Maintaining a Good Commit History

 (https://github.com/learn-co-curriculum/git-github-commit-messages)  (https://github.com/learn-co-curriculum/git-github-commit-messages/issues/new)

Learning Goals

- Follow best practices for committing your work.
- Follow best practices for writing commit messages.

Introduction

Maintaining a clean, well-documented commit history is important for several reasons:

- It provides documentation of your project's history.
- It makes it much easier to find, view, and modify changes that were made in the past. (We'll talk more about this in the next lesson.)
- It's a skill that many potential employers will be looking for.

In this lesson, we'll talk about best practices for making commits and creating a good commit history.

When to Commit

Commit often! You want to keep your commits relatively small. In fact, you could conceivably have a commit in which only a single character has been changed! (e.g., fixing a typo).

There are a couple of reasons for keeping your commits small. First, if you've been coding along for a while without committing and then something goes terribly wrong, you may need to revert to an earlier version of your code. If that happens, and you've written dozens or even hundreds of lines of code since your last commit, that's a lot of time and work lost!

The second reason to keep commits small is that it makes it much easier to track down problems that may arise. Imagine that, as you're working on your code, you discover that a bug got introduced at some point, but you aren't sure when. One way to track it down is to back up through

your commit history, commit by commit, until you find the last commit before the bug was introduced. You can then start searching through the changes in the next commit to see if you can find the problem. If that commit is small — a few lines, say — it will be much easier to find and fix the bug than if there are dozens of lines of code to check.

It is easy to get involved in coding and forget to commit — it happens to all developers, even experienced ones! The sooner you get in the habit of making regular small commits, the more headaches you'll avoid down the line.

Keep Your Commits Specific

Ideally, all the changes saved in a single commit should be related to a single task, fix, or feature. Some examples:

- Adding references for information included in the README.
- Fixing a bug in a particular workflow.
- Updating a broken link.
- Modifying the CSS to use a new color palette.

Doing this has two big benefits. First, using well-defined, specific commits (combined with meaningful commit messages, which we'll talk about next), will give you very specific and complete documentation of the history of your code. Second, if you need to go back and revert changes related to some fix or feature, the task is much easier if you know 1) all the code is encapsulated within one commit (which you can find easily because you've used a descriptive commit message), and 2) that commit does not contain any changes that aren't related to that fix or feature.

Note that there will be times when the specific task you're working on (e.g., building out a new feature) will involve writing a lot of code. In this case, you'll want to break it down even more, into smaller individual tasks.

Writing Commit Messages

Different organizations have different guidelines for writing commit messages, so ultimately you'll want to use the guidelines adopted by the organization you work for. However, there are some general best practices you should get in the habit of following. The first two are basically the same as the guidelines for the commits themselves:

First, commit messages should be specific. Avoid commit messages like "Updated filename.txt". For one thing, the commit message is already attached to the file that was updated so you don't need to include its name in the message. But, more importantly, the commit message says

nothing about what was actually changed.

Second, commit messages should be short: try to keep them to 50 characters or less. If you need a lot of characters for your message, odds are you've tried to accomplish too much in a single commit (which will be a good reminder to keep your commits short and specific).

The third guideline is not as important as the first two, and there's also a lot more debate about it in the Git community, but it's still a good thing to be aware of: commit messages should use a consistent grammatical style. There are different opinions about what is the best grammatical style to use, and there is no right answer. Git itself recommends using the imperative case, i.e., commit messages are phrased as if you are giving someone an order. For example, you would use "fix bug" rather than "fixes bug" or "fixed bug". This is very much a matter of preference, however, so here too, you should adopt whatever approach is used by the organization you are working for. In the meantime, consistency is more important than the specific approach you choose because it makes your commit history easier to read: whatever style you decide to use, try to stick to it.

Multiline Commit Messages

In addition to the single-line commit command we've been using in these lessons (`git commit -m 'some message'`), Git also gives you the option of creating multiline commit messages using a text editor. The process that is used to do this can cause problems for people who stumble into it accidentally (which is easy to do!). Therefore, even though you may not need to use multiline commit messages often, we recommend that you read through this section to avoid problems later.

When making a commit, if you forget to add the `m` flag and message and just type `git commit` instead, whatever text editor is configured as Git's command line editor will open automatically. If you have set up a text editor to open when you type a command into the terminal (e.g., `code` for VS Code), you might expect that text editor to be opened, but it won't be. The Git command line text editor needs to be configured in Git separately.

