React app

<https://medium.freecodecamp.org/part-1-react-app-from-scratch-using-webpack-4-562b1d231e75>

If npm run start fails, try installing babel-loader v7

npm i babel-loader@7 rimraf -D

One more thing to install. We’ll explain it later.

npm install @babel/polyfill --save

in package.json > scripts > start (line 8 ) change that line to

"start": "webpack-dev-server --mode development --port 4000 --open",

This will allow us to run development server on a port of our choice.

We’ll add one more line in that file, just below the previous line.

"build-dev": "webpack --mode development && rimraf ../public/index.html",

We need to change our webpack config a bit. To make it easy to get start, just replace the contents of webpack.config.js with this:  
**You can keep a copy of old webpack.config and compare it with new one to see what is different.**

**const** HtmlWebPackPlugin = require("html-webpack-plugin");

**const** path = require('path');

module.exports = (env, argv) **=>** {

**const** htmlPlugin = new HtmlWebPackPlugin({

template: "./src/index.html",

filename: argv.mode === 'production' ? "../views/index.ejs" : 'index.html'

});

return {

module: {

rules: [

{

test: /\.jsx?$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

}

]

},

plugins: [htmlPlugin],

devtool: 'source-map',

devServer: {

historyApiFallback: {

publicPath: "/",

contentBase: "./public",

hot: true,

rewrites: [

{ from: /^.+$/, to: '/' },

]

}

},

output: {

path: path.resolve(\_\_dirname, '../public' */\*ifProd('../', 'build')\*/*),

publicPath: '/', *//ifProd('./public', ''),*

filename: '[name].js' *// output bundle.js and vendor.js*

}

}

};

Now before getting started, there is one more thing to do. It is called a source-map.

A lot of the code we write are going to be transformed by a bunch of tools, mainly babel and webpack. When there is an error in our code, it won’t be shown until it is run on the browser. The code that is being run on the browser is the transformed version , not the original code we wrote. This is where the source map comes handy. Sourcemap is usually a mapping file generated for the debugger tools of your browser, for example chrome’s dev tools. These devtools read the mapping file and show you the source of the error as you wrote it.

To do that, go to webpack.config.js we use the devtool proprety

devtool: 'source-map'

What we don’t have is, babel is still not setup to generate the map. Go to .babelrc file and add the following as the last property

"sourceMaps": true

Okay, now react project is setup, if you are the thinking type you should have noticed that we are not using our nodejs server but still getting the webpage in the browser. Well, the react development setup includes a development server which is good to have if your react app doesn’t depend on your api server.

This might be confusing to you, but another way to put, A react app is just client side javascript that runs on a browser. It can work from any location within your computer as long as it is embedded in an html file. To make it available online you’ll need a web server though.

The development server is used so that each time a change is made to frontend, the api server doesn’t have to be restarted. In production you may make changes to api server so that it will serve the final react app too.

Now we are going to replace our html templates we rendered using ejs with a react app.

React app doesn’t have multiple template files but just one html file which is the entry point for the app. All the other screens are dynamically inserted using javascript. However we are not going to remove ejs completely, our entry point will be an html file rendered by ejs.

In our react app we’ll have just 2 pages.

1. Login
2. Profile page

With this 2 pages we will setup the basic framework so that you can add more pages if you want

Every code sections in this document references the files inside /react-app directory

Now take a look at ./src/index.js and you’ll see this

At the top of the page you need to add this line

import "@babel/polyfill";

This is because babel is a transpiler, which means it converts the code you write to a different form. A polyfill is a piece of code that mimics a new language feature which is not available in older version browsers ( or javascript engines ). If we use a language feature in our code that would not work on an old browser, polyfill will substitute that with a predefined code that can achieve the same result. This won’t happen magically, babel will replace your code with the polyfill while it is transpyling.

import React from "react";

import ReactDOM from "react-dom";

This simply imports react and react-dom, two dependencies required to make a react app work.

*“react “* does all the react magic while “react-dom*”* is the component that couples everything with the DOM.

**const** Index = () **=>** {

return <div>Hello React!</div>;

};

This is the most basic thing in a react app called a component. In react everything is composed of components. A simplest form of a component is a function that returns a react element.

Now you’ll be confused , that the function returns a <div> which is not a react-element but a native html element.

The trick here is that what you see inside the function is not html. It is called JSX, a special syntax used by react apps. Now, JSX is not mandatory, you can do the same using React functions, but this looks nice and easy to follow.

In our development environment, we are using something called babel to convert the JSX to equivalent javascript. So our <div>Hello React!</div>; would look something like this when compiled.

