

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Team Week Objectives

Text

You're a team and you're building something awesome!

Spend time together as a team brainstorming your vision. Decide how you want to communicate and work together over the next 4 days. All ideas should be welcome and all members given roles. You can change partners daily or play to the strengths and preferences of your members (business logic vs user interface logic). Remember that in addition to coding skills, communication is one of the most important elements of building great applications. Let your team be one that fosters direct, honest communication and encourages every member's voice!

For the presentation, determine what the minimum viable product for demonstration is. A minimum viable product, or MVP, is a development approach where an application is created with the minimum sufficient features necessary to demonstrate it to users and/or investors. Additional features are implemented when/if time and resources allow. When you create an MVP, it should be a prototype of your idea and have the basic core elements in place so your audience (e.g. peers, investors, clients, future employers, future users, etc.) can understand what your vision is. Try to be

both ambitious and realistic. Use a whiteboard or paper or online storyboard application if that would help everyone with the overview and the plan!

You are going to create something incredible. Dive in!

Code Review Objectives

At the end of the week, you will present your group project at Thursday's Trade Show, where it will be reviewed for the following objective:

- Participation in creating and presenting a project, and collaborating effectively with teammates.

Next (/intermediate-javascript/team-week/git-with-collaborators-setup)

Lesson 1 of 13

Last updated more than 3 months ago.

disable dark mode



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Git with Collaborators: Setup

Text

Working in a team rather than alone can allow you to build more advanced projects, but it can be a little tricky to divide up the work and make sure that no one is overwriting anyone else's code unintentionally. We have used Git to backup our work and save our changes as we go, but it is also a powerful tool for working in a group.

To determine the best workflow for team development, a few decisions have to be made regarding the roles of team members. On GitHub, developers contributing to the project can be owners, collaborators or contributors. In this lesson, we detail how a team member's role will determine the workflow for team development.

Remote Repository Ownership

One of the first decisions a team will need to make is to determine who will own the GitHub remote repository. Every project should have one main repository that is created with a single GitHub account. If there is a logical owner or team leader for the project, it makes sense that the main repository is created with their account. If not, the team should determine who will create the main and therefore be the owner.

The owner will have the following privileges for the repository:

- add and remove collaborators
- change the visibility of the repository (public vs private)
- delete the repository
- implement all collaborator privileges

Once an owner is determined, other developers can work on the code as either collaborators or contributors.

Collaborators

Collaborators are members of the core development team and are officially designated as collaborators by the owner. When working in teams at Epicodus, team members are collaborators.

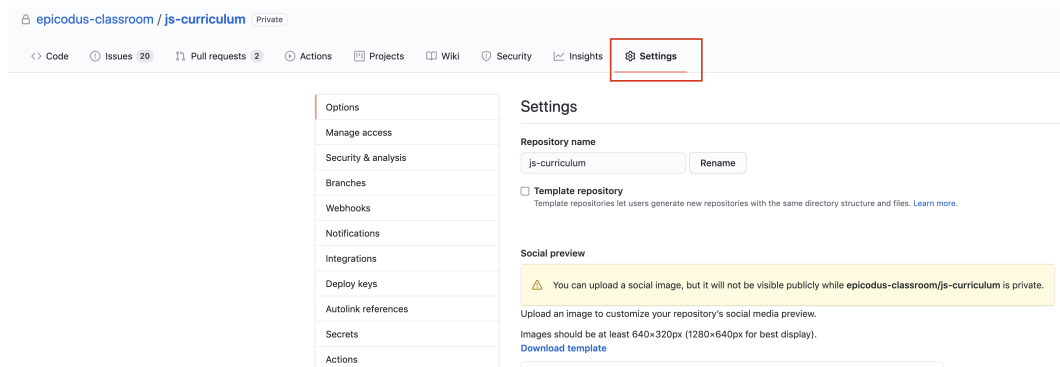
Collaborators have the following privileges for the repository:

- push to (write) and pull from (read) the main repository
- manage issues
- merge and close pull requests (code from contributors)
- remove themselves as collaborators

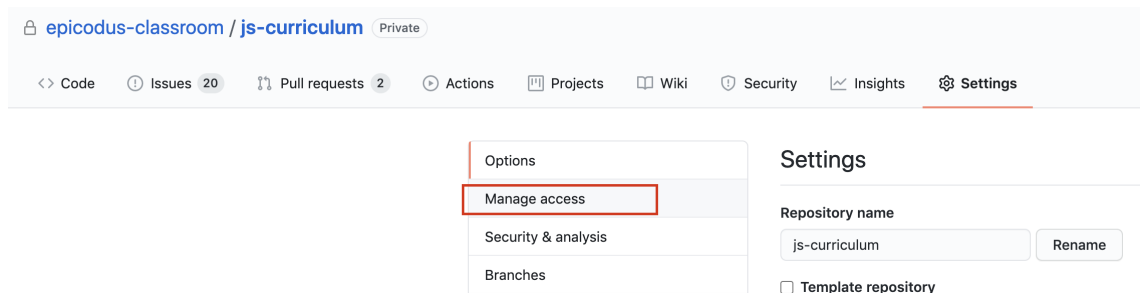
They will use a [branching and merging] workflow when they add and modify code for the project.

To assign a person as a collaborator, the owner will need to follow these steps:

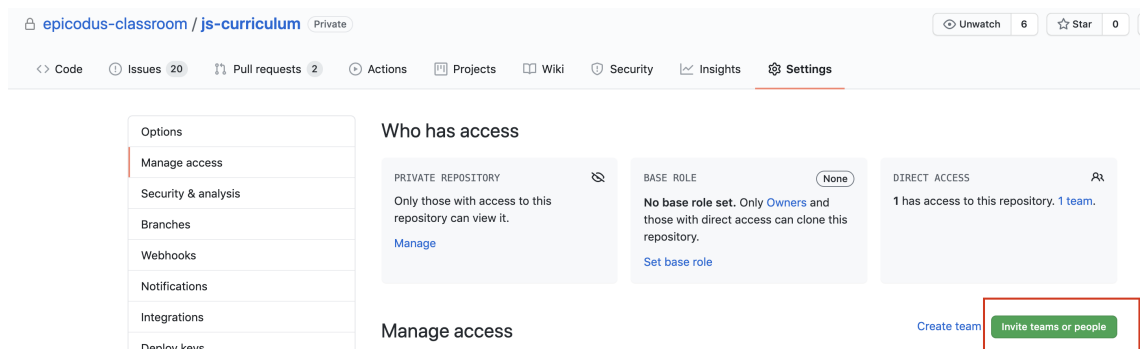
1. Go to the `Settings` menu on the right side of the main repository (see in red box below).



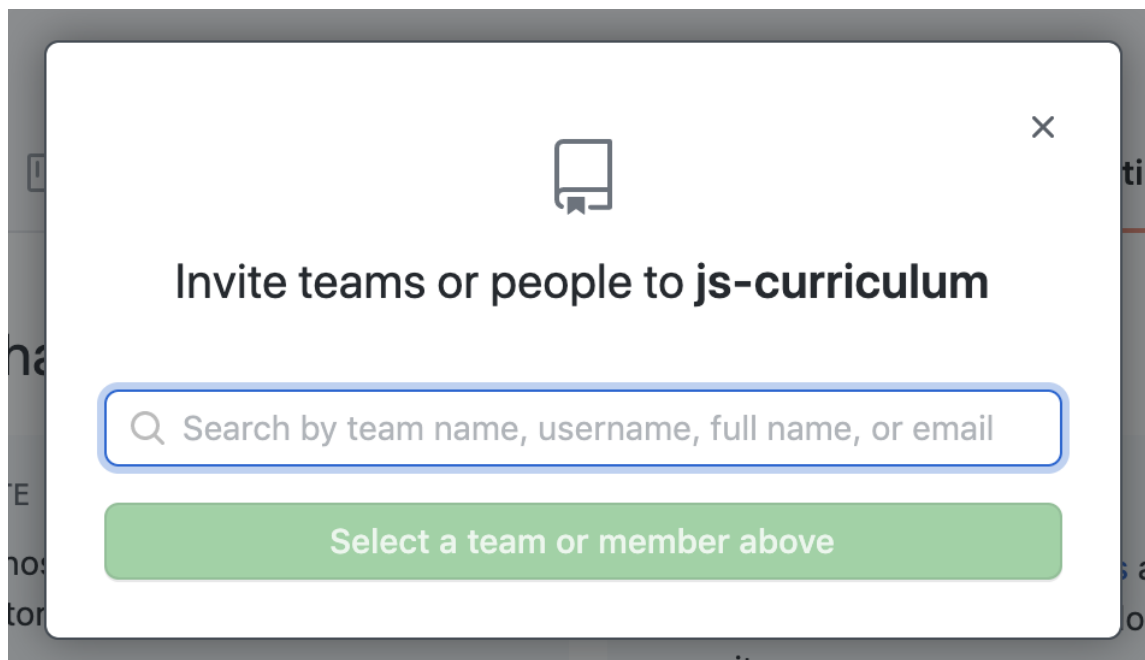
2. Go to Manage Access on the left side menu:



1. Click on Invite Teams or People :



1. Add a team member by GitHub username, full name or email in the Collaborators form. GitHub will helpfully offer to autopopulate, but be careful! It's easy to give a random account access otherwise.



1. Click `Select a team or member above` to add the GitHub user to the project. At that point, the GitHub user will be sent a confirmation email. Once they have confirmed, they will be added to the project.
2. You may optionally update a user's role in the `Manage Access` pane. If you want to be very careful about access, you'd give `Read` access, which allows contributors to create and comment on issues and also create pull requests. For group projects, it's more likely that you will give `Write` or `Admin` access. `Write` access will allow contributors to make direct changes to the repository, including pushing to the repository and managing issues. `Admin` access allows additional functionality such as adding collaborators and updating repository settings.

