Week 8: Collaborative Coding 2

This week's material is based on the **Version Control Course** from the School of Mathematics and Statistics from the University of Glasgow. The content has been reduced to fit the class structure. At the end this week, please provide some feedback on the materials on https://uofg.qualtrics.com/jfe/form/SV_56jF2LNgmA6qrhY.

Add collaborator

To collaborate with users in a repository that belongs to your personal account on GitHub.com, you can invite the users as collaborators.

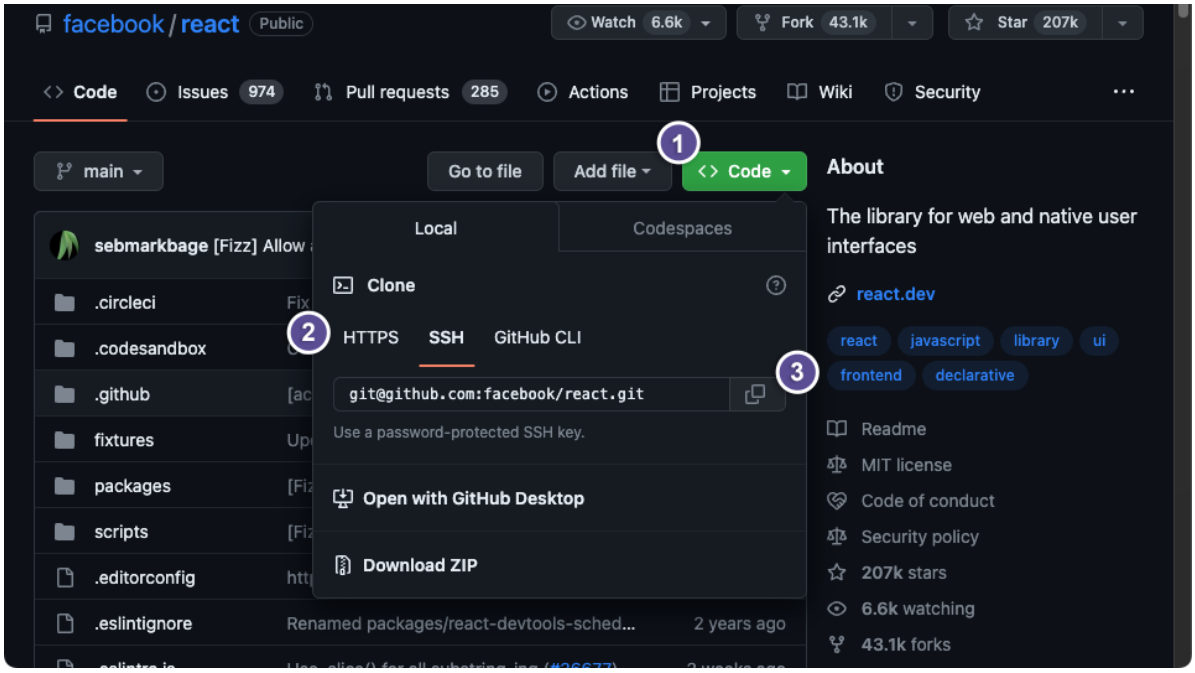
Clone

One way to download a project to your computer is to use Git's `clone` method. To use `clone`, we need to tell Git installed on our computer to make a copy (or 'clone') of another repository, in this case from GitHub.

Clone is a Git feature that allows a user to make a copy of a repository, including all the associated metadata on their computer so they can work on it. This is an essential feature and makes sense for a team of collaborators, with read-and-write privileges to the repository, all working on a project together.

Command-line

First, we need to copy the address of the repository we want to clone.



1. Click on the green button labelled "Code"
2. Choose either HTTPS or SSH, depending on how you connect to GitHub with Git on your computer (see the Configuration section of Unit 1)
3. Copy the address

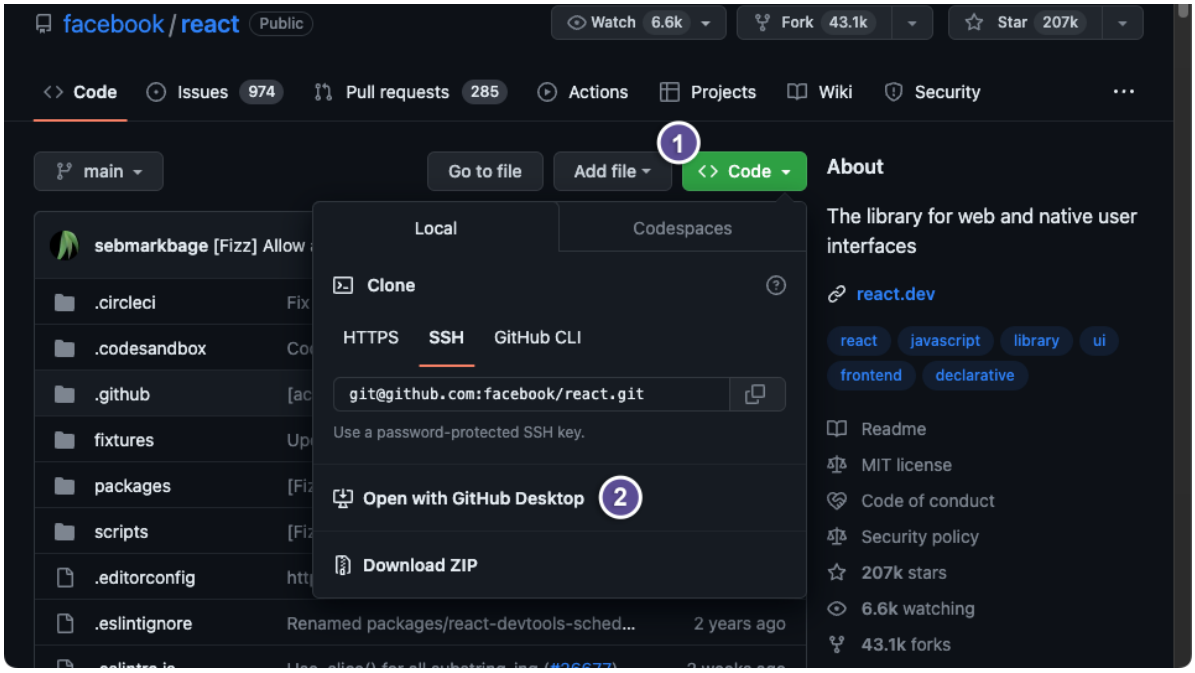
Then in your terminal, navigate to your desired folder and issue the following command:

```
git clone <address>
```

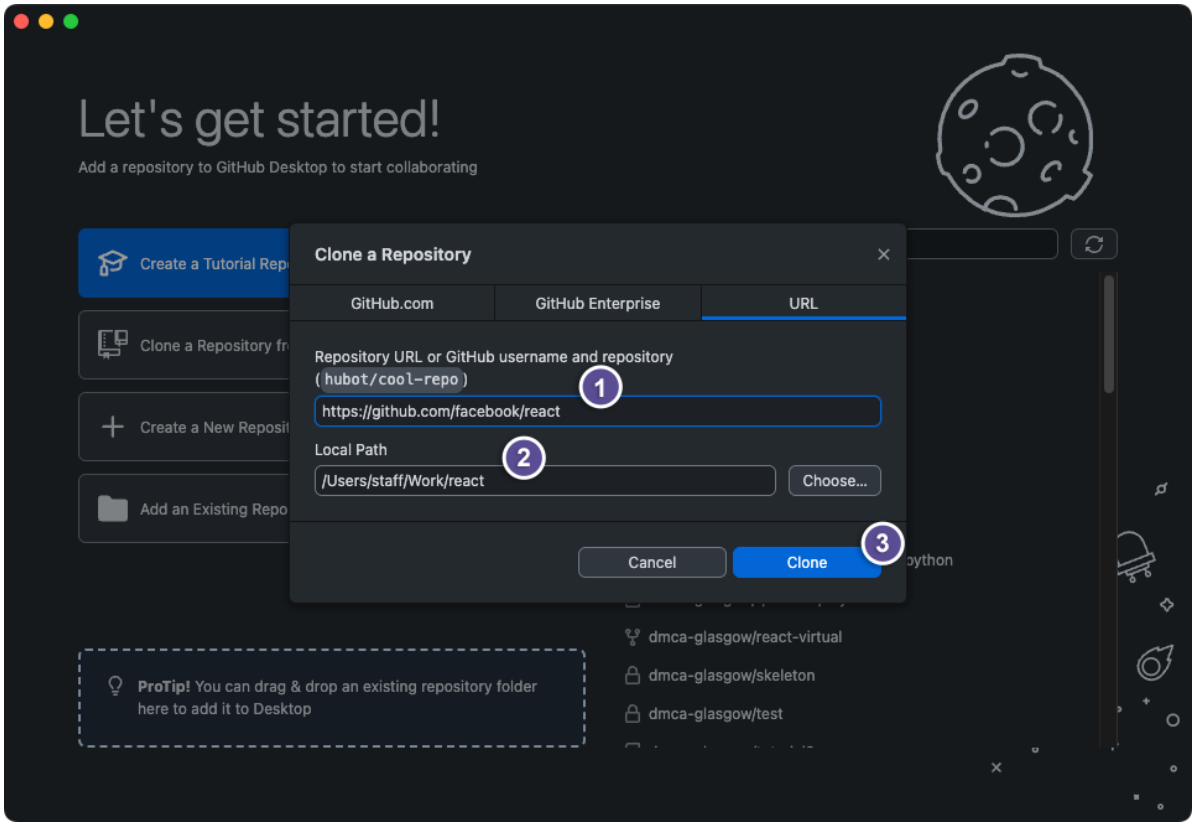
You should now see the repository has been cloned to your computer.

GitHub Desktop

First, navigate to the repository you want to clone on the GitHub website.



1. Click on the green button labelled ‘Code’
2. Click on ‘Open with GitHub Desktop’
3. You should then see a window like this pop up in GitHub Desktop:



1. The URL of the repository on GitHub has automatically been populated
2. Choose where you would like the repository to be cloned to on your computer
3. Click 'Clone'

Using this approach, we have a copy of the files and folders in the repository, but we have also copied the metadata which makes this into a Git repository. To answer the example above, now if a project maintainer makes an update to the project on GitHub, we can easily sync both versions:

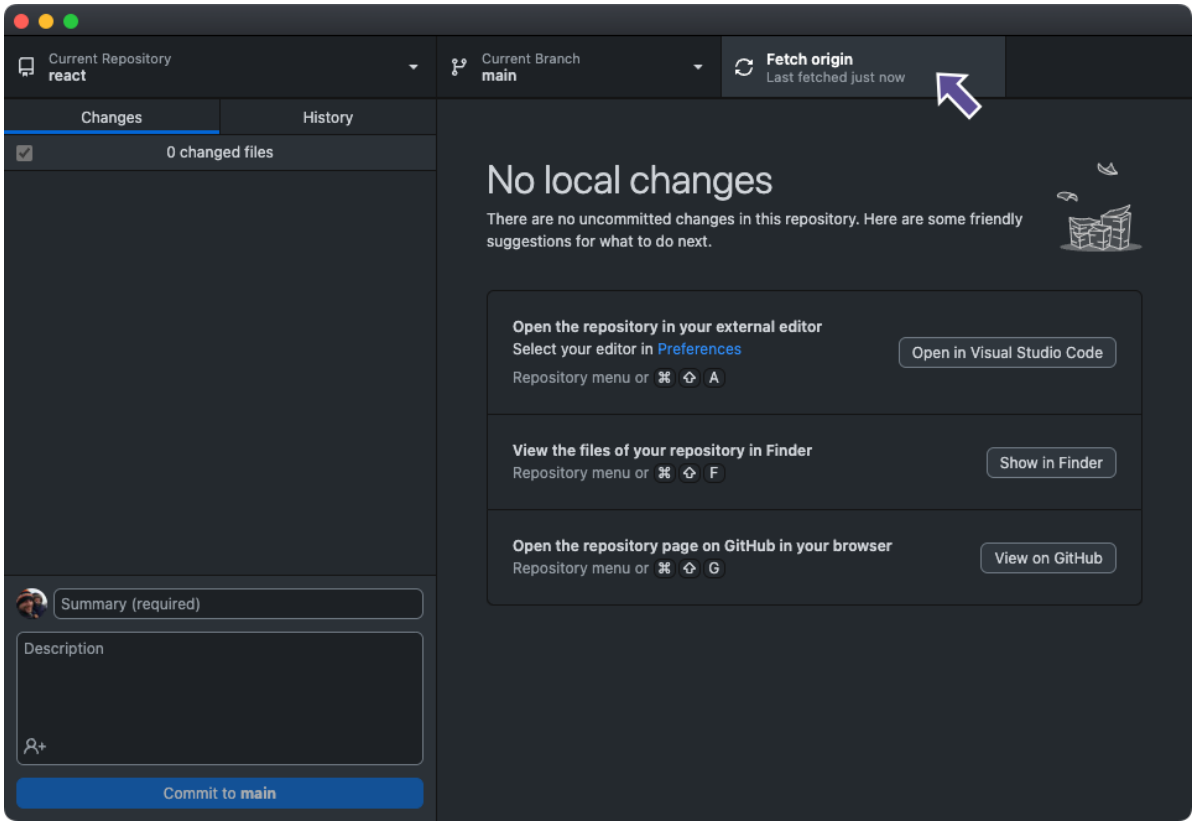
Command-line

In the project folder just issue the `pull` command:

```
git pull
```

GitHub Desktop

On GitHub Desktop, click the 'Fetch origin' button in the top bar:



Branching and branching strategies

Introduction

Typically a project will have one main direction of development— to reach a predetermined goal, such as a fully working prototype or to complete a course assessment. When Git uses the analogy of a tree (with branches), this main direction could be thought of as the 'trunk' of your development flow.

So far, we have only committed changes to one branch, the default branch called `main`, which, in our tree analogy, can be thought of as the trunk. If you are working on your own, or with a small group and you are not all committing changes at the same time, this can be a valid approach and avoids a lot of complexity in using Git.

Branches allow you to 'branch off' from the trunk in a new direction, which we may then add back to our trunk at a later time (see next paragraph). On this new branch, you can experiment with new ideas or work away in isolation, committing to Git and pushing your changes to GitHub, safe in the knowledge that nothing you do will affect the main direction of development. Likewise, the main direction of development on `main` can carry on in parallel with its own changes, and these will not affect your isolated branch.

It's possible to **merge** the changes made in a branch, into another branch (usually the `main` branch). For example, if you created a branch to try out an experiment and the experiment was a success, Git enables you to merge those changes back into the main branch, or main direction of development, then you can delete your branch as it's no longer needed. Or, if the experiment failed, you can leave the branch and never speak of it again, although it will still be stored if you decide later it is a good idea after all, or you can safely delete the branch.

Similarly, you can merge the changes from`main` into your branch. This is useful if, for example, `main` has had updates made to it since you 'branched off', and you would like to see these updates in your branch.

Git offers another method to merge branches called **rebase**, which is why the Git term for merging branches is **integration** as this covers both methods. Although rebase has some advantages, you can risk losing work when using it, so it won't be covered in this course.

Motivation

Working with other people

Branches enable a team to work on the same files at the same time. However, it doesn't work like other collaboration tools like Google Docs or Microsoft 365, where you can see other users editing parts of a file as you work. Instead, branches allow collaborators to work in isolation, not worrying about what others are up to, and when the time comes to **integrate** changes into the default branch, Git provides the tools to ensure you don't overwrite someone else's work (or, if you do, you do it very intentionally!).

Grouping commits

Even if you're working on your own, branches can be used as a tool to group multiple commits. For example, when working on a new feature for a project, you can create a branch named after your feature, then continue to use Git best practices by **breaking your task into small manageable chunks**, and **complete one thing at a time and commit it**. Then when it comes time to integrate your changes to the main branch, you can choose to "squash" those small commits down into one "merge commit" named after your new feature. This helps to keep the commit history on the default branch minimal and tidy, only showing completed tasks such as new features and bug fixes.

Branching and integrating

This week we are going to learn how to create a branch and make changes to it, then integrate this branch back into the `main` branch. This is particularly helpful when working collaboratively.

In Git, a branch represents a distinct version of the `main` repository, serving as a snapshot of your changes. When implementing new features or addressing issues, regardless of their scope, creating a new branch is the practice to isolate your modifications. This approach enhances code stability in the main branch and affords you the opportunity to refine your history before merging it into the primary codebase.

When the work is complete, a branch can be merged with the main project. You can even switch between branches and work on different projects without them interfering with each other.

We are also going to learn about Git's stash feature, and how it can help you when switching between branches.

Create a new local Git repository

Let's create a new folder on our computer called 'tutorial3' and initialise Git:

Command-line

Create a directory for your 'tutorial3' work:

```
mkdir tutorial3
cd tutorial3
```

and run git init inside:

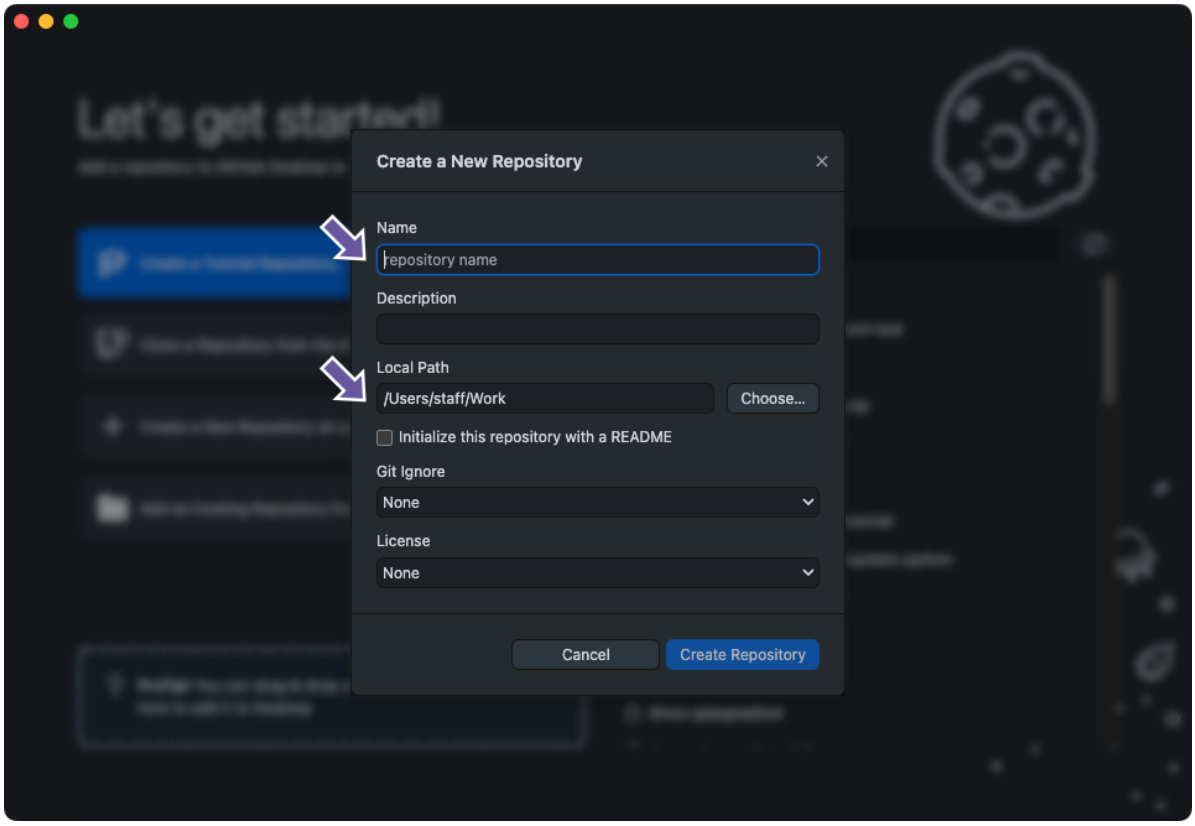
```
git init
```

```
Initialized empty Git repository in /Users/staff/Work/tutorial3/.git/
```

GitHub Desktop

Choose `File > New Repository...` from the menu.

In the "Create a New Repository" form, name the repository "tutorial3", set the "Local Path" field to your preferred location and click the "Create Repository" button:



Add the following file to the repository:

example.md

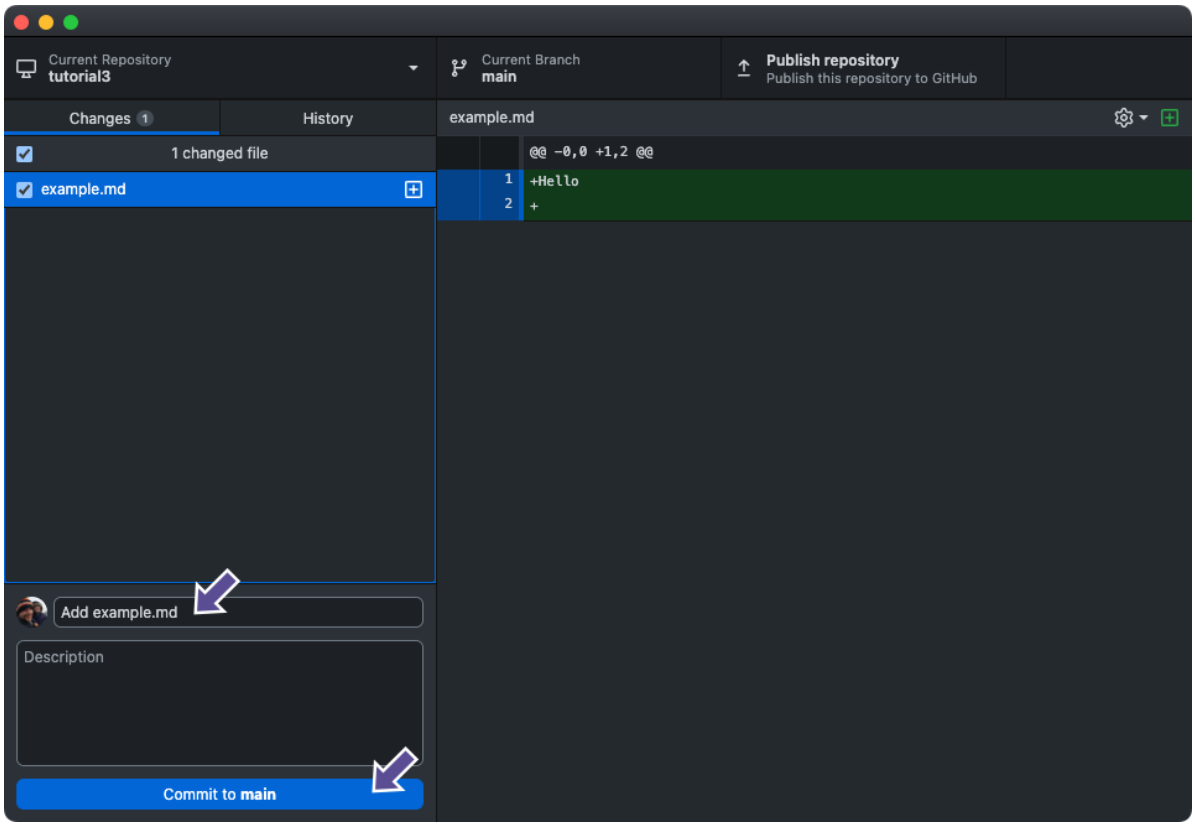
```
Hello
```

Then add and commit the file:

Command-line

```
git add example.md
git commit -m "Add example.md"
```

GitHub Desktop



Next, let's create a branch named `shout` based on this commit.

