MERN Base App

These are the step by step instructions for creating a base MERN App with the main functionality. Once the steps are completed you will have a full stack MERN App with authentication using JWT, Firebase and Google OAuth and all the signin/signup, delete and update functionality for users with the image storage in firebase and the database in mongoDB.You can follow the instructions in the root directory or just clone the git repo and have all the basic functionality ready to go for your next project.

Installation

1. For your front end, Install React using the build tool VITE in a sub folder of the root folder and call it client.
2. Create the main app folder and inside *npm create vite@latest client* and select React from the drop down and then JavaScript + SWC for fast loading. CD into client and install node modules
3. Install Tailwind CSS search tailwind vite for instructions
4. Clean up unnecessary files. Delete App.css public/vite.svg and assets/react.svg and delete everything inside App.jsx and replace with rfc. Now you can run the app using npm run dev and check if tailwind is working.
5. Add the root not the client project to a github repository so we can use the git repository for both client and api and deploy the whole app to production later on.

Creating pages and routes

Create folder src/pages and in there create each page with an rfc (react functional component) eg. About.jsx Signin.jsx .We don't need to import React as we are importing it in App.jsx before we import all our pages.

```
export default function About() {
  return <div>About</div>;
}
```

To create the routes for these pages we have to install React Router DOM
npm install react-router-dom in the client folder.
Then in App.jsx we need to import { BrowserRouter, Routes, Route } from 'react-router-dom'; and import every page we wish to use in our routes.

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
```

inside the App function wrap everything inside BrowserRouter

```
<BrowserRouter>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/profile" element={<Profile />} />
        <Route path="/signin" element={<SignIn />} />
        <Route path="/signout" element={<SignOut />} />
      </Routes>
    </BrowserRouter>
```

Create Header for all pages

In src create a components folder src/components and add a Header.jsx with an rfc inside then add the header component to the BrowserRouter outside of Routes so it will be there on all pages

```
    <BrowserRouter>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/profile" element={<Profile />} />
        <Route path="/signin" element={<SignIn />} />
        <Route path="/signout" element={<SignOut />} />
      </Routes>
    </BrowserRouter>
```

Make your header using html and tailwind css. You'll need to install react-icons to have some icons to use like the magnifying glass in the search box
npm install react-icons
Then import the icons you need in each page, fa stands for font-awesome
https://react-icons.github.io/react-icons/icons/fa/

```
import { FaSearch } from "react-icons/fa";
```

```
<form className="bg-slate-100 p-3 rounded-lg flex items-center">
```

```
          <input
            type="text"
            placeholder="Search...."
            className="focus:outline-none w-24 sm:w-64"
          />
          <FaSearch className="text-slate-600" />
        </form>
```

We also need to import Link from react-router-dom and wrap any elements which we want to link to other pages. Link allows browsing from one page to another without refreshing the page.

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

export default function Header() {
  return (
    <header className="bg-slate-200 shadow-md">
      <div className="flex justify-between items-center
max-w-6xl mx-auto p-3">
        <Link to="/">
          <h1 className="font-bold text-sm sm:text-xl flex
flex-wrap">
            <span className="text-slate-500">Recipe</span>
            <span className="text-slate-700">App</span>
          </h1>
        </Link>
```

Make our api for backend

We will be using Express.js for our web application framework, read the installation guide here https://expressjs.com/en/starter/installing.html

Create a folder in the root of our app called api

name_of_root/api

Create a package.json file in the root folder, this is where your hosting platform (render) will look for the backend package.json

npm init -y

The structure will be like this
recipe_app
recipe_app/api
recipe_app/client
recipe_app/node_modules
recipe_app/.gitignore
recipe_app/package.json

If you look in the package.json the main file it is looking for is index.js

```
"main": "index.js",
```

Make this file in the api folder (api/index.js)
touch index.js
Now install express in the root folder and check the package.json to see if it has been added to the dependencies.
npm install express
Open the index.js and add basic express code from express docs

```
import express from 'express';
```

```
const app = express()
const port = 3000

app.get('/', (req, res) => {
   res.send('Hello World!')
})

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

Run the file node api/index.js which will give an error of module syntax was detected because you use an import without adding "type": "module" to your package.json file

```
"type": "module",
```

so add it now and run the file again. Now you should get a console log of Express Server running on port 3000

Add nodemon to restart the server automatically during development. In the root folder run

npm install nodemon --save-dev

then add the script to run it in the package.json

