## Configuring Git

### Overview

Welcome to the course, Advanced Git Techniques. I'm excited that you're here to learn even more about Git and how you can integrate it into your workflow. Initially, we're going to be talking here about configuring Git and utilizing the power of Git's configuration system. Now let's talk first about a fictional company, Globomantics, because they too are looking to use Git at a deeper level. So first of all, what they're looking to do is configure NuGet installs for their developers. They want to be able to follow a set of best practices and they want to be sure that everyone is set up for success with that. They also want to enable code reuse through common repositories. So common code they use across multiple projects, they want to be sure they're using Git in the proper way to pull that into the new projects that they create. They also are developing a local workflow for running tests and linting per commit, so they catch those things before they even end up on the build server. They also want to enforce a commit message template on the server so that every commit that developers make references a ticket within their ticketing system. And ultimately, they want to enable their developers to work more efficiently with Git. So here in this initial module, here's what we're going to cover to work towards those goals. First of all, we're going to be reviewing the structure of Git's configuration files. Then, we'll be modifying Git configuration for common tasks. Then, we'll be reviewing Git attributes and their use, we'll be implementing a clean and smudge filter within Git, which will leverage Git attributes.

### Git Configuration Structure

So the first step in understanding how to configure Git is understanding its configuration structure. So first of all, let's talk about Git configuration. So Git does provide a robust configuration system, and you can configure both standard and custom configuration values across multiple levels of your installation. Let's first talk about those levels and let's give an example. So to begin with, on your machine, you might have a repository, let's say you actually have multiple repositories, but ultimately those repositories are under your user account on that machine. And on the exact same machine, you could have another user account that has other repositories in it, and then over all of that, you have the actual Git installation on that computer. Now let's see how that ties into the Git configuration levels. So first of all, we have three different levels primarily. First, we have the repository, which is known as the local level. Then we have the user account, or global level. And then finally, we have the Git installation, or system level. And configuration values are evaluated from the repository up, so Git config will use what is in the repository, and if it doesn't find a value there it will bubble up to the global setting, and if it doesn't find something there, it will ultimately bubble up to the system level setting. Now let's talk about the Git config locations. Now I'm using an example here from my machine, which is a Mac, and in this case you can see here where the repository Git config is found. It is actually in the git directory of your repository, and then for global it's under my usr in .gitconfig, and then for the system it is actually under usr/local/etc, and then gitconfig. Now all this said, even on a Mac, it is going to be dependent on the method you chose to install Git on your machine. Now let's look at a sample Git config file. So here within a Git config file, you'll see that you have sections and in some cases you even have a qualifier on a section. But, for example, will look first at the user section and this contains two keys being name and email and then the values for those specific settings. And so this holds true for every use of Git config that you will see whether you're talking about local, global or system level configuration. So let's look at how you set a Git config value. So you would actually specify within each of these the qualifier on what level you want to save it as, and in all three cases here I'm choosing to set the user, which is the section, and then the name, which is the actual key for the setting, followed by the value, which in this case is David Tucker. So I'm first setting the local value. This would be the value that is set within the repository. Then I'm setting the global value, and then I'm sitting here the system value. Now you wouldn't need to set all three if you simply set this at the system level or at the global level, it would evaluate no matter what repository you're using. However, if you ever wanted to have a different name within a different repository, you could choose to set that at the local level. Let's talk about common configuration settings that you would configure for Git on your machine. First of all, we have identity settings, and that's what we just saw. This is where you would configure your name and email address. Then you have the file editor. So maybe you're a total command line pro and you want to use Vim or Emacs for your primary editor, or maybe for you you're very familiar with VS Code or Sublime Text and you want to use those for your editor. Those are some examples of things you can configure. And the same holds true for the merge tool. Also, you can configure terminal color output, whether you want to include it or not, and even customize the colors that it uses. Also, command aliases. So if there are certain commands that you perform over and over again, you can create aliases so that you can execute those commands more easily. Next, let's look at how we view our config values. So from any point on your machine, you can run git config ‑‑list ‑‑show‑origin, and this will show you all config values and also it will show you in which file they are defined. So if you run this command within a repository, it will show you those that are configured for the repository, as well as for the global value for your user and for the system value. So at any point, if you want to see what Git config is evaluating for a specific setting, you can simply run git config and then pass in the actual setting that you want to evaluate, and it will return what the value is for you at that point. Now let's look next at how we remove Git config values because there are times that maybe we've entered in a customization at our repository level that we want to remove, or maybe there's a setting that we've set globally that we don't want to have set globally. So at any point you can run git config, pass in the level, and then say ‑‑unset and then tell it what you want it to unset, in this case user.name. You can also edit the config files directly, so without even having to do it through the command line. You can simply run git config ‑‑global and ‑‑edit, and it will launch this in your default file editor. You also can remove an entire section of config by simply running git config ‑‑global and then ‑‑remove‑section, and then pass in the section that you want to remove. Now it's important to note here that Git config is not the only configuration that changes how Git works. In addition to Git's configuration, there are environment variables that also affect how Git works. However, in most cases, these are more advanced use cases that you would be modifying environment variables for. Your first step should always be to leverage Git config to modify the default behavior.

