🧪 Lab 2 - Git workflow

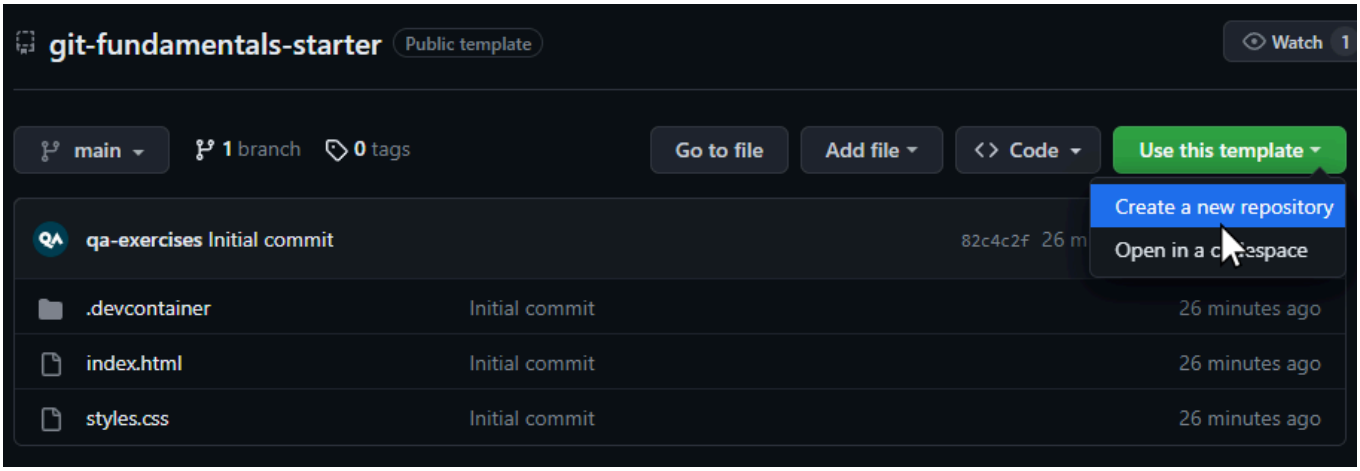
In this exercise, you'll practice the most important components of a software developer's daily Git workflow. Along the way, you'll create a simple website which you're free to customise as much as you like. 👩‍🎨

Table of contents

Part 1: Creating a repository

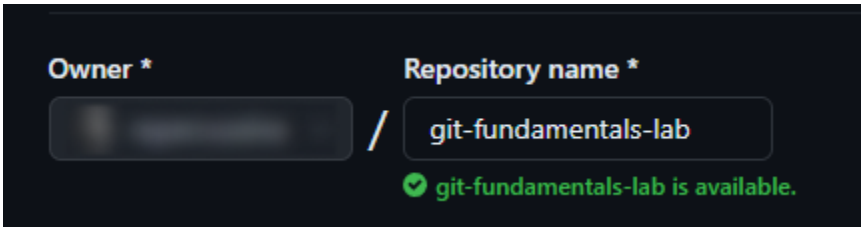
To get started, you'll create a clone of an existing repository hosted on GitHub.

1. Log into **GitHub**.
2. Click **this link** to view the starter repository for this exercise.
3. Click "**Use this template**", then "**Create a new repository**":



You'll then be prompted to configure settings for your new repository.

