CIS 4900 Independent Study

Project 2 Tutorial Document

By: Igli Duro

Email: duroi@usf.edu

Preface

In this document I will be going over the steps to build a website that utilizes a MERN stack designed to allow users to register, login, and access tools that are uploaded by administrators. This document will be divided into sections based off features that were added in order and will be starting off from a mostly blank state. While not every feature desired is implemented I will go over the steps to build the current state of the site, while offering insight on portions that I ran into issues developing and potential avenues to take my work and build off.

Original Statement of work

Proposed statement of work: For this independent study project the student will create and document a cloud-hosted website that does the following:

1) **Allows for user registration and login**
    a. **Registration includes name and other information via a form**
    b. **Registration information is logged for later access**
    c. Registration can be vetted automatically or manually by a human
    d. Include a provision for required payment to be registered
2) ***Login to a website that contains downloadable tools (e.g., Excel spreadsheets), documents, and videos. The student is not developing the tools, they already exist.***
    a. All logins and tool downloads are logged for later access
3) **Provides a webpage for user feedback, generates an email to the administrator**
4) ***Provides admin access to registration and download logs, and to easily add tools to download page***
5) **Matches style ("look and feel") of, and can be integrated with, https://smhcollaborative.org/**
6) **Provide a step-by-step tutorial document for the construction of the website**

The student is to implement this using a MERN stack (with focus on React and Node.js) or some other current stack to be discussed with the instructor. Weekly group meetings to be scheduled. Some cost for hosting may be incurred. All code to be delivered with MIT license for future use by others. The student will learn React and Node.js.

***Bolded portions are features that are currently working**

****Bolded and italicized are features that were started on but not in a finished/working state.***

Starting off

If you are new to MERN stack development or web development overall I highly encourage you to spend some time going through some learning resources. Work through some tutorials and smaller projects to learn the different components that go into development. Perhaps even go through some free courses to learn the basics.

To be brief MERN stands for

- MongoDB - document database
- Express(.js) - Node.js web framework
- React(.js) - a client-side JavaScript framework
- Node(.js) - the premier JavaScript web server

Learning Resources

The following are learning resources I used when I was starting out from scratch. I have a background in programming but next to none in web development going into this.

- https://www.geeksforgeeks.org/mern-stack/
- https://www.mongodb.com/mern-stack
- https://www.mongodb.com/languages/mern-stack-tutorial
- https://create-react-app.dev/
- https://www.w3schools.com/react/
- https://reactjs.org/tutorial/tutorial.html
- https://www.w3schools.com/nodejs/nodejs_npm.asp
- https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/
- https://create-react-app.dev/docs/getting-started
- https://reactjs.org/community/courses.html

- https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/
- https://classic.yarnpkg.com/en/docs/install#windows-stable

Third Party Services

Throughout my development of this website, I made use of the following third party services to run and power it. As part of following this tutorial you will need to create accounts for each of them and I will be going over each of them in more detail as we integrate and use. Each of these services have free plans that you can utilize for testing purposes.

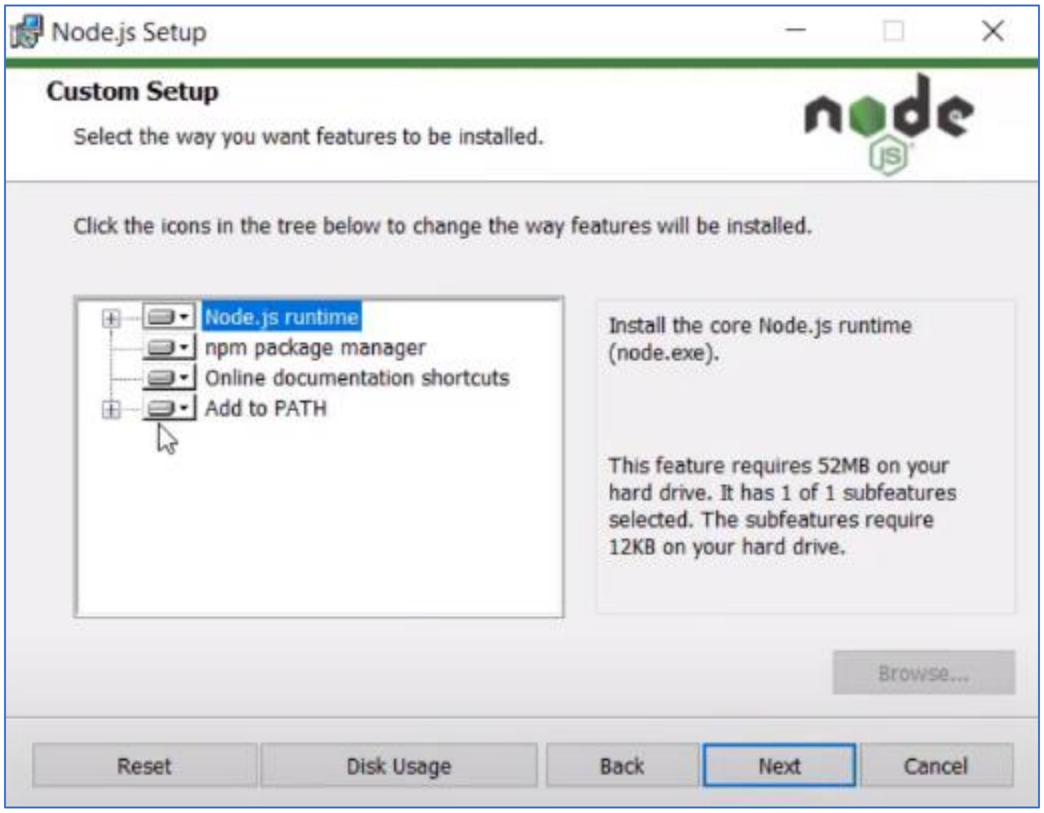
- MongoDB Atlas – this is a database hosting service that will be storing user information
- EmailJS – A service used to send emails and create templates all through the client side.
- Heroku – A platform where you can deploy your MERN stack application to have it hosted online.

1. Initial Setup

The first steps we will need to do is setting up our workspace. For the purposes of this document my development workspace is running on Windows using Visual Studio Code. So, I will be working from that perspective. To start we will need to install Node.js and npm. You can use the node.js installer on the official site found here which includes npm https://nodejs.org/en/download/.

npm is a package manager for installing/uninstalling different modules for nodejs. Another popular package manager is called yarn. If you want to read up more on yarn I have included some links in the Learning Resources section of this document. I recommend using one or the other to avoid issues later in development.

When running the installer make sure the option for adding to the PATH at this step is enabled.


Node.js Setup

Custom Setup

Select the way you want features to be installed.

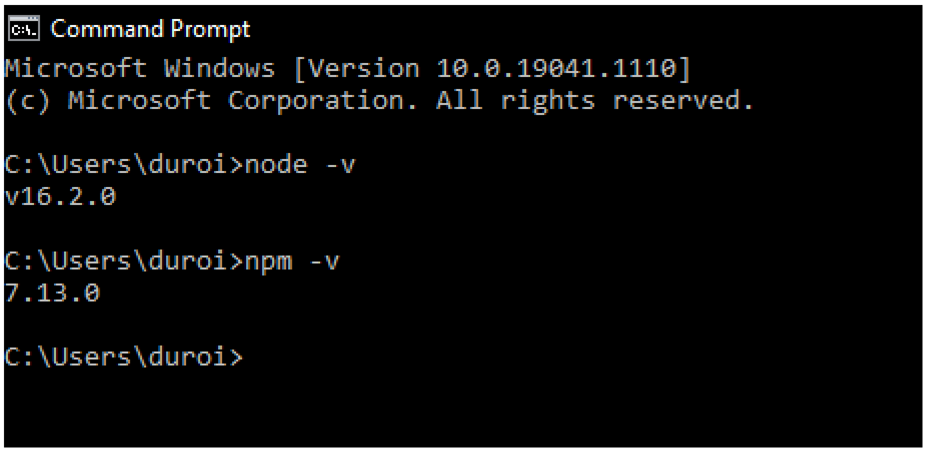
Click the icons in the tree below to change the way features will be installed.

- Node.js runtime
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

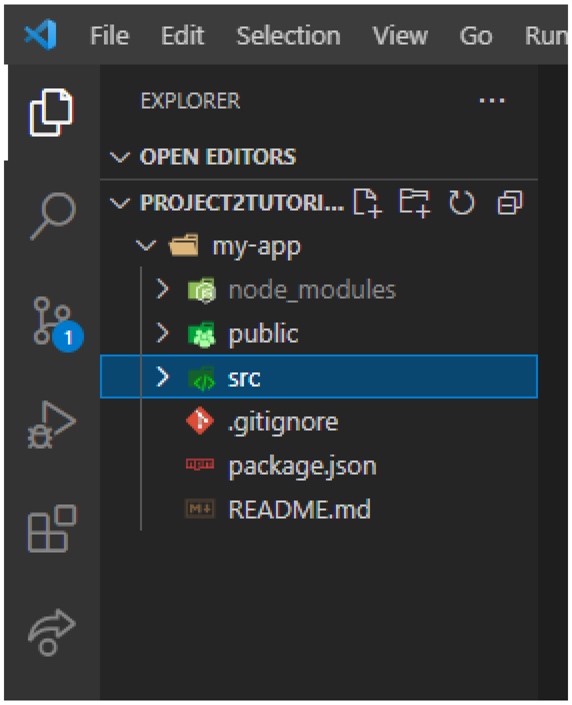
This feature requires 52MB on your hard drive. It has 1 of 1 subfeatures selected. The subfeatures require 12KB on your hard drive.

Afterwards, open a terminal and run the commands node -v and npm -v to check the versions to make sure they have installed properly. If so, you'll get a response back like this.

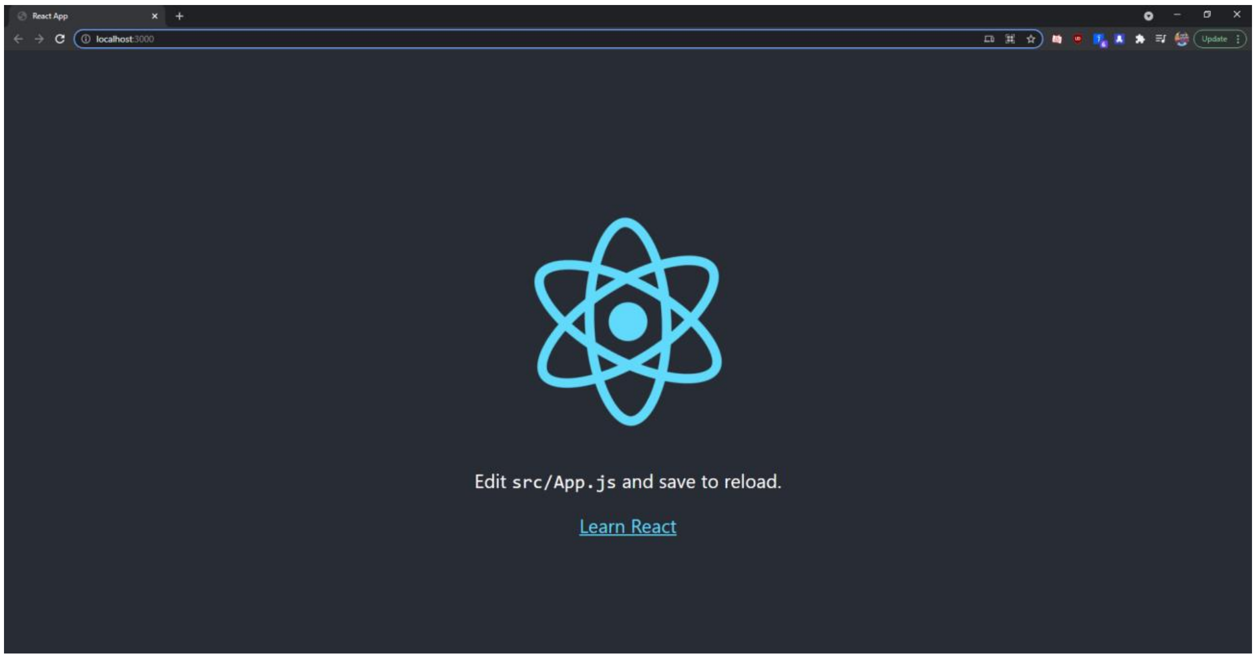

Command Prompt

```
Microsoft Windows [Version 10.0.19041.1110]
(c) Microsoft Corporation. All rights reserved.

C:\Users\duroi>node -v
v16.2.0

C:\Users\duroi>npm -v
7.13.0

C:\Users\duroi>
```

Next, open up Visual Studio Code along with the folder that'll act as the workspace for our application. Then run the command **npx create-react-app my-app** in the terminal. This creates a basic react application to start with. Your project folder should now look like this.



Next do **cd my-app** to change directories and then **npm start** run the react app in development mode. This will also open new window in your browser with the url http://localhost:3000. This is your react-app being hosted locally on your computer.



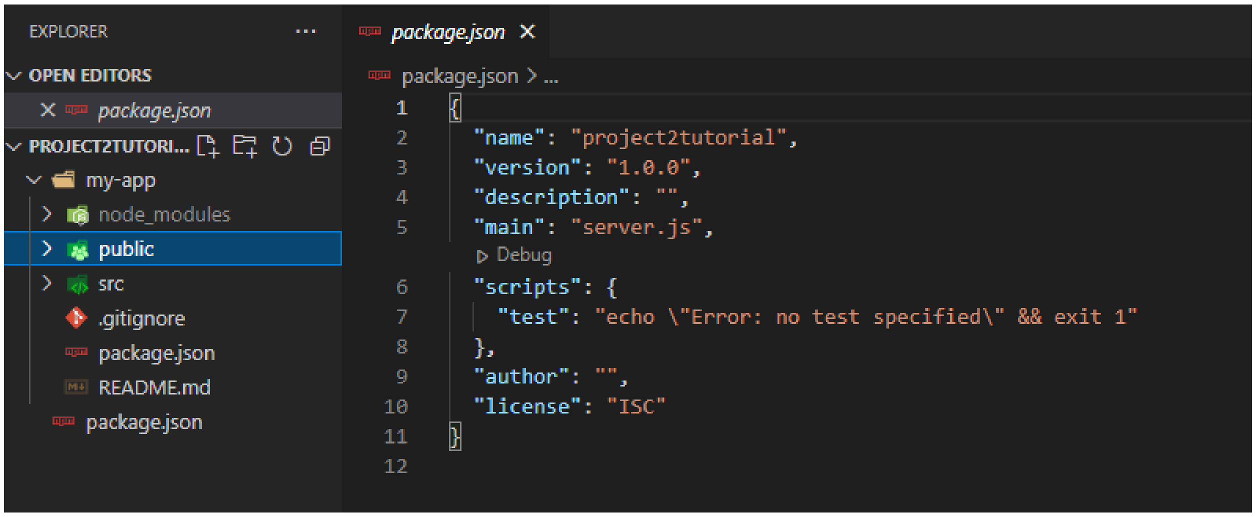
Edit `src/App.js` and save to reload.

At this point out initial setup is complete and from here you can begin modifying your App.js file.

2. Login and Registration

To begin work on our Login and Registration functionality we are going to first need to setup our backend and database that we will be connecting to. Change directories to your root folder, this would be the folder you initially opened for your workspace, one layer above the react-app we created. We are going to run the command **npm init** to initialize a package.json file.

