

# A Team-Based Workflow for using Git and GitHub

- We've already explored Git and GitHub from an individual's perspective, covering the most common workflow for working on a Git/GitHub-based project by yourself.
- However, one of the major benefits of Git and GitHub is that they can be used by an entire team to make it easier to collaborate on a shared code project.
- Working with a team on a shared code project using Git and GitHub takes some care, though, because each developer needs to be aware of the work being done by each other developer and to make sure their own work is compatible with everyone else's.
- Without coordination, individual developers can accidentally undo someone else's work or otherwise break the shared code because they introduced a change that wasn't compatible with the rest of the codebase. For this reason, teams working on a shared Git/GitHub project usually adopt a special workflow that helps team members collaborate more effectively.
- An ideal team-based Git/GitHub workflow allows team members to stay aware of what their teammates are working on while at the same time ensuring that the code each developer contributes works with the rest of the codebase.
- Here, we'll explore one of the most popular Git/GitHub workflows for teams, known as the [GitHub flow](#) (also sometimes called the [feature branch workflow](#)).
  - Note that there are many other Git/GitHub workflows. Another popular one is the [Gitflow workflow](#), though Gitflow is losing popularity due to its relative complexity in comparison to the GitHub flow.

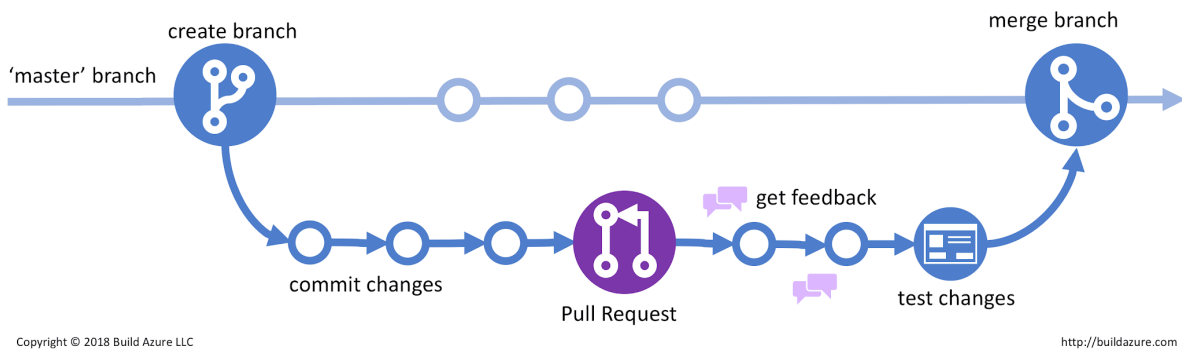
## The GitHub flow

- The GitHub flow is a multi-developer workflow that is based on **Git branches** and **GitHub pull requests**. We'll cover both of these tools in detail here. Combined, the use of branches and pull requests means that the GitHub flow does the following things:

- It isolates each developer's work from the work of other developers, protecting against conflicts between different developers' code.
- It ensures that all developers on the team are aware of what other developers are working on and, through a procedure known as **code review**, gives the team a chance to test and approve each developer's work before it is accepted into the "production" version of the code.

- Graphically, the GitHub flow looks like this:

### GitHub Flow

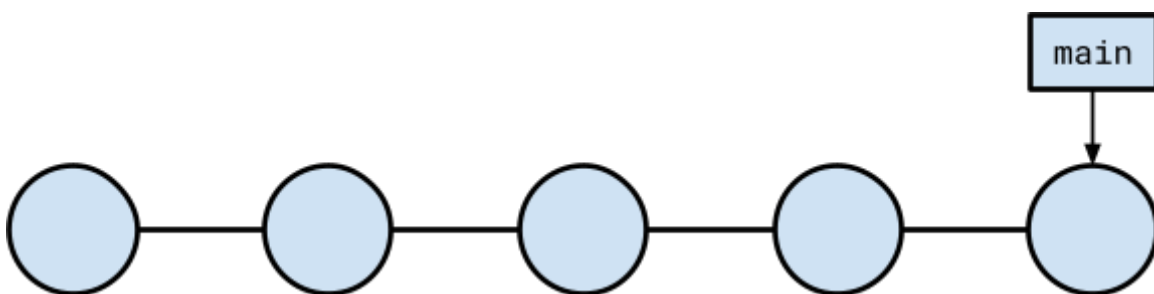


- Here is a high-level overview of how the GitHub flow works:
  - In the GitHub flow, the **main** branch of a repo represents the production version of the code. In particular whatever code lives in the **main** branch should be deployed.
  - To work on anything new (a new feature, a bug fix, anything), a new Git branch should be created off the **main** branch.
  - All development should be done in this new branch. Commits should be made locally to this branch and frequently pushed to a branch with the same name on GitHub.
  - When the work in the new branch is ready for production, or when you need feedback or help, a pull request is opened to merge the new branch into **main**.
  - Discussion about the new code takes place in the pull request. The developer who made the pull request should respond to feedback here and make any requested changes to their code (or argue why those changes shouldn't be made).
  - All changes in the pull request should be tested, both through an automated test suite and, if possible, through manual testing by members of the team.

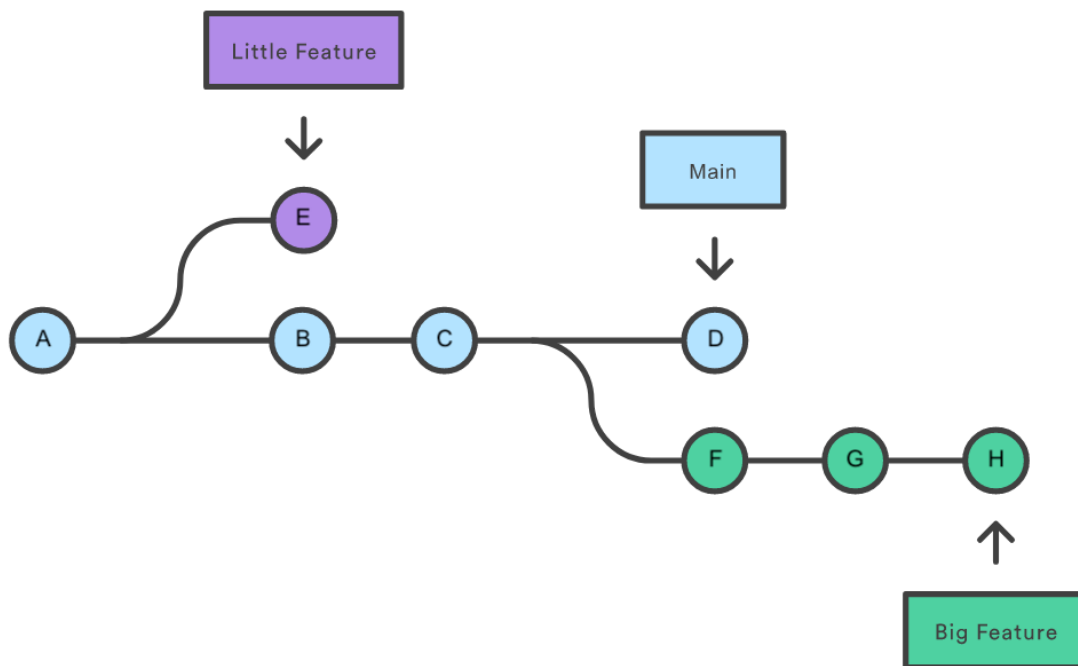
- After the right number of teammates have reviewed the new code and approved the pull request, the new branch can be merged into the **main** branch.
- Once the new branch is merged into **main**, the new code should be immediately deployed to production.
- One of the most important corollaries of the GitHub flow is that **no code should ever be committed directly to the **main** branch**. Instead all new work is done in separate **feature branches**.
- Don't worry if you don't totally understand how the GitHub flow works just based on the description above. We'll break it down into more detail in what follows.
- Let's start by exploring what Git branches are in more detail.

