', and would be what is popped. As you add more to the stash, the original entries get pushed farther down. This means stash@{1} won't necessarily always be my web.config, which is another important reason to name them.

b) Checking out a branch from stash:

[git reset –hard head]

[git status]

[git stash list]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git reset --hard head
HEAD is now at 852ebe2 Create readme.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git status
On branch StashingDemo
nothing to commit, working tree clean

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash list
stash@{0}: On StashingDemo: A new file and modified info.txt
stash@{1}: On StashingDemo: web-config-changes

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
```

[git stash branch feature-changes]

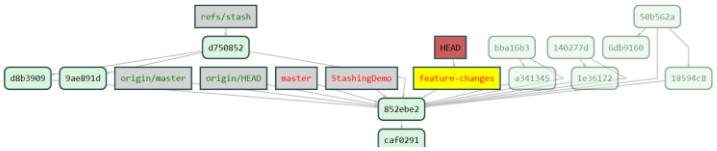
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (StashingDem
o)
$ git stash branch feature-changes
Switched to a new branch 'feature-changes'
On branch feature-changes
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   info.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        aNewFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (50b562a862d884a27b13df8fc5e9e1d30348360e)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash list
stash@{0}: On StashingDemo: web-config-changes
```

Notice that it took the first stash only by default. The branch also changed.

[git stash save -u “A new file and modified info.txt”]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash save -u "A New file and modified info.txt"
Saved working directory and index state On feature-changes: A New file and modif
ied info.txt

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
```

Our repo is getting pretty messy with all this stashing going on…



[git stash list]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash list
stash@{0}: On feature-changes: A New file and modified info.txt
stash@{1}: On StashingDemo: web-config-changes
```

c) Removing from stash without applying or popping

[git stash drop stash@{0}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivi
nges)
$ git stash drop stash@{0}
Dropped stash@{0} (d750852437c510e65c0668c639c8ea415cda2a88)
```

Keep our changes by applying them, then clearing stash

[git stash apply stash@{0}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash apply stash@{0}
On branch feature-changes
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:   web.config

no changes added to commit (use "git add" and/or "git commit -a")

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash list
stash@{0}: On StashingDemo: web-config-changes
```

[git stash clear]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash clear

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash list
```

[git stash save web-config-changes]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash save web-config-changes
Saved working directory and index state On feature-changes: web-config-changes

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/GitStashingActivity (feature-cha
nges)
$ git stash list
stash@{0}: On feature-changes: web-config-changes
```

This concludes our stashing activity.

Closing Thoughts

In this activity we saw how it is possible to stash changes and get them back at a later time.

We also learned how we can save the stash with a name to help us remember what is stashed, and how to apply them while keeping the stash intact.

Stashing is useful for small changesets that need to be applied across multiple branches repeatedly. If the changes are fairly major and/or are specific to a branch, I would recommend just committing the changes to a branch rather than working with stash.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

Section 10 – Using Tags to manage releases

Order and simplification are the first steps towards mastery of a subject – Thomas Mann

Learning:

In this section we'll cover tags, and how to use them to create releases. A tag is nothing more than another ref object, so the great news is that we can checkout branches from a tag. We can also use tags to place a marker in our history for where a release or deployment has taken place. At GitHub, we can formally document a tag as a release.

Goals:

- Understand how to work with annotated and light tags
- Use tags to version our code and create releases

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we will understand how to create and work with tags, including the subtle nuances of pushing and deleting tags to and from the remote repository from our command line.

Activities:

- Tagging

Videos:

- Tagging commits with version and other information

Notes

GIT: From Beginner To Fearless

GIT Tagging Activity:
Using tags to mark releases and specific commits

Brian Gorman, Author/Instructor/Trainer

Introduction

Tags are a great way to place an important milestone on a specific commit in a repository. Often, a release is marked with a tag. Additionally, tags might denote a specific feature implementation or even something like a bug fix.

There are two types of tags: Lightweight and Annotated. The major difference is in how they are stored behind the scenes and what can be displayed from the tag details. Both types are bookmarks to a specific commit, but the annotated tag lists information about the commit and committer, while the lightweight tag is more of just a pointer to a specific comit.

