



Welcome to the Git 'r Done workshop!

Getting Started With Git

What is Git? How is it different from GitHub? In a nutshell, Git is a powerful version control system that allows for easy collaborative coding in a non-linear way. GitHub is an online host for Git repositories where people can collaborate on open-source projects of any size.

This tutorial assumes no previous knowledge of Git, GitHub, or Linux command line interface (this one is optional throughout the tutorial), so if you're new to this have no fear!

Some Terms

- Repository: A repository is essentially a codebase surrounding a project or group of projects. As an example, check out the official Brock CSC GitHub organization at <http://github.com/BrockCSC>. Here you can see a number of repositories. There is one called "acm-icpc", and this is repository is used as a space to do practice problems for the ACM International Collegiate Programming Competition. (We send students every year and it's totally awesome so if you don't know what that is, keep an eye out next year!)
- Branch: Consider a tree. As it grows, branches stem from the base of the tree and develop independently. A branch in a Git repository is very similar to this. It is like duplicating the current state of a project to work on a certain feature, without interfering with other people's progress on a different part of the project.

- Merge: Consider a tree again, only this time it's branches are deformed and warp back into the base of the tree itself. This is a merge. Once you have completed work on a feature in a branch of the repository, it can be merged into the main code base, referred to as the master branch.
- Commit: A commit is like a snapshot of your project at one point in time. It is (in a literal sense) *committing* to the work you've done. After you've done some coding, it is a way to save all of your changes locally to the repository.
- Push: If your repository has remote references (that is, a repository that is not *just* local, like something on GitHub) once you have committed your changes, they have been saved but will not be seen in the remote repository until they have been pushed there.
- Pull: Similar to a push, but in the opposite direction. If someone has made changes to the repository and you want them to appear on your local version, you will need to pull them from the remote version.
- Git Ignore: When you initialize a repository, you can include a file called '.gitignore', this will sometimes be automatically created, depending on how you initialize a repository, and it can be used to tell Git that certain files in your repository should not be tracked at all. This is particularly useful if you have certain files with sensitive information, like server information or auto generated files such as build files that change with every commit.

Throughout the remainder of the workshop we'll be using the GitHub desktop application, as well as Git Bash, a command line interface. You can download this at <https://desktop.github.com/>, or if you prefer use any alternate Git tool you like. There are many other options

Creating Your First Repo

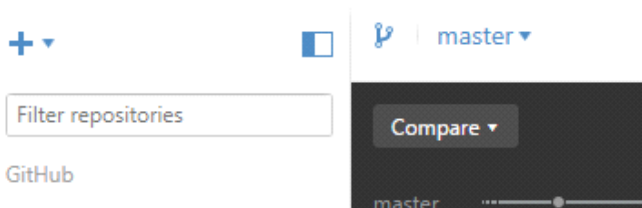
There are a few different ways to create a repository. We will look at initializing a new repository, cloning (creating a copy of an existing repository), and turning an existing codebase into a Git repository.

Initializing a New Repository

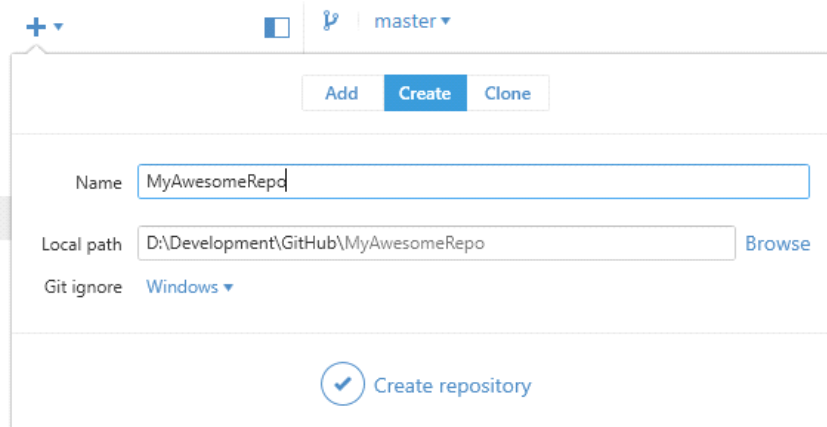
We can create a new repository using Git Bash, or through the GitHub desktop application. This is what you want to do when you're creating an entirely new project. We're going to look at both methods, so go ahead and try either, or both!

GitHub Desktop Application

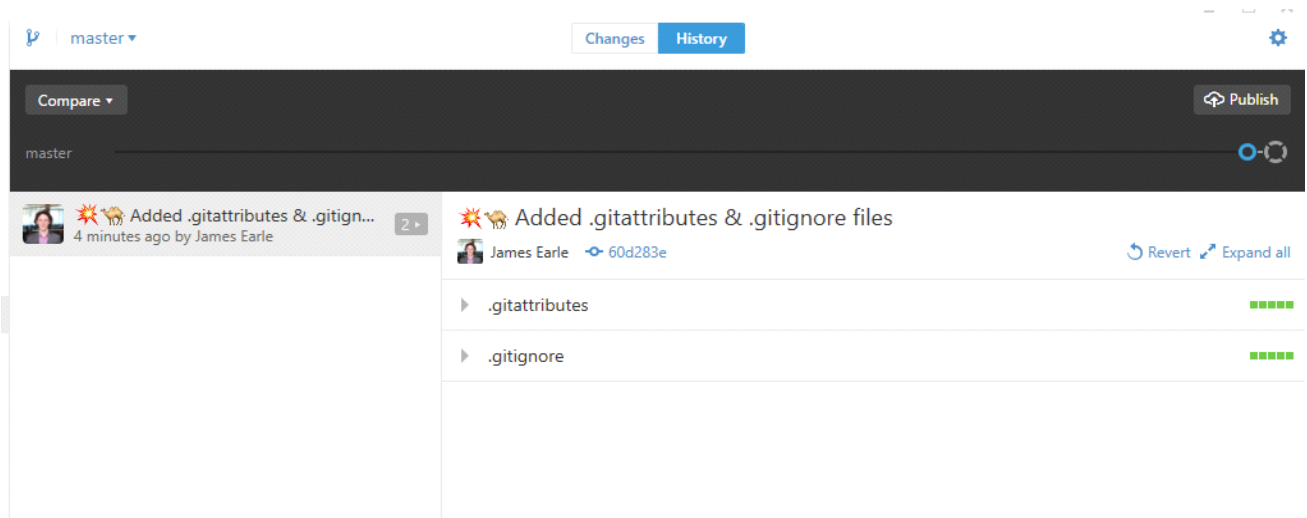
Likely the easiest way to do this for a beginner (although for those of you eager to learn some command line tools today, check out the following alternate method with Git Bash), the GitHub desktop app allows you to simply click and create. To begin, launch the desktop application and select the plus button in the top left-hand corner.



