Git Hooks

Git Hooks: Git hooks are scripts that run automatically every time a particular event occurs in a Git repository.

There are two groups of Git hooks:

1. client-side / local hooks, which are prompted by events on the local repository, such as when a developer commits or merges code.
2. server-side / remote hooks, which are run on the network hosting the repository, and they are prompted by events such as receiving pushes.

Local Hooks

In this section, we'll be exploring 6 of the most useful local hooks:

1. pre-commit
2. prepare-commit-msg
3. commit-msg
4. post-commit
5. post-checkout
6. pre-rebase

Server-side Hooks

There are 3 server-side hooks that we'll be discussing in the rest of this article:

1. pre-receive
2. update
3. post-receive

Hook Name	Event	Description
pre-commit	git commit	This hook is called before obtaining the proposed commit message. Exiting with anything other than zero will abort the commit. It is used to check the commit itself (rather than the message).
prepare-commit-msg	git commit	Called after receiving the default commit message, just prior to firing up the commit message editor. A non-zero exit aborts the commit. This is used to edit the message in a way that cannot be suppressed.
commit-msg	git commit	Can be used to adjust the message after it has been edited in order to ensure conformity to a standard or to reject based on any criteria. It can abort the commit if it exits with a non-zero value.
post-commit	git commit	Called after the actual commit is made. Because of this, it cannot disrupt the commit. It is mainly used to allow notifications.
pre-rebase	git rebase	Called when rebasing a branch. Mainly used to halt the rebase if it is not desirable.
post-checkout	git checkout git clone	Run when a checkout is called after updating the worktree or after git clone. It is mainly used to verify conditions, display differences, and configure the environment if necessary.

Default file format.

1. applypatch-msg.sample
2. post-update.sample
3. pre-rebase.sample
4. commit-msg.sample
5. pre-applypatch.sample
6. update.sample
7. pre-push.sample
8. pre-commit.sample
9. prepare-commit-msg.sample
10. fsmonitor-watchman.sample
11. pre-merge-commit.sample
12. pre-receive.sample
13. push-to-checkout.sample

To make it executable file

1. chmod +x pre-commit
2. chmod +x prepare-commit-msg
3. chmod +x commit-msg

Different type of script

1. Python is located in /usr/bin. So, the first line would look like:

   #!/usr/bin python

2. If you want to use Bash, on the other hand, the first line would be:

   #!/bin/bash

3. And for shell:

   #!/bin/sh

Note: To enable hooks in your local repository: git config core.hooksPath .githooks

Format of commit message?

Note: git commit -m "feat(scope): Jira-001 add new functionality"

git commit -m "fix: Jira-001 description summary"

git commit -m "merge testing_hooks into feature/testing_hooks"

Branch Naming?

1. **feature/**description
2. **bugfix/**description
3. **prerelease/**description
4. **release/**description
5. **hotfix /**description
6. **master**
7. **develop**

```
CDW-6536: My custom type with JIRA issue id

Feat(scope): (new feature for the user, not a new feature for build script)
Fix(scope): (bug fix for the user, not a fix to a build script)
Docs(scope): (changes to the documentation)
Style(scope): (formatting, missing semi colons, etc; no production code change)
Refactor(scope): (refactoring production code, eg. renaming a variable)
Test(scope): (adding missing tests, refactoring tests; no production code change)
Chore(scope): (updating grunt tasks etc; no production code change)

Feat: (new feature for the user, not a new feature for build script)
Fix: (bug fix for the user, not a fix to a build script)
Docs: (changes to the documentation)
Style: (formatting, missing semi colons, etc; no production code change)
Refactor: (refactoring production code, eg. renaming a variable)
Test: (adding missing tests, refactoring tests; no production code change)
Chore: (updating grunt tasks etc; no production code change)
```

