December 7, 2021

GitHub: Getting Started

Git and GitHub

Now that we have set the expectations right, let's look at what Git and GitHub are and what is their relation. Git is, in essence, a source control system, a very popular one as already mentioned and very widely adopted one as well. It allows many users, even distributed, to work on small and large software projects in a very simple, yet powerful way. Git is a distributed source management system. Source control systems, such as Team Foundation Version Control, are centralized version control systems. In a nutshell, being distributed basically means that instead of having a single place for all the sources, every developer also has a full history of all the changes. It was created by Linus Torvalds and is itself, of course, also open source and completely free to use. Using Git as your source management system has, of course, quite a few advantages. For starters, it is fast, really fast and scalable. Performance tests have shown that compared to other source management systems, it's a lot faster. Since it is distributed, you'll often work with a local history, and that too is a lot faster than working with a remote server. Now since Git is a distributed source management system, it can also work disconnected. Distributed means, in essence, that every developer has a full local copy of the complete repository. All your work is done in the local repository, and only when you are satisfied with the work you have done, you can synchronize with the remote server. Centralized source management systems also can cause havoc when someone breaks the production branch. With distributed source management systems, everyone can continue working on a local branch. Git is very powerful, yet it's easy to use. In the beginning, when coming from other source control management systems, often developers need to make a paradigm shift. Once you have taken that step though and learned the basic commands, you'll become a Git wizard in no time. Using branches is another benefit of Git. In fact, the entire typical workflow in Git is based around branching. Development typically is done on the feature branch, and only when the level of quality is where it should be will we be merging back into the main branch. This way, the main branch always contains quality code. Very closely related to this is the concept of pull requests. It's not really a Git feature, but I'm adding it here anyway. Pull requests are typically added on top of the core Git feature set by sites, such as GitHub. It enables collaborating on code. Using a pull request, a developer can ask for a review and a merge back into another branch of the changes that he or she has performed. This intrinsically brings discussion and reviews with it, thus improving the quality of the code. Now, it's not all perfect in the land of Git. There are certainly some challenges to take when working with Git. For starters, yes, it is different, although that can be a good thing. If you have been working with a traditional source management system for some years, you'll need to make a mind switch. It has multiple local levels and a remote typically. But as in traditional centralized version control systems, we only have a local working directory, and changes are pushed to remote. That is basically it. Developers working with Git also complain about the learning curve they have to go up when learning to work with Git. As said, some things are different, and sometimes things we know in other systems have a different meaning in Git. For example, checking out has a different meaning in Git than it does in other source management systems. Adding to the learning curve is sometimes the available tooling. Git is most powerful when used from the command line, and that isn't always easy for developers coming from other systems where a powerful GUI could help. No doubt for Git, several GUI clients also exist, but Git really shines when using it from the command line. In this course, we'll be using the command line as well. Finally, one area where Git also suffers a bit is large binary files. But then again, other source control management systems have issues with this very often. Now that we know what Git is all about, I want to make sure that you see GitHub in combination with Git. As mentioned, we'll start the exploration of GitHub in the next module, but I already wanted to make things as clear as possible here. GitHub is a Git‑hosting site. It provides a hosting service for Git repositories, and it's used by many developers for personal projects, as well as numerous companies around the globe for small and large software projects. GitHub runs on top of Git, so it will add many more things. Of course, source management is one of the key concepts here, but Git offers a broad variety of extra services, including issues management, working with teams, adding a wiki, and so much more. Some source‑based actions can also be done directly in the GitHub interface, such as creating a pull request or performing a merge. Some others still require the use of local tools. GitHub offers a free tier, as well as paid options. It's possible to create both public and private repositories with the free plan, and that's what we will be using in this course. GitHub is the key learning of the schools, so as said, we'll start with the exploration of GitHub with the next module. For the remainder of this module, we'll first explore Git itself, so you will be comfortable using GitHub later on.

Getting Your Machine Ready

Now before we start exploring and using Git, let's make sure your machine has everything installed for use with Git. Again, although we will be using GitHub, not everything is possible in the interface from GitHub, so we'll need to install some local tools as well. Git is available for everyone. It's widely supported on all operating systems. So no matter if you're using Windows, macOS, or Linux, Git is available for your machine. Since we'll be using the command line for our interactions with GitHub in this course, you will be able to follow along with everything I'm doing here no matter what operating system you are using. That's yet another advantage. In terms of what you need and need to install in order to follow along with this course, we can be pretty brief. You'll need to have Git installed locally on your system. Head over to the link shown here on the page, and there you will find the download for your operating system. Next, you'll need an editor since we will be making changes to some files. I'll be using Windows in my demos. And as editor, I'll be using Visual Studio Code. Any editor will basically do. Just use your favorite here. and you'll be good to go. Once we start in the next module with GitHub itself, quite logically, you'll need a GitHub account. We'll talk more about that in the next module, so no need to worry about that just yet.

Demo: Setting up Your Environment

In the first demo of this course, let me show you in a bit more detail which steps you need to follow to set up your environment. To do just anything with GitHub, we'll need Git to be installed. Head over to git‑scm.com /downloads, and there you can download the version for your operating system. I'm running on Windows, so I have downloaded this Windows client here. Next, we will be needing an editor. I'll be using Visual Studio Code throughout the entire course. But if you want to use another editor, feel free to do so as any editor will work. If you want to get hold of Visual Studio Code, head over to code.visualstudio.com/download. And you can see the link here on the slide as well. The nice thing about Visual Studio Code is that it's also available for just any operating system. As mentioned, throughout this course, we will be using Git through the command line. If you have installed Git, it'll come with Git Bash, and that is what we will be using throughout the entire course. Now the first thing I need to do is configure Git locally on my machine. When interacting with Git, everything we'll do will start with the git command, so configuring it as well. If you just time git here, we'll get a whole list of options that we can pass through the git command. Quite a few of these we'll be using throughout the course. Now back to the configuration part. Let me maximize window here, and let's clear the window by typing clear, and I'm back to the configuration part. I want to configure Git, so the command to do that will be git config. What I'll do here is I want to set global settings, so we'll use ‑‑global as a parameter here. Notice that I have prefixed global with two dashes to indicate that you want to edit the global settings. And the first thing I want to change is the username that will appear for all the commits that we'll be doing. So for that, we'll need to pass in that we want to change the username, so type user.name. And then, we'll specify the actual value, and that will be my name, of course. And I suggest you use your own name here. Hit Enter, and then Git will have changed the global username. Next, we'll do the same for email. We'll again use git config, passing in again ‑‑global, indicating that we want to change a global setting, and we'll type user.email, and then I'll pass in my own email address. Now, where are these settings stored? Well, they are stored locally on my machine in a file, and we can actually ask Git to visualize these global settings. By typing, again, git config, but now typing ‑‑edit ‑‑global, Git will open an editor showing my settings that I have just applied, and you can see here that the username and the email has been said like we asked Git to do. Now as you can see, Git already opened Visual Studio Code. That's the editor that I indicated I want to use. Now why did it do that? Well, that's because I set Code as my default editor in Git. Now, depending on the installation, Git will already ask you what you want to set as its default editor. But you can also change it later on. Let me show you how to do that. And the first thing I've done is I've added an environment variable. In Windows, under Environment Variables, I've added Code, Visual Studio Code that is, to the path. If you look here, you'll notice here that indeed the Visual Studio Code directory\bin, in fact, is added to the path. That will trigger when I type the word code here to open Visual Studio Code. Bash will indeed find Visual Studio Code, and it will open that for me. Now that is just Bash. How does Git then know that it needs to use Code as my editor? Well, again, I can use a command to configure it. Let me do that. To set Code as the default editor, we need to again call the git config ‑‑global, and now we specify the value to be core.editor. Then, as the value, we specify that we want to use code, which is known to bash, and we also pass in a few parameters. I specify that I always want to use code in a new window. And secondly, I also specify that I want to use wait as an extra option here. Git will then wait to continue until the code window that has been opened has actually been closed. That can be useful if you're doing editing work in one of the code windows. Let us hit Enter again, and now code has been specified as the default editor with these extra options. If we look again at the global configuration, by calling config with the edit parameter again, we'll indeed see that Code has now been set as our default editor with these extra options specified as well. At this point, Gt has been configured, and Code is now our default editor.

Foundations of Git

With everything installed, we can now take a look at the base concepts of Git. If you are already familiar with how Git works and want to focus purely on GitHub, feel free to skip this section and start with the next module. As mentioned already, one of the main differences when comparing Git to other source management systems is the fact that Git is a distributed source management system. But what is that exactly then? Well, before we look at that, let's first take a look at how a centralized source management system works. In a centralized source code management system, we have a central copy of the source on a server. That central server is the main repository, and it contains all versions of the code. When we want to make changes to the code, we first need to copy the code from the server. This will get the current version of the code down, and that will give you a local copy of that code. When I make changes to the code and that would then be committed again back to the central repository. Once that is done, all the developers can pull down this change, and their files will be updated as well. In the centralized system, we work with changesets. A changeset is a number of changes that is treated as a whole. In a distributed system, things work slightly different. In such a system, a developer will clone the entire repository and therefore get the entire history on their machine. This way of working therefore doesn't require a central store, although typically, there will be one. In our case, that central store will be GitHub. Because each developer has a full copy of the entire repository, including history, we could work offline in this scenario. We can create local branches and only when we are ready to share the work with others, we can send it back to the central repository from where it can be distributed again. The process of sending information back to the repository is pushing. And once the new code is available, other developers can pull those changes back to their own machines. Of course, pushing and pulling does require to be connected. Most other operations can be done in a disconnected way. Next to being distributed, another very important concept in Git are the three states that files can be in. These three states are committed, modified, and staged. These three states are basically a promotion‑based system. When content matures, it can move up a level. In the committed state, the data is basically stored in the local database. The modified state means that the file has been changed, but it hasn't been committed to the database yet. Finally, staged means that the modified file is marked to be part of the next commit snapshot. Keep in mind though that all these changes are still local. Since files can be in three states, there will also be three areas in which files can live, and we'll soon add a fourth one as well. The first is the working directory, and that's where content will be created, edited, and deleted. Typically, at least for existing files, these files will be extracted from the local Git database. The staging directory can be seen as the area where changes from the local directory will be staged before they are committed. Basically, when files are in the staging area, they are waiting to be committed typically. They will be probably part of the next commit. Once we perform a commit, all changes will be stored in the local Git directory. This is entirely managed by Git itself. This is also created when we clone source from another computer or from GitHub. Although in a distributed source management system it is not required to have a remote, many environments will have a central place that is basically the main. When we have made the local commits to the local repository, we can send these off to the central location. And in our case, that will be GitHub. Indeed, GitHub will be the fourth area where a file can be stored. Although we are adding a remote, we still have all the benefits we had before of a distributed source management system. As mentioned already briefly, we can work with Git and GitHub in several ways. The first, and probably the most commonly used approach, is using the command line interface. Using the command line really gives developers access to all commands available in Git. In this course, although we are focusing on using GitHub as part of the workflow, we will need to use a local tool in several occasions. That will be the command line, as most developers prefer this approach, even though there's a learning curve involved. Next to the command line, we can also use a GUI. Although they're also powerful, typically they only support part of the available commands. Typical clients that you'll often see being used or GitHub Desktop, SourceTree, and Tower. Some of these are free to use; others require a commercial license. In this course, we'll do all the demos using the command line. Let's look at the commonly used commands now. All interactions with Git start with the dollar sign and then git. Just entering the git command really won't do anything since we'll need to pass more options to Git before it knows what you want to do. Using the git config command allows us to get and set configuration options, both globally, as well as for the current repository we're working with. Most of the time we'll be working with repositories. We'll explore in a lot of detail what repositories are. For now, see them as a folder for your project. When we want to create an empty repository, we can use the git init command. The net result of executing this command will be that the .git directory will be created in a specified directory with a number of subdirectories, all internal to Git. The git clone command will do exactly what the name says. It will clone a repository into a local directory on the machine and will also create a remote tracking branch and check out an initial branch. If the concept of branches isn't fully clear right now, don't fear as we will be exploring that in much more detail later in the course. Another important concept and commonly used command for Git is using git add. Using the add command, we can add a file or multiple files into the staging area. It basically makes Git include these files to the index, making them ready for a commit later on. Options are available on the add command to include only the selected fires as well so that not all files within the directory will get staged. Finally, another vital command is the commit command. Git commit will, as the name implies, commit the changes to the local repository. A new commit is created, and typically a commit message is sent along.

Demo: Getting Started with Git

Now that you already have an idea about the basic Git commands, let's go to another demo where we will use these commands to do some basic interactions with Git locally. First, we'll create a local repository, and then we'll add and edit some files, and we'll then create a local commit. In this demo, I will show you some basic Git commands and a bit of the flow to work with Git. We'll start by creating a new directory, and we'll do it also from the Bash command. To create a new directory, I'm going to use the mkdir command, and I'll pass in the path of the new directory I want to create that's going to be d:\code\pluralsight. Now we are, at this point, still not in that directory, so we're not working directly from that yet. We'll need to change that directory. For that, we'll use the cd command, so the change directory command, passing in the name of the directory. At this point, this is now our working directory, but this is just a plain directory, a plain folder on my disk. It's not a Git repository or Git directory, let's say, just yet. Let us change that. As you already saw, every interaction we'll do with Git starts with the command git, and this is no different. When I want to initialize Git inside of a directory, I'll use, well, the git init command. This rule will, let's say, convert this directory to become a Git directory. And as you see, indeed, Git now specifies that it has created an empty Git repository in our working folder. In fact, it has now created the .git folder, and that is the Git database directory, really used for its internal use to keep track of things that have changed, to bring down changes that have been made remotely, and so on. This is not a directory that you want to change manually. This is internal to Git. Now let's play a game. What has changed between these two lines? Indeed, the word main has been added here. In fact, when we initialized Git, it automatically also created a new branch for us, the main branch, that is now the default. We'll talk about branches in a lot of detail later in the course. But just for now, keep in mind that a new branch has been initialized in our Git repository as well, and that is the main branch. Let's see if there's anything already in that directory. It would be weird that we already have some code magically in there. Let's check it out. I'm going to call the ls command, and I'll pass in the la option. Well, there is, in fact, already something in there, and it is that .git directory, so the Git database. That is the place that Git uses to keep track of all the changes made by me or made by someone else in the remote repository. Now, if we want to ask Git what is the status really of my current working directory, we can use the git status command. That command will ask Git this repository, what is its status? Git will show me at this point that we are, indeed, on the main branch, and there haven't been any commits yet, which is kind of logical. We haven't done anything, so we also obviously didn't create a commit yet. And also, there are no new files that it can commit. We haven't really added anything to that working directory just yet. It also specifies that we can create files manually and then use the git add command to start tracking these files. Now a good idea would now be to create a new commit. Let's do that. And I'll use the code command again to invoke Visual Studio Code. I'll pass in the name of the file I want to create, and I'll use readme.md as the file name. We'll see soon that this is a file with a special meaning in GitHub. Md stands for markdown format, and we'll talk about that later on as well. Don't worry about that just yet. Now this will again, of course, open Visual Studio Code, but I can now edit my readme.md file. As always, we'll start by typing Hello world. Don't forget to save this file now. There we go. You see that the file has now been saved. We close Visual Studio Code. And, in fact, in Git, it seems that nothing has changed, but let's see if we do another git status if, in fact, nothing has changed. Oh, there we go. Something has changed. Git has now noticed that we have an untracked file, the readme.md file, that is waiting inside of the directory. Now the file is untracked. It's not being tracked. It's not being looked at yet by Git. But we can change that. We're going to add the file to Git. Adding the file to Git so that it does track the file can indeed be done by calling the git add command. And which file are we going to add? Well, I need to specify the file name, so that will be there readme.md file. So this will then bring the file to Git. It will basically start tracking that file. Initially, it's not doing that. Only when I do a git add command will it now become part of the files that are being tracked, that are being looked at inside, by Git. If we do now the git status, we indeed see that the status has changed again. Git is, in fact, now tracking that new file, that readme.md file, but it's not part of a commit just yet. Only when I call the git commit command will a new commit be created, and that new file will be part of that new commit. That commit is really, let's say, a bundle of all of us that we're going to store in the current state in the Git database. So let's do that now. Let's create a new commit. If you want to place your bets on what the command will be, this is your time. Indeed, it will, of course, be the git commit command. I'm going to specify a commit message, let's say a piece of text that indicates what changes I have done. Let's keep it simple here, and let's just pass in some comment as the commit message. Let's hit Enter again. There we go. And now a new commit has been created locally on my machine in which one file has changed. A new file was inserted. That was our readme.md file. Let's do another git status, and we are back where we started. We are again on the main branch. That hasn't changed. Nothing is ready to be committed at this point. The working tree is clean, meaning that no new files or no edited files are being tracked by Git at this point. Now, of course, we're not going to do all of this per file. Let me come back to you with a number of files precreated already. I've now created a couple of new files as you can see here. If we do another git status, we'll see that Git says, well, I see that you have new files created, but I'm not tracking them yet. Do we need to do that git add thing now for every new file individually? Well, in fact, we don't. We can use git add and then the dot saying that all files can just be added. There we go, and I've done quite a few things already. Why don't we take a look at the history. Well, Git also has a command for that, and that is git log. Git log says that, at this point, one commit was made on the main branch made by Gill Cleeren on this particular day on this particular time of the day. And there was a comment added, some comment. But didn't we create another commit? Well, we added those several files, but we just added them to Git. We haven't created a new commit for those new files. I'm going to change that. I'm going to create another commit by again typing git commit, but I'm now going to do it using a bit of a shortcut. I'm going to use the ‑am parameter, and I'm going to specify the commit message again. There we go. And now a new commit was created, and those three new files were added as part of that commit. If we do another git log now, we'll see that indeed two commits have been made on this main branch in our Git repository. Now this should already give you a basic understanding of Git. We will be focusing on GitHub mostly in this course. There are other great courses on Pluralsight that will help you getting a deeper understanding of the different Git commands.

Summary

With that, we have reached the end of this module. Let's wrap it up properly with a small summary of what we have seen already. Although this is a course on GitHub, there's basically no GitHub without Git. So the focus of this module was entirely on Git. After watching this module, you now have a good understanding of the concepts of Git and how it works. Most developers will use Git and GitHub through the command line, and we have looked at some commands already. In the next module of this course, we will start with the exploration of GitHub. See you there.

Getting Started with GitHub

Module Introduction

Hi there, and welcome to this module in the GitHub: Getting Started course here, on Pluralsight. I'm still Gill, and I will guide you through this module as well. This is a GitHub course, but so far, we haven't been doing an awful lot with GitHub just yet. Well, that is about to change with this module. In this module, I'm going to get you started with GitHub. Let's see what we will be covering in this module. We'll start the module with a part I've named Understanding GitHub. As the name implies, I'm going to give you an overview of GitHub, its features and what you can and cannot do with it. Then, we'll take a look at the start of it all, creating a GitHub account. In this module too, we'll look at the different ways you can have your local machine connecting with GitHub, including how we can use SSH to do so. In the final part of this module, we'll spend some time understanding how we can search the vast amount of data that lives on GitHub. Since there's really so much to be found here, GitHub offers some tools that we can use to find what we're after, and we'll learn about that here as well.

