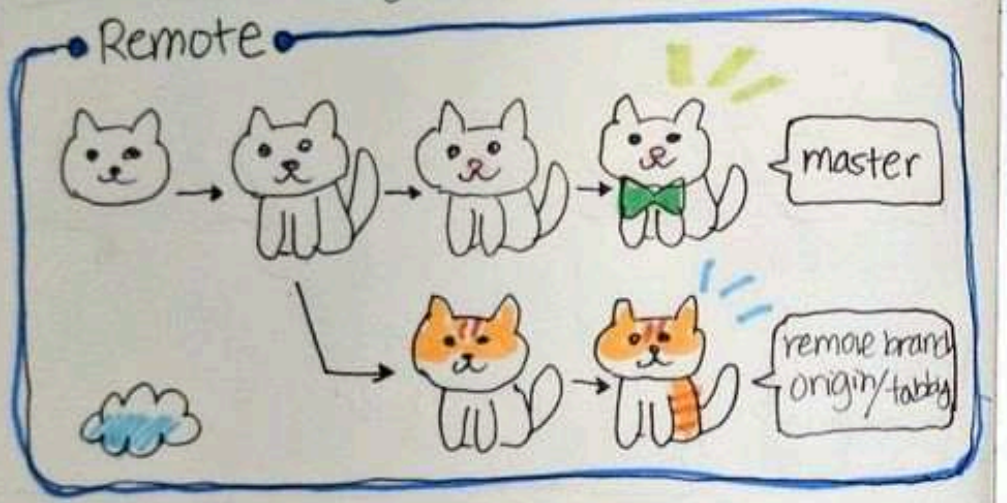
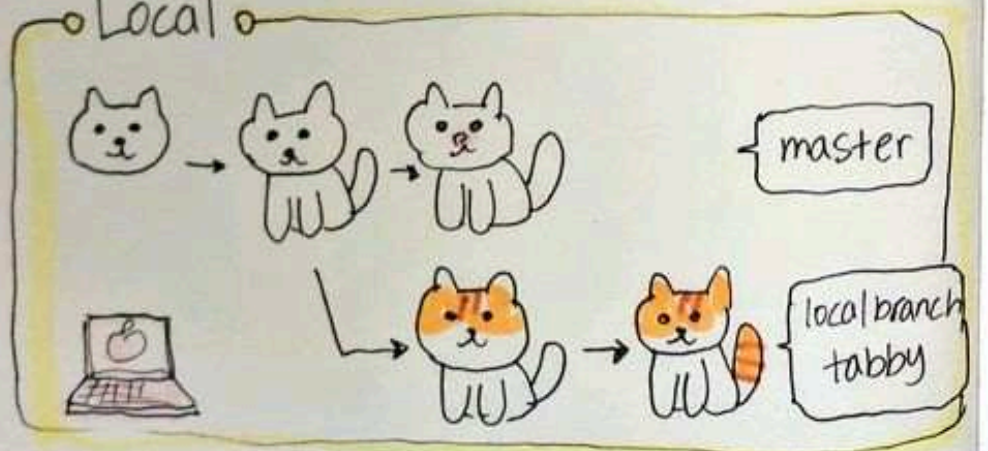
git Pull

purr

@girlie_mac

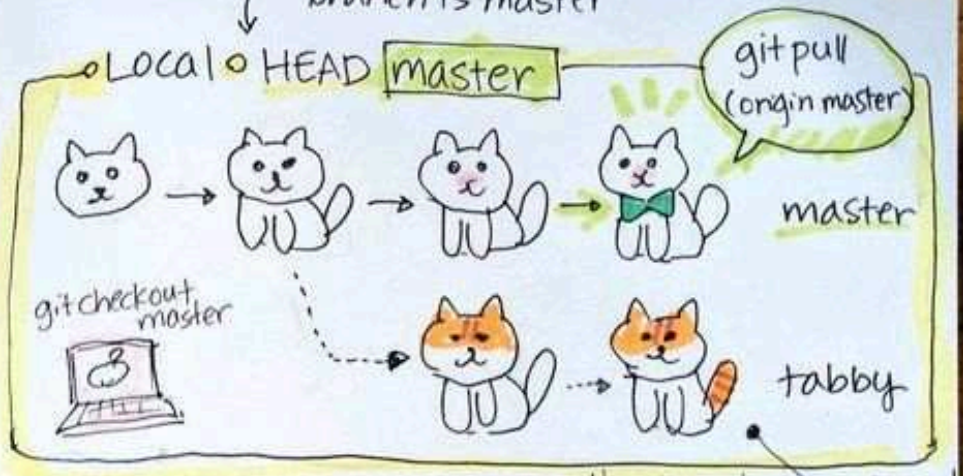
git pull updates your current HEAD branch w/ the latest Δs from remote





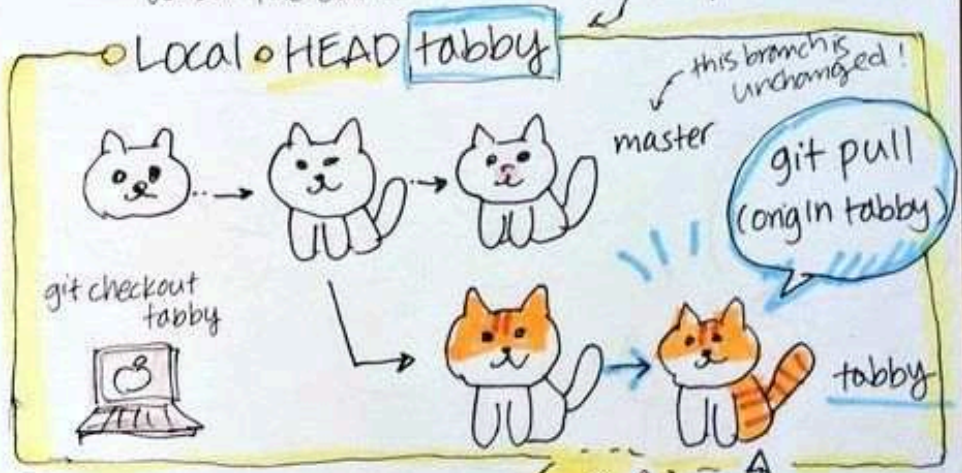
Now I'm going to 'git pull' to update a branch!

When the current branch is master



This is same as:
- git fetch origin
- git merge origin/master

When the current branch is tabby



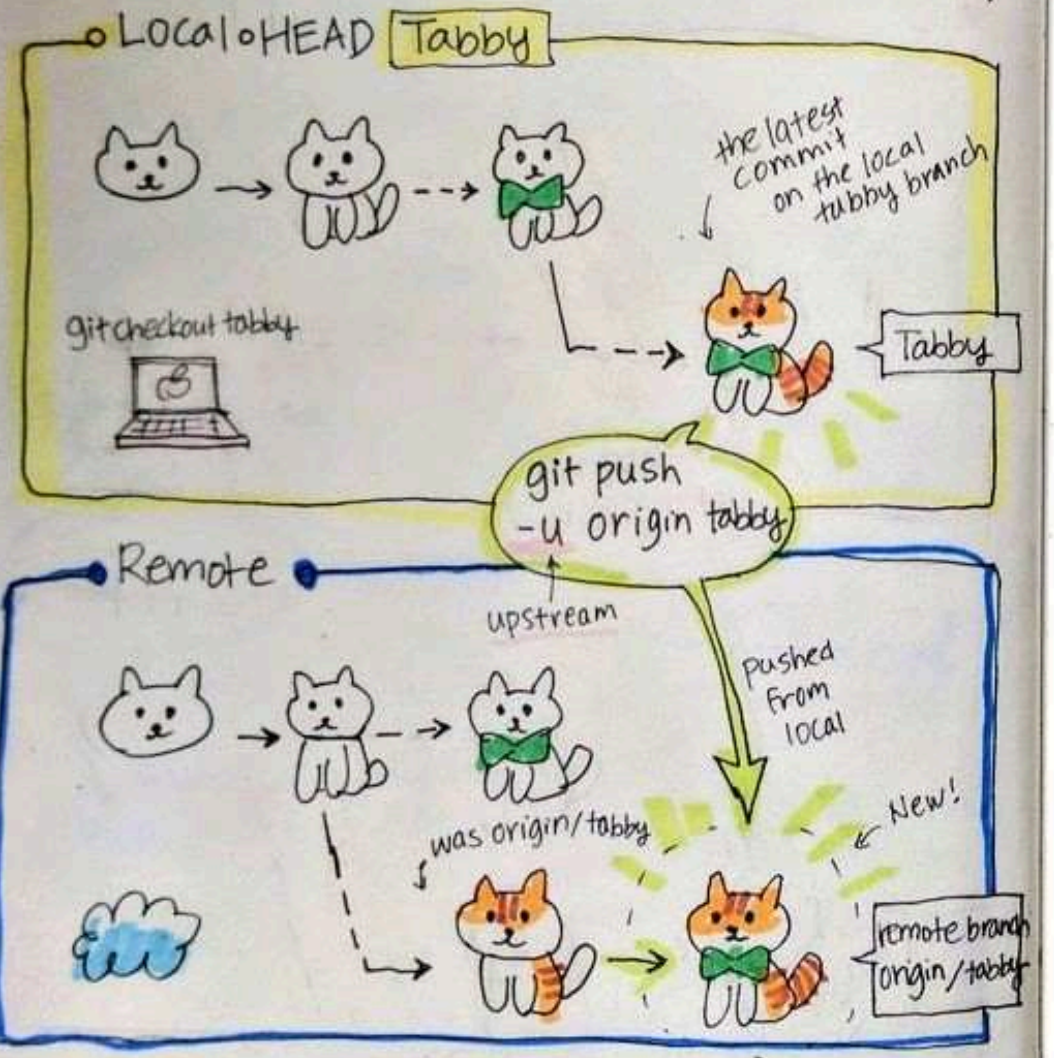
Same as:
- git fetch origin
- git merge origin/tabby

