I.T Professional Skills

KEITH NOLAN & CUAN O'CONCHUIR

Important note

During the creation of this application our github commits got wiped and a substantial amount of the project got uploaded at once.

This was due to a human error by (Cuan O'Conchuir).

It occurred because I was unable to perform a push, due to pull related conflicts and I performed a –force push which wiped our pervious commits.

Martin Kenirons, our tutor was notified of this and we didn't receive any advice as to how to proceed so we operated under the assumption it would not hinder our grades.

Another important thing to note:

There appears to be an error which we are aware of but unsure to fix. The state/lifecycle management between the application and mongo database seems to be one step behind itself, which causes the app to sometimes crash. A simple refresh of the page will fix this issue and the states are loaded in correctly. We were unable to identify the root cause of this, trying multiple fixed but to no avail. The app works fine despite this, but can be a little temperamental.

Introduction

The objective of this project is to build a M.E.R.N (Mongo, Express, React, Node) social media platform for students that implements C.R.U.D (Create, Read, Update, Delete) features and functions. A M.E.R.N application has a front end, back end and a database which all talk to each other and respond to user requests. C.R.U.D on the other hand enables the user to create, read, update, and delete the data they created.

Just like any social media platform or website they can create an account, using their email and creating a password.

The site will incorporate the option to sign up with a gravatar email, upon doing so the photo of that users email will be posted on their account and will display on their profile and will be associated with any posts they write. If the user does not have a gravatar account linked it will just show a default photo.

The system would then authenticate them, and they would be able to continue with the sign-up process. Immediately after being authenticated they will be brought to a page that will inform the user that they have not created a profile yet. They then have the option to select a button "Create Profile" which will enable them to fill in their information.

When creating their profiles, they will enter in information such as their name, their role in the college for example are they a postgrad, a lecturer or exchange student etc. Other things such as what course they are studying, and a bio can also be filled in. This information would then be saved and displayed on that user's profile.

When filling in their profile information there are a few mandatory options that they must include. These are marked by a * the site will not let the user submit their profiles unless these points of information have been filled in. This will be outlined by a red box at the top of the page informing them or what they have to fill in. The box will disappear after a few seconds. If they make the same mistake again the box will appear again. When the user has filled out their information or minimum amount of information, a green success box will appear informing them that their information has been saved the users web page will be populated with the information they have entered.

The information the user entered can be updated or changed at any time with their being options to edit their information or add information to their profile. The information that has been entered will also be able to be deleted by the click of a button if the user wishes. If they do not like a piece of information, they have entered they can simply delete it from their profile and redo it. This includes the user's full profile. The only thing that does not get deleted upon deleting their profile will be the comments they have left on the comments page if they have not already deleted them when their profile was active.

When a user is logged in there will be a navigation bar at the top of the site that allows them to view the different pages and features that the site has to offer. Below are the options that are available.

View Posts

The site will also enable them to communicate with each other through posts. A member can create a post and then other logged in members will be able to view the posts and comment on them, like them or if after they have liked it, they can unlike it. This will enable them to have conversations with each other by viewing posts and commenting on them giving it the feel of a real social media site.

After a post has been submitted to the webpage, The posts will include the name of the person who posted it, their picture (if any) The post will also include a timestamp (time and date the post was created) a like and unlike button to enable users to like and (if they have liked) unlike the post, and a discussion button. The discussion button will enable the writer of the post or any other registered member to comment on this post. Thus starting a correspondence with each other. These posts can be viewed by any registered member and they can partake in the correspondence if they wish or they can start their own conversation. If a user clicks on the name of the person who commented on a post or made the comment in the first place they will be brought directly to their personal profile.

Members

A registered user can also view the profiles of registered members by clicking on the "members" button on the nav bar. This will display a page that contains all the user's profiles including their own. If the user wishes they can click on the profiles to view that user's information. When they are done, they can go back and view other profiles if they wish.

The information included on that person's profile will be their picture, their status in college, the subjects they entered, what year they are in or title. What subjects they are doing, the location they are etc. In other words, all the information that the user entered when creating their profile will be displayed on their public profile.

Dashboard

The dashboard is where the user can update their entered information delete their information or delete their account. There are several buttons enabling them to carry out these actions.

There is an edit profile button, which will bring the user to the page where they can edit their information. There will be an edit education button where they can update the education, they have such as what year in college they are in. They can also update and edit their job history. When the relevant information has been filled out and the minimum requirements have been met, they can update their profiles. The information will be displayed on the dashboard, it can also be deleted here as well. At the bottom of the page there will be a delete profile button where the user can delete their account. Upon clicking this they will be asked if they are sure they want to proceed with deleting their account, if they click yes their account will be deleted, otherwise they can go back to the dashboard.

Logout

When the user clicks the "Log out" option they will be immediately logged out of their account. This ends their session and they will have to log back in with the correct credentials if they want to log back into their profile. When the user has logged out all their information will still be stored the next time, they log back in.

Other information

When a user wishes to signup to the account there are a few strict guidelines they must follow.

The first being that they must enter <u>a valid email address</u>. This means that they must enter in a string of text that includes an @ symbol. If the user does not enter in a valid email address, the field will become red prompting them to enter in a valid email address.

The user <u>must enter in a name</u> into the name section, if they do not the same will happen a red box prompting them to fill in that field.

