Full Stack Hello World

Our Controller Logic

Now that we have the basic set up, let's review each step as we create the rest of our project in the form of a "Hello World" full stack MERN application.

Let's start by adding a controller within the 'controllers' folder, called `person.controller.js` :

Code Block 1 - server/controllers/person.controller.js

```
module.exports.index = (request, response) => {  //We are exporting a key:val pair of index : function
    response.json({     // This is where we're setting the API's response to the requesting client
        message: "Hello World"
    });
}
```

Our Routes

Next, in the routes folder, let's create the `person.routes.js` file.

After importing (`require`) our controller file and assigning our controller's exported logic to a variable, we will export an anonymous arrow function that requires an "app" as its parameter. The parameter's value (argument) will be assigned in our server.js, which will be the express server object itself.

The function consists of lines that include our HTTP verb, the request's route and what our API is supposed to do when we hit that route. In the following, a "get" (verb) request to the "/api" route is to look to PersonController, find the value that goes with the "index" key and execute that value (which is a function):

*(See **Code Block 1** for that key:val pair)*

Code Block 2 - server/routes/person.routes.js

```
const PersonController = require('../controllers/person.controller');  //Import the code from Code Block 1
module.exports = (app) => {
    app.get('/api', PersonController.index);
}
```

Linking Our Routes to Our server.js

After setting up our controller logic and our routes, we have a route that ends in '/api' which will simply respond to the client request with a response object containing a "message" key and value of "Hello World". Let's link to this in our server.js:

Code Block 3 - server/server.js

```
const express = require('express');
const app = express();
require('./routes/person.routes')(app);   // We're importing the routes export.
// These two lines are the long-hand notation of the code above. They better show what's going on.
    // const personRoutes = require("./routes/person.routes");  <-- assign the exported function to a const
    // personRoutes(app);     <-- invoke the function and pass in the express "app" server
app.listen(8000, () => {
    console.log("Listening at Port 8000")
})
```

Now, when we make a request to 'localhost:8000/api', our server will send back a response object with the following JSON object included: `{message: "Hello World"}`

We have our back end set up, but let's display this in our front end.

Let's now start setting up our React Front-end

Change directories into your React project, called `client`, and run the following:

```
npm install axios
```

We are installing the axios library so we can easily make a request to our backend.

In planning ahead, let's create a new folder inside the `src` folder to hold all of your functional components. We will name it "components". We will discuss the purpose of our "views" folder later on!

Setting Up Our PersonForm Component

Within the `src/components` folder, create a new file called `PersonForm.js`. In the `PersonForm.js` file, we will make an api request and display our message. We will be using the `useEffect` hook in order to make the api call immediately upon rendering and save the message in state.

Code Block 4 - client/src/components/PersonForm.js

```
import React, { useEffect, useState } from 'react'
import axios from 'axios';
const PersonForm= () => {
    const [ message, setMessage ] = useState("Loading...")
    useEffect(()=>{
        axios.get("http://localhost:8000/api")
            .then(res=>setMessage(res.data.message))
            .catch(err=>console.log(err))
    }, []);
    return (
        <div>
            <h2>Message from the backend: {message}</h2>
        </div>
    )
}
export default PersonForm;
```

We are setting the default message as "Loading...". Next, in your `App.js` file, you will need to import the `PersonForm` component and include it as a child of our App component as follows:

Code Block 5 - client/src/app.js

```
import React from 'react';
import PersonForm from './components/PersonForm';
function App() {
  return (
    <div className="App">
      <PersonForm/>
    </div>
  );
}
export default App;
```

At this point, you can start your backend server using the command `nodemon server.js` and your React app via `npm run start` in two different consoles / terminals. Once you have them both running, you should be able to visit 'localhost:3000'. Once you visit it, you will end up just

seeing "Loading..." as your message. Why is that? This is because we are making a request to our server from a different origin.

If you look in your browser's console it will show an error message pointing to CORS (Cross Origin Resource Sharing). The browser considers the two servers to be different "origins" because React is running on port 3000 and your node / express API server is running on port 8000. Browsers by default consider this an unsafe practice and so they require the API server to specifically allow this type of request. The React client **cannot** be configured to override this behavior. For more information see the MDN documentation here.

So, let's stop your express server and install an extra package within your server.

CORS

```
npm install cors
```

This will install the ability to make cross-origin requests. Now, we will need to add the cors functionality to our `server.js` as the following code shows:

Code Block 6 - server/server.js

```
const express = require('express');
const cors = require('cors')    /* This is new */
const app = express();
app.use(cors())                 /* This is new */
/* This is a short-hand notation we use: */
require('./routes/person.routes')(app);
/* These two lines are the long-hand notation of the above code, to better illustrate
what's going on: */
/* const personRoutes = require("./routes/person.routes"); */
/* personRoutes(app); */
app.listen(8000, () => {
    console.log("Listening at Port 8000")
})
```

Restart your server again and refresh your browser to see that we are now able to make cross-origin requests in our project. When you refresh your React app, you should briefly see "Loading..." as the message, which will then be replaced by "Hello World". Congratulations! You have made your first full stack MERN app!

Next, we will use this basic boiler-plate in order to build out a Full-Stack CRUD (Create, Read, Update, Destroy) MERN application.