Getting Started with GIT

Course Overview

Understanding the fundamentals of Git, the popular version control system, can make your life so much easier if you're working on any type of file-based project that needs version control. Learning a new subject, especially a technical one, can be a difficult and stressful experience, but knowing where to start can make all of the difference. My name is Aaron Stewart, and my latest course is Getting Started with Git. Now I've been a Git advocate and GitHub enthusiast for several years now, but I can still remember the challenges I faced when learning Git for the first time. I wanted to learn from content that was fun, relatable, and interactive. I found great content in several places, but I never found one great resource that had all three. This course will teach you the fundamentals of Git by taking you on a journey with Steve, a local coffeeshop owner, where you'll take a hands-on approach to learn Git while working on his coffeeshop recipe project. You'll learn why Git is such a popular version control system choice, not just for software developers, but for project managers, team members, students or really anyone who wants the flexibility to keep track of changes to any type of file-based project. You'll also learn what GitHub is, and how it can transform your Git project to the next level. And the great thing about Git is that you don't even have to be a computer geek to use it. After this course, you'll harness the power of Git, and be well on your way to becoming a Git master.

What Is Git and How Does It Work?

Introduction

Hi, and welcome to Getting Started with Git. Since you're here, I think it's safe to assume that at some point you've heard about Git. Maybe some project at work is using Git, and you're not quite sure how it works, or even if it's worth the time to understand. Maybe you're taking a class and your

professor wants you to use Git to keep track of source code files, and you feel like you don't quite understand the concept yet or even how to get started with using it. Perhaps you've attended a conference where the speaker talked about the wonders and the powers of Git, or you've read some article where it presented data on how Git increased efficiency on various projects and you want to learn more. Whatever the situation you find yourself in, this course is a great place to start with understanding the basics of Git, and how you can start using it in your projects today. The great thing about Git is it doesn't take a computer geek to understand the basic concepts and quickly see how it can be useful to any type of software project. By the end of this course, you'll understand why Git is such a powerful and popular tool among software developers, project managers, team members, students, or really anyone who wants the flexibility to keep track of changes to any type of plain text file project. And that's the great thing about Git, it doesn't just apply to source code files. In this course, you'll also learn how Git, combined with the popular web hosting service of GitHub, can transform your Git project to the next level by maximizing collaboration within a project among multiple contributors. Git and GitHub are often talked about in unison, so it's common for newcomers to think that they're the same thing. But for now, just know that they are separate technologies, but they complement each other seamlessly. So before we get ahead of ourselves, let's start with the first basic question, what is Git?

What Is Git?

Git is a version control system, also known as a VCS. A VCS is basically some software designed to record changes made to a file, or a set of files, over time. This gives you the ability to revert or recall changes to one file, or a set of files, after you've made them. If you're working on a project that contains a lot of files, version control gives you the ability to control that entire project, and you can revert only one file to its previous version, multiple files to their previous versions, or the entire project back to its previous version. You can compare changes made to files from one version to another, and you can see exactly what was changed in each file, when a change was introduced, or who introduced the change. You can even see why a certain change was made. The great thing about a VCS is that you can version control nearly any type of file on a computer, not just source code files. Students often use version control to keep track of different versions of an essay or a research paper. A designer can use version

control to keep versions of an image without worrying about modifying the only version of that one image, and version control gives you the ability to explore new ideas with your project, with the assurance that you can always revert back to a previous version no matter how far you've altered the content. If you haven't noticed by now, version control is a pretty powerful tool that gives you complete dominance over the life of your project. And if you think about it, version control is the ultimate time travel machine for your project, and you're in control. But Git is not the only version control system out there, in fact, Git is just a type of version control system. So to better understand how Git works, and why GitHub decided to build its product on top of Git, we need to understand the main different forms of version control. Before version control systems, programmers often copied files into a new directory for each version change of their files. Well, this approach is common due to its simplicity, but it's incredibly error-prone. Forgetting which directory you're in and possibly saving changes in the wrong directory, would completely overwrite those files that you didn't intend to. So to fix this problem, programmers created a local version control system.

Local Version Control Systems

A local version control system is a database stored locally on your computer, and each time changes are made to a file, those changes are stored as version patches in that database. Each version patch only contains the changes made from that previous patch. So to see what the file looks like at any point in time, it needs to add up all the patches previous to that time to recreate that file. Now, this was a major improvement to saving copies to a new directory, but it still has its flaws. If anything happened to that local database, those files would be lost. If one of those patches became corrupt, the changes introduced in that patch would also be lost. And since this is a local database, all changes made to these files are being made and stored locally on that computer, and because of this, collaborating with other developers or team members is a bit of a problem. So to deal with these issues, and help address the collaboration bottleneck, centralized version control systems were developed.

Centralized Version Control Systems

Central version control systems have a single server that contain all of the version files. This makes it possible for multiple clients to check out files

from that central server at the same time, and since every client is pulling and pushing changes to a central server, everyone generally knows what everyone else is working on in that project. This also introduced authorization rights for administrators to control who is working on what, instead of having to deal with local databases with each individual client. In other words, centralized version control systems keep the version history of changes on one central server that all collaborators request the latest version of that project and then push their changes back up to a centralized server. And because of this, everyone collaborating on the project server also shared everyone's work. Now this was a big upgrade to local version control systems, but again, it still has some flaws. The biggest problems was that everything was being stored on one central server. Now if that server happens to go down, nobody can save changes, pull files or collaborate on anything that they're working. Similar to a local version control system, if that central database becomes corrupt, and no backups have been kept, everything is lost for that project except for those single file snapshots that each collaborator has on their local client computer that they're working on at that time. So, to solve this problem, another type of version control system was created called distributed version control systems.

Distributed Version Control Systems

With a distributed version control system, each client doesn't just pull down the latest file snapshot, they fully mirror the project by pulling down the entire project so everyone has a local copy of that entire project's history. Every time a client pulls down that repository or project to their local client computer, they essentially are doing a full backup of all of the data each time they check out that project. Now with this model, if the server dies or becomes corrupt, any of the client projects collaborating with that server can be copied back up to the server to restore it. A distributed version control system allows for collaboration with many clients simultaneously within the same project. This allows for many different project workflows that a local or centralized version control system don't support. Because of this, and the advantages that a distributed version control system brings, this is the version control system of choice for most projects, and it's the type of version control system that Git is. So now that we know that Git is a distributed version control system, and why software teams should use it in

their projects, I think it's important to know why Git was created in the first place.

Short History of Git

In 1991, Linus Torvalds created the Linux kernel as an open source software project where changes to the software were passed as patches and archived files. Now this was done for a little over a decade, until in 2002 the Linux project began using a proprietary distributed version control system called BitKeeper. Now this worked out well, as it brought the advantages of a distributed version control system to the Linux kernel, and at the time, it allowed open source developers to use the basic version for free. But since BitKeeper is a proprietary software, many of the Linux developers were against the idea of using a proprietary software for contributing to Linux's open source kernel. And in 2005, BitKeeper decided to revoke its free-of-charge status for using the tool. Not wanting to revert to their previous model of using patches, this change prompted Linus Torvalds and the Linux development community to develop their own tool based on the advantages of BitKeeper. Now, they wanted a tool that would be fast, have a simple and easy-to-use design, have strong support for non-linear development, be fully distributed, and be able to handle large projects with efficiency. The tool that was created was called Git. Now, since its birth in 2005, Git has retained these initial qualities as it has evolved and matured to be the popular distributed version control system that it has today, that so many developers, team members, project managers, and project collaborators choose for their projects. Now that we're familiar with the different types of version control systems, and why Git was created, let's now look a little closer and understand exactly how Git works.

Git Uses Snapshots, Not Patch Differences

The first thing to understand about Git is the way that it stores data for your project. Other version control systems store data as a list of file-based changes. With this model, each version of that project stores data as changes made to that base version of each file. As a version control system, this model works because it is keeping track of changes made to files. But as you can see, each version only shows the changes made to those files in that version. Now this makes it difficult to see the status of the entire project, especially the files that didn't have a change made in that version. Git, on the other hand, stores and thinks about the project's data a

little differently. Git thinks of its data as a series of snapshots taken of the entire project's file system. Every time you commit or save the state of your project, Git takes a picture, or a snapshot, of what your project looks like, including all of its files at that specific moment in time, and then stores a reference to that snapshot. You can recall any snapshot previously taken at any point and see the state of that entire project, not just the files being changed in that version. If files have not changed for that snapshot, Git is efficient enough to not store that file again, but it links to that previous identical file that it stored during the last snapshot that that file had changed. By understanding that Git stores snapshots of your project's data similar to a file system, and not just version differences, you can start to realize that Git is much more than just a version control system that simply keeps track of versions to your project.

Almost Everything in Git Is Local

Another main advantage to Git is its ability to function locally. Now since Git is a distributed version control system, you can pull down that project's repository to your local computer, and when you pull down that project locally, you have the complete project and all of its history, not just the files you're currently working on. Now because of this, Git doesn't need to pull the history from the server each time you want to work on a file, it just reads it from your local database, so this makes most local operations seem almost instantaneous. For example, if you want to see the changes made to the project several weeks, or even several months ago, Git simply checks your local database to run the comparison instead of asking for that information from the remote server. This means that there is very little that you can't do with Git when working offline. Imagine working from home when your VPN fails to connect, or you need to work on a plane or a train where you don't have access to the origin server. This would make life pretty difficult with many other version control systems. However, with Git, you simply make the changes to your local repository of that project, as normal, and then you simply commit your changes to the remote server when you're connected back to the network.

