

# Gitting git

Derek Hunter

derek.hunter@aggiemail.usu.edu

# What is it?

- “Initial revision of "git", the information manager from hell” -Linus Torvalds, when he first committed git into a git repo.
- Version control system created to aid the development of the Linux Kernel.
- Basically a database that tracks changes to a folder.

# Getting started

- Install git
- Create a git repo (locally or on github)
- If local push the repo else clone the repo
- Start working

# States of a file

- Created but untracked
  - This is when you first create a file but you haven't told git to keep an eye on it
- Tracked but not committed
  - This is after you have told git to look after the file, Like adding a column to a database.
- Committed but not pushed
  - This when you have told git to track exact changes to a file. Like adding a row to the database.
- Pushed
  - This sends your changes to github (or other git host).

# Branching

- A `Branch` is a separate line of commits separate from the main branch.
  - This is like creating a new table in the database.
- A `Merge` is when you pull the changes of one branch into another.
  - Merging two branches is like unioning the two sets of changes.
  - Specifically it's like unioning the two last rows from the tables.
- A `Pull Request` is a request to merge some branch A into some branch B
  - This is really a concept created by github and other git hosts, it is not a git specific concept.

# DEMO

Whoops I forgot to press record =(

# DISCLAIMER

This introduction assumes that you will be using github, as your remote repository source. **This is not a requirement.** There is no reason you have to use github with git. There are many git repository hosting services, but they all function about the same. There's also nothing that says you have to use git with a remote repository host at all. You could keep everything local.

# First step.

- The first step when starting any project that you want to be tracked by git is to make the repository, and get a copy on github, and locally.
- Instructions on how to create the repository on github first can be found on the right

<https://help.github.com/articles/create-a-repo/>

After you have your repository created on github, we need to clone it to get it on our local machine. Go to the repository page, and click `clone` and copy the url that shows up.

Note: make sure the url begins with https. There's nothing wrong with using ssh. Setting it up is a little out of the scope of this tutorial, more information can be found here:

<https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/>



# Cloning

- Now, that we have the url to the repository, it's time to clone.
- 

In your commandline client, navigate to where you want to store your git repository, and type

```
git clone <url_to_git_repo>
```

Example:

```
git clone https://github.com/DerekHunter/MyRepo.git
```

You should see a message similar to the following:

```
Cloning into 'MyRepo'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

# New Files and Git Status

For this part, we need to create some files. I'm going to make some junk files, but when working on an actual project, these new files will be your source files.

Once we have new files, we'll tell git about them, and get them up onto github to share with the world.

For this part I'm going to create a new file called `newFile.txt` and store the text `Hello World!` into it.

Now to see that git is aware this new file existed we type ``git status`` which produces output similar to the following

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

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

`newFile.txt`

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

This tells us git know about the file ``newFile.txt``, but it's in the untracked files sections, so it's not tracking changes made to it.

# Staging Files

Now that we have a new file created, we need to tell git to keep track of it for us. We do this by adding it to the staging area, then committing it.

To add the file to the staging area, we need to say

```
git add newFile.txt
```

We won't get any output so to verify git has done what we want we need to say `git status` and again we get output that looks like the following

```
On branch master
```

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

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file:   newFile.txt
```

# Committing

Now that our file is staged we need to commit it.

Note: you can stage multiple files for one commit. Git will take everything in the staging area, and build a commit based on just those files.

Note2: The -m flag allows us to add a commit message right there, if we leave that off it will open git's default editor (vi unless otherwise specified) information about changing git's default editors can be found here:

<https://git-scm.com/book/tr/v2/Customizing-Git-Git-Configuration>

To commit the changes in the staging area, we say

```
git commit -m "Commit message"
```

Example:

```
git commit -m "Adding newFile.txt"
```

From here we should get output similar to

```
[master bdc1015] Adding newFile.txt  
1 file changed, 1 insertion(+)  
create mode 100644 newFile.txt
```

# Quick Check

From here we can verify that the commit was made by saying `git status` again, and looking more closely at some of the output

The main thing to notice is we no longer have a list of staged files.

The next thing to notice is that our branch is ahead of our remote branch by 1 commit by looking at this line of output

```
Your branch is ahead of 'origin/master' by 1 commit.
```

Note: If you had any modified or unstaged files they will still show up in their respective categories for `git status`

# Pushing to Github

Now that we've made a commit let's get it onto github.

To do this we say

```
git push origin master
```

This will produce output similar to

```
Counting objects: 3, done.
```

```
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (3/3), 304 bytes | 0  
bytes/s, done.
```




```
Total 3 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/DerekHunter/MyRepo.git  
f96b329..bdc1015 master -> master
```

This output is not incredibly useful other than it didn't say we had any errors.

