Lesson Monday

Introduction to Programming (/introduction-to-programming) / Git, HTML and CSS (/introduction-to-programming/git-html-and-css) / Git Best Practices

Text

Before we move on, let's cover a few Git best practices that are initially very confusing for students. We do not expect your commit histories to be perfect but we do expect students to work on forming good Git habits.

In this lesson, we'll focus on three key things:

- When you should commit
- What your commit messages should look like (and why they should be concise)
- Labels you can use to make your commits clearer

Then, at the end of the lesson, we'll go over Git expectations for independent projects throughout the program.

When Should I Commit?

There are three situations in which we expect you to commit.

When we add a new feature or functionality to an application.

This is when you'll most commonly be committing your code. But what exactly is a new feature? Well, if we add a sidebar with a list, this would be a new feature. However, just adding a new bullet point to that list would not be a new feature. That being said, you might realize that you forgot to add a bullet point after committing the list. It is okay if you have to write an additional commit if you forgot to add something to a feature. You've just started coding and you don't need to be perfect! Here's an example of a good commit message using the example above:

```
"Add sidebar with bullet list of greetings"
```

When we are fixing a bug in our code.

Bugs are inevitable. When we find one, we should fix it as quickly as possible. Ideally, we should do so with a label such as `BUGFIX` to indicate that the commit fixes a bug. This differentiates it from a commit that adds a new feature. For instance, if we discovered that our sidebar list wasn't formatting properly, we might have a commit message like this:

```
"BUGFIX: Fix formatting error in sidebar bulleted list"
```

When we reach a stopping point or we are stuck for a long time.

When we stop coding for lunch or at the end of the day, we should commit our code. That way, if something happens to the machine we are using, our code is safe (as long as we've pushed it, which you should be doing regularly). This is commonly called a work-in-

progress commit (WIP). In this case, the WIP should note what you are working on and it should include the WIP label. Here's are a few examples:

```
"WIP: Trying to fix broken sidebar"
```

```
"WIP: Working on new navbar with styled heading"
```

Note that if the above features are *complete* by lunchtime or the end of the day, they *should not* be WIP commits. After all, in that case, they aren't WIP commits, they are complete features and the commit message should reflect that.

Also, note that if you've gone more than an hour without making a commit because you are stuck, it is okay to make a WIP commit to indicate your progress. You will generally not do this in the industry, but especially on independent projects, it can help instructors determine whether you are working even when you get stuck.

Writing a Good Commit Message

Let's look at some more examples of good commit messages. But before we do, let's cover the rationale behind why commits should be short and concise. Understanding *why* we are writing commits this way can help us write better commit messages.

Commit messages should generally be 50 characters or less. But why so short? Well, in large projects, you might need to look through hundreds or thousands of commits to find the one you are looking for, especially if you are tracking a difficult bug. (There is actually a feature called `$ git blame` used in the industry to track bugs, but we won't cover that here.) More concise commits are better for that reason. Also, you've learned about `$ git log`, but if

you're a developer working on a big projects, you'd probably use the command `$ git log --oneline` instead, which puts every commit on a single line. (Try it out in the project you are working on.) Now imagine that running this command outputs *dozens* of pages of commits. You'd want them to be concise and to the point, too!

By the way, 50 characters isn't a hard maximum. It's just a guideline. Some developers might say 70 or 80 characters is okay. But 50 is a good guideline.

Now let's look at some more examples of good commit messages. Some of these messages touch on coding concepts we haven't addressed yet. But that's okay. It's more about getting a sense of what a good commit message looks like.

Examples of commits tracking completed features

These are examples of clear, concise commit messages tracking completed features or functionality.

```
"Add personal bio to home page"
"Add green and gold styling to navbar"
"Add styled box to heading"
"Add radio buttons to favorite color form"
"Add README that lists completed features"
"Update README to include project setup instructions"
"Create hide function to hide form on sign up"
"Write temp method to track current temperature"
```

Keep in mind that a project could have many forms, many updates to a navbar, or multiple changes to a heading. That's why a commit like `"Style navbar"`, while better than something vague like `"Add styles"`, still isn't very concise. If that navbar takes a lot of work, you could easily end up with twenty variations on `"Style navbar"` — and if you were trying to track down a specific change to the

navbar, you'd have a very difficult time doing so based on the commit messages. Likewise, `"Update form"` isn't very helpful if there are twenty different forms on different parts of the site.

