Intro to Frontend Development

1.9: Git & Version Control

Learning Goals

- Practice version control with Git

Estimated Read Time: 50 Minutes

.

Introduction

Welcome back! In the previous Exercise, you learned (almost) all there is to know about CSS preprocessors, taking a look at their general syntax and capabilities through the lens of a specific preprocessor, Sass. You also looked into the differences between pre- and postprocessors, along with how one of the most useful postprocessors, autoprefixer, works. Finally, you learned about CSS variables and why they've become a good alternative to preprocessors.

As discussed previously, you should always keep a copy of your original "styles.css" file (before it's been run through a postprocessor). The bigger your project grows, the more important it will be that you keep track of different versions of the same (or slightly different) files. This is especially true for projects that require collaboration from multiple parties. This is where version control comes in handy, which is why in this Exercise, you'll learn how you can save and manage different versions of a project over time with the version control system, Git. Let's start with a look at what the term "version control" means before creating a new repository for your portfolio project!

What's Version Control?

If you've ever used collaborative tools such as the shared document feature on Google Drive, you probably know more about version control than you realize! When working on any project, you'll inevitably make changes to some files, then upload them to a web server for your users to collaborate on or view. What happens, though, if you accidentally make a mistake or delete the wrong file? Or, perhaps you remove a feature only to discover weeks later that the feature hadn't been so bad after all. If you're familiar with Google Docs or Sheets, you'll know that you can easily view the revision history to see who did what and when. You can even revert back to a previous version if you should so desire. Believe it or not, this is a form of version control.

For these types of issues, version control can be a truly invaluable tool. It allows you to save each change you make to one or multiple files so that you can go back and view (or revert) those changes at a later point in time.

Keep in mind that version control isn't a specific tool, but rather a generic term used to describe the overall process and concept of tracking changes to a project or working file. There are many different ways to perform version control. Manual version control, for example, involves duplicating your project folder every time you finish something or start something new; this ensures you can always revert your changes if need be. While this may be the most obvious way to manage version control, it's unfortunately quite ineffective, which is why you'll be looking into an option that's not only best practice but also free and effective: Git.

The Power of Git

Git is a free and open-source version control system. In recent years, it's emerged as a quasi-standard for version control in new projects, which is why you'll be focusing on it for the remainder of this Exercise. Do keep in mind, though, that there are other (less popular) options such as Mercurial and SVN.

Git is a distributed version control system. The "distributed" part means that each team member (or collaborator) can have a local copy of the files on their computer; they make changes on their own computer without interfering with what others are doing to the same files, then share new versions of the files with their team members once they're done. This special quality of Git is great for collaboration on development projects.

Before diving deeper into how you can use Git for your own projects, you'll want to first familiarize yourself with some of the key terms associated with Git and version control.

Repositories

A repository (or repo) is a single Git project, and it can have one or more branches. Think of it like a parent folder for your project that will contain all of the other folders and files.

Branches

A branch is, essentially, a copy of a set of files in a repository where you can make changes without interfering with the original files in the main branch. By default, each repository has a main branch. This is where the development work takes place.

TIP!

A repository can also have more than one branch. This is especially handy for larger-scale projects. Imagine the following scenario:

You have a movie review blog that you're actively maintaining. At some point, you decide you also want to cover TV show reviews, which requires adding an entirely new section to your site, new links and navigation, and new branding. While you're working on creating all the code and assets for this new section, one of your readers reports a bug on the homepage, and you need to fix it and release it as soon as possible. Just great! What should you do with all the code you've been working on? You don't want to throw it away, as you want to finish it on time, but you need to go back to the version that's currently live to fix the bug ASAP.

This is where a custom branch can swoop in and save the day. You can create a branch called "tv_update" where you save all the work you've done for the new section as well as related changes. Then, you can switch to the main branch, fix the bug there, and push it live. After that, you can switch back to the "tv_update" branch and continue your work where you left off.

Pretty handy, right? The reason this functionality is called a "branch" is because Git works in a tree-like structure:

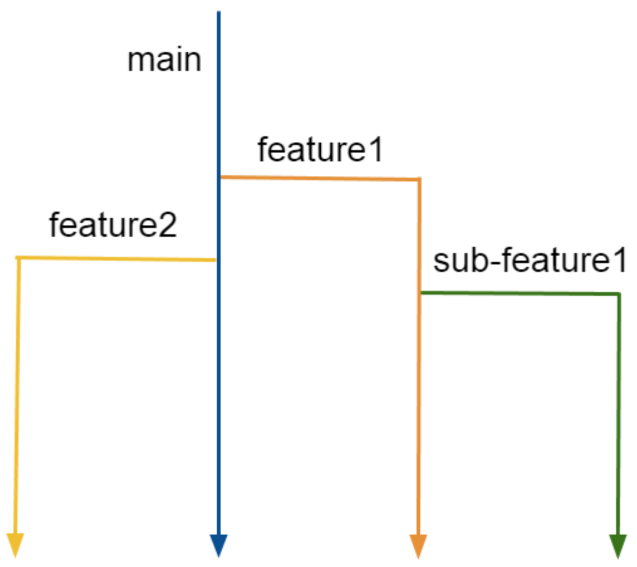


Figure 1. Visual representation of Git's branches.

The blue arrow is the "main" branch, storing what's currently live on the site. At some point in time, a new branch, "feature1," is created off of "main." That branch will start out with the same files as "main." Later, the branch "sub-feature1" is created off of "feature1," which will, again, start out with all the same files as "feature1." Meanwhile, another branch, "feature2," has also been created off of "main" again. You can have as many branches as deeply nested as you want. Do note, though, that the more branches you have, the more complicated it will get, just like if you were to have too many folders within folders on your computer. Only create branches where necessary, and always clean up your branches when you're finished with them.

Commits

In order to save your changes to a branch in your repository, you need to make a commit. A commit is a set of changes to one or multiple files in a branch. Generally, a commit can consist of the following:

- A file that's been added

- *A file that's been deleted*
- *A file that's been changed*

A commit also comes with a commit message, which should indicate its purpose (what exactly the commit is doing). The goal here is to create meaningful, compact commits. In order to understand what this means, let's look at two examples for the same changes. The commit messages are above the boxes, which detail the actual files that changed:

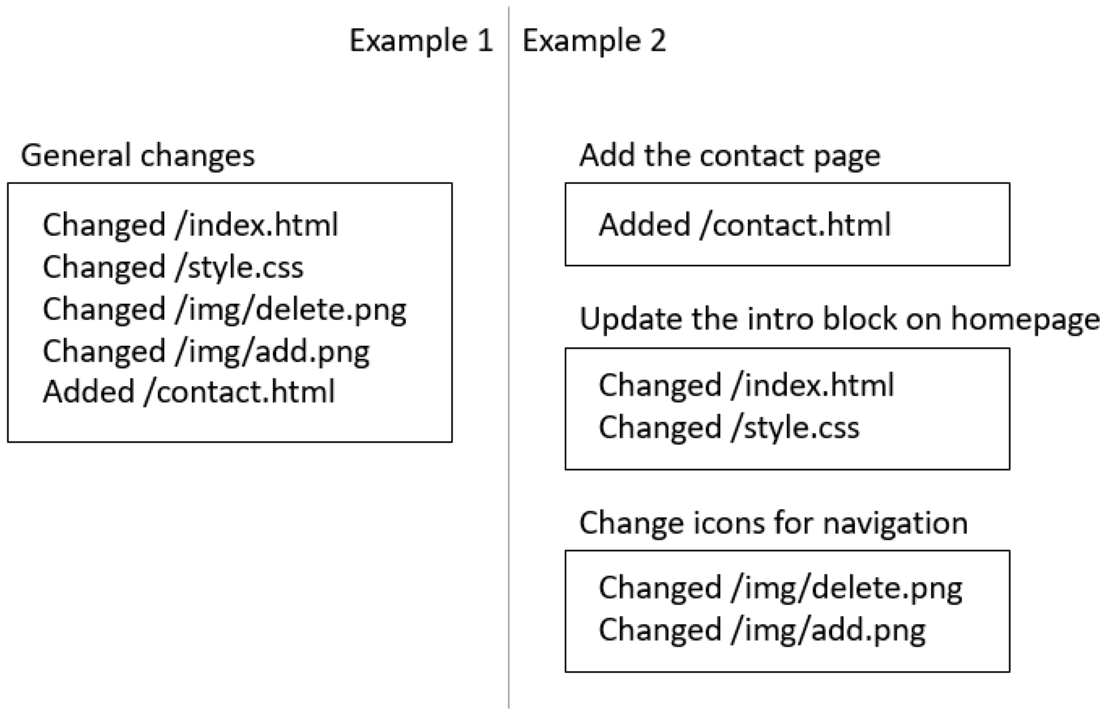


Figure 2. Side-by-side examples of different commit messages

Both examples describe the same overall changes, but let's look at the commit messages on their own:

Example 1:

- *General changes*

Versus

Example 2:

- Added the contact page
- Updated the intro block on homepage
- Changed icons for navigation

Which of the above gives you a better idea as to what exactly has changed? Example 2 is the obvious answer. In addition to having more detailed descriptions of the changes made, the commits in Example 2 are also much more atomic than the single commit in Example 1. Atomic refers to commits that address a specific and "irreducible" fix or feature—in other words, commits consisting of the smallest possible set of related changes.

Let's revisit the image above. For each step in the list of commits, can the changes be made without affecting (or depending upon) the other commits? Changing icons and adding a new page aren't necessarily dependent upon one another—as in, one can be done without the other—so they should go into separate commits. On the other hand, the styling changes made in the commit "Update intro block on homepage" can't be split into multiple commits because the two files are dependent upon one another.

Why do atomic commits matter? Let's say you push out several unrelated changes to an existing feature or page, just like what was done above, and something breaks in your code because of one of those changes. If you haven't split your commits into atomic commits, you'll end up having to revert all of the changes you've made; in other words, you won't be able to target the specific change that broke your code and will, instead, need to revert even unrelated changes that are working just fine.

Writing atomic commits is considered one of the best practices in version control, and you'll find many resources out there on how to properly write and structure Git commits. (If you're curious, you can find some in the Resources section of this Exercise). The key thing to remember is that each commit message should be able to complete this sentence:

"If applied, this commit will have ________."