Contributors

Anyone can be a contributor on an open source repository. Contributors are interested developers that want to offer code but can NOT make commits directly to the main repository. The workflow of a contributor uses [forking and pulling] to submit their

code for review (pull request) to the core development team. The core team can then determine whether to merge the code with the main or not.

[Previous \(/intermediate-javascript/team-week/team-week-objectives\)](#)

[Next \(/intermediate-javascript/team-week/git-with-collaborators-workflow\)](#)

Lesson 2 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Git with Collaborators: Workflow

Text

When we work on a development project as a collaborator, we have commit privileges to the main repository on GitHub. The development workflow for collaborators in a team takes advantage of some Git functionality that we don't often use on pair projects: **Branching** and **merging**. Below is a *general* outline of this workflow. Use this lesson as a reference when working on your group project this week:

Git Team Workflow

1. Setup Github Repo

Build a repo on Github and add all team members as collaborators (<https://www.learnhowtoprogram.com/lessons/git-with-collaborators-setup>). Clone the GitHub main repo to each pair's desktop with `$ git clone <repo-url>`. Navigate to the project with `$ cd <project-directory-name>`.

2. Create Branches

Each pair creates (and switches to) their own feature branch locally by running `$ git checkout -b <branch-name>`. Note that repos come with a main branch by default; you will not need to create one manually.

Tip: If you're ever uncertain which branch you're currently on, run `$ git branch`

3. Code

Pairs complete work on their own branches, adding and committing throughout the process.

4. Pull Origin Main into Local Main

Before pushing work, pairs pull any new code teammates may have merged into the Github main branch (AKA *origin main* or *remote main*) into their local main branch.

This is done by navigating into local main with `$ git checkout main`, then running `$ git pull origin main`. This pulls code from Github's main into the local main.

Generally speaking, this command triggers one of the following three results:

Example A - Pulling down no new changes. There is no new content to pull into the local branch, it is already up-to-date.

```
$ git checkout main
Switched to branch 'main'
Your branch is up-to-date with 'origin/main'.
$ git pull origin main
From https://github.com/test-user/my_project
* branch          main      -> FETCH_HEAD
Already up-to-date.
```

Example B - *Pulling down new changes with no merge conflicts.* There was new content from the Github's main (also known as *origin main*), but Git was able to merge it into the branch automatically.

```
$ git checkout main
Switched to branch 'main'
$ git pull origin main
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/test-user/my_project
* branch          main      -> FETCH_HEAD
  2833d6c..51f2f03  main      -> origin/main
Updating 2833d6c..51f2d03
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Example C - *Pulling down new changes with merge conflicts.* There was new content from the Github's main, but Git was *not* able to merge it into the branch automatically. The user will need to do so by hand.

```
$ git checkout main
Switched to branch 'main'
$ git pull origin main
From https://github.com/test-user/my_project
* branch          main      -> FETCH_HEAD
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

5. Resolve Conflicts (if necessary)

If conflicts occur (as seen in Example C above), conflict tags will appear in the impacted files:

```
...
<<<<<< HEAD
    This is code from the local main branch!
=====
    This is code from the origin main branch!
>>>>>>
....
```

Content above the ===== refers to code from the branch you wish to update (in this case, local main). Content below comes from the branch that is *not* being updated (in this case, Github's main). Resolve these conflicts by replacing everything between <<<<<< and >>>>>> with the code you ultimately want in the project's 'final draft' or main. Then remove conflict tags, and commit the changes.

6. Merge Local Feature into Local Main

After gathering the most up-to-date code from main origin, ensure code in the local feature branch works correctly with code from the main branch by merging feature branch into local main.

Confirm you're still located in your local main branch by running `$ git branch`. Then, merge local feature into local main with `$ git merge <feature-branch-name>`. If any merge conflicts occur, follow the steps above.

7. Add Local Main Code to Origin Main

Pairs ensure the application still looks and functions correctly. Once everything is working as desired, the local version of the main branch can be added to origin main.

There are two primary ways to do this. If all of the collaborators for a project are not physically present to review code, then it is best practice to submit a Pull Request via GitHub so that each member has a chance to review the changes before any new code is added. If all collaborators are present and approve of the changes to the code, or if you are working solo, then it is acceptable to merge into main directly and push the updated main to Github.

If all collaborators are **not** present (most common):

Pull Requests

a. Push Branch to Github

Feature code has been merged into local main branch successfully, and any subsequent refactoring committed. Push this to GitHub as the new feature branch by running `$ git push origin <branch-name>`.

b. Pull Request

A pull request is created by navigating to the feature branch on Github, selecting "New Pull Request", clicking "Create Pull Request", and including a description of new features in the resulting form. Finally, "Create Pull Request" will create and send the request to project collaborators.

c. Review and Merge

Repo owners and collaborators may then view this notification for more information, and options to merge the request into main. More information on that is available here (<https://help.github.com/articles/merging-a-pull-request/>).

If **all** collaborators are present, and have okay'd changes, or you are working solo:

Merge to Main

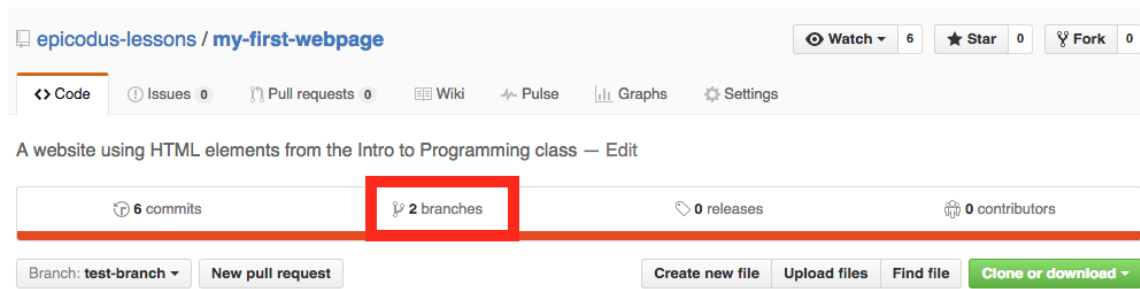
a. Merge Feature into Main

Feature code has been merged into the local main branch successfully, and any subsequent refactoring committed. Contents of local main are then pushed directly into origin main by switching to main with `$ git checkout main`, and merging the appropriate feature branch into main with `$ git merge <branch-name>`.

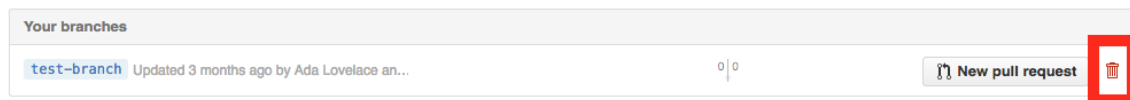
Again, the result should be very similar to one of the code snippets listed in Step 5. If examples A or B are received, there is either nothing new to merge or new code was merged automatically. If something akin to example C is received, pairs must repeat Step 6 in order to resolve any merge conflicts.

8. Delete

If merging was successful, the feature branch may be deleted if it is no longer in use. The easiest way to do this is through the GitHub repo. Simply visit the repository, and select the "branches" option:



Then, in the "branches" area of your GitHub repository, select the red delete icon next to the branch your group would like to delete:



Previous (/intermediate-javascript/team-week/git-with-collaborators-setup)

Next (/intermediate-javascript/team-week/rewriting-history-with-git)

Lesson 3 of 13

Last updated more than 3 months ago.

disable dark mode



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Rewriting History with Git

Text

We always want to include clear and detailed messages with our commits. However, even the most careful developers can make mistakes. Thankfully, git offers several options to re-write commits.

But first, note that we need to be very careful when modifying Git history. This is especially true if you are working collaboratively with others: If you modify an existing commit, it actually *removes* the old commit from the project and makes a *new* one in its place. If others are working on the same branch they will still have your old commits when they pull down any changes, and will be asked to merge your *rewritten* commits with your *new* commits. This can create a real mess. For that reason, **you should not modify commit history on a branch others are also working on.**

However, in some circumstances it is necessary to modify your **local** commit history (that is, commits that exist locally on your machine, but have not been pushed to Github yet) before pushing your repository. In this lesson, we'll cover several ways to make changes to your Git commit history.

git log --oneline

We can see a project's commit history with the `git log` command. And we can add the `--oneline` flag to display the history in an easier to read format:

```
$ git log --oneline
d7e33de something css related
32ccc0b do something with html
9db82b6 update readme
79e9de2 add readme
827ad58 add initial files
```

We can see that this project contains some poorly-worded commits. There are also two consecutive commits about the same feature. Let's clean up this Git commit history!

git commit --amend

First we'll learn how to modify the **most recent** commit. Let's say we committed the right files, but we messed up the commit message. In this case we can use `git commit --amend` to simply update the commit message:

```
$ git commit --amend -m "add styling to main page"
```

This changes the commit message on the most recent commit from "something css related" (as seen in the git log above) to "add styling to main page". Note that it also assigns that commit a new id, since it deleted the old commit and created a new one in its place. The Git history reflects the change:


```
$ git log --oneline
6ad9b62 add styling to main page
32ccc0b do something with html
9db82b6 update readme
79e9de2 add readme
827ad58 add initial files
```

If we accidentally forgot to include a file in the previous commit, we could also use `git commit --amend` to add a file to the previous commit:

```
$ git add index.html
$ git commit --amend -m "add styling to main page"
```

git rebase -i

`git commit --amend` is an easy way to modify our most recent commit, but if we need to modify history going *further* back then we'll need to use the powerful (but potentially dangerous!) `git rebase -i` command.

`git rebase -i` allows us to change commit messages and combine multiple commits by "squashing" them together. Be particularly careful with this command. It permanently deletes all commits from the point you're modifying onward, replacing them with new commits.