4. Give your repository any name you like.

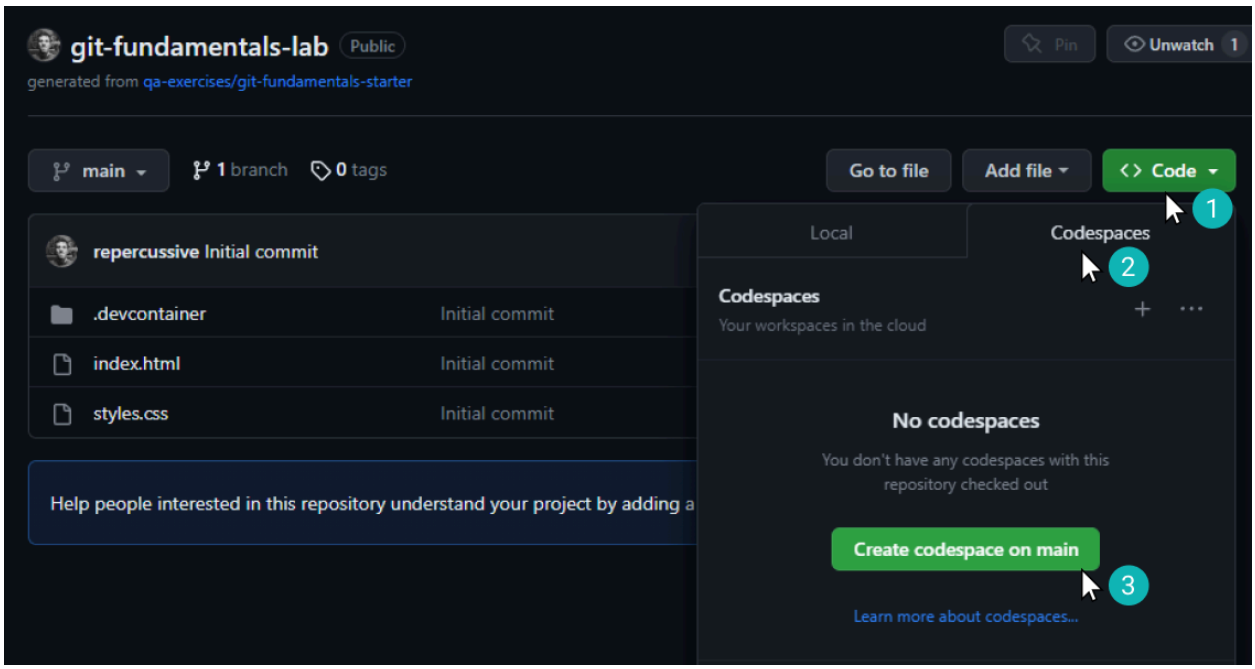


You can ignore all the other options for now.

5. Click "**Create repository**" and wait a few seconds. This creates a copy of the starter repository; the copy belongs to your GitHub account. The repository page should automatically open.

Next up, you'll dive into your project so that you can make changes. For now, we suggest simply opening your project in a web-based codespace.[1] This is to keep things simple for the exercise and allow you to focus on the Git workflow.

6. With your repository page open, click "**Code**", then "**Codespaces**", then "**Create codespace on main**".



Wait a moment for the codespace to load, and you're all set up for Part 2.

---

[1] *It's also common practice to run Git on your local computer and make changes to the repository there. This requires you to have Git installed, and you also need to authenticate with your GitHub account via SSH or HTTPS.*

For this course, it's recommended to use the codespaces provided, not only for simplicity, but also because they are configured with the necessary VS Code extensions pre-installed.

Part 2: The add/commit cycle

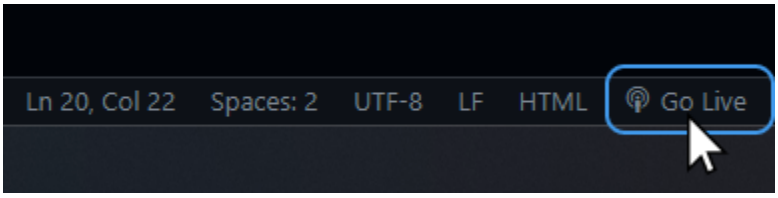
In this section, you'll make incremental changes to your repository and save them as commits.

You should have the codespace for your repository open.

Ensure that you can see the Explorer and Terminal panes in your codespace. If you ever lose these panes:

- Press **Ctrl+Shift+E** to restore the Explorer
- Press **Ctrl+J** to restore the Terminal

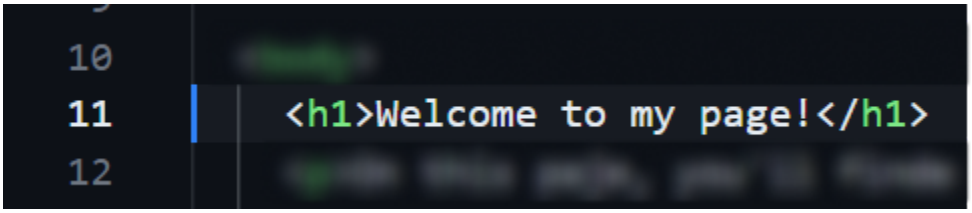
1. In the Explorer pane, click on the **index.html** file to open it. You'll see the HTML for a simple webpage.
2. You can view this webpage in your browser. In the bottom right corner of the screen, you should see a button that says "**Go Live**".



   Click it, and a new tab will open displaying the page. Keep this tab open, as it will update in real time to reflect the changes you make in upcoming steps. *(Along the way, we'll fix the typos, add content, and improve the appearance of the page.)*

📝 ***Change #1:*** *Edit the main header*

Notice the header text on line 11 of **index.html**.



3. Change this text so that instead of "Welcome to my page!", it's personalised to you, e.g. "Welcome to <insert your name here>'s page!"

4. In the terminal, type **`git status`** and press Enter. You'll see a message saying your modification to **index.html** has not been staged, which means it won't be included in the next commit.

5. To fix this, *stage* the change by entering **`git add index.html`** in the terminal.

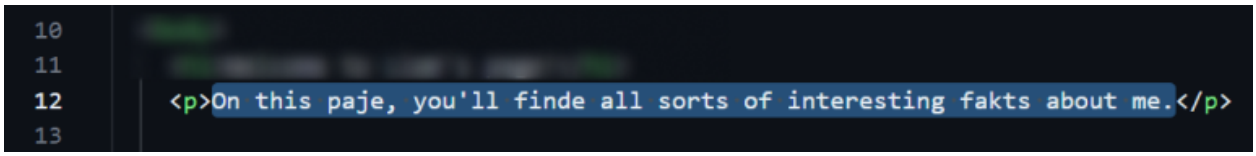
6. Enter **`git status`** again in the terminal. You'll see that your modification to **index.html** is now set to be included in the next commit.