Let's try this with the examples from above:

- If applied, this commit will have added the contact page
- If applied, this commit will have updated the intro block on home page
- If applied, this commit will have changed icons for navigation

In most cases, following this schema will help ensure that your commit messages are imperative and appropriate in describing what the commit actually does. For example, if you tried to fit "general changes" into this pattern, you'd get something like, "If applied, this commit will have general changes," which clearly doesn't work.

Also note, here, that while the term "commit" describes a single set of changes on a branch, it's also used as a verb to create such a set of changes. Thus, committing the changes for "delete and add icon" would involve creating a new commit with these changes.

Gitignore

Sometimes you have files in your project folder that you don't want to commit as part of your repository at all. There are many types of files that can be part of a project folder but that aren't part of the actual code repository—log or cache files, for example. These files are updated constantly, so every time you make a commit, they would be part of that commit even though they're not directly related to what you worked on. Other types of files that you wouldn't want to commit include the output version of your SCSS code (see the previous exercise on CSS Preprocessors) or, if you use a Mac, a file called ".DS_Store" that appears here and there.

Git has a nice way of dealing with these files. In the root folder of your repository, you can create a new file and call it ".gitignore". The leading dot is very important! On Mac, this dot makes the file a "hidden file" that's not visible in the Finder. If you open your project in your text editor, however, you'll see it. In this file, you can list all the files and paths you want to ignore. Because developers often ignore the same types of files, there are .gitignore templates you can use depending on whatever language your project is built in.

To add file names or paths to this file, simply list them each on their own separate line. The order doesn't matter.

Pulling

Most of the time when working with Git, you'll store your repository on a server (e.g., on GitHub). This has two main benefits: easier collaboration and an automated backup of our project. Collaborating with Git will, of course, require a server, but Git can work 100% offline, as well.

Pulling involves grabbing the latest changes from the server and merging them into the branch you're currently working on. This will update your local copy of the project to the same state as the project on the server.

Remember, Git works like a cloud-based system that many people can be editing. If someone makes changes to the files on the server while you're working on your own copy of the files locally, you'll need to update your local copy to capture those changes before you merge your own copy with the files on the server. In other words, you need to sync your copy of the files with any changes that have been made on the server.

Pushing

Once you've made some changes and created one or multiple commits, you can push them to the server. This will update the branch on the server to the same state as your local branch.

Note that this will only push commits to the server. If you've made some changes on your computer but haven't committed them yet, they won't be included in the push. Always be sure to commit your changes!

Merging

The last important concept in Git is merging. While you can always create a new branch, you can also merge branches together. Simply put, this will take all the changes you've made in one branch and apply them to another branch.

Going back to the previous example, once you've finished with your experimental feature1 changes, you'll want to put them back on the "main" branch (as that's where you'll continue your work and, later, publish the new feature for your users). You can achieve this by merging "feature1" into "main."

As you might imagine, merging can be a rather complex process behind the scenes. For example, what happens when you've changed a file on "main" and made a different change on "feature1"—which change will win out? In such cases, Git has a process called "resolving merge conflicts" that allows you to determine which changes from which files should end up in the final version. Keep in mind that wherever possible, Git will automatically merge files for you. However, you should always try to sync your branches as often as possible anyway. "Syncing" ensures you have the most up-to-date version of the code. If you don't sync often, more possibilities for conflicts will pop up, which can be very time-consuming to resolve!

In summary, using Git for version control requires that you save your work with a commit (and commit your changes with an appropriate commit message) and then push your local changes to the main repository. When you're working on a complex project with other developers, you'll also need to sync your copy of the project in order to pull down any changes that other contributors may have made, and you'll sometimes need to merge different branches to keep the work seamlessly integrated.

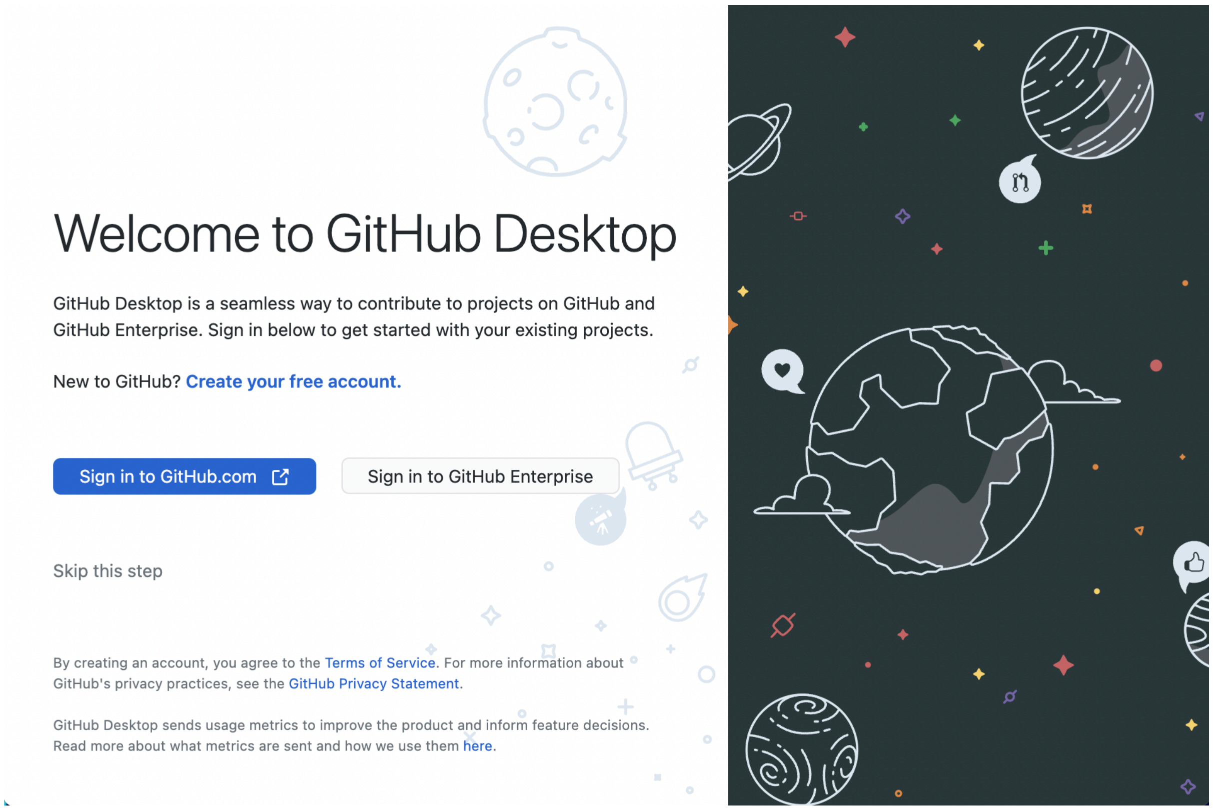
Working with Git

Whew! That was quite a lot to take in. Take some time to review the terms defined above until you've gotten a good grasp of what they mean. You'll be seeing them a lot throughout the rest of the Exercise.

It's time to try out Git for yourself and see how you can use it in your project! You can go about using Git in a number of ways, such as using the command line, which is a direct line to your computer's operating system. This approach is very powerful, but it requires a basic understanding of command line syntax to get started quickly. You won't be looking at the command line approach for this Exercise, but if you're already familiar with it and prefer to use it, feel free to do so. You might want to check out this simple guide to Git, which has a list of commands for the tasks that will be discussed in the upcoming sections.
For now, since you're just getting the hang of how version control works, you'll be taking a look at Git clients with their own user interfaces, instead.

As far as Git clients go, there are quite a few options, and you can find a fairly comprehensive list on the Git project page. For your purposes, however, you can stick to GitHub Desktop.

Before installing GitHub Desktop, you'll first need to create a free GitHub account. Don't worry about what exactly GitHub is at the moment—you'll learn all about it in the coming sections. Once you're done creating your account, download GitHub Desktop (available for OS X and Windows) and start the setup. During the installation, you'll be asked to sign in with your GitHub.com account (this is why you needed to create an account!), so go ahead and do so:


Welcome to GitHub Desktop

GitHub Desktop is a seamless way to contribute to projects on GitHub and GitHub Enterprise. Sign in below to get started with your existing projects.

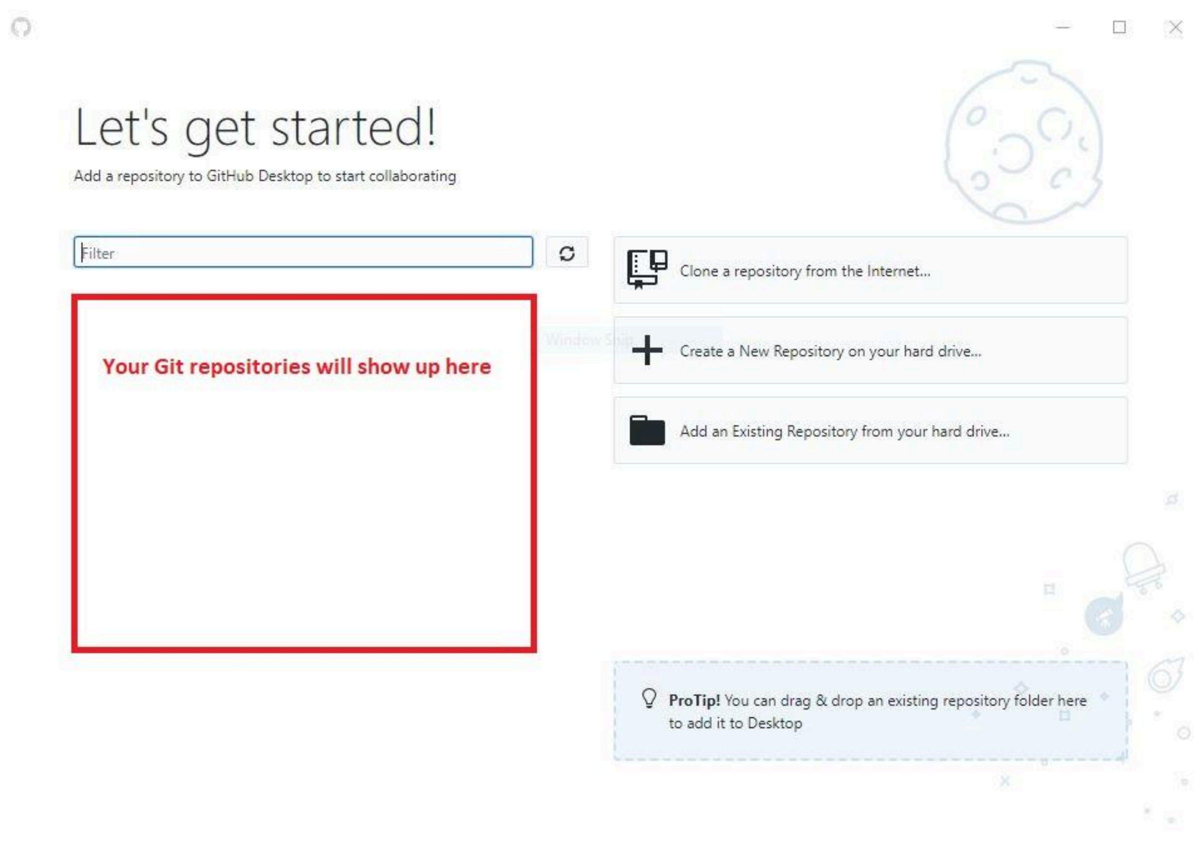
New to GitHub? Create your free account.

