## Client-side Hooks

### Introduction

Welcome to the first of two modules about git hooks. In this module, we will be talking about client‑side hooks, those that go into the client side of the Git repository. So, let's get started!

### Module Overview

Okay, before we move any further with this, let us take a minute to discuss some of the items that we'll be talking about in this module. We will start this module by talking about some of the basic concepts about Git hooks. We will then move on to the next section of this module to talk about a handful of hooks that are defined on the client side of the Git repository. And we're going to be mainly talking about hooks that are related to the commit process. There are three of them. And then we're going to close this module and talk about two more hooks on the client side, one that is related to the rebase operation and one that is related to the checkout.

### Basic Concepts

Okay, some basic concepts about Git Hooks. Let us start with the definition. Git Hooks are extensibility points that are triggered when specific Git actions happen or when specific Git workflows are triggered, like the commit workflow, as we're going to see later in this module. There are several hooks that will be invoked by Git when it executes the commit command. There are several types of hooks that are defined by Git, and which I'm going to group by their name prefixes. The first name prefix is pre, which includes all the hooks that are called by Gits before Git commands are actually started, and these hooks have the capability of aborting the Git command if they exit with a non‑0 exit code. The second group of hooks are those that start with the post prefix, and as the name indicates, these hooks are called after the Git command has finished executing. And as a result, their exit code is not going to affect the Git command. The third group are a miscellaneous set of hooks that are called by Git during the execution of Git commands. And typically, Git uses these hooks to obtain specific data that it needs during the execution of the Git command, and also. these hooks have the capability of aborting the Git command if they exit with a non‑0 exit code. One of the main workflows in Git, where all three of those types of hooks exist, is the commit workflow, which is basically the process of applying changes to the Git repository. And so the process starts by Git invoking a hook that is called the pre‑commit hook, which can be used to do any modifications to the changes before they are committed to the Git repository, such as applying code styles or do any validations and checks. And if this hook exits successfully, the actual commit process starts. However, if it exits with a non‑0 exit code, which means that it has failed, then the entire commit workflow is going to be aborted, and so this hook falls into the first type of hooks that we have discussed previously. After that, Git is going to invoke another hook to obtain the default commit message for that commit. If this hook successfully exits, then Git is going to present the committer with an editor to edit the default commit message. However, if this hook fails, then the entire commit workflow is going to get aborted. Then before the changes are fully written to the Git repository, Git is going to invoke another hook to validate the commit message. If that hook exits successfully, then the changes are going to be written to the Git repository. But if it fails, then the entire commit workflow is going to be aborted. The previous two hooks fall into the third type of hooks that we have discussed previously. And after the commit workflow finishes, Git is going to invoke one final hook. And the exit code of this hook does not have any effect on the commit process, and so this hook falls into the second type of hooks that we discussed previously. Now speaking about the nature of Git Hooks, it is very important that we know that Git Hooks are ultimately plain old executable, and this is what all Git cares about. It cares about an executable that it can run. And this means, that we can basically write the Git Hook using any programming language that we would like, that could be either Perl, Python, or even a Bash script, or we could even go with, you know, languages such as C#, Java, whatever you'd like. At the end of the day, all that Git cares about is that there is an executable that it can run, okay? When Get runs the executable, it's going to use the primitive ways of communicating with that process, and that means it's going to use the command line arguments, the input stream, the output and error streams to receive results from the Git Hook, and even environment variables, okay? And also, Git is going to check for the exit code of that executable to determine whether it succeeded or failed. And since hooks are running at the Git level, they are neutral to the IDE that the developer is using, so you could basically use whatever IDE you'd like whenever that IDE integrates with Git and Git decides that it needs to invoke the Git Hook, the Git Hook will get executed. Finally, since this is an executable again, Git is going to set the working directory for that executable to be the root of the Git repository on which the hook is installed. Another important aspect of Get Hooks is the fact that some of them are intended to be executed on the client side, while the others are intended to be executed on the server side. The main difference between the two groups is the fact that those that run on the client side are under the total control of the developer, and they could be easily bypassed or disabled in effect by the developer or even forgotten, so their sole purpose would be for the developer's own productivity and maybe defensive style to keep up with the project standards. However, the server‑side hooks, even though they can still be easily disabled by removing them from their location on the filesystem, they cannot be bypassed by client‑side switches or arguments, and their main use case would be to enforce server‑side policies over the Git repository.

