In the early stages, brad shows us the basic setup/installation of the backend and the database, mongodb atlas. Although different libraries are used, they all do the same thing and I already know the concept thanks to helsinki fullstack mooc.

**1.we create a route for registering users.**

**2.we create the auth middleware**

Now we get a **token** back, which has the **payload** (info about the user). Now we need to be able to send this **token** back to authenticate and access protected routes. We build a **middleware** for this

const jwt = require('jsonwebtoken')

const config = require('config')

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

    //Get token from header

    const token = req.header('x-auth-token') //x-auth-token can be any name

    //Check if no token

    if(!token) {

        return res.status(401).send({ message: 'No token, authorization denied'})

    }

    try {

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

        req.user = decoded.user

        next()

    } catch(err) {

        res.status(401).send({ message: 'Token invalid'})

    }

}

And in **auth.js** route we simply add the middleware as a 2nd argument:

const router = require('express').Router()

const auth = require('../../middleware/auth')

router.get('/', auth, (req, res) => {

    res.send('Auth route')

})

module.exports = router

**the above 2 steps are enough to make a route protected.** For example now in postman I cannot make a get request to http://localhost:5000/api/auth because there is a middleware, which acts as a barrier.

So from the 2nd last diagram above, **decoded** will print the same as the **payload**. In my case, **payload in user.js route is:**

const payload = {

        user: {

            id: user.id

        }

    }

        jwt.sign(payload, config.get('jwtSecret'), {expiresIn: 360000}, (err, token) => {

            if(err) throw err;

            res.send({

                token

            })

        })

And **decoded’s value** is:

{

    user: { id: '5e94f830e8b3230d1075b408' },

    iat: 1586821168,

    exp: 1587181168

  }

Notice the user object with the id? The same. **Basically once a token is verified, it will be equal to the payload**

**3. after making sure authentication is done and we get token back for both logging in and signing up, we work on profile route**

Basically each time we wanna work on a **resource**, the steps are:

1. create a **schema,**
2. create the route

When the backend is done, we move to the **frontend** and **redux**

**1. We first setup a store.**

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

then of course we want all components to have access so wrap **app.js** in the **provider:**

import {Provider} from 'react-redux'

import store from './store'

const App = () => {

    return (

        <Provider store={store}>

            <Fragment>

                <Navbar />

                <Route exact path='/' component={Landing} />

            </Fragment>

        </Provider>

    )

}

**2. work on the actions and reducers. Below is just the root reducer, where all reducers are combined**

import {combineReducers} from 'redux'

import alert from './alert'

import auth from './auth'

export default combineReducers({

    alert,

    auth

})

Remember that each reducer will have its own state.

and for **action**, a little different from **context** because the **actions creator has to return a function first**, then, **dispatch (notice the double arrow function)**:

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

        })

    }

}

**3. if a component needs access to the state, mapStateToProps:**

const mapStateToProps = state => ({

    //state.alert is from the root reducer.

    alerts: state.alert

})

export default connect(mapStateToProps)(Alert)

“alerts” now becomes available through normal props that can be destructured. Also remember the **2nd argument of connect is an action (put it in curly brackets)**:

export default connect(null, {setAlert, register})(Register)

**so don’t overcomplicate it , it really is quite simple now that I understand context. Basically create a store (for the state), actions, reducers. Precisely like context. Then if a component needs access to the state, use connect. 1st argument for connect is mapStateToProps, 2nd argument is all the actions you wanna use in that component**