Git Has Integrity

A very important part of Git is how changes are saved to your project. Everything in Git is checksummed before it is stored in the database, and it is then referred to by that same checksum. Now if you've messed around with Git or are familiar with other version control systems, you'll notice that

these checksums are not simple version references such as V0123, V0124, and so on, but are instead composed of 40-character long hashes of hexadecimal characters, 0-9 and A-F. The reason for such a long checksum reference for each commit, well, to put it simply, Git has integrity. This means that it's impossible to modify the contents of a file without Git knowing about it. If two people look at the same checksum reference on their local computers, they can be 100% sure that they're looking at exactly the same files. Now this is especially important in a distributed version control system. You can't lose information in transit or have corrupted files without Git knowing about it. The mechanism that Git uses to generate these 40-long revision check-sums is called a SHA1 hash. A SHA1 hash is an algorithm that takes some data as input and generates a unique 40-character string from it. It's unique because no other input of data should ever produce the same 40-character hash. However, this same input of data should always produce exactly the same 40-character hash. Now, you'll get used to seeing these hash values in Git because it uses them quite often. Git stores everything in its database not by filename, but by the hash value of its contents.

Steve and the Coffee Shop Analogy

Now that we understand some of the main benefits to Git, and how it manages data, hopefully you can see that managing changes made to a file or files is Git's main purposes, and it does it extremely well. Understanding exactly how Git manages these changes is one of the most, if not the most important thing to remember about Git. Everything Git does is based upon how it manages data. But before we get started learning the inner workings of Git, I found that it's much easier learning a new technical subject by using an analogy to make these concepts more relatable. So, for the rest of this course, we'll use an analogy of a coffeeshop to help us understand a little more about Git. Our story starts with Steve, and Steve is the single owner of a well-known coffeeshop downtown called Wired Brain Coffee. Steve manages all operations, and has even developed most of the shop's recipes himself. In order for Steve to keep the popularity of his shop, he constantly tries out new ideas by altering existing recipes and changing out certain techniques and ingredients that are used. He wants to keep the existing recipes, but also keep track of all of his changes in case he decides to make a change to the new menu item of the month. Currently, Steve stores each of these recipe files on his computer, and when he wants

to make a change, he alters the actual file and resaves it with a different recipe name. His approach to saving files is common and simple, but as we learned earlier, it's incredibly error prone, as Steve has made several mistakes and has accidentally resaved over files he didn't intend to. Another issue Steve has encountered is the accessibility of these files. Steve occasionally works on his home computer, as well as other devices when not at his work computer, and this has made it difficult to keep everything up-to-date on the changes Steve is making across these multiple devices. And Steve's whole approach is a very manual and time-consuming process, and so Steve is looking for a better way to manage his files that adds both clarity and simplicity to this process. So, with this analogy in mind, let's look closer at Git and see how Steve can harness its power to improve the way he manages his files within his Wired Brain Coffee project.

The Three Stages of a File

Git has three main stages a file can occupy, Committed, Modified, and Staged. A file that resides in the committed stage, means that the data in this file is safely stored in Steve's local database with that project. When Steve makes changes to his recipe file, the status of the file moves from committed, or unmodified, to modified. This modified state means that the file has been altered from the last committed version that is stored in the local database. Basically, this state just means that Steve has introduced new changes to the recipe file but hasn't committed them yet. You can think of this state as a work in progress where more changes can be made to that file. Now when Steve has finished the changes he wants to make to a file, the file then moves to the staged state. This means that Steve wants to take the changes made to that file during the modified state, and then commit them to his local database to be in his next commit snapshot. Now it's important to note that these three stages are applied to files in his project that are being tracked by Git. In other words, these stages only apply to files that were in the project's last commit snapshot. A file can also be untracked. Untracked files are any files added to the project since the last snapshot that Git hasn't been tracking. To start tracking an untracked file, Steve can simply add it to his Git project and then Git places that file in the staged state, and it will be added into the next commit snapshot.

The Three Main Parts of Git

Similar to how a file can reside in three different states, Steve's Git project is made up of three main parts. We start with the .git directory, also known as a repository. This is the origin of Steve's projects data, and is what is pulled down from the remote server, which in Steve's case is currently being stored on his local computer. This is where Git stores the metadata and object database for his project. Next we have the working directory. This is a single copy, also known as a checkout, of one version of the project. These files are pulled from Steve's local compressed database in the .git directory and placed in the working directory that he can then modify. The staging area is also known as the index. This is an area that sits between his working directory and the .git directory. Steve can use the staging area to build up a change or set of changes that he wants to commit by taking a snapshot of his project with those changes in place. When he creates a commit, what is committed is only what is currently in the staging area, not what is in his working directory. This gives Steve the freedom to modify several files, but only commit the changes of a file or files that he wants to. The uncommitted changes remain in his working directory until Steve decides to commit them or stash them away for future use. After Steve commits his changes, he then pushes his commits up to the .git directory to update the origin data of his project. So, in short, if a file version is in the .git directory, it's committed and recorded in that project's repository. If a file is modified and has been added to the staging area, it's staged. And if a file was modified since the last checkout of that file from the .git directory, but has not been added to the staging area, it's modified.

Summary

So far, we've learned about the different types of version control systems, and unlike a local or central system, Git is a distributed version control system that allows for collaboration with many clients simultaneously within the same project. With Git, each client doesn't just pull down the latest file snapshot, they fully mirror the project by pulling down the entire repository, essentially doing a full backup of all the data each time they check out. We then covered a short history of Git, and the events that prompted Linus Torvalds and the Linux development community to develop their own tool based on the advantages of BitKeeper, that would be fast, have a simple and easy-to-use design, strong support for non-linear development, be fully distributed, handle large projects with efficiency, and most of all, be free to use. We also learned that Git uses snapshots, and not patched differences,

to manage its data. Every time you commit or save the state of your project, Git takes a snapshot of what your project looks like, including all of its files at that moment in time and stores a reference to that snapshot. We learned that almost everything in Git is local. Git doesn't need to pull the history from the server each time you want to work on a file, it just reads it from your local database, which makes most local operations seem almost instantaneous. If you need to work offline, no problem. Make the changes to your local repository as normal, and then simply push your changes to the remote server when connected to the network. We then covered the integrity of Git, and how everything is checksummed before it is stored in the database. We learned that these 40-character long checksums are generated by using a SHA1 hash to produce a unique checksum identifier for the contents of that snapshot. Basically, no other input of data should ever produce the same 40-character hash, but the same input of data should always produce exactly the same 40-character hash. We learned the stages of a file and how it can be committed to your repository, altered in the modified stage, and then staged to be committed back to your repository. And lastly, we learned the three main parts of a Git project, and the basic workflow of how the .git directory, or repository, interacts with the working directory and the staging area. So, at this point, you may be wondering exactly how Git performs these functions. What exactly are the specific Git commands used to move these files from stage to stage, or how to actually get started using Git. But trust me, I promise we'll cover that in just a bit. There is a lot more detail surrounding these subjects, but really understanding the basic workings of Git will make your learning much easier. Now that we've got some of the basics of Git out of the way, it's time to build upon what we've learned and continue our story with Steve's coffeeshop, and see how he harnesses the power of Git while using the popular web hosting service of GitHub.

Get up and Running with Git and GitHub

Introduction

Hi, now that we have a good understanding on what Git is, and how it adds version control to a file project, this module will cover what you need to know to get up and running with Git so you can start using it in your project today. If you recall from the first module, Steve, who is the owner of Wired Brain Coffee, is looking for a better way to manage his recipe files instead of copying the existing files and saving them under different file names to keep track of his changes. Since Steve has read about the powers of Git, he's excited to finally start managing his files efficiently by adding Git to his existing project. So, let's continue Steve's story and see how he uses Git to manage his project moving forward. First, we're going to cover the process of installing Git on Linux, Windows, and Mac systems, and learn how to configure Git regardless of which system you're using. We'll then learn how to initialize or start a new Git project, and then be introduced to GitHub and learn how to start using it by first creating a GitHub account to host our Git project. We'll then learn how to push our local project to our GitHub account so we can then have access to our project on our multiple devices. So let's get started.

Using the Command Line

Before we install Git, we need to become somewhat familiar with the command line. Now, if you've never used the command line before, this can seem a little daunting, but here we'll cover a few simple commands that will give you the confidence to begin your next Git project, as well as start you on a path to continued learning. As the name implies, the command line is used to run commands by typing specific commands and then hitting Enter to execute that command. It's important to note that many of these commands are dependent on your current directory location, so you'll want to make sure that you're in your projects directory before running your commands. Now for Windows users, you can open up the command prompt or PowerShell, and for Linux or Mac users, which is what Steve will be using in this course, you can open up Terminal, which comes pre-installed with every Mac OS X system, and can be found in the Applications folder inside of the Utilities subfolder. Throughout this course, we'll be using a few Terminal commands to perform some actions. There are some slight differences with these commands depending on what system you're using, but the result will be the same. The first command to know is the pwd command. Pwd, which stands for print working directory, just returns your current folder path location. This is a simple, yet useful

command that simply lets you know where you are at currently inside Terminal. Now that we know where we are, you can use another command to change your location. The cd command, which stands for change directory, will move us into another directory. To go one directory upwards in the current folder path, we can type cd dot, dot. You can also cd into a subfolder by typing cd, and then the name of the subfolder. If you want to jump to the beginning home folder, you can type cd tilde. Now, there are several other options on how to use the cd command, but for this course we will mostly be using the cd.. command to go up one directory. Another useful command is how to list the files in a directory. For Linux and Mac users we type ls, and for Windows users you can type d-i-r, or dir. To create a new empty file, we can use the command touch for Mac and Linux users, and the command copy con for Windows users. And you can create a new empty folder by using the mkdir command. As mentioned, these are just a few simple commands that you can use in the command line. Now there are many, many other commands, and even other options to use with these few commands that we just learned, but for now we'll stick to just these few commands.

