Recap

Last week we went over a brief introduction to Javascript and flexbox. We explored how you can make a website actually dynamic by updating it using javascript and how flexbox can be used to place elements on our site.

Today

Today we're going to look into how to actually manage a web development project using git, and then we'll breifly look at an introduction to react.

We're going to need two pieces of software for today's tutorial. The first is `git`. We'll head over to https://git-scm.com/downloads and grab git from there. Also, while we're installing software let's head over and grab `node.js` from their official website which we'll use to install react: https://nodejs.org/en/download/.

Git

Git is an open source version control system that allows for multiple developers to work on the same project at once. If you're going at a project solo then git will allow you to keep track of your progress and ensure that you don't break everything by adding a new feature. I'm sure you've found while developing hat you've copied and pasted a file in order to ensure you still have the original - git allows us to do that and so much more!

When working with a team, overwriting each other is a chaotic mess, so it's better for us to work independently and push our changes for

everyone to see on to a remote repository. Github allows us to host that remote repository.

First, we're going to want to get git on our local machine. If you have a mac, or are running linux than you most likely already have it installed. Open up a terminal (or powershell in windows) and type `git --version` to see your version. If this command errors out, then you'll need to install git according to your operating system.

You're pretty much never going to want to use something like git without a remote repository like GitHub or GitLab. For the purposes of our tutorial, we're going to use GitHub.

First, we're going to head over to GitHub, and create a new repository. We'll then notice that there is a blank repository. It is a remote repository, and note that there is a label stating that there is 1 branch named main. We'll go into branches later, but for now notice that we have the main one selected.

Git uses these branches for people to work on many different features simultaneously. Now that we've got a GitHub repository, we're going to want to work on it locally. The first thing that we need to do is identify ourselves with a name and an email.

```
git config --global user.name <name>
git config --global user.mail <email>
```

Now that we've set both our name and email, we can begin to use git & as a result GitHub.

Heading back over to GitHub, we'll click the green "CODE" button and copy the link for this repository. Within the terminal, we'll type:

```
git clone <link>
```

We now have a local folder that links to this remote repository, and will allow us to work on this repository.

Notice that `clone` is the first git command that we've actually used other than config. You can use `git clone` followed by any GitHub link to download a local repository of any GitHub repo. We could for example grab the VSCode repository, and clone it.

Git Commands

Next we're going to go over some git commands.

`git status`

The first thing we'll do with our repository is run `git status`. You'll notice a few things when we do this - the first being that we are `On branch main` - if you remember from earlier the `main` branch was the only one, and this is confirming that we've selected it.

It also says that the branch is up-to-date. This means that our local version of this repository is up-to-date with GitHub's version. We also have "nothing to commit", since we've completed nothing locally that should be added to the remote repository.

`git pull`

Remember with `git status` how it would report back to us if our version is up-to-date. Let's go back to our GitHub repository and actually add a file using the "Add file" button. We'll create a new file called `README.md` since our repository is asking us to do so. We'll add some fake content in here and then add the file. Once we've added this

file, we'll go back to our local version of the repository and realize that unfortunately our changes haven't synced up. Even though we added this file ourselves, we can pretend that another developer added this file. We'll first run `git status` which will tell us that our repository needs to be updated! We can then run `git pull` to take the changes (in this case the addition of a new file) on to our local machine. Furthermore, we've now updated our local version of the repository, and we're ready to continue to work on it!

`git add`

Next we're going to create a file locally. Let's create a new HTML file called `index.html` and let's add some starter code using a neat VSCode shortcut `html:5` to auto generate some code. First we'll run `git status` which will show the addition of our new file! `git status` doesn't know if it should be tracking the changes that we've made (sometimes you don't want to provide your changes to everyone else).

Next, we can add a file to the set of files that are tracked by `git add [filename]`.

We can check the status and see that this file is now tracked. If we quickly add 1 more file here, then we can show that you can juse use `git add file2.html` in order to add another file without seeing it's changes. Essentially, you can use `git add [filename]` to add a specific file and `git add *` in order to add all files within this directory.

`git commit`

Now that we've added files, we need to actually "commit" these files. This is similar to the commit that we created earlier when creating the `README.md` file directly on GitHub. A commit is essentially the

opportunity for us to describe all the changes we've made and for git to note down every single change we've completed.