```
 "scripts": {
    "dev": "nodemon api/index.js",
    "start": "node api/index.js"
  },
```

The reason for the "start" script is that when the app is deployed in production your hosting platform (render) will look for the script to start the backend.
Now you can run the backend in development using npm run dev and add to git

Install and create our database MongoDB with Mongoose

If you don't have a mongdb account create a free one here https://www.mongodb.com/

We will use Mongoose with MongoDB for easier access. Install mongoose in the root folder by running

npm install mongoose

then import mongoose in your index.js
Set up a database in mongodb and put them in a .env file as below.
Use dotenv to hide your database passwords and connection string

npm install dotenv

then create a .env file in the root folder and add your environment variables

```
MONGO_URI=mongodb+srv://your_user_name:**password**@cluster0.bnt
bfci.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0
JWT_SECRET=df;kdfkkdf**********fj943r0
```

Now create an example.env for anyone cloning your repo and add .env to .gitignore so that it is not saved to your git repo with your passwords
Connect to the database and check for errors

```
import 'dotenv/config';
import express from 'express';
import mongoose from 'mongoose';
import userRouter from './routes/user.route.js';

mongoose.connect(process.env.MONGO_URI)
```

```
   .then(() => {
      console.log('Connected to MongoDB')
   })
   .catch((err) => {
      console.log(err);
   })
```

Create user models for database

Inside the api folder create a models folder, we will use a naming convention like user.models.js to name our models. Create user.models.js and add schema for a user with a timestamp.

```
import mongoose from "mongoose";

// create the schema
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
    },
    email: {
        type: String,
        required: true,
        unique: true,
    },
    password: {
        type: String,
        required: true,
    }
},{ timestamps: true });
```

Create the user model and use the schema above and export as default

```
const User = mongoose.model('User', userSchema);

export default User;
```

Create a test api route

Inside the api folder create a folder called routes and inside that create a file called user.route.js This file will create and export the express router for our desired route. We will also create a controllers folder and place all our route controllers in here. user.route.js will have the code

```
import express from 'express';
import { test } from '../controllers/user.controller.js';

// create a router
const router = express.Router();

router.get('/', test)

export default router;
```

And api/controllers/user.controller.js will have the code

```
export const test = (req, res) => {
    res.json({
        message: 'Hello World! from
controllers/user.controller.js'
    })
}
```

We have to import the route from our routes/user.route.js into index.js and use app.use() to add the routes we want

```
import 'dotenv/config';
import express from 'express';
import mongoose from 'mongoose';
import userRouter from './routes/user.route.js';
```

```
mongoose.connect(process.env.MONGO_URI)
   .then(() => {
      console.log('Connected to MongoDB')
   })
   .catch((err) => {
      console.log(err);
   })


const app = express()
const port = 3000

 app.use("/api/user", userRouter)
```

Now when we go to the port that the express server is running on (in our case 3000) open browser window at localhost:3000/api/user we will get the json returned from our controller user.controller.js

```
{
        message: 'Hello World! from
controllers/user.controller.js'
    }
```

Create the signup api route

We need to get the user information like user name, password and store these in the database while hashing the password.
Create a new route called auth.route.js ***routes/auth.route.js***

```
import express from 'express';
import { signup } from '../controllers/auth.controller.js';

const router = express.Router();

router.post('/signup', signup);
```

```
export default router;
```

Create a new controller called auth.controller.js ***controllers/auth.controller.js***

```
import User from '../models/user.model.js';

export const signup = async (req, res) => {
    const {username, email, password } = req.body;
    const newUser = new User({username, email, password});

    // save to the database
    await newUser.save();
    res.status(201).json("User created successfully");