7. Next, enter **`git commit -m "Edit page header"`** in the terminal. This will create a commit with the message "Edit page header".

8. Enter **`git log --oneline`** in the terminal to see a condensed summary of all the commits that have been made. Each commit has a **hash** (the scrambled numbers and letters that act as each commit's ID). You should see that your most recent commit appears at the top of the list.

📝 ***Change #2:*** *Fix typos*

Notice the typos on line 12 of **index.html**.



Your task is to create a new commit with the typos resolved. You'll follow a similar process as before:

a. Make changes (fix the typos)
b. Run **`git status`** to review the unstaged changes
c. Run **`git add index.html`** to stage your changes to **index.html**
d. Run **`git status`** to confirm the changes were staged
e. Run **`git commit -m "<message>"`** with a suitable message, such as "Fix typos"
f. Run **`git log --oneline`** to review the commit history.

📝 ***Change #3:*** *Complete the "About me" section*

Notice the "About me" section, which hasn't yet been populated with information.

```
14    <h2>About me</h2>
15
16    <h3>Hobbies and interests:</h3>
17    <ul>
18      <li>??????</li>
19      <li>??????</li>
20      <li>??????</li>
21    </ul>
22
23    <h3>Places I've visited:</h3>
24    <ul>
25      <li>??????</li>
26      <li>??????</li>
27      <li>??????</li>
28    </ul>
```

Your task is to create a new commit, where the "About me" section is filled in with information all about you. The categories are just suggestions; you can add whatever interesting facts you'd like about yourself 🙂 You'll follow a similar process as before:

a. Make changes (fill in the "About me" section)
b. Run **`git status`** to review the unstaged changes
c. Run **`git add index.html`** to stage your changes to **index.html**
d. Run **`git status`** to confirm the changes were staged
e. Run **`git commit -m "<message>"`** with a suitable message
f. Run **`git log --oneline`** to review the commit history.

If everything went smoothly, your commit history should look something like this:

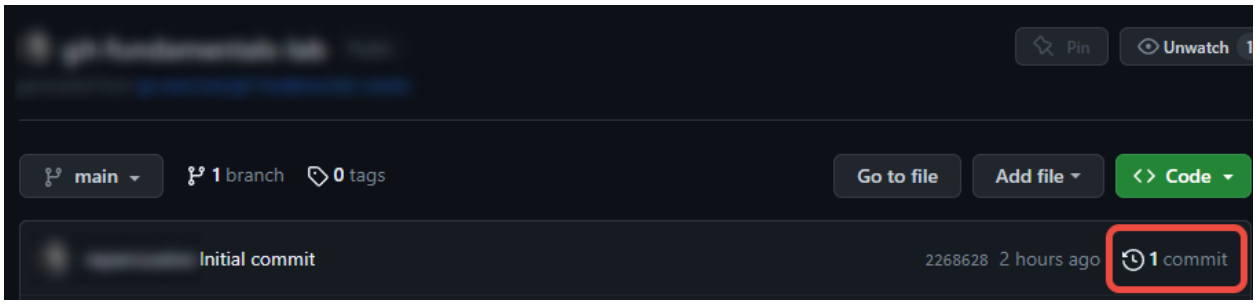
```
de25d5a (HEAD -> main) Fill in the about me section
fe2ecc3 Fix typos
7249e5d Edit page header
2268628 (origin/main, origin/HEAD) Initial commit
```

Part 3: Pushing your changes

Right now, all the changes you've made only exist inside the codespace and haven't been backed up to GitHub.

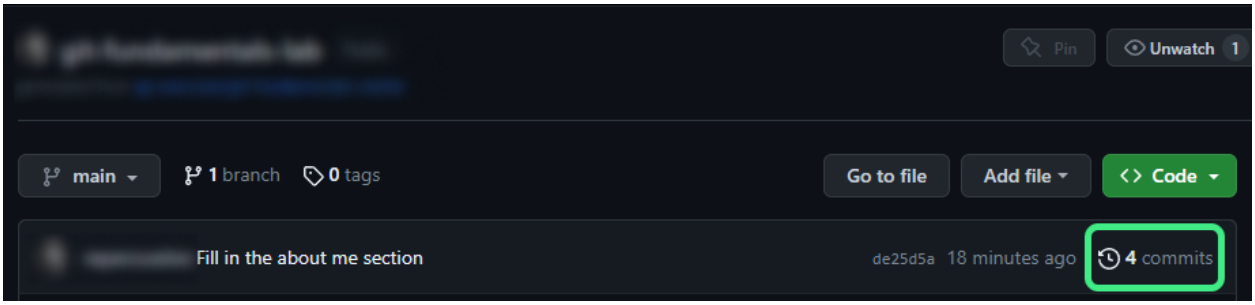
1. To see that this is the case:
    a. Open **GitHub** in a new tab.
    b. Click on your profile icon in the top right corner.
    c. Click "**Your repositories**".
    d. Click on the repository you created for this exercise.

