Full Stack Project

Introduction

This document is going to talk through a few of the concepts that you need to understand to get a full stack JS project up and running. It will also talk through some of the steps that you will need to follow to set up the project.

Folder structures

We want to create two separate directories for this app. In a way we are going to create two projects that are going to be linked. One will be our front-end, which will be our JS, HTML, and CSS, and the other will be our backend, which will be our Express API.

We can create a whole new directory somewhere on our computer for this app. Mine is called `record-store`. In this we can have a folder called `front-end` and one called `backend`. Or we could set it up totally separately, so that we have one directory called `record-store-front-end`, and on called `record-store-backend`. It's a matter of preference as to how you would like to do it.

Front-end folders

```
/front-end
    |- server.js
    |- package.json
    |- /public
        |- index.html
        |- /js
        |   |- index.js
        |
        |- /css
            |- styles.css
```

This is just the typical setup for a basic front end, but with a parent directory that has our server in it. Our server is going to serve the static content from the `/public` folder.

Backend folders

```
/back-end
    |- server.js
    |- package.json
    |- albums.http
    |- /models
    |   |- Album.js
    |
    |- /data
        |- data.js
```

Remember that in both of these folder structures, the `package.json` is created by the `> yarn init` command. Also you might have a slightly different setup, and that could be totally fine. Please make sure that you are also changing the relative path names for your circumstances.

Front-end server

In the early couple of weeks of the course you were writing JavaScript - browser JS. This is a language that runs in the browser, and so it has to be served to the browser somehow. Typically it's easiest to get that up and running using the VSCode live server. With the click of a button up comes the server, and then you can find your project at `http://localhost:5500/`.

The way it works is that the files are all sent up to the browser (the index, with the connected JS and CSS if they are referenced), and after that they are navigated by the user. Only if the URL is refreshed are all those files sent up again.

Because as we started this project we now had some more experience with creating servers, when setting up the structure this time, I thought it better not to rely on the inbuilt server in VSCode, and whip up a small server. It's not something that I have done for static content very often, so mine was a bit more convoluted.

This time it looks like this:

```
const express = require('express')
const app = express()

const port = 8080

app.use(express.static('public'))

app.listen(port, function() {
    console.log(`Example app listening at http://localhost:${port}`)
})
```

All we are doing is creating a little server to get our `index.html`, `index.js`, and `styles.css` up to the broswer. It will run at `http://localhost:8080/`.

Our `index.html` looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="./css/style.css" rel="stylesheet"></link>
    <script src="./js/index.js" defer></script>
    <title>Document</title>
</head>
<body>
    <h1> NEW PROJECT </h1>
    <div id="attach-here">
        CLICK THIS THING
    </div>
</body>
</html>
```

and our `index.js` looks like this:

```
console.log("index.js is running!")

const attachPoint = document.querySelector('#attach-here')
console.log(attachPoint)

attachPoint.addEventListener('click', function(event) {
    console.log("The attach point was hit")
    const newElement = document.createElement('div')
    newElement.innerHTML = "here's a new element on the page"
    attachPoint.append(newElement)
})
```

These don't do much, and you can have a play around with the code. They are just to get things set up, and get a start on the project. Where you take things from here is totally up to you.

We will need to run our `> yarn init`, `> yarn add express`, and `> yarn add nodemon --dev`, and then run the project using nodemon. We can add a script to the package.json if we like:

```
{
  "name": "record-store-backend",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.6"
  },
  "scripts": {
    "dev": "nodemon server.js"
  }
}
```

And we can run that script by running `> yarn run dev`.

Working in the front end

We are now set up on the front end. The server to serve these files is going to just do this small job, and not be anything like the API we will build on the back end. It's not going to have any or many routes, and will be there to just serve up those files, in almost exactly the same fashion as the VSCode Live Server. We can now do all our JS work here, and build up the front end in any way we would like. It's just a normal JS project.

Before we build out this project, and attach it to anything else by getting Axios involved, please note that there is a heap that we can do with the project as it is. We can make all sorts of apps with just the HTML, CSS, and JS, and *no external data*. We could make a little game like noughts and crosses, or we could make an app that presents some information about a topic, or really any number of possibilities. We have the power of those three technologies at our fingertips. We can set up whatever structure we would like in the HTML. We can add event listeners to buttons, and present different things based on how the user interacts with it. There is no necessity for it to be connected to the rest of the internet.

Setting up the backend project (API)

Here we are going to set up a separate project. It will be an API. In this case I am going to make it an API that will serve some data relating to some albums. At this point we are concentrating only on this API project, and for now we are going to just make an API for the albums without thinking beyond that. We are leaving the front-end behind altogether for now.

We are here going to start by creating an Express server, and put in some basic routes:

```
const express = require('express')

const data = require('./data/data.js')

const app = express()
const port = 4400

app.use(express.urlencoded({ extended: true }))
app.use(express.json())

app.get('/', function(req, res) {
    console.log('routes are working?')
    res.send('hello world NEW PROJECT')
})

app.get('/albums', function(req, res) {
    console.log('the GET ALBUMS route was hit')
    res.send(data.albums)
})

app.post('/albums', function(req, res) {
    res.send('POST ALBUMS route was hit')
})

app.get('/get-first-album', function(req, res) {
    console.log('the GET FIRST ALBUMS route was hit')
    res.send(data.albums[0])
})

app.listen(port, function() {
    console.log(`Example app listening at http://localhost:${port}`)
})
```

A few things to chat about here. Firstly, remember that we will need to add our dependencies, as above in the front-end. We are temporarily keeping our data in the file given below, and this will be our 'database' while we get things set up, and we can incorporate MongoDB later. We are adding a couple of lines of middleware that we haven't discussed too much, and that will help us to parse the body of requests.

We are adding the root route `'/'` as a first check that the API is working, and then some RESTful routes. I put in the route `'/get-first-album'` just to show that it is ultimately up to us how we design the API and decide to release the data. We can stick with convention, but also we have full control.

This is the data file we are using temporarily in lieu of a DB:

```
const albums = [
    {
        artist: "Horace Silver",
        title: "Song for My Father",
        year: 1964,
        genre: "Jazz"
    },
    {
        artist: "Charli XCX",
        title: "How I'm Feeling Now",
        year: 2020,
        genre: "Pop"
    },
    {
        artist: "Charli XCX",
        title: "Charli",
        year: 2019,
        genre: "Pop"
    },
    {
        artist: "Morrissey",
        title: "Vauxhall and I",
        year: 1994,
        genre: "Rock"
    }
]

exports.albums = albums
```

As you can see, it is just some static JS in a file - an array of objects.

Another big aspect to note here is a change in mindset from when we first started making an Express API. Initially we were serving various strings, and some HTML (which is just a specific type of string that a browser knows how to parse). At that time we were seeing code like this:

```
app.get('/simple-form', function(req, res) {
    res.send(`
          <form action="/my-handling-form-page" method="post">
              <div>
                  <label for="email"> Name: </label>
                  <input type="text" id="email" name="user_email"
value="">
              </div>
              <div class="button">
                  <button type="submit"> Send your email </button>
              </div>
          </form>
        `
    )
})
```

The broswer will parse this response and present a form to the user. There are many frameworks that do this type of server-side rendering, with Ruby on Rails (RoR) being probably the most famous example. But we have also seen this type of thing with Express where you used `.ejs` files to dynamically send HTML in response to requests (using embedded JS).

There is nothing wrong with this approach. However, we are going to do things a little differently here. Our API is going to send data only - JSON data. We are not going to preempt what the person requesting the data will do with it - we are going to remain agnostic on the end use of our data. We will just set up a server, and create some routes, and deliver JSON data from these routes.

We can even set up an `albums.http` file with the HTTP Client extension in VSCode:

```
GET http://localhost:3000/albums HTTP/1.1

###

POST http://localhost:3000/albums HTTP/1.1
content-type: application/json

{
    "title": "Fear Innoculum",
    "artist": "Tool",
    "year": 2020,
    "genre": "Rock"
}

