What is GitHub?

- Hosting platform for version control and collaboration
    - Lets people work together on projects from anywhere

GitHub Flow

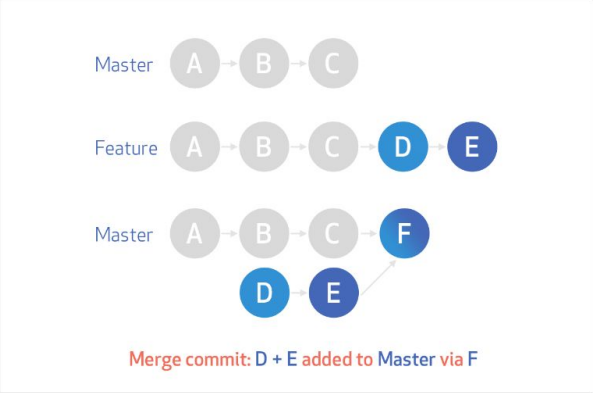
- Lightweight, branch-based workflow that supports teams and projects where deployments are made regularly.
- **Branching**
    - When you create a branch in your project, you're creating an environment where you can try out new ideas.
        - Changes you make on a branch don't affect the **master** branch.
        - Free to experiment and commit changes
        - Won't be merged until it's ready to be reviewed by someone you're collaborating with
    - New Branch should be created off of master
    - Branch names should be descriptive so that others can see what you are working on.
- **Add Commits**
    - Whenever you add, edit, or delete a file, you're making a commit, and adding them to your branch.
    - Process of adding commits keeps track of your progress as you work on a feature branch.
    - Each commit has an associated commit message
        - A description explaining why a particular change was made.
    - Each commit is considered a separate unit of change.
        - Able to roll back changes if a bug is found, or if you decide to head in a different direction.
    - Make sure to write clear commit messages.
- **Open a Pull Request**
    - Initiate discussion about your commits.
    - Anyone can see exactly what changes would be merged if they accept your request.
    - Can ask for a pull request at any time
        - Want to share some screenshots or general ideas
        - When you're stuck and need help/advice
        - When you're ready for someone to review your work
    - By using **@mention** system in your Pull Request message, you can ask for feedback from specific people or teams
    - Useful for contributing to open source projects and for managing changes to shared repositories.

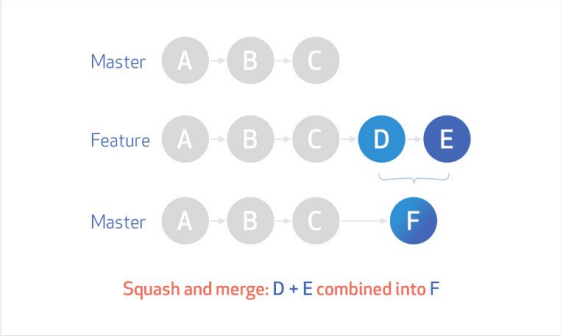
- **Deploy**
    - Deploy changes to verify them in production.
    - If your branch causes issues, you can roll it back by deploying the existing master into production
- **Merge**
    - Once merged, Pull Requests preserve a record of the historical changes to your code.
    - By incorporating certain keywords into the text of your Pull Request, you can associate issues with code.
- **Fork and Pull Model**
    - Anyone can fork an existing repository and push changes to their personal fork without needing access to the source repository.
    - **Fork**
        - A copy of a repository
        - Forking a repository allows you to freely experiment with changes without affecting the original project
        - Used to propose changes to someone else's project
            - Fork the Repo
            - Make the fix
            - Submit a **Pull Request** to the project owner
        - Use someone else's project as a starting point for your own idea
            - When creating a public repo from a fork of someone's project, make sure to include a **license file** that determines how you want you project to be shared with others.
    - In the top-right corner of a page, click **Fork** to fork a project
    - **Creating a local clone of your fork**
        - Navigate to **your fork**
        - Under the repo name, click **Clone or Download**
        - In the Clone with HTTPs section, click the clipboard to copy the clone URL for the repo
        - Open Git Bash
        - Type **git clone** and then paste the URL you copied.
        - Press **Enter** .
        - Your local clone will be created.
    - **Configure Git to sync your fork with the original Repo**
        - To **Pull Changes** from the original, or upstream repo into the local clone of your work.
        - Navigate to the original Repo
        - Under the repo name, click **Clone or Download**