Understanding GitHub

Now we'll start at the very beginning. That's always a good plan. In the first part of this module, I'll make sure that you understand in full what GitHub is all about, and you'll understand why it is so popular today. As already discussed briefly, GitHub is a web‑based hosting service for Git. It's mostly used by software developers to put their code in. GitHub, therefore, basically extends what we can do with Git. GitHub offers the features offered by Git, such as distributed version control and source code management. Did I say it's popular? Official numbers from January 2020 say that over 40 million developers are using GitHub today, and there are more than 200 million code repositories. Let that sink in. That is a lot of code. Next to source management, it offers a lot of extra features that we'll discover in this course, making it a go‑to place not only for storing code, but also to manage your entire project. GitHub is praised because it's featured around the social aspect of coding. While you'll often use it to store your code in, it is really built around collaborating around projects. Everyone can get involved very easily and can participate in existing projects easily. We'll see this, for example, when we explore pull requests later in this course, which is one of the fundamental building blocks of the GitHub flow you'll learn about later as well. Since 2018, GitHub is owned by Microsoft. Now GitHub has been around for quite some time already. It was founded back in 2007/2008, a couple of years after Git itself had been created. Already in 2009, they passed the 100,000‑members barrier. And 1 year later in 2010, they already hosted 1 million repositories. That shows that a lot of developers were discovering GitHub very early. In 2015, Microsoft started to grow more closely to GitHub as they released the first edition of Visual Studio with support for GitHub baked in. At that time also, it was announced that the company was already worth $2 billion. In 2018, as said, the relation between Microsoft and GitHub was now set in stone as Microsoft acquired the company for around 7.5 billion. As said in a previous slide, in 2020, GitHub passed the 40‑million user mark, yet another milestone. So what do we get when using the GitHub then? Well, as mentioned, the cornerstone is code, code and, yes, more code. All that code lives in repositories. Repositories are a very important concept in GitHub. Another core concept of GitHub is pull requests. Using pull requests, part of the so‑called GitHub flow, developers can request that their changes are pulled into another branch. Typically, this is combined to the code review, which can be conducted directly from GitHub's interface. This is very relevant for the social aspect of GitHub as well where just anyone can propose changes to existing projects. As said, we'll explore every aspect of pull requests throughout this course in a lot of detail as it's also a very important concept that GitHub adds on top of Git. GitHub is much more than just source code. It's a fully fledged project management tool. We can work with issues to register work that needs to be done, such as a bug that needs to be fixed, for example. It also contains a project board to manage the work. GitHub also comes with CI/CD features, so continuous build and deploy features, in the form of GitHub Actions. There are many other features that we will be discovering together in the upcoming modules. As I already briefly mentioned in the previous module, GitHub offers several plans for us to choose from. Of course, these will change over time, so I haven't included any pricing information here. You can find that all on github.com/pricing. Most importantly, GitHub offers a free account type that offers you a lot out of the box, including an unlimited amount of both public and private repositories, a large number of minutes for GitHub Actions, issue management, and much more. We will be using this type of account in this course. The Pro version, also targeting the individual developer, brings more build minutes and unlimited amount of collaborators on the repository. Collaborators are users that you can invite to work on your project. Next to these, you also have options for teams named the Team and Enterprise accounts. Again, take a look at the link you see on the slide to learn more about the details.

Demo: A Tour of GitHub

Now that you already understand what GitHub is and its main features, I'm going to take you on a guided tour around GitHub. Of course, all features that we touch on here will be explained in much more detail in the remainder of the course. When you go to github.com, you'll be greeted with this magnificent landing page. But the first thing you'll need to do is signing in, I think. So, when we sign in and we pass the two‑factor authentication, we will arrive on this page here. This is the landing page that, once you are signed in, you'll typically arrive at. On this page, you'll see quite a number of things already. You'll see some of your repositories. You'll see the teams you are a part of, if any. You'll see the recent activity here in the middle, so people that started following you, changes you have made in repositories, and so on. And you'll also get a suggestion of some interesting repositories that GitHub at least thinks are interesting for you. When you click here on your profile, you'll go to your profile page. In my case, it is github.com/GillCleeren. This is the page you'll typically post on LinkedIn, on Twitter, and so on because this gives an overview of all the contributions you have made on GitHub. It is really the personal landing page. It contains a picture, a short biography, and you can also edit the profile here. You'll see some other achievements and highlights and so on. Here in the middle, we have some pinned repositories, so repositories that you have selected as being your most interesting ones. You'll see the contributions, and you'll also see a contribution history. Via the pinned repositories, or alternately via the repository step, we can go to all the repositories in my account. And that can actually become a long list as you can see here and then select a sample repository here. Code is, of course, the most important thing that we'll keep track off with GitHub. But as you can see by the Tabs here at the top, we can do quite a number of other things with this repo. Did I already mention that a repo is the building block of GitHub? Well, they should really make that clear. On the repo level, we can create issues. For example, if someone wants to log a bug, we can do that here. We can also create a pull request. We'll talk a lot about pull requests as they are very important in the way that you will work with GitHub. A pull request will basically be someone asking you to merge changes into another branch. We'll talk about actions to automate things, and we'll also see projects later on. Projects are a way to manage the work inside of a repository. Think of it as a project board. A wiki can also be created on the level of repository. Typically, a wiki will be used to store documentation. We can work with security settings, and we can also work with insights. Insights give us, well, insight in the activity of the repository, how many changes, how many commits have been made? How many people are working on the project? How many issues are there? And finally, we have these settings here, which give us access to a lot of settings on the repository level. Now, this is not a very active repo. This is just a sample repo. Let's take a look at a real live repository. And let's, for example, take a look at the ASP.NET Core repo. So this is the ASP.NET Core repository. As you can see, there's quite a number of issues, pull requests, projects, and so on available on this repo level. Quite normally, there's also a lot of code in here, and you can browse all the source code directly here in the GitHub interface. If you're a bit bored, why not take a look at the Explore page? The Explore page is based on your interests, and it will show repositories that GitHub things are interesting for you. There's always something interesting to discover here on the Explore page. Also, on the Explorer page, you can click on Topics, and then you'll find a list of popular topics on GitHub. You can, for example, browse all the repositories based on JavaScript, for example, and there's going to be quite a few interesting ones here to discover.

Demo: Setting up Your GitHub Account

With the guided tour finished, it's time to start working with GitHub yourself. Of course, the first part is setting up that account. In the next demo, we'll do exactly that. We'll also take a look at the options we have to configure a profile, and we'll learn how you can also follow other interesting uses on GitHub to see their activity. As already mentioned, to do just anything with GitHub, you'll need an account. So head over to github.com and click on Sign in if you don't have one just yet. Here you can click on Create an account, and then you can just enter regular information to create your account. Now you're all smart people. I assume you can create your own account. Once you have created the account, you'll get this personal landing page that we already looked at in the previous demo. From here, you can change some information directly, such as your picture or your profile. And on this page as well, you can also see your contributions, your pinned repositories, and your contribution activity. Any achievements you may have or if you are a pro account user will also be shown on this default landing page on your personal landing page, let's say. And most settings can actually be changed here by clicking on your profile picture here at the top right and then clicking on Settings. Here, you can change your public profile. You can change your name, your bio, your URL, and so on. These are my public profile settings. I can also change settings on the account. You can, in fact, change the username. So for me, that was Gill Cleeren. But if I change that, I will also break all existing links, so that is something to be careful with. You can export your data, and you can also delete your account should you want to do so. Under Account security, there's a number of important settings that have to do with security. You can change your password here, or you can enable or disable two‑factor authentication. If you have seen some suspicious activity on your account, take a look at the security log. There you can see when events have happened on your account. You may have logged in, you may have done two‑factor authentication, and so. All events that have to do with your account will be shown here in his log. Now this is my personal landing page. Of course, I can look at other people's personal landing pages. Let's, for example, take a look at James Montemagno's home page. He's one of the guys behind Xamarin and MAUI. We can, on this landing page, look at his repositories and his information, but you can also decide to follow him if you're interested in seeing updates he's making on his repositories. And if I want to do the opposite, I can also block users. Here, under Blocked users, I can, as the name implies, block users from doing just anything with my GitHub account. I can, for example, say that they cannot give me a star, they cannot watch me, or they cannot even post issues. This is something that you would do, for example, if you see someone making inappropriate comments on your repositories. There are quite a number of other settings that we won't be taking a look at, but you can explore them at your own pace.

Using SSH

Before we start connecting our machine with GitHub, it's important to ensure that things are happening in a secure way. Let's now look at using SSH, which we will be using starting from the next module. Connecting with GitHub is possible in two ways. We can use HTTPS. And while there is definitely nothing wrong with this approach and it's definitely secure, it might not always be the best approach. Depending on your operating system that you're using, it may be so that you are prompted several times for your username and password when you are, for example, pushing changes to GitHub. Over time, that will become annoying. In comes SSH, short for Secure Shell. SSH is a protocol that allows for secure communication. Using SSH, we will create an SSH key, which can be linked to your GitHub account. When you then connect from your local machine with GitHub, you won't be prompted every time to enter your credentials. If needed, you can also combine this with a single sign‑on approach if it's been set up for your GitHub access. And don't fear. Using SSH is just as secure as a HTTPS. So although you're taking a bit of a shortcut, you are not compromising on security.

Demo: Working with SSH

In the next demo, we're going to create SSH keys, and we're going to use these then to connect with GitHub. Throughout the rest of this course, this will be the way we will communicate with GitHub. To connect with GitHub, we can use HTTPS, but we are going to use SSH. So we're going to create an SSH key next, add it on GitHub, and then when we connect with GitHub, it'll notice we are passing in that key, and it'll know it can verify really that we are who we say we are. Consider it as a token to validate the communication. First, let's see if we already have such a token, such a key, installed. If you have just installed Git, you probably don't have one yet. But let's verify to be sure. Return to Git Bash and go to your root directory, so the Users directory. That would be for me C:\Users\gill. There is a command that we can use to check if we already have an SSH key installed, and that'll be ls ‑al and then tell ~/.ssh because that's the default directory where the SSH key, if any, will be installed. Get Bash indicates that we don't find that directory, so we don't have an SSH key installed yet. So, we are going to set it up again using Bash. Through the ssh‑keygen command, passing in these parameters, we can create a new SSH key. We'll be asked a few questions. We'll accept the default for the place to put the file. We then get the question to enter a passphrase. This will be like a password really, but just for the sake of simplicity, we'll leave it empty for now. And now, the SSH key has been created. Next, we'll need to add it to the SSH agent. We first need to ensure that the SSH agent is actually running. It should, in fact, already be running, but let's verify to be sure. There's another command that we can use, and you see it here. It is running, so that's good. So now we can add the SSH key. There's again a command for that, the ssh‑add command, and that specifies the folder in which our key was stored. Now that our key has been created locally, we now need to add it to GitHub. And then GitHub will know that once we make a commit and we start communicating with it, we are, in fact, who we say we are. Let's try that. Let us go first to this local directory. So I'm going to go to my Users directory, so that's C:\Users\gill. And in there, you'll find a .ssh directory. In here, you'll find two files typically, one that is mistakenly known by Windows as a Microsoft Publisher document. Don't double‑click it, but instead open it with Notepad and just copy the entire key to your clipboard. Next, return to GitHub. And there, under SSH keys, click on New SSH key. You'll be asked to give this SSH key a name. I'll give this the name Gill Laptop. The name is important to later on being able to track where this key was used from. Then I'll paste in the SSH key and click on Add SSH key. From now on, we can actually use this key. Let's now verify that everything works. For that, you need to use ssh ‑T git@github.com. You'll need to confirm this warning with yes. There we go. And now it says that we have successfully authenticated. So from now on, we can always use SSH. That will use our token, and we don't have to provide a username and password anymore. Our SSH token is used instead.

Searching GitHub

GitHub is vast, and it grows each minute of the day. Such a massive amount of code is also a treasure in terms of knowledge. There's so much to look at really. But how are you going to find in there what you're looking for? In the final section of this module, we're going to do some searching through GitHub using the tools it brings. To make searching the vast amount of data a bit more feasible, GitHub comes with some interesting search tools. Of course, we can launch a search that basically searches throughout everything on GitHub. That will effectively launch a search that's going to search through all the types of information, including source code, issues, wikis, and much more. Since the amount of results can be, well, a bit overwhelming and may not even be what you are after, you can also define a scope for your search. If you want to search for something within a repository, simply navigating to that repository limits the search scope to that single repository. Next to that, you can also scope your search so that GitHub will only search in code, wikis, issues, and so on. To give you a visual guidance in creating the correct search query, GitHub comes with an advanced search page. On this page, you will get a long list of options to finetune your search, including things such as the created date, indicated how many stars a repo has, in other words how popular it is, and much more. And filling in some of the fields, as you'll see in the upcoming demo, GitHub will create a search query for you with a specialized search syntax. You can use the same syntax as well when creating a search. Let's take a look at some samples that show you how this search syntax works. Imagine you are searching for dotnet‑related projects. Of course, there are quite a few of these, so you may want to search only for the popular ones. On GitHub, repositories can receive a star, so when you are being favorited by a user. So we can search for popular dotnet repositories by searching for dotnet and adding that we want the repo where the search result belongs to to have at least 1,000 stars. Alternatively, we can indicate that the repository should have between 10 and 50 stars. Another search query is this one. The word dotnet should be in there, but the words hello world should not. So basically we're saying don't give me those intro demos.

Demo: Searching GitHub

In the last demo of this module, we're going to experiment with searching on GitHub. To do this, we'll use both the advanced search page and the syntax, and we'll explore the search results retrieved by GitHub. From basically anywhere, you can search GitHub through the search box here at the top of the screen. GitHub is home to a vast amount of information, and we can search it to find tons of information. And let's, for example, use the search box to search for aspnetcore. We've already used that report before. Here you can see that I can search all GitHub or directly jump to the aspnetcore repository. And let's search for now all GitHub. The search is super fast and shows me a number of repositories where it can find aspnetcore. Of course, the top result is the aspnetcore repository itself, but many other places have been searched. So it searched in Repositories, but it also searched in Code, and it found no less than 7 million instances where aspnetcore is used. That's quite a few. You can also see the results in Commits, Issues, Discussions, Packages, and so on. And as with all searches, we can also sort based on the best match, recently indexed, or least recently indexed. Let's go back to the Repositories section here and click on the aspnetcore repository itself. It was our top search result, so let's dive into that one. Now I've used basically the search to search the entire GitHub pool of information. Now, in fact, I can also search just a single repository. Let me show you that. For example, let's click on src here, and this is aspnetcore, so there must be some Mvc in there and then MvcRazor and then src. And in here, I must find the RazorPage. There we go. If we now search for RazorPage, let's go back to the aspnetcore repository itself. So if we now search for RazorPage, you can search in this repository in the organization or all GitHub again. As for now, search in this repository, and I will find results filtered to the aspnetcore repository. So that's, of course, a lot less than when you would search the entire GitHub site. Now this search is multifunctional. I can, for example, also search for Gill Cleeren. I'm going to search for myself, which sounds a bit philosophical. But anyway, let's not get distracted. So let's search Gill Cleeren on all GitHub, and you'll find just one repository result, but notice you'll also find Users here. So indeed, my user account is found, and another demo account that I use for Pluralsight is also found. Now the search is pretty simple as you can see. I think it's very straightforward to find a lot of information. But if you're looking for something more particular, well, then you can also use the advanced search. On this page, GitHub proposes you a lot of extra options to perform your search. You can, for example, indicate that you want to only use popular repositories. Repositories in GitHub, as we'll see later, can be starred, basically meaning that a lot of people find them interesting. You can, for example, say that you only want a repository that has at least 10 stars. I'm going to type greater than 10 stars. I also want the repository to have between 10 and 1,000 forks. Forks are basically personal copies of a repository. Notice at the top, based on what I entered here as search options, that GitHub has created for me this search query. It uses a particular syntax that I can then use to make my search more fine‑grained. For example, let's also used another option and say that this repository needs to represent in the C# language, and notice it also updates the advanced search option. If we now make the search, we'll definitely find a lot less results than we did when we did the GitHub‑wide search.

Summary

And we have arrived at the end of this module that really focused on the use of GitHub. Let's summarize what we have seen in this module. We have done a high‑level overview of the GitHub features, and I hope that you already saw that GitHub is really a social coding platform. Repositories are the heart of GitHub. They will contain the code that we create, but also everything else, such as issues are linked to repositories. GitHub is so vast. It also comes with a powerful search engine that gives us relevant search results. So are you starting to like GitHub? I really hope you do. In the next module, we will dive much deeper in the use of repositories, and I hope to see you in that module as well.

Working with Repositories

Module Introduction

Hi there, and welcome to another module in the GitHub: Getting Started course here, on Pluralsight. My name is Gill Cleeren, and I'm happy that you are joining me in this course to learn about GitHub. In the previous module, we made sure that we had a correctly configured GitHub account, so we can now get started using the most important building block, repositories. That will be the focus of this module. Let's dive in. Before we start exploring repositories, let's take a look at what you will be learning throughout this module. We'll start with a very brief part on how GitHub works with repositories. In the next part, which is mostly demo‑based, we will see the different interactions we can do with repositories, both on GitHub, as well as when connecting from our local machine. You'll see how we can change files remotely, pull changes in, push to GitHub, and much more. Next, we'll look at some of the special files that we can include in a repository and which are known to GitHub. In the final part of this module, we will explore the many settings we can use on a GitHub repository, including how we can add contributors to a repository, see how many people have looked at a repository, and much more. By the end of this module, GitHub repositories won't have any secrets left for you. Let's do this.

Understanding Repositories

As mentioned, I want to kick off the module on GitHub repositories in a proper way, so we'll start with a short introduction on what they are really. Repositories are the base building block for GitHub. They can be seen as a folder for your project, and all files that are related to your project will reside in that repository. That includes also the history for your files in that repository. When you want to start using GitHub, you'll need to create a repository. It's possible to create a repository through the GitHub interface and clone that to your local machine using the command line interface. Alternatively, you can initiate your work locally and push that to a repository on GitHub. Repositories can be public or private. If a repo is public, it means that its code and basically everything surrounding the repo is publicly available from the GitHub website. Everyone can see the repo, but not everyone can make changes to it directly. Only collaborators can do so, and we'll see that later in this module. In a private repo, only users with access to the repo can see it and interact with it. Every type of account has the ability to create public or private repos on GitHub. You already saw this slide earlier in the course, but I want to repeat it here since you now already have a better understanding of what GitHub will do for you and your projects. When working with Git locally, we had three areas, so the working directory, the staging area, and the local Git repo. The latter is basically Git's internal database that keeps track of the changes. We can work with Git this way, but in most cases, there will be a remote as well. And when using GitHub, that is the remote repository. GitHub will be the place where several users on the team will go to sync changes from their local repositories. Using this remote is also possible to share code with others. Although GitHub works in a distributed way, the remote in this scenario can be compared with the central server in traditional source management systems. What you see here is the repository landing page on GitHub. It shows all the files that are included in the project, and it offers many actions that we can do on the repository. We can, from GitHub's interface, create or edit existing files, see the projects README, clone it to our local machine, or work with its settings.

Demo: Working with Repositories

In the first demo of this module, we are going to create a repository through GitHub's interface. Let's take a look. We've talked already quite a lot about repositories, but on GitHub, we haven't created one so far. Let's change that now. I am here on my profile page, and I'm in the Repositories tab where I'll create a new one. So I'll click on this New button here, and then I'll see this Create a new repository page. The first thing I do is I need to give my repository a name. If I don't have a particular project name in mind, I can also ask GitHub to create one for me, and it has done that already. Every time you refresh this page, a new one will be created. And I really like this one, the refactored‑telegram one. Let's use that. I can also pass in a description if I want to. Next, we have some options that we can specify for our newly created repository. The first question we get is if you want this repository to be public or private. Public means, well, as the name implies that everyone can see it, and other users can also suggest changes. We are still in charge then to choose who can commit changes to it. The other option, Private, basically means that your code is hidden, and only people with access to the repository, collaborators that is, can access the repository. And let's make this repository private for now. Next, I get some other initialization options. We can add a README file. We'll do that manually soon, but keep in mind for now that the README file is a very special file for GitHub that is shown by default on the repository landing page. Next, we also get the ability to let GitHub create a gitignore file. A gitignore file is very useful as it will be used to exclude files that don't make sense to be uploaded as part of your code in your repository. You can check this, and then you get to choose which gitignore file template you want to use. I'm using Visual studio, so I'll use the Visual Studio Code gitignore file. Depending on the project, this definitely might be a different one. I can also select a license here that will be added by default to my repository. I suggest you always do that as it will give people a clear understanding of what they can and cannot do with your code. And this is a private repo, but still it makes sense to add that license here. Again, GitHub comes with a number of suggestions. As you can see here, there's quite a number of possible licenses that we can use. Let us take the GNU General Public license v3. There we go. The repository has now been configured correctly. Let us ask GitHub to create it. Here's our repository. It's called the refactored‑telegram repository. We are now the proud owners of our first GitHub repository, and it's private. Let me show you that. If I take this link and I go to a private window, it cannot be found. You cannot access this repo because it's set as private. If you want, we can also change it back to public later on. Let's now take a look at our repo in a bit more detail. Here's the name of the repository. It is under my account, and it is called refactored‑telegram. For our repository, we can add some additional information as well. It's always good to give people a little bit extra information to explain what this repo is all about. If you click here on the little settings icon here, you see this dialog. In here, we can add a description, a short description of the repository. Don't start writing a whole book here. We have other places to do that. You can also point people to a website. We'll talk about the websites attached to the repository later, and we can also add topics. Again, we'll talk about this later. I'll go Save changes, and the repository details have been saved, and we see our repository information there at the top right. Now we are here under the Code tab, and that shows, well, not surprisingly our code. We don't have a lot of code yet. We'll do that in just a minute. All the information in this block here has to do with the code. We currently have, for example, one branch. That's the main branch, and we'll talk about branches very soon. Keep in mind already that through this pop‑up, we can also create a new branch. We currently have indeed just one branch. If we click here, we see that we have one branch that's the main branch, and it's even the default branch. The fact that the main branch is the default branch can actually also be considered quite important since that is the branch that we'll be creating pull requests to by default. Through the filters here at the top, you can also view a different selection of branches in your repository. It's possible through the GitHub interface to also start editing code. We'll do that very soon, but already see here that I can create a new file or upload files here through this Add file button. And we can also, of course, download our code from here. Through the Code button, you get access to the SSH or HTTPS links to download your code. We'll use this very soon to download a copy of our repository to our local machine. It is also possible to open this repository directly with GitHub Desktop. GitHub Desktop is a companion desktop application available for Windows and other operating systems that allows you to work through a GUI with GitHub. I won't be doing that in this course. There's another course in the path that goes deep into the GitHub Desktop. You can also open it directly with Visual Studio if installed or, very simple, download the zip file that contains all the code in this particular branch. We also see that our repository has already had one commit. You saw commits the previous module. What have we actually already committed? Well, let's click on Initial commit to see what was created as part of this commit. An initial commit was made when the repository was created. Remember, we actually added already two files. That was the gitignore file and the license file. There are the two files that have been created as part of the initial commit. We can add a comment here on this commit by typing it here directly in the comment box. The commit also has an ID that you see here, and I will go back to this overview. You'll also see part of that ID over here. Again, in the Code tab, we, of course, can browse the code. Well, we don't have a lot of code. We'll change it very soon. But if you just click, for example, on the license file, you'll see the license file being shown. And as this is this is a special file, GitHub will show me a bit of a different interface to visualize this license file. So this part here at the top is, in fact, special to the license file. GitHub is also still warning me that I should be adding a README file. We'll do that very soon. The contents of the README file will actually be shown here on the repository landing page as well. And then here back at the top, we see a lot of other tabs that have to do with the repository. So all of this information is managed on the level of the repository. When we will start adding issues, so bugs or tasks, we'll do them under Issues. When we get pull requests, we'll see them appear here. When we get wiki pages, we can access them from here. And also settings and insights are all managed on the level of the repository. We'll dive into all of this in a lot of details in this and in the coming modules.