The <u>passwords must match</u>, and they must be more than 6 characters long. If theses requirements are not met a red box will appear at the top informing them that the passwords do not match.

If the user tries to enter in an email address of an existing member, the field will turn red and a red popup will appear informing them that this user already exists.

These popup boxes last for a specified amount of time but will appear each time an incorrect piece of information is entered.

Technologies Used

There were various dependencies and technologies used in this project. I will first discuss the backend as this was built first.

Backend Dependencies

The technologies we used for the backend were as follows:

Node.js

Runtime environment

Express

Express is a node framework. It provides features for creating web applications. Express simplifies the server setup process. Make it possible to do so in just a few lines of code. It also enables us to handle various requests such as Get, Post, Delete, Update.

MongoDB

MongoDB is a document database (NO-SQL) that stores its information in JSON format. It enabled us to populate our user information on the mongo cloud.

Mongoose

To manage the relationships between data, provide schema validation

Mongoose was used as a middleman to enable our application to talk to and update the cloud database.

Postman

Postman is an application that allowed us to hit end points that were set up in our application and make various requests such as get, post, delete without having our frontend set up.

Vs Code

This is the code editing software we used to create this application.

Dependencies

Express Validator

This was used for data validation. This was used to validate things such as if we make a request and data fields that had to be included in the request were not it would display an error. Below is some documentation on express validator.

https://express-validator.github.io/docs/

https://auth0.com/blog/express-validator-tutorial/

Bcrypt.js

Bcrypt.js is used for password encryption. When a user enters their password Bcrypt will jumble it all up. Storing plain text passwords in a database is not a good idea.

Below is a link to an example of bcrypt in action:

https://www.youtube.com/watch?v=rqAlF2K0eVU

Below is an example of how we used bcrypt to encrypt the user's passwords.

```
/ Encrypt password - using bcrypt
        // Docs:  "https://www.npmjs.com/package/bcrypt"
        // the salt is random data that is used as an additional input to a on
e-way function that hashes data
        const salt = await bcrypt.genSalt(10);

        user.password = await bcrypt.hash(password, salt);// creates a hash ba
sed off the salt on password param

        await user.save();
```

An example of bcrypt unjumbling passwords

```
// making sure password matches
        const isMatch = await bcrypt.compare(password, user.password);// will
compare encrypted and plaintext passwords
```

```
        if(!isMatch)
        {
            return res.status(400).json({ errors: [{msg: 'Invalid credentials'
}]});// 400 bad request, array
        }

        const payload = {
            user: {
                id: user.id
            }
        }
```

Config

This dependency was used to create global variable, so certain things could be accessed in different areas of the program.

```
"mongoURI": "mongodb+srv://Admin:admin@myfirstcluster-
0o9yh.mongodb.net/test?retryWrites=true&w=majority",
```

```
const config = require("config");
const db = config.get("mongoURI");
```

Gravatar

This is used to enable us to use a user's gravatar image and apply it image to their profile image. The email they are using to sign up must be associated with a gravatar account.

```
const gravatar = require('gravatar');
```

```
// Get users gravatar (profile picture) that is associated with their email
        const avatar = gravatar.url(email, {
            s: '200',
            r: 'pg',
            d: 'mm'
        })
```

JsonWebToken

This is used to pass along token validation. When a user signs into their account, an individual token will be applied to their account. This token must be valid for them to access and use the site for example post a comment etc, the token has a period of expiration and after that they will not be able to access or use their account unless they get a new token, by signing in.

Extra info: https://www.npmjs.com/package/jsonwebtoken

```
// Handles the in-between on the token and confirms or denies it

const jwt = require('jsonwebtoken');
const config = require('config');

// has access to all request and response objects
module.exports = function(req, res, next)
{
    // Get the token from the header
    const token = req.header('x-auth-token');

    // Check if no token
    if(!token)
    {
        return res.status(401).json({ msg: 'No token, authorization denied' })
; // 401 not authorized then -> error message
    }

    // verify token
    try
    {
        // decode the token
        const decoded = jwt.verify(token, config.get('jwtSecret'));

        req.user = decoded.user;
        next();
    }
    catch (error)
    {
        res.status(401).json({msg: 'Token is not valid'});
    }
}
```

```
// sign the token, pass in the payload(data), pass in the secret, then validat
e on whether a token is received
```

```
        jwt.sign(payload, config.get('jwtSecret'),
            { expiresIn: 360000}, // 360000 is a test value! CHANGE to 3600 (i
n seconds, i.e. 1 hour lifespan)
            (err, token) => {
                if(err) throw err;
                res.json({ token });
            }
        );
```

Request

This is used to make HTTP requests to other Api's . This is used to link the users GitHub repo with their account.

Nodemon

This dependency is always watching the changes made in the server, it makes it so we don't have to continuously save and keep running npm start. It just automatically looks for changes and restarts or refresh the server itself.

Nodemon information

https://nodemon.io/

Concurrently

This dependency allows us to run our backend express server and our frontend react server at the same time. It is set up so we just have to run "npm run dev" instead of npm start server and npm start. This starts the backend and front end together, so they concurrently run together. This was done by setting up the following in our server side package.json.

```
"scripts": {
    "start": "node server",
    "server": "nodemon server",
    "client": "npm start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\""
  },
```

https://www.npmjs.com/package/concurrently

Frontend Dependencies

React

This is a JavaScript library that we used to enable us to build the user interface. We set up react in a folder called client.