        - In the Clone with HTTPs section, click the clipboard to copy the clone URL for the repo
        - Open Git Bash
        - Go to your Home Directory by typing **cd** with no other text
        - Type **ls** to list the files and folders in your current directory
        - Then type **cd <your directory>**
        - To go up one directory, type **cd ..**
        - Type **git remote -v** and press **Enter**.
            - You'll see the current configured remote repo for your fork
        - Type **git remote add upstream**
        - Paste the URL you copied and then press **Enter**
        - Then type **git remote -v** again
            - Your fork URL should say **origin**
            - URL for original repo as **upstream**
    - **Syncing a Fork**
        - To Keep it up-to-date with the upstream repo
        - Open Git Bash
        - Change the current working directory to your local project
        - Fetch the branches and their respective commits from the upstream repo
            - Commits to **master** will be stored in a local branch, **upstream/master**
            - **git fetch upstream**
        - Check out your fork's local **master** branch
            - **git checkout master**
        - Merge the changes from **upstream/master** into your local **master** branch.
            - This brings your fork's **master** branch into sync with the upstream repo without losing your local changes
            - **git merge upstream/master**
        - If you local branch didn't have any unique commits, Git will instead perform a "fast-forward"
        - Syncing your fork only updates your local copy of the repo
        - To update your fork on GitHub, you must **push your changes.**
- **Shared Repository Model**
    - Collaborators are granted push access to a single shared repository and topic branches are created when changes need to be made.
    - Pull requests are useful in this model as they initiate code review and general discussion about a set of changes before the changes are merged into the main development branch.

- Model is more prevalent with small teams and organizations collaborating on private projects.
- Use a Topic Branch for your pull requests
    - Can push follow-up commits if you need to update your proposed changes.
    - When pushing commits to a pull request, don't force push as it can corrupt your pull request.

Pull Request Merges

- **Merging pull request by retaining all the commits in a feature branch**
    - Request is merged using the **--no-ff** option
    - Must have **write permissions** in the repo
    - 

- **Squash and Merge**
    - The pull request's commits are squashed into a single commit.
    - Instead of seeing all of a contributor's individual commits from a topic branch, the commits are combined into one commit and merged into the default branch.
    - These are merged using the fast-forward option
    - Repo must allow squash merging
    - 

    - Can be used to create a more streamlined Git history in your repo.
        - Work in progress commits are helpful when working on a feature branch but aren't necessarily important to retain in the Git History.

        - If you squash these commits into one commit while merging to the default branch, you can retain the original changes with a clear Git history
- **Rebase and Merge your pull request commits**
    - All commits from the topic branch (or head branch) are added onto the base branch individually without a merge commit.
    - Merged using the fast-forward option
    - Repo must allow rebase merging
    - You will not be allowed to rebase and merge on GitHub when:
        - The pull request has merge conflicts
        - Rebasing the commits from the base branch into the head branch runs into conflicts
        - Rebasing the commits is considered “unsafe”

How To’s

- **Pull Requests**
    - Changes are proposed in a branch, which ensures that the **master** branch only contains finished and approved work.
    - Pull requests can only be opened if there are differences between your branch and the upstream branch.
        - You can specify which branch you’d like to merge your changes into when you create your pull request
    - **Changing the branch range and destination repository**
        - By default, pull requests are based on the parent repo’s **default branch**
        - If the default parent repo isn’t correct, you can change both the parent repo and the branch with the drop-down lists.
        - **Base branch**
            - Where the changes should be applied
        - **Head Branch**
            - What you would like to be applied
        - When you change the base repo, you can also change notifications for the pull requests.
        - Everyone that can push to the base repo will receive an email notification and see the new pull requests in their dashboard the next time they sign in.
    - **Creating the Pull Request**
        - Navigate to the main page of the repo
        - In the “Branch” menu, choose the branch that contains your commits.
        - To the right of the Branch menu click **New Pull Request**
        - Use the **base** branch dropdown menu to select the branch you’d like to merge your changes into

