Advanced - Git 2
Git Hooks

EECS 201 Winter 2022

Some background: Git Hooks

One feature of most version control systems is **hooks**. A hook is an automated script or tool that runs at various points in time. For example, you can automatically run test cases every time anyone commits code.

A slightly easier one to wire up, however, is a hook that will automatically check the spelling of your commit messages for you, letting you know if you made any mistakes. While spelling rarely counts for a class projects, it adds a nice bit of professionalism to any future work you'll share with others.

I recommend creating a temporary Git repository to play with while you test things and try to get things working.

Git stores all of its information and configuration in a folder named `.git` in the root of each project. Navigate to `.git/hooks` and rename `commit-msg.sample` to `commit-msg`. This activates the commit message hook, which runs after you write and save your commit message. Now go back to your repository, make a change, and commit it. Hmm, doesn't look like anything changed. Make another change, but this time make this your commit message (**exactly this, capitalization matters**):

```
This is a test.

Signed-off-by: Me!
Signed-off-by: Me!
```

After trying to commit, type `git status`. What happened? Go back and look at the commit hook. Do you understand what it does? Try running `git commit --no-verify` with the same commit message. Go edit the commit hook and delete the line `exit 1`. Try making a new commit with the same commit message, what happens now? What is `$1` in this script? (not sure? try adding lines like `echo $1` or `echo $(file $1)` or `echo $(cat $1)` to the hook and making commits, what happens?)

Submission Instructions

Submission Instructions

This assignment will be submitted as a repository on the UMich GitLab server. Create a private, blank, **README-less (uncheck that box!)** Project on it with the name/path/URL `eecs201-adv-git2` and add `brng` as a Reporter. The submission branch will be `hooks`. If this branch is not already the default initial branch, you initialize the local repo with an additional argument: `git init --initial-branch=hooks` if your version of Git is recent enough. Otherwise you can create a branch with this name after your first commit. The repository should have the following directory structure, starting from the repository's root:

```
/
|-- commit-msg
|-- pre-commit
|-- report.txt
```

`commit-msg` and `pre-commit` are the hook scripts that you will be writing in the following sections. Unfortunately, Git does not version things inside the `.git` directory, so you'll have to copy them to a submission repo.

How to go about this

You can do whatever you want that results in a GitLab repo of the correct URL, with a branch called `hooks` with the above directory structure, but to make things a bit less confusing here's my suggested workflow:

1. Create a directory called `eecs201-adv-git2` to serve as the directory whose contents will be submitted. I will call this the "submission" directory from now on.
2. Initialize a Git repository inside of `eecs201-adv-git2` with the appropriate initial branch.
3. Set up the appropriate remote repository information.
4. Copy the `commit-msg.sample` and `pre-commit.sample` files from the newly initialized Git repo's `hooks` directory into the repository's root directory (the `eecs201-adv-git2` directory you just created). Rename them so they don't have the `.sample` extension.
5. Exit this directory. We won't be working with it directly until you're ready to submit.
6. Make sure you're outside of your `eecs201-adv-git2` submission directory.
   Download the
   `https://www.eecs.umich.edu/courses/eecs201/wn2022/files/assignments/adv-git2-test.tar.gz`
   archive, extract it, then initialize a Git repository inside of the extracted directory. This repository will serve as a testing bed for your hooks and its history will not be submitted: I will refer to it as the "testing repository" from now on. The files included won't be needed until problem 2, however.
7. Navigate to the `hooks` directory of your testing repository.
8. Create a symbolic link called `commit-msg` that leads back to the `commit-msg` file in your submission directory and do the same for the `pre-commit` script. A symbolic link is kind of like a shortcut to another existing file: it pretty much says "looking for a file? It's actually over there". If you are unfamiliar with them, you can check the last ungraded section of Basic - Shell or do a bit of reading on your own. It would looks something like `ln -s <path-to-submission-dir> <path-to-.git-directory-of-testing-repo>`
9. The symbolic links will allow you to "trick" Git in the testing repository to run the scripts over in your submission repository. This will allow you to edit your scripts and have them be usable in the testing repository while submittable in the submission directory without needing to copy them back and forth.
10. Perform all the testing of your scripts inside of your testing directory and leave the actual submission directory alone until it's time to write your report and submit your hooks.

Of course, you are free to ignore this and do whatever you want as long as the assignment is submitted properly. I just found this to be a neat use of symbolic links :)

1 Automating Professionalism (5)

`aspell` is a simple command-line utility that checks spelling. Install it and play with it a little.

Write a commit message hook that checks the spelling of a commit message. Your hook **should not** prevent the commit from going through (that'd be annoying...). Your hook should print out "Spelling errors detected!" and then print out a `sort`ed list of offending words, with one word per line.

