Version Control, Git, and Github

Version control is a class of tools that programmers use to manage software projects. Version control does a great many things, the two primary uses are:

- Making it easy to share codebases across many collaborators.
- Making it easy to create 'save points' in the code that can be rolled back.

Git is a free and open source software for version control. The project home is here https://git-scm.com/. Github is a hosting service that is build to host Git repositories.

While there are many different version control systems, git is incredibly popular and powerful. Many companies use git, and if you understand git it will be easy to learn another version control paradigm.

Objectives -- You Should Learn To

- Create a git repository and put it onto github
- Clone a git repository
- Fork a git repository
- Describe the difference between clone and fork
- Make a commit
- View the commit history
- View the pending changes
- Make a branch
- Merge a branch into master
- Pull changes from a remote
- Push changes to a remote

Content

Today we're going to learn by doing - lets go for it!

Excecise 1: Initalize a Repo

Git is, at it's core, just another terminal command. So open up a command line and:

1. Decide on (or create) a location for your repositories related to Galvanize.
2. Make a directory in that location `$ mkdir gitCheatSheet`
3. Move into the repository `$ cd gitCheatSheet`
4. Initalize a git repository `$ git init`

Now, we have a git repo. Simple, but what really happened?

But, in programming, nothing is truly magic. Try this:

```
$ ls -la | grep .git
drwxr-xr-x  10 Tyler  staff  340 Oct 16 09:42 .git
```

What is this directory? (and pop quiz, how did I know it was a directory?)

```
$ cd .git
$ tree
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

> Pro-tip: if you don't have tree installed, `$ brew install tree`

Here's a little cheat sheet about what's in that directory. As you become more familar with git throughout the day, and throughout the course, and further into your career (I'm still learning new git-fu all the time) many of these directories will start to feel familiar.

For now, be aware that 100% of each local git repository lives in that little .git folder. Since we just made an empty repo, lets try this:

```
$ cd ..
$ git status
$ rm -r .git
$ git status
fatal: Not a git repository (or any of the parent directories): .git
```

Without a .git directory, you do not have a git repository. But we still need a git repo to continue with our exercise, so please:

```
$ git init
```

So What is Git

Git offers developers a safety net, and tools to collaborate. Git tracks and saves changes made to everything in the repository over time. Every time a new **commit**is made, git saves that work. This gives developers tremendous freedom to experiment! Because git has a saved copy of our last working **commit** we can destroy the code as much as we want without fear that we'll never get it working again.

Lets explore commits and rollbacks quickly.

Excercise 2: Do Some Work

We want you to create a git cheat sheet, to track some of what you've learned. Lets create a markdown readme file for our `gitCheatSheet` repository.

```
touch readme.md
```

> Pro-tip: github handles files named readme.md specially. Any github repo has the readme.md file displayed on it's main page.

Now, open that file in MacDown and paste the following:

```
## Cheat Sheet

Whenever you're confused about git, come read this cheat sheet. Remember that all git

`$ git branch --help` or `$git log --help`

### Essential Git Commands

####Create a new git repository
`$ git init` - Create a new, local repository
```

```
#### Repo Status
`$ git status` - Check the status of your current repository and see which files have

`$ git diff` - __Fill Me Out__

#### Repo History
`$ git log` - __Fill Me Out__

`$ git log --oneline --decorate --color --graph --all` - __Fill Me Out__

`$ git log -p [filename]` __Fill Me Out__

#### Stage files to commit
`$ git add <filename>` - __Fill Me Out__

`$ git add -A` - __Fill Me Out__

#### Commit changes in staged files
`$ git commit -m "<commit message>"` - __Fill Me Out__

#### Branching
`$ git branch <branch name>` - __Fill Me Out__

`$ git branch` - __Fill Me Out__

`$ git checkout <branch name>` - __Fill Me Out__

#### Merging

`$ git merge <branch name>` - __Fill Me Out__
```

Once you've saved that file, try this:

```
$ git status
```

What do you see? We have one **untracked** file, which means that our git repository has *never seen that file* before. To tell git that we want to add that file to our repository, we have to **add** it. **commit** our work.

```
$ git add readme.md
$ git status
```

What do you see now? We have staged our readme, which means we can commit it now. Commiting code means making a save point that we can always roll back to. For now, we're happy with the readme, so lets commit it!

```
$ git commit -m "initial commit/added a readme"
$ git log
```

Log lets you view the commits in any given **branch** and repository. So, now we have a good baseline. Lets do

something wrong, and rollback our changes!

Try This:

```
$ touch badFile.txt
$ echo "This code does not belong" >> badFile.txt
$ git status
```

You should see that we have an "untracked" file called badFile. Add this badfile to our repo, so that we can practice rolling back.