        - Use the **Compare** branch dropdown menu to choose the topic branch you made your changes in.
        - Type a title and description of your pull request
        - Click **Create Pull Request**.
        - Once you've created a pull request, you can push commits from your topic branch (feature branch) to add them to your existing pull request.
            - These commits will appear in chronological order within your pull request
            - The changes will be visible in the "Files Changed" tab
        - Contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.
        - After you're happy with the proposed changes, you can **merge the pull request.**
    - **Closing a Pull Request**
        - Under your repo name, click Pull Request
        - Choose the pull request you would like to close
        - At the bottom of the pull request, click **Close Pull Request**
        - Optionally, **delete the branch**. This keeps the list of branches in your repo tidy.
    - **Merging a Pull Request**
        - Anyone with push access to the repository can complete the merge.
        - Repo admins can require that all pull requests receive at least one approved review from someone with write or admin permissions or from a designated code owner before they're merged into a protected branch.
        - Under your Repo name, click Pull Request
        - Click the pull request you'd like to merge
        - Depending on the merge options enabled for your repo, you can:
            - **Merge all of the commits into the base branch** by clicking **Merge Pull Request**.
                - If the Merge Pull Request option isn't shown, then click the merge drop down menu and select **Create a Merge Commit**
            - **Squash the commits into one commit** by clicking the merge drop down menu, selecting **Squash and Merge** and the clicking the **Squash and Merge** button.
            - **Rebase the commits individually onto the base branch** by clicking the merge drop down menu, selecting **Rebase and Merge** and then clicking the **Rebase and Merge** button.

        - If you clicked **Merge Pull Request/Squash and Merge,** type a commit message or accept the default message.
            - Click **confirm merge** or **confirm squash and merge**
        - If you clicked **Rebase and merge**, click **Confirm rebase and merge**
        - Optionally delete the branch to keep things tidy
- **Branches**
    - **Default Branch**
        - The **base** branch in your repo against which all pull requests and code commits are automatically made unless you specify a different branch.
        - It is named **master.**
        - If you have admin rights over a repo, you can change the default branch on the repository.
    - **Creating a Branch**
        - Navigate to the main page of the repo
        - Click the branch selector menu
        - Type a unique name for your new branch
        - Press Enter
    - **Deleting a Branch**
        - Navigate to the main page of the repo
        - Above list of files, click the symbol NUMBER branches
        - Scroll to the branch that you want to delete, then click the trash icon
    - **Setting the Default Branch**
        - Navigate to the main page of the Repo
        - Under your repo name, click settings
        - In the left menu, click **Branches**
        - In the default branch sidebar, choose the new default branch.

Git Command Shell

- Cloning
    - **git clone <url> <Optionally Include the name of the file it should be>**
    - Example:
        - **git clone https://github.com/libgit2/libgit2 mylibgit**
    - Creates a directory named “libgit2”, initializes a .git directory inside it, pulls down all the data for that repo, and checks out a working copy of the latest version.
- Recording changes to the Repository
    - Want to make changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.
    - Each file in your working directory can be in one of two states:
        - **Tracked**

            - Files that were in the last snapshot
            - Can be unmodified or staged
        - **Untracked**
            - Everything else
            - Any files in your working directory that were not in your last snapshot (commit)
            - Files that are not in your staging area.
    - When you first clone a repo, all your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.
    - As you edit files, Git sees them as modified since it has been changed since the last commit.
        - Stage these modified files and then commit all your staged changes
    - Use **git status** command to check the status of your files
    - Use the command **git add <fileName>** to begin tracking it
        - The file is now tracked and staged to be committed
        - If you commit now, the version of the file at the time you ran **git add** is what will be in the historical snapshot.
- Ignoring Files
    - A class of files that you don't want Git to automatically add or even show you as being untracked.
        - Usually automatically generated files such as log files or files produced by your build system.
    - Create a file listing pattern to match them named **.gitignore**
        - **cat .gitignore**
        - ***.[oa]**
            - Tells Git to ignore any files ending in ".o" or ".a"
        - ***~**
            - Ignore all files that end with a ~
    - Setting up a **.gitignore** file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.
    - Examples:
        - ```
          # no .a files
          *.a