Notice that only **1 commit** exists in the GitHub repository. This must mean that the 3 commits you made previously haven't been synced.



2. Return to your codespace.
3. In the terminal, run **`git push origin main`**
    - ***`git push`*** *is the command used to upload commits from a local repository (your codespace) to a remote repository (GitHub).*
    - *The name associated with the remote repository, by convention, is* ***`origin`****.*
    - ***`main`*** *is the name of the branch you made your commits on.*

After running the **`git push`** command, refresh the GitHub repository page. You should see that the additional commits you made have appeared, which means your changes have been backed up.



Part 4: Creating a branch for a new feature

So far, you've been committing all your changes linearly onto the **main** branch. This is not good practice for several reasons, including:

- *The main branch typically represents the production-ready codebase. If you commit directly to main, you might introduce bugs and other issues that could **disrupt the production environment** and the experience of end users.*
- *If you only ever committed onto the main branch, it would be **impossible to track incremental revisions** to work-in-progress features.*
- *Having a single branch makes **collaborative development** unfeasible.*

In this section, you'll overhaul the aesthetics of your webpage, but you'll make your changes in a **separate branch**. In the next step you'll *merge* those changes back into the main branch. Let's go!

View existing branches

1. Open the terminal in your codespace.
2. To see the list of current branches, run the **`git branch`** command in the terminal.

You should see that there is a single branch (main). The asterisk (*) signifies that you are currently "on" the main branch

Create a new branch

To create a new branch, you can use the **`git branch`** command followed by the desired name of the new branch.

3. In the terminal, run **`git branch feature/ui-update`**
    - *This creates a new branch called "feature/ui-update". This new branch stems from the last commit on the main branch (as that's where we are currently located).*

Switch to the new branch

Although you just created a new branch, you're still on the main branch. You can see this if you run **`git branch`** again:

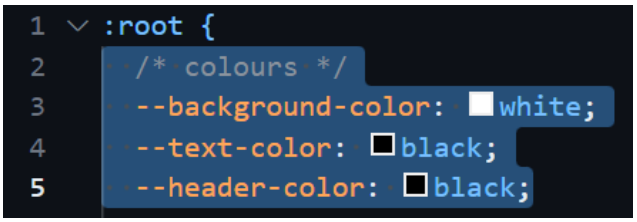
```
(main) $ git branch
  feature/ui-update
* main
```

4. Run **`git switch feature/ui-update`** to switch to the new branch.
5. Run **`git branch`** to confirm that you have moved onto the new branch.

Make changes on the new branch

6. Open **styles.css** from the Explorer pane.

First, you'll change the colours of the page. At the top of **styles.css**, you'll see various properties that correspond to the page's colour scheme:



You can hover over any of these colours to pick new ones:

7. Change the colours to create whatever colour scheme you like.
8. When you're done, **`git add`** and then **`git commit`** your changes with a suitable commit message, such as "Change colour scheme".

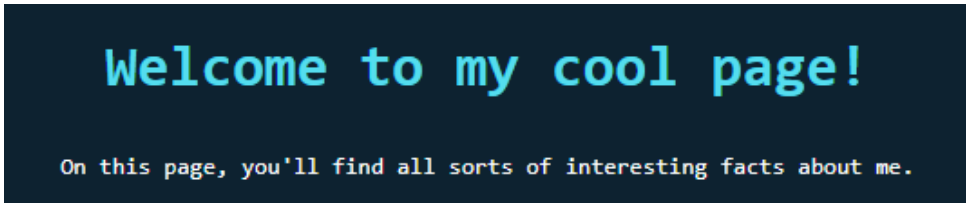
Next, you'll change the typography on the page.

9. On line 8 of **styles.css**, change the font property from “serif” to something else - for example, “sans-serif” or “monospace”:

```
7     /* typography */
8     --font: monospace;
9   }
```

10. Now, **`git add`** and then **`git commit`** your changes with a suitable commit message, such as “Change page font”.

Your page should look a little more visually appealing now, for example:



11. **Important:** push your changes to GitHub by running **`git push origin feature/ui-update`**

Switch back to the main branch

The changes you’ve made only exist on the “feature/ui-update” branch, and aren’t present on the main branch.

12. Run **`git switch main`** to return to the main branch.

Notice that your visual improvements no longer appear:

That’s OK - the changes are still saved on the “feature/ui-update” branch, but the new branch hasn’t yet been merged with the main branch.

You’ll merge the branches in the next part.

Part 5: Merging changes into the main branch

It's possible to use the **`git merge`** command to merge branches locally. The merged branch can then be pushed to the remote repository.

However, in a collaborative environment, it's more common to use a "**pull request**".