git Push

puss

@girlie_mac

♥ git push transfers your commits from your local repo to a remote repo!



Now the tabby branch on local + remote are sync'd!

♥ git push

- `git push <remote> <branch>`
  - e.g. origin
  - e.g. tabby

  push the specified branch w/ all commits

- `git push -u <remote> <branch>`
  - ↳ = --set-upstream

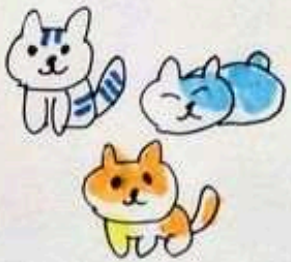
  the -u flag sets up the association between your branch + the remote branch explicitly.
  you don't need it once you've done!

- `git push -f <remote> <branch>`
  - ↳ --force

  force the push, even if it results in a non-fast-forward merge.
  just ignore + push!

- `git push --all <remote>`

  push all of your local branches!

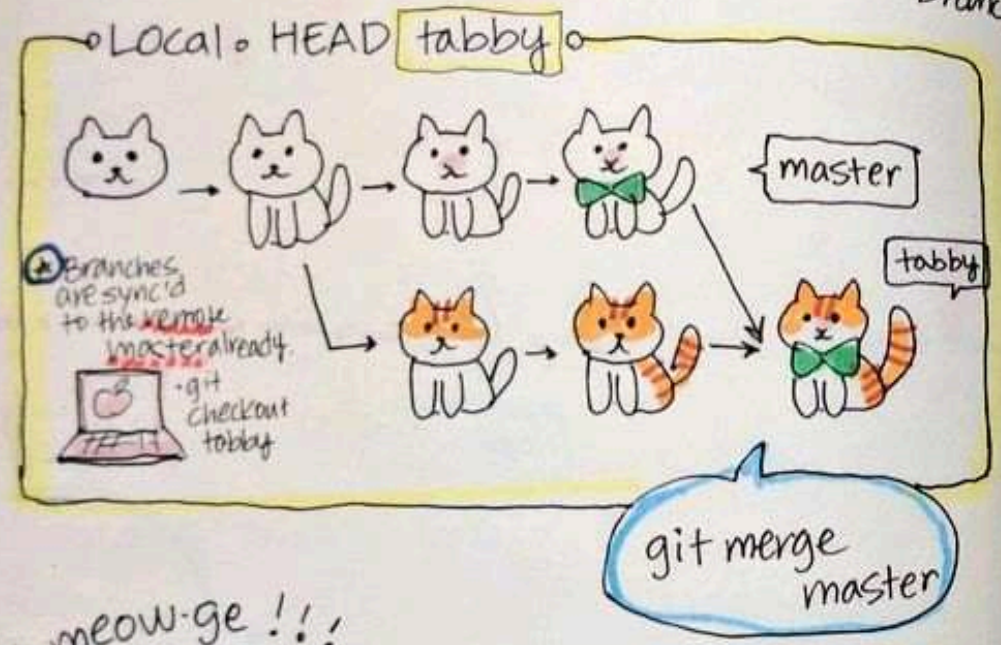


git merge

meow! ge

@girlie_mac

♥ git merge incorporates changes into the current branch.



git meow-ge !!!

★ merge is like having 2 parents & 1 resulting child !

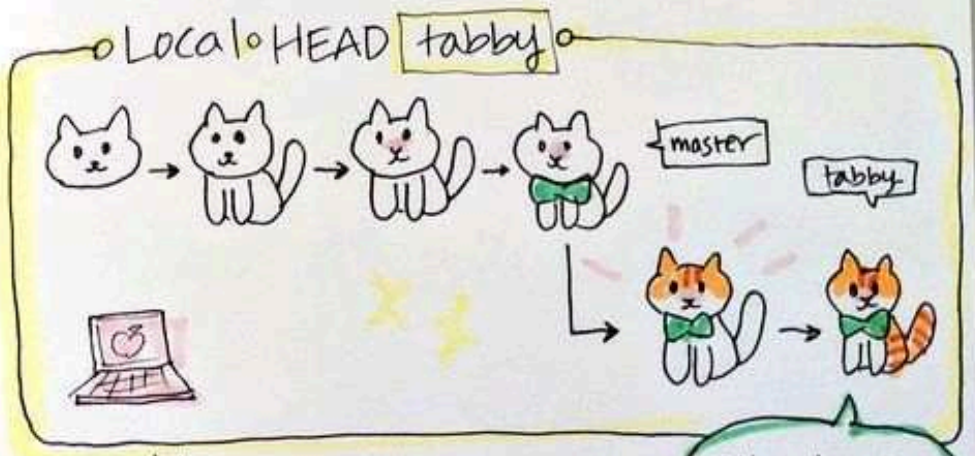
★ rebase adds all new △s on top of one parent.

PAWSOME!

git rebase

@girlie_mac

♥ git rebase moves a branch from one commit to another.



Cleaner commit history!

git rebase master

If you want to rebase from the remote master instead of local. do either:

$ git fetch origin
$ git rebase origin/master

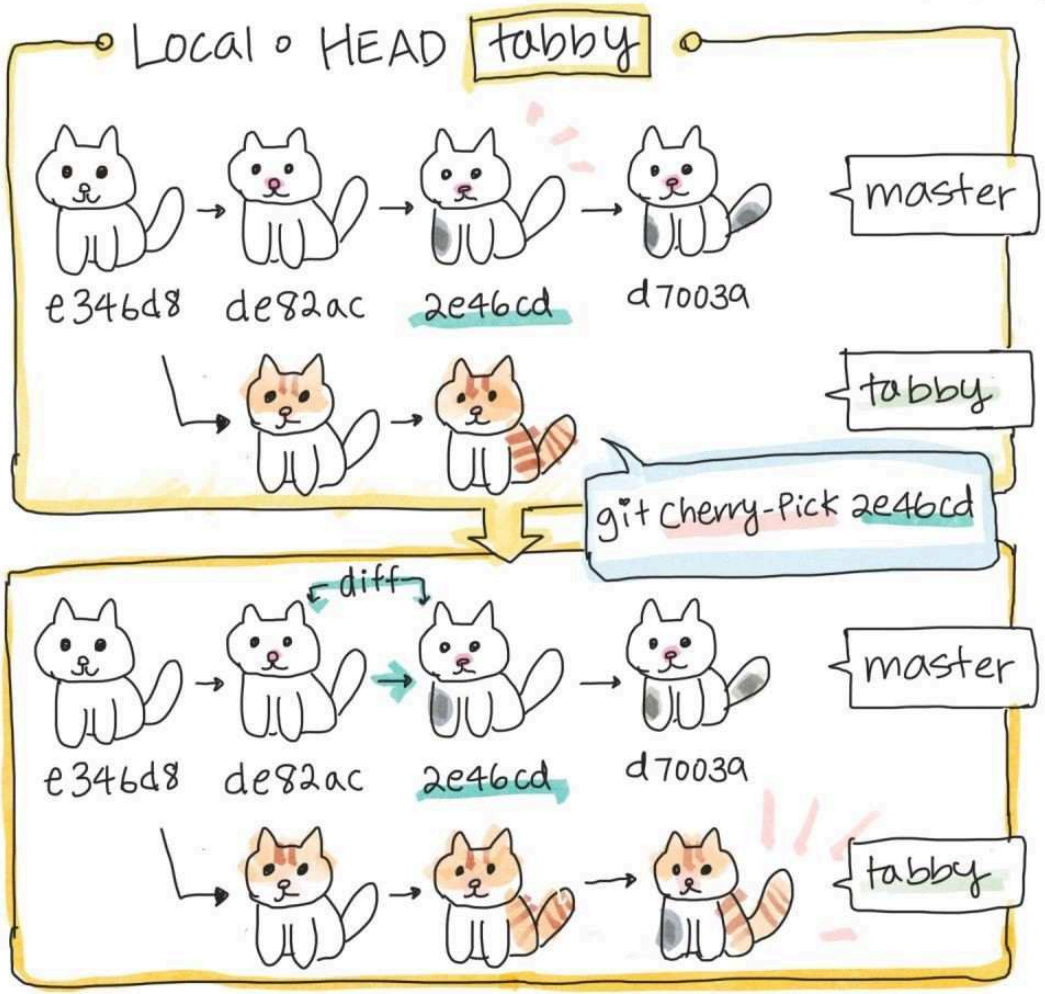
or

$ git pull --rebase origin master

git cherry-pick

@girlie_mac

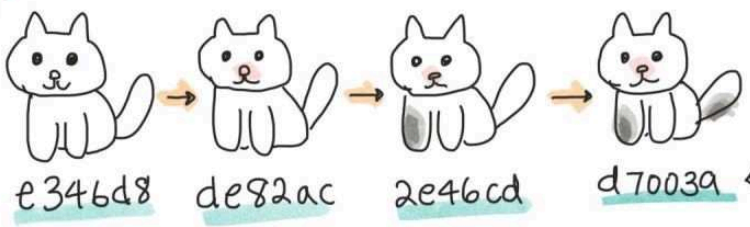
♥ git cherry-pick lets you grab some commits from one branch & apply it into another branch!



git log

♥ git log lets you view the commit history

@girlie_mac



♥ git log prints out

```
Commit e346d8
Author: Little-princess Leia <leia@kitty.cat>
Date : Sun Sept 23 17:30:42 2018 -0700
    Add body
commit de82ac
Author; Jamie <jam@kitty.cat>
Date :
....
```

git cherry-pick this SHA into the tabby branch

Simpler log with:

♥ git log --oneline

```
e346d8 Add body
de82ac Edit nose
2e46cd Add gray dot on leg
```

Git & GitHub 101

Basic CLI Commands

1. To list all files or folder in a folder

   `ls`

2. Make a new folder

   `mkdir folder_name`

3. Go inside a folder

   `cd folder_name`

4. To delete a whole non-empty directory/folder

   `rm directory_name -rf`

5. Write a file in Git Bash Vim

   `vim file_name`

   1. use insert key to enable the writing mode in any file
   2. then after finishing edits, press the left-right arrow key to disable the writing mode and then write `:x` to exit out

6. Copy + Paste in CLI

   1. Use the insert key to paste in CLI or highlight the statement then right click and copy that statemen and then right-click on CLI shows the options.

Basic Git Commands

1. To make a new file

   `touch names.txt`

2. To check if git is installed in your PC

   `git`

3. To initialize an empty Git repository in your folder

   `git init`

4. To view the changes or the untracked files in the project that's not been saved yet

   `git status`

5. Staging the files

   `git add file_name` or `git add .` (to stage everything in the current folder)

6. Committing the files

Working with Existing Projects on GitHub

Use Git Bash for Windows.

You can't directly change the contents of a repo unless you have access to it. To solve this, you create a copy (fork) of this project in your own account. In our own copy, we can do anything we want with it. After forking, we:

1. Cloning the forked project to local machine

   `git clone forked_repo_url`

2. The public repo that we forked out local copy from is known as the upstream url. We can save it as

   `git remote add upstream insert_upstream_url`

3. Creating a new branch

   `git branch branch_name`

4. Then shift the head to the above branch using the checkout command

5. Then stage. Then commit.

6. Then push. We can't push to upstream (no access). Can push to our forked repo though (origin)

   `git push origin your_branch_name`

7. Always make different branches for different pull requests if you're working on different features. 1 branch = 1 pull request (never commit on main (2))

8. To remove a commit

   1. we can remove a commit with the reset command Now it's unstaged.
   2. then add. to stage the remaining files
   3. then we can use the stash command to stash it elsewhere
   4. then, we'll have to force push this branch since the online repo contains a commit which the local repo does not

   `git push origin your_branch_name -f`

9. To make forked project even (updated) with the main project

```
git commit -m "your_message_here"
```

7. To unstage or remove a file from the staging level

   `git restore --staged file_name.txt`

8. To view the entire history of the project

   `git log`

9. Removing a commit from the history of a project

   `git reset insert_commit_hash_id_to_which_you_want_to_go_back_to_here` (all the commits or changes before this will go back to the unstaged area now)

10. After you stage a few files but then you want to have a clean codebase or reuse those files later, we can stash those changes to go back to the commit before they were staged

    `git stash`

11. Bringing back those changes or pop them from the stash

    `git stash pop`

12. To clear the changes or files in your stash

    `git stash clear`

How Git works

1. Connecting your Remote Repository to Local Repository

   `git remote add origin insert_https_project_link_here`

2. Pushing local changes to remote repository

   `git push origin master` (we're pushing to the url origin, and the branch master)

3. To view all your remote urls

   `git remote -v`

4. Never commit on the main branch since it's the one used by the people, to prevent any mishaps

5. Shifting the head to a branch (head is the pointer which points to where all you do your changes)

   `git checkout branch_name`

6. Merging your branch to main of project

   `git merge branch_name`

1. Shift the head to your main branch

   `git checkout main`

2. Fetching all the commits/changes from the main project (upstream)

   `git fetch --all --prune` (here prune gets deleted commits too)

3. Reset the main branch of origin (forked) to main branch of upstream (main project)

   `git reset --hard upstream/main`

4. Check and verify your changes

   `git log` *click q for exit from log*

5. Then push all these local changes to your online forked repo

   `git push origin main`

Method 2

1. To fetch all **at once**

   `git pull upstream main`

2. Then push to the origin url or your forked project

   `git push origin main`

Method 3

1. Update using the Fetch Upsteam button on forked repo

10. Squashing all your multiple commits into one commit

    `git rebase -i insert_hash_code_of_commit_above_which_all_your_required_co`

    If there's 4 commits. Keep 1 as the pick and then `s` or squash the other 3 into that one

11. Merge conflicts and how to resolve them

    1. They happen when multiple users edit the same code line and then push it. Git won't know which one to merge and then there'd be a conflict
    2. This has to be resolved manually by repo maintainer

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/94dcc4e7-259a-4c09-bf05-d1ce5eb2d9e3/atlassian-git-cheatsheet.pdf

GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`
create a new branch at the current commit
`git checkout`
switch to another branch and check it out into your working directory
`git merge [branch]`
merge the specified branch's history into the current one
`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`	show the commit history for the currently active branch
`git log branchB..branchA`	show the commits on branchA that are not on branchB
`git log --follow [file]`	show the commits that changed file, even across renames
`git diff branchB...branchA`	show the diff of what is in branchA that is not in branchB
`git show [SHA]`	show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`	add a git URL as an alias
`git fetch [alias]`	fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`	Transmit local branch commits to the remote repository branch
`git pull`	fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`	delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`	change an existing file path and stage the move
`git log --stat -M`	show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`	apply any commits of current branch ahead of specified one
`git reset --hard [commit]`	clear staging area, rewrite working tree from specified commit

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`	system wide ignore pattern for all local repositories

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`	Save modified and staged changes
`git stash list`	list stack-order of stashed file changes
`git stash pop`	write working from top of stash stack
`git stash drop`	discard the changes from top of stash stack

GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`
create a new branch at the current commit
`git checkout`
switch to another branch and check it out into your working directory
`git merge [branch]`
merge the specified branch's history into the current one
`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`	show the commit history for the currently active branch
`git log branchB..branchA`	show the commits on branchA that are not on branchB
`git log --follow [file]`	show the commits that changed file, even across renames
`git diff branchB...branchA`	show the diff of what is in branchA that is not in branchB
`git show [SHA]`	show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`	add a git URL as an alias
`git fetch [alias]`	fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`	Transmit local branch commits to the remote repository branch
`git pull`	fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`	delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`	change an existing file path and stage the move
`git log --stat -M`	show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`	apply any commits of current branch ahead of specified one
`git reset --hard [commit]`	clear staging area, rewrite working tree from specified commit

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`	system wide ignore pattern for all local repositories

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`	Save modified and staged changes
`git stash list`	list stack-order of stashed file changes
`git stash pop`	write working from top of stash stack
`git stash drop`	discard the changes from top of stash stack

Git - 0 to Pro Reference

By: supersimple.dev
Tutorial link: https://www.youtube.com/watch?v=hrTQipWp6co

Command Line (Terminal / PowerShell)

`ls`	List the files and folders in the current folder
`cd ~/Desktop/folder`	Change the folder that the command line is running in

Note: git commands must be run inside the folder that contains all the code.