Running this command will ask for some metadata such as name, version, and description. You can use the defaults for all these options but when asked for the main file replace index.js with server.js.



```
{
  "name": "project2tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Next we'll need to install the packages that we will be necessary to make our backend function. Here's an overview of these. If you want to learn more feel free to read up on each of the packages on www.npmjs.com where you can find overviews and usage examples.

- bcryptjs – for security purposes, it'll encrypt passwords by hashing
- concurrently – for running our front and backend at the same time
- express – works with nodejs for routing & requesting information
- body-parser – parse requests for middleware
- validator – validates certain input types
- passport – authenticates requests
- passport-jwt – authenticates with a json web token
- jsonwebtoken – used for authorization
- mongoose – interacts with MongoDB
- is-empty – used in tandem with validator
- nodemon – restarts server once you save changes

To install each of these you can run **npm i "package name"** one by one or at the same type with each other separated by spaces for example, **npm i bcryptjs concurrently express body-parser validator**

passport passport-jwt jsonwebtoken mongoose is-empty for nodemon we'll want to do **npm i -D nodemon** to install it as a dev dependency.

We'll also want to add two lines to the scripts section in our package.json. The scripts section acts as commands we can run.

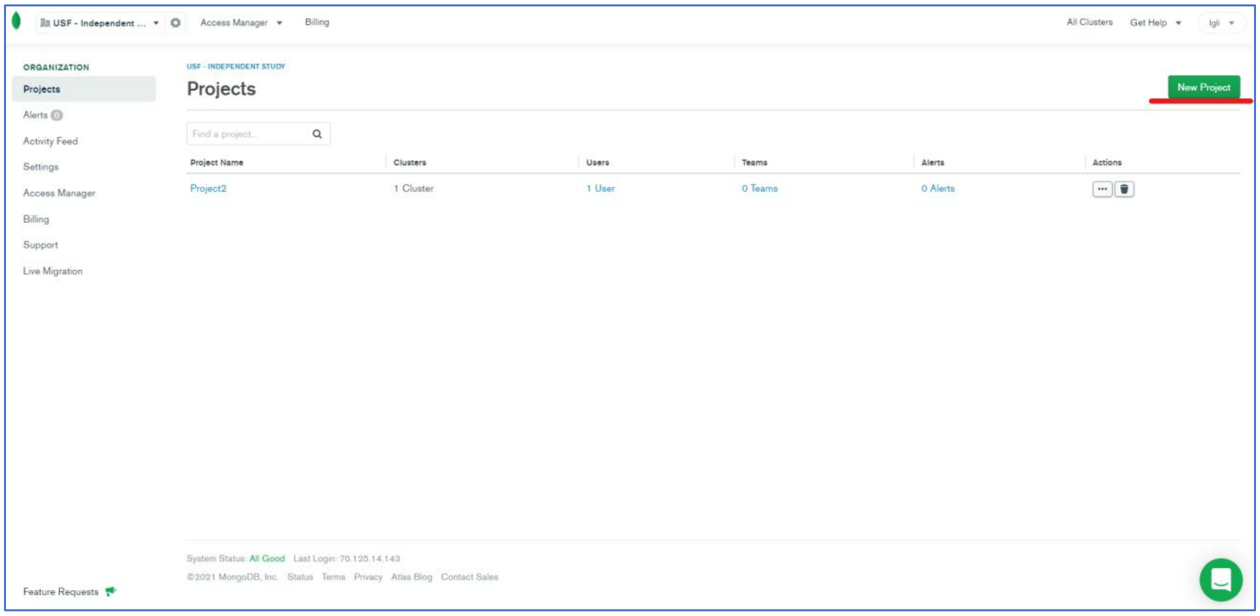
```
"start": "node server.js",
"server": "nodemon server.js"
```

At this point our package.json folder should look like this after install all dependencies mentioned.

```
{
  "name": "project2tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js",
    "server": "nodemon server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "body-parser": "^1.19.0",
    "concurrently": "^6.2.0",
    "express": "^4.17.1",
    "is-empty": "^1.2.0",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^5.13.3",
    "passport": "^0.4.1",
    "passport-jwt": "^4.0.0",
    "validator": "^13.6.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.12"
  }
}
```

Next we'll need to setup our MongoDB Atlas account. You can follow my steps here or use this tutorial, specifically the "Connecting to MongoDB Atlas" section https://www.mongodb.com/languages/mern-stack-tutorial.

Once signed in you'll want to setup a **new project**. Give it a name and add additional members if you happen to be working with multiple people that'll need access to the database.

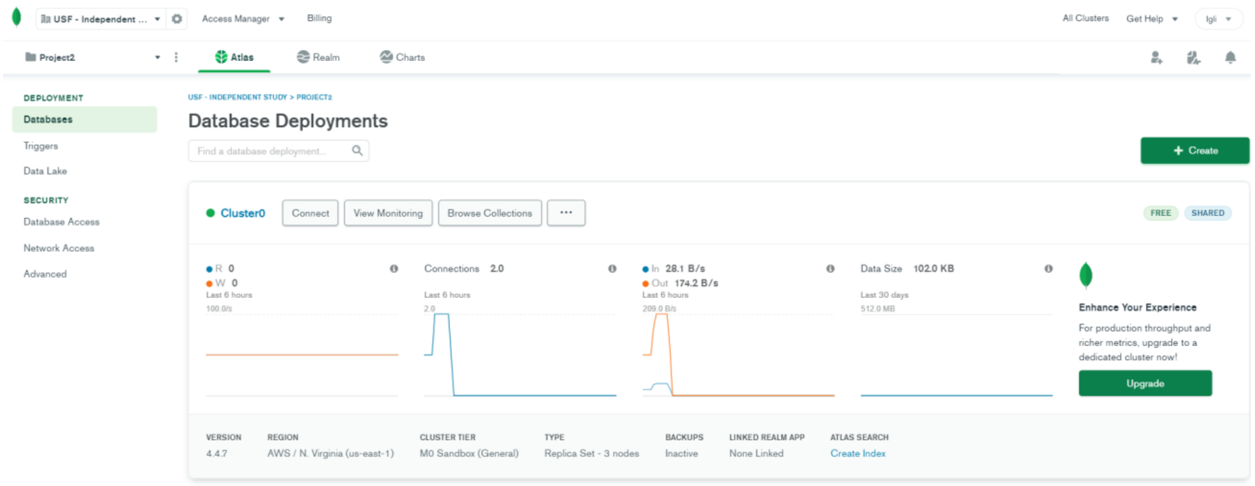


Project Name	Clusters	Users	Teams	Alerts	Actions
Project2	1 Cluster	1 User	0 Teams	0 Alerts	··· 🗑

System Status: All Good   Last Login: 70.125.14.143

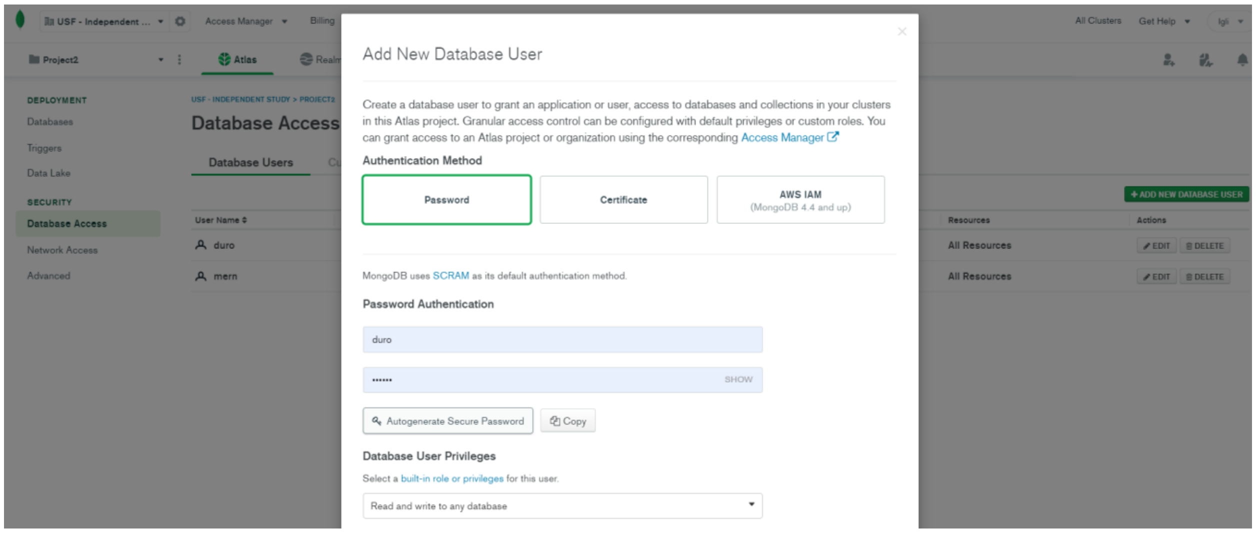
©2021 MongoDB, Inc.   Status   Terms   Privacy   Atlas Blog   Contact Sales

Once done create a **new cluster**, for the free version you'll be using a shared cluster. And you can upgrade later to scale the project up. Pick any of the providers with a free tier available with a server in a preferred region. You can create one free cluster per account. You can see my cluster here already created.



VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED REALM APP	ATLAS SEARCH
4.4.7	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Create Index

You'll next want to create a new **database user**, navigate to the **database access** section and click **add new database user** give a name and password along with read and write access. This will be needed to access the database later in our backend.



Add New Database User

Create a database user to grant an application or user, access to databases and collections in your clusters in this Atlas project. Granular access control can be configured with default privileges or custom roles. You can grant access to an Atlas project or organization using the corresponding Access Manager

Authentication Method: Password | Certificate | AWS IAM (MongoDB 4.4 and up)

MongoDB uses SCRAM as its default authentication method.

Password Authentication

duro

••••••

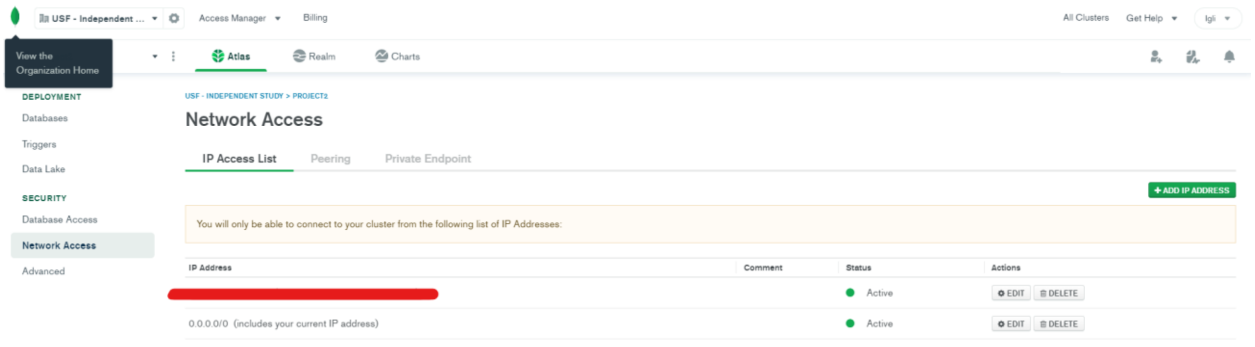
Database User Privileges

Select a built-in role or privileges for this user.

Read and write to any database

User Name	Resources	Actions
duro	All Resources	EDIT DELETE
mern	All Resources	EDIT DELETE

You'll also want to navigate to network access under security settings and whitelist your IP



USF - INDEPENDENT STUDY > PROJECT2

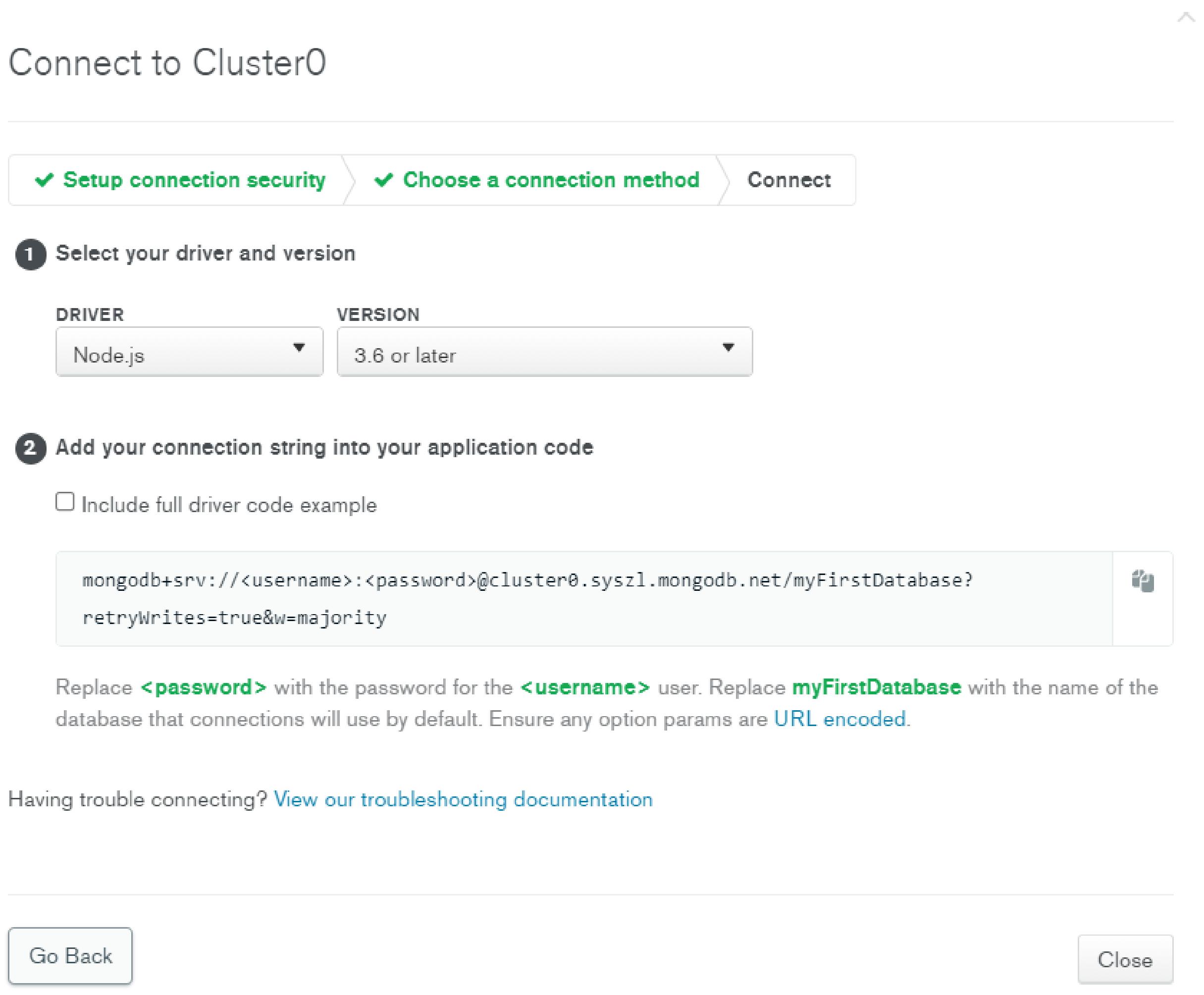
Network Access

IP Access List | Peering | Private Endpoint

You will only be able to connect to your cluster from the following list of IP Addresses:

IP Address	Comment	Status	Actions
[redacted]		Active	EDIT DELETE
0.0.0.0/0 (includes your current IP address)		Active	EDIT DELETE

Once all that is done we can go to back to our **databases** overview and click connect on the cluster we created. And select the second option that reads **connect your application**. Upon doing so you'll see the driver code we'll be using in our application. Replace **<username>** and **<password>** with the database username and password you created, including the angle braces. You can also replace **myFirstDatabase** with something of your choosing. MongoDB Atlas will create a folder in which to contain your database collections.


Connect to Cluster0

1 Select your driver and version

DRIVER: Node.js

VERSION: 3.6 or later

2 Add your connection string into your application code

☐ Include full driver code example