# Verifying

From here we can now go back onto our repo page on github and verify the changes were pushed. As you can see from the image below, it has been pushed properly.

 <b>DerekHunter</b> Adding newFile.txt		Latest commit bdc1015 7 minutes ago
 <a href="#">README.md</a>	Initial commit	31 minutes ago
 <a href="#">newFile.txt</a>	Adding newFile.txt	7 minutes ago

# Branching Intro

Awesome! Now that we know how to get stuff from our machine to github, it's time to talk about branching.

So, what is branching? So far, we have committed everything to master, and when we generate commits on a branch, there's some meta data about the ordering of those commits. So If I start on commit A, and change some files and make commit B, commit B has a reference to commit A saying "That's where I came from" and if we make commit C from commit B, commit C has a reference to commit B saying "I came from commit B". Branching is when you have two commits that reference the same base commit.

So If commit A is our base commit, and we make commit B from commit A. We've just continued master, but if we also make commit C from commit A, that's creating a new branch.



# Why Branching.

When working in an environment with multiple developers branching is an absolute must have. It helps keep work between developers separate while they're in progress.

For example, let's say that developer A is working on feature x that depends on feature y, and developer B is working on making feature y more efficient. Developer B could go completely break feature y while he's rewriting it, and it work affect developer A. If they these two developers didn't create branches for their work, then when developer B breaks feature y, developer A now has to wait for developer B to finish what's he's doing to continue work on feature x.

# Looking at branches

When the branches exist on the github copy of the repo, there's a handy tool on the github website for looking at them.

This tool can be found by going to the main repo page, clicking on **graphs**, then **network**. And you'll get a screen that looks similar to the one below.



As you can see to the left, we have one branch master, with two commits.

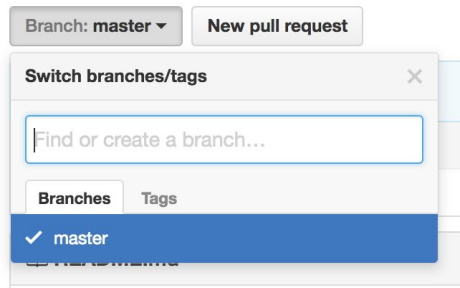
Note: each commit is represented with a dot.

# Creating a branch on github.

There are two ways to create a branch. You can create it locally, and push it to github, or you can create it on github, and download it. I'm only going to show you how to do it on github, but more information about how to do it locally can be found here

<https://git-scm.com/book/en/v1/Git-Branching>

To create a new branch on github go to the main repository page, and look for the button that says Branch: <branch\_name>, and click on it.



The search box serves two purposes, it lets you filter current branches, or create a new one. So let's type in `FirstBranch` and press enter to create the new branch.

# Branching cont.

Now that we have a branch created on github, it's time to get that to our local copy. To do this we need to introduce a new term

The new tool we need is `git fetch` what this command does it tells git to go to github, and grab a copy of the git database that's stored there and download and save the changes.

Note: this does not change your current workspace, just the underlying git database

To get the new branch from github, we simply say

```
git fetch
```

And this will produce output similar to the following:

```
From https://github.com/DerekHunter/MyRepo
* [new branch]      FirstBranch -> origin/FirstBranch
```

What this says, is that it there was a new branch, and git downloaded it, and set it up to 'track' the version that's on github.

# Git Checkout

If you type `git status`, you'll notice that it still says something similar to 'on branch master'.

This is because all we've done is downloaded the underlying database but we haven't changed the workspace or anything.

So we need to switch branches.

To switch branches we say

```
git checkout <branch_name>
```

Example: `git checkout FirstBranch`

And we get output similar to the following:

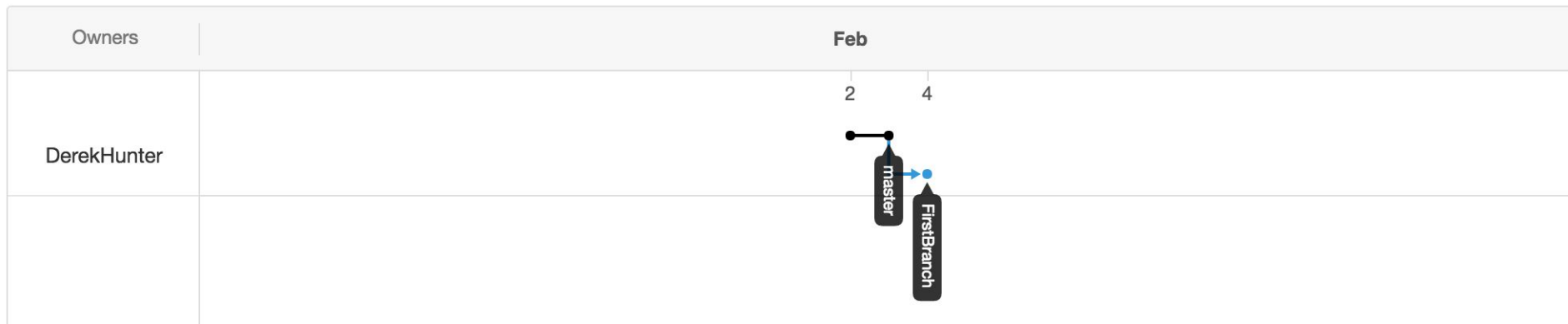
```
Branch FirstBranch set up to track remote  
branch FirstBranch from origin.  
Switched to a new branch 'FirstBranch'
```