Creating Commits

In Git, version = commit
Version history = commit history

`git init`	Git will start tracking all changes in the current folder
`git status`	Show all changes since the previous commit
`git add <file|folder>`	Pick changes to go into next commit
`git add file`	Pick individual file
`git add folder/`	Pick all files inside a folder (and subfolders)
`git add .`	Pick all files (in folder command line is running in)
`git commit -m "message"`	Creates a commit with a message attached
`git commit -m "message" --amend`	Update previous commit instead of creating new one
`git log`	View the commit history
`git log --all`	Show all commits (not just current branch)
`git log --all --graph`	Show branching visually in the command line

Configure Name & Email for Commits

```
git config --global user.name "Your Name"
git config --global user.email "email@example.com"
```

Working Area = contains changes start in the working area

Staging Area = contains changes that will go into the next commit

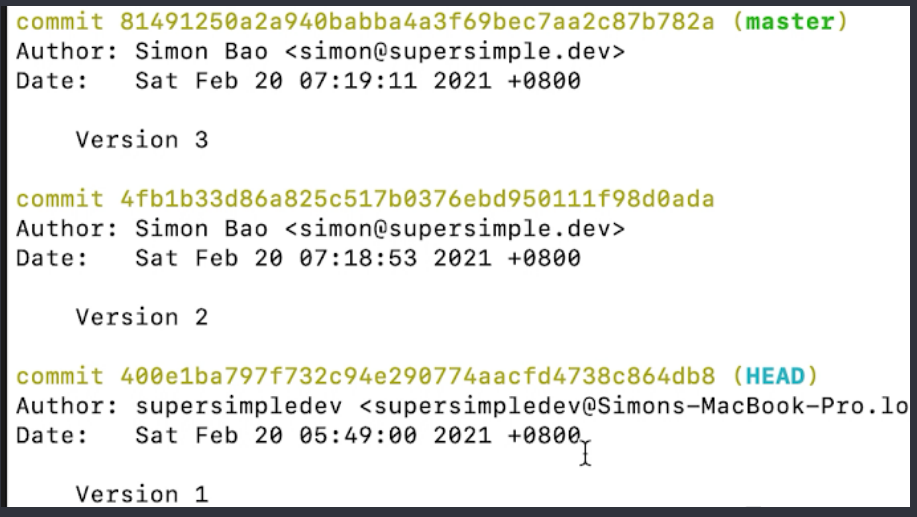
```
git add .                          working => staging
git commit -m "message"            staging => commit history

git reset <file|folder>            staging => working
  git reset file
  git reset folder/
  git reset .
git checkout -- <file|folder>      working => remove the  changes
  git checkout -- file
  git checkout -- folder/
  git checkout -- .
```

Viewing Previous Commits

```
git checkout <commit_hash|branch_name>
```

View a previous commit



master = branch name
1. You can `git checkout branch`
2. Always points to latest commit on the branch.

HEAD = indicates which commit you are currently viewing

Restoring to a Previous Commit

```
git checkout <hash|branch> <file|folder>
```

Restore the contents of files back to a previous commit

```
  git checkout <hash|branch> file          Restore a file
  git checkout <hash|branch> folder/       Restore all files in folder (& subfolders)
  git checkout <hash|branch> .             Restore all files in project
```

Other Features of Git

```
git config --global alias.shortcut <command>
  git config --global alias.s "status"
```

Creates an alias (a shortcut)

```
  git s = git status
```

`.gitignore`	Tell git which files/folders it SHOULD NOT track
`rm -rf .git`	Remove git from project

GitHub

Repository = a folder containing code where any changes to the code are tracked by git.
(To create a repository, we create a new folder on our computer, and then run `git init`)

GitHub = a service that lets us save our git repositories online. It also helps us:

- backup our code in case we delete it on our computer
- see the history of our code changes more easily
- alternatives include Bitbucket and GitLab

Local repository = a git repository saved on our computer
Remote repository = a git repository saved online (for example on GitHub)

Uploading Code to GitHub

`git remote add <remote_name> <url>`	Link a local repository to a remote repository and give a name for this link

```
git remote add origin https://github.com/SuperSimpleDev/repository1
^
```

The above command links a local repository to a GitHub repository (located at the url `https://github.com/SuperSimpleDev/repository1`) and gives it a name "origin"

`git remote`	List all remote repositories that are linked
`git remote -v`	List all remote repositories (but with more detail)
`git remote remove <remote_name>`	Removes a link to a remote repository
`git remote remove origin`	Removes the link to the remote repository named "origin"
`git config --global credential.username <username>`	Configure your GitHub username so you can get access to your Github repository
`git push <remote_name> <branch>`	Upload a branch of your git version history to your remote repository
`git branch`	Shows a list of available branches
`git log --all --graph`	Shows the branches visually in the history

```
  git push origin main
```
Upload the branch "main" to the remote repository named "origin"

```
git push <remote_name> <branch> --set-upstream
```
Sets up a shortcut for this branch and remote repository

```
  git push origin main --set-upstream
```
Next time you are on the `main` branch and you run `git push`, it will automatically push the `main` branch to `origin`

```
git push <remote_name> <branch> -f
```
Force-push the branch to the remote repository (it will overwrite what's on the remote repository)

Downloading Code from GitHub

```
git clone <url>
    git clone https://github.com/SuperSimpleDev/repository1
```
Download a remote repository from a `url`

```
git clone <url> <folder_name>
```
Download the repository and give it a different folder name

```
git fetch
```
Updates all remote tracking branches. Remote tracking branches (like `origin/main`) show what the branch looks like in the remote repository

```
git pull <remote_name> <branch>
```
Update the local branch with any updates from the remote repository (on GitHub)

```
  git pull origin main
```
Downloads any new commits from the `main` branch on `origin`, and updates the local `main` branch with those new commits

```
  git pull origin main --set-upstream
  ^
```
Sets up a shortcut so that the next time you are on the `main` branch and run `git pull`, it will automatically `git pull origin main`

Branching

Branching = create a copy of the version history that we can work on without affecting the original version history. This lets us work on multiple things (features + fixes) at the same time.

```
git branch <branch_name>
```
Creates a new branch

```
  git branch feature1
```
Create a new branch named `feature1`

```
git checkout <branch_name>
```
Switch to a different branch and start working on that branch

```
  git checkout feature1
```
Switch to the `feature1` branch. New commits will now be added to the `feature1` branch

```
* commit 9bb22ff9063a3e1134e5cea3fb289df492868cef (HEAD -> feature1, master)
| Author: Simon Bao <simon@supersimple.dev>
| Date:   Sat Jun 5 09:27:25 2021 +0800
|
|     version3
|
* commit 8464f5b7dc7d0271f8a00f9dc0b707b4ecc64301
| Author: Simon Bao <simon@supersimple.dev>
| Date:   Sat Jun 5 09:27:16 2021 +0800
|
|     version2
|
* commit 285addbf98ee4d450c226a410acf38ab16ba7696
  Author: Simon Bao <simon@supersimple.dev>
  Date:   Sat Jun 5 09:27:01 2021 +0800

      version1
```

HEAD = points to which branch we are currently working on
HEAD -> `feature1` = we are currently working on the `feature1` branch. Any new commits will be added to the `feature1` branch

`git branch -D <branch_name>` `git branch -D feature1`	Deletes a branch Deletes the `feature1` branch

Merging

`git merge <branch_name> -m "message"`	Merge the current branch (indicated by **HEAD ->**) with another branch (`<branch_name>`). Saves the result of the merge as a commit on the current branch
`git checkout main` `git merge feature1 -m "message"`	1. First switch to the `main` branch 2. Then merge the `main` branch with the `feature1` branch. The result of the merge will be added to `main` as a commit (a "merge commit")

Merge Conflicts

`<<<<<<< HEAD`
`code1`
`=======`
`code2`
`>>>>>>> branch`
	If there is a merge conflict (git doesn't know what the final code should be), it will add this in your code.
	
