Branches & Merging

Class 4 of 5

Welcome back.

Hi everyone, welcome back – today we'll be talking about branching and merging – super important; you can't collaborate without understanding either of these things. Once you're familiar you will appreciate the flexibility git gives you even more.

Just a reminder – not a whole lot of you took the survey – please do this, it's super quick. Also very few of you posted a URL of your forked repo to the forum. I hope that doesn't mean that you're confused – either way, take the survey – I have left the 2 questions there very open ended, so anything goes.

Branches & Merging: what we'll cover

- Review: what is a branch?
- A look at merging
- Merge conflicts
- Misc: remote review, fork updates, workflows
- For next week: reading & practice

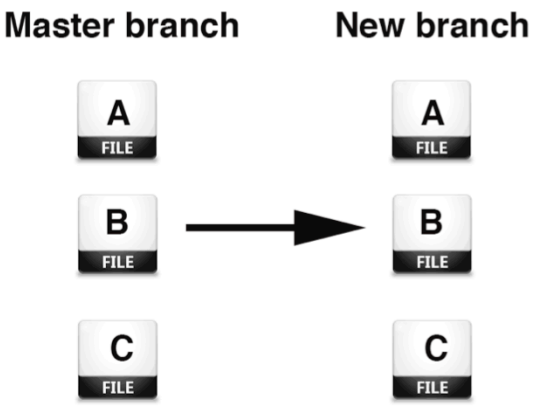
We'll look more deeply at branches, I'll do a demo of how they work. We'll talk about merging, I'll show you what happens when you have a merge conflict and explain what that is in the first place. We'll look a bit at working with a team and if time I can cover some miscellaneous features about checking out and forks because some folks asked about that in the survey.

Branch

A copy of the repository that lives within your project

As we have seen before, a branch is a copy of your repository. It lives in your project. When you start a git project, you are on the master branch by default. You can create new branches, and if needed push those branches up to GitHub or to other external hosts.

Branch



When you create a new branch, all your files and project history (all the commits, in other words) are copied over to the new branch.

Create a new branch

Git branch *branch_name*

Git checkout *branch_name*

OR

Git checkout -b *branch_name*

There are a couple of ways to create a new branch. You can create a new branch with the branch command, but you wont be moved to the new branch. You move between branches by the git checkout command. You are only on one branch at a time in git, and this is the branch your computer sees.
An easier way to create a branch and check it out is to do it all in one command, with this git checkout –b command

Create a new branch

DEMO

See demo at 10:50 – 11:46

The “git branch” command

- Create a new branch
  - **Git branch *new_name***
    - Still have to use *git checkout new_name* to work on it.
- Show the branches available
  - “**Git branch**” on it’s own = list of branches on local
  - “**Git branch -v**” = list of branches with commit note & hash
  - “**Git branch -a**” = local + remote
  - “**Git branch -av**” = local + remote w/ commit note & hash

Running git branch can be a way of showing branches, creating branches, or deleting branches.
The flags a & v can be helpful to see more info. V shows the URL; a shows all available branches the repository knows about. This will be helpful for dealing with remote branches.

Move to a branch
Again, the way you move between branches is to check them out. You will be doing this a lot.

The “git branch” command

- Delete a branch
  - **git branch -D** *branch_name*
- Delete a branch on remote
  - **Git push** *origin* **--delete** *branch_name*

Demo: 15:00 – 18:00 (includes deleting a local branch)

Move to a branch

- Move around branches by the git checkout command
  - **git checkout** branch_name

Branches on GitHub

- You have multiple branches on GitHub
  - To pull down an additional branch, run:
    - Git branch -a to see the branches
    - Git checkout branchname automatically brings down the branch from GitHub and checks it out on local for you
- Push up/pull down branches to local

You can have multiple branches on Github as well. It doesn't have to be the same branches you have on local. There are good reasons why you might want to push up an experimental or test branch, and then be able to test your code out in new environments or have other people work on the branch with you.

Demo of checking out a remote branch: 19:10 – 21. Also includes demo of git diff between branches.

Demo of moving between branches and how the computer sees only one branch at a time: 22 – 26. When Kate says "you won't see anything different on your computer" she means you won't see a reference to different branches (unless you issue a git command like status or branch), but your computer will only see how the files look on whatever branch is checkout out at that time.

Merging branches

- Merging is how you move work from one branch to another
  - Git compares each file in each branch looking for possible conflicts

It's a standard convention to have a branch that you mark as the main branch, where everything works great (a lot of people use the master branch as this branch), and then work on new development on branches. Let's say you have created a new branch, did some work on it, it looks great, and you've decided that you'd like this work to be a part of your project for the long run. How do you get it to the master branch? You ask git to merge the branches.

