Harvard John A. Paulson School of Engineering and Applied Sciences

F. Wermelinger
Office: Pierce 211

Pair-Programming 2

Bash scripts and semantics, Example Git workflow

Issued:	September 9, 2022
Due:	September 23, 2022 11:59pm

In this pair-programming session you will write some simple Bash script to *automate* example workflows. Bash is a bit clumsy but it can be a very powerful tool for the automation of common tasks. Such scripts are often ran periodically via *cron*[1] on GNU/Linux.

You should work on the two exercises in groups of 3 to 4 students via a `tmate` session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.[2] Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Scripting a Git Workflow

Deliverables:

1. `lab/pp2/exercise_1.sh`

Write a Bash script to automate an example Git workflow. Write your script in the file `exercise_1.sh`. The specific tasks are listed below. The exercise is focused on Bash scripting but you will also be in touch with some basic Git commands. You may be unfamiliar with some of them at this point. We will study them in more depth in the lectures.

To test your script without breaking anything, you will setup two local git repositories to play around with. Create a sandbox directory and let Git ignore it[3]:

```
$ mkdir -p lab/pp2/sandbox
$ echo '/sandbox' >>lab/pp2/.gitignore
```

Now change into `sandbox` and create two Git repositories like this

[1] https://en.wikipedia.org/wiki/Cron

[2] https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp

[3] This assumes you are at the root in your private Git class repository. If you are using the class Docker image with `lab/pp2` mounted, simply do not type `lab/pp2` in the commands shown since in that case you are already located inside the `lab/pp2` directory.

```
$ mkdir remote work
$ (cd remote && git init --bare)
$ cd work && git init && git remote add origin ../remote
```

You should now be inside the `work` Git repository where you can play around without affecting your actual private Git repository. The task is to write a script `exercise_1.sh` (you can create it inside the `work` directory and move it later into `lab/pp2` when you are done) that can interactively add new files to the Git repository you have just created. For example:

```
$ echo 'Brand new file' >foo
$ ./exercise_1.sh
```

will start a dialogue with the user to add new files to the Git repository you are in. The script should do the following:

1. Prompt the user for a file to commit.
2. Stage the file with `git add`.
3. Display the status using `git status`.
4. Ask the user whether to continue with `Y` or `N`. If the input is `N` the script will exit using `exit 1` for example.
5. If the response is `Y` then you should prompt for a commit message next.
6. Commit the file to the sandbox repository `work` you have just created (you should run the script inside this directory). You can use `git commit -m <message>`.
7. Display the status using `git status`.
8. Ask the user whether to continue with `Y` or `N`. If the input is `N` the script will exit using `exit 1` for example.
9. If the response is `Y` perform a `git push` and exit.

Note that this is just an common Git workflow. In practice you would not write a script for this because the routine will become second nature such that you prefer to interact with `git` directly instead of via this script. Note that the command `git add .` (notice the ".") will add all untracked files recursively down from the current location. Using a command like this is *bad practice* as it will also pickup junk you may not want to stage. Refrain from using it.

Hint: *You can use the Bash built-in functions* `printf` *to print text and* `read f` *to read from the standard input. Try this in a separate terminal to test. The command* `echo $f` *will print what has been read. See* `man read` *as well.*

Please see `solution/exercise_1.sh` for the solution code.

Exercise 2: Bash Loop and File Permission Checks

Deliverables:

1. `lab/pp2/exercise_2.sh`

Write a Bash script that checks all the files in the directory and prints out the file name if it is executable by the *owner*. Save this script as `exercise_2.sh`.

For testing purposes, you should create an executable file in your working directory to make sure your script picks up on it. You should make use of the command substitution[4] feature in the Bash shell. Try the following command from the command line:

```
$ echo "$(ls)"
```

Note that the string quoted with "" *expands* shell variables or the output of command substitutions. Try the string version that is quoted with ” instead:

```
$ echo '$(ls)'
```

Hint: *In practice, the find command may be a better option for this task rather than writing a script (unless you have specific needs that must be met).*

Please see `solution/exercise_2.sh` for the solution code.

[4] https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html