## Git branches

- To understand what a Git branch is, remember that, at its core, Git's main purpose is to store a series of snapshots of a project over time. This series of snapshots, or **commits**, is called the **commit history** for the project.
- Since we haven't yet dealt with creating new branches, up until now the commit histories of our projects have each been just a straight line, with one commit after the other in a straight series within the project's **main** branch:



- Introducing new branches changes this picture. Specifically, creating a new branch in our repo allows us to create a commit history with divergent, parallel, well... branches, each with its own commits, which don't exist in other branches:



Adapted from [Atlassian](#)

- Each new Git branch must be created off of an existing branch. Specifically, each new branch must start with an existing commit (in an existing branch). Up to and including that existing commit, the original branch and the new branch will share a commit history, but after that commit, the histories of the two branches will diverge.
- For example, in the image above, the “Big Feature” branch is created off the “Main” branch at commit C. Thus, “Big Feature” and “Main” share the commits A, B, and C in their history. However, after commit C, the two branches have different histories.
- Because a branch has its own commit history, at least back to the common commit it shares with another branch, it represents an isolated environment in which a developer can work without impacting other branches in a repository. In other words, a branch can allow you to work on a new feature, fix a bug, or experiment with new ideas in a contained area of the repository, isolated from the work other developers are doing in other branches.
- This means if each developer is working in a separate branch, no developer’s work will impact the work of another developer until the branches are **merged**

back together (more on that later).

- Now that we understand a little bit about what branches are and why we might want to use them, let's start to explore how to work with branches in a Git repository.

## Working with Git branches

- Let's imagine we're working in an existing repository with a single branch called `main` that already has a few commits:

```
$ git log --oneline
54575a9 (HEAD -> main, origin/main) Add document title.
8436964 Add "Hello, world!" header.
46c83e2 Create index.html. Add HTML skeleton.
```

- We can see a list of all the branches in our project by running the command `git branch` (the current branch will be marked with a `*` in the output):

```
$ git branch
* main
```

- Let's say that the project we're working on currently just contains a single file, `index.html`, with the following contents:

```
<html>
  <head>
    <title>World greeter</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

- Imagine we've been tasked with adding some CSS to this app. This involves new development work, which we'll need to do in a new branch off of the main branch.

- To create a new branch, we need to start by making sure the current branch is the one we want to branch from (i.e. the `main` branch). Then, we can use the `git branch` command with the name of the new branch we want to create. New branch names should be descriptive, capturing the work that will be done in that branch, e.g.:

```
git branch begin-css
```

- After running that command, if we run `git branch` again, we'll see the new branch listed:

```
$ git branch
  begin-css
* main
```

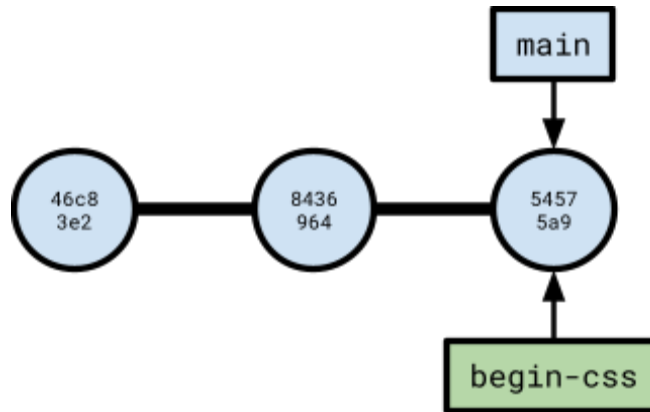
- Note that in this output, the `main` branch is still selected as the current branch. This means any new commits we make now will still be added to the `main` branch. If we want to start making commits to the `begin-css` branch, we need to select it with the `git checkout` command:

```
git checkout begin-css
```

- Note that as an alternative to the two-step “create then checkout” sequence here, Git provides a shorthand command we can use to create and checkout a new branch, all in one command. Specifically, starting from the `main` branch, we could have run the following command to create and checkout the `begin-css` branch:

```
git checkout -b begin-css
```

- At the moment, our repository's two branches point to the same commit, the last commit in `main`:



- Let's start the work on our new feature by creating the CSS file (`style.css`) and linking it to the HTML:

```
<html>
  <head>
    <title>World greeter</title>
    <link rel="stylesheet" href="style.css">
  </head>
  ...
</html>
```

- Before writing any CSS code in `style.css`, let's make a commit that includes our changes:

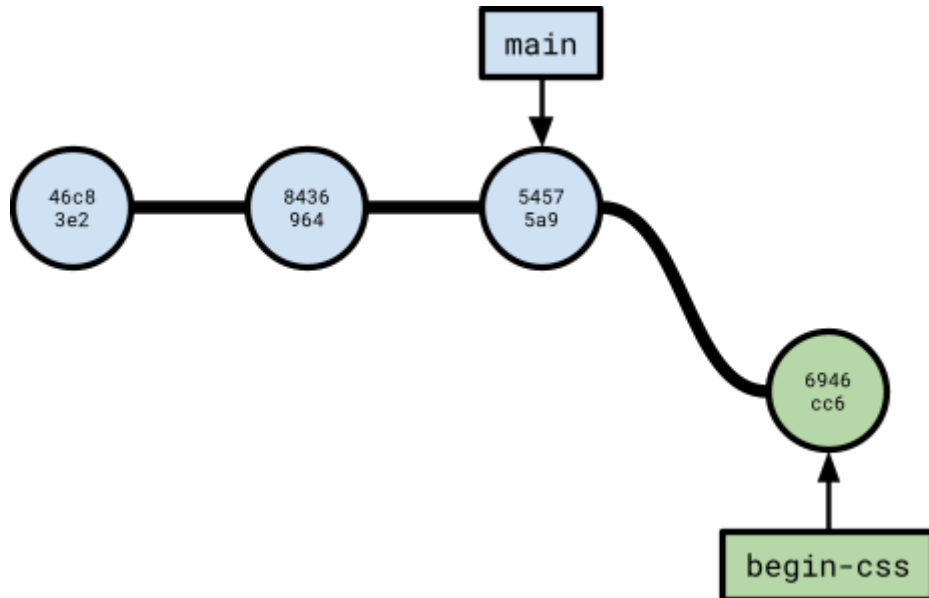
```
git add style.css index.html
git commit -m "Create style.css and link it to HTML."
```

- Now, if we look at the commit history for our branch, we'll see that the histories of the `main` branch and the `begin-css` branch will have diverged (as indicated by the locations of the branch names in parentheses):

```
$ git log --oneline
6946cc6 (HEAD -> begin-css) Create style.css and link it to HTML.
54575a9 (origin/main, main) Add document title.
8436964 Add "Hello, world!" header.
```

46c83e2 Create index.html. Add HTML skeleton.

- Visually, our repo's commit history looks like this right now:



- The `origin/main` that appears in the commit log above indicates that we have a remote repository (e.g. on GitHub) tied to our local repo and that the `main` branch is synchronized there. Remember that we can see information about the remote repository using the `git remote` command:

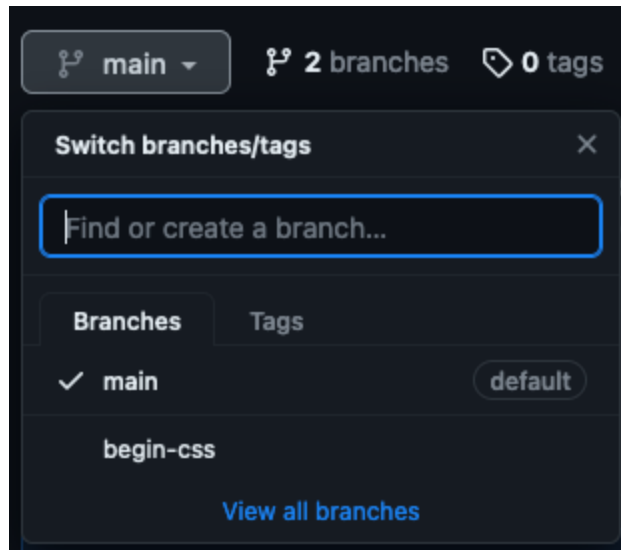
```
$ git remote -v
origin    git@github.com:... (fetch)
origin    git@github.com:... (push)
```

- We want to make sure our `begin-css` branch is also synchronized to the remote repo on GitHub. We can do this by running the following `git push` command:

```
git push --set-upstream origin begin-css
```

- If we visit our repo on GitHub, we should now be able to see the `begin-css` branch listed there in the branches dropdown:





- We could select `begin-css` in that dropdown to browse the contents of the branch, view its commit history, etc., just like we've previously done with the `main` branch on GitHub.
- For good measure, let's add some CSS to `style.css`:

```
html {  
  font-family: sans-serif;  
}  
  
h1 {  
  font-variant: small-caps;  
}
```

- Then, let's commit our changes and push the new commit to GitHub:

```
git add style.css  
git commit -m "Do some work on font styling."  
git push
```

- Let's imagine that as we're working on our CSS feature, another developer has been tasked with adding some client-side JS to the app. Imagine this developer is starting off with just the `main` branch, containing the same three commits we started with here:

```
$ git log --oneline
54575a9 (HEAD -> main, origin/main) Add document title.
8436964 Add "Hello, world!" header.
46c83e2 Create index.html. Add HTML skeleton.
$ git branch
* main
```

- To begin their feature, the developer would create and checkout a new branch, giving it a descriptive name:

```
git checkout -b handle-title-click
```

- Within this branch, let's say the other developer creates a file `index.js` with the following code:

```
const title = document.getElementById("title")
title.addEventListener("click", function () {
  alert("You clicked the title!")
})
```

- And, let's say they also make the following changes to the HTML:

```
<html>
  <head>
    <title>World greeter</title>
    <script src="index.js"></script>
  </head>
  <body>
    <h1 id="title">Hello, world!</h1>
  </body>
</html>
```

- Finally, let's say the other developer commits their changes and pushes their new branch to GitHub:

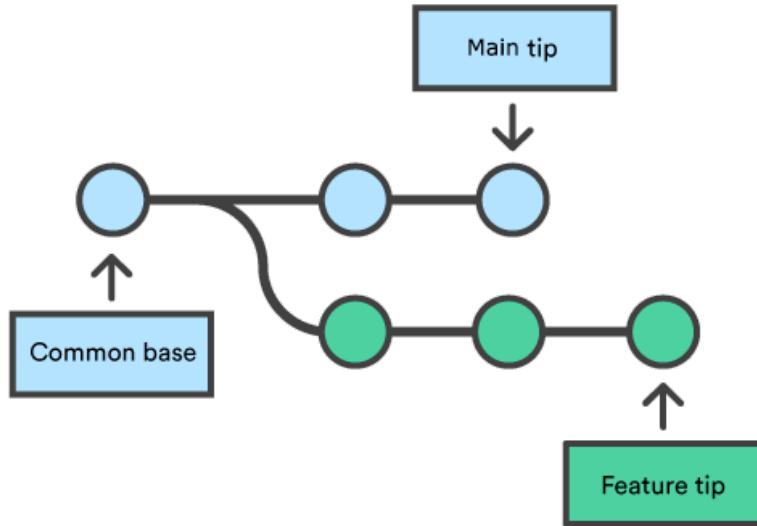
```
git add index.html index.js
```