```
$ git add .
$ git commit -m "added a bad file"
$ git log
```

We have 2 commits, but we only want the first to stay in our repository. Try this:

```
$ git reset --hard HEAD~1
$ ls
$ git log
```

As you can see, there is no record of badFile.txt anywhere. `git reset` can move you through the git log. `--hard` means "completely throw away the changes" and `HEAD~1` means "go back in time one commit from the top of the log".

We've made our first commit, and corrected our first mistake! Great job. Programming is all about experimentation and failure is the norm. In programming we fail once, then we fail better.

Instead of avoiding mistakes use git as a safety net, and mistakes as a learning experience.

Git is a Rocket Ship, Github is Mars

By using a version control system, developers know that they can always roll back code to the latest master or to any previous commit. Github acts is a remote backups service for git repositories. Once we've **pushed** to a **remote** we know our code is safe, even if our hard drive dies.

Recall the stages that readme.md and badText.txt went through:

1. unstagged.
2. staged.
3. committed.

As we'll see in a moment, there is one final stage: pushed.

A useful metaphor is that git is a rocketship. That rocketship delivers packages of our work to Mars (Github).

When code is "unstaged", we're making changes that we're not really tied to yet. We're assembling the raw materials for our next rocketship launch. It's very easy to assemble more materials than we need, and not pack them in the ship.

Eventually we'll decide what should go into the rocketship, and put those changes into boxes. This is "staged"

work. To move something from "unstaged" to "staged" we use `git add`. Once the code is 'staged' it's still easy to 'unstage' it, just as it's fairly easy to remove an item from a box. To unstage work we use `git reset HEAD <file>`.

The third stage is "committed". Once we have staged all the changes we want, we use `git commit` to put all of our packed boxes into the rocketship. Once that box is on the ship, we have to pack any new boxes *around* the existing box. Similarly, any code we add has to play nice with code that's already committed. We can still take these boxes off the rocket ship (as we did with `badFile.txt`) but it's more costly than unstaging work.

Finally, when we've 'committed' some packages to our rocketship, we want to back those changes up to Mars. We use `git push` to do this. Push sends the rocketship to space. It is **exceedingly difficult** to get your boxes back after you've already sent them to Mars.

Just like any space mission, you should be absolutely sure you have what you want before liftoff. Next, lets send off our first mission to Mars!

Excercise 4: Using a Github Remote

Github is a cloud hosting (and more) service for git repositories. We assume you all have github accounts, but if not, you'll have to make one now. There are two steps we need to take in order to get our code from our local git repo into github:

First, SSH Keys

Github uses SSH keys as a security mechanism in order to determine who you are. Github provides a wonderful tutorial on these keys so lets give it a go:

Generating SSH Keys: https://help.github.com/articles/generating-ssh-keys/

Second, Create a Remote

Again, Github has a great tutorial on this. We have already completed some of the steps here (like `$git init`, and our first commit). So skip things you've already done to your `gitCheatSheet` repository.

Creating a remote: https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/

Lets talk about the new commands you ran:

```
$ git remote add origin [remote repository URL]
$ git push origin master
```

A remote is an offsite git repository. We basically created a .git folder on github, and when we used `push` we told the remote to synchronize with our local .git folder. What do you think `git pull` does?

Add the following commands to your cheat sheet, and push your changes to your remote!

```
## Commands for working with a remote repository (e.g. Github)
```

```
`$ git clone <repo path or URL>` - clone a repository into a new directory.

`$ git remote` - List all remotes for the current repo.

`$ git remote add <remote name> <remote path or URL>` - adds a remote to your repo.

`$ git pull <remote name> <branch name>` - Pull down changes from a remote and integra