### Configuring a Git Installation

So next, we're going to be configuring a Git installation. So let me tell you what we'll be configuring over the course of this demo. First of all, we will need to review the current configuration settings so we know where we're starting from. Then, we'll be configuring a first time Git installation for use. So we'll look at it just as if you were starting from scratch. Then, we'll be configuring the editor, diff, and merge tools, and then we'll be utilizing local config for different identity settings to demonstrate the different configuration levels. So I realized that for many of you, one of the first things you did when you installed Git was to set your identity values. We're going to walk through that again here and look specifically at how we're setting those values. So the first up here is we're going to set the name. Now in this case if you notice, we're using the global level, this means it will apply to any repository for the current user. You'll also notice here that we're setting the user category and the name value. Now next, we need to set the email address. In this case, we'll say that I am one of the developers at Globomantics. So we'll say in this case that for any repository that I have under this user account that I'm going to be using my globomantics.com email address. However, we also will later on in this clip walk through the process of how we would customize this for a specific repository. Now that we have that added in, the next thing we need to do is we need to create a sample repository that we can work in. Now that we've added in a sample repository, we can now review all configuration. So we're going to run the git config ‑‑list command and pass in ‑‑show‑origin. So with this, we're now able to see every current Git config value that is being evaluated, all the way from the system level, which we see at the top, then through the global level, and then after the global level, we see the specific repository settings that are included. And we can see here that our identity settings have been set globally within the Users/davidtucker/.gitconfig file. And let's say that I'm not going to be editing this as david@globomantics.com, let's say this specific repository is a personal project that I'm going to be working on. I can choose here to update the email address specifically for this repository, and in this case I've set this to be the local scope. But if we look here at our configuration list, we can see that we are currently evaluating this locally within the repository. We can also see that if we run git config user.email. Now the next thing we need to do is we need to configure the standard tools that we want to leverage for our editor, for our diff tool, and for our merge tool, and this will heavily rely on Git config to make these settings work. Now I'm going to choose to leverage VS Code for all of these. So the first step to making this work is that I need to set globally the setting for core.editor, and in this case I'm going to use VS Code, which is just code from the command line, and I'm going to specify that it would open a new window each time this is executed and that it waits to return until the editor is closed. So in this case, we'll go ahead and enter this configuration value, and then we can actually test that it works by editing our global config file. We can see here now that VS Code has launched, and it's launched with our configuration file. And this global configuration file includes the identity settings that we set previously. If we need to, we can make any changes to this and save it and this will then be reflected in our global config. So in this case, we're going to choose to set the diff tool that we want to leverage. So I'm going to paste in a value here that will allow us to use VS Code for that as well. So now that I've pasted this in, we're going to save the file and then we'll close the editor. Now the next step is for us to test out our diff tool. In this case, we'll make a change to our README file and then we'll open our diff tool. We'll see here that it does open up VS Code. On the left we have the original value, on the right we have the modified value for this file, and we can see that it is properly showing us the diff between the two. Now I'll close the editor. Now the next step is for us to actually set the merge viewer. So in this case we'll go back and edit our global config file. We'll paste in the value for our merge config, and then we'll save our config file. We'll then close the editor. Now next, we're going to walk through steps so that we can actually create a merge conflict. And now that we've ended up in a situation where we have a merge conflict, we can now run git mergetool. We'll see that that now opens VS Code and we are able to in this case accept either the current change, incoming change or both changes. In this case, we'll accept the incoming change. We can then save the file, and then we can close the editor. Next, we can commit, and in this case I'm purposely not going to pass in the message through the command line. By simply hitting git commit, we'll see the editor come up. This allows us to edit our commit message from within VS Code because it's set as our default editor. I'll save the file and close. And now we can see we've solved our merge utilizing our merge tool. Now, the next step we can take in making our use of Git even more efficient is we can set aliases inside of Git config. So one of the commands that I use a good deal in Git is the status command. So in this case, instead of having to type git status every time I want to get the status, I simply want to be able to type git st and have it return the status. We can accomplish this utilizing config values. Now here I am setting an alias, and we can see here that it is an alias because it is included within the alias category, and in this case the value that we pass in is the shortened version of the command, which is st, and then we can see the full version of the command after status. Now simply by typing git st, we're able to get our status command back. We can also build on top of our aliases. So another command that I leverage, and to be honest I actually leverage it more than git status, is git status and then by passing in the short argument. So here, by utilizing git config, we can pass in an alias for ss, or short status, and in this case we can pass in st, which was the alias that we just defined previously and then pass in ‑‑short. Now we can utilize the short status, which in this case returns nothing because we don't have any modified files. And the more that you work with git config, you'll want to actually create an alias for how you edit your global, your local, and your system config. So in this case, we'll utilize an alias for egc, which stands for edit global config, and this will call git config, it'll pass in the global option, and then ‑e to edit it within our editor. Now we can simply type in git egc. Now this opens up the config file, and we can see at the bottom of the config file where our alias values have been set. Now here we can choose to pass in even more aliases that will allow us to work more efficiently within the system. So I've pasted in options here for editing the local config, editing the system config, check out branch, commit, and amend. We can now save this file and close the editor. Now from this point forward, I will not be using the aliases in this course. This is simply because if someone doesn't watch this specific clip, I don't want them to be lost on what commands I'm executing. So while in this course, I'll be using the full versions of the commands. I encourage each of you to figure out the aliases that can make you most efficient when you're leverage Git on the projects you're working on.

### Git Attributes

So now that we've talked about Git configuration and how we can use it to change how Git functions, we're now going to talk about a different aspect of configuration, which enables us to configure how Git deals with specific files, and that's called Git attributes. Now there's two different ways that Git really can handle dealing with files differently. The first is .gitignore, and this is what allows us to determine files in which Git will simply ignore and not include it within the repository. But then we also have .gitattributes, and this allows us to set specific configuration values for how specific files or groups of files are handled. So at a high level, Git attributes are placed on a file or a collection of files within a repository, and it dictates how this file is handled. Now, in some cases, different Git hosts may offer specific functionality that leverages these attributes. So let's talk about some common use cases for Git attributes. First of all, you can configure line endings in files with Git attributes. You also can specify files that are binary versus files that are not. You also can enable large binary file support with Git LFS. You also can exclude files from the exported version of a repository, and you can specify clean and smudge filters, which we'll talk more about in just a minute. So let's look at a sample .gitattributes file. So here is an example file from within a repository, and you can see first we are setting it to Auto‑detect the line endings for specific files. You can also see here that we are setting this specific file, such a JPEG files and PNG files are binary files. Now as a note, in most cases, you don't have to specify this yourself. Git does a great job detecting and determining what is binary. However, if you ever run into a situation where it is not handling it properly, you can choose to set that specifically within the .gitattributes file. You can also see an example here of using Git LFS for storing zip files within a repository, and this again is dependent on your host support for Git LFS. And we also have an example here where we're using the export‑ignore attribute to set that value for our test directory so it is not included within the exported version of the repository. But let's talk next about a more complicated example, utilizing something called Git clean and smudge filters. So, first of all, within any project, you're going to have a staging process when you have files that are ready to be committed. You're going to have your working directory, which are the files that you're choosing to work with, and then you're going to have this staging area, so once things are ready to push into the repository. Now we can choose to define, inside of our .gitattributes file, a filter. And in this case, we're choosing to specify a filter just on the JavaScript files within our repository. And there are two functions for this filter. One is smudge and one is clean. So let's say, for example, we have an API key that is present within our JavaScript files, and we want to remove that so it's not stored in our repository. We can choose in this case to use the clean and smudge filter to make that happen. So let's look at what would happen to these files. First of all, because this is a JavaScript filter, our README file would be passed through without going through the filter. However, our JavaScript files would pass into the clean filter, and this function would remove references to our specific API key. And then we would see it go from the clean into the staging area. However, this is just the staging process. We would need to have the reverse of this happen in the checkout process, and that's what the smudge filter will do. So first, within our checkout process, we would see the README file continue to pass through, just as it did before. So next, we would see our JavaScript files moved through the smudge filter, and this would take the default placeholder that we put in place and then replace it with the specific API key. And we would then see the proper values reflected within our working directory. And this is just another example of what can be accomplished by utilizing Git attributes. So next, we're going to walk through the process of implementing specific Git attributes within a Git repository, and using a clean and smudge filter is one of the attributes that we'll include.