```
git commit -m "Add title click handler."
git push --set-upstream origin handle-title-click
```

- We're feeling pretty pleased with our own feature at the moment, and the other developer probably is, too. In fact, we each think our respective feature is ready to be merged into `main` to be deployed as production code.
- According to the GitHub flow, when we think our work in a new branch is ready to be deployed to production, our next step is to create a pull request. We'll cover how to do that in a minute, but first we have to understand what "merging" is in Git.

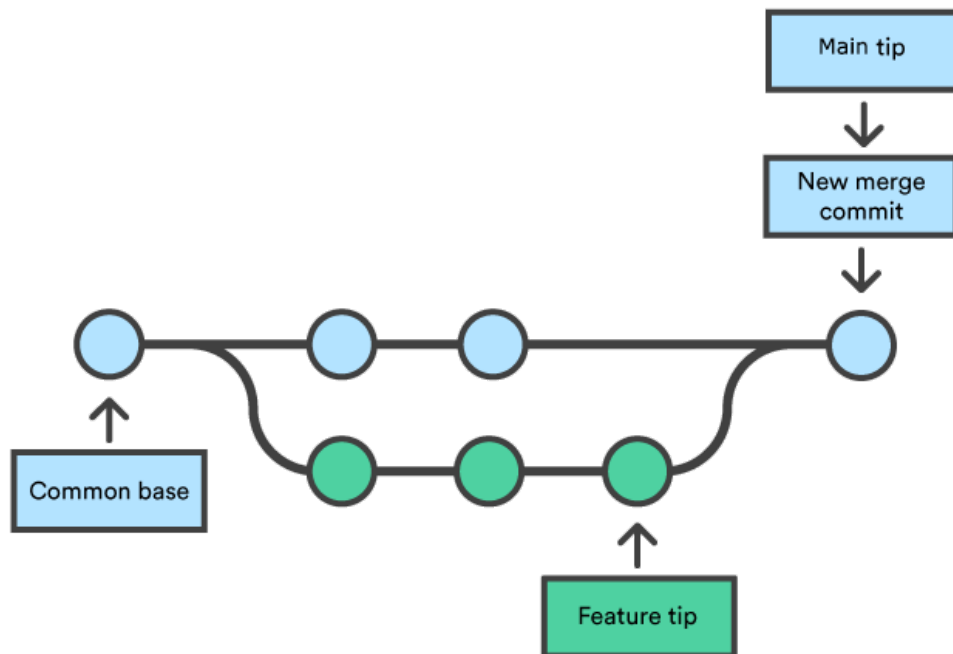
## Merging branches in Git

- Every pull request starts with changes that live in a branch in a GitHub repo. At its core, a pull request is simply a request to **merge** that branch into another branch. Most often (though not always), pull requests are used to merge features implemented in other branches into the `main` branch.
- In Git, the term "merge" has a specific meaning, referring to the `git merge` operation, which is used to combine multiple separate sequences of commits into a single, unified commit history. Usually, `git merge` is used to combine two branches, like in a pull request.
- When `git merge` is used to merge two branches, changes from the **target branch** are always merged *into the current branch* (i.e. into the branch listed as selected by the `git branch` command).
- Unless the conditions are met for a special type of merge known as a **fast-forward merge**, when Git performs a merge operation it always creates a special kind of new commit in the current branch known as a **merge commit**. A merge commit simply represents the changes from the target branch that are being merged into the current branch.
- For example, imagine we have a new feature branch that we want to merge into the `main` branch of our repo:



Courtesy of [Atlassian](#)

- After merging the feature branch into `main`, a new merge commit will be added to `main` that incorporates the changes from the feature branch:

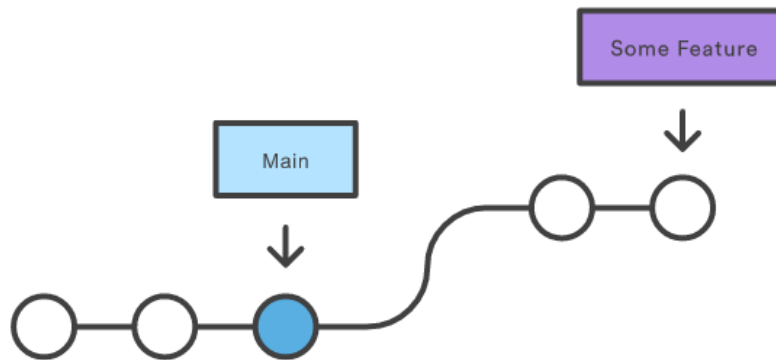


Adapted from [Atlassian](#)

- After the merge, the `main` branch will contain all of the code that was implemented in the feature branch.

## Fast-forward merges

- As we mentioned above, there is a special kind of merge known as a **fast-forward merge**, and during a fast-forward merge, Git does not create a merge commit.
- A fast-forward merge can only be performed under specific circumstances. Specifically, a fast-forward merge can only take place when there is a direct linear path in the commit history between the tip of the current branch and the tip of the target branch, i.e. when the tip of the current branch is a direct ancestor of the tip of the target branch in the commit history, e.g.:



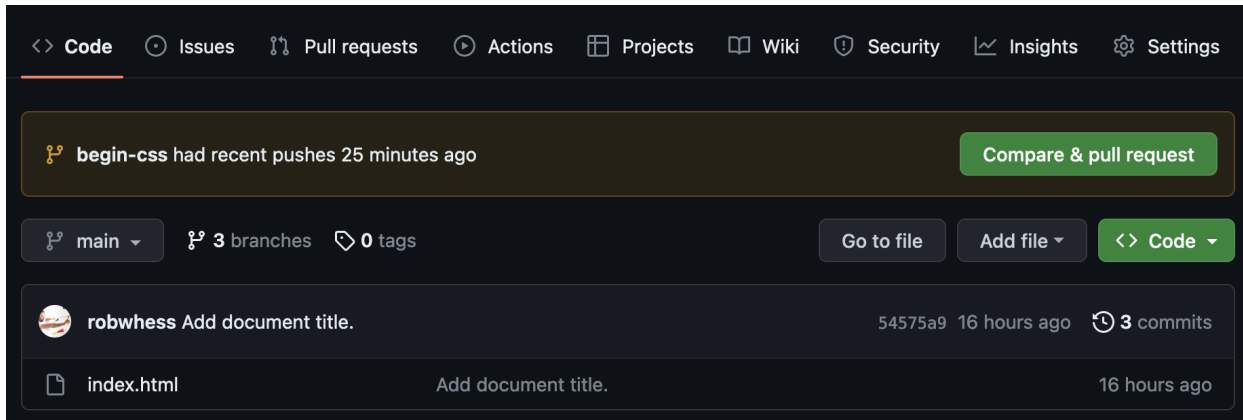
Adapted from [Atlassian](#)

- In this case, Git can merge the target branch (the feature branch in the picture above) into the current branch (the **main** branch in the picture above) by just moving (i.e. “fast forwarding”) the tip of the current up to the tip of the target branch, i.e.:

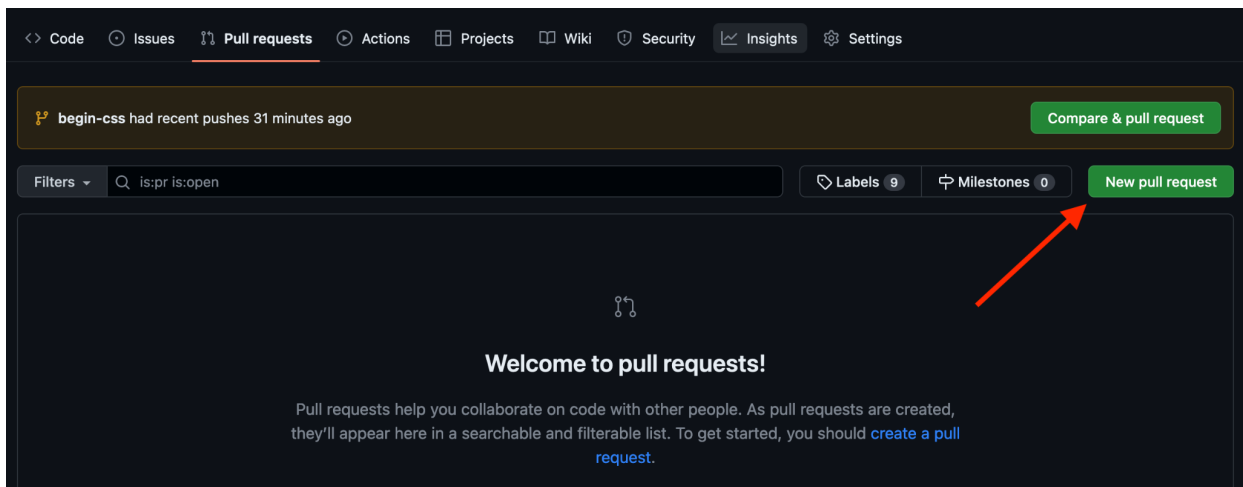


- ## Creating a pull request

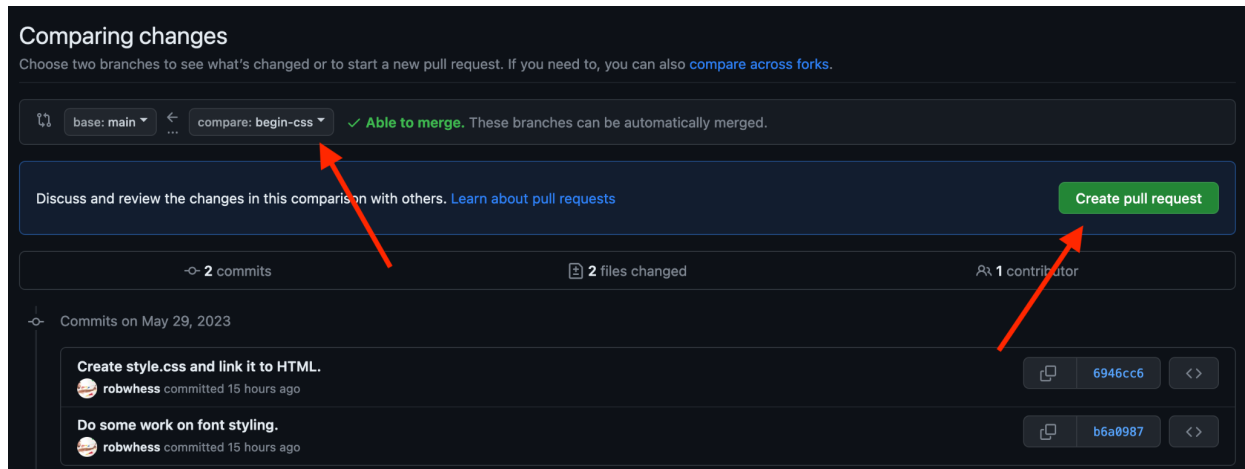