Git GUIs

Other than using Git in a command line tool, Git comes with built-in graphical user interface tools calls GUIs that you can use to perform Git actions to manage your project, as well as many other third-party client tools, as shown here at git-scm.com/downloads/guis. At this point, I advise against using one of these GUIs until you fully understand the Git commands that you'll be using and how they work together. Now, these GUIs or client tools can be nice and very useful to use, especially for making more advanced Git operations that would be complex using the command line. However, with using GUIs, you only see the result of that action, and not how that action was made, or the command or series of commands that were used to make that Git action. In fact, a lot of Git-proficient users prefer to use Git in the command line tool over GUI, because it gives them complete control over what is happening. You'll learn a lot better, and understand really how Git operates and the Git commands that are used if you take the time now to learn and understand the basic individual Git commands. So for this course, we're going to skip the use of GUIs, and use Git in the command line for most of our interaction.

Install Git

Now that we've learned some terminal commands, we now need to install Git on our machine. Regardless of the system you're using, this is a pretty easy and straightforward process. That is basically a one-line command or a click on a downloadable link. If you're running Linux, Git is easiest to download using the package manager for your Linux distribution. Debian users can use sudo apt-get install git, where Fedora users can use sudo yum install git. If using Windows, there are several different programs that you can use to install Git, but the easiest that I've found is at get-scm.com/download/win. This automatically starts the download of the Git installer, and this installation also installs a command line tool called git bash, that will give you a more Linux feel on a Windows system to perform the git commands. For Mac users, you can use the same URL that was used to download Git for Windows, but instead of /win, type /mac. Like the Windows installer, this will automatically download the latest version of Git. Mac users can also use Homebrew, which is a popular free and open source software package manager that simplifies the installation of software on Mac OS. If already installed, you can use Homebrew to download a number of different software packages, but to download Git using Homebrew, just type brew install git in the Terminal window. Now since Steve is on a Mac computer, we'll be using the git-scm.com/download/mac to download Git.

Configure Git

After you have finished installing Git, you can verify if the installation was successful by typing git --version. The response of this command will be the numbered Git version that was just downloaded. Now that we have verified that Git has been installed, the first thing that we should do is set our username and email address. When we use Git to make changes to our project's files, Git uses this information to identify who made these changes. Git has a tool called git config that allows you to set configurations to Git. So to set a username, type git config --global user.name, and then the name in the parentheses. Let's use Steve's name. And then to set an email address, type get config --global user.email, and then the email in parentheses. Again, let's use Steve's information. Now that we have Steve's username and email address configured in Git, his file changes will now be identified to him, and since we're using the --global option, we'll only need to do this once, and Git will apply it to everything we do in the system. If you want to change this global setting with a different

username or email address, but only for a specific project, you can run the same commands, but without the --global option when you're in the project. This will overwrite your global settings, but just within that project, and keep the global settings for everything else. If you want to check your configuration settings, you can type get config --list, and it will list all of the settings Git has at that point. If you want to check a specific setting, like the username, for example, you can type get config user.name, or any of the other settings that you have configured, and it will respond back with a set value just for that setting.

Git Help

Now, if you ever need help with Git, or if you forgot what Git command does what, you actually already have the complete Git manual at your fingertips, because you downloaded it when you installed Git. In fact, you have two help options. First you have the full manual, which can be seen by typing man git, and it covers pretty much every Git command that you can make with a description of what it does. As you can see, it's quite long, and scrolling down to the bottom can take a minute. You'll also notice that the full manual has a section for high-level commands, or what Git calls porcelain commands. These are your main Git commands that are most commonly used day-to-day, and are the commands you'll use for maintaining your Git project, and the commands that we're covering in this course. If we scroll a bit further down, you'll notice the lower level plumbing section. Now these are commands that give you access to the deeper inner workings of Git. Many of these commands aren't meant to be used manually on the command line, but are meant to be used in tools or scripts. You can shortcut to a specific section in this manual by typing git help, and then the section you want. So, for example, if I wanted to read more about the config section, type git help config, and it takes you straight to that section in the Git manual. The second help option is a quick reference guide that has some of the most common and useful commands used in various situations. You can see this list by typing git help, and then it will return a much smaller list with commands that you'll use most of the time. This is a shorter list of the porcelain commands that is easier to get to than finding them in the full manual. At this point, don't pay too much attention to a lot of the commands in the full manual, especially most, if not all, of the plumbing commands. Most of them are for advanced Git usage, and using them without understanding exactly what they do can really mess things

up. For now, just know that you have these helper guides whenever you need a quick reference.

Initalize a Git Repository

Now that Git is installed, it's time for Steve to add Git to his recipe project. This is done by initializing a new Git repository. A Git repository, which is often abbreviated as just repo, is a virtual storage of your project. In other words, each project in Git is referenced as a repository. To initialize a new project or a project that already exists, as is the case for Steve, you first need to cd to that project folder using the command line. Now Steve has his project folder saved on his desktop, so let's cd to his project by typing cd desktop/, and then the name of the folder, which in Steve's case is called café_recipes. Now that we're in his folder, initializing a Git repository is done by typing the git init command. Once executed, you'll then see a response message that says Initialized git repository in the specific file path of our project. By initializing your project, it added the Git configuration to a .git subdirectory that has marked this project folder and all of its contents to be version controlled. This .git subdirectory is hidden by default, but you can go into that folder by typing cd .git, and then you can use the ls command to provide a list of the files in this folder. You'll notice that there are several files that were just created, but for the scope of this course, just know that this .git directory just contains all of the important Git metadata for this new repository. Now, as you learn more about Git, you'll understand more about what these files do, but for now, just know that they exist. We can get out of the .git subdirectory and go back up a level to our parent directory by typing cd dot, dot. If we now type ls again, we get a list of all of the recipe files that Steve has saved under his project folder. Even though we've initialized Git on this project, the contents in these files haven't changed, they've only been added to be version controlled by Git.

Introduction to GitHub

Steve has now installed, configured, and added Git to his project by by initializing his project as a Git repository. His files are now being version controlled by Git, and they are ready to be tracked of any changes he now wants to make. But this only solves one of Steve's main problems. He now has a better way to manage his project, but everything is still local on his café computer. If he forgets his computer at work and wants to contribute to his project from his house, or if he wants to access his project on one of his several other devices, he still can't work on his project because it's still only

on his local computer at the coffeeshop. In order for Steve to solve this problem, he needs a way to host his Git repository in an online cloud-like setting. That way he can sign into his Git repository online, and pull down his project on his multiple devices. Now, there are several companies that specialize in this type of product offering, but one of the most popular, if not the most popular option, is GitHub. GitHub is a web-based Git repository hosting service, which is exactly what Steve is looking for. With using GitHub, Steve can push his local Git repository to their online hosting service where he can access his repository any time he has internet access. In addition to being a Git repository hosting service, GitHub provides a lot of additional features to help improve your Git repository. The most important feature GitHub offers is its ability to integrate collaboration seamlessly into your Git repository. Being able to discuss and collaborate changes to your repository with other developers, contributors, team members, is really what makes GitHub one of the best at what they do. With so many great features that GitHub has to offer, there is a lot of information and detail around GitHub that won't be covered in this course. Regarding our Wired Brain Coffee project with Steve, he currently is the sole contributor to this project, so a lot of these collaboration features with GitHub won't be discussed in full detail. However, as Steve's café and project continue to progress and then see success, he's working towards adding additional contributors to his project in the near future, and he's excited to leverage the collaboration-rich features that GitHub has to offer and take his project to the next level. But before Steve could start pushing his repository to GitHub, he needs to first create a GitHub account.

Create a GitHub Account

Creating a GitHub account is pretty easy, and can be done in just a few simple steps. So let's go to github.com where Steve can get signed up. At the home page, you'll see at the top right corner the option to sign in or sign up. Let's click the Sign up option to get started. We're then brought to a quick form to fill out by entering a username, email address, and password. Let's enter in Steve's information. We'll need to pick a username that is available. Let's try wired-brain-coffee. With that username available, we'll enter Steve's email address, and then we'll enter in Steve's password. Make sure to go over the terms of service and the privacy policy. We won't take time to read and discuss those in this course, but it's a good idea to understand what those are, and to agree to GitHub's policy before

continuing on. The next step is to choose the type of personal plan you want to have. Now there are two options. The first is a free plan to create unlimited public repositories that will be hosted on GitHub. The second is a paid plan to create unlimited private repositories. Now, since Steve will be pushing his sensitive café recipes to GitHub, it would make sense for him to sign up with the paid plan so that his repositories will remain private. The steps to push content to GitHub won't change with these two plans, other than the fact that only authorized collaborators that Steve can manage will be able to see his project. For now, let's go ahead and stick to the free public plan. Note that you can always cancel or upgrade your plan at any time. There are also a few other optional checkmarks that you can read more about if interested. One is for setting up an organization, which is best suited for businesses who need to manage permissions for many employees, and the other is to sign up for GitHub news, offers, and events. Again, for our purpose, let's just leave those two blank and then hit Continue. The last step is optional, but it contains a few questions to answer so that GitHub understands its users better. They're basic questions, but again, for our purpose let's go ahead and skip this step by clicking the option at the bottom of the screen. We then see a page that says Learn Git and GitHub without any code. GitHub then provides two options, one to read a guide and the other to start a project. Since we'll be going through these steps with Steve's project, let's get started by clicking on Start a project. This then takes us to one last screen where it asks us to verify our email address that we entered at the beginning. So let's jump over to Steve's email so GitHub can verify our account. In the email, we'll click on the Verify email address, which will verify that our email address with our GitHub account, and will take us back to GitHub where we see the Start a project button again. Let's click that button to push our local Git repository to GitHub.