Merges in git are relatively easy and fast, especially compared to other version control systems. Behind the scenes, git compares the current branch, the one getting merged in, and will add in the most recent changes, unless a file/line has changed in the 2 branches at the same time – then you get a merge conflict, which we will see in a bit. For now, let's just look closer at merging itself, because while it's true, you will eventually have to deal with merge conflicts, most merges you will make will have no problems.

Ability to merge = better collaboration

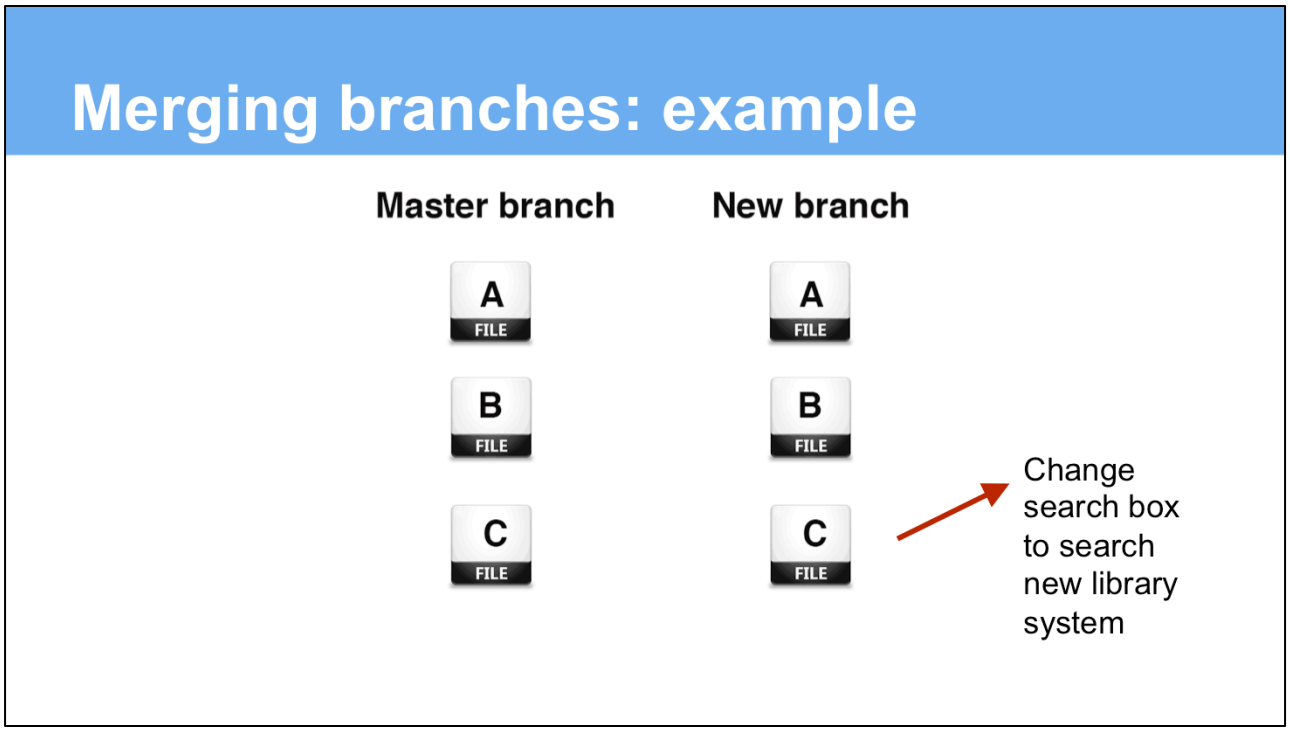
Merging allows collaboration

Pushing and pulling to GitHub - Git is merging remote branches with local ones

Merging is where git and other decentralized version control systems excel compared to centralized (like subversion) – with git you can merge on your local computer and on some place like GitHub, doesn't have to only be on one centralized server.