If we run `git status` we'll notice that we have all of our files added. Now we can run `git commit` which will open up a text editor, so we can write our description. The first line will be our title, and the rest will be our description. We can type all of that in, save it, and then we've officially created a commit!

`git push`

Now that we've created some changes locally, and stored them on a commit, we can run git status. From here, we can "push" our changes up to the remote repository by using `git push`!

Branching

Next we can introduce the idea of a branch. Branches allow us to take the current commit, and continue to work off of it without effecting any other developer or work. We'll use `git checkout -b adding-css` in order to create a branch called `adding-css`. We will also automatically switch to this new branch when we do this.

We can check all of our branches using `git branch`. You'll notice that our new branch isn't on our GitHub page, that's because we haven't run `git push`. When we run `git push` it will ask us where we want to push our changes. Generally the built-in response to this error is what we want to run - don't worry about memorizing it!

If we run that very quickly, and refresh our GitHub page, we'll see our new branch is selectable from the list of branches! It worked!

From here we are going to add a CSS file, and then we'll run `git status`, `git add -p` and `git commit -m` in order to make a commit for

this new CSS file. We'll also do `git push` one last time in order for our new CSS change to show up on GitHub.

We'll open our GitHub page and see here that the new branch is there. We'll click on it, and we'll see our commit. From here, we can actually use the shortcut to create a pull request. We could also compare the changes of the two branches in the upper right and be able to create a pull request from there. A pull request can be thought of as a formal request to **merge** or combine your new code with the code that is currently on another branch.

In this case we have a `main` branch which we're assuming is the final version of the site. We'll then create this pull request, we could do some cool stuff like assign someone to look at it, but we'll skip that for now. Now we're able to click the merge button to put our commits on to the main branch!

You would then normally delete the branch that you used, but we'll keep it for now. We can go back to our main branch using git checkout main. And we'll notice that it hasn't been updated yet -- we can then run `git pull` once again and all of our changes are there!

Merge Conflicts

Unfortunately, combining the work of two developers isn't always this easy! Sometimes we'll have a merge conflict when we attempt to combine our work with another developers.

A merge conflict happens when two developers change the same line in a file, or one developer deletes a file that another worked on. We can begin to simulate this process by making a change to our `temp.html` file's title line.

The best way to display this is to change back to our branch using `git checkout adding-css`, and then we'll make some changes to our HTML file. We'll delete a line and maybe replace it with something else.

Once we've committed both of these changes on to our two branches, we'll switch back to the branch that we want to merge with, in this case `git checkout adding-css`. Then we'll say hey, we want to merge this branch with the main branch. To do that, we can do `git merge main`. Git will automatically say hey, that's not going to work, and we'll get some information placed right into our file about the conflict. We can open VSCode to easily resolve this merge conflict by picking two of the options to go forward.

Final Remarks

- Git can be tricky
- a good few of us have lost hours of work by using it improperly once or twice, but at the end of the day it's used in the industry for good reason → it allows you to make sure you never lose any previous work you've done!
- You can use gitexplorer in order to learn more, or official documentation if you so choose.

React

Why React

React is a JavaScript library for building web applications. It uses JavaScript, HTML, CSS, and JSX. React is pretty linear, in the way where you can learn only the things you actually need to use, and continue to build your knowledge base that way.

Getting Started

Before we get started with react, we're going to have to install `node` and `npm`. We actually had you install `node.js` at the beginning of today's tutorial, so that's already completed. `node.js` is a JavaScript environment that allows you to create web servers and networked applications. `npm` is a package manager, which makes installing `node` packages easier. A package (sometimes called a module) is a code library that can extend Node's features by adding some code written by other developers.

Now that we have Node.js, we can begin to create our first react project.

First project

We can create our first react project by using `npx create-react-app club1`. This will then create a simple react app quickly and easily and it will be called club1. We'll then open up that directory by using `cd club1` and we can then actually run our react app for the very first time by using `npm start`.

While we wait a few minutes to create a brand new project we'll use an existing one that I've created. The first thing we'll want to do is run `npm start` to get our react site to be actually running. We'll see this spinning react logo to inform us that our site is indeed working! Then we're going to do with our new project is delete some of the pre-made files, specifically `logo.svg`, `reportWebVitals.js`, `App.test.js`, and `setupTests.js`.