Push a Local Repository to GitHub

Creating a new repository on GitHub starts with giving it a name. Steve has chosen wired-brain-recipes. Next we can add an optional description. You then have the option to make this repository as public or private, and then there is an option to initialize this repository with a README file. A README is a quick and simple way for collaborators to learn more about your repository. We can add a README file by clicking the button here, or by adding it using the command line. It's as easy as clicking this checkbox

on GitHub to add one to our new repository, but we'll add it to our local project using the command line before pushing it to GitHub. There are two other options. One is to add a gitignore, which is a way to add files that you don't want version controlled by Git, and is something we'll discuss a little later on in this course. The other is to add a license to your repository. You can learn more about the different licenses you can add by clicking the information icon next to the drop-down option. For real projects, these options are probably a good idea to add, as recommended by GitHub, but for our example project with Steve, we'll leave both of these options as their None default, and continue with clicking the Create repository button. We're now taken to a GitHub page that provides some different scenarios and steps on how to push our repository to GitHub. We're first prompted to either use HTTPS URLs or SSH URLs. The HTTPS URLs are recommended and are the default option. Now, these URLs will work really everywhere, even if you are behind a firewall or a proxy. Using HTTPS URLs to push or pull changes to or from your repository, GitHub will ask you for your GitHub username and email address for verification. There is also an option to use the SSH URLs to provide access to your Git repository via the SSH secure protocol. In order to use these SSH URLs, you need to first generate an SSH key pair, and then add that public key to your GitHub account in your profile settings. Using SSH URLs to push or pull changes to or from your repository, GitHub will prompt you with a password, and you'll need to enter in your SSH key passphrase. Now, we're not going to go over setting up SSH, but you can find more information at the URL listed below. So for Steve's project, we're just going to use the default HTTPS URLs. Since we already have a Git repository, we're going to use the second option to push an already existing repository from the command line to GitHub, but we'll use some of the steps in the first scenario to create a README file and add our first commit before adding the remote origin and pushing our repository to GitHub. Before we switch over to the command line, we'll need to copy the HTTPS URL at the top of the page. Jumping over to Terminal, and once there, let's add a README.md file by typing echo, and then in parentheses, hashtag, or the pound symbol, space, wired-brain-recipes, and then two greater than sign symbols with the word README.md. The echo command says that we want to write wired-brain-recipes with an h1 HTML header into a new file called README.md. Now, you can really put anything you want into this README file, but by default GitHub just includes the name of your

repository. Now that we've added the README file to our project, it's currently being listed as an untracked file similar to the rest of our recipe files. Now, we've initialized this project to be tracked by Git, but we need to add these files to our staging area so that they can be committed, and thereby tracked by Git. We can add these files by typing the Git command git add, and then dot. This command will add all of these files to our staging area. And now that these files have been added, we can now make a commit with a message that says first commit. We can do this by typing get commit -m, for message, and then in parentheses first commit. Git then responds back with some interesting information, and it can tell us what we just did. We then need to connect our local repository to the one we just created on GitHub by linking the two repositories with the Git command. This can be done with the remote add origin, and then the HTTPS URL that we just copied. This command adds the origin GitHub repository to our local repository by creating a link that allows us to push and pull changes between the two. Now we need to push our repository that is now pointing to our GitHub repository by typing git push -u origin master. We're then prompted to enter in our GitHub username and password. And if verified, Git responds back with some information that at the bottom says, Branch master set up to track remote branch master from origin. Now, this means that everything worked, and we have successfully set up to track our local Git repository from the origin GitHub repository. Now, if we go back to our GitHub profiles main page, we now see that we have one repository listed, and if we click on that repository section, and then on the actual repository, we can see that our wired-brain-recipes Git repository that was local is now being hosted and tracked on GitHub.

Summary

In summary, we learned how to install Git on Linux, Windows, and Mac systems, as well as how to configure Git by adding our username and email address as global settings to Git, and even how to set individual settings to specific projects. We then learned how to initialize a Git repository by using the git init command to our existing recipe project. This command added a .git subdirectory to our project that contains the necessary metadata for Git that works behind the scenes. We then learned about the two Git helper guides, with one being a full Git manual that details the Git porcelain commands, which consists of the basic most common Git commands, and the more advanced plumbing commands,

which are used as building blocks for creating scripts and tools. We were then introduced to GitHub, and learned that it's a web-based Git repository hosting service, and learned how to push a project repository to GitHub. We can then access our project from any computer or supported device that we have internet access to. When we pushed our repository to GitHub, we used the git add and git push commands to accomplish certain tasks. Git then responded back with information on what just happened. In the next module, we're going to dive more into Git commands such as these, and learn more about what they do and what the information actually means that Git responds back with. These next Git commands are referred to as the common porcelain Git commands that we referenced earlier. This next module is pretty important, so sit back, adjust your seat, and get ready to master Git.

Basic Commands of Everyday Git

Introduction

Hi, welcome to Getting Started with Git. This module is all about understanding some of the basic commands of everyday Git. In the previous module, we installed Git and learned how to set some global and local configurations using the git config command. We also learned how to locate and use two main help resources, one being a full Git manual by using the man git command, and the other being a quick reference guide by using git help. We were then introduced to GitHub, the web-based Git repository hosting service, where we learned how to create a new account and create a new repository to push our existing local Git repository to GitHub. As we continued our story with Steve and his Wired Brain Coffee project, we then pushed his repository to GitHub using the command line. By doing so, we used some basic high-level commands referred to as porcelain commands, such as git add and git push. We then saw some interesting information that Git provided as a response when we executed these commands. Now in this module, we're going to understand this information that Git provides in these responses, and learn more about

what these Git commands do and how we can use them to manage our project. The commands introduced in this module are common everyday Git commands that you'll be using quite frequently when working on a Git repository. So, let's get started by going back to our Terminal command line where we left off.

Git Status

As you have probably noticed, a nice thing about Terminal is that we can see the history of commands that we've already executed. They don't go away when we run another Git command. Now this is great for remembering the steps taken to reach our current state, but as you can see, the majority of our screen is taken up with our previous Git commands. This can leave things a bit cluttered, especially when using help guides or similar commands that respond back with a lot of information. It's often nice to clear this information out and start with a clear Terminal window. We can do this by typing the clear command. This clears out our window and moves our command prompt to the top of the screen. Now, we didn't erase this command history. We can always scroll back up to view our previous commands. All the clear command does is move this history out of view. Now, before we ran clear, the last command we ran was to push our committed changes to the origin master by typing git push -u origin master. Remember, the origin is the repository version that sits in our GitHub profile. Since we just pushed our local repository to GitHub, these two repositories, our local repository and the remote origin repository on GitHub, should be the same. But with Git, you don't have to guess or try to keep track of the current state of your project or files. You can check the status of your project at any time by using the git status command. If we type git status in our command line, Git will respond back with a few lines of information. The first line is stating the branch that we are on, which is the master branch. Now, back in a previous module when we initialized Steve's Git repository, Git created a master branch automatically. A branch in Git is a lightweight movable pointer to your project at a specific point in time. Now since our two repositories are in sync with each other, our local master branch is equal to the origin master branch. This default branch that was created is called the master branch. If you're a little confused on the idea of a branch, don't worry, we're going to cover branching in detail in the next module, but for now just understand that when you initialized your Git repository, Git created a branch called master. And when you make

changes in Git, you make them on a specific branch. Now branching is a core concept in Git, and a feature that adds a lot of value by allowing many different types of workflows that can involve many different types of branches. But at least for right now, we'll keep things simple and make all of our changes directly on the master branch. The second line in the response mentions that our branch is up to date with the origin master. Now this one is pretty easy to understand. It's saying our local branch, again, which is master, is up to date with the origin master branch, the one being hosted on GitHub. And the third and final line is saying that there is nothing to commit, and that our working tree is clean. If you remember from the last module, we haven't modified any files, so there are no changes yet in our working directory, and nothing yet that has been staged that is still waiting to be committed. The git status command is a simple, but important command, because it shows us the state of our project at this particular point in time. Now this is a git command that we'll be using quite frequently moving forward.

File Stages

Now that our local repository is equal to our origin repository, Steve is ready to make some new changes in these files. Each file in our project can be in one of two states. It can be tracked or untracked by Git. Remember, tracked files are ones that were in the previous Git snapshot. Tracked files can be in one of three states, committed, which are unmodified changes from the last snapshot; modified, which are files that we've added changes to; and then staged, which are files that we've marked to be added to our next commit snapshot. When we make a change to our committed or unmodified file, it moves from a committed stage and into the modified stage. When we're satisfied with our changes and we want to stage them for commit, we add them to our staging area. We then take the changes in the staged area and commit them to our repository. These changes are now saved in our repository in that commit snapshot. and they are then moved back to the committed stage. Now, going back to the Terminal command line, we can run git status and see which files are in which state. Now since we have nothing yet to commit, and our working tree, again, is clean, and all files are in the committed state from the last snapshot, as we make changes to Steve's project, keep in mind the three stages and see if you can predict the state of file we'll move into as we run these commands.