Axios

This dependency is a JavaScript library that enabled us to make HTTP requests from node.js across multiple domains and create global headers. Below is an example of us using axios to log a user in, in react.

```
/*
    Function to Log in a user takes in a email and password.
*/
export const login = (email,password) => async dispatch =>{
    const config = {
        //Sending data so need to send headers
        headers:{
            'Content-Type': 'application/json'
        }
    }
    //Stringifying the email, password getting it ready to be sent over the re
quest
    const body = JSON.stringify({email, password});

    try {
        /*
            waiting the post request to hit the end point
            passing in the body of stringified data and the headers
        */

        const res = await axios.post('/api/auth', body, config);
        //dispatch a succeslful registration and get the response.data (token)
        dispatch({
            type: LOGIN_SUCCESS,
            payload: res.data

        });
```

React Router Dom

This was used for making connections to various routes within the application.

Below is a list of all our routes used in this application

```
Provider store={store}>
      <Router>
        <Fragment>
          <Navbar />
          <Route exact path="/" component={Landing} />
          <section className="container">
            <Alert />
            <Switch>
              <Route exact path="/register" component={Register} />
              <Route exact path="/login" component={Login} />
              <Route exact path="/profiles" component={Profiles} />
              <Route exact path="/profile/:id" component={Profile} />
              <PrivateRoute exact path="/dashboard" component={Dashboard} />
              <PrivateRoute exact path="/create-
profile" component={CreateProfile} />
              <PrivateRoute exact path="/edit-
profile" component={EditProfile} />
              <PrivateRoute exact path="/add-
experience" component={AddExperience} />
              <PrivateRoute exact path="/add-
education" component={AddEducation} />
              <PrivateRoute exact path="/posts" component={Posts} />
              <PrivateRoute exact path="/post/:id" component={Post} />
            </Switch>
          </section>
        </Fragment>
      </Router>
    </Provider>
```

Redux

We used Redux to control our state.

Below is a result of the various methods used to set up Alerts in our application. This will be explained in further detail further down. This will display a message saying “ The passwords do not match”, and will have the colour danger

```
 setAlert('The passwords entered do not match', 'danger');
```

React-Redux

This lets our react components read data from the redux store and dispatch actions to the store to update data.

Below is an example of our Redux Store:

```
//boiler plate store file for redux
import {createStore, applyMiddleware} from 'redux';
import{composeWithDevTools} from 'redux-devtools-extension';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const initialState = {};
const middleware = [thunk];

const store = createStore(
    rootReducer,
    initialState,
    composeWithDevTools(applyMiddleware(...middleware))
);


    export default store;
```

https://medium.com/the-web-tub/managing-your-react-state-with-redux-affab72de4b1

Redux Thunk

Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object. That function receives the store's dispatch method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have completed.

https://www.npmjs.com/package/redux-thunk

Moment

Moment is a date and time library.

React- Moment

This allows us to use moment with react.

```
<p class="post-date">
                Posted on <Moment format='YYYY/MM/DD'>{date}</Moment>
            </p>
```

Proxy

On the client side package.json we used a proxy, so when making requests it would always be to the same local host. So instead of every time writing axios.get(http://localhost0000/whatever) we just have to do axios.get("/whatever").

```
 "proxy": "http://localhost:5000"
```

System Architecture

Routes

Profile routes are the end points that need to be hit when making a request to make a get, post, delete request. This will in turn make a change to or view the data that is stored in the Mongo database.

Some of the routes that we worked on are as follows. I will go over a few just to get the general gist of what we did to implement these routes, as after a few they are all very similar.

Get profile by the users id

When a user signs up to the site a random id is generated for that user. This is one example of how it is done. This request is Get request because it is just getting data from the database. It is not updating it in anyway.

```
const Profile = require('../../models/Profile');
```

```
router.get('/user/:user_id', async (req, res) => {
    try {
        //Using mongo findOne() to find and populate a single profile by their
 user_id
```

```
        const profile = await Profile.findOne({ user: req.params.user_id }).po
pulate('user', ['name', 'avatar']);
        //if the profile doesnt exist, send a status 400 along with a json mes
sage
        if (!profile) {
            return res.status(400).json({ msg: 'There was no profile found for
 this user' });
        }
        //response
        res.json(profile);

    } catch (err) {

        console.error(err.message);
        //dont need this
        //If the kind of error relates to an object id send back this response

        if (err.kind == 'ObjectId') {
            return res.status(400).json({ msg: 'There was no profile found for
 this user' });
        }

        res.status(500).send('Server Error');
    }

});
```

The end point the router needs to hit is (http:/localhost:5000/user/user_id) the full url is not needed here as we set up a proxy in our client side because of this it always looks for localhost5000 we just need to fill in the blanks.

This request uses the mongo method findOne() by looking in the parameters of the request and finds and populates the response with the data that has been applied to that particular profile.

The response is then sent back.

If no profile is found, the server will send back an error message. If the kind of error message is that the object id (does not exist) it will display an error message saying there was no profile found. The object id being the user profiles id.

