**BACK END SET UP**

**Set up Database**

Using Mongo via mLab

* Create database.
* Create database user

**Install Dependencies**

In parent directory: **npm init** // creates package.json;

* Make entry point – **server.js** // makes it clearer that this is the server backend

Install dependencies:

* express // backend routes
* mongoose // connect to mongo
* passport // authentication
* passport-jwt // to use jsonwebtoken as auth system
* jsonwebtoken // auth system
* body-parser // helper library for url route
* bcryptjs // generate hash for the webtoken
* validator // assist with validation

**npm i express mongoose passport passport-jwt jsonwebtoken body-parser bcryptjs validator**

Install nodemon: **npm i -D nodemon** // monitors the node application and updates it.

**Server.js**

Create server.js.

Final server.js will be something like the below with the initial code highlighted:

const express = require("express");

const mongoose = require("mongoose");

const bodyParser = require("body-parser");

const passport = require("passport");

const path = require("path");

// pointers to route folders

const users = require("./routes/api/users.js");

//const profile = require("./routes/api/profile");

//const posts = require("./routes/api/posts");

// initialize app

const app = express();

// middleware for body-parser.

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

// DB config

const db = require("./config/keys").mongoURI;

// connect to DB

mongoose

.connect(db)

.then(() => console.log("MongoDB connected"))

.catch(err => console.log(err));

// Passport middleware.

app.use(passport.initialize());

// Passport Config. Passport needs a strategy for the relevant authorisation mode. ie a strategy for jwt or whatever the auth kind is used.

require("./config/passport")(passport);

// establish initial test route - will later put routes into separate files

//app.get("/", (req, res) => res.send("Hello!!!"));

// set routes

// points url to relevant folder ie this is the users end point.

app.use("/api/users", users);

//app.use("/api/profile", profile);

//app.use("/api/posts", posts);

// Server static assets if in production.

if (process.env.NODE\_ENV === "production") {

// Set static folder.

app.use(express.static("client/build"));

app.get("\*", (req, res) => {

res.sendFile(path.resolve(\_\_dirname, "client", "build", "index.html"));

});

}

const port = process.env.PORT || 5000;

app.listen(port, () => console.log(`server running on port ${port}`));

to start server: **node server**

**Nodemon**

In package.json, create script to run nodemon:

“start”: “node server.js”,

“server”: “nodemon server.js”

Can now do **npm run server**

**Gravatar**

If you want to use gravatar: **npm install gravatar**

**Git**

**Create .gitignore**: // tells git what to ignore

**/node\_modules  
package.json**

Initiliase git repository: **git init**

Add files – **git add .**

Commit – **git commit -am “initial commit”** // make regular commits after this. Later will push project to Heroku using git.

**Connect to Database**

Store connection sting in a file: **create** new folder/file: **config/keys.js**

In server.js:

* bring in mongoose: **const mongoose = require(‘mongoose’);**
* configure the db connection:
  + bring in the connection path: **const db = require(‘./config/keys).mongoURI;**
  + connect via mongoose:   
    **mongoose  
    .connect(db) // returns a promise  
    .then(() => console.log(“MongoDB connected”));  
    .catch(err => console.log(err));**

**Create Initial Routes**

Will use express to create routes(end points) for each resource – each proposed view in the front end eg users, customers, orders.

Each route will have its own file in a folder/subfolder - **create routes/api.**

**Create users.js** and any other proposed routes in routes/api // will hold all the end points to access the users collection.

Create the route in the file:

* Use express – **const express = require(‘express’);\**
* Bring in express’s router = **const router = express.Router();**
* Instead of app.get as you have in server use router.get to set the path, and serve the json that is returned from the database via res.json: a simple example is: **router.get("/test", (req, res) => res.json({ msg: "Users Works" }));**

In server.js:

* bring in the route files: **const users = require("./routes/api/users.js")**;
* use the routes via app.use()..**app.use("/api/users", users);** // sets path to the file that has been brought ie to the constant to which the file has been assigned

**Authentication**

Create Model with schema for Users.