### Using Attributes in Git

So next, we're going to walk through the process of actually utilizing attributes within Git. So over the course of this demo, here's what we're going to do. First, we're going to be adding in a gitattributes file for a new repository, and then we'll be implementing exif diffs for image files. Then, we'll be excluding files from release packages on GitHub. And then we'll be adding a clean and smudge filter to hide secret values within a repository. So let's dive in. So first, as a prerequisite, we're going to be leveraging the exif tool from the command line to help perform diff files on our images. Now for this to work, you will need to have the exif tool installed on your machine. You can look in the exercise files that are included within this module to get instructions on how to install that on your machine. So first, we're going to create a repository that we can work in. Once we create a repository, we're now going to populate it with both a README file and an index.js file. And next, we need to add in our gitattributes file. Next, we'll go in and edit our gitattributes file. And so next, we're going to define a Git attribute for JPEG files within our repository, telling them to use the exif strategy for performing diffs. We'll now go ahead and save our file. Next, we need to define how we're going to leverage the exif tool from within our Git config, and we're going to choose to set this globally. So next, by using this command, we're able to now specify how Git config is going to utilize the exif strategy and what tool it is going to be leveraging, which in this case will be the exif tool. We're now going to add these to our repository and commit. I'm now going to open the exercise files for this module. I'm going to choose to go into the original image folder and I'm going to copy 1.jpg, and then we'll move this over into our repository. Now that I've moved that over, I'm now going to add that into the repository. Now that I've committed that, I'm going to open up again the exercise files for this module. And from within the exercise files, I'm now going to navigate to modified\_image an d I'm going to copy this version of 1.jpg into our repository. And we'll choose to replace the existing image. From here, we can now run git diff. When we do, we'll notice that we're able to see the diff between the exif on the old version of the file and the new version of the file. And in this case, if we continue to scroll down, so in this case among other things you'll notice that the image width and height is different between the new and old versions of the file. The new version of the file is an optimized version of the image that is smaller, both in terms of size and file size. We can now close out of the diff. So next, I'm going to open up the exercise files for this module again. I'm going to take the test directory that is included, and I'm going to copy it. I'm now going to paste that within the repository. Now what we want to do with this is we want to configure gitattributes so that within GitHub, our test directory is not included within the exported files. So to make this work, we first need to include our test directory. So we'll now navigate back to the terminal and we'll add this to the repository. Next, we'll commit it. Now the next step is that we need to add in our origin. I've created a repository on GitHub specifically for this, so I'm going to choose to add this in. Now that we've added in the remote, I'm going to push the master branch to origin. Now that we've successfully pushed, I'm going to add a tag in at this point. I'm going to follow semantic versioning standards, and we'll call this 0.0.1. Now we'll push that specific tag to origin. Next, I'm going to navigate to my repository on GitHub. We can see now that GitHub has been populated with the files that we've added into our repository. I'm going to choose to select the Releases option. From here, we can see our release for 0.0.1. I'm going to next choose to download the ZIP file. And here we can see within the export for our tag, 0.0.1, all of the files that are in our repository, including our gitattributes file, are included within the downloaded ZIP for the release. So next, let's leverage Git attributes to change that. I'll navigate back over to the terminal. Next, we'll actually navigate to our gitattributes file. From within our gitattributes file, we're going to add in two new lines. So the first line that we added puts an attribute of export‑ignore on anything that begins with a dot in the repository, which would include our gitattributes file. And next, I've added in the same attribute on the test directory. So if this works properly, once we push this tag to origin with the updated commit, we should see that the release would not include those files. So let's go ahead and save our gitattributes file. So next will commit these changes. Then we'll add a tag and then we'll push the tack. Now that that tag has been pushed, we will now navigate over to GitHub, and we can see now on the Releases page we now have a release for 0.0.2. I'll click on the ZIP file, and now we can see that the files for this specific release on GitHub only include the files that we wanted to. We have excluded the gitattributes file along with the test directory. And so next, we'll navigate back over to the terminal. And so next we're going to be implementing our clean and smudge filter as we discussed in the previous clip. So our first step here will be to edit our index.js file. So from within this file, we want to include some sample code, and this sample code won't do anything, but it'll simply be an example of what we could implement with real code. So in this case, we have a pretend API key. You can see it listed here. And we're going to say that to make some API call we need to pass in this API key. But here's the issue. We don't want this secret to end up within our repository. So this becomes a good use case for leveraging a clean and smudge filter. We'll go ahead and write our index.js file. So just as when we were creating the exif strategy for how we would handle diffs for images, we define the specifics of that within Git config, and then we simply told specific files to use it within our gitattributes file, and the same will be true for the clean and smudge filter. Now in this case, we're going to use sed to simply replace one value and put in a placeholder. So our real API key will be replaced by that placeholder. So let's go ahead and implement the clean and smudge filters within the local level of Git config. And so now that we've added both of those filters in, both the clean and smudge filter, we're now ready to utilize Git attributes to specify what files it needs to act on. And so now we've specified this so that the clean and smudge filter will work with each JavaScript file within the repository. When we go through the staging process, we'll see it run the clean filter, and then when we see it run the checkout process, it will run the smudge filter. Now that we have this in place, let's save our gitattributes file. Next, we'll commit our gitattributes file. Next, we're going to push this to master. Then, we'll navigate over to GitHub. So from here within our repository, we can now navigate to the index.js file. And when we navigate to the file, we can see that our actual API key has been replaced by the placeholder for SECURE\_API\_KEY. So by using the clean and smudge filter, we're able to have the real value locally while having a different value on the remote version of the repository. So over the course of this demo, we've been able to utilize Git attributes for several things. First of all, we have configured images to use the exif tool to perform diffs. Then, in addition to that, we were able to exclude specific files from the exported version of our repository using the export‑ignore attribute. Then, we were able to use a clean and smudge filter to help protect secret values within our repository from ending up on a remote.

### Summary

So over the course of this module, we have looked at several different ways that we can configure Git. Now let's look specifically at what we've been able to cover. First of all, we reviewed the structure of Git's configuration files. We examined the three different levels that Git config works within. Initially we have the local level, which handles the repository that you're in. Then we looked at the global level, which is what is for the specific user account that you're using. And ultimately we looked at the system level, which covers the specific Git installation that you're using. And then we modified Git configuration for common tasks. First, for things like our identity settings, our name and our email, but then we went deeper and looked at how we set our merge tool and our diff tool to be VS Code. And then we also looked at how we can use aliases to make ourselves more efficient by taking common Git commands that we leverage and creating a shortened version of them. And then we reviewed Git attributes and their use. We talked about how you can leverage Git attributes for things like setting line endings on files. And then we also talked about how you can use it to perform things like using a different diff strategy, which we use the exif tool to perform diffs on two different versions of an image. And we also used Git attributes to specify within GitHub what should and should not be included within the exported version of a tag. And then, we utilized both Git config and Git attributes to implement a clean and smudge filter within Git. We utilized sed to replace a sensitive API key and put a placeholder in place before it ended up in our remote repository. And then when we checked it back out, it was able to then replace that value with the real API key.

## Utilizing Git Submodules

### Overview

So in this module of the course, we're going be walking through how you leverage Git submodules. Let's take a minute and review what we're trying to solve for Globomantics. First of all, they are looking to configure new Git installs for their developers, and we covered several aspects of configuration in a previous module. But they're also looking to enable code reuse through common repositories, and we're going to be utilizing Git submodules to make that happen within this specific module. They're also looking to develop a local workflow for running tests and linting per commit, as well as enforcing a commit message template on the server. They're also looking to enable developers to work more efficiently. So over this module, let's talk about what we're going to accomplish. First of all, we're going to be reviewing submodules and their support within Git. Then, we'll be configuring Git to better use submodules. We'll be adding a submodule to a project, removing a submodule from a project, working with nested submodules, and reviewing best practices for utilizing submodules as a team.