          # but do track lib.a, even though you're ignoring .a files above
          !lib.a

          # only ignore the TODO file in the current directory, not subdir/TODO
          /TODO
          ```
          ```

```
# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

- **Viewing Your Staged and Unstaged Changes**
    - To know exactly what you changed, not just which files were changed, you can use the **git diff** command
        - Shows you the exact lines added and removed
        - What have you changed but not yet staged?
        - What have you staged that you are about to commit?
    - To see what you've staged that will go into your next commit, you can use **git diff-- staged**.
        - Compares your staged changes to your last commit.
- **Committing Your Changes**
    - **git commit**
        - Will launch your editor of choice.
        - Type the message for your commit
        - You can pass the **-v** option to **git commit** to put the diff of your change in the editor so you can see exactly what changes you're committing.
        - Exit the editor
- **Skipping the Staging Area**
    - If you want to skip the staging area, Git provides a shortcut:
        - **git commit -a** makes Git automatically stage every file that is already tracked before doing the commit
        - Lets you skip the **git add** part
- **Removing Files**
    - To remove a file from Git, you have to remove it from your staging area and then commit.
        - **git rm <file name>** removes the file from your working directory so you don't see it as an untracked file
        - To force the removal of a file because you modified the file and added it to the staging area already, use the **-f** option.
    - To keep the file on your hard drive but not have Git track it anymore, use the **--cached** option

- **Moving Files**
    - To rename a file use **git mv file_from file_to**
- **Viewing the Commit History**
    - After several commits or if you have cloned a repo with an existing commit history, you'll probably want to look back to see what has happened.
        - **git log** lists the commits made in that repo in reverse chronological order, meaning the most recent shows up first.
        - **git log -p** shows the difference introduced in each commit
            - **-2** option added limits the output to only the last two entries
        - **--stat** option shows a list of modified files, how many files were changed, and how many lines in those files were added and removed.
        - **Author** is the person who originally wrote the work
        - **Committer** is the person who last applied the work
        - https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History for customization of the history
- **Undoing Things**
    - When you commit too early and possibly forget to add some files, or you mess up your commit message. To try the commit again, you can run commit with the **--amend** option
        - **git commit --amend**
        - Command takes your staging area and uses it for the commit
        - If you want to change your message, use this command and the text editor will pop up
- **Unstaging a Staged File**
    - **git reset HEAD <filename>** will let the file still be modified but unstaged
- **Un-modifying a Modified File**
    - **git checkout -- <filename>**
    - This command is dangerous, any changes you made will be absolutely gone
- **Working with Remotes**
    - Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
    - Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.
    - **Showing Your Remotes**
        - **git remote**
            - Shows which remote servers you have configured
            - Lists the short-names of each remote handle you've specified
        - **git remote -v**

            - Shows you the URLs that Git has stored for the short-name to be used when reading/writing to that remote
    - **Adding Remote Repositories**
        - How to add a new remote explicitly
        - **git remote add <shortname> <url>**
            - To add a new Git repository as a short-name you can reference easily
            - Now you can use the short-name string on the command line of the whole URL
    - **Fetching and Pulling from your Remotes**
        - **git fetch <shortname> or [remote-name]**
            - Fetch all the information that <shortname> has but that you don't yet have in your repository
        - **git pull**
            - Automatically fetch and then merge that remote branch into your current branch
    - **Pushing to Your Remotes**
        - **git push [remote-name] [branch-name]**
            - When you want to share a branch with the world
            - Must have write access to
