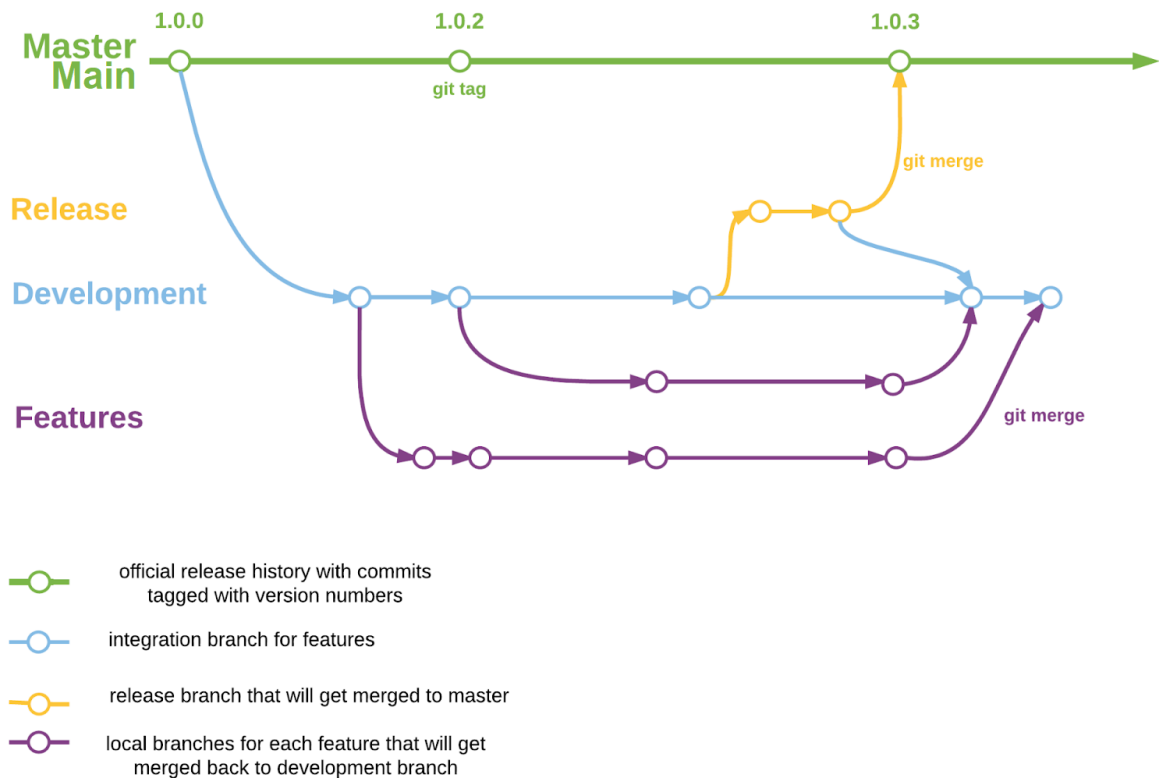


Git Branching Cheatsheet

- Jump to [Workflow Examples](#) for a quick example reference if you've already read the basics.
- [Simple Agile Trello Board Template](#)

Diagram

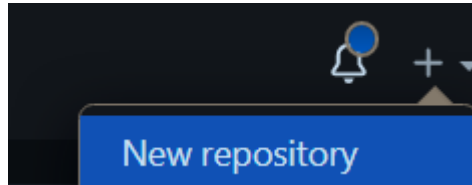
- This is a simplified diagram. In the real world there are often additional branches so that the development branch is not merged directly into master. For example, a User Acceptance Testing (UAT) branch may be between master and dev.
- Master, now more commonly named 'main', contains the code that the live / deployed website would run off of--so it should only have code merged into it from the dev branch that has already been tested and verified as bug-free so the main branch is kept 'pristine'. There are edge cases like creating a hotfix branch to quickly fix a critical bug that made it to production.



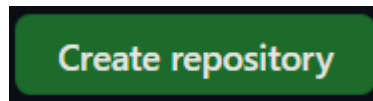
Repository Creation

- After creating your repository you can give collaborators access via settings -> collaborators -> add people.

GitHub (remote - not your PC)



1. At the top right, click
2. Name the repo, use hyphens or underscores instead of spaces. Leave the rest as default.



3. At the bottom, click
4. Leave the following page open, we will need it later. Proceed to the Your PC steps.

Your PC (local) & Link It To GitHub (remote)

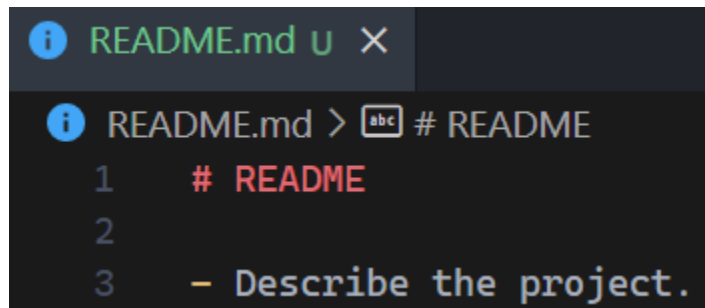
1. Create a new folder for your project and open your terminal to it.
2. Verify the new project folder is not already a git repository with the terminal open to the new folder.
 - a. **git status**
 - b. It should say - **fatal: not a git repository**. Proceed to the next step if it does, otherwise, read below.
 - i. If it doesn't say **fatal: not a git repository**, the project is already a git repository because you have already made it one, or you placed it inside a parent folder that is already a git repository which is a problem.
 - ii. Move the new project folder to a different folder that you know is not already a git repository and verify with **git status** again. If **git status** still does not say **fatal: not a git repository**, then delete the

hidden **.git** folder in your new project folder (look up how to show hidden files & folders). If there is no **.git** folder in your project folder, then one of the folders your project is in must be a git repository – move the project folder again or find and delete the **.git** folder in one of the parent folders if it should not have been a git repository.