React.createElement('div', null, “Hello React!”);

The interesting part about react components is it is composable, meaning a component can be made of other components. For example:

**const** AnotherComponent = () **=>** {

return <div>I am parent of “<Index/>”</div>;

};

This will result in a html as this:

<div>I am parent of “<div>Hello React!</div>”</div>

There is more about components but we’ll move on for now.

ReactDOM.render(<Index />, document.getElementById("index"));

This basically finds the element with id “index” in our html file where our js files are embedded and inserts the rendered react elements as its equivalent . In this example this file is the index.html.

Now we start building our app.

Our app has 2 major states, that is when user is logged in and when he is not.

We have to do a bit of planning ahead of that on what all pages we are going to have and what all parts of these pages are going to be reused.

In this new app, we’ll have mainly 3 pages namely signup, login and profile page.

Login and signup are pretty straight forward.

Profile page is going to be a bit more interesting.

One more thing is input fields and forms are going to be a bit more complex that a plain html form.

We can definitely use the html form but it causes the page to reload, which defeats the purpose of using react. So to do it nicely, we’ll create a react component which mimics the behaviors of an html form. Luckily someone has already done it for us and published in on github, (power of a large js community 😉, that’s one of the reasons for the huge success JS has attained)

Unfortunately, this code is not in a ready to use module form. So we’ll have to create a component out of this code. So head over to <https://gist.github.com/gearnode/1bfafc0cd060fbc505ff42be38f8169a>

Download the [**Form.jsx**](https://gist.github.com/gearnode/1bfafc0cd060fbc505ff42be38f8169a#file-form-jsx) and [**FromObject.js**](https://gist.github.com/gearnode/1bfafc0cd060fbc505ff42be38f8169a#file-fromobject-js)  files to your components folder. Well, even better, create a folder /**Form** inside the **/src/components** folder and move these files there. Don’t worry about the .jsx extension, babel can read that too.

Remember to remove this section from **Form.jsx**

**static** propTypes = {

        method: PropTypes.string.isRequired,

        action: PropTypes.string.isRequired,

        encType: PropTypes.string.isRequired,

        serializeForm: PropTypes.func.isRequired,

        onResponse: PropTypes.func.isRequired

    }

Now in the handle submit function do these changes ( from line 52)

**const** serializeForm = this.props.serializeForm || JSON.stringify

**const** fetchRequest = { body: serializeForm(fromData),

                 headers: { 'content-type': this.props.encType || "application/json" },

                 method: this.props.method || "GET"

}

Not done yet! We copied someone’s code. We should give attribution. Well no one is going to sue you if you don’t do that here but down the road when you are working in a corporate setting, these things come into play. Giving attribution for an opensource work is just the nice thing to do. Beyond that there are legal implications to that too. The way you attribute depends a lot on the license involved which is to be dealt with legally, not something a developer should spend a lot of time on. Stackoverflow has a policy that all code you use from them is under *Creative Commons Attribution-ShareAlike 3.0* which requires you to add a link to the post as comment in your code. So when you are not sure what to do, just do that, or get help from the legal team in your company.

Open these files, in the beginning of these files add the following line

*/\**

*Author: Bryan FRIMIN*

*github/@gearnode*

*Link: https://gist.github.com/gearnode/1bfafc0cd060fbc505ff42be38f8169a*

*\*/*

If you are copying code and using it within a file of your own, may be slightly modified, you could still do the following

*//Attribution: https://gist.github.com/gearnode/1bfafc0cd060fbc505ff42be38f8169a*

There is another huge advantage for doing this. When you go back to this code after 3 months and have no clue what this does, head over to the link and it’ll tell you everything.

Okay, we can start with our signup form

Create a file **./src/components/signup.jsx**

Import react.

Create a class component which extends React.component , well yeah this is new.

import React from "react";

export default **class** Signup **extends** React.Component {

render() {

}

}

We saw earlier that a component can be a plain function which returns some JSX. It can also be a complex component which extends a react.component class.

This article very well explains this difference. <https://medium.com/@Zwenza/functional-vs-class-components-in-react-231e3fbd7108>

But the TLDR version is a functional component is pretty basic. It can do a few things within its scope, but that’s pretty much it. This function will be called each time there is a change in props. In return the function creates new components each time it is called.

On the other hand, the class component gives you much more control. It can give you lifecycle update calls, like when a prop is going to change, when this component is created, or is going to be destroyed etc. Above all, it allows you to have a **state** which a functional component doesn’t have.

Okay, so what is this state and prop. Yeah, we didn’t talk about any of that, I know you don’t understand most of what is going on. But now we have a working environment where we can try out more stuff.

So a react app works on a basic concept of building up an app using blocks called components. A component can be composed of other components or just basic html components.