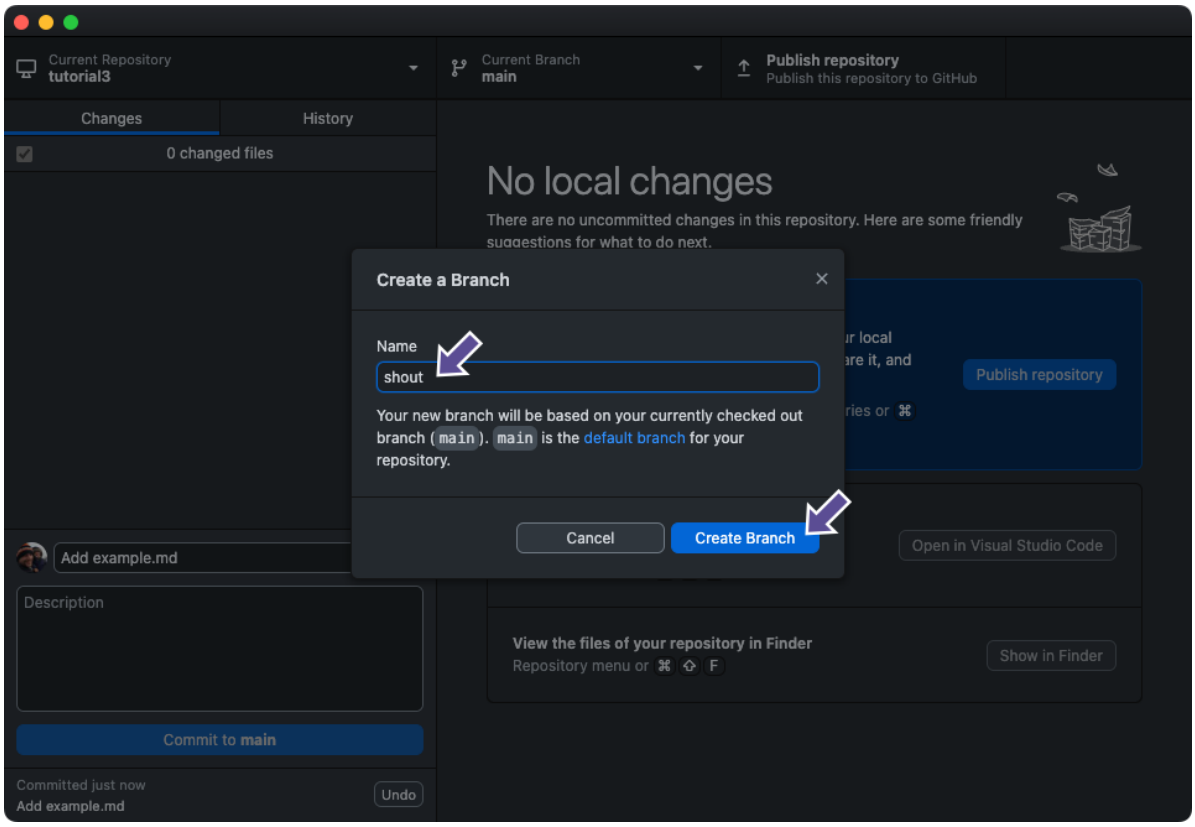
Command-line

We can create a new branch and switch to it using the switch command with the -c flag:

```
git switch -c shout
```

```
Switched to a new branch 'shout'
```

GitHub Desktop



Add an exclamation mark to the end of the word within the `example.md` file:

> example.md
>
> Hello!

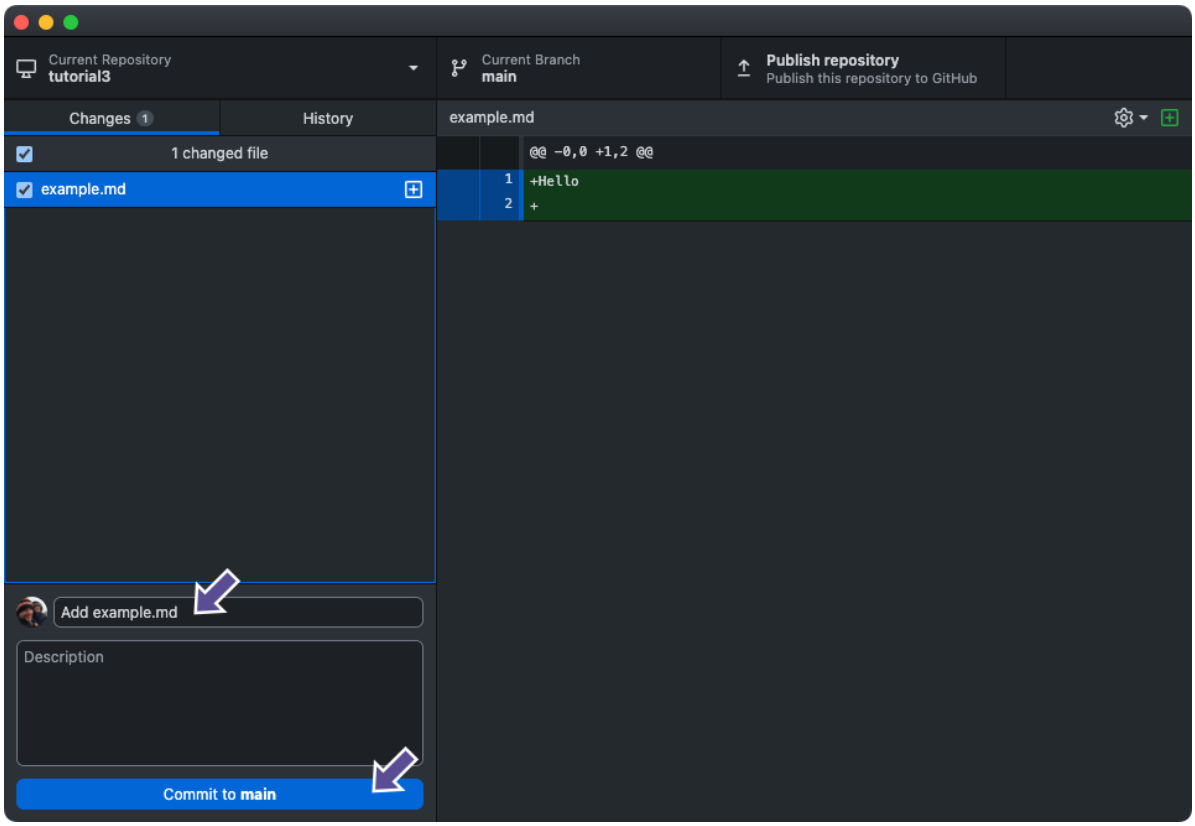
Then commit the change:

Command-line

```
git add example.md
```

```
git commit -m "Add exclamation mark"

[shout eed4222] Add exclamation mark

1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop



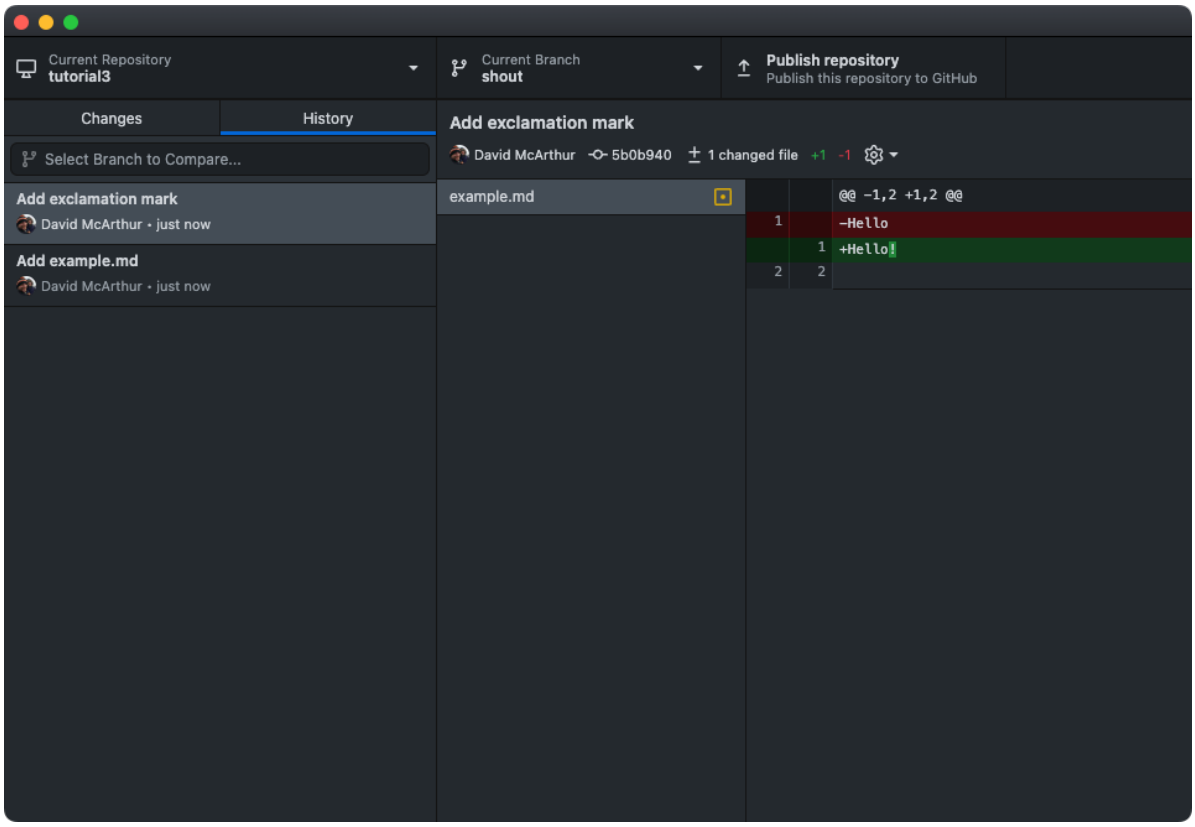
Let's take a quick look at the log:

Command-line

```
git log --all --decorate --oneline --graph
```

```
* eed4222 (HEAD -> shout) Add exclamation mark

* 8f601cd (main) Add example.md
```

GitHub Desktop



Now let's try merging the branch back into the `main` branch. First, switch back to `main`:

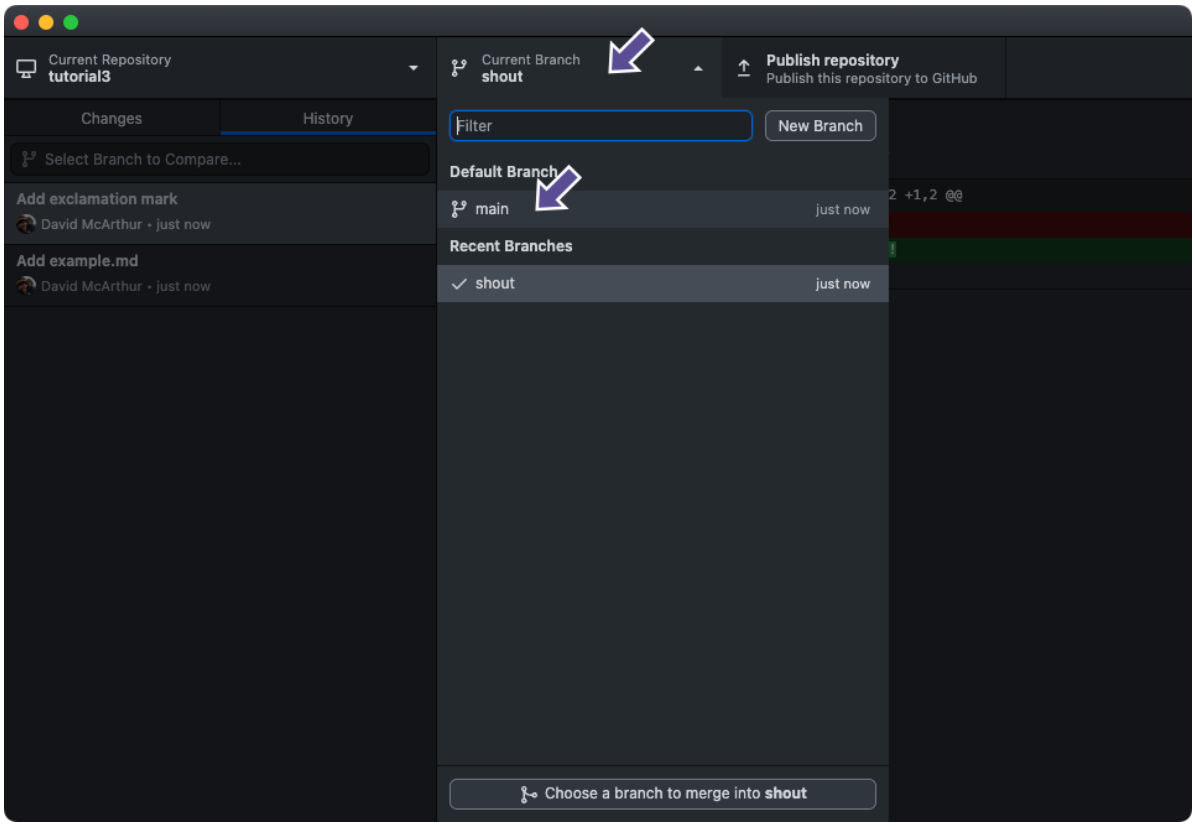
Command-line

```
git switch main
```

```
Switched to branch 'main'
```

(Note, we don't include the -c flag as this branch already exists)

GitHub Desktop



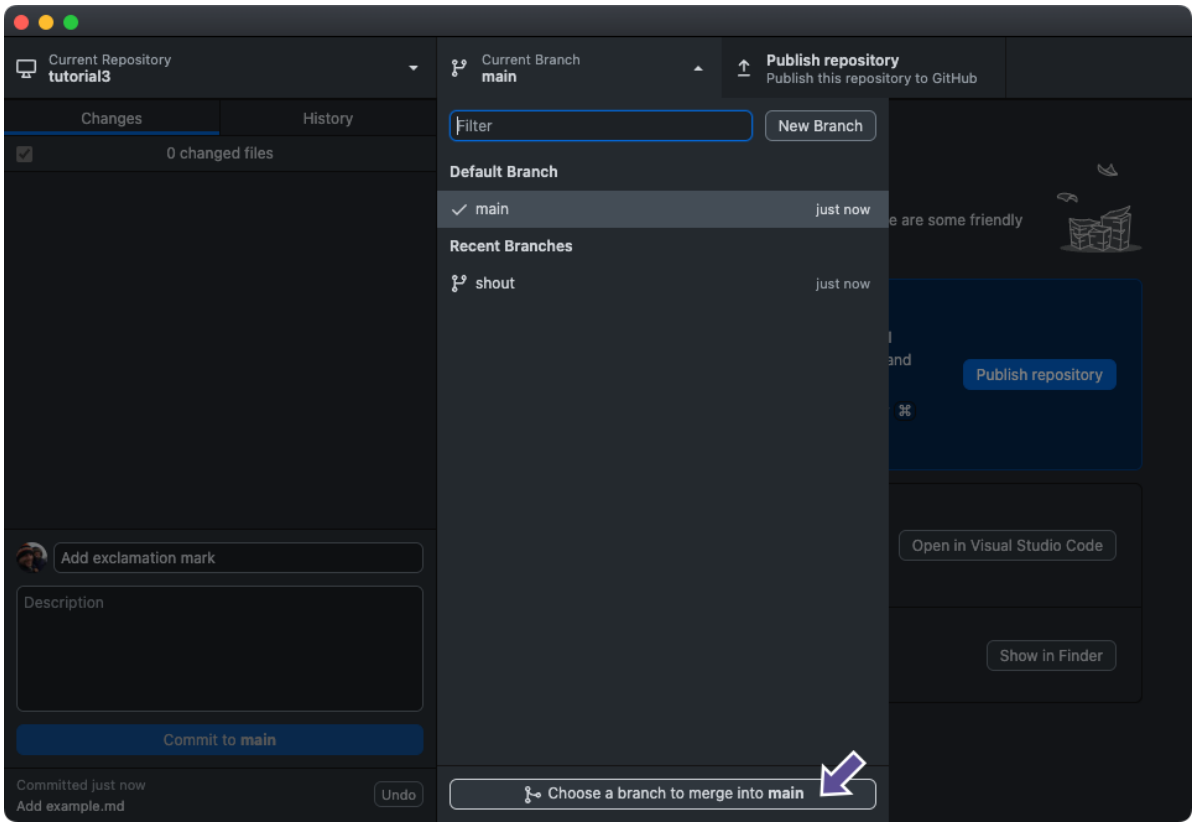
Then merge our `shout` branch into `main`:

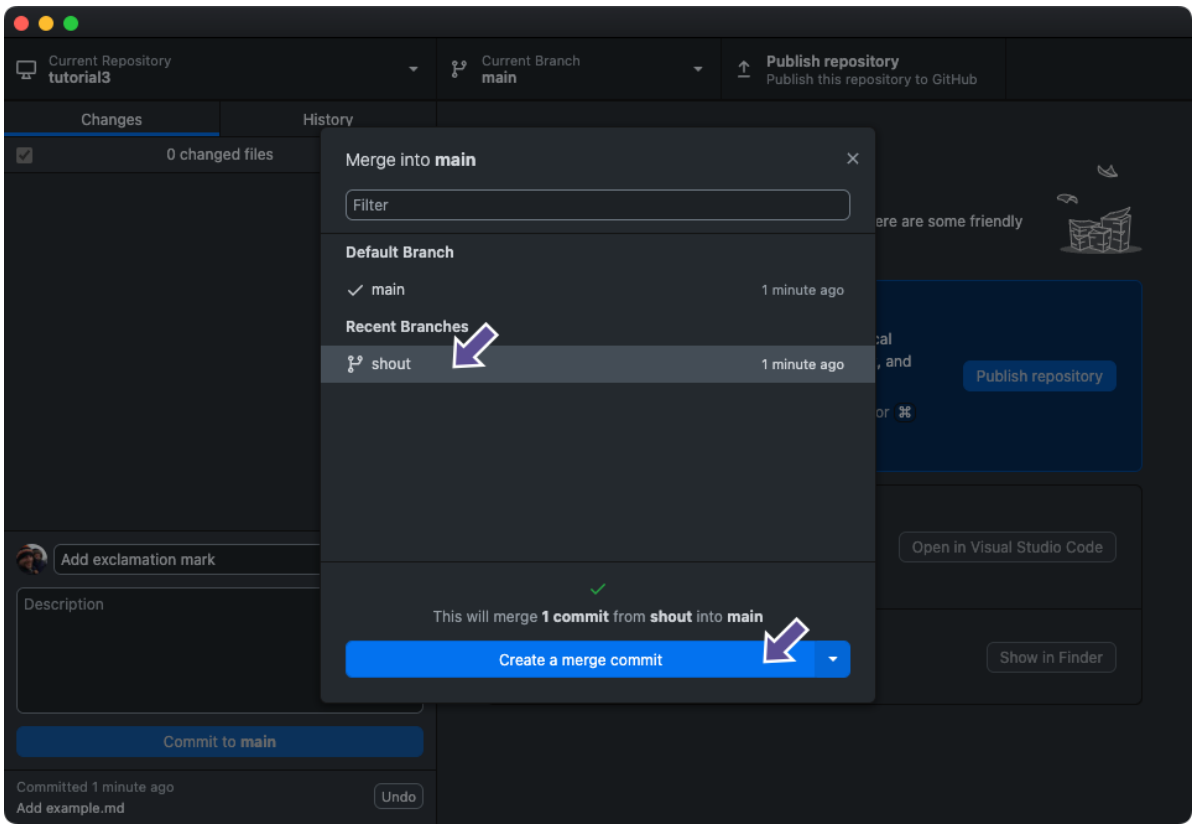
Command-line

```
git merge shout
```

```
Updating 8f601cd..eed4222

Fast-forward

example.md | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop





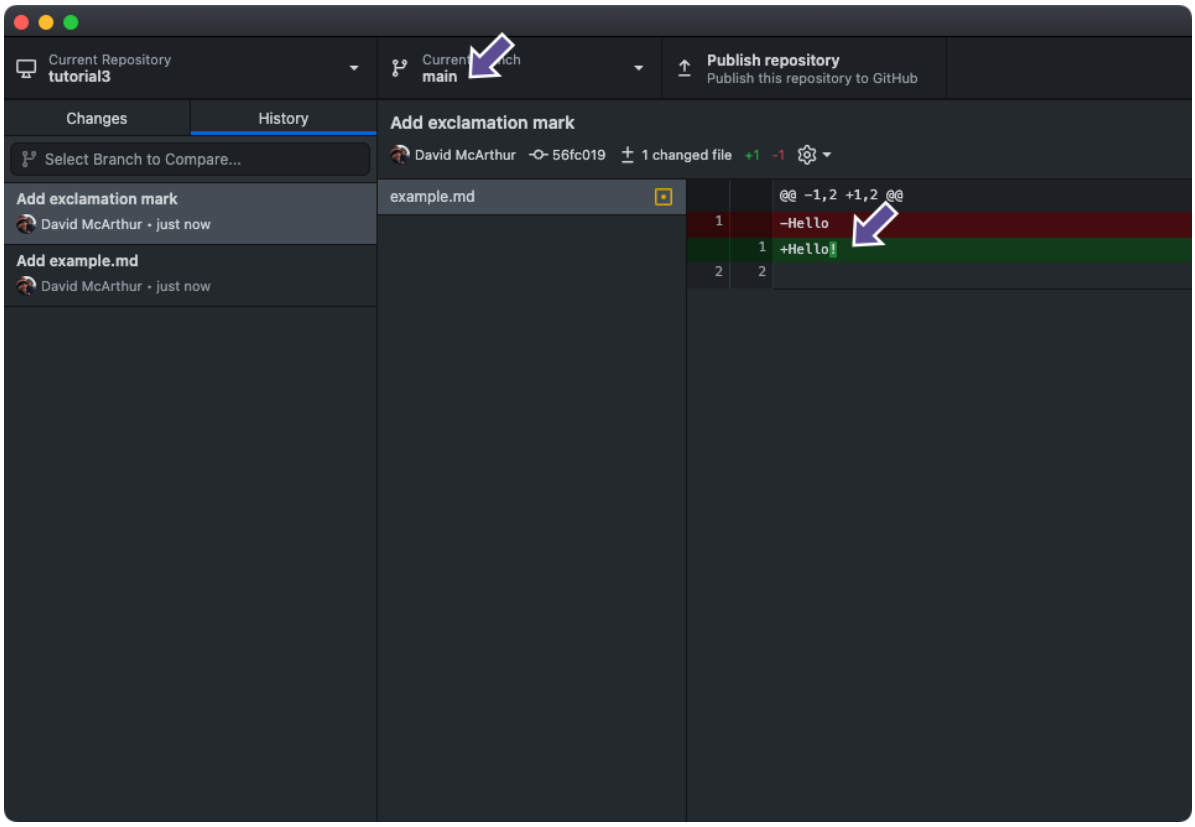
We can check that the version of our file in the main branch now has the exclamation mark:

Command-line

```
cat example.md
```

```
Hello!
```

GitHub Desktop



As the change is now reflected in our `main` branch, we can safely delete our `shout` branch:

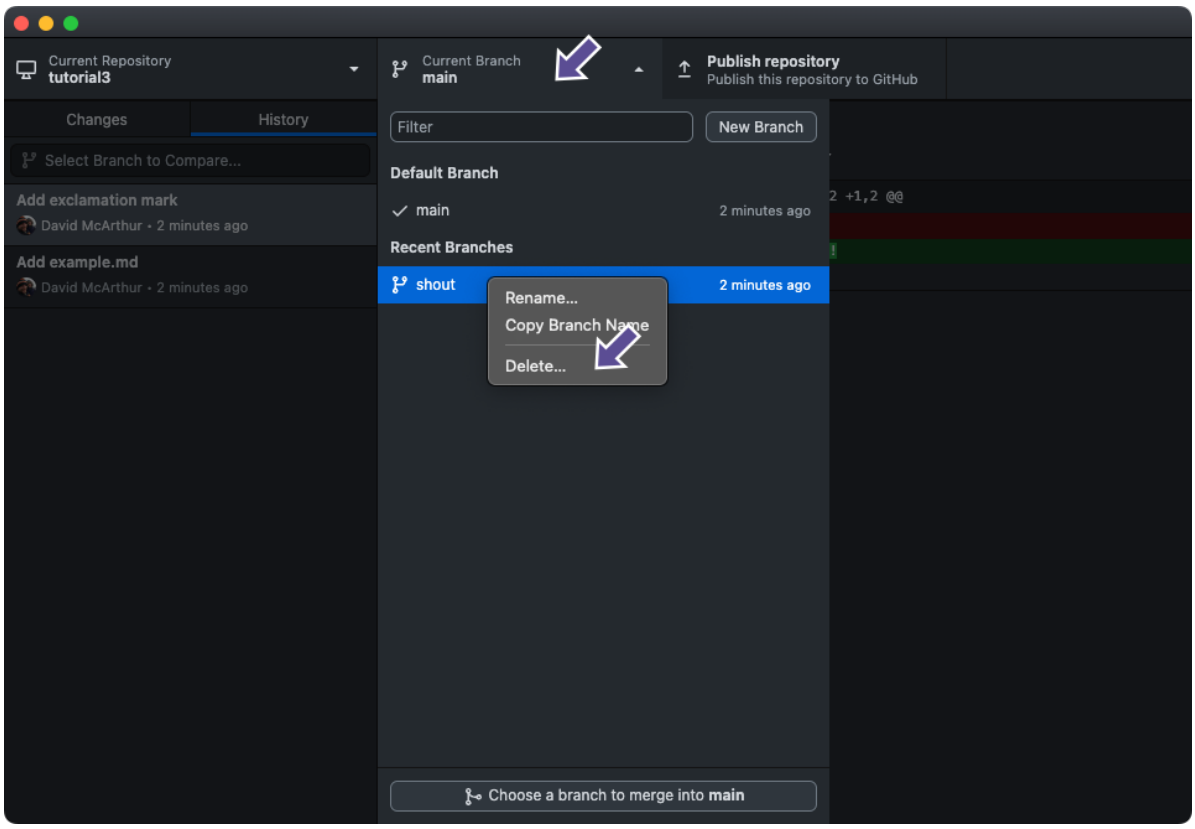
Command-line

```
git branch --delete shout
```

```
Deleted branch shout (was eed4222).
```

GitHub Desktop

Simply right click on the `shout` branch here and select Delete



Switching branches with the help of Git stash

Given the example above, branching with Git appears straightforward. However, this is not always the case. Sometimes Git won't let you switch branches. Let's create a scenario where we see an error when switching branches, try to understand why Git is stopping us, and look at a simple universal solution.