Changing old commit messages with git rebase -i

Now, let's reword our second-to-last commit's message. If we type `git rebase -i HEAD~2` it will launch the system editor, where we'll see a Git rebase file containing the two most recent commits. It will look something like this:

```
pick 32ccc0b do something with html
pick 6ad9b62 add styling to main page
```

To reword some of the commit messages, we can change `pick` before those lines to `reword`, as follows:

```
reword 32ccc0b do something with html
pick 6ad9b62 add styling to main page
```

When we save and close that file, we're immediately presented with a commit message file in the editor, allowing us to update the commit message we marked for rewording:

```
do something with html

# Please enter the commit message for your changes. Lines s
tarting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# Date:      Sat Mar 19 10:56:31 2016 -0700
#
# rebase in progress; onto 9db82b6
# You are currently editing a commit while rebasing branch
'main' on '9db82b6'.
#
# Changes to be committed:
#       modified:   index.html
#
```

Let's change the commit message at the top to *add welcome message to index.html*. We save and close the file, and now the changes are reflected in the Git history:

```
$ git log --oneline
0e3e0bc add styling to main page
1a940bb add welcome message to index.html
9db82b6 update readme
79e9de2 add readme
827ad58 add initial files
```

Note that the last two commits (the two we rebased) now have new id's, because the old commits were deleted and replaced with new ones.

Combining Multiple Commits with `git rebase -i`

Now, let's say we want to combine multiple commits for one feature into a single commit before publishing to Github. We want to combine the "add readme" and "update readme" commits.

We can type `git rebase -i HEAD~4` to bring up the four most recent commits in the rebase editor window. To combine a commit with the previous commit, we can change `pick` to `squash`.

Let's change `pick` to `squash` on the "update readme" commit in order to combine that one with the "add readme" commit directly preceding it:

```
pick 79e9de2 add readme
squash 9db82b6 update readme
pick 1a940bb add welcome message to index.html
pick 0e3e0bc add styling to main page
```

After saving and closing this file, we're presented with a commit message file. We can write the commit message for the new *combined* commit in this file:

```
# This is a combination of 2 commits.  
# The first commit's message is:  
  
add readme  
  
# This is the 2nd commit message:  
  
update readme
```

We can update the first commit message to "create project readme" and simply delete the "update readme" line, since we're eliminating the second commit from our project:

```
# This is a combination of 2 commits.  
# The first commit's message is:  
  
create project readme  
  
# This is the 2nd commit message:
```

After saving and closing that file, the Git history reflects the change:

```
$ git history --oneline  
59e7083 add styling to main page  
2b1b367 add welcome message to index.html  
4c876ed create project readme  
827ad58 add initial files
```

Pushing Rewritten History to GitHub

Again, be extremely careful when changing history on a public repository. Do not make history changes to a remote repository being worked on by other developers. However, you may push rewritten history to a public repository if you're the *only* developer working on it.

Normally you'll receive the following error if you push a project to Github, rewrite history, and then attempt to push the project again:

```
$ git push origin main
To https://github.com/epicodus-lessons/hello
! [rejected]          main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/epicodus-lessons/hello'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This is simply because the local commits and remote commits no longer match. In order to force Github to throw away the old version of the project and replace it with the newly-edited local commit history, you must use the `--force` option:

```
$ git push origin main --force
```

Rebasing to Clean Merge

Note that `git rebase` without the `-i` flag is often used for doing a "clean merge" of different branches. We won't cover this concept yet, simply know this option exists in case you encounter it in the

future.

Or, if you'd like to explore it now, you can find more information here (<https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase>).

[Previous \(/intermediate-javascript/team-week/git-with-collaborators-workflow\)](#)

[Next \(/intermediate-javascript/team-week/learning-more-about-git\)](#)

Lesson 4 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Learning More About Git

Text

At this point, you should have a solid grasp of the basics of git. However, it's likely that some concepts remain intimidating. For instance, merging branches and the possibility of merge conflicts or losing code can still feel scary. Also, the basic underpinnings of how git works (it's based on graph theory) probably aren't very clear.

Check out the site Think Like (a) Git (<http://think-like-a-git.net/>). The site is a comprehensive and fun overview of git by Sam Livingston-Gray, a Ruby developer based in Portland, Oregon.

We recommend reading through each of the sections (which are short, quick reads) starting from the beginning. Also, make sure you read the following sections, which may be particularly useful during team week:

- Testing Out Merges (<http://think-like-a-git.net/sections/testing-out-merges.html>): This section illustrates several patterns for beginners to merge their code before they move on to "Black Belt Merging."
- Rebase From the Ground Up (<http://think-like-a-git.net/sections/rebase-from-the-ground-up.html>): This section goes into more depth about how git rebase and git

cherrypick work. You may not use these commands much — yet — but they will be helpful long-term in your growth as a developer.

[Previous \(/intermediate-javascript/team-week/rewriting-history-with-git\)](#)

[Next \(/intermediate-javascript/team-week/practicing-the-git-workflow\)](#)

Lesson 5 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Practicing the Git Workflow

Text

We've covered the basics of using a git workflow — now it's time to practice a little. We recommend going through this practice *before* you actually start working on your group project. This way, you'll be more equipped to use this workflow, and to deal with any issues that come up.

This will be an extremely simple exercise — we will just make some small modifications to a README. That's because the goal here isn't to code, but to practice the workflow itself and also deal with a merge conflict.

Start by creating a new directory called `workflow-practice`. It will have just one file: `README.md`.

README.md

Hello!

Next, let's run the `git init` command. Create a `git-workflow` repository in GitHub and then connect it to this project using `git remote add origin git-workflow`. This will be a repository we can

throw away later — it's just for practice!

Save the code in the README and make your first commit:

```
$ git add .  
$ git commit -m "add greeting"
```

Once the code is committed, push it to GitHub.

Next, let's create a new *local* branch called `development` :

```
$ git checkout -b development
```

This will create a new branch *locally* (the `-b` flag), and then `checkout` will automatically take us to that branch.

It's common for large companies to have a `development` branch. This isn't the production branch — instead, the developers will merge code into this branch, test that it works, and so on — *before* it gets merged into the main branch, which is usually the production code. In this example, we'll just have the development branch along with the main branch to keep things simple. But in real world applications, there'd be additional feature branches which would get merged into the development branch first, and then tested, all before they are ever merged into main.

It's always a good idea to verify the branch we are on with `$ git branch`. That way, we won't accidentally start modifying code in the wrong branch.

If we run `$ git log`, we'll see that our first commit is in this branch as well. This is expected — when we create a new branch, it copies the code of the branch we are in along with its commit history. This is great for creating experimental features or for testing code.

Now let's modify our README:

README.md

Hola!

Save and commit this code:

```
$ git add .  
$ git commit -m "change greeting to Spanish"
```

At this point, this branch has diverged from the main branch. Couldn't we just merge it into main?

Well, if we are working on the project alone, that would probably be fine. But that's not how development teams work in the real world. Other teams are *also* making changes to the code.

So let's simulate that process. We'll make another change to our code, this time *directly in GitHub*. This represents a second team working on the same code as us. They finished their update first and they've already merged their code into the main branch and pushed it to GitHub.

We can imitate this process by going to the repository in GitHub and clicking on the pencil icon, which allows us to modify the code directly in GitHub's UI.

Now let's change the greeting to French in GitHubs:

Bonjour!

Name the commit "change greeting to French" .

Now we are getting closer to a real world process.

We now have a remote main branch in GitHub which has two commits. We have a local main branch which has one commit. And we have a local development branch which has two commits. All three of these branches *have different greetings*.

So now let's say we want to merge our development branch into main and push that code to GitHub. How would we go about doing that?

Well, we need to switch back to the main branch and pull the latest code from GitHub. Then we need to switch back to the development branch, merge the code from main, make sure it is good to go, and then switch back to main and merge the development branch into main before we push it to GitHub. We'll walk through this whole process more slowly in a moment.

Why this convoluted workflow?

Well, we don't ever want to merge experimental code into main until we know it works. If we have any merge conflicts or other problems, we want them to happen in the development branch. Once everything is fixed and good to go, we'll be ready to merge the code into the main branch, but not before then.

So now let's slow this down. Here's what we need to do locally. We are currently on the development branch. We need to switch over to the main branch.

```
$ git checkout main
```

Next, we need to pull the latest code from GitHub:

```
$ git pull origin main
```

Now the main branch is up to date with the code that the other development team has already merged into main. We *could* merge our development branch into the main branch, but that could potentially lead to a big mess, especially if there are merge conflicts. That's not a good workflow. Instead, we need to play it safe and switch over to the development branch. Alternatively, if we're really worried about making a mistake and messing up code on either the main or development branch, we could create a safety branch called something like `savepoint`. This is one of the techniques suggested in Think Like A Git (<http://think-like-a-git.net/sections/testing-out-merges/the-savepoint-pattern.html>). It's not necessary to do this but it does create an extra layer of security when you're new to merging branches.

Now let's go back to the development branch. We're ready to merge the code from main *into* development. This is a really good time to verify that we are on the correct branch with `$ git branch`. If we are on the wrong branch, we could seriously mess up our code when we try to merge!

Once we've verified that we are on the development branch, we can run the following command:

```
$ git merge main
```

This will merge the main branch *into* the branch we are on — the development branch.

Before we move on to the next step, what do you think is going to happen?

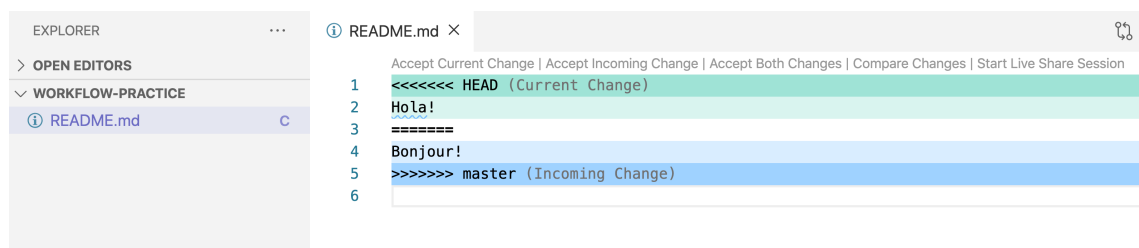
If you guessed that there will be a merge conflict, you're correct.