A component will have an internal state (it is optional) which is a JS object (a key value store). This state is updated by calling a *setState(updated\_state)* function. This sets in motion an update to the UI. It basically calls the render function (only available in a class component).

Now there is one more data store which is **props** . A prop is whatever data you pass into a component from its parent component. For example if you have a component as below:

<FancyBorder color="blue"/>

The props received in FancyBorder is

{ color: "blue" }

Which again, is a simple javascript object. A change in props will also trigger a re-render (call the render function). Yeah basically that’s what all react does.

Now think about this scenario, we have some component called <BigBox> and we have a state inside it as { color: “blue” }

And Bigbox has FancyBorder component used inside it like this

<FancyBorder color={this.state.color}/>

That’s right, we are assigning the prop value for FancyBorder from a state variable.

If we change the state to { color: "green" }, it will re-render the <BigBox>. But now the value passed in to props of <FancyBorder> is changed. So it will trigger a re-render of <FancyBox > too. This keeps on going until there is no changes. This is how react updates your UI. And that is the only thing react does.

There are other frameworks to manage states in a different way so that your component states are consistent with all the actions you do and the states in other components. For this tutorial we are staying away from them as it will complicate everything too much. One tool you might have heard about (if you have heard about react) is Redux.

Back to our signup component

Remember the ejs files we had from our server app?, head over to that ../views/signup.ejs, copy everything from <form… to </form> including the tags, past it inside the render function (as return). Change <**form …>** to <**From** ..>

One more thing, you’ll have to import **Form** which is the file we created from github gist. So the finished file will look like this

import React from "react";

import Form from "./Form/Form.jsx";

export default **class** Signup **extends** React.Component {

**constructor**(props, context) {

super(props, context);

this.state = {};

*// this.state = {error: "sample error"};*

};

render() {

return <Form method="POST">

<input type="text" name="firstName" placeholder="firstName" /><br />

<input type="text" name="lastName" placeholder="lastName" /><br />

<input type="password" name="password" placeholder="Password" /><br />

<input type="password" name="repassword" placeholder="Repeat Password" /><br />

<input type="email" name="email" placeholder="Email" /><br />

<input type="submit" value="Register" />

</Form>

}

}

Now, you’ll need to display any errors that may occur when doing signup. To do that, we’ll add the following code inbetween email field and submit button as shown below

...

<input type="email" name="email" placeholder="Email" /><br />

<br/>

<b>{this.state.error}</b>

<br/>

<input type="submit" value="Register" />

...

This won’t display anything right now as we don’t have any errors defined in the state. But to test it, you can uncomment line 11.

For now, this is going to be the structure for our other 2 pages too ( login and profile… they are forms just like this.) . So what you should do next is , make 2 copies of **signup.jsx** and rename them as **login.jsx** and **profile.jsx**

Next, go to our ejs templates folder again. Open the corresponding ejs file for both these components and copy everything inside the **<form>** tag except the last <input type="submit" value="…"> tag. Replace everything inside the **<Form>** tag in your respective jsx file (react component) with what you copied from ejs file. Remember to keep the last 4 lines that goes like this.

<br />

<b>{this.state.error}</b>

<br />

<input type="submit" value="…Action name…" />

Do this for both **login.jsx** and **profile.jsx**

Remember to replace the …Action name… to whatever action you are doing. (Login, Signup, Save etc)

Also, while doing this for profile.jsx, remember to remove all the ejs bit like **value="<%= firstName %>"**, value="<%= lastName %>" and value="<%= email %>"

These components do nothing right now as we haven’t used it anywhere. So we’ll do something to display these components in our app.

To do that, one more thing we need is routing, that is, in both these states, we’ll have multiple pages that are accessed with different url paths. To do routing, we’ll be using a router module. For react apps,  
react-router is the defacto router module.   
You might be a little confused now, What is routing in react. Well, we did routing in our server side app. When you make a request, it hits our node server first, then our routing in server will analyse the url and use the appropriate controllers to handle that request. Then it will display the appropriate page.

Routing in react is pretty much the same thing but happens after the page is loaded. Once the react app is loaded and initialized, the router in the react app will analyze the url again and determine which components to display and which not to.

Basically, for a url resulting in a react-app page, you can just skip routing on the server side. i.e do not do anything in the server side and just display the react app. Further routing will be done on the react app.