Connecting with GitHub Locally

A lot of actions can be done through the GitHub interface directly. However, since GitHub is the remote when working with Git, we will have to perform a lot of interactions with GitHub locally. Let's take a look at how we can do this using what we have learned in the previous module. Let's look at how we can do this. When working with a remote using GitHub, developers will typically use the following flow. Assume to start that you want to start working with the repository which already exists on GitHub. So basically, the remote or quote quote central repository is already there. To be fully clear, in the full workflow, GitHub will be the remote repository or, for short, the remote. It's, in fact, nothing more than the central place where the code is stored for everyone to access it. Often, we talk also about the upstream repository, which is another name for the remote repository. The first thing that we'll do is a git clone. This will create a local copy on your machine of the remote repository for you to work with. A clone command will initially also create remote tracking branches for all branches in the cloned repo, and it will check out the default active branch of the remote. Now, we haven't talked about branches just yet, so no worries if you're not completely following the branches aspect here. That will come later. For now, just remember that the clone will perform a full copy of the remote repo into the local repo, and we can then start working in our own working directory. You'll see in the demo that GitHub also has in its interface an easy way to start the cloning process. We can now edit files and add new files. The latter we can add to Git so they will be tracked using the git add command. Once we are pleased with the work, we can perform a commit so that are changes are stored in the local repository. Just to be clear, so far, we have done just one interaction with GitHub. We have performed the initial clone, which gave us the local copy. All other work so far was local. But now we've made changes to the code, and we want to share these with others. So, we'll now need to push these changes back up to GitHub, so the remote. And for this we'll use, well, the push command. Now of course, when working with multiple people on a project, chances are that others have made changes while you were working on that feature as well. In that case, your push command may actually result in a conflict, and it's your responsibility to fix these issues. Therefore, it's recommended in most cases that before you perform the push, you'll basically check if other changes have been made. If so, you'll want to bring in those changes to your local machine and fix things first. We basically have two options here. We can perform a git fetch. That will bring down the changes made in the upstream repository, which you do not have locally just yet. However, it will stop there. No merge will be done just yet. Only when you perform a git merge will your changes be merged. In this case, some things might be performed automatically, while others may require a manual intervention from you. Alternatively, you can also perform a pull. A pull can be seen as a bit of a shortcut where the fetch, as well as the merge with changes in your local repository, will be done in one go. We can basically say that the Git pull is the sum of a git fetch and a git merge. After a successful merge, we can then still perform the push back to GitHub, so others can now see the changes you have made locally.

Demo: Cloning from and Pushing to GitHub

Now that we have talked about the flow of how we can work with GitHub as a remote, it's time to look at the demo where we'll see this in action. We'll start from an already existing repository, which we will clone to the local machine. We'll then make some changes to it, and then we want to publish those changes back to GitHub. We'll be using the commands we talked about in the previous slide. Although we can create files directly in the GitHub interface, most of the time, I will be coding directly locally on my machine. So, what I'm going to do is I'm going to clone my project to my local machine and work on it from there. Remember that we had the Code button here that, when we clicked on it, showed this link here. That's a link to clone my repository directly to my local machine. Click on this Copy button to copy it to the clipboard. Now I did create locally on my machine that D:\Pluralsight\Projects directory. That will be the directory I want to work in for our repository that we're going to be cloning. Don't use the one that you already created earlier since that will already contain a Git repository, and you cannot have two repositories in the same folder. That's not possible. So, what I'm going to do is, in a new Bash window, I'm going to switch to that directory. So I'm going to do cd to that particular directory. We are now in that directory. Now in this directory, I want to clone my remote repository from GitHub. So I'm going to do that again using the git command, and I'm going to say that I want to do git clone here and then paste in the link I copied to the clipboard on GitHub. Now since we are using the SSH key, we should automatically be able to do this without being asked for username or password. We see here that the git clone command brought in a number of files, and they are now locally on my machine. Let's verify that. Let's do an ls ‑la again. And indeed, you see that the refactored‑telegram folder has now been created because it cloned that repository from GitHub. We're now still in the project directory. I actually want to go into my refactored‑telegram directory, so we'll type cd refactored‑telegram. I still like that name. It's a very good name. There we go. You'll now see that Git has noticed that this is, in fact, a Git repository. We didn't have to say to Git that this is a git repository. No. It automatically noticed that, and we can see that because it puts the main here as the name of the branch we're currently working on. If we now do an ls ‑la again, we'll see a little bit more that has been added to that directory already. We again see the .git directory, which was the Git database. We shouldn't be changing that. That's internal to Git. But also notice that the files that we already created on GitHub, so the gitignore file and the license file, have also been added to this directory. That is because of the clone that simply went successful. So our clone went successfully. Let's ask Git what is the status. Our things have changed a little bit compared to what we saw, well, earlier in the course. Now it again says that we are in the main branch, and we are up to date, but now we are up to date with origin/main. Now this it didn't say before. Now it actually has a remote, and that's known as origin. And our local main branch now has a remote tracking branch, main. These two branches are now, let's say, copies of one another. If I now make changes on this main branch, we can then push them back up to the main branch remotely on GitHub. Now we don't have a lot of code yet. Let us change that. I'm going to copy in a number of files. I'm going to do that simply with my File Explorer. When you download the course assets for this course, from the Pluralsight website that is, you'll find in there a very basic HTML‑based website. Let me quickly show it to you. It's very simple. It shows just the landing page with some Lorem ipsum text and a nice background image. In the folder, you'll find this index.html, some images, and also some CSS, some custom fonts, and some JavaScript. These are the files that, for the most of this course, we'll be working with. So this is what you see here on the right. On the left, I'm going to go into my refactored‑telegram folder. That is my local folder in which I've initialized Git. Let me copy the assets in there. Let's now go back to our Bash window. And if we now do a git status again, now Git will say, hey, I noticed that you've added a number of directories and a number of files again that I'm not tracking. We already know what happened here. Git noticed that the files have been added, but it's not actively tracking them. So we need to change that. Again, I'm not going to be adding all the files individually to Git. I'm going to use the git add command again, and I simply specify the dot. That will take all files and directories and add them to Git. It will trigger Git to start tracking them. We get some warnings about line feeds. Don't worry about those. Let's do another git status. And it now noticed that quite a few files have been added. It looked at those directories recursively. It noticed .css files, some fonts, some jQuery, and, of course, also the index.html. So now we've added those files so that Git is tracking these files. Let's now created a new commit. I'm going to do to git commit, and I'm going to specify a message again. Let's specify website files added as the commit message. Now we're going to create a commit that contains all these new files that have been added. There we go. Our files are now part of a commit. Does GitHub know about this commit yet? No. At this point, this is just a local commit. Nothing more. No one that is working with our repository in GitHub will know about this commit just yet because it's just a local commit that we've made. That means, at this point, we cannot work with someone else on this repository because they don't see our changes yet. Let us now change that. I'm now going to bring in another git command. That will be the push command. And we'll use that to push up our changes, to push up our local commit, to GitHub. So let's use the git push command. I'm going to be specifying that I want to push this to origin. That was my remote, and it will be GitHub in my case because I cloned it from GitHub, and it's automatically set as the remote. And then I specify also that I want this to go to the main branch, which is our default tracking branch for our local main branch. Let's hit Enter, and after a few seconds, our changes have been committed. Notice what it says here. My commit has been pushed to the GillCleeren/refactored‑telegram repository, and the local main is now in sync with the remote main again. Let's verify on GitHub that we see all our files. When I refresh my page here on GitHub, I'll notice that a new commit was added just 2 minutes ago. I will also see the files I've added. The index.html has successfully been added. There we go. Also, the other files in the assets folder and the images folder have been successfully pushed to our GitHub remote repository.

Demo: Working with Multiple People on a Repository

Of course, working with a repository is rarely a one‑man show. Typically, we are working with several people on the same code. That means that code will have changed on GitHub, and we need to get these changes in. In this demo, we are going to see how we can do that using fetch and pull. Now we'll need to work with multiple people on a repository, and thus changes might have happened in the repository that we don't have local yet. Now I'm going to simulate that here. We're going to make a file change directly on GitHub and then bring that in local. Now, we are going to use for that another feature of GitHub where we can directly add and edit files in the GitHub interface. Let's start with that. So, I'm going to go to my index.html file. And, as you see here, there's a little pencil icon here. We're going to edit this file. I'm going to make our site, which seems a bit boring to me, a little bit more enthusiastic. So instead of just saying Welcome to our site, I'm going to add a few exclamation marks to make it a bit more enthusiastic to welcome people. There we go. Now I've made this change directly in the index.html in the main branch of our repository. Now everything that we do, even though it's such a small change, it will be part of a commit. So a new commit will be created. We cannot go around that. We cannot just edit the file out of the blue. That's not possible. So I'm going to create a new commit. I need to also add in a message. The suggestion was actually quite good. I can also add an extended description. We're now going to commit it directly to the main branch. We could, in fact, also create a new branch and do this with a pull request. We'll do that later. For now, let's keep it simple and just commit this to the main branch. Click on Commit changes, and there we go. If we go back to the repository's home page, we'll see that the commit, the third commit, was made on this repository. That's all good. And a change was made on index.html. Indeed, there is a change that we can see here. The three exclamation marks were added on the right, so that is the new version. I'm also going to add a new file, and I'm going to do that again through the GitHub interface directly. I'm going to click here on Add file, create a new file. And here, I need to add the name of the file, and I'm going to call this instructions.txt, and I'm going to put some sample content in there. To add this new file, again, we need to create a new commit. I'm going to leave the defaults as is, so I'm going to click on the Commit new file button. And a new commit, the fourth one, was now created, and we can see that here. Now at this point, our repository on GitHub has evolved. Now our local repository does not know about this. The local repository is totally unaware of these changes that have been made directly on GitHub. So imagine this being someone else having done some changes on GitHub. And us, in a local repository, we don't know about that just yet. So we're going to change that. We're going to let our client know that these changes have been made. So let's go back to Git Bash. Now we are still in our local repository, and now I have a couple of options, let's say. I can actually use two commands here, very popular commands that is, and it's the fetch and pull. The fetch will basically only look at the changes that have been made, but not merge them into our working copy just yet. A pull, however, will do that in one go. It will automatically try to merge the changes made in our remote repository into our local repository. And a merge can go either way. It can go automatically if no conflicting changes have been made remotely compared to what we have locally. That's called a fast‑forward merge. But sometimes, there can be conflicts. And then, of course, we'll also need to solve these. So let's first check the status again of our repository, the local repository that is, and it says it's up to date. Again, it doesn't know about these changes that have been made remotely. It didn't actually go out to look at the status. That it didn't do. I'm going to have to ask it to do that. I'm going to use first the fetch command because I want to, let's say, peek into the changes, if any, that have been made on our remote. This will pull the changes already in the Git database, but not in our working directory. So our local directory hasn't changed. If we do another git status, now we see that the message has changed. Now it says that our local branch is behind by two commits, and it also already did some work for us. It noticed that we don't have any conflicting changes. It can actually be fast‑forwarded by basically doing a git pull to bring in the changes that are made remotely into our local repository. So it's actually a very simple one that we have right now. The remote is just a little bit ahead of what we have local. So, let us do that. Let's do a git pull. It'll now do it fast‑forward, and we see that changes have been made to index.html. And a new file, instructions, has also been added. Now our local repository is again up to date with the remote. Now we can verify that. If I now call code and I say index.html, that is going to open the title here with the exclamation marks as you can see here. So indeed, we now have the latest version locally available. We're good. We're back in sync. Now this is, of course, a very simple scenario. Let me show you some other situations. Assume that we now have our exclamation marks here, so the more enthusiastic title. But frankly, I still think we could do better. Let us add here some more enthusiasm by making it our awesome site. So we've made a local change. Again, let's save that. Close Visual Studio Code. If we do a git status, we'll see that we have modified index.html. We have made a change. It's not being tracked, so let's add it by using git add and dot again. And I'm also going to make a local commit because I think I've done a great job. So let's do a git commit. Let's use ‑am, so the shortcut version again, and let's say local change of title. Now I've made this change locally, but instead of already pushing it up to GitHub, I'm also going to make another change there. Let's return to GitHub. On GitHub, I'm going to go back to the instructions.txt file and also make a change there. Again, using the pencil icon here, I'm going to change the title and add here to the server. We just want to make an edit and a commit on GitHub. So let's change this to New instructions added as our commit message. Again, let's push to the main branch, and let's commit these changes. So now imagine what we have. We now have changes. We have commit on GitHub, and we also have a commit locally. Let's now see what happens when we want to push the changes that we have made locally up to GitHub. So, I've made changes, but someone else has also made changes. Let's see what the result will be. Let's return again to Git Bash where we have our local commit pending. If I do a git status, we see indeed that we have a clean working tree again. Nothing is ready to be committed, but we have one commit locally that's ahead of origin/main. Let's do a git push like we did before. Do a git push origin main. We're going to try to push our changes to GitHub. And now we get an error. GitHub is rejecting our changes. Now why is that? Well, it already gives us a message on what is happening. The remote, so the origin, already contains changes that we do not have locally. So we first need to get those changes in, merge them if necessary, and then push the result back to GitHub. So let's do that. So I'm going to do another git pull. I could do a fetch, but I'm going to do a pull because I do need to merge in any case. So I'm going to do to get pull. Automatically now I see that Visual Studio Code opens. A merge has been made automatically for me, and a merge message is created also. If I want, I can add some more text here. I'm not going to do that. I'm going to close this. There we go. And our new commit, the merge commit, has been created. So now, basically, if we do a git status, we see that now there are two commits waiting locally to be pushed back to origin/main. And we can now simply do what we did before, git push origin main, so that GitHub will also know about our original commit and the merged commit. And now GitHub will accept our push, and there we go. And if we now return to GitHub, well nice to see the local change of the title that was adding awesome. On GitHub, we added the new instructions, and then there's another commit that was the result of merging everything together that has now been pushed to GitHub.

Adding Special Files

GitHub has a number of known or special files it works with in our repository. These files are used to allow you, as the repository owner, to communicate expectations for your project and also manage how users will contribute to the project. In this part, let's take a look at some of these already. The best known file that we can or, should I say, must add to the repository is the README file. In the README file, you typically put information about the project, such as what it does, how users can help with the project, who owns it, who contributes to it, and so on. It's a file known to GitHub when you put it in the correct location. This includes the root of the repository, the .github folder, or the docs folder. What I mean by known file is that when it's available, GitHub will perform a certain action with it. And for the README file, that means that this file will be automatically shown on the landing page of the repository. And when you make changes to it, those will be reflected right there on the landing page. The README file is typically written in Markdown format, also known as MD format. On the screenshot here, you can see the result of adding a README file. GitHub noticed the file, and it renders it automatically on the landing page of the repository. Not only the README file is a special known file to GitHub; others exist as well. The LICENSE file, named LICENSE or LICENSE.md, contains your open source license, allowing aspiring contributors to know what is the applied license for the project. Next, we have the CONTRIBUTING or CONTRIBUTORS file, which is used to contain a list of people or organizations who have contributed to the project. The CHANGELOG file contains a list of all major changes between different versions of the project. Another file is the SUPPORT or SUPPORT.md, which is a file to inform people about possible ways to get help with the project. Another very important file is the CODE\_OF\_CONDUCT file. This file contains guidelines for people on the rules they have to follow when interacting with the project, signaling a welcome and inclusive project and outlining procedures for handling abuse. Finally, the CODEOWNERS file is used to specify which users are responsible for the code in the repository. Note that not all of these files are required for each project. When browsing through projects on GitHub, you'll notice that these README files typically have a very common formatting style. That's because they're written in Markdown, a very lightweight markup language using just plain text. It can be converted to HTML pretty easily. Here you see some examples of Markdown. By adding a hash or pound sign, we are indicating that this line should be an H1 header. By adding a second hash, that line will be rendered as an H2. Adding some asterisk in front of items renders these as a list. Markdown is pretty simple, but showing you the entire syntax is not the goal of this course. You can find to complete syntax list at the following URL.

Demo: Adding a Readme and Codeowners File

All right, let's now return to GitHub and add a couple of these special files. We'll start with README, and then we'll bring in a CODEOWNERS file. Now GitHub has been begging us to add this README file. This message here has been staring at us from the very beginning of the creation of our repository. So, let us do that now. The README file is really a special file, which is going to be shown automatically where this placeholder is now showing. And it will typically be used to show, well, the name of the repository and also some general information, instructions, or even code samples. So, let's add one by clicking on this Add a README button. And the file needs to be called README because otherwise it won't be picked up automatically by GitHub. MD stands for Markdown, and that's a special type of syntax that we can use to easily mark up our code. So I'm going to remove the default contents here. I'm going to bring in some Markdown. We'll talk about Markdown later in the course. There's a number of special characters. The hash here makes this line an H1. Two hashes make this line an H2, and three hashes make this an H3. Adding a dash makes this a list item. Those things are typical for Markdown. They're a very simple way to create, well, your basic HTML markup. Let's create a new commit to create that README file. Click on Commit new file, and there we go. We automatically see our README file appearing here on the landing page of our repo. It looks a lot better already if you ask me. Now the README file is probably the most well‑known special file on GitHub, but there are quite a few others. Let us add another special file. That'll be the CODEOWNERS file. Again, the name of the file makes this a special file. So I need to name this CODEOWNERS. Adding CODEOWNERS file can only be done by a certain number of people within the repository. Typically, it's the admin or people with owner permissions that can create that CODEOWNERS file. And what is its purpose then? A review will be automatically requested when someone opens a pull request that changes code that the code owner owns. The CODEOWNERS file is also typically placed in the root. It can also be placed in the GitHub directory or in the docs directory. Now to make the CODEOWNERS work, we need to use a bit of a special syntax. We need to put in this type of syntax here. I use the asterisk and then the name of the GitHub user account that I want to do the review. Again, we can do a commit to add the CODEOWNERS file, and I will see that the CODEOWNERS file is in the root, and it will automatically now make sure that a review is requested when my code is changed.