In the drop-down menu that appears, under the “Create” tab provide a name for your repository and where you would like the project to be located. Once you’ve filled this in, you can select “Create Repository”



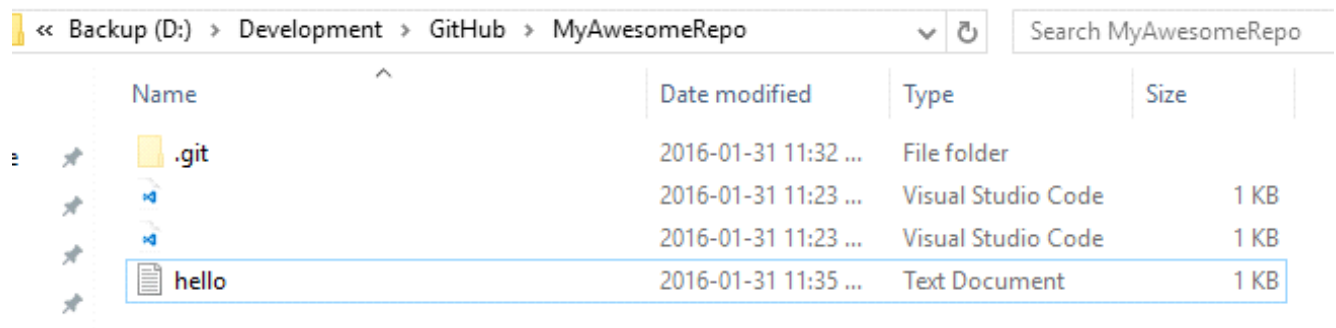
Once this has completed, you'll be shown the initial commit that is created automatically. In this commit, .gitignore and .gitattributes files are automatically generated for you. You can use the .gitignore file to specify documents or folders that should you don't want Git to pay attention to. It is a good idea to add the build folder to this file if not already done so. These files are recreated with every build so they will often unnecessarily complicate your commits.



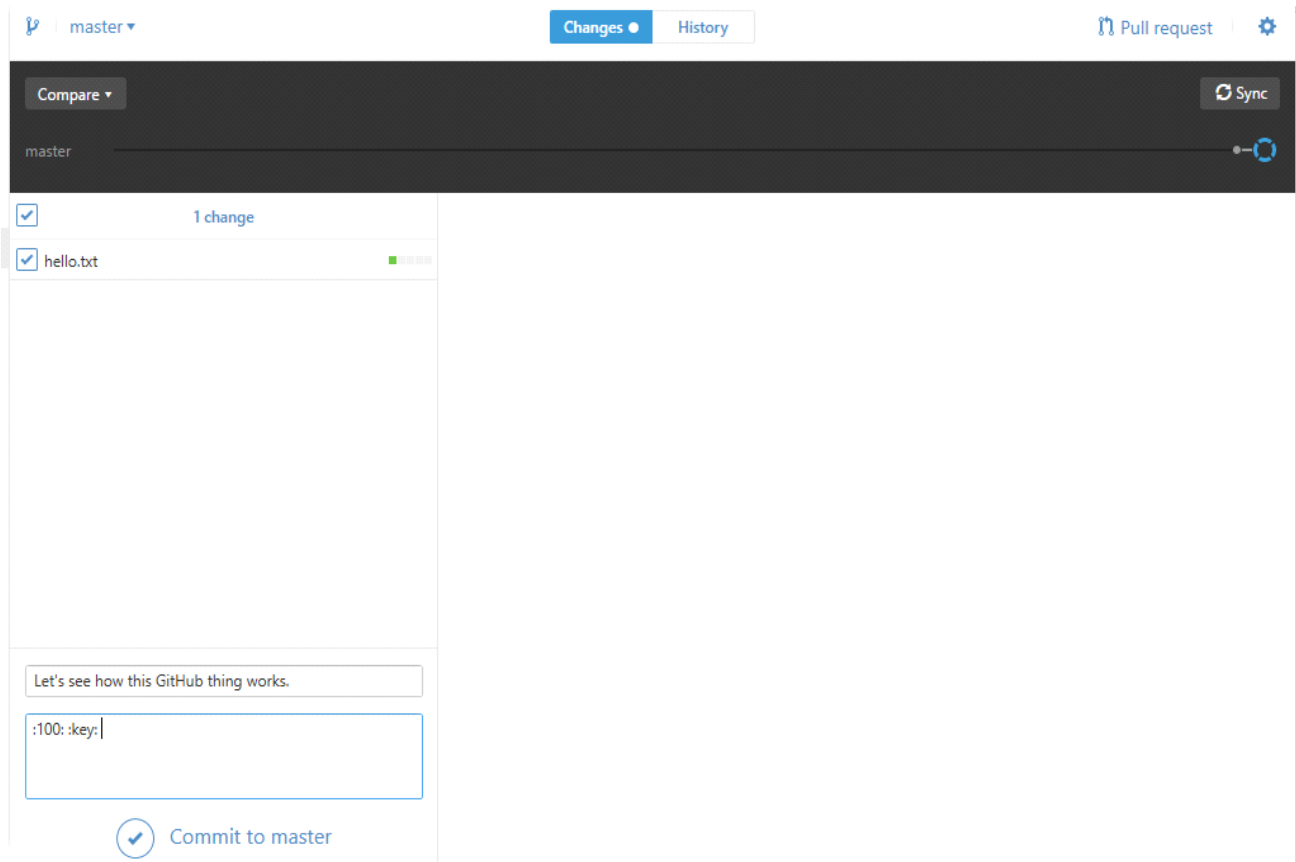
Side note, GitHub supports emoji characters in your Git messages and descriptions of your repositories. Why not checkout <http://www.emoji-cheat-sheet.com/> to find the codes for something you'd like to include in the description of your repo? :tada:
:confetti_ball: :beers:

Congratulations! You've now created your first repository using the GitHub desktop application. By clicking the "Publish" button in the top right corner you'll be prompted to provide a description for your repo (where you can put those fancy emoji's, if you like) and then you're done! Your repository is now visible on your GitHub account.

Now let's create a new file to try committing changes to our repository. Open Windows Explorer and go to the directory where you've initialized your repository. Create a new text file here. You can name it anything you like, and just put the words "Hello World!" inside, then go back to the GitHub desktop application.