	(This is just for your convenience, the `<<<<<<<` and `>>>>>>>` don't have special meaning)

```
<<<<<<< HEAD
...             <-- Code in the current branch (indicated by HEAD ->)
=======
...             <-- Code in the branch that is being merged into HEAD
>>>>>>> branch
```

To resolve a merge conflict:

1. Delete all the extra code and just leave the final code that you want.

~~<<<<<<< HEAD~~
~~code1~~
~~=======~~ => code2
code2
~~>>>>>>> branch~~

2. If there are conflicts in multiple places in your code, repeat step 1 for all those places.

3. Create a commit.

```
git add .
git commit -m "message"
```

Feature Branch Workflow

A popular process that companies use when adding new features to their software.

1. Create a branch for the new feature (called a "feature branch").

```
git branch new-feature
git checkout new-feature
```

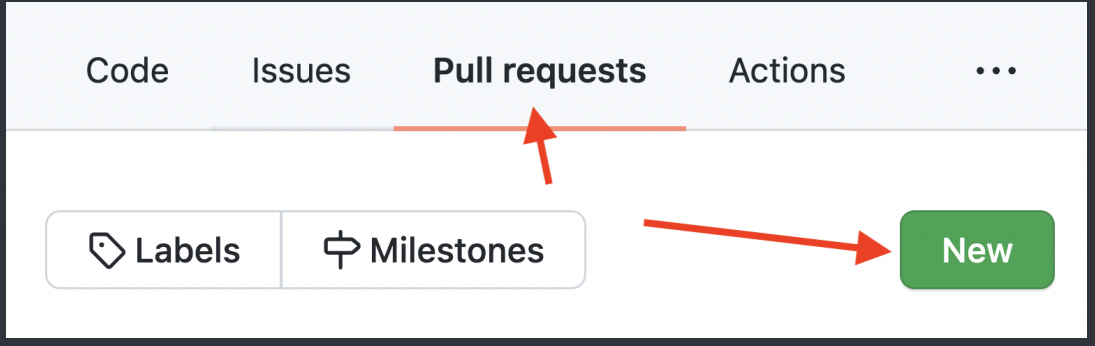
Make some changes to the code...

```
git add .
git commit -m "new feature message"
```

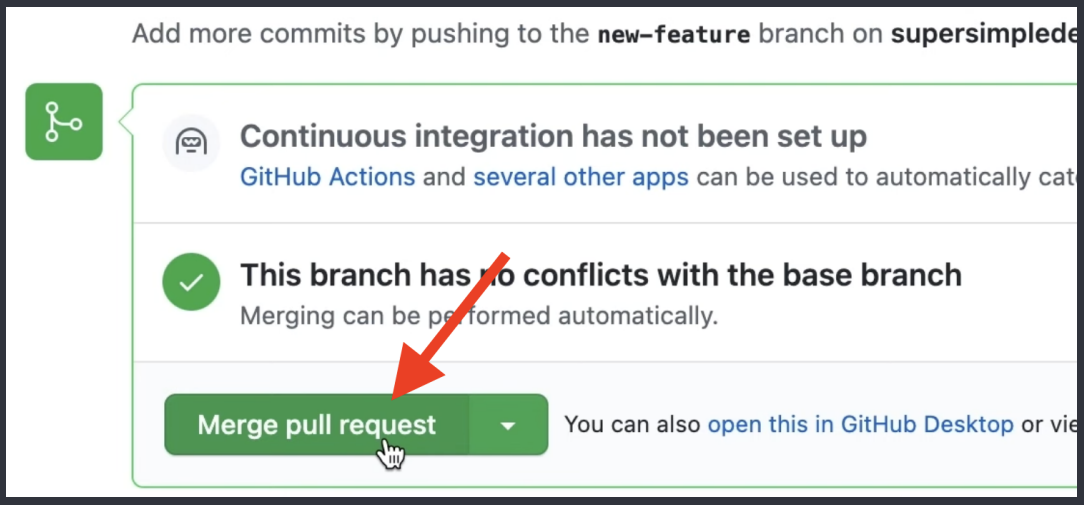
2. Upload the feature branch to GitHub.

```
git push origin new-feature
```

3. Create a pull request on GitHub (a pull request lets teammates do code reviews and add comments).



4. Merge the feature branch into the main branch (by opening the pull request in the browser and clicking "Merge pull request")



5. After merging, update the local repository (so that it stays in sync with the remote repository on GitHub).

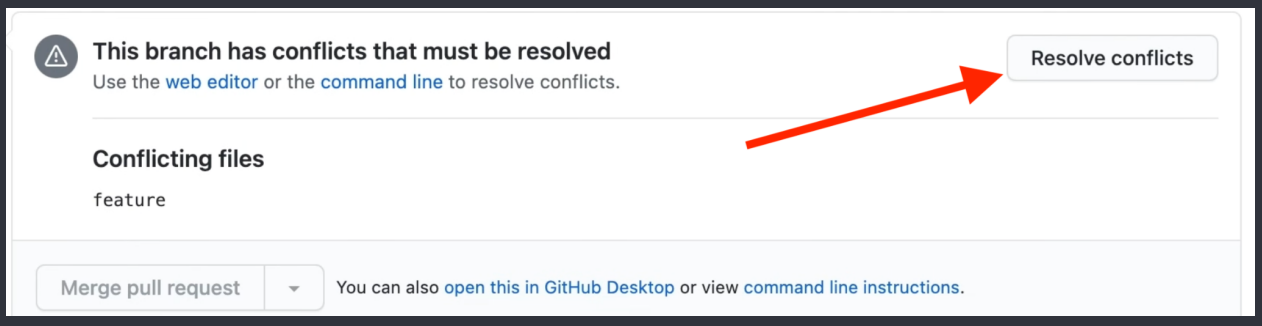
```
git checkout main
git pull origin main
```

Merge Conflicts in the Feature Branch Workflow

A merge conflict can happen if 2 or more pull requests change the same file and the same line.

We can either:

1. Resolve the merge conflict on GitHub.



2. Resolve the merge conflict on our computer.

1) Get the latest updates from `main`

```
git checkout main
git pull origin main
```

2) Get the latest updates from the feature branch.

```
git checkout feature4
git pull origin feature4
```

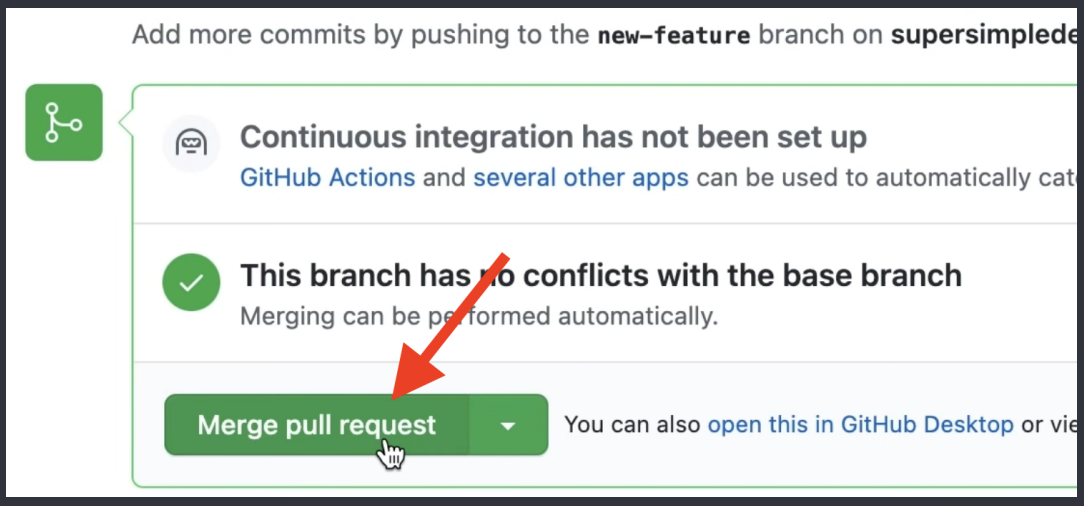
3) Merge `main` into the feature branch (`feature4`). Notice the direction of the merge: we want the merge commit to stay on the feature branch so our teammates can review it.

```
git checkout feature4
git merge master
```

4) Push the resolved feature branch to GitHub.

```
git push origin feature4
```

Now the pull request should be ready to merge again.



