Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Team Week (/intermediate-javascript/team-week) / Practicing the Git Workflow

Text

We've covered the basics of using a git workflow — now it's time to practice a little. We recommend going through this practice *before* you actually start working on your group project. This way, you'll be more equipped to use this workflow, and to deal with any issues that come up.

This will be an extremely simple exercise — we will just make some small modifications to a README. That's because the goal here isn't to code, but to practice the workflow itself and also deal with a merge conflict.

Start by creating a new directory called `workflow-practice`. It will have just one file: `README.md`.

README.md

```
Hello!
```

Next, let's run the `git init` command. Create a `git-workflow` repository in GitHub and then connect it to this project using `git remote add origin git-workflow`. This will be a repository we can throw away later — it's just for practice!

Save the code in the README and make your first commit:

```
$ git add .
$ git commit -m "add greeting"
```

Once the code is committed, push it to GitHub.

Next, let's create a new *local* branch called `development`:

```
$ git checkout -b development
```

This will create a new branch *locally* (the `-b` flag), and then `checkout` will automatically take us to that branch.

It's common for large companies to have a `development` branch. This isn't the production branch — instead, the developers will merge code into this branch, test that it works, and so on — *before* it gets merged into the main branch, which is usually the production code. In this example, we'll just have the development branch along with the main branch to keep things simple. But in real world applications, there'd be additional feature branches which would get merged into the development branch first, and then tested, all before they are ever merged into main.

It's always a good idea to verify the branch we are on with `$ git branch`. That way, we won't accidentally start modifying code in the wrong branch.

If we run `$ git log`, we'll see that our first commit is in this branch as well. This is expected — when we create a new branch, it copies the code of the branch we are in along with its commit history. This is great for creating experimental features or for testing code.

Now let's modify our README:

README.md

```
Hola!
```

Save and commit this code:

```
$ git add .
$ git commit -m "change greeting to Spanish"
```

At this point, this branch has diverged from the main branch. Couldn't we just merge it into main?

Well, if we are working on the project alone, that would probably be fine. But that's not how development teams work in the real world. Other teams are *also* making changes to the code.

So let's simulate that process. We'll make another change to our code, this time *directly in GitHub*. This represents a second team working on the same code as us. They finished their update first and they've already merged their code into the main branch and pushed it to GitHub.

We can imitate this process by going to the repository in GitHub and clicking on the pencil icon, which allows us to modify the code directly in GitHub's UI.

Now let's change the greeting to French in GitHubs:

```
Bonjour!
```

Name the commit `"change greeting to French"`.

Now we are getting closer to a real world process.

We now have a remote main branch in GitHub which has two commits. We have a local main branch which has one commit. And we have a local development branch which has two commits. All three of these branches *have different greetings*.

So now let's say we want to merge our development branch into main and push that code to GitHub. How would we go about doing that?

Well, we need to switch back to the main branch and pull the latest code from GitHub. Then we need to switch back to the development branch, merge the code from main, make sure it is good to go, and then switch back to main and merge the development branch into main before we push it to GitHub. We'll walk through this whole process more slowly in a moment.

Why this convoluted workflow?

Well, we don't ever want to merge experimental code into main until we know it works. If we have any merge conflicts or other problems, we want them to happen in the development branch. Once everything is fixed and good to go, we'll be ready to merge the code into the main branch, but not before then.

So now let's slow this down. Here's what we need to do locally. We are currently on the development branch. We need to switch over to the main branch.

```
$ git checkout main
```

Next, we need to pull the latest code from GitHub:

```
$ git pull origin main
```

Now the main branch is up to date with the code that the other development team has already merged into main. We *could* merge our development branch into the main branch, but that could potentially lead to a big mess, especially if there are merge conflicts. That's not a good workflow. Instead, we need to play it safe and switch over to the development branch. Alternatively, if we're really worried about making a mistake and messing up code on either the main or development branch, we could create a safety branch called something like `savepoint`. This is one of the techniques suggested in Think Like A Git (http://think-like-a-git.net/sections/testing-out-merges/the-savepoint-pattern.html). It's not necessary to do this but it does create an extra layer of security when you're new to merging branches.

Now let's go back to the development branch. We're ready to merge the code from main *into* development. This is a really good time to verify that we are on the correct branch with `$ git branch`. If we are on the wrong branch, we could seriously mess up our code when we try to merge!