### Understanding Git Submodules

So next, we're going to tackle the process of understanding Git submodules and what they provide. Let's first talk about different ways that we can utilize code from other Git repositories. And remember, our goal here is that we can have shared repositories that can be used across projects. The first approach is you can leverage a package manager, one that's specific for the technology that you're working in. This could be a solution, like npm or NuGet or RubyGems. The next approach is utilizing something called Git subtrees. Now this is an emerging approach, and it does leverage a command git subtree. However, we also have Git submodules, and Git submodules are the most common approach for integrating other repositories, and it is the approach that's been around the longest, and so we're going to choose to focus on Git submodules. So for a quick definition, a Git submodule is just a construct within Git that enables you to keep a separate Git repository as a subdirectory within an existing repository, so this enables you to effectively keep two separate repositories linked together within a project. So let's see how this plays out. So here we have our normal Git repository, and within this, we're going to have our working copy, and all of this exists within our repository. But then, we can look to integrate a separate repository. In this case, we have Submodule A, and this is held under a directory external/submoduleA. And then we could have another one even, Submodule B, and this one would be an external/submoduleB. But then it goes even deeper than this because you can also have nesting. So we could even potentially have two separate submodules that exist under Submodule A. So with this being the case, let's talk through a few facts about Git submodules. First of all, utilizing submodules does add complexity, especially when you're working with a team, so you want to be sure that you understand the guidelines that you need to follow, and we'll be covering those in more detail later on within this module. Next, package managers should be leveraged, if possible. Package managers have scripted many aspects of taking specific versions of a Git repository, making them available in an easy way to include within one of your projects. There will be much more additional work if you choose to go the submodule route over the package manager route. However, submodules are the fully supported way to include other repos into your project. And submodules, as mentioned, are the most common manner to include other repositories, and you'll see this approach leveraged in many open source projects. So let's next look at how you add a submodule. So Git does have a submodule subcommand that you can leverage, and in this case, we're going to pass in the add command to that and then pass in the repo that we want to add, followed then by the directory that we want to live in within our current repository. Now there is a file that gets created the first time we add a submodule, which is our .gitmodules file. Now this file will be checked into our remote, and this is what allows us to then share this with other members of our team that are going to be leveraging the same Git submodules because in this case there is a process with submodules where it will take this information and then integrate it in with the config for your repository. Now you'll notice this looks much like the configuration files that we covered in a previous module, and that is very true. Much of this will get integrated in within your git configuration for your specific repository. But for each submodule, two pieces of data are included. First, the path within the current repository and the URL for the remote that we're pulling it from. Let's look next and how you clone a repository that has submodules in it. So if you pull down a repository that has submodules, you can utilize clone just as you have before, but next, you'll need to init the submodule. And by doing that, you'll then integrate that in with the config for your local repository. Once you do that, you can run the update command, which will actually grab the content of the reference for the specific commit of that submodule. Then we can talk about how you actually remove a submodule from a repository. Now there are two different approaches you can take. If you just want to temporarily remove it, you can use the deinit command, and that will remove it, but you can also go back and init again at a later time. You also can permanently remove it by running the deinit command, followed by git rm, and then committing your repository with that change. Now let's talk about a few facts of working with Git submodules. First of all, submodules are truly their own repositories. So that means once you change into that directory, you're utilizing the config of that repository, you're committing to that specific repository, and when you push, you're pushing the contents of that repository. So it's very important when you're working with submodules to know what director you're in and what repository you're working with. Now submodules do not track to a branch, but rather to a specific commit. Now this is to your advantage. You can make sure that someone else doesn't have the ability to change the code that is included in your project because you have to update explicitly if you want to include the new code for your submodule. Now, as mentioned, submodules can also contain other submodules. And while that adds some complexity, we'll walk through that also in a later clip. In addition, you can edit and update your submodules within your project just as you would any other normal repository.

### Adding a Submodule to a Project

So next, we're going to walk through the process of adding a submodule to a project. Over the course of this demo, here's specifically what we're going to cover. First of all, we'll be verifying our version of Git. We do need to be sure we have the correct version because Git submodule support has changed slightly in certain releases. Then we'll be adding a submodule to a project. Then we'll be configuring Git to better work with submodules as there are some small changes we can make that will help your interaction with Git. Then we'll be updating a submodule. And finally, we'll be removing a submodule. So let's dive in. So, as mentioned, we want to be sure that we verify the version of Git that we're leveraging. Now here on my local machine, I'm using Git version 2.22. Now there are changes that have been made, some significant changes to submodule support within Git 1.75, 1.78, and even 2.7, so we want to be sure that you're using at least version 2.7 to be following along with this clip. Now the next thing we're going to do is we're going to make a repository that we can leverage for this clip. Now the next thing we're going to do is we're going to create a directory external, and this will be a directory where we will store the submodules that we will add to this project. Now next, we're going to actually run our git submodule add command, and in this case, we're telling it to pull down the GitHub repository that I'm referencing, and we're telling it that it needs to store it in the external/example‑submodule directory. Now that we've added that in, we can now navigate to that directory. Once we do, we can see that it has pulled down the contents of that specific repository. We'll now navigate back up to the parent repository. From here, we can now take a look at the gitmodules file that was included because we ran the git submodule add command, and we can see here that it does define a submodule within the directory external/example‑submodule. Now it also includes the URL that I passed in that it is actually pulling the repository from. And so next, I'm going to run git status. Now we can see here that there are two different files that are included. The first is gitmodules, which this is the file that we just looked at. The other is a reference to the specific commit of the repository for our submodule. Now this doesn't give us a ton of information about the submodule. So because of that, we can choose to set a different Git configuration setting that will give us more information when we run git status when we're leveraging submodules. And this configuration setting is status.submoduleSummary. So we'll set this value to true, and then we'll run git status again. Now, once we do this, we'll be able to see here that we do have submodule changes that need to be committed. It even lets us know the specific commit that we're actually referencing and what the commit message was for that specific commit. So next, we'll go ahead and add both of these, and then we'll commit. And that is the configuration that allows us to also get submodule summaries when we're doing diffs on our repository, so we want to be sure that we set this value as well. So next, I'm going to navigate outside of this specific repository, and then I'm going to use the file protocol of Git, and I'm actually going to clone the repository that we just created. Next, I'm going to change into the clone of our repository. From here, we'll list the contents, and we can see that we do have the same files that we had previously. But next, we will change into our external/example‑submodule directory. And when we do, we can see that there are no files currently within this directory. And this is because there are some additional steps that we'll need to perform to be sure that we have fully initialized our submodule. I'll navigate back to the parent module, and then from here, we will first run git submodule init. And next, we'll actually update our submodule to pull down the code. Next, we'll change back into our submodule directory, and we can now see that the files have populated from our submodule. So next, we'll talk about how we update the reference for our submodule. So in this case, at Globomantics, let's say that we want to leverage the specific beta branch of this shared repository that were pulling in as a submodule. In this case, we would first need to fetch. After we fetch, just like any normal repository, we can check out a different branch. In this case, we'll check out the beta branch. We will then change back into our parent directory, and from here, we'll run git status. When we do, we can see that we do have new commits for our submodule. So next, we will add and commit by updating this to now leverage the beta branch of the submodule that we pulled into our project. Now next, we'll walk through the process of deleting the submodule from our project. The first step will be for us to deinit the specific submodule. Now, at this point, we could choose to go back in and reinit that submodule, so its not removed permanently from our repository. However, next, we will actually run git rm. Now if we do this, we can now run git status. And part of what it's letting us know here is there is no longer a git repository at the location where it was previously. We've modified our gitmodules file, which has removed the module that we had previously included, and it has deleted our submodule. Next, from here, we'll finalize this by committing to our repository. Now that we've done this, we've successfully shown how we can add, update, and then ultimately remove a submodule from a Git repository.

