## Customizing Git with Configuration

### Version Check

Hello there. Before we begin, let's check the versions of Git that apply to this course. This course was created using the displayed version of Git, and here are the versions of Git for which the information in this course definitely applies.

### How Git Evaluates Configuration

Hello, and welcome to Git Configuration and Attributes. I'm AJ Foster, a software engineer at CodeSandbox. I've been working as a developer since 2012, and Git is one of those skills that just doesn't go away. What you may not know is just how configurable Git can be. In this course, we're going to take a practical look at how Git's configuration and attributes features can make your life easier every day. The first step is to understand how Git finds and evaluates configuration. But, before we dive in, I need to make a quick side note. This course relates to configuration, and configuration is very personal. Git allows you to change so much of how it behaves because reasonable people can disagree on the best way to use Git in a variety of circumstances. You will see examples throughout this course, but, that doesn't necessarily mean you have to use the options shown. Talk with your team, evaluate your own needs, and seek out what works best for you. And, if you're looking for more opinionated recommendations from me, that will come later in this course when we talk about how configuration and attributes can impact your workflow. For now, let's talk about how Git configuration works. Git has several hundred configuration options available, which it refers to by name, such as user.name or core.editor. Every option has a default value or a default behavior that Git will use if it's unset. These defaults form the base layer of how Git evaluates configuration. Git then divides its configuration into several levels. At the bottom, we have system‑wide configuration that affects all users. This isn't often used, but we'll see an example of when it can be useful. Next, we have per‑user config, which Git calls global configuration. This is the most common place to put your preferences, as different users can choose their own options without affecting one another. Finally, at the top, we have per‑repository configuration. This is limited to a single copy of a repository, but it can affect any user on a system who interacts with that directory. When Git needs to decide how to behave, it evaluates all three levels of configuration from the top down. Per‑repository config has the highest precedence, meaning Git will use options for a specific copy of a repository when they are available. Then, it will look in the user's global config. Finally, if it still hasn't found what it's looking for, it will check the system config and ultimately fall back to the default value or behavior. With this in mind, let's talk more about each level of configuration, starting with the system level. As I mentioned, configuration at this level affects all users on a system. It's stored in a central location and manipulated using the system flag when using the Git command‑line interface. We don't often use system configuration because it's usually not the right level of abstraction for our preferences. Either we are working on a computer with a single primary user, in which case the user config is just as helpful as the system config, or we're working on a computer with multiple users where each individual user is likely to have different preferences about how to work with Git. System configuration isn't useless, however. Imagine you are working on a computer where several commonly used programs are installed in a different location than normal. Git may not be able to find these binaries when it needs them. Luckily, using configuration, we can tell Git to look in an alternative location, and by doing this in the system config, it will solve the issue for every user on the machine. While there are occasional uses for system‑wide preferences, most options end up in the user‑specific or global configuration. These options are stored inside a user's home directory and can be manipulated using the global flag with the Git CLI. Many people's first interaction with Git configuration is setting their name and email address in their global configuration. When in doubt, start by adding your options here. Occasionally, we need to modify an option for a specific repository. These options are stored in the .git folder of the repository, but, don't be fooled by this, they don't get transferred when you push or pull the repository. So while they will affect anyone who interacts with this copy of the repository, they won't propagate to other computers. We use the local flag for this config or we can omit the flag altogether. As an example, we can use local configuration to set a custom email address for a repository. This can be useful when you have personal and work‑related projects on the same computer. So, that's the hierarchy of Git configuration. Local config for one‑off special case changes, global user config for most of your preferences, and system config for items that affect everyone on a machine. Before we finish, I want to give you two additional notes about how Git evaluates configuration. First, there are a number of environment variables that can alter the way Git behaves. For example, setting the GIT\_AUTHOR\_EMAIL variable will override the user.email setting in the config. These variables are not often used and I recommend using configuration instead. Second, it's possible to introduce additional levels of configuration if necessary. Git allows conditional inclusion of configuration files based on the repository path or branch, using the includeIf option. This is beyond the scope of this course, but it's available if you need it. Now that we understand the hierarchy of Git configuration, let's start adding our own options to change how Git behaves.