Delete request to api/profile

```
const Profile = require('../../models/Profile');
const User = require('../../models/User');
const Post = require('../../models/Post');
/Delete request to api/profile
```

```
//Delete profile, user and posts

router.delete('/', auth, async (req, res) => {
    try {
        //Remove user posts
        await Post.deleteMany({ user: req.user.id });
        //Remove a profile by using the mongo command findOneAndRemove
        await Profile.findOneAndRemove({ user: req.user.id });
        //Remove the user
        await User.findOneAndRemove({ _id: req.user.id });

        res.json({ msg: "User has been deleted " });

    } catch (err) {

        console.error(err.message);

        res.status(500).send('Server Error');
    }

});
```

Above is an example of a delete request. When this request is implemented it will delete the user’s profile, their posts, and the user themself. Similarly, to the above request it is using mongo methods to search for these documents and delete them. In the case of the posts it calls the delete many methods, it finds this as every post has the users id assigned to it in the database. The other mongo methods here used are findOneAndRemove which will remove the profile and the user by their user_id.

It then sends back a response message “User has been deleted”.

Otherwise it sends back a server-side error message.

Post request to api/posts

```
// Post request
// Create a post
router.post('/',
    [   //text is required in post
        auth,
        [
            check('text', 'You are required to enter text into a comment').not
().isEmpty()

        ]
    ],
    async (req, res) => {
        //error checking, takes in request
        //validation result method provided through express validator
        //it extracts the validation errors from a request and makes them avai
lable in the error var
        const errors = validationResult(req);
        //if there are errors send back status 400 with the contents of the er
ror
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }

        try {
            /*
                User logged in, users id can be found by req.user.id
                don't send the password
            */
            const user = await User.findById(req.user.id).select('-password');
            //new post
            const newPost = new Post({
                //text is in the req.body.text
                text: req.body.text,
                //name= users name
                name: user.name,
                //users avater
                avatar: user.avatar,
                //users id
                user: req.user.id
            });
            //save post
            const post = await newPost.save();
            //send back post in response
            res.json(post);

        } catch (error) {

            console.error(err.message);
            res.status(500).send('Server error')

        }



    });
```

Above is a post request, this request is used to create a post. The first thing it does is look for authentication. It checks to make sure that the user has entered in some text. It will not let the user post an empty post. If the user tries to enter in an empty post it will display the message “You are required to enter text". It then uses express validator to check and see that errors do not exist. If there is an error, it will catch it and return a status 400 message.

If no errors exist it will find the user by their id. It will send back all the relevant message minus their password.

It then creates a new instance of the post model passing the contents of the post into the request, along with their user_id, name, and their picture. It then saves the post and returns the newly created post in the response.

If the try fails it will catch the error and send back a status 500 (Server error)

Models

Below is the postSchema this schema is used to populate the post document in the mongo database.

It is implemented by brining in mongoose. This in turn connects to the database and saves the schema to the database. It takes the user id from the user schema which it references. The text is the text that is entered into the post or comment, and sets it as required. It also takes the username and their picture if any.

The likes are an array stored by the user id(the person who liked the post)

Comments are also an array which stores the user id of the person who wrote it and the text they entered the comment along with their name and their picture and the date the comment was posted. The date the initial comment was posted is also stored as well.

The other model I created is like this one, both are implemented in a similar manner.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
```

```
const postSchema = new Schema({
    //connects a user to a post
    //references users schema
    user: {
        type: Schema.Types.ObjectId,
        ref: 'user'
    },
    //user required to enter text into their post
    text: {
        type: String,
        required: true,
    },
    //user name
    name: {
        type: String
    },

    avatar: {
        type: String
    },
    //array of likes
    likes: [
        {   //array of user object that contains the id's
            user: {
                type: Schema.Types.ObjectId,
                ref: 'user'
            }

//array of comments and contents
    comments: [
        {
            //user data
            user: {
                type: Schema.Types.ObjectId,
                ref: 'user'
            },
            //comment entered
            text: {
                type: String,
                required: true,

            },
            //user name
            name: {
                type: String
            },
            //user avatar
            avatar: {
                type: String
            },
            //date the comment was posted
            date: {
                type: Date,
                default: Date.now
            }
        }
    ],
    //date of the post
    date: {
        type: Date,
        default: Date.now
}
```

React Router

For our routing we used the React router library. For react router to work everything must be put into a Router container. We then simply just put the component we want to load from each route between each route tag. This sets up the path of the page and handles all the routeing. If we want to redirect to a page or link to a page, we then simply put the relevant tag and route.

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom'
```

```
          <section className="container">
            <Alert />
            <Switch>
              <Route exact path="/register" component={Register} />
              <Route exact path="/login" component={Login} />
              <Route exact path="/profiles" component={Profiles} />
```

Instead of using something like a href to go to a page we used Link also from react router. This works in the same way as href. If the user clicks on this link it will bring them to the page that displays the profiles made on the site.

```
<Link to="/profiles">
          Members
        </Link>
```

To redirect our pages i.e. if after the user has filled in their required information during the signup process, they must be redirected to a new page to continue their sign-up process we used redirect. This does exactly what is say on the tin and redirects the user to that specific page.

```
 if(isAuthenticated){
        return <Redirect to = '/dashboard' />
    }
```

Using Redux

Redux is a state manager. Components like profiles, posts are app level state that need to be accessed from anywhere in the program.

Data gets put into a redux store. In the context of the profiles when an action is called it gets the data and puts it in the store. We can use any component to call an action to do something that we need it to. If for example the user wants to update or delete their profile a request to the server will be made, do the needed action on the server or database a response will be sent and the U.I will need to be updated to reflect these changes. These actions happen through a reducer which is a function that takes in an action. When an action is dispatched to the reducer it then decides what action to take i.e. how to handle the state and how to pass it down to the U.I components. Any components that use that state will then be updated.