Hook Name	Invoked By	Description	Parameters (Number and Description)
applypatch-msg	`git am`	Can edit the commit message file and is often used to verify or actively format a patch's message to a project's standards. A non-zero exit status aborts the commit.	(1) name of the file containing the proposed commit message
pre-applypatch	`git am`	This is actually called *after* the patch is applied, but *before* the changes are committed. Exiting with a non-zero status will leave the changes in an uncommitted state. Can be used to check the state of the tree before actually committing the changes.	(none)
post-applypatch	`git am`	This hook is run after the patch is	(none)

		applied and committed. Because of this, it cannot abort the process, and is mainly used for creating notifications.	
pre-commit	`git commit`	This hook is called before obtaining the proposed commit message. Exiting with anything other than zero will abort the commit. It is used to check the commit itself (rather than the message).	(none)
prepare-commit-msg	`git commit`	Called after receiving the default commit message, just prior to firing up the commit message editor. A non-zero exit aborts the commit. This is used to edit the message in a way that cannot be suppressed.	(1 to 3) Name of the file with the commit message, the source of the commit message (`message`, `template`, `merge`, `squash`, or `commit`), and the commit SHA-1 (when operating on an existing commit).
commit-msg	`git commit`	Can be used to adjust the message after it has been edited in order to ensure conformity to a standard or to reject based on any criteria. It can abort the commit if it exits with a non-zero value.	(1) The file that holds the proposed message.
post-commit	`git commit`	Called after the actual commit is made. Because of this, it cannot disrupt the commit. It is mainly used to allow notifications.	(none)

pre-rebase	`git rebase`	Called when rebasing a branch. Mainly used to halt the rebase if it is not desirable.	(1 or 2) The upstream from where it was forked, the branch being rebased (not set when rebasing current)
post-checkout	`git checkout` and `git clone`	Run when a checkout is called after updating the worktree or after `git clone`. It is mainly used to verify conditions, display differences, and configure the environment if necessary.	(3) Ref of the previous HEAD, ref of the new HEAD, flag indicating whether it was a branch checkout (1) or a file checkout (0)
post-merge	`git merge` or `git pull`	Called after a merge. Because of this, it cannot abort a merge. Can be used to save or apply permissions or other kinds of data that git does not handle.	(1) Flag indicating whether the merge was a squash.
pre-push	`git push`	Called prior to a push to a remote. In addition to the parameters, additional information, separated by a space is passed in through stdin in the form of "<local ref> <local sha1> <remote ref> <remote sha1>". Parsing the input can get you additional information that you can use to check. For instance, if the local sha1 is 40 zeros long, the push is a delete and if the	(2) Name of the destination remote, location of the destination remote

		remote sha1 is 40 zeros, it is a new branch. This can be used to do many comparisons of the pushed ref to what is currently there. A non-zero exit status aborts the push.	
pre-receive	`git-receive-pack` on the remote repo	This is called on the remote repo just before updating the pushed refs. A non-zero status will abort the process. Although it receives no parameters, it is passed a string through stdin in the form of "<old-value> <new-value> <ref-name>" for each ref.	(none)
update	`git-receive-pack` on the remote repo	This is run on the remote repo once for each ref being pushed instead of once for each push. A non-zero status will abort the process. This can be used to make sure all commits are only fast-forward, for instance.	(3) The name of the ref being updated, the old object name, the new object name
post-receive	`git-receive-pack` on the remote repo	This is run on the remote when pushing after the all refs have been updated. It does not take parameters, but receives info through stdin in the form of "<old-value> <new-value> <ref-name>". Because it is called after the	(none)

		updates, it cannot abort the process.	
post-update	`git-receive-pack` on the remote repo	This is run only once after all of the refs have been pushed. It is similar to the post-receive hook in that regard, but does not receive the old or new values. It is used mostly to implement notifications for the pushed refs.	(?) A parameter for each of the pushed refs containing its name
pre-auto-gc	`git gc --auto`	Is used to do some checks before automatically cleaning repos.	(none)
post-rewrite	`git commit --amend`, `git-rebase`	This is called when git commands are rewriting already committed data. In addition to the parameters, it receives strings in stdin in the form of "<old-sha1> <new-sha1>".	(1) Name of the command that invoked it (`amend` or `rebase`)

What is a changelog?

A changelog is a kind of summary of all your changes. It should be easy to understand both by the users using your project and the developers working on it.

Note: git log > CHANGELOG.md