In this activity, we'll take a look at working with tags in our repositories.

Let's gets started!

Step 1: Make sure you have a working repository.

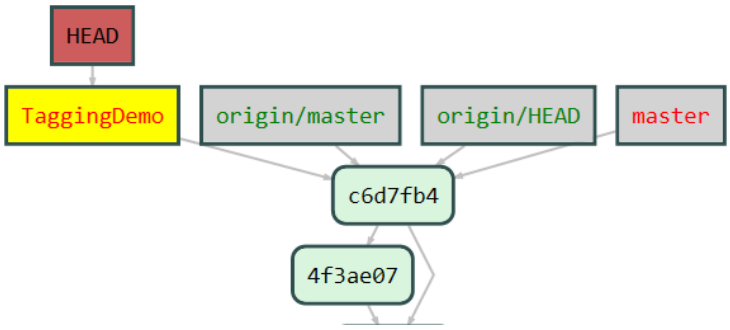
h) Either clone a repo or get the latest on master for a repo. Ideally, the repo would have a few commits in it at least, as well as no tags.

[git clone https://github.com/majorguidancesolutions/SimpleActivityRepo TaggingDemo]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ git clone https://github.com/majorguidancesolutions/SimpleActivityRepo TaggingDemo
Cloning into 'TaggingDemo'...
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 58 (delta 28), reused 47 (delta 17), pack-reused 0
Unpacking objects: 100% (58/58), done.
```

[cd TaggingDemo]
[git checkout master]
[git fetch origin]
[git pull origin master]
[git checkout –b TaggingDemo]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder
$ cd TaggingDemo/

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (master)
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (master)
$ git fetch origin

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (master)
$ git pull origin master
From https://github.com/majorguidancesolutions/SimpleActivityRepo
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (master)
$ git checkout -b TaggingDemo
Switched to a new branch 'TaggingDemo'

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$
```



Notes

Step 2: Commit and tag a few times, list tags

a) Commit and create a lightweight tag, then list the tags on the repo

[code info.txt]
//make a change
[git commit -am “tagging demo commit 1”]

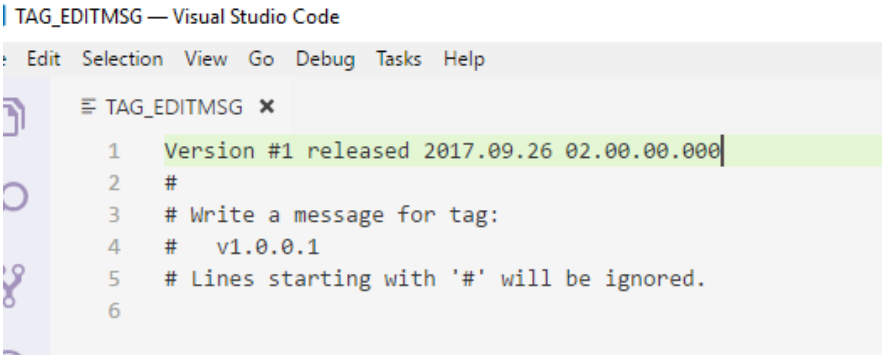
First we’ll add a lightweight tag:
[git tag beginning-tag-demo] //creates a lightweight tag
[git tag] //lists tags
[git tag -l] //lists tags
[git tag --list] //lists tags

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag beginning-tag-demo

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
beginning-tag-demo

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -l
beginning-tag-demo

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag --list
beginning-tag-demo
```

b) Commit and create an annotated tag, then list the tags on the repo

[code info.txt]
//make a change
[git commit –am “tagging demo commit 2”]
[git tag v1.0.0.1]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git commit -am "Tagging demo commit #2"
[TaggingDemo e161f83] Tagging demo commit #2
 1 file changed, 1 insertion(+)

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -a v1.0.0.1
```



[code info.txt]
//make another change
[git commit –am “tagging demo commit 3”]
[git tag -a -m “version 1.0.0.2 released 2017.09.26 02:05:00.000” v1.0.0.2]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -a -m "version 1.0.0.2 released 2017.09.26 02:05:00.000" v1.0.0.2
```

[git tag]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
beginning-tag-demo
v1.0.0.1
v1.0.0.2
```