**Create folder models**

**Create user.js** // will hold schema for Users collection.

The model requires mongoose and sets a constant to a mongoose schema object.

A constant is then set to a NameSchema as a new Schema with the structure of

({ fieldname: {type: String, required: true} })

module.exports is set to a constant which is the name of the model and the constant is set to a mongoose model which sets out the name of the collection in the mongodb, and the name of the schema for the model

**const mongoose = require(“mongoose”);**

**const Schema = mongoose.Schema;**

**const NameSchema = new Schema({ fieldname: {type: String, required: true} })**

**});**

**module.exports = Name = mongoose.model(‘name’, NameSchema);**

users is the one collection/table that is essential for all databases. It will hold the email/login and password for all users – admin and customers.

Install and//or open Postman // needed to test calls to database

**Registration**

* Load the user model - **const User = require("../../models/User")**;
* Use Router.post – **router.post(‘/register’, (req, res => {  
  });**
* In server.js:
  + In order to use req.body (next step), body-parser must be brought at file head – **const bodyParser = require(‘body-parser’),** and the body parser middleware setup **app.use(bodyParser.urlencoded({ extended: false }));  
    app.use(bodyParser.json());**
* Check if registration email is already in the database:
  + Mongoose has query functions that attach to a Model that can be used to act on the data in the request – **User.findOne({email: req.body.email});** // the request comes from the frontend which will be set with a login form that has a field called emai
  + Can then use a promise attached to function to act on the response, which will be the user.
  + If the user is found via the email you return an error status and an error
  + If the user is not found, create an object to hold the user data as a new instance of the model and pass in the data as an object within the new model eg const newUser = new User({data}):
  + If there is an avatar field, fetch it using the gravatar module and put avatar data into a constant which can be added to the new data object:

**const avatar = gravatar.url(req.body.email, {  
 s: "200", // Size  
 r: "pg", // Rating  
 d: "mm" // Default  
 });**

* + Must encrypt the password using bcrypt:
    - Must bring it in: **const bcrypt = require(‘bcryptjs’);**
    - Use bcrypt to generate a hash of the password, after checking that a password has been entered by the user – **if(newUser.password){  
      bcrypt code  
      }**
  + The bcrypt.hash function has a callback which is used to:
    - Set the newPassword = hash
    - Save the newUser - newUser.save()
    - Which save function has a callback which returns the saved data which is put in json - .then(user => res.json(user))   
        
      **FRONT END SET UP**

**Create the react app:**

**create-react-app client** // this puts the app into a directory called client. Separating it from the backend/server code.

**cd client**

**npm start**

**Bootstrap:**

Either basic bootstrap

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>

Or

Bootstrap React

The best way to consume React Bootstrap is via the npm package which you can install with npm (or yarn if you prefer).

If you plan on customizing the Bootstrap Sass files, or don't want to use a CDN for the stylesheet, it may be helpful to install [vanilla Bootstrap](https://getbootstrap.com/docs/4.2/getting-started/download/#npm) as well.

**npm install react-bootstrap bootstrap**

React-Bootstrap doesn't ship with any included css. However, some stylesheet **is required** to use these components. How and which bootstrap styles you include is up to you, but the simplest way is to include the latest styles from the CDN.

**<!-- Bootstrap -->**

**<link**

**rel="stylesheet"**

**href="https://maxcdn.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css"**

**integrity="sha384-GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS"**

**crossorigin="anonymous"**

**/>**

**Font Awesome:**

**<!-- Font Awesome -->**

**<link**

**rel="stylesheet"**

**href="https://use.fontawesome.com/releases/v5.6.3/css/all.css"**

**integrity="sha384-UHRtZLI+pbxtHCWp1t77Bi1L4ZtiqrqD80Kn4Z8NTSRyMA2Fd33n5dQ8lWUE00s/"**

**crossorigin="anonymous"**

**/>**

**Concurrently**

If there is a back end, install concurrently so that the backend and front end server can be started with one command.