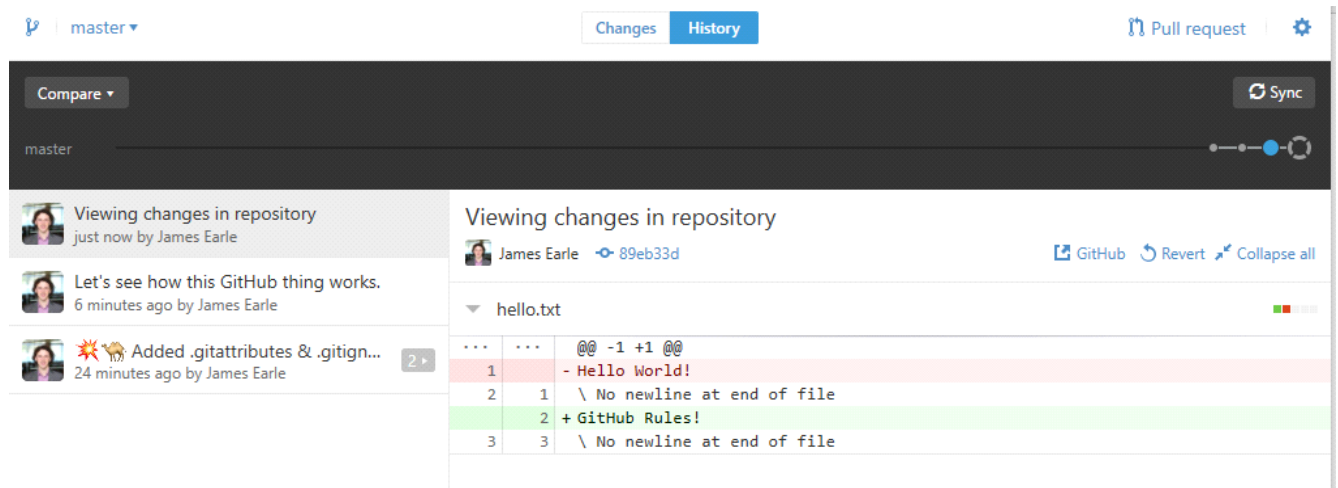


The GitHub application should now show that there are untracked changes in your repository. Be sure you select the "Changes" tab in the top center of the application, as before we were viewing "History".



Once you provide a title and description you can commit these changes to the repository. A notification will appear confirming that you've done this successfully, but remember our definition of a commit? These changes are stored locally, but haven't been pushed to the remote copy of your repository yet, and so aren't visible on the GitHub website. To push them to the remote version of your repo, you need to press "Sync" in the top right-hand corner of the application. Similarly, if the remote version is ever ahead of your local repo because another user pushed their changes, pressing "Sync" will pull them to your local copy.

The GitHub desktop application additionally shows changes to files when you are like at previous commits to a repository. To view how this works, go to the file you created previously. Delete the words "Hello World!" that we put in there before, and now type something new, like "GitHub rules!". Follow the same steps above in committing your changes and syncing them with your remote repository. Once you've done this you can look at the "History" tab in the desktop application to look at changes made to this file.



Any additions to a file are shown in green, prefixed with a plus sign. Similarly, deletions are shown in red prefixed by a delete sign. This is a great way to compare changes made to files between commits, and is also possible in your browser on the GitHub website when you're viewing the details of any given commit.

Git Bash

The process can seem a little daunting at first if you're not familiar with using the command line for things like this, but using Git in the command line can certainly yield benefits in that it is the simplest way to work between different platforms without relying on GUI software, and command line knowledge can be useful! Keep in mind throughout this entire process, if you're ever lost you can use the help command in git to understand more about the commands we are using. For example, '`$ git help`' or will give you general help and a list of Git commands, and '`$ git commit --help`' will provide documentation on the specific command "commit", which we'll learn about more soon. Here are some commands to get started.

```
# Create a new directory
$ mkdir MyAwesomeRepo

# Move into that directory
$ cd MyAwesomeRepo/

# Initialize the repository
$ git init
```

Congrats! You've just created a repository. You can verify this by typing '`$ git status`' and seeing the output, which should look something like this.

```

James Earle@JAMESEARLE-PC /d/Development/GitHub
$ mkdir MyAwesomeRepo

James Earle@JAMESEARLE-PC /d/Development/GitHub
$ cd MyAwesomeRepo/

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo
$ git init
Initialized empty Git repository in d:/Development/GitHub/MyAwesomeRepo/.git/

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git status
On branch master

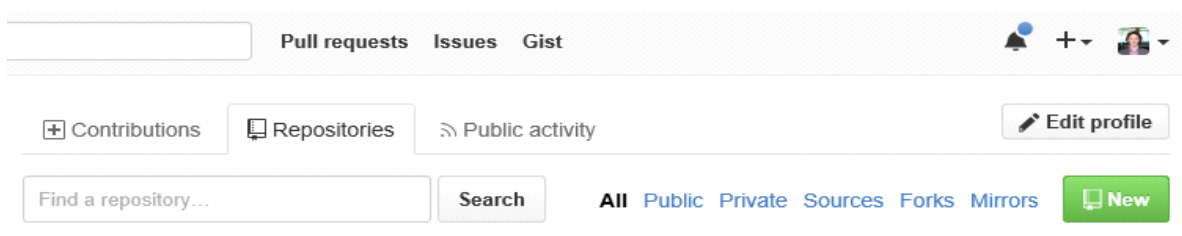
Initial commit

nothing to commit (create/copy files and use "git add" to track)

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$

```

Now what if we want to put this repository on our GitHub accounts? The easiest way to do that is to go to your GitHub profile, click on the “Repositories” tab, and select the green “New” button, as shown below.



Provide the correct name for your repository, a description, and then select “Create Repository”. Don’t worry about the other options for now.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: JamesEarle / Repository name: MyAwesomeRepo ✓

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

Description (optional)

A cool new repository.

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

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

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None**

Add a license: **None**

Create repository

Once you've done this, you can go back to the command line to add this as a remote reference for your local repository. This is done by typing

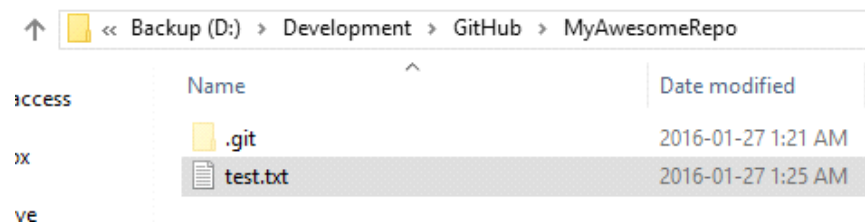
```
$ git remote add origin <URL to your repository on GitHub>
```

So for me, this would be

```
$ git remote add origin
```

```
https://github.com/JamesEarle/MyAwesomeRepo.git
```

Now we can add a file, commit it, and push it to the remote repository to see GitHub in action! Create any text file, just to create an untracked change in the repo.



| | Name | Date modified |
|--------|----------|--------------------|
| Access | .git | 2016-01-27 1:21 AM |
| ix | test.txt | 2016-01-27 1:25 AM |
| ve | | |

Before adding a file, try typing '`$ git status`' to see your untracked changes in the repo. You should see something similar to below.

```
James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git status
On branch master

Initial commit

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

        test.txt

nothing added to commit but untracked files present (use "git add" to track)
James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$
```

Before you are able to commit changes to the repository, they have to be added as tracked changes. This is like filling a box with something you want to send in the mail, but you haven't put it in the post box yet. To add a file, you can type '`$ git add --all :/`', which essentially says to add all new files in the entire repository directory. You can also add specific files by using the filename, such as '`$ git add test.txt`'.