Track a New File

In Steve's repository, he realizes that he doesn't yet have a file to keep track of his suppliers, so Steve wants to add a new file into his repository called vendors. So let's add this new file to Steve's repository by typing touch vendors.txt. If we run the git status command, we now see that the vendors.txt file is now showing under the untracked files. Remember, untracked means that Git sees a new file that didn't exist in our last commit snapshot. Now Git won't track anything unless we explicitly tell it to do so, but Git does provide us with some helpful information on how to add this untracked file. The bottom message says, nothing added to commit, but untracked files present. Git then lets us know how to add an untracked file by using git add, and then the name of the untracked file. So let's start tracking this file by typing git add vendors.txt. If we now run git status, we now see that the vendors.txt file is showing as a new file being tracked, and that it's a new change waiting to be committed. Modified files that have been added are now in our staged state. Git has provided us with another message that says use git reset HEAD, and then file, to unstage. Resetting or removing files from the staged area is something we're going to cover in the next module. For now, we just want to commit this new file. We can tell that this file is staged, because it's now under the Changes to be committed section.

Stage Modified Files

If we committed this new vendors.txt file now, the contents of this file when we perform the git add command will be committed to our project in that snapshot, but before committing this file, let's add some changes to a file that was already being tracked. If we type the ls command, we see a list of all files under our project directory. We see the vendors.txt file because we've added it to our local master branch, but we haven't committed it yet. Let's open up employees.txt, a file that is already being tracked, and add a new employee that was just hired. We can do this by using the command open -a, and then the name of your text editor, which in Steve's case is atom, and then the name of the file, employees.txt. Now let's add the new employee and his information to the file. Let's now save that file and go back to our Terminal window. If we type git status again, we see that the employees.txt file is now under the Changes not staged for commit section. This means that the file is being tracked by Git and has been modified in our working directory, but it hasn't yet been added to our staging area for commit. Now we can stage this file by using the git add command. The git

add is used to track new files, as well as add modified files to be staged. You can think of the git add command as add this file to be in the next commit snapshot. So, let's type git add employees.txt, and add it to the staging area. Let's now run git status again and see what happened. Both files are now staged, and are showing as changes to be committed. Git provides us with information on what kind of file is being staged next to the file name. We can see that the employees.txt file is a modified file, and the vendors.txt file is a new file ready to be committed. But before we commit these files, Steve realized that he recorded the new employee's phone number incorrectly, so let's re-open the employees.txt file and make that change. We can open the file again by typing open -a atom employees.txt, and let's make the change. After making the change, let's save our file again and go back to our Terminal window. Let's run git status again to see what happened. Notice that employees.txt is now showing as staged and ready to be committed, as well as showing as modified changes not yet staged. Now, this may seem a little confusing, I mean, how can a file be in two states at the same time? Well, the answer is how and when Git stages a file. When we first staged our employees.txt file by using the git add command, those changes were added at that point in time to be staged. When we fixed the phone number and then resaved the file, git sees this as a new change that was not added when we previously staged that file. If we committed the files in our staged area right now, the version of employees.txt when we added it to the staging area is what will be committed into the snapshot. This will not include the fix we just made, since we haven't added that change to be included into our next commit snapshot by staging it with the git add command. If you change a file after you run git add, and you want those changes to be included in your next commit snapshot, you have to run git add again on that file to add those changes to be staged. So, let's run git add employees.txt again, and then run git status to see what happened. We now see that the employees.txt file is now only showing in the staged area, which again, are changes to be included in our next commit snapshot.

Short Status

So far, we've run the git status command several times already, and each time we run git status, Git provides us with a pretty comprehensive response. But after getting used to Git's response when using git status, it can be useful and nice to have a way to see these changes in a more

compact response. Git has provided a short status that provides the same amount of information, but in a much smaller and concise way. Now we can see the shortened response by typing git status -s, or git status --short. By running this command on our project, we get back a response that has our two files that are waiting to be committed, but now with some interesting letters in front of them. Right now I bet you're pretty confused by looking at this response, because it doesn't make much sense unless you understand the format response that the short status uses. The key to understanding the short status response is knowing what the letters in front of the files mean, and their placement. After running this command, Git will place the name of the files in the far right column. There is then a middle column and a far left column. The far left column indicates that the file has been staged, and the middle column indicates that the file has been modified, but remains in the working directory. The letter m means modified, and the letter a means new files that have been added to the staging area, and two question marks mean that there is a new file that is untracked by Git. So, in this example, file 1 is a modified file in the staging area, file 2 is a modified file in the staging area, as well as modified in the working directory. This was the same case with our employees.txt file when Steve made a change after staging it. And file 3 is a new file added to the staging area, and file 4 is a new file created, but not yet being tracked. So, jumping over to our Terminal window again, we can see that our employees.txt file is showing modified in the staging area, and our vendors.txt file is a new file that has also been staged.

Git Diff Explained

The short status is nice if you just want to see which files reside in which states, but what if you want to go the other direction and see more information than just what git status shows? Well, Git has a command for that too. If the git status command is too vague, and you want to see the actual changes being made in these files and not just know which files have been changed, you can use the git diff command. You can use git diff to answer two questions, what changes have I staged that are ready to be committed, and what changes have I made, but not staged? If we want to see the changes we've staged that will go into our next commit, we can use git diff --staged. This command compares our staged changes to our last commit snapshot. When we run git diff --staged, we're going to see an interesting detailed response. So let's go over how to read this output. The

first thing we see are the compared file versions. Git diff is comparing file a with file b. Now in almost every case, file a and file b will be the same file, but just in different snapshot versions. This is shown with diff --git, and then file a followed by file b. The next line is the file metadata. Now this is technical information of our files. The first two numbers are the hashes or IDs of the two file versions that are being compared. As we were briefly introduced to SHA1 hashes in a previous module, git identifies a file object at a specific version by using the SHA1 hashes. The last number is most likely going to be 100644. Now this is just an internal file mode identifier that represents a normal file. Now, there are other number identifiers, but diving into this metadata information is more advanced Git knowledge that won't be covered in detail in this course. Again, for now, just know that it's there, and the basics of what it is. Next we see the change markers for file a and b. File version a is assigned with the minus symbol, and file version b is assigned with the plus symbol. Next there is the diff chunk, which is made up of two parts, the chunk header and the chunk changes. At the top of this chunk, is the chunk header, and the header is enclosed with two @ symbols on each side and will tell you which lines were changed in the file versions. File a is represented with a minus sign, and file b is represented with a plus sign. In this example, the -12 is saying that the minus sign, again, being file a, begins at line 12 and then has two lines that were modified. The +12 is saying that the plus sign, being file b, also begins at line 12 and then has three lines modified. After the chunk header, we have the chunk changes that contain the actual line changes. Now this often includes a few lines of unchanged lines before or after the changes to help provide context of where these changes are being made in the files. These changes will either have a minus or a plus sign in front of the change to show you what version that change is occurring in. However, for files with only one or two lines of content, these extra lines may not show up. So let's jump over to our Terminal window again and run this command to see how it looks in our project.

Git Diff in Action

Let's clear out our window again by typing clear to bring our command prompt up to the top of the window. By running the git diff --stage command, we see that the compared a and b version files is employees.txt. Next we have the metadata with these file version hashes, and the file mode identifier, which is the ID for a normal file. We then see the change

markers for the two file versions of employees.txt with the minus being the last committed snapshot version and the plus being the new changes that are in our stage state. We then see the chunk header that shows us the line changes being made in these files, and then we see the chunk changes where we actually see what is being changed. Now we also see another compared a and b version files, but unlike the employees.txt file comparison, this file a and file b comparison is with different files. File a is employees.txt, and file b is vendors.txt. Below, we see some interesting information that says similarity index 100% copy from employees.txt and copy to vendors.txt. So, what's going on here? Well, the reason for this is how Git thinks about file changes. Now Git doesn't care what the name of the files are, Git really only cares about the content inside of those files. Now we added a new vendors.txt file that is currently an empty file, and the other file in our staging area is the employees.txt, which was also an empty file before we added some content to it by adding the new employee's information. Now Git sees this similarity and thinks that we just copied the employees.txt file and then renamed it to vendors.txt. The similarity index is the percentage of unchanged files. Well, and since these two file versions are the same, the similarity index is 100%, so Git thinks it's the same file. Now, since this is not the case, we can get around this by explicitly telling Git that this is not a renamed file, but it is, in fact, a different file, and to show us the git log as it should be. We can see the correct diff information by again typing get diff --staged, but this time let's add the --no-renames option. We now see the compared a and b version files for vendors.txt, and some similar formatted information that was in our employees.txt version differences. Git tells us that this is a new file with the normal file identifier, and then we see the file metadata. Notice that the first hash reference is a hash of zeroes. Because this is a new file, Git wasn't previously tracking this file, so there is no hash reference to compare it with. The second hash, which is e69de29, is the new hash for the contents inside vendors.txt. But just a second, the vendors.txt file is empty. There isn't any content in this file. Since Git is now tracking this file, Git needs to create a hash of contents from the file even if the file contents are empty. Creating a hash on an empty file responds back with a hash reference of e69de29. Remember, Git only cares about the hash reference of a file to keep track of its content. Since the vendors.txt file has a hash of e69de29 and the beginning file version of employees.txt before the added employee change being file a also has the same has reference of e69de29, this is why Git

thought we just copied and renamed the file. Now the git diff --staged shows us the information on changes that are staged, but what about changes that haven't been staged and are still in our working directory? We can see this information by just using git diff. By running that in our Terminal window, we see that we get nothing in response. Well, this is because we currently don't have any changes in our working directory. So let's add some content into our vendors.txt file, and see where these changes show up. We can open it using open -a atom vendors.txt. Let's now add a new vendor with a name, location, and phone number. Okay, let's save this file and return back to our Terminal window. If we run get diff, we now see this exact change in our vendors.txt file being compared to its previous version. If we run git status, we see that this file change is modified, and in our working directory. So let's add this change to be staged with the other changes by typing git add vendors.txt. We can verify our actions by typing git status again. And now we see that both our employees.txt and vendors.txt files have staged changes ready to be committed.

