Pro Git Supplement

Joshua B. Teves

November 9, 2022

Contents

Preface

This is a set of supplemental questions and exercises to work through in conjunction with Pro Git version 2, chapters 1 through 3. Pro Git is freely available at `https://git-scm.com/book/en/v2`.

1 Getting Started

1.1 About Version Control

1. What is the big weakness of Centralized Version Control?
2. In distributed version control, why is there no longer one point of failure?

1.2 A Short History of Git

1. Which community was motivated to create a new version control tool?
2. When was Git created?

1.3 What is Git?

1. Does Git store snapshots or differences?
2. Do you need to be online to use Git?
3. What are the three states that files in a repository can reside in?

1.4 The Command Line

1. Where is the only place you can run all Git commands?

1.5 Installing Git

1. What version of Git are you running?

1.6 First-Time Git Setup

1. What's the difference between `--global` and `--system` when running `git config`?
2. Why might you need to check your configuration settings?
3. Set up your git config
   (a) Name
   (b) Email
4. Verify that your name is correctly configured with 'git config'

1.7 Getting Help

1. Use `git help` to answer the following:
   (a) Is `git stash clean` a valid command?
   (b) Is `git stash drop` a valid command?
   (c) What flags can be used to show the short version of `git status`?

2 Git Basics

2.1 Getting a Git Repository

2.1.1 Questions for Understanding

1. What are the two standard ways of obtaining a git repository?
2. What do you receive when you clone a repository by default?

2.1.2 Exercises

1. Create a directory, 'recipes', and initialize it as a git repository.
2. Clone this git repository from the url below

```
https://github.com/nimh-sfim/nimh-sfim.github.io.git
```

2.2 Recording Changes to a Repository

2.2.1 Questions for Understanding

1. If you add a file to an empty repository, is it tracked or untracked?
2. Can you add folders with the 'git add' command?
3. When running `git status`, you can see that a file has been modified with changes to be committed, *and* modified with changes not staged for commit. What happened?
4. When running `git status -s`, what is the left column? The right?
5. What expression would you use to ignore all `.pdf` files in a repository in the `.gitignore`?
6. What's the difference between `git diff` and `git diff --cached`?
7. If you use `git commit` with no arguments, when does the commit get created?
8. If you modify a file or stage it, and then attempt to remove it with `git rm`,which flag must you add?
9. To stop tracking a file but retain it in the working area, what flag must you add to `git rm`?
10. Does git track whether files are renamed?

2.2.2 Exercises

Throughout the exercises, it is recommended that you routinely run `git status` in order to see how the repository views your changes. Exercises should be done in order.

1. Go to your recipes repository, which you made in the last chapter. Create a file called `README.txt`, and put a brief message in it, perhaps, "I like cookies." Stage, and then commit, the file.
2. Create a file, `cookies.txt`, and add some ingredients to it. Stage the changes. Then, go back to the file and add cooking instructions. Commit your changes, stage the remaining ones, and then commit them. This exercise is to reinforce the nature of the working and stagin areas.
3. Create a file, `ignoreme.txt`, and set up the repository to ignore it. (Hint: you will need to commit changes to the `.gitignore`). Commit the changes to the `.gitignore`. Check the working area and `git status` to verify that the file is present, but ignored by the repository.
4. Set up the repository to ignore all `.pdf` and `.docx` files.

5. Create a file, `hello.txt`, with “Hello, world!” in it, and then commit it. Stop tracking it without deleting the file from disk. What does the status say after you do so? After checking, unstage the change.

6. Change `hello.txt` to `hello_world.txt` without using `git mv`. Then, change it back using `git mv`.

2.3 Viewing the Commit History

2.3.1 Questions for Understanding

1. Which specifier for ‘pretty‘ gives the author email?

2. Why are there so many options for showing the git log?

2.3.2 Exercises

Use the ‘recipes‘ repository for this, even though there’s not much there.