By creating an account, you agree to the Terms of Service. For more information about GitHub's privacy practices, see the GitHub Privacy Statement.

GitHub Desktop sends usage metrics to improve the product and inform feature decisions. Read more about what metrics are sent and how we use them here.

Figure 3. GitHub Desktop Welcome Page.

Once logged in, the program will ask you to share usage stats. This is up to you to decide. For now, turn your attention to the startup screen (if the startup screen doesn't pop up automatically, you'll need to start the software manually):



Let's get started!

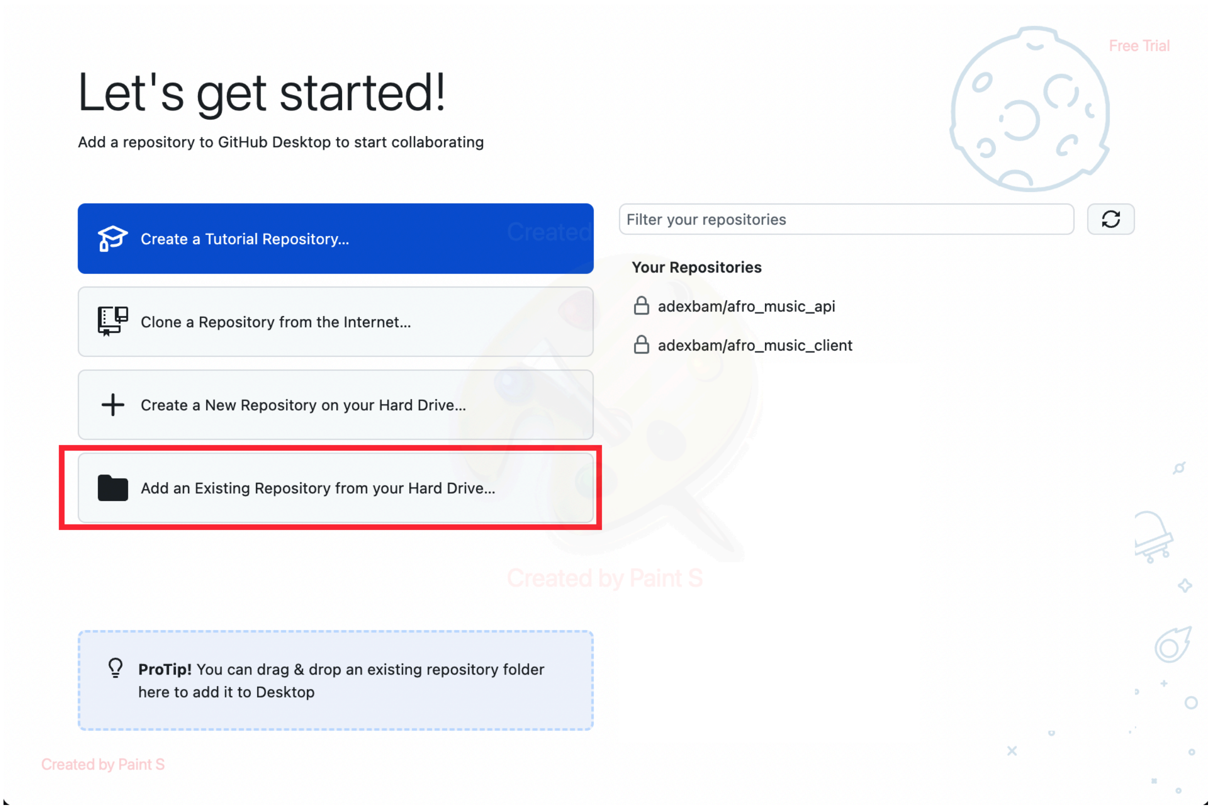
Add a repository to GitHub Desktop to start collaborating

Filter
Your Git repositories will show up here
Clone a repository from the Internet...
Create a New Repository on your hard drive...
Add an Existing Repository from your hard drive...

ProTip! You can drag & drop an existing repository folder here to add it to Desktop

Figure 4. Github Desktop start screen.

The first thing you need to do is initialize Git for your portfolio website project. To do so, click on "Add an Existing Repository from your hard drive...":



Free Trial

Let's get started!

Add a repository to GitHub Desktop to start collaborating

Create a Tutorial Repository...
Clone a Repository from the Internet...
Create a New Repository on your Hard Drive...
Add an Existing Repository from your Hard Drive...
Filter your repositories

Your Repositories

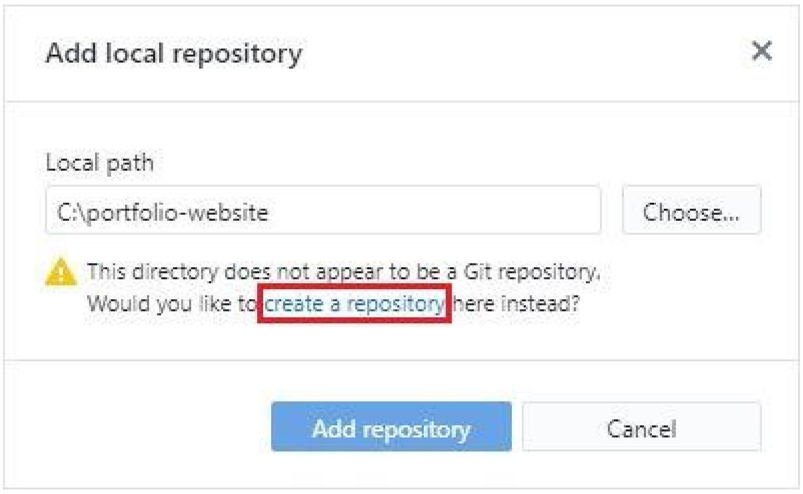
adexbam/afro_music_api

adexbam/afro_music_client

ProTip! You can drag & drop an existing repository folder here to add it to Desktop

Figure 5. Adding an existing project repository to GitHub.

Then, select the folder in which your project lies. A message will show up telling you to "create a repository" because Git hasn't been initialized in your project folder yet:



Add local repository

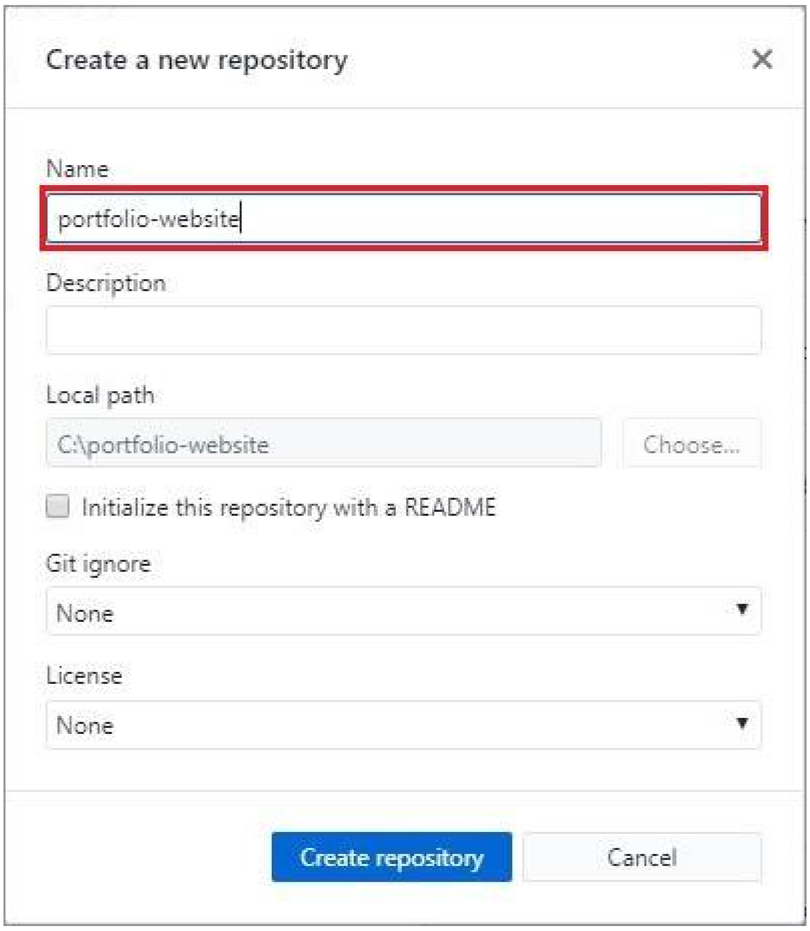
Local path

C:\portfolio-website

This directory does not appear to be a Git repository. Would you like to create a repository here instead?

Figure 6. Opening the repository creation form.