            - **git push [remote name] [branch name]:[new name]**
                - To change the name of the branch after you push it
    - **Inspecting a Remote**
        - **git remote show [remote-name]**
            - If you want to see more information about a particular remote
    - **Removing and Renaming Remotes**
        - **git remote rename [shortname] [new name]**
            - To change a remote's short-name
        - **git remote remove** or **git remote rm**
            - To remove a remote
- **Tagging**
    - Tag specific points in history as being important
    - People can use this functionality to mark release points such as v1.0 and so on
    - **Listing your Tags**
        - **git tag**
            - Lists the available tags in alphabetical order
        - **git tag -l "pattern"**
            - Example : **git tag -l "v1.8.5*"**
    - **Creating Tags**

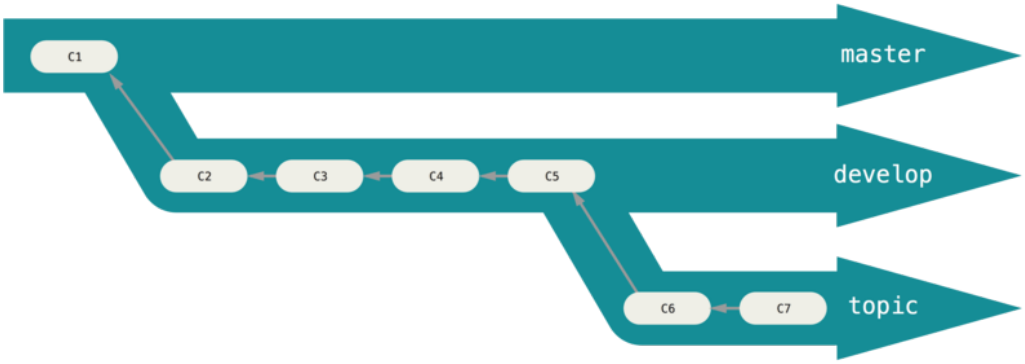
        - There are two main types of tags, **LightWeight** and **Annotated**
        - **LightWeight**
            - Like a branch that doesn't change, a pointer to a specific commit
            - **git tag [name]**
        - **Annotated**
            - **git tag -a [name]**
                - May specify **-m** for a tagging message
                    - **git tag -a v1.4 -m "my version 1.4"**
                - Stored as full objects in the Git database
                - Are checksummed
                - Contain the tagger name, email, and date
                - Have a tagging message
                - Can be signed and verified with GNU Privacy Guard
            - **git show [name]**
                - Shows the tagged information
    - **Tagging a Past Commit**
        - **git tag -a [name] commit-checksum**
            - Example: **git tag -a v1.2 9fceb02**
    - **Sharing Tags**
        - **git push origin [tagname]**
        - **git push origin --tags**
            - Transfers all of your tags to the remote server that are not already there
    - **Checking out Tags**
        - To view the versions of files a tag is pointing to
        - **git checkout 2.0.0**
- **Git Aliases**
    - If you do not want to always type the whole command
    - **git config --global alias.[abbreviation] command-name**
        - Example: **git config --global alias.co checkout**

Git Branching and Merging

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- **Creating a new Branch and Switching to it at the same Time**
    - **git checkout -b <branch name>**
- **Creating a New Branch**
    - **git branch <branch name>**
- **Switching Branches**
    - **git checkout <branch name>**

    - To switch to an existing branch
    - If your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches.
    - When you switch branches, Git resets your working directory to look like it did the last time you committed on that branch.
        - It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.
- **Deleting a Branch**
    - **git branch -d <branch name>**
    - If the master branch points to the same place as the branch
    - If you no longer need it
- **Merging**
    - **git merge <branch name>**
    - Make sure that when you use this command, to **checkout** into the appropriate branch to merge it with.
        - Can use **git merge master** from another branch to pull in the changes made by the master branch into your branch.
    - **Merge Commit**
        - Git determines the best common ancestor to use for its merge base
    - **Fast Forward Merging**
        - When you try to merge one commit with a commit that can be reached by following the first commit's history
        - Just moves the pointer forward because there is no divergent work to merge together
    - **Merge Conflict**
        - Run **git status** to see which files are unmerged at any point after a merge conflict
        - Git adds standard conflict resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts
            - Everything above the "======" is the version in **HEAD**
            - Everything below the "======" is the other conflicting version
        - After you resolved **each of these section in <u>each conflicted file</u>**, run **git add** on each file to mark it as resolved.
            - Staging a file marks is as resolved in Git.
        - **Opening a Graphical Tool to Resolve Issues**
            - **git mergetool**
- **Branch Management**
    - **Listing of your current branches**
        - **git branch**

        - The * behind a name indicates that if you commit at this point, the **master** branch will be moved forward with your new work.
    - **To see the last commit on each branch**
        - **git branch -v**