Administering Your Repository

Now that you have seen how we can work with a repository hosted on GitHub, let's explore some of the many other features which are offered on repositories by GitHub. I have already mentioned a couple of times that the repository is the foundational building block on GitHub. Everything basically evolves around repositories. We've now seen how we can work with them, but GitHub offers many extra features that allow us to work with them even better. We can assign topics to a repository. Now you can think of a topic as a tag, which is used to classify your project that can be assigned by you onto your repository. Once you have applied one or more topics, those can help your project to be more easily discovered when someone is performing a search. Also, when clicking on a topic, other repositories with similar topics will be shown. From the settings of a repository, we can also work with projects. As the name implies, we typically use a project or project board to track progress. We'll explore this later in the course. On a repository, we can also see and manage issues. Issues are a way to track all things, such as bugs, to dos, feature requests, and more. We'll spend a whole module later exploring issues in much more detail. Another important concept that is linked to the repository is the concept of a pull request. We've already talked briefly about these, and we will explore them in much more detail in the next module. Using pull requests, GitHub makes it easy to collaborate on code, allowing people to suggest changes and allowing repository owners to review these proposed changes. Also, in repositories, GitHub offers the ability to use so‑called insights. As the name gives away, this gives you a lot of information about your project, including the traffic to your repository page. Finally, using the settings of a repository, we can add several other settings about the repository, including the collaborators, the default branch, and more.

Demo: Working with the Repository Settings

Instead of talking about these repository features and its settings, let's go straight back to GitHub and take a look at these in the next demo. We'll also manage the topics for our repository so that it is now categorized. Now on our repo, we can also add topics. Topics are basically going to be a way to categorize our repository, and then people browsing GitHub might come across our repository, or we can also direct them from our repository to repositories that cover the same topics. When you click here on the settings wheel, then you get the ability to add topics. Now our repo contains some HTML, so that might be a good topic. You will see that automatically GitHub will suggest me some topics based on what I've already entered. Our repo also contains some CSS, and there's also some jQuery in there, and let's use that as well. I can now save the changes. And now, as you see here in the little tag cloud, the topics will start appearing. And now, as said, from here, I can click on, for example, jQuery, and I will start browsing GitHub based on that topic that I've added. And this is really a way to explore similar repositories starting with our repository. It's always a good idea to add these topics to your repository. And there's much, much more that we can do with a repository. I have already touched on that a few times, but let's now already start diving in to a few of these options in a bit more detail. For example, on a repository, I can start creating an issue. Issues is really used to track all kinds of items around the repository where there are bugs, to dos, feature requests, and so on. Let's already created a new issue, and let's call it Sample new issue. I can also add a comment here, and this is again Markdown that we can use here. We can preview the contents of our issue, and it's also using Markdown. So if you want to learn about Markdown, you can really add to the editor here, and you'll also see how it looks in Markdown. Let's submit our new issue, and there we go. Now we see the issue here appearing in the issue tracker of our GitHub repo. Pull requests, they are really important. They are a way to announce your changes and ask for review and possibly also that your changes are going to be merged into another branch. We are going to ask that our changes are going to be pulled in to another branch, hence the name pull request. I'm not going to create one now, but we'll do that very soon. Let's take a look at projects. As mentioned before, projects are a way to create a project board. We can, from here, create a project. Let's call it Sample board. And I can also select a template. I'm going to use the basic kanban template here, and that's going to create already for me a sample board for the project. It has different lanes. By default, it chose To do lane, an In progress lane, and a Done lane. And if you think of what I've done before, I created an issue. Well, issues are basically your work items. I can drag an issue onto my board. Now it's in the to‑do list. When someone is working on it, they can put it into the progress lane. And when it's done, they can actually move it to the Done lane. So that's how issues and projects can relate. Again, we'll dive into this in a lot more detail later in the course.

Bringing in More Users

You hopefully already have understood that GitHub is a social coding platform. On a repository, we can work with many to achieve a goal. In GitHub, we can distinguish two groups when working on a project. First, collaborators. Collaborators are a fixed group of people, so the core development team working on a project. They typically have more permissions on the repository. The most important one here is commit access in the main repository. In other words, you, as the project owner, will trust one or more people to be able to change the code in the main repository of the project. This is indeed a curated group of people that can be managed through the repository settings. Next, we have contributors. This can be basically everyone from outside the core team, so the team of collaborators. A contributor is someone who perhaps uses your project in his or her daily work and is proposing a change or improvement. The big difference is, of course, that contributors, so basically everyone, has lower permissions. They cannot commit to your main branch. The proposed changes will come in as a pull request, which can then be reviewed.

Demo: Bringing in More Users

In the next demo, let's take a look at how we can add a collaborator to the core team of the project. And since we are the owner of the repository, we can control who has access to it. Our repository is set to Private, so that means that people cannot just browse to it. Let's verify that. If I copy the link here and I put it in another tab in a private window, I will not be able to access that repository even though I'm logged in as different GitHub user in this browser. That's because this repo is set to Private. But imagine that I want other people to have access to it so that they can work on it. Then, I can make them collaborators, and that I can do by going to the Settings. And in there, I can click on Manage access. I'll need to enter my password again. There we go. At this point, it says here that indeed the repository is private. Only people that have access can view it. I want to manage that, and I can do that by inviting a collaborator. I'm going to bring in my other account. That's gillcleerendemo. That's the one I had logged in in the other browser. When I click on Add gillcleeren to this repository, this user will now get an invite. I will get a notification, typically via email. But I'll copy the invite here, and I'll go to that invitation. So now I can go to the link, so the refactored‑telegram invitation's link, and I see that I am invited. And now, my second user has access to our repository. If we refresh here, we indeed see that gillcleerendemo, so the second account, is now a collaborator as well. And we see here also that repository has one collaborator.

Demo: Repository Insights

In the final section of this module, I want to highlight another interesting feature that is part of the repository settings, and it is insights. As mentioned before, insights gives you insight information in your project. Using insights, you can get information about contributors to your project, basically showing you who was the most important contributor. This can perhaps help you in deciding if they can become part of the core project of a team, so becoming a collaborator. Commits, quite logically, gives us an overview of the commits made to the repo. Using traffic, we can see information as to the number of clones or visits to the repository. Other graphs, such as the code frequency, give us other insight into the activity of a project. In the final demo of this module, we will take a look at the insights information of a repository. Using insights, we get, well, as the name gives away, insights in our repository. How many people are using it? What's the traffic? Let's take a look. So on the home page of the repository, click on Insights. And there we'll get the pulse. The pulse is really the dashboard, the overview, of our repository. As I had expected, there's not that much going on for our repository just yet, but we'll take a look anyway. There are, at this point, no active pull requests, but there is one active issue. That's the one we created earlier. When we click on Contributors, we can see who is contributing to the project. Well, it's a one‑man show at this point. It's only me doing all the work. We can also look at the traffic, the commits, code frequency, and so on. But since this is really repository where not a lot is happening, it doesn't make a lot of sense to look at it here. It might be more useful to look at it in a very active repository. Let's jump again to aspnetcore, and some of these settings will actually be available here as well, even though I'm not part of the ASP.NET co team. Here again on the pulse, we see how many pull requests there are, how many issues there are, how many contributors there are. There are quite a few as you can see here. You see also the most active ones. We also see the commits. There's quite a few commits on this repository, which was to be expected. So as you see, the insights give you a lot of insight information into the inner workings of your repository.

Summary

We have reached the end of yet another module, which was entirely dedicated to repositories. I think to summarize, it suffices to say that everything in GitHub evolves around repositories. In our workflow, where we use Git locally, Git will be the remote. On top of what a repository offers by default, GitHub offers many additional services on top of repositories, and we have technically just scratched the surface of that in this module. In the next module, we'll continue to work with repositories, and we'll be introducing the GitHub flow.

Collaborating Using the GitHub Flow

Module Introduction

Hi there. It's great that you are joining me for another module in the GitHub: Getting Started course here, in Pluralsight. My name is Gill Cleeren, and I'll be your instructor for this module as well. In the last module, we had a lot of fun exploring the many facets of repositories, one of the most important aspects of GitHub. While we have covered a lot of ground, there are quite a few things that we haven't covered yet so far around repositories. So in this module, we'll extend the knowledge of the previous module, and we will be learning about how we can work with branches in our repositories and use the very important concept of pull requests in GitHub. Let's take a look at the agenda for this module to start with. We'll start this module by taking a look at how Git works with branches. If you're coming from a traditional source code management system, you may be thinking of branches in a different way than Git and therefore also GitHub thus. So it's definitely worth taking a look at how this is handled in Git. There are different ways of how you can work with branches. I'll explain the most commonly used ones in the next part where we look at branching strategies. Next, I will explain you what is known as the GitHub Flow, which is a typical way of working with GitHub. While we learn about this, we'll learn about pull requests and how they work with GitHub. I'll show you also what happens when you perform a merge, which can be an automatic merge if there weren't any conflicts or, alternatively, what should be done when there are merge conflicts. Finally, still in this area, we'll learn about forks and what is the difference between branches and forks. Excited? All right. This module contains some very foundational concepts that you'll use all the time when working with GitHub, so it's important that you grasp these in full. Let's do this.

Working with Branches

As mentioned, we need to start our exploration of branches with an overview of what branches are in Git and therefore also in GitHub, so let's do that first. The concept of branching is certainly nothing new, and it's also not something that is unique to Git. Other traditional version control systems also offer the ability to create a branch. If you aren't familiar with the concept, well, it's really nothing complex. Basically, instead of writing all the code on one line and continue to build on that, we, at some point, diverge from the main development line. See it as a train track that goes on and on and on, and at some point, a side track rises. On that side track, we can do all kinds of things with the code without impacting the main line of code, so the main track. If we like what we have done in the side track though, so we can, at some point, just like train tracks again, converge them. The changes we have made in the side track are then merged with what we have in the main development track. Branching is available in other source management systems as well. However, in traditional ones, it's often an expensive operation. It typically requires that we create a copy of the source code directory and work on that one. If the directory is launched, this will take time. The concept of branching in Git is different though. It's done in a completely different way, making it much more lightweight. Hence, the creation of a branch is really fast. In Git, and thus also in GitHub, it's really recommended that you work with several branches in your application development. Creating branches, switching between branches, and merging branches is all really fast. And therefore, it's the recommended way of working. Initially, Git was created to support large and widespread teams working on things such as the Linux kernel. If everyone would be working on the same branch, that would be really hard. Even with branch support in traditional source code systems, that scenario would be hard since a lot of copying would occur to work with branches. As said, in Git, it's much faster, and it works, in fact, because of the way that Git works internally. Git doesn't work with differences. Instead it works with snapshots, and those are basically the enabler for the lightweight branching and merging system. Let's take a look at those snapshots in a bit more detail as we haven't covered them so far in this course. So, in a traditional system, we are checking in information. That information is stored as a list of file‑based changes. Basically, all information is stored as files and the changes made to these files over time. The deltas to these files are stored, which is what a changeset consists of. That's what you see here on the slide. We have a couple of files, and, over time, changes are made to this files, and these changes are stored as deltas. As said, Git and therefore also GitHub, work in a different way. In Git, we work with a series of snapshots. When we perform a commit, a snapshot of all files is made, and a reference to the way that all files look at that point is created. Of course, that should sound an alarm since now you may be thinking, well, then it will be creating a copy of each file with every commit. Well, it does not do that. If a file was not changed, Git will just store a link to the previous identical file. This fundamentally different way of working is one of the main differences between Git and other source code management systems. Now that you understand the concept of a snapshot, let's try to go back to branching because that's what we were talking about. Every time we create a commit, Git will store a pointer to one of these snapshots. The commit object of which that pointer is part will also contain things such as your email and the commit message. Also, the commit will contain a pointer to the commit that came before it. It's possible that there is no earlier commit if this was the first one, of course. In Git, a branch is, in fact, nothing more than a pointer to one of these commits. We'll always have the main branch that will typically be the default in Git, and it gets created automatically. Having now performed commits on that branch through the main branch, a pointer will point to the last commit we have made, and it moves forward with every commit we create. So that's why creating branches in Git is such a cheap operation. A branch is nothing more than a pointer to a commit. When using Git, branches are being used for almost all development. You will typically create a new feature on a specific branch. When a bug needs to be fixed, you will typically create a new branch. If you want to experiment, same thing. We create another branch. What now happens when we want to create a branch then? Well, as said, a branch is nothing more than a pointer, so a new pointer gets created. By default, we work on the main branch, and creating a branch doesn't change this. It keeps track of this using the head, which is a special point used to keep track on which branch you're on currently. Only when we have now performed a so‑called check out will we change to the new branch. Once we are now on that new branch and we start making changes, we are building these on top of the newly created branch. The original main branch hasn't changed. At this point, with no changes in the main and we are happy with the changes that we made in that new branch, we can now easily merge these back into the main branch since that hasn't changed. However, if, in the meantime, the main branch did change, we'll have to perform a merge. Now I did it mention that Git defaults to using main as the default branch. At the time of the recording of this course, Git actually still defaults to using this default branch master. But when installing Git, you can now actually change this to main. We are using main in this course since GitHub itself also changes this default branch to be named main. Please note that if you have an older version installed, it might still be using the old naming conventions.

Demo: Creating a Branch on GitHub

Creating branches is something we can do locally, of course, but also on GitHub. In the first demo of this module, let's start by taking a look at how we can create branches on GitHub itself. We will use this branch later in the module. The typical GitHub workflow is based on working with branches. We'll explore this workflow in a lot of detail later. We have already been working with a branch, the main branch, although we haven't really spent any time looking at it yet. It is currently also the only branch we have in our repo that you can see here effectively in the drop‑down where we can see that we currently have one branch that is the main branch. We can directly, from the GitHub interface, create a new branch, and that's what I'll do here. We can, in this text box here, find an existing branch. Or when we start typing, it will automatically create a new branch if it doesn't find it by the given name. So let's do that now. I'm going to create a new branch called sample‑branch. Because it does not exist yet, GitHub indicates it will create a new branch, and it will base it off of main. So let's do that now. Automatically, we have now been switched to this new branch. GitHub indicates that the branch has been created, and this branch is now even with main. Quite logical if you think about it. It is a new branch based on the main branch, but you haven't made any changes in this branch just yet. We're automatically also switched to that new branch, and that we can see here. You saw that the switch was very quick. Making a new branch is indeed a very lightweight operation in Git and therefore also in GitHub, and it is a normal thing to do. As mentioned, the branch is currently even with main. That means that all files that we have in our new branch are identical to the files in the main branch. This drop‑down also shows a bit more information about our branch. It does notice that no new commits have been made to this branch, so we cannot create a pull request just yet. Pull requests we'll dive deep into in the next demos, but pull requests are used to bring these changes back to the main branch. But since they're still identical, it doesn't make a lot of sense to bring the changes back in because there are none. We can also compare two branches from this dialog. Now, what is the use of making branches? Well, of course, making changes to files that do not impact yet the files in the other branch, in our case the main branch. So let's do that now. What I'm going to do is I'm going to go to my instructions.txt file, and I'm going to make changes to this file directly from the GitHub interface. So I'll click on Edit, and we'll add some changes here. These changes have now been made directly on the instructions.txt file in the sample‑branch. That means that only if we're in the sample‑branch we can see these changes. From the main branch, we do not see these changes to the instructions.txt file. Now why does GitHub do it on the sample‑branch? Well, that's because that is the active branch currently. If you scroll down, indeed, GitHub says that it wants to do to commit. It suggests that we do the commit, let's say, on the sample‑branch. We could also create a pull request. Again, we'll talk about that later on. We'll accept the default message, and we'll then click on Commit changes. The file has now been changed. If you go back, now GitHub says that our sample‑branch has had recent changes, and it is now possible to do a compare or also create a pull request to bring these changes back into the main. Now, our changes, as said, are done at this point only on the instructions.txt file in the sample‑branch. If we change branches here and jump into the main branch, you'll see that that file does not have these changes yet. Indeed, these changes have been made only on the instructions.txt within the sample‑branch. Now GitHub also mentioned here that this branch is one commit ahead of main. Indeed, comparing the two branches, GitHub notices that we have a change in our sample‑branch that is not known yet in the main branch, and it is therefore ahead of main. And if we've done a lot of work in this branch, or even if we just changed the instructions.txt file, we can bring those changes back to the main branch, and that we will do through a pull request. Pull requests are a very important thing in GitHub, and we'll learn about them in the next part.

Demo: Creating a Branch Locally

The previous demo was pretty short in that we just created a branch on GitHub, and you also saw that we did some changes directly on GitHub. Although that is possible, it's definitely not the way that we'll be interacting with GitHub most of the time. You'll typically be working locally, pushing your changes, including branches, up to GitHub. If you are working on your local machine, commands exist specifically to work with branches. You probably won't be amazed that this command is called git branch and then the name of the branch you want to create. If you have a local branch you want to start working with, you'll need to switch to that branch. That is what the checkout command will do. You will specify which branch you want to check out. At that point, the files in your working directory are replaced with the ones in that branch. Another command that is very relevant in this context is again the push command. You've already seen the push command in a previous module. Well, we'll use it here again. When we create a new branch, of course, that branch is not pushed automatically up to GitHub. Only when we use the push command, passing in the u parameter to specify the origin and name of the branch we want to push, will it be pushed up to GitHub. Now that we know the relevant commands, let's head back to a larger demo where you'll see this flow. We'll create a local branch, and then we'll push that up to GitHub. Now although we can use GitHub to create branches, very commonly we'll be creating new branches from a local machine and push these up to GitHub. This can be done through an IDE, such as Visual Studio, but, of course, also from the command line, and that is what we will do here. Now we are back here in our refactored‑telegram local repository, and let's do a git status to check on the status of the local repo. My local repo is currently still on the main branch, and it is up to date with origin main. Now do you remember in the previous module we added the README file and the CODEOWNERS file? Do bring those in with the git pull if you haven't done that yet. I've already done that, so mine is up to date with origin main. Now from the CLI, we have a couple of options to create a new branch. We can use, as you may have expected, the git branch command. As the name gives away, the branch command is used to create locally that new branch just like we did on GitHub in the previous demo. When you do this, we will create a branch, but we won't automatically be switching to it. There's actually a shortcut that I tend to use more often, and it is the git checkout command. The git checkout command will create a new branch if it doesn't exist yet and automatically switch to that branch once created. If the branch does already exist locally, it will automatically switch to it. So, doing the checkout basically means switch to that branch, and if it doesn't exist yet, well, just create it for me. Remember, branches are a very lightweight concept in Git. So what I'll do is I'll do the ‑b, and as the value, I'll pass the name of the new branch I want to switch to. I'll name my new branch add‑more‑instructions, and Git indicates that we are indeed switched to that new branch, add‑more‑instructions, automatically. And you see that here as well. Indeed, add‑more‑instructions is now the active branch and no longer the main branch that we have been using so far. Let us now open the instructions.txt file. We'll use the code command, and then we'll pass instructions.txt. And here Visual Studio Code opens our instructions.txt file. But hey, where are the changes that we made in the previous demo, you may be asking? Well, that will be a great question. But we don't see those here because they were made on a different branch that we created on GitHub. So these changes we're now doing, we're doing on a local branch, the add‑more‑instructions branch. The changes we made on GitHub were on a different branch, the sample‑branch. So we don't see those changes here, and there are also no conflicts whatsoever because these changes are currently made on different branches. So I'll bring in my changes, my local changes that is. In here, I'll add some installation instructions and save this, close Visual Studio Code, and we're back here in the CLI. If we now do we git status, we now indeed see that instructions.txt has been modified, but that hasn't been part of a commit just yet. So, we'll do that next. Now through the git commit, I'll create a new local commit. I'll again use the shortcut, so ‑am, and my commit message will be instructions changed. So there we go. Our new local commit has been created. Instructions changed is the message, and one file has been changed. Now at this point, this change that we've made is local. On top of that, our branch also is only known locally. GitHub is totally unaware of what we are doing on our local machine, right? So what I now want to do is I want to bring my changes back to GitHub so that maybe other people can also look at them and so on. What I'll do for that? Well, we have already seen that we're going to use the git push again. What I'm going to do is I'm going to say that this local branch, our active branch that is, now needs to be pushed to the origin. So that will be our remote. That's GitHub. And I'm also going to pass in the name of the remote tracking branch, indicating that this local branch should be linked to a remote branch on GitHub. And I'll do that by just passing in the same name. I'll want to use the same name, but then on GitHub. So I'll use here also add‑more‑instructions. That will be the name of the remote tracking branch on GitHub on the origin. And just to reinforce, the ‑u creates a tracking relationship between the two branches, so the local add‑more‑instructions and the remote add‑more‑instructions. And there we go. The remote new branch has been created, add‑more‑instructions. And a tracking branch has been set up as well. Let's go back to GitHub to see the results over there. Of course, by default, GitHub doesn't notice, so I need to refresh this page. And now it say, hey, a new branch has been created, and it had recent pushes 1 minute ago, and that is the add‑more‑instructions. So GitHub already suggests that we can do again a compare, and we can also create a new pull request to bring the changes back into, for example, the main branch. The main takeaway here now is that I have, through the git push command, pushed my local branch to GitHub, and all the changes that I've made locally are now known also on GitHub.