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git log --oneline
90a9cab (HEAD -> TaggingDemo, tag: v1.0.0.2) Tagging demo commit #3
e161f83 (tag: v1.0.0.1) Tagging demo commit #2
54821bf (tag: beginning-tag-demo) Tagging demo commit #1
c6d7fb4 (origin/master, origin/HEAD, master) Merge pull request #7 from majorgui
dancesolutions/feature-picked-branch
4f3ae07 CherryPickingActivity - commit #2
```

Step 3: Show tag info

Once we have commits tagged, we can actually use the tag just the same as we would use a commit id. This means we can check them out, show them, diff them, etc.

a) Show tag info, see the difference between lightweight and annotated tags:

[git show v1.0.0.2] //annotated

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git show v1.0.0.2
tag v1.0.0.2
Tagger: Brian L. Gorman <blgorman@gmail.com>
Date:   Tue Sep 26 01:59:53 2017 -0500

version 1.0.0.2 released 2017.09.26 02:05:00.000

commit 90a9cab6fe8c7523ef719bcb52b01e35249a50a1 (HEAD -> TaggingDemo, tag: v1.
0.2)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Tue Sep 26 01:57:39 2017 -0500

    Tagging demo commit #3

diff --git a/info.txt b/info.txt
index 67a6f7c..6a3b104 100644
--- a/info.txt
+++ b/info.txt
@@ -23,5 +23,6 @@ Squash and Merge commit #9
 Cherry Pick merge #1
 Cherry Pick merge #2
```

[git show beginning-tag-demo] //lightweight

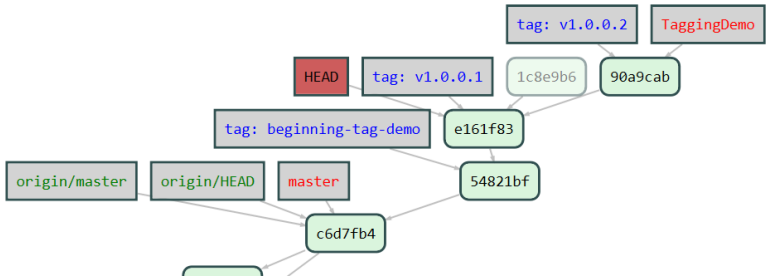
```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git show beginning-tag-demo
commit 54821bf864da49add8feb19a07f31df2e833539f (tag: beginning-tag-demo)
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Tue Sep 26 01:50:35 2017 -0500

    Tagging demo commit #1

diff --git a/info.txt b/info.txt
index 0a7e1d4..abd6b6f 100644
--- a/info.txt
+++ b/info.txt
@@ -22,3 +22,5 @@ Squash and Merge commit #9

 Cherry Pick merge #1
 Cherry Pick merge #2
+
+Tagging Demo 1

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
```

b) Checkout a tag

[git checkout v1.0.0.1]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git checkout v1.0.0.1
Note: checking out 'v1.0.0.1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example

  git checkout -b <new-branch-name>

HEAD is now at e161f83... Tagging demo commit #2

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo ((v1.0.0.1))
```

The checkout happens in a detached head state. When that happens, we can look around and do stuff, but if we want to use it for a commit we need to then checkout a branch and commit on that branch.



Notice the head is pointing to e161f83, where tag for v1.0.0.1 is also pointing.

[git checkout TaggingDemo]

Step 4: Use expressions/wildcards to list specific tags

a) Get all the tags with v1 in the tag

[git tag –l “v1.*”]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -l "v1.*"
v1.0.0.1
v1.0.0.2
```