    - Options **--merged** and **--no-merged** can filter the list to branches that you have or have not yet merged into the branch you're currently on.
- **Branching Workflows**
    - **Long Running Branches**
        - You can have several branches that are always open and that you use for different stages of your development cycle
        - Having only code that is entirely stable or that will be released in the **master** branch
        - They have parallel branches called **develop** or **next** that they work from or use to test stability
            - They are used to pull in **topic branches,** short lived branches
            - Make sure they pass all the tests and don't introduce bugs
        - Some larger projects also have a **proposed** or **pu** branch that has integrated branches that may not be ready to go into the **next** or **master** branch.
        - Idea is that your branches are at various levels of stability, when they reach a more stable level, they're merged into the branch above them.



    - **Topic Branches**
        - A short lived branch that you create and use for a single particular feature or related work
        - Do a few commits on them and delete them directly after merging them into your main branch or above branch.
        - Your work gets separated into silos where all the changes in that branch have to do with that topic
            - Easier to see what has happened during code review
- **Git Branching**

- **Remote Branching**
  - Remote references are pointers in your remote repositories, including branches, tags, and so on.
  - **git ls-remote [remote]**
    - Retrieves a full list of remote references explicitly
  - **git remote show [remote]**
    - Displays remote branches as well as more information
- **Remote-Tracking Branches**
  - References to the state of remote branches
  - Reminds you where the branches in your remote repositories were the last time you connected to them
  - Take the form **<remote>/<branch>**
    - **origin/master**
    - If you wanted to see what the **master** branch on your **origin** remote looked like as of the last time you communicated with it
- **Updating Your Remote References**
  - **git fetch**
    - You do not get the branch, you get a pointer
    - Cannot edit these
    - Will fetch down all the changes on the server that you don't have yet, but will not modify your working directory
    - Will only let you get the data, will leave merging to yourself
  - **To merge a branch into your current working branch**
    - **git merge <branch name>**
  - **To create one of your own branches that you can work on, you can base it off your remote-tracking branch**
    - **git checkout -b <branch name> <remoteName>/<branch>**
      - git checkout -b serverfix origin/serverfix
      - Your new branch serverfix will now automatically pull from origin/serverfix
- **Tracking Branches**
  - Checking out a local branch from a remote-tracking branch automatically creates what is a called a **tracking branch**
    - Tracking branches are local branches that have a direct relationship to a remote branch.
      - If you are on a tracking branch and type **git pull,** Git automatically knows which server to fetch from and branch to merge into

            - The branch that a **tracking branch** tracks is called an **upstream branch**
        - If you have a local branch and want to set it to a remote branch you just pulled down or want to change the upstream branch you're tracking
            - **-u or --set-upstream-to** option to **git branch**
                - git branch -u origin/serverfix
        - **To see what tracking branches you have set up**
            - **git branch -vv**
            - Gives a list of your local branches and what branch they are tracking
            - **Ahead**
                - Means that it has a # of commits that are not pushed to the server
            - **Behind**
                - There is a # of commits that we haven't merged in yet
            - These numbers are not up to date, to keep them update them, you have to fetch from all your remotes before
                - **git fetch --all**
    - **Pulling**
        - **git pull**
            - A **git fetch** command immediately followed by a **git merge**
            - If you have a tracking branch set up either by explicitly setting it or by having it created for you by the **clone** or **checkout** commands, **git pull** will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.
    - **Deleting Remote Branches**
        - You can delete a remote branch using the **--delete** option to **git push.**
            - Example: deleting serverfix branch from the server