GitHub Pages Reference

By: supersimple.dev
Video Tutorial: https://youtu.be/p1QU3kLFPdg

GitHub Pages

Lets us put websites created with HTML, CSS, and JavaScript code on the Internet

Overall Process

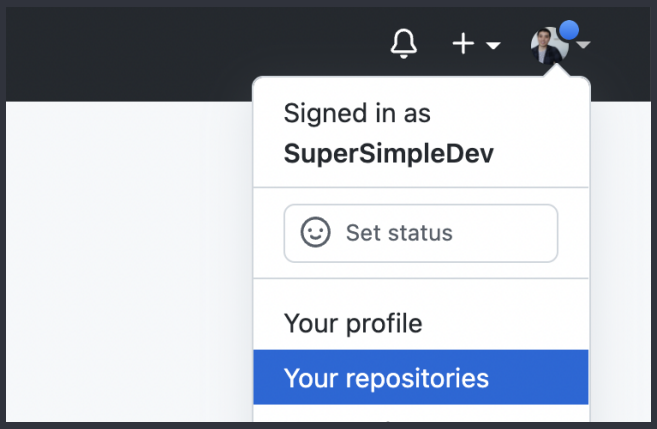
1. Write your HTML, CSS, and JavaScript code on your computer (as normal)
2. Upload your code to a GitHub repository
3. Turn on GitHub Pages from the repository's settings

That's it!

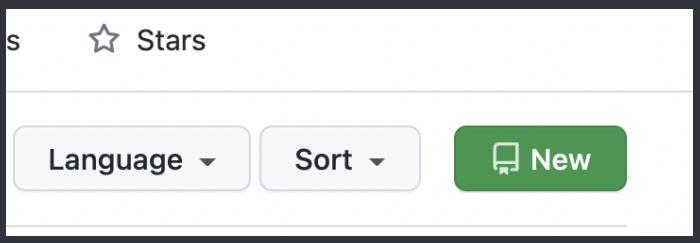
Step By Step

1. Write the code for your website as normal (make sure your website works when you open the HTML file in your browser)
2. In your browser, go to "github.com" and sign in

3. Click your icon in the top-right > Click "Your repositories"
   (repository = folder that contains all your code)



4. If you already have a repository, open it. Otherwise, on the right side click "New"



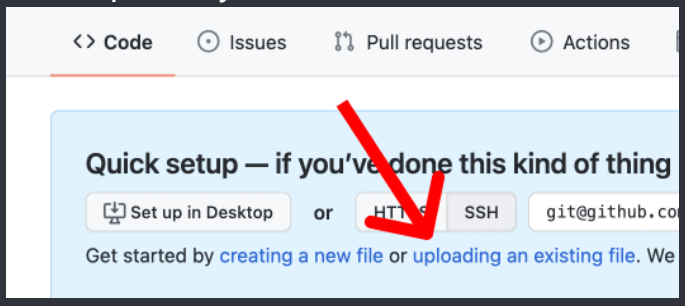
4a. When creating a new repository:

- Give it a name (any name is fine)
- Make sure it is "Public" (private repositories need to pay a fee to use GitHub Pages)
- Everything else can be left as default

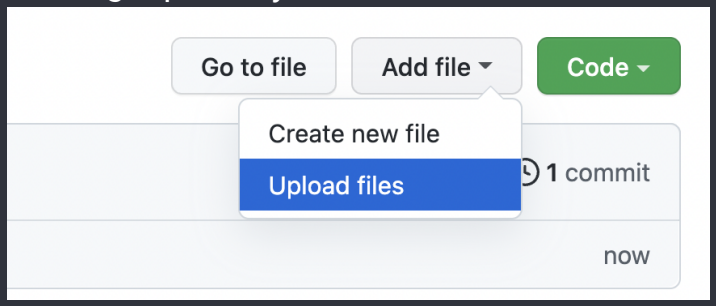
Upload Your Code

5. Click "Upload files"

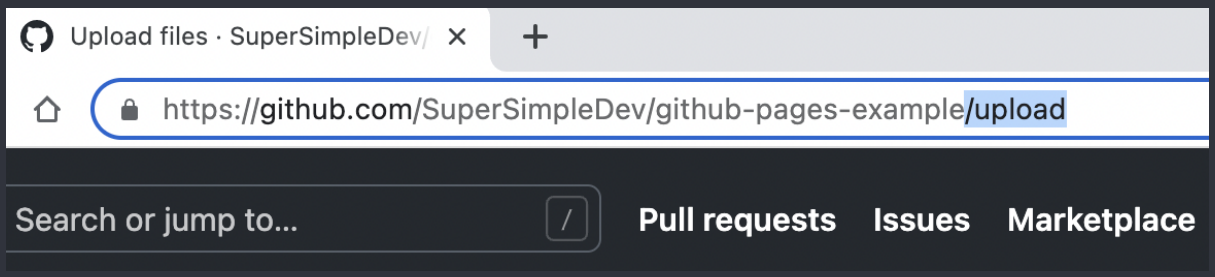
New repository:



Existing repository:



You can also add "/upload" to the end of the URL:



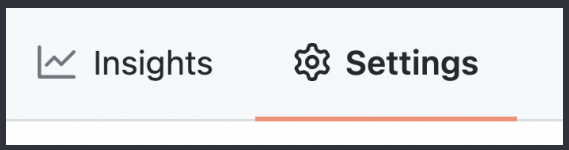
6. Drag and drop the code you want to upload and click "Commit Changes" at the bottom.

❌ Don't drag the folder containing all your code into GitHub

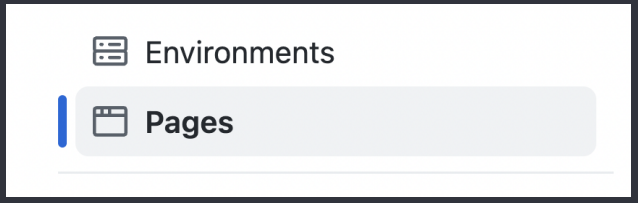
✅ First open the folder, then select all your code and drag and drop into GitHub

Turn on GitHub Pages

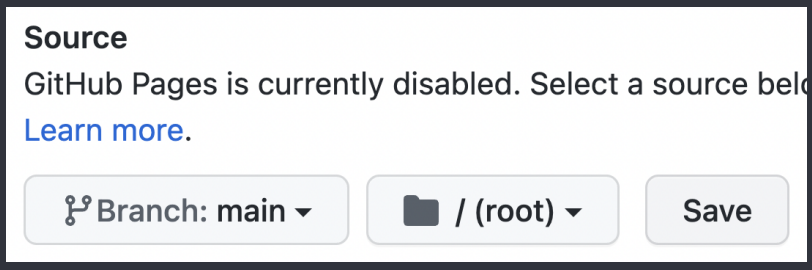
7. In your repository, click "Settings" on the top-right.



8. Click the "Pages" tab on the left.



9. Select Branch: main (or master) and click Save.



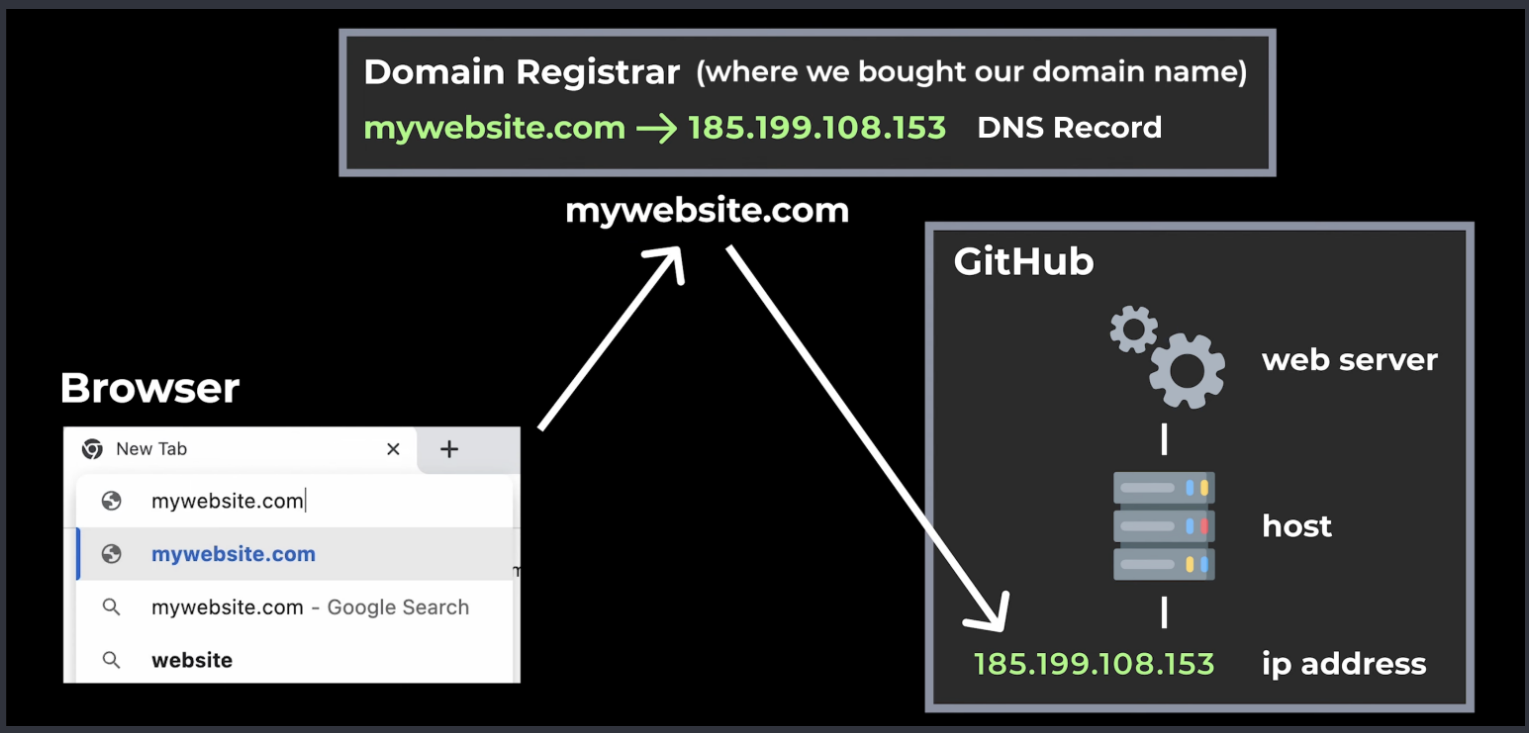
10. To visit your GitHub Pages website, enter this URL at the top of your browser:
`[your_github_username].github.io/[repository_name]/[file_name].html`