Step 5: Create tags on previous commits

Need to have some commits in the history. If not enough, create two or three commits so that a couple of them don't have tags. Find a commit without a tag on it

a) Create a lightweight tag on a previous commit

[git log --oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git log --oneline
90a9cab (HEAD -> TaggingDemo, tag: v1.0.0.2) Tagging demo commit #3
e161f83 (tag: v1.0.0.1) Tagging demo commit #2
54821bf (tag: beginning-tag-demo) Tagging demo commit #1
c6d7fb4 (origin/master, origin/HEAD, master) Merge pull request #7 from majorgui
dancesolutions/feature-picked-branch
4f3ae07 CherryPickingActivity - commit #2
59a4cde Update readme.txt
3fd06fc Squash and merge feature2 (#6)
fa75127 Merge pull request #5 from majorguidancesolutions/SquashAndMergeFeature
3741e4f A single new commit on feature
728f97e Merge branch 'master' into SquashAndMergeFeature
5449a4a Squash and merge feature (#4)
729bc24 Squash And Merge Commit#4
c2d0c0d Squash And Merge Commit#3
a0b8ada Squash And Merge Commit#2
7ab5f41 Squash And Merge Commit#1
1a3444a Merge pull request #3 from majorguidancesolutions/GitAmendDemo
0cac76b Changed info.txt and added readme.txt
```

Here there are plenty of candidates. I'm going to put a lightweight tag on 728f97e for squash and merge completed

[git tag squash-and-merge-completed 728f97e]
[git tag]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag squash-and-merge-completed 728f97e

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
beginning-tag-demo
squash-and-merge-completed
v1.0.0.1
v1.0.0.2
```

[git log --oneline]

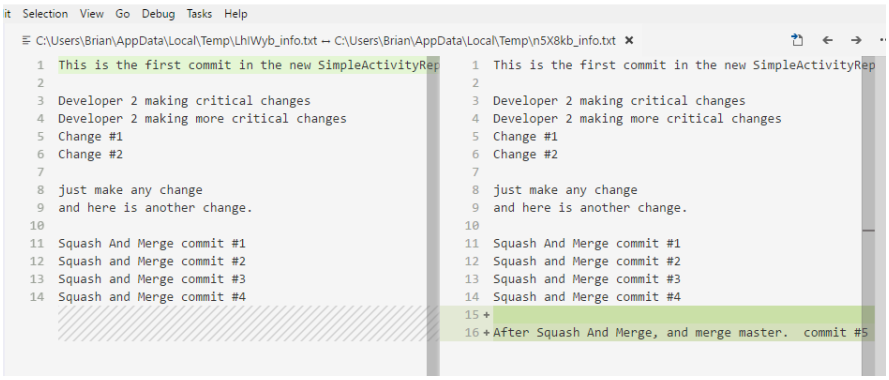
```
3fd06fc Squash and merge feature2 (#6)
fa75127 Merge pull request #5 from majorguidancesolutions/Squash
3741e4f A single new commit on feature
728f97e (tag: squash-and-merge-completed) Merge branch 'master'
rgeFeature
5449a4a Squash and merge feature (#4)
729bc24 Squash And Merge Commit#4
```

[git show squash-and-merge-completed]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git show squash-and-merge-completed
commit 728f97ed574a90737bd6b6a4ed51ca3d3b995e6b (tag: squash-and-merge-completed
)
Merge: 729bc24 5449a4a
Author: Brian L. Gorman <blgorman@gmail.com>
Date:   Mon Sep 25 00:42:51 2017 -0500

    Merge branch 'master' into SquashAndMergeFeature
```

[git difftool squash-and-merge-completed 3741e4f]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git difftool squash-and-merge-completed 3741e4f
```



b) Create an annotated tag on a previous commit

[git log –oneline]

```
54821bf (tag: beginning-tag-demo) Tagging demo commit #1
c6d7fb4 (origin/master, origin/HEAD, master) Merge pull request #7 from majorgui
dancesolutions/feature-picked-branch
4f3ae07 CherryPickingActivity - commit #2
59a4cde Update readme.txt
3fd06fc Squash and merge feature2 (#6)
fa75127 Merge pull request #5 from majorguidancesolutions/SquashAndMergeFeature
3741e4f A single new commit on feature
728f97e (tag: squash-and-merge-completed) Merge branch 'master' into SquashAndMe
rgeFeature
5449a4a Squash and merge feature (#4)
729bc24 Squash And Merge Commit#4
```

How about fa75127 this time.