Branching Strategies

You now already understand the concept of branches in Git, and you've see that branches in Git are, in fact, an easy and cheap operation to perform. How you work with these branches is actually pretty important. With this, I mean when do you create a new branch, and from which branch do you actually branch off? Defining this is what is known as a branching strategy. Let's take a look at this next. As mentioned, since creating branches in Git and GitHub is an easy and cheap operation, it can easily get out of hand. You or others working on your project may start creating branches and doing changes spread across several branches. That's a good thing, but it can also be hard to figure out what is on which branch. Therefore, it's good to define a branching strategy, which is basically a set of rules to define how we work with branches. You can specify when we should create a new branch and also when not. It could be that we say when working on a new feature or fixing a bug, create a new branch. Another rule can define that when doing so, from which branch are we branching off? And typically, we'll also need a rule that specifies where we are merging back and to which branch. We are, in fact, defining a workflow for working with branches, giving team members a defined structure of working with branches. It's often recommended to keep this as simple as possible so that it can easily be understood by team members. Certainly, when working on open source projects on GitHub, it is recommended to define this as part of the project. You don't have to come up with your own ideas here. There are quite a few options already being used. Now let me show you a couple of these commonly used workflows. One of them is the centralized workflow, which may seem a bit weird when working with a distributed source code management system like Git. In this type of workflow, we are essentially working with one branch, typically the main branch, to which all changes are committed. Now while this works and will feel familiar to a lot of developers, it is, in my opinion, not the best solution when working with Git. A second option is using what is referred to as Gitflow. Whereas the centralized workflow is very simple, the Gitflow is quite complex and uses a number of branches. It relies on two long‑lived branches, typically main and development. Main reflects the stable version of the product, while development is unstable, and it's where development happens. Development itself is done through the use of several other branches. In fact, different categories or branches even exist. Feature branches are branches created to work on a specific feature rollback. Release branches are used to prepare a release, and hot fix branches are used to, well, perform fixes on code that is already in production. The idea is that we can work on several things in parallel. A third commonly used workflow is the forking workflow. Now we haven't discussed forking yet. We will do that very soon. For now, just remember that a fork is a copy of a repo. Where a developer will work on, each developer will indeed work on a copy of the repo, that is their own copy. Developers can then push their changes to their own server‑side copy, and the owner of the original repo whether or not to integrate changes proposed by other developers back into the main repo. This approach is often applied for open source projects. Finally, we have the GitHub workflow. Since we are working with GitHub in this course, I think this flow deserves some more attention.

Using the GitHub Flow

So how does the GitHub flow work then? Let's take a look. The GitHub flow is, compared to some other flows, a very lightweight flow. And, not surprisingly, it is favored by GitHub. This flow is based heavily around branches as well. We'll typically create branches for doing several tasks, such as fixing a bug or creating a new feature. On this branch, you will make the changes you need to do your work, such as creating new files, editing existing ones, or the deleting files. Everything you do, you do in this branch, and this won't affect the work of others since you're doing this all on your own local branch. You can create several commits. And once you're good to go to, you'll create what is known as a pull request. Pull requests deserve some more attention. Let's see what they are. We'll come to a point where we want to integrate the changes that we have made into another branch, perhaps the main branch. What we're going to be doing then is creating a pull request. Using a pull request, or PR for short, we're basically telling others about the fact that we've made changes on a separate branch. When we open the pull request, we're announcing those changes, and we're opening it for discussion. Others, such as collaborators, can now review the changes that we've made, and it's even possible that other commits are added to the same pull request. After things are basically approved, there can be a merged into another branch of the branch that we have just created. To support this, GitHub contains a number of screens that will help you greatly in the creation of the pull request. Also, GitHub contains an interface that allows, for example, the collaborator to review the pull request, annotate it, and so on. Here, you see the interface you'll get to work with when creating a pull request. You can enter an overview of the changes, labels, milestones, and so on. We'll learn about milestones and labels in the next modules of this course. Now this way of working is a typical way of interacting with GitHub, and it is what is often referred to as the GitHub flow. Let's take a look at this flow with a schematic overview. So we start with a branch. That is the main line of development. In our example, this is the main. But in many real‑life projects, this could be the development branch. We're now asked to create a new feature, fix a bug, or maybe we just want to test something out. So we'll create a new branch from that initial branch. Remember, when we create this branch, we don't have any impact whatsoever on the main branch. Only when we explicitly ask to bring these changes back to the main branch will there be an impact. Okay, so now we have our branch, and we're coding in our favorite editor on a local machine that is. Since we are working with Git locally, we can add multiple commits. All of this happens on a local feature branch. At some point, we'll want to get some feedback on our work. This can be very early, even with a limited amount of code. Maybe we want to have someone do a review on our code or even just on the ideas we have. Or it can also be at a point that we think the feature that we had to work on is ready, so we are going to create a pull request. With the request open, others can now participate in a discussion about the proposed changes. They can suggest improvements, indicate if something is missing, and so on. You can still, at this point, add to the pull request and bring in additional commits. Hopefully, at some point, your work is ready to make the big jump, and it can be merged back into the main branch, and you can now even delete your own branch.

Demo: Working with Pull Requests

I think it's pretty obvious we need to see all of this in a demo. So in the next demo, we'll make some changes. And when we are happy with our changes, we'll create a pull request. I'm back here in the GitHub interface, and it notices that we have three branches currently. We have three branches currently. One is the default branch, that's the main branch, and then we have two other branches that we added already in this module. We have created the sample‑branch first, and then, in the previous demo, we added the add‑more‑instructions branch. Through this filter here, we can also see which branches we created ourselves, which branches are active, which branches are stale, meaning that no changes have been made for some time. There aren't currently any of these. And then we can also view an overview of all branches again. Let's go back to Overview, and notice that on each of the lines of the branches here, we can see this button here to create a new pull request. Pull requests are very important, and they are going to be used to bring in changes made in this branch into another branch. That could be the main branch, but it could also be another branch. So if you want to bring the changes made in the add‑more‑instructions branch back into main, you can click on this button here. Alternatively, and that's what I often do, is I'll use the Pull requests tab. In the Pull requests tab, again, GitHub is suggesting us some possible pull requests it thinks could be useful because we've made some recent changes on these branches. But we can also create a new pull request from scratch. Currently, there are no pull requests open as we can see here by the filter. So let us click here on the New pull request because we want to do this from scratch ourselves. On this page, we now get the ability to compare two branches and then let GitHub decide what can be done with these branches. And then GitHub will give us an indication on what we can do in terms of bringing the changes from one branch to another one. What I want to do is what we'll typically do is we'll bring the changes made into our own branch, that will be the add‑more‑instructions branch, into main. So we'll use this direction here. GitHub now already looked at the differences between these two branches, and it saw that it can do an automatic merge. These branches can be automatically merged it says here. That's good. It means that we have no conflicts yet. We'll talk about conflicts and how to resolve them later on. Before we create the actual pull request, there's a lot of information also shown on this page. Currently, there's only one commit that would be part of this pull request. So indeed, we can have made multiple commits, and all these commits together will be the result of what we take into the pull request. Currently, we just have one. That is the commit that we created in the previous demo. There's also only one file that has been impacted, and currently there are also no comments made, and there's only one contributor involved. Again, indeed on one change on one pull request, multiple people can be involved. We also see that indeed just one single file has changed. We've got five extra lines. That's what it says here. We are currently seeing the unified view in which we can see what is added and what is removed. We just have added a couple of lines, so we only see green lines. Alternatively, what I also often use is this split view here. In the split view, you basically get the main branch's version of this file and our own branch's file, so the add‑more‑instructions branch, version of this file. That gives you an easy overview of the changes between the two branches. Now once we're happy with the changes, we can create the pull request. So let's click here on Create pull request. That will bring us to this page that creates the new pull request. This pull request will have a name, the name it got here automatically from the name of the commit that was last done. That is instructions changed. And we can also add a comment here. Pull requests are all about communication. When you've made a change, you'll want to communicate about what you have done, and that you can do here in this comment. Remember, GitHub is a social coding platform. Maybe you don't even know the people owning the branch you want to push to personally. Well, in most cases, you'll want to explain why you want to bring this code back into that other branch, and that you do here through this comment. My comment is not very good, but let's keep it at that for now. We can again still review the changes, but we've done that already, and then we'll click on Create pull request. GitHub is now checking, and it already noticed that indeed there is no conflict to bring this change into our main branch. Now just to be clear here, I can now go back to the Pull requests page, and I can then see the active, the open, let's say, pull requests. I can click on that. We already see a little bit of information about what we want to do here. I want to bring add‑more‑instructions into main. But if you want to get more information, I see all the information about that pull request. We see that add‑more‑instructions wants to go to main. We see the conversation with the comment we made earlier. We can get an overview of all the commits. There's just one commit which is involved right now. And there's also only one file that has been changed. And then, again, we get that split view. Now when working with pull requests, you'll want to get that conversation going. You may want to get a review going about the changes that are being pulled in. Say now, for example, I am now reviewing these changes, I can maybe not agree with one of these changes. So I can, therefore, go on this plus sign here. Click on that. And then, it will expand this comment option here, and that can then trigger a review process. We'll talk about the review process in a later demo, but this is how you can start that conversation about code in the context of a pull request. Now let's go back to the conversation here. And indeed here, we saw that the merge can be done without any conflicts, so the pull request can be merged in. So I'm going to click here on Merge pull request. I can add an additional comment, and then I click on Confirm merge. GitHub has now made the change as we can see here. It now says Merged over here. And the branch that was involved, the add‑more‑instructions branch, is, in fact, now safe to be deleted because we've made the changes. They are not part of the main, so we already know these changes now back in our main branch. So I'm going to go ahead and delete this branch now. We can afterwards still restore it. It's not really gone at this point yet. If we now go back to the Code tab, we now see that we have 11 commits. We have the original changes that we made on the main branch, and then we have two other commits that are made today, and that is the commit I made on the other branch, so the add‑more‑instructions branch, as well as the commit that was created by merging the pull request of add‑more‑instructions into main. So that also generated a new commit. And finally, if we go back to pull requests, currently, there are no pull requests open at this point.

Demo: Reviewing Pull Requests

Another person can then review a pull request. The pull request itself can also be attributed. We can add comments and refer to specific lines of code and the changes that we've made. Let's see this in action in this demo. So let me now show you how we can do that review process. I have gone ahead, and I've already created a new branch called installation‑file‑created. Let's compare that with the main branch to see what are the differences. So if we look here, we see that I have made two changes. I have effectively added a new file, installation.txt. And I basically started removing the installation instructions from the instructions.txt file. But something has gone wrong there. Notice I still have that Installation on IIS thing that is still there. Only the Content goes here part has been removed. So, maybe that's not really what we want. So unknowingly that my change is now really perfect, and I'm going to ask to bring this into the main branch again by opening a pull request. Let's leave the default name for now and create that pull request. So now the pull request is pending. There's two commits part of that pull request, which I want to bring into main. And there are no conflicts, so we could go ahead and merge that pull request. Now I said there is this review process, which is baked in the whole pull request scenario where someone else can go ahead and review those changes before they actually merge back into the main. So what I'm going to do is I'm going to show you that process here now. I'm going to switch accounts, and I'm going to do the review from that different account. I am now logged in as my gillcleerendemo account as you can see here, so that's that different account. And I am a contributor on this repository, so I can look at the pull regrets, of course. And since I am a contributor, I can also merge these pull requests if I agree. So assuming that this demo account is going to be the reviewer, I click on this pull request, and I will now start my review process. I get all the information about that pull request. So I can see that two commits are part of this pull request, and I also see that two files have been changed. And let's start this review process now. Let's take a look if everything is the way I expect it to be. So this installation.txt file, that looks okay. But then the instructions.txt file, hmm. It seems as if the developer has made a change here, but I don't really like it. This part, well, we don't need that here because let's move to the installation.txt file. So I'm going to click on the plus sign here and add a comment saying that this should be cleaned up properly because that is part of what is moved to the installation.txt file. I don't agree with this change therefore. So I'm going to start a review here. There we go. This is all I want to send back to the original developer, and now I can click on Finish your review here. And I'm going to say that I want to request changes. I don't want to merge this back into main in this state. So I'm going to submit my review here. There we go. Changes are requested. We can still do the pull request, but I don't want to do that. I've now sent this pull request back to the original developer. This is that conversation that we're getting around the pull request. So let's jump back to the original account. We're now back in my original gillcleeren account, and there we see now that a change is requested on the pull request before it's actually going to be merged into main. It is still open. It's not pulled back into the main branch. Now, of course, I could do this change again directly here, but let's, to learn a bit more, go back to the CLI and make the change from there. Now, of course, we make these changes directly on GitHub, so they are not known locally yet. Even those extra branches are not known locally yet. If I do a git branch command and I type ‑r, I will get an overview of all the branches I have locally. At this point, I have the main branch, the origin head points to origin main, and I have that old add‑more‑instructions branch that we created earlier in this module. I do want to get those new branches locally before I can now do that change that is requested. So I'm going to do for that a fetch that will bring in the branches added remotely on the origin. Indeed, a new branch, installation‑file‑created, has been found on origin on GitHub, and also sample‑branch is also a new branch that we still have own GitHub as well. Now origin installation‑file‑created is known locally. That has already been fetched. That code is already on my local machine, so I can now simply switch to that bench. Do you remember how we switched to branches? Indeed that we do through the checkout command. So I'm going to do git checkout, and I now pass in installation‑file‑created. That is the branch that is fetched from GitHub. Automatically, we switch to that new branch, and it already is tracking that remote branch on origin. Let's now open that instructions.txt file. That was the one we needed to clean up. So here's that file that still had that Installation on IIS thing, which the reviewer didn't like. So I'm going to clean it up. There we go. Delete all that. Save this instructions.txt file. Close code. And I'm back here. So I'm now going to commit my local change as a new commit on the installation‑file‑created branch. I'm going to use git commit again ‑am. And as comment, I'm passing instructions updated for PR reviews, so for pull request review. A new commit has been created, three lines have been deleted, and we simply now do we git push to push those changes back up to our remote tracking branch. And that looks good. Let's jump back to GitHub, and we are back in the demo account, so the reviewer account. And that now says, hey, new changes have been made to what is in this pull request since we last viewed it. Let's click on View changes. Here the original development cleaned this up, and I agree with the changes that were made. So I review, and I approve of merging these changes. So I click on Submit review. GitHub checks if we can still go ahead and merge the pull request without conflicts. And we then merge the pull request, in this case, as the reviewer account, so the demo account. I click on Confirm merge, and there we go. The changes made in installation‑file‑created are now merged into the main branch.

Merging with Conflicts

Having no conflicts when performing a merge can happen, but definitely won't always be the case. That is for sure. Now don't panic as GitHub has tools to make the merge process, basically where a manual intervention will be required, an easy one. So merge conflicts can and will happen. If you have made a change on a line of code that someone else has made a change on as well, this will cause a merge conflict. Git will to know what to do here. Similarly, if I would make a change on a certain file and commit that, while someone else has already, in another commit, deleted that file, that too will cause a merge conflict. Now, before we can perform the merge, all conflicts will need to be resolved. Performing the merge can actually be done inside GitHub. GitHub comes with an interface that you can see here, allowing us to make changes and resolve the conflict. It's definitely also possible that the change you have made locally will cause a conflict with another already pushed commit. In that case, you'll have to perform the merge locally. Doing this can be done using a diff tool. Quite a few exist, including KDiff, vimdiff, P4Merge, and Beyond Compare.

Demo: Merging with Conflicts

In the next demo, we'll perform a merge where we have to resolve a conflict. We'll do this using GitHub's interface, as well as locally. And being able to merge without any conflicts, well, that's the ideal situation. But in a real‑life project that won't always be the case, so we'll get merge conflicts. And, of course, that's no reason to panic. We need to solve the conflicts, and we can do that in two ways. We can do it through the GitHub interface, or we can use the CLI for that. Now let me show you how I can do that. I've gone ahead and I've already prepared some changes. We're currently in the main branch. And in the index.html, on line 14, I have removed the three exclamation marks that we had before. Now before I made that change directly in main, I also have made a different branch, the changes‑to‑index branch, and that contains an index.html file that has another change on line 14 where it says Welcome to our awesome site of our company. If you want to bring these changes back into main, we'll get a conflict because we have changes on the same line. That will trigger a conflict when merging. Let's try it. Let's see what we get. So we'll go to Pull requests, and we'll create a new pull request. And for now, I'll use this shortcut here where I can bring the changes into main. And now for the first time, we see that GitHub is saying sorry, I cannot automatically merge this. If we take a look at why this conflict is occurring, we indeed see that on line 14 in our index.html file, we have a conflict. In the original main, it had the three exclamation marks. But the one that we want to bring in has the of our company. GitHub does not know what to do with this change, so it cannot automatically merge things. We'll go ahead and I will create the pull request. As indicated by GitHub, we shouldn't be worrying. We can still create a pull request, so let's do that now. And then, we arrive here on this page that says, well, we have a conflict, so we cannot automatically merge this for you. You'll have to fix this before the changes in this changes‑to‑index branch can be pulled into the main branch. So, we can solve these conflicts in two ways as said. We can do this directly in the GitHub interface, but we can also do it from the CLI. Let me show you these two options, and we'll start with the GitHub interface to resolve these conflicts. So GitHub already gives us this resolve conflicts, and notice that the Merge pull request button is still disabled. So I click on Resolve conflicts. We now get the merged file that contains the changes on the changes.index branch and the original line on our main branch. This is an editable file. I can now change this file directly here. And I guess the changes made on the new branch actually make more sense. So I'm going to remove this part manually, and I also need to remove this comment added by GitHub itself. There we go. And let's remove the new line here as well. And so the changes in our branch now become the final solution. So we'll click on Mark as resolved, and now a new merge commit is being created. And now we go back to our pull request page, and now GitHub notices there are no more conflicts. The pull request can now actually be merged into main, and so let's do that here. We confirm the merge. And as before, the changes we wanted to bring into main through our pull request have now been merged in, and we can delete the branch as we did before as well. Now we've seen how we can resolve conflicts through the GitHub interface, but it's also possible to do this through the CLI. Let me show you that next. First, I'm going to jump back to my main branch, so I'm going to do a git checkout main. Now Git already says that my local main is behind by two commits, on the one on GitHub that is. So, we'll do a git pull. The change is made to index and installation.txt that we did already earlier in this module and are brought in locally. Let's now go and make a local change. So I'm going to go to Visual Studio Code again and change the index.html. We indeed see the the merged version of our title, and we'll make again some changes on our index.html. So I'm going to bring in that three exclamation marks again. We'll save that locally. There we go. We now have a change, which is pending locally. We haven't committed it yet. But let's jump back to GitHub for a second here. To save us some time, I'm going to do a change directly of index.html in the main branch. We'll edit the file here, and I'm going to bring in again one exclamation mark. I think that is more than enough in terms of enthusiasm, right? So let's commit those changes directly to the main branch. Those are indeed on GitHub. Now since they are on GitHub, locally I don't know anything about them just yet. It could be that someone has made a change. It has been pulled into the main already that we don't have locally. We have been working as offline. Now I have been working here locally, and I want to bring those changes also to GitHub. Now I have worked here directly on main as well, so when I will push this up to GitHub, I will definitely get another merge conflict. Let's try that. I'm going to make a local commit first because we don't know anything about what happened on GitHub itself, so on the remote. I'm going to simply do a git commit and say index changes. A local commit has been made. Now, I want to push this. We are on main, but remember, there are changes also made on main on GitHub. So I'm going to do a git push, and GitHub is rejecting my change. We already saw this before because what it says here is that main has changes that we don't have locally. All right, we know all that. So I need to do a get pull first. Otherwise, we cannot push our changes. We need to bring in the changes made on GitHub on the remote. And now we indeed see the same thing as we saw directly on GitHub itself. We are entering into a merge conflict state. So, since we have a conflict in index.html, we'll need to open code again, and we'll go back to our index.html. And now notice again, we get these markers. The first block points to our local version that conflicts with the incoming change with just one exclamation mark. Now Visual Studio Code offers us some help here. I'm not going to use it because you may be using a different editor, so I'm going to do this manually. Now I have created this local change, so I think this looks better, so I'm not going to use the incoming change. So I'm going to remove this mark in here, and do not forget that we also remove this part. There we go. So this will now be the final version that we want to push up to our main. Let's save this. Close Visual Studio Code. At this point, we're still in the merging state, so we need to do another commit. I'll use git commit ‑am, and I'll pass in a message, resolve merge conflict. We're back in main, so the merge conflicts have been solved. Now I can try another git push, and there we go. Our local changes that originally had the conflict have now been merged back to our remote. Let's take a look at the final result on GitHub. So if we look back here at main and we look at commits, we indeed see that resolve merge conflict was the last commit after index changes. As for the result, index.html is the result of what we did locally.