**cd ..** // cd to the master directory

**npm I concurrently**

change package.json in server directory:

“scripts”: {

“client-install” “npm install –prefix client”,

“start”: “node server.js”,

“server”: “nodemon server.js”,

**“client”: “npm start –prefix client”,**

**“dev”: “concurrently \”npm run server\” \”npm run client\”**

}

Can now run both backend and frontend servers with

npm run dev

from master directory.

**Clean up Create React App**

**Delete logo.svg in src**

**Amend app.js to remove import logo**

In the render return delete all in main div and add some temporary code such as <h1>App Name</h1>

**Delete content of App.css**

Can use App.css as the global style sheet for the app.

**Client Directory Structure**

Within src create a basic directory structure

**assets** // will hold images etc.

**components** // for react components. Will create subdirectories for each logical view. Eg layout to hold the header, footer etc.

[will add **actions**, **reducers** for redux if relevant]

**utils** // for utility files such setAuthToken.js

**validation**

**Install dependencies in client**

* axios
* classnames
* jwt-decode
* moment
* react-moment
* redux
* react-redux
* redux-thunk
* **npm i axios classnames jwt-decode moment react-moment redux react-redux redux-thunk**

**Styling – css**

**Sass**

To use sass: **npm install node-sass –save:**

* rename src/App.css to src/App.scss and update src/App.js to import src/App.scss
* To share variables between Sass files, you can use Sass imports. For example, src/App.scss and other component style files could include @import "./shared.scss"; with variable definitions. This will allow imports like
  + **@import** 'styles/\_colors.scss'; // assuming a styles directory under src/
  + **@import** '~nprogress/nprogress'; // importing a css file from the nprogress node module

**Note:** Must prefix imports from node\_modules with ~ as displayed above.

* node-sass also supports the SASS\_PATH variable.
* To use imports relative to a path you specify, and from node\_modules without adding the ~ prefix, you can add a [.env file](https://github.com/facebook/create-react-app/blob/master/docusaurus/docs/adding-custom-environment-variables.md#adding-development-environment-variables-in-env) at the project root with the variable SASS\_PATH=node\_modules:src. To specify more directories you can add them to SASS\_PATH separated by a : like path1:path2:path3.
* If you set SASS\_PATH=node\_modules:src, this will allow you to do imports like
  + **@import** 'styles/colors'; // assuming a styles directory under src/, where \_colors.scss partial file exists.
  + **@import** 'nprogress/nprogress'; // importing a css file from the nprogress node module

**Css Modules**

Its recommended to use css modules so that styles are scoped to the component to which they relate. To use modules:

* use the [name].module.scss file naming convention. Ie/eg **rename app.scss to app.module.scss etc**
* to use the stylesheet in the related component:

**import** React, { Component } **from** 'react';

**import** styles **from** './Button.module.scss'; // Import css modules stylesheet as styles

**import** './another-stylesheet.scss'; // Import regular stylesheet

**class** **Button** **extends** **Component** {

render() {

// reference as a js object

**return** <**button** className={styles.error}>Error Button</**button**>;

}

}

* class names must be camelCase.
* They are used within curly braces and to use more than one style, incl normal styles such as bootstrap, use classnames :
  + **npm i classnames --save**
  + import classnames from ‘classnames’;
  + className = {classnames( styles.errors, ‘btn’, ‘btn-info’)} //

**Navbar**

If there will be authentication eg for customers or staff, use a class component.

If using Navbar from bootstrap name the component Header.

To create transparent navbar delete the bg attribute, create a class (eg navbar-transparent) and in the custom css file (eg App.css) make the background-color transparent and have a media query which sets a background color if the max-width is 768px or maybe 812 (landscape on an iphone 10)