You can read more about react-router here <https://reacttraining.com/react-router/web/guides/basic-components>

So to start with routing, we’ll first remove everything from the index.js except the first 2 import lines. Then add the following

import { BrowserRouter } from "react-router-dom";

ReactDOM.render(

<BrowserRouter>

<Routes/>

</BrowserRouter>,

document.getElementById("index")

);

Okay, now we actually do not have a <Routes /> component yet, so we’ll create that now, right next to index.js and paste the following in there

import { Route, Switch } from "react-router-dom";

import React from "react";

import Signup from "./components/signup.jsx";

import Login from "./components/login.jsx";

import Profile from "./components/profile.jsx";

**const** NotFound = () **=>** <div>Page not found</div>;

export default **class** Routes **extends** React.Component {

*// a render function is the absolute minimum requirement for a react component extended from React.Component class*

render(i) {

return <Switch>

<Route path="/signup" component={Signup} />

<Route path="/login" component={Login} />

<Route path="/profile" component={Profile} />

<Route component={NotFound} />

</Switch>

}

}

This is basically a react component. I hope you have a basic understanding of a react component by now. We could have just made a functional component here, because, as you can see this component is doing nothing much, just rendering a couple of components. This is a perfect occasion where we could use a functional component. But we will need to add more functionality later. That is the reason we are using a class component here. NotFound is another example of function component, because it just displays “page not found”.

**const** NotFound = () **=>** <div>Page not found</div>;

The content returned in the render function is basically how you setup routes in react-router. This is documented in react-router documentation page you saw earlier.

Remember that we are using the Routes component inside the index.js. So the content there will be replaced by what we are configuring here depending on the current url. That’s it. If the current url doesn’t match anything given here, it won’t display anything. Notice that we haven’t given a component for base route ‘/’, which means when the app initializes, nothing will be displayed. We need to navigate to one of the three pages defined in the routes.

Notice that, the last route doesn’t have a path specified, so it will match anything. This is a good place to add a place holder.. if nothing matches, say you’re on the wrong page. So this matching sequentially, from top to bottom and when something matches, router won’t look for further matches.

<Route component={NotFound} />

Unfortunately, our server (development server ) is not configured to handle anything other that ‘/’ so if you try to go to any of these pages, it’ll simply say it couldn’t find the page.

To change that, go back to webpack.config.js and add the following after your devtool setting and save it

...

devtool: 'source-map',

devServer: {

historyApiFallback: {

rewrites: [

{ from: /^.+$/, to: '/' },

]

}

}

...

Now restart the server and reload the page in browser.

You can now go to **/signup** route and see the signup form.

To add a little more fun, go to routes.js under..

**const** NotFound = () **=>** <div>Page not found</div>;

add the following..

**const** Menu = () **=>** <div>

<Link to="/" >Home</Link> |

<Link to="/signup" >Signup</Link> |

<Link to="/login" >Login</Link> |

<Link to="/profile" >Profile</Link>

</div>;

Now, Link is a component supplied by react-router-dom, so change your 1st import to include that.

import { Route, Switch, Link } from "react-router-dom";

and change the render function to as follows, make use of the menu component

...

render(i) {

return <div>

<Menu />

<Switch>

<Route path="/signup" component={Signup} />

<Route path="/login" component={Login} />

<Route path="/profile" component={Profile} />

<Route component={NotFound} />

</Switch>

</div>

}

...

Now, go to the browser and see that we have a menu bar (not very pretty…) but it gets the work done.

We have alittle more work to do.

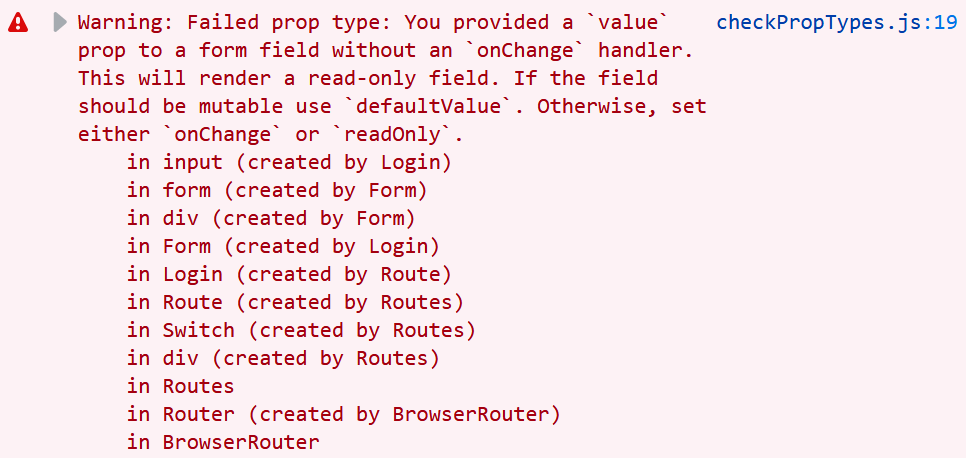
There is a problem with input fields in react. If you specify a value to prefill the field, then it will throw some errors. Lets go ahead and try that.