}
```

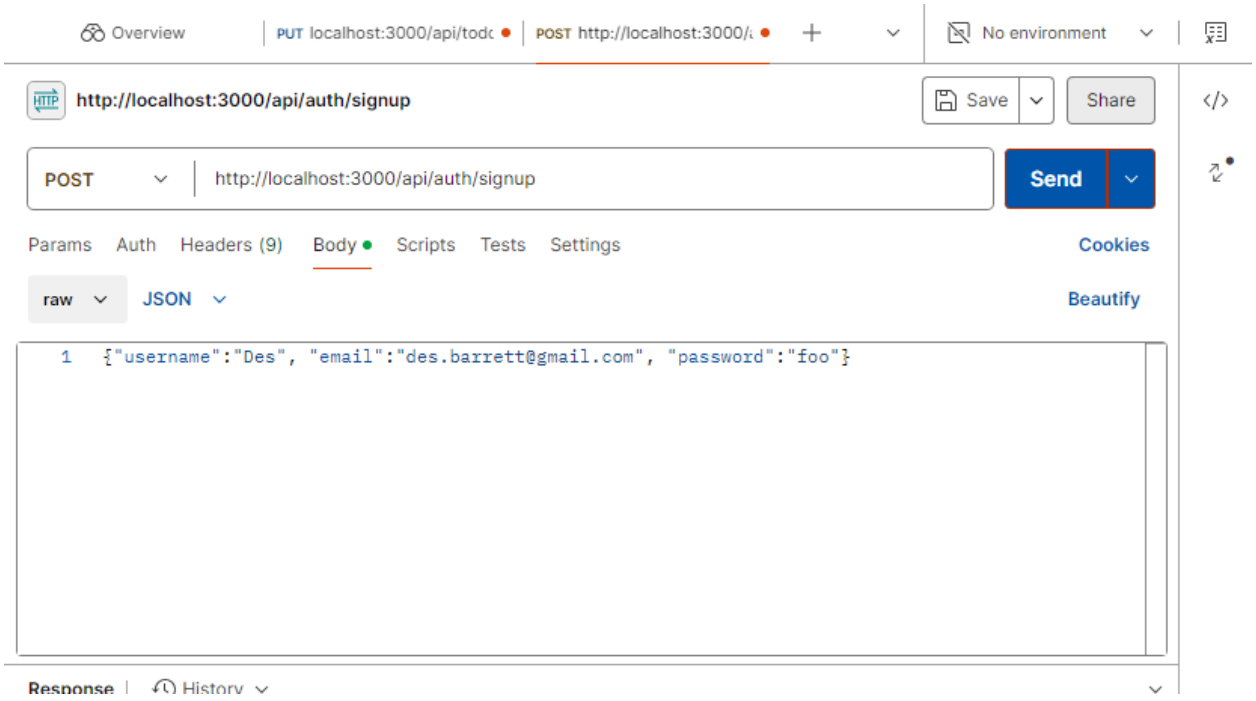
In the index.js import your auth.route.js and use it for the route '/api/auth'

```
import authRouter from './routes/auth.route.js';
 app.use('/api/auth', authRouter);
```

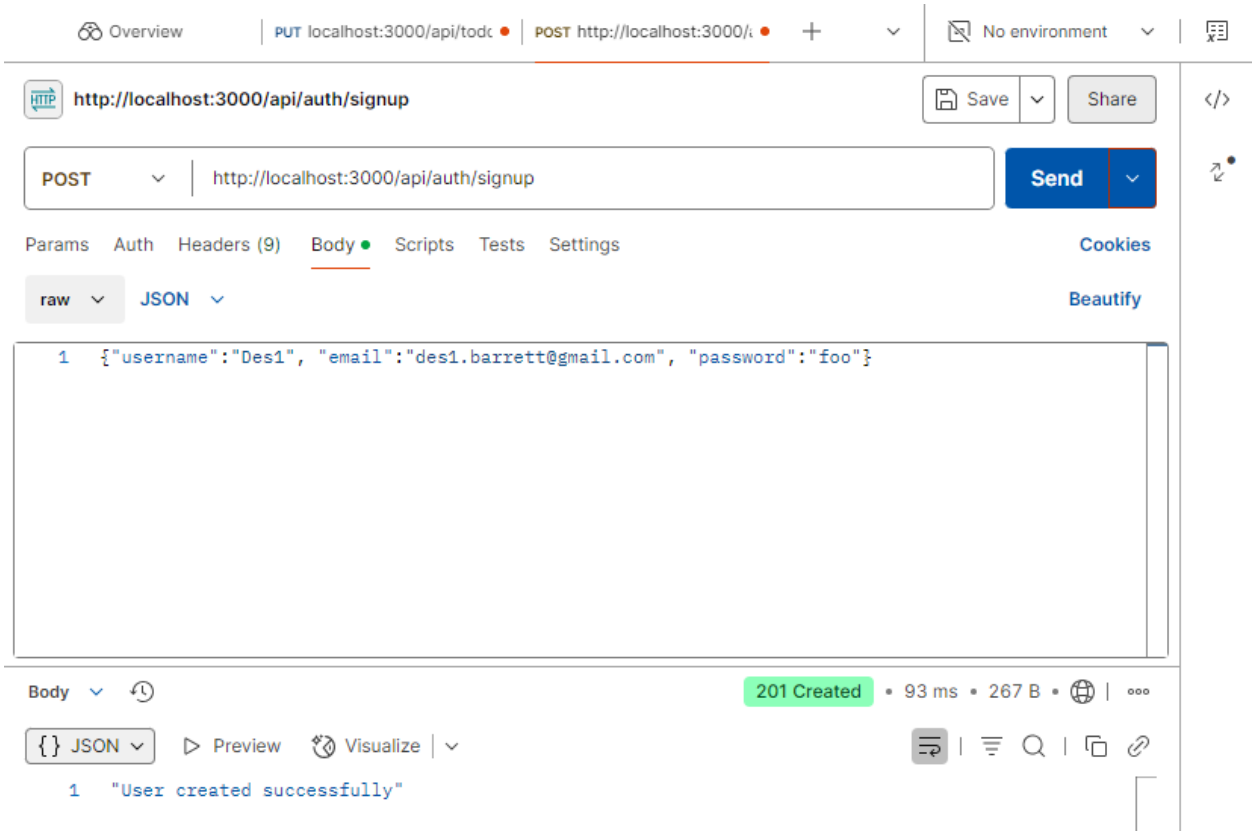
If we try to look in req.body now we will get undefined from the server and that is because by default the server does not accept json. To fix this we have to make it allow json by modifying the index.js file with the line