Demo: Creating a PR Template

When we have a larger project, it makes sense that collaborators follow a certain path to create a pull request. For this reason, we can create a PR template. We'll do that in this demo where we'll use this template to create a pull request. Now the pull requests we have created so far were not the best examples of how a good pull request should be created. It could definitely be adding more useful information to it. It's better to create a template so that contributors will always be creating a pull request in a certain format. We can create multiple templates for pull requests. We'll create just one here. Now in order for it to be recognized as a pull request template, the pull request template needs to have a certain name. So we'll create one directly to through the GitHub interface, and I'll do that as part of a new commit. So I'll create a new file here. And in order for it to be recognized, the file name should be pull\_request\_template.md, and I'm going to create it directly in the root. As we have already seen, pull request templates are also created in Markdown. There we go. We can use Markdown to create that template. In the template, I ask that pull request creators do enter a description, as well as an indication of the type of changes. Is it a bug, is the feature, and so on. If we look at the preview, this is how it will be rendered. People are asked for a description, and they can indicate the type of change. Let's commit this directly to the main branch. So now we see here that the pull\_request\_template is now indeed in our branch as a regular file. If we now create a new pull request, indeed, I will get this template to fill in. This is the information that the contributors expect from me as a creator of a pull request. So I need to enter the description and also indicate what type of change I'm suggesting.

Demo: More Options with Pull Requests

In the next demo, we'll continue to work with pull requests. I'm going to show you how we can revert a pull request, see the history, and use the blame feature on GitHub. And reverting a pull request on GitHub will create a new pull request that basically reverts the changes made in the original already merged pull request. Let us see how we can do that. I have gone ahead and I created a new pull request, which I already merged into main. If we go to the Pull requests page, we, by default, see no open pull requests, which is okay, but that is because I have a filter active here that says only show me the open pull requests. If I remove this filter, I will indeed see all the pull requests that we have done so far. I've created new pull request that I, in fact, already merged into main. It originated on a branch, the revert‑me‑branch that I also created. Now assume that I'm not happy with what was merged into the main with this pull request. I can now go here on this pull request and revert it. This will, as said, create a new pull request that will revert the changes made in this pull request. As so indeed, a new pull request will be created that will revert my pull request with ID 5. We'll talk about these IDs later on. But you'll already notice here that this points back to the original pull request with ID 5. I'll create that pull request now. And that pull request we can then also merge if we want to. It could also be that this creates new conflicts that we also need to solve manually again. Now when we go back to the Code tab, we have this button here that shows me all the commits. And indeed, here, we see all the commits that have been made on our branch. In this case, this is the main branch that shows a list of all the commits that we have been doing. For each of these commits, we can take a look at what was changed. For example, if we look here at index changes, we see the changes that we have made in there. That was one where we added those two exclamation marks, these are the files that were changed, and so on. This, of course, we can also do for our other branches. For example, the installation‑file‑created branch had this particular history, and we can also browse those individual commits. If we go back to the main branch, we can browse the repository at a certain point in time. So now we're looking at the repository in the way it looked with this commit. If I go to the index,html here, I can now click here on the line I want to look at. I want to see the history of our title here. I then get this ellipsis icon here, and I can now see that git blame feature. If I click on View git blame, I will see all the changes made to this line. So indeed, I can now browse through the history of all the changes that were made on this particular line on the index.html. Now, of course, my history is not that large. If we look at a more active repository again, we can see this git blame feature in all of its glory. Let's take a look. I'm now here in a random file, the MvcServiceCollectionExtensions part of the aspnetcore MVC repository. When I click on one of these lines here, I get this ellipses again. I can view the git blame feature again, and then I can see the history made to all the lines inside of this file.

Demo: Setting the Default Branch

We can set a specific branch as our default branch on GitHub. We'll see how we can do this, as well as how we can restore an earlier deleted branch. The default branch is considered the base branch in your repository against which all pull requests and code commits are automatically made, unless we specify a different branch, of course. So the default branch, as the name gives away, is used by default when using the GitHub interface. It is stored on a per‑repository level, and we can change it through the GitHub interface. When we go to Settings, and then we click on Branches, we indeed see what is the default branch. We can change this. We can switch here to another branch. We get a list of all branches that we have, and then we click on Update if we want to change that. Now on this page too, you can also see branch protection rules. Now branch protection rules can be used to basically block users from automatically pushing to, for example, the main branch.

Forking a Repository

In the last section of this module, let's talk about forks. I've mentioned forks before. Let me explain them in a bit more depth. So what are forks exactly? To put it simply, a fork is a copy of a repository. Thus, it's, in fact, nothing more than a copy of an existing repository. We can make changes to the copy without impacting the original repo. Now, of course, being GitHub, it's more than just a blank copy. When we create a fork, the copy basically remains linked to the original repo so that if there are updates in the original repo, we can get these changes into the fork as well. Also, the opposite is true. If we have created a fork, it is possible from this fork to create a pull request so that our changes can optionally be merged back into the original repository. Forking is available for everyone. Once you have created a fork, that fork is now your own repository, and you can manage it entirely yourself. So that means that on the new project, you can bring in collaborators, change the repo settings, and so on. Forks are typically created to test ideas. And yes, forks are definitely related to working with branches. Forks are also a GitHub thing. They don't come with Git itself. That is why I talk about them here since they really highlight the social aspect of the platform. One major difference between forks and branches is that branches all work on the same repository; whereas, forks work on different repositories. In terms of being able to create pull requests to merge your changes back to the upstream repository, things are pretty similar here. So before we head back to GitHub, let's take a look at the graphical representation of how forking on GitHub works. It all starts with the repository. As a GitHub user, well, I think I'll need to make some changes to it so that it works perfectly for my situation. A good choice, in this case, is therefore forking this repository into my own account. At that point, I have created a copy, so that copy will now be my origin. Things are also still on GitHub as well. This is now my own repository, so I can now do what I want with this repository. For example, I can decide that I want to make some changes, so I will clone and then pull my changes to my local machine. Changes that I've made will then be also pushed back to my own fork. The original repo isn't bothered by all this. None of the changes that I may have pushed back to the origin are automatically going back to the upstream repo. Only when I think that that actually would be a good idea can I submit a pull request to the upstream repository. Of course, just to be clear, from a single repo, multiple forks can be created. If someone else also likes this repo, they can do the same process as I originally did. These forks aren't related either, apart from the fact that they still know that they are forks from the original upstream repository.

Demo: Forking a Repository

In the final demo of this module, we are going to create a fork of a repository and make changes to that repository. I'm now looking at our repository again through the eyes of someone else. I have the repository open through the demo account. And now the active user here, so the demo account, can now also click here on Fork. This will then clone our repository into this user's account. So I can click here on gillcleerendemo, and I want to fork the refactored‑telegram repository into this account. Let's do that. What I've now done is I've created a copy owned by the gillcleerendemo account, but it is still linked. It's forked from the gillcleeren refactored‑telegram repository. These are now two repositories. They're two separate repositories. This is the demo account's personal copy of this repository. It's important to understand that at the point of the creation of the fork, the entire history of that original repository is copied. Indeed, I can look here at all the commits in the main branch. That's an exact copy of all the commits made on the main branch. But indeed, I also have all the other branches at were copied over as part of the fork to my demo account. Now they are now standalone. That means that if I am in my demo account and I change the index.html, I change it to demo company, I commit to main. Now these are changes made in the forks during the copy, and they are not known in the original repository. If we jump back to the gillcleeren account and we look here at the index.html, also in the main, the changes are not visible there. That is because the fork is a copy that can live independently. If we want, we can also, from the fork, create a pull request back into the upstream repository to bring the changes made in the fork back into the original repository.

Summary

We have completed together another hopefully very exciting module. Let us summarize what we've seen here. We've seen that Git itself has a very strong support for creating branches. Due to it's nature, it's much more efficient in creating branches and other source code management systems. GitHub has based its GitHub flow on this concept as well. Through the interface, we can easily work with branches and pull requests. The concept of the pull request also already emphasizes the collaborative nature of GitHub. And to close this module, we looked at forks, which are used to create a copy of a repository. Another module done. Great. In the next module, we will look at how we can work with issues in GitHub. Stay tuned for more GitHub fun.

Tracking Issues and Creating Releases

Module Introduction

Building software is more than writing code. Managing the project means following up on bugs, issues, and at east about the application. I'm Gill Cleeren, and I want to welcome you to the Tracking Issues and Creating Releases module in the GitHub: Getting Started course here, on Pluralsight. Let's see what we will be learning in this module, shall we? So what will we be covering in this module? Well, I have four items for you that I think are closely related to how you can use GitHub to manage a software project. We'll start with what I think is the most important one, issues. You'll see that issues on GitHub's way of managing work items. Next, related to issues, we'll also look at milestones. Then, we'll look at tags. And finally, I will show you how to label a certain commit to be a release. I hope this all sounds interesting to you. Let's get started straightaway.

Working with Issues

Issues are a fundamental block for GitHub in allowing us to manage our work through the interface. We'll start by taking a look at what types of issues GitHub defines. The word issues can trick you in thinking that they are a negative thing. Hey, there's an issue with your code. But that's not the case. In GitHub, issues is just a name to indicate everything that is going on from indeed, yes, bugs, so the negative one, to enhancements to work tasks and even ideas. All these feedback items are basically collected in one large tracker called issues, and each repository will have an issues tracker. New features that need to be created can be tracked by the creation of an issue. Issues can also be linked to a pull request. We can then when we merge the pull request close the corresponding issue. Closing the issue does not mean that the information is gone. All of the related information remains available in GitHub. When we use issues, we can also be notified using the notification system. Imagine that you have created an issue. When someone then creates a comment on that issue, you'll get notified. This helps in staying up to date regarding the state of the issue. Google requests can also be associated with an issue. We can with a pull request mention an issue number combined with a keyword. Issues can also be used to work with milestones, which I will explain you very soon. For now, you can see a milestone as a point you want to reach. That point is reached when you complete a number of issues, hence the relation of issues and milestones. Since issues are a GitHub feature, the site has all the required interface to work with. Probably, the most relevant screen in this area is, of course, the issues tracker itself. This issues tracker is linked to a repository. In the main screen of the issues tracker, you can see a list of all open issues by default. Of course, we can also set it to display other lists. Let me explain you one more thing before we go to the next demo, and it is labels. Labels can be applied on issues, but also on pull requests. They are typically used to organize and prioritize items, such as issues. Again, labels are a GitHub thing. Quite logically, if you think about it, they are added on issues and pull requests, and they too are specific GitHub things. By default, GitHub already comes with a set of built‑in labels. But if these do not suffice for you, you can create your own set or even delete the built‑in ones. By default, nine labels are built into GitHub, and so they are available on each repository for you to use out of the box. Here you see the list of default labels. Assume, for example, that you are creating an issue that describes a bug. You can then use the bug label to categorize this issue. When you and other people working on the repository consistently apply labels, it will also make the overview in the issue list much easier to digest.

Demo: Working with Issues

All right, time for a demo on working with issues. We'll take a look at the issue tracker related to the repository, and we'll create an issue. We'll also take a look at how we can link an issue with a commit, and we'll take a look at working with labels. So far, we have been working with our repo, well, pretty much on an island. Now while that's okay for a demo, that is typically not how GitHub works since GitHub is a social coding platform. People will report bugs, and those we can follow up on through the issue tracker. The issue tracker is available under Issues, and it is, of course, also under the repo. Each repo has its own issue tracker. Now, you may not see this Issues button here at the top. If that is the case, well then go to Settings and scroll down here under Options. You can see that issues can be enabled or disabled for your repository. Let's go back to the issues tracker, and you can see my repository issues page. It contains just one issue. We'll add a new one in just a minute. There's a filter here at the top that is filtering by open issues. I can remove this filter and set it to just give me all issues, and also closed issues will be shown here. We already created an issue earlier in this course that I have closed just before this demo. There is also filter here at the top that I can use to filter the list of issues. I can filter by author, for example for all issues created by Gill Cleeren. I can also filter based on a label. We will talk about labels in just a minute, but I think you can already imagine what a label will do. It's basically a tag that we place onto an issue. We also filter based on who the issue is assigned to, and there's also some sorting issues. I think you can imagine what this will do. Now, this is, of course, not very useful if you have a very small list of issues. If we go to the aspnetcore repository, we'll find an issue tracker that contains many issues, many more issues than what we have in our repository. Currently, there's about 2,700 issues, which are open. And here we can also filter. For example, we can also see who created a certain issue. So it's possible to go into an issue and read more information about the issue. Here too I can write comments, so we can also get that conversation going about a certain issue. Let's return to our own repository where we will create an issue and experiment with that. Let's created a new issue. There's a New issue button here at the top right. I'm going to create a new issue that the background color of the page needs to be green. I can also add some comments here, and again that is using Markdown. Now this issue, see it as a task in this case. It could be a bug. It could be something to remember. Basically, anything goes when talking about issues. You can basically put all the things related to the repository inside of the issue tracker, and then you can follow up onto it. There's a couple of options that we have when we create the issue. For example, I can assign the issue to someone. I can assign it to myself. The issue is now ready for submission. Let's click on Submit new issue, and then the issue will appear as an open issue. From here, I can also start a conversation by adding a comment here. And from here, I can go ahead and make some changes to the issue. Of course, we can edit the current issue, but we can also from here directly, example, apply a label. This could be an enhancement, for example. Now the issue is labeled with enhancement, and that can be also used as a filter. We'll talk about labels in just a minute. I can also assign it to a milestone. We'll talk about milestones very soon. But they can be used to group certain issues to reach a certain point, a certain milestone. If you go back to the list of issues, we indeed see that our new issue has been added. It's open. It's green, meaning that it's found through the current filter, and it's also labeled with the enhancement label. And it's also sent to me, as we can see here. When I go back to the details of the issue, we can, from here, also write a comment or close the issue. But I'm not going to close the issue right away. I'm going to show you something else first. Notice here at the top that this issue, Background color of page needs to be green, has received the number 8, #8 it says here, and that 8 is also visible in the URL here in my browser. That is the ID of the issue. That's quite logical. Now, if you think about it, where did GitHub get number 8 then? Did I make some issues that I've hidden before? Well, no. Let me show you. Within the repository, a number of items are grouped together when it comes to numbering. If you look here at the issues, we indeed see that I think I currently have three issues in the repository. So that's not where we got to 8 from. But if we now look at pull requests, you can see here when I moved the filter for open that also the pull requests have received an ID. And this #6, #5 is a way to point to a certain issue, a certain pull request in comments. For example, if we go to this pull request here, and we go into the comments, and we write here Fixes, and then we write a hash, and I will see a list of issues, pull requests, and so on that can be linked to this current pull request. For example, this pull request might be fixing our background color issue. When I now preview this, you see that I can hover over this, and I can see the short information about this issue. And when we comment, also, this link to #8 is shown. I can click on it, and we jump directly to our issue. Now while we can reference the issue, we can also close it directly. For example, let's create a new pull request. I've created a sample branch here, close‑issue‑keyword‑test, and I'm going to create a pull request with that. I'm going to create a new pull request, and I'm going to write here a keyword, Closes, and I'm going to refer to our issue, Closes #8. I'm going to create now a pull request. So in the pull request message, I'm referencing again our issue with ID 8. There we go. Now you see here that closes is highlighted. It actually will when I now merge this pull request close the referenced issue. So why it is now useful, you've made all the changes to fix issue 8, well then you create a pull request and you reference issue 8. When I run the pull request, it's going to be merged. You will see here under Issues that our issue is indeed closed automatically. If we remove this open here, then indeed we see that background color is now closed. Let's create another new issue here. Let's call it sample issue. Let's take a look at the labels. Labels are indeed a way to triage your issues. I can take your own labels and then label, indeed, my issue with a certain label. For example, this could be a bug. There we go. One issue can also be labeled with multiple labels to be clear. If I want to, I can also go ahead and create my own labels. I can click on Edit labels, and this will take me to this page where I can create my own labels. These are the nine labels that you get for free basically with every repository, but you can use your own triaging system. I can click on New label, and let's say that this could be a task, and I'm going to use a certain color. Let's take this nice pink here. Click on Create label. If we now go back to Issues and we take again a new issue, then we come back to our issue that we were currently editing, now I go to Labels. And from here, I now see my label task that I can use here now as well. Now the issue is labeled with the task label. I can submit it. And this now is also visible in the list of issues, both here on the issue, as well in the list of labels, which I can then use as a filter.

Demo: Creating an Issue Template

We've looked before at working with templates when creating a pull request. In a similar way, we can create a template for issues being submitted. Let's look at that in this demo. I've already created a pull request template. That's the one you saw over here. Now, we can also create, in pretty much the same way, a template for creating issues so that people creating new issues know how they should be doing so. So we have, as contributors, an easier way to look at these issues. Now you can also do it manually, but I'm going to go through Settings, and here I have the ability to set up templates. Now scroll down here to Features, Issues, and here I have the button Set up templates. GitHub already comes with a number of predefined templates for issues. For example, I can say I want a bug report. There we go. I can preview this. This is what the bug report looks like. It contains typical information that you'd want people to enter when reporting a bug. This is useful information, labeled what is required, as well as what is optional items that they can add. I can also add another template for a feature request so that when people suggest a feature, they can also follow a certain template for that. Now with these two templates added, we're going to click on the Proposed changes button, which will then create a new commit in our repository. Now this we are going to directly do in our main, so I'm going to add a new commit message, Creation of issue templates, and I'm going to commit this directly to my main branch. There we go. And now GitHub has noticed indeed that issue templates have been created for our repository. Automatically, GitHub will place them in a certain specific directory, the .github/ISSUE\_TEMPLATE directory. Here, now, we see our two issue templates, bug\_report and feature\_request. Now, these templates again are known by GitHub so that when someone goes ahead and creates a new issue, they will get the ability to choose which template they want to use. So I can now click on New issue, and I get a list of templates that are included in this repository. I can still open a blank one, but I can go ahead and create a bug report based on this template. And we can enter all the required information about this issue.

Creating a Milestone

We have now had a good look at issues. And for the next topic, we will stay in the same area where we'll look at milestones. Of course, the first thing is making sure that you understand what a milestone really is. If you think about the concept of a milestone in real life, it's a point that you reach. Well, in GitHub, a milestone is pretty much the same. It allows you to track if we have reached a certain point. Typically, this is done using a list of issues that we tracked the progress of. Milestones are, again, a GitHub thing, and so working with them is done in the interface. A milestone will typically include a due date, a completion percentage, and a list of open issues and pull requests. We can, within a milestone, also prioritize the associated issues.

Demo: Working Towards a Milestone

Now instead of talking about milestones, let's go straight back to GitHub and take a look at how we can create and manage a milestone. We're getting ready for the launch of our site, but there are still a few things that we need to fix before we can actually do so. A milestone can actually be created to group these things together and get an overview of the issues that still need to be fixed before we actually do that launch. Now we are again here on the Issues page here, and we see the milestones we have in our repository. Currently, there are no milestones, but we can create one and group a number of issues or pull requests together before a certain release. Let's click on Create a milestone and call it Launch of the website. I can add a due date, but let's leave that open for now, and let's add a comment here as well. With the milestone ready, let's create it. So here we see now the details of the milestone. It's currently open. Let us now go back to our issues. I've created a few sample issues here, and I'm going to associate these with my milestone. So I'm going to open my issue here, and I'm going to link that with the Launch of the website milestone. There we go, and that's now been associated. I'm going to do that with a few other issues. So now I've changed my issues, I've changed four in total, and I've associated these with the launch of the website. In other words, these will need to be fixed before we can actually do the launch. If we now go back to the milestone, we'll see that indeed we are at 0% complete because these issues are still open. If we go back and, for example, say that we have fixed the logo scaling issue, we could, of course, associate this with a pull request But to save us some time, I'm going to close this now. There we go. This issue is now closed. And if we go back to the milestone, we'll see the milestone is now 25% complete. Some milestones can be used to track progress on a number of pull requests or issues.