Go to profile.jsx. add a valueattribute to the firstname field

<input type="text" name="firstName" placeholder="firstName" value="yourname" /><br />

Save it and now go to profile page in browser. Try to type something in the field… Nothing , right?

Okay if you check the browser console, you’ll see this warning



What it says is that, when you set a value to an input field, it should be from a variable (preferably a state variable) then use an onChange listener to change that value in the state so that it’ll be consistent with state update cycles.. lame but, yeah.. that’s how react rolls..

Since we are using a browser defined form and form handling methods, we don’t need these values in our state. To get around this, we can define a component that takes care of this.

Head over to **Form.jsx** and add the following code at the end

export **class** Input **extends** Component {

**constructor**(props) {

        super(props)

copy value to state

        this.state = { value: props.value }

Binding, keeping the class context

        this.changeHandler = this.changeHandler.bind(this);

    }

Simply updates the value in state so the, next render will display the updated value

    changeHandler(event) {

        this.setState({ value: event.target.value });

    }

    render() {

*// creates a shallow copy of this.props*

**let** otherProps = Object.assign({}, this.props);

*// removes value property from otherProps*

        delete otherProps.value;

*// now, we have a copy of all props except value in otherProps*

        return <input {...otherProps} value={this.state.value} onChange={this.changeHandler} />

    }

}

*The {...otherProps} is the spread syntax for jsx, it simply puts all the key-values in the variable (it should be an object) as props to that component.*

Now go to profile.jsx and change the **Form** import to this…

import Form, {Input} from "./Form/Form.jsx";

Now change the firstname input field from <input..> to <Input…> (note the capitalization)

<Input type="text" name="firstName" placeholder="firstName" value="yourname" /><br />

Save the file, go to browser and try typing something in place of yourname.

**Do the same for lastname and email fields too. These fields are going to be autofilled with your details when you are logged in.**

**The next bit is just for fun.**

Our profile page looks like a form. Well, that’s a little odd, that’s not how a profile page is supposed to look like.

So we’ll make it look a little different, We’ll remove all text boxes, print the text instead and give an edit button next to each of them. When you click the edit button, it’ll change to a textbox.

Again, we’ll make a component for that so that we can re-use it. Head over to Form.jsx

Paste this in the end of the file.

export **class** EditableText **extends** Component {

**constructor**(props) {

        super(props)

        this.state = {

            edit: !props.value,

            originalVal: props.value,

            currentVal: props.value

        }

        this.toggleEditState = this.toggleEditState.bind(this);

        this.blurHandler = this.blurHandler.bind(this);

    }

    toggleEditState(isEditable) {

        this.setState({ edit: isEditable });

    }

    blurHandler(event) {

        this.setState({ currentVal: event.target.value });

        console.log(this.state.currentVal, event.target.value)

        this.toggleEditState(!event.target.value || false);

    }

    render() {

**let** updated = (this.state.originalVal != this.state.currentVal);

**let** strikeStyle = updated ? { textDecoration: 'line-through' } : {};

*// creates a shallow copy of this.props*

**let** otherProps = Object.assign({}, this.props);

*// removes value property from otherProps*

        delete otherProps.value;

        return <div>

            {this.state.edit ?

                null

                : <div>

                    <div style={strikeStyle}>{this.state.originalVal}</div>

                    {updated ? <b>{this.state.currentVal}</b> : null}

                    (<a onClick={() **=>** this.toggleEditState(true)}>edit</a>)

            </div>

            }

            <Input {...otherProps} value={this.state.currentVal || ''} onBlur={this.blurHandler} type={this.state.edit?'text':'hidden'}/>

        </div>

    }

}

What does it do, well, Try to figure that out yourself..

Now go to profile.jsx and change all the **Input** component reference to EditableText (import and usage)

Save everything, refresh your browser. Go to profile page and see how it looks. Try typing in some values.

**Submit actions**

Take signup.jsx file to start with. When you click the register button, it will submit the form but unlike before, nothing will happen in your UI because the submission happens as an ajax request. It won’t reload your entire page, but submits it in background. To do something in response to the submission, show the errors or go to another page, we will add an onResponse attribute to our form component. This was actually defined in the code we just copied from github. So change it like this

...

return <Form method="POST" onResponse={this.onSubmit} action="/api/signup">

...

The *action* prop is the url where the form is to be submitted. Earlier in the ejs file, we hadn’t mentioned it because it was being handled by the same url of the current page (that is the default behaviour). But now we are using a different url (api end point ) to submit to.

But we haven’t defined the onSubmit function yet. So we’ll do that next. Add the following above our render function

...

**async** onSubmit(response) {

**let** error = ""

if (!response) error = "Something went wrong while submitting. Please try again";

else if (!response.ok) error = await response.body.json();

else this.props.history.replace('/)

this.setState({ error: error && error.errorMsg })

}

render() {

...