### Using Submodules with Teams

So in this clip, we're going to be walking through how you utilize submodules with teams. So let's take a look at what we're going to cover in this demo. First of all, we're going to be leveraging nested submodules because the moment you start pulling in submodules, many of those submodules will have submodules underneath them. Then, we're going to be understanding team considerations for submodules. Now this is critical, and this is where some of the complexity comes in in using submodules. Then, we'll be editing a submodule in place. And then finally, we will be pushing to remote with submodules. So first, we're going to clone a repository that has submodules. It actually has nested submodules. Now if we follow the path that we took in the last clip, we would need to first clone it and then go into a submodule directory, initialize, and update it. And then we would need to go into, if that submodule had more submodules, we would need to do that again for each of those. Now this could be quite time consuming if you had a repository that had a lot of submodules. However, in this case, Git provides a command that does all of this for us, and we accomplish this by passing in the recursive option to git clone. This will go through each of the tree of submodules, and it will perform both initialization and updating on each of those. And we can see here that we're pulling in a lot of different submodules within this repository, so let's change into our repository directory. Now from here, we want to take a look at our gitmodules file. And here we can see within our parent repository we have two submodules. The first is external/repo1, and the second is external/repo2. Now we also want to take a look at the submodules that are included within each of these submodules, and we can do that by leveraging a command called git submodule foreach. So what we're going to do is we're going to pass in a command, which will be the same command we used to look at our gitmodules file here. We'll pass that in for each of the submodules. And so it will simply go and run this command within each of our submodules. And we can see first here with our external/repo1, it has to submodules, and then for external/repo2, it has one submodule underneath it. Now next, we're going to talk about when working as a team how we integrate the changes that our team members have made into our repository. So the first step is going to be the same as if we were working without submodules, and that's just using git pull. Now the next step is a little different because let's think about what could happen here with our submodules. First of all, it's possible we could have some of our teammates delete submodules. We could have them bring in new submodules. We could even have them change a submodule. So let me explain what I mean by that. Within our gitmodules file, when we initialize it, it integrates in that into our local config, but it's possible that we could have one of our teammates even change the remote URL for a repository. Let's say, for example, that they cloned a third‑party library. They now have their own version of it to make specific edits, and now they want to reference that repository. So because of that, we need to sync our configuration with our gitmodules file, and we need to do that recursively. And so we're going to use to git submodule sync command to sync our config with the gitmodules file, and we're using the recursive option so it will navigate the tree of submodules for us. So first of all, it's possible that our submodules have been updated, and they're now pointing to a different commit. It's also possible that we have new submodules, as we mentioned earlier. So first, we need to initialize any new submodules, and then we need to make sure that we're updating each of our submodules. Now, fortunately, Git also provides this within a single command, and here we're going to use git submodule update by passing in both the init and recursive options. So it's important to note here that there are three distinct steps that we're taking when we're working in a project with submodules, and these three steps need to be taken every time we're pulling in changes from our team members. First git pull, then git submodule sync recursive, and finally, git submodule update init recursive. You could even create your own alias utilizing what we learned previously about Git configuration so that each of these can be performed by passing in a single command. Now next, we're going to talk about how we edit our submodules and make sure that those changes can make it to our team members. So I'm going to next change into one of the directories for one of our submodules. So in this case, we are still tracking against master, but in most cases, that's not going to be the case. We're going to be referencing a specific commit, and we're going to assume that the submodule that we're working with probably has its own ongoing development lifecycle, so we're probably not going to be on the head of a specific branch. So we need to make a decision here. We need to make a decision about what branch we're going to track against for this change that we're going to make. So in this case, I'm going to choose to check out master, and then next, I'm going to go ahead and pull and perform a rebase, and now we're going to make the changes within our submodule. And so in this case, I'm going to choose to append a value to are README file. Now next, I'm going to commit this within the submodule. Now just remember, we have to think about the directory that we are in. Currently, we are in the directory for one of our submodules, so we're going to commit this to the submodule. Now I'm purposely not going to push this now because I want to highlight one of the risks that can exist. I'm going to now navigate back up to the parent repository. So if we run git status, we can see here that there are new commits, and so we need to go ahead and commit from this repository, from the parent repository, to pick up on the new reference to the submodule. And so in this case, I'm going to go ahead and commit that. Now let's think about what happens if I were to push this right now. If I were to push it, we would be referencing a commit that I haven't pushed from the submodule into the submodule's remote. And so by doing so, I would, in essence, break that submodule for each of my teammates because they wouldn't be able to update the submodule to the commit that it's referencing, so we want to avoid this. Now, we could go through and manually check each of the submodules every time we want to push, but chances are we would miss it. Luckily, since Git 2.7, there is a config value that will go through and detect when you're doing a push in the parent repository to see if any of the child submodules have commits that haven't yet been pushed, and it will push them for you. And so we're going to go ahead and set this config value, and we're going to set it globally because we want to do this by default moving forward. Once we have that in place, we can now push. And you can see here that we have pushed both our submodule, which in this case was example‑submodule‑1, as well as the parent repository, which was the example‑nested‑submodule repository. So throughout this clip, we have first been able to deal with nested submodules when we have been cloning a repository. We have then covered how we integrate changes from our team and how we push out changes to our team utilizing Git submodules.

### Summary

So now we've made it through our focus on Git submodules, and we've covered quite a lot from the beginning. We first looked at submodule support within Git. We gained an understanding of what submodules are, why we would use them, and what some of the challenges could be. We then configured Git to better use submodules. We updated things like how git status and diff handles submodules. We also made sure that we recursed our submodules when we're doing pushes. Then, we added a submodule to a project, we removed a submodule from a project, and we worked with nested submodules, for example how we used the recursive option with clone so that we can handle initialization and updating of all of the submodules when we clone a repository that includes submodules. And then we also reviewed best practices for utilizing submodules as a team. We talked about how when you're leveraging submodules as a team and you're doing pull statements, you need to also sync your configuration for submodules, as well as performing an update and init recursively across all of your sub modules.

## Implementing a Git Workflow

### Overview

In this module, we're going to be walking through how we implement a Git workflow and taking many of the concepts that we've learned so far and piecing them together. So let's quickly review Globomantics, the fictional company that we have been following throughout this course. Here is what they're looking to do. First of all, they're looking to configure new Git installs for the developers, and we talked about in the first module how you leverage Git config to optimize different aspects of how individuals leverage Git. They also want to look at enabling code reuse through common repositories, which we solved through looking at Git submodules. But now we're going to look at two aspects within this module. First of all, developing a local workflow for running tests and linting per commit, as well as enforcing a commit message template on the server. We're also going to look at enabling developers to work more efficiently, and we'll be covering that in a later module. So here, over the course of this module, here's what we're going to be focusing on. First of all, we'll be reviewing client‑side and server‑side Git hooks, and we'll be utilizing client‑side Git hooks to perform pre‑commit tests. And then we'll be leveraging server‑side Git hooks to enforce a commit message template. Then, we'll be creating a custom Git command to automate repetitive steps.