Click on "Create a repository" to open the repository creation form. For the repository "Name" field, enter "portfolio-website", then click on "Create Repository" button to initialize Git and create the local repository:


Create a new repository

Name: portfolio-website

Description:

Local path: C:\portfolio-website

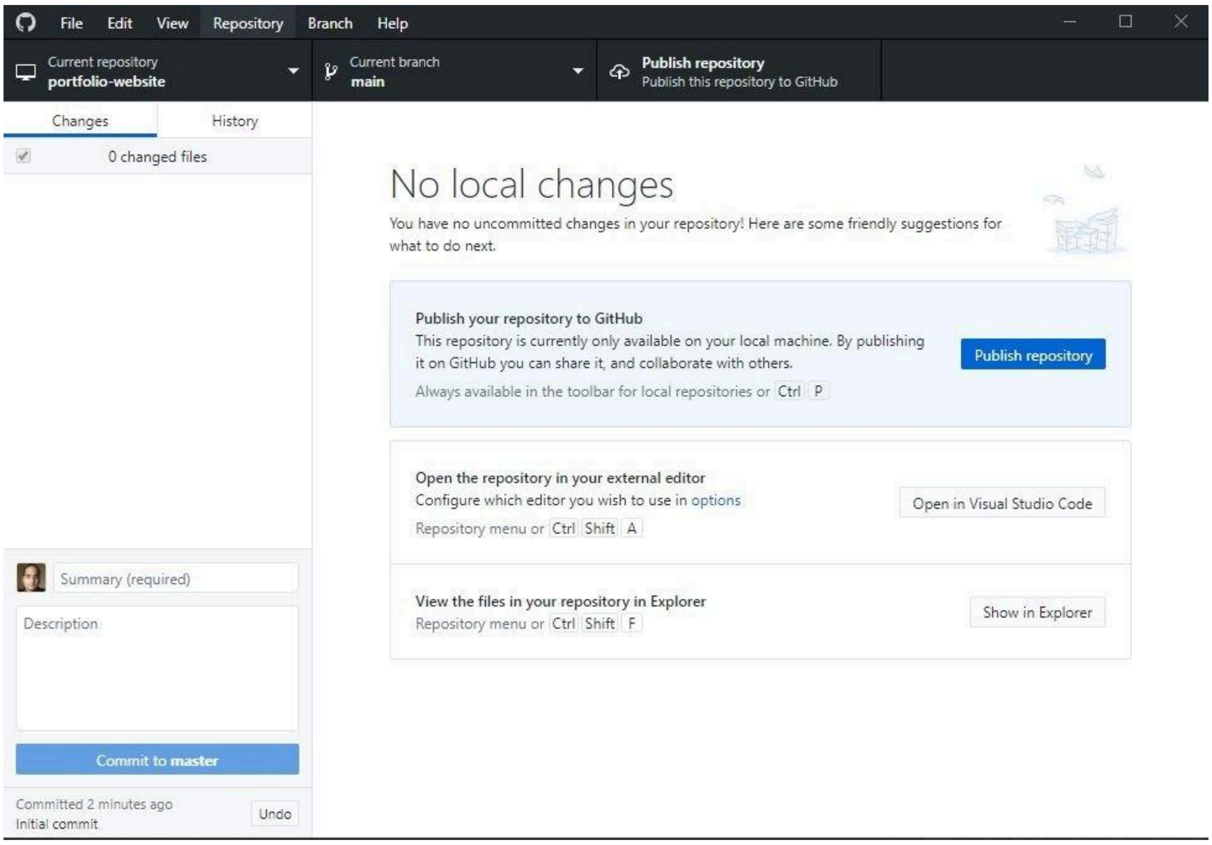
Initialize this repository with a README

Git ignore: None

License: None

Figure 7. Naming the portfolio project's repository.

This will bring up the workspace window:



No local changes

You have no uncommitted changes in your repository! Here are some friendly suggestions for what to do next.

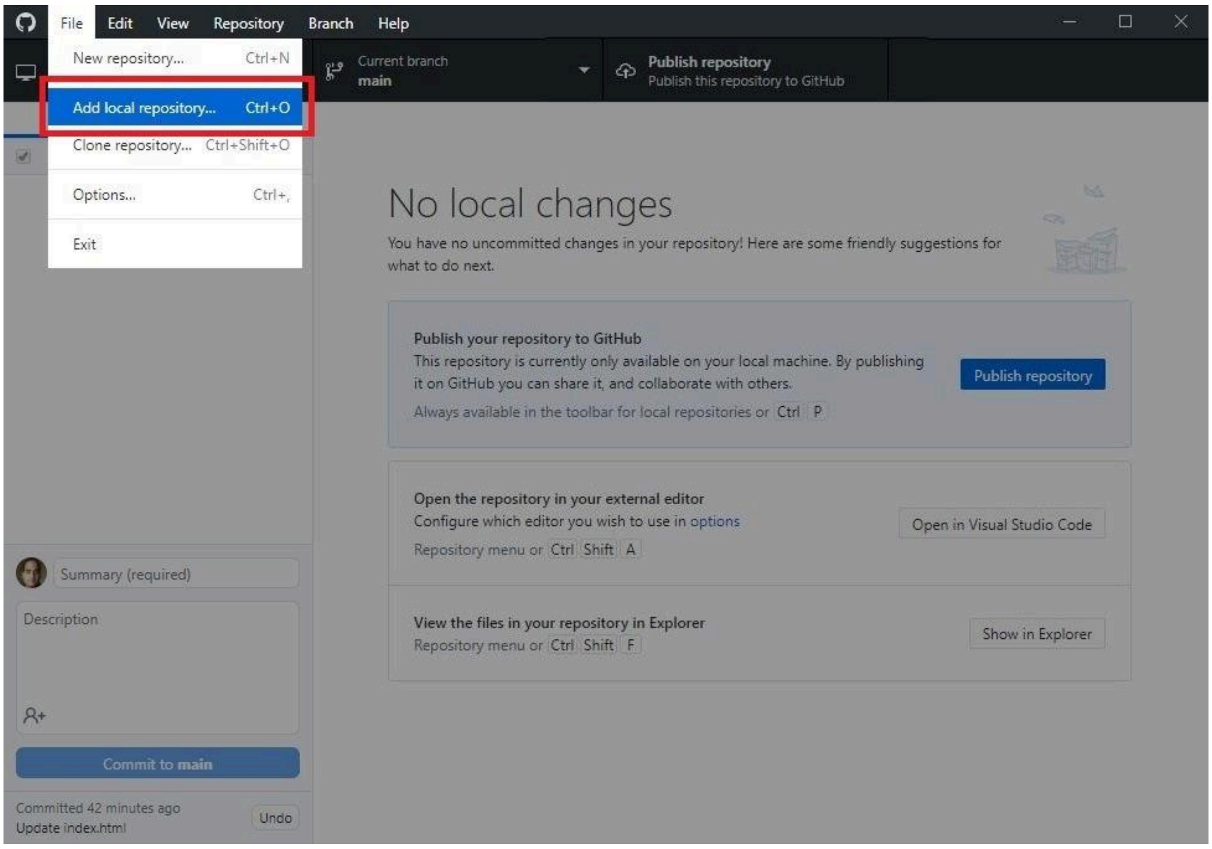
Publish your repository to GitHub
This repository is currently only available on your local machine. By publishing it on GitHub you can share it, and collaborate with others.
Always available in the toolbar for local repositories or Ctrl P

Open the repository in your external editor
Configure which editor you wish to use in options
Repository menu or Ctrl Shift A

View the files in your repository in Explorer
Repository menu or Ctrl Shift F

Figure 8. GitHub workspace window.

It's important to note here that you can access the same repository functions shown on the startup screen (e.g., for your future projects) via the "File" menu:



No local changes

You have no uncommitted changes in your repository! Here are some friendly suggestions for what to do next.

Publish your repository to GitHub
This repository is currently only available on your local machine. By publishing it on GitHub you can share it, and collaborate with others.
Always available in the toolbar for local repositories or Ctrl P

Open the repository in your external editor
Configure which editor you wish to use in options
Repository menu or Ctrl Shift A

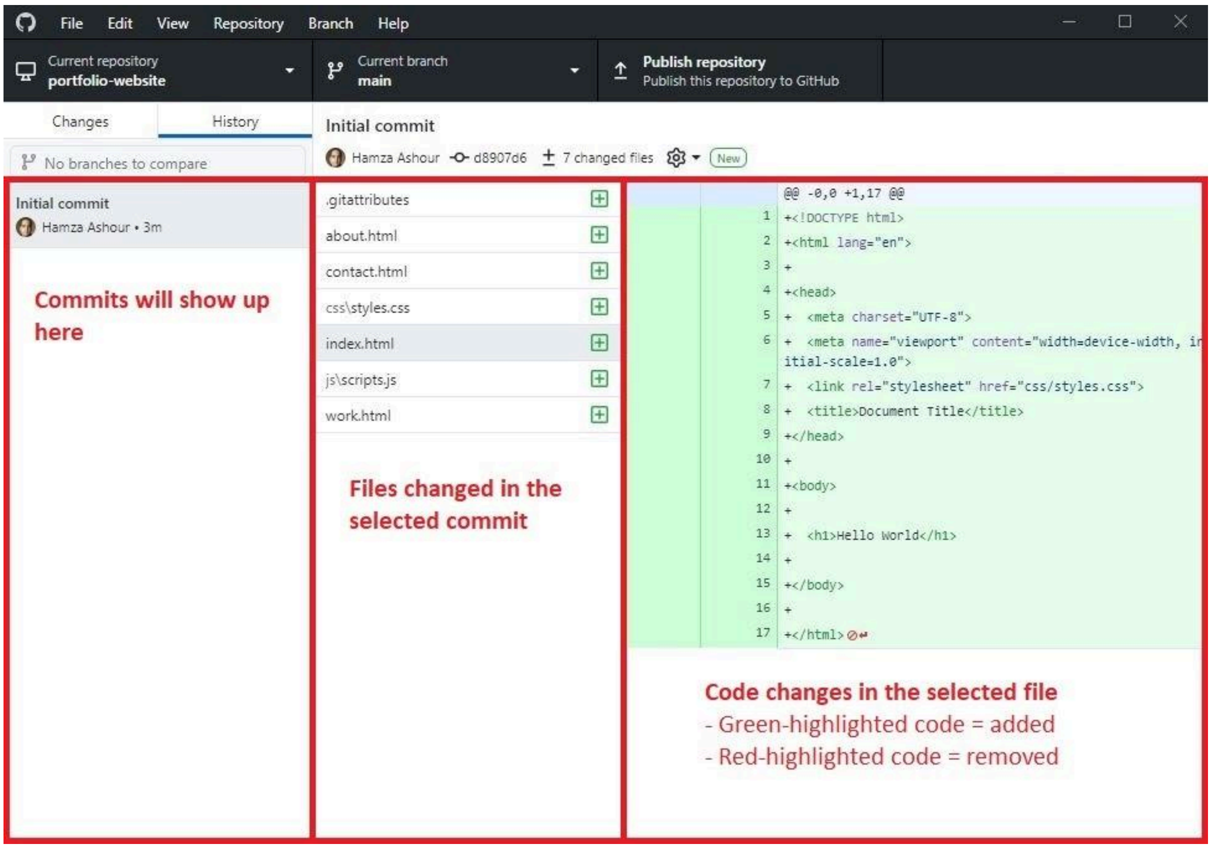
View the files in your repository in Explorer
Repository menu or Ctrl Shift F

Committed 42 minutes ago
Update index.html

Figure 9. GitHub's "File" menu also contains repository functions, including "Add local repository".