###

DELETE http://localhost:3000/albums/5f8f85efe1e990f36ceefce3 HTTP/1.1

###
```

This will make requests to our API, and allow us to test our routes without having to set up a whole lot of front-end code to ping our backend. It will also exist with the project, and it provides a bit of a record of the progress we have been making as we build out our routes.

Before we move on, we have work to do here yet. We need to build out several other routes, and we would be well advised to set up our DB and Models prior to going forward. While the temporary measure of some data in a file is ok to get things up and barking, as we get deeper into the code of the API, it would be better to write the code to connect to the DB now, and not have to rewrite things later. The information relating to that is in the Mongo/Mongoose PDF.

Still, here we are going to move back to the front-end project, and expand our horizons.

Back to the front-end - connecting to the internet

As we have talked about, the front-end already can make many different apps without connecting to any data source. There are limitations to this of course, but there's also much that can be done. Still, given that the internet is available to us, we might like to connect to the wider world and take advantage of the options out there.

Axios is the way that we can get hold of this data. The whole of the internet is based around a bunch of computers making HTTP requests and responses. We make a request (a GET request) every time we hit enter on the URL bar of a browser. We make an HTTP request when we sumbit a form (quite often a POST request). And now we will have a third way to make http requests via Axios.

We add this line to our `index.html`:

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
```

and this will download the Axios module from a CDN (Content Delivery Network). This is somewhat similar to requiring a package in Node.

Then in our `index.js` we can make HTTP requests:

```
const starWarsDock = document.querySelector('#star-wars-dock')
console.log(starWarsDock)

axios.get('https://swapi.dev/api/planets/3/')
    .then(function(res) {
        console.log(res.data)
        starWarsDock.innerHTML = `<h2> ${res.data.name} </h2>`
    })
```

Here we are finding a node in the DOM with an id of `#star-wars-dock` (which you will have to put into your HTML to have this work), and then we are hitting the internet with an HTTP GET request, and receiving a response. When we get that response then we are changing the innerHTML attribute of the node we selected.

This small bit of code should open up a whole new world to you. There is data out there on the internet covering just about any conceivable topic, and you are really only limited by time and your imagination.

Attaching the backend to the DB

Most of this is outside of the scope of this document (see the document on Mongo for details), but I will post the `server.js` code here so that you can look through it. Note that you will need to configure things for your DB, and that you will need to make the Album model (at least). I have managed to send JSON data from the front end, and have removed 'multer' for now (for simplicity - it has other uses).

```
const express = require('express')
const mongoose = require('mongoose')
const cors = require('cors')

// Import the model for Album
const Album = require('./models/Album.js')

// A data store, here used for seeding the database (see routes below)
const data = require('./data.js')

// Connect to our DB
mongoose.connect('mongodb+srv://matt:password@cluster0.pxhul.gcp.mongodb.n
et/recordStore?retryWrites=true&w=majority', { useNewUrlParser: true,
useUnifiedTopology: true },
    function(err, database) {
        if (err) {
          throw err;
        }
        console.log("Connection made to database.")
    }
)

const app = express()
const port = 3000

// Release our API to be hit from other origins
app.use(cors())

// Parse form data if sent in URL encoding
app.use(express.urlencoded({ extended: true }))
// Parser JSON data in the POST body
app.use(express.json())

app.get('/', function(req, res) {
    // Just getting some responses as we build our app.
    console.log('routes are working?')
    res.send('hello world')
})

app.get('/albums', function(req, res) {
    // The argument to find is an empty object, telling the model to not
restrict the search at all.
```

```
    // We can refine this search using this parameter if needed.
    Album.find({})
        .then(function(albums) {
            console.log(albums)
            res.send(albums)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        })
});

// an example of adding one album (to show the functioning of the Album
model, to test routing, and for seeding). It is a non-essential route.
app.get('/add-one-album', function(req, res) {

    albumToAdd = new Album({
        title: "Highway to Hell",
        artist: "AC/DC",
        year: 1980,
        genre: "rock"
    })

    albumToAdd.save()
        .then(function(album) {
            console.log("ITEM SAVED!")
            console.log(album)
            res.send(album)
        })
        .catch(function(err) {
            console.log(err)
        })
})

// A route to load several albums - basically a seeding route
app.get('/add-several-albums', function(req, res) {
    // Mongoose's bulk insert option
    // We are passing the array of albums from our file through to the
database.
    Album.insertMany(data.albums)
        .then(function(albums) {
            console.log(albums)
            res.send(albums)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        });
})

app.post('/albums', function(req, res) {
    console.log("ALBUM POST route hit")
    console.log(req.body)
    albumObject = {
        title: req.body.title,
        artist: req.body.artist,
        year: req.body.year,
        genre: req.body.genre
    }

    albumToAdd = new Album(albumObject)

    albumToAdd.save()
        .then(function(album) {
            console.log("ITEM SAVED!")
            console.log(album)
            res.send(album)
        })
        .catch(function(err) {
            console.log(err)
        })
})

app.delete('/albums/:id', function(req, res) {
    console.log("DELETE ROUTE hit")
    console.log(req.params.id)
    Album.findByIdAndDelete(req.params.id)
        .then(function(x) {
            console.log(x)
            res.send(x)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        })
})

app.listen(port, function() {
    console.log(`Example app listening at http://localhost:${port}`)
})
```