[git tag -a -m “Code Review Completed” code-review-completed fa75127]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -a -m "Code Review Completed" code-review-completed fa75127
```

[git tag]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
beginning-tag-demo
code-review-completed
squash-and-merge-completed
v1.0.0.1
v1.0.0.2
```

[git show code-review-completed]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git show code-review-completed
tag code-review-completed
Tagger: Brian L. Gorman <blgorman@gmail.com>
Date:   Tue Sep 26 02:25:48 2017 -0500

Code Review Completed

commit fa751272965cd550ac657226e70a47a180cda3b0 (tag: code-review-completed)
Merge: 5449a4a 3741e4f
Author: majorguidancesolutions <brian@majorguidancesolutions.com>
Date:   Mon Sep 25 00:57:05 2017 -0500

    Merge pull request #5 from majorguidancesolutions/SquashAndMergeFeature

    Squash and merge feature
```

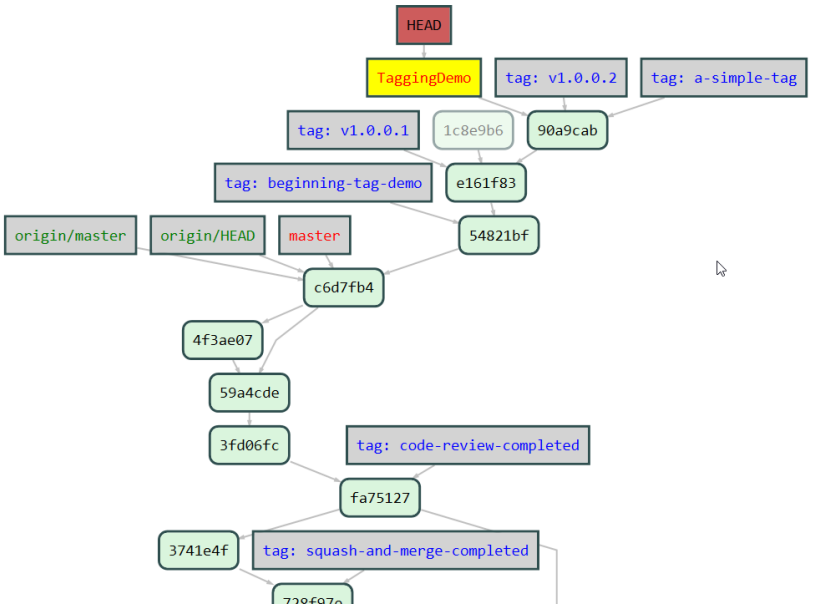
Step 6: Delete a tag locally

a) Create a simple tag then delete it

[git tag a-simple-tag]

[git tag]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag a-simple-tag

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
a-simple-tag
beginning-tag-demo
code-review-completed
squash-and-merge-completed
v1.0.0.1
v1.0.0.2
```



[git log –oneline]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git log --oneline
90a9cab (HEAD -> TaggingDemo, tag: v1.0.0.2, tag: a-simple-tag) Tagging demo com
mit #3
e161f83 (tag: v1.0.0.1) Tagging demo commit #2
54821bf (tag: beginning-tag-demo) Tagging demo commit #1
c6d7fb4 (origin/master, origin/HEAD, master) Merge pull request #7 from majorgui
dancesolutions/feature-picked-branch
```

Looks like we have two tags on commit 90a9cab now…

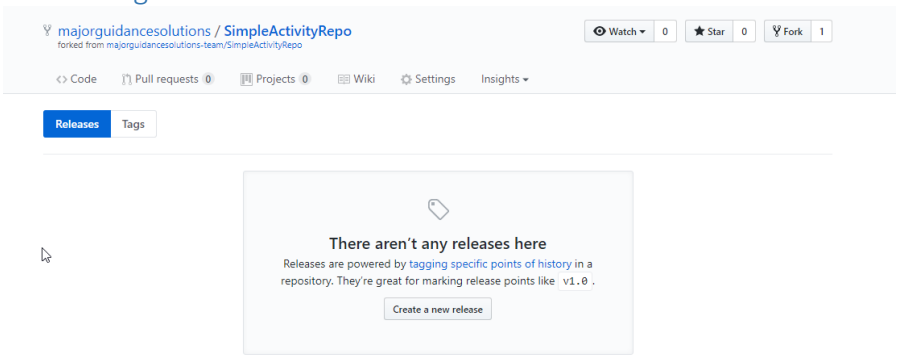
[git tag –d a-simple-tag]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -d a-simple-tag
Deleted tag 'a-simple-tag' (was 90a9cab)
```