Let's start by exploring the UI. Any new changes made in a file within the repository (and which are awaiting an action from you) will be shown in the left-hand pane under the "Changes" tab. There, you can decide what you want to do with those changes. For instance, do you want to commit the change? Delete it? Was the change an accident and you want to revert it? Is there a conflict with another file?

When you add a new repository that hasn't been initialized before, GitHub Desktop will, by default, create an initial commit that adds your files to the repository. You can view a list of all commits made in your repository by clicking on the "History" tab. Click on a single commit in the left-hand pane to bring up a list of the individual changes within that commit on the right. The right-hand pane is further divided into two columns—the left of which lists all files that contain changes, and the right of which shows the actual code changes for each file. This makes it easy to 1) click into an individual *commit, 2) click into an individual file within a commit, and 3) check the changes that have been made to that file:*



```
@@ -0,0 +1,17 @@
+<!DOCTYPE html>
+<html lang="en">
+
+<head>
+  <meta charset="UTF-8">
+  <meta name="viewport" content="width=device-width, in
itial-scale=1.0">
+  <link rel="stylesheet" href="css/styles.css">
+  <title>Document Title</title>
+</head>
+
+<body>
+
+  <h1>Hello World</h1>
+
+</body>
+
+</html>
```

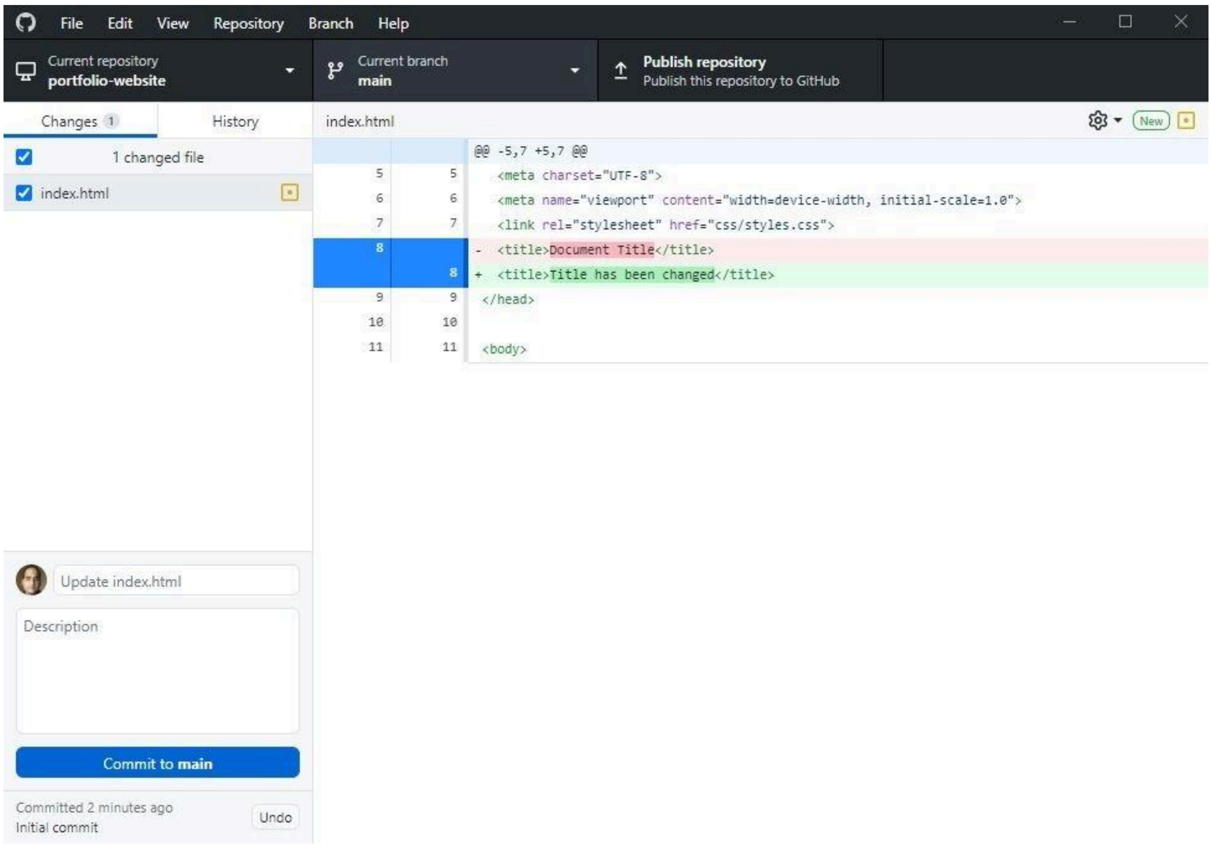
Commits will show up here

Files changed in the selected commit

Code changes in the selected file
- Green-highlighted code = added
- Red-highlighted code = removed

Figure 10. GitHub "History" tab explained.

Next, you'll want to create your first (real) commit for this project. To do so, you'll need to make some changes in a file (otherwise, nothing will show up in GitHub Desktop). Go ahead and make a change in your "index.html" page; for instance, you could change the `<title>` of your page or another similar small change, then save the changes in your text editor. Now, go back to GitHub Desktop and click back into the "Changes" tab in the left-hand pane. The new change you just made should pop up!



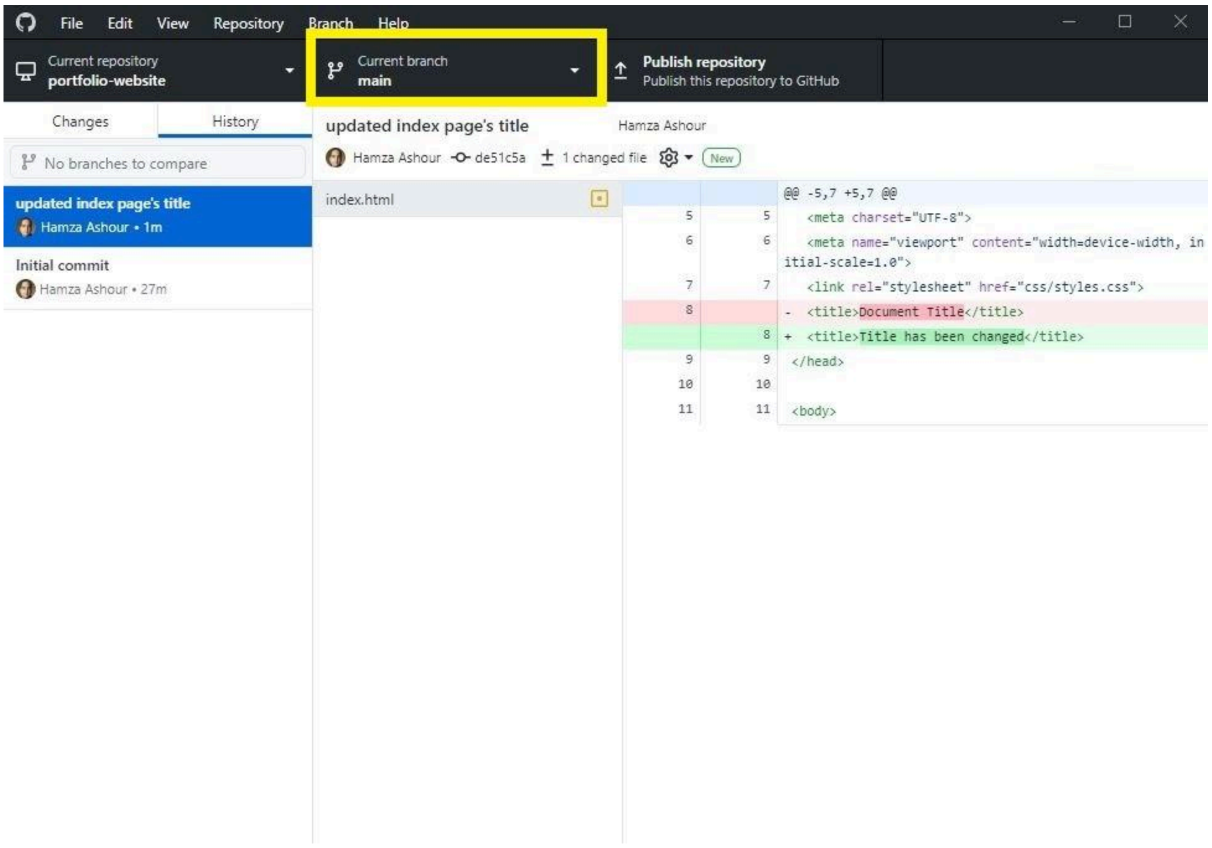
```
@@ -5,7 +5,7 @@
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="css/styles.css">
-   <title>Document Title</title>
+   <title>Title has been changed</title>
  </head>

  <body>
```

Figure 11. GitHub's "Changes" tab, which shows the changes that have been made.