### Introduction to Git Hooks

So next, we're going to talk about Git hooks, and Git hooks are one of the features of Git that allow you to optimize aspects of workflow. So let's talk about what Git hooks are. So first of all, a Git hook is a predefined script that is executed in response to a specific action that has occurred within your Git repository. Now generally, we break these down into two different groups. We first have client‑side hooks, and these are hooks that would run on your machine with your version of the repository based on things that you do to your repository. So next we have server‑side hooks, and these are scripts that execute on the Git server to a different set of events that may happen. So let's look at the overall Git hooks process. First of all, we're going to divide down the middle between client and server. Let's start by looking at the client side. So this will include some but not all of the areas for integrating Git hooks. So first of all, we're going to look at pre‑commit. Now this happens after the user chooses to commit, but in this case the commit message template hasn't yet been generated. Then we have prepare‑commit‑msg, and in this case we do have a commit message that has been generated and we can choose to customize it at this point. Then we have commit‑msg, and this is where the user has entered in their commit message. And we could, in this case, choose to either accept or reject that message. And then we have post‑commit. Now, any of the Git hooks that start with post are generally just used for notifications because in this case the action has already been completed, the commit has already been made. But if you wanted to tie this into some notification system, you could do it through post‑commit. Now at this point, we're now going to say that we have pushed things from our client to the server to our remote. Now the first area for Git hooks that we see is pre‑receive. Now pre‑receive receives all of the refs that are being updated. So you could have many that are being pushed from the client to the server, and you could choose in this case to examine each of those and reject that push that is coming in to the server. You also have update. Now update will get called per ref that gets pushed to the server. So you'll get a chance to look at each of them individually, and you can choose to reject some of those that are coming in. And just as with post‑commit, we also have a post‑receive, which also would be focused for notifications after things have been pushed to the server. Now let's talk for a minute about client‑side Git hooks. First of all, client‑side Git hooks are designed to improve workflow for developers that are using the repository. Now, I note this here that this is designed to improve workflow and not to enforce policy because you cannot enforce policies because the users have their own version of the repository and they can reconfigure those and adjust those. They could even completely remove those Git hooks. Now it's often leveraged to run pre‑commit checks against code, so if you want to do just some simple checks of code before it leaves your developer's machine, this is one of the ways that we would leverage client‑side Git hooks. Now let's look at some of the common use cases. First of all, we have things like linting files, checking to be sure that the files that the developer is going to commit meet the standards for your organization. Running tests, so actually kicking off an execution of the test suite to make sure that we're not injecting code that breaks the tests. Verifying things like no TODOs and committed code. So we have different ways that we can look at the content of those files and make sure that the developer is not breaking any of the best practices that we have defined. We can also use them for preparing a commit message, so potentially providing a template that our users can then go in and edit, and in some cases even cleaning up files after specific actions. So if there is some sort of compilation depending on the language that we're in that we would need to perform before we actually push, we could then go through, for example, and clean up some of those artifacts once that has completed. Now let's talk next about server‑side Git hooks. So these are different because, unlike client‑side hooks, server‑side hooks can enforce team policies because our developers don't have the ability to go in and edit the server and change what those scripts are. So, first of all, when we're looking at the different opportunities to inject our Git hooks, if we want to reject pushes overall based on overall policies, we would use pre‑receive. If we want to look at individual branch pushes, this is where we would leverage update. And for notifications and hooks after the commit, this is where we would use post‑receive. So for server‑side Git hooks, here are some common use cases. We have things like enforcing a commit message format because in some cases organizations want to be sure that developers are referencing a ticket in their ticketing system with every commit. It could also include enforcing user identity information, making sure that they're using their corporate email address as the identity to push this to the server. Enforcing the signing of tags and/or commits. Now there's a lot of talk around whether or not there's value in actually signing commits, but some organizations want to enforce that, and that's something that you can do with server‑side Git hooks. You can also block access for specific IP addresses, and you can even block specific file extensions.

### Implementing Client-side Hooks

So next, we're going to be implementing client‑side Git hooks. So over the course of this demo, here's what we're going to do. First, we'll be verifying our version of Git again because there are some things we will be leveraging that require a specific version of Git. We'll then be reviewing a sample project with testing and linting integrated into it. We will then create a pre‑commit hook to run tests and to lint our files. We'll then update config to include hooks within the repository. So first, I'm going to check the version of Git that I'm using. Now, I'm on Git version 2.22. Now in this case we'll be using a config variable that is only available in Git version 2.9 and later. So I'm fine here. You should be past version 2.9 as well. Now next, we're going to be cloning a sample JavaScript project repository. Now that we have that in place, we're going to change into the directory. Now from here, we're first going to run npm install to pull in some of the project dependencies. Now I'm going to open this in VS Code. Now from here, I'm going to go in and view the terminal. Now the first thing that we need to do is make sure that our index.js file is executable. Now next, we're going to run this script, and this script is a relatively simple one. It will simply calculate all of the prime numbers between 1 and the number that we pass in. So in this case, we'll choose to pass in 100. And we can see that it found 25 prime numbers between 0 and 100. Now the next thing we're going to do is we're going to highlight the different capabilities that we have built into this repository. So first, we do have a test suite that is configured in package.json, so we can run npm test. And when we do, we see that we do have to passing tests. It first tests to be sure that it is returning 25 primes between 0 and 100, and then we're checking to be sure that it is returning the correct prime numbers. Now the next thing I want to highlight is that we also have linting integrated into this repository. And so in this case, we can run npm run lint, and we'll now use ESLint to check over the files that we pass in. Now in this case, we have no linting errors. Now the next thing we're going to do is we're going to navigate to settings for VS Code so that we can see our .git directory inside of the editor. Now chances are for whatever tool you're using to edit your code, you'll need to make this adjustment as well. So from here, I'm going to search for exclude, and then I'm going to make this a workspace setting. I'm going to remove the glob pattern that includes the .git directory. Now we'll see that our .git directory has shown up. We'll now close the settings. Now next, I'm going to open the .git directory and I'm going to open the hooks directory. Now from within here, we can see that there is a pre‑commit.sample file. This is the file that we're going to be using. Now the pre‑commit file is one, again, that gets executed before we actually commit the file and even before the commit message is generated. So in this case, we're going to use this to run both the tests and linting prior to committing. And we will rename this. I'm going to remove the .sample from the end of the file. Now the next thing I'm going to do is I'm going to close the terminal, I'll then highlight and delete everything that's currently in this file, and will now add in some additional logic that will enable us to run both the tests and the linting. So first of all, I've specified that this needs to exit with a non‑0 status if any of the following scripts fail. Now that's critical here because the way that Git hooks communicate with Git is based on that status code. So if anything fails, either the linting or the test, it needs to return a non‑0 status code, that way Git will know that this failed and we should not accept the commit. Now next, I'm going to open back up the terminal, and from here I'm going to close our .git directory and I'm going to navigate to index.js. Now I'm going to add an extra tab in. Now this should cause linting to fail because that's not where we would want this tab. So I'll save this file, and then now I'll try to commit this. And when we do, we can see that there is an error. The linting error does come up, and it lets us know that we cannot commit. Now next, I'll reset the repository to get rid of this change. Now next, I'm going to navigate in the lib directory to findPrimes.js. Now this is the function that is actually being tested by the test suite. So in this case, I'm going to change, instead of returning the output, which would include the prime numbers that it has detected, I'm going to return a fixed array. We'll now save the file. Now from here, I'll now try to commit this change as well. And now we can see that we do have a failure. In this case, we're not returning the correct primes and we're not returning the correct number of primes, so we have failed our tests, and because of that it is failing to let us commit. So in this case, we'll now go back in and we will reset the repository. Now, all of this has worked well so far except there's a problem. Our .git directory is our config for our repository. This is not information that is going to get populated for any of the members of my team. Luckily, Git provides a config option, and it's been around since Git 2.9 that allows us to use a different directory within the repository to store our Git hooks. So I'll go ahead and set this value, and this is core.hooksPath, and I'm going to set this to be .githooks. So next, I'm going to actually make this directory, and then I'm going to move the pre‑commit file into that directory. Now just as a note here, this file is already executable, but if you're creating this file from scratch, you'll want to take the extra step to be sure that you do make it executable before you actually attempt to commit. Now the next thing I'm going to do is I'm going to navigate back into my index.js file, I will add in a few misplaced tabs again, I'll save the file, and now I'll attempt to commit this. And in this case, we do see that it failed yet again. So in this case, this failed, not because of a hook in our .git directory, but because of a hook within our .githooks directory, and that is a directory that will be included within our repository. And as long as our teammates have the correct config value set for where their githooks directory is, it will automatically integrate in with the Git hooks that we have created.