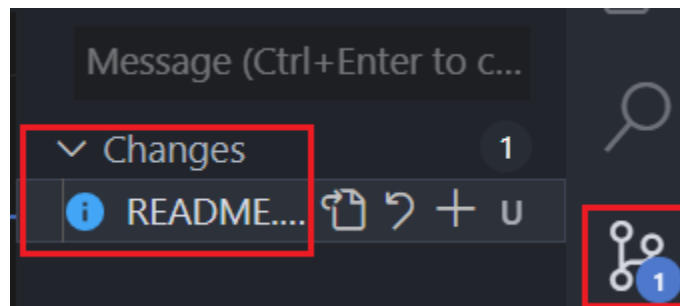
3. git init

- a. Do this only after you have verified your new project folder is not already a git repository itself or inside of one.
- b. It should say: **Initialized empty Git repository**

4. Create a README.md file.



- a.
- b. In the source control panel you should now see git is tracking the README and any other files to watch for changes:



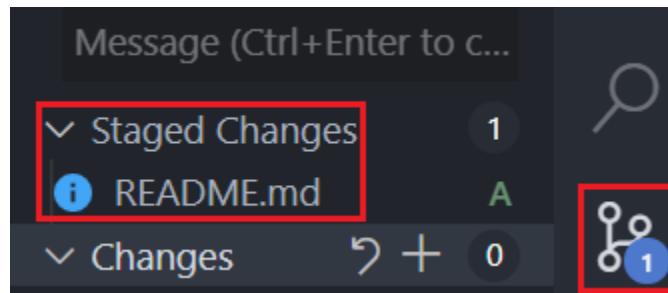
i.

5. Create a **.gitignore**. A gitignore should be tracked and should be added to GitHub. This file tells git which files it should ignore so they won't end up on GitHub. The files that should be ignored are any files that contain sensitive information such as passwords and API keys. Also files with configuration settings that are specific to the developer or environment (dev / prod). Lastly, files that can be automatically generated on the local computer, such as build files, installed package files, debug files, etc.

- a. Create a file named **.gitignore** at the root of your project folder.
- b. Google "*MERN* gitignore example" (replace *MERN* with the framework you are using) and copy paste the example code into the **.gitignore** file. Alternatively, some languages / frameworks have terminal commands to create a gitignore automatically, for example in dotnet: **dotnet new gitignore**
- c. Also add any files to **.gitignore** that contain secret information, such as API keys and passwords. If teammates need this information, share it with them directly so they can create their own local files instead of it being visible on GitHub.
- d. In the future, when git is set up in your project, you can right click a file in staged changes and click "add to gitignore" if needed.

6. git add .

- a. This will tell git to track changes in all the files that are in the repository except the ones listed in the gitignore file.
- b. You should now see all your project's files have moved to the Staged Changes category. They are now ready to be committed. Walking onto a stage is a big commitment.



i.

7. Run commands from GitHub repository page.

a. git commit -m "first commit"

- i. All future commits should be frequent, small, and concisely describe what changes were made. This helps to easily revert if something goes wrong without reverting too far.

b. git branch -M main

- i. Renames the branch. If it is already named main this can be skipped.

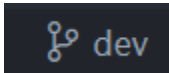
c. Paste the **git remote add origin** line. This is what connects your local (on your PC) repository to the corresponding remote repository on GitHub.

d. **git push -u origin main**

e. Refresh the GitHub repository page and you should see your README file, .gitignore file, and any other files that were not ignored.

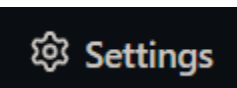
f. **git checkout -b dev**

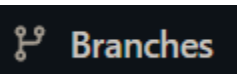
i. Create the dev branch for the first time on your local PC. It will come with all the code from the main branch since we are branching off of it. The main branch is only used for deployed code.

