Before Lecture

- create `club1` as an empty directory, and initalize a koa project with `npm init -y` and `npm install koa`.
- pull `club2` from the github repo and `npm install`

Recap

Last week we went over the creation of a react project for the creation of a club homepage. We went over how a react project actually works, and created an image carousel.

Today

Backend

Today we're going to introduce you to a backend. This concept that we alluded to in the first lesson, we'll return to as a way for us to be able to provide information dynamically to the frontend.

The backend of our site would generally handle any important data, and create endpoints in order to display specific amounts of data. Later on, authorization can be used in order to ensure only specific users can access specific data.

We're going to introduce you a simple version of a backend for our current events page. I've brought up an example of this site here, and you can see that on this page we have a few events each with some information.

If we open up our inspect page and refresh the site, we'll be able to see that we actually have a request coming in on the requests page called `events`. Within the response, we can see a bunch of information. We'll also realize that each of these events corresponds with one of the events on this page.

Basic Backend

Let's start to create a basic backend, we'll create a folder called `club0`, and within it we'll run a new command called `npm init` in order to initalize a node project. We'll then run `npm install koa` in order to install koa, the library that we're using in order to create this simple backend.

You'll start to notice that between `react` and `koa` that the simplicity of installing a new library, and then beginning to use it in your project is the reason that we use `node.js` and for that matter `javascript` to create web applications.

Now that we've installed koa we can begin to create our project. First we'll need to make an `index.js` file, just like our react project, this will be the basis of our backend. We'll then start to write some simple sample code in order to see how this backend works:

```
const Koa = require('koa');
const app = new Koa();

app.use(ctx => {
  ctx.body = 'Hello World';
});

app.listen(1234);
```

First we use a `constant` in order to import `koa`. We then create an "application" or a backend in order to start to use this new library.

We'll then use `app.use` to "add" a function to be called everytime our API is used. This is referred to as `middleware` and we'll get more into this later but for now let's just say that the function passed will be called every time our API is used.

We'll then set the body of our API response to be 'Hello World', and we'll set our app to listen on port 1234. When we run our application by running `node index.js`. Nothing seems to happen in our terminal, but if we open our web browser and head to `localhost:1234` we'll see "Hello World", and in essense we've written our very first backend, in this case it only returns Hello World but that's alright!

A Get Request

As you continue to learn more about creating a backend you'll learn about different types of requests that can be made to an API. For now, we're going to introduce you to a GET request, which at the end of the day is used to "get" information from that backend.

We're going to start by making our backend actually return something to us, that way we can eventually reach a point where we're only asking for specific data. We are going to begin to create closer to a final version of our site, we'll head into `club1` which has been setup as just an empty koa project, with no changes.

We'll create `index.js`, and this time we'll define what's called a `router`. This will let us define different sub directories or routes on our page in order to provide different information.

```
const Koa = require('koa');
const router = require('@koa/router')();

const app = new Koa();
```

```
router.get("/events", async (ctx) => {
        ctx.body = "Event Information Goes Here"
})

app.use(router.routes());
app.listen(3001);
```

When we try to run our application using `node index.js` we get an error, because we didn't actually install our new dependency, the `@koa/router`. We'll then install it using `npm install koa-router`, and after that when we run `node index.js` our application will run. When we head over to `localhost:3001`, we'll see that we are met with the response of `Not Found`. It turns out that we only defined what happens when we head over to the events endpoint. If we head to `localhost:3001/events` we'll see that hey our Event Information Goes Here text is provided.

Sending JSON

Your next thought may be that hey, we're sending some basic text, what if we could send some more useful information. We could for example, send some JSON, recall that its just a dictionary, or a set of keys and values. We can check out `club2` where we've defined `events.json` with an array of these key value pairs. Then within `index.js` we've slightly changed what data is provided to our site:

```
const Koa = require('koa')
const router = require('@koa/router')()
const fs = require('fs')

const app = new Koa()

let events = []

function getEvents() {
  events = JSON.parse(fs.readFileSync('events.json')).sort((a, b)
=> {
    return new Date(a.date) - new Date(b.date)
  })
}
getEvents()

fs.watchFile('events.json', getEvents)

router.get('/events', async (ctx) => {
  ctx.body = events
})

app.use(router.routes())
app.listen(3001)
```

We've added an array for all of the events, created a function in order to pull all of these events from the file. We can pull files using this built-in `fs` library, we say that we want to read them using readFileSync, we provide what file we want to read. Then we sort them, according to their dates using .sort(). It compares the two values, and we provide the difference between the two which allows the function to sort them. We then set events to the array created.

We'll then say hey we want to ensure that updates done to our json file reflect in our api. So we're going to watch that file for changes using watchFile, and call a function everytime it changes, in this case getEvents so we update the events variable.

Instead of returning random text we're now going to return the events using this API, and so now if we head over to our site we'll see that our events json show up in our web browser! I'm using a nice extension in order to format this json, but regardless of what it looks like, it will be all the information provided.

From here if we head back into our file structure, and `events.json`, if we remove one of the values within and refresh our site we'll see that the changes are seen!

Basic Frontend

Now that we have an extremely basic version of our backend finished, we need to display this information in a bit of a nicer way, hopefully actually on our site!