An action is dispatched to a reducer then the reducer decides what to do with that state. It will then in turn pass down the state to any components that have that state.

Below is an example of using redux.

Types

```
export const GET_POST = 'GET_POST';
export const POST_ERROR = 'POST_ERROR';
```

The types are fist created in the types folder and then exported to the relevant actions.

Actions

```
import {GET_POST, POST_ERROR} from './types';
```

```
//GET POSTS FUNCTION
export const getPosts = () => async dispatch => {
    //try catch to make request
    try {
        const res = await axios.get('/api/posts');
        dispatch({
            type: GET_POST,
            payload: res.data
        });
    } catch (err) {
        dispatch({
            type: POST_ERROR,
            payload: { msg: err.response.statusText, status: err.response.stat
us }
        });
    }
};
```

The above function makes a request to get all the posts in the database. It dispatches a type of GET_POST to the reducer and sends the data in the payload. If there is an error, it dispatches a POST_ERROR type and sends an error message in the payload.

Reducers

```
import {
    GET_POST,
    POST_ERROR} from '../actions/types';

//object containing state
const initialState = {
posts:[],
post: null,
loading: true,
error:{}
}
```

The initial state is just an empty object containing the empty types.

As you can see here below a reducer is a function that takes in any state that has to do with that reducer. This function takes in the state and an action. The action contains a type which is the action type on and a payload which contains the data.

It then dispatches an action which gets dispatched from an actions file. Inside the reducer there is a switch for each type it then specifies what to do with that specific type. There is a reducer for every component in this project, profiles, posts, registration, alerts etc.

```
export default function(state=initialState,action){
    //pulling type and palоad from the action
    const {type, payload} = action;

    switch(type) {
        //returning the state and the array of posts in the payload
        case GET_POST:
            return{
                ...state,
                posts: payload,
                loading: false
            };
        case ADD_POST:
            return {
                ...state,
                //current array with state.posts and add new post in payload
                //will return post down to any component that uses post compon
ent in their state
                //putting payload first will put the comment at the top
                posts: [payload,...state.posts],
                loading: false
            };
```

The type is what is being evaluated in the switch method. The types are constants which come from src/actions/types

```
export const SET_ALERT = 'SET_ALERT';
export const REMOVE_ALERT = 'REMOVE_ALERT';
```

Components

The state is then passed to the components where it can be used to control the app level state. It is done through connect. Connect allows the component to connect to redux. This comes with the react redux package. When using connect it must be exported along with the name component must be exported.

When being exported connect takes in 2 parameters. The first is any state that you want to map and the second is an object that contains any actions that need to be used.

```
export default connect(mapStateToProps,{getPosts})(Posts);
```

```
import {addPost} from '../../actions/post';
const PostForm = ({addPost}) => {
    //Setting string to be entered in as text
    const [text,setText] = useState('');
    return (
        <div class="post-form">
        <div class="bg-primary p">
          <h3>Leave A Comment</h3>
        </div>
        <form class="form my-1" onSubmit={e => {
            e.preventDefault();
            addPost({text});
            setText('');

        }}>
          <textarea
            name="text"
            cols="30"
            rows="5"
            placeholder="Write your post!!!"
            value = {text}
            onChange = {e => setText(e.target.value)}
            required
          ></textarea>
          <input type="submit" class="btn btn-dark my-1" value="Submit" />
        </form>
      </div>
    )
}

PostForm.propTypes = {
addPost: PropTypes.func.isRequired
}

export default connect (null, {addPost})(PostForm)
```

Dispatch keyword:

https://stackoverflow.com/questions/42871136/dispatch-function-in-react-redux

Redux Store

A store is a JavaScript object that allows components to share state. A store is like a database.

The variable initial state is inside all the reducers but is initialised here.

Redux-Thunk is a middleware which lets you call action creators that return a function instead of action object. This function receives the store's dispatch method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have completed. This is initialised below.

Redux Thunk:

https://daveceddia.com/what-is-a-thunk/

The store const is assigned to the created store, which was imported above. It takes in the root reducer. There are multiple reducers post profile etc these are all combined using the root reducer. It then takes in the middleware.

Store

```
import {createStore, applyMiddleware} from 'redux';
import{composeWithDevTools} from 'redux-devtools-extension';
import thunk from 'redux-thunk';
import rootReducer from './reducers';
```

```
import rootReducer from './reducers';

const initialState = {};
const middleware = [thunk];

const store = createStore(
    rootReducer,
    initialState,
    composeWithDevTools(applyMiddleware(...middleware))
);


    export default store;
```

Root reducer (index.js)

```
import {combineReducers } from 'redux';
import alert from './alert';
import auth from './auth';
import profile from './profile';
import post from './post';

//creates object containing any reducers created
export default combineReducers({
    alert,
    auth,
    profile,
    post
});
```

All the reducers are imported into the index.js (Root reducer) then they are exported using combine reducers.