If you don't provide an HTML file, GitHub Pages will look for a file called "index.html":
`[your_github_username].github.io/[repository_name]`

Set up a Domain Name

1. Purchase a domain from a domain registrar (Namecheap, domain.com, Google Domains, etc.)

2. We'll link our domain name to GitHub Pages' IP address. Here's the diagram for reference



IP Address

- Is like a phone number (but for the Internet)
- Any device that connects to the Internet is given an IP address. At home, you connect to the Internet through a router. In this case, only the router will get an IP address that is public
- A device can communicate with another device across the Internet using an IP address

IPv4 example: 185.199.108.153
IPv6 example: 2606:50c0:8000::153

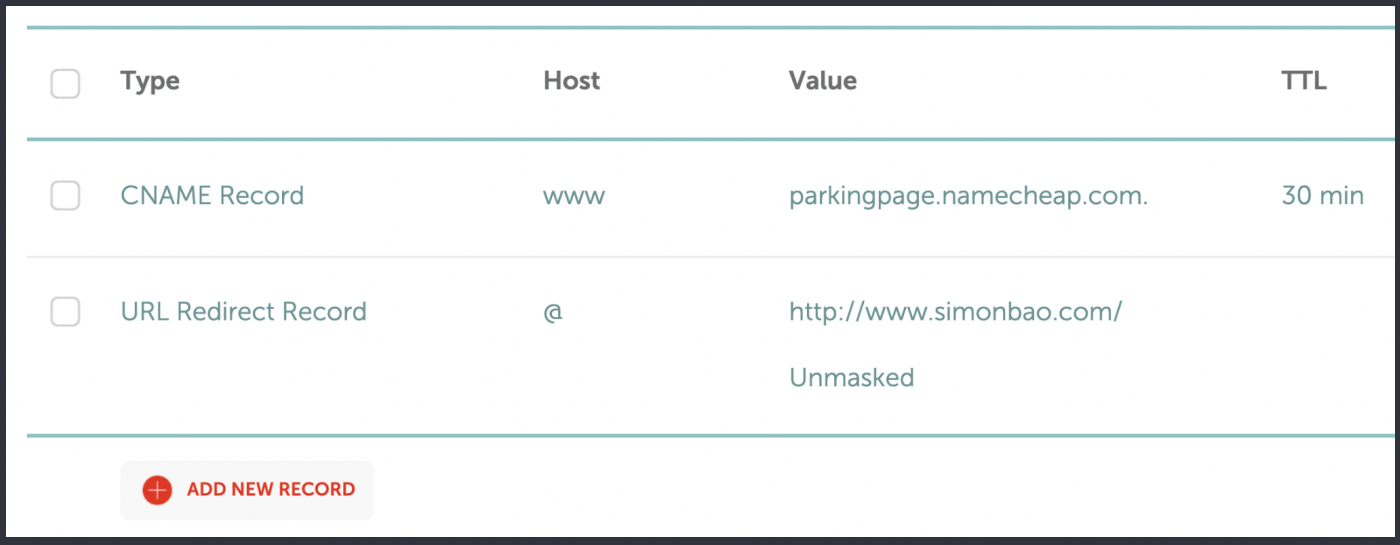
3. Find GitHub Pages IP address. Go to https://mxtoolbox.com/DNSLookup.aspx and search for "[your_github_username].github.io"

Create DNS Records

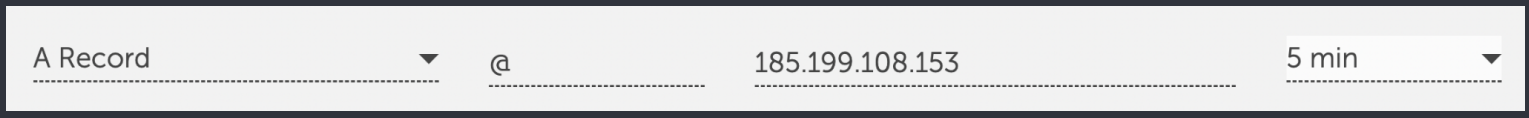
DNS records let us link IP addresses (and other information) to a domain name.

4. Log into your domain registrar > open the "Settings" for your domain.
If you're not using Namecheap, see https://supersimple.dev/infrastructure/dns-records
In Namecheap:

- Go to your Dashboard > click "Domain List" on the left
- Find your domain > click "Manage" on the right > click "Advanced DNS"
- Look for this table (that lets us create DNS records):



5. Create an A Record = links a domain name to an IP address



- 1st value, select "A Record"
- 2nd value is the subdomain, which is the part of a URL in front of a domain name (like subdomain.mywebsite.com). Namecheap, use "@". Other domain registrars, leave it blank
- 3rd value, enter an IP address found in step 3
- 4th value is TTL = time to live (how long DNS records will live before being refreshed). Namecheap, select "5 min". Other domain registrars, enter "300" (300 seconds)

Create a DNS A Record for each IP address from step 3. Here's the end result:

Type	Host	Value	TTL
A Record	@	185.199.108.153	5 min
A Record	@	185.199.109.153	5 min
A Record	@	185.199.110.153	5 min
A Record	@	185.199.111.153	5 min

6. Create a CNAME Record = links a domain name to another domain name.
Set up the subdomain "www" (www.mywebsite.com). This is recommended by GitHub Pages.

CNAME Record	www	supersimpledev.github.io.	5 min

- 1st value, select "CNAME Record"
- 2nd value, enter "www"
- 3rd value, enter your GitHub Pages domain name = [your_github_username].github.io
- 4th value, enter "5 min" or "300"

Here's the end result:

Type	Host	Value	TTL
CNAME Record	www	supersimpledev.github.io.	5 min

Summary
A Record = mywebsite.com -> ip address
CNAME Record = subdomain.mywebsite.com -> domain name

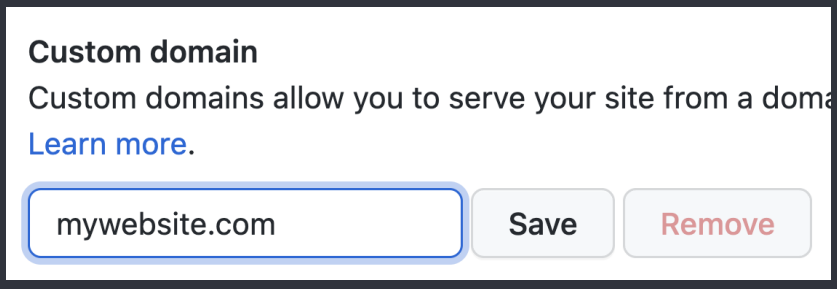
Note
A CNAME Record can only be used when there is a subdomain. Some registrars offer an ALIAS Record which can link the base domain name ("mywebsite.com") to another domain.