Integration

Now that we have a front end with the power to reach out to the internet with HTTP requests, and a backend that can receive incoming requests and respond with JSON data, we have the ability to make a full-stack app. We can choose to turn Axios from our front-end `index.js` file towards our API, and then take hold of the data that is returned.

We can build our record store.

We might start with this code in the `index.html` (in the `<body>`):

```
<h2> Record Store </h2>
    <div id="all-records-section">
    </div>
    <h2> Add an album: </h2>

    <form id="new-album-form" name="album-form">
        <div class="form-input">
          <label for="title">Album title:</label>
          <input type="text" name="title" id="title" required>
        </div>
        <div class="form-input">
          <label for="artist">Artist: </label>
          <input type="text" name="artist" id="artist" required>
        </div>
        <div class="form-input">
            <label for="year">Year: </label>
            <input type="text" name="year" id="year" required>
        </div>
        <div class="form-input">
            <label for="genre">Genre: </label>
            <input type="text" name="genre" id="genre" required>
        </div>
        <div class="form-input">
          <input type="submit" value="Add the album!">
        </div>
    </form>
```

And then some code like this in our `index.js`:

```
console.log("index.js is running")

// Just an early basic check to see that we are connecting to our API
axios.get('http://localhost:3000/')
    .then(function(response) {
        console.log("OUR API!")
        console.log(response.data)
    })
    .catch(function(err) {
        console.log(err)
    })

// Reaching out to our API to get all the albums
axios.get('http://localhost:3000/albums')
    .then(function(response) {
        console.log("OUR API!")
        console.log(response.data)
        // Find where we want to put the data
        const recordsSection = document.querySelector('#all-records-
section')
        // renaming for clarity
        const allRecords = response.data
        // Looping through the data, and appending to the DOM
        allRecords.forEach(function(record) {
            const newDivElement = document.createElement('div')
            newDivElement.innerHTML = record.title
            recordsSection.appendChild(newDivElement)
        })
    })
    .catch(function(err) {
        console.log(err)
    })

// Find our form
const albumForm = document.querySelector('#new-album-form')

// Process the sumbission of the form
albumForm.addEventListener('submit', function(event) {
    // Stopping the normal action of the form so that we can handle things here
    event.preventDefault()
    console.log(event.target)
    console.log("Submitting the ALBUM form")
    // Getting the form data and converting it to JSON
    var formData = new FormData(albumForm)
    const plainFormData = Object.fromEntries(formData.entries())
    const formDataJsonString = JSON.stringify(plainFormData)
    console.log("form data")
    console.log(formDataJsonString)
    // Posting that data to our API
    axios.post('http://localhost:3000/albums', formDataJsonString)
})
```

The above code can get you started if you need. Very happy to get questions about anything that is going on in these files - please don't hesitate.

Next steps

Note really well that much of this is predicated on getting things properly set up in the back end, with a DB up and running. The code above should point you in the right direction. But otherwise we have the nascent stages of a real app, a real web app.

From here it is up to you. There are several more routes to complete on the backend to get things fully rolling. The `index.js` code on the front end is a little messy and will run into trouble. Our Express API is going to get unweildy soon, and so cleaning up the code and separating things out will be a good idea soon. There is plenty of other obvious things to do on the front end too, from styling to more and better functionality. But you are on your way.

We can learn about those next steps in the coming weeks. Hopefully though, this document helped you to be able to put it all together so that you might be able to get a start. Or it may have triggered some questions that I can answer.

Good luck, and keep at it.