Adding Tags

Let's now take a look at how we can work with tags. Before we start a discussion about tags, I want to make sure that you remember the following. A branch points to a snapshot in the history of your project. It indeed has a reason that I'm saying that, and that reason will become clear very soon, in fact already in the next slide. In Git, we can indicate that a certain point in history is important. Many other source code management systems allow for this. Through tagging, we can indicate that a certain point has a certain value to remember. For example, this can be used to indicate that a certain commit is v1.0. In Git, we have two options to create tags. A lightweight tag is the first option, and it basically is nothing more than a pointer to a certain commit. The second type is an annotated tag. Such a tag is, as the name implies, more complete as it will contain information, such as the email, date, and also perhaps a tag message. In general, these are the tags we'll typically want to create. Tags can also be added later on, so that means if you already have other commits after the one you had planned on tagging. Adding a tag is done using the git tag command. Adding the ‑a parameter makes this into an annotated tag where you can then add other information. Once we have our tags pushed to GitHub, we can then explore these via the website. With a tag, we can download the corresponding code that is linked to that a tag. And remember, it's nothing more than a commit. So, via GitHub, we can download the code that is associated with that tag. Through GitHub also, we can then see a list of commits that leads to this commit. Also, some more information about the tag, such as the message, is available in GitHub.

Demo: Adding Tags

Now that we have an understanding of tags, let's go back to the demos. We'll start with the creation of tags locally, and we'll then push these up to GitHub. Once we have them pushed up, we'll take a look at the tag page on GitHub. Now I'm back here to start this demo in the CLI, and I'm going to start by showing you the Git history of the main branch, and we can do that here in the CLI by using git log. The log command will show me a list of the history that has happened on my repository in descending order. Indeed, I have done a pull, and the last commit we did on main, I believe, was actually the addition of the issue templates. So indeed, we see here that the last commit with ID b8 and so on was on origin main, and that was the creation of the issue templates. We can see all the other commits that have happened as well, and this is a long list. So we can actually scroll through it, and then we'll see all the commits that have happened. So through the git log command, we can indeed see the full history of the main branch of our repository. Now, let's see what tags we currently have. I can use to git tag command, and we'll see, well, nothing. There are no tags at this point that we have created. Let's create our the first tag and then scroll up for a second here. We have, of course, our head, which is pointing to the main and, in fact, to the last commit on the main. Since the head points to that, when I create a tag without any annotation, that tag will point to that commit. So I'm going to do a git tag again. This is going to be a lightweight tag. What I'm going to do is I'm going to say stable. Stable will be the name of the tag. And I'm going to point that to main. And now, I've created a lightweight tag that points to the latest commit on main. If I didn't git tag again, we'll see that we currently have one tag available, and that's a stable tag. If I do another git log, we'll see these tags also pointing to these commits. Now, instead of using the git log directly again, I'm going to use an alternative here. I'm going to use git log ‑‑oneline ‑‑graph. Here we go. Now we see a more readable overview of what has happened on our main branch. And notice that the latest commit, that's the b8b, is also tagged with stable. Now let me create another tag, and I'm going to create an annotated tag. So I'm going to add a message to that tag. And let's say that, for example, for this version here, the 3aO, I want to create a tag. I'm going to use git tag again, and I'm going to use an annotation, so I do ‑a, and I'm going to say that this will be v0.1. That's the annotation. I'm going to bring in a message, so I use ‑m, and the message will be 0.1 release, and I point this to this commit here. That's the one that set, so the 3a06a16. Let's do another git tag, and we should now have two tags, stable, that's our regular lightweight tag, and 01, that's our annotated tag. Let's do another one for v0.2, and that's going to be a tag that points to this commit, so 7e02dbc. Hit Enter, and now we have indeed three tags available. So now we have these tags locally, so those are now commits that we have given a name. It's nothing more than that. Let us now make sure that we get these also known on GitHub. So what do we typically do when we have done something locally and we want to let GitHub know about that? We use a git push. Let's go and do a git push. And it says everything is up to date. That's weird. So let's see. Let's go back to GitHub, and I'll do a refresh here to be sure. But it says here zero tags. Our tags don't seem to have gone up to GitHub. Indeed, by default, where we do a git push, tags are not pushed up to the remote. So I need to go ahead and now make sure that these tags are also pushed. So I'll do that again using a git push, but I'll do ‑‑tags. And now, you see that it is pushing our a tag to GitHub. And if we now do a refresh here on GitHub, we'll effectively see our tags being shown. Indeed, we see the stable 0.1 and 0.2 tags. Now let's take a look at the annotated tag, the 0.2. You see quite a few things here that are related to this tag. So this is related to that 7e02dbc commit. We can click on that. And then we see the changes made in that commit. There was not a lot of changes in there, but anyway, we see the changes related to that tag. We also see which commits have been made since this tag was created because it was not the last one. And most importantly, what we also get here is a zip version of the code associated with this tag. There we go. I can click on that, and that will open a zip file that contains the source code at this point in time. We also see that this contains a message, and that is that 0.2 release. If we go back to Tags, we also see that we can download from the overview that zip file. Let's take a look at the lightweight tag. It's very similar what you get. Of course, it doesn't contain a specific message. It just takes the commit message of the commit it was associated with.

Publishing a Release

You've now seen tags. Let's now turn our attention to releases. Already a word of warning though. Tags and releases are very closely related. Releases are based on tags. And as mentioned, both are closely related. However, with a release, we can add other information. Probably the most notable difference is that with the releases, we can create release notes. Through the GitHub interface, we can create release notes in Markdown format. And on the same page, we can also add binary files related to this release. Let's return once more to GitHub and explore how we can work with releases. We'll create a new release and take a look at the result. Now we've just seen tags, and we'll see that releases are very closely related to that. Now releases are a way to package software in GitHub and distribute that to our users. It basically replaces some downloads that we need to, for example, point to from our README file. With a release, we can typically link in a binary file that people can then use to start the installation, as well as release notes. The big difference, in fact, is release notes. Let's take a look. If we click here on Releases, you'll see that the overview is actually very similar to what we got with tags. The page actually looks very similar. So automatically, when we create tags, GitHub will also create for us a release based on that tag we have created. Now I do want to create my own release, there we go, by clicking on Draft a new release. And from here, we can now create a new release, and let's call this v0.3. Let's say that we are creating an installation package of the website for installation on a server. We can, of course, add some comments. And instead of just having GitHub create a zip file based on the sources on a certain point, I can now include a binary file that will then get associated with this release. Let's call it installation.zip. So this could be something that you have created separately and that you want to include as part of the release. Now since this is 0.3, I think it is definitely prerelease, and we can indicate that here as well. When we click on Publish release, this release will appear as part of the releases. So we now have a release called Website installation, and this now contains the installation zip files or the binary file, as well as the sources. Now I didn't point to a specific commit, so it automatically goes back to the latest commit on the main branch. So if we go back to Releases, we indeed see that this prerelease is now included here as well. And by the way, you can also go to Tags, and from a tag, you can also create a release from here. And that will automatically be based on that existing tag. It will create a new release to which you can then also associate binary files.

Summary

And we have reached the end of yet another module. I hope that you have learned a lot again. Let's quickly summarize what we have learned in this module to wrap it up. I hope by now that you have understood that GitHub offers a great selection of tools that help you in managing the project. It all revolves around the issue tracker, which is the central point to store information in the form of issues, including bugs, ideas, and feature requests. We also looked at tags and releases. They can be used to create a named commit in the history and give us the option to download the code related with that commit. In the next module, we'll look at how we can add automation in GitHub using GitHub Actions. See you there.

Automating Tasks with GitHub Actions

Module Introduction

GitHub allows us to do many tasks as part of the software development lifecycle. It is pretty mind‑boggling. Some tasks are pretty repetitive, meaning that we'll need to do them often. So why not automate them and save us some valuable time? That is why GitHub Actions comes in, a way to automate tasks inside GitHub, and that will be the focus of this module. I'm still Gill Cleeren. Please let me know if you have any questions via the discussion board here, on Pluralsight. All right, how we will be learning about GitHub Actions then? Well, a good place to start is understanding what they are, and that's what we'll kick off the module with. We'll create our first workflow in this section too. Then, I will show you how we can really put these actions to use in the next part called Automating Tasks. Finally, we will learn about continuous integration, and we'll see how GitHub Actions can be used in this context. Let's dive in.

Understanding GitHub Actions

So first, let's see what GitHub Actions is all about then. Like I mentioned in the introduction already, GitHub supports many tasks related to the software development lifecycle, also known as the SDLC. The SDLC is the process used to execute a software project, and it contains several phases, including the planning, defining the requirements, developing the code, testing the project, and deploying the application. We have already seen that, indeed, GitHub helps us with a lot of this. Think, for example, of issues we've looked at in the previous module. They can be used to track the status of bugs reported in the testing of a project. When new issues are created, we'll often give them a label. Tasks like these can actually be automated, making us more productive. GitHub Actions will be the place to create this automation. Many tasks, or better workflows, we'll typically do manually in GitHub can be hooked into actions, helping us in several ways to avoid repetitive tasks. A workflow will typically contain different tasks that will be triggered. And what will trigger this workflow? Well, events within GitHub are typically used for this. An event can be many things. For example, pushing code to the main branch can be an event that we'll use to trigger a workflow. Or creating an issue can be another one, and a workflow can be many things as well. It can be a continuous integration build to verify that all of our code still builds, triggered after a push was done on a given branch. GitHub Actions live inside of the repository. They are code file and are thus part directly of the repository itself. That means that they can be versioned as well. Our actions need to be placed in a specific folder for GitHub to pick them up, namely the .github/workflows directory. We create them based on YAML syntax. YAML is a human‑readable data serialization language. It's often used for configuration files and in applications where data is being stored or transmitted. I will show you this in a minute. GitHub Actions, as the name gives away really, are a GitHub thing. They are not part of Git itself. The fact that they are so closely integrated into the GitHub platform is great since it means that we can now do everything in one place, the same place as we manage our sources or perform our pull requests. We can create our own actions, but there are a lot of templates available we can use directly or customize if we want. Many temples exist already, for example, for setting up continuous integration for several project types, meaning that we can be productive with it pretty quickly. Therefore, setting up a build for your project is very simple. And even in the free plan, you can use GitHub Actions. At the time of the creation of this course, the free plan comes with an unlimited amount of automation minutes for public repos and a 2,000‑minutes‑per‑month limit for nonpublic repos. These automation minutes are used to perform your continuous integration builds. If that isn't enough, you can still look at the paid options. You've heard me say quite a few terms already, actions, workflows, events. I think your head must be spinning right now. Let's take a look at how all these things relate then since, indeed, there are quite a few parts to GitHub Actions, and understanding what does what is, I think, important. So, using GitHub Actions, what we will be doing is creating a workflow. The workflow is where it all starts really. This is the automated procedure, which we will add to our repository. So it will be based on code. The workflow can be used to perform some continuous integration build, to run tests, to perform a deploy, and much more. It's made up out of one or more jobs. Now to kick off the workflow, something needs to happen around the repository within GitHub, and that is the event part. An event is an activity triggered within GitHub, and it can be a push to a certain branch, the creation of an issue, the creation of a pull request, and so on. Events can also be external and can be received through a webhook. So the workflow is what is triggered through the event. Now inside the workflow, we'll have, as said, one or more jobs. And a job will contain one or more steps. Execution of these steps will happen on a runner. And okay, what's a runner then? Well, as the name gives away, that's what will execute the jobs. So it's the server that runs the automation code. It will typically be a server hosted by GitHub, but it can also be your own. Now for the sake of simplicity, we'll use the ones provided by GitHub in this module. The steps within the job are the individual tasks that run commands. And a step can be an action or a shell command. Actions are the finest building block of a workflow, and they are standalone commands. Very often, we'll use actions created by the GitHub community, but we can also create our own. Actions will always be wrapped in a step for it to execute within a workflow. Now GitHub Actions, so the workflows, are created as code, more specifically in YAML. Now I won't be explaining the details of YAML syntax here. That is definitely out of the scope for this course. But I'll show you what you need to know, so no worries here. Let's take a look at the very simple first workflow written in YAML. This file is often used to explain the different steps, part of a GitHub Action. You may come across it in the docs as well. The first section is the name of the workflow you'll see this appear in the Actions tab on GitHub very soon. The on section defines what event is used to trigger this workflow. Here we use push, meaning that whenever a push is made in this repo, this workflow will be triggered. It's possible to limit this to certain branches, but here we do this across the entire repository. Then, under Jobs, we will find all jobs that are contained within this workflow. It's possible to have multiple jobs within a single workflow indeed. The check‑bats‑version here is the name of the job in this section. Next, we see a line that configures the job to run on Ubuntu, so a Linux server. And where is this machine then running? Under your desk? No. It's hosted by GitHub, so no need to set up something yourself, although you can. As we already saw, a job will contain one or more steps. All steps within this job are grouped under the steps section. Next, we'll have the list of one or more actions within this step. These actions are the smallest commands contained in a workflow. The uses checkout action v2 tells GitHub to use an action already created by the community, namely the checkout action. I want to use v2. What this does, it'll check out the repo, so it will get downloaded to the runner. So at this point, your code in your repo is now on a machine, and we can start doing things with it, such as running tests or performing builds. This is a very common command, so you'll see it appear very often since you'll often perform workflows against your code, and the first step will be getting your code on the runner. Next, we ask to set up Node, the package manager, onto the runner. Once Node is installed, we will get access to the npm command, which in turn will give us many more options. In the next line here, we are specifying to run a command on the runner. And in one go, we are using the npm command that became available by installing Node in the previous line. So in this line, we ask through npm to install the bats package, a package typically used for testing. And in the final line, we are then calling the bats command itself, and we use the ‑v to get the version of the package. So here you go. That is your first GitHub workflow. As said, we need to place this workflow YAML file in a specific location, the .github/workflows folder. Once you place it there, GitHub will pick it up. Under the Actions tab, you can now see that your workflow was found. And inside the file, all information to execute the flow was contained . So GitHub knows when it'll need to be triggered and what needs to be done when it's triggered. This we can all see in the interface. Our workflow is triggered whenever a push happens in the repository to get workflows, and the output can also be seen in the GitHub interface. We can indeed see log output once we have done a push and the workflow was triggered.

Demo: Creating a First Workflow

All right, time for the first demo. We'll take a look at the creation of the workflow inside the GitHub interface, and we'll then perform a push, and we'll see the result of triggering the workflow. So let's create our first action. Again, on the repo level, we can find the Actions tab. And when you have no actions in there yet, you'll see this getting started page. On this page, based on your repository, you will get some suggestions from GitHub to get started with GitHub Actions. It will basically look at what your repository contains, and, based on what it thinks it contains, it will give you a sample workflow and some other suggestions as you can see here. There's some suggestions around deploying the code, around continuous integration, and also about automating steps in the process. We'll take a look at this in the next demos. Let's get started building our first workflow. I'm not going to follow the suggestions that GitHub is giving me. I want to show you how you can set up a workflow yourself. So you can click here on set up a workflow yourself, and that will give me this page in which I'm going to be editing my YAML file that will contain my workflow. Notice here at the top that GitHub will automatically place this workflow file in a specific folder, and that is the .github/workflows directory. I'm doing this inside of the main branch. And so GitHub needs these workflows to be in this very location because, otherwise, it won't be able to find it. We can, of course, give our workflow a name. I'm going to call my workflow sample‑workflow. Now although I haven't asked for it, GitHub does give me a sample workflow. I'm going to remove this and include my own. I've pasted in now the first workflow. I'm not going to ask you to wait while I type this, but let me take you through it. So, first, we see the name, and I'm calling this sample‑workflow. It is going to be triggered, and that is defined here in the on when I do a push to my repository. And when making this very broad, every push within my repository will then trigger this workflow. We can, of course, also constrain this, for example saying that only a push on a certain branch will trigger this workflow. Then we have the jobs part, which is, of course, the most important part, and it will contain all the work that needs to happen. This line here is the name of my job, and I've also specified the runner. It needs to run on ubuntu‑latest, which will be the environment in which GitHub will execute my workflow. Then, we have the steps section, and the steps section contains individual commands or actions that need to be executed. And I have quite a few here. This is actually a simple workflow that is created by GitHub, so let's take a look at what this will do. The run here indicates that a command needs to be executed on the runner. In this case, I'm going to use the echo command, which is simply going to display some text. But notice in here that I'm also using some parameters. I'm going to ask which event on GitHub triggered the execution of this workflow. The event will, in this case, have been a push. I'm also going to ask for some other information about the workflow. I'm asking which operating system did it run an execute on. I'm also displaying which branch and which repository this was executed on. Then, we have another action that looks a bit different. It has a name and also has the uses keyword here. It is going to use an action, a predefined action, which is going to check out my code. That is one that you will come across very often. It's a predefined action provided to you by GitHub. But this will do. It will basically copy the code onto the runner, so the code that is currently in a repository, in this case on our main branch. And it will copy that to the runner so that the runner can then execute tasks on it. Then we do a couple more echo commands that will display that the repo has been cloned to the runner, and it is basically ready to test our code. We then do another command in which we are going to do an ls on the workspace, which will simply list all the files. And then we display the job status, again through a parameter. Now we have created the workflow. Let us now commit this. It is simply going to be a file, so we need to do a new commit. We can give it a name, and I'm going to directly commit this to the main branch. Now the funny thing is that because this is going to push to the main branch in my repository, my workload is automatically going to be triggered. So I'm going to commit a new file, so this new workflow file. And then when we click on Actions, we'll see that this page has now changed. Notice that it is now executing this workflow. At this point, it is yellow. Now it starts spinning, meaning that it is executing. And after some time, it will hopefully turn green. There we go. Now take a look at this page. It contains an overview of all the workflows. Indeed, our repository can and will have multiple workflows. I'll add a second one in just a minute. So these are the workflows. We can click on that workflow, and then it will show me all the runs that have executed for this workflow. So each workflow can execute multiple times. In this case, every time I do a push to my repository, this workflow will be triggered. We can then click on the workflow run to see its details. So here we see the summary of the run. It took about 14 seconds. That's actually not very long. And it did cost me 4 seconds of my billable time. Based on your GitHub subscription, you will get a number of free minutes. And if you go above that, you will have to pay. On the left here, you see the jobs included in this workflow. And so when I click on that, I will see all the steps logged by GitHub Actions that have executed in my workflow. So first, there was a setup step. We can take on that to see the details, and you can drill into that if you want. Then, we indeed see the execution of the run echo, and it also filled in that parameter. So this was triggered by a push event. And it's hosted on a Linux server. Again, you can expand these and see more information if you want. Here's that checkout repository code action that basically copied the code from the gillcleeren refactored‑telegram repository onto the runner in a different directory. On that code, we then called the ls command, which will then list all the files within our repository in this main branch, so in the current state. And although this works, well, actually did a good job, we are going to create another one that actually will do something more interesting. Let's go back to Actions and create a new workflow. We'll again arrive on this getting started page, but I'll skip that, and I'll set up another work for myself. I'm now going to use a workflow where I'm going to use NPM, the Node Package Manager. Once I install NPM, I can then use NPM commands to install other, for example, CLI tools. I'm going to do that here, for example, using the Netlify CLI. Netlify is a platform to build web applications, but that's not really important right now. What I'm going to do is I'm going to create a workflow that installs the Netlify CLI. Now you see the code for that workflow. I've given it a name, and I'm also doing it again on a push. Then we have the job named check‑netlify‑version. Indeed, I'm going to simply install the Netlify CLI and then ask for the version of that. So we again have the runner specified, and then I have a couple of steps. First, I'm going to check out my code, and then I'm going to use another built‑in action called setup‑node@v1. That is going to install Node, which brings with it the NPM Package Manager. Then, once I have NPM installed, I can then use that in a run action. So I just call npm install netlify‑cli. Once Netlify is installed, I have the ability to call the netlify command. So I then call npm netlify‑v, which will give me the version of Netlify that is installed. Now while this may not be really a useful thing, you actually see that you can do a lot more than just echoing text to the command. So let's commit this. And just as before, because we are doing this on a push, pushing this to the main branch will actually already trigger the execution of this workflow. So I'm going to go to Actions, and you'll see indeed that the Create netlify.yml push triggered two workflows. Let's take a look at this one, which is going to execute the install‑netlify‑package workflow. This will take a bit longer. So while it's running, we can see what it's doing. So it's already set up Node. Now it's installing netlify‑cli. So we do get some output, and then it executed netlify‑v to we see what is the current version of Netlify. And so here, under Actions, we see our workflows and the ones that have executed on our repository.