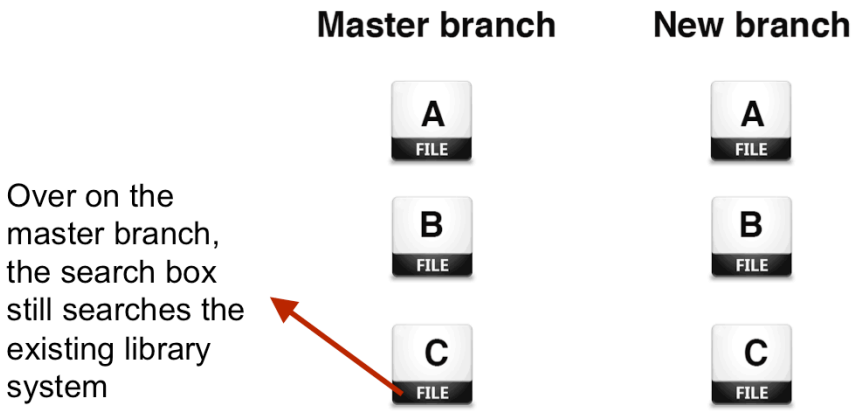
This act of merging is also behind working with remotes, even if you aren't explicitly calling the git merge command when you push & pull, git is doing merges to combine 2 copies of a repository, whether it's 2 different branches you made on your local computer, or if it's comparing the local branch you are pushing up to origin with GitHub's existing copy of that branch.

Merging branches: example



Lets say I need to change a search box on my library website to search a new library catalog system. I don' t want to break the existing site, so I checkout a branch. Now, I'm working on local, so it's not like my local website is going to effect users. But let's say as I'm working on this search box I get a request to make a new form – if I were to push up the project now when I am working on my search box, then the website were broken. So with branches, I can just let that experimental code for changing the search box on a branch, and either work on the new form on the master branch or check out a third branch to work on the form.

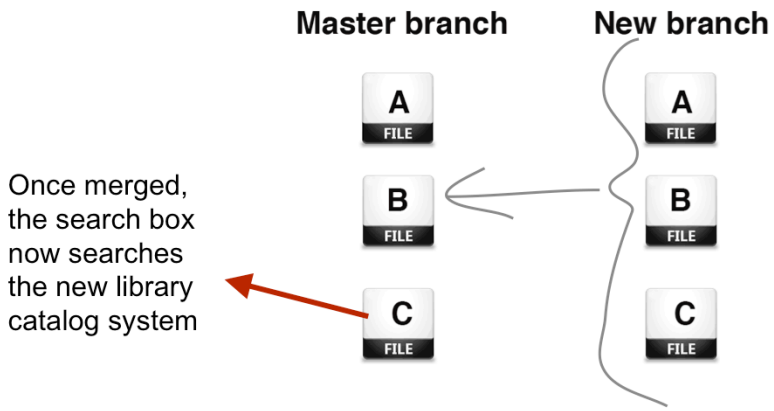
Merging branches



Over on the master branch, the search box still searches the existing library system

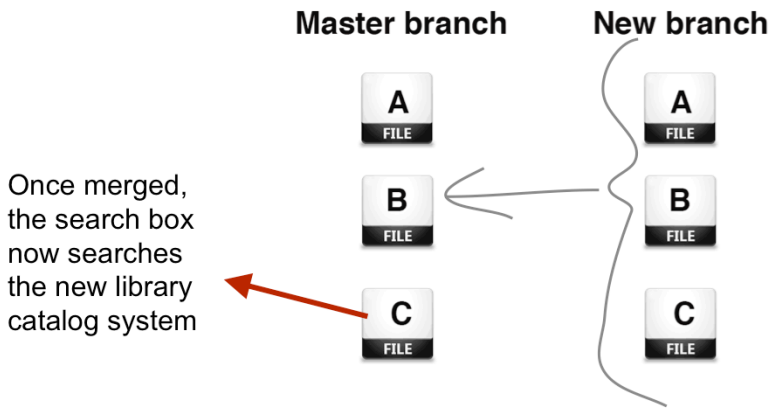
Either way, back on master, I know I have a search box that works.

Merging branches



I finish the library search box for the new system, it works fine – so when its time to switch to the new catalog, I can merge in my branch

Merging branches



For this merge to happen, I checked out master, and merged new branch into it.

In order to merge in the new branch, I need to actually move to the branch I want to merge it to. So I checkout master, and then run the git merge new_branch command

Merging branches

Demo

Demo of merging branches: 32:48 – 34:20

Merge Conflicts

When Git doesn't know which change to prioritize when merging, it flags a "merge conflict"

It will happen to you - especially if you're collaborating with others.

Dealing with merge conflicts

Fix in text editor, or use a tool. Also, merge often so you can fix a few at a time instead of all at once.