- *A pull request is a mechanism for **proposing changes** to a repository.*
- *If a developer has created changes on a separate branch that they wish to be merged with the main branch (for example), they can create a pull request.*
- *A team member can then **review the code** before deciding to **merge it** with the main branch.*

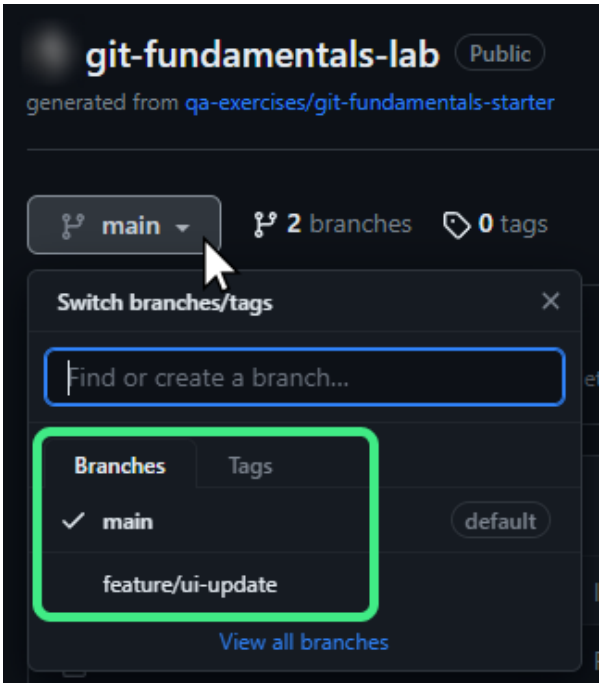
Pull requests promote collaboration, code quality, and knowledge sharing.

It's not a feature built in to Git - it's part of GitHub. Other hosted repositories have a similar concept (e.g. GitLab's merge requests).

View branches on GitHub

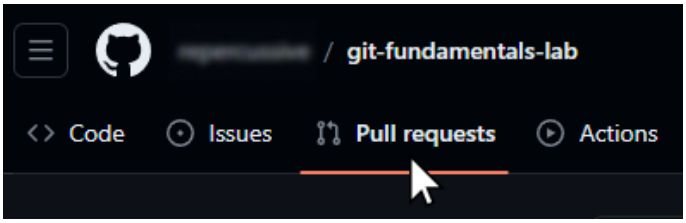
1. Open the GitHub page for the repository you've been working on.

You should see that there are now two branches. If not, return to your codespace and ensure you've pushed both branches to the remote repository.



Create pull request

2. Navigate to the "**Pull requests**" tab at the top of your GitHub repository page.

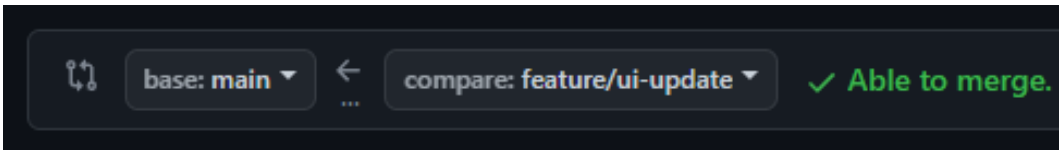


3. Click the "**New pull request**" button. A screen should display with the title "Compare changes".

We now need to specify two things:

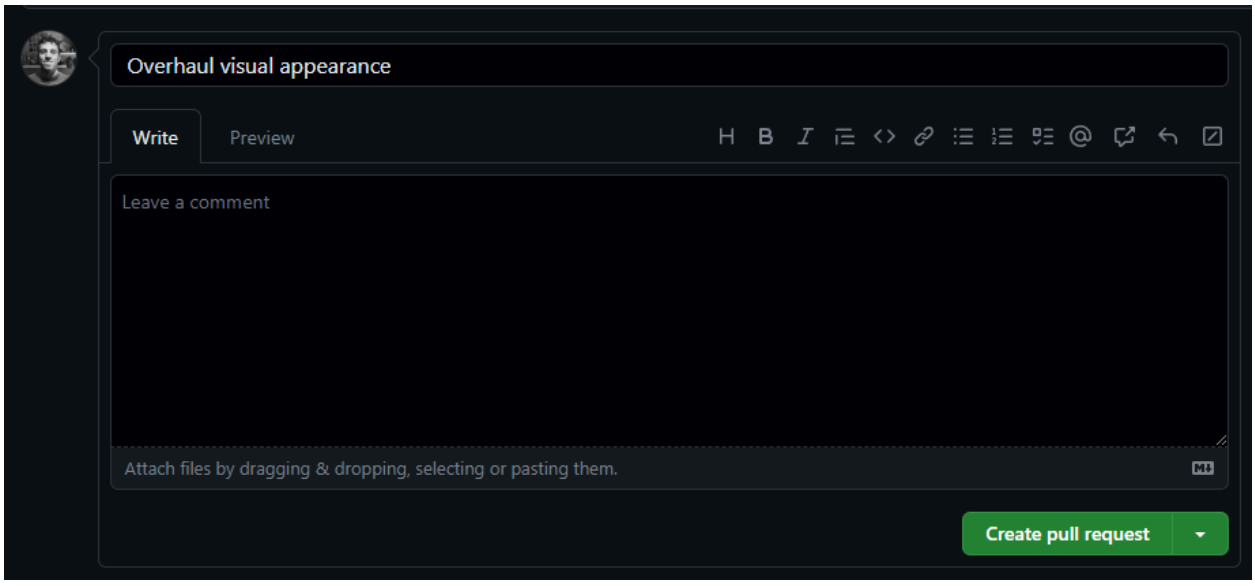
- The branch which the changes will be **merged into** ("*base*")
- The branch where the changes are **coming from** ("*compare*")

4. Select the "base" and "compare" branches using the dropdown menus as shown.



★ The base branch is **main**
★ The branch we are comparing with is **feature/ui-update**

5. Click "**Create pull request**".
6. On the next screen, you can give your pull request a name, and add comments for whoever will be reviewing your request.