### Adding New Configuration

We're ready to add some configuration of our own. The first interaction many users have with Git configuration is adding their name and email to be used in the author section of a commit. Using the Git command‑line interface, we can set a name and email at the global level. Git config is the base command we'll use throughout this module. To set the user's name, we can use git config ‑‑global user.name followed by your name. Breaking this down, we're calling git config, specifying the global configuration for the current user, giving the name of the option, in this case, user.name, and finally, the value to set. If the value includes spaces, then quotes are necessary. If we want to confirm that our change is in place, we can call the same command again without supplying a value. Instead of setting the configuration, Git will read it back to us. We don't have to use the command line to edit Git configuration, however. Let's take a look at the configuration file that was just created when we set the user's name. We'll talk more about the structure of this file soon. For now, let's follow the pattern and add an email address. Once we save this file, it is just as if we had set the option via the CLI. Of course, not all configuration has to go in the user‑specific global config, we can swap out the global flag in our command to system to set a system‑wide preference. For example, this will set the editor program used by Git if a user doesn't have their own configuration. Similarly, we can set an alternative email address for a particular repository with local config. This command should be run inside the repository we want to change. Here, the local flag is optional because working with local configuration is the default. To review, we can use the git config command to get and set options at each level of configuration. With no value supplied, the command will read back the current value of the option. Supplying a value after the name of the option sets it. Using the local, global, and system flags, we can control which level of config we want to work with. The command‑line interface is a convenient way to work with configuration, but it's helpful to remember that everything is stored in a file behind the scenes. Next, we'll take a closer look at these files and how they're organized.

### Structure of a Configuration File

Whenever we alter our configuration using the command line, Git transparently edits a file in the background. Let's take a look at those files and their contents. Remember that each level of configuration has its own file, and the exact location of those files might be different between systems. Regardless of the level of config or its location, the structure of the file is the same. For our examples, we're going to focus on the global configuration located in your user's home directory. This is often a .git/config file directly in the home directory, although Git also supports another location compatible with the Cross‑Desktop Group standard. Let's open up the file and take a look. So far, we've modified two options at the global level, the name and email address to use in commits. To do this, we ask it to modify options called user.name and user.email. Now, in the configuration file, we see those terms divided up in an interesting way. User is split out at the beginning and each option, name, and email has its own line. This is typical of configuration options in Git. The first part of an option forms the section, which is wrapped in brackets and acts as a header in the file. Later parts form the name of the option, which is followed by an = sign and its value. Let's look at a larger configuration file now. This one has multiple sections with multiple options in each section. It also has a subsection wrapped in quotes with options that apply only some of the time. Configuration files can have extra white space, as well as comments using the # sign or semicolon. The order of the sections and their options doesn't matter as long as the options end up in the correct section. If you find yourself editing a configuration file manually, it may help to reorganize the sections and names alphabetically. A quick side note. If you've seen a lot of configuration files, this syntax probably looks familiar. Git's configuration files look a lot like INI or TOML files. However, don't be fooled by the similarities. Git does not support multiline strings or other advanced features of TOML, and it considers the subsection syntax of INI files to be deprecated. To summarize, when we write user.email on the command line, we're referring to the email option in the user section of the config file. Most documentation will refer to options using the dot syntax for consistency. But now that we understand the division of options, let's talk about some common sections that we see in configuration files. As we've seen, user contains options related to the author information that appears in commits. Core contains options that change fundamental parts of how get behaves, often affecting many different Git commands. This ranges from which programs are used to display information to the level of compression used for objects in the repository. Each Git subcommand has options associated with it, allowing you to change how git push interacts with remote repositories and how git rebase deals with unstaged files. If this is starting to sound like a lot of options, it is. There are several hundred options listed in the documentation for git config, and the list is known to be incomplete. Subcommands tend to document their own options, and it's even expected that config files may include options for tools besides Git itself. It can be overwhelming. My advice is to start by evaluating your personal pain points. For example, maybe you get annoyed at having to type the set‑upstream flag when you call git push on a new branch. There's an option for this. It's called push.autoSetupRemote. If you don't like something about the way Git behaves, you can probably change it. Next, talk with your team. There may be configurable best practices that help to avoid problems with your team's particular workflow. For example, perhaps the team has standardized on dev or trunk as the name of the default branch. You can use init.defaultBranch to set this name automatically when creating a new repository. Finally, search for other people's Git configuration online. Many people share their global configuration files and other so‑called dotfiles to provide inspiration for others. Once you get a configuration that you like, you can do the same. Now that we have a better understanding of Git configuration and the breadth of options available, let's take a moment to see how we can manage our ever‑growing configuration files.