1. In ‘recipes‘, view the patches of the latest four commits.

2. View all commits since last week.

3. View all commits since the first of the month, using the ISO datetime format (YYYY-MM-DD).

4. View all commits, with a format showing only:

    (a) the short hash
    (b) the author name
    (c) the commit subject

5. View the last three commits, with a format showing only:

    (a) short hash
    (b) commit subject

2.4 Undoing Things

2.4.1 Questions for Understanding

1. Should you amend a commit if it has been pushed somewhere else?

2. Does unstaging a file change its contents in the working directory?

3. In which version of Git was ‘restore‘ introduced?

4. Are changes made with

```
git checkout -- FILE
```

or

```
git restore FILE
```

recoverable?

2.4.2 Exercises

1. Make a commit with a typo, such as "I lik cookies." Then, fix the typo and amend the commit. Try the same with a typo in the commit message itself. What do you notice about the commit hash afterwards?
2. Stage a commit with some typo. Correct the typo incorrectly, such as "I lik cookies" to "I likee cookies." Then try
   (a) Discarding the second typo and
   (b) Unstaging the first.

   Note how after the unstage operation, the typo remains.

2.5 Working with Remotes

2.5.1 Questions for Understanding

1. Can a remote be on the same local machine?
2. What is the remote name that is added when you perform a ‘git clone‘ operation?
3. What is the syntax for adding a new remote?
4. How do you list all remotes?
5. What is the difference between ‘fetch‘ and ‘pull‘?
6. Are changes automatically sent to the remote?
7. How do you modify remote names?
8. If you delete a remote, do your branches continue to track it?

2.5.2 Exercises

1. You should have cloned this repository previously. If not, clone it now at:

   ```
   https://github.com/nimh-sfim/nimh-sfim.github.io.git
   ```

   Check to see which branches are configured to push and pull.
2. Fetch the repository and list all branches fetched.

2.6 Tagging

2.6.1 Questions for Understanding

1. Do tags update with a branch?
2. Can lightweight tags have messages?
3. Are tags automatically synchronized with remotes?
4. When checking out a specific tag, what type of state are you in? What's one way to get out of it?

2.6.2 Exercises

1. On the recipes repository, tag the very first commit as "v0.0"
2. Check out the tag "v0.0" and then add another commit. Then, return from the detached HEAD state by checking out 'main'.

2.7 Git Aliases

2.7.1 Questions for Understanding

1. What is the syntax for adding an alias from the command line?

2.7.2 Exercises

1. Add the unstage alias recommended in the section. It is recommended to type instead of copy for understanding.
2. Create each of the following aliases as specified:
   (a) `slog`: an alias for a git log which has only the short hash and commit subject line.
   (b) `stat`: an alias for a short git status

3 Git Branching

3.1 Branches in a Nutshell

3.1.1 Questions for Understanding

1. What is a branch pointing to?
2. Do commits point to their "children"?
3. When you create a new branch, do you automatically move to it?
4. When you check out a branch, what special pointer is moved?

5. When you make a new commit on a particular branch, are other branches updated?
6. What does it mean when two branches' history diverges?
7. As of Git v2.23, how can you switch to the previous branch?

3.1.2 Exercises

1. In recipes, create and switch to a branch called 'cake'. Make some commits on it to create a recipe for cake, and then switch back to the 'main' branch.

3.2 Basic Branching and Merging

3.2.1 Questions for Understanding

1. Can you switch branches with modified but uncommitted changes?
2. Does your working area change when you change branches?
3. What is a "fast-forward"?
4. How do you delete a branch?
5. In the example given in "Basic Merging," why is the merge not a "fast-forward"?
6. Do merge commits have one parent?
7. If there is a merge conflict, is a merge commit made?
8. What command can you use to find out which files have merge conflicts?
9. How can you tell which branch one side of the merge conflict came from?
10. Will `git status` indicate if you've finished merging conflicts?

3.2.2 Exercises

Do exercises in order in the recipes repository.

1. Create and checkout to a new branch, `muffins`. Create a recipe for muffins on the branch, and then merge it into main. What kind of merge was this?
2. Create and checkout to a new branch, `cake2`. Make a cake recipe that is identical to the one in the branch `cake`, but change the ingredients or baking instructions slightly. Merge `cake` into `main`, and then merge `cake2` into main. You will need to resolve the merge conflict.
3. Delete all branches except for `main`.