Example output:

```
Spelling errors detected!
alkdj
dakfjll
dklak
ldkafjlk
notarealword
```

Some tips

- Remember to look only at the **commit message** itself. Commit message files have "commented out" stuff with lines beginning with `#` to communicate with the user what's going on in a commit, but they aren't in the commit message itself.
- This can be a little intimidating to get started. *Try some things.* Make a bunch of garbage commits, modify the hook a little, see what happens.
- Remember that a shell script is exactly like working in a terminal. The only magic is that you've typed all the commands in advance instead of one at a time. So try some things in your terminal! Mess around until you get some commands that do what you want, then copy them to your script.
- There is **zero** need to be efficient. This hook is called rarely and operates on hundreds of bytes of text. Read the file 6 times. Write 7 temporary files. *Who cares.* The goal is not to be pretty, the goal is to work.
- Speaking of temporary files, the `/tmp` directory can be a great place to throw those. There's even a command called `mktemp` to create temporary files or directories here.

Further exploration and some gotchas:

- Take a look at some of the other hooks, are any of them useful?
- For a list of all available hooks, type `man githooks`.
- Hooks have to be marked as executable to run (`chmod +x`). The sample hooks already had the executable bit set, which is why renaming the existing sample worked above.
- Hooks can call other scripts. Because invocation of hooks is controlled by the name of the script, if you want multiple scripts to run for a single hook, you'll need to have one script named correctly that calls your other hooks.
- Git hooks need not necessarily be shell scripts; they can be any sort of executable. You could possibly implement your hooks as, for example, Python scripts (not in this assignment though).
- Note that the shebang on the example hooks run the very basic `sh` shell. You may change this to `bash` if you want to use Bash's more advanced constructs.

2 Automating security (5)

When you are developing a web application, sometimes you need to access the services of some API to access things like weather information from one source or a user's Google calendar from another.

Such services tend to require the use of an *API key* that is used to authenticate the client requesting services and can be used to do things like rate limiting and tracking API use.

If someone else (say, another competing application) gets a hold of your application's API key, they could mooch off of whatever quotas that API key has.

Another security issue is when people unwittingly put private SSH keys into a repository. Recall that when you generated an SSH key it created a private and public key pair (e.g. public: `id_ed25519.pub`, private: `id_ed25519`.) The public key, being "public" is provided to various servers to authenticate your computer with. Your private key is what verifies that you are you: if it gets out of your hands, others can access things that have saved your public key for authorization. (This is simplifying it greatly: just know that your private key is private!).

Such things shouldn't necessarily be put onto a public Git repository being hosted somewhere (unless you actually intended for the entire internet to be able to use your key).

In this exercise you'll be writing a Git **pre-commit hook** that **stops** the commit from happening when an API key is directly in the staged code files or if a private SSH key file is staged. If your hook fails the commit, it should print out "Keys detected!" and a `sort`ed list of what files are the culprits, one file on each line. **Every** offending file should have its name printed out; do **not** exit early once a key is detected.

Example output:

```
Keys detected!
example_ed25519_private_key
example_key_assigns
```

Here are some things to identify each component with (these are simplified for this exercise).

API keys:

- They are a string literal enclosed by double quotes (`"`) or single quotes (`'`)
- They are a string composed of alphanumeric characters of both cases (A-Z, a-z, 0-9)
- They are being assigned via '`=`' to some variable whose name ends in "`key`" or "`Key`". There can be an arbitrary amount of whitespace around the '`=`'

SSH private keys:

- The file's first line is "`-----BEGIN OPENSSH PRIVATE KEY-----`"
- The file's last line is "`-----END OPENSSH PRIVATE KEY-----`"
- You may assume that a valid SSH private key must have both of these conditions met.

Helpful hints:

- The `https://www.eecs.umich.edu/courses/eecs201/wn2022/files/assignments/adv-git2-test.tar.gz` archive has an example private key and API key assignments. You may have downloaded this already if you followed my suggested workflow.
- You can get a list of staged files via `git diff --name-only --cached`.
- `head` and `tail` can get the first and last lines of a file.
- If you want to look up the manpage for the `[ ]` conditional, remember it's really just a shorthand for `test`: you can use `man test` to see how it works and what operators it has.
- If a `grep` pattern needs to start with a `-`, to differentiate the pattern from arguments you can put a `--` in between to signify the end of the optional/flag arguments. For example: `grep -E -- '--test--'`

3 Conclusion

1. Make sure your finished hook script files are in the right place.
2. Create a file called `report.txt`.
3. On the first line provide an integer time in minutes of how long it took for you to complete this assignment.
4. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".
5. Add and commit this `report.txt` file. Push your repo.