### Showing and Removing Configuration

We've seen that there are a lot of configuration options available in Git. Since options tend to be spread across multiple files and often build up over time, let's take a look at the tools available to help us manage our configuration. First, let's imagine that we've recently added an option to our global configuration, but Git isn't behaving the way we expect. We've double checked the name of the option and its value, but something still isn't right. A great first step is to ask Git to show us the current configuration options. We can accomplish this using git config ‑‑list. This command displays the configuration active for the current directory. It's a great way to quickly check if an option is set and it has the value you want. By default, Git uses the less program to page the output, and you can type Q to quit. Depending on your shell environment, if you're looking for a particular value, you can take the output of command and either sort it or search the text. Now, that's great for confirming the currently active configuration, but it doesn't help diagnose why a configuration value is different than what you might have expected or why it shows up multiple times. For this, we can use the show‑origin flag to get some additional information. This shows us the name of the file where the configuration can be found. We can go one step further and use the show‑scope flag to add the scope, such as system, global, or local, to the output. Doing this can make it clear when we have the same configuration in multiple scopes, possibly overriding the value we want to use. In this case, I have custom user information in the local scope of the current repository. If I no longer want to override my global name, we can remove the local configuration by using the unset flag for git config. This will remove the configuration from that scope, allowing the more general scope or default value to be used. If we want to take it one step further and remove an entire section of configuration from a file, we can do that too using the remove‑section flag. As you evaluate and debug your configuration, keep in mind the general guidelines for which options should go in each scope, local config for one‑off, special‑case changes, global user config for most of your preferences, and system config for items that affect everyone on a machine. We've now seen how Git evaluates configuration across various scopes. Each scope is backed by a file that can be edited manually or manipulated using the Git command‑line interface. There are flags to set, unset, show, and remove options from each scope, and we've seen how to learn the source of an active configuration option. Remember that Git has hundreds of configuration options available affecting many aspects of its behavior. Sometimes, however, we want Git to behave a little bit differently depending on the type of file it's working with. For these kinds of options, Git provides attributes, which we'll learn about next.

## Managing Files with Attributes

### How Git Evaluates Attributes

Git has a large number of configuration options controlling many aspects of its behavior. Sometimes, however, we want Git to behave a little bit differently depending on which file it is working with. This is where attributes come in. To help understand what we mean by changing behavior depending on the file, it's helpful to think about something that isn't really an attribute, but acts like it, ignoring files. With a .gitignore file, we specify a list of paths, optionally including wildcard and glob patterns that Git should treat differently from other files. Attributes are going to look very similar with an even higher level of control. Let's dive right in and look at an example of a gitattributes file. Much like a .gitignore file, each line starts with a pattern that matches one or more file paths. Be careful though, because there are two major differences between patterns and a gitattributes file and to .gitignore file. First, you cannot use negative patterns. Instead, you'll have to use a regular pattern and set a different attribute from whatever matched above. Second, you cannot match every file in a directory by writing the directory name with a trailing slash. Instead, you'll have to use a glob pattern to match every file. Returning to the example file, after the file pattern, we can list one or more attributes. These attributes may be set to a value, like on the first line where we tell Git to process .ex files as elixir code when generating diffs. This will include elixir, module, and function information in the headers of diff chunks. Attributes could also be Boolean in nature, like the ‑text attribute that is set on the last two lines and unset on the second line. This attribute controls whether Git will automatically manage the type of line‑ending character, character return, or line feed when a file is checked in or out. As a general rule, attributes are available when Git needs to make an assumption about how to work with a particular type of file, and it might make the wrong choice. Much like configuration, attributes exist in multiple levels, with each level backed by a file. At the bottom, we have the default behavior and a system‑wide attributes file that will affect every user on a given computer. We also have a user‑specific attributes file whose location can be controlled with the core.attributesFile configuration. There's even a local attributes file inside of the .git directory, which affects a single copy of a repository. None of these files travel with the repository on push or pull, so they act just like Git configuration in that respect. Git even evaluates them in the same order from the top down. However, there's an extra set of locations where Git attributes can live, .gitattributes files inside of the repository itself. Like a .gitignore file, these files can be placed in subdirectories and are committed into the repository to be shared by everyone. The local, non‑committed attributes still have the highest precedence, but these files are evaluated next in the order of how close they are to the file Git is currently working with. This makes gitattributes quite powerful, but it also makes it more difficult to determine where any given attributes should live. Next, we'll see some common attributes and how they might be divided among the files.