### Demo: First Look at Git Hooks

In this demo, we're going to take a first look at Git hooks. And so I'm starting work on a calculator project, and I have decided to use Git for it as my source control management system. And if we take a look at the .git folder, there is a subfolder inside it called hooks that Git has repopulated with a set of files that end with the .sample extension. For every new Git repository, Git is going to take a copy of these files from a shared folder called usr/share/git‑core/templatess/hooks, and it also provides configs to changing both of these locations, the one at the repository level and the shared folder. And all of these files are shell scripts. And since all that Git cares about is that it has an executable in that folder, I am going to create for myself a .NET console application. And for now, I'm just going to only read the ProcessName and the standardInput, and I'm going to record the time of invocation of my executable inside a file called hooks‑trace.log. And I'm going to create a file by the name of my process, and I'm going to write in there the environment variables, command line arguments, and the standard input that were passed to me by Git. And so if I go back now to my calculator project and do a simple commit, you can see that the hooks‑trace file was created. And there were also three log files that were created by three different hooks that were invoked by Git. And if we take a look at the hooks‑trace file, we can see that there were indeed three hooks that were invoked by Git. The first one is called pre‑commit, second one, prepare‑commit‑msg, and the final hook is called commit‑msg.

### Demo: pre-commit Hook

In this demo, we're going to be talking about the pre‑commit hook, which is the first hook that is going to be invoked by Git during the commit operation. The hook is given a snapshot of the changes that are going to be committed to Git, so it is typically used to do different code checks and validations or even applying code styles. And so I'm going to use this hook to run my test cases and make sure that all of them pass before my commit is written to my Git repository. And so when this hook is executed by Git, I am going to, in turn, execute the dotnet test command, and I'm going to also configure this command to output the test results to a folder that is going to be called TestResults. And if the dotnet test command exists with a non‑zero exit code, which means that some of the test cases failed, I am going to also exit with the same exit code, which is in effect going to abort the commit process. And so going back to my calculator project, I have added a test case for a nonexisting functionality and so I do expect that test case to fail. So if I try and go ahead and do a commit adding a test case, the pre‑commit hook is going to run the test cases, and the test case is going to fail, and the commit command is also going to fail, and my changes will remain staged. But in some cases such as this one, I would like to be able to separate the commits that, test cases from the commits that implement them, so I don't really want the pre‑commit hook to keep blocking my commit process because I don't have passing test cases. And so I'm going to go back to my commit, but now add the no‑verify switch, which is going to tell Git to bypass the pre‑commit hook. And so you can see that now my commit successfully goes through. You could also see that the commit‑msg hook was also bypassed by the no‑verify switch; however, the prepare‑commit message hook was not affected by that switch.

### Demo: prepare-commit-msg Hook

In this demo, we're going to be talking about the second hook that is going to be invoked by Git during the commit process, which is the prepare‑commit‑msg hook. And so if you go ahead and take a look at the contents of the log file that this hook has generated, we can see that Git has passed this hook two command line arguments. The first one is a file path to a file that contains the commit message and the second argument is a string value that indicates the source of the commit message inside the file. And in this case, it's going to be message because I have specified the message as a parameter to the git commit command. However, in the cases where I call the git commit command without specifying a commit message, the prepared commit message hook can be used to populate the committed message file with a default commit message that is going to be presented to the committer. And so going back to my git‑hook implementation, I am going to define a very simple mapping between user emails and their active task descriptions. And so when the prepare‑commit‑msg hook executes, I am going to get the commit AUTHOR EMAIL from the environment variables, and then I am going to use that to look up the active task description and then populate the commitEditMsgFile with this task description. And so in my calculator project, I have implemented the functionality to make the test case pass, so if I go ahead now and do a git/COMMIT, you can see that Git now has prompted me to enter the commit message because I didn't provide one when I call in the git/COMMIT command. And as you can see, the default commit message now has been populated by the value that was provided by the prepare‑commit‑msg hook. And so I am going to accept this default commit message because it correctly describes what I have changed. And so if I save and exit, you can see that my commit has been successfully completed.