```
mongodb+srv://<username>:<password>@cluster0.syszl.mongodb.net/myFirstDatabase?
retryWrites=true&w=majority
```

Replace **<password>** with the password for the **<username>** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

Switch back over to VSCode and inside your root folder create a new directory called **config** and a file inside it called **keys.js** within that file type the following code. However, use your specific driver code, username, and password or else it will not work. By doing this we'll be able to make a reference to **mongoURI** every time we read and write from our database.

keys.js

```
module.exports = {
    mongoURI: "mongodb+srv://<username>:<password>@cluster0.syszl.mongodb.net/myFirstDatabase?retryWrites=true&w=majority", //Replace with your MongoDB driver code.
    secretOrKey: "secret"
  };
```

Next, in the root folder create a new file called **server.js** we'll be setting up our node.js server and be pulling in some of the dependencies we installed earlier. Go ahead and type in the following code. Afterwards run the command **nodemon run server** withing the root folder. If you see the message "MongoDB successfully connected" then congratulations we've set up a server and connected your backend to your database.

server.js

```
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const app = express();

// Bodyparser middleware
app.use(
  bodyParser.urlencoded({
    extended: false
  })
);
app.use(bodyParser.json());

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

// Connect to MongoDB
mongoose
  .connect(
    db,
    { useNewUrlParser: true }
  )
  .then(() => console.log("MongoDB successfully connected"))
  .catch(err => console.log(err));
const port = process.env.PORT || 5000; // process.env.port is Heroku's port if yo
u choose to deploy the app there
app.listen(port, () => console.log(`Server up and running on port ${port} !`));
```

Now, its time to setup a User Schema for our database. Essentially we're setting up a framework of information that will be stored for each entry in our database. For our User Schema, we'll want information like a name, email, and password. And if you want to make additions to the User Schema make sure you update any corresponding code that makes use of it to avoid issues.

Go ahead and create a folder called **models** in our root directory to hold any Schemas we create. Then make a file called **User.js** within that folder. We'll need to pull in dependencies for mongoose as well as Schema. Then create a new schema type. Go ahead and type the following code within the **User.js** file. As you can see, the elements within our User Schema are name, email, school_name, password, admin, and date. The admin element defaults to false. In the current state of the application, if you want to make an "Administrator" account you will have to manually edit the admin field to true within MongoDB Atlas.

Afterwards we need to export the Schema model so that we can use it in other files in our application.

User.js

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


// Create Schema
const UserSchema = new Schema ({
    name: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    school_name: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },
    admin: {
        type: Boolean,
        required: true,
        default: false
    },
    date: {
        type: Date,
        default: Date.now
    }

})

module.exports = User = mongoose.model ("users", UserSchema);
```

Next we'll need to be able to validate information being posted to our application and reject inputs if they don't meet our requirements. In your root directory create a folder called **validation** along with two files called **register.js** and **login.js** within it.

Starting with your **register.js** file. Import the Validator and isEmpty dependencies and we'll be checking to see if all the fields of our User Schema.

register.js

```
const Validator = require("validator");
const isEmpty = require("is-empty");


module.exports = function validateRegisterInput(data) {
    let errors = {};

  // Convert empty fields to an empty string so we can use validator functions
    data.name = !isEmpty(data.name) ? data.name : "";
    data.email = !isEmpty(data.email) ? data.email : "";
    data.school_name = !isEmpty(data.school_name) ? data.school_name : "";
    data.password = !isEmpty(data.password) ? data.password : "";
    data.password2 = !isEmpty(data.password2) ? data.password2 : "";

    // Name checks
    if (Validator.isEmpty(data.name)) {
    errors.name = "Name field is required";
    }

    // Email checks
    if (Validator.isEmpty(data.email)) {
        errors.email = "Email field is required";
    } else if (!Validator.isEmail(data.email)) {
        errors.email = "Email is invalid";
    }

    // School Name checks
    if (Validator.isEmpty(data.school_name)) {
        errors.name = "School name is required";
    }

    // Password checks
    if (Validator.isEmpty(data.password)) {
        errors.password = "Password field is required";
    }
    if (Validator.isEmpty(data.password2)) {
        errors.password2 = "Confirm password field is required";
    }
    if (!Validator.isLength(data.password, { min: 6, max: 30 })) {
        errors.password = "Password must be at least 6 characters";
    }
    if (!Validator.equals(data.password, data.password2)) {
        errors.password2 = "Passwords must match";
    }
    return {
        errors,
        isValid: isEmpty(errors)
  };
};
```

Then we'll be doing something similar for our **login.js**. However, logging in will only require an email and password so we only need to check with those fields.

login.js

```
const Validator = require("validator");
const isEmpty = require("is-empty");


module.exports = function validateLoginInput(data) {
    let errors = {};

  // Convert empty fields to an empty string so we can use validator functions
    data.email = !isEmpty(data.email) ? data.email : "";
    data.password = !isEmpty(data.password) ? data.password : "";
  // Email checks
    if (Validator.isEmpty(data.email)) {
      errors.email = "Email field is required";
    } else if (!Validator.isEmail(data.email)) {
      errors.email = "Email is invalid";
    }

  // Password checks
    if (Validator.isEmpty(data.password)) {
      errors.password = "Password field is required";
    }

  return {
      errors,
      isValid: isEmpty(errors)
    };
  };
```

For the next step we'll need to setup some API routes to pass in information a user is entering from the frontend to the backend and database. From your root directory create the **routes** folder and an **api** folder nested within. Then create a file called **users.js** in the **api** folder. We'll need to import the dependencies for express, router, bcrypt, and jwt. Along with that we'll need import the routes to the **keys.js, User.js, register.js, and login.js.** We'll also be using bcrypt here to hash passwords so they aren't stored as plaintext.

users.js

```
const express = require("express");
const router = express.Router();
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const keys = require("../../config/keys");


// Load input validation
const validateRegisterInput = require("../../validation/register");
const validateLoginInput = require("../../validation/login");

// Load User model
const User = require("../../models/User");

// @route POST api/users/register
// @desc Register user
// @access Public
```

```
router.post("/register", (req, res) => {
    // Form validation

  const { errors, isValid } = validateRegisterInput(req.body);

    // Check validation
    if (!isValid) {
      return res.status(400).json(errors);
    }

    User.findOne({ email: req.body.email }).then(user => {
      if (user) {
        return res.status(400).json({ email: "Email already exists" });
      } else {
        const newUser = new User({
          name: req.body.name,
          email: req.body.email,
          school_name: req.body.school_name,
          password: req.body.password
        });

        // Hash password before saving in database
        bcrypt.genSalt(10, (err, salt) => {
          bcrypt.hash(newUser.password, salt, (err, hash) => {
            if (err) throw err;
            newUser.password = hash;
            newUser
              .save()
              .then(user => res.json(user))
              .catch(err => console.log(err));
          });
        });
      }
    });
  });


// @route POST api/users/login
// @desc Login user and return JWT token
// @access Public
router.post("/login", (req, res) => {
    // Form validation

  const { errors, isValid } = validateLoginInput(req.body);
```

```
// Check validation
  if (!isValid) {
    return res.status(400).json(errors);
  }

  const email = req.body.email;
  const password = req.body.password;

  // Find user by email
  User.findOne({ email }).then(user => {
    // Check if user exists
    if (!user) {
      return res.status(404).json({ emailnotfound: "Email not found" });
    }

    // Check password
    bcrypt.compare(password, user.password).then(isMatch => {
      if (isMatch) {
        // User matched
        // Create JWT Payload
        const payload = {
          id: user.id,
          name: user.name
        };

        // Sign token
        jwt.sign(
          payload,
          keys.secretOrKey,
          {
            expiresIn: 31556926 // 1 year in seconds
          },
          (err, token) => {
            res.json({
              success: true,
              token: "Bearer " + token
            });
          }
        );
      } else {
        return res
          .status(400)
          .json({ passwordincorrect: "Password incorrect" });
      }
    });
  });
});


module.exports = router;
```

With that done create a new file called **passport.js** in our **config** folder. We’ll be using passport-jwt to authenticate using a JSON Web Token.

passport.js

```
const JwtStrategy = require("passport-jwt").Strategy;
const ExtractJwt = require("passport-jwt").ExtractJwt;
const mongoose = require("mongoose");
const User = mongoose.model("users");
const keys = require("../config/keys");

const opts = {};
opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();
opts.secretOrKey = keys.secretOrKey;

module.exports = passport => {
  passport.use(
    new JwtStrategy(opts, (jwt_payload, done) => {
      User.findById(jwt_payload.id)
        .then(user => {
          if (user) {
            return done(null, user);
          }
          return done(null, false);
        })
        .catch(err => console.log(err));
    })
  );
};
```

Now we must make some updates to our **server.js** file to include the passport dependency, middleware, config, and api routes. The updated file should look like this.

server.js (updated)

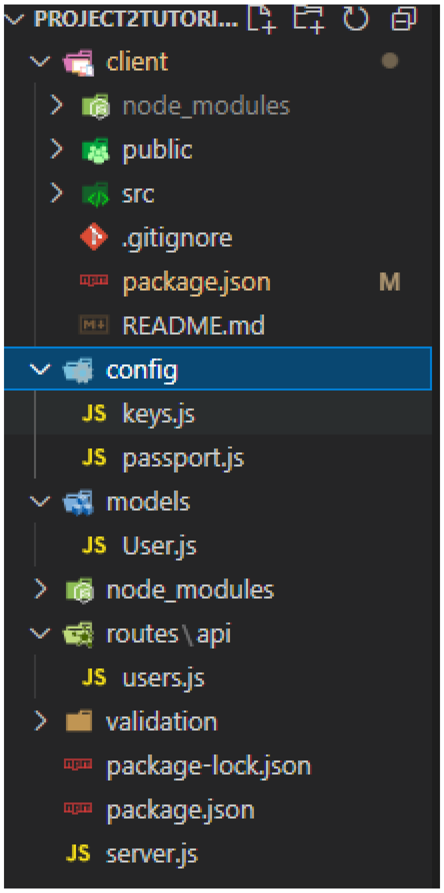
```javascript
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const app = express();
const path = require('path');

const passport = require("passport");

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



// Bodyparser middleware
app.use(
  bodyParser.urlencoded({
    extended: false
  })
);
app.use(bodyParser.json());
// DB Config
const db = require("./config/keys").mongoURI;
// Connect to MongoDB
mongoose
  .connect(
    db,
    { useNewUrlParser: true }
  )
  .then(() => console.log("MongoDB successfully connected"))
  .catch(err => console.log(err));



// Passport middleware
app.use(passport.initialize());

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

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

// process.env.port is Heroku's port if you choose to deploy the app there
const port = process.env.PORT || 5000;

// Serve static assets if in productions
```

```
if (process.env.NODE_ENV === 'production') {
  app.use(express.static('client/build'));

  app.get('*', (req,res) => {
    res.sendFile(path.resolve(__dirname, 'client', 'build', 'index.html'));
  });
}

app.listen(port, () => console.log(`Server up and running on port ${port} !`));
```

Next we'll be moving to the front-end and making the react-app. Before that go into your root folder's **package.json** and make an addition to the scripts section. Add the following lines before your last script. Now we can do **npm run dev** to start up our backend and frontend at the same time.

```
  "client-install": "npm install --prefix client",
  "client": "npm start --prefix client",
  "proxy": "http://localhost:5000",
  "dev": "concurrently \"npm run server\" \"npm run client\""
```

Along with that change the file name for the folder containing your react app to **client**. Our folder structure should look like this now.



```
PROJECT2TUTORI...
  client
    node_modules
    public
    src
    .gitignore
    package.json          M
    README.md
  config
    keys.js
    passport.js
  models
    User.js
  node_modules
  routes\api
    users.js
  validation
  package-lock.json
  package.json
  server.js
```

Next in your terminal change directories to your client folder as we need to install some packages we will be using. Once again feel free to look up additional documentation regarding the packages. The following packages are

- axios – client for making requests to backend
- jwt-decode – used to decode jwt to get user data
- react-redux – allows use of Redux
- react-router-dom – allows for rerouting to different pages
- redux – used to manage component states
- redux-thunk – redux middleware
- classnames – used for conditional classes.

You can use **npm i axois jwt-decode react-redux react-router-dom redux redux-thunk classnames** in the terminal.

Now we can clean up our client folder and remove some starter files we don't really need. In the **src** folder remove **logo.svg** and remove the import of that file in **App.js**. Remove all the CSS in **App.css** to use custom CSS instead. Finally remove everything inside the div tags in **App.js** and replace it with the following.

```
import './App.css';

function App() {
  return (
    <div className="App">
      <h1>Hello World!!!!</h1>
    </div>
  );
}

export default App;
```

Upon saving our react app will look like this.

Hello World!!!!

Next let's create a **components** folder in **src.** In our components folder create another folder called **layouts**, and then the files **Navbar.js** and **Landing.js** inside layouts.

For our **Navbar.js** import the Component dependency from react and Link dependency from react-router-dom. We'll be using link to redirect to different webpages.