You may have noticed that both the new code (highlighted in green) and the old code (highlighted in red/pink) are both displayed in the right pane when you select "index.html" in the left pane. This will help greatly in keeping track of what changes you made in which files. But there's still more you need to do—actually commit your changes.

First, write a new commit message for your change. Make sure that the checkbox next to file's name is ticked, then enter a descriptive message in the field below. (Think back to the first half of this Exercise, where you learned how to write commit messages.) You can add a description, as well, if you want, though this usually isn't necessary unless there's something very specific you want to point out in the commit. Once done, hit the "Commit to main" button, and the changes should disappear from the "Changes" tab pane. Now, open the "History" tab and select "main" branch as your current branch (selectable via the "Current branch" button highlighted in yellow in the image below). You should now see all the commits that have been made on the main branch in the left-hand pane, which, for you, will likely just be your "Initial commit" and the commit with your most recent change:

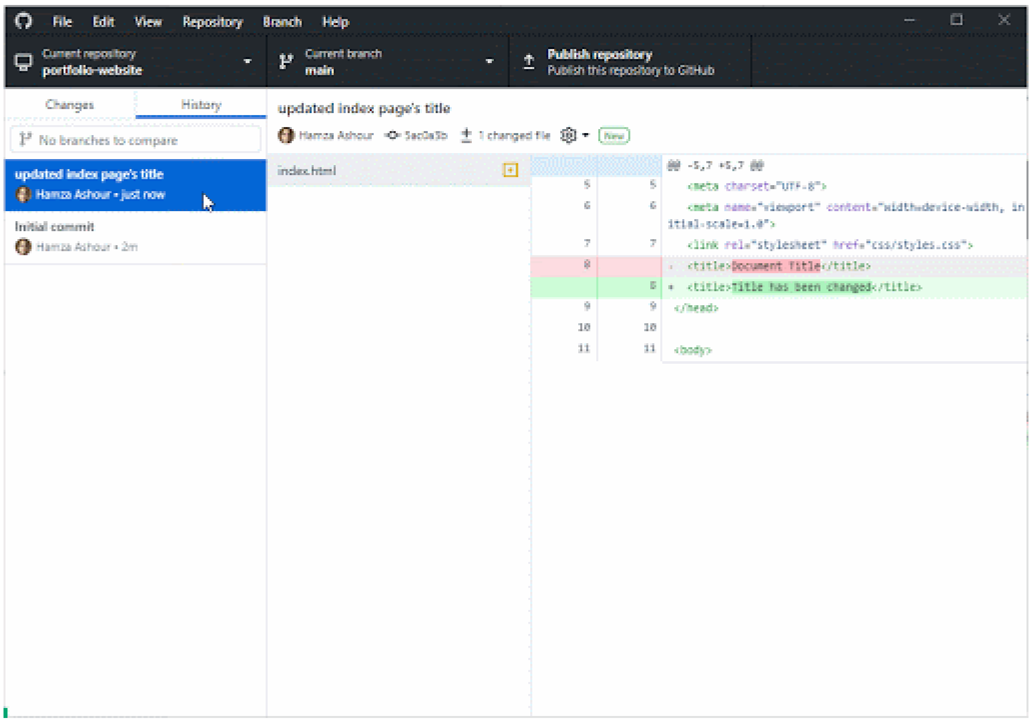


```
@@ -5,7 +5,7 @@
     <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <link rel="stylesheet" href="css/styles.css">
-    <title>Document Title</title>
+    <title>Title has been changed</title>
 </head>

 <body>
```

Figure 12. GitHub's "Current branch" view, with all commits visible.

Of course, since you only made this change now to play around with Git, you'll want to go back to the previous state before that change was made (known as "reverting"). To do so, simply right-click on the commit and select Revert changes in commit:



```
@@ -5,7 +5,7 @@
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <link rel="stylesheet" href="css/styles.css">
-  <title>Document Title</title>
+  <title>Title has been changed</title>
 </head>

 <body>
```

Figure 13. How to revert changes.

You should now see another commit with the same message as the original one, only this time, prefixed with "Revert." For example, if the original commit message were "Changed the title of homepage," then the new reverting commit will have the commit message "Revert 'Changed the title of homepage.' " When you look into this commit, you'll see that the title was changed back to what it was before. This is an easy way to roll back accidental changes you may have made.

While this is handy to know, there are also certain times where you shouldn't revert changes. In the example above, the commit only had a single, small change in it. Reverting this change would be easy and simple and wouldn't affect anything else in your files. If, however, the change that you wanted to revert were part of a large commit with a wide variety of changes, you wouldn't want to revert the entire commit, as this could lead to potential problems and simply create more work. In this case, you could make a brand-new commit on top of the old commit that only changes back the one thing you want to "revert." This way, you keep all the parts of the original commit that you want to keep and only change what actually needs to be reverted.

TIP!

If you're using an operating system other than Windows (i.e., Linux or Mac), the screenshots above may look a bit different than what you see on your screen; however, the basic UI and functionality remain the same. If . If anything's unclear, however, don't hesitate to reach out to your tutor.

Version Control with GitHub

So far, you've learned some of the key vocabulary surrounding Git. You've also created a Git repository on your computer for your project; however, all of this so far has only been living on your computer—what if you want to collaborate on a project with a team? And what if you want to keep a copy of your source code online as a backup?

For that, there are several services that allow you to host your Git project on their servers. The most famous is GitHub, which is also what you'll be using for the remainder of this course. Other good options include Bitbucket or GitLab.

While Git itself is open source and can be used by anyone, online services built around Git aren't necessarily free. GitHub, for instance, (which, despite its name, isn't directly affiliated with Git) allows you to host public and private projects for free so long as you don't have more than three collaborators. This means that, for your purposes as an up-and-coming developer, GitHub is completely free; however, if you were working in a company, that company would likely have to pay for an account, as they would need to allow multiple team members to collaborate on the same company repository. Regardless, GitHub is still the solution with the highest mind and market share, which is why you'll be using it instead of its competitors for this course.

Perk: GitHub for Education

As a CareerFoundry student you're also eligible for a GitHub Education account. This gives you free access to

GitHub Pro and also GitHub's Student Developer Pack, which offers discounts and free access to an awesome array of developer tools and services. Head to the Perks tab in Settings on the platform for information on how to redeem this perk.

As a quick note before moving on, be careful with your use of terminology! As many of the terms being used are similar to each other, you'll want to ensure you're always referring to the correct thing:

- "Git" is the actual version control tool.
- "GitHub" is an online repository where you'll be hosting the code for your portfolio page project (rather than just on your local computer).
- "GitHub Desktop" is the application that utilizes Git under the hood. What it does essentially is provide a clean, easy-to-use UI for using the Git version control tool to upload projects to GitHub, the online repository (which you'll be learning all about in the next section!).

Publishing Your Local Repository

Head on back to GitHub Desktop and, with your project open, click on "Repository" in the navigation bar and. From there, select "Repository settings..." from the bottom of the pulldown list:

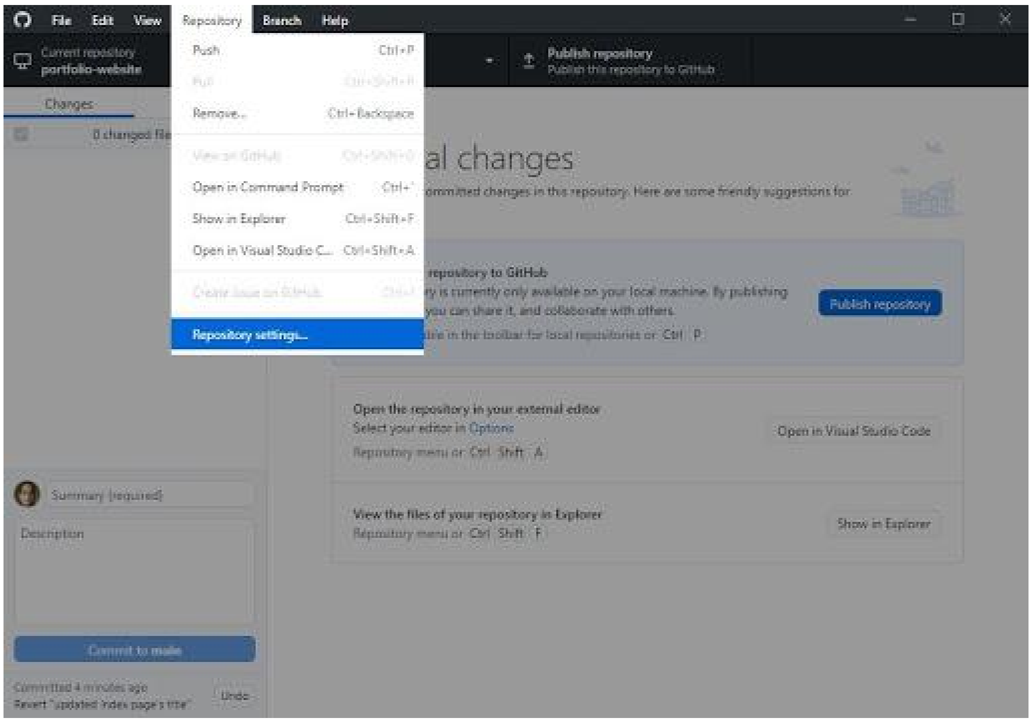
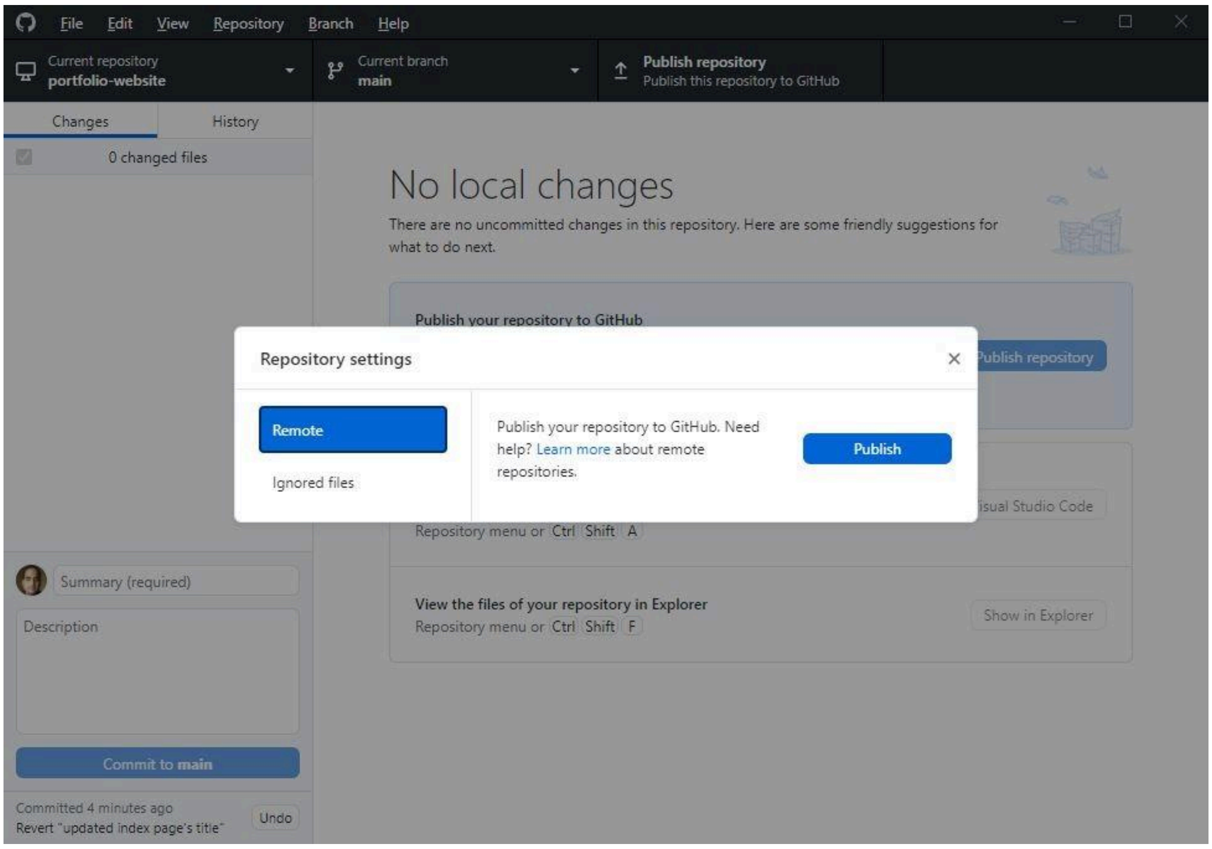