Typing `git status`, should now say on branch `origin/FirstBranch`

# Adding a Commit to the New branch

Now that we've got a branch it's time to start working. To mimic this I'm just going to add the text `git is an awesome tool` to line 2 of `newFile.txt` and commit, and push the changes. This is done the same way outlined earlier. Make changes, stage files, commit, and push.

Now if we look at the network page on github we'll see two branches. With some commits on each.



# Making a similar change on master.

Now, let's switch back to master for a minute and add the text `git is an awful tool` to to line 2 of `newFile.txt`.

Just like before we say `git checkout master` this should give that familiar message saying it switched to master.

Then add the line `git is an awful tool` to line 2 of `newfile.txt`, and commit and push.

# Merging

Merging is the concept of bringing changes from one branch into another. It looks at the current state of both branches, and does a union of the line changes.

Typically, someone will have changed line 1 in file A and someone else will have changed line 15 of File B. When this happens the merge just applies both of those changes.

If however, line 1 in file A gets changed on one branch, and line 1 of file B gets changed on another branch, and you try to merge you have to tell git which change you want to keep. This is called a merge conflict.

Note: We have purposefully created a merge conflict by changing line 2 of newFile.txt on FirstBranch to ``git is an awesome tool``, and changing line2 on newFile.txt on master to ``git is an awful tool``.

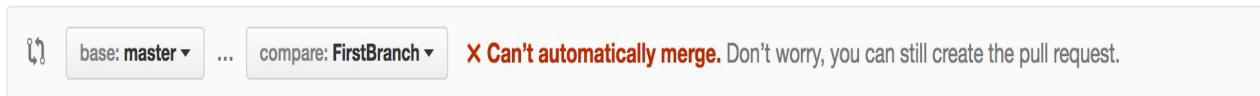


# Pull requests

Pull requests are what make Github awesome. They're a nice web ui for submitting changes to a repository. They're a request to merge branch B into branch A. Typically, this is merging a feature branch into master.

So let's submit a pull request for our new branch.

Go to the main repo page, and click the button New Pull Request. You'll get something similar to the following



the left branch. In this case we want to merge `FirstBranch` into `master`. Ignore the bit about 'Can't automatically merge' for a minute, it's caused by the merge conflict we intentionally created. Typically you would write a description about the merge, like 'This merge adds functionality for x'. But we don't care so much about that right now.

Click the button 'Create pull request'

# Open Pull Requests

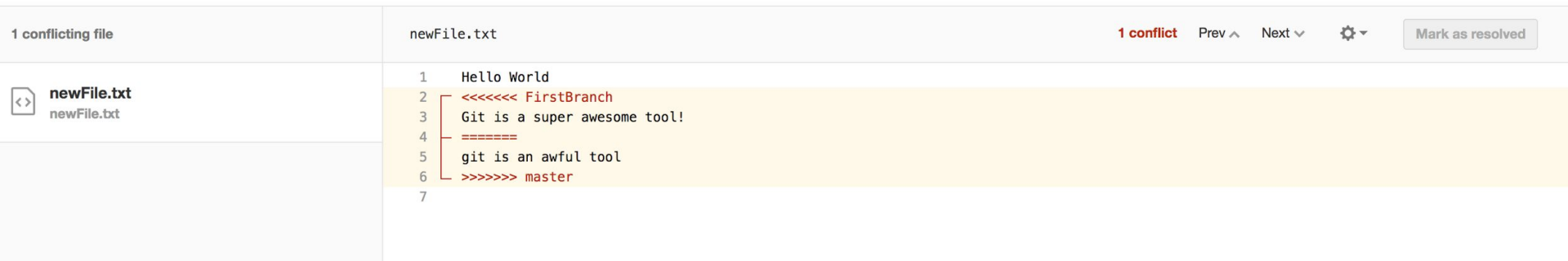
On this screen, this is where different developers can look at the changes made and write comments about whether or not the change should be merged. Or if things need to get fixed.

Once this review processes is done and the branch is okay to merge, and there are no conflicts. There will be a green button near the bottom that says merge.

This button does not exist because we have a merge conflict.