Navbar.js

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
```

```
class Navbar extends Component {
  render() {
    return (

      <div className="navbar-fixed">
       <nav class="navbar navbar-expand-lg navbar-light bg-light">

          <div className="nav-wrapper white">
            <Link
              to="/"
              style={{
                fontFamily: "wolf2"
              }}
              className="col s5 brand-logo center black-text"
            >
              <i className="material-icons"></i>
              Project 2
            </Link>


          </div>
        </nav>

      </div>

    );
  }
}
export default Navbar;
```

For our **Landing.js** file, we'll use it to link a login and registration page we that we'll create soon.

Landing.js

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

class Landing extends Component {
  render() {
    return (
      <div style={{ height: "50vh" }} className="container valign-wrapper">
        <div className="row">
          <div className="col center-align">
```

```
            <h3>
              <span style={{ fontFamily: "wolf2" }}> Login to continue or create
an account.  </span>

            </h3>
            <p className="flow-text grey-text text-darken-1">

            </p>
            <br />
            <div className="col s6">
              <Link
                to="/login"
                style={{
                  width: "140px",
                  borderRadius: "3px",
                  letterSpacing: "1.5px"
                }}
                className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
              >
                Log In
              </Link>
            </div>
            <div className="col s6">
              <Link
                to="/register"
                style={{
                  width: "140px",
                  borderRadius: "3px",
                  letterSpacing: "1.5px"
                }}
                className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
              >
                Register
              </Link>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
export default Landing;
```

Now, import our Navbar and Landing into your **App.js** file and add t hem to the render function.

```
import React, { Component } from "react";
import "./App.css";


import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
class App extends Component {
  render() {
    return (
      <div className="App">
        <Navbar />
        <Landing />
      </div>
    );
  }
}
export default App;
```

Let's also make some changes to **index.html** in our public folder. Here we can use CSS and HTML along with scripts that take effect across our application. The updated index.html should look like this.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-
to-fit=no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <!-- Compiled and minified CSS -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materiali
ze/1.0.0/css/materialize.min.css">
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="sty
lesheet">
    <title>Project 2</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>
  </body>
</html>
```

Up next we’ll be setting up our login and registration forms. Create a new folder called **auth** inside our components folder and **login.js** & **register.js** inside **auth.**

For our **register.js** we’ll need to write conditional events whenever the user changes some information inside a text both or clicks a button to submit their form. And we’ll need to use JavaScript and HTML in the render function to display the forms.

register.js

```
import React, { Component } from "react";
import { Link, withRouter } from "react-router-dom";
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { registerUser } from "../../actions/authActions";
import classnames from "classnames";
class Register extends Component {
  constructor() {
    super();
    this.state = {
      name: "",
      email: "",
      school_name: "",
      password: "",
      password2: "",
      errors: {}
    };
}

componentDidMount() {
    // If logged in and user navigates to Register page, should redirect them to dashboard
    if (this.props.auth.isAuthenticated) {
      this.props.history.push("/dashboard");
    }
}


componentWillReceiveProps(nextProps) {
    if (nextProps.errors) {
      this.setState({
        errors: nextProps.errors
      });
    }
  }
onChange = e => {
    this.setState({ [e.target.id]: e.target.value });
  };
onSubmit = e => {
    e.preventDefault();
const newUser = {
      name: this.state.name,
      email: this.state.email,
      school_name: this.state.school_name,
      password: this.state.password,
      password2: this.state.password2
    };
this.props.registerUser(newUser, this.props.history);
  };
render() {
    const { errors } = this.state;
return (
      <div className="container">
        <div className="row">
          <div className="col s8 offset-s2">
            <Link to="/" className="btn-flat waves-effect">
              <i className="material-icons left">keyboard_backspace</i> Back to
              home
            </Link>
            <div className="col s12" style={{ paddingLeft: "11.250px" }}>
              <h4>
                <b>Register</b> below
              </h4>
              <p className="grey-text text-darken-1">
                Already have an account? <Link to="/login">Log in</Link>
              </p>
            </div>

            {/* Registration Form */}
            <form noValidate onSubmit={this.onSubmit}>

              {/* Name Field */}
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.name}
                  error={errors.name}
                  id="name"
                  type="text"
                  className={classnames("", {
                    invalid: errors.name
                  })}
                />
                <label htmlFor="name">Name</label>
                <span className="red-text">{errors.name}</span>
              </div>

              {/* Email Field */}
              <div className="input-field col s12 ">
                <input
                  onChange={this.onChange}
                  value={this.state.email}
                  error={errors.email}
                  id="email"
                  type="email"
                  className={classnames("", {
                    invalid: errors.email
                  })}
                />
                <label htmlFor="email">Email</label>
                <span className="red-text">{errors.email}</span>
              </div>

              {/* School Name Field */}
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.school_name}
                  error={errors.school_name}
                  id="school_name"
                  type="text"
                  className={classnames("", {
                    invalid: errors.school_name
                  })}
                />
                <label htmlFor="name">School Name</label>
                <span className="red-text">{errors.name}</span>
              </div>

              {/* Password Field */}
              <div className="input-field col s12">
                <input
```

```
                  onChange={this.onChange}
                  value={this.state.password}
                  error={errors.password}
                  id="password"
                  type="password"
                  className={classnames("", {
                    invalid: errors.password
                  })}
                />
                <label htmlFor="password">Password</label>
                <span className="red-text">{errors.password}</span>
              </div>

              {/* Confirm Password Field */}
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.password2}
                  error={errors.password2}
                  id="password2"
                  type="password"
                  className={classnames("", {
                    invalid: errors.password2
                  })}
                />
                <label htmlFor="password2">Confirm Password</label>
                <span className="red-text">{errors.password2}</span>
              </div>

              {/* Sign Up Button */}
              <div className="col s12" style={{ paddingLeft: "11.250px" }}>
                <button
                  style={{
                    width: "150px",
                    borderRadius: "3px",
                    letterSpacing: "1.5px",
                    marginTop: "1rem"
                  }}
                  type="submit"
                  className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
                >
                  Sign up
                </button>
              </div>
```

```
            </form>
          </div>
        </div>
      </div>
    );
  }
}
Register.propTypes = {
  registerUser: PropTypes.func.isRequired,
  auth: PropTypes.object.isRequired,
  errors: PropTypes.object.isRequired
};
const mapStateToProps = state => ({
  auth: state.auth,
  errors: state.errors
});
export default connect(
  mapStateToProps,
  { registerUser }
)(withRouter(Register));
```

Our **login.js** will look similar, however with less fields.

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { loginUser } from "../../actions/authActions";
import classnames from "classnames";
class Login extends Component {
  constructor() {
    super();
    this.state = {
      email: "",
      password: "",
      errors: {}
    };
}

componentDidMount() {
  // If logged in and user navigates to Register page, should redirect them to da
shboard
  if (this.props.auth.isAuthenticated) {
    this.props.history.push("/dashboard");
  }
```

```
}


componentWillReceiveProps(nextProps) {
    if (nextProps.auth.isAuthenticated) {
      this.props.history.push("/dashboard"); // push user to dashboard when they
login
    }
if (nextProps.errors) {
      this.setState({
        errors: nextProps.errors
      });
    }
  }
onChange = e => {
    this.setState({ [e.target.id]: e.target.value });
  };
onSubmit = e => {
    e.preventDefault();
const userData = {
      email: this.state.email,
      password: this.state.password
    };
this.props.loginUser(userData); // since we handle the redirect within our compon
ent, we don't need to pass in this.props.history as a parameter
  };
render() {
    const { errors } = this.state;
return (
      <div className="container">
        <div style={{ marginTop: "4rem" }} className="row">
          <div className="col s8 offset-s2">
            <Link to="/" className="btn-flat waves-effect">
              <i className="material-icons left">keyboard_backspace</i> Back to
              home
            </Link>
            <div className="col s12" style={{ paddingLeft: "11.250px" }}>
              <h4>
                <b>Login</b> below
              </h4>
              <p className="grey-text text-darken-1">
                Don't have an account? <Link to="/register">Register</Link>
              </p>
            </div>
            <form noValidate onSubmit={this.onSubmit}>
```

```
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.email}
                  error={errors.email}
                  id="email"
                  type="email"
                  className={classnames("", {
                    invalid: errors.email || errors.emailnotfound
                  })}
                />
                <label htmlFor="email">Email</label>
                <span className="red-text">
                  {errors.email}
                  {errors.emailnotfound}
                </span>
              </div>
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.password}
                  error={errors.password}
                  id="password"
                  type="password"
                  className={classnames("", {
                    invalid: errors.password || errors.passwordincorrect
                  })}
                />
                <label htmlFor="password">Password</label>
                <span className="red-text">
                  {errors.password}
                  {errors.passwordincorrect}
                </span>
              </div>
              <div className="col s12" style={{ paddingLeft: "11.250px" }}>
                <button
                  style={{
                    width: "150px",
                    borderRadius: "3px",
                    letterSpacing: "1.5px",
                    marginTop: "1rem"
                  }}
                  type="submit"
                  className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
```

```
                >
                  Login
                </button>
              </div>
            </form>
          </div>
        </div>
      </div>
    );
  }
}
Login.propTypes = {
  loginUser: PropTypes.func.isRequired,
  auth: PropTypes.object.isRequired,
  errors: PropTypes.object.isRequired
};
const mapStateToProps = state => ({
  auth: state.auth,
  errors: state.errors
});
export default connect(
  mapStateToProps,
  { loginUser }
)(Login);
```

Now we'll need to update **App.js** with our login and registration page.

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route } from "react-router-dom";


import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register from "./components/auth/Register";
import Login from "./components/auth/Login";


class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <Navbar />
          <Route exact path="/" component={Landing} />
          <Route exact path="/register" component={Register} />
          <Route exact path="/login" component={Login} />
        </div>
      </Router>
    );
  }
}
export default App;
```

Next we'll need to setup a Redux router for our application. Redux helps manage React the states of components but will take some setup that we'll go into next. In our **App.js** we can make the following changes after importing react-redux.

```
import React, { Component } from "react";

import { BrowserRouter as Router, Route } from "react-router-dom";

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

import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register from "./components/auth/Register";
import Login from "./components/auth/Login";


class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router>
          <div className="App">
            <Navbar />
            <Route exact path="/" component={Landing} />
            <Route exact path="/register" component={Register} />
            <Route exact path="/login" component={Login} />
          </div>
        </Router>
      </Provider>
    );
  }
}
export default App;
```

We’ll now need to setup a **store.js** file inside our **src** folder, along with two folders in our folder called **actions** and **reducers.**

In our **actions** folder make two files called **authActions.js** and **types.js**

In our **reducers** folder make three files called **index.js**, **authReducers.js**, and **errorReducers.js**

In our **store.js** file we’ll need to store states to send to components. We’ll be importing redux and redux-thunk dependencies.

store.js

```
import { createStore, applyMiddleware, compose } from "redux";
import thunk from "redux-thunk";
import rootReducer from "./reducers";


const initialState = {};

const middleware = [thunk];

const store = createStore(
  rootReducer,
  initialState,
  compose(
    applyMiddleware(...middleware),
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
  )
);

export default store;
```

In our **types.js** file we’ll need to define the following actions.

```
export const GET_ERRORS = "GET_ERRORS";
export const USER_LOADING = "USER_LOADING";
export const SET_CURRENT_USER = "SET_CURRENT_USER";
```

In our **authActions.js** we’ll need to use JWT tokens to define actions for user login and registration.

```
import axios from "axios";
import setAuthToken from "../utils/setAuthToken";
import jwt_decode from "jwt-decode";
import {
  GET_ERRORS,
  SET_CURRENT_USER,
  USER_LOADING
} from "./types";
// Register User
export const registerUser = (userData, history) => dispatch => {
  axios
    .post("/api/users/register", userData)
    .then(res => history.push("/login")) // re-
direct to login on successful register
    .catch(err =>
      dispatch({
        type: GET_ERRORS,
        payload: err.response.data
      })
    );
};
// Login - get user token
export const loginUser = userData => dispatch => {
  axios
    .post("/api/users/login", userData)
    .then(res => {
      // Save to localStorage
// Set token to localStorage
      const { token } = res.data;
      localStorage.setItem("jwtToken", token);
      // Set token to Auth header
      setAuthToken(token);
      // Decode token to get user data
      const decoded = jwt_decode(token);
      // Set current user
      dispatch(setCurrentUser(decoded));
    })
    .catch(err =>
      dispatch({
        type: GET_ERRORS,
        payload: err.response.data
      })
    );
};
// Set logged in user
```

```
export const setCurrentUser = decoded => {
  return {
    type: SET_CURRENT_USER,
    payload: decoded
  };
};
// User loading
export const setUserLoading = () => {
  return {
    type: USER_LOADING
  };
};
// Log user out
export const logoutUser = () => dispatch => {
  // Remove token from local storage
  localStorage.removeItem("jwtToken");
  // Remove auth header for future requests
  setAuthToken(false);
  // Set current user to empty object {} which will set isAuthenticated to false
  dispatch(setCurrentUser({}));
};
```

For our **authReducer.js** we'll need to import our types.js and reduce the number of requests for user logins and registrations.

```
import {
    SET_CURRENT_USER,
    USER_LOADING
  } from "../actions/types";
  const isEmpty = require("is-empty");
  const initialState = {
    isAuthenticated: false,
    user: {},
    loading: false
  };
  export default function(state = initialState, action) {
    switch (action.type) {
      case SET_CURRENT_USER:
        return {
          ...state,
          isAuthenticated: !isEmpty(action.payload),
          user: action.payload
        };
      case USER_LOADING:
        return {
          ...state,
          loading: true
        };
      default:
        return state;
    }
  }
```

For our **errorReducer.js**, it'll look like this.

```
import { GET_ERRORS } from "../actions/types";
const initialState = {};
export default function(state = initialState, action) {
  switch (action.type) {
    case GET_ERRORS:
      return action.payload;
    default:
      return state;
  }
}
```

In our **reducers** folder, we'll need to setup the **index.js** file there to combine the authReducer and errorReducer.

```
import { combineReducers } from "redux";
import authReducer from "./authReducer";
import errorReducer from "./errorReducer";
export default combineReducers({
  auth: authReducer,
  errors: errorReducer
});
```

With our Actions and Reducer files set up, create a new folder within **src** called **utils**. And in it, the **setAuthToken.js** file. Here will set a token for every request if a user is logged in. If they aren't we delete they token.

```
import axios from "axios";
const setAuthToken = token => {
  if (token) {
    // Apply authorization token to every request if logged in
    axios.defaults.headers.common["Authorization"] = token;
  } else {
    // Delete auth header
    delete axios.defaults.headers.common["Authorization"];
  }
};
export default setAuthToken;
```

With that setup. Let’s create a dashboard that users will go to after logging in. In our components folder, create a **dashboard** folder, and the **Dashboard.js** file inside that. Our dashboard file will act as a hub to go to different portions of the site, such as the feedback page or the tools page. As-well as allowing for the user to logout. If you want to make additional webpages for users to use once logged in you can add them to the dashboard file as-well importing them into the main App.js.