Once you've done this you can see that your changes have been added to the list of files ready to be committed. To commit from the command line, you simply type '`$ git commit -m`

"My Commit Message" , where the '-m' tag associated with the command just tells git that you'd like to provide a message describing this commit.

```
James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git add --all :/

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git status
On branch master

Initial commit

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

    new file:   test.txt

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git commit -m "My first commit"
[master (root-commit) 4290ec2] My first commit
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ |
```

And finally, now that we have committed our changes to the repository, we can push them to the remote. If you take a look at the remote on your GitHub profile before doing this, you'll see that the changes you've made locally are not there yet. To push them to the remote, simply type '**\$ git push origin master**'. You will be prompted to type in your username and password, and once you have successfully authenticated the changes are sent to the remote repository!

```
James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git push origin master
Username for 'https://github.com': jamesearle
Password for 'https://jamesearle@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 228 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/JamesEarle/MyAwesomeRepo.git
 * [new branch]      master -> master

James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ |
```

Cloning

Cloning a repository is essentially creating a copy for yourself locally, but any changes you make will be visible and applied to the same repository. You can only clone a repository that already exists from another location. This is slightly different from something called Forking which creates a complete duplicate repository, including all history of development

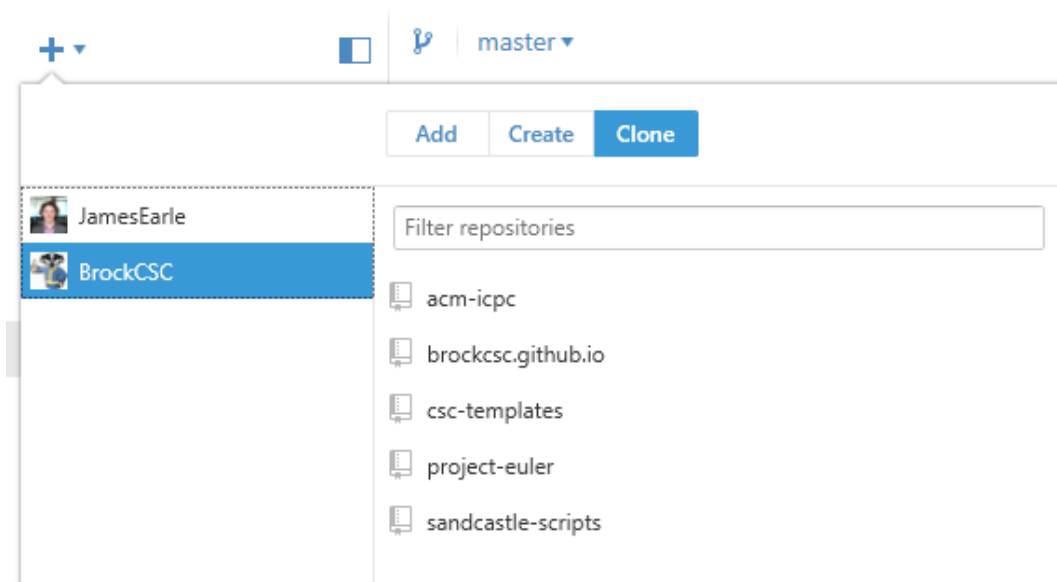
branches and previous commits. When forking another repository, any changes you make to your copy will remain in your copy, while cloning means your changes will be merged to the *same* version you cloned from. This means anyone else who uses that repository will get to see your changes.

If you want to clone a repository that you've found on GitHub so that you can contribute, you can do this using Git Bash, or the GitHub desktop application. This is a very easy way to get a Git repository setup on your machine locally, because all of the work in initializing it (and likely in setting up the code inside the project) has already been done. What's easier, building an Ikea couch, or finding one that's already made?

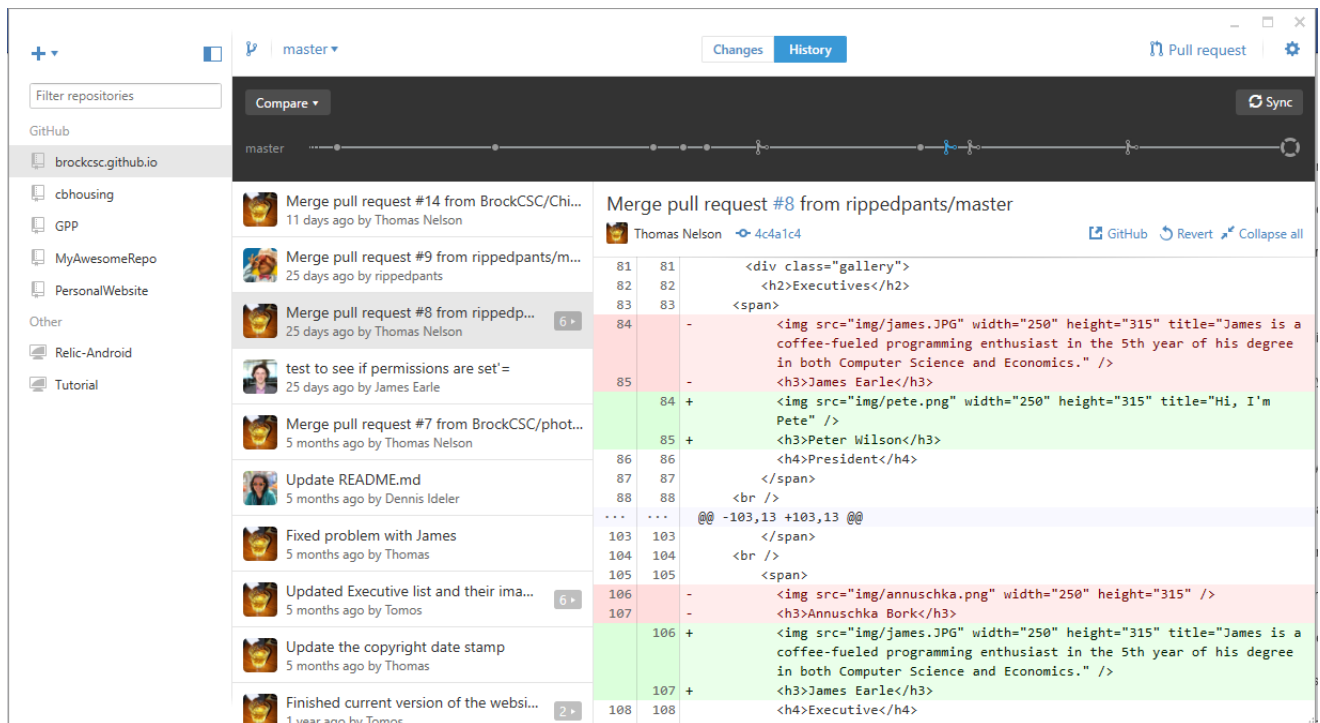
GitHub Desktop Application

Begin by opening the application. You'll be faced with the default view. Go to the plus sign in the top right, and similar to when you create your new repository, there is a tab that says "Clone", where you can easily clone one into a new folder.