```
$ git merge main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

There have been two recent changes made to the code — how is Git supposed to know whether the greeting should be hola or bonjour?

When there is a merge conflict, a list with the file (or files) that have merge conflicts will be printed to the command line. We can see this above: `CONFLICT (content): Merge conflict in README.md`.

Merge conflicts can be scary. Fortunately, VS Code makes managing conflicts much easier. If we navigate to the application in VS Code, we'll see the following:



VS Code has helpfully organized the code into two parts: the *Current Change*, which is the code from the branch we are on, and the *Incoming Change*, which is the code coming from the branch we are merging. It's easy to remember which is which by thinking of the branch we are merging as being the *incoming* branch and the branch we are on as being the *current* branch.

We have several choices here. If we click *Accept Current Change*, the text will just be `Hola!` If we click *Accept Incoming Change*, the code will just be `Bonjour!` And if we *Accept Both Changes*, the code will read:

```
Hola!  
Bonjour!
```

Alternatively, we can modify the code directly — but if we do so, we have to make sure we remove ===== and >>>>>> main from our code in addition to making the changes. Git inserts these lines automatically to show where the code diverges in each branch.

Because we are collaborative and both teams have done great work, we are going to *Accept Both Changes*. Note that your use cases will vary greatly depending on the situation — sometimes you'll want both changes, sometimes you'll want just one, and sometimes you'll want parts of both — which means you'll need to update some of the code manually before declaring the merge a success and committing it.

Next, we are ready to save and commit our code.

```
$ git add .  
$ git commit -m "update code to include both greetings"
```

At this point, we've cleared up the merge conflict on the development branch, and we've made sure that the conflict posed no risk to the main branch. Once we've verified our code looks good (hopefully by testing it), it's ready to merge into main. Now we can switch over to the main branch and merge our code again:

```
$ git checkout main  
$ git merge development
```

As we'll see, the merge will go smoothly. That's because we've taken care of the merge conflict in the other branch!

At this point, we can safely push our code to the main branch.

While this example is very simple, it should hopefully make the whole process of merging code a bit clearer. Ideally, when working with your group, you should be working on different parts of the code and create different features — which means you won't run into as many merge conflicts. Make sure you practice a good git workflow by following the steps above when merging and dealing with merge conflicts.

If you're still feeling a bit foggy on these concepts, go ahead and repeat the process — add a few more files and then make different changes directly on GitHub and then on your development branch. This will give you the chance to work through a few more simple merge conflicts before you run into one that is potentially bigger and more confusing during your group project.

[Previous \(/intermediate-javascript/team-week/learning-more-about-git\)](#)

[Next \(/intermediate-javascript/team-week/hosting-a-webpack-project-with-gh-pages\)](#)

Lesson 6 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Hosting a Webpack Project with GH-Pages

Text


In this lesson we'll learn how to host our webpack project on **GitHub Pages**, also referred to as **gh-pages**. GitHub Pages is free, and lets us host our HTML, CSS, and Javascript right from the repository. It's a great way to quickly host a web app, and since it's integrated into GitHub, it's a nice option for developers. You may have hosted some projects this way by creating a `gh-pages` branch in your project, and while we'll end up with the same result here, the process for setting it up is slightly different. We'll make some edits to our `package.json`, and run some new commands from our terminal. In this lesson we'll also cover how to deploy a React app built with webpack for students in the React course.

Before we start, one important note about GitHub Pages is that it is **not meant** to host sensitive information, like API keys. The GitHub docs explain more:

GitHub Pages is not intended for or allowed to be used as a free web-hosting service to run your online business, e-commerce site, or any other website that is primarily directed at either facilitating commercial transactions or providing commercial software as a service (SaaS). GitHub Pages sites shouldn't be used for sensitive transactions like sending passwords or credit card numbers.

In other words, any information we put in our code will be available to someone visiting our site. If we want to host a site with sensitive information we would need some kind of backend, or server-side code, to store our sensitive information. For now, that is beyond the scope of what we'll do.

To read more about prohibited uses for gh-pages, visit the following documentation:

-  **Link to GitHub Pages Docs**
(<https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages#prohibited-uses>)

Update the `package.json`

First, we'll add a `repository` key to our `package.json`. We're adding it under the `version` key but the order doesn't matter as long as it is in the top-most level of our package object, the same level as `name` and `version`. After we add that, let's set it equal to an object with a `url` key. Use the following code snippet as an example, replacing `{REPONAME}` with your repo and `{USERNAME}` with your GitHub username respectively.

```
./package.json
```

```
{
  "name": "project name",
  "version": "1.0.0",
  "repository": {
    "url": "git+https://github.com/{USERNAME}/{REPONAME}.git"
  },
  ...
}
```

Next we'll add two new commands to our `scripts` object:

- `predeploy` builds our site and bundles it in the `dist` folder.
- `deploy` pushes the contents of that folder to a new commit on the `gh-pages` branch, creating that branch if it doesn't already exist.

`./package.json`

```
{
  ...
  "scripts": {
    "build": "webpack --mode=development",
    ...
    "predeploy": "npm run build",
    "deploy": "gh-pages -d dist"
  },
  ...
}
```

Install gh-pages Package

Install the `gh-pages` package via the terminal by running the following command in the root of your project's folder:

```
$ npm install --save-dev gh-pages
```

You should see `gh-pages` listed in your `package.json` under `devDependencies`.

Deploy the site

At this point we can run the following command to deploy our site. Make sure to run this command in the root of your project's folder.

```
$ npm run deploy
```

This will build our project and then publish it to GitHub Pages. You may be curious where the `predeploy` script gets used if we aren't using it ourselves. Well, `gh-pages` runs the `predeploy` script automatically before deploying the site to ensure there's an up-to-date build of our project. We can look in our terminal to see exactly which scripts are executed and in what order. Our terminal should show us that the `predeploy` script is run, which itself calls the `build` script, all before the `deploy` script executes. Check out the code snippet below that shows an example of the terminal output for deploying. Note that 'PROJECTNAME' will be the name of your project as listed in the `package.json` file.

```
$ npm run deploy

>PROJECTNAME@1.0.0 predeploy
> npm run build

> PROJECTNAME@1.0.0 build
> webpack --mode=development

Hash: e6e0a675ea138d2edcd8
Version: webpack 4.46.0
Time: 2553ms
...

> PROJECTNAME@1.0.0 deploy
> gh-pages -d dist
```

After we've successfully deployed our site, it will be hosted at a URL like this, where {USERNAME} is your github username and {REPONAME} is the name of your repository :

```
https://{USERNAME}.github.io/{REPONAME}
```

If we want to update our live site with some changes, we'll need to `git checkout` to the branch that has the most up to date code, then run our deploy script. Typically we deploy from the `main` branch but it's possible to deploy from any branch. Each time we deploy, another commit gets made on our `gh-pages` branch. The commit message will be "Updates" by default, if you want a custom commit message you can specify it by using the `-m` option.

```
$ npm run deploy -- -m "Deploy site with new colors"
```

Finally, note that `deploy` can take a minute or two to update, so be mindful of that when checking the live site.

For React Projects

The process is almost identical for deploying a React site to GitHub Pages. We'll still update our `package.json`, install `gh-pages`, and add our new scripts. The difference between this and basic webpack hosting is in the `deploy` script. We need to configure GitHub Pages to deploy the contents of the folder that contains our bundled code, and in a React project that folder is called `build`, as opposed to `dist` in our basic webpack projects. Check out the code snippet below that shows how we've updated the value of the `deploy` script to point to the `build` folder.

./package.json

```
{
  "scripts": {
    ...
    "predeploy": "npm run build",
    "deploy": "gh-pages -d build"
  },
  ...
}
```

Outside of that the steps are exactly the same. Optionally, see this repo for a walkthrough of deploying a React project. (<https://github.com/gitname/react-gh-pages>)

Further Exploration

Configure a Publishing Source

It's possible to configure our GitHub Pages site to publish when changes are pushed to a specific branch, or we can write a GitHub Actions workflow to publish our site. So for instance we could set our `main` branch as the publishing source and have GitHub

automatically deploy any time changes are pushed to the `main` branch. See this page in the GitHub docs for how to set a custom publishing source. (<https://docs.github.com/en/pages/getting-started-with-github-pages/configuring-a-publishing-source-for-your-github-pages-site>)

Enforce HTTPS

By default GitHub enforces HTTPS protocol (not HTTP), so you may get errors if your links use HTTP. For the deployed site you'll need to remove any HTTP content. See this page on enforcing HTTPS in GitHub (<https://docs.github.com/en/pages/getting-started-with-github-pages/securing-your-github-pages-site-with-https#resolving-problems-with-mixed-content>) for more information.

Un-Publish the Website

If you want to remove a site from GitHub Pages so it is no longer hosted, you can find instructions for un-publishing here. (<https://docs.github.com/en/pages/getting-started-with-github-pages/unpublishing-a-github-pages-site>)

And that's it! Our site should be live on GitHub Pages for the whole world to visit.

[Previous \(/intermediate-javascript/team-week/practicing-the-git-workflow\)](#)

[Next \(/intermediate-javascript/team-week/backend-course-preparation-software-installation\)](#)

Lesson 7 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Backend Course Preparation: Software Installation

Text

It's time to prepare for the next course by installing all relevant software! Depending on what track you signed up for, you will install software for C#/.NET or Ruby/Rails.

We have students install software during the weekend before team week for a couple of reasons. The first reason has to do with installation and configuration issues. It's normal to run into some issues when setting up your personal environment, and the way we deal with this is to work through the installation process early so that we can identify and address any issues sooner.