### Implementing Server-side Hooks

Now that we've implemented client‑side hooks, we're now going to transition to implementing server‑side hooks. So over the course of this demo, we will first be cloning a bare repository to enable testing of server‑side Git hooks. Then, we'll be creating a pre‑receive hook to verify the commit message format. Then, we'll be amending commits so that we can enable pushing with the correct format. So first, one of the challenges that can exist when you are testing your server‑side hooks is effectively being able to test that on your local machine. Now the first thing we're going to do is we are going to clone a repository, but we're going to clone it as a bare repository, and this will allow us to simulate the remote on our local machine. And so in this case, we have cloned the bare repository into example‑git‑remote. Now next, we're going to clone another version of the repository that is not a bare version, but we're going to clone it from this version that we just cloned. And this will be our local copy of the repository. And in this case, we're going to use the Git file protocol, but we will be able to leverage this just like we would if we had cloned a repository from a Git host like GitHub. Now first, let's change into our example‑git‑remote directory. And in this case, if we list the directory, you'll see that we're basically inside of the normal.git directory for a repository. However, since a bare repository does not have a working copy, in this case the config directory is the root of the repository. Now next, we're going to create a file for pre‑receive because we want to use the pre‑receive hook point to be able to reject any push that includes commits that don't follow our desired commit template. And what we want to do here is we want to leverage our ticketing system. And let's just say that our ticketing system, something like JIRA or other solutions out there, has a key for each issue. And in this case, the key includes letters and then a dash and then numbers. This allows us to find a specific number within our ticketing system. Now, because of this, we want to be sure that all of our developers have a ticket that is associated with each of their commits. So first, we will create our pre‑receive file. Then we need to make sure that it is executable. Now next, we're going to go in and edit this file. Now from here, we're going to drop in code that will allow us to check for this format. So at a high level, first we have a regular expression, and we've included the letters GSP for Globomantics Standard Project. And then we have a dash, and then it can include any number of numbers after that. Now this is the format that we're going to be looking for within each of the commits. Now this specific version of this is a modified version that comes from some platform samples that GitHub makes openly available. Now in this case, we're going to check with each of the refs that get included because remember, when we're using the pre‑receive hook, we're going to be getting one or more refs that are going to be pushed at a time. But in this case, if it doesn't meet the regular expression, we're going to reject the push to the server. So now that we have this in place, we'll now save the file. So we're now going to change into our local repository. In this case, we'll choose just to edit a sample file. We'll just choose gitignore. We'll then say that we want to ignore the vscode directory. We'll now save the file. Now that we've made that change, we're going to commit that change. Now, we're ready to actually push to the server. Now again, because we're using the file protocol, we're actually going to be pushing it just to the other repository that is our bare repository that we checked out first. So next, I'll push. And we can see here that we do have an error that comes back. It lets us know here that it was rejected because a specific commit is missing the Globomantics JIRA Issue key, which in this case as an example we're saying is GSP‑123. So to make this work, we're going to have to actually go in and amend our commit. Now we're going to choose to add in our issue key first. We'll say that this is our ticket, we'll then save this, and we'll close the editor. Now that we've amended our commit, we'll now push. And we can see now that our push has been accepted. Now this is just the beginning of what you can do utilizing server‑side Git hooks. So if you're interested in learning more about server‑side Git hooks and seeing more examples of how they work and even learning how to set up your own Git server, you can see another course that I've authored here on Pluralsight, Git Administration.

### Creating Custom Git Commands

So one aspect of improving your workflow with Git revolves around automating things that normally take multiple steps. And we're going to leverage a feature of Git, which is custom Git commands to do just that. So over the course of this demo, here's what we're going to be doing. First, we will be creating a directory to store custom commands for our user. We'll, then, make sure that that directory is in the PATH variable. And then we will create a custom command that combines multiple Git commands into one, and then ultimately we will run and verify our custom command. So first, we need to create a directory for our user to store our custom Git commands. In this case, we're going to call that git‑scripts. Next, we'll change into this directory. Next, we're going to edit a sample file. In this case, we'll just call this git‑testscript, and you'll notice that every command that we create needs to start with git and then dash. So inside of this file, we're just going to include an echo statement. This will allow us to test and make sure that this custom script is working. Next, we need to make sure that this file and any of our custom Git commands are executable. Now from here, the next step is that we need to make sure that our Git scripts directory is in the PATH variable for our user. So in this case, for me, this is going to involve me putting this into my zshrc because I'm using Zsh as opposed to Bash. Now, if you're using Bash, which is the default on Mac, you can simply run this and drop this into /.bash\_profile. Now you will need to update this directory to be the specific directory for your user. I'll go ahead and put this back in, and then I'll hit Enter. Now the next step is I need to actually source this value, and this will actually pick up on that PATH variable. Now next, if everything works properly, I should be able to run Git and then pass in as the subcommand the name of my custom command. So in this case, git and then testscript. Now as I mentioned, any Git custom command in terms of the file needs to start with git‑, and it will only work if this is on the path. And we can see here that we do see the echo statement indicating that the testscript worked. So our custom Git command is working just fine. But now we're going to create one that we actually will use. Now if you remember previously when we talked about working with submodules, there were three different steps that we needed to integrate when we were looking to pull down commands for a repository that contains submodules. In this case, we're going to automate those steps into one custom Git command. And in this case, we're going to call it git‑subpull. Now I'm going to paste in what would be the logic that we will be automating for this particular custom Git command. So first, you'll notice that I'm using Bash, and then I'm going to call git pull. This will allow us to pull changes from the remote. Then, I'm going to call git submodule sync ‑‑recursive, and this is what will allow us to sync our config in the local repository with any changes that have come in from one of our direct submodules or any of the child submodules. And then finally, we're going to call git submodule update ‑‑init ‑‑recursive, which is what will update the submodules to the correct commit. And if they haven't yet been initialized, it will initialize them. And we can perform all of those across the entire submodule tree, recursively. I'll now save the file. Now from here, the next step is I need to make sure this particular file is executable. Next, I'm going to clone the same module that we leverage before the integrated git‑submodules. And next, I'm going to change into that directory. Now that I'm in that directory, I'm going to now use our new custom Git command, git subpull. And we can see that it has walked through each of the three different steps that we included within our custom Git command.