Commit Changes

At this point, let's add another vendor to our vendors.txt file by opening it back up in atom using open -a atom vendors.txt. Underneath Jones Coffee Cups, let's add another vendor called West Summit Supplies, with their location and phone number. Now let's save this file and go back to our Terminal window and run git status. As expected, we now see our vendors.txt file as modified, and in our working directory. At this point, we can add it to our staging area by using the git add command, and then run the git commit command to commit this change to our repository. Now going through these series of steps for multiple changes is nice for organization, but it can be a bit unnecessary for small simple changes such as this one. Now, git provides a shortcut if you want to skip the staging area completely and jump right to committing your changes. You can use this shortcut by typing git commit, and then the -a option. The -a option automatically stages all changes in your working directory that are being tracked by Git, essentially skipping the git add step. In the same command, you can then use the -m option to add a commit message as we did before. So let's type git commit -a -m, and then our message. If we then run this command, we can see it in the git output that our commit snapshot was successful. Let's now run git status to double check. As you can see, there is nothing to commit, and now our working tree is clean again. Alright, we

just made our first commit. Git also responded with some output to verify that everything was committed successfully. Git provides the branch we made the commit on, which in our situation is the master branch. We then see the SHA1 hash from the commit, as well as the commit message that we provided. We then see the number of files that we changed, and some stats on how many insertions and/or deletions that were made in these files for this commit. If a new file was added with this commit, Git shows us that too on the next line that says create mode 100644, which is saying a normal file was created, and then the name of the new file, vendors.txt.

Summary

In summary, we've learned how to check the status of our project using the git status command. This is a command that we use often to frequently check how changes can move from our working directory to our staging area. We then briefly revisited the different file states that we learned in depth from a previous module, and then used the git status command to actually see how our files move around in Git. We then learned how to track a new file by using the git add command, as well as using the same command to add or modify changes to the staging area. then we learned about a condensed version of the git status command by using git status -s or git status --short, and then learned how to read the output of that command. In contrast, we then learned how to get more status information by using the git diff command, and then learned how to read that output that consisted of some Git metadata, chunk headers, and the chunk changes for each modified or staged file. We then learned how to commit our staged changes by using the git commit command, committing our changes, we then learned how to skip the staging area by using the -a option in our commit command that resulted in taking any modified changes that were being tracked and then adding and committing them in the same git command. Now, we still have some very important topics to discuss. In fact, we've only yet introduced one of the most important things about Git. So, in the next module, we'll cover a few more basic Git commands to help us manage our project better, and then we'll dive into one of Git's core features and see how we can elevate our Git project to the next level.

Extended Commands of Everyday Git

Introduction

Hi, welcome to getting Started with Git. Up to this point, we've learned quite a bit about what Git is, why Git is useful, and how Git manages your project. Now that we have a basic understanding on some common everyday Git commands, we'll be building upon that knowledge and learn some more fundamental commands that, depending on your project's workflow, may or may not be used as frequently as the previous commands, but nonetheless, these are very important commands to know and use to better improve and manage your project. Now, we have a lot to cover in this module, including a core concept that we've only yet briefly introduced in previous modules, and something that we've been building up to understanding by learning some of the basic Git commands first. So, let's continue our story with Steve and the Wired Brain Coffeeshop project, and see how Steve can continue to use Git to improve his project and fully manage his files with transparency and optimization.

Push to Remote Origin

The last command we ran, other than the git status command, was to commit our staged files to our local project. By committing these files, we created a new commit snapshot that now includes our changes and moves our local master branch pointer to our newest commit. We did this by adding a commit message, and then Git returns some information to confirm a successful commit. Git then provides some information on the files changed with the number of insertions and deletions made to each file. We then ran the git status command to see the status of our local project, and this time Git returned some new information we hadn't yet seen with the git status command. So let's jump back over to our Terminal window to where we last left off. Let's run git status again to see the current state of our project. Git now tells us that our branch is ahead of origin/master, which, again, is our project being hosted on GitHub, by 1 commit. Git then provides a helpful note on what is a common next step after commit is

made. Git says, use git push to publish your local commits. Now, remember, when we made this commit to our local project, those changes were only on our local project. If we were to view our project version on GitHub right now, it wouldn't have any knowledge of this commit. Notice that we don't have to push our commit to our project at origin/master every time we make a commit. Now we can continue to make commits to our local project as much as we want, and then when our local project is at a point where we want to send our changes to our origin project, we need to push our local project upstream. We can send our changes to our origin project by using the git push command, and then our remote name, which is origin, and then the branch name we want to push up to, which in this case is master. So let's go ahead and run this command now to push our local changes to our origin project on GitHub. If verified, Git now returns some confirmation data to let us know the push was successful. We see some compression information, as well as the GitHub URL path to our project repository that we were pushing to, along with the SHA1 hashes and the local master to origin/master mapping. Now, there's a lot of specific information here to digest, but the important thing to note is that our push was 100% successful, as shown in the Git response, and that our one commit was pushed to the origin repository without error.

Check Commit History

Let's run clear to bring our command prompt back up to the top. Now that we've pushed our local changes upstream to our origin repository on GitHub, let's run the git status command to see the state of our local project. We now see that our branch is up to date with origin/master, and that there is nothing to commit, and currently nothing in our working directory or working tree. Now at this point we've only made two commits so far on our local project, the first being our initial commit when at the beginning of our project we ran git init, created a README file, and then made a commit with the message of first commit as we then added the remote origin project URL that was provided by GitHub when we created our repository placeholder to host our project, and then pushed our local project to GitHub. The second commit being the one we just made with the commit message of add new vendor. Now, as you can probably guess, we don't have to keep track of each commit we make. As our project progresses and more and more commits are made, it will become unrealistic to mentally keep track of all commits and changes that have

taken place. So Git has a way to view our existing commit history by using the git log command. By running git log, Git returns some information related to our previous commits, and by default, the git log command lists the commits made to our project in reverse chronological order, being the most recent commits will show up first. Up to this point, we've only seen part of the SHA1 checksums that we've made commits or compared file versions using git diff, but with using git log, we see the full SHA1 hash of our commits. We also see the name and email of the author who made these commits, which in our case is Steve. The author and email configurations were set up using the git config command earlier in this course. We also see the date with the timestamp of when the commit was made, as well as the commit message for each commit. And as you can see, this is a quick way to have a high-level overview of our previous commits and know who made each commit, when a commit was created, and what that commit was changing. Currently, viewing our commit history like this works well for us now, but as Steve's project progresses and more and more commits are made, looking for a particular commit by just using git log, we'll end up with a lot of scrolling to find the commit that we're looking for. Also, git log just shows us some basic information on each commit. It would be nice to dig into a specific commit to see what exactly was changed in that commit, similar to the results that we saw from using git diff. Well, as it turns out, there are a lot of options that can be used as parameters when using the git log command to give us exactly what we're looking for. Navigating our project's commit history is pretty important, but with so many options it can be a bit confusing on knowing how to get the data we want, or even where to get started. So, to begin, let's go over some of the most commonly used options when using git log. But, remember, there are many, many more options to use with git log in getting the exact data the way you want it, and the different ways to modify the commands we're about to learn, but this will give us a good starting point to move forward. If we want to limit the number of commits shown in the log, we can type git log and then the number we want to limit the log to. Now, since there are only 2 commits so far in our log, let's type -1 to limit the git log output to 1 commit, and then hit Enter. We see that only the new add vendor commit was returned in the log. Remember, the log output returns the commits in reverse chronological order, so we get the latest commit first. Notice that nothing has changed in the log output response with this command, we simply just ask for a limited number of commits to show,

which was 1 commit. But what if we want to see all the commits, but in a more simplified way? Well, we can use the git log --oneline command. Now if we type that into our command line and hit Enter, we get a list of all our commits in a single line format. Each line in the response is a commit, and it has the commit message, as well as a few digits of the SHA1 hash of that commit. This is a great way to see a simplified high-level overview of all the commits in our project history. Now let's go the other direction and get a detailed view of each of our commits. And we can do this by typing git log --stat, and then hit Enter. We now see that the log output is the same as if we just use git log, but now with the addition of the files changed in each commit. This output may look familiar because it's very similar to the output that git status provided when we were making the changes in our staging area and then when we ran the git commit command. It's nice that we get this detailed information back on any commit by running this command in git log. Now, if we want to get even more more detail than this, we can type git log --patch, and then hit Enter. This command will give us the full diff of the exact changes in each file with each commit. This is similar to the results that we got back when using the git diff command, but instead of doing it on changes in our staging area, we can go back to any commit in our project's history using git log and see this detailed view. We can navigate these changes using the up and down arrow keys, and when finished, we can type q for quit. Now, again, there are so many options when viewing the commit log. We can search for author, date range, or even for certain commit messages. But for now, just know that we've only scratched the surface of what we can do when using git log.