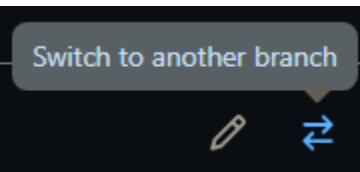
ii.  you should see in the bottom left of VSCode it shows **dev** as the active (checked-out) branch.

g. **git push origin dev**

i. To create this branch also on the remote repository as well, so your collaborators can see and use it. All other branches where features are developed and bugs are fixed will be merged into **dev** when they are completed. In the future this command will just update the remote **dev** branch with your local **dev** branch changes.

h.  go to your GitHub repository settings at the top.

i.  click branches on the left.

j.  switch the default branch to **dev**. **main** should only be updated when preparing to deploy.

k. Tell your collaborators to create the **dev** branch locally and pull it.

i. **git checkout -b dev**

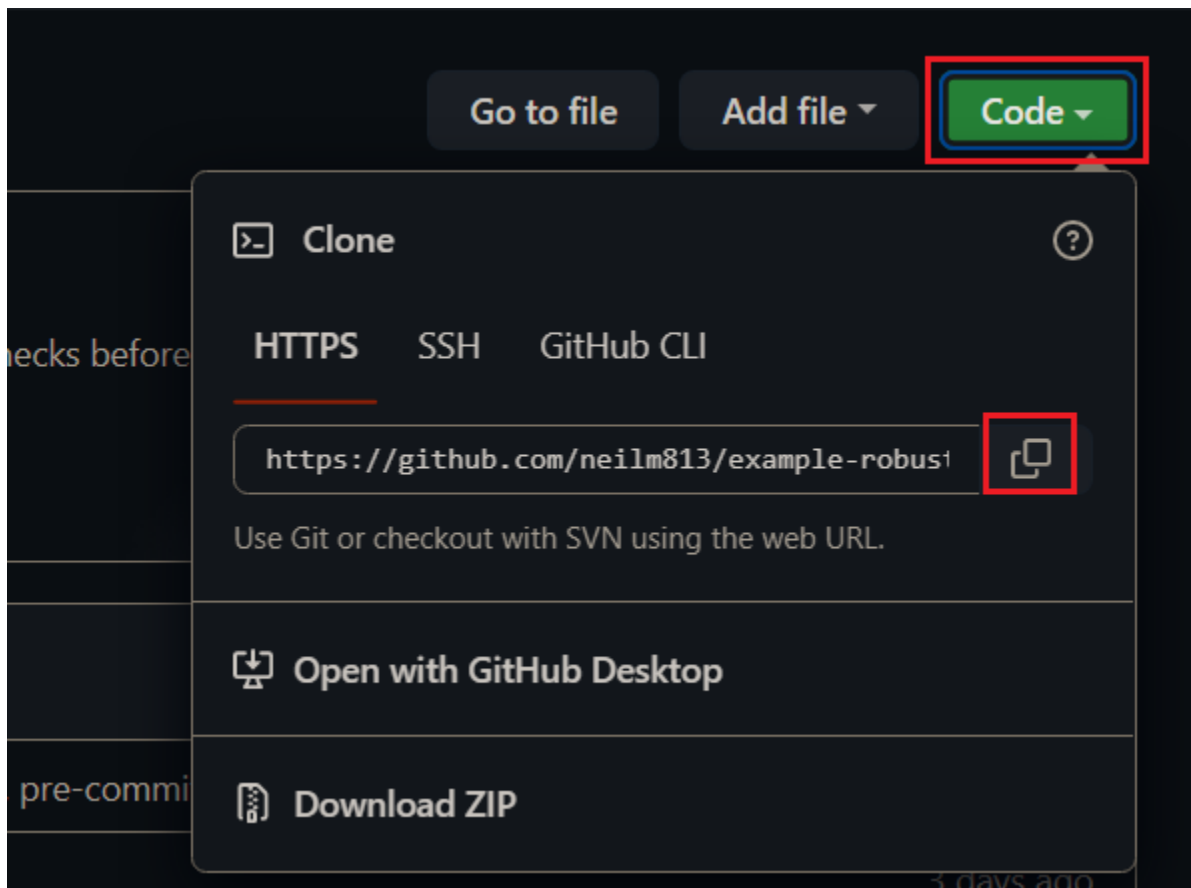
1. Create the branch locally on their PC.

ii. **git pull origin dev**

1. Pull the remote **dev** branch code onto their PC.
1. Now you and your team are ready to start creating and working on feature and bugfix branches. See [Definitions and Commands](#), [Branch Naming](#), [Workflow Examples](#).

Cloning A Repository

- Once you or a team member has created a repository to collaborate on, the other team members need to clone it so they have a local copy (a copy on their computer).
1. Go to the root level of the repository on GitHub (don't open any folder or file).
 2. Click Code and then the copy button



3. Open your terminal to the location on your computer where you want the repository to be saved to.
4. ***git clone paste-link-here***
5. Press enter

6. Open the newly created folder in VSCode (the folder name matches the GitHub repository name).
7. Ask the repository creator if there are any gitignored configuration files that are needed for the project.
 - a. Configuration files are typically ignored and not on the remote GitHub repository, so they need to be sent directly to each collaborator so each collaborator can add those files to their local cloned repository.
8. When there has been new code added to the remote repository you can get it by opening your terminal to the cloned repository and using the **pull** command. See the below Definitions and Commands if you are new to git.

Definitions and Commands

- Examples given below are generic, for more specific examples, see [Workflow Examples](#).

branch

- A branch contains some version of code.
- A single branch may exist locally, only on your computer, or only remotely, on GitHub, or both (named the same). If it exists in both places, the branches may not be synchronized if edits have been made to one and haven't been merged with the other.
- Editing code will result in the checked-out (currently active) branch's code diverging (branching) away from the branch that the code was copied from originally. When editing is complete, typically the branch will have it's code merged back into the branch that it diverged from so that multiple branches of code edits that people have completed can converge to result in a complete version of the application—the **dev** branch is the full dev version of the application and the **main** branch is the full production (deployed) version.