                - **git push origin --delete serverfix**
    - **Rebasing**
        - There are two main ways to integrate changes from one branch into another
        - To take all the changes that were committed on one branch and replay them on another one.
            - Goes to the common ancestor of the two branches,
            - getting the diff introduced by each commit of the branch you're on,
            - saving those diffs to temporary files,

            - Resetting the current branch to the same commit as the branch you are rebasing onto
            - Applying each change in turn
        - **Steps:**
            - **git checkout <branch>**
                - Moves to the branch specified
            - **git rebase <branch to rebase with>**
            - **git checkout <branch to rebase with>**
            - **git merge <branch>**
    - **More interesting rebases**
        - https://git-scm.com/book/en/v2/Git-Branching-Rebasing
        - Needs pictures to clarify
    - **Do not rebase commits that exist outside your repository**
        - When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different

Git on the Server- Protocols

- First choose which protocols you want your server to communicate with.
    - **Local Protocol**
        - The remote repository is in another directory on disk.
        - Uses if everyone on your team has access to a shared file system such as an NFS mount.

Git fixing errors

- **git checkout -f**
- Work by creating branches and merge once you are done, if you have made a mistake, you can erase the branch
- **git checkout sha#**
    - Use this technique to inspect the state of the project and figure out any necessary changes, then check out the **master** branch

Git Commands

- **git init**
    - Initializes a folder to become a git repository
- **git clone**
    - **git clone <clone url> customRepoName**
- **git log**
    - Prints out a log of all of the commits and works like the **less** command
    - **-p** option
        - The default only shows a very summarized version, with this option, you will be able to see an extended detailed log of each commit

- **git diff**
    - Default shows the difference between the last commit and unstaged changes in the current project
    - **--staged** option
        - To see the differences between staged changes and the previous version of the repo
    - **git diff branch-1 branch-2**
        - Used to show differences between branches
    - **git diff branch**
        - Used to show differences between the current branch and **branch**
- **git show <SHA>**
    - Shows the differences applied to that commit
- **git remote add origin repositoryURL**
    - Sets the **repositoryURL** in Github as the origin
- **git add -A**
- **git commit -m**
    - **git commit --amend -m "Fixed Message"**
        - If the commit only exists in your local repository and has not been pushed to GitHub, you can amend the commit message
- **git status**
- **git push**
    - **git push -u origin master**
        - Sets whatever origin repository setup as the **upstream respository**
    - **git push -u origin branchName**
        - Sets a **branchName** branch as the upstream repository for this branch
        - If you are on this branch, subsequent **git push**'s will directly push to **branchName**
- **git pull**
- **git checkout branchName**
    - Moves to **branchName**
    - **git checkout -b branchName**
        - Creates a new branch with the name **branchName** and switches to it
    - **git checkout -f**
        - Forces Git to checkout HEAD and wipeout all of the changes you've made
        - **HEAD** is the state of the repository as of the most recent commit
        - If you want to erase a new file that was created, you must first stage the file and then use this command
    - **git checkout sha#**
        - Checks out an earlier version of the repository based on **sha#**

    - **git checkout -b gh-pages**
        - Creates a branch **gh-pages**
        - Allows the creation of GitHub Pages along with the following command at the location http://username.github.io/reponame
            - **git push -u origin gh-pages**
- **git branch**
    - Shows the list of all branches with an asterisk * indicating the currently checked-out branch
    - **git branch branchName**
        - Creates **branchName**
    - **git branch -d branchName**
        - Deletes the topic branch **branchName** if and only if it has been merged into the master branch
    - **git branch -D branchName**
        - Deletes the topic branch **branchName** even if its changes are unmerged
- **git merge**
    - **git merge branchName**
        - Merges in **branchName** with the current branch