### Setting Git Attributes

Sometimes repositories contain images, in this example, a JPEG file. We can use attributes to make sure Git treats this file appropriately throughout its lifecycle. First, we want to ensure that Git won't see a byte that looks like a carriage return or line feed character and think that it needs to change these characters when the file is checked in and out. We can do this by unsetting the text attributes for this type of file. As something that would be beneficial, mandatory even, for everyone who works with this repository, we can include this attribute in the Git attributes file at the root of the repository. If there's a subdirectory that contains all of the JPEG files, it could go there as well. This also goes for the attribute that tells Git it should not try to generate text diffs for this file. Now, instead of trying to print out the differences in the binary version of the image, Git will simply state the image has changed. As a side note, unsetting these two attributes is so common that Git has a special macro called binary to accomplish this. Now let's imagine that we have installed a tool called exif, which takes an image file and displays metadata about the image. We can tell Git to use this program when generating a diff of a JPEG file with the help of some configuration. I'll go ahead and put that configuration in my user‑specific global config file. This defines a new subsection of the diff configuration that will only affect files with the diff attributes set to jpeg and tells it to use the exif program. Unfortunately, because we unset the diff attribute earlier in our repository and the repository attributes have higher precedence than global attributes or configuration, we won't see the results of this change. Luckily, there's a place we can override even those attributes defined inside of the repository, the local attributes file. Here, we can target JPEG files and turn the diff attribute back on. This only affects our copy of the repository so everyone else will get the safe default behavior while we get the benefit of using the exif tool. Now when we look at a diff of the JPEG file, we see text generated by exif, including changes to the metadata of the image. This pattern is fairly typical when working with attributes. Mandatory or highly‑beneficial defaults can be committed into the repository, and individuals can override these attributes with their own local attributes. These overrides, as well as the user and system‑wide attributes, can make use of extra programs and configuration that might not be available to everyone. There's one other common attribute found in many repositories that controls the end‑of‑line characters used in a file. We've already seen the text attribute that controls whether Git will automatically change line endings when files are checked in or out, but sometimes a file needs to always have a certain type of line ending. For example, PowerShell scripts are native to environments that use carriage returns at the end of each line. We can set the eol attribute to crlf to ensure that these files are always committed and checked out with the right characters. Similarly, UNIX shell scripts are native to environments that use line feed characters, and we can set the eol attribute to LF to ensure that they stay that way. Each of these are attributes that should probably be enforced for everyone that works in a repository, lest the files stored in the Git index have the wrong data. So you'll usually find them in a Git attributes file in the repository and not in someone's user or system‑wide attributes. You might have picked up on the idea that attributes can affect how files are modified when checked in or out. The text attribute is a perfect example of this. On Windows, Git may insert carriage return characters into files when they are checked out and remove them before storing the data in the repository on commit. This idea of checkin and checkout changes is called a filter in Git, and we'll learn more about those next.