Commands

- **git branch**
 - To view all the locally available branches on your computer and which one is checked out (active). To view all the remote branches, see the branches section of the GitHub repository.

- **git branch -D branch-name**
 - This deletes a branch from your local computer, not from the remote (GitHub) repository. After a feature or bug fix has been completed and the branch has been successfully merged into **dev** via a pull request, the branch is typically deleted both on GitHub and on the local repository (your PC) since the code is now available in the **dev** branch.

commit

- Commit to the changes you've made. See [stage, commit & source control panel](#).

Example

- **git add server/routes/user.routes.js**
- **git add server/controllers/user.controller.js**
 - This adds the changed files as staged changes that are ready to commit. You can also view the changes made and stage / unstage files in the source control panel:



- **git commit -m "added api route to create a user"**

checkout

- Checking out a branch means switching to a local (existing on your PC) branch so that it is active. Meaning your code will be displayed as the code from that branch, and edits will only edit the code on the checked out branch.
- The checked out branch can be seen in the bottom left of VSCode, and in some terminals.
- If there is a remote branch (a branch on GitHub with the same name), the code in your local (on your PC) branch may or may not be up-to-date. You may need to pull the remote branch (of the same name) to get updates if there have been changes.

Commands

- **git checkout branch-name**

- This will switch (checkout) to the existing branch with the given name.
- **git checkout -b branch-name**
 - This will create a new branch with the given name and also check the branch out.

merge

- I use [pull](#) instead most of the time, since it ensures getting the latest changes and then merges.
- Merging is when two branches with diverging code have their code merged together into one of the branches. New code is merged into the **dev** branch to add new features and bug fixes to the current version of the app. However, two branches that aren't **dev** can also be merged together, for example when two features are being developed by two different developers but they need to share their code because the features depend on each other.

Merge conflicts

- If you use VSCode's integrated terminal and use [pull](#) to merge, merge conflicts will show up in VSCode and can be handled more easily in VSCode's GUI.
- Merging can result in merge conflicts if there have been edits to the same line of code in both branches which means for each conflict you will have to specify how the code should be merged.
 - The common options are to accept the code that is being merged into your branch to have it overwrite the code in the checked-out branch, or keep the current code in the checked-out branch instead, or if some of the code is needed from both versions, you will have to write out the merging of the code by hand in the file that has the conflict to resolve it. You can keep code from both if the code does not conflict with each other. Make sure to save the file after merging is done for VSCode to recognize the conflict is resolved.

Example

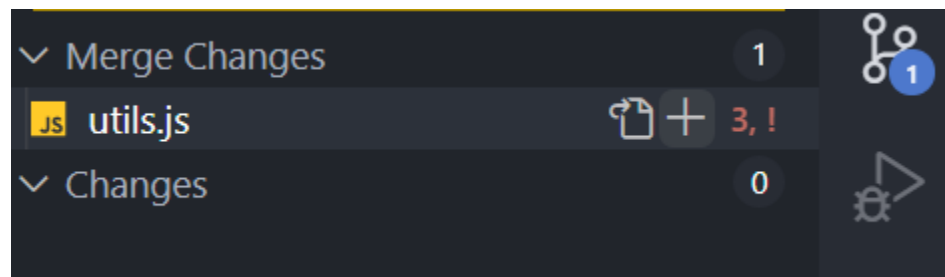
- Imagine you have a very simple file of utility functions:

```
function add(a, b) {  
  return a + b  
}  
  
function subtract(a, b) {  
  return a - b;  
}  
  
module.exports = {  
  add: add,  
  subtract: subtract,  
};
```

- You get assigned a feature to update the add function to support adding many numbers instead of just two so you create a branch to start working on this.
- **git checkout dev**
- **git pull origin dev**
 - Make sure your **dev** branch is up-to-date so the new branch you create will start with up-to-date code.
- **git checkout -b feature/add-many**
 - Create the new branch to work updating the **add** function.
- However, there was a miscommunication and somebody else is already working on it and they finish it before you while you are still working, and they merge their finished branch into the **dev** branch, so now, you unexpectedly don't have the up-to-date code as you keep working.
- You finish your work and prepare to [stage and commit](#) so you can [Open a Pull Request](#) to merge your feature branch into **dev**.
- **git pull origin dev**
 - When you run this command after committing your code changes, you get a merge conflict because someone else edited these same lines of code and merged them into **dev** while you were working.

```
Js  utils.js > add
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes | Start Live Share Session
1  <<<<<<< HEAD (Current Change)
2  function add(... nums) {
3      let sum = 0;
4
5      for (let i = 0; i < nums.length; i++) {
6          sum += nums[i];
7
8  =====
9  function add(numbersArr) {
10     let sum = 0;
11
12     for (let i = 0; i < numbersArr.length; i++) {
13         sum += numbersArr[i];
14     }
15     return sum;
16 }
17
```

-
- Source control panel:



- You now need to decide how the code should be merged, you can accept the changes coming from **dev** (Accept Incoming Change) to overwrite your code, or keep your code instead (Accept Current Change). These are the two most common choices.
- Let's say after talking with the team, your feature code is the preferred solution, so you click Accept Current Change.
- Save the file.
- Click the + icon (see above image) for this file in the source control panel to stage and resolve the merge conflict. The file should disappear from the panel.
- **git commit -m "merged dev into feature/add-many"**
 - Sometimes when you try to commit with the prefilled message in the source control panel GUI it will say nothing to commit. That is why this step shows the command instead.