`$ git push <remote name> <branch name>` - Send your changes to the remote to be merge
```

Exercise 5 - Collaborating with Github!

Setup

Partner with 2 of your classmates to make a team of 3.

You'll notice that the cheat sheet we provided for you is not very complete. Between the three of you, we'd like to replace all of the `__Fill Me Out__`'s with descriptions of the command. Divide the commands between yourselves and start filling in the descriptions in your own readme files **for only the commands you've been given**.

> Pro-tip: The next steps will be easier if commands are divied out in blocks. e.g. person 1 takes the first 3 commands, person 2 takes the next 3 commands, person 3 takes the last 3 commands...

Don't worry if the descriptions aren't perfect, and remember to use `git [command] --help`.

When you've added the commands you've been assigned to your own readme, push your changes to your remote. Each partner should now have a repository with different commands filled in on their cheat sheets.

Wait for your partners to finish before moving on. You can help each other, but make sure everyone's cheat sheets only have your own answers filled in.

Forking

Each of your partners should now have their own cheat sheet that needs your commands. You'll need to **fork** each of your partners repositories. To fork a repository:

1. Go to your partners' GitHub repo and **fork** it using the button in the top right.
2. Clone the new forked repo (the url should have your username in it) to your computer. `$ git clone [repoUrl]` (you can find the url on the middle right of the repository's webpage)
3. Open your partner's cheat sheet, and add your command descriptions to it!
4. Add, then commit the change to your local repo.
5. Push the change up to GitHub. Your change should now be reflected in**your** forked repo.
6. Click the green pull-request button. Review the changes. Click "Create Pull Request".

7. Give your pull request a title and (optional) comment. Click "Create Pull Request" to submit a pull-request to you partner.

Congrats! You've submitted your first pull-request!

STOP! Make sure your teammates have caught up before moving on! Help each other out if any of you are stuck!

Accept Pull Requests

Go to your original cheat sheet GitHub repo (not one of your forks). You should see a pending pull-request from your partner. Review the pull-request and accept it if it looks good to you.

Congrats! You've accepted your first (and second) pull-request!

Git Workflow

Now that you've done some collaboration on git, lets introduce the idea of a **"git workflow"**.

Typically, when someone says **git workflow** they mean, the procedure by which code changes are **"merged to master"**, or the protocol for packing boxes into our ship, and controling when ships go to Mars.

Once code is in the master branch of the main repository (on Mars), a code change is said to be complete; all other developers now must base their changes on the new master.

There are many flavors of **git workflow** and they come with tradeoffs. Some of the competing values of a git workflow include:

Overhead Cost: Every git workflow is "extra work" compared to simply writing and saving the code (though I promise they will also all save you time in the long run). Some workflows are more work for individual developers to get their code "into master".

Roll-back-ablility: One of the most explicit value propsitions of version control is the ability to roll changes back if they don't work as expected.

Deployability: At places with commitment to Continous Integration (Etsy and Airbnb for example) the **master** branch must always be deployable. At other places, you may find commits in the master history that contain bugs.

Historical Granularity: Git maintains a history of *every commit*. If you make commits more frequently, your history is more granular.

Control Over Master: In a corporation, the senior engineers might want to gate everything that gets into the master branch. Some organizations give all developers admin access to all repositories.

Here is an example of a very low overhead workflow.

1. Do some work
2. `$ git status` to review your changes.
3. `$ git add <filename>` to add files to the staging area. (Use `$ git add -A` to add all files and changes.)
4. `$ git status` to see what you're about to commit

5. `$ git commit -m "<message>"` to commit the changes in the staging area with a message.

Here's another example with a little more overhead, but that is much more common in the industry:

1. `$ git checkout master` master is always the "source of truth"
2. `$ git fetch origin` to grab the latest code from the remote.
3. `$ git rebase origin/master` to rebuild our local master from the code we just fetched.
4. `$ git checkout -b new_branch_name` to create a branch for our work.
5. Do the previous workflow's steps 1-5 for committing to our branch. We might repeat these steps several times until we're happy with the code in the new branch.
6. Do THIS WORKFLOW'S steps 1-3 to ensure that our code is built on top of the latest (in case someone else made changes while we were working)
7. `$ git merge new_branch_name` to pull our branch changes into master.

It's a lot more complex, but if you're working with others keeping your branches up to date wit the remote is crucial to avoiding conflicting changes.

Even in solo projects, it can be nice to create branches for doing chunks of work so that you can just go back to master if things don't pan out. But when working alone you won't have to worry about `git fetch origin` and `git rebase origin/master`.

Advanced Git - Homework

Aliases - Make git commands easier and faster to type with command line shortcuts. For more: http://git-scm.com/book/en/v2/Git-Basics-Git-Aliases

your task is to create aliases for branch, commit, status, and checkout. Write about how to do this in your cheat sheet, and push those additions to your remote!

Additional Resources

Rebasing - Another way to merge that keeps your git history cleaner and more streamlined. For more: http://git-scm.com/book/en/v2/Git-Branching-Rebasing

Understanding Workflows - There are lots of workflows, understanding these concepts can make you a much more flexible developer, and save you time and headache as you begin to work on more complex products:

- https://guides.github.com/introduction/flow/index.html
- http://scottchacon.com/2011/08/31/github-flow.html

Extra goodies

- Pro Git (free e-book) - http://git-scm.com/book
- Try git - https://try.github.io
- Github Training Videos - https://www.youtube.com/user/GitHubGuides

- Git for Ages 4 and Up (Video) - https://www.youtube.com/watch?v=1ffBJ4sVUb4
- A Practical Git Introduction - http://mrchlblng.me/2014/09/practical-git-introduction/
- Git Branching Game - http://pcottle.github.io/learnGitBranching/
- Github Guides - https://guides.github.com/