```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { logoutUser } from "../../actions/authActions";
import { Link } from "react-router-dom";


// User logout

class Dashboard extends Component {
  onLogoutClick = e => {
    e.preventDefault();
    this.props.logoutUser();
  };
render() {
    const { user } = this.props.auth;
return (
      <div style={{ height: "75vh" }} className="container valign-wrapper">
        <div className="row">
          <div className="col s12 left-align">

            {/* Greeting Message */}
            <h4>
              <b>Welcome,</b> {user.name.split(" ")[0]}
              <p>
                You are now logged in.
              </p>
            </h4>


            {/* Logout button */}
            <div className="col s6">
            <button
              style={{
                width: "150px",
                borderRadius: "3px",
                letterSpacing: "1.5px",
                marginTop: "1rem"
              }}
              onClick={this.onLogoutClick}
              className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
            >
              Logout
            </button>
            </div>


            {/* Feedback button */}
            <div className="col s6">
            <Link
                to="/feedback"
                style={{
                  width: "150px",
                  borderRadius: "3px",
                  letterSpacing: "1.5px",
                  marginTop: "1rem"
                }}
                className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
              >
                feedback
              </Link>
              </div>


            {/* Tools Button */}
            <div className="col s6">
            <Link
                to="/tool"
                style={{
                  width: "150px",
                  borderRadius: "3px",
                  letterSpacing: "1.5px",
                  marginTop: "1rem"
                }}
                className="btn btn-large waves-effect waves-
light hoverable blue accent-4"
              >
                Tools
              </Link>
              </div>



          </div>
        </div>
      </div>
    );
  }
}
Dashboard.propTypes = {
  logoutUser: PropTypes.func.isRequired,
  auth: PropTypes.object.isRequired
};
const mapStateToProps = state => ({
  auth: state.auth
});
export default connect(
  mapStateToProps,
  { logoutUser }
)(Dashboard);
```

Next, its import to protect pages so not everybody can access them. Perhaps you want only a user who's logged in to access something, or an admin only page. So we'll need to create a private route. In the components directory, create a folder called **private-route** and the **PrivateRoute.js** file inside it. In this file, we'll make it so any link marked private will redirect the user to another webpage if they don't meet the conditions. In this case, redirecting a user who isn't logged in back to the login screen.

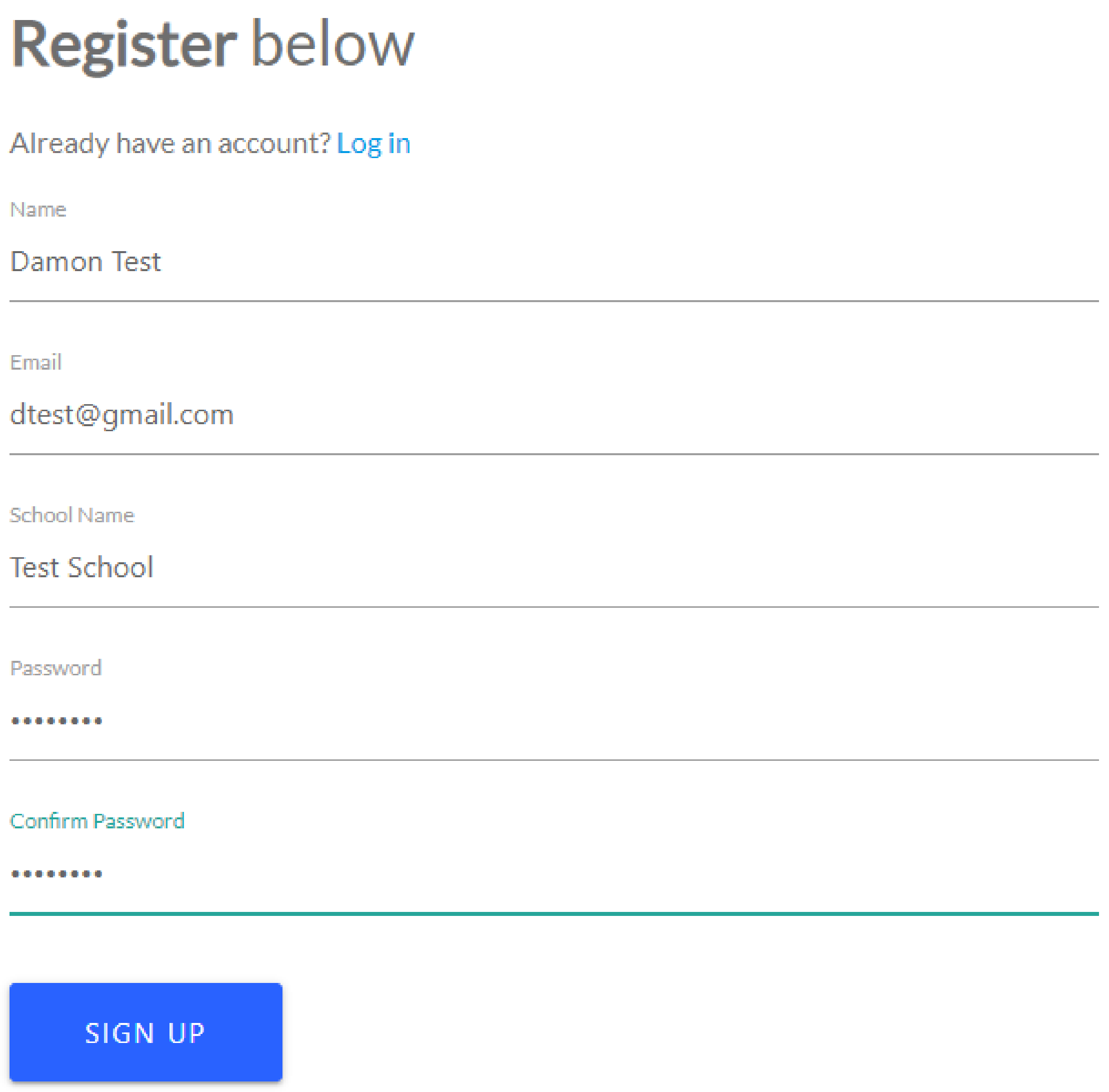
```
import React from "react";
import { Route, Redirect } from "react-router-dom";
import { connect } from "react-redux";
import PropTypes from "prop-types";
const PrivateRoute = ({ component: Component, auth, ...rest }) => (
  <Route
    {...rest}
    render={props =>
      auth.isAuthenticated === true ? (
        <Component {...props} />
      ) : (
        <Redirect to="/login" />
      )
    }
  />
);
PrivateRoute.propTypes = {
  auth: PropTypes.object.isRequired
};
const mapStateToProps = state => ({
  auth: state.auth
});
export default connect(mapStateToProps)(PrivateRoute);
```

Lastly, for the Login and Registration page, we need to update the **App.js** file. The updated **App.js** file will look like this.

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

import jwt_decode from "jwt-decode";
import setAuthToken from "./utils/setAuthToken";

import { setCurrentUser, logoutUser } from "./actions/authActions";
import { Provider } from "react-redux";
import store from "./store";

import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register from "./components/auth/Register";
import Login from "./components/auth/Login";
import PrivateRoute from "./components/private-route/PrivateRoute";
import Dashboard from "./components/dashboard/Dashboard";


// Check for token to keep user logged in
if (localStorage.jwtToken) {
  // Set auth token header auth
  const token = localStorage.jwtToken;
  setAuthToken(token);
  // Decode token and get user info and exp
  const decoded = jwt_decode(token);
  // Set user and isAuthenticated
  store.dispatch(setCurrentUser(decoded));
// Check for expired token
  const currentTime = Date.now() / 1000; // to get in milliseconds
  if (decoded.exp < currentTime) {
    // Logout user
    store.dispatch(logoutUser());
    // Redirect to login
    window.location.href = "./login";
  }
}
class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router>
          <div className="App">
            <Navbar />
            <Route exact path="/" component={Landing} />
            <Route exact path="/register" component={Register} />
            <Route exact path="/login" component={Login} />
            <Switch>
            //Mark routes with PrivateRoutes to protect them.
              <PrivateRoute exact path="/dashboard" component={Dashboard} />
            </Switch>
          </div>
        </Router>
      </Provider>
    );
  }
}
export default App;
```

With that, our login and registration system is finished.

Users can register new accounts.



Register below

Already have an account? Log in

Name

Damon Test

Email

dtest@gmail.com

School Name

Test School

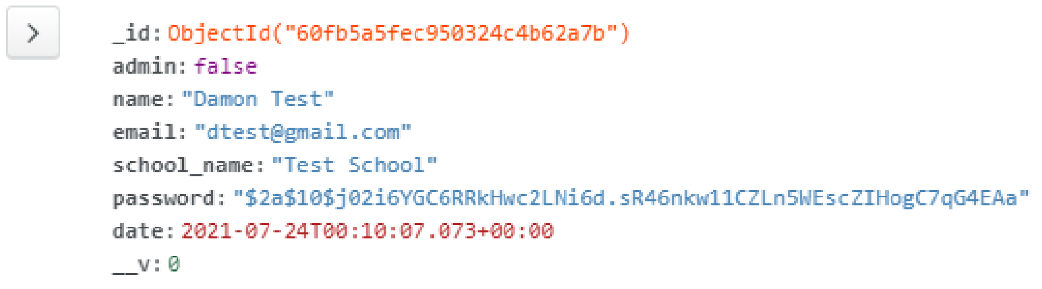
Password

••••••••

Confirm Password

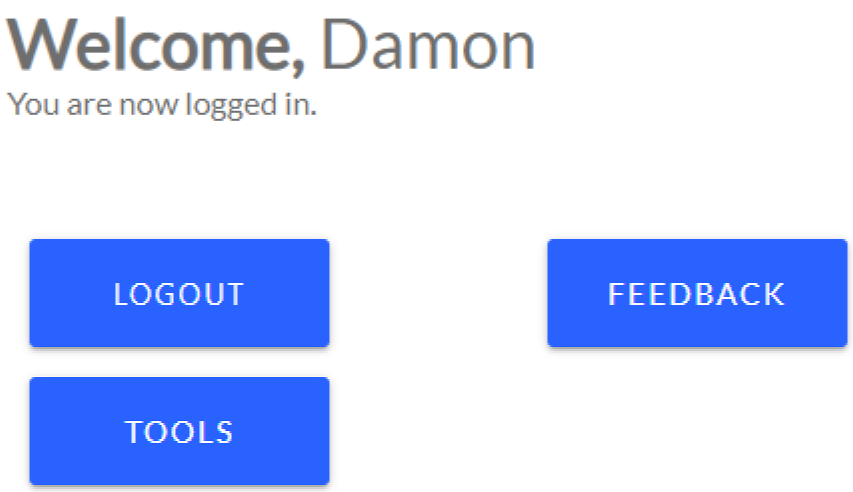
••••••••

Information is passed to the backend and database.



```
_id: ObjectId("60fb5a5fec950324c4b62a7b")
admin: false
name: "Damon Test"
email: "dtest@gmail.com"
school_name: "Test School"
password: "$2a$10$j02i6YGC6RRkHwc2LNi6d.sR46nkw11CZLn5WEscZIHogC7qG4EAa"
date: 2021-07-24T00:10:07.073+00:00
__v: 0
```

Then users can login and information is retrieved. With this a user that is logged in.


Welcome, Damon

You are now logged in.

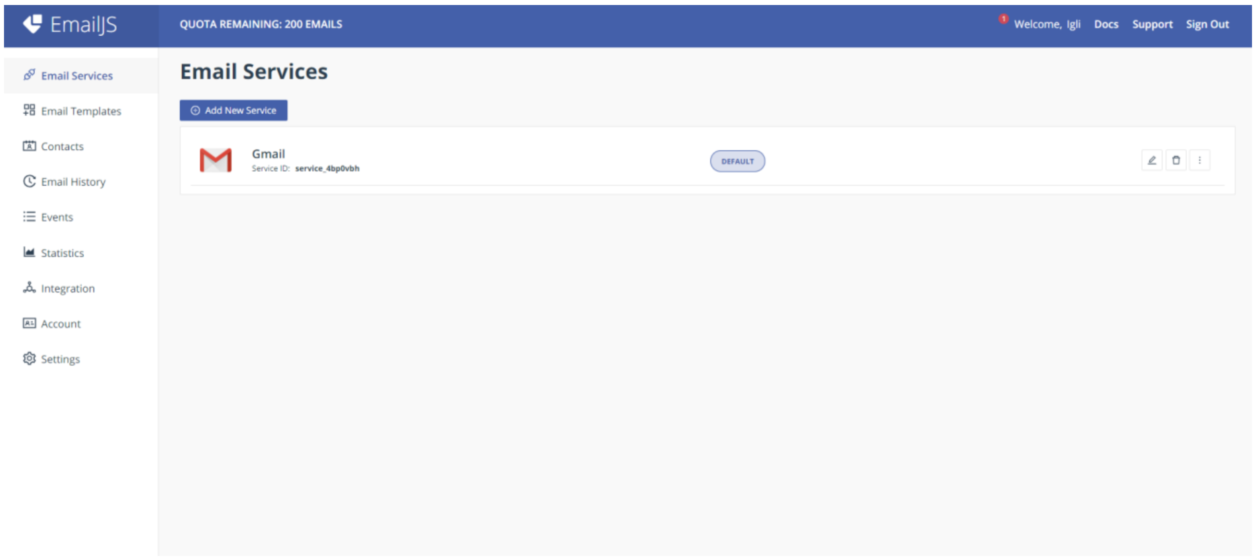
While this section has been lengthy, a lot of it was setting up the foundation. With it set making additions to other webpages you wish to implement becomes a lot easier. In the next section I'll go over the feedback page.

3. Feedback Page

In this section we'll be going over the steps to get a working feedback page added to our application. There are a few steps in the setup process. But once that's done writing the code necessary is actually very easy.

First off, if you haven't already, create and account with EmailJS. Once logged in, let's get familiar with the layout.

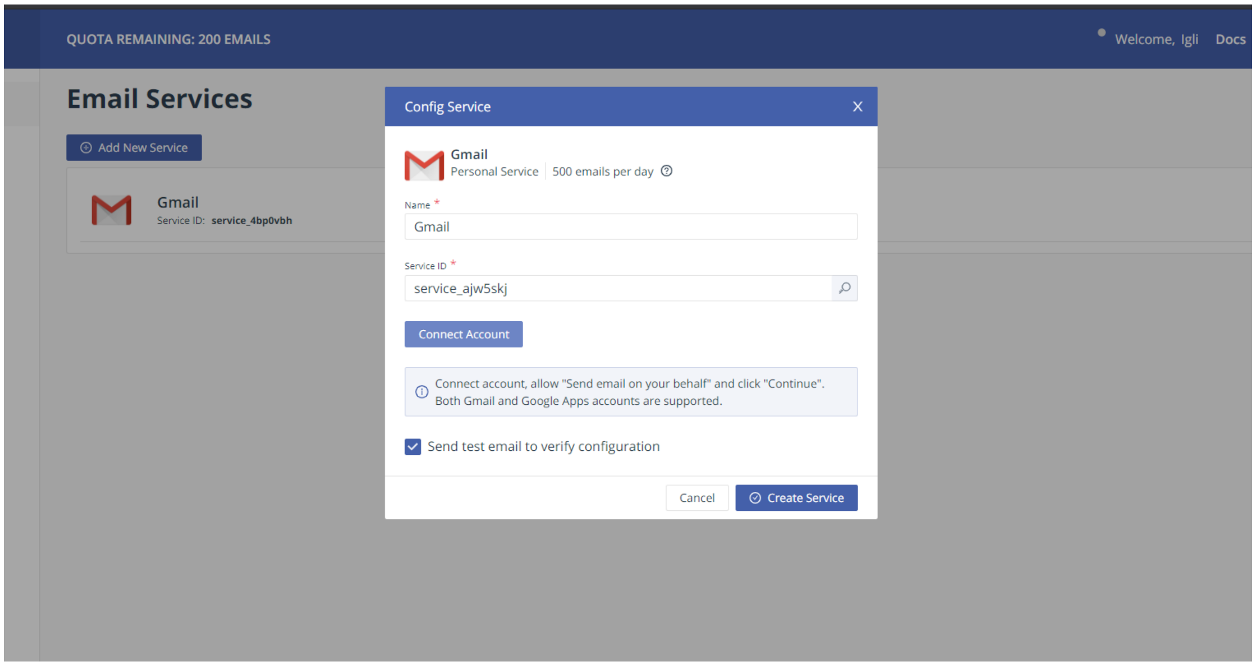
Here we have the main dashboard. You have your different email services you can setup in the middle and your navigation bar is on the side. On top you'll see a remaining quota. Using the free plan of EmailJS allows 200 monthly emails to be sent. If needed you can scale the service up by going to account and then subscriptions.



Email Services

Gmail
Service ID: service_4bp0vbh

The purpose of using EmailJS is to essentially authorize the service to have access to an email account and send emails using it. For example we can create a new service using a Gmail account. Hitting connect account will have you choose a Gmail account to sign in with. For the purposes of this project, I created a dummy account. For actual use I would recommend creating a Gmail account with a name along the lines of ServiceName.feedback@gmail.com as its easy to identify.