- **git push origin feature/add-many**
 - Pushes your merged feature branch to GitHub
- Now you can [Open a Pull Request](#) and your code should be able to be merged into **dev** without any more conflicts, because you already handled them.
- If the other person's code was going to be kept instead, you could have just accepted incoming code and then deleted your branch since their code was already in **dev**. Unless your branch had other changes that **dev** needed then you would need to push your branch and open a pull request for it.
- Alternatively, you could have handled the merge conflict in the opposite direction.
 - Stage and commit changes on your branch.
 - **git push origin feature/add-many**
 - **git checkout dev**
 - **git pull origin dev**
 - **git pull origin feature/add-many** to pull your change into **dev**, handle the conflict, commit the merge changes in **dev** and then, **git push origin dev** to push the merged code.

pull

- Fetches the most up-to-date code from the specified branch in the remote repository AND merges it into whatever branch you have checked out. Because this command does both, it will be favored over **git merge** in other examples in this document.

Example

- **git pull origin checked-out-branch**
 - To make sure the branch you have checked out is up-to-date.
- **git pull origin source-branch**
 - Fetch the most recent code from **source-branch** and merge it into **checked-out-branch**.

push

- Pushes local code on the checked-out-branch to the branch of the same name on the remote repository. If the remote branch doesn't

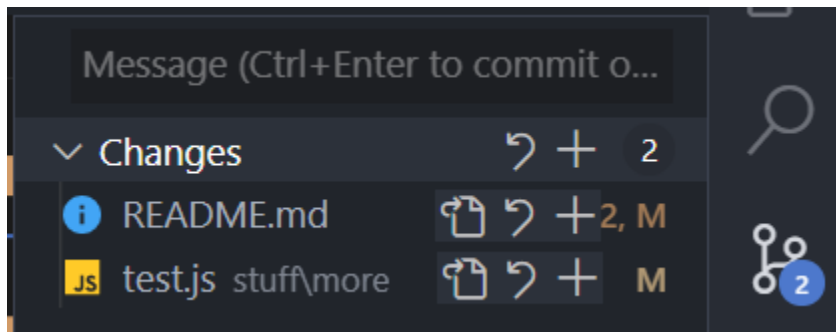
exist, it will be created on GitHub. This is how you get your new code to be available to your collaborators so they can pull it.

Example

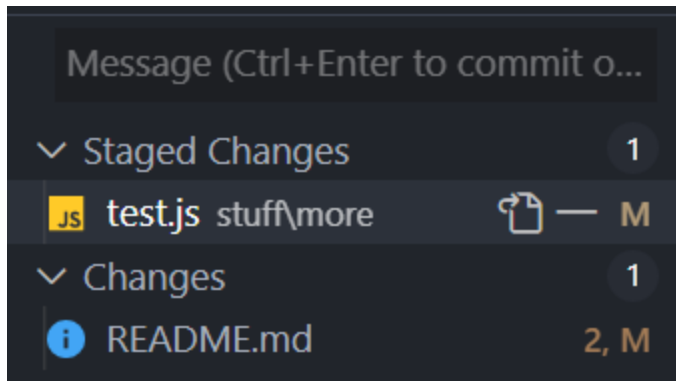
- **git push origin checked-out-branch-name**

Staging, Committing & Source Control Panel

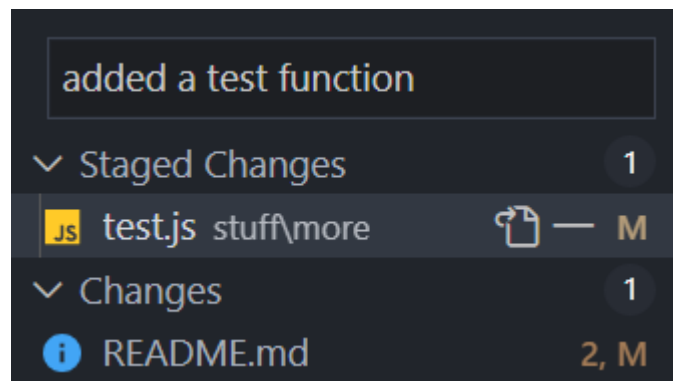
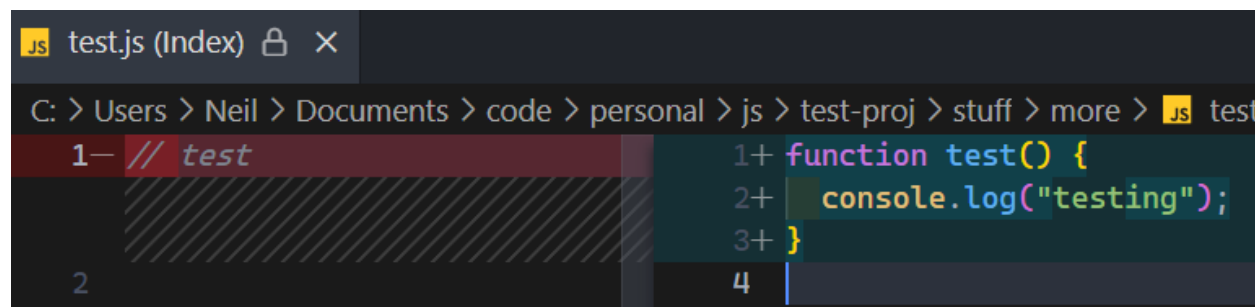
- Commit your changes frequently so they are small and so it's easier to have concise but descriptive commit messages. This makes it easier to revert back to a prior point if anything goes wrong without reverting too much because commits were too few and far between.
- Before you can commit your changes, you need to stage them first. Committing changes means they are ready to be pushed to the remote repository on GitHub.
- You can choose to stage and then commit only some of your changes.