You'll be asked to select from one of your repositories to clone from, or the repositories of any organization you contribute to. In this case, I can see all of my own repositories as well as those belonging to the Brock CSC GitHub page.



From here you are able to select any repository you'd like to clone and it will be sent directly to a folder you specify when prompted. Why don't you go ahead and try cloning something from the BrockCSC account? If you're not listed as a member, we can add you as soon as you have an account and the repositories will be listed for you. Now that you've cloned this repository, you can see all previous commit history and changes made to the codebase.



You are now free to contribute actively to the repository you've just cloned.

Note: Before contributing be sure you understand the best practices and conventions that other contributors on the project want you to follow. For example, if you want to contribute to a Brock CSC repository, we follow the official Git Workflow (which we'll be going over), so we ask that you follow this convention if you want to contribute to any projects.

Git Bash

Fortunately, cloning in the command line is just as easy as using the desktop application, and also does not only let you select from a list of your own repositories or repositories of your associated organizations. To get started, simply open Git Bash and enter the following command, but replace the URL with the correct URL for any repository you'd like to clone.

```
$ git clone http://github.com/BrockCSC/acm-icpc
```

```
James Earle@JAMESEARLE-PC /d/Development/GitHub
$ git clone http://github.com/BrockCSC/acm-icpc
Cloning into 'acm-icpc'...
remote: Counting objects: 542, done.
remote: Total 542 (delta 0), reused 0 (delta 0), pack-reused 542R
Receiving objects: 100% (542/542), 6.10 MiB | 4.48 MiB/s, done.
Resolving deltas: 100% (103/103), done.
Checking connectivity... done.
```

And similar to using the desktop application, if you want to view the commit history associated with the app, you can type `$ git log` and type `':q'` when you've finished.

```
James Earle@JAMESEARLE-PC /d/Development/GitHub/acm-icpc (master)
$ git log
commit 52605d39cda265f2a8ce8870c6ddc9588abfa453
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Thu Jan 7 23:54:17 2016 +0000

    Fix broken image

    Replace with an image hosted on GitHub. Should last longer.

commit 3c1d912c29b892ebb2fb0c9cb3b71b7c60d20ca1
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Fri Jun 19 23:00:54 2015 +0100

    Update README.md

commit efb90c8d6ed6f2b6f8570045158b8f57498baa4
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Thu Jul 3 03:29:53 2014 -0400

    Add another size definition

commit 1cf4b57e0003676af6f4ef5450cc0284b32dacda
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Sat Aug 17 12:09:20 2013 -0400

    Update geometry code, add new operations and tests

commit 65dcd2a95ed2bbb4ad0091340a0098717a3d7fed
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Sat Aug 17 12:05:56 2013 -0400

    Update vector of pair of ints.

    Pair of integers changed from pi to pii, but vector of integer pairs is still
    using pi in typedef vector<pi>. It must be typedef vector<pii>.

commit 355b892abc71dcbc3a6a50ee774213bcf4f1d7c2
Author: Dennis Ideler <ideler.dennis@gmail.com>
Date: Sat Apr 27 18:45:16 2013 -0400

    Update geometry library and add tests

commit 2628619551ea5b8740d3329d7e49f7937b59b8fd
```

Creating a Repository from an Existing Code Base

Fortunately, this process is almost identical to creating a brand new repository. In the **GitHub desktop application**, you can simply go to the plus sign, and under the “Create” tab provide a name for the repository and navigate to the pre-existing codebase. All of the files that already exist will be untracked, so you’ll have to commit them, push, and then publish the repository for it to be visible on your GitHub account.

When using **Git Bash** you follow the exact same steps outlined above for initializing a new repository, only this time there is no need to make a new directory. Simply navigate to the directory of interest using the command line and follow all the instructions beginning at when you type the command ‘\$ git init’

Help! I’ve made a mistake!

At some point in your life as a programmer you are going to end up making a mistake and want to go back to a previous commit in your repository. Sometimes it isn’t even a mistake, and instead you just come up with a new idea and want to go back to a cleaner version of your branch. This comes in many forms, but here are some other typical examples:

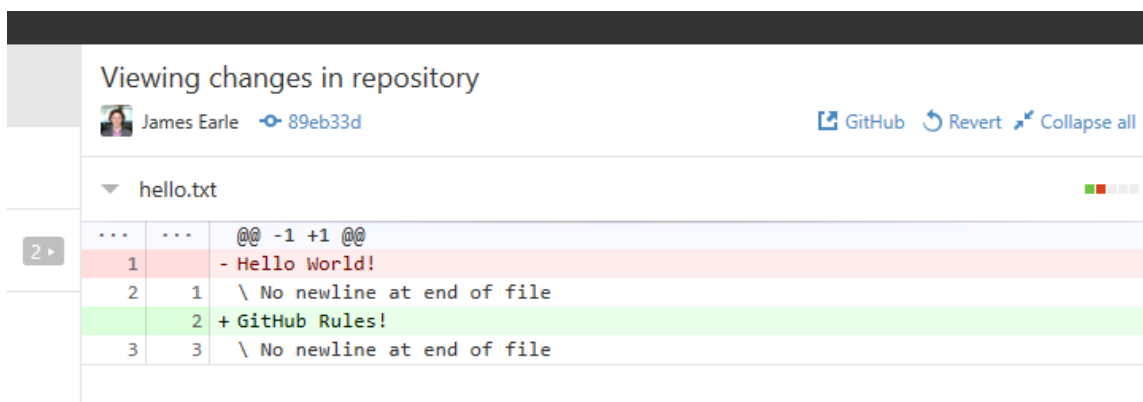
- You want to try to optimize some code. You make some changes, but it turns out that your idea won’t work out. You don’t want to go back and undo all of your changes manually, so what should be done?
- You spend an hour or two building a class to read in some particular data, and suddenly come up with a much better way of accomplishing this task! Instead of just deleting your branch and losing any initialization of your object across the project that you did in earlier commits, what else could be done here?