```
// allow server to accept json
app.use(express.json());
```

Now you can test the req.body response by sending a post request to api/auth/signup with a json of {"username": "John", "email":"john@gamil.com", "password": "foo"}
Use an api tester like postman to send it to http://localhost:3000/api/auth/signup. Click the body tab and select raw and JSON to send json.



Check to see if you get a 201 created from the api server after sending it



But we don't want to save the password as a string that anybody can see. We need to hash it using bcryptjs ***npm install bcryptjs*** and once installed use hashSync to hash and add salt to

the password and then store that in the database import it into auth.controller.js and update the auth.controller.js

```
import User from '../models/user.model.js';
import bcrypt from 'bcryptjs';
export const signup = async (req, res) => {
    const {username, email, password } = req.body;
    const hashedPassword = bcrypt.hashSync(password, 10);
    const newUser = new User({username, email, password:
hashedPassword});

    // save to the database
    await newUser.save();
    res.status(201).json("User created successfully");
}
```

Create middleware and function to handle any possible errors

Inside in index.js add middleware code to send json on error

```
// middleware for handling errors
app.use((err, req, res, next) => {
   const statusCode = err.statusCode || 500;
   const message = err.message || 'Internal Server Error';
   return res.status(statusCode).json({
      success: false,
      statusCode,
      message
   })
 });
```

Now we can add a try catch in our signup function and pass it to the middleware for handling errors

```
export const signup = async (req, res, next) => {
  const { username, email, password } = req.body;
  const hashedPassword = bcrypt.hashSync(password, 10);
  const newUser = new User({ username, email, password:
hashedPassword });
```

```
  // catch any errors while trying to create user like dublicate
email etc.
  try {
    // save new user to the database
    await newUser.save();
    res.status(201).json("User created successfully");
  } catch (error) {
    console.log(error);
    // use middleware for handling errors in index.js
    next(error);
  }
};
```

Create a new folder called utils in our api folder and add file error.js with our error handler. We will use this function later to handle custom errors like password too short.

```
export const errorHandler = (statusCode, message ) => {
    const error = new Error();
    error.statusCode = statusCode;
    error.message = message;
    return error;
```

Create UI for our Sign Up page

Add a background color to your body in index.css

```
@import 'tailwindcss';

body {
    background-color: rgb(241, 245,241 );
}
```

Open ***client/pages/signin.jsx*** and add the following code for your signup form

```
import React from "react";

export default function SignIn() {
  return (
    <div className=" p-3 max-w-lg mx-auto">
      <h1 className="text-3xl font-semibold my-7
text-center">Sign Up</h1>
```

```
      <form className="flex flex-col mx-2 gap-4">
        <input
          type="text"
          placeholder="username"
          className="border p-3 rounded-lg bg-white"
          id="username"
        />
        <input
          type="email"
          placeholder="email"
          className="border p-3 rounded-lg bg-white"
          id="email"
        />
        <input
          type="password"
          placeholder="password"
          className="border p-3 rounded-lg bg-white"
          id="passwordame"
        />
      </form>
    </div>
  );
}
```

Now add your Sign Up button

```
<button className="bg-slate-700 text-white p-3 rounded-lg
uppercase hover:opacity-95 disabled:opacity-80">
          Sign up
        </button>
```

Import useState and then add an onChange event listener to each input for catching changes to the form. formData will hold all changes made.