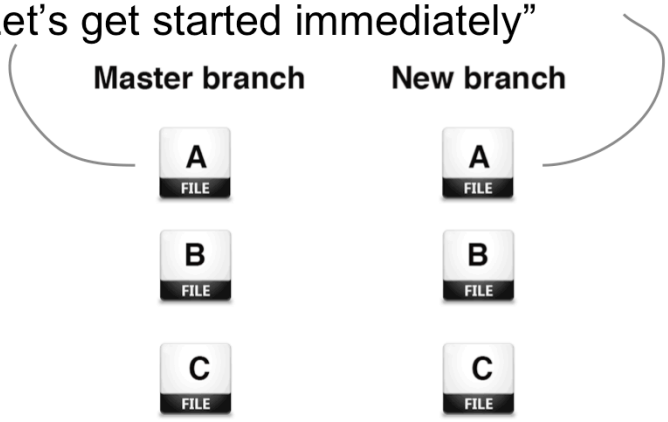
Merge conflicts: this will happen to you. You're merging in a branch, and git complains about conflicts.
You have to fix them before git will let you finish the merge.
The conflict is that there are changes where git doesn't understand which one should go ahead.
In the previous example, let's say that when I was changing the new branch to search a new library catalog, someone came in and changed the master branch to have an additional change in the same lines- git won't know what to do with both new changes to the file. So I have to go into the file - with a text editor - and git will have a message for me, so I'll know what to change. Then I add the changes and commit.

When you get a merge conflict, how do you fix it? Git will tell you where the merge conflicts are, at least which files. The manual way to fix it is simply to open that file in a text editor, and git will have actually left markers where the two changes are. You edit the file as you see fit, and then add and commit the file, leaving a note that it's a merge.

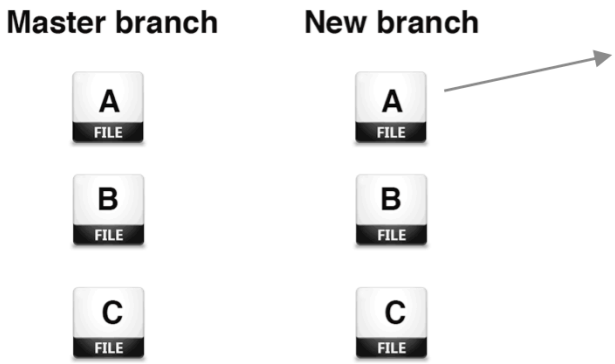
There are also tools that help you visualize the merge conflict – you can set them up in your config file. I am going to suggest you read an article about how to deal with merge conflicts, I will have the link in the slides later on. For now let's look at an example of a merge conflict.

Merge conflict

Intro sentence in file A in both the master and new branch reads “Let’s get started immediately”