Commit Messages

Now, let's run git log again, but without any parameters, and then hit Enter. We see the typical output response of git log with the commit SHA1 hash, author, date, and commit message. At first glance, it can be very difficult to know generally what was modified in each commit if the commit message said something vague or didn't make much sense. Now, this may not be too much of a problem if you're working on the project by yourself, but as most working projects include other contributors, it's incredibly important to have meaningful commit messages so that others, as well as ourselves, can understand the changes that are being made in that commit. For example, let's look at our first commit, the second in the list, with the message of first commit. Now this doesn't tell us anything that happened in

this commit except that it was our first commit. Now without looking further, we don't know which files were changed, or what was changed in those files. This can make life very difficult if we needed to revert a commit that introduced a bug and we needed to find that specific commit, or if we just need to review specific changes introduced in a commit. If the commit message isn't detailed enough, life for every contributor on the project can get pretty frustrating. So for this reason, there are some general rules and guidelines when writing a commit message. Of course, these are just guidelines, and can change depending on how the project is structured to be maintained, but nonetheless, these guidelines are followed in most cases. Chris Beams wrote a great blog post in 2014 about writing commit messages that outline the general 7 rules to a great commit message. We can find this blog post online, and we can even open it up in our default web browser from our command line by typing open http://, and then the URL, which is chris.beams.io/posts/git-commit. Now there is a lot of really good information here supporting why good commit messages really matter, and Chris goes on to list the 7 rules of a great commit message. First, to separate the subject line from the body with a blank line. Now, up to this point, our commit messages have only been a one-line message written from our command line, and they didn't include a body. A lot of commits don't require both a subject and a body. Sometimes a single line is sufficient for the changes being made. For example, add a new ingredient to coconut oil coffee recipe. A change like this may not need a body in the commit message. In addition, adding a commit message with a body is best done using a text editor to write the message, rather than using the -m option in the command line. Second, limit the subject line to 50 characters. The commit message should be short and to the point. If it's too long, it can be a tedious process to filter through lines and lines of messages to find exactly what you're looking for. And third, capitalize the subject line. And fourth, do not end the subject line with a period. Fifth, use the imperative mood in the subject line. Sixth, wrap the body to about 72 characters. Remember, this is the body, not the subject line. And seventh, use the body to explain the what and why versus the how. This one ties in a little bit with the first rule, if your commit needs an explanation on what is going on, the body is used to explain the what and the why. Remember, when you're writing a commit message, it's important to keep in mind that you're writing the message for other people to read and understand what is going on. The better the explanation, the easier it will be to navigate through the git log,

and the more transparent the log will be. So, let's jump back over to our command line and run git status again.

Remove and Move Files

At this point, we don't have anything yet to commit and our working tree is clean. Now, Steve has a bit of a problem, one of Steve's recipes, eggnog latte, has been getting some negative reviews and is the least-ordered drink on his menu. Now Steve wants to completely get rid of this recipe because no one is ordering it, and he's convinced he won't ever want to bring it back. So completely removing a file from his Git project is something that needs special consideration, especially if we've modified the file in our project and its file versioning is being tracked by Git. Nonetheless, Steve wants it gone, so let's type ls in the command line to bring up the file. Now that we've verified that the eggnog_latte file is present and currently being tracked by Git, Steve wants it removed from his project. So to remove a file in our project, we need to tell Git to stop tracking the file. We can do this by using the git rm command. Since Steve wants to remove the eggnog_latte recipe, let's type git rm eggnog_latte, and then hit Enter. We see that Git responded with rm, and then our file, 'eggnog_latte'. This is confirmation that we have staged this file's removal, and Git has stopped tracking the file. We can then type git status to see our project's current state. We see that our eggnog_latte file is now showing as a deleted file in our staging area, and the next time Steve commits this file will no longer be tracked by Git and it will be removed from our project. We can also type ls to see the updated list of files under our project. Notice that the eggnog_latte file is no longer showing in our list. This is exactly what Steve wanted, but he also wants to untrack the irish_coffee file from Git, but not remove the file from the project. Like we did with the eggnog_latte file, he wants to keep the file around, but not have Git track it for now. We can do this by typing git rm --cached, and then the file name, irish_coffee. We see a similar git response saying that the irish_coffee recipe was removed, but let's type git status to get a better picture of what was changed. We now see that the irish_coffee file is under the untracked files. We can also confirm this by typing the ls command to see the list of our files. Notice that the irish_coffee file is still present. Now on a similar note, Steve wants to rename his README.md file, but he isn't sure how to do it by using Git. Now, we can do this by typing git mv instead of rm, and then the filename that exists, followed by the desired filename. So in Steve's

case, he wants the README.md file renamed to just README. We can change this by typing git mv README.md, space, README. If we now run git status, we see that in our staging area the file is now showing as renamed and ready to be committed into our next commit snapshot.

Introduction to Branches

At this point, we have some files in our staging area waiting to be committed into our next snapshot, but Steve has some new recipe ideas he wants to mess around with, ones that he doesn't know if he wants to incorporate yet into his main list of recipes, but he wants Git to track the progress of these changes. Now, Steve can make these changes right on his master branch, and Git would track those, but his staging area and working directory will get messy with all sorts of different changes. That can get pretty confusing. This is where we introduce one of the best things about Git, branches. And a really great way to see the power of branches is to visit a visualizing tool created by some developers at GitHub. We can open this URL using our command line by typing open http://git-school.github.io/visualizing-git. So let's jump over to this URL to see the power of branches in action. Here we see a green dot that represents our project's master branch. We can see this by the labels given in this visual. In the bottom left text, it says Enter git command. We can type some git commands here to visualize how branching works. Now in Steve's project we've made some commits onto the master branch, so let's add some commits to the visual by typing git commit a few times. We see that our HEAD, which is where we are currently pointing at on our project, is moving along with master. This makes sense. We haven't done anything yet to divert any work that is being done, and this is what Steve has been doing on his project. Everything has been done on the master branch up to this point. Now, a branch in Git is simply a lightweight movable pointer to a commit made in a project. A commit needs to be made on a branch, but you have full flexibility on telling Git what branch to make that commit on. Let's create a new branch off of the master branch. We can do this by typing git checkout -b, and then the name of our branch. For now, let's just call it new_branch. Notice now that a new pointer has been created called new_branch. Let's dissect the command we just ran. The git checkout command let's you check out to a different branch, and the -b option is telling Git that we want to create a new branch and then check out to it. Notice that our new branch, master, and HEAD all point to the same

commit in our project, but that our HEAD pointer is now under new_branch, and no longer directly under master. This is because we have checked out to our new branch, and we are no longer in our master branch. HEAD is just a pointer to our current state on the branch we are on. Let's now make a couple of commits on our new_branch. Notice that master stays behind, and our new_branch moves along with the commits. We are making commits on the new_branch, and no longer on master, and we can check out back to our master branch by typing git checkout master. Notice that HEAD is now pointing back to the master branch because we are now checked out to the master branch. Notice that the other commits stay on our new_branch, even if we have checked out to a different branch. If you commit on a branch, those commits, unless explicitly changed, are linked to that branch. If we now want to make changes to the master branch, those changes will only be on the master branch; they will not be included in the new_branch. So let's make a couple of commits on master and see what happens. Notice how the work on our project has now diverted. The direction on our master branch is now slightly different than the work done on our new_branch. If we check out back to our new branch, we now see that our HEAD has moved back to that branch. If we make another commit, the work continues on that branch, and that branch alone. The changes made on our new_branch won't affect anything on master. Branching off of master allows us to create a parallel environment where we can try out new ideas and make changes without affecting the master branch, and this is huge when trying out new ideas or working with multiple collaborators. Now, this only scratches the surface of what we can do with branching in Git. Branching is a key concept when using Git, especially with working with multiple collaborators. Now that we have the basic understanding on branching, let's jump back over to our Terminal window and create a branch off of master similar to what we did here, and then make some commits on our new branch.