The second reason is that you'll be ready to tackle the remaining pre-work for the next course when it comes time to complete it. Students are required complete pre-work for the next course before the first day of the course. Most students complete the pre-work during the weekend before the course starts, but you can start as early as you like. Take a moment to follow the link below to look at your upcoming course, and notice all of the sections that are labeled "pre-work":

For full-time students:

- Full-time C# and .NET (<https://www.learnhowtoprogram.com/c-and-net>)
- Full-time Ruby and Rails (<https://www.learnhowtoprogram.com/ruby-and-rails>)

For part-time students:

- Part-time C# and .NET (<https://www.learnhowtoprogram.com/c-and-net-part-time>)
- Part-time Ruby and Rails (<https://www.learnhowtoprogram.com/ruby-and-rails-part-time>)

Software Installation by Course

Find the course you are signed up for below and install the necessary software.

C#/.NET

If you are in a track that includes the C# and .NET course, install the packages in the following lessons.

- Installing C# and .NET (<https://www.learnhowtoprogram.com/c-and-net/getting-started-with-c/installing-c-and-net>)
- Installing dotnet script (<https://www.learnhowtoprogram.com/c-and-net/getting-started-with-c/installing-dotnet-script>)
- Installing and Configuring MySQL (<https://www.learnhowtoprogram.com/c-and-net/getting-started-with-c/installing-and-configuring-mysql>)

Ruby/Rails

If you are in a track that includes the Ruby and Rails course, install the packages in the following lessons.

- Ruby Installation and Setup
(<https://www.learnhowtoprogram.com/ruby-and-rails/getting-started-with-ruby/ruby-installation-and-setup>)
- Installing Ruby on Mac
(<https://www.learnhowtoprogram.com/ruby-and-rails/getting-started-with-ruby/installing-ruby-on-mac>)
- Installing Ruby on Windows
(<https://www.learnhowtoprogram.com/ruby-and-rails/getting-started-with-ruby/installing-ruby-on-windows>)
- Installing Postgres
(<https://www.learnhowtoprogram.com/ruby-and-rails/getting-started-with-ruby/installing-postgres>)

Installation Issues and Setup Assistance

Sometimes issues can arise in the installation process. Because setup issues naturally arise, students and teachers collaborate on solving setup issues together during an in-class troubleshooting session at the start of the backend course. Let's review the guidelines and expectations for that now.

Schedule and Expectations

1. Before the course starts, every students installs all tools at home.

The first step is for you to take the time to install the necessary tools for the course at home. You start this process individually during the weekend before team week, and students often finish this process without help. If you run into any installation issues, that's ok. See #2 below.

2. If any installation issue happens, troubleshoot them and take notes.

It's important to take notes and screen shots for any issues or error messages that comes up. You can then share these notes with your peers and teacher during the in-class troubleshooting session. It's

also important to take the time to work through the issue yourself. Whatever you try in the troubleshooting process, take notes. The same goes for any helpful resources you find online. We recommend reading through terminal output and any error logs that come up in the installation process, as they can point what to look into that may be missing.

3. Follow your instructor's directions.

Your instructor may have further instructions for you as far as how or where they want you to take notes on any issues. Your instructor may also ask that you "check in" about how your installation process went on a spreadsheet or whiteboard so they can track issues and organize help. Whatever it is, make sure to follow your instructor's directions.

4. Finish troubleshooting during the first class of the course.

For anyone who hasn't been able to resolve an installation issue, you will group up during the first class of the backend course and continue troubleshooting together. Your instructor will have more directions for you about grouping up, and they will be present and available during this time to also help troubleshoot installation issues. This troubleshooting session is focused on collaboration, so discuss your issues and share your notes with your peers. Situations happen where your classmate has solved a problem you are currently facing and has advice for you, and vice versa!

As you work through your installations, keep in mind that it's common for environment issues to crop up and for them to take a while to resolve. This is why we begin the installation process over the weekend before team week and continue into the first class of the backend course. Make sure to do your part by troubleshooting any issues that come up, and by taking notes on what the issues are and what you did to try and fix them.

If you have any questions or concerns, talk to your instructor about them as soon as you can.

[Previous \(/intermediate-javascript/team-week/hosting-a-webpack-project-with-gh-pages\)](#)

[Next \(/intermediate-javascript/team-week/journal-7\)](#)

Lesson 8 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Weekend

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Journal #7

Text

You were asked to keep a journal while going through the program. Each weekend you'll receive a brief journaling assignment in addition to your coding homework. (Review the *Journaling at Epicodus* (<https://www.learnhowtoprogram.com/introduction-to-programming/git-html-and-css/homework-journaling-at-epicodus>) lesson for a refresher.)

Journal #7 Prompt

Spend several moments thinking critically about the following questions, and record brief yet honest responses. Include a date or timestamp and a brief summary of the prompt to refer back to later.

Think about accomplishments. Specifically, think about the last time you felt legitimately proud and accomplished. Feeling accomplished is about knowing what you can do, asking yourself (or your team) for the best possible work, then putting in the effort to deliver.

You're about to hit a big milestone in your Epicodus career: developing your first group project. Let's reflect on how we can ensure this section is yet another accomplishment in your journey.

- What are your goals for this group project? Besides creating a codebase, what do you hope to accomplish? (For example, are you hoping to explore technologies above and beyond what's offered in our curriculum? Practice your project management skills? List anything applicable!)
- What do you anticipate could potentially stand in your (or your teammates') way to feeling accomplished this section? Consider referring back to previous journal responses. For instance, did you identify yourself as someone that occasionally falls prone to imposter syndrome? Or someone that can become unproductively frustrated by bugs and issues if you don't catch yourself? List anything applicable.
- How could you increase the feeling of accomplishment experienced with your teammates this section? For each potential roadblock you recorded above, list *at least* one thing you can do to proactively prevent it from negatively effecting your group project experience.

Discussion

We'll discuss our responses with our group project teammates at the beginning of our next class. Make sure your responses are recorded before then!

[Previous \(/intermediate-javascript/team-week/backend-course-preparation-software-installation\)](#)

[Next \(/intermediate-javascript/team-week/journal-7-discussion\)](#)

Lesson 9 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Exercise

Monday

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Journal #7 Discussion

Text

As discussed in the *Journaling at Epicodus* (<https://www.learnhowtoprogram.com/introduction-to-programming/git-html-and-css/homework-journaling-at-epicodus>) lesson, each weekend you'll receive a journaling assignment in addition to your coding homework. You'll then discuss your responses with a partner at the beginning of the next class session.

Journal Response Discussion

This Section's Prompt

As you'll recall, you were asked to write responses to the following in your journal over the weekend, as detailed in the Journal #7 prompt:

- What are your goals for this group project? Besides creating a codebase, what do you hope to accomplish? (For example, are you hoping to explore technologies above and beyond what's offered in our curriculum? Practice your project management skills? List anything applicable!)
- What do you anticipate could potentially stand in your (or your teammates') way to feeling accomplished this section? Consider referring back to previous journal responses. For instance, did you identify yourself as someone that occasionally falls prone to imposter syndrome? Or someone that can become unproductively frustrated by bugs and issues if you don't catch yourself? List anything applicable.
- How could you increase the feeling of accomplishment experienced with your teammates this section? For each potential roadblock you recorded above, list *at least* one thing you can do to proactively prevent it from negatively effecting your group project experience.

Discussion Questions

Before beginning your group project, meet with your team and discuss your journal responses as a group using the following questions to guide your discussion:

- What did you all identify as your goals (beyond just completing your project idea) for the section? Are they similar? Different? What can you do as a team to ensure everyone meets these goals together, even if some individuals' goals slightly differ?
- What did you identify as possible roadblocks to your project/team's success, and feelings of accomplishment? How many of these are shared amongst team members? How many are unique to individuals?
- Determine what your team can proactively do *now* to both prevent potential issues later, and maximize what you can accomplish together this section. Take **all** goals and potential roadblocks identified by all team members into account, and develop a plan. (For instance, if one member has a goal to gain project management experience, and another member shared

hesitations about communication skills, perhaps the first team member could be responsible for holding a couple regular team stand-ups a day, to check in and keep communication flowing?)

[Previous \(/intermediate-javascript/team-week/journal-7\)](/intermediate-javascript/team-week/journal-7/)

[Next \(/intermediate-javascript/team-week/pull-requests-with-branches\)](/intermediate-javascript/team-week/pull-requests-with-branches/)

Lesson 10 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Monday

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Pull Requests with Branches

Text

In this lesson, we'll practice making pull requests with branches. This is a common workflow in the industry. In this workflow, a developer pulls down the latest code from the main branch and creates a new branch. When a feature is completed, the new branch is committed and pushed to GitHub. Then the developer makes a pull request so the code can be merged into the main branch via GitHub's UI.

We've already learned one way to merge our code into the main branch in [Practicing the Git Workflow](https://www.learnhowtoprogram.com/intermediate-javascript/team-week/practicing-the-git-workflow) (<https://www.learnhowtoprogram.com/intermediate-javascript/team-week/practicing-the-git-workflow>). While it's important to be fluent in this workflow, it doesn't allow for any oversight of the process. All of the merging happens locally.

In the case of group projects, the entire team should be able to comment on and merge pull requests. In order to do so, everyone must be a collaborator on the project. See the [Git with Collaborators Setup](https://www.learnhowtoprogram.com/intermediate-javascript/team-week/git-with-collaborators-setup) (<https://www.learnhowtoprogram.com/intermediate-javascript/team-week/git-with-collaborators-setup>) lesson for instructions on adding collaborators.

Pull Requests with Branches

Once again, we'll create a very simple project with a README to walk through the process of making pull requests. After all, the goal isn't to code here — it's to practice a new git workflow.

Start by creating a repository in GitHub called `pr-practice`. Clone that down to your local machine and add a README with the following text:

README.md

```
This code needs some updates. Fork it and make a pull request!
```

Next, commit this code and push it back to the main branch. We need to have some starter code before we practice our new git workflow.