The answer to all of your problems and more is Git’s ability to revert to a previous commit. This will effectively turn the branch you are currently on right back to the state of any previous commits. This does not delete the commits that you have pushed but instead builds a commit that undoes all changes since the revert point. This means that it is possible to *revert a revert*, which also means that your work is technically never lost!

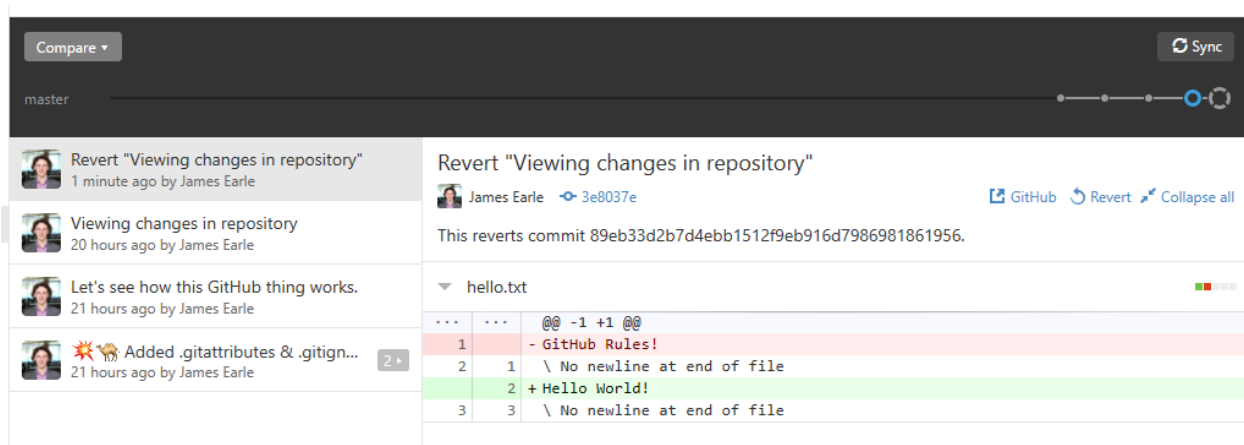
So head back to the “MyAwesomeRepo” repository we made earlier, and now we can experiment with undoing our previous committed changes. Turns out we really like “Hello World.”

GitHub Desktop Application

When inside the app and viewing the commit history, select the commit you want to undo. In my case, that commit was titled “Viewing changes in repository.” Next you need to select the “revert” button above the portion of the screen that shows you additions and deletions to the file.



Phew! That was close one, but don't worry you reverted your changes and all is well. Simply press "Sync" in the top right of the screen, and the revert will be pushed to the remote. Remember, if it turns out you've changed your mind, a revert is really only a new commit that undoes changes up to the chosen commit, so you can revert this revert to get the old state of the file back!



Git Bash

Because we just reverted our previous commit in the above desktop version, let's try reverting that revert using Git Bash! It's just as easy as above, only with one small hitch. If you're new to the command line and don't know what Vim is, don't worry! We'll walk you through the process.

Open up Git Bash and navigate to your repository. There, type ``$ git reset HEAD``. You should see a new page popup that looks a little unfamiliar. This is the Vim environment. Vim is a command line text-editor that is based around control commands to write to a file, delete text, etc. It's not always as simple as just typing what you want to type, but fortunately right now all you need to do is confirm the changes made. Enter the command in Vim to write changes to the file by typing ``:w`` and hitting enter, then quit using ``:q`` and hitting enter. After this you're brought back to the basic CLI and should see some details about your reversion, like below.

```
James Earle@JAMESEARLE-PC /d/Development/GitHub/MyAwesomeRepo (master)
$ git revert HEAD
[master a0dcf76] Revert "Revert "Viewing changes in repository""
1 file changed, 1 insertion(+), 1 deletion(-)
```

Help! I have Merge Conflicts!

Oh no! When I try to merge my two branches it says there's a merge conflict! What do I do?! Where do I go?! Who do I blame?! First of all, there's not necessarily anyone to blame. As for what to do and where to go, that depends!

Let's say you're working on your branch and finally finish everything you set out to do. You tested everything, it works, it's beautiful and you open a pull request into master. Depending on if there are conflicts or not you will be able to merge right away, but we care about what to do if there is a conflict. First of all, a conflict only occurs if there exist changes which are unambiguous. This means it isn't just an addition of some code or the like. Inside of your file where the conflict is found the following will be placed:

```
<<<<<<< HEAD
My work
=====
Other branch work stuff
>>>>>>> walrusBranch
```

This is Git's way of directly showing you where the conflicts are. From here there are generally four things you can do depending on your Git client.

1. Resolve using "mine".
2. Resolve using "theirs".
3. Manually edit and piece together the code via working with the above Git plaintext stuff.
4. Use an external Git resolution client.

The first two options are fairly self-explanatory. These are just signals to Git to just take either your branch or the others "conflict section". Manually editing would be achieved through opening up the file where the conflict exists in something like Notepad++ and editing from there. This is where you can take the conflict section from both branches and piece together what you need to make it work. Lastly, any Git conflict resolution tool can make this all easy, but that's up to you to explore.

Git Workflow and Best Practices

Version control and Git Hub can be a powerful tool for team projects if used effectively, but can just as easily become an unruly nightmare if not. However, with project members knowing the basics of branching and git workflow you will be able to take advantage of all the features of version control to make the most of your project.

Committing

To get the most out of Git, be sure to commit early and often. Consider the commit as the new line break at the end of each paragraph. Whenever you complete a section of code

that you have tested to ensure it works, commit it to your branch. Remember, you can always revert to any commit along your branch so having a plethora of working backups will only help you if you get stuck down the line. Just be sure to only commit WORKING CODE! It's not going to benefit you one bit if you revert to code that won't compile or doesn't do what you are expecting. Worst off, if you are working on the same branch as someone else (which shouldn't be happening, see below) and push those broken changes, it will effect your teammates as well.