A response is a **Response** object returned by the browser and passed to the function by our form component. You can read more about this object here <https://developer.mozilla.org/en-US/docs/Web/API/Response>

response.ok will have a truthy value if the http request returned an OK status code (2xx)

if its not true, whatever the server responded with is an error message.

If no errors, then switch to login page.

One more thing, in the constructor of each of signup.jsx , login.jsx and profile.jsx add the following

this.onSubmit = this.onSubmit.bind(this);

Now copy the onSubmit function and paste it in login.jsx and profile.jsx. In these pages we have forms and submissions has to be handled there too. We are going to handle them the same way so that we don’t have to write new handlers for each form. But there is going to be one difference though.. when a submission is successful, each form reacts differently. For example, signup when success, redirects to home (‘/’) . Login when success, calls a loginSuccess callback (which we will add later)

Profile form when success, just reloads the data and displays updated values.

One more thing you need to change is

In login.jsx

**async** onSubmit(response) {

**let** error = ""

if (!response) error = "Something went wrong while submitting. Please try again";

else if (!response.ok) error = await response.body.json();

else this.props.onLoginSuccess();

this.setState({ error: error && error.errorMsg })

}

And

return <Form method="POST" onResponse={this.onSubmit} action="/api/login">

in profile.jsx

**async** onSubmit(response) {

**let** error = ""

if (!response) error = "Something went wrong while submitting. Please try again";

else if (!response.ok) error = await response.body.json();

else {

this.props.refreshSession();

alert("Updated !!");

}

this.setState({ error: error && error.errorMsg })

}

And

return <Form method="POST" onResponse={this.onSubmit} action="/api/profile">

## Server side

We’ll now need to write the required api end points. Go to the serverside code, we have a controller in there called **apiControllers.js** if its not there, create the file in the controllers folder. Remove everything in there and add these functions (controllers there)

**const** UserController = require('./userControllers');

Handles login requests. This is similar to login in viewsController. It uses the userController to do the actual login and produces the output to api call.

exports.login = **async** (ctx, next) **=>** {

try {

**let** email = ctx.request.body.email;

**let** password = ctx.request.body.password;

await UserController.login(email, password);

ctx.status = 201;

ctx.body = '';

} catch (errorMsg) {

ctx.status = 401;

ctx.body = {errorMsg};

};

}

For signup

exports.signup = **async** (ctx, next) **=>** {

try {

await UserController.signup(ctx);

ctx.status = 201;

ctx.body = '';

} catch (errorMsg) {

ctx.status = 401;

ctx.body = {errorMsg};

};

}

Returns the user details.

exports.profile = **async** (ctx, next) **=>** {

ctx.status = ctx.session.user ? 200 : 404;

ctx.body = ctx.session.user || '';

}

Saves changes to profile

exports.updateProfile = **async** (ctx, next) **=>** {

try {

await UserController.update(ctx);

ctx.status = 201;

ctx.body = '';

} catch (errorMsg) {

ctx.status = 401;

ctx.body = {errorMsg};

}

}

For signout

exports.signout = **async** (ctx, next) **=>** {

await UserController.signout(ctx);

ctx.status = 201;

ctx.body = '';

}

This is a fallback response, i.e if the api endpoint being accessed is not defined in the server, then this controller will be called

exports.unknownEndpoint = **async** (ctx) **=>** {

ctx.status = 404;

ctx.body = "Method not defined";

}

### Explaining…

I hope you know what a try-catch block is, what you might not know is this part,

ctx.status = 201;

ctx.body = '';

ctx.status, is where you set the http response status code.

200, 201 means OK, 404 means not found, 403 – forbidden, 401 – unauthorized.

500- server error.

These are the most common we use. For more code reference read the wiki page

<https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>

ctx.body = '' is used to set the response body. You can set a string or a JS object there. A string will be transferred as text while js object will be converted to JSON.

Once you set this value, your response is sent. You are not allowed to change this value after that.

That’s basically everything for controllers here as we already discussed everything going on in these controllers when we did the serverside part earlier.

Next thing is setting up the routes so that these parts of the controllers are accessible through http requests.

Go to routes.js (server side)

Just before

module.exports = router;

add the following code:

create a new instance of router to keep the api routes separate from our older routes.

*//Api routes*

*// just to keep them seperate from all other routes*

**const** apiRoutes = new Router();

A new session checker middleware. The previous one was for html pages, it would redirect to login page. Now we don’t need that. We just want to tell that request is not authenticated. The requesting app should be able to decide whether to go to login page or not. That is how api’s should be.