Automating Tasks

Now that we have our first workflow working, let's learn a bit more about which tasks we can automate using GitHub Actions. You've seen before that in order to execute an action, so a workflow, we need to link it with an event. When the event fires or triggers, the workflow or flows that are attached to it will execute. The first group of events that we can use are events within GitHub. You have seen before how we could attach to a push happening in the repo. That's a very common one, just like, for example, creating a pull request or creating an issue. You can also specify in the home, so where we specify the event, a list of events saying to GitHub that either a pull request or a push will trigger the flow. It's possible for each event to specify some configuration options, such as the branch on which the pull request was triggered. Next, it's also possible to define scheduled events, so time‑based events. In this case, we'll simply specify a UTC time on which we want to work for it to be triggered. The smallest interval between two negated workflows is 5 minutes, by the way. Finally, it's also possible to hook into webhook events from GitHub. GitHub indeed defines a webhook that is commonly used for integrations, and it specifies several events that will be triggered. On the most of these, we can also attach a workflow. Now events will trigger the workflow, and inside the workflow, we will, like we already saw, define what needs to happen. The smallest piece of functionality is an action. You saw a few actions in action already, such as the checkout action. We can create our own actions or use already created actions. The latter is what you'll do most often since a lot have already been created. We can browse the marketplace for existing actions when defining a flow. Workflows are triggered when the event fires, and then all its contained actions will execute. We have so far seen a workflow that shows a version number of an NPM package. And honestly, while it's interesting to know that, I don't think it is the type of automation we were hoping for. Am I right? In general, I see two large buckets of types of workflows. The first one is automating tasks within GitHub, such as labeling a newly created issue automatically, which I'll show you in just a second. The second group is everything around CI/CD, so continuous integration and continuous delivery. Using continuous integration, we will, as developers of code inside the repository, perform commits, and that we'll do often. The reason is that we are then bringing code together in a given branch. We are going to build things together, so our changes merged with the ones created and committed by others. By doing so, so by integrating all the time, so continuously, we can detect errors early, hopefully reducing the amount of time we need to spend later on on getting things to work again. Merge conflicts are typically also smaller, causing us to lose less time on merging different branches together. Setting up this CI process is often done using GitHub Actions. Next is the step to continuous deployment where we'll also deploy this merged good to an environment such as Azure App Services. That too can be automated using GitHub Actions.

Demo: Labelling Issues

In the next demo, we will create a workflow for an automation, namely adding a label automatically when a new issue is being created. Now GitHub Actions don't always have to do with code. We can use them to automate work within the repository. Say that we want to automatically label issues that are being created within our repository with a certain label, we can do that using actions. Let's try doing that. I'm going to go to Actions again, and I'm going to create again, like we did before, a new workflow. Again, we'll set it up ourselves. Thank you very much, GitHub. And I'm going to call this label‑issues since, indeed, this workflow will be used to automatically attach a label onto issues once they are created. That will then give us the ability to filter in our issue tracker newly created issues. So again, remove the default here, and let's bring in our workflow. I've given it a name as before, but now notice that we're not going to trigger this workflow on a push. I'm actually going to attach this onto issues being opened or reopened. Now what I then want to do is I want to label these issues with a given label. So in the Jobs section, I have a label‑issues job, which again will run on Ubuntu. And now in the steps, I'm going to use the andymckay labeler. Now it is actually an action that you can get from the GitHub Marketplace. So with it, if you look in the marketplace for andymckay, you will indeed find that simple issue labeler, which is that particular action that we're now using. I'm going to parameterize it a little bit so that the label that we're going to use is called triage. So let's commit this to our repo again. Here, we now see that our workflow file was saved in the .github folder. And if we now go to Issues and we create a new issue, let's create a blank issue like we did before, and let's call it label tester. We'll submit a new issue. We are not attaching a label ourselves. There we go. If we now go to Actions, we see that the label tester run is scheduled. There we go. It's now running. And it should be pretty quick. There we go. We now see that it executed successfully. Let's go to the issue tracker. And indeed, the label tester has now been labeled with triage. So only after the execution of the workflow did it get that label. So this shows you how you can use GitHub Actions to automate tasks in your project management.

Demo: Setting up Continuous Integration

In the last section of this module, we'll see how it can set up continuous integration using GitHub Actions. Setting up CI or CD can be quite some work, namely to get the YAML right. But no fear as GitHub actions comes with a lot of templates out of the box. It is so common to use a template, even that, in the interface, a section to show you the common templates around your repository's code is included. Instead of talking about this, let's return to GitHub and learn how we can use a template to set up continuous integration. The repository that we have been using throughout this course with its HTML and CSS code might not be a great fit to show you how this works. So for the sake of this demo, I'm using a different .NET Core‑based repository. Now for the purpose of this demo, I'm going to use a very simple .NET Core application. And in case you're not familiar with .NET Core, let me give you a very small sample of what it will do. It is a console application that will execute this Program.cs, which will simply write to the console Hello World. We're not here to learn about .NET Core. We're here to learn about how it can use GitHub Actions to set up CI, so continuous integration builds. Again, this will be triggered using an action. So let's go to Actions again, and this time we'll actually follow GitHub's suggestions. As you can see, it has detected that this is a .NET Core repository. So I'm going to follow its suggestion and set up this workflow. So I'm going to click here on Set up this work flew to create a CI, a continuous integration, build. It's just that I call this workflow dotnet.yml. We can, of course, give it a different name. But let's now take a look at what the sample code contains. This workflow will be triggered on pushes to the main branch, as well as pull requests on the main branch being created. Then, I will execute the CI steps, the continuous integration steps that is. .NET has a CLI, which is accessible through the .NET Driver called dotnet. So dotnet is basically the command you need to execute for all of this. Now under the steps, you will see again that we are doing a checkout, and then we call another built‑in action called Setup .NET, and this is using .NET 5, which is the latest version at the time of this recording, so I'm going to set up .NET version 5. This will install .NET Core on my runner. I'm first going to call dotnet restore, which will restore the dependencies, and then I use a dotnet build, passing in that it doesn't have to do a separate restore anymore. Once that is done, if there are unit tests, we will also execute these tests using the dotnet test command. So basically, we are doing a build of our project continuously when a push on the main branch or a pull request on the main branch is happening. So let's commit this. And again, because this is a commit on the main branch, this workflow will be triggered. Let's take a look at its jobs. First, it's checking out the code. Then, it is setting up .NET. And it's building our code. And it almost seems to have run successfully, so it did actually do a successful build of our project. So we are sure that this commit hasn't broken the build. That's what continues integration is all about.

Summary

And we've reached the end of this module, and now you have had your first encounter with GitHub Actions. I hope you liked what you saw. Actions are a great way to automate many tasks within GitHub. Very often, you'll see actions being used to set up CI, so continuous integration, and CD, continuous deployment. In the next module, we will learn about the ability to create a wiki for our repository. I hope to see you there as well.

Creating a GitHub Wiki

Module Introduction

Hello again, and welcome to this module where we are going to create a wiki on GitHub for our repository. A wiki is a knowledge‑based site where people can collaborate on documentation and other content. I'm Gill Cleeren, and I will be guiding you also through this module. I assume most, if not all of you, are familiar with the concept of a wiki, but maybe you didn't know that GitHub helps you in creating one for your repository. Let me explain you all about that in this module. The agenda for this module is pretty straightforward, as you can see here. I'm going to take you through the steps to set up a wiki for the repository.

Creating a Wiki

Let's dive straight in and see how we can create a wiki for the repository we have been working with in this course. A project without good documentation is... a regular project? While it's often the least interesting part, good documentation on things such as coding guidelines or best practices are valuable for each project. GitHub comes with the ability built in to add a wiki to a repository, giving you the central location to manage all information about the project. Wikis in GitHub are linked to a repository, thus describing the project. The goal, of course, as already mentioned, is giving developers and other team members a central location to store project documentation. It can contain information for end users of the project, such as how to use the code, why certain things work the way they are created, and much more. You can choose, of course, what you put in there. The wiki can contain many pages and can be edited online directly from GitHub, as we'll see soon, or offline, and then the changes can be pushed, as always. In that view, it's different from the README file, which is also a form of documentation for the project. The README will typically contain an overview or a getting started while the wiki can be used to provide more complete information about the project. Of course, you can reference the wiki from within the README. And before I forget, the way you perform the markup and layout of the content in the wiki is, again, what we've seen before already. It's done using markdown syntax, and we'll see that in the demo.

Demo: Creating a Wiki on GitHub

In the first demo of this module, we'll go back to our repository, and we'll set up a wiki for documentation of the project. We'll add a page, and we'll add some other content, such as images. We have been working on our repository quiet a lot, and it's actually in good shape. But what if people want some information on how to work with our code? And we can use a wiki for that, and a wiki can indeed be used to put all kinds of information, even the landing page says it here. We can put documentation there, how to get started, installation instructions, roadmaps, you name it, you can all put it in a wiki. So, we can click here on the wiki button, and then we can start creating our wiki. And every novel has started with the first page, so our wiki will too. So let's click here on Create the first page. Let's call that our Home page, and we are going to be bringing in some content here. As always, this is Markdown that we'll use, so we can use the one hash and the two hashes to indicate H1s, H2s, and so on. Let's preview this. This looks quite good. And Edit message is Initial Home page because this also goes into the repository. I'll come to that later. Let's click on Save Page, and now we have our home page, our landing page, for our wiki. So this is the home page of our wiki. GitHub is saying that we can add a custom footer and a custom sidebar. Let's start with the footer content, which will be content that is placed on each page. It has a special underscore footer name. And because of this footer content, I'm going to use this to track when this page was updated. Click on Save Page. There we go. And now we see this footer showing here. Let's create another page because, of course, we're not limited to one page. We can add as many pages as we want. Let's click here on New Page, and let's call this the Getting started page. And, of course, we'll bring in some content here as well. Save the page. Now we have two pages at this point, and notice that in the list here, GitHub is showing me the pages that we currently have. But we may want a hierarchy of the pages in our wiki. We can, for example, do that in the sidebar. Let's click on Add a custom sidebar, and here we'll bring in that hierarchy I just mentioned. As the first item, I'll want a link to the Home page of the wiki. This points to the text, and this points to the link. Let me show you that. Let's add a page here, and we see that we have Home page, and that goes to the wiki home. Let's add another list item here and say that this will be Quick starts, for example, and we'll bring in another page, Getting started, and that links to Getting started. And we can bring in another page that points to Samples. I don't have that page yet, but you can imagine what that will look like. Now before I save, I need to make sure that this contains a dash because that's the link to the page Getting started that we already created. Let's take a look at the preview, and that seems to be okay. So this now goes to Getting started. Let's save the page, and here's the result. We now have our sidebar, which has a link to Home and also a link to Getting started. And that's our Getting started page. Now, of course, this page looks a bit dry at this point. Let's add in an image. Let's edit the current page. Let's bring in a logo, an image, the Pluralsight logo that is. Let's save the page, and now we have the Pluralsight logo showing at the top. So bringing in images and other rich content can be done in these wiki pages as well. Now if you do not like editing this directly on GitHub, you can, in fact, also work with this locally. Your wiki becomes a repository, and you can just clone that as before. You may have noticed already that it says here that we can clone this wiki locally. Just like before, I can clone this. And if we now go to Git Bash again and we do git clone and we paste in that copied link, you see that it got the name refactored‑telegram.wiki. It will now simply clone the files in there. Let's take a look. So here's that wiki folder. And indeed, all the pages we created so far are now available there as Markdown files, and we can simply open them using, for example, Visual Studio Code. And we can then edit them locally, push them up, as we would do with a regular repository.

Summary

Let's quickly summarize what we have learned in this module to wrap it up. This module was entirely devoted to working with a wiki, which is a great source of documentation for the project. As you saw, the content for the wiki is managed entirely as source files in a repository. In the next and final module of this course, we are going to work with some more social features of the GitHub platform.

Working with Social Features

Module Introduction

Hello, and welcome to what is already the last module of the GitHub: Getting Started course here, on Pluralsight. I'm still Gill Cleeren. For the last module, I thought it would be a good idea to keep my name unchanged. Anyway, in this last module, we are going to explore some more social coding features that come with GitHub. Let's kick off the module by taking a look at what I still have planned for you in this last module. We'll start by taking a look at gists. Although it's a small feature, it's definitely worth taking a look at. In the next topic, we'll take a look at GitHub Projects. And finally, we'll explore GetHub Pages.

Demo: Creating a Gist

So let's kick off the module by taking a look at gists in GitHub. So what is a gist really? A gist is, aside from GitHub, meant to allow users to share snippets and notes. If you've ever worked with Pastebin, well, it's very closely related to that. Using gists, as mentioned, you can share files or snippets, so a part of a file. Use it to remember something for yourself or share with your colleagues or even post to your blog. While it's aimed at sharing files, you can combine multiple files into one single gist. Behind the scenes, gists use the same infrastructure as GitHub itself, meaning that everything here is still a repository. When working with gists, there are basically two types, public and private ones. As the name implies, public gists are really public. They're searchable so everyone can find them pretty easily. The private ones, on the other hand, aren't really private. It's basically more something that you don't want to be discovered or aren't ready to share with everyone yet. However, the link itself isn't private. While private gists don't show up in a search, for example, if you have the URL, you can still access it. So it's definitely not good to put something secret in there. Finally, gists are also downloadable in a zip file. All right, time for a demo. Let's take a look at creating a gist. Let us now create a gist. To create a gist, you go to gist.github.com. And there, you can instantly start creating gists to share code and snippets. Indeed, gists are typically used to share small amounts of code that you may want to embed in a blog post or something. You typically won't be including entire projects in gists. Let's create our first gist. I'm going to create a gist, and I'm going to give it a name. I'm going to call that the ToObservableCollection gist. It's a snippet I often use in my work. Now, GitHub also asks me the file name including the extension before I actually put in my snippet. Now, a good name would be CollectionExtensions since there are C# collection extensions. So I'll call this CollectionExtensions.cs. .cs is useful because then you also get some code highlighting here. And then next, I'll put in the code I'll want to be part of this gist. And like I said, because we used C# as the file extension, we also get some code highlighting here. We can add multiple files as part of one gist as well. Next, we get to select it is gist should be secret or not. Now a secret is not really secret. It means that it's not findable through a search. But if you give the link to someone, that link is still public. A public gist will also be searched through the search engine. So let's, for now, create a secret gist. Click on Create gist, and then our gist will get a unique link, and that I can now use to send this gist to someone. Of course, we can also go back to the gist and start editing it because I do notice that my formatting is not right here, so I'll change this here a little bit. There we go. Let's leave it like this for now. Update the gist, and now it's been updated. We have two revisions that we have on this gist. We can see here the typical GitHub interface that shows me a list of revisions. This was the second revision, and this was the initial revision that was when I added my code in my gist. Of course, again, like everything in GitHub, we can add a comment onto that gist. And it's, of course, again, Markdown. Like I said, we can also embed this gist. If we copy this link here, you'll get an embed link that you can then use to embed this gist, so the code in my gist in, for example, your blog posts. You can also download the zip of the gist that will contain the file name, so CollectionExtensions.cs that contains our code. And again, like pretty much everything in GitHub, you can also clone it because, indeed, a gist is also a repository. So if I click here on the Clone via SSH link, and I go to the CLI, and I do a git clone, a repository will be created. Indeed, here we now see our gist as a local repository with a .git folder and that CollectionExtensions file as well.

Demo: Working with GitHub Projects

In the next topic, we'll take a look at projects on GitHub. When someone talks about managing a project on GitHub, they are probably referring to a project board. As the name implies, a project board can be used to give you an at‑a‑glance overview of the status of the project's work. The main idea behind the board is, of course, giving you an overview to help with the organization and prioritization of the work on the project. You can choose to create a specific board to manage the work and the related issues for a certain feature. A board can also be used to manage a checklist or even a roadmap. In the next demo, I'll create a project board using one of the built‑in templates. We'll use some of the issues we have already created and manage them through our board. I may be sounding repetitive now, but I hope you understand that GitHub is much more than just source code management. It can also be used to track a project, and we've already looked at projects briefly earlier in this course. They too are linked to the repository, and you can actually create multiple projects per repo. And we can use these to organize work, as well as issues and therefore use it as a project management system. So let's go to Projects and then click on Create a project. We'll start by creating a new board, which is going to be the Website release board. This could be the board that we'll use to track all the issues and bugs and pull requests related to getting that website released. We can also, of course, add a description, and you can also use a template. There are a couple of templates which come by default with GitHub. You can use basic kanban boards, an automated kanban board, and so on. We'll use for now no template. We'll create the board ourselves. So finally, click on Create project to generate the project. So here, we have the project we just created, but we can't do anything with it, says GitHub. We don't have any columns nor cards yet. So what I'm going to do is I'm going to create a couple of columns to start organizing my work. And this, by the way, would have been done automatically if I would have chosen to use a template. That would already have created a couple of columns for me. But let's do it ourselves. It's pretty easy. Let's add the first column, Not started. That's going to contain the work that effectively hasn't started yet. Next, we are asked if you want to use some presets for automation that will basically automate some work. For example, if issues get created, they will automatically appear in this column. We'll come to that very soon. Notice here we have a couple of presets that we can use. Let's click on Create column. I'll do the same for two more columns. I'll use a Doing column in which developers can drag items that they're working on. And finally, we'll create another column, which I'll name Done, and that will contain work that is finished. I think I have enough lanes for now. I have enough columns, let's say, for now. Now I can click on Add cards here. This will show me a number of open issues, and I can also, again, go to the issue search to get more issues available here. By default, it gives me the open issues. I can now drag all of these into the Not started column. There we go. These are the useful ones, let's say. Let's close this now. So these are the work items, the issues which are still open before we can go ahead with the release, so before we reached the milestone, let's say. So now, for project management, I can drag this item into the Doing column. Once that is done, I can actually then drag it over here to the Done column. Now we can use the project entirely manual as you can see here, but I can also bring in some automation. I can click on the ellipsis here and then click on Manage automation. I can now use one of the presets to automate a couple of things around the flow for this project. For example, when I used it To do preset, I can then say that when issues or pull requests are being created as newly added issues or pull requests, they will automatically appear in this column. Let's click on Update automation, and let's now go and create a new issue. So I create a new issue. I'll use the blank issue. I'll call it Background image on contact page. I do need to associate it with the project, so that's going to be the Website release. There we go. And now, GitHub knows that this is effectively associated with that project. So when I now click on Submit new issue and I go back to the project, you'll see that the Background image on contact page is now automatically in this lane, so in this column, as a not started issue. And the same goes for pull requests, but I suggest you try that out yourself.

Demo: Publishing Information Using GitHub Pages

Sharing knowledge is one of the best things we all can do. While repositories on GitHub contain a wealth of knowledge, it's not always the easiest way to share that knowledge. GitHub offers other ways to share information. One of them is the GitHub Pages. In the final section of this module, we'll take a brief look at Pages, and we'll see how we can easily set it up. Before we look at how we can set them GitHub Pages, it's a good idea to start by looking at what they are exactly. Basically, it's a site‑hosting service where you can host just any website. It could be a site that's related to a repository, but it could also be your personal blog or even a company website. One of the main points of interest here though is that it's made to host static pages. You can't run any server‑side code. All pages will need to be generated and then placed online. Underneath the covers, your site is being hosted from a repository. Pages are not an exception to everything on GitHub being hosted in a repository. Creating the pages site can be done online or offline using the command line again. Yes, it's just a repository. I think it's best that we take a look at GitHub Pages in a demo instead of talking about it. I'll give you an introduction demo. Note that there's actually much more that can be done with Pages. It could, in fact, be a course in itself. Now our project lends itself very well to being hosted directly on GitHub. It is just static content. It's static HTML, JavaScript, and CSS. So instead of hosting it, well, somewhere else, we can put it, we can host it, let's say, directly on GitHub Pages. If you want to do so, go to Settings and then go to Pages. It is, by default, disabled, so we can enable it by clicking here on this drop‑down here, and we are asked to select which branch we basically want to push to GitHub Pages, which we want to host, let's say, on GitHub Pages. I'll use the main branch here. Next, I get the question which folder do I want to use. In our case, that would also be the root, but it can also be a different folder. With that selected, let's click on Save. And now our site is available through gillcleeren.github.io/ the repository name. So it will be your username.github.io/ the repository name. So let's open this, and there we go. Our site is now automatically hosted, and it is, indeed, the main branch with the three exclamation marks that we're seeing here. If we go ahead and make a change in the main branch directly, let's go to the index again and edit it once more. Let's include fantastic company in the header. Click on Commit changes. We push this automatically to main. GitHub Pages is reading from main. So if we refresh this, after a while, it will show your new content.

Summary and Course Closing

The main message of this module was that GitHub stimulates social coding, and I hope I've given you a good overview of how GitHub achieves this. Let's recap what we have seen. Through gists, we can share small amounts of code. Projects enable distributed teams of coders to manage the work within a project. And through GitHub Pages, we can use a repository as a way to share static content on a site. It's time to say goodbye since we've reached the end of this course. The only thing left for me is congratulating you on finishing this course. I wish you a lot of success in using GitHub, and thanks for watching this course.