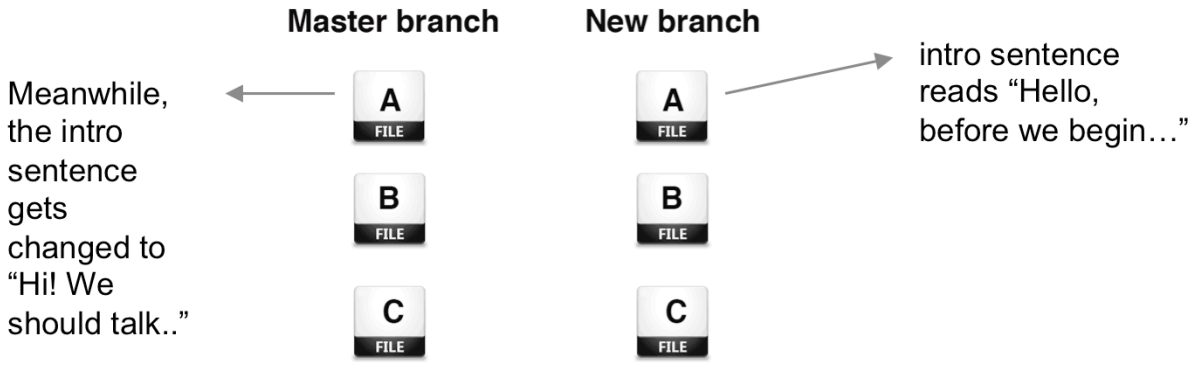


Merge conflict



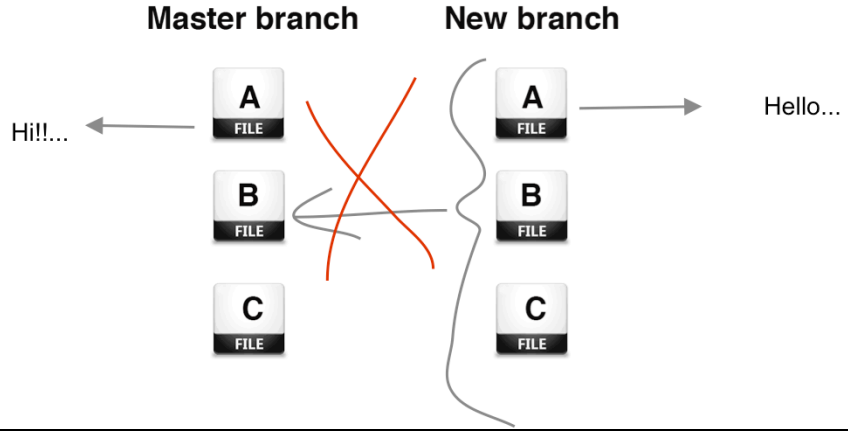
Changed the intro sentence to read “Hello, let’s talk about this important thing…”

Merge conflict



Merge conflict

When you try to merge the branches, Git doesn't know which change to make. You'll have to choose.



Merge conflict

```
TSLOSX413A:mt katebronstad$ git merge second_branch
Auto-merging afile.txt
CONFLICT (content): Merge conflict in afile.txt
Automatic merge failed; fix conflicts and then commit the result.
TSLOSX413A:mt katebronstad$
```

When you have a merge conflict, you will see a message like this on your command line. It happens after you run a git merge command. To fix manually, open up the affected files in a text editor.

Merge conflict

```
1   <<<<<<< HEAD
2   Hi, lets talk about this important thing.
3   =======
4   Hi we should talk about MassMoca.
5   >>>>>>> second_branch
6
7   Mass MoCA has always been about the vast and the
    become an unlikely but very real success.
8
9   Building 6 expands dramatically on that fact. An
    which will showcase installations by art-world st
```

Git will have marked the files with notations that show you exactly where the conflicts are. Take out the <<<<HEAD and ==== and >>>>branch_name notation, and leave in the code you want. Then add & commit the file.

Manually dealing with a merge conflict

- Edit the file(s) with conflicts in a text editor
- Add and commit the new edit

Help with merge conflicts

- You may prefer to use a visual tool to deal with merge conflicts
  - Git UI or other merge tools
- “14 Tips and Tools to resolve conflicts with Git”

Some people like working with the files manually, others want some visual help – this article has some advice.

Merging: take theirs vs. take ours

- Instead of managing any merge conflicts yourself, tell git to either merge in favor of your commits or the ones you are bringing over from a remote
    - Git merge --strategy-option theirs
    - Git merge --strategy-option ours

Something else you can do if you get a merge conflict is back out of it and try what's called a merge strategy (when I say back out, I mean, reset the head on the branch you are on and try the merge again with a different command).

Let's say I did a merge and I got a conflict on a bunch of files that don't really matter – or it's going to take too much work to clean up, and you maybe haven't updated your branch in a while – you can tell git, any change you see, take their branch, meaning the branch you are merging in, or "Our branch", meaning the branch you are on. If you use this, you probably want to have your existing work on another branch in case you need to refer back to it, or you can use the commands git stash and git pop.

A note about keeping forks up to date

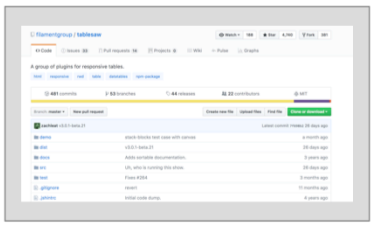
- Can i keep a fork up to date in GitHub?
  - Now possible through pull requests
  - http://stackoverflow.com/questions/7244321/how-do-i-update-a-github-forked-repository

I wanted to review how to keep forks up to date with the original project – this is something that was mentioned on the survey – someone asked if they can keep their project up to date within GitHub. Up until recently, this wasn't possible at all. It is now but to be honest I think it's a bit unintuitive and I think for now it's best to do it on the command line or through a GUI, which we will show next week. But this link is here if you are committed to the idea of keeping your branch up to date on GitHub

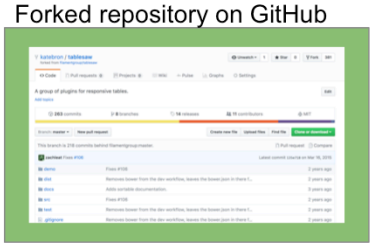
A note about keeping forks up to date

.