### Git Filter Attributes

At its heart, Git's job is to efficiently move files between its storage and the working directory where we edit them. Sometimes the data shouldn't look exactly the same in each of these places. For example, Git prefers to store files with simple new line characters in its storage, but these files may need carriage returns when they are edited. This process of changing files as they move between Git storage and the working directory isn't only accessible to Git, we can use it too, with filters. A filter requires both an attribute and some configuration. First, in our attributes file, we match on one or more files and define an attribute filter with a string name as its value. This name is the driver of the filter. Over in our configuration, we need to define a subsection of the filter config with a matching driver name. We've heard this before, but it's worth repeating. While attributes can be committed into a repository, configuration cannot. So, if you decide to implement a filter for your repository, consider whether it might belong in the local or user‑specific attributes so that it doesn't travel with the rest of the repository. Otherwise, it's probably a good idea to document the setup very well so that others can create the necessary configuration. Back in our configuration, we can define two options, clean and smudge. Each one is a command that gets run with the contents of a file as standard input. Let's take a moment to understand when each step runs. Thinking about our data from Git's perspective, a Git repository's internal storage is like a pristine file cabinet where Git efficiently organizes everything. In contrast, the working directory where we edit files is like a messy workshop with changes happening everywhere. Keeping this in mind, the clean part of a filter is where Git takes information from the messy workshop and cleans it before storing it away. The smudge part is where Git takes a clean piece of data and rubs some dirt on it so that it will match the rest of the workshop. As an example, we can use a filter to format code before it's committed. A number of languages have an auto‑formatter, and all we need is a command that accepts code as standard input and prints the clean result as standard output. With the elixir language, our clean filter would be mix format ‑, with the dash signaling that the command should listen for file contents from standard input. Now, whenever a file is checked in, Git will format the code and store the formatted output. In this case, we don't really want to deformat the code when it's checked out, so, we can omit the smudge filter or use cat to leave the contents alone. Another feature of Git, hooks, might be better suited for formatting code on commit. However, this should give you an idea of just how powerful filters can be. Before we see a full implementation of clean and smudge filters, I should mention one other type of filter that Git supports, a process filter. When a large number of files are checked in and out of a repository, it might become computationally expensive to start and stop the clean and smudge programs for each file. Process filters allow a program to start once and receive files asynchronously from Git. There's a special protocol for this, which we won't cover in this course, but it's there if you need it. Now let's see a practical example of clean and smudge filters.

### Using Clean and Smudge Filters

When we think about changing files as they're committed or checked out, it's natural to think about what kinds of data we don't want to be committed in a repository. Let's see an example of using clean and smudge filters to transparently keep our API keys and other secrets out of our commits. First, we need to identify the files that should be processed by our new filter. In this example, let's focus on the .env file where environment variables are stored for my project. In this file we have a few API keys that we need during local development, but shouldn't be shared with the outside world. I'll define a filter called secrets and scope it to just the file we care about. Right now this filter won't do anything because we haven't defined it in our configuration, so let's do that next. Over in the local configuration for this repository, I'll define a new filter driver called secrets using the subsection syntax. It will have both options, clean and smudge, and I will also mark this filter as required so that Git will inform me if it fails rather than silently continuing the commit or check out. The clean filter will use a script we haven't defined yet to look for secrets and replace them with placeholders that are safe to push to a centralized repository. The smudge filter will do the exact opposite, replacing those placeholders with the real secret so that we can use it. Most projects use ignored files for this sort of thing, but that may not always be an option. Git filters give you the flexibility to keep a file in the repository while still rendering its contents safe. Our last step is to define the script where we clean or smudge the file. For this example I'll use sed to quickly replace the values. Filters can be highly complex scripts or binaries compiled or interpreted from any language. If defined outside of the attributes file, the program must be executable and it must be located in a path where your shell looks for binaries. Your password manager might have a command‑line tool that can help. With everything in place, let's commit our .env file. I'll go ahead and push this to GitHub so we can see the results of our work. Looking at the file here, we can see that our secrets have been replaced with placeholders. A similar script in the other direction would work for smudge. This isn't necessarily the best way to handle secrets in a modern application, but it's a tool at your disposal. We've now seen how attributes can control Gits behavior depending on the type of file it's working with. If you would like to see recommended attributes for use with a variety of languages and frameworks, there is a GitHub repository that collects attributes files, and we'll see some more opinionated recommendations for configuration and attributes, next.