Provider

To be able to use the store is must be imported into app.js. It needs to have a Provider which comes from the react redux package. This provider connects react and redux together. This is done by wrapping everything inside this provider container. The store is then passed into the provider. This is how all the created components are able to access the app level state.

```
import { Provider } from 'react-redux';
import store from './store';

<Provider store={store}>
      <Router>
        <Fragment>
          <Navbar />
          <Route exact path="/" component={Landing} />
          <section className="container">
            <Alert />
            <Switch>
              <Route exact path="/register" component={Register} />
              <Route exact path="/login" component={Login} />
              <Route exact path="/profiles" component={Profiles} />
              <Route exact path="/profile/:id" component={Profile} />
              <PrivateRoute exact path="/dashboard" component={Dashboard} />
              <PrivateRoute exact path="/create-
profile" component={CreateProfile} />
              <PrivateRoute exact path="/edit-
profile" component={EditProfile} />
              <PrivateRoute exact path="/add-
experience" component={AddExperience} />
              <PrivateRoute exact path="/add-
education" component={AddEducation} />
              <PrivateRoute exact path="/posts" component={Posts} />
              <PrivateRoute exact path="/post/:id" component={Post} />
            </Switch>
          </section>
        </Fragment>
      </Router>
    </Provider>
```

Site Registration

This works by using form data which is an object with all the field values such as name, email etc this is the state. There is also a function that is used to update the state this is called setFormData in the image below. These are both pulled from the useState hook. The useState hook allows you to have state variables in functional components.

I will also use this as a way to demonstrate how the state is controlled in this application as it is all done similarly.

The initial state is the name, email, password, and password2. Password 2 being the password verification i.e. the user must confirm their password and they must match.

```
const [formData, setFormData] = useState({
        //initial state
        name: '',
        email: '',
        password: '',
        password2: ''
    });
```

The name, email, password, password2 are then pulled from the formData state. Each value is associated with a value on the registration form so when the user enters a value into that field it then updates that state with using setFormData entered value.

```
<form className="form" onSubmit={e => onSubmit(e)}>
            <div className="form-group">
                <input type="text" placeholder="Name" name="name" value={name}
 onChange={e => onChange(e)} required />
            </div>
            <div className="form-group">
                <input type="email" placeholder="Email Address" name="email" v
alue={email} onChange={e => onChange(e)} required />
                <small className="form-text">
```

SetFormData is the object changing the initial state of the name, email, password, password2. In the code below first the setFormData is making a copy of what's in the formData using the spread operator, it then changes the value of name attribute to the value of the input (e. target.name).

e. target.name is the key or value that has been entered into that field. Without using this it would just change the actual name of the value. This allows the onChange to be used for every field in the form. Validation is also used here as well using the required keyword.

```
 const onChange = e => setFormData({ ...formData, [e.target.name]: e.target.va
lue });
```

The registration form is inside of a **form** tag when onSubmit is called when the user clicks the register button the onSubmit method is called.

The default action is prevented, the passwords entered are then checked against each other, if they done match an error is displayed using redux. If everything has been entered correctly, they will be registered using a redux action which makes the request to the backend.

```
  <form className="form" onSubmit={e => onSubmit(e)}>
```

```
const onSubmit = async e => {
        e.preventDefault();
 if (password !== password2) {
 setAlert('The passwords entered do not match', 'danger');
  } else {
   register({name, email, password});
        }
    }
```

The error message like all of the other error messages are displayed as follows.

```
import {connect} from 'react-redux';
//importing set alert action
import {setAlert} from '../../actions/alert';
import {register} from '../../actions/auth';
import PropTypes from 'prop-types';
```

The correct imports are brought in.

```
const Register = ({setAlert, register, isAuthenticated}) => {
    /*
```

setAlert along with other components are pulled in from the props.

```
//setting up proptypes for Regsiter
Register.propTypes = {
    setAlert: PropTypes.func.isRequired,
    register: PropTypes.func.isRequired,
    isAuthenticated: PropTypes.bool
}
const mapStateToProps = state => ({
    //Need to check if the user is authenticated (reducers/auth)
    isAuthenticated:state.auth.isAuthenticated
})

export default connect(mapStateToProps, {setAlert, register})(Register);
```

Connect is then exported along with the state that you want to map in this case mapStateToProps and the object that contains the actions that you need. Which is the setAlert and the register.

The prop types are also declared which are assigned to the proptype objects.

The objects are Proptypes, then what type of proptype it is (bool, int, function) and then whether it is required or not.

```
setAlert('The passwords entered do not match', 'danger');
```

This is called from the setAlert type below.

```
export const setAlert = (msg, alertType, timeOut = 5000) => dispatch => {

    //generating a unique id from uuid
    const id = uuid.v4();
    dispatch({
        type: SET_ALERT,
        payload:{msg, alertType, id}
    });

```

Here It then takes in the alert message `'The passwords entered do not match'`

It then dispatches that alert with the above message, alertType (Danger) and the randomly generated id.