Original repository on Github



Forked repository on GitHub



Fetch from original;
Merge to local

Local copy of forked repo

Push merged branch up to GitHub forked version

I do want to make it more clear about the workflow of working on a forked repository – fetch from the original, from upstream, and you might want to do this in a branch, and then merge it into your master branch. You push this back up to GitHub so it reflects both your changes and the changes most recently made on the original repository.

“Remotes”: what is a remote

- A remote points to external sites your repository also lives
  - Example: origin = git@github.com:katebron/Moby_dick.git
    - If I run “git push origin master”, it pushes my repository changes in my local master branch to git@github.com:katebron/Moby_dick.git
  - Run git remote -v to see a list of remotes in your project and the urls associated with them.

Also on the survey someone asked for some clarity on remotes. A remote points to a URL where your project also lives, so you can easily push and pull back and forth. Git Push origin master pushes my repository – the new changes, really – up to my GItHub version, so the two can be up to date with each other.

You can see the remotes you have by running git remote –v.

“Remotes”: already in GitHub clones

When you clone a repository from GitHub, it will automatically have an “origin” set up for you.

- The origin will be the default branch of the project (usually master)

Repositories you clone from GitHub will have a remote alias set up as origin by default.

Usually this is the master branch of the origin, but you can set your default branch in GitHub to be something other than master.

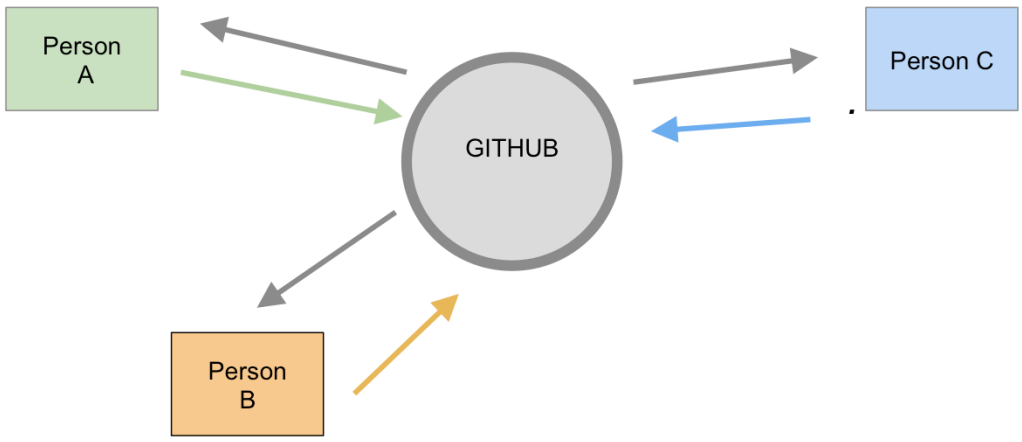
If you saw the last class, I pulled down this repository called devfreebooks (go to this), and I was using git pull, and I thought it was the same thing as git pull origin master, but when I ran git pull origin master, I didn’t get the result I was expecting – this is because the default branch for this project is actually the dev branch - you can see that here on the command line and also when I go to the GH page for this project.

“Remotes”: how to set up

- Can add them in your project by using the **git remote add *aliasname*** command.
  - Example: **git remote add origin git@github.com:katebron/Moby_dick.git**
  - Example: **git remote add upstream git@github.com:devfreebooks/devfreebooks.github.io.git**
  - Example: **Git remote add bitbucketcopy somebitbucketaddress**