We've setup `club3` as a very basic version of our frontend. You'll notice that the image carousel from last week is gone, and we've replaced it with an upcoming events page. Suffice to say, that we are going to return to that home page we created last week, once we form our website a bit closer together. For now, let's continue on.

Right now, our events page is just displaying some sample text and no actual event information. If we inspect our code we can scroll down and realize that we have this `events` value, and it's currently empty. Then we are checking to see if the `events` value is empty, and if it is, we display some text saying events will be here.

If we uncomment our events array, we'll see that our page updates with the information, all nicely formatted like it was in our example from earlier. We can inspect this file further and see how it actually works, it looks like we are checking to see if our `data` value has actual data in it, and if it does we are creating 4 of these `upcoming event cards`, passing along a value of our data to each one. We then add some extra text, and all of that is just some HTML and CSS classes, of course we've imported our CSS classes at the top of our file.

We then provide an alternative statement, with some JSX if we don't have any events currently in our events array, and that was the sample text that we saw earlier.

Unfortunately, our backend and frontend aren't linked. We are still stuck with all of our data for our website being defined on our react frontend.

Frontend & Backend

Now that we've shown off how we can create a frontend worthy of our backend, we want to link these two in some way. We want it so that our events on the frontend, and fueled by the data on the backend.

We can combine our server into a new server folder, and our frontend into one project within `club4`. First, let's have our frontend running using `npm start`, and let's start our server using `node index.js`. We know that these aren't connected in any way so let's begin to set up our API so that we can connect it.

Cors

We have to link our backend to our frontend, and so we have to introduce the idea of **CORS** or **Cross-Origin Resource Sharing**. CORS are essentially this small header defined within requests that define the locations from which a server can load resources.

We have to say that our frontend is allowed to load resources from our backend. To do this, we need to use a `koa cors` library. First we'll install it with `npm install @koa/cors`. We'll then import it by using `const cors = require('@koa/cors')`, at the start of our file, and then we'll insert it into our program by using `app.use(cors())`.

Middleware

`Cors` is an example of middleware, with every API request, every `app.use` command is executed in order. Essentially we figure out our

CORS, and then we deal with sending back the information for our request.

Connection

We can now begin to actually connect our frontend to our backend since we've set up the backend. In order to do that, we need to run our fetching of the backend whenever we load our site. We can use a new `hook` called `useEffect` in order to do so. Remember back to last week, we created a hook called useState in order to create a counter, and later an image carousel. The way hooks work in react, is that whenever the value of the hook changes, it updates the website.

In this case, we're going to want to create a useEffect hook, and so first we'll need to update our import statement to include `useEffect`. We then create our useEffect, and it takes in a function which we'll define as an arrow function, and a list. That list will contain a bunch of values, and whenever one of these values change, the useEffect "hook" will call the function we provided. Very similar to how our backend will update whenever the `events.json` file changes.

We then need to actually call our api, and we can use the fetch function, `const response = await fetch('http://localhost:3001/events')`. We use the `await` keyword, because a request is not instantaneous. It's usually a few fractions of a second, but it's still not instantaneous, and so we need to have our program continue to function and work while we wait for this data to come back, hence `await`.

We then turn our data into `json` because us calling our api just returns us back with text, we can turn it into `json` using `const newData = await response.json()`. We'll then set our data to the new version of events using `setData(newData)`. Then within our `useEffect`, we will call our new async function called `fetchData`.

```
useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(`http://localhost:3001/events`)
      const newData = await response.json()
      setData(newData)
    }
    fetchData()
  }, [])
```

We'll see that hey look, when we refresh our page, all of our events are now populated! It works! If we change some of the information that our API responds with, we'll see first that our API location changes, and even better when we refresh our page the API is called once again and updated with new information!

Continuing with our theme of making sure that our code is reusable and dynamic, we can notice that if we add another entry to our upcoming events, they don't show up on our site. We know that `data` stores all of these events, we can print this out and see it in our console.

From here, we need some way to display our 5th event. We could simply add another event, and put in the 5th one, but that won't work once someone adds another event. And it turns out if we add this extra event, and don't provide its information, our site will throw us some errors.

Instead, we can use a `map` statement, one of those cool functions that are a part of the array object. We introduced the idea that there are a bunch of these functions when we started on JavaScript, and you could of course use a classical for loop, but this way makes our code a bit cleaner.

We can say hey let's use `data.map` in order to loop over every event, we then define a function that gets called with every event in our array. In this case we'll take in the event as our parameter, and we want to return our `UpcomingEventCard` so we'll do exactly that! If we now add some more events, or even remove them, our site with dynamically update!

`club5` has a final version including all of these changes.

Recap

So today, we introduced the idea of a backend, we created one to display our events' information. We then incrementally updated our backend in order to allow it to connect to our site using CORS. We created a frontend based on some pre-written code to display our events, and then we linked our backend to our frontend so that changes made on our backend update on our site.

You may think that all of this hassle wasn't really worth it for this example site, and that may be partially true. But you can continue to work on this to make it worthwhile. Maybe you make a dashboard in order to add more events to your site -- allowing a non-technical person to manage the day-to-day running of the site. Or maybe it even populates the events automatically from a google calendar. Regardless, the idea of a backend and a frontend is deeply important, and how we separate our and control data when we're creating websites.