It then goes to the setAlert reducer where the type SET_ALERT the message, alertType and id are added to the array.

```
export default function (state = initialState, action) {
    //pull out type and payload from action
    const { type, payload } = action;

    switch (type) {
        case SET_ALERT:
            return [...state, payload];
```

The alert component is getting the state passed from the reducer

Alert Component (Alert.js)

Pulling the alerts from mapStateToProps. Making sure it is not equal to null and if the array of alerts is greater than zero. Map through the array of alerts with a key of the alert.id(the randomly generated uuid) pulling out the alert message and the alert type (colour).

```
const Alert = ({alerts}) => alerts !== null && alerts.length > 0 && alerts.map
(alert => (
    <div key={alert.id} className={`alert alert-${alert.alertType}`}>
        {alert.msg}
    </div>

));
```

Setting alerts up as a proptype which is an array of alerts.

```
Alert.propTypes = {
    alerts: PropTypes.array.isRequired

}
```

Mapping a redux state to a prop so that it has access to it through connect. Which is the array of alerts. Then pulling the state from the root reducer “Alert”. This makes props.alerts available to use.

```
const mapStateToProps = state => ({
    alerts: state.alert
})
//connect needs to be used when interacting with function or getting state whe
n using redux
export default connect(mapStateToProps)(Alert);
```

```
import Alert from './components/layout/Alert';
```

It is then imported into app.js and embedded

```
  <Alert />
```

How the alert is removed

In actions/alert.js

A method called setTimeOut will dispatch and object that contains the type of remove alert with a payload of id. It also has a timeout variable which is assigned to 5000 milliseconds or 5 seconds.

```
export const setAlert = (msg, alertType, timeOut = 5000) => dispatch => {
```

```
setTimeout(()=> dispatch({type: REMOVE_ALERT, payload: id}),timeOut)
```

setTimeOut:

https://www.w3schools.com/jsref/met_win_settimeout.asp

Then in the alert reducer it will filter out everything except for the alert with that id.

```
/*
            Remove a specific alert by it's id
            Return the array state and filter through the alerts
            Then for each alert check and see if the alert.id is not equal to
the payload
        */


case REMOVE_ALERT:
            return state.filter(alert => alert.id !== payload);
```

All the state works like this example above in the registration there is also register and isAuthenicated, the login, the profiles, posts etc all work very similar to the method mentioned above. Please see the code for more in-depth explanation of how we applied the above methods to our other components, stores, reducers, and actions.

Token

When the user has logged in or registered, they receive a token into their local storage. This token is sent to the backend for validation and then the user is loaded. This happens every time the main app component is loaded.

The route that is hit to authenticate the user is in routes/api/auth

What it does is, it checks for a user with the user id that has been sent. This is validated by the middle ware.

```
// GET api/auth - Test route
// by adding auth as a param, we protect our route using the middleware
router.get('/', auth, async(req, res) => {
    try
    {
        const user = await await User.findById(req.user.id).select('-
password'); // exclude password from return
        res.json(user);
    }
    catch (error)
    {
        console.error(err.message);
        res.status(500).send('Server error');
    }
});
```

In the utils folder there is a file called set auth token

If the token exists it sends it in the global header to the backend for validation. This will get sent with every request. If there is no token it gets deleted.

```
import axios from 'axios';


const setAuthToken = token => {
    //Checks to see if there is a token in local storage
    if (token) {
        //if there is send this global header and assign it to the token
        axios.defaults.headers.common['x-auth-token'] = token;
    } else {
        //if what is passed in is not a token, delete it from the global heade
rs
        delete axios.defaults.headers.common['x-auth-token'];
    }
}

export default setAuthToken;
```

The types that are used for this validation.

They are then imported into the actions/auth file.

actions/auth file.

```
export const AUTH_ERROR = 'AUTH_ERROR';
export const USER_LOADED = 'USER_LOADED';
```

First it checks to see if there is a token in local storage

If there is a token it will use the set auth token function and pass it will pass that token into local storage. This sets the header with the token of there is a token in local storage.

If there is a token a request will be made to the end point that needs to be hit. If everything goes to plan it will dispatch the USER_LOADED type and the res.data that is the user's data.

I it's unsuccessful it will dispatch the auth error token.

```
//load user
export const loadUser = () => async dispatch => {
    if(localStorage.token){

        setAuthToken(localStorage.token);
    }

    try {

        const res = await axios.get('/api/auth');

        dispatch({
            type: USER_LOADED,
            payload: res.data
        })

    } catch (error) {
        dispatch({
            type:AUTH_ERROR
        })

    }

}
```

Reducers/auth

Below is what happening if either types are dispatched

Everything will be loaded except for the user password

```
  //functionality for USER_LOADED type
        case USER_LOADED:
            /*
                make a copy of the state, set isAuth to true, loading false,
                and return the user in the paylaod as it includes the user dat
a
            */
            return{
                ...state,
                isAuthenticated: true,
                loading:false,
                user: payload
            }
        //testing case of REGISTER_SUCCESS
```

```
 case AUTH_ERROR:
                //remove items from local storage
            localStorage.removeItem('token');
            return{
                /*
                using spread operator to return whats currently in the state
                Set the token to null, isAuthenticated to false, and loading t
o false
                */
```

```
                 ...state,
                token: null,
                isAuthenticated:false,
                loading:false
            }
```

The loadUser action is then used in app.js where it can be used throughout the application. Using the useEffect hook its accesses the store then loaduser is dispatched.

```
/*
    useEffect Hook can be thought of as componentDidMount, componentDidUpdate,
 and componentWillUnmount combined.
    The useEffect will run in an infinite loop, the the [] is used at the end
to make sure that it is only run once, when it's mounted.
    Like a component did mount
  */
  useEffect(() => {
    store.dispatch(loadUser());

  }, []);
```

https://reactjs.org/docs/hooks-effect.html

Conclusion

When we first started this project, we did it with the intention of expanding our knowledge of how M.E.R.N applications work. I believe the objectives to this project were achieved to a high standard and implementing the new things we learned have been both rewarding and frustrating (at times) to learn how to use. Although the result and the process far out ways any troubles that we encountered while building this application.

