First we work on the **registering** and **logging in**. then we create a **middleware** for **authentication**:

const jwt = require('jsonwebtoken')

const config = require('config')

module.exports = function (req, res, next) {

    //Get 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' })

    }

    try {

        const decoded = jwt.verify(token, config.get('jwtSecret'))

        console.log('decoded is', decoded)

        /\*

        At this stage, decoded contains the payload which is

        const payload = {

            user: {

                id: user.id

            }

        }

        \*/

        //Then we assign that payload to the request object

        req.user = decoded.user

        next()

    } catch (error) {

        res.status(401).json({ msg: 'Token invalid' })

    }

}

For the 2nd argument, sometimes we use **express-validator** or **auth middleware**. If we need to use both, just put them in square brackets:

router.post('/', [auth, [

    check('name', 'Name is required').not().isEmpty()

]], async (req, res) => {

    const errors = validationResult(req)

    if (!errors.isEmpty()) {

        return res.status(400).json({ errors: errors.array() })

    }

So the backend im quite ok with, just need more practice. We move on to the frontend

In the frontend, brad says this project is unlike the github finder, where he explains things in detail. This project, he will do more like if hes doing it on his own. He starts off with……**context**. Interesting to see that the very first thing he works on in the frontend, is **context**. He said he wants to have a single source of truth before working on any react component.

After that he begins with hard coded data in the frontend. **Nothing to do with backend yet.** Just use hardcoded data in frontend.

*Notice that all the axios requests are located in the* ***action creators****.*

Now we connect the frontend to the backend. We first create a new **context** for authentication, then for alert. Finally, we start interacting with our backend. We will make a **request** to the backend, **store** the data in the database, send a **token** back and store it in **localStorage**

Remember, whenever we are making a **post** request, we are obviously sending data. Therefore we need to set the content type like so:

const register = async formData => {

        const config = {

            headers: {

                'Content-Type': 'application/json'

            }

        }

        try {

            const res = await axios.post('/api/users', formData, config)

            dispatch({

                type: REGISTER\_SUCCESS,

                payload: res.data //res.data will be the token

            })

        } catch (error) {

            dispatch({

                type: REGISTER\_FAIL,

                payload: error.response.data.msg

            })

        }

    }

Next, after successfully registering, we need to load that user into our user **state**.so far we are only getting the token back, never the user itself. So we do that now. we also obviously need a token to access protected routes. To do that we create a simple separate file to store the token in **axios’ global headers**:

//what this file does is: check if a token is passed in. if yes, set it to th

e global header. otherwise delete it from global header

import axios from 'axios'

const setAuthToken = token => {

    if(token) {

        axios.defaults.headers.common['x-auth-token'] = token

    } else {

        delete axios.defaults.headers.common['x-auth-token']

    }

}

export default setAuthToken

and implement it – basically if the localStorage has a token, then set it to be global:

const loadUser = async () => {

        //Load token into global headers

        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

            })

        }

    }

*(at this stage, I am beginning to think……how can I possibly do all this without* ***context****…..lol too much state to manage)*

This project is done. Overall, I just need more practice. This particular course, the **contact keeper**, is a great reference for when im stuck.