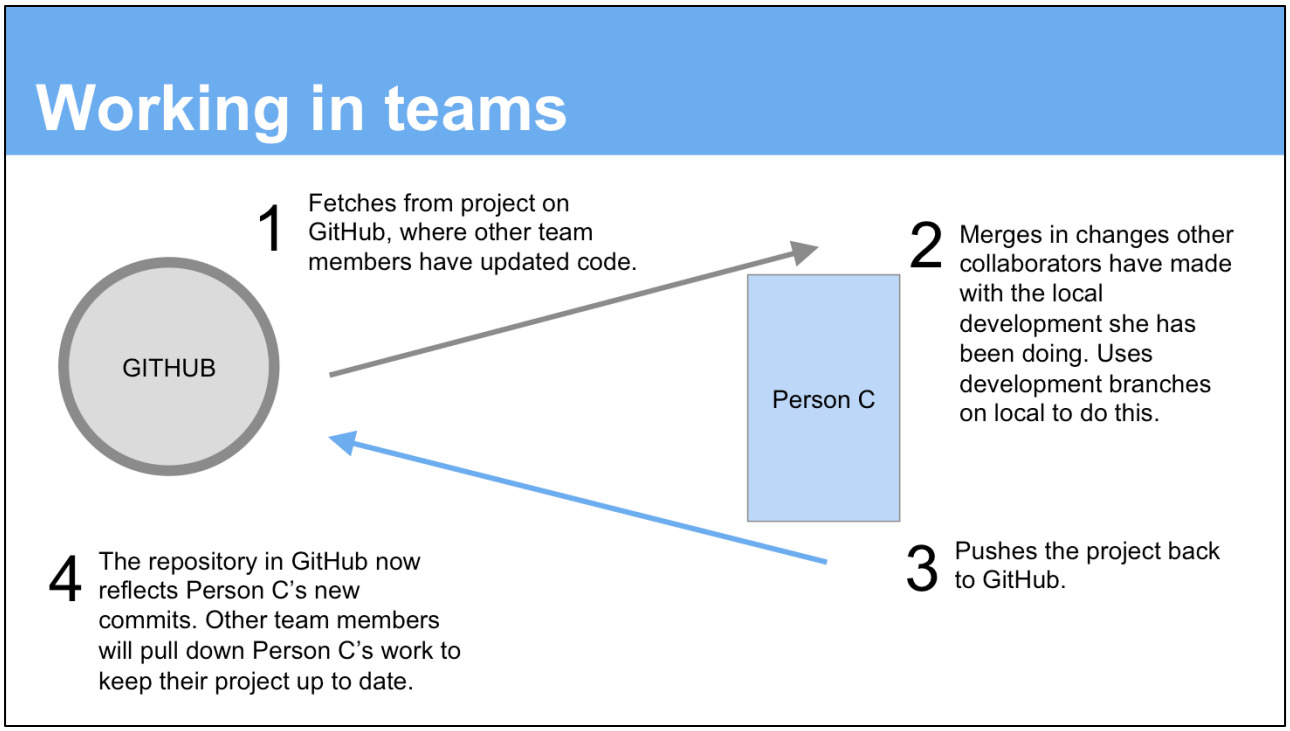
You’ll need to set up remotes for a couple of different situations:
1, when you started the project on your machine and not from a GitHub clone
and 2, when you already have origin set up from a clone, but you need to set up a separate remote, either to keep the project uptodate with the original fork, or if you for whatever reason are using 2 separate git hosts, like an internal server or bitbucket. I don’t think there is a limit to how many remotes you have.

Working in teams



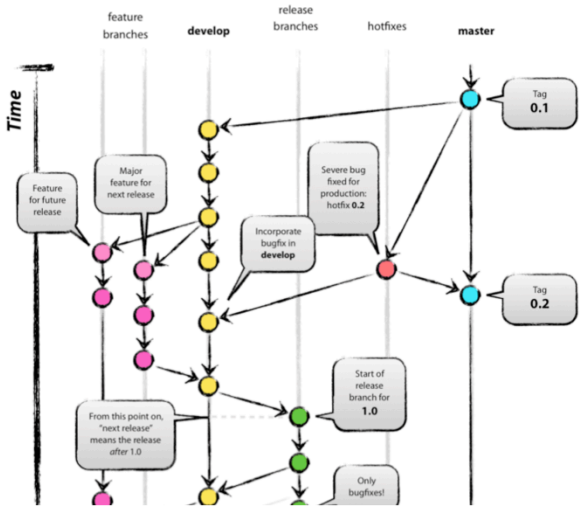
Someone else mentioned wanting to know about working in teams. Git is incredibly flexible about how to set up your workflow, and work with others, so I am going to mention some really basic workflows.

Let's say for an example there are 3 people working on a website – each member is going to be pushing and pulling from the GitHub copy, whether it's through a fork and pull request or directly to a project. Let's look at just one slice of this –



Let's just look at Person C and assume that everyone else is essentially working the same way – so, person C pulls down from the team's project page on GitHub. The changes from the GitHub project – changes made most likely by her team mates – get merged into her local copy. She probably uses a dev branch to do her own work and to merge in her own code when it's ready for production. If she's pushing directly to a project instead of a pull request, she'll probably pull down one last time before she pushes up so she can make sure her code is completely up to date and doesn't conflict with any new code.
Once she pushes to GitHub, the repository now reflects her new changes. Persons A and B will eventually pull down her code and merge it in with their changes.

Git branching model



- From "A successful Git branching model", by Vincent Driessen
- Basis of Git-flow rules and techniques for collaborating.