Next, create a branch called `new-feature` locally:

```
$ git checkout -b new-feature
```

Now let's update our README (in the local `new-feature` branch, not in the GitHub UI):

README.md

```
This code needs some updates. Fork it and make a pull request!
```

```
New feature added!
```

Next, we need to save and commit our code. Finally, we'll push our `new-feature` branch to GitHub. As of now, it only exists locally.

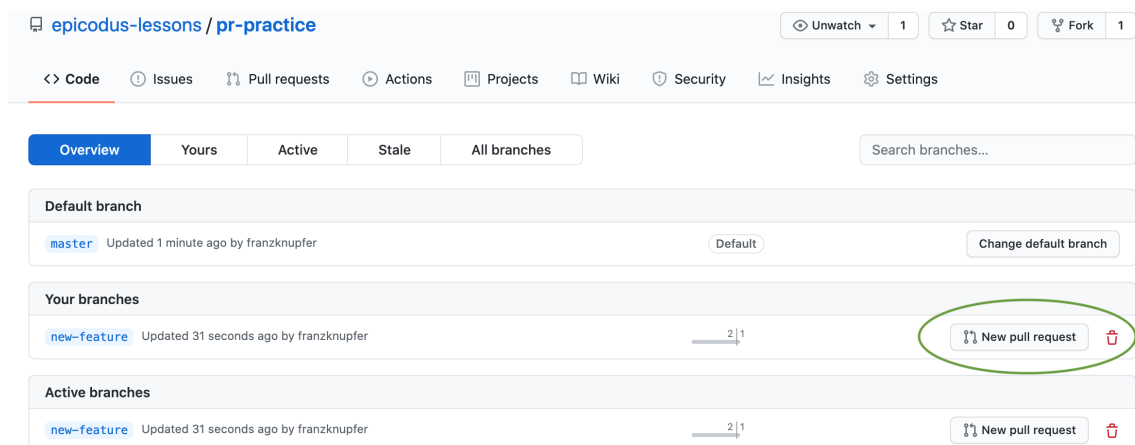
```
$ git push origin new-feature
```

Now our `new-feature` branch exists both locally and remotely, but it still hasn't been merged into the main branch yet.

At this point, we're ready to make our first pull request.

Creating a Pull Request

At this point, we can go to the repository in GitHub and click on the *Branches* tab. This will show all of the branches you've created as well as all of the active branches in the project.



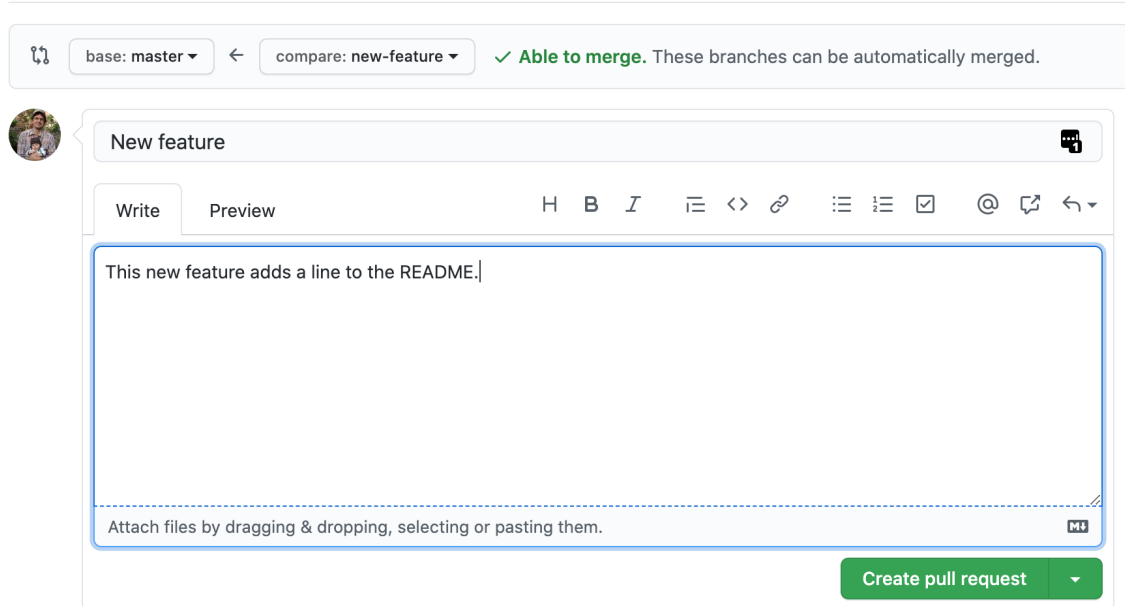
In the image above, we can also see that our branches have a button on the right side of the screen that reads *New Pull Request*. That's exactly what we want. Click on this button to open a new pull request.

Note: If you don't see the branch you've been working on in GitHub, that probably means it only exists locally. Make sure to commit and push the branch and then refresh GitHub.

Once we click on the *New Pull Request* button, we'll be taken to a screen that looks like this:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



The screenshot shows the GitHub interface for creating a pull request. At the top, there are two dropdown menus: 'base: master' and 'compare: new-feature'. To the right of these is a green checkmark and the text 'Able to merge. These branches can be automatically merged.' Below this is a section titled 'New feature' with a user profile picture on the left. Inside this section, there are two tabs: 'Write' and 'Preview'. The 'Write' tab is active, showing a text area with the text 'This new feature adds a line to the README.' Below the text area is a dashed line and the text 'Attach files by dragging & dropping, selecting or pasting them.' To the right of the text area is a green button labeled 'Create pull request'.

There are several important things to note about this screen.

First, at the top, we see two dropdowns. The one on the left reads *base: main*. The one on the right reads *compare: _new_feature*. Then, just to the right of that, we see a green message with a checkbox that says *Able to merge*.

Base is the branch we want to merge our new feature *into*. It will often be the main branch but not always. For instance, if we also had a development branch, we might want to target that branch with our PR instead.

Compare is the branch that includes the code we want to merge. In this case, it's the *new-feature* branch. However, we could also change it to be another branch instead. Why is it *compare* instead of *merge*? Well, just because we are creating a PR doesn't mean the code will eventually be merged. It is just a request. We are asking

someone else to review the code and compare it. Based on that, the reviewers will determine whether the PR should be merged, if it needs more changes, or if it should be rejected outright.

Next, we have an *Able to merge* message. This is important. This verifies that our PR, if it is approved, can be merged without any conflicts. We can still make a PR even if it will cause merge conflicts but in general we want to avoid that if possible. Merge conflicts are often a sign that developers haven't been communicating about the code they are working on, resulting in two sets of code that has conflicts. We will work through an example of a PR with merge conflicts after we are finished creating and merging this PR.

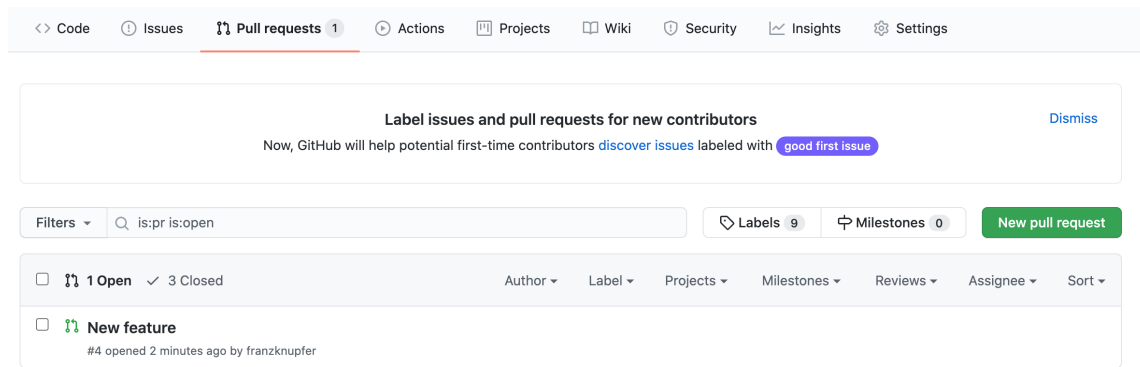
Next, we need to give our PR a title and add a comment. It's important to communicate clearly and concisely. While it's technically optional to add a comment to a PR, doing so (and having a clear title) makes it easier for reviewers to see what your PR is supposed to do. In an actual job, you should always strive to have good communication. Your senior devs will not be happy if they have to review poor PRs that create more work for them.

Once this is done, we can click on *Create pull request*. There are actually two options here. We can click on the dropdown attached to the button to have an option to *Create Draft Pull Request*. This simply means that the PR can't be merged until you mark it ready for review. Generally, we'll be making PRs only when the code is ready to merge. However, a draft pull request can be helpful because it allows others to see the new code — and comment on it — while it's still in progress.

Click on *Create pull request* and we're done making the request.