Set up Domain Name in GitHub Pages

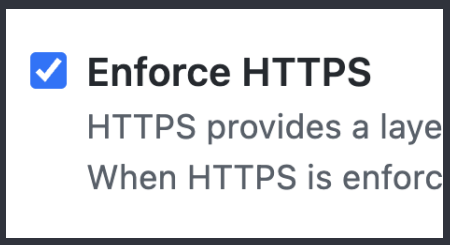
1. Go to the GitHub repository for your website

2. Click "Settings" on the right > click "Pages" on the left

3. In the "Custom domain" section, enter your domain name. Click Save

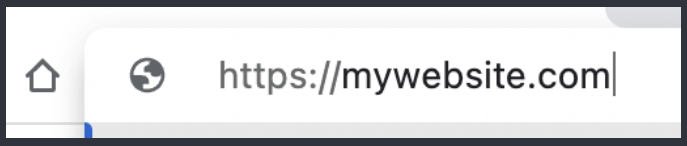


4. Wait for GitHub's process to finish. Then check the box to activate HTTPS



HTTPS encrypts all data flowing between your browser and your website.

5. Finally visit your website by entering in the top of your browser:



7 Git

Techniques and Shortcuts

a quick guide by @meakcodes

Technique 1:

Git Commit

I assume that you already know how to use:

```
git-demo $ git add .
git-demo $ git commit -m "hi mom"
[master d6b7949] hi mom
 1 file changed, 1 insertion(+)
git-demo $
```

But theres actually a better way how to do it:

```
git-demo $ git commit -am "that was easy!"
[master 4bc37a8] that was easy!
 1 file changed, 1 insertion(+)
git-demo $
```

Just get rid of the **“git add . “** and **“git push”** and just type * **git commit -am “ “** *

note: new files will be not tracked.

Technique 2:

Git stash

Let's say you have implemented some code, but you don't want to share it yet. You just want to save it without publishing it for now.

Thats where git stash comes in:

```
git-demo $ git stash save coolstuff
```

the code know will be hidden and not visible, its like it never existed.
example:** it will be saved under the name **"coolstuff"

Then when you're ready to use it again:
use git stash list to view a list of all stashes

```
git-demo $ git stash list
stash@{0}: On master: coolstuff
git-demo $
```

Then choose your respective stash:
choose the stash with the current index

```
git-demo $ git stash apply 0
```

Technique 3:

Git log

Imagine you want to view the latest commits from the past. You might currently be using **'git log'** for this.

```
fireship $ git log
```

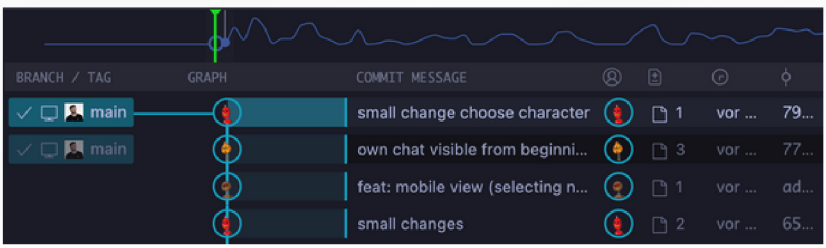
But that's actually not a great idea because it's hard to follow due to all the information being so unstructured: *it will look like this!*

```
commit a8db8aecd3fc5912fb611923d6f0b397fa3a6a1f (orig
in/master, origin/HEAD, master)
Author: Jeff Delaney <hello@fireship.io>
Date:   Fri Sep 3 15:26:31 2021 -0700

    add dispatch to gh actions
```

Just use the VsCode Extension **GitLens**:

there so many features, definitely give it a try!

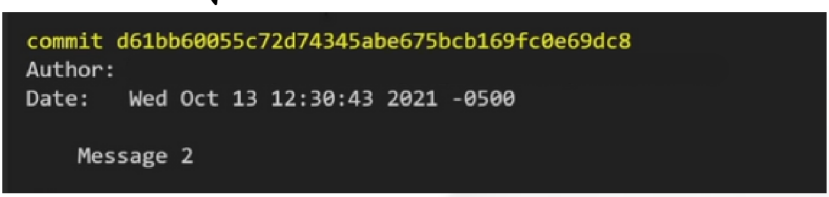


Technique 4:

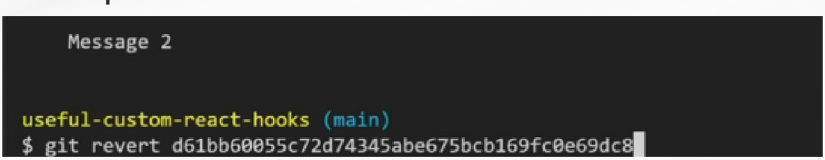
Git revert

Consider a scenario where you need to switch to a different commit because you made a mistake and want to revert to a specific section of your commit history:

Use **git log** to show the last commits and copy that commit id:



Now, enter the command **'git revert'** followed by the copied ID from the last commit:



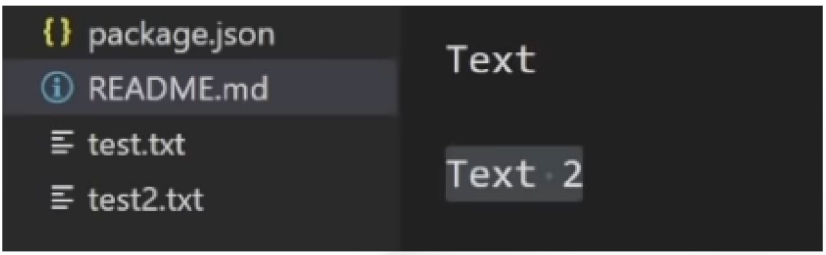
Using **'git revert'** does not delete your previous commit. Instead, it generates a new commit that effectively undoes the changes made in the specified commit. It allows you to selectively undo a particular commit without affecting the rest of your commit history.

Technique 5:

Git log -S “ ”

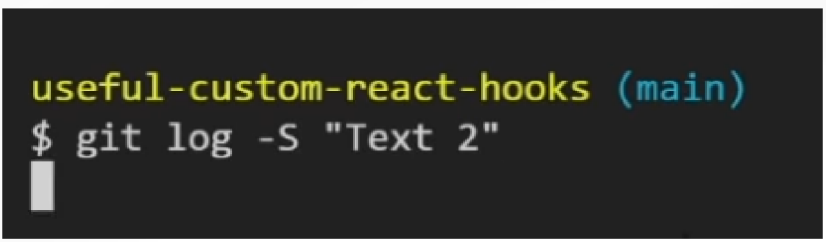
Picture a situation where you recall having specific words in your code, but now you can't locate them. You wish to investigate where those words are currently within your codebase.

For example our **README.md file** looks like this:



Let's find out where **'Text 2'** was changed and see who made those modifications.

Just use **git log -S “ ”** and in our example with *“Text 2”*
you will then all commits related to “Text 2”



Technique 6:

Git branch --prune

Imagine you have an old branch that you no longer use, but it still exists locally. Learn how to completely delete it:

To view the current and last branches type: **git branch -vv**

```
useful-custom-react-hooks (main)
$ git branch -vv
  21-25 170b8b7 [origin/21-25] Revert "Revert "Message 2""
* main  170b8b7 [origin/main: ahead 8] Revert "Revert "Message 2""
```

As you can see, we have one **old branch** and one **main branch**. Now, we want to delete that old one!

Use **git remote update --prune** , it will delete the reference to your remote repository if the branch no longer exists.

```
useful-custom-react-hooks (main)
$ git remote update --prune
Fetching origin
From github.com:WebDevSimplified/useful-custom-react-hooks
 - [deleted]         (none)     -> origin/21-25
```

If you view it again, it will say **gone**:

```
useful-custom-react-hooks (main)
$ git branch -vv
  21-25 170b8b7 [origin/21-25: gone] Revert "Revert "Message 2""
* main  170b8b7 [origin/main: ahead 8] Revert "Revert "Message 2""
```

Technique 7:

Git rebase

Imagine you working in a Team and you need to implement a feature and want to merge it later:

Create another branch, let's name it '**feature1**' for example, and then check out that branch:

```
git branch feature1
git checkout feature1
```

→

```
Switched to branch 'feature1'
* feature1
  master
```

Now let's commit something to that branch:
note: that is not the master branch!

```
touch f1
git add .
git commit -m "f1 added"
```

```
touch f2
git add .
git commit -m "f2 added"
```

```
touch f3
git add .
git commit -m "f3 added"
```

Lets go back to the Master Branch:

```
git checkout master
```

→

```
Switched to branch 'master'
  feature1
* master
```

Now, simply use rebase to integrate your features into the master branch.
Simply use **'git rebase'** with your current branch, let's say '**feature 1,**' and indicate the branch where you want to merge it.
In our example it's the master branch.

```
git rebase feature1 master
```

your feature branch to master

Before you go!

Please leave a positive review on Gumroad!

good luck & keep learning!