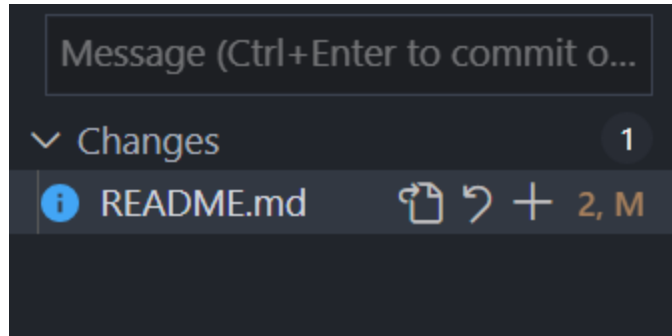
- - The + sign at the top next to Changes will stage all changed files. You can commit multiple staged files at once with one commit message but it's often better to commit them separately so you can have more specific and concise commit messages.
 - Equivalent to **git add .**
 - The + sign next to individual files will stage just that file.
 - Equivalent to **git add path/to/file.ext**



- Only the test file is staged and ready to be committed. The minus sign will unstage it.
- Clicking on the name of the file (not any of the icons) will open a preview of what code was added / removed. This is helpful to remind yourself what you changed and how to describe it in your commit message.



- Commit
 - Ctrl / Cmd + Enter to commit.
 - Equivalent to `git commit -m "added a test function"`



- - The committed file is no longer shown unless more edits to it are made.
- **git pull origin dev**
 - After you commit your changes, make sure whatever branch you are on has the latest dev code pulled and merged into it.
 - Handle any [merge conflicts](#) so they don't show up when you open a pull request.
- **git push origin your-checked-out-branch-name**
 - Push your branch's code to GitHub.
- [Open a Pull Request on GitHub](#) to merge your branch into dev.

Branch Naming

- A more simplified version (not as good of a practice) so that you don't have to create so many branches and merges could be using branches that are named after the developer that is working on them so they can be reused for multiple features and bug fixes.

feature/describe-the-feature

- For branches where code is added to create a new specific functionality. Do not add 5 new functionalities in a single feature branch, make multiple feature branches for separate features.
- Example: **feature/shopping-cart-item-quantity**
 - Building a feature to adjust the quantity of an item in the shopping cart.

bugfix/describe-the-bug

- For branches that fix bugs instead of adding new features.

- Example: **bugfix/stale-cart-total**
 - Fixing a bug where the shopping cart total is not updating after an item is removed.

refactor/describe-the-refactor

- Refactoring (re-writing code to make it cleaner) code, but not to fix bugs or add new features.
- Example: **refactor/cart-total-calculation**
 - Refactoring the code that calculates the shopping cart total to be easier to read / understand. Perhaps this would entail breaking it into small functions and using more descriptive variable names.

docs/describe-the-docs

- Adding some documentation (README) or code comments, but not actual code.

experimental/describe-the-experiment

- Experimenting with some code that may or may not be used. If it ends up being used for a new feature, create a feature branch for it.

test/describe-the-test

- Adding code that tests other code (unit testing, etc).
- Example: **test/cart-total-calculation**
 - Adding code that tests the code that calculates the shopping cart total to make sure it gives expected results with multiple tests, including edge cases.


GitHub Pull Request

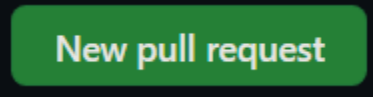
- Local branches that are pushed to GitHub can be merged into other branches through the GitHub website GUI. [Alternatively, you can also merge the code on your PC](#) and then push the updated branch

that had code merged into it onto GitHub so others can pull the updates. However, large projects will usually require pull requests on GitHub or whatever website they use for code to be merged so it can go through a review process.

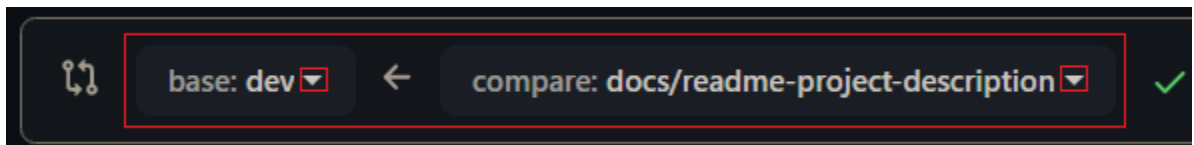
Create A Pull Request

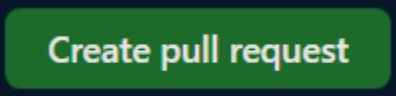
- In this example, the branch **docs/readme-project-description** is checked out.
- **git pull origin dev**
 - To make sure your branch has the latest **dev** code pulled and merged into it.
- **git push origin docs/readme-project-description**
 - To make sure GitHub (remote) has your (local) latest changes for this branch.