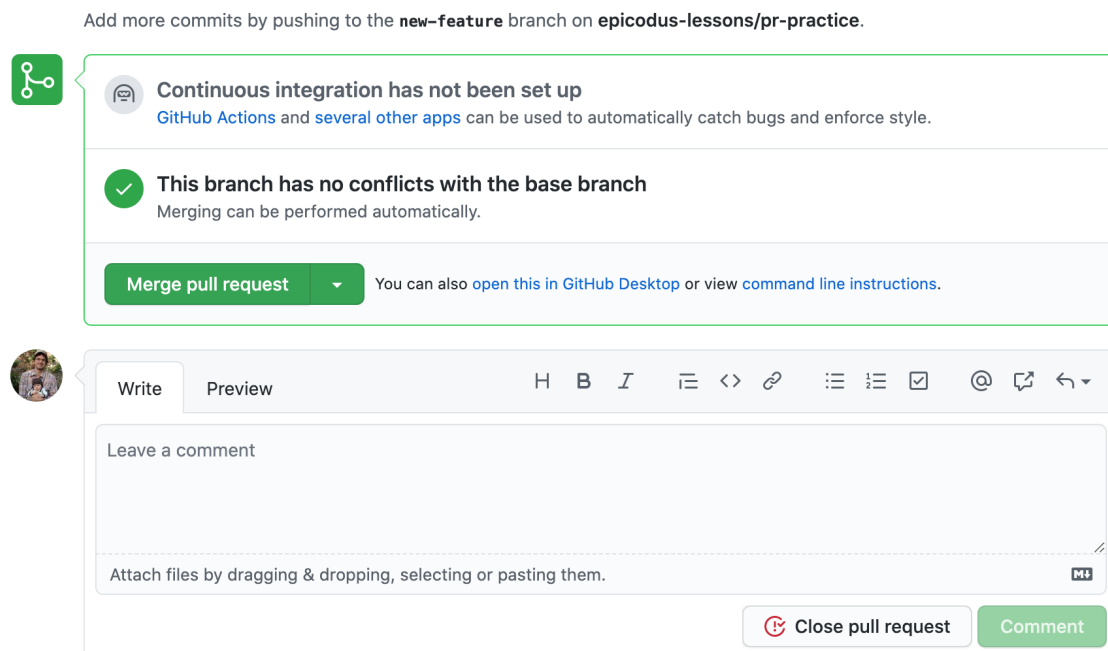
Approving a Pull Request

After we create the pull request, GitHub will display the page with the pull request. We can also navigate to our new PR, or any other PR in the repository, by clicking on the *Pull requests* tab in the repository.



As we can see in the image above, the *New feature* title isn't so great — it's fine for practice, but if we had many pull requests, the purpose of this one wouldn't be clear.

The image below shows the page for the PR itself:



There are several important things to note about the PR page:

- The PR will show a list of commits. If we need to, we can click on any of the commits to review the code. This is one thing a senior dev would do before approving the PR. (Note that the image above doesn't show the list of commits but you'll see it in your own UI.)

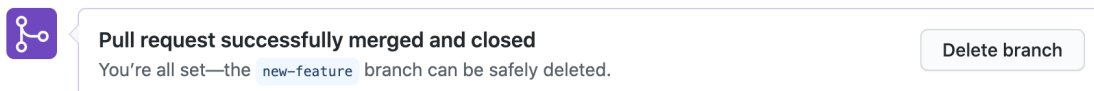
- As the image above shows, *This branch has no conflicts with the base branch*. We already know that. The good news here is that we can easily merge without running into conflicts.
- The green button gives us the option to *Merge pull request*. You can also click on the dropdown by this button to choose to *Squash and merge* or *Rebase and merge*. Generally, we will just click *Merge pull request*. The advantage of *Squash and merge* is that it will take all the commits from the PR and squash it down into one commit. This makes the commit history on our main branch tidier. For instance, if we added a navbar, we could *Squash and merge* to a single commit that reads `complete navbar feature`. Finally, *Rebase and merge* means that we can reorganize and rewrite all the commits. However, rebasing should be avoided unless it's necessary due to a poor commit history.

At the bottom of the page, we can leave a comment. There's also a *Close pull request* button (if you have privileges to close pull requests, which you will in your own account). We'd only click the *Close pull request* button if we want to reject it.

The comments are a great place to discuss the PR, including any changes or updates that should be made to it. Try adding a message for practice.

Now let's click on the *Merge pull request* button to actually merge the pull request. We can name this commit whatever we want and then click *Confirm merge*.

Once the branch is merged, we'll be given the option to delete it by clicking on the *Delete Branch* button.



Generally, once a feature is complete, the branch should be deleted. Then a new branch should be created for new features. For that reason, it's a good idea to delete the branch so it doesn't clutter the GitHub repository. The branch will still be available locally (unless it is deleted there, too).

If we were to continue working on that branch locally, we could still make a new PR with that branch later. That's actually what we'll do with our second PR — no need to create a new branch when we are focused on making and accepting pull requests in this lesson.

Our first PR is complete! We've learned how to make a PR and approve it.

Dealing with Merge Conflicts

Let's create one more PR. This time around, we'll intentionally introduce a merge conflict. That way, we can practice resolving a merge conflict in the GitHub UI. While we want to avoid merge conflicts if possible, they will happen from time to time, and you should be prepared when it does.

First, we need to introduce a merge conflict in our code. We'll start by making an update to the README in the GitHub UI. Click on the pencil icon by the README in GitHub and update the final line of the README to the following:

New feature updated...

Commit this code in GitHub.

Next, we'll make a change in the `new-feature` branch locally. We'll update the final line to this:

```
New feature changed...
```

It's a fairly similar line of text, but just different enough to create a merge conflict.

Save and commit this code. Next, push the branch to GitHub:

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

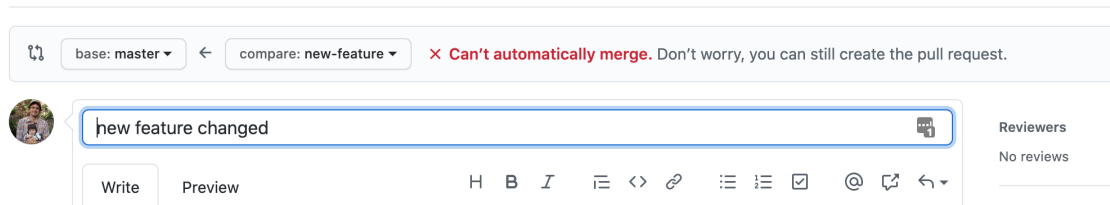
Our `new-feature` branch will be in direct conflict with the `main` branch — a great opportunity to practice resolving a conflict!

Go to the *Branches* tab of the repository in GitHub and click on the *New Pull Request* button to the right of the `new-feature` branch. (In general, all of the steps involved in creating a PR will be the same as they were for our first PR.)

Now, when we're taken to the page to make a PR, we'll see a message that states we can't automatically merge.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

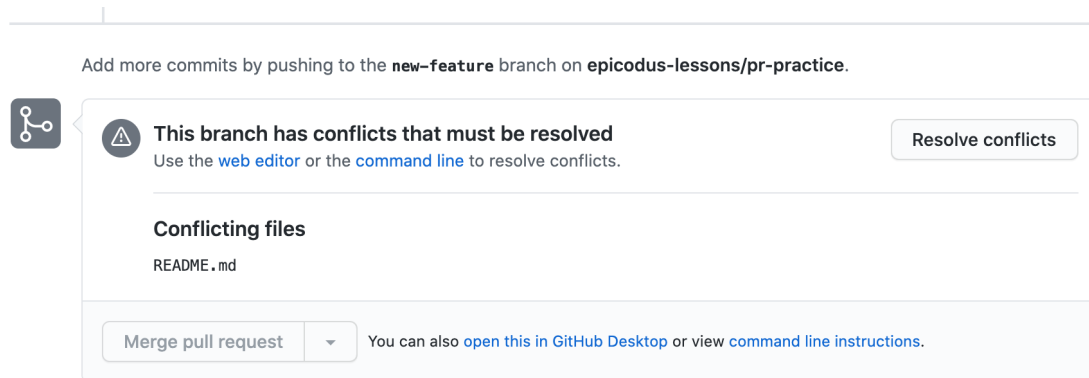


The screenshot shows the GitHub 'Open a pull request' page. At the top, there's a header with 'base: master' and 'compare: new-feature'. A red warning message states: 'Can't automatically merge. Don't worry, you can still create the pull request.' Below this, there's a text input field containing 'new feature changed'. To the right of the input field is a 'Reviewers' section showing 'No reviews'. At the bottom, there are tabs for 'Write' and 'Preview', and a rich text editor toolbar with various formatting options like bold, italic, link, and list.

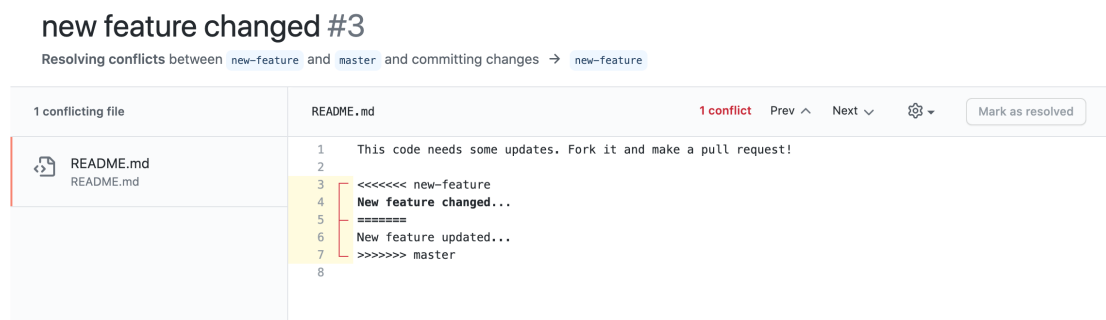
In our own projects, we should definitely heed this message if we weren't expecting a merge conflict. Generally, this means that two different teams have been working on the same code in different ways, which is a likely sign of a communication breakdown.

Go ahead and submit the PR following the same steps as stated for our first PR. The only difference in this entire process is the message that states that we can't automatically merge the PR. If you need assistance, review the steps from when we created our first PR in the previous section of this lesson.

Now we're ready to merge our PR. If you haven't already, navigate to the PR in GitHub. Instead of the green *Merge pull request* button, we'll see a new message:



The message reads *This branch has conflicts that must be resolved*. We can click the button to the right of this message to *Resolve conflicts*. Click that button now. The GitHub UI will allow us to resolve the merge conflicts.



In this example, there is only one file with a merge conflict. However, if there are multiple files that have merge conflicts, we'll see the name of each file in the left-hand pane. At the top right hand corner of the pane, we'll see a greyed-out *Mark as resolved*

button. We cannot click this button until we remove all lines that indicate where the conflicts are. These lines will always include `=====`, `<<<<<<`, and `>>>>>>`.

To the left of the *Mark as resolved* button, we'll see a message in red that says *1 conflict* — and then just to the right of that we'll see we can click *Prev* and *Next*. This is so we can navigate between files with conflicts. We can't do that here because we only have one conflict to resolve.