Figure 14. Selecting "Repository settings" from GitHub's "Repository" menu.

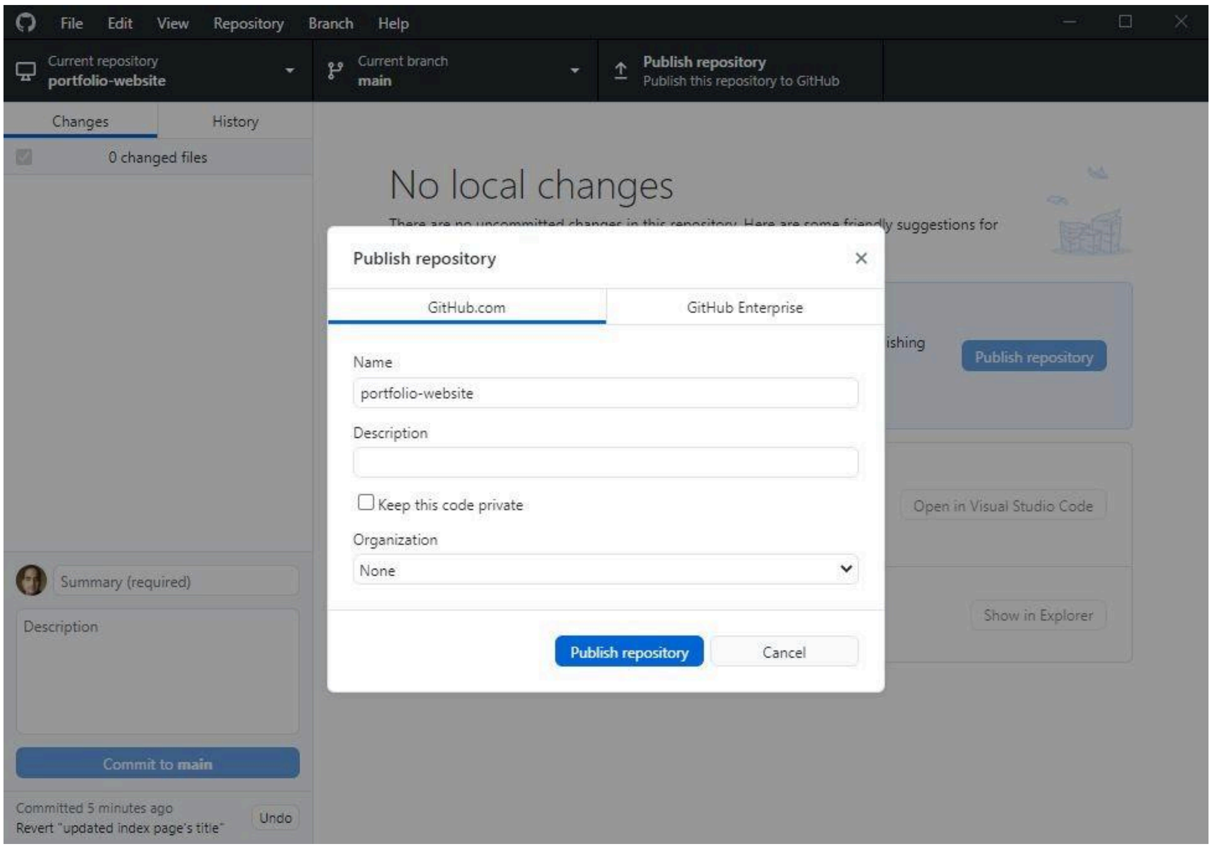
This will open the following pop-up window:



Publish your repository to GitHub. Need help? Learn more about remote repositories.

Figure 15. "Repository settings" popup, with "Publish" button.

While the "Remote" tab on the left is selected, click "Publish." (The term "Remote," here, describes the server where you'll be storing your Git project, which is separated, or "remote," from the local repository on your computer.). This will open up another pop-up window, allowing you to name your new online repository. Give it a name such as "portfolio-website." Also, make sure the "Keep this code private" checkbox is unticked. Otherwise, you won't be able to share your repository with your Tutor and Mentor! Finally, click "Publish repository":



Name: portfolio-website

Description:

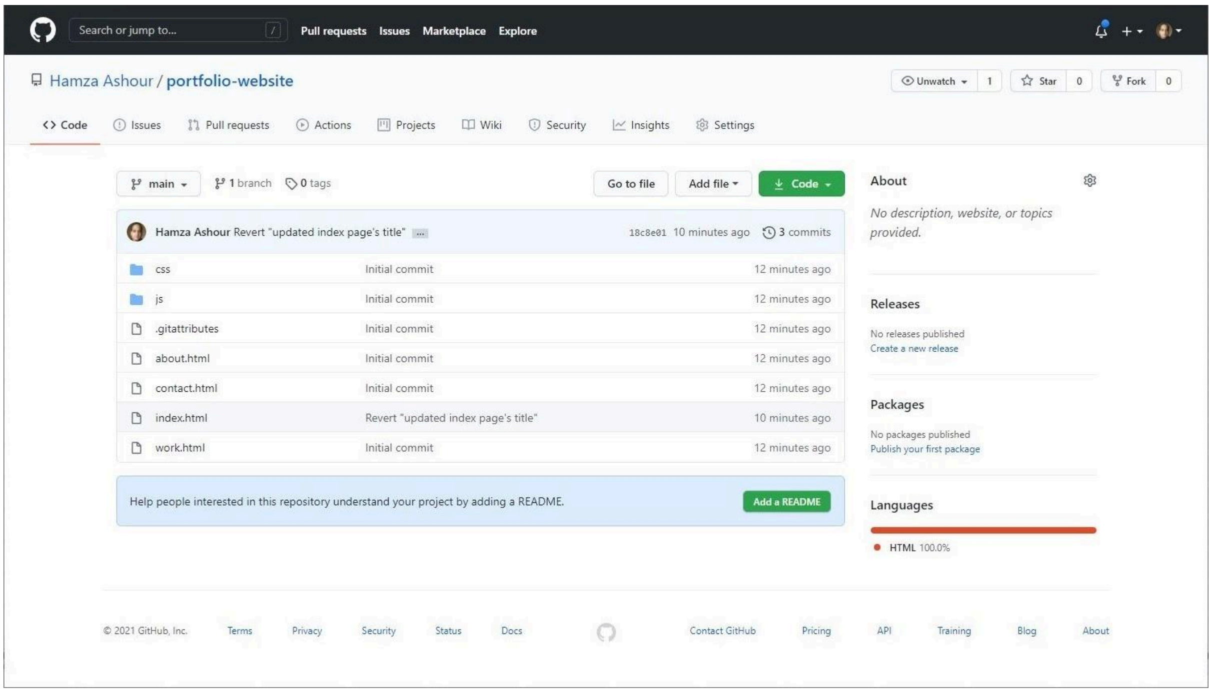
☐ Keep this code private

Organization: None

Figure 16. Pop-up modal confirming repository's details before publishing.

After a few moments, your online repository will be ready to go, and GitHub Desktop will automatically push the commits you already made.

At this stage, your local files should now all be uploaded to your online repository. To see for yourself, sign in with your GitHub account in your browser, then select "Your repositories" from the dropdown menu that appears when you click on your avatar image in the top navigation bar. You should see your new repository, "portfolio-website," there, and if you click on it, you'll be taken to your repository dashboard:



Hamza Ashour Revert "updated index page's title" ... 18c8e01 10 minutes ago 3 commits

css	Initial commit	12 minutes ago
js	Initial commit	12 minutes ago
.gitattributes	Initial commit	12 minutes ago
about.html	Initial commit	12 minutes ago
contact.html	Initial commit	12 minutes ago
index.html	Revert "updated index page's title"	10 minutes ago
work.html	Initial commit	12 minutes ago

Help people interested in this repository understand your project by adding a README.

No description, website, or topics provided.