3.3 Branch Management

3.3.1 Questions for Understanding

1. What do you get if you run `git branch` with no arguments?
2. Can you delete a branch with `git branch -d` if it isn't merged?
3. What command would you use to find out which branches aren't merged with the branch `fluffy`?
4. What are the dangers of renaming a branch?
5. How do you delete a branch on a remote?

3.3.2 Exercises

Do exercises in order in the recipes repository.

1. Create a branch called `salad`. Add a salad recipe and commit it to the branch. Then, switch to main. Rename the branch to `salad_recipe`. Switch to the renamed branch.

3.4 Branching Workflows

3.4.1 Questions for Understanding

1. Rank the stability of `master` (note: in most new workflows, `main`), `develop`, and `topic` in Figure 27.
2. In Figure 29, why are commits C2 and C4 included in `master`, when they were created in the now-deleted branch `iss91`?

3.4.2 Exercises

Do exercises in order in the recipes repository.

1. Similar to Figure 29, do the following:
   (a) Create a new branch, `toast`, with a commit holding just the ingredients for a toast recipe.
   (b) Branch off of `toast` to create a branch `toast_v2` that has a recipe for toast that you think is fancy. (Consider "French Toast" or "Cinnamon Toast.")
   (c) Return to the original toast and complete a recipe for plain toast on that branch.
   (d) You want the fancy stuff! Delete the original `toast` branch and merge in `toast_v2`. View the history and make sure you understand why it looks the way it does.

3.5 Remote Branches

3.5.1 Questions for Understanding

1. What flag do you use to override the default name `origin` when running `git clone`?
2. Why might there be several remotes for a project?
3. State the difference between the following:

   ```
   git checkout -b branch remote/branch
   ```

   and

   ```
   git checkout --track remote/branch
   ```

   and then explain why you might choose one over the other.
4. According to the text, it is preferable to use `fetch` and `merge` rather than `pull` by itself. Why might it be confusing to use `pull`?

3.5.2 Exercises

Do exercises in order in the recipes repository.

1. Create a remote repository and add it as a remote. Call it `test`.
   (a) Create a branch, `eggs`, and put a recipe for scrambled eggs on it. Push it to `test`.
   (b) Checkout to `main` and create a new branch, `fried_eggs`. Push it to `test`.
   (c) Delete your local `eggs` and `fried_eggs` branches, then fetch them from `test`. Note: the ability to do this is one of the strengths of `git`.
   (d) Merge `eggs` and `fried_eggs` into `main`, and push `main` to the `test`.
   (e) Delete `eggs` and `fried_eggs` both locally and on the remote.

3.6 Rebasing

3.6.1 Questions for Understanding

1. Why does rebasing appear to make the history cleaner?
2. What will attract the scorn of friends and family?
3. What are two cases in which rebasing is fine?
4. Discuss the pros and cons of rebasing vs. merging.

3.6.2 Exercises

Do exercises in order in the recipes repository.

1. Clean rebase
   - (a) Make two branches, one called `bread` and one called `sandwich`.
   - (b) Create the `sandwich` recipe while referencing the incomplete `bread` recipe.
   - (c) Switch to the `bread` recipe and complete it.
   - (d) Rebase `bread` onto `main` and merge it.
   - (e) Rebase `sandwich` onto `main` and merge it.
   - (f) View the history. Note how it implies that `bread` came first in history, even though you actually made the sandwich recipes first.
2. Scorn-inducing rebase
   - (a) Create a branch, `broccoli`, with instructions on making broccoli. Make sure the branch has several commits.
   - (b) Push `broccoli` to `test` (as set up in the previous chapter).
   - (c) Locally, rebase `broccoli` onto `main`.
   - (d) Add an additional recipe on your `broccoli` branch.
   - (e) Attempt to push to the `test` remote with your `broccoli` branch.
   - (f) Force-push the new `broccoli` branch to the remote.
   - (g) What would happen if somebody had already based new work after your first push? In this scenario, is it likely that something like that has happened? What if the project was larger?