Fixing the issue itself is a bit more involved than doing so in VS Code. In VS Code, we can click whether we want the current, incoming, or both changes to take effect. In the GitHub UI, we have to modify the code manually.

The code below `<<<<<<` is the incoming change from the `new-feature` branch. The code below `>>>>>>` is the current change from the `main` branch. If the arrows mix you up, just focus on the names of the branches instead and keep in mind that the code in the branch you choose will take precedence.

This is how the code currently looks with GitHub's merge messages inserted:

```
<<<<<< new-feature
New feature changed...
=====
New feature updated...
>>>>>>
```

Assuming that we want to accept the code from the `new-feature` (since that's the point of the PR), we'd remove `<<<<<< new-feature` and then also remove:

```

=====
New feature updated...
>>>>>>

```

That would leave just:

```

New feature changed...

```

Alternatively, we can modify the code in other ways as well as long as the GH merge conflict messages are removed. Once again, those are all the lines with `=====`, `<<<<<<<`, and `>>>>>>>`.

When the merge conflict lines are removed, the *Mark as resolved* button will no longer be greyed-out and we can click it. This is a nice little feature to ensure we don't accidentally leave any of those lines in our code.

Once our merge conflict is marked as resolved, we'll see some changes in the UI.

new feature changed #3

Resolving conflicts between `new-feature` and `master` and committing changes → `new-feature` [Commit merge](#)

This merge commit will be associated with franzknpfer@hotmail.com.

1 conflicting file	README.md	✓ Resolved
<div>README.md</div> <div>README.md</div>	<div>1 This code needs some updates. Fork it and make a pull request!</div> <div>2</div> <div>3 New feature changed...</div> <div>4</div> <div>5</div>	

Most importantly, we can now click *Commit merge*. When we click on this button, we'll be able to make a new commit (as always, with a clear, concise commit message) that commits the changes we just made to resolve the conflict. At this point, we've just made a new

commit, but we still haven't merged the PR yet! Now that the conflict is resolved, we'll see the green *Merge pull request* button. We can click that button and finish merging the PR.

In this lesson, we made and merged two simple pull requests, including one with a merge conflict. We recommend practicing this workflow as you work on your group project — and give your project teammates the chance to look at your pull requests, make comments, and then approve them!

[Previous \(/intermediate-javascript/team-week/journal-7-discussion\)](#)

[Next \(/intermediate-javascript/team-week/pull-requests-with-forks\)](#)

Lesson 11 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Tuesday

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Pull Requests with Forks

Text

Many of the JavaScript libraries and tools we use at Epicodus are **open source** including webpack, React, and Jest, to name a few. **Open source** means the code is available publicly and that anyone can contribute to it.

In order to do that, maintainers of open source projects welcome pull requests from the community. However, it wouldn't be a good idea for maintainers to make everyone in the community collaborators. That would be difficult to manage and also lead to potential security risks.

Instead, it's possible to make a PR with only read access to a repository. Instead of cloning the repository and making a branch, contributors **fork** the project, make changes to a branch on the fork, and then submit the PR.

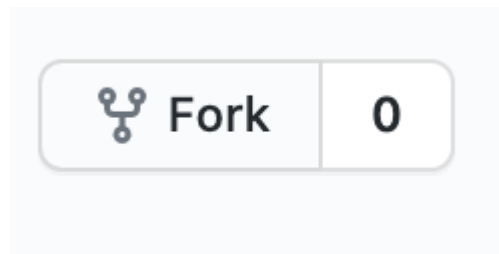
In this lesson, we'll walk through the necessary steps to make a PR via a fork. The steps for approving a PR are the same regardless of whether the PR comes from a branch or a branch on a fork so we won't cover that part.

Once you know how to submit a PR via a fork, you can start contributing to open source projects! There are many, many ways to do so — and you don't need to be an expert coder to contribute. We'll talk about contributing to open source more at the end of this lesson.

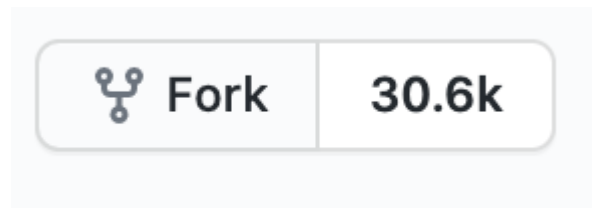
Making a PR Request Via a Fork

We covered the basics of forking in Practice: GitHub Remote Repositories (<https://www.learnhowtoprogram.com/introduction-to-programming/git-html-and-css/practice-github-remote-repositories>).

To fork a repository, we just need to click the *Fork* button in the upper right corner of the repository we want to fork.



There's even a little number by the button that shows how many times the repository has been forked. The above repository doesn't have any forks — but check out the *Fork* button for React:



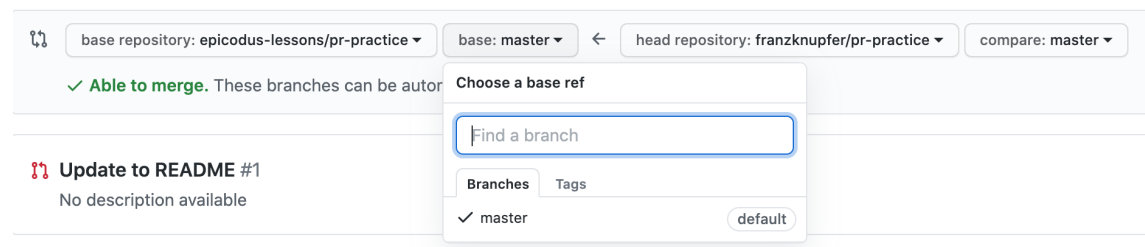
This one has nearly 31,000 forks at the time this screenshot was taken.

When you click the *Fork* button, you'll be prompted to choose a GitHub account to fork the project to. Once you choose an account (you likely only have one at this point), GitHub will automatically create the fork.

Once the project is forked, we can clone that project to our desktop, make a branch, and then make any updates to the branch as needed. *Do not make changes directly to the main branch.* We are really following almost the exact same process that we follow when making a PR without a fork.

Once the branch is updated, committed, and pushed back to our fork of the project, we can make a PR.

To do that, we can open a PR *in our fork of the project*. We don't even need to navigate to the original repository.



The image above shows the process of making a PR from a forked repository. GitHub automatically points at the original repository for the base repository while the compare repository is automatically the fork. Note that we should always verify that we are pointing at the correct repositories; we also need to make sure we are pointing at the right branches, too. In the example above, we can see that the compare branch is still on main — not what we want if we've just made changes to a branch on the fork! So even though GitHub should automatically point to the correct repositories, we still need to verify they are correct and update the branches as needed.

After that point, every other part of the PR process is the same as when we make a PR via a branch on the original repository itself.

Contributing to Open Source Projects

Open source projects are a huge part of the developer ecosystem. As we mentioned at the beginning of this lesson, many widely-used libraries are open source. By contributing to open source projects, we can get more involved in the developer community, improve our favorite libraries, and add great experience to our resume.

At this point, you might be thinking that you don't have enough experience to make an open source contribution. However, that is not the case. You don't need to be an expert coder to make a contribution. In fact, many open source projects have a pool of good first issues for people new to making contributions. Open source maintainers may need help on everything from updating READMEs to writing tests to updating the code itself. Updating a README for a library can be a great way to get to know the library better and get involved in an open source project.

We recommend taking the time to make a contribution. For more information on making open source contributions, see [How to Contribute to Open Source](http://opensource.guide/how-to-contribute/) (<http://opensource.guide/how-to-contribute/>), an excellent tutorial from Open Source Guides on making contributions.

[Previous \(/intermediate-javascript/team-week/pull-requests-with-branches\)](#)

[Next \(/intermediate-javascript/team-week/team-week-presentations-and-independent-project\)](#)

Lesson 12 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.

Lesson

Thursday

Intermediate JavaScript (/intermediate-javascript)

/ Team Week (/intermediate-javascript/team-week)

/ Team Week Presentations and Independent Project

Text

Trade Show Presentations

Today, teams will give an informal presentation and demonstration of their project to staff and students at their work stations. Determine how you want to present and demonstrate your work. Each team member may speak or a spokesperson may be designated for the group.

Your instructor will organize the format, length, and requirements for presentations. Plan to spend about 5 minutes covering the following information:

- **Team Members** — Who worked on the project?
- **Project Name and Objective** — Why did you choose this project? What purpose does it fulfill?
- **Demonstration** — Show and describe the features of your work.

- **Process** — Describe how the team development process worked in your group.
- **Challenges** — Share the biggest challenges you faced.

Also, allow a bit of time for any questions your audience may have.

Team Week Code Review

Group projects count as one of your required code reviews, and must be submitted through Epicenter. Remember, successful completion of each code review is required to pass the course. Additionally, transcripts depicting your performance on all code reviews will be sent to internship companies.

Your presentation at the Trade Show will take the place of a one-on-one code review with your instructor. You will receive feedback directly through Epicenter.

Objectives

Your code will be reviewed for the following objective:

- Participation in creating and presenting a project; including collaborating effectively with teammates.

Submission

Each team member should have a copy of the project repo on their own GitHub account. Then, each team member needs to submit the link to the project's GitHub repository to the **Team Week** code review on Epicenter (<https://epicenter.epicodus.com/>) by the deadline:

- Friday at 5 pm for full-time students.
- Sunday at 8 am for part-time students.

It's easy to forget to submit by the deadline. Because of this, we suggest submitting your code review **before the end of class** after the trade show is finished.

Visit Independent Projects and Code Reviews

(<https://www.learnhowtoprogram.com/introduction-to-programming/getting-started-at-epicodus/independent-projects-and-code-reviews>) lesson for details on how to submit your code, how feedback works and course completion requirements.

[Previous \(/intermediate-javascript/team-week/pull-requests-with-forks\)](#)

Lesson 13 of 13

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.