The site registers, and signs in a user. If the user already exists it displays this to the screen. If the passwords are to short it informs the user, if the sign in credentials are not correct it displays this to the screen.

The user can create their profile and it will show the information they entered in a formatted way. The user can write, comment on and like posts. The user's profiles on the platform can be viewed. They can edit, remove their information or profiles and comments. The database stores all the information successfully, the routes all hit their end points.

Learning perspective

From a learning perspective this project ticked all the boxes for creating responsive M.E.R.N applications and the technologies that it includes.

Our understanding of C.R.U.D has also been expanded from building routes and making quite a lot of requests to out backend.

We have learned lots of new approaches, technologies, and dependencies for building these kinds of application and we have built upon and improved our prior knowledge of building backends and implementing the react front end.

Backend

From the point of view of building the backend we built upon our knowledge of how the backend works and learned new ways of how to get it done in a different sometimes more easy way. Below are a few of the major new things we learned about.

For example:

nodemon to always listen for changes made to the backend so it constantly updates when a change is made.

Config using the config dependency to easily allow us to take constants such as the mongo uri and call them from any part of the application by just simply using config.

Postman we used this tool to hit all our end point before we had the front end built. This proved an invaluable tool for debugging and making sure all our routes were working correctly.

JsonWebToken was used to send the user a token when they logged in and their token is used to do various things throughout the application. This tool has opened my eyes about how web authentication works along with bcrypt which we used to encrypt our passwords, so they were not stored in plain text in the database, which of course would be a security issue. Using these tools, we were able to create a middle ware which was used to authenticate a user.

Request we used this to be able to make a request to the Github api to fetch the user's repository data.

Models we learned new way of manipulating database schemas to create things like likes and comments by using arrays again building upon our prior knowledge of them.

Routes creating various kind of routes and learning how they work really taught us how they work and how they can be used to access a database and hit endpoints. Making different and new types of requests such as a put request, post, delete, update. Using authentication with these routes to enable us to validate what the user is doing is correct.

React Frontend

The front end is what posed the biggest challenge when building this application.

Learning completely new topics such as **Redux**, was easily the hardest thing to get our head around. After figuring it out through taking the time to understand exactly what it is doing it really helped us not only understand the topic of redux but how state in react is handled.

Everything from creating the **redux store**, the middleware using **redux-thunk, creating types, creating reducers**, **alerts** and then making it all work with the numerous **actions** and then in turn to work with the various **components** to control the state of the application. We used these methods numerous times repeatedly which enabled us to get a firm grasp and understanding of them and is what the front end is built around.

Hooks

A few hooks were used throughout this program such as the useState hook and the useEffect hook. Learning about these hooks and what they do really gave us a bigger understanding of the features and power that react has to offer.

Concurrently using this dependency made our lives so much easier when working with the frontend and back end as it ran them both at the same time.

Using a proxy using a proxy really helped us when making requests as we didn't have to put in the full url every time we wanted to make a request just the end point, we wanted to hit

Using Github and teamwork

During the creation of this project we were with collaboration on github. Both I and Cuan were in constant contact with each other discussing pushes and fixing errors. It was a great experience having a team member to work with on a project like this. I imagine it reflects what it is like out in the world working as a software developer. We had a few issues with github and our pushes where the first part of our commits were deleted but we learned from that experience and what we did wrong in that instance to make sure that it never happened again. After doing this project I feel like I have improved my overall understanding of github and how it works. Especially when working in teams, which will be invaluable to the both of us when out in the working world as will this project, I believe it will really stand to us both in terms of our studies and when out working in the industry.

Small bugs in the application

In the profile page sometimes, it pulls other users' information onto that person's profile.

Sometimes the page must be re-loaded in order to view the saved data of that page.

References

Traversey media this YouTube channel taught us a lot about making the front and backend as well as redux

https://www.youtube.com/channel/UC29ju8bIPH5as8OGnQzwJyA

https://www.youtube.com/watch?v=PBTYxXADG_k

Redux

https://www.youtube.com/watch?v=iI5h4-pChho

https://www.youtube.com/watch?v=93p3LxR9xfM

Document on redux https://redux.js.org/

Hooks https://www.youtube.com/watch?v=mxK8b99iJTg

Document on hooks https://reactjs.org/docs/hooks-effect.html

private routing - https://www.youtube.com/watch?v=Y0-qdp-XBJg

Link to concurrently information: https://www.npmjs.com/package/concurrently

useEffect hook - https://reactjs.org/docs/hooks-effect.html

Fragments - https://reactjs.org/docs/fragments.html

Destructuring - https://www.youtube.com/watch?v=5_PdMS9CLLI - This allows us to unpack values from arrays or properties from objects into distinc objects

improves our codes readability connect - https://react-redux.js.org/api/connect

mapStateToProps - https://react-redux.js.org/using-react-redux/connect-mapstate

PropTypes - https://reactjs.org/docs/typechecking-with-proptypes.html

History - https://scotch.io/courses/using-react-router-4/using-history

withRouter - https://github.com/ReactTraining/react-router/blob/master/packages/react-router/docs/api/withRouter.md

useState hook - https://reactjs.org/docs/hooks-state.html

moment date formatting - https://momentjs.com

conditional rendering in react - https://reactjs.org/docs/conditional-rendering.html