```
export default function SignIn() {
  const [formData, setFormData] = useState({});
  const handleChange = (e) => {
```

```
    setFormData({
      ...formData,
      // whatever is changing set to that value
      [e.target.id]: e.target.value,
    });
  };
  console.log(formData);
  return (
    <div className=" p-3 max-w-lg mx-auto">
      <h1 className="text-3xl font-semibold my-7
text-center">Sign Up</h1>
      <form className="flex flex-col mx-2 gap-4">
        <input
          type="text"
          placeholder="username"
          className="border p-3 rounded-lg bg-white"
          id="username"
          onChange={handleChange}
        />
```

Once we have the data in formData we want to submit it using an onSubmit event listener calling handleSubmit function. Add this function to the Signin.jsx

```
const handleSubmit = async (e) => {
    e.preventDefault();
    // in order to use the url '/api/auth/signup' from the
client side
    // to the api side we need to use a proxy configured in
vite.config.js
    const res = await fetch("/api/auth/signup", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(formData),
    });
```

```
    const data = await res.json();
    console.log(data);
  };
  return (
```

Alson add the onSubmit to the form element

```
<form onSubmit={handleSubmit} className="flex flex-col mx-2
gap-4">
```

In order to use the url `"/api/auth/signup"`
In our fetch we need to configure a proxy server in our vite.config.js file

```
export default defineConfig({
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3000', // The target URL where
the requests will be proxied
        changeOrigin: true, // Changes the origin of the host
header to the target URL
        secure: false, // If the target URL uses HTTPS, set this
to true
        // rewrite: (path) => path.replace(/^\/api/, ''), //
Rewrites the path if needed
        // agent: new http.Agent(), // Optional, can be used to
configure the HTTP agent
      },
    },
  },
```

Now we can submit the form and test if we get a successful response creating a new user.

We also need to handle errors and give the user a good UX by changing the signup button to loading while creating the user. Add two more useStates for error and loading and set each to false or null depending on success or failure.

```
export default function SignUp() {
  const [formData, setFormData] = useState({});
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);
```

```
  const navigate = useNavigate();

  const handleChange = (e) => {
    setFormData({
      ...formData,
      // whatever is changing set to that value
      [e.target.id]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      setLoading(true);
      // in order to use the url '/api/auth/signup' from the
client side
      // to the api side we need to use a proxy configured in
vite.config.js
      const res = await fetch("/api/auth/sign-up", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(formData),
      });
      const data = await res.json();
      if (data.success === false) {
        setError(data.message);
        setLoading(false);
        return;
      }
      setLoading(false); // loading done
      setError(null);
      navigate("/sign-in");
```

```
    } catch (error) {
      setLoading(false);
      setError(error.message);
    }
  };
  return (
```

We also need to navigate to sign-in page if successful so import

```
import { Link, useNavigate } from "react-router-dom";
 const navigate = useNavigate();
navigate("/sign-in");
```

Change the button text and opacity using the loading variable

```
<button
          disabled={loading}
          className="bg-slate-700 text-white p-3 rounded-lg
uppercase hover:opacity-95 disabled:opacity-80"
        >
          {loading ? "Loading..." : "Sign Up"}
        </button>
```

Add error to bottom of page using the error variable

```
</div>
      {error && <p className="text-red-500">{error}</p>}
    </div>
```

Create sign in api route

Create a signin function inside ***api/controllers/auth.controller.js*** to check if we have valid email and password.

```
export const signin = async (req, res, next) => {
    const { email, password } = req.body;
    try {
        // check if we have valid email
        const validUser = await User.findOne({ email });
```