Branches

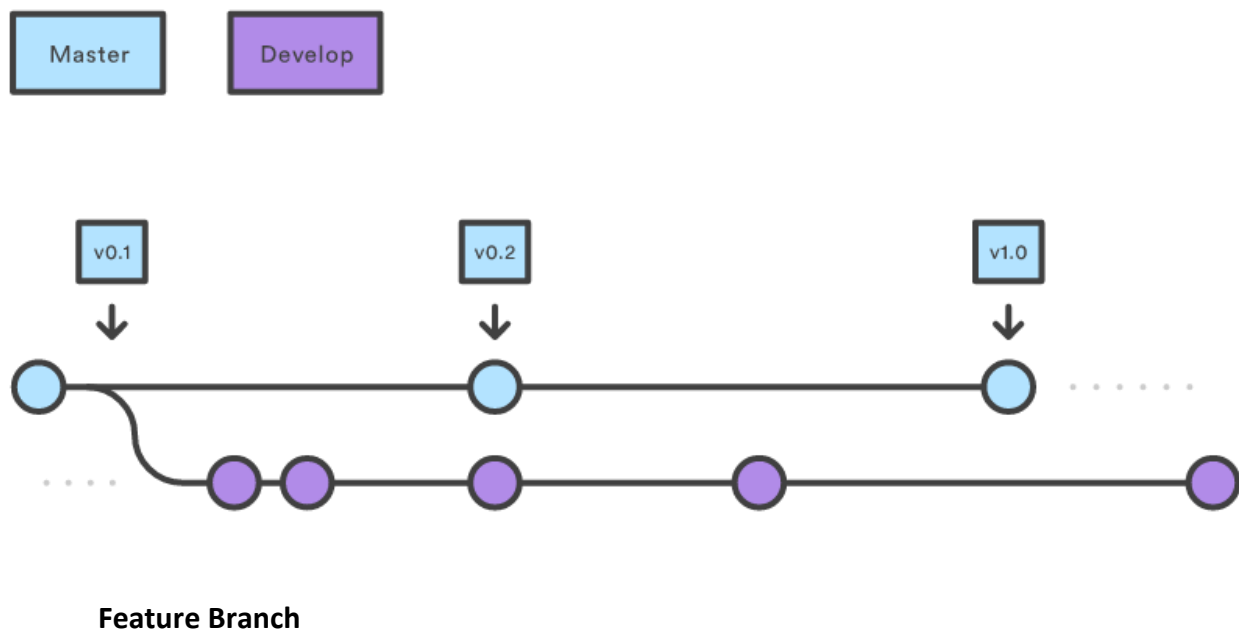
Branches allow for multiple different versions of the application to be maintained at the same time. If used correctly, branches could be a great help to teams with projects that support different platforms or continue to maintain previous versions of the app.

Master Branch

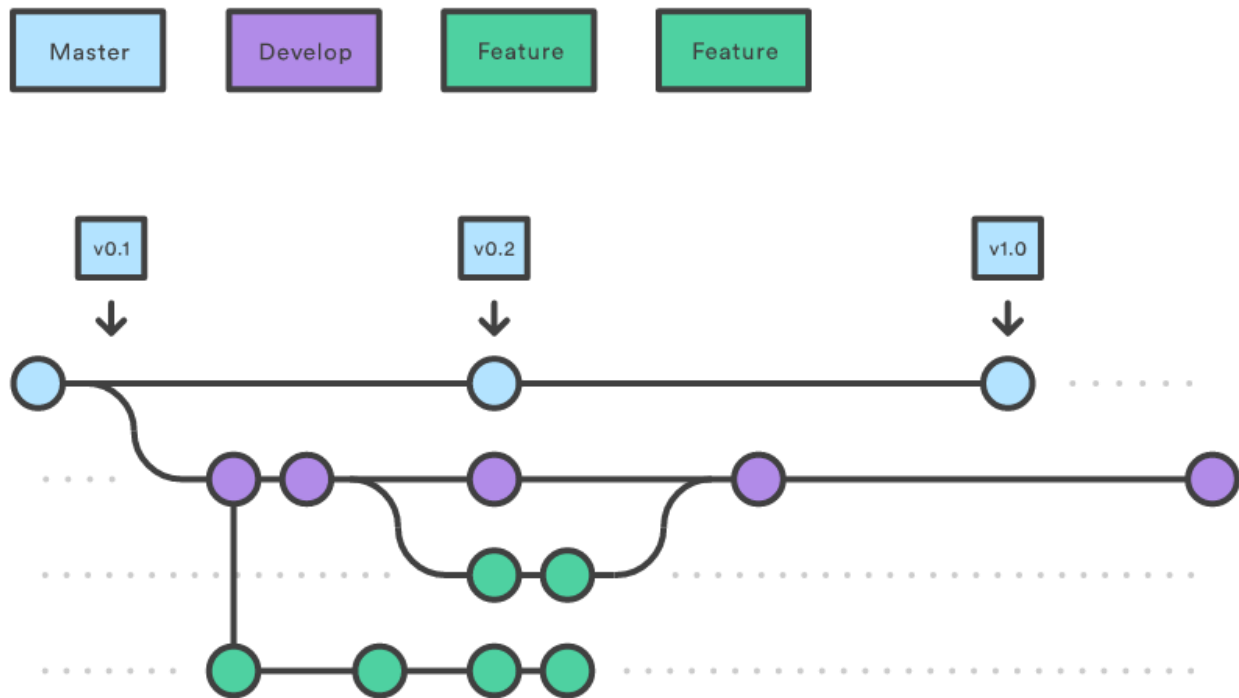
Each GitHub repository comes with a single branch called 'master'. Consider this to be your release version branch. This should only contain the finished product or the version of the application currently released to the public. Please do not do any active development on this branch, especially if you are releasing or have released the application to the public. This can save your life if you need to perform a hotfix or bug fix to the existing product. Prior to release, it isn't a big deal if you alter this, but it's a good idea to not get into the habit of it.

Develop Branch

The first thing you should do is create a branch called 'develop', this is the branch you should be doing you main developing and testing on. You will merge this branch into the 'master' branch when you are ready to release the application.