- Pull requests are a GitHub feature that allow a developer to notify collaborators about changes they want to incorporate into the codebase. A pull request allows collaborators to see what changes are being proposed and to discuss those changes, provide feedback about them, request modifications, and ultimately to approve the changes (or reject them) and merge them into the codebase.
- A pull request is specifically a request to merge one branch into another branch (usually a feature branch into the `main` branch, though not always). Thus, in order to create a pull request, both branches must live on GitHub.
- Since we made sure our `begin-css` branch is on GitHub and we want to merge it into `main`, which is also already on GitHub, we're all set up to create a pull request for our work in `begin-css`.
- Though there are other ways to do so, we'll create a pull request through GitHub's web interface. We'll start by navigating to our repo on GitHub. If you've pushed commits to a non-`main` branch recently, GitHub will actually present you with a banner asking if you want to create a pull request from that branch:



- Just in case you didn't get to GitHub fast enough to catch that banner, you can also navigate to the “Pull requests” tab and click the “New pull request” button there to create a pull request:



- In the page that opens, we'll use the “compare” dropdown to pick our **begin-css** branch. We'll leave the “base” branch as **main**. Once we select **begin-css** as the “compare” branch, we'll see the screen change to present us with a list of commits from **begin-css** that will be merged into **main** when the pull request is accepted, along with a diff showing the differences in the code between **main** and **begin-css**. We can inspect these, and if they look good, we can click the “Create pull request” button:



- After clicking the “Create pull request” button, we’ll be brought to a screen where we can type in a title and description of the pull request. These should be descriptive, capturing the purpose of the pull request and summarizing the changes it makes as well as the problem they solve, e.g.:

*Start styling the application*

*The app didn't have any styling before. This PR begins to add some styles, focusing mainly on font styling.*

- Once typing the title and description of the pull request, we can click the (second) “Create pull request” button.
  - Note that we *don't* want to create a “draft” pull request here. [Draft pull requests](#) are intended for branches that are not yet ready to be merged.
- After the pull request is created, you will be brought to the pull request’s “conversation” tab. This tab summarizes the pull request, lists its commits, and provides a forum for general discussion about the pull request.
- There are other tabs associated with the pull request as well, such as the “files changed” tab, which displays a line-by-line diff of all of the actual code changes made by the pull request. This is where other developers can perform **code review** on our pull request. Let’s see what this looks like.

## Performing code review on a pull request in GitHub

- One of the most important features of a pull request is that it gives other developers a chance to review changes before they are accepted into the main

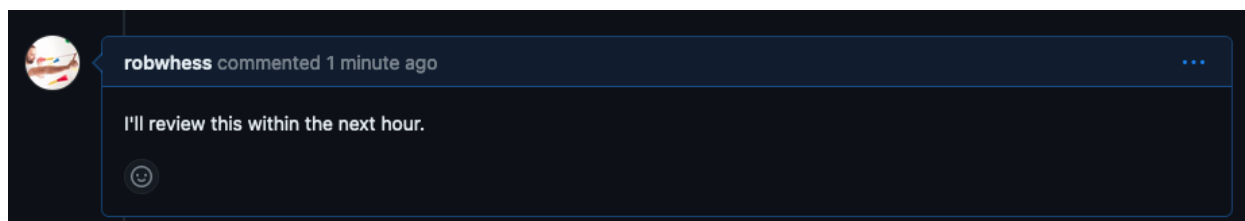


branch.

- Performing code review is an important part of the development process for many teams because it helps to improve code quality and to distribute knowledge about the code itself across the team.