Email Services

Gmail
Service ID: service_4bp0vbh

Config Service

Gmail
Personal Service | 500 emails per day

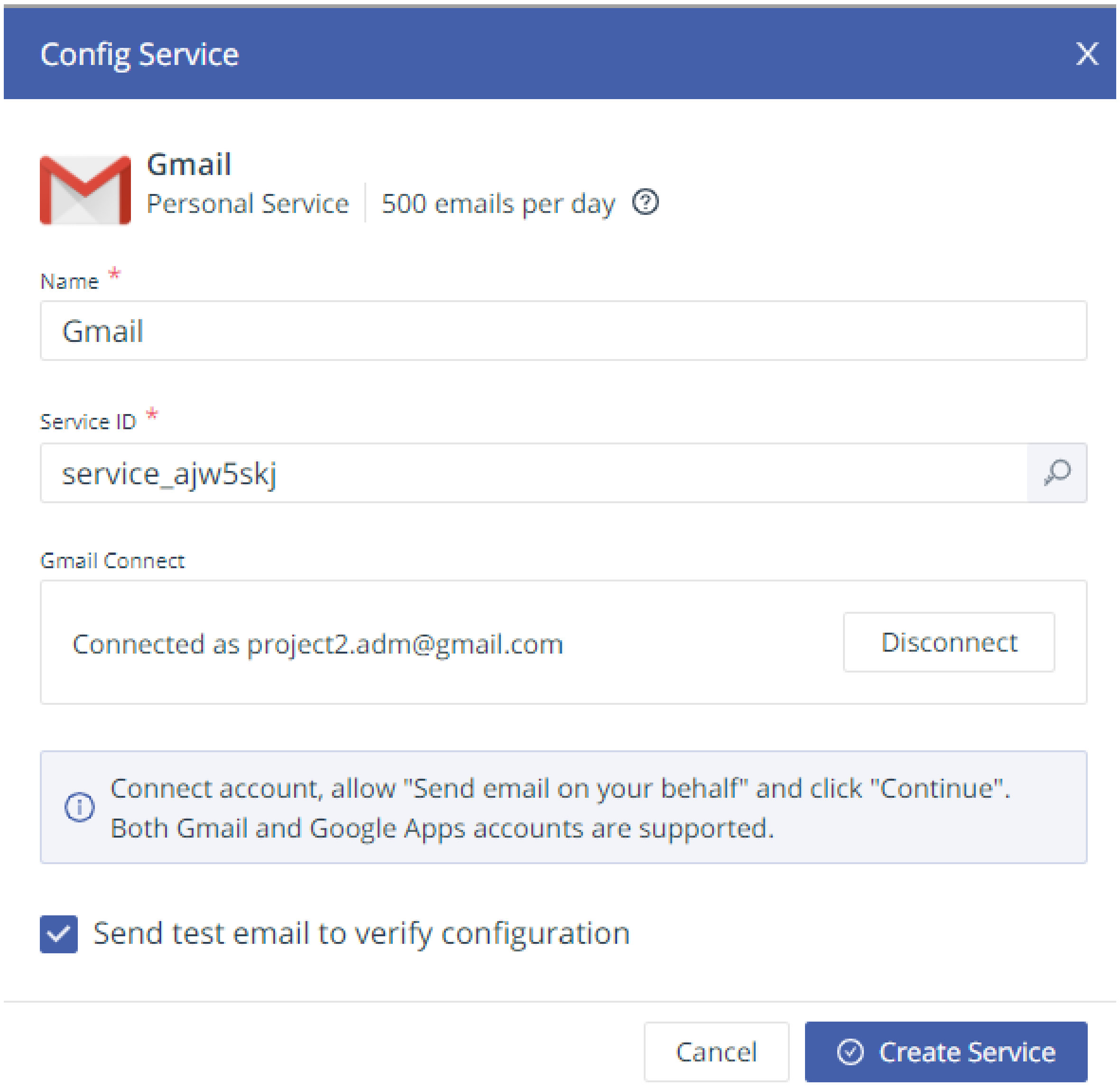
Name *
Gmail

Service ID *
service_ajw5skj

Connect account, allow "Send email on your behalf" and click "Continue". Both Gmail and Google Apps accounts are supported.

☑ Send test email to verify configuration

You can also disconnect and replace it with another email later down the road if you wish.


Config Service

Gmail
Personal Service | 500 emails per day

Name *
Gmail

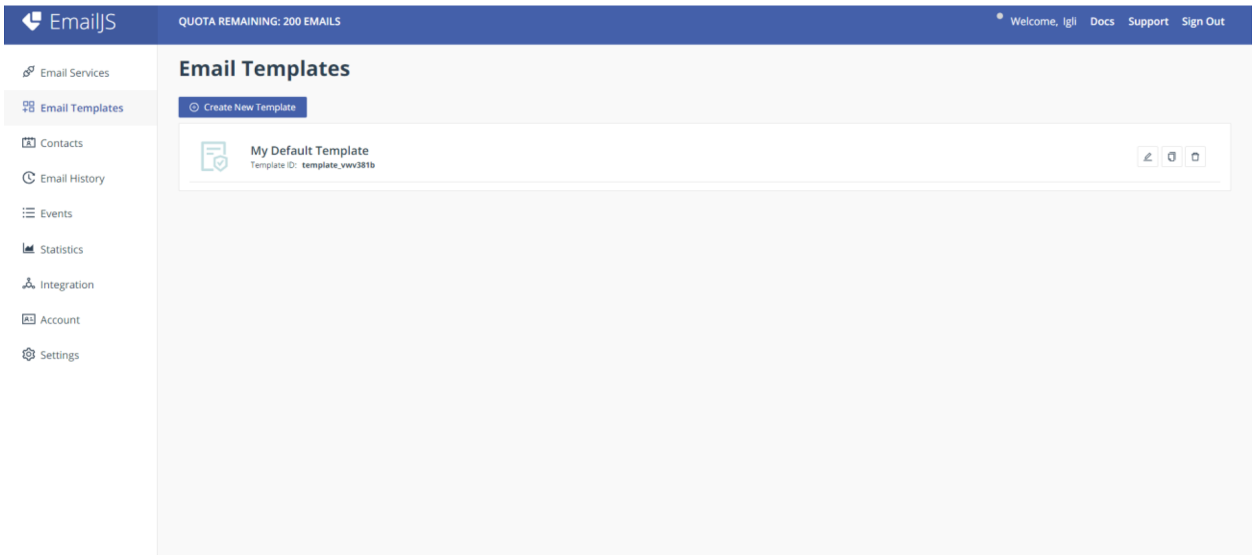
Service ID *
service_ajw5skj

Gmail Connect
Connected as project2.adm@gmail.com

Connect account, allow "Send email on your behalf" and click "Continue". Both Gmail and Google Apps accounts are supported.

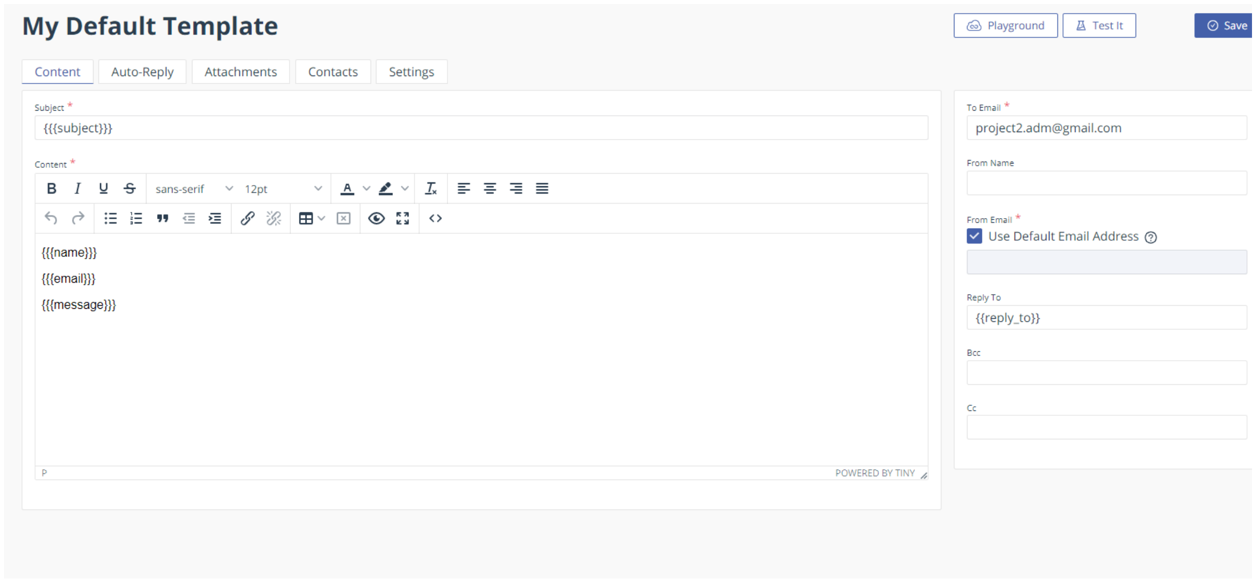
✓ Send test email to verify configuration

Next up is another feature of EmailJS, email templates.



Email Templates

My Default Template
Template ID: template_vwv381b



My Default Template

Subject *: {{{subject}}}

Content *:

{{{name}}}
{{{email}}}
{{{message}}}

To Email *: project2.adm@gmail.com

From Name:

From Email *: [✓] Use Default Email Address

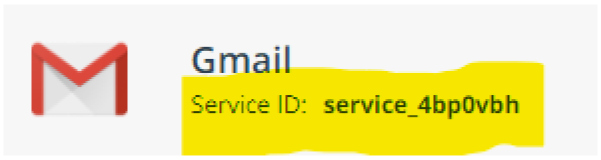
Reply To: {{reply_to}}

Bcc:

Cc:

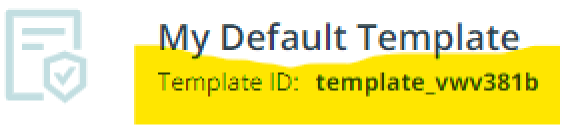
Here we can create templated emails that will pull information off a form we create. You can see the fields subject, name, email, message, and reply to that are setup. We can tie these fields to a form to automatically generate an email and send it to a desired account.

Now, to link this up to our application we need a few bits of information. That being a,

- Service id found underneath the service you setup
  - 

Gmail
Service ID: service_4bp0vbh

- Template id found underneath the template you setup

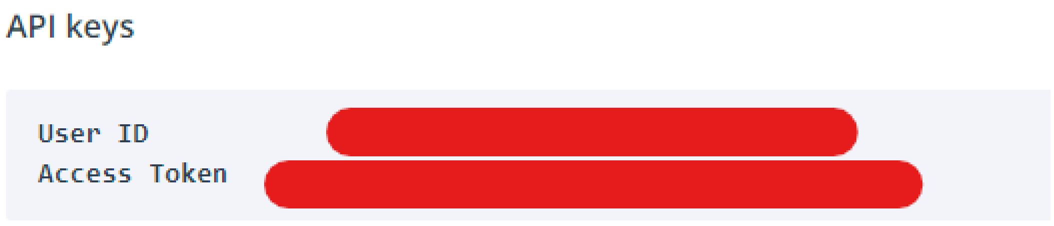


My Default Template

Template ID: template_vwv381b

    ○

- User id found in the integration tab underneath API keys.



API keys

User ID [redacted]

Access Token [redacted]

    ○

With those things we can move back to VSCode to implement our feedback form.

It's necessary to install dependencies for EmailJS within our client folder. This can be done by running the command **npm install emailjs-com**

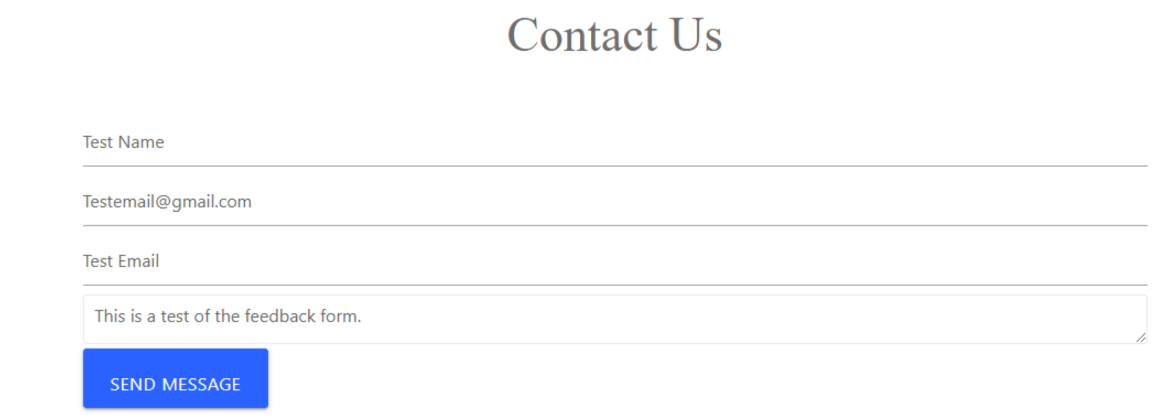
Then create a new folder inside **components** called **feedback** and a **Feedback.js** file inside it. Import emailjs and React. We'll need to create a send form event that'll reset the text fields afterwards.

```
import emailjs from "emailjs-com";
import React from 'react';

export default function Feedback() {

    function sendEmail(e) {
        e.preventDefault();

    //Replace first field with your own service id, found underneath email servic
e you create on EmailJS
    //Replace second field with your own template id, found underneath template y
ou create on EmailJS
    //Replace fourth field with your own user id, found in integrations underneat
h API keys on EmailJS

    emailjs.sendForm('service_4bp0vbh', 'template_vwv381b', e.target, 'user_EY4qi
L8ValKN3waEJEfET')
        .then((result) => {
            console.log(result.text);
        }, (error) => {
            console.log(error.text);
        });
        e.target.reset()
    }

    return(
        <div>
        <div className="row">
          <div className="col s12 center-align">
            <h3>
              <span style={{ fontFamily: "wolf2" }}> Contact Us </span>
              </h3>
              </div>
              </div>
            <div className="container">
            <form onSubmit={sendEmail}>
                    <div className="row pt-5 mx-auto">

                        {/* NAME FIELD */}
                        <div className="col-8 form-group mx-auto">
                            <input type="text" className="form-
control" placeholder="Name" name="name"/>
                        </div>

                        {/* EMAIL FIELD */}
                        <div className="col-8 form-group pt-2 mx-auto">
                            <input type="email" className="form-
control" placeholder="Email Address" name="email"/>
                        </div>

                        {/* SUBJECT FIELD */}
                        <div className="col-8 form-group pt-2 mx-auto">
                            <input type="text" className="form-
control" placeholder="Subject" name="subject"/>
                        </div>


                        {/* MESSAGE FIELD */}
                        <div className="col-8 form-group pt-2 mx-auto">
                            <textarea className="form-
control" id="" cols="30" rows="8" placeholder="Your message" name="message"></tex
tarea>
                        </div>


                        {/* SEND MESSAGE */}
                        <div className="col-8 pt-3 mx-auto">
                            <input type="submit" className="btn btn-large waves-
effect waves-light hoverable blue accent-4" value="Send Message"></input>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    )
}
```

And that's pretty much it for the feedback form. With the power of EmailJS we can having this fully working on the client side. No need to set up messy backend stuff. Now just update the App.js file by importing the feedback page and making sure its protected.