(By the way, don't worry about things like functions and methods yet. You'll learn more about them in the next section.)

Examples of commits tracking fixed bugs

```
"BUGFIX: Fix typos in favorite things list"
"BUGFIX: Fix overlapping text box on home page"
"BUGFIX: update addMoney function to increment correctly"
```

All of these use the `BUGFIX` label and specifically state the bug that's being fixed. We recommend always using the `BUGFIX` label for clarity.

Examples of work-in-progress commits

Work-in-progress commits should only happen in a few situations:

- You are stopping work for a significant chunk of time (lunch or end of day are examples) *and* you are in the middle of a feature or a bug — or you're just plain stuck (which is totally normal).
- You have gone several hours without a commit because you are stuck and you want to document your progress (or lack thereof).

Otherwise, you should always be committing completed features or bugfixes.

```
"WIP: updating navbar to add styles"
"WIP: trying to fix bug in addMoney function"
"WIP: working on logic for Record class"
```

As you can see, these give a brief overview of what you are currently working on — and they communicate what needs to be done when the work resumes.

Examples of Bad Commits

Now, let's look at some examples of *bad* commits — and why they are bad.

The biggest offender is being too vague. Here are some examples:

```
"Update styles"
"Add h1"
"Fix bug"
"Finish work for day"
"Add forms"
"Add business logic"
"Add frontend code"
"Add bullet point"
"Add code"
```

The problem with these commit messages should be obvious — they don't really say anything about the work that was actually done. Some of these may seem laughable — like `"Add code"` — but we have seen these kinds of commits from students many times in the past. And fair warning — your instructor *will not* look favorably on commits like the ones above on your independent projects. Nor will potential employers be impressed if they see these kinds of commits on your portfolio projects.

Another issue is trying to put too much into one commit. This can happen for two reasons (or a combination of both): being too wordy or trying to put too much into one commit.

Here's an example of the former:

```
# This commit is too long.
"Add sidebar with a bulleted list of how to say hello in En
glish, French, Kinyarwanda, Japanese, German, Spanish, and
Pig Latin."
```

As you can see, the commit above is too long. It's very specific — in fact, it's *too* specific, which leads to an overly wordy commit message.

On the other hand, we might find that we've gone way too long without making a commit, so when we do commit, we need a long message to accurately convey the work that's been completed. Here's an example:

```
# This commit covers too much.
"Add sidebar, navbar, page content, and complete styles for
all pages plus add user interface code.
```

This commit is too long, not surprisingly, but it also covers too much. A general guideline is to watch out for using *and* or commas to separate clauses in a commit message. For instance, if your commit says you did X, Y, and Z, that suggests your commit is covering three separate features when there should be three commits instead, one each for X, Y and Z. However, this does not mean that you can't use *and* or commas in a commit message. Based on the work you've done, you will know whether a commit message is covering too much. At the very least, if it's not fully clear yet, it will become clearer with more practice.

In order to write a good commit message, you need to stop and think for a moment — just as when we choose our words carefully in a conversation. It's not quite crafting a haiku but it does involve thinking things over — and if you are working with a pair, checking in with them. The more effort you put into good Git practices now, the better off you'll be. Also, try to avoid typos in your commit messages. Sure, they happen. Just be careful and proofread your commit before completing the message.

In the next lesson, we'll go over some best practices for committing your work on the independent projects at the end of each section.

Previous (/introduction-to-programming/git-html-and-css/practice-tracking-changes-with-git)
Next (/introduction-to-programming/git-html-and-css/practice-github-remote-repositories)

Lesson 9 of 64
Last updated February 28, 2023

disable dark mode

(http://www.epicodus.com)