7. Click “**Create pull request**” again to open the request.

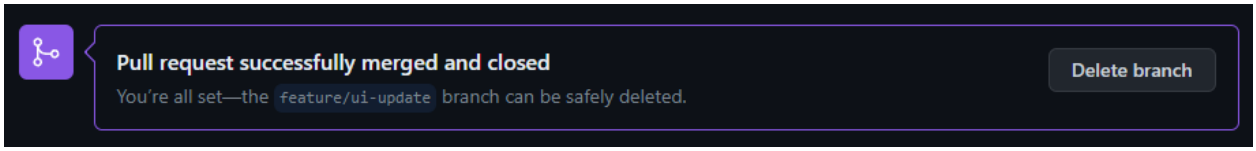
In a real scenario, someone else would review this request and either reject or accept it.

8. To accept the proposal, click the green “**Merge pull request**” button and then the “**Confirm merge**” button.

You should see a message that says “Pull request successfully merged and closed”.

Because the changes have now been merged, you can safely delete the **feature/ui-update** remote branch.

9. Delete the branch with the “**Delete branch**” button.



Sync changes locally

10. Return to your codespace, and ensure that you are located on the main branch.

Notice that the changes you made in **styles.css** are not up to date. This is because you still need to download the latest updates from the remote repository.

11. To do this, run **`git pull`** in the terminal.

You should now see that **styles.css** reflects the merged changes.

It’s also a good idea to delete the local version of the **feature/ui-update** branch, as it now doesn’t serve any purpose.

12. Run **`git branch -d feature/ui-update`** in the terminal. This will delete the unnecessary branch.
13. Run **`git branch`** in the terminal to see that only the main branch remains.

Part 6: Resolving a merge conflict

In the last part, Git was able to automatically merge the new feature into the main branch.

However, sometimes Git isn't able to merge automatically. For example, if the same line of code is modified in branch A and branch B, and the two branches are then merged together, there will be a merge conflict - because Git doesn't know which branch to accept the changes from. In these cases, the **conflict needs to be resolved** before the merge can complete.

Make a change on a new branch

You'll start by making some changes in a separate branch.

1. Open the terminal in your codespace.
2. In the terminal, enter `git switch -c experimental`
    - *This command **creates a new branch** called "experimental" and also **switches to the new branch**.*
3. Make some kind of change to the main header on line 11 of **index.html** - something that you'll remember, like adding a 🐸 emoji:

4. Now **commit** and **push** your changes (with the following commands):
    a. `git add index.html`
    b. `git commit -m "Change main header"`
    c. `git push origin experimental`

Make a conflicting change on the main branch

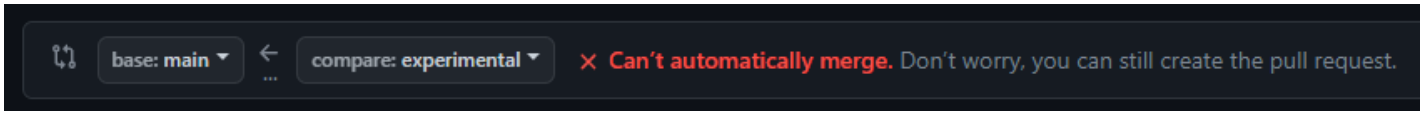
5. In the terminal, enter `git switch main` to return to the main branch.
6. Make a **different change** to the **the same line** (line 11) of **index.html** that you edited before - e.g., adding a 🐧 emoji to the header:

7. Now **commit** and **push** your changes (with the following commands):
    a. `git add index.html`
    b. `git commit -m “Change main header”`
    c. `git push origin main`

Make a pull request

Now, you’ll go through the same process as before to create a pull request - this time attempting to merge the **experimental** branch into the **main** branch.