**let** apiSessionCheck = sessionCheck(ctx **=>** {

ctx.status = 401;

ctx.body = ''

})

The routes that need to be mapped to controllers

apiRoutes

.post('/login', ApiCallHandler.login)

.post('/signup', ApiCallHandler.signup)

.get('/profile', ApiCallHandler.profile)

.post('/profile', apiSessionCheck, ApiCallHandler.updateProfile);

.get('/signout', ApiCallHandler.signout);

This attaches the new routes to our main route object as a sub route going under ‘/api’. All the above mentioned api endpoints should have the ‘/api’ prefixed to be accessed. Eg: ‘/api/login’

*// now all requests starting /api/{path} will be handled by apiRoutes*

router.use('/api', apiRoutes.routes(), ApiCallHandler.unknownEndpoint);

Note that we are using our unknownEndpoint controller here at the end. What it does is, if none of the routes match, it’ll give control to this function which would simply respond with a 404 code.

One more thing to do, is that currently we are depending on the development server to serve the react app. We are going to change that. From now on, our api server and the react frontend will be served by the same server at port 8080.

To do that we are going to remove all the routes we previously had (the ejs templates)

Comment out all of these (line 10 to 20)

router

.get('/', console.log('sssssss') Views.welcome)

.get('/about', Views.about)

.get('/contact', Views.contact)

.get('/login', Views.loginPage)

.post('/login', Views.login)

.get('/signup', Views.signupPage)

.post('/signup', Views.signup)

.get('/profile', viewSessionCheck, Views.profile)

.post('/profile', viewSessionCheck, Views.updateProfile)

.get('/signout', viewSessionCheck, Users.signout, viewSessionCheck);

Now right before the last line, add the following

*// now all requests starting /api/{path} will be handled by apiRoutes*

router.use('/api', apiRoutes.routes(), ApiCallHandler.unknownEndpoint);

router.get('(.\*)', ctx **=>** ctx.render('index'));

module.exports = router;

Again, we don’t have the index.ejs template yet. But we are going to make webpack create that for us.

Open **webpack.config.js**

Find this section (line 3-6)

**const** htmlPlugin = new HtmlWebPackPlugin({

template: "./src/index.html",

filename: "./index.html" to -> ifProduction("../views/index.ejs", 'index.html')

});

Then go to the end of exported object and add this

...   
output: {

path: path.resolve(\_\_dirname, '../public'),

publicPath: '/,

filename: '[name].js' *// output bundle.js and vendor.js*

}

...

This path is where our bundled javascript files will be saved.

We have one more problem though, our generated files go to /public But we cant access it through our server as that path is not handled in our routes. To serve static files either we could write a controller and route (which is cumbersome) or we could just use a module

We will be using a module called koa-static

npm install koa-static

open **index.js**

before app.use(koaBody());

add this

**const** serve = require('koa-static');

app.use(serve('public'));

To make it run, first we need to build our react app. This step will generate a couple of files. We made the changes to webpack.config to specify where each built file should be saved. To build, go to the react app directory (in command line) and run this once

npm run build

and then run

npm run build-dev

Now restart your server, and go to localhost:8080, it should show you the react app.

For the following builds, you just need the build-dev part alone. In case the server throws an error that the template file is not present then run npm run build part again

We are not giving up the dev-server yet.

## Navigation

Back to react app. Now our server side is ready, so we can change our react app’s navigation to work in the proper flow. First thing, remember the menu bar we put there to preview all the pages we have? Remove that.

Go to routes.js and comment out the <Menu /> component from render block.

...

return <div>

{*/\* <Menu /> \*/*}

<Switch>

<Route exact path="/" component={(routeProps) **=>** this.ifLoggedIn(Home, routeProps)} />

<Route path="/signup" component={Signup} />

<Route path="/login" component={Login} />

<Route path="/profile" component={Profile} />

<Route component={NotFound} />

</Switch>

</div>

...

You’ll notice that we have added something inside the switch. You’ll also notice that the ifLoggedIn function is not defined as of now. So above the render function, you’ll define that.

*//custom methods*

ifLoggedIn(component, defaultProps) {

if (this.state.session)

return React.createElement(

component,

Object.assign({ user: this.session }, defaultProps)

);

return <Login

onLoginSuccess={**async** () **=>** { await this.updateSession(); }} />

}

This function checks if the session is set (if loggedin or not.) if yes, it’ll create an element of the component passed and return it. If not, it will return an element of login component.

But in the login component, we are passing a onLoginSuccess callback that calls updateSession which we need to define.

**async** updateSession() {

**let** user = await fetch('/api/profile', { method: "GET" });

this.setState({ session: user });

}

This function is async because it uses await in its body. It has to wait for the ajax call to complete.

Whatever we receive from the api call, we set it in state.