No releases published
Create a new release

No packages published
Publish your first package

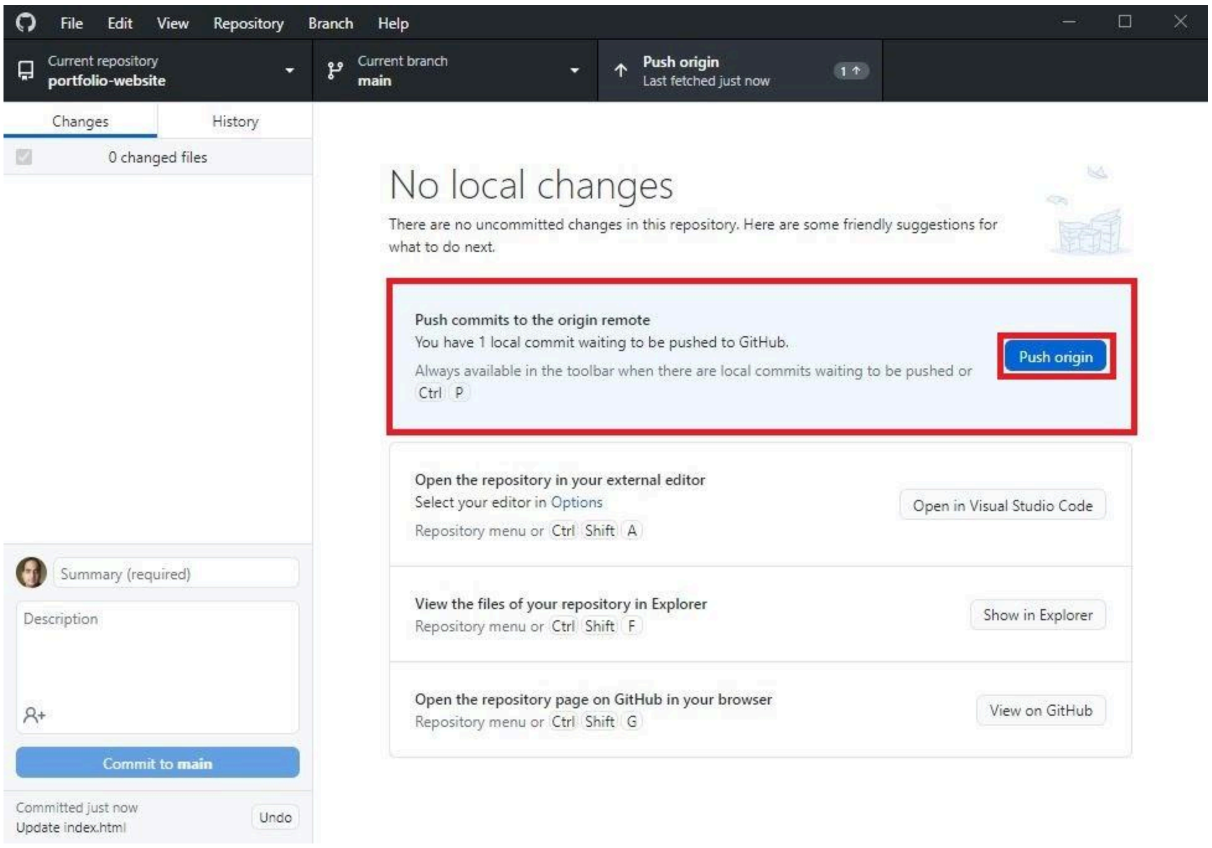
Figure 17. GitHub repository dashboard.

Amazing! You can now see all your files, when they were last changed, as well as the commit message of the most recent commit that affected each file.

Try It!

Take a little time to play around with the GitHub UI. Try clicking on "3 commits," for example, to see a list of commits or check out the file explorer. The more familiar you become with GitHub now, the easier you'll find it to work with as you make progress on your course project.

Each time you make additional changes to your website, you must push them to GitHub after committing them in order to update the online repository on GitHub. To test this, try changing the title of your "index.html" file once again (you may revert this change later), then commit the change in GitHub Desktop. Once the change is committed, GitHub Desktop will inform you that there's a local commit waiting to be pushed. Also, the "Fetch origin" button at the top should have changed to a "Push origin" button:



No local changes

There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.

Push commits to the origin remote
You have 1 local commit waiting to be pushed to GitHub.
Always available in the toolbar when there are local commits waiting to be pushed or Ctrl P

Open the repository in your external editor
Select your editor in Options
Repository menu or Ctrl Shift A

View the files of your repository in Explorer
Repository menu or Ctrl Shift F

Open the repository page on GitHub in your browser
Repository menu or Ctrl Shift G

Figure 18. GitHub Desktop's "Push origin" button, which updates the online repository when pressed.

Give that button a click now. It will take a few seconds to push the latest commited changes into your online repository. Once done, go back to the dashboard for your GitHub repository in the browser. The number of commits should have been updated, and, if you directly open your "index.html" file, you'll see that it contains the change you just committed.

The process of making changes to your project files can be summarized as follows:

1. Make changes to your files in your preferred text editor (e.g., VS Code)
2. On GitHub Desktop, add a commit message that describes the changes you just made, then commit the changes.

3. Still on GitHub Desktop, push the changes to your online repository via the "Push origin" button.
4. (Optional) Open GitHub in your browser and check that the changes in your commit are now reflected in your repository.

More on GitHub

One cool thing GitHub allows you to do is collaborate with others on the same project. You can even invite other people to collaborate on a project in GitHub by going to "Settings"→ "Collaborators."

What if, however, someone else were currently working on the same project as you, committing new changes of their own? In order to make sure their changes show up in your local version of the project, you'd need to do a "pull" before starting to work on the project with their latest changes incorporated.

You can also share the URL to your project. So long as your project is public on GitHub, anyone with the link can view it. You'll notice that GitHub URLs always follow the same structure: "https://github.com/USERNAME/PROJECTNAME".

Checking out other developers' public projects is actually a great way to learn some coding best practices and stay on top of web development trends. All you have to do is locate their profile in the list of public repositories.

You can treat your GitHub account as an extended portfolio where potential employers can go and investigate the quality of your code. Employers may also look at your GitHub account to see if you have recent activity—in other words, to see if you're keeping your skills and portfolio up to date. Remember, as you create more projects, you can also link to these projects and to your GitHub page from your portfolio website.

Summary

In this Exercise, you learned a great deal about version control and Git, from the most important terms you'll need to know along with its basic functions. You also set up a Git project on your computer and made some small changes to it to practice working with commits. You learned how to easily view your changes, move between branches, and revert commits, as well as how to set up a project for collaboration on GitHub. Finally, you looked at how to push changes to GitHub and view and share project files online.

In the next Exercise, you'll wrap up everything you've learned in this course by talking about testing—namely, cross-browser testing. You'll also take a look at how you can improve the quality of your code via tools such as the W3C validator and code linters. For now, let's take what you've learned and put it into practice by setting up a GitHub account of your very own!

Resources

If you're curious to read more about the topics covered in this Exercise, then we recommend taking a look at the resources below. Note that this reading is optional and not required to complete the course.

Git:

- Getting Started with Git
- Git Documentation

- Understanding the Github flow

GitHub Desktop Documentation:

- Getting Started with GitHub Desktop

Writing Commit Messages:

- How to Write a Git Commit Message
- The Art of Writing a Good Commit Message

You scored 3/5 on this Exercise last time.

Take Quiz Again See Details

Task

Estimated Task Time: 1 - 3 hours.

In this Task, you'll set up a GitHub account for your project and make some changes to your website before pushing them to GitHub. You'll also work on finishing up your portfolio website by replacing any remaining placeholder text and images with your own content.

Important Note!
If, at this point in the course, you're uncomfortable uploading your real name and photo to your portfolio site, you're free to use an alias and placeholder image. We know that many of our students are already working jobs while taking this course, and that releasing info about wanting to change careers could lead to trouble with a current employer. If you're worried about this, you're free to use an alias until you're ready to start promoting your site and begin your job search. In this case, please still follow the directions for the Task as closely as possible, but switch out any mention of your real name with an alias (and find a stock photo or other image to use as your profile picture). This way, once you're ready to release your portfolio site into the wild at the end of the course, you'll be able to switch to your real information quickly.

Directions

1. Set up a GitHub account.

    - Head to the Perks tab in your Settings of the CareerFoundry platform for information on how to sign up for a GitHub Education account. This gives you free and discounted access to an array of awesome developer tools!
2. Download and install GitHub Desktop.
3. Create a new project for your website in GitHub Desktop.
4. Publish your local repository to an online repository called "website-portfolio" (or something similar) on GitHub.
5. Make some real changes to your project:
    - Change the page title of your "contact.html" file to something indicating that this is a contact page, for example "Contact Lisa Gringl," then commit that change using a reasonable commit message.
    - Change the page title of your "about.html" file in the same way.
    - If you haven't done so yet, replace the placeholder titles, logo (text or image), and project images in all your HTML files.
    - If you haven't done so yet, finalize the text on your pages, as well.
6. Commit the changes you've made to your project. Make sure to follow the standards you learned when creating commit messages.
7. Push your changes to GitHub.
8. Submit the GitHub repository link to your project here. Feel free to share additional thoughts or ask questions on your submission page.

Rubric

Refer to the categories below to see how to meet the requirements of the approved stage

Not Yet

GitHub link doesn't include a portfolio project repo, the portfolio project repo is empty, or the repo shared is unrelated to the student project; OR

GitHub link hasn't been provided

A Little More

GitHub link shows correct project repo with sufficient commits made, but project repo is poorly named, commit messages are poorly written, insufficient changes have been made to portfolio page titles, and placeholder content hasn't been replaced; OR

GitHub link shows correctly named and synced project repo, but insufficient commits have been made to project

Almost There

GitHub link shows project repo with multiple commits

Repo is named incorrectly or commit messages are incorrect; OR

Portfolio page titles haven't been renamed or have been inappropriately renamed; OR

Not all placeholder text and imagery has been replaced with final content

Approved

GitHub link shows correctly named and synced project repo and multiple appropriate commit messages

Portfolio page titles have been appropriately renamed

All placeholder text and imagery has been replaced with final content