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

import jwt_decode from "jwt-decode";
import setAuthToken from "./utils/setAuthToken";
import { setCurrentUser, logoutUser } from "./actions/authActions";
import { Provider } from "react-redux";
import store from "./store";


import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register from "./components/auth/Register";
import Login from "./components/auth/Login";
import PrivateRoute from "./components/private-route/PrivateRoute";

import Dashboard from "./components/dashboard/Dashboard";
import Feedback from './components/feedback/Feedback';



// Check for token to keep user logged in
if (localStorage.jwtToken) {
  // Set auth token header auth
  const token = localStorage.jwtToken;
  setAuthToken(token);
  // Decode token and get user info and exp
  const decoded = jwt_decode(token);
  // Set user and isAuthenticated
  store.dispatch(setCurrentUser(decoded));
// Check for expired token
  const currentTime = Date.now() / 1000; // to get in milliseconds
  if (decoded.exp < currentTime) {
    // Logout user
    store.dispatch(logoutUser());
    // Redirect to login
    window.location.href = "./login";
  }
}


class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router>
          <div className="App">
            <Navbar />
            <Route exact path="/" component={Landing} />
            <Route exact path="/register" component={Register} />
            <Route exact path="/login" component={Login} />
            <Route exact path="/list" component={FilesList} />

            <Switch>
              <PrivateRoute exact path="/dashboard" component={Dashboard} />
              <PrivateRoute exact path="/feedback" component={Feedback} />
            </Switch>
          </div>
        </Router>
      </Provider>
    );
  }
}
export default App;
```

Here we can see the feedback form in action.

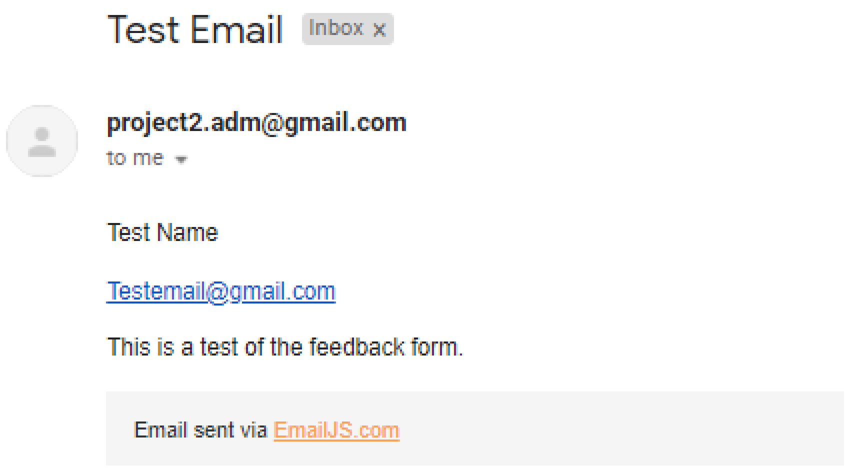

Contact Us

Test Name

Testemail@gmail.com

Test Email

This is a test of the feedback form.


Test Email

project2.adm@gmail.com
to me

Test Name

Testemail@gmail.com

This is a test of the feedback form.

Email sent via EmailJS.com

4. Tools Page

For the tools page I will have to admit that unfortunately I wasn't able to create a fully functional version of it in time. What I was able to get done is setting it up so files can be uploaded and the pathways to those files are saved on MongoDB. The issue I ran into in development were attempting to retrieve those files.

To start, similar to how we created our User Schema, we need to create a File Schema. In the **models** folder create a new file called **file.js**. This will look very similar except with different fields. Namely, we'll have a title, decription, a file path, a file mimetype, and timestamp.

```
const mongoose = require('mongoose');
const fileSchema = mongoose.Schema(
  {
    title: {
      type: String,
      required: true,
      trim: true
    },
    description: {
      type: String,
      required: true,
      trim: true
    },
    file_path: {
      type: String,
      required: true
    },
    file_mimetype: {
      type: String,
      required: true
    }
  },
  {
    timestamps: true
  }
);
const File = mongoose.model('File', fileSchema);
module.exports = File;
```

Next create a file in our config folder called **db.js** that'll contain the mongodb database we'll connect to. Similar to the previous time, replace it with your specific information.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb+srv://<username>:<password>@cluster0.syszl.mongodb.net/
file_upload?retryWrites=true&w=majority', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useCreateIndex: true
});
```

After that, create a new file in our **routes** folder called **file.js**. This is where we'll define where to store these files on the server, and limits to the file size and type as desired. We'll be importing the multer dependency for this.

```
const path = require('path');
const express = require('express');
const multer = require('multer');
const File = require('../model/file');
const Router = express.Router();
const upload = multer({


  //Store in files folder.
  storage: multer.diskStorage({
    destination(req, file, cb) {
      cb(null, './files');
    },
    filename(req, file, cb) {
      cb(null, `${new Date().getTime()}_${file.originalname}`);
    }
  }),
  limits: {
    fileSize: 5000000 // max file size 1MB = 1000000 bytes
  },

  //filter file type
  fileFilter(req, file, cb) {
    if (!file.originalname.match(/\.(jpeg|jpg|png|pdf|doc|docx|xlsx|xls|gif)$/))
{
      return cb(
        new Error(
          'only upload files with jpg, jpeg, png, pdf, doc, docx, xslx, xls forma
t.'
        )
      );
    }
    cb(undefined, true); // continue with upload
  }
});

//upload file
Router.post(
  '/upload',
  upload.single('file'),
  async (req, res) => {
    try {
      const { title, description } = req.body;
      const { path, mimetype } = req.file;
      const file = new File({
        title,
        description,
        file_path: path,
        file_mimetype: mimetype
      });
      await file.save();
      res.send('file uploaded successfully.');
    } catch (error) {
      res.status(400).send('Error while uploading file. Try again later.');
    }
  },
  (error, req, res, next) => {
    if (error) {
      res.status(500).send(error.message);
    }
  }
);
Router.get('/getAllFiles', async (req, res) => {
  try {
    const files = await File.find({});
    const sortedByCreationDate = files.sort(
      (a, b) => b.createdAt - a.createdAt
    );
    res.send(sortedByCreationDate);
  } catch (error) {
    res.status(400).send('Error while getting list of files. Try again later.');
  }
});
Router.get('/download/:id', async (req, res) => {
  try {
    const file = await File.findById(req.params.id);
    res.set({
      'Content-Type': file.file_mimetype
    });
    res.sendFile(path.join(__dirname, '..', file.file_path));
  } catch (error) {
    res.status(400).send('Error while downloading file. Try again later.');
  }
});
module.exports = Router;
```

This will setup the backend for our tools page. Next we'll be working in the frontend to create a tools page to upload files and a file list to browse uploaded files.

In the **dashboard** folder, create a new file called **Tools.js**. We'll be importing the dropzone dependency.

```
import React, { useState, useRef } from 'react';
import { Form, Row, Col, Button } from 'react-bootstrap';
import { Link } from "react-router-dom";
import Dropzone from 'react-dropzone';
import axios from 'axios';

import { API_URL } from '../../utils/constants';


const Tools = (props) => {
  const [file, setFile] = useState(null); // state for storing actual image
  const [previewSrc, setPreviewSrc] = useState(''); // state for storing previewI
mage
  const [state, setState] = useState({
    title: '',
    description: ''
  });
  const [errorMsg, setErrorMsg] = useState('');
  const [isPreviewAvailable, setIsPreviewAvailable] = useState(false); // state t
o show preview only for images
  const dropRef = useRef(); // React ref for managing the hover state of droppabl
e area
  const handleInputChange = (event) => {
    setState({
      ...state,
      [event.target.name]: event.target.value
    });
  };

  const onDrop = (files) => {
    const [uploadedFile] = files;
    setFile(uploadedFile);
    const fileReader = new FileReader();
    fileReader.onload = () => {
      setPreviewSrc(fileReader.result);
    };
    fileReader.readAsDataURL(uploadedFile);
    setIsPreviewAvailable(uploadedFile.name.match(/\.(jpeg|jpg|png)$/));

    dropRef.current.style.border = '2px dashed #e9ebeb';

  };

  const updateBorder = (dragState) => {
    if (dragState === 'over') {
      dropRef.current.style.border = '2px solid #000';
    } else if (dragState === 'leave') {
```

```
    dropRef.current.style.border = '2px dashed #e9ebeb';
  }
};

const handleOnSubmit = async (event) => {
  event.preventDefault();
  try {
    const { title, description } = state;
    if (title.trim() !== '' && description.trim() !== '') {
      if (file) {
        const formData = new FormData();
        formData.append('file', file);
        formData.append('title', title);
        formData.append('description', description);
        setErrorMsg('');
        await axios.post(`${API_URL}/upload`, formData, {
          headers: {
            'Content-Type': 'multipart/form-data'
          }
        });
        props.history.push('/list'); // add this line

      } else {
        setErrorMsg('Please select a file to add.');
      }
    } else {
      setErrorMsg('Please enter all the field values.');
    }
  } catch (error) {
    error.response && setErrorMsg(error.response.data);
  }
};

return (
  <React.Fragment>
    <Form className="search-form" onSubmit={handleOnSubmit}>
      {errorMsg && <p className="errorMsg">{errorMsg}</p>}
      <h4>File upload</h4>
      <p className="grey-text text-darken-1">
            Click <Link to="/list">here</Link> to view list of uploaded files.
            </p>
      <Row>
        <Col>
          <Form.Group controlId="title">
            <Form.Control
```

```
                type="text"
                name="title"
                value={state.title || ''}
                placeholder="Enter title"
                onChange={handleInputChange}
              />
            </Form.Group>
          </Col>
        </Row>
        <Row>
          <Col>
            <Form.Group controlId="description">
              <Form.Control
                type="text"
                name="description"
                value={state.description || ''}
                placeholder="Enter description"
                onChange={handleInputChange}
              />
            </Form.Group>
          </Col>
        </Row>

        <div className="upload-section">
        <Dropzone
            onDrop={onDrop}
            onDragEnter={() => updateBorder('over')}
            onDragLeave={() => updateBorder('leave')}
        >
            {({ getRootProps, getInputProps }) => (
            <div {...getRootProps({ className: 'drop-zone' })} ref={dropRef}>
                <input {...getInputProps()} />
                <p>Drag and drop a file OR click here to select a file</p>
                {file && (
                <div>
                    <strong>Selected file:</strong> {file.name}
                </div>
                )}
            </div>
            )}
        </Dropzone>
        {previewSrc ? (
            isPreviewAvailable ? (
            <div className="image-preview">
                <img className="preview-image" src={previewSrc} alt="Preview" />
            </div>
            ) : (
            <div className="preview-message">
                <p>No preview available for this file</p>
            </div>
            )
        ) : (
            <div className="preview-message">
            <p>Image preview will be shown here after selection</p>
            </div>
        )}
        </div>


        <Button variant="primary" type="submit"  className="btn btn-large waves-
effect waves-light hoverable blue accent-4" >
          Submit
        </Button>

      </Form>
    </React.Fragment>
  );
};
export default Tools;
```

After this create a new file in the **components** folder called **FilesList.js**

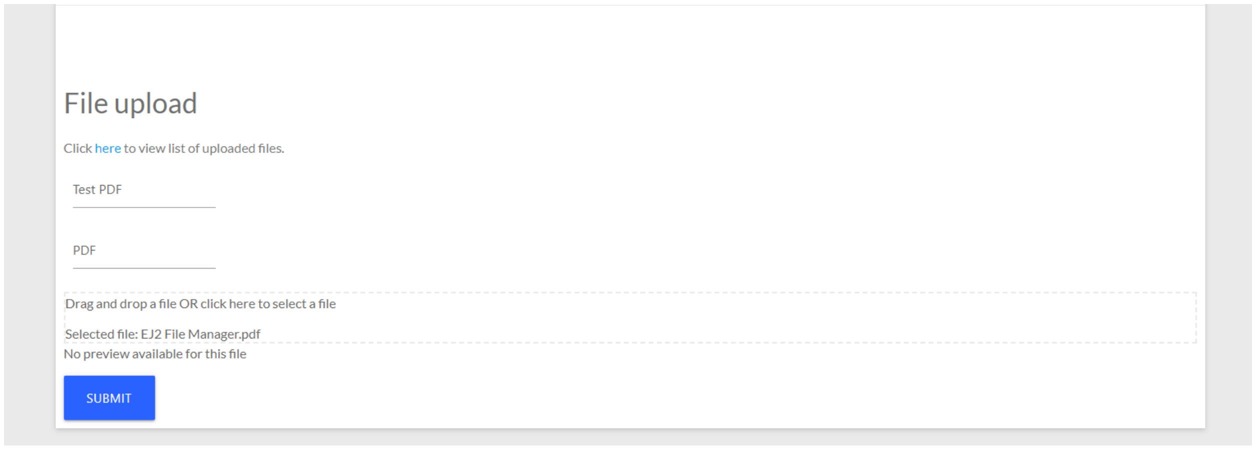
```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { API_URL } from '../utils/constants';
const FilesList = () => {
  const [filesList, setFilesList] = useState([]);
  const [errorMsg, setErrorMsg] = useState('');
  useEffect(() => {
    const getFilesList = async () => {
      try {
        const { data } = await axios.get(`${API_URL}/getAllFiles`);
        setErrorMsg('');
        setFilesList(data);
      } catch (error) {
        error.response && setErrorMsg(error.response.data);
      }
    };
    getFilesList();
  }, []);
  const downloadFile = async (id, path, mimetype) => {
    try {
      const result = await axios.get(`${API_URL}/download/${id}`, {
        responseType: 'blob'
      });
      const split = path.split('/');
      const filename = split[split.length - 1];
      setErrorMsg('');
      return download(result.data, filename, mimetype);
    } catch (error) {
      if (error.response && error.response.status === 400) {
        setErrorMsg('Error while downloading file. Try again later');
      }
    }
  };
  return (
    <div className="files-container">
      {errorMsg && <p className="errorMsg">{errorMsg}</p>}
      <table className="files-table">
        <thead>
          <tr>
            <th>Title</th>
            <th>Description</th>
            <th>Download File</th>
          </tr>
        </thead>
        <tbody>
          {filesList.length > 0 ? (
            filesList.map(
              ({ _id, title, description, file_path, file_mimetype }) => (
                <tr key={_id}>
                  <td className="file-title">{title}</td>
                  <td className="file-description">{description}</td>
                  <td>
                    <a
                      href="#/"
                      onClick={() =>
                        downloadFile(_id, file_path, file_mimetype)
                      }
                    >
                      Download
                    </a>
                  </td>
                </tr>
              )
            )
          ) : (
            <tr>
              <td colSpan={3} style={{ fontWeight: '300' }}>
                No files found. Please add some.
              </td>
            </tr>
          )}
        </tbody>
      </table>
    </div>
  );
};
export default FilesList;
```

And that’s as far as I got. Last is to update **App.js** with the new webpages and imports. The final version of **App.js** that I’m leaving off with is this.

```

import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

import jwt_decode from "jwt-decode";
import setAuthToken from "./utils/setAuthToken";
import { setCurrentUser, logoutUser } from "./actions/authActions";
import { Provider } from "react-redux";
import store from "./store";

import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register from "./components/auth/Register";
import Login from "./components/auth/Login";
import PrivateRoute from "./components/private-route/PrivateRoute";
import Dashboard from "./components/dashboard/Dashboard";
import Feedback from './components/feedback/Feedback';


import Tools from "./components/dashboard/Tools";
import FilesList from "./components/FilesList";



// Check for token to keep user logged in
```

```
if (localStorage.jwtToken) {
  // Set auth token header auth
  const token = localStorage.jwtToken;
  setAuthToken(token);
  // Decode token and get user info and exp
  const decoded = jwt_decode(token);
  // Set user and isAuthenticated
  store.dispatch(setCurrentUser(decoded));
// Check for expired token
  const currentTime = Date.now() / 1000; // to get in milliseconds
  if (decoded.exp < currentTime) {
    // Logout user
    store.dispatch(logoutUser());
    // Redirect to login
    window.location.href = "./login";
  }
}
class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router>
          <div className="App">
            <Navbar />
            <Route exact path="/" component={Landing} />
            <Route exact path="/register" component={Register} />
            <Route exact path="/login" component={Login} />
            <Route exact path="/list" component={FilesList} />

            <Switch>
              <PrivateRoute exact path="/dashboard" component={Dashboard} />
              <PrivateRoute exact path="/feedback" component={Feedback} />
              <PrivateRoute exact path="/tools" component={Tools} />


            </Switch>
          </div>
        </Router>
      </Provider>
    );
  }
}
export default App;
```

Here is the current tools page. Users can upload files to be stored on the server.