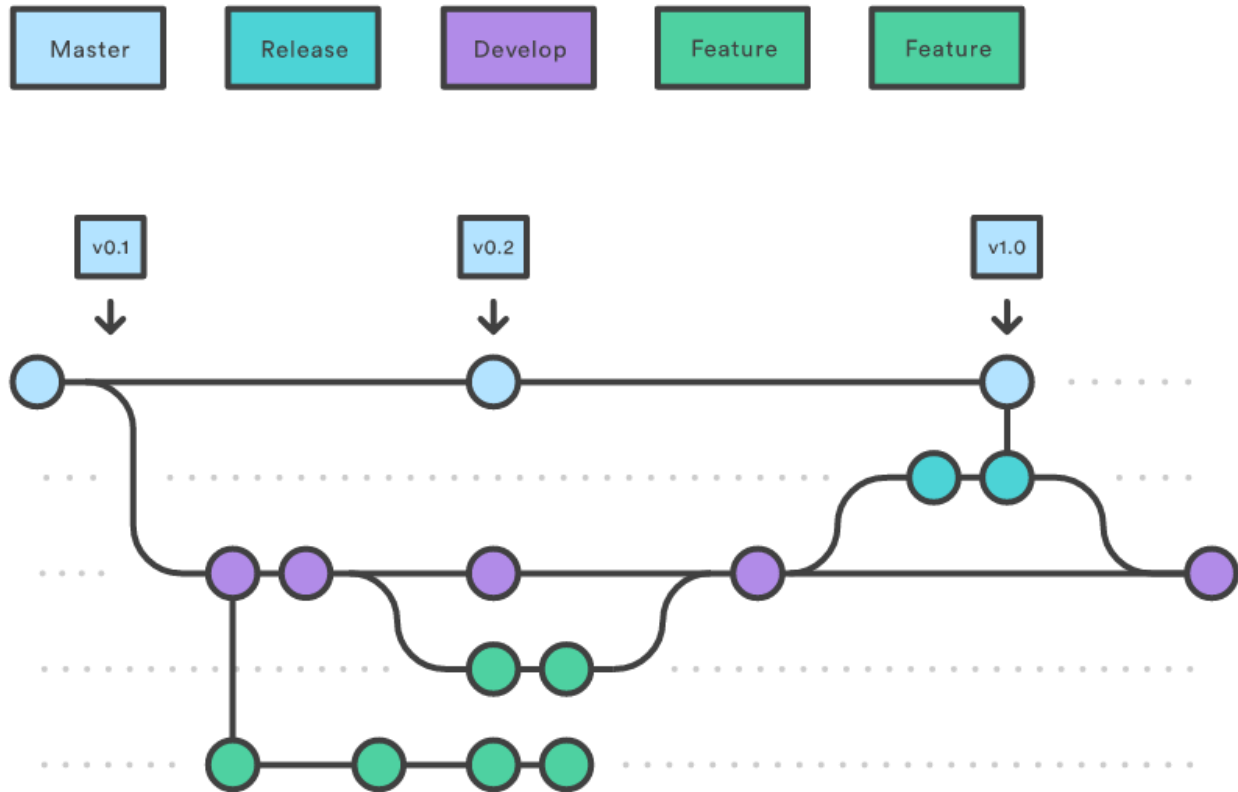


When you are adding or upgrading a feature of the application, it is a good idea for you to do this in a feature branch. In the case of group projects when you are working with multiple people, each person should take a feature and do their work independently in their own feature branch. Once the feature is completed and tested to ensure it is working exactly as expected, you will then merge this feature branch back into the develop branch. Since the develop branch may have changed since you first created your feature branch it is a good idea to test the develop branch to ensure your new feature is still working. Once you have confirmed this, you can then destroy the feature branch since all your commits are now in the develop branch.



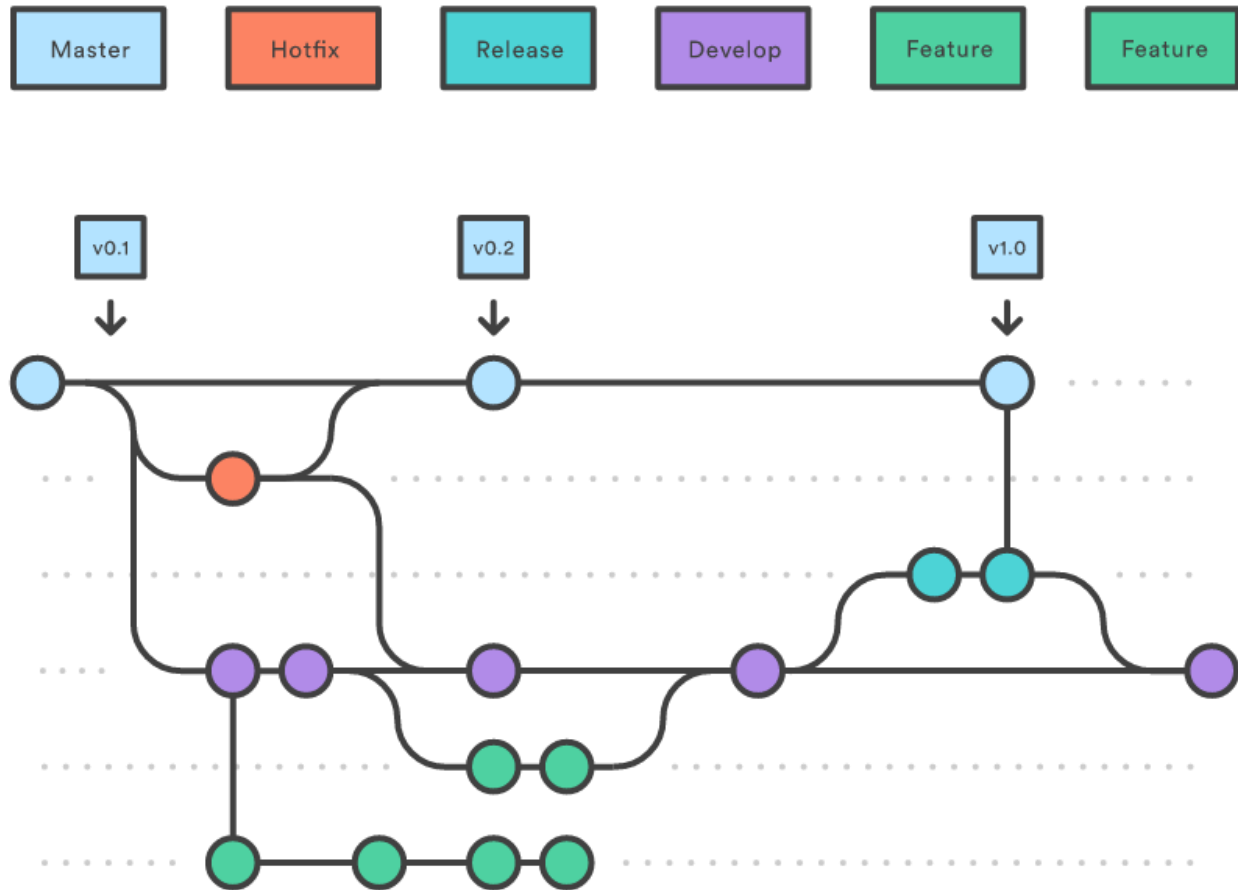
Release Branch

Once your application is tested like crazyballs and you think it is finally ready to release to the public, it is now time to create a release branch. This branch isn't always necessary. This would be for the cases when your testing/developing environment is different than the release environment. Use this release branch to make the necessary changes to get the app ready to release in the release environment. You will also use this time to update the version number of the app and add documentation. Merge this branch into master and you are good to release.



Maintenance Branches

This may not be relevant in the course of a school term but is good to know regardless. So what happens if you have a completed published version 1.0 of your application and are hard at work on version 2.0. You are well into development when a widespread app breaking bug is reported about version 1, how would you solve this. Luckily, because you are maintaining the released version in your master branch, it is a relatively easy process. First, you branch off master and create a 'hotfix' branch. Fix the problem in that and merge it back into master. If the fix is also applicable for version 2, you may want to merge it into the develop branch as well.



By following this git workflow, your team will be able to tackle massive projects simultaneously without the fear of breaking the entire project with one change. You will have the peace of mind that every commit can be reverted and every branch can be abandoned without any harm to the develop and master branches. Now start collaborating like the pros and git like a champ.