Working with Branches

To create a branch off of master, let's type git checkout -b new_branch. Remember, this creates our new branch and checks out to it at the same time. If we just wanted to create a branch, but not check out to it at that moment, we can just type git branch, and then the name of our branch. Let's now run git status. We see our renamed file and deleted eggnog_latte file as changes to be committed. Now, we made these changes on our

master branch, why are they showing up in our new_branch? In the Git visualize tool that we just used, it showed us that the commit stayed on the branch that you made them. Well, this is correct. When we removed and renamed these files on our master branch, they remained in our staging area. We didn't commit these changes yet to a branch, and since we created a new branch off of master at this point in time, the changes in the staging area remained. So let's commit these changes to our new_branch. If you're ever unsure which branch you are checked out to, running git status will let you know. Here we see that Git tells us we are on branch new_branch. Since these changes are already in our staging area, we don't need to add them to be committed by running git add. Let's commit them as we've done before. We can type git commit -m, and then our commit message, rename README.md and remove eggnog_latte. Git now responds back with information on what was committed. We see that the README.md we renamed, and the eggnog_latte file was deleted. Running git status lets us see that we now have nothing to commit. But wait, we still have that irish_coffee file as an untracked file. But, remember, this is all only done on our new_branch. These changes were not committed to our master branch. If we run the ls command, we see that our changes on this branch are present, however, if we check out back to our master branch by typing git checkout master, we see an error saying, The following untracked working tree files could be overwritten by checkout. Please move or remove them before checkout. Now, since we untracked our file with git rm --cached, we deleted the version control cache but did not remove or delete it from our file system. This is why it's still staying in an untracked state. When we try to check out to master, or a different branch, Git is telling us that we need to do something with this file. Now, we can remove it, or we can add it back in to be tracked. But Steve doesn't quite want to get rid of this file, but he doesn't want to add it right now to be included in his local project, so Git provides us with an option to stash this change. We can do this by adding the file back in by typing git add, and then dot, and then using the stash command by typing git stash. We see that Git responded by saving our working directory in a Work in progress state on our branch. We can use git stash to record a current state of the working directory and staging area but want to go back to a clean working directory. Now that this change has been stashed, we can type git status and see that our branch is clean, and nothing to commit, and nothing in our working directory. So where did this change go? Now, if we type git stash list, Git provides us

with a list of work in progress changes that we've stashed, along with the branch and some commit data. We can also type git stash show to see a better detailed view of what file was stashed, and a high-level overview of the changes. Now, if Steve wanted to get this stage out of stash, he simply needs to run git stash pop, and the change would be dropped back into his working directory. But since we don't want to do that right now, we'll just leave it in our stash for the moment. Let's now check out back to our master branch with git checkout master, and then run the ls command again. We see that the README.md file is not renamed, and the eggnog_latte file is still present. Now that we're on our master branch, let's create a new file named cold_brew by typing touch cold_brew, and then we can open it up in our Atom text editor by typing atom space dot. In our cold_brew file, let's add some content text, Placeholder for adding a new cold brew recipe. Let's save the file and return back to our Terminal window. Let's now run git add cold_brew, and commit our changes by typing git commit -m, and then our commit message, add cold brew placeholder file. Now running git status shows us that our local master branch is ahead of origin/master by 1 commit, and this is true because our other commit was done on our new_branch, and not master. Let's return back to our Git visualization tool to see what we've done.

Merge

Let's clear out our last example and recreate what we just did with Steve's project. We can do this by typing clear in the text command prompt, and clicking OK to clear out what we've done. We're back on our master branch, so let's add a few commits. We then created a new branch by typing git checkout -b new_branch. Remember, we've just created a new_branch pointer from master. Nothing has changed yet as far as content in the project. We then made a commit on our new_branch by removing and renaming some files. So let's make a commit on our new branch to represent that change. We then returned back to our master branch with git checkout master and created a new cold_brew file with some placeholder content. So let's represent this change on master with a commit. Notice that our project commit history has diverted. This is perfectly okay. Remember, branching lets you work on content without affecting the master branch, sort of like working in a test sandbox environment. And this is where we left off in our Terminal command prompt. But what if we now want to incorporate the changes from our new_branch

into master? Well, this is a pretty common step. It wouldn't make much sense to create a separate branch if we never intended to incorporate those changes back into our production-ready master branch. Merging a branch into another branch is a key concept with branching, and really the purpose of creating a branch. Since we want to merge our changes into the master branch, and we're already checked out to master, we can type git merge, and then the name of the branch we want to merge into. So in our case we want to merge new_branch into the master branch, so let's do this. We now see that our HEAD is still on master, but a new merge commit was created that is pointing to both branch paths. Our local master branch now has the changes that were made in our new_branch. This is pretty cool. Let's now jump over to our Terminal window to see this in action. Now, we're currently already checked out to master, so let's merge our new_branch into master by typing git merge new_branch. This brings up a tool called Vim, where we need to add our merge commit message similar to a regular commit. Now, we won't be going over Vim in detail, but we can add our message by typing i to enter the interactive mode, and then writing our merge commit message, which in our case we'll just leave it as it already is. And then when done, hitting the escape key followed by :w, and then q. The w saves our changes, and the q quits Vim. We can see that the changes from the new_branch have now been added to our master branch. If we run git status, we now see that our local master branch is ahead of origin/master by 3 commits. One commit was on our new_branch, another was a direct commit on mater, and the last commit was the merge commit to merge the new_branch into the master branch.

Reset

Now, at this point, Steve could push his changes up to the origin/master on GitHub, and it would include these three commits. His origin/master and local master would then be in sync again with each other. But before we do so, what if we didn't want one of these commits, or all of these commits anymore. Now that we've merged these commits into our local master, how can we reset some or all of these already committed changes? Or change the history a bit? Well, this is a common enough situation that Git has a command just for this, and it's called git reset. Git reset allows us to move commits from history back into our working or staging area. It can also be used to throw changes away. Now this can be a destructive command, so careful consideration is highly recommended when using reset. Since we're

changing our Git history with reset, it's best practice not to reset changes once we've pushed our commits to our remote origin where other collaborators could have accessed the changes. Now, there are three different options when using git reset that will decide what happens to the commits that you want to alter. The first is called git reset --soft. This reset command will move the specific commit or commits back into your staging area. This is helpful if you want to regroup these changes into different commits, or add more changes before committing. The second option is git reset --mixed, or just git reset. This is the default reset option if something else isn't already specified. This reset command is similar to --soft, but instead of moving the changes to the staging area it moves the changes back to the working directory. And the last option is git reset --hard. Now, this option moves the changes to the trash. This could be helpful if you've messed around with making changes, and then you decided you didn't want to keep anything and just start over. But be careful when using this command. If you already have changes in your staging or working directory, and you decide to run git reset --hard, those changes will also be deleted. So let's jump back over to our Terminal and see how this would work. Now, let's run clear to bring our Terminal back to the top. If we run git log --oneline, we see a list of commits that we've made in our project. We can see where our local HEAD is pointing and where our origin/master is pointing. Now this makes sense, since our local master is ahead of origin by 3 commits. Our most recent commit is the merge commit when we merged the new_branch into our master branch. The next commit was on our master branch where we created the new cold_brew file, and the commit before that was made on our new_branch where we committed the changes to rename the README file and remove the eggnog_latte file. The following commit is where origin/master is currently pointing to. So let's reset our master branch back to where our origin/master is pointing. Let's use git reset --mixed or just git reset, to move these changes back into our working directory. We can do this by typing git reset --mixed, and then the commit that we want to move our HEAD to, which again, is the one where the origin/master is pointing to. Let's now run git status to see what happened. We see that our changes from master, as well as the changes from our new_branch, are now all showing in our working directory. And we can add these changes by typing git add, space dot. And if we now run git status again, we see that all changes made, ones originally made on our master branch, as well as the changes on our new_branch, are now all

together, ready to be committed in our staging area. Let's now commit these changes and add a commit message. Now if we run git status, it now shows us that our master branch is now only 1 commit ahead of origin/master. Remember, the changes are all there when our local master was 3 commits ahead of origin/master, we've just changed our commit history to group these commits together into one commit. So let's run clear to bring our command prompt back up to the top.

Pull from Remote Origin

Let's now push our local master branch to origin/master by typing git push origin master. Our origin master is now up-to-date with the changes we've made locally, nice. Now that Steve has made the changes he wants, and has pushed them up to origin/master on GitHub, the next thing he wants to do is to pull down those changes to his other computers so that he can work on his project using his work and personal computers. We can do this by returning back to Steve's GitHub profile where Steve's repository is located. If we navigate to the Repositories tab, and click on Steve's wired-brain-recipes project, we see a list of files and content inside of the project. In order for Steve to clone this repository down to another computer, he can click on the green Clone or download drop-down menu button located toward the top right of the page. We can copy this HTTPS link by clicking on the copied clipboard icon next to the web URL. Now that we've copied the URL, we can open up the command prompt to clone the project down. So let's jump back over one last time to our command prompt. Now, Steve can clone down his project to his other computers simply by navigating to the directory he wants to store the project in, and then cloning the project into that directory. So let's navigate out of our café_recipes folder by typing cd .. to go back one directory. Steve can now clone down his repository by typing git clone, and then the URL that we copied. We see that Git has cloned the wired-brain-recipes project into the same directory name. So let's cd into that new directory by typing cd wired-brain-recipes. We can now type git status to see that we are up to date with the origin/master, and then typing the ls command shows us the full list of files in our project that were pulled down from origin/master. Now Steve can follow these same simple steps to clone down his project to his other computers that he needs to to work on this project.

Summary

In summary, we've learned how to check our commit history by using git log, and the different options we can use with git diff. We also learned how to remove and move files around in our project by using git rm and git mv. We then jumped into the core concept of branches and learned how using branches can be useful in a workflow. We learned how to create a branch by using git checkout -b to check out to our newly created branch, or just git branch to create a new branch. We learned how to move around to different branches with checkout, and then learned how to merge branches together to incorporate the changes made on one branch into another branch with git merge. We learned how to reset committed changes with git reset --soft, git reset --mixed, and git reset --hard. And lastly, we learned how to clone down a repository from GitHub so that we can work on our project from multiple computers by using git clone and then the URL of our repository found on GitHub. Now, I am so excited on the possibilities of what lies ahead. Now that you have a basic understanding of Git and how it works with GitHub, there is so much more to learn when using Git and GitHub. We have just scratched the surface of the power of Git, but now you have the fundamentals learned to get you started on learning even more. Thank you for taking the time to watch this course. I had a great time creating it, and I hope you learned a lot, and are as excited as I am to continue your Git and GitHub journey. Have a great day and thank you for watching.