The last slide was pretty simplified. There's a lot of thought that goes into the best workflow to have. If you are interested in workflows, I recommend this article by Vincent Driessen. It had a huge impact on developer communities and is the basis of git-flow, which is a model of an ideal workflow in git – some people have criticized it for being too complicated – but a lot of groups have adopted it.

This image – shows the branching model Driessen wrote about – you have dedicated branches for different kinds of development work, depending on plans and needs, and this gets reflected in a team's overall workflow, which he also addresses.

Checkout as a quick file change

- You can check out a file
- Can check it out from a branch or another commit.
    - Git checkout HEAD name_of_file
    - Git checkout branchname name_of_file

I just wanted to mention one last thing – this goes back to the 2nd class when we were talking about undoing changes. One thing you can do with branches is you can check out one file at a time – from a branch or from a commit. Let's say I am on the new branch to alter the library search box – and I have a file, let's say it's javascript, that I'm using for some user interaction with the form. I royally mess it up on the new branch – I could check it out from HEAD if I haven't committed it yet – or I could check it out from the other branch

Gitignore

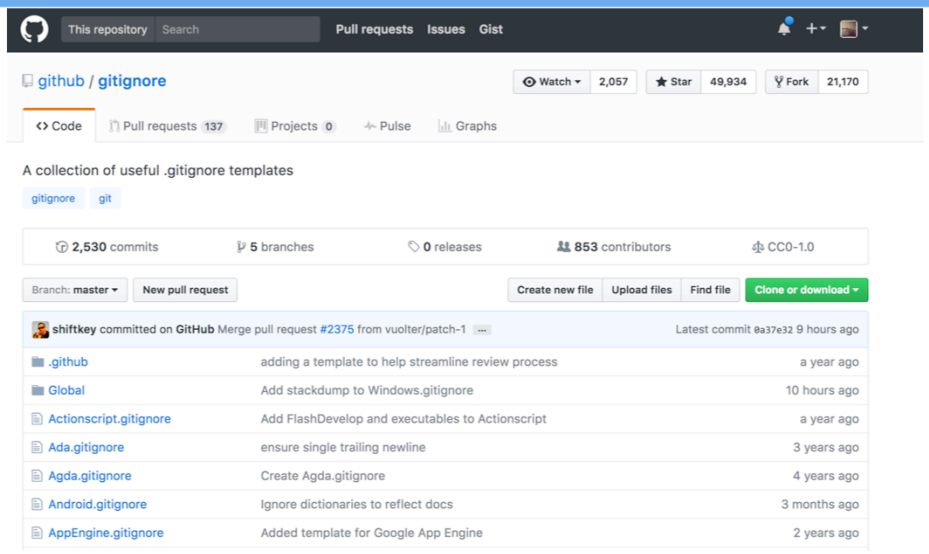
A file, not a command

Git Ignore files you don't want to track

- privacy: user information, confidential information, including config files
- Files used in only in local development or that get installed by package managers (ex. NPM, bower)
- Images

Sometimes there are files you don't want to track. This could be because you have confidential info – like configuration files, which may have passwords – or because they are huge, clunky application files that aren't needed for the program, and are only used for local development to compile files – things like sass cache directories that get created whenever sass is compiled into CSS, or files that get downloaded with package managers like with NPM install.

Gitignore



The good news is that there are templates for gitignore files out there on places like GitHub – so depending on the language or framework you are using someone may have already created a gitignore file you can edit for you own purposes, but they have an indication of where the sensitive documents are, or where are the files that get created that aren't worth tracking.

Example Gitignore file

WordPress .gitignore

19 lines (16 sloc) | 300 Bytes

```
1  *.log
2  wp-config.php
3  wp-content/advanced-cache.php
4  wp-content/backup-db/
5  wp-content/backups/
6  wp-content/blogs.dir/
7  wp-content/cache/
8  wp-content/upgrade/
9  wp-content/uploads/
10 wp-content/wp-cache-config.php
11 wp-content/plugins/hello.php
12
13 /.htaccess
14 /license.txt
15 /readme.html
16 /sitemap.xml
17 /sitemap.xml.gz
18
```

Basically a .gitignore file is a list of exceptions. By default, git is going to track a change to ANY file it sees under your git repo. The .gitignore file tells git to not even bother tracking any changes it sees in this directory, or has this naming convention, etc

Next week

- Practice dealing with a merge conflict
  - Instructions
- Read about the branching model and/or merge tools and post to forum one thing you've learned, or want to know more about