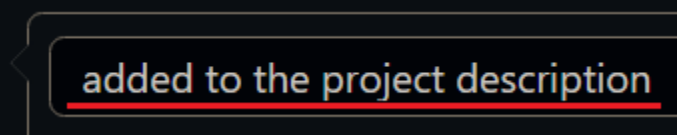
-  Pull requests At the top of the repository page.

- 

- Select the branches to merge



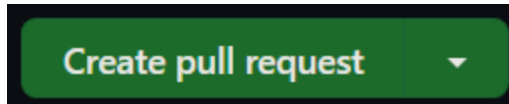
- 

-  describe the code being merged.

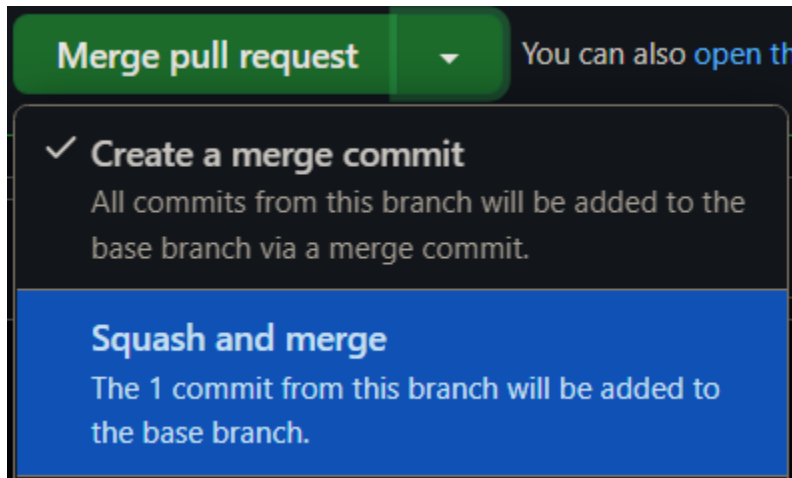
- If you want to practice the code review process, you can do this and wait for approval before merging the pull request.



-



- Wait for approval from code reviewers if you added any.



-

- It is typically a good idea to squash commits because it will squash lots of commits made while working into something more streamlined that is easier to follow in the remote repositories commit history.
 - Optionally leave a comment in the box below if there is useful context that happened between the time of opening the pull request and merging it.
- You can now click the delete branch button on GitHub as long as it isn't deleting the **dev** or **main** branch or some other important branch. The merged branch is typically a branch that is done being used. Deleting a branch on GitHub does not delete it on your computer. You can **git branch -D branch-name** to do that.
- **git checkout dev**
- **git pull origin dev**
 - Since the remote **dev** branch just got updated, pull it to your local PC so it is up-to-date. Let your teammates know the **dev** branch has been updated so they can pull it too.

Terminal Pull Request Link

- When a local branch is pushed to the remote GitHub repository for the first time, it automatically provides a link in the terminal to open a new pull request.

- Hold ctrl or cmd and click the link:

```
remote: Create a pull request for 'docs/readme-project-description' on GitHub by visiting:  
remote:      https://github.com/neilm813/test/pull/new/docs/readme-project-description
```

Merge Conflicts During Pull Request Merging

- If there are a small amount of merge conflicts, they can be resolved on GitHub through their GUI. If there are a decent amount, it's easier to cancel the pull request and then [merge it yourself](#) in VSCode and resolve the conflicts with VSCode's merge conflict GUI.

Workflow Examples

- These examples assume you have a GitHub repository [already created](#) with a **dev** branch created and have already read the introductory info above.

New Shopping Cart Feature

- Imagine you are tasked with adding adjustable item quantities in a shopping cart.
- **git checkout dev**
 - Prepare to create a new branch that branches off of **dev**.
- **git pull origin dev**
 - The new feature branch should have the latest **dev** code when it's created.
- **git checkout -b feature/shopping-cart-item-quantity**
- Let's say you add some HTML (input box and label) and CSS (alignment and input box styling) first.
- [Stage and commit](#) the changed HTML and CSS files. Example commit message: **"Added quantity input, label, and styles."**
- **git pull origin dev**
 - Make sure any updates to **dev** are merged into your feature branch. Do this frequently to help avoid merge conflicts.
 - **git push origin feature/shopping-cart-item-quantity**
 - The first push on a new branch creates it on the GitHub remote repository. It's now safer that your code isn't only saved on your computer. You or anyone else with access to the repository can get this code on another computer.

- Maybe you take a break, or it's the end of the day and you resume working later.
- **git pull origin dev**
 - If you took a break.
- Edit a JS file to add the functionality of updating the carts price on quantity change, verifying stock, etc.
- [Stage and commit](#) the completed JS file changes.
- Now that the feature is complete, it's ready to be pushed to GitHub so it can be merged into **dev**.
- **git pull origin dev**
 - Make sure any updates to dev are merged into your feature branch. Do this frequently.
- **git push origin feature/shopping-cart-item-quantity**
- [Open a Pull Request](#) on GitHub to merge your completed branch into the **dev** branch.