### Demo: commit-msg Hook

In this demo, we're going to talk about the commit‑msg hook, which is the last hook that is going to be invoked by Git before the Git command successfully finishes. And if we go ahead and take a look at the log file that this hook has generated, we can see that Git has passed to it a single command line argument to the same COMMIT\_EDITSMG file that was passed to the prepare‑commit‑msg hook, and so in this file now is the final version of the commit‑msg that is going to be added to the commit before it's written to the Git repository. And so I'm going to use the commit‑msg hook to make sure that the commit message now has a valid task number because one of the project standards that all commit messages should have valid task IDs in them. And so going back to my Git Hook project, and when the commit‑msg hook is going to be invoked, I am going to read the contents of the commitEditMsgFile and check that it has a valid task number. And if it does not, then I'm going to exit with a non‑0 exit code to abort the Git commit operation. And so now in my calculator project, I have another commit to make. So if I go ahead and do a git commit, and I am going to accept this default commit message because it describes what I'm still working on. However, I'm not going to include any task ID in this commit message. So if I go ahead and save and exit, you can see that my commit was aborted. And if I check my changes, I can see that they are still staged.

### Demo: post-checkout Hook

In this demo, we're going to talk about the post‑checkout hook, a hook that is invoked by Git whenever repositories are cloned, files are checked out, and even when switching between branches. And so, going back to my calculator project, I can see that the more commits I do to my Git repo, the more log files that are being generated by my hooks, and the more test result files that are generated by the dotnet test command. And so I definitely need to get rid of this clutter, especially when I'm switching between branches, because I want to be able to distinguish which test results logs were generated on which branch and also which hook log files were generated on which branch. And so going back to my Git Hook project and inside a function that is going to be executed when the post‑checkout hook is going to get invoked, I am going to delete the log files that were generated by the commit‑related hooks. I am also going to delete the TestResults folder. And finally, delete the hooks‑trace.log file. And so back to my calculator project. And if I wanted to switch to one of my other branches, we can see that the TestResults folder and the other log files that we had in there were actually deleted.

### Demo: pre-rebase Hook

In this demo, we're going to talk about another Git hook called pre‑rebase, and this hook is going to get invoked by GIT whenever branches are rebased. And, so I'm going to use this hook in my GIT repository to prevent any branch rebase operations because rebase operations rewrite the history of source control, and so, in general they are considered dangerous operations. And, so in my Git hook project, whenever the pre‑rebase hook gets invoked by GIT, I am going to immediately exit with a non‑0 exit code, which in effect is going to abort the rebase operation. And, so if I try to rebase my master branch onto one of my other branches, the pre‑rebase hook is going to reject that and is going to abort the rebase operation. And, if we go ahead and take a look at the log file that the pre‑rebase hook has generated, we can see that GIT has passed as a command‑line argument, the name of the branch that we were rebasing on.

### Module Summary

We started this module by talking about some of the basic concepts of Git Hooks. And then we talked about the hooks that are used at the client side of the Git repository. And then we discussed some of the different use cases. But the fundamental takeaway from this module is that client‑side hooks are under the full control of the developer, and so they cannot be used to enforce any kind of repository policies. And that's the reason why we need server‑side hooks, which we are going to talk about in the next module.

## Server-side Hooks

### Module Overview

Welcome to the second module about Git hooks. In this module, we're going to talk about server‑side hooks, and so we're going to take a look at a few of the server‑side hooks that will be invoked by Git when we do a Git push operation, and then we will see how we can use those hooks to enforce a few server‑side policies over our Git repository.