The default command line text editor varies for different operating systems and operating system versions, but, in general, they are quite confusing to use if you aren't familiar with them. Many Mac users have had the very frustrating experience of getting "stuck" in Vim, the default command line text editor for MacOS. If this happens to you, to exit you first need to hit the `esc` key to ensure you are in **command mode**, then type the following:

```
$ :q!
```

Obviously, this is not an easy solution to remember, and it only applies if your default command line editor is Vim. Therefore, we recommend that you configure Git to use whichever text editor you use normally.

If you're using VS Code, this is the command to use:

```
$ git config --global core.editor "code --wait"
```

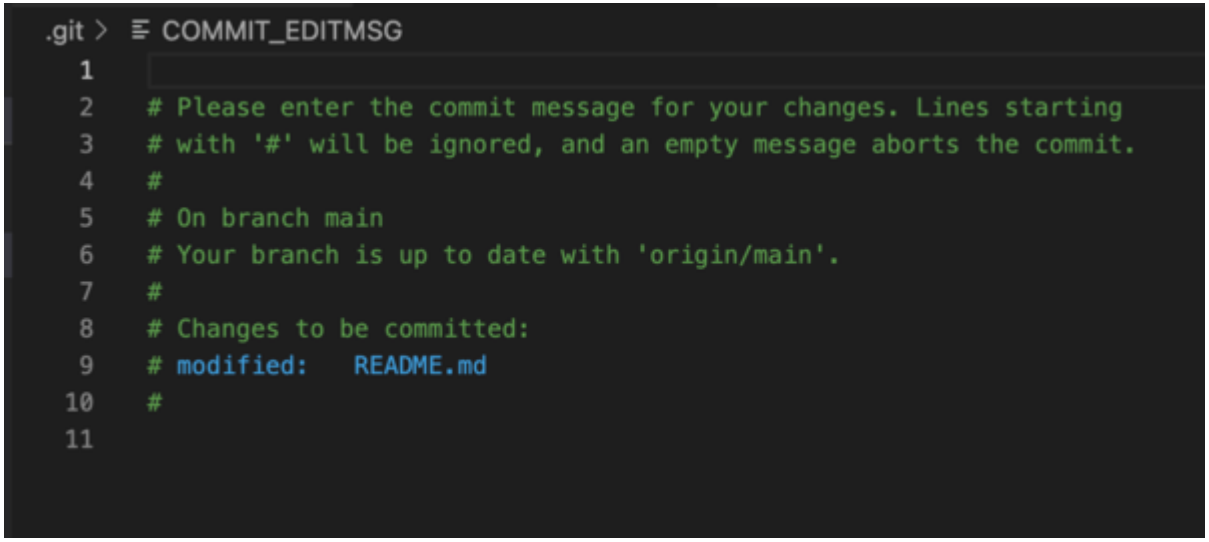
For other text editors, see **these configuration instructions (https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Setup-and-Config)** .

Note: In order for this to work, you need to configure your system to recognize the `code` command (or the command for whichever text editor you use) if you haven't yet. The steps to do this for VS Code are:

1. Go to the "View" menu in VS Code and select "Command Palette...".
2. Type "shell command" in the box and click on "Shell Command: Install 'code' command in PATH".
3. Close and reopen VS Code.

If you're using a different text editor, you should be able to find instructions for configuring the relevant command in their installation instructions or using a Google search.

Once you've set the default editor, the process for entering a commit message is quite straightforward. When you enter `git commit` in the terminal, your text editor will be launched and you will be navigated into a page that looks something like this:



Everything currently in the file is commented out and will be ignored by Git (i.e., it will not be included in your commit message). You can simply ignore it as well.

To make the commit, enter your commit message, save the file, and close it. When you close the file, the commit will be made, and whatever you typed in the text editor will be its message. If you close the file without entering a message or without saving the file first, your commit will be aborted:

```
Aborting commit due to empty commit message.
```

When creating a multiline commit message, the first line should follow the usual guidelines for commit messages: it should be specific and no more than 50 characters. This is the high-level summary of the changes made in the commit. You can then add as much information as you like on subsequent lines to fully document the commit.

See **these guidelines (https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html)** for more information about creating a good multiline commit message.

Check for Understanding

Before moving on to the next lesson, check for your understanding of this material by describing in your own words the two main reasons it's important to follow best practices for committing your work.

Conclusion

While the guidelines for creating good commits and good commit messages are not difficult, it is very easy to forget or to slip into more sloppy habits. This **will** happen to you! We encourage you to start practicing the good habits we've outlined in this lesson as early as possible. The efforts you make now will pay off for you down the road!