**What are Git Hooks?**

A custom action (script) triggered after certain git event/action is occurred.

* Client-side - triggered by operations such as committing and merging
* Server-side -  run on network operations such as receiving pushed commits
* The hooks are all stored in the **hooks** subdirectory of the Git directory.( .git/hooks)
* If you want to use the bundled hook scripts (example script/hooks after **git init**), you’ll have to rename them; their file names all end with **.sample**.

### Client-Side Hooks

* Client-side hooks are **not** copied when you clone a repository.
* Client Side Hooks reside on one’s local repository and are executed when a git event is triggered.

(commit, push, rebase etc.)

#### Committing-Workflow Hooks

The first four hooks have to do with the committing process.

* + **pre-commit**
    - Run first, before you even type in a commit message
    - Used to inspect the snapshot that’s about to be committed, to see if you’ve forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code.
    - Exiting **non-zero** from this hook **aborts the commit**, although you can bypass it with **git commit --no-verify**
  + **prepare-commit-msg**
    - Run before the commit message editor is fired up but after the default message is created.
    - It’s good for commits where the default message is auto-generated
  + **commit-msg**
    - Hook takes one parameter
  + **post-commit**
    - Run after the entire commit process is completed.
    - **git log -1 HEAD**

#### Email Workflow Hooks

Invoked by the **git am** command.

* **applypatch-msg**
* Takes a single argument : the name of the temporary file that contains the proposed commit message
* Used to make sure a commit message is properly formatted, or the message is normalize by having the script edit it in place.
* **pre-applypatch**
  + Run after the patch is applied but **before** a commit is made.
  + Used to inspect the snapshot before making the commit.
  + We can run tests or otherwise inspect the working tree with this script.
  + **If** something is **missing** or the **tests don’t pass**, exiting **non-zero aborts** the **git am** script **without committing** the patch.
* **post-applypatch**
  + Runs after the commit is made.
  + Use to notify a group or the author of the patch you pulled in that you’ve done so.

#### Other Client Hooks

* + **pre-rebase**
    - Runs before you rebase anything and can halt the process by exiting non-zero.
    - You can use this hook to disallow rebasing any commits that have already been pushed.
  + **post-rewrite**
    - Run by commands that replace commits, such as **git commit --amend and git rebase** (though not by **git filter-branch**). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on **stdin**. This hook has many of the same uses as the **post-checkout** and **post-merge** hooks.
  + **post-checkout**
    - After you run a successful git checkout, the post-checkout hook runs;
    - You can use it to set up your working directory properly for your project environment.
    - This may mean moving in large binary files that you don’t want source controlled, auto-generating documentation, or something along those lines
* **post-merge**
  + Runs after a successful **merge** command.
  + You can use it to restore data in the working tree that Git can’t track, such as permissions data.
  + This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.
* **pre-push**
  + Runs during **git push**, after the remote refs have been updated but before any objects have been transferred.
  + It receives the name and location of the remote as parameters, and a list of to-be-updated refs through **stdin**.
  + You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).
* **pre-auto-gc**
  + Invoked just before the garbage collection takes place,

### Server-Side Hooks

* You can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project.
* These scripts run before and after pushes to the server.
* The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that’s as complex as you wish.

#### pre-receive

#### The first script to run when handling a push from a client is pre-receives.

#### It takes a list of references that are being pushed from stdin;

#### if it exits non-zero, none of them are accepted.

#### You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they’re modifying with the push.

#### update

* The **update** script is very similar to the **pre-receive** script, except that it’s run once for each branch the pusher is trying to update.
* If the pusher is trying to push to multiple branches, **pre-receive** runs only once, whereas update runs once per branch they’re pushing to.
* Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push.
* If the update script exits non-zero, only that reference is rejected; other references can still be updated.

#### post-receive

* The **post-receive** hook runs after the entire process is completed and can be used to update other services or notify users.
* It takes the same stdin data as the **pre-receive** hook. Examples include emailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed.
* This script can’t stop the push process, but the client doesn’t disconnect until it has completed, so be careful if you try to do anything that **may take a long time**.