Once we've verified that we are on the development branch, we can run the following command:

```
$ git merge main
```

This will merge the main branch *into* the branch we are on — the development branch.

Before we move on to the next step, what do you think is going to happen?

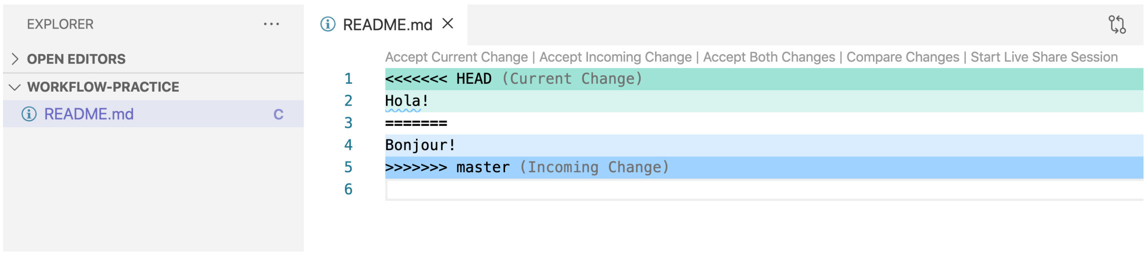
If you guessed that there will be a merge conflict, you're correct.

```
$ git merge main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the r
esult.
```

There have been two recent changes made to the code — how is Git supposed to know whether the greeting should be hola or bonjour?

When there is a merge conflict, a list with the file (or files) that have merge conflicts will be printed to the command line. We can see this above: `CONFLICT (content): Merge conflict in README.md`.

Merge conflicts can be scary. Fortunately, VS Code makes managing conflicts much easier. If we navigate to the application in VS Code, we'll see the following:


EXPLORER
OPEN EDITORS
WORKFLOW-PRACTICE
README.md C
README.md
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes | Start Live Share Session

```
<<<<<<< HEAD (Current Change)
Hola!
=======
Bonjour!
>>>>>>> master (Incoming Change)
```

VS Code has helpfully organized the code into two parts: the *Current Change*, which is the code from the branch we are on, and the *Incoming Change*, which is the code coming from the branch we are merging. It's easy to remember which is which by thinking of the branch we are merging as being the *incoming* branch and the branch we are on as being the *current* branch.

We have several choices here. If we click *Accept Current Change*, the text will just be `Hola!` If we click *Accept Incoming Change*, the code will just be `Bonjour!` And if we *Accept Both Changes*, the code will read:

```
Hola!
Bonjour!
```

Alternatively, we can modify the code directly — but if we do so, we have to make sure we remove `=======` and `>>>>>>> main` from our code in addition to making the changes. Git inserts these lines automatically to show where the code diverges in each branch.

Because we are collaborative and both teams have done great work, we are going to *Accept Both Changes*. Note that your use cases will vary greatly depending on the situation — sometimes you'll want both changes, sometimes you'll want just one, and sometimes you'll want parts of both — which means you'll need to update some of the code manually before declaring the merge a success and committing it.

Next, we are ready to save and commit our code.

```
$ git add .
$ git commit -m "update code to include both greetings"
```

At this point, we've cleared up the merge conflict on the development branch, and we've made sure that the conflict posed no risk to the main branch. Once we've verified our code looks good (hopefully by testing it), it's ready to merge into main. Now we can switch over to the main branch and merge our code again:

```
$ git checkout main
$ git merge development
```

As we'll see, the merge will go smoothly. That's because we've taken care of the merge conflict in the other branch!

At this point, we can safely push our code to the main branch.

While this example is very simple, it should hopefully make the whole process of merging code a bit clearer. Ideally, when working with your group, you should be working on different parts of the code and create different features — which means you won't run into as many merge conflicts. Make sure you practice a good git workflow by following the steps above when merging and dealing with merge conflicts.

If you're still feeling a bit foggy on these concepts, go ahead and repeat the process — add a few more files and then make different changes directly on GitHub and then on your development branch. This will give you the chance to work through a few more simple merge conflicts before you run into one that is potentially bigger and more confusing during your group project.

Previous (/intermediate-javascript/team-week/learning-more-about-git)
Next (/intermediate-javascript/team-week/hosting-a-webpack-project-with-gh-pages)
Lesson 6 of 13
Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)