File upload

Click here to view list of uploaded files.

Test PDF

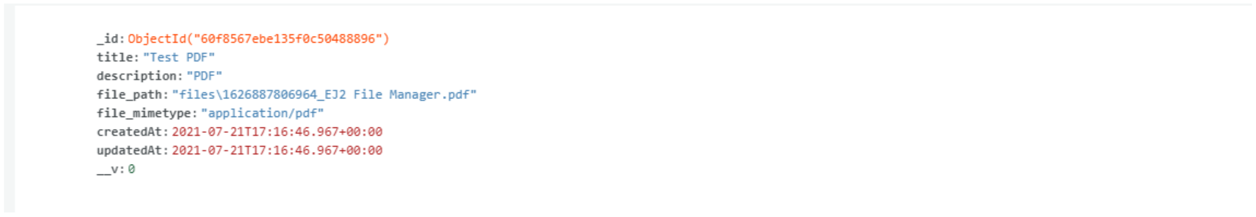
PDF

Drag and drop a file OR click here to select a file

Selected file: EJ2 File Manager.pdf

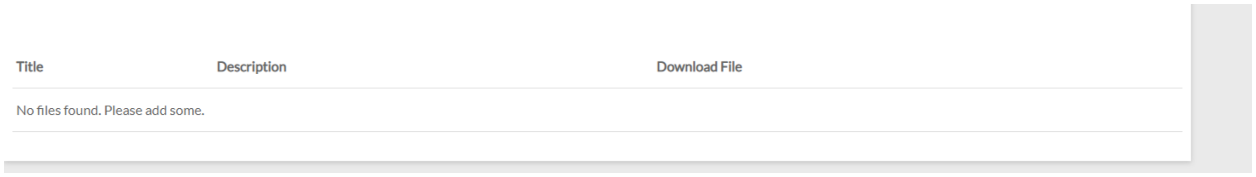
No preview available for this file

The file schema is saved onto the database.



```
_id: ObjectId("60f8567ebe135f0c50488896")
title: "Test PDF"
description: "PDF"
file_path: "files\1626887806964_EJ2 File Manager.pdf"
file_mimetype: "application/pdf"
createdAt: 2021-07-21T17:16:46.967+00:00
updatedAt: 2021-07-21T17:16:46.967+00:00
__v: 0
```

But I couldn't get it to display on the files list page.



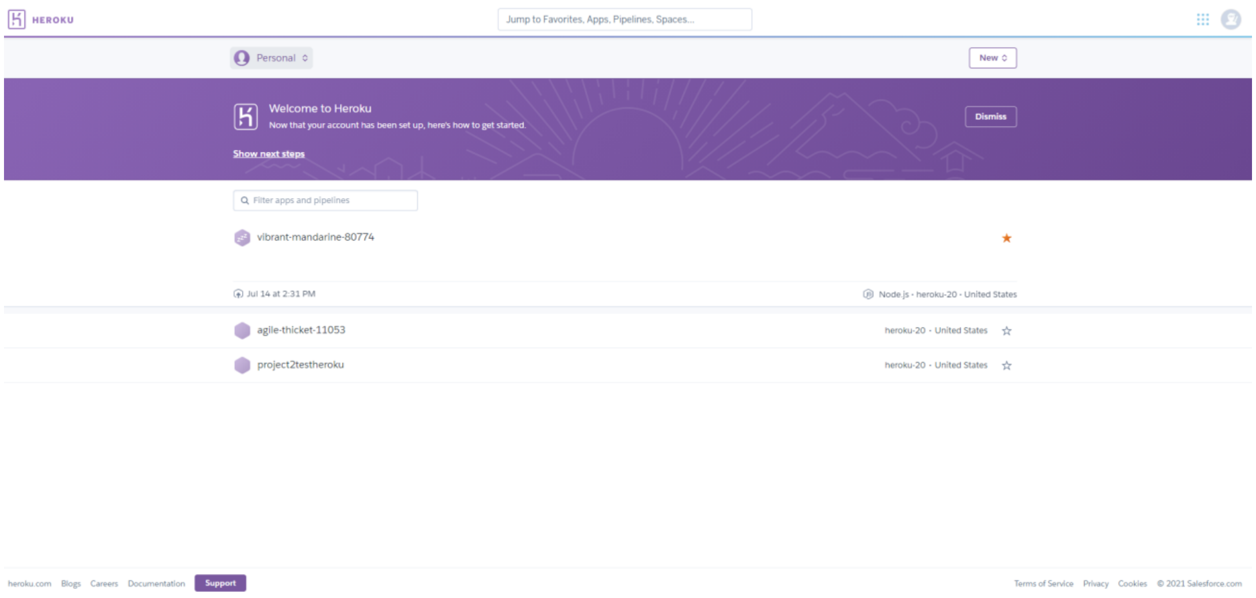
Title	Description	Download File
No files found. Please add some.		

5. Pushing to Heroku

In this section we will be going over on getting the application hosted on the web using Heroku as a host. If you haven't already, create an account with Heroku.

Here we have the dashboard where you Heroku applications will show. To start, click show next steps and go through this help guide.

https://devcenter.heroku.com/articles/getting-started-with-nodejs



Welcome to Heroku
Now that your account has been set up, here's how to get started.

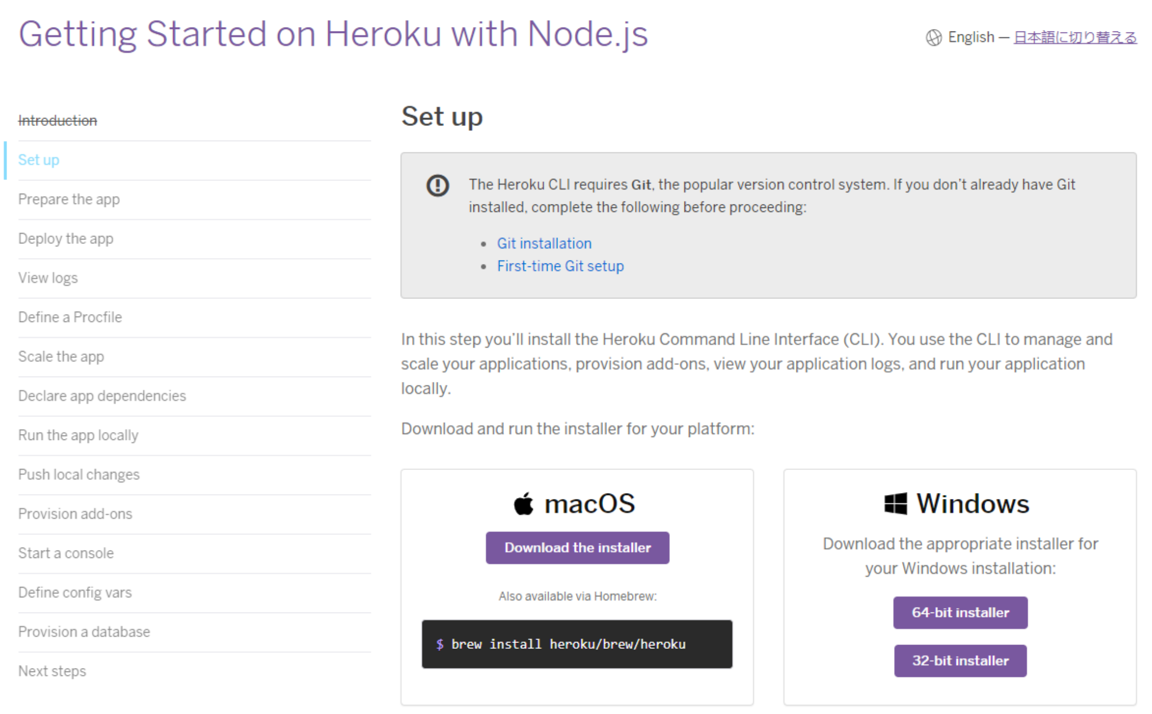
vibrant-mandarine-80774
Jul 14 at 2:31 PM   Node.js · heroku-20 · United States
agile-thicket-11053   heroku-20 · United States
project2testheroku   heroku-20 · United States

There's a lot that can go wrong when deploying your application to Heroku. Thankfully Heroku has an extensive guide on how to set up and deploy an application that uses nodejs.

To start, download Heroku onto your computer and follow the setup process. In addition to Heroku you will also need to install Git.

https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git



Getting Started on Heroku with Node.js

English — 日本語に切り替える

~~Introduction~~
Set up
Prepare the app
Deploy the app
View logs
Define a Procfile
Scale the app
Declare app dependencies
Run the app locally
Push local changes
Provision add-ons
Start a console
Define config vars
Provision a database
Next steps

Set up

The Heroku CLI requires **Git**, the popular version control system. If you don't already have Git installed, complete the following before proceeding:

- Git installation
- First-time Git setup

In this step you'll install the Heroku Command Line Interface (CLI). You use the CLI to manage and scale your applications, provision add-ons, view your application logs, and run your application locally.

Download and run the installer for your platform:

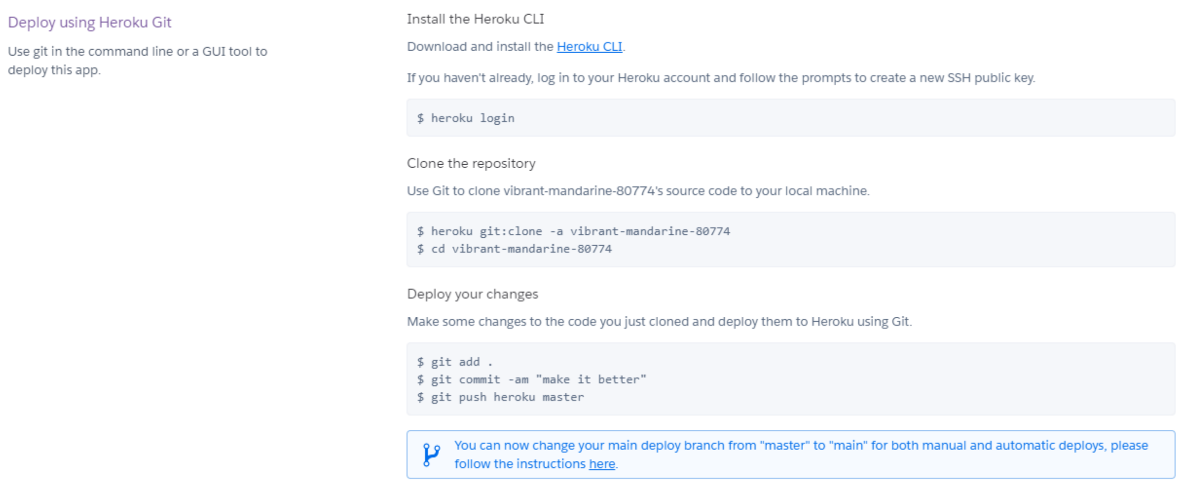
```
$ brew install heroku/brew/heroku
```

Windows
Download the appropriate installer for your Windows installation:
64-bit installer
32-bit installer

Once you have Heroku and Git setup run the command **heroku login** to log into the heroku cli.

Also make sure to check your git version with **git --version** to make sure you installed it correctly.

Afterwards you can create a new heroku app with **heroku create** and it'll generate a random url for you to use. Now looking at the Heroku website you can see the app that was created on the dashboard. Go to it and click on the deploy tab where you can see how to deploy it online using Heroku Git. If it fails its time to troubleshoot and repeat the process. This can be arduous as you develop but if the build fails the build log generated will show the step at which it failed.



Deploy using Heroku Git

Use git in the command line or a GUI tool to deploy this app.

Install the Heroku CLI

Download and install the Heroku CLI.

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Clone the repository

Use Git to clone vibrant-mandarine-80774's source code to your local machine.

```
$ heroku git:clone -a vibrant-mandarine-80774
$ cd vibrant-mandarine-80774
```

Deploy your changes

Make some changes to the code you just cloned and deploy them to Heroku using Git.

```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

You can now change your main deploy branch from "master" to "main" for both manual and automatic deploys, please follow the instructions here.

6. Where to go from here

In the past 10 weeks this is what I was able to accomplish. I'd say roughly 5 weeks were spend learning MERN Stack development. This comprised of going through free courses. Doing small projects. And following along on online tutorials each week and trying to learn something useful that I can apply to the project. The other 5 weeks were spent developing this application. To sum up this application currently has the following.

- Working Login/Registration System
  - Users can register using a form, registration information is then passed to the server and stored onto a database so that they can log in.
  - Authorization tokens to remember the user's session.
  - Password hashing via bcrypt.
  - Ability to manually set admin accounts via MongoDB Atlas
  - Protected routing so that unauthorized users are redirected elsewhere.
- Working Feedback System
  - Set up an email account to connect to EmailJS
  - Users can fill out a form that uses EmailJS's templates to pull data from the forms.
  - Emails can be redirected to an email address as desired.
- Semi functional Tools page
  - Files can be uploaded and stored on the server. Information regarding the file name, description, path and mime type is stored onto a database.
  - The FilesList page still needs to be fixed to then retrieve and display files.
- Web Hosted
  - In its current state the website is able to be hosted online using Heroku

Along with that there's also some features from the original statement of work that are missing. Currently there is no payment form for user registration. I did begin looking into this during development but it was not a priority at the time. Based off what I've seen and if I have more time I would attempt to create this form using Web Payments by Square.

https://developer.squareup.com/docs/web-payments/overview

https://www.npmjs.com/package/react-square-web-payments-sdk

In addition to that, while I've set up a foundation for administrative roles there is currently no robust tools to support that. I had features such as user management, changing credentials, deleting accounts in mind but couldn't get to it in time.

Other features I had ideas for and felt they would be beneficial for the website

- User profile page
    - Manage account
        - Subscription
        - Change email
        - Change password
        - Two-Factor Authentication