## Improving Git Workflows

### Increasing Status Visibility

For many developers, Git is an integral part of their daily workflow. It's a great tool, and it helps to address many of the common problems that can come up during the software development process. The thing is, Git is meant to be a background tool. Every interaction we have with Git is a chore that distracts us from what we're really trying to accomplish. That's why in this module, we're going to focus on ways that we can streamline our Git experience. We'll start by making it effortless to see the current status of a repository. Then, we'll configure some aliases to simplify and automate common tasks. Finally, we'll see some opinionated recommendations for your configuration and next steps. Let's begin with the most fundamental piece of information we need when working in a repository, the current status. The output of Git's status tells us a lot about the state of our project in a small space. Even without reading everything, the output is a visual indication of the current state of the project. Are there many files waiting to be committed or a few? Are they existing files that were modified or brand new and untracked? Have we already staged anything for commit? We need this information so often, I have the command Git status aliased in my shell to the single letter g for easy access. We'll talk more about shell and Git aliases later. Many IDEs, including VS code, keep Git status information visible at all times. But, you may have noticed one other place where I keep status information visible, the prompt of my shell. Similar to the full output of Git status, this area of my shell prompt gives a quick visual indication of the state, the current branch, files changed or staged, and how many commits ahead or behind we are from the remote copy. This particular plugin, which is available for ZSH, keeps output dense to save space, but it's also highly configurable. If you use ZSH as your shell in a Unix‑like environment, you can install the plugin directly or simply enable it if you use a framework like Oh My ZSH, where it comes bundled. This also works in Windows subsystem for Linux. If you use Bash or Fish as your shell in a Unix‑like environment, a port of the same plugin is available. Installation requires a quick download and the addition of some lines to your shell profile. In non‑Unix‑like environments like Windows, you don't have to give up this feature. Although the legacy command‑line terminal doesn't support customized prompts like this, PowerShell does. Git has an appendix to their documentation that describes how to integrate Git with PowerShell, including the installation of the posh‑git plugin for status information. Just like configuration, our shell prompts are highly personal. If you choose to integrate a Git status plugin into your shell, I highly encourage you to investigate the configuration options available. For example, I prefer to keep my prompt as small as possible with the Git status seamlessly hidden for directories that are not Git repositories. I also find it easier to read the status indicators when they are separated by spaces instead of pipes or other characters. My goal when working through these options is to find a way for Git to communicate information quickly when I need it and fade into the background when I don't. Another way of pushing Git into the background is by reducing the amount we need to type with aliases, which we'll talk about next.

### Using Git Aliases

Although text editors and IDEs like to give us graphical ways of interacting with Git, many developers find themselves working on a command line at some point in a project's lifecycle. Here we have to type commands, many including flags, and often. Aliases allow us to reduce typing and simplify each step. Now, before we dive in, we need to clarify what we mean by aliases because there are two main types we can talk about. First, there are aliases in our shell. This is where we type something other than Git followed by a subcommand name. In my case, I use the single letter g for git status, ga for git add, and a few others. These are configured in your shell profile, such as a ZSHRC file or a Bash profile. ZSH even has a plugin that defines common aliases for Git. Shell aliases are great because they allow you to collapse entire Git commands into just a few letters. Beyond the sheer speed of interacting with Git in this way, this also has long‑term benefits for your hands by reducing the risk of repetitive stress injuries. I encourage you to learn more about this. The other type of alias we talk about are Git aliases. These are defined in our Git configuration, and they allow us to simplify the way we call Git subcommands. Let's create an alias to see how they work. In my user‑specific global configuration, I'll create a new subsection called alias. The name of the configuration option will be the thing we type on the command line after Git, in this case, git stash‑all. I want this alias to save all of the currently modified files in the stash, including untracked files, which requires an additional flag. Think of the alias subsection of your configuration as a find and replace for Git commands. When we type git stash‑all, it will replace the stash‑all part of the command with the value of the alias. Back on the command line, we can now quickly stash all of the currently‑modified files using our new alias. This is much easier to remember and much faster to type than the full git‑stash command, including the necessary flag. If you search online for recommended Git aliases, you'll find quite a few that might be useful to your workflow. Here are a few of my favorites. Along with the stash‑all alias we just saw, Scott Nonnenburg describes aliases for undoing a commit and getting a nice graphical log of recent Git history. Michael Kaufmann has a helpful alias for unstaging files. Despite the fact that this command is usually included in the output of Git status, I often struggle to remember and to type it correctly. Finally, for projects that use Git branches as the basis for deployments, I'd like to define aliases that easily push work to those branches. For example, the first alias takes whatever I'm currently looking at and sends it to the staging remote branch. The second resets the staging branch with whatever is on the remote copy of the main branch. These are dense commands that are easy to mistype with potentially negative consequences. In general, if you find yourself struggling to remember or type a long command, a Git alias might be a good addition to your configuration. All of the examples we've seen so far were single command aliases that worked on parts of the command that come after Git. It's also possible to run arbitrary code by prefacing the alias with an exclamation mark. Large, complicated interactions can be defined by creating a shell function and immediately invoking it within the alias. As with status visibility, using aliases is about letting Git fade into the background while you go about your work. It's time to look at a few next steps for you as you leverage Git configuration and attributes in your workflow.