Working on Multiple Branches

- Say you are working on a new feature to replace a multi-select dropdown menu with alphabetized checkboxes that are for adding categories to a product, but then you need to pause work on this to work on something else. Imagine you are told to prioritize a more important bug where buying a specific combo of products is supposed to trigger a discount but it's not.
- **git checkout dev**
 - Prepare to create a new branch that branches off of **dev**.
- **git pull origin dev**
 - Get the latest **dev** code from GitHub so when you create a new branch it will have the latest.
- **git checkout -b feature/category-checkboxes**
- Say you added HTML & CSS code to display the checkboxes but haven't alphabetized them yet or finished the functionality of them, and now you are told to switch to fixing the above mentioned bug.
- [Stage and commit your changes](#).
- **git pull origin dev**
- **git push origin feature/category-checkboxes**

- Make sure GitHub has your work in progress feature branch up-to-date for later. Maybe someone else will take over working on it or you will come back to it. Any teammate can create this branch locally and then pull it now.
- **git checkout dev**
 - Prepare to create a new branch that branches off of **dev**.
- **git pull origin dev**
 - Make sure your **dev** is up-to-date and also so the new branch has the latest code when it's created.
- **git checkout -b bugfix/make-it-a-combo-discount**
- Make some code changes, but not finished yet, but it's getting late so you will continue working on it tomorrow.
 - [Stage and commit your changes](#).
- **git pull origin dev**
- **git push origin bugfix/make-it-a-combo-discount**
 - Make sure GitHub has your work in progress so it's not only on your PC.
- In the morning you start working on the bug fix again. The bug fix branch is still checked out.
- **git pull origin dev**
 - To make sure you have the latest **dev** code. Doing this often makes it less likely you'll run into merge conflicts.
- Finish fixing the bug.
- [Stage and commit your changes](#).
- **git pull origin dev**
- **git push origin bugfix/make-it-a-combo-discount**
 - Get your finished bug fix code onto the remote GitHub branch.
- [Open a Pull Request](#) on GitHub to merge your completed branch into the **dev** branch.
- It's the end of Friday, happy Friday. You are going on a weekend vacation. On Saturday you spend all day driving in traffic, vacations are fun!
- On Sunday you decide you aren't coming home until the middle of the week but are going to work from your AirBnB during the week on the laptop that you brought which was not the desktop PC you were working on at home. Good thing you followed good git

branching practices so you can easily access your work-in-progress feature you need to finish.

- **git checkout dev**
 - Assuming your laptop has already cloned the GitHub repository, created local branches, installed required packages, and set up any local configuration files that were not on GitHub because they were in the **.gitignore**
- **git pull origin dev**
 - Get the latest code, some maniacal **dev** worked all weekend and merged a bunch of code to **dev**.
- **git checkout -b feature/category-checkboxes**
 - Create the branch locally on your laptop since it was created on your home computer and not your laptop originally.
- **git pull origin feature/category-checkboxes**
 - Pull your partially completed code from GitHub.
- Edit files, write code, drink coffee, commit frequently and descriptively, write code, complete the feature. You're a digital nomad, nothing can stop you, except bad WiFi.
- [Stage and commit your changes](#).
- **git pull origin dev**
- **git push origin feature/category-checkboxes**
- [Open a Pull Request](#) on GitHub to merge your completed branch into the **dev** branch.

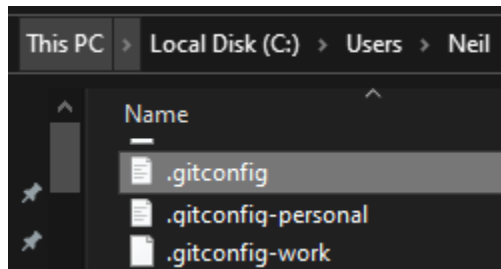
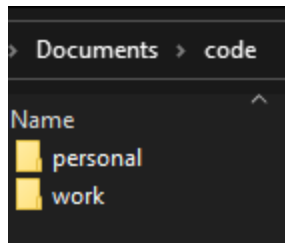
Configuring Multiple Git Accounts

Windows Example: Work <-> Personal

- Say you have a git account specific for work, and a separate one that is used for personal projects outside of work and you want git to automatically know which one to use depending on which folder your repository is in on your computer.

Folder Structure

You'll need to organize your git repositories into separate folders.



.gitconfig

```
[includeIf "gitdir:C:/Users/Neil/Documents/code/work/"]
  path = C:/Users/Neil/.gitconfig-work
[includeIf "gitdir:C:/Users/Neil/Documents/code/personal/"]
  path = C:/Users/Neil/.gitconfig-personal
```

.gitconfig-personal

```
[user]
  email = personal-email@gmail.com
  name = Your Name
[credential]
  helper = wincred
[filter "lfs"]
  clean = git-lfs clean -- %f
  smudge = git-lfs smudge -- %f
  process = git-lfs filter-process
  required = true
```

.gitconfig-work

```
[user]
  email = work-email@gmail.com
  name = Your Name
[credential]
  helper = wincred
[filter "lfs"]
  clean = git-lfs clean -- %f
  smudge = git-lfs smudge -- %f
```

```
process = git-lfs filter-process  
required = true
```