# Resolving merge conflicts

Github has a new web based UI for resolving merge conflicts. You can get there by click `Resolve Conflicts`. You should see a screen similar to the following



The screenshot shows the GitHub web interface for resolving merge conflicts. On the left, a sidebar indicates '1 conflicting file' and lists 'newFile.txt'. The main area displays the file's content with a conflict highlighted in yellow. The conflict occurs between line 2 and line 6. Line 2 contains '<<<<<<< FirstBranch' and line 6 contains '>>>>>>> master'. The text between these lines is 'Git is a super awesome tool!' followed by a separator line '=====' and then 'git is an awful tool'. The interface includes navigation controls like 'Prev', 'Next', and a 'Mark as resolved' button.

```
1 Hello World
2 <<<<<<< FirstBranch
3 Git is a super awesome tool!
4 =====
5 git is an awful tool
6 >>>>>> master
7
```

On the left are the files with merge conflicts(only one), and on the right it shows the contents of the file, as well as some weird lines that look like `<<<<<<< FirstBranch`

# Format of Merge Conflicts

Those funny tags take on the format

```
<<<<<<<< BranchA
```

```
<change_that_happened_on_branch_a>
```

```
=====
```

```
<change_that_happened_on_branch_B>
```

```
<<<<<<<< BranchB
```

# Resolving Merge conflict pt2

All we need to do. Is pick the line change we want, and deleted everything else. Obviously git is a super awesome tool, so we pick the change that says that.

So in our specific example

```
<<<<<< FirstBranch  
Git is a super awesome tool!  
=====  
git is an awful tool  
>>>>>> master
```


Turns into



Git is a super awesome tool!

# Finalizing

The final result of the merge conflict is the following

1 conflicting file	newFile.txt	1 conflict	Prev ^	Next v	⚙	Mark as resolved
 newFile.txt newFile.txt	<pre>1 Hello World 2 Git is a super awesome tool!</pre>					


Once all our conflicts are resolved, we click `Mark as Resolved`.


Then we click `Commit Changes`

# Closing the pull request.


Now that our conflict is resolved it will take us back to the pull request page. From here we should have the green button that says `Merge Pull Request`.

Add more commits by pushing to the `feature` branch on `yourname/myrepo`.





**This branch has no conflicts with the base branch**  
Merging can be performed automatically.

**Merge pull request** 

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Click merge pull request, and optionally add a message when prompted. Then click `Confirm Merge`.

From here, depending on how branching style for your organization you can delete the branch right there, or not. We're going to delete it.

# Pulling Changes

Now that we have successfully merged, FirstBranch into Master. We need to get those changes locally. We do that by saying `git pull`.

Git pull does two things. It performs

`git fetch` then `git merge`

`git merge`, is the underlying command used by github in pull requests. However, this is a special merge that git pull performs. Instead of saying merge branch B into branch A.

It says merge remoteBranchA into localBranchA. So it has the effect of just grabbing what is on github for than branch and applying those changes to current branch.

Note: This DOES change your local workspace.



# Pulling

So now we need to pull that merge to our local copy of master.

Make sure you're on branch master. Then say

```
git pull
```

You should get output similar to the following

```
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 4 (delta 1), reused 0 (delta 0),  
pack-reused 0  
Unpacking objects: 100% (4/4), done.  
From https://github.com/DerekHunter/MyRepo  
6f66d47..0df82a2 master -> origin/master  
Updating 6f66d47..0df82a2  
Fast-forward  
newFile.txt | 2 +-  
1 file changed, 1 insertion(+), 1 deletion(-)
```

All this says, is that there was 2 changes to newFile.txt

# Checking

Now that we've pulled lets open newFile.txt to verify that it has the contents we chose for the pull requests. You should see the contents below in that file.

```
Hello World
```

```
Git is a super awesome tool!
```

# Conclusion

There's no way I could include everything about git in these slides. I really just scratched the surface. But they should give you enough background to know what questions to ask.

In the next few slides, are some notes and resources to help you be successful using git.

And, as with any new tool. You're going to be bad. You're probably going to be really bad. You'll probably break your repo, end up on random commits, and not know how to fix it, but that's okay, because everytime that happens you'll learn more.

# Good workflow

Github has prepared a really useful starter workflow. I would recommend following something similar to this. Even if you're one person, it will help get you comfortable with branching and merging.

<https://guides.github.com/introduction/flow/>

# Resources

- <https://githowto.com/> ← Very good super basic introduction to git
- <https://git-scm.com/doc> ← Some fantastic official git resources
- <https://help.github.com/> ← Github also has some incredibly useful resources for new users. This is where you should look for github specifics
- <https://git-scm.com/book/en/v2> ← The book on git. There's a whole lot of esoteric information in this book about how git works under the hood. Like, how it calculates the differences, and how it stores changes. Not for the faint of heart.