The first thing we'll do is go into `index.js` and remove the `reportWebVitals` information from that file. Next we'll open up `App.js` and we'll run into the first instance of JSX. JSX is a version of javascript that allows us to easily return some HTML. Notice how this function actually returns a HTML div! This is why we started with HTML

because you'll still most likely continue to use HTML when you're developing in React.

We're going to delete the entirety of `App.js` and type out a new version.

First we import a css file → in React we can directly import these CSS files instead of having to link the HTML and CSS together! It's much easier. Then we've defined a function called `App` which is currently returning a `div` that has a bunch of HTML in it.

The only new syntax we have here is the import statement, the function which actually returns some HTML, and then at the very bottom of the file we have `export default App;`, and this just allows us to import this function to be used in another file.

```
import "./App.css";

function App() {
    return (
        <div>
            <div className="title-card-container">
                <div className="title-card-item">
                    <h1>People watching club</h1>

                    <p>
                        Humann Homo sapiens are the most abundant
and widespread
                        <b>species of primate</b>, characterized
by <i>bipedalism</i> and
                        large, complex brains. This has enabled
the development of advanced
                        tools, culture, and language.
                    </p>
                </div>
            </div>
```

```
        </div>
    );
}

export default App;
```

We'll then go back to `index.html` and you'll be able to notice that hey, we are actually importing the App on line 4, and then we are grabbing the root element of our website using some javascript, and then we are rendering our app right within that root by using this weird syntax in which we use the name of the function and a slash so that it doesn't close. This seems very weird at first glance, but we'll get used to the syntax as we use it more.

We can then of course grab some CSS that we've defined and put it into `App.css`. We've just defined some title cards with some css styling. We'll leave `index.css` be.

As we've seen our react website slowly update as we type in more code, we'll actually get to the final version of our site in `club1`. We can once again, run it with `npm start`. You'll see that `club1` is this final version of a simple club website.

Next we'll checkout `club2`, first we need to once again run our react project so we'll open up that folder in our terminal and run `npm start`. Next we're going to notice that we have added some cool title cards in order to show off some different locations around the world.

Notice how there is a large amount of similarity between these different locations. They are all using the same two divs, an image, a heading, and some coordinates.

React's biggest benefit is it's reusability. We can create what is called a **component** in order to reuse the same set of code and styling multiple

times on our site. First we'll create a folder called `components` within our project. Next we'll create a file called `InfoCard.js`, next we'll want to import our CSS and react:

```
import "../App.css"
import React from "react";

function InfoCard(props) {
    return (
        <div className="title-card-container">
            <div className="title-card-item">
                <img className="card-image" src={props.image} />
                <h1>{props.name}</h1>
                <p>{props.coordinates}</p>
            </div>
        </div>
    );
}
```

We'll then create a function for our InfoCard, and we'll accept `props` as our parameter. We'll get more into that later, but for now know that `props` are a way for us to move values around and make things truly unique. Let's pretend that props are just a dictionary of values for us to use in that same json format that we alluded to before. Now we're going to start to create the same HTML format from our containers. We'll make those divs, and we'll make the image, but now we need to actually say what image we wish to display. For now, we'll just do `props.image` and come back to it later. Same with finding the `name` and the `coordinates`.

From there we can move back to `App.js` and we can import our `InfoCard` by using `import InfoCard from "./components/InfoCard";`.

Then we'll need to create new InfoCards for each of our previous locations:

```
<InfoCard
                name="Eiffel Tower"
                coordinates="48.8584° N, 2.2945° E"

image="https://www.toureiffel.paris/sites/default/files/actualite/
image_principale/vue_depuisjardins_webbanner_3.jpg"
            />

            <InfoCard
                name="Empire State Building"
                coordinates="40.7484° N, 73.9857° W"

image="https://media.timeout.com/images/101705309/image.jpg"
            />

            <InfoCard
                name="CN Tower"
                coordinates="43.6426° N, 79.3871° W"

image="https://upload.wikimedia.org/wikipedia/commons/9/96/Toronto
_-_ON_-_Toronto_Harbourfront7.jpg"
            />
```

Notice how we are providing a name, a coordinate, and an image. These are the prop values that will be used within InfoCard in order to find those values. If you wish, we can print out those props to console so we can actually see the values using `console.log(props)`.