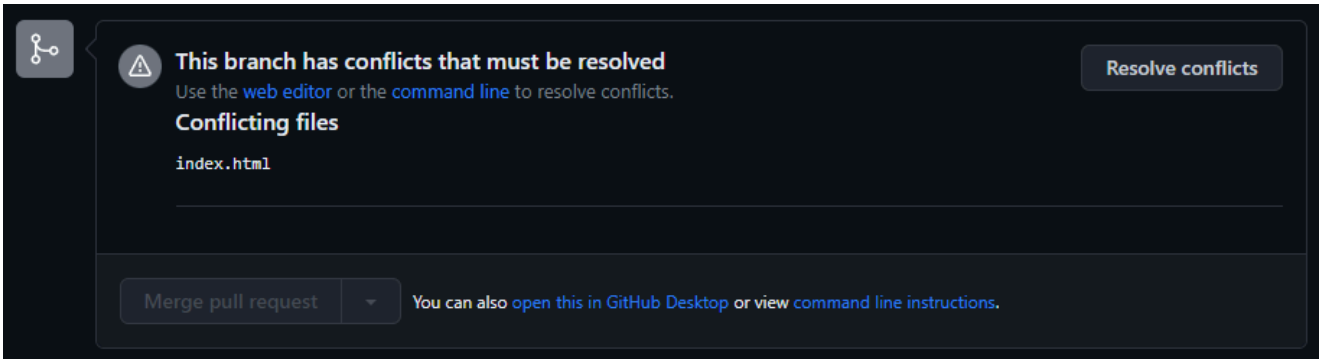
8. Open the GitHub page for your repository and navigate to the “**Pull requests**” tab.
9. Click the “**New pull request**” button.
10. Set the base branch to *main*, and the compare branch to *experimental:*



You should see a warning: “Can’t automatically merge”, which means that there is a merge conflict. You’ll resolve this soon.

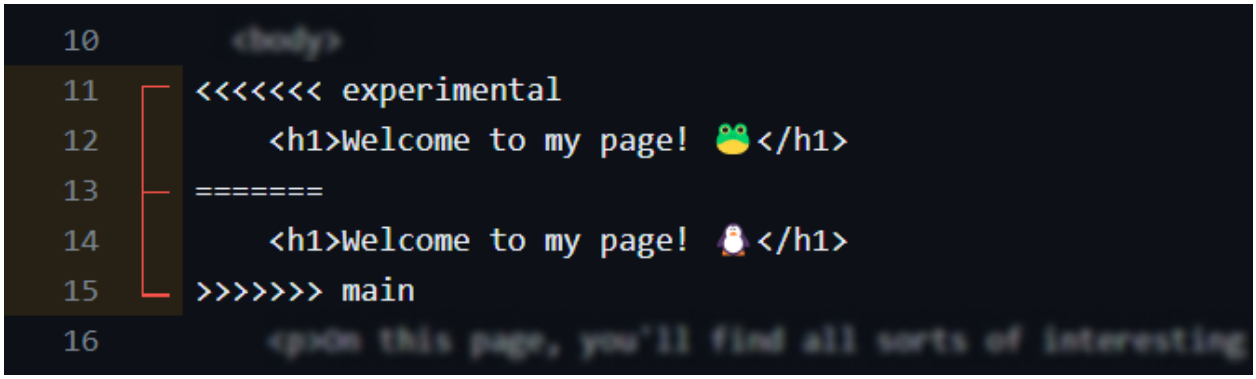
11. Click the “**Create pull request**” button, and then press “**Create pull request**” again on the following page.

Notice that you can’t merge the pull request because of the unresolved conflict.



12. Click the “**Resolve conflicts**” button.

You'll be taken to a text editor which will allow you to resolve the conflicting changes. Notice that the **lines related to the conflict are highlighted**:



You can also see 3 extra lines which have been generated by Git, called "conflict markers", which show us what the conflicting pieces of code are and which branches they comes from.

To explain what you are seeing here:

- **`<<<<<<< experimental`**: This is a conflict marker that marks the beginning of the conflicting changes from the "experimental" branch.
- **`<h1>Welcome to my page! 🐸</h1>`**: The content of the **index.html** file as it exists in the "experimental" branch. In this case, it's an HTML <h1> element.
- **`=======`**: This is another conflict marker that separates the changes from the "experimental" branch from the changes in the "main" branch.
- **`<h1>Welcome to my page! 🐧</h1>`**: This is the content of the file from the "main" branch. It's also an HTML <h1> element but with different text content.
- **`>>>>>>> main`**: This is the final conflict marker, indicating the end of the conflicting changes from the "main" branch.

To resolve the conflict, you need to **choose which version of the code you want to keep** (or make some modification that combines both changes).

Then, you need to **remove all the conflict markers**, only leaving behind the desired content.

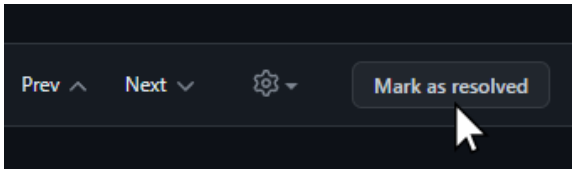
Here's an example of how you might do this:

```
10      <body>
11        <h1>Welcome to my page! 🐧🐸</h1>
12        <p>On this page, you'll find all sorts of interesting facts about me.</p>
```