- It's important for reviewers to do a thorough job of reviewing the changes proposed in a pull request, since the pull request is the last hurdle those changes have to make it through to be deployed into production.
- To perform high-quality reviews, it helps to have a good understanding of how the code review tool itself works, so let's explore some of the code review features GitHub offers.
- Here, let's assume the other developer we're working with also created a pull request for their `handle-title-click` branch. We'll perform code review on their pull request.

## General discussion on the “conversation” tab

- If we have general comments, questions, proposals, etc. about a given pull request, we can submit those on the “conversation” tab for that pull request. This tab provides a general discussion forum for the pull request. Comments are entered in the box at the bottom of the “conversation” tab:

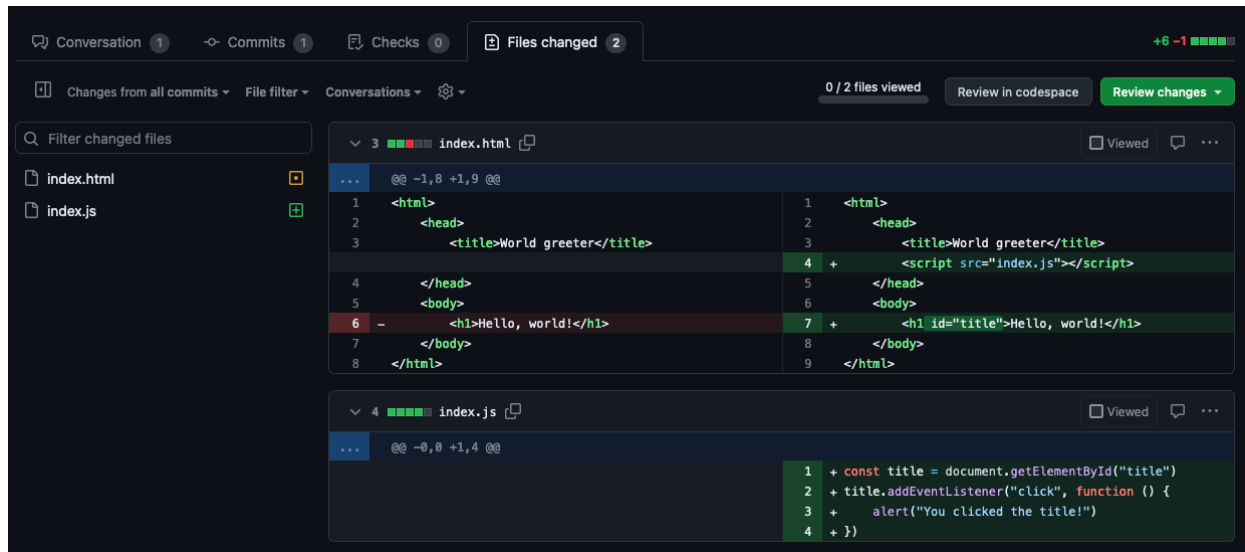


- Some high-level code review, including approval of the pull request, is sometimes appropriate in the “conversation” tab, but a better place for more detailed code review is under the “files changed” tab.

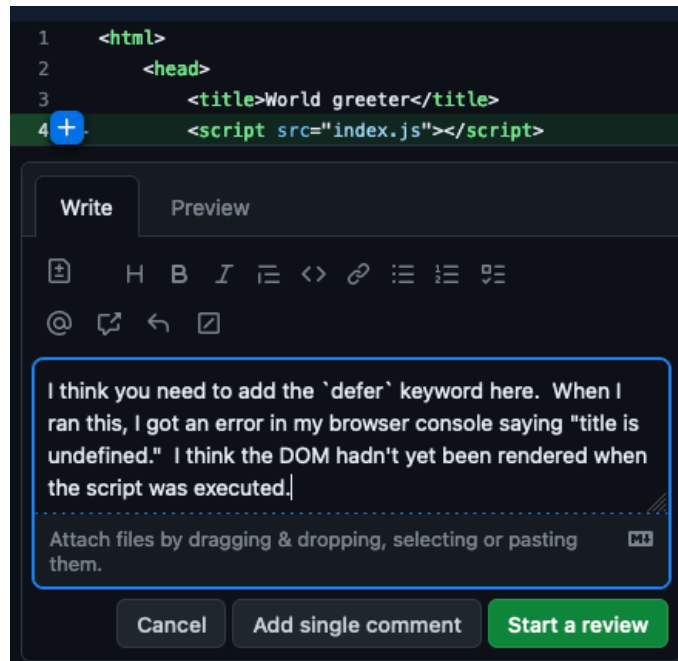
## Performing code review on the “files changed” tab

- A pull request's “files changed” tab is the right location to perform more fine-grained code review, and it is set up for this purpose.

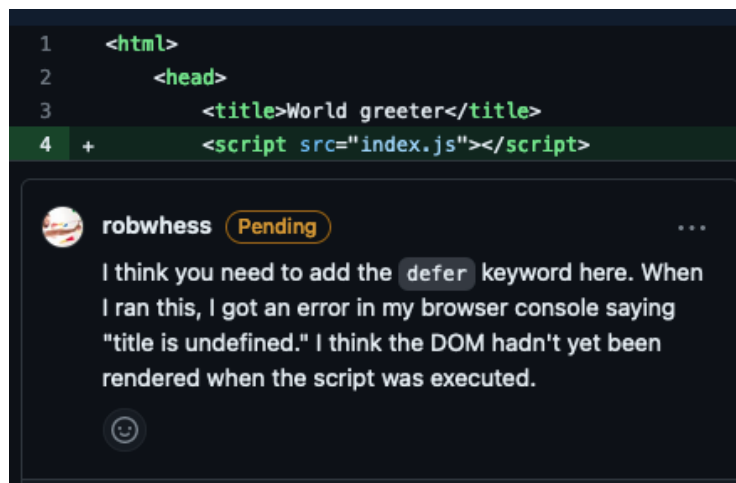
- The “files changed” tab displays a line-by-line diff representing the changes made by a pull request and provides some tools for opening and performing a code review:



- On this tab, we can review the pull request a file at a time. We can leave comments or request changes on a file-by-file basis, or we can make comments and requests on specific lines of code.
- For example, let's say we ran the code in the branch associated with the pull request and identified a potential bug. If we knew which line(s) of code that bug originated from, we can hover over that line and click the blue “+” button that appears to leave a comment on that line itself (or we can drag to select multiple lines on which to comment):

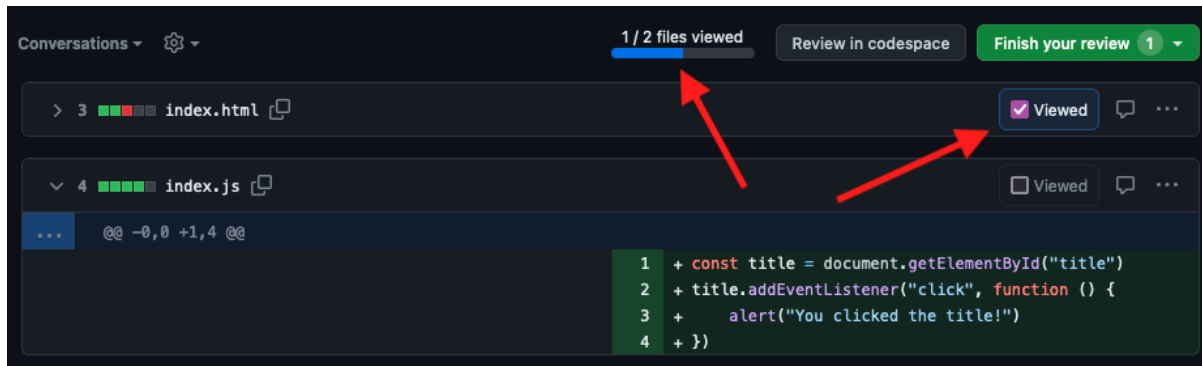


- After typing a comment, we can either click the “Add single comment” button or the “Start a review” button.
- If we want to start an official review that should be resolved before the pull request is merged, we should click the “Start a review” button. This will add our comment and mark it as “pending” indicating that it is part of a larger review that hasn’t been submitted yet:

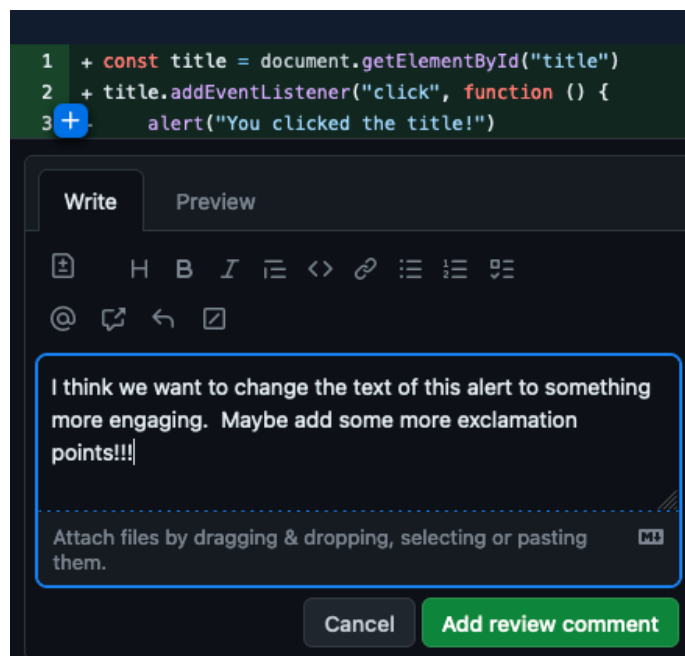


- Once we are done reviewing a specific file, we can mark that file as “viewed”, which will collapse the diff for that file and update our progress on the review.

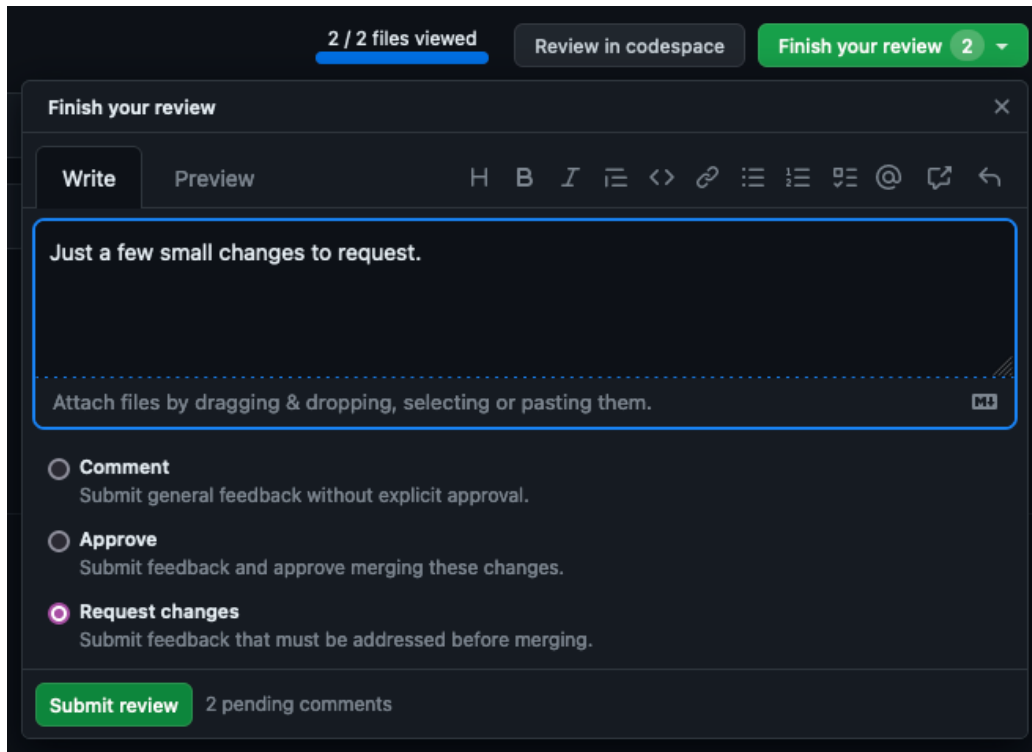
This doesn't have an effect on the review itself but can be a useful way to help us remember what we've already reviewed:



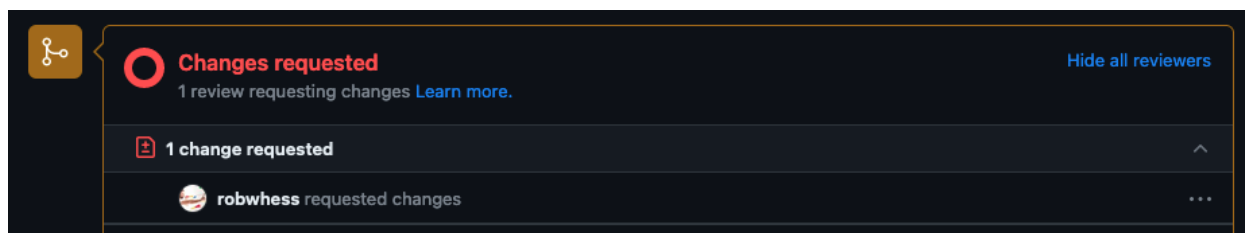
- We can continue adding comments this way to the remaining files that are part of the pull request:



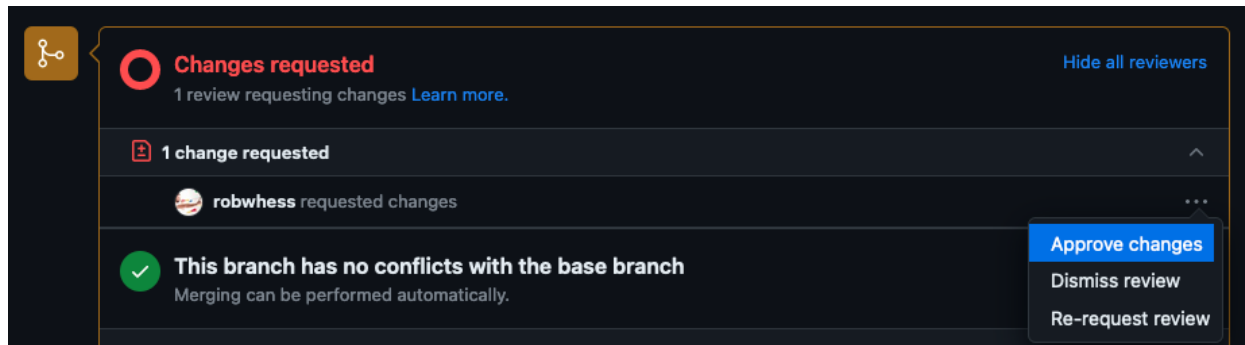
- After we've gone through and reviewed all the files in the pull request, our comments will remain pending until we submit the review. We can do this by clicking the "Finalize your review" button, typing a summarizing comment, and then indicating whether we're just leaving general feedback, approving the pull request, or requesting changes to the code:



- If we select the “Request changes” option and click “Submit review”, the conversation tab will be updated to indicate that changes have been requested:



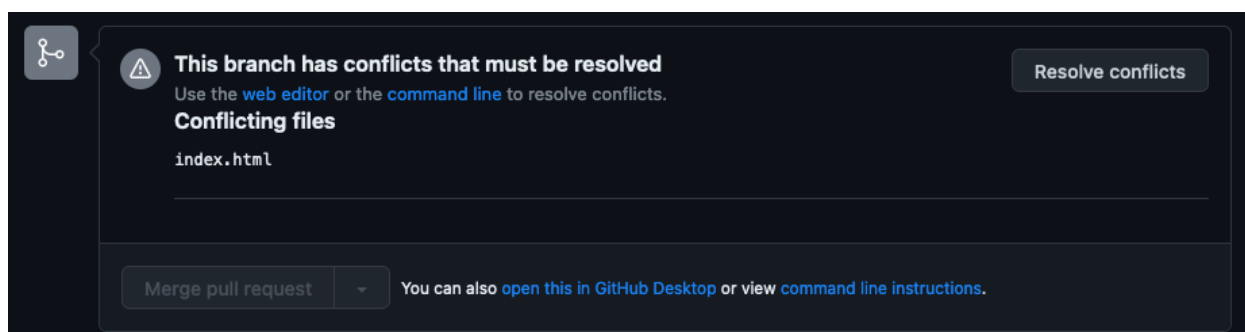
- It is now up to the author of the pull request to respond to our review. Generally, when we request changes, the author of the pull request should either make new commits to incorporate the changes we requested or argue why those changes are not necessary.
- If the author of the pull request makes the requested changes, or if they convince us why they’re not needed, we can either approve the changes or dismiss the review:



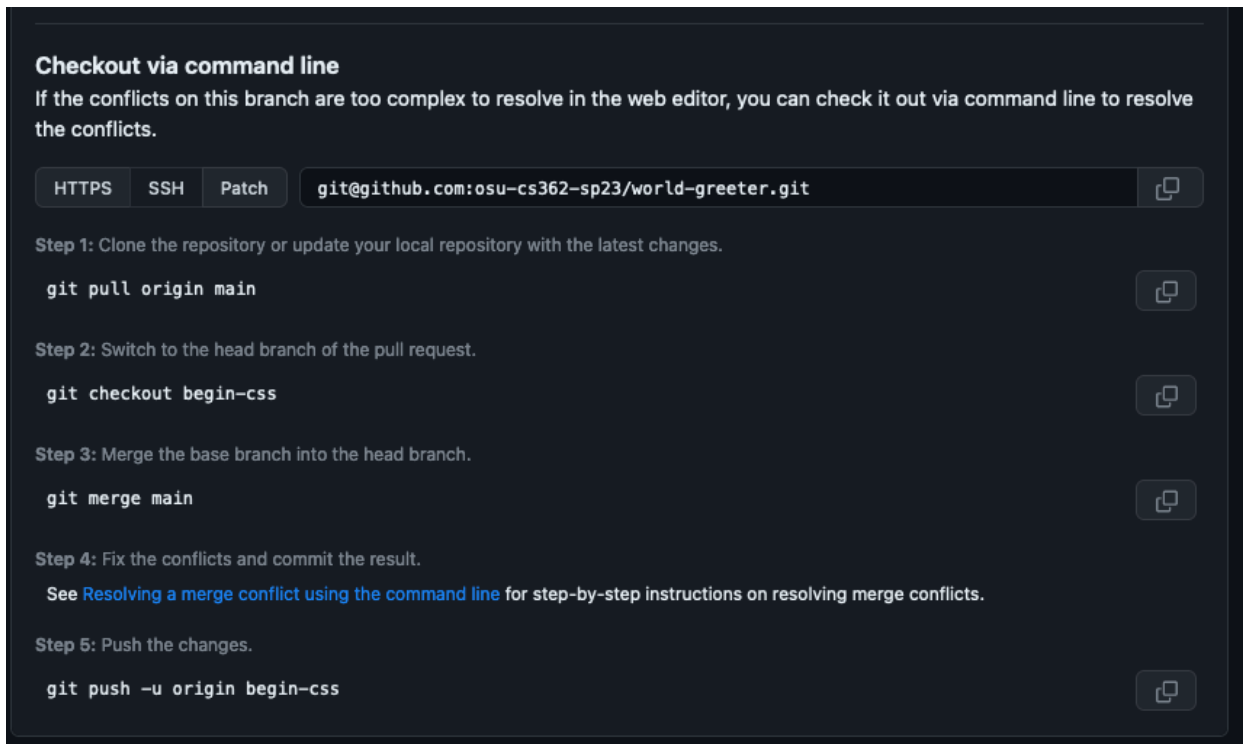
- If we approve the changes, the conversations tab will update to indicate the approval.
- Each team can decide how many approvals are needed before a pull request can be merged. Once the required number of approvals is obtained, we can merge the pull request by clicking the “Merge pull request” button on the conversation tab. This will merge the feature branch into **main** on GitHub.
- **Importantly, once a pull request is merged, all developers should pull the updated **main** branch into their local repos to incorporate the new changes.**
- After the pull request is merged, it is generally safe to delete the feature branch on GitHub. The developer can also delete the branch in their local repository. Deleting branches once they’re merged helps to keep clutter to a minimum.

## Resolving merge conflicts on a pull request

- If we check on our own pull request after the other developer’s pull request is merged into **main**, we’ll unfortunately see that our pull request can no longer be automatically merged because the other developer’s code introduced conflicts with ours:



- We will have to add at least one commit that resolves these merge conflicts before our pull request can be merged.
- If we wanted to, we could click the “Resolve conflicts” button, and GitHub would open a simple web-based editor for us to use to manually resolve the merge conflicts. Usually, though, it is better practice to resolve the conflicts locally in our branch within our normal development environment.
- We discussed how to resolve merge conflicts earlier in the course. If you need a reminder about the Git commands to use to get going on resolving these conflicts, you can click the “command line” link in the warning message in the picture just above, and GitHub will show us a list of steps:



**Checkout via command line**  
If the conflicts on this branch are too complex to resolve in the web editor, you can check it out via command line to resolve the conflicts.

HTTPS SSH Patch `git@github.com:osu-cs362-sp23/world-greeter.git`

Step 1: Clone the repository or update your local repository with the latest changes.  
`git pull origin main`

Step 2: Switch to the head branch of the pull request.  
`git checkout begin-css`

Step 3: Merge the base branch into the head branch.  
`git merge main`

Step 4: Fix the conflicts and commit the result.  
See [Resolving a merge conflict using the command line](#) for step-by-step instructions on resolving merge conflicts.

Step 5: Push the changes.  
`git push -u origin begin-css`

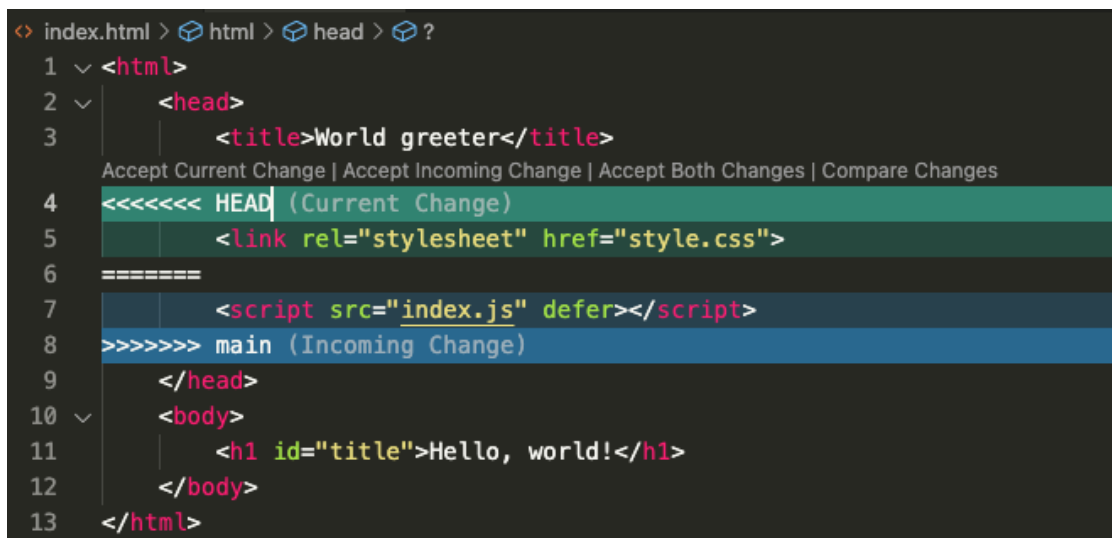
- We don't need to perform the first step, since we already have the repo cloned on our development machine. We can start at step 2 and make sure we have the `begin-css` branch checked out:

```
git checkout begin-css
```

- Then, **assuming we've already pulled the most recent changes to the `main` branch**, we can run step 3 to merge the `main` branch back into `begin-css`. When we do this, Git will report the merge conflicts, which appear in the file `index.html`:

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

- If we open `index.html` in our editor, we'll see the conflict:



```
<> index.html > html > head > ?
1 <html>
2   <head>
3     <title>World greeter</title>
4 <==== HEAD (Current Change)
5   <link rel="stylesheet" href="style.css">
6   =====
7   <script src="index.js" defer></script>
8 >==== main (Incoming Change)
9   </head>
10  <body>
11    <h1 id="title">Hello, world!</h1>
12  </body>
13 </html>
```

- Here, the conflict arises because the other developer's pull request added a line to `index.html` (the `<script>` line) at the exact same place we added a new line in our own feature branch (the `<link>` line), and Git doesn't know how to reconcile the different changes.
- In this case, we want to keep both changes, since each change is compatible with the other. We can edit `index.html` to keep both of the conflicting lines:



```

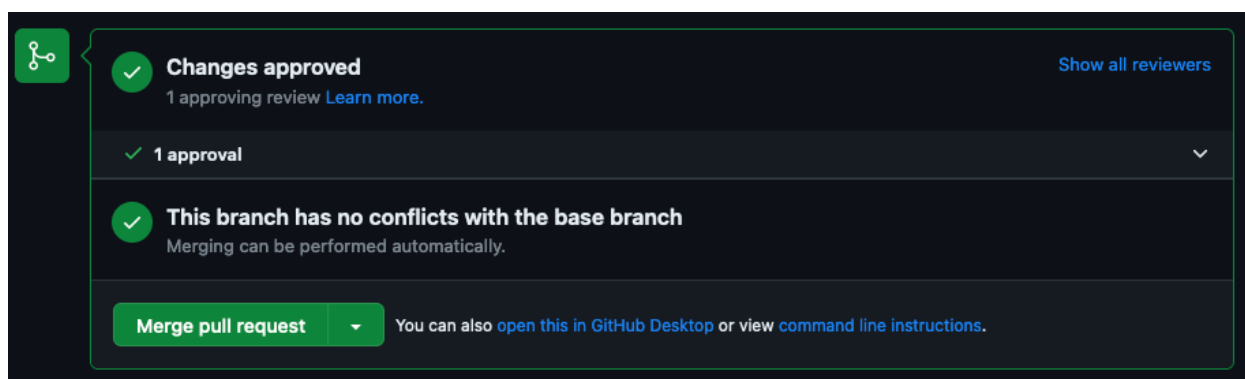
<> index.html > ...
1  <html>
2  <head>
3      <title>World greeter</title>
4      <link rel="stylesheet" href="style.css">
5      <script src="index.js" defer></script>
6  </head>
7  <body>
8      <h1 id="title">Hello, world!</h1>
9  </body>
10 </html>

```

- Once we resolve the merge conflict, we can go back to the terminal and run the following command to mark the conflict as resolved (running `git status` will remind you of this):

```
git add index.html
```

- Then, we can run `git commit` to conclude the merge and add the merge commit. Finally, we can pick back up at step 5 in the list of commands GitHub displayed for us and run `git push` to push our conflict resolution to the `begin-css` branch on GitHub.
- This should make our pull request auto-mergeable again. We can wait for the required approvals, and once we get them, we can merge the pull request:



- Again, it's important to emphasize that once a pull request is merged, that is a signal to all developers on the team that they need to `git pull` the `main` branch of their local repo! It's very important for every developer to

make sure they have the most up-to-date code in `main` so that when a developer makes a new feature branch off of `main`, they are starting their branch with the current version of the code.

## Enforcing the GitHub flow on a GitHub repository

- You might have noticed as we were going through the examples above that it was possible to merge either pull request right from the beginning, even when the pull request hadn't received any review or approval.
- In fact, at the moment, it would be possible for any developer with write access to our repository to push commits directly to the `main` branch without opening a pull request.
- In other words there's currently nothing *enforcing* use of the GitHub flow on our repository other than the honor system.
- If we want, we can put more strict enforcement mechanisms in place by creating a **branch protection rule** in GitHub (provided we have administrative permission on the repository). We can do this in our repository's settings on GitHub, specifically under "Settings → Branches".
- Under the "Branches" settings, there is a button we can click to add a new branch protection rule. If we click this button, it will open up a menu we can use to customize the branch protection rule we're creating.
- The first thing we'll need to specify is the branch name pattern. We have [some flexibility here](#) to specify a pattern that matches multiple branches, but at the moment, we just want to protect the `main` branch, so we can enter "main" here.
- After setting the name pattern, we can choose the specific protections we want to put in place for the branch. If we want to simply enforce the GitHub flow the following protections are the ones we need to turn on:
  - **Require a pull request before merging**
  - **Require approvals**
    - This will not appear until we select "require a pull request before merging".
- We can customize the protections further from there based on our team's preferences. For example, once we set up a continuous integration pipeline that

automatically runs our tests against every pull request, it will be useful to turn on the “**Require status checks to pass before merging**” protection. This will prevent a pull request from being merged into `main` unless the tests pass for the branch being merged.

- Once we create the branch protection rule with the above protections, it will no longer be possible to push commits directly to the `main` branch on GitHub. Instead, commits will have to be pushed to a feature branch and merged into the `main` branch via pull request. In other words, we will have to follow the GitHub flow.