```
        if(!validUser) return next(errorHandler(404, 'User not
found'));
        // if valid email check if valid password
        const validPassword = bcrypt.compareSync(password,
validUser.password);
        if(!validPassword) return next(errorHandler(401,
'Invalid credentials'))
    } catch (error) {
        // handle errors using middleware in index.js
        next(error)
    }
}
```

Once we are sure we have valid email and password we need to hash the user ID and store this in a cookie in the browser so that the user will be authenticated while browsing the app. We will use JSON Web Token (JWT) to create that hash token.
In the root of the app folder ***npm install jsonwebtoken*** and import it at the top of the auth.controller.js then add a JWT_SECRET variable to the .env so we can use it as an extra level of security to hash our id in our cookies. The key can be any random keystrokes you choose for your app. Now update your signin function and store the cookie in the browser

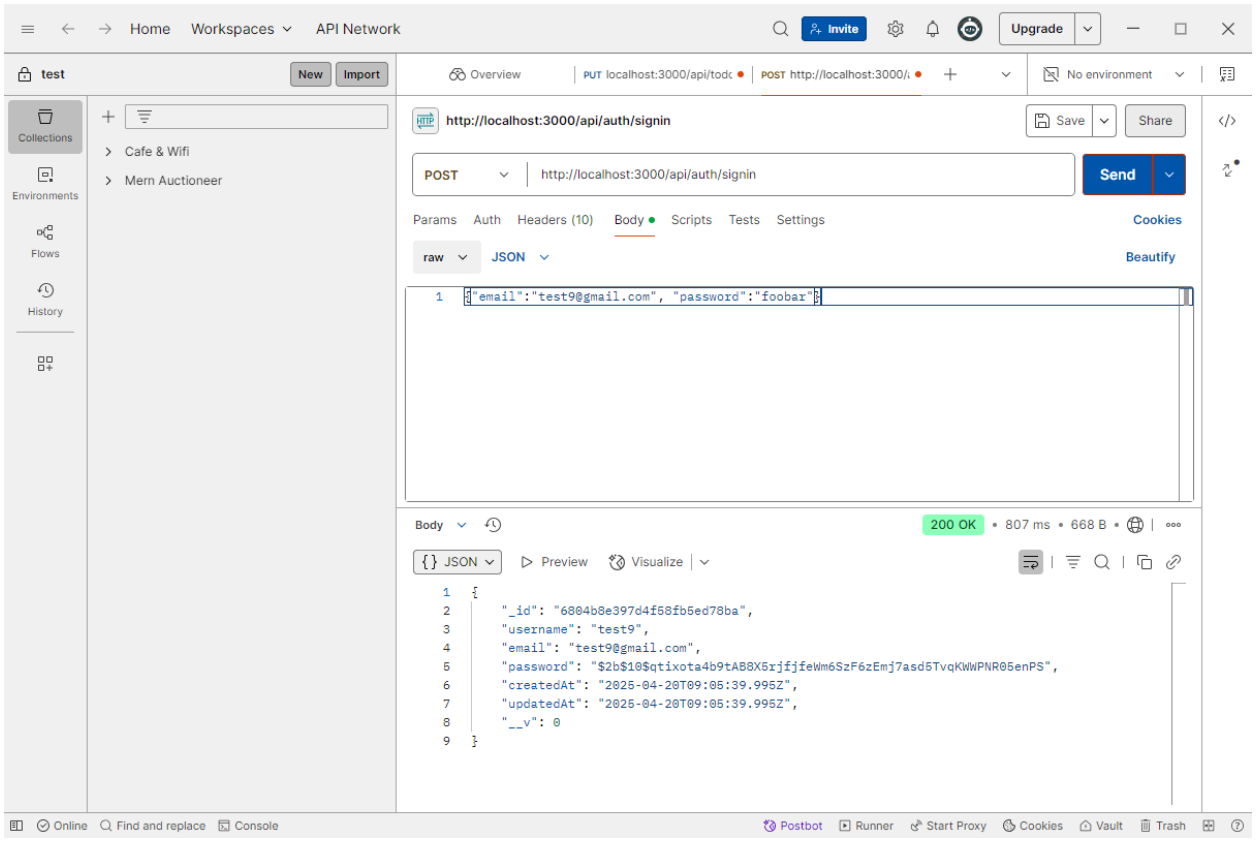
```
export const signin = async (req, res, next) => {
    const { email, password } = req.body;
    try {
        // check if we have valid email
        const validUser = await User.findOne({ email });
        if(!validUser) return next(errorHandler(404, 'User not
found'));
        // if valid email check if valid password
        const validPassword = bcrypt.compareSync(password,
validUser.password);
        if(!validPassword) return next(errorHandler(401,
'Invalid credentials'));
        const token = jwt.sign({ id: validUser._id },
process.env.JWT_SECRET);
        // store the cookie in the browser as a session
```

```
        res.cookie('access_token', token, {httpOnly:
true}).status(200).json(validUser);

    } catch (error) {
        // handle errors using middleware in index.js
        next(error)
    }
}
```

Now we need to add the route to api/routes/auth.route.js

```
import express from 'express';
import { signin, signup } from
'../controllers/auth.controller.js';

const router = express.Router();

router.post('/signup', signup);
router.post('/signin', signin);

export default router;
```