### Demo: First Look at server-side Hooks

In this demo, we're going to take a first look at server‑side hooks, and we're going to do something similar to what we have done with client‑side hooks. And so after I have installed my hooks on the server repository, I am going to go ahead and push all my branches. And so going to the server repository, and if we enumerate the contents of the repository folder, we can see that the hooks‑trace file was generated. And we have also three hooks that were invoked by Git and that have written out log files. The first one is the pre‑receive hook, and then we have two log files from a hook called update, and then we have another hook called post‑receive. And we can verify that this was indeed the order of execution by taking a look at the contents of the hooks‑trace.log file. And if we take a look at the contents of the pre‑receive log file, we can see that it has received in the Standard Input two lines, one line for each branch that we have pushed from the client side. And if we take a look at the first log file from the update hook, we can see that it has received now in the command line arguments the same data that is in the pre‑receive log file for the first branch that we have pushed. And the second update log file includes the second line from the pre‑receive log file. And finally, the post‑receive hook log file includes the same data that is in the pre‑receive log file.

### Demo: pre-recieve Hook

In this demo, we're going to talk about the pre‑receive hook. And as we have seen in the previous demo, the pre‑receive hook receives in the standard input a line for every branch that is being updated by the client. And also, for every branch it receives the range of commits that are being pushed by the client. And so, it is generally a very suitable place to perform global validations and checks for all the branches that are being updated by the clients. And so I'm going to use this to enforce a global policy that any commit that is being pushed to any branch must have a task number in its message. And so inside a function that is going to get executed specifically for the pre‑receive hook, I'm first going to parse the lines that are passed to me by Git, and remember that each line belongs to a branch that is being updated by the client. And then I'm going to scan the range of commits that are being pushed to this branch and extract the commit message. And then, if the commit message doesn't have a task number in it, I'm going to exit with a non‑0 exit code to abort the push operation. And so, in my calculator project, I have merged the subtraction branch into the master branch, and as part of that merge, a couple of commits have been added to the master branch that have no task numbers in them. And so if I try to go ahead and push the master branch, you can see now that the pre‑receive hook has rejected the push operation because one of the commits doesn't have a task number in its message.

### Demo: update Hook

In this demo, we will be talking about the update hook, which is the second hook that is going to get invoked by Git during the push operation. And, in a previous demo we have seen that this hook is going to get invoked for every branch that is being updated by the push operation. And, each time this hook gets invoked, it's going to receive in the command‑line arguments, the name of the branch that is being updated and also the start and end commit IDs of the range of commits that are being pushed to that branch. And so it's the best place to perform any branch‑specific validations and checks before accepting the commits that are being pushed to that branch. And so, in my case, I am going to use this hook to implement access controls to my branches on the server to allow for each branch only a subset of users to push commits to that branch. And so I am going to start by defining an Access Control List to my branches, and in my Access Control List, I am going to only allow push commands to the master branch and I'm going to only allow myself to push commits to this branch. And so, when the update hook is going to get invoked by Git, I am going to first check whether the branch is allowed to receive commits, and if it does, then I'm going to scan the range of commits that are being pushed to that branch, and for each commit I'm going to extract the authorEmail and make sure that the email is in the list of allowed users. And if it's not, then I'm going to exit with a non‑0 exit code to reject the push operation. And now, back to my calculator project, I I have another commit to make; however, if we take a look at that commit, we can see now that the authorEmail of that commit is not my email. And so if I try and push that commit to the master branch on the server, you can see that the update hook has rejected that because the email in the commit isn't allowed to push commits to the master branch.

### Demo: post-recieve Hook

In this demo, we're going to talk about the post‑receive hook, which is the last hook that is going to get executed by Git during the push operation. And in a previous demo, we have seen that this hook receives the same data as that of the pre‑receive hook. However, I'm not going to use this information to do any kind of validations and checks. I'm just going to use the fact that this hook is invoked at the end of the push operation to incorporate a simple, continuous‑integration functionality within my server‑side Git repo. And so when this hook is going to get executed by Git, I am going to clone the server‑side repo into a workspace folder and then I'm going to run a dotnet build on the server side. And so in my calculator project, I have one more commit to make my calculator usable from the command line. And so if I go ahead and push that commit to the server side, the post‑receive hook has finished cloning the repository and building it on the server. So if I go to the build location and check the output, I can see that my calculator has now been successfully built. And trying to use it, I can see that it actually works.