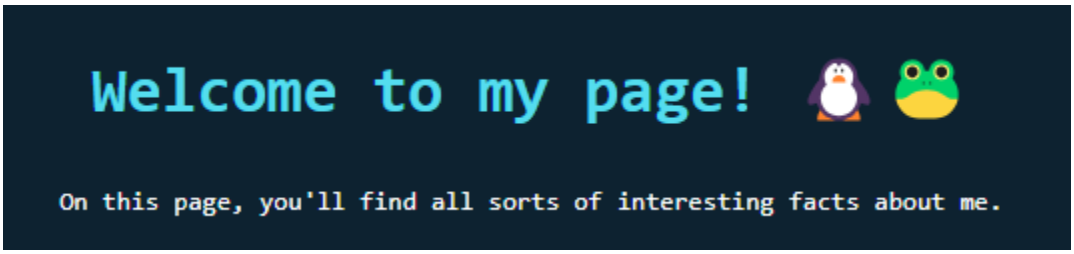
Notice:

- *The conflict markers have been removed.*
- *The conflict itself has been resolved by combining the changes.*

13. Once you've resolved the conflict, press the "**Mark as resolved**" button at the top of the page, then "**Commit merge**".



14. You'll return to the pull request page. Press the "**Merge pull request**" button, then "**Confirm merge**".
    a. *Feel free to delete the experimental branch.*
15. Return to the terminal in your codespace, and run **`git pull`**
16. Confirm that your webpage displays the correct content (as you specified when you resolved the conflict).



Part 7: Reverting changes

Sometimes we make mistakes, including when we're working on a software application.

For example, you might accidentally delete some code, or introduce a bug, or you might just decide that a new feature is unwanted.

Thankfully, even if we commit these unwanted changes, Git allows us to easily revert to a previous commit.

Make a bad commit

1. Open the terminal in your codespace, and ensure you are on the main branch.
2. Open the **index.html** file and delete all of the content in it. Woops! 🤦
3. **Stage** and **commit** the change (using the following commands):
    a. `git add index.html`
    b. `git commit -m "Bad commit"`

Understand how to reference the commit you want to revert

To undo a change, we can use use **`git revert`** followed by whatever commit we want to undo. To reference that commit, we could use its commit hash (ID). However, there's another way to reference the accidental commit we just made.

4. Run **`git log --oneline`** to view the list of commits, with the most recent "Bad commit" at the top of the list.

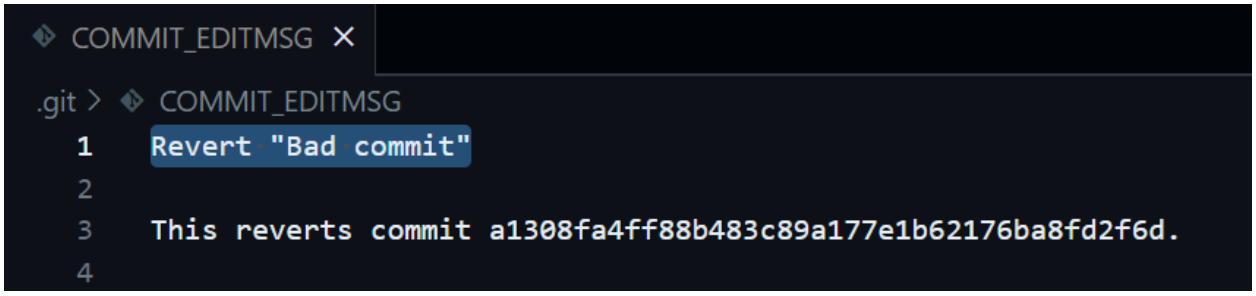
```
@repercussive ➜/workspaces/git-fundamentals-lab (main) $ git log --oneline
a1308fa (HEAD -> main) Bad commit
```

Note the commit hash. But the commit is also marked with **HEAD**. HEAD, like a commit hash, is a reference to a commit - but unlike a hash, it's context-dependent: HEAD points to wherever you currently are in the commit history. Because HEAD currently represents the bad commit that we wish to undo, we can supply HEAD to the **`git revert`** command.

Revert the commit

5. Run **`git revert HEAD`**

A text editor will appear. On the first line of this file, Git has suggested a commit message for your reverting change.



Why?

The **`git revert`** command doesn't remove any commits from the history. Instead, **`git revert` creates a new commit** that contains the **opposite changes** of the selected commit - effectively cancelling out changes from the previous commit. The original commit history stays instact, and the new "revert" commit is added to the branch.

Git is simply asking you to provide a commit message for this new commit.

This behaviour of **`git revert`** is generally desirable. By preserving the commit history, you provide transparency and help collaborators understand which changes were reversed.

Git provides other commands like **`git reset`** and **`git rebase`** for scenarios where modifying history is necessary, but you should only use these if you know exactly what you're doing.

6. Close the COMMIT_EDITMSG tab. This will complete the revert operation.
7. Run **`git log --oneline`** to see how your reverting change fits into the commit history.

Important: when you're finished

Congrats - you completed the exercise!

You'll want to stop your codespace to avoid wasting your 60 free monthly codespace hours. To do this: with the codespace tab open, press **Ctrl+Shift+P**, which should open up the command palette. Type "**stop codespace**" - after which the option to stop the current codespace should be highlighted. Press Enter, then wait a few moments for it to stop.

Bonus

Try out Learn Git Branching for a series of interactive exercises.