Now that have a state, it needs to be initialized in the constructor. So, we will add a constructor next.   
Another thing is that currently when you open the app the 1st time, we haven’t set any session, it will be empty. SO the app will show the login screen, then when you are logged in, it will set the session and then show the profile page. But if we have already logged in previously, Then the app has to show the profile page directly when you open it. ( As in facebook, when you close the tab and come back later, it won’t ask for login). To do that, we should call the updateSession before everything is initialized. We will do that in constructor as well. There are other places, like a lifecycle callback, to make this call, but this is our routes component and it will be initialized only once, so doing that in constructor is fine.

*// constructor*

**constructor**(props) {

*// if there is something to be run in constructor, it goes here*

super(props)

this.state = {};

}

Now, we are showing login page based on the login state, which means we don’t need a route for login page anymore. So you can remove the following lines.

<Route path="/login" component={Login} />

And

<Route path="/profile" component={Profile} />

As we removed the menu, now we can’t access the signup page. We can add a link to signup page from our login page.

Go to login.jsx and add the following after the submit button.

...

<input type="submit" value="Login" /><br />

<p>Dont have an account?

<Link to="/signup" >Signup</Link>

</p>

</Form>

...

Remember to include Link to the import block

import { Link } from "react-router-dom";

### put back the Nav menu

Remember the nav menu we removed earlier? We might need it back. Why? Any guesses?

Well, once we are logged in, we need to log out.

So we will add a logout mechanism in our menu. But, then, this menu should only be shown when logged in.

So go back to routes.js, uncomment <Menu>, but this time, we will add a few props.

* Session, to decide if the user is logged in or not.
* updateSession, so that when logged out, the app state will update accordingly

as below:

<Menu session={this.state.session} updateSession={this.updateSession} />

We will make little changes to Menu component too (Beginning of same file)

**const** Menu = (props) **=>** {

**async** **function** logout() {

await fetch('/api/signout', { method: "GET" });

props.updateSession();

}

if (props.session) {

*// menu to show if loggedin*

return <div>

<a onClick={logout}>Logout</a>

</div>;

}

*// menu to show if not logged in*

return <div>

{*/\* <Link to="/" >Something</Link> |*

*<Link to="/signup" >Something</Link> | \*/*}

</div>;

}

With this, our app is functional. The functional part is over.

As an exercise, students try to create an about page and a contact page. (doesn’t need to be functional), Add them in menu. They can decide if they won’t to show it in logged in state or not.

### Welcome to Bootstrap.

Now We may add a little visual flare (we wont go deep )

For styling in html (explain what styling means), we use a set of styling rules called css.

Fortunately, some good engineers at twitter made a set of predefined css rules which we can use in our own applications. And they call it bootstrap. Bootstrap is now in version 4 (yeah 3 versions came before this)

Here is a link to a comprehensive hands on tutorial of bootstrap4 , <https://www.w3schools.com/bootstrap4/>

Or the official one here <https://getbootstrap.com/docs/4.0/layout/grid/>

Open our html template file (/react-app/src/index.html) and add the following inside <head> tag.

Bootstrap comes with a javascript component as well, which we are not using as it may interfere with react’s working. We are only using the styling components. Bootstrap has some advanced features like popup windows, carousals etc. These require the js file to be embedded as well (which is highly discouraged in a react app. You can create custom components for that or you could use react specific modules from npm registry. There is a project called react bootstrap available if you are interested but we are not going there as this is too much for the scope of this tutorial. https://react-bootstrap.github.io/

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css" integrity="sha384-GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS" crossorigin="anonymous">

<link href="https://getbootstrap.com/docs/4.2/examples/floating-labels/floating-labels.css" rel="stylesheet">

Also add the attribute class="form-signin" to <section>

So it will look like this now,

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>React and Webpack4</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css" integrity="sha384-GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS" crossorigin="anonymous">

<link href="https://getbootstrap.com/docs/4.2/examples/floating-labels/floating-labels.css" rel="stylesheet">

</head>

<body>

<section id="index" class="form-signin"></section>

</body>

</html>

Now run

npm run build

npm run build-dev

Reload the page and it already looks slightly different right?

Yeah, we are almost there.

Now , in our profile page, the firstname, lastname sections looks a little odd, we need to work on that. To do that we need to go to where the actual html for these sections are defined. Remember, we used <EditableText> component for that. So we go to where EditableText is defined , i.e in Form.jsx

**Note to self**

~~To do next, profile input fields to these dynamic objects~~

~~Write Submit handlers,~~

~~Write api endpoints~~

~~Remove menu~~

~~Implement proper navigation.~~

Ask students to add an about, contact pages. (copy from ejs templates)

Also ask them to hide the menu when logged in and show a logout button in that place.