### Summary

So here we have walked through multiple ways for how we can implement, and then ultimately improve our overall Git workflow. Let's quickly review what we've covered. First of all, we reviewed client‑side and server‑side Git hooks. We learned that client‑side Git hooks actually execute on the machine where our local repository is. And the server‑side Git hooks execute the remote server. We then utilize client‑side Git hooks to perform pre‑commit tests. And we verified in this case that we could both run tests and linting as a part of that process, and that those commits were rejected if either of those processes failed. And then we utilized server‑side Git hooks to enforce a commit message template. Because we know that Globomantics wants to ensure that developers are referencing a ticket for every commit they make, we included this change and because it is a server‑side Git hook, we can use it to actually enforce a policy for our developers. And then we created a custom Git command to automate repetitive steps. In this case, we looked at the situation where we are dealing with submodules inside of a repository. And when we want to pull down a new code, we want to ensure that we are pulling it down, sinking our config, and then ultimately updating and if need be, initializing the submodules within our project.

## Finding Bugs with Git

### Overview

So in this module, we're going to be using Git and one of the tools provided by Git to help find bugs within our repositories. But first, let's quickly review Globomantics, the fictional company that we have been following throughout this course. They were first looking to configure new Git installs, enable code reuse, develop a local workflow for running tests and linting per commit, and enforcing a commit message template on the server. And currently through the previous modules, we have been able to meet all of those needs. Next, we're going to talk about enabling developers to work more efficiently. And one of the key aspects of any development workflow is finding bugs, and so we're going to see the tools that Git provides to help us with that process. So first of all, in this module, we will be reviewing the Git Bisect tool and its capabilities. Then, we'll be utilizing Git Bisect to manually determined when bad code was injected within our repository. And then ultimately, we will be utilizing Git Bisect to automatically determine when bad code was injected.

### Using Git Bisect

So next we're going to talk about how you leverage the Git Bisect tool to help find the point at which bad code was injected into your project. But let's first review some general version control benefits. First of all, version control does enable us to see a history of the entire project. We can see what changes were made to what files and when they were made. We also can see specifically who made them. We can know which developer committed a specific line of code. Now because of that, with any good version control system, we can identify the point at which bad code was injected into an application. We simply need to spend time reviewing all of the logs, we can try to determine where a problem is happening, and then try to track down changes that have happened to that specific project. However, with this being said, this could be horribly inefficient. This can take a good deal of time. Now Git provides a tool that will help us in this process, and that tool is called Git Bisect. So Git Bisect is a tool that is included with Git that enables you to specify a start and end commit and then either manually or automatically determine the point of failure within that range of commits. Let's see how this plays out. So let's say that we have a sample repository, and these are all commits within this repository, and we know that at this point code was working correctly. However, by the time it got to the end, code was not working correctly. So we need to determine where within this range that bad code was injected into the project. So first we'll call this code working correctly commit, or good commit, and we'll call the code not working correctly commit, or bad commit. Now first, Git Bisect will identify a commit near the middle. It will know in this case that this one is not working correctly and so because of that, it knows that the commit must happen before this point. It's then going to go back and search, and we can see here that there is now a good commit happening here, and then it will continue to proceed and we'll see here that there is another good commit, which means that by the process of elimination that we're going to have a first bad commit identified here. Now this is the power of Git Bisect in that it helps guide you through this process, either manually or automatically, which we'll talk more about in the next clip. Now let's look at how you enter bisect mode. You simply need to run Git Bisect start and then specify both the good and bad commits, like we mentioned previously. Now next, we need to classify the commits that Git Bisect takes us to as either good or bad, and that is simple as just entering Git Bisect good or Git Bisect bad. But as I mentioned, there is an automated way that it can help guide us through this process. So if we have a command that will test our code from within our repository, we can simply utilize Git Bisect run and then specify what command it needs to run. And as long as that command will use the status code to correctly indicate whether or not something passed or failed, we can then use this to determine when our first bad commit was. Now whether you're using the automated or manual process when you are finished, you need to call Git Bisect reset, and this will allow us to exit out of bisect mode and we can now go fix the code that we found.

### Detecting Bad Commits with Git Bisect

So next we're going to utilize Git Bisect to detect bad commits within our repository. So over the course of this demo, we will first utilize Git Bisect's manual mode to find when bad code was injected. But then we'll be integrating Git Bisect with tests to automatically find a bad code commit. So the first thing we need to do is clone the example JavaScript project that we used within a previous module. Next, we're going to change into that directory. From here, the next step is we need to run npm install to pull down the project dependencies. Next, we'll check out a branch, I'll call git fetch and then we'll check out the bisect branch. From here, we're going to run our index.js file. Now just as a reminder, this file allows us to pass in a number, and that number is the max and we'll find prime numbers up to the max value. Now, in this case, I'm going to run up to 100, and when we do we'll see that it found 15 primes between zero and 100. But that actually is incorrect. We should be finding 25 primes between zero and 100. So we know that there is a problem in our code, and we can verify that by actually running npm test. And we can see here that we have four different failures. We have no test passing and all failing. Next, we need to look at the commits that have been made into this branch. So we have one commit here, which is breaking the findPrimes.js logic. Now, this is the commit ultimately, that we want Git Bisect to find for us. We're going to specify the commit for updating our gitignore file as our good commit, because we know it was working at this point. And we'll utilize the add more test commit to be the commit where this is our bad commit, because we know that this is a point where it was no longer working. So I'm going to go ahead and exit out of here and we'll go ahead and enter Git Bisect mode. Now from here, our next step is to specify our good commit. Then, we need to specify our bad commit. Now we've currently entered Git Bisect mode and Git Bisect is waiting for us to determine whether or not the current commit that we're on is good or bad. Now, we can test this by simply running our index.js file. In this case, we can see that it is returning 15 primes instead of 25. So we know that we still are bad. Now, next, we're actually on the commit where it is breaking. And so in this case, we're going to run our index.js file again, and we can see that it is still returning the incorrect value. So we'll specify this commit is bad. And it lets us know then that this is the first bad commit. It has done the analysis and it is determined that this is the point where the bad logic was injected. We can now run git show. From here, you can see that there was a single line that was passed in. In this case, it is cutting the max value in half from the max value that is being passed into the function. So we know that this is the logic that caused it to break. So now I'm going to exit out of git show. Now, once we're done to exit out of Git Bisect, we simply need to run git bisect reset. Now we have exited out of Git Bisect mode. Now the next step is we're going to inter Git Bisect mode again, and in this case, instead of using the manual process, we're going to use the automated process. But our first step is still the same. We need to specify our good commit and we need to specify our bad commit. Now in this case, we're going to use Git Bisect and we're going to tell it to run our test suite and it can run our test suite by running npm test. It now went through the process and it landed at the same conclusion, that the breaking the findPrimes.js logic commit was indeed the one where the bad code was injected. Now, some of you might be worried if you don't have tests at the point when you think that the bad code was injected. However, if you include tests, they're not a part of your repository, they will not be wiped out when Git Bisect switches between different commits. And so in that case you can still use the automated process, as long as you keep those tests that you're using to validate whether or not a commit is good or bad out of the repository for the time being.