Create and switch to a new branch called `question`:

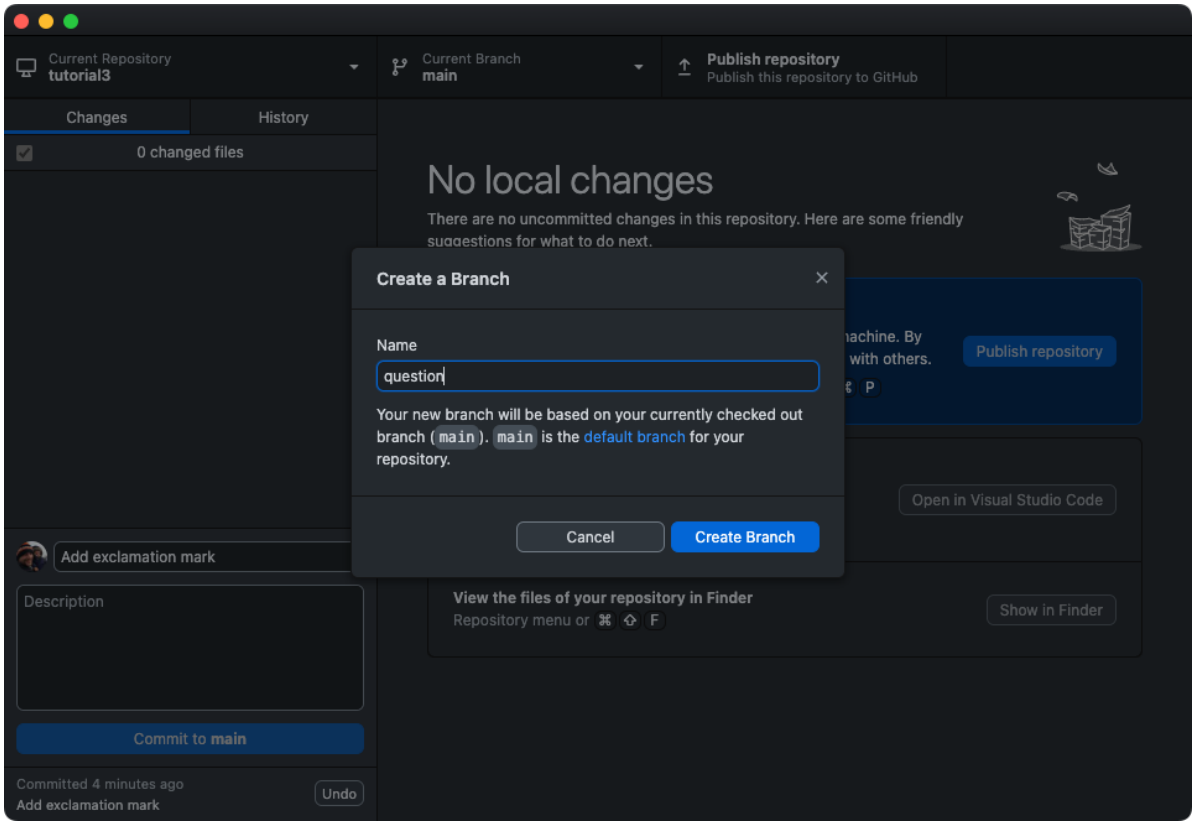
Command-line

```
git switch -c question
```

```
Switched to a new branch 'question'
```

GitHub Desktop

From the main menu select Branch > New Branch and name the branch 'question':



Let's make a change to the file by changing the exclamation mark to a question mark:

> example.md
>
> Hello?

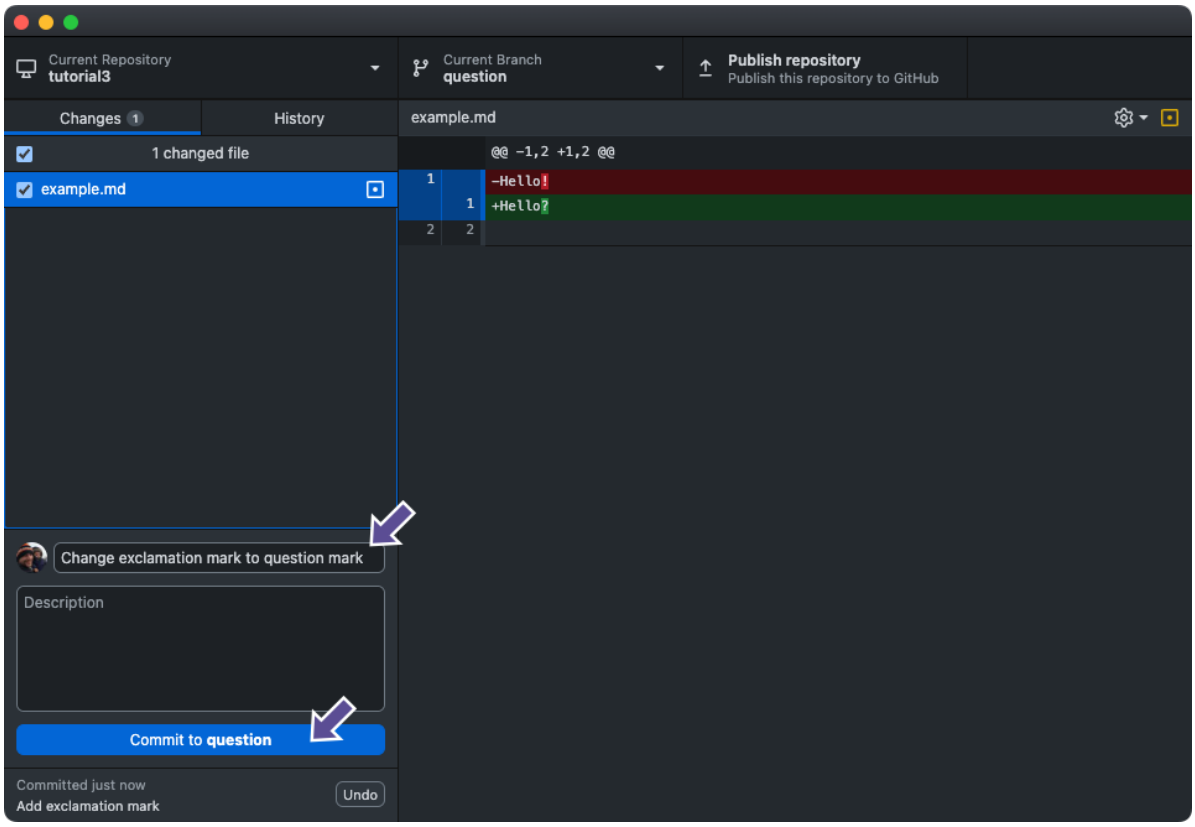
Add the change to the stage and commit:

Command-line

```
git add example.md
git commit -m "Change exclamation mark to question mark"
```

```
[question 49806f4] Change exclamation mark to question mark

1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop



Now let's make another change to our file:

> example.md
>
> Hello?
> Is it me you're looking for?

This time, instead of committing, we will try to switch back to the main branch, and Git will tell us we can't switch branches currently as we would lose our local changes (see the next example for the error). Committing our latest change would let us then switch the branch, but what if we don't want to commit the change?

Command-line

A common scenario is that you have started working on a problem, then when you check Git you realise you are working on the wrong branch, but now you have uncommitted changes and can't switch branches. For this scenario, there is the **stash**. Think of it as a clipboard with cut-and-paste functionality like you would have in a text editor, which will let us 'cut' our changes and store them somewhere safe, let us switch branches, and then paste them.

Let's try it now:

```
git switch main
```

```
error: Your local changes to the following files would be overwritten by checkout:

example.md

Please commit your changes or stash them before you switch branches.

Aborting
```

Git complains that we will lose our local changes. Let's try out the stash:

```
git stash
```

```
Saved working directory and index state WIP on question: 49806f4 Change exclamation mark to
```

Git has used the abbreviation 'WIP' which stands for 'Work In Progress'.

Now if we look at our file we'll see the 2nd line has disappeared:

```
cat example.md
```

```
Hello?
```

We can ask Git to show us our stash:

```
git stash show
```

```
example.md | 2 ++

1 file changed, 2 insertions(+)
```

This shows us which files have changed. We can add a `-p` (for 'patch') flag which will show us a diff:

```
git stash show -p
```

```
diff --git a/example.md b/example.md

index ebbc6cd..472b4c5 100644

--- a/example.md

+++ b/example.md

@@ -1,2 +1,4 @@

Hello?

+Is it me you're looking for?

+
```

Now our local copy is 'clean' and we can safely switch branches:

```
git switch main
```

```
Switched to branch 'main'
```

And we can apply our change to a different branch:

```
git stash apply
```

```
Auto-merging example.md

On branch main

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified:   example.md

no changes added to commit (use "git add" and/or "git commit -a")
```

And now if we look at the file, we'll see our stashed change has been applied, but there is no question mark on "Hello?" as this change still resides on the `question` branch:

```
cat example.md
```

```
Hello!

Is it me you're looking for?
```

We can now safely remove our stashed version:

```
git stash drop
```

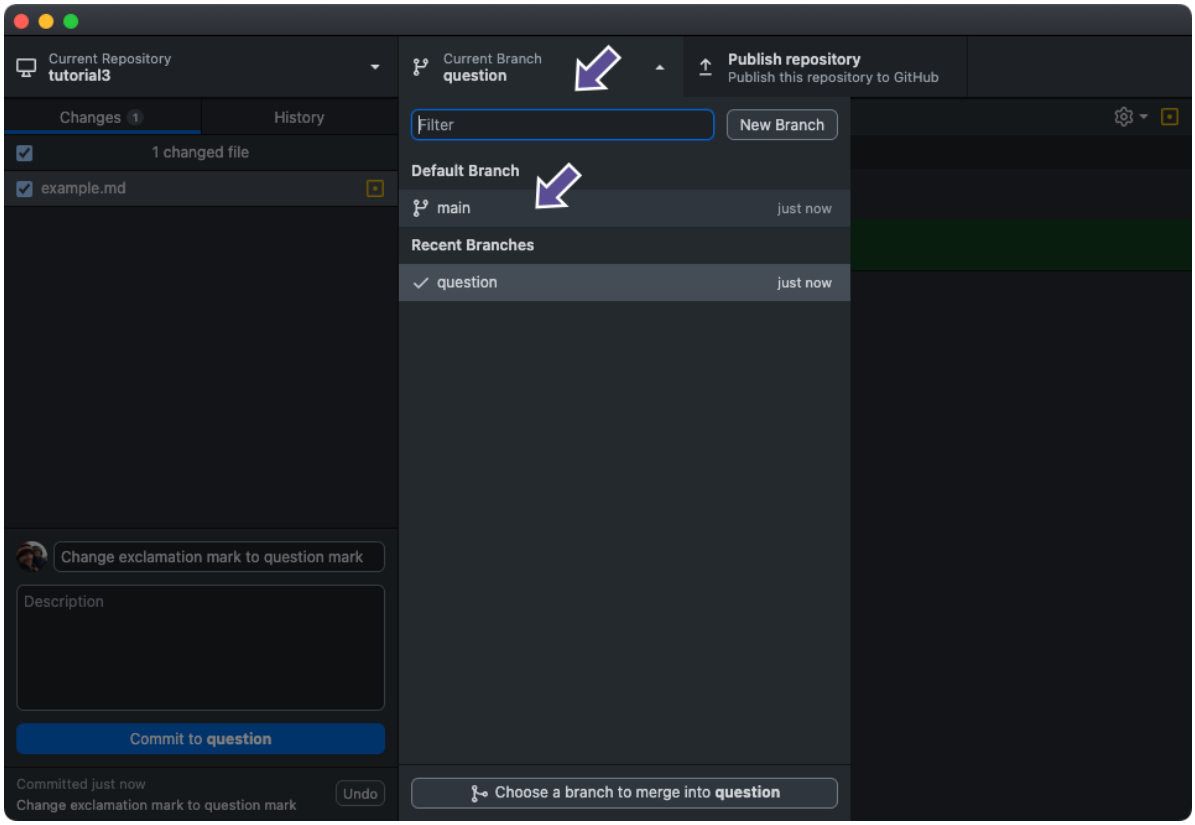
```
Dropped refs/stash@{0} (00b8825ce0b6a0e3c2877f3c42c828fb4da4c922)
```

It's important to `drop` the stash afterwards, or you'll end up with several items in the stash that could lead to applying the wrong one. Git has a shortcut for 'apply then drop' which is `git stash pop`.

GitHub Desktop

Luckily for GitHub Desktop users, this stashing and stash-applying process is all handled by clicking a button.

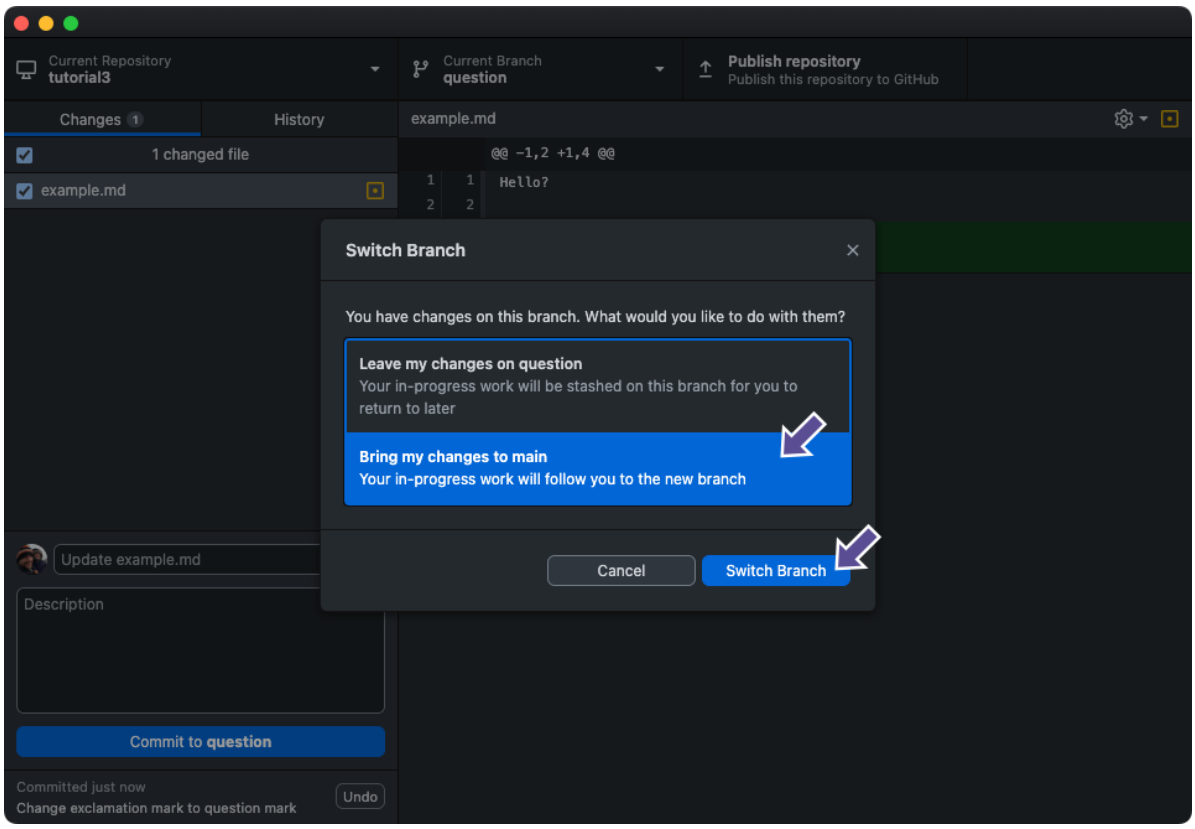
Once we try to switch to the `main` branch:



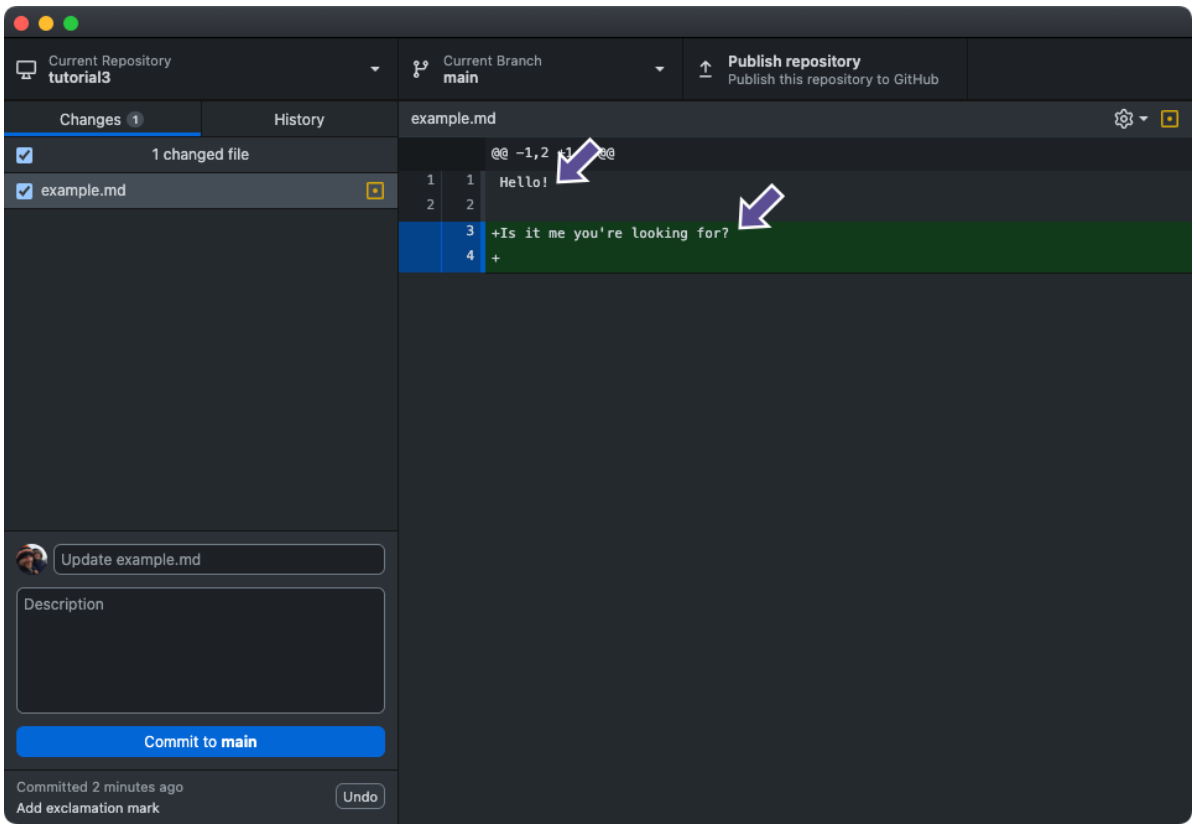
GitHub Desktop offers us two options:

- **Leave my changes on question** allows us to stash the changes and come back to the branch later and apply the stash
- **Bring my changes to main** automates the process of stashing the changes, switching to the desired branch, and applying the stash.

For this tutorial, we will choose the latter option— 'Bring my changes to main':



And now if we look at the file, we'll see our stashed change has been applied, but no question mark as this change still resides on the `question` branch:



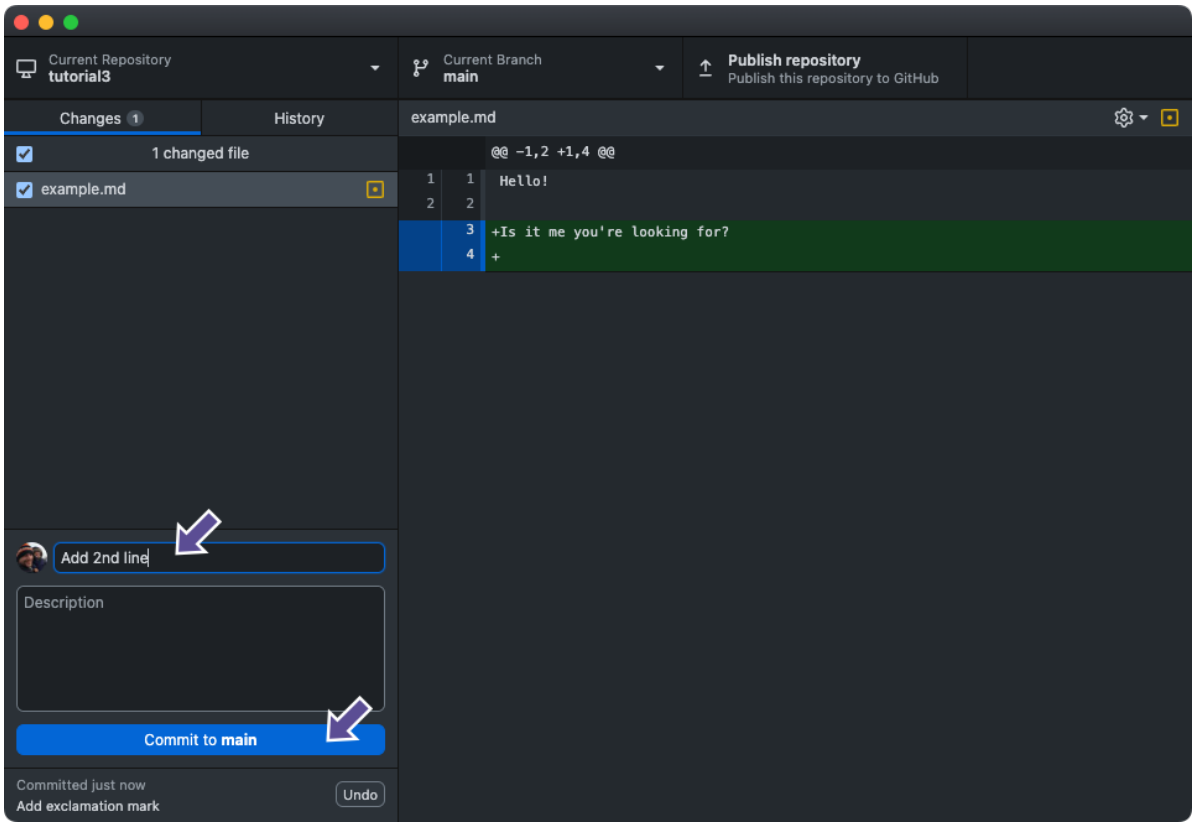
Let's stage and commit our 2nd line:

Command-line