Step 7: Working with Tags at GitHub

We need to be able to push our tags, as well as delete tags that are pushed.

a) Push all tags to GitHub

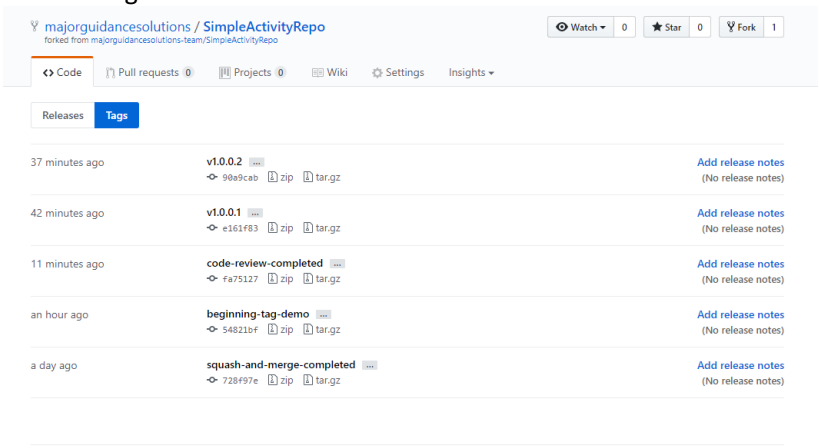


Right now, there are 0 releases at GitHub, and 0 tags. Note that we can create a tag right at GitHub with the button above, "Create a new release" If we did this, we could then get our local repository up to date with tags from REMOTE with [git fetch --tags]

However, we aren't going to worry about that. Instead, let's push our tags [git push --tags]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git push --tags
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 1.26 KiB | 0 bytes/s, done.
Total 12 (delta 6), reused 0 (delta 0)
remote: Resolving deltas: 100% (6/6), completed with 2 local objects.
To https://github.com/majorguidancesolutions/SimpleActivityRepo
 * [new tag]         beginning-tag-demo -> beginning-tag-demo
 * [new tag]         code-review-completed -> code-review-completed
 * [new tag]         squash-and-merge-completed -> squash-and-merge-completed
 * [new tag]         v1.0.0.1 -> v1.0.0.1
 * [new tag]         v1.0.0.2 -> v1.0.0.2

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$
```

And looking at GitHub:



The cool thing to note is that you can get a download of the repo at any of the release points.

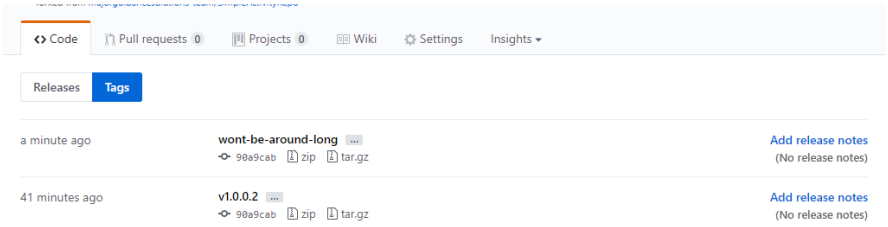
b) Set config to always push tags

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git config --global push.followTags true
```

c) Delete a tag from remote

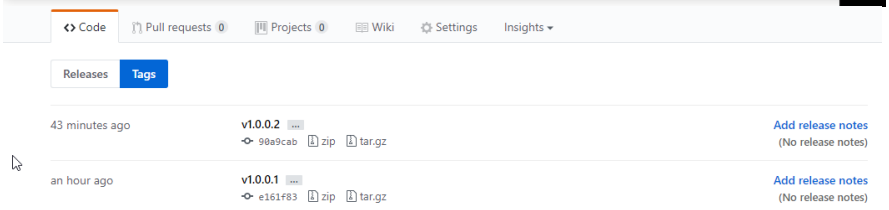
[git tag -a -m “This Tag is going to go away” wont-be-around-long]
[git push --tags]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -a -m "This Tag is going to go away" wont-be-around-long

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git push --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 187 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/majorguidancesolutions/SimpleActivityRepo
 * [new tag]         wont-be-around-long -> wont-be-around-long

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$
```



[git push origin :wont-be-around-long]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git push origin :wont-be-around-long
To https://github.com/majorguidancesolutions/SimpleActivityRepo
 - [deleted]         wont-be-around-long

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$
```



Even though we removed from origin, we still need to delete locally
[git tag]
[git tag –d wont-be-around-long]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag
beginning-tag-demo
code-review-completed
squash-and-merge-completed
v1.0.0.1
v1.0.0.2
wont-be-around-long

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DemoFolder/TaggingDemo (TaggingDemo)
$ git tag -d wont-be-around-long
Deleted tag 'wont-be-around-long' (was 8ed11c7)
```

This concludes our tagging activity.

Closing Thoughts

In this activity, we took a look at creating tags on our repository. There are two different types of tags, annotated and lightweight. Both can be useful, but if we want to tag a major release for public knowledge we should use the more verbose annotated tag. The lightweight tag is great for private use or simple pointers to commits along the way.

Just like a commit id in GIT, a tag can be interacted with to get information about the commit, differences between commits, and even checked out to a branch for further development.

Adding and getting tags from the public repo requires using push and pull with the --tags flag. Deleting from a public repository is much like deleting a branch from a public repository, by pushing with a : (colon) in front of the tag name.

The really cool thing at GitHub is that tags allow us to be download the repo as it was at the state of that tag directly for release/deploy.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes

Section 11 – Workflows

Order and simplification are the first steps towards mastery of a subject

Learning:

Outside of the normal team flows we've been studying are two other important workflows. These workflows are more important when there are distributed teams and/or there are segregated sections of the code. Even so, they can be done in any situation.

Goals:

- Have a general understanding of the Integration Manager workflow
- Understand working with the Dictator and Lieutenant workflow

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

At the completion of this section, we'll understand that there are different workflows that work well when using GIT. Our basic team doesn't have to go to a lot of effort to protect the code, but a large team (like a Microsoft, for example) will likely need a more robust solution. Having studied these different workflows, we're in a great position to help any team discover the best path to their solution.

Activities:

- None

Videos:

- None

Notes

Section 12 – Working with GIT and GitHub from Visual Studio

There are only two mistakes one can make along the road to truth; not going all the way, and not starting. - Buddha

Learning:

Any IDE you are using likely has some GIT integration. In this section we'll look at a couple of the commands we've already learned to see how we can perform them in visual studio. We'll also see a couple of the pitfalls for working with GIT in Visual Studio and how to overcome them.

Goals:

- Become fluent with GIT in visual studio
- See how we can recover from a couple of issues with the Visual Studio git solution.

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

Understand that even though the command line exists, it is not always necessary to use. Instead, we can use the built-in tools of our editor (Visual Studio). In some cases, we'll have to recover with a simple command to the command line.

Activities:

- None

Videos:

- None

Notes

Section 13 – Conclusion

End? No, the journey doesn't end here. Death is just another path. One that we all must take – J. R. R. Tolkien

Learning:

There is no learning in this section; this section will a review of concepts already learned, followed by a brief plan of attack to move forward.

Goals:

- Review our learning
- Reflect on our missing fear

Tasks:

- Watch the videos for section seven
- Complete the activities for each video, don't just watch them
- Take notes and/or make reminders for yourself for the future.
- Practice the steps that give you trouble more than once

Achievements:

Completion is the only achievement here. Great job to everyone@

Activities:

- None

Videos:

- None

Notes

APPENDIX A

Connect with me and other offerings

These courses are older so the quality may not be as good as what you experienced in this course

Please visit: https://courses.majorguidancesolutions.com/

To review and see other offerings that I have made available.

LinkedIn: https://www.linkedin.com/in/brianlgorman/

Twitter: https://twitter.com/blgorman