```
git add example.md
```

```
git commit -m "Add 2nd line"
[main 6859b82] Add 2nd line
1 file changed, 2 insertions(+)
```

GitHub Desktop



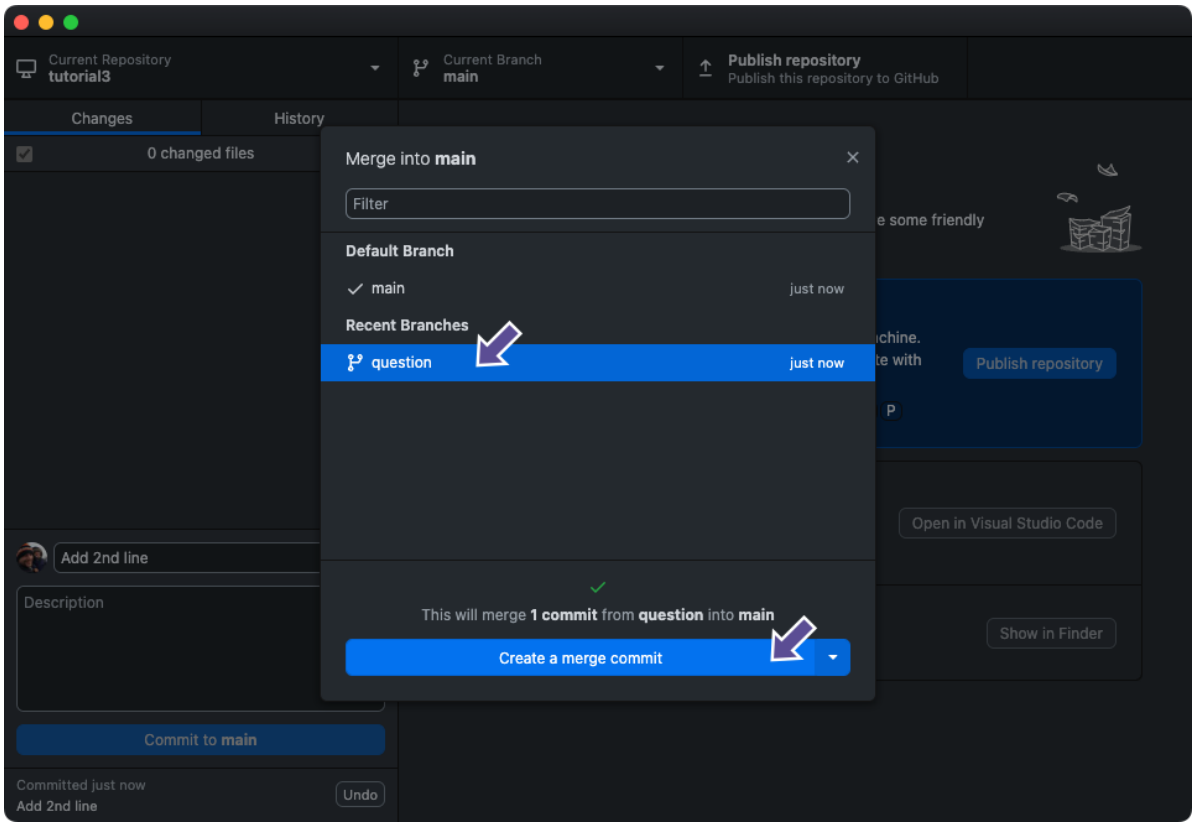
Let's merge in the question mark.

Command-line

```
git merge question
```

```
Auto-merging example.md

Merge made by the 'ort' strategy.

example.md | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop



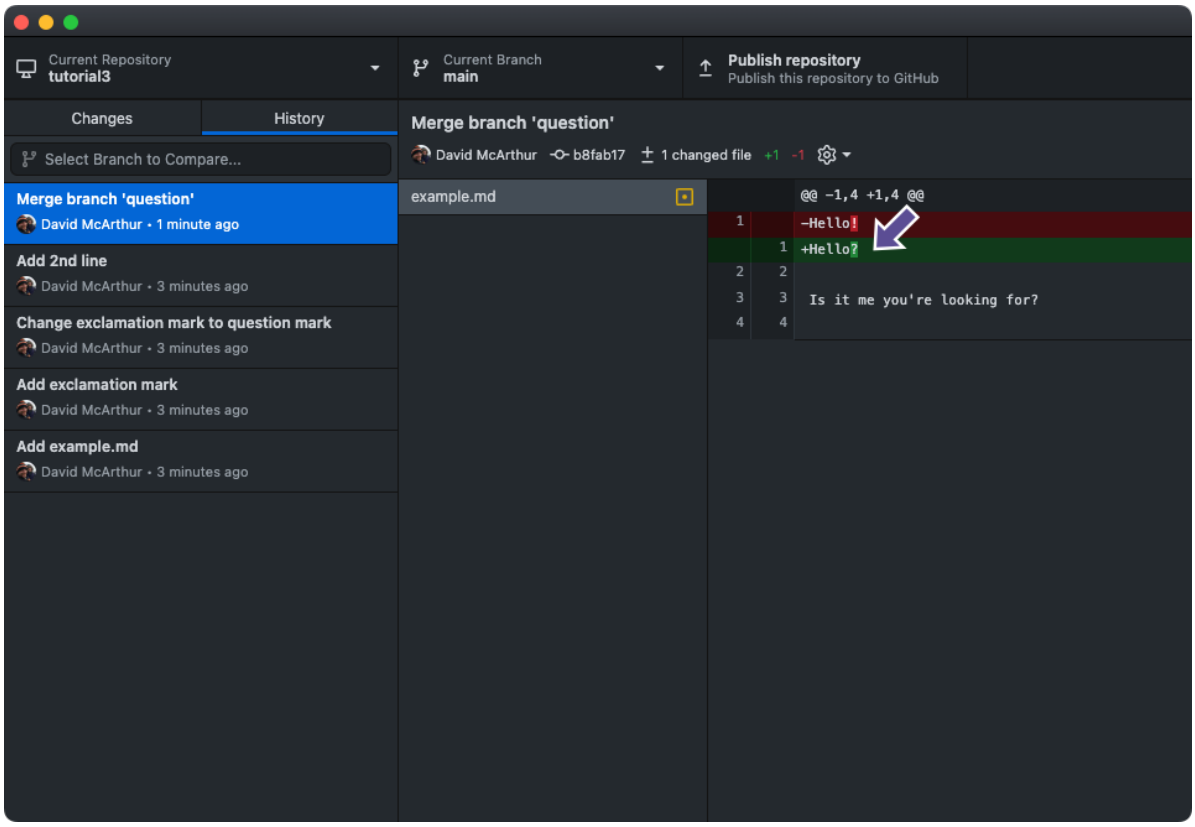
Let's check the file:

Command-line

```
cat example.md
```

```
Hello?

Is it me you're looking for?
```

GitHub Desktop



Merge conflicts

In the examples above we have introduced create a branch, making some changes and then integrating this back into main. However, what happens if you make a change to a line in a file on one branch, then you (or someone else) changes the same line of the same file on a different branch, what should Git do when you want to merge? This is a so-called **merge conflict**.

In this section, we're going to create a simple scenario where we will run into **merge conflicts** and explore two different ways to resolve them.

When using `merge` to combine two branches, Git will first find a common "base commit" between the two branches, then compare the changes since this commit and attempt to compile a change list known as a "merge commit".

Git tries to automate this process as much as possible, but if it finds two conflicting changes it won't try to guess which change should overrule the other. This usually happens when the same line in a file has been changed in both branches. In this case, while attempting to

compile the merge commit, Git will go into a "merge conflict" state, which involves updating the conflicting files in the filesystem (or "working tree") with some special formatting to show you what it cannot guess, and Git will not allow you to commit the merge until you have manually decided what you want to happen.

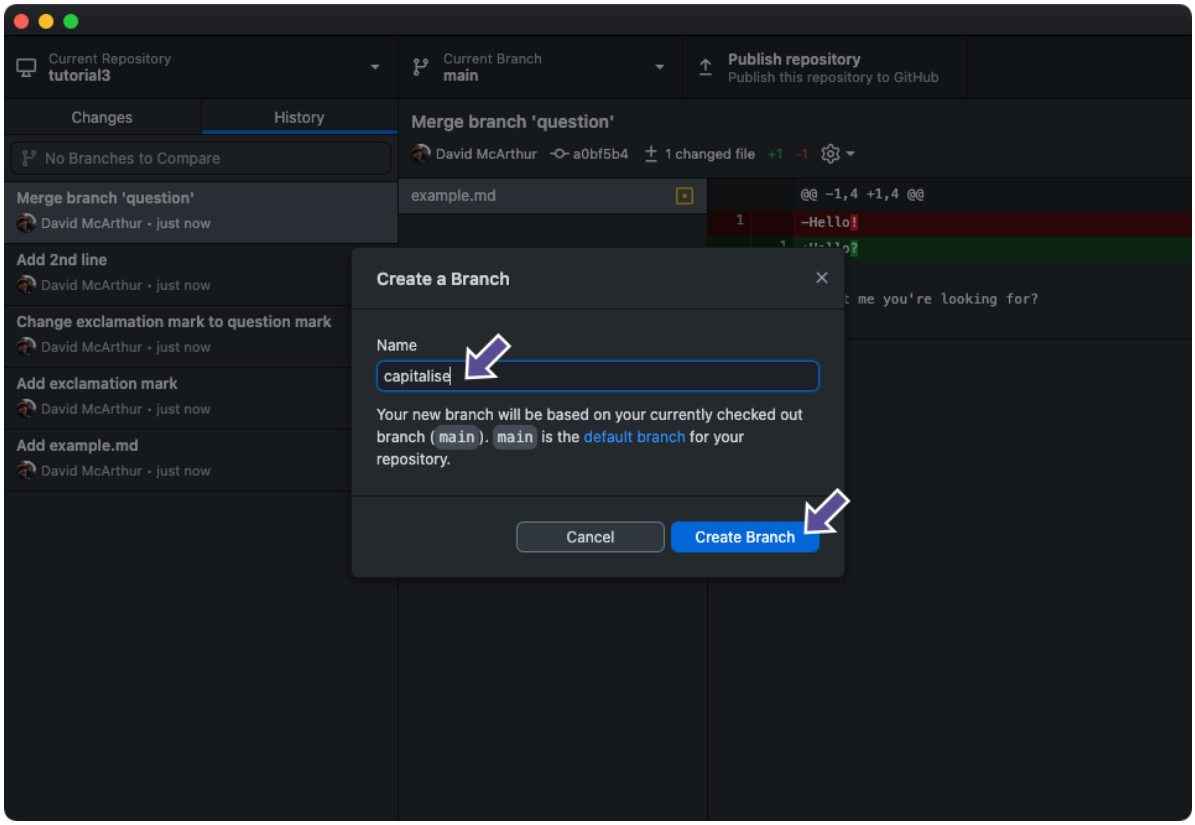
Let's create a branch named `capitalise` and switch to it:

Command-line

```
git switch -c capitalise
```

```
Switched to a new branch 'capitalise'
```

GitHub Desktop



Now we're going to make a different change to the file on the main branch. This time change the contents to the following:

example.md

HELLO

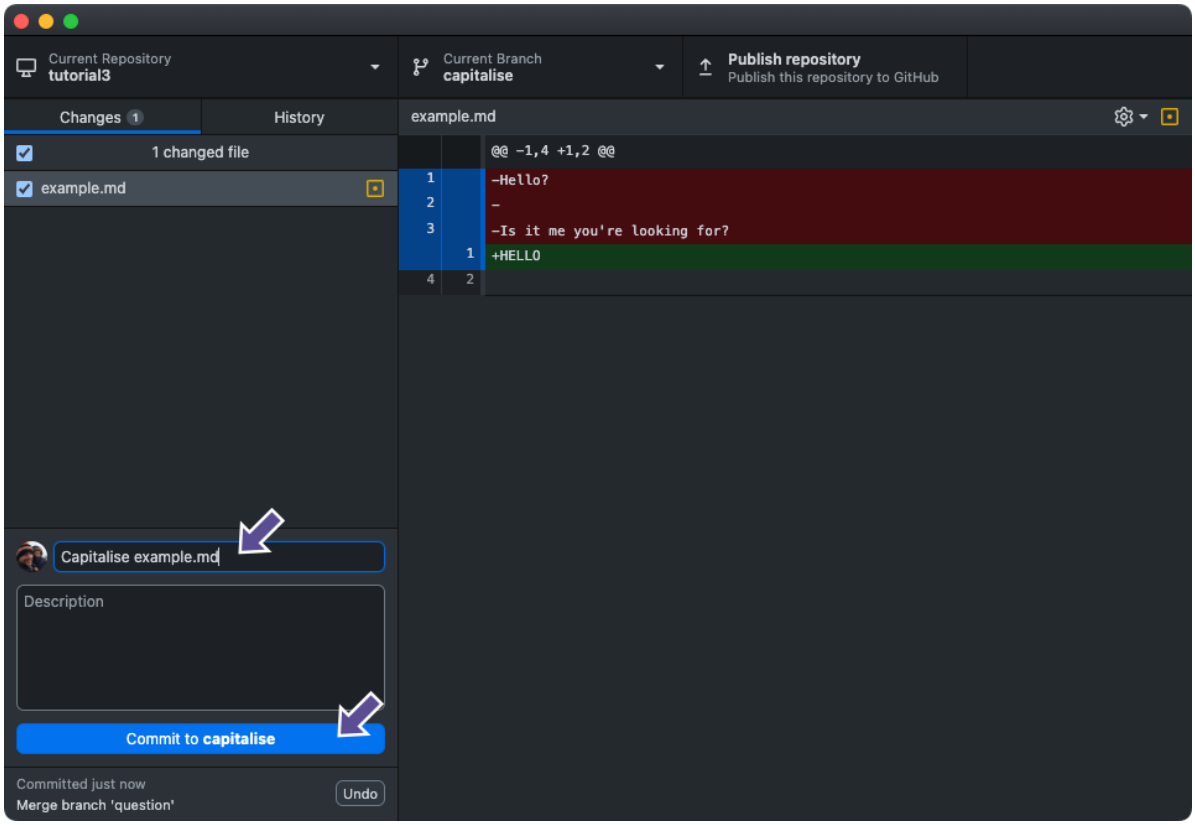
Then commit the change:

Command-line

```
git add example.md
git commit -m "Capitalise example.md"
```

```
[capitalise 3da9dfb] Capitalise example.md

1 file changed, 1 insertion(+), 3 deletions(-)
```

GitHub Desktop



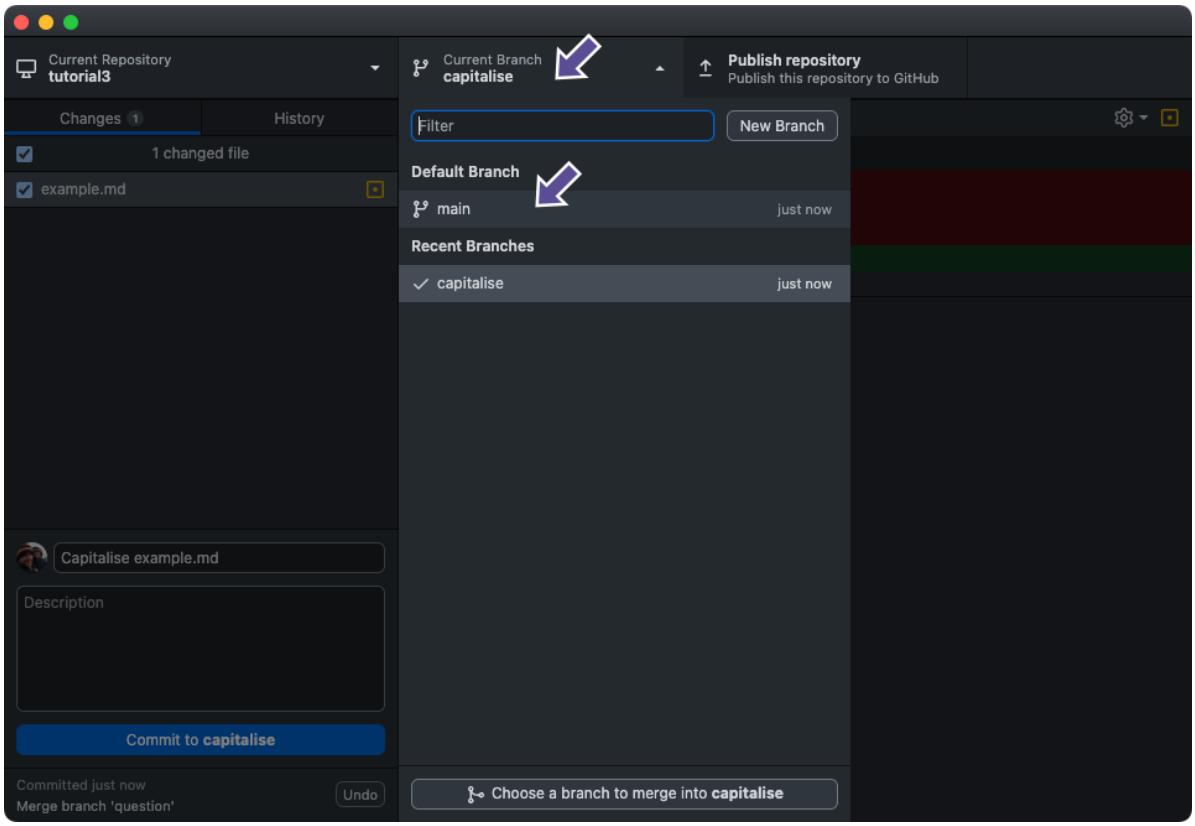
Now switch back to the `main` branch:

Command-line

```
git switch main
```

```
Switched to branch 'main'
```

GitHub Desktop



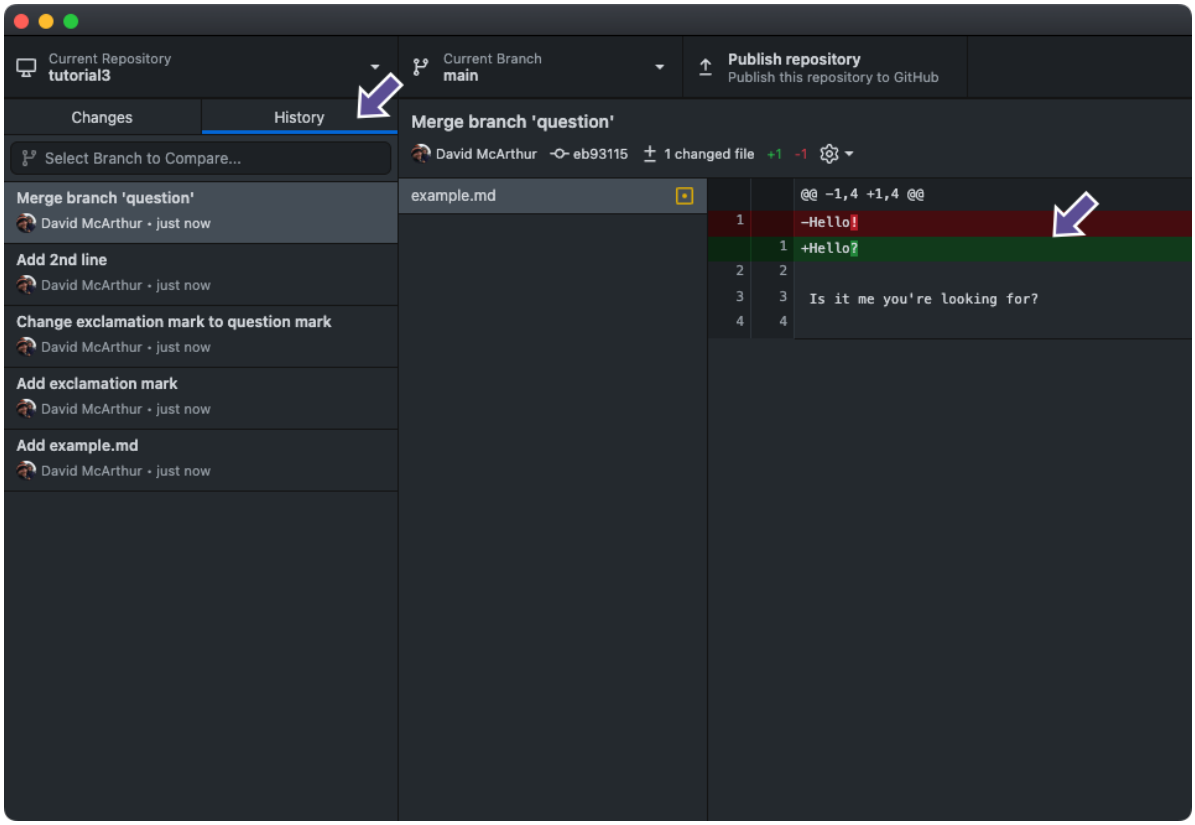
The file should look like it did before the capitalisation, as that change is isolated on the `capitalise` branch:

Command-line

```
cat example.md
```

```
Hello?

Is it me you're looking for?
```

GitHub Desktop



Now we're going to make a different change to the file on the main branch. This time change the contents to the following:

> example.md
>
> Hello.

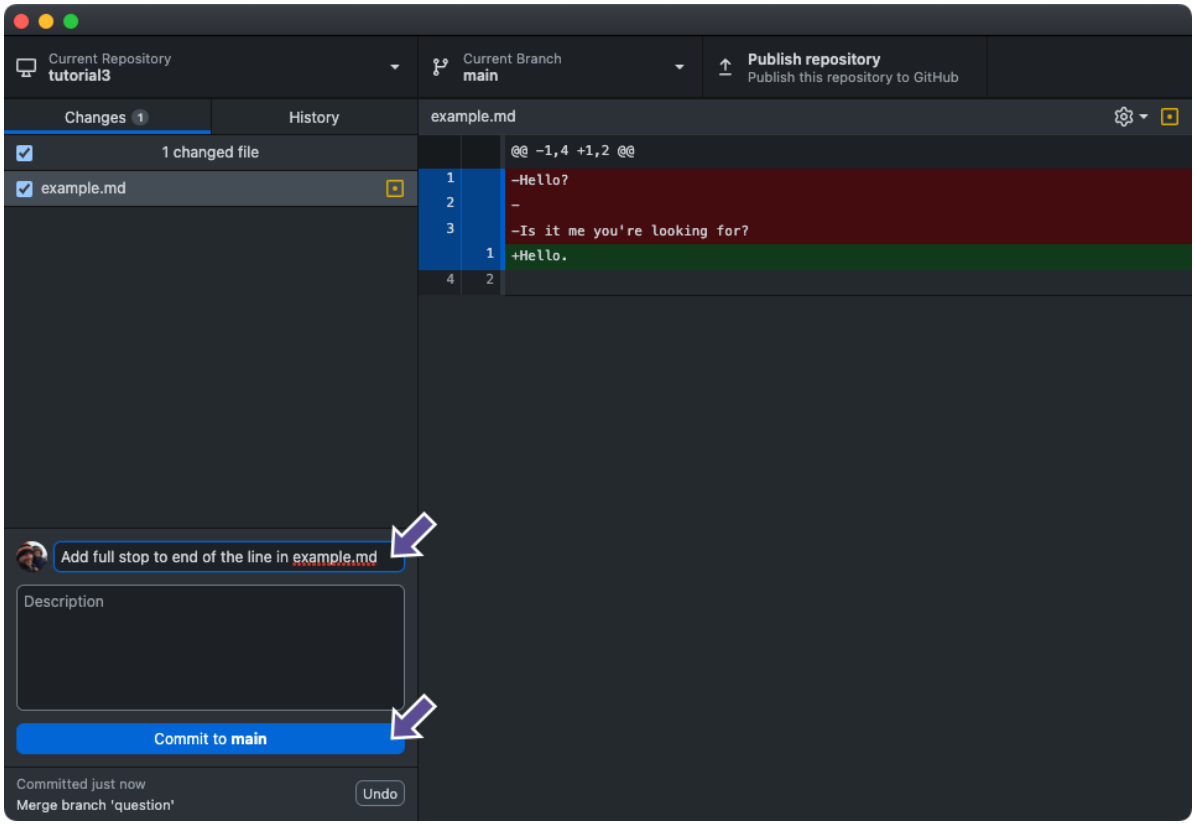
Stage and commit the change:

Command-line

```
git add example.md
git commit -m "Add full stop to end of the line in example.md"
```

```
[main 7888fef] Add full stop to end of the line in example.md

1 file changed, 1 insertion(+), 3 deletions(-)
```

GitHub Desktop



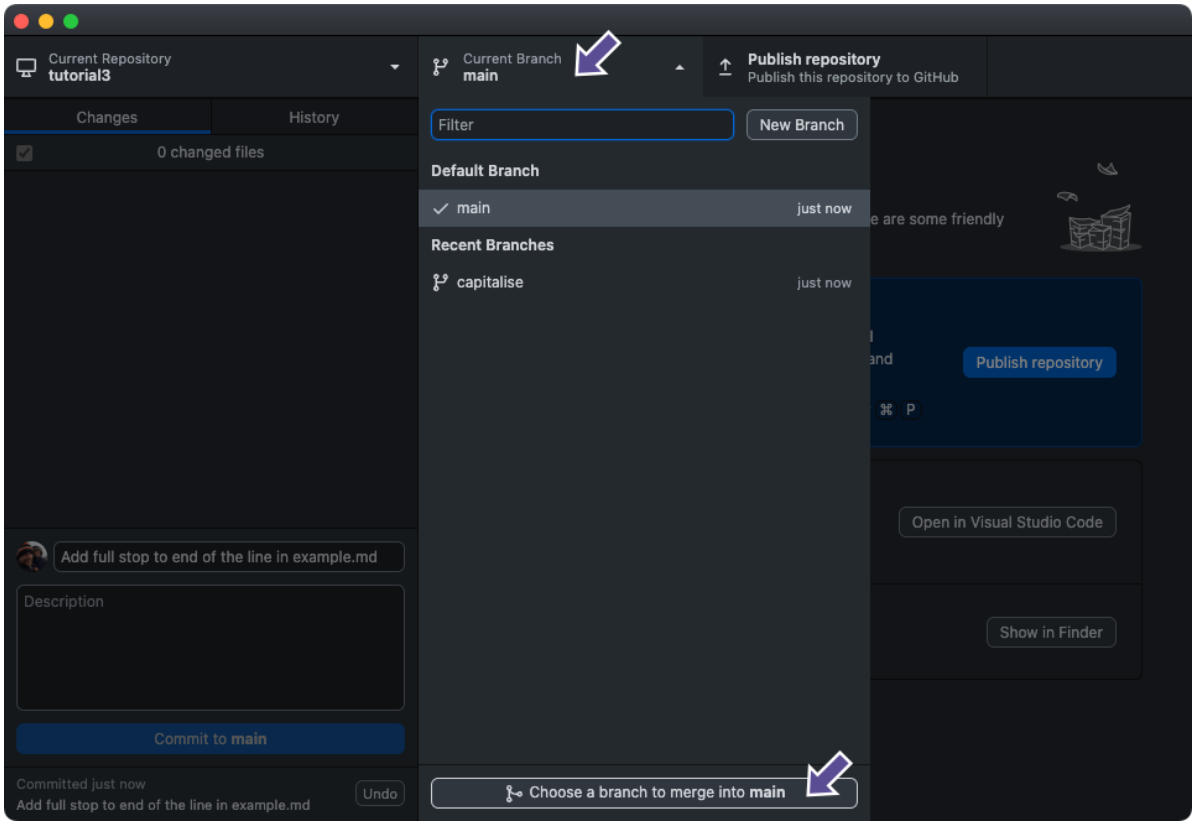
Let's see what happens when we try to merge:

Command-line

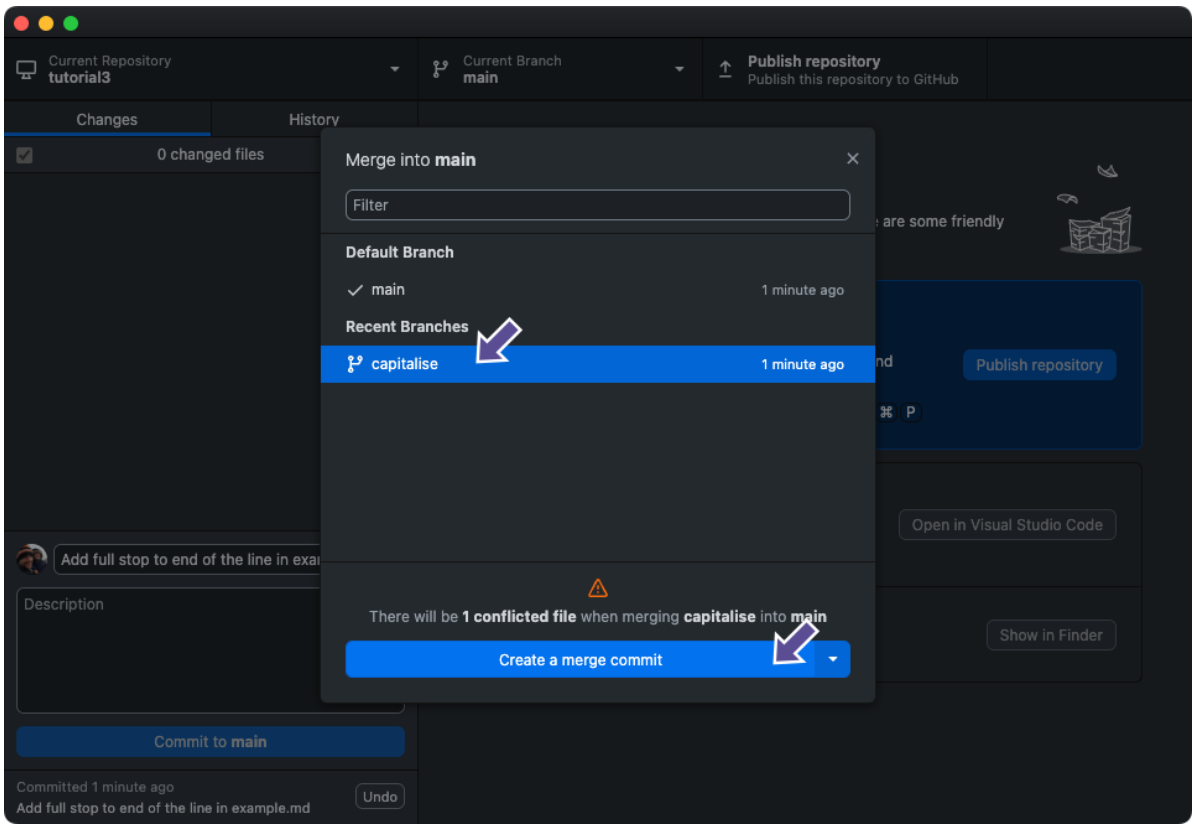
```
git merge capitalise
```

```
Auto-merging example.md

CONFLICT (content): Merge conflict in example.md

Automatic merge failed; fix conflicts and then commit the result.
```

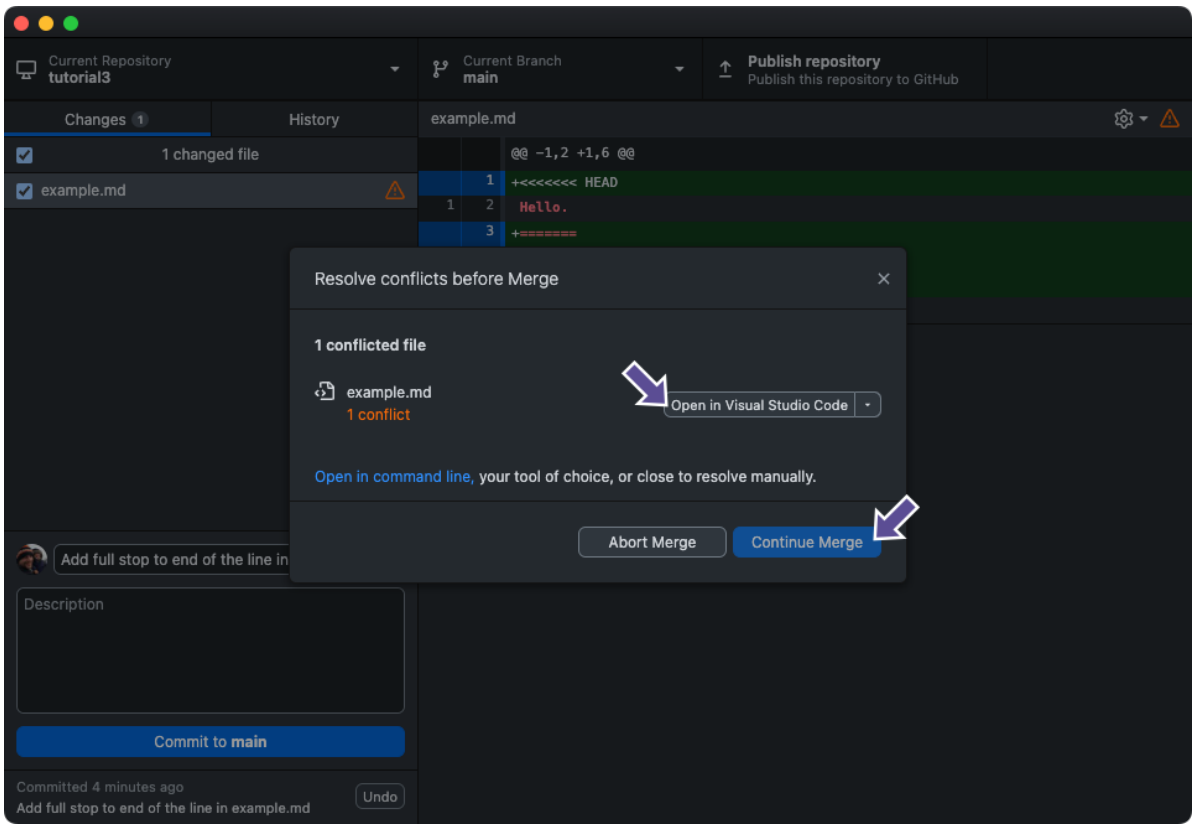
GitHub Desktop



Let's see what happens when we try to merge:



Next, GitHub Desktop will tell you which files need attention, and instruct you to open them in your preferred code editor.



The Continue Merge button is disabled, as we need to resolve the conflicts in our files.

Let's open up `example.md` in our preferred editor and see what is happening:

```
<<<<<<< HEAD

Hello.

=======

HELLO

>>>>>>> capitalise
```

Git has modified our file to delineate a change that it cannot merge. The line (or lines) between `<<<<<<<` and `======` here show what you already had (you can tell because HEAD points to your current branch). The line (or lines) between `=======` and `>>>>>>>` is what was introduced by the other commit, in this case from the capitalise branch.

If this was a large file with multiple lines changed, Git will automatically merge the parts that it can, and will highlight the parts it can't like this, and we will have to go through each conflict and manually resolve them.

Let's have a look at Git's status:

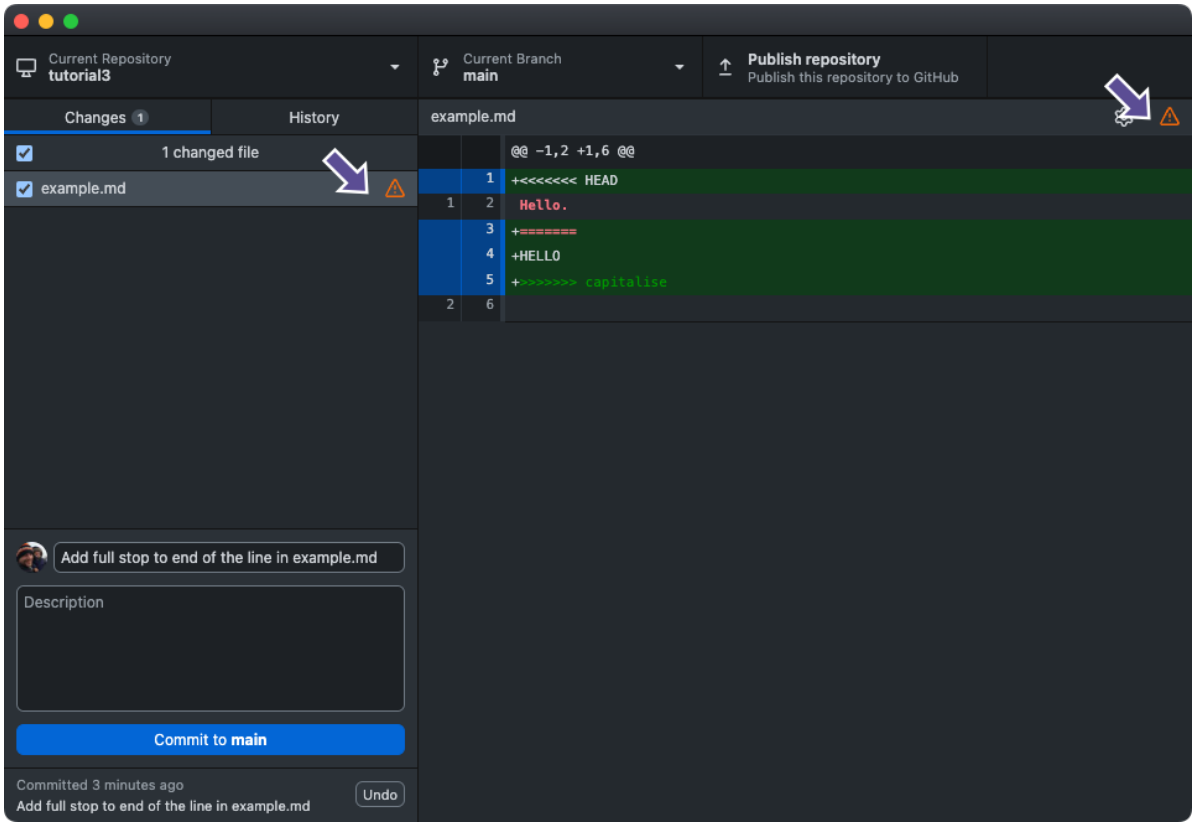
Command-line

```
git status -v
```

```
On branch main

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified:   example.md

* Unmerged path example.md

no changes added to commit (use "git add" and/or "git commit -a")
```

GitHub Desktop

Note that GitHub Desktop displays the conflict icon besides each file that needs attention:



Git's method for displaying merge conflicts by modifying the source files makes it very straightforward for us to resolve, by just manually editing the files as we want. However Git knows that there is now a danger that you could commit this merge conflict syntax, and when working with code, this will almost certainly cause a syntax error and break your project. So Git tries to protect us against this by going into a "merge conflict" state, where it wants you to either resolve or abort this action before you do anything else.

Let's manually edit our file to what we want, like a mixture of both changes:

> example.md
>
> HELLO.

Command-line

Add the file to the stage to mark it as resolved:

```
git add example.md
```

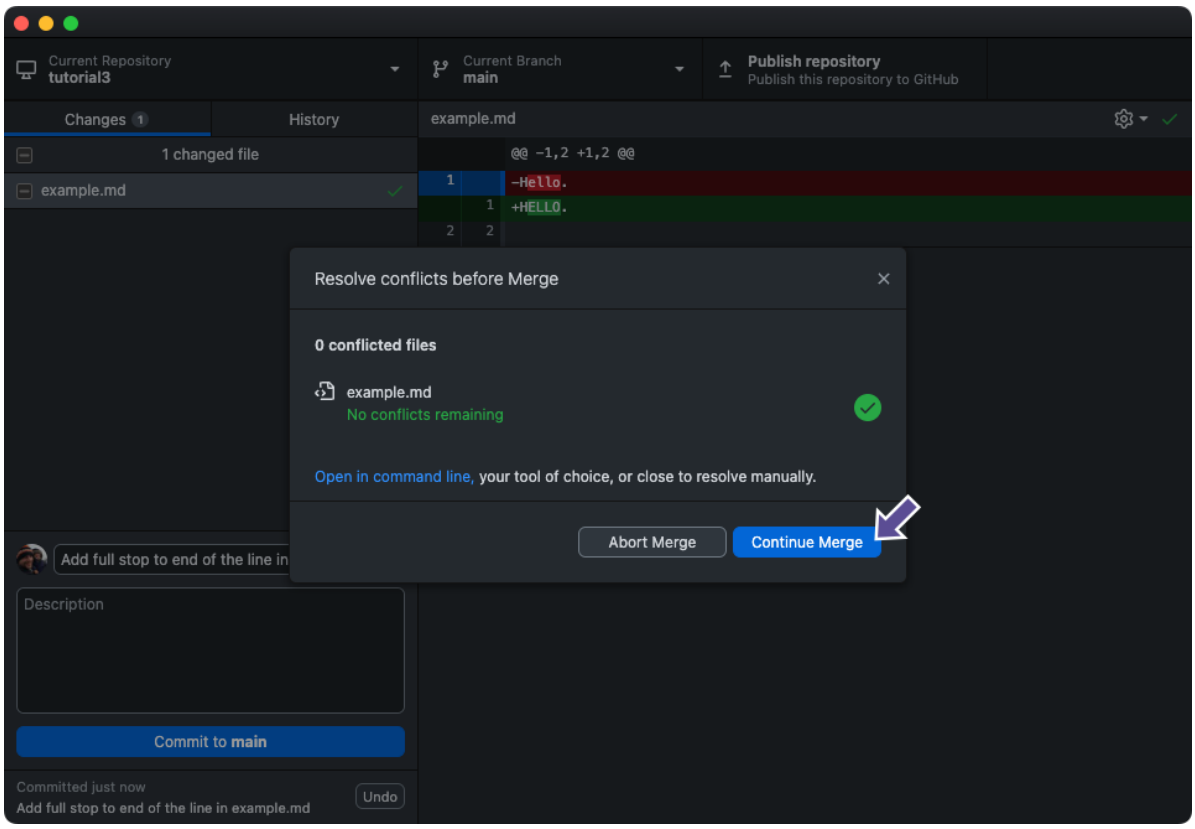
We can see Git is no longer in a merge conflict state:

```
git status -v
```

```
On branch main

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified:   example.md

diff --git a/example.md b/example.md

index 8b94e8e..c886167 100644

--- a/example.md

+++ b/example.md

@@ -1,2 +1,2 @@

-Hello.

+HELLO.
```

GitHub Desktop

Once the special `<<<<<<<`, `=======` and `>>>>>>>` syntax is removed from our files, GitHub Desktop detects that our merge conflict has been resolved and allows us to 'Continue Merge':



And we can conclude the merge.

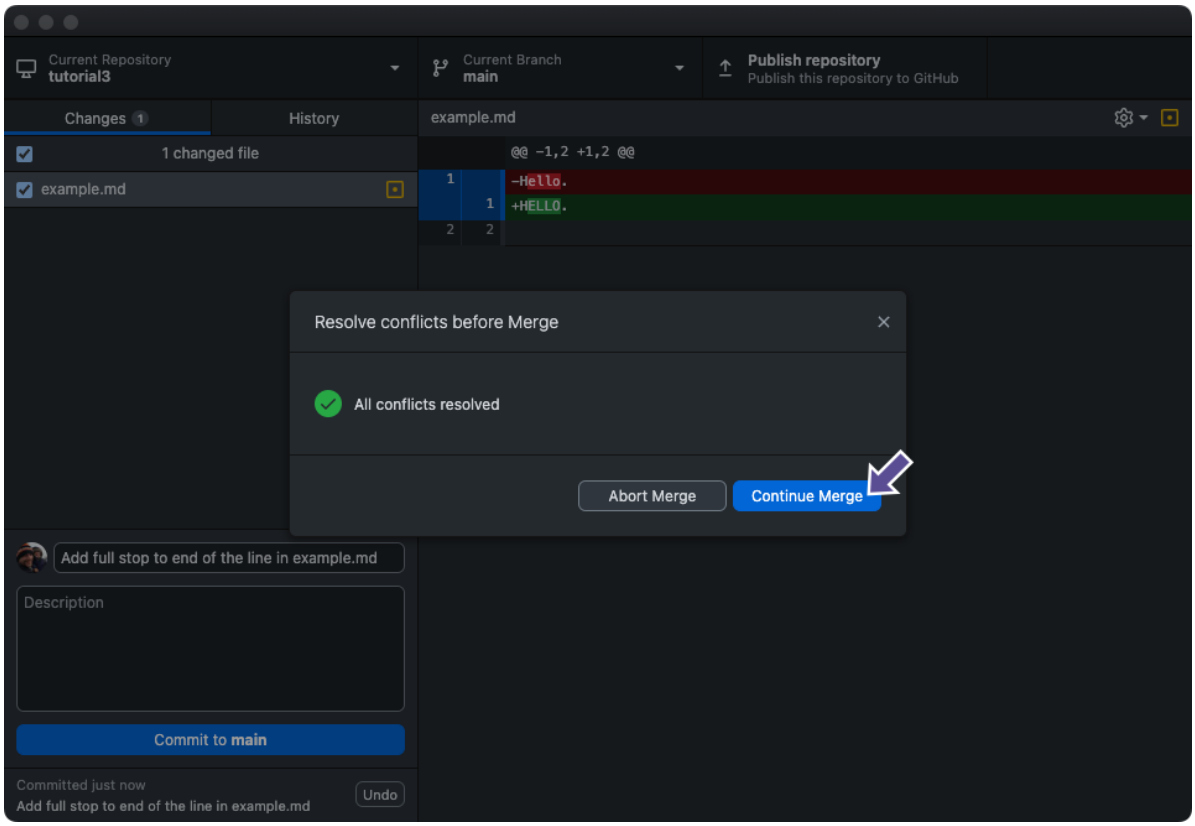
Command-line

I use the `--no-edit` flag to tell Git to use the default message for a merge commit:

```
git commit --no-edit
```

```
[main fe2d762] Merge branch 'capitalise'
```

GitHub Desktop



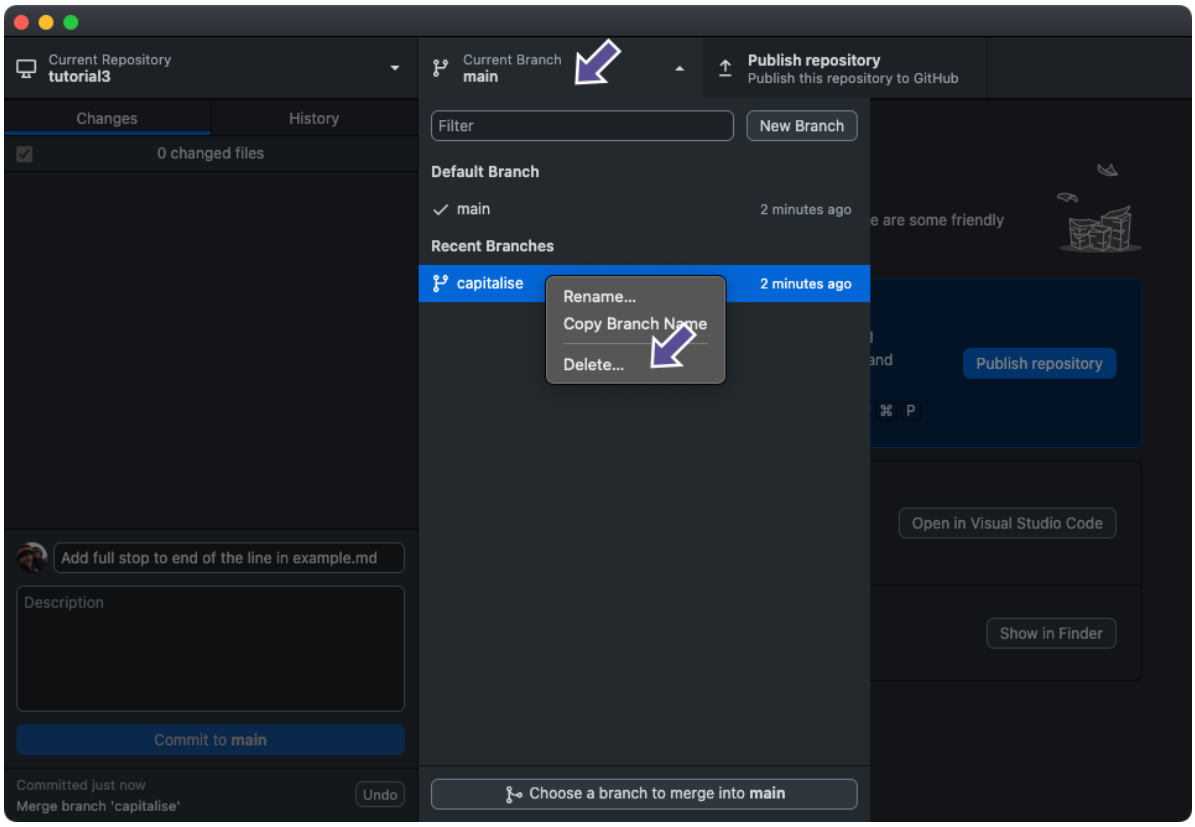
We have integrated our branch into the `main` branch and we can safely delete our short-lived branch `capitalise`:

Command-line

```
git branch --delete capitalise
```

```
Deleted branch capitalise (was 3da9dfb).
```

GitHub Desktop



Code Review: A better approach to resolving merge conflicts

Although Git allows us to resolve a merge conflict as above, this is not considered the "best practice" way to achieve this result. We ended up with something different from both our branch contents and the main branch contents all in one merge conflict resolution, with no opportunity for oversight by someone else working on the project- i.e. we managed to skip the Code Review (aka. Pull Request) stage.

A Pull Request on GitHub should be thought of as a preparation and checking stage for a Git `merge` (or `rebase`) operation. GitHub needs a 'base' branch to 'merge into', usually `main`, and a 'compare' branch to 'merge from'. If Git has indicated that a merge will result in a merge conflict, for the base branch as above, GitHub will not easily let you perform the merge process until this is fixed. So let's see how this could have been done differently.

Let's create a new branch called `goodbye`:

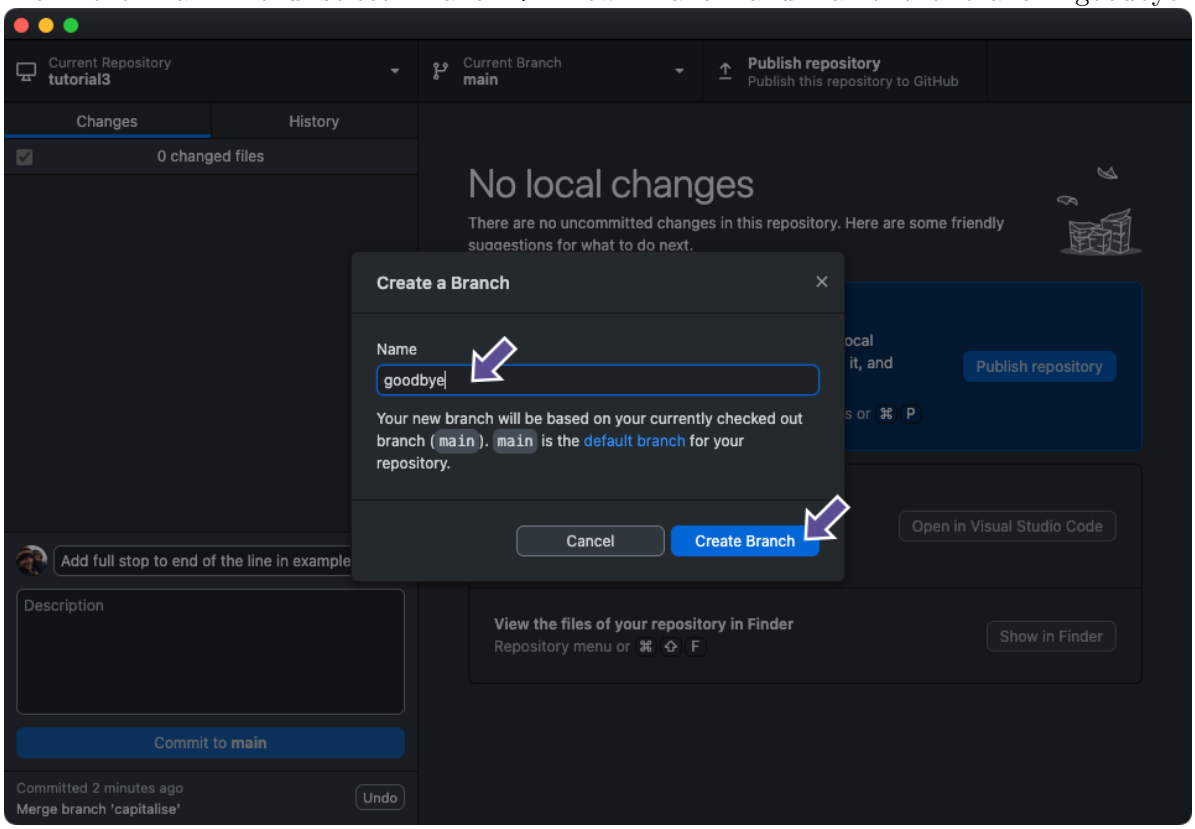
Command-line

```
git switch -c goodbye
```

```
Switched to a new branch 'goodbye'
```

GitHub Desktop

From the main menu select Branch > New Branch and name the branch 'goodbye':



Let's update the file like so:

example.md

GOODBYE

And commit the change:

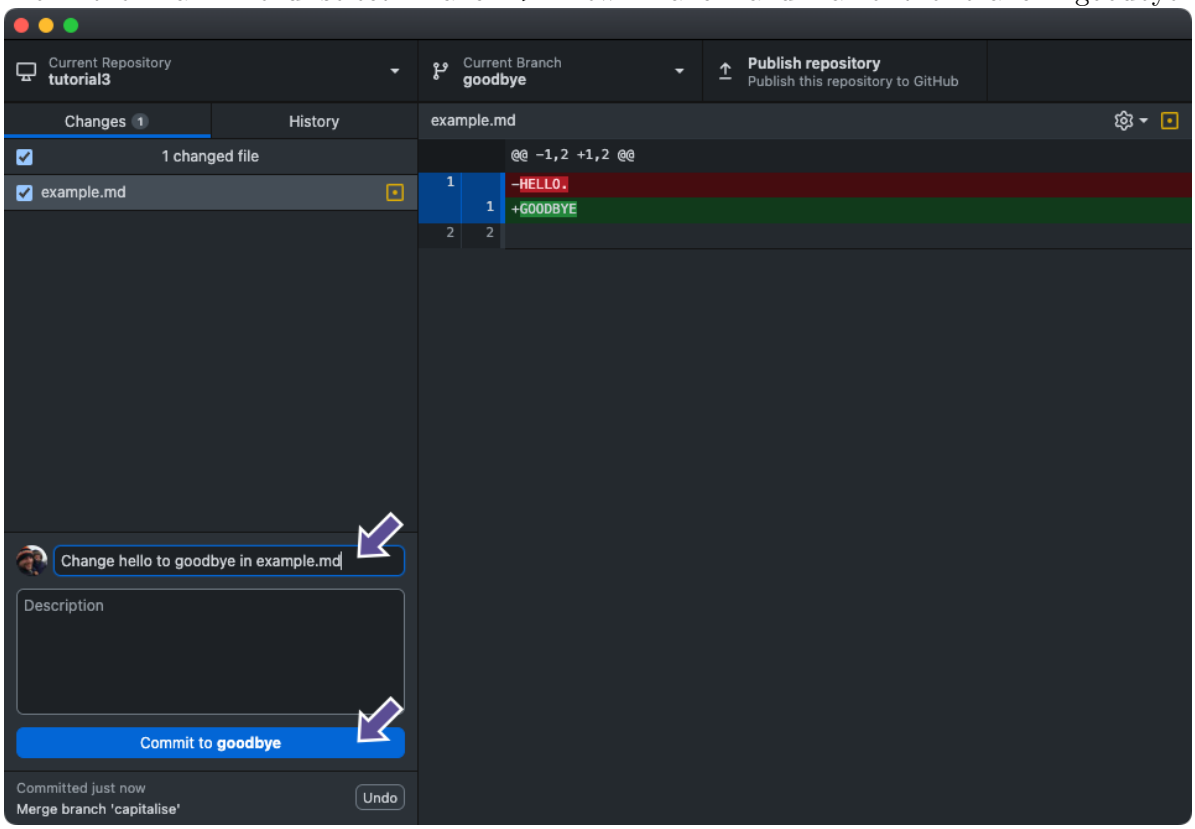
Command-line

```
git add example.md
git commit -m "Change hello to goodbye in example.md"
```

```
[goodbye 753e225] Change hello to goodbye in example.md
1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop

From the main menu select Branch > New Branch and name the branch 'goodbye':



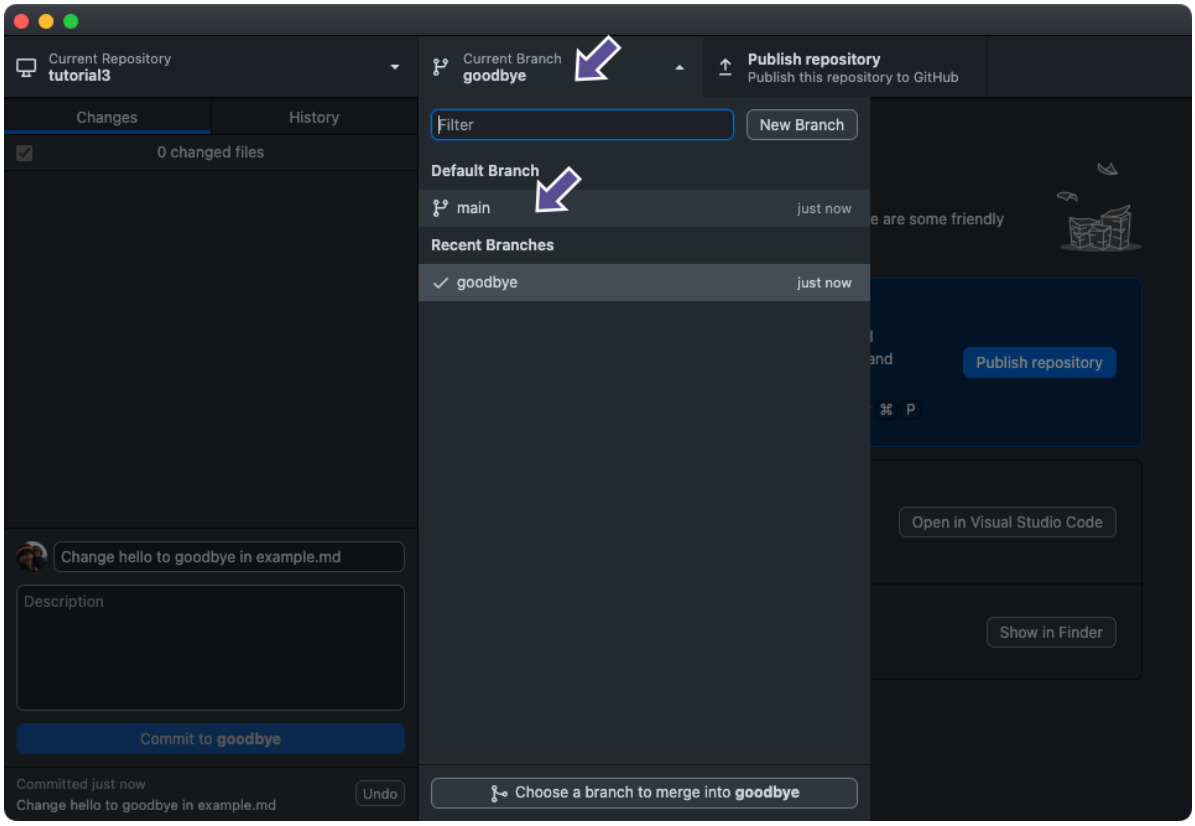
Switch back to the main branch:

Command-line

```
git switch main
```

```
Switched to branch 'main'
```

GitHub Desktop



Make a change to the file on the main branch:

example.md

HELLO

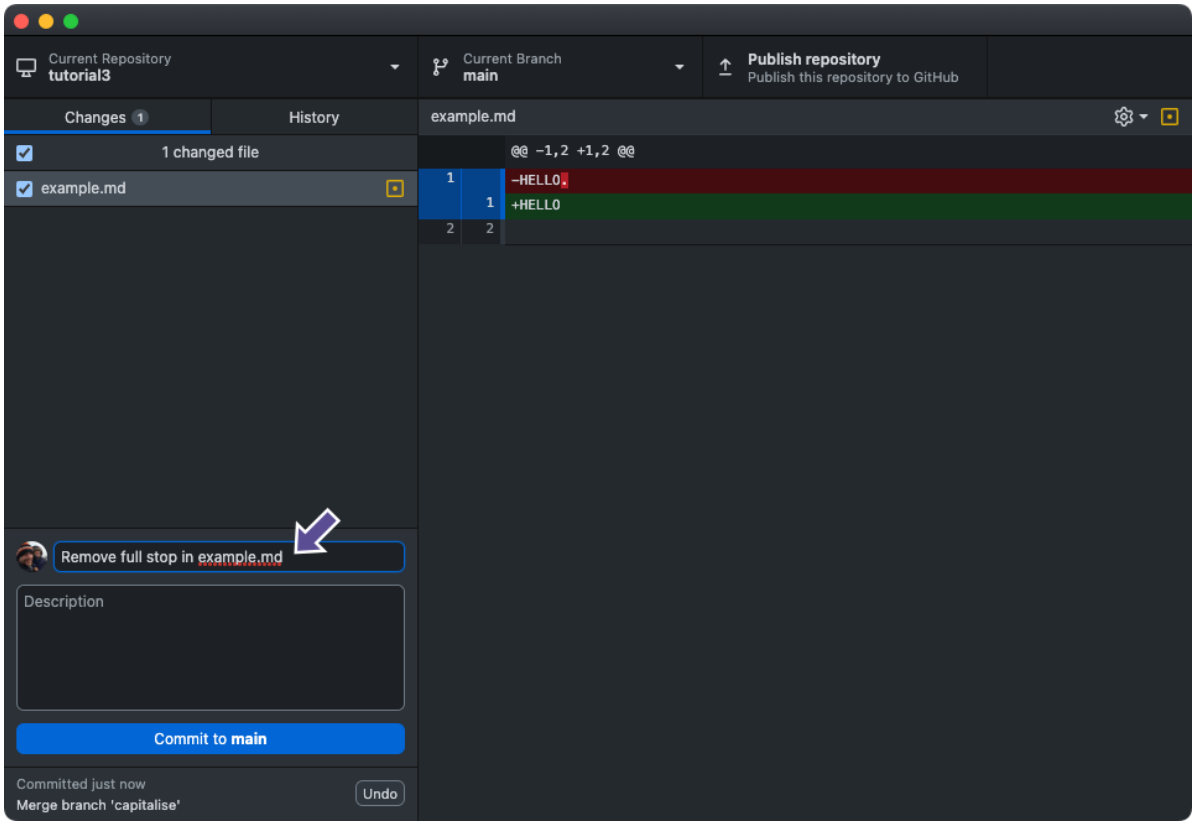
And commit.

Command-line

```
git commit -m "Remove full stop in example.md"
```

```
[main 27c4cc0] Remove full stop in example.md
1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop



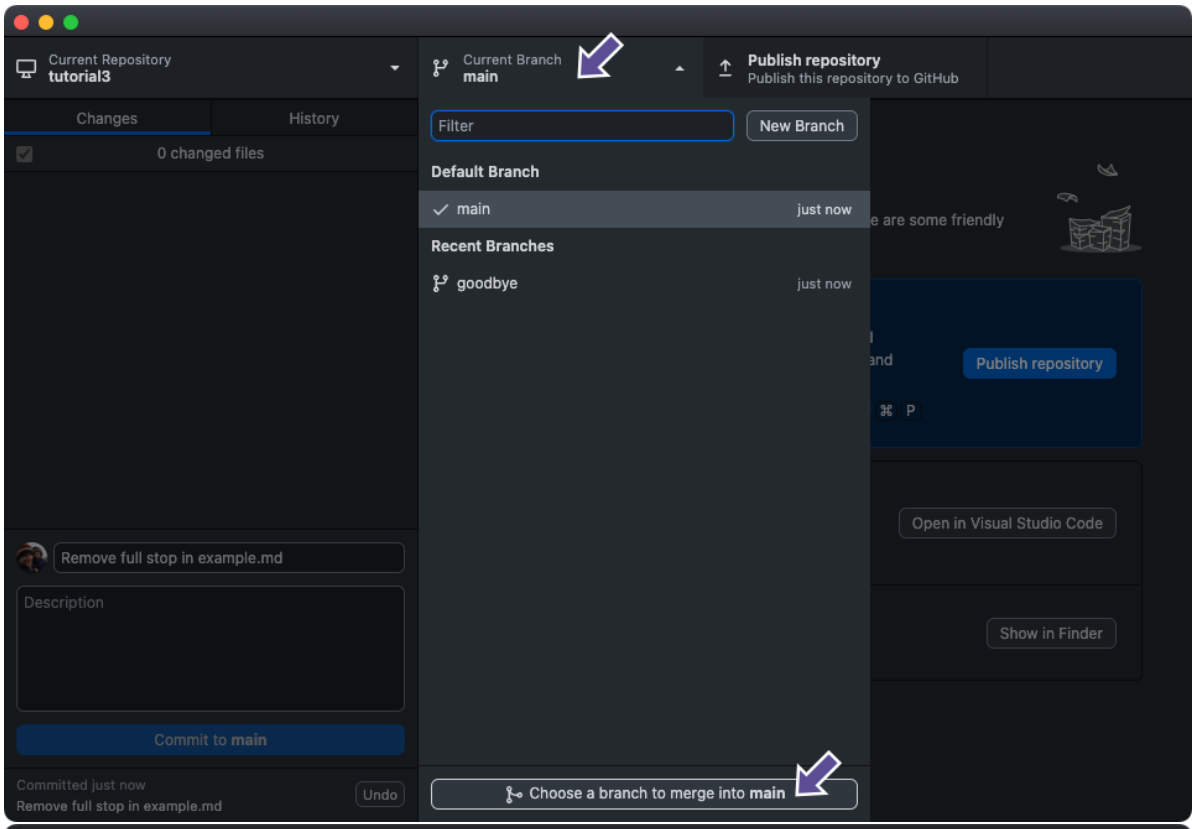
Then try to merge:

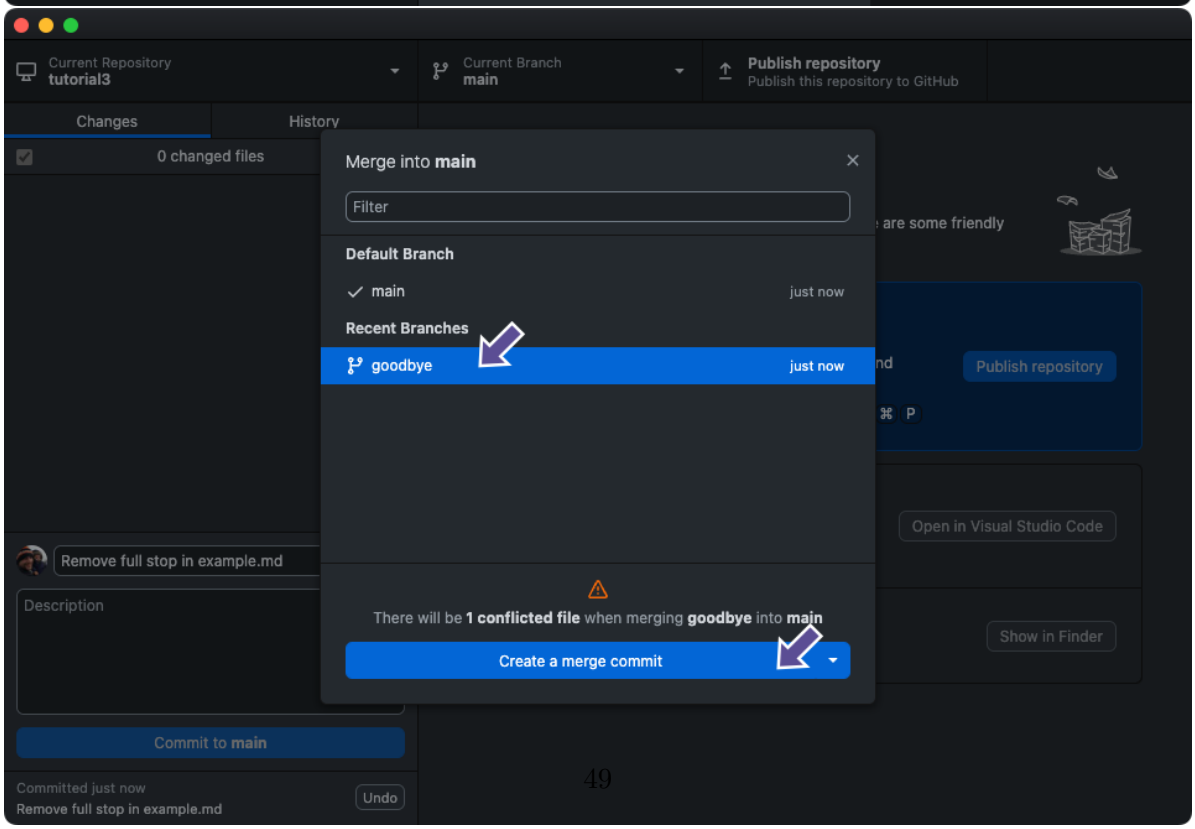
Command-line

```
git merge goodbye
```

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

GitHub Desktop



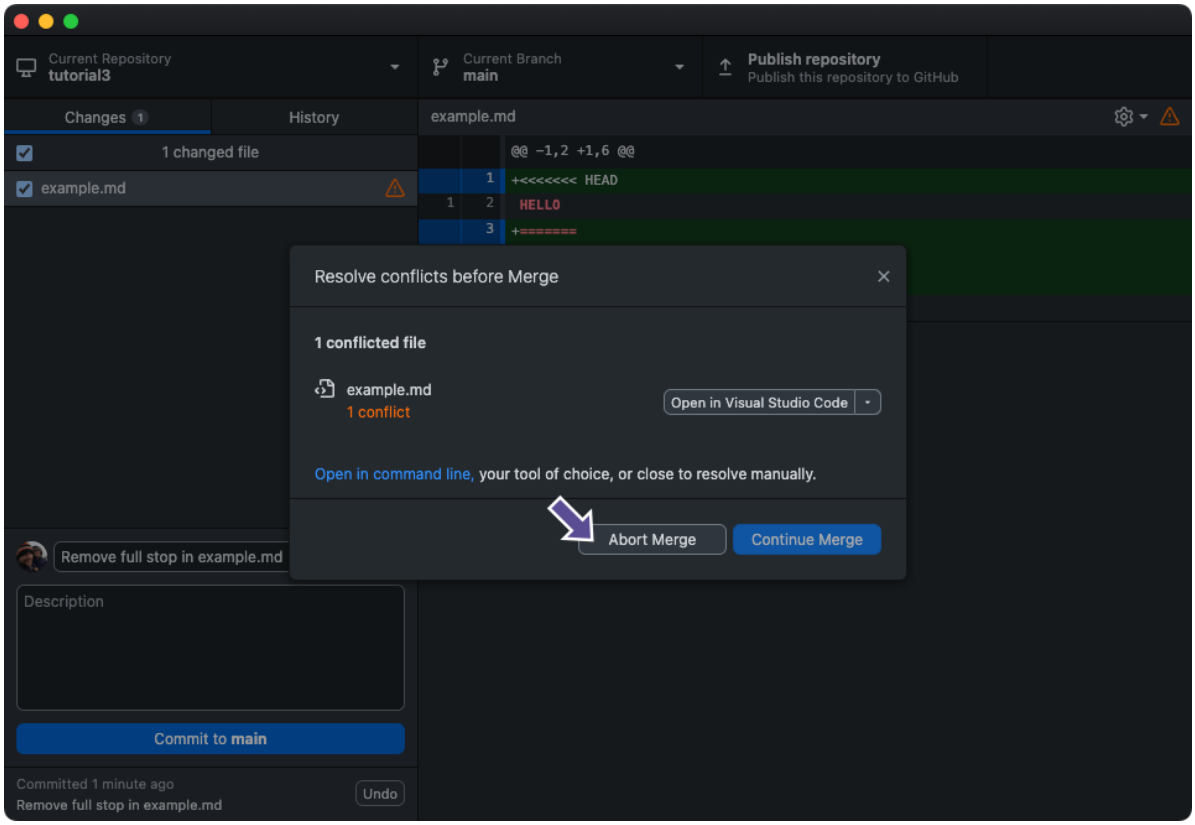


And again we have a merge conflict. This time, let's abort the merge:

Command-line

```
git merge --abort
```

GitHub Desktop



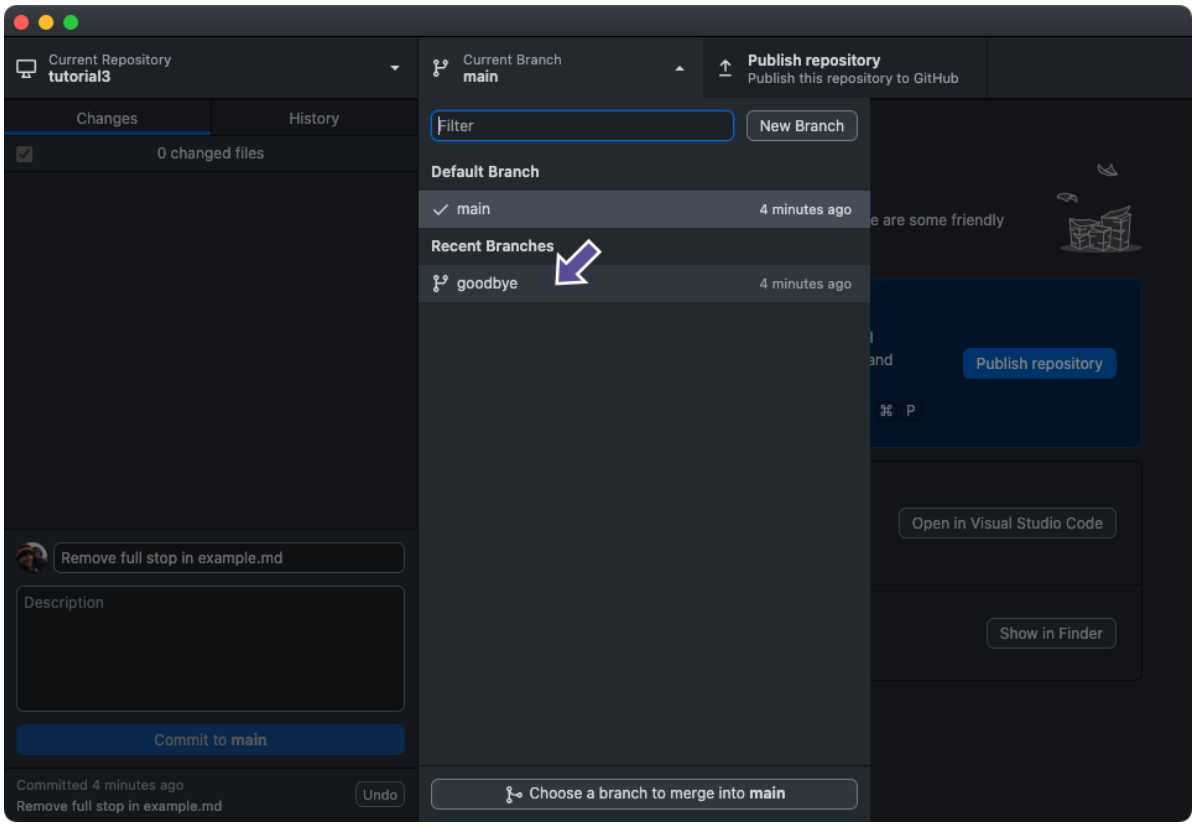
Now switch back to the `goodbye` branch:

Command-line

```
git switch goodbye
```

```
Switched to branch 'goodbye'
```

GitHub Desktop



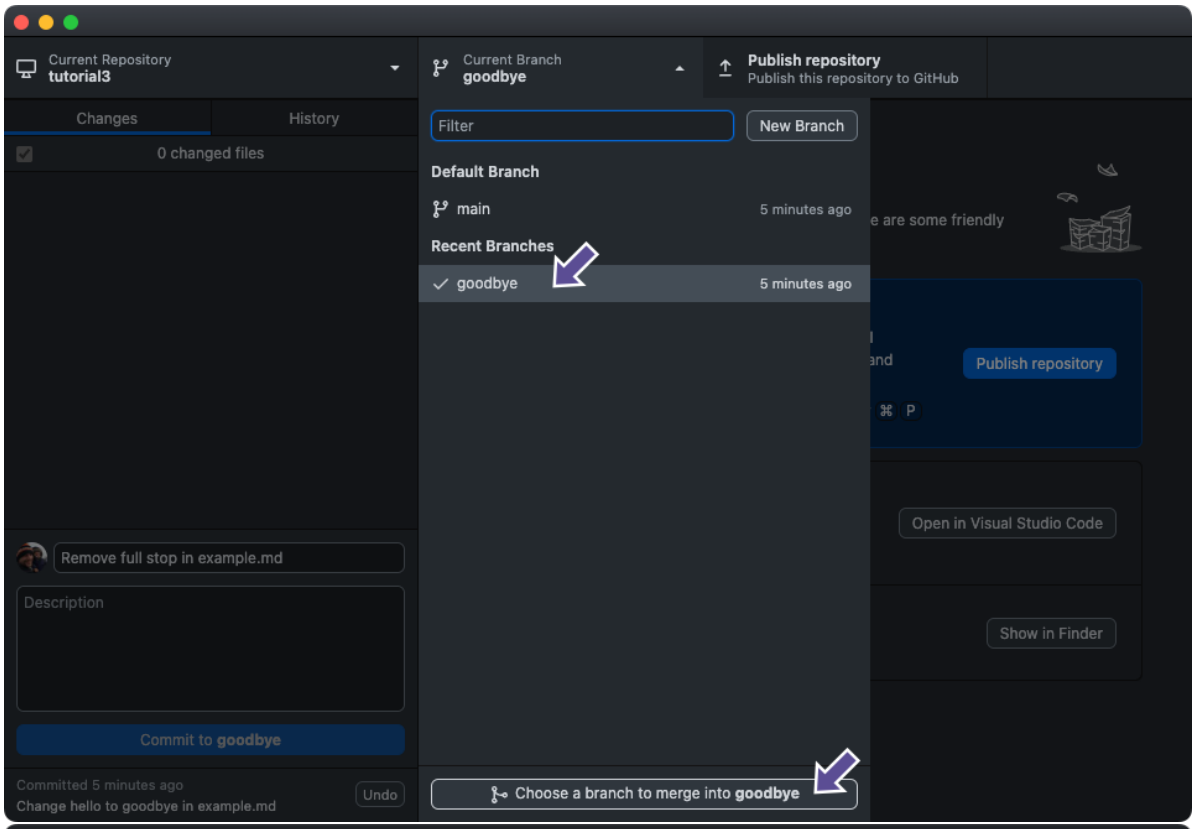
And **merge `main` into our short-lived `goodbye` branch**:

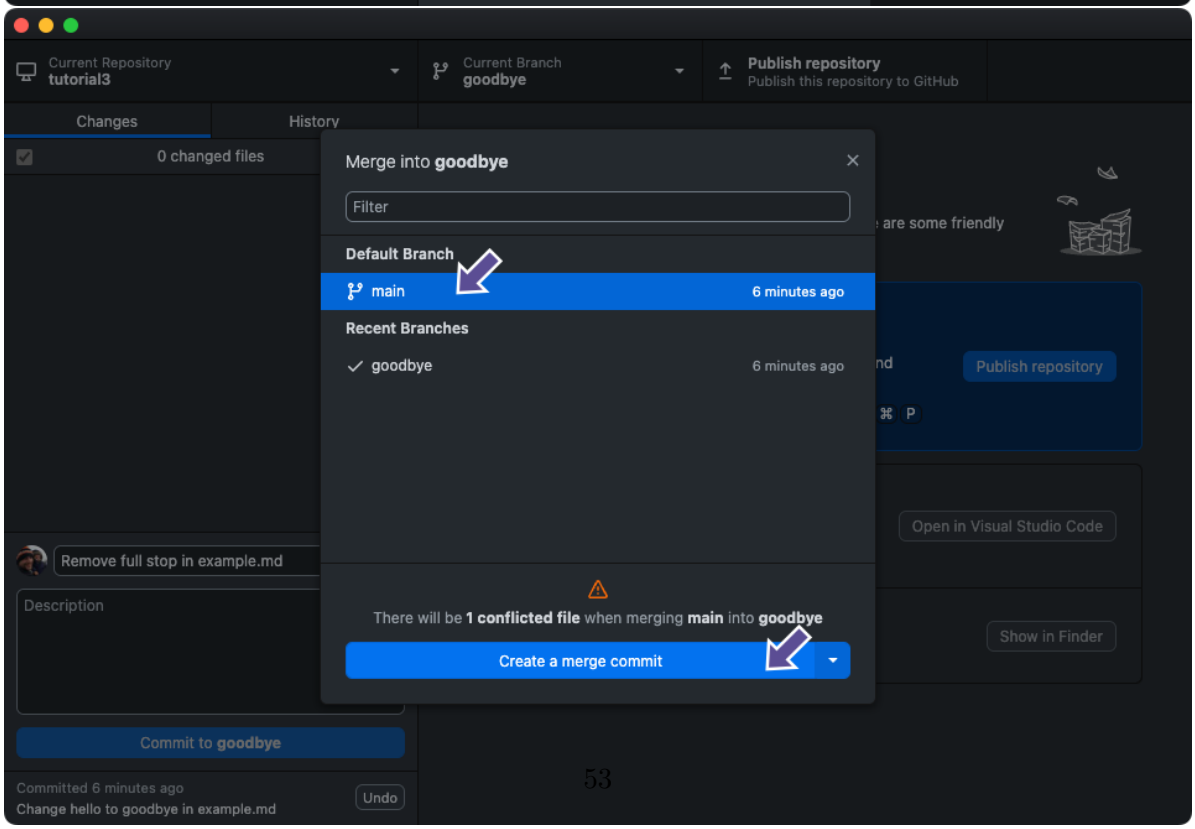
Command-line

```
git merge main
```

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

GitHub Desktop





Now if we head to our preferred code editor, we have almost the same merge conflict to resolve (except `HEAD` is now our `goodbye` branch):

```
<<<<<<< HEAD
GOODBYE
=======
HELLO
>>>>>>> main
```

And we can manually resolve it as before, this time with some Beatles lyrics:

> example.md
>
> You say GOODBYE and I say HELLO

Command-line

And `add` to mark the conflict resolved:

```
git add example.md
```

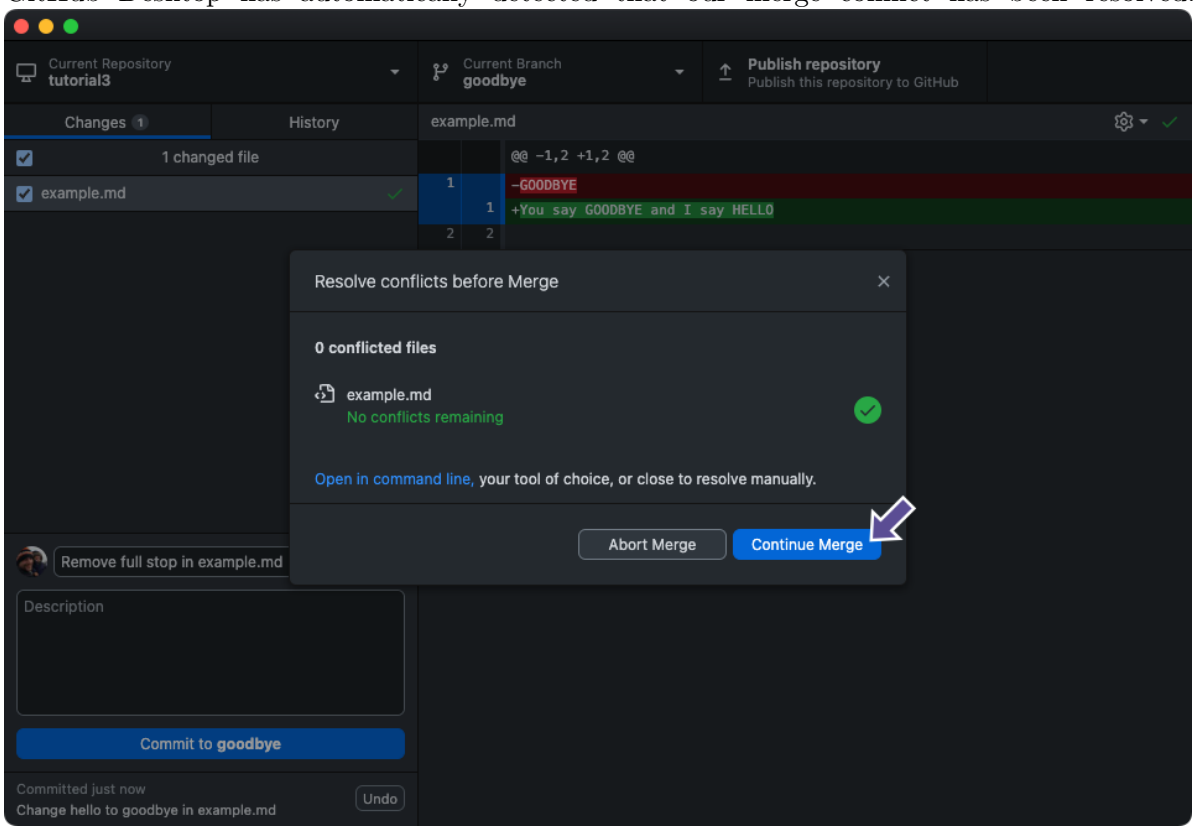
And commit the change.

```
git commit --no-edit
```

```
[goodbye 34ca84f] Merge branch 'main' into goodbye
```

GitHub Desktop

GitHub Desktop has automatically detected that our merge conflict has been resolved:



> **Note**
>
> And now our branch could be ready for Code Review on GitHub via a Pull Request (see Supplement). This makes sense when you are contributing to someone else's project.

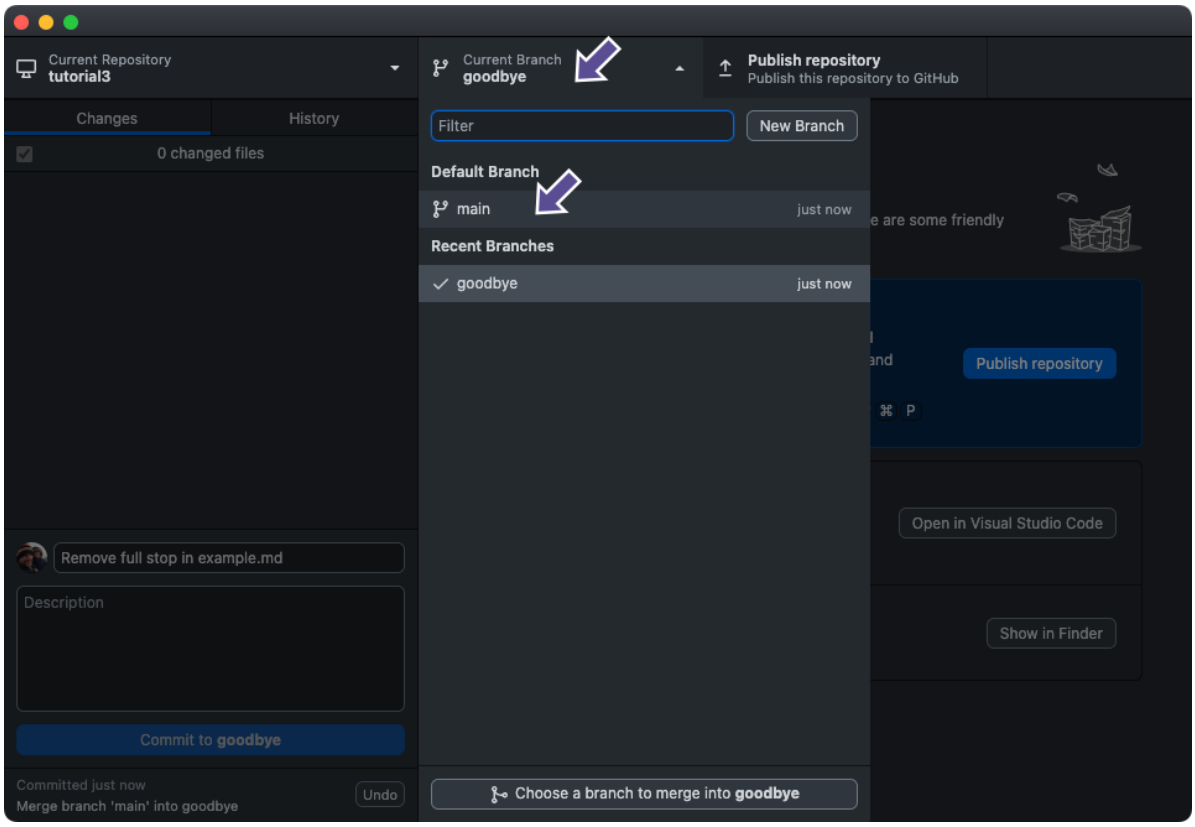
When we switch back to the `main` branch:

Command-line

```
git switch main
```

```
Switched to branch 'main'
```

GitHub Desktop



Our change can be merged without a conflict. We will use `git squash` in the next steps. Squashing allows you to combine multiple commits in your branch's history into a single commit. This can help keep your repository's history more readable and understandable.

Command-line

"Fast-forward" in Git terms means an automatic merge, or in other words, our merge will not result in a conflict.

The `goodbye` branch has 2 commits, which we can see by asking `git log` to compare it to the `main` branch:

```
git log --oneline main..goodbye
```

```
34ca84f Merge branch 'main' into goodbye
753e225 Change hello to goodbye in example.md
```

Since we resolved the merge conflict with Beatles lyrics when merging the `main` branch into the `goodbye` branch, neither of these commit messages does a good job describing what we are about to integrate into `main`. So instead of transferring both commits to the `main` 'as is', let's 'squash' them into one:

```
git merge --squash goodbye
```

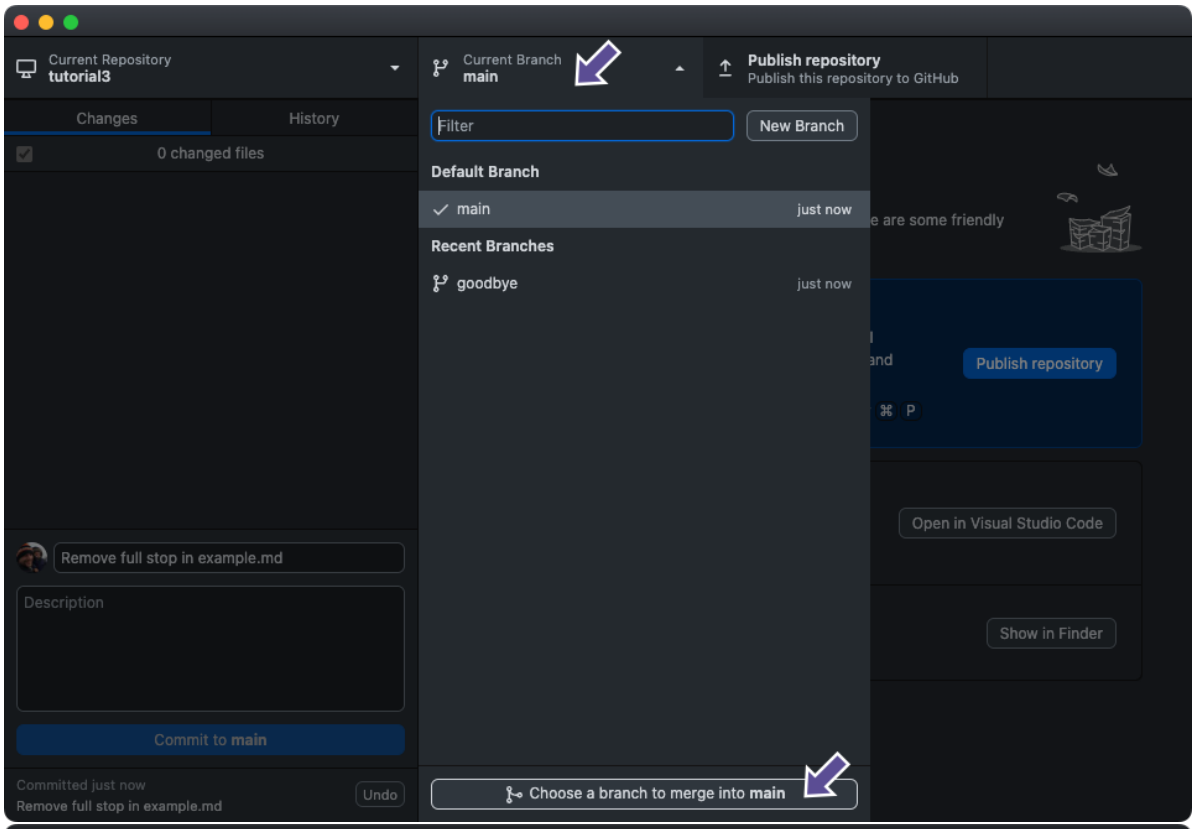
```
Updating 27c4cc0..34ca84f
Fast-forward
Squash commit -- not updating HEAD
example.md | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

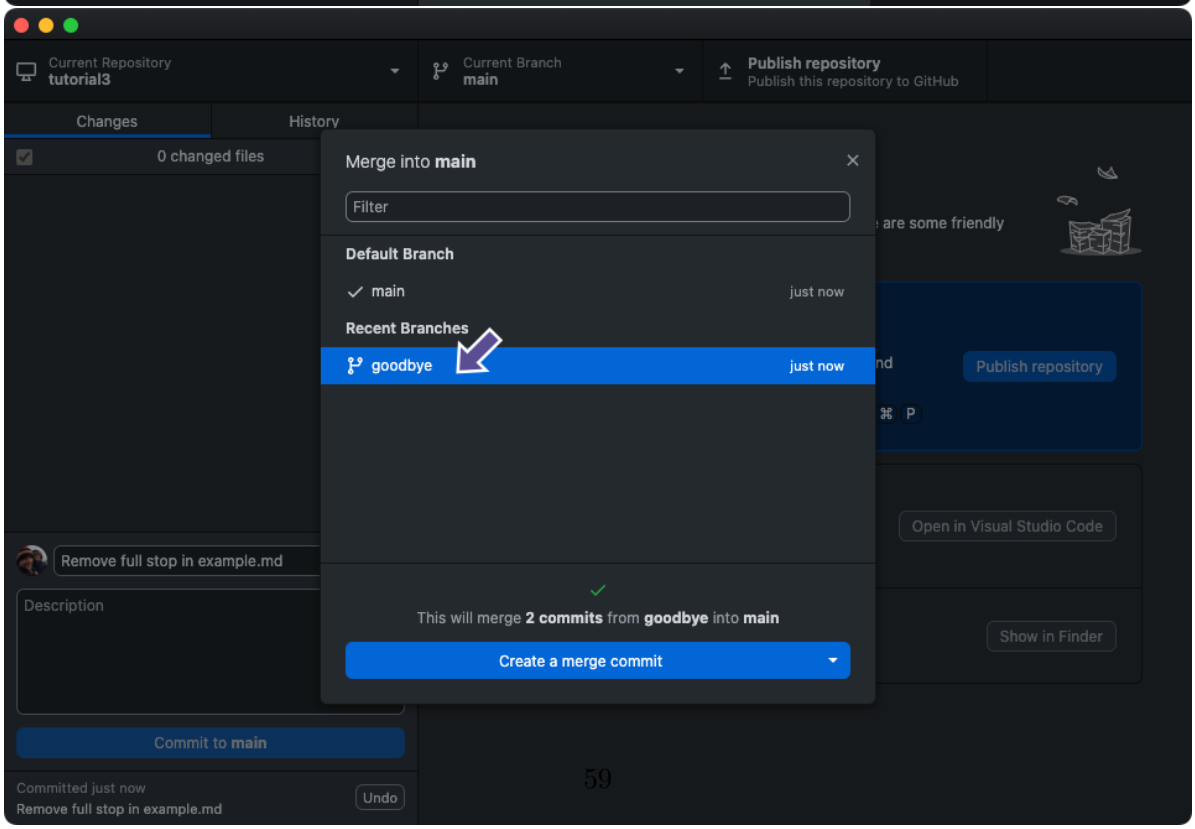
Then commit the squashed change with a more descriptive message:

```
git commit -m 'Change to Beatles lyrics'
```

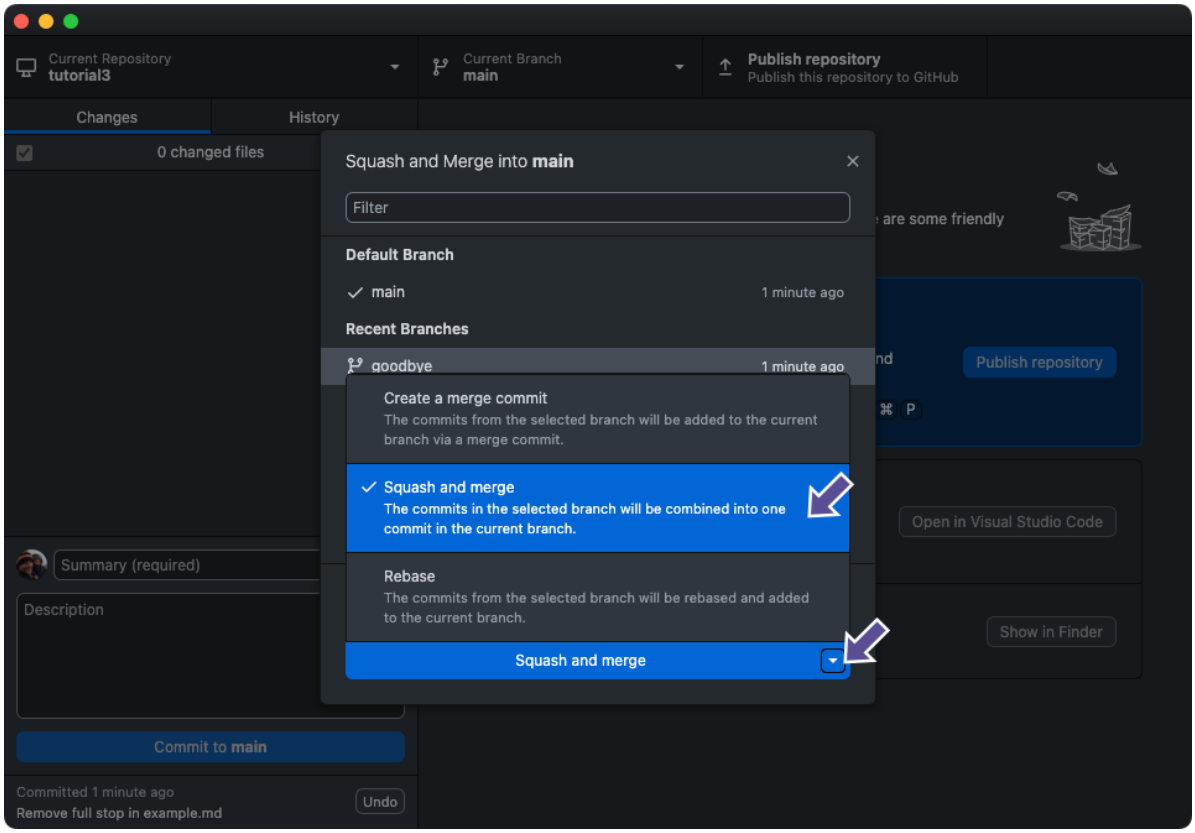
```
[main 08a1ff2] Change to Beatles lyrics
1 file changed, 1 insertion(+), 1 deletion(-)
```

GitHub Desktop

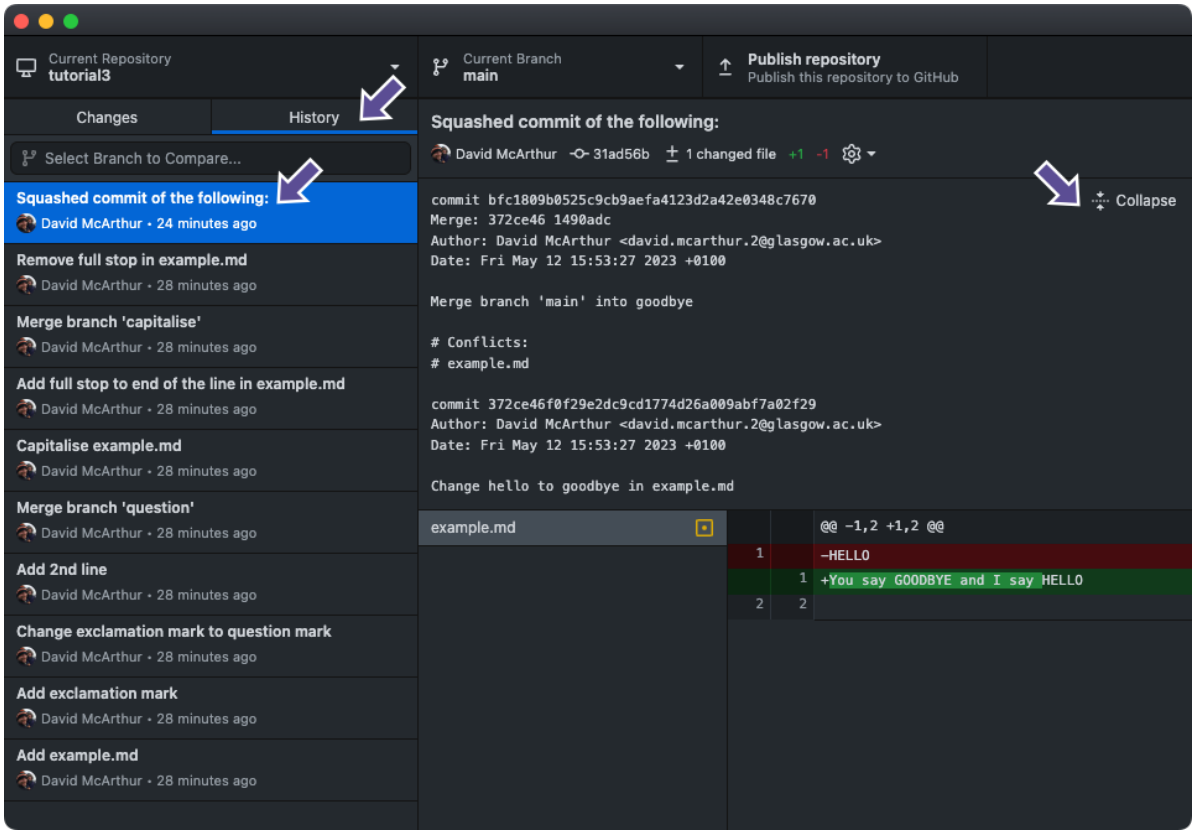




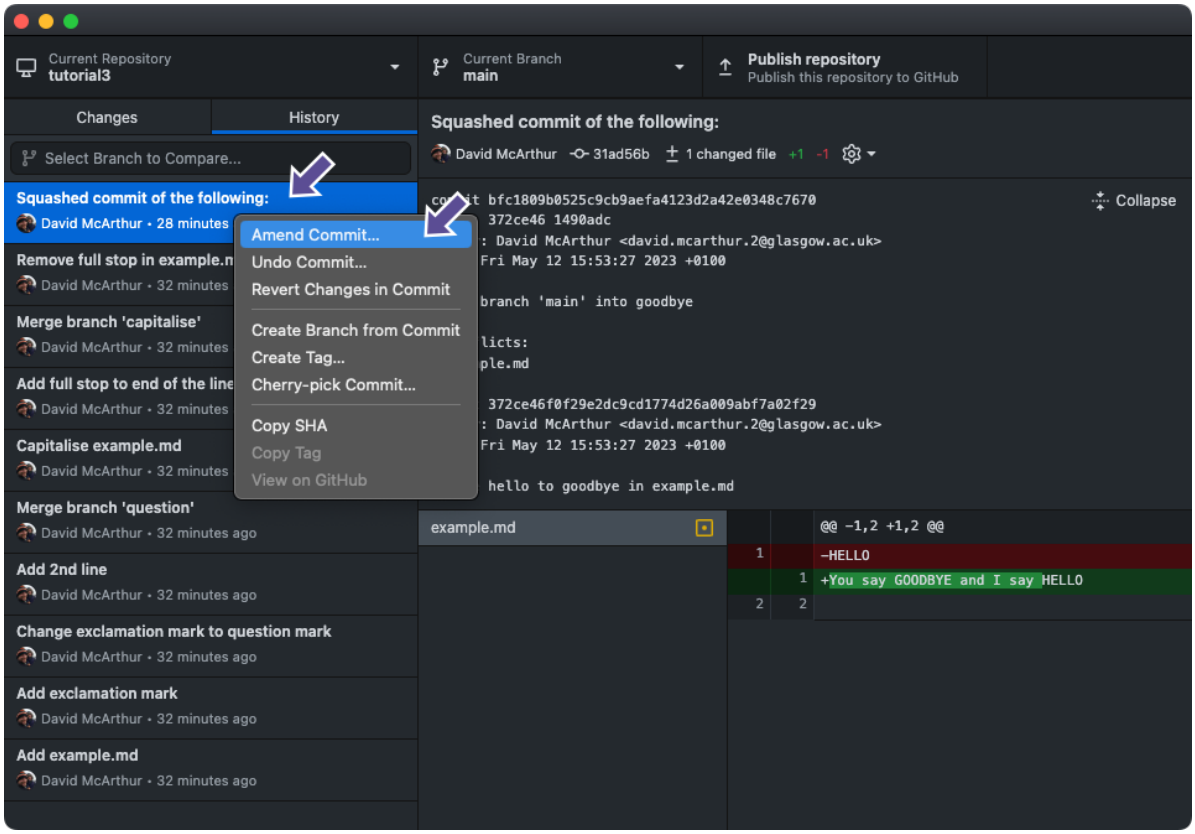
This time instead of clicking ‘Create a merge commit’, click the little arrow on the right of the button and select ‘Squash and merge’:



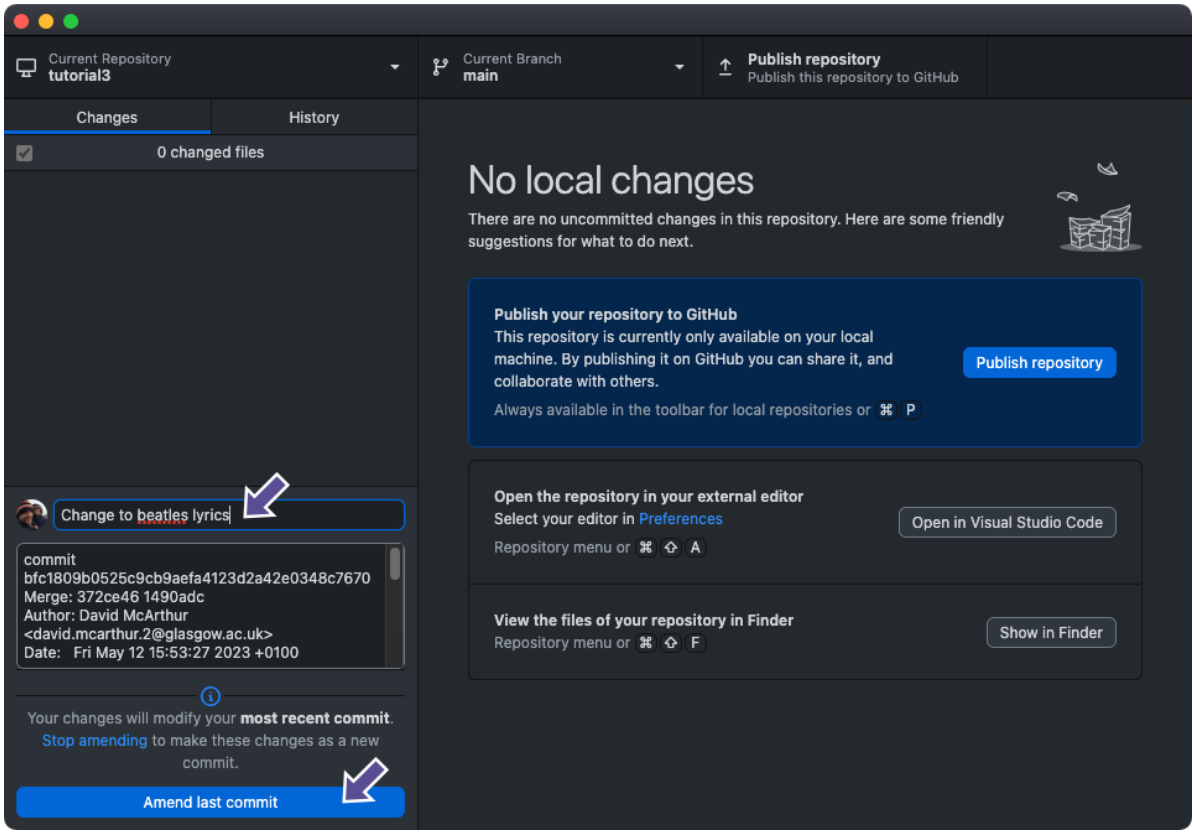
Now if we look at the history tab we can see a commit with the message ‘Squashed commit of the following:’. If you click on ‘Expand’ on the right, we can see the description of the commit has a log of the commits that were squashed into this one:



Let's change the title to something more descriptive by right-clicking on the commit message and selecting 'Amend commit…':



Then change the message to ‘Change to Beatles lyrics’ and click ‘Amend last commit’:



"Fast-forward" in Git terms means an automatic merge, or in other words, our merge will not result in a conflict.

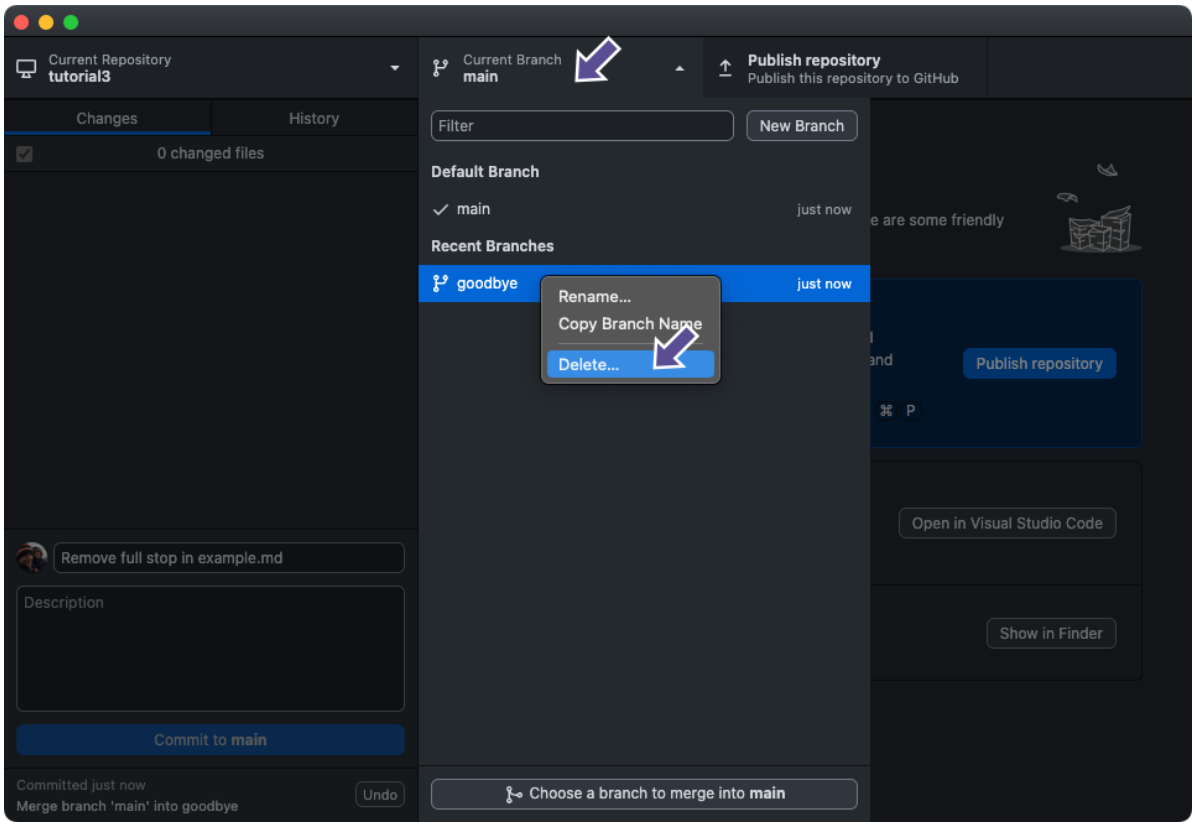
Finally, you can safely delete the goodbye branch:

Command-line

```
git branch --delete goodbye
```

```
error: The branch 'goodbye' is not fully merged.
If you are sure you want to delete it, run 'git branch -D goodbye'.
```

GitHub Desktop



So to recap, we have prepared our branch for integration by first combining any new changes from the `main` branch into our branch and resolving the merge conflicts in isolation, creating a "final version" of the change we want to make with our branch ready to be checked, before attempting to integrate our branch back into the `main` branch.

If you need to keep your short-lived branch open for some reason, it's advised to merge any recent changes from `main` into your branch often, to avoid painful merge conflicts later.

Supplement: Create a Pull Request to an existing project

This tutorial is about contributing to someone else's project. The sample project that we are going to contribute to can be found here:

https://github.com/UofGAnalyticsData/tutorial2

We're simply going to add a line to the existing readme.md file, but you may have found a bug, or thought of a new feature, in someone else's project, and you've taken it upon yourself to contribute and submit it to the project maintainer for approval.

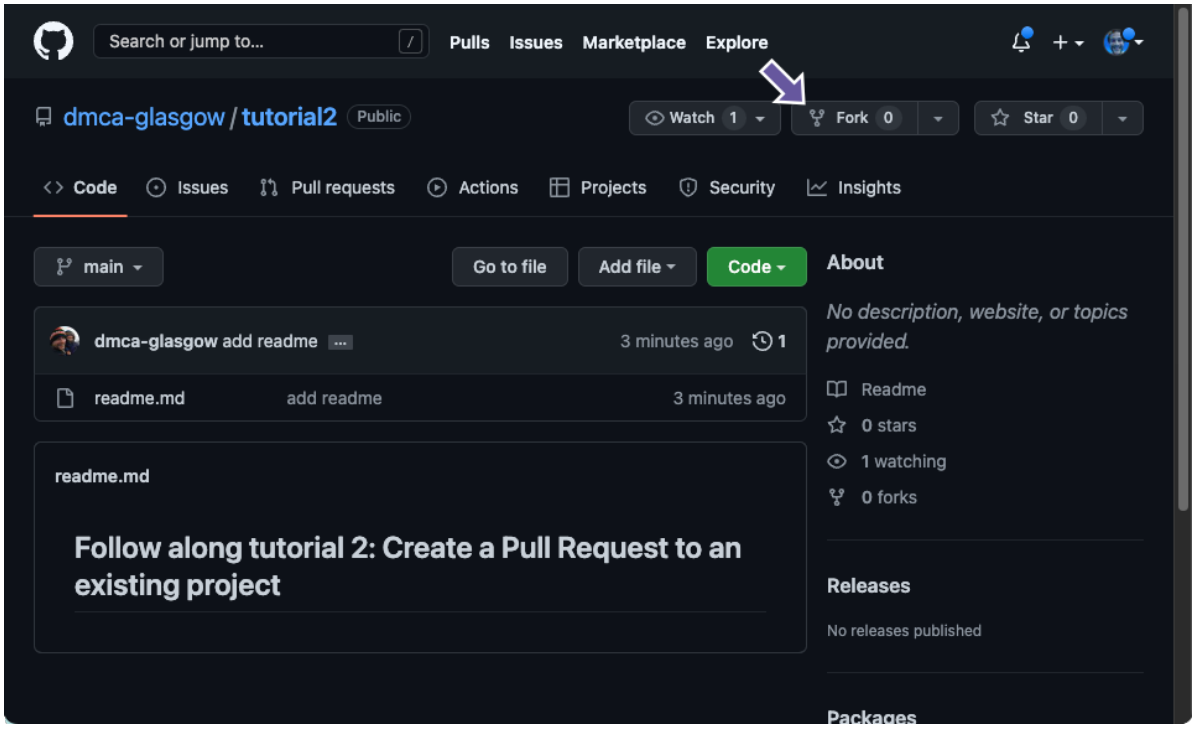
In Git terms, what you need to do is:

1. **Fork** the project on GitHub to your account
2. **Clone** it to your local machine
3. Make the change
4. **Stage**, **commit** and **push** the change to GitHub
5. Create a **Pull Request** on the original project which points to your forked repository
6. **Code review**: the project maintainer will review your change to decide if they want to integrate it, close it, or request modifications

> **Note**
>
> What is the difference between Clone and Fork?
> **Clone** is a Git feature that allows a user to make a copy of a repository, including all the associated metadata on their computer so they can work on it. This is an essential feature and makes sense for a team of collaborators, with read-and-write privileges to the repository, all working on a project together.
> If you have read privileges but not write privileges to a repository, like for example an open-source GitHub project where you're not a project maintainer like the React repository we explored above, it's easy to consume open-source projects but difficult to contribute to them.
> **Fork** is a GitHub feature aimed at making it easier to be part of the open-source community. When you fork a repository, you make a copy of a repository, including all the associated metadata, but crucially, you become the owner of the new copied repository, and you now have read and write privileges necessary to make changes.
> If you would like to contribute your changes back to the original repository, you can create a **Forked Pull Request** (see the next section) which will notify the original project maintainers that you are requesting to have your changes integrated with their project. Those maintainers reserve the right to approve, reject, or request changes to your Pull Request.

Fork

To fork a GitHub repository, first, click the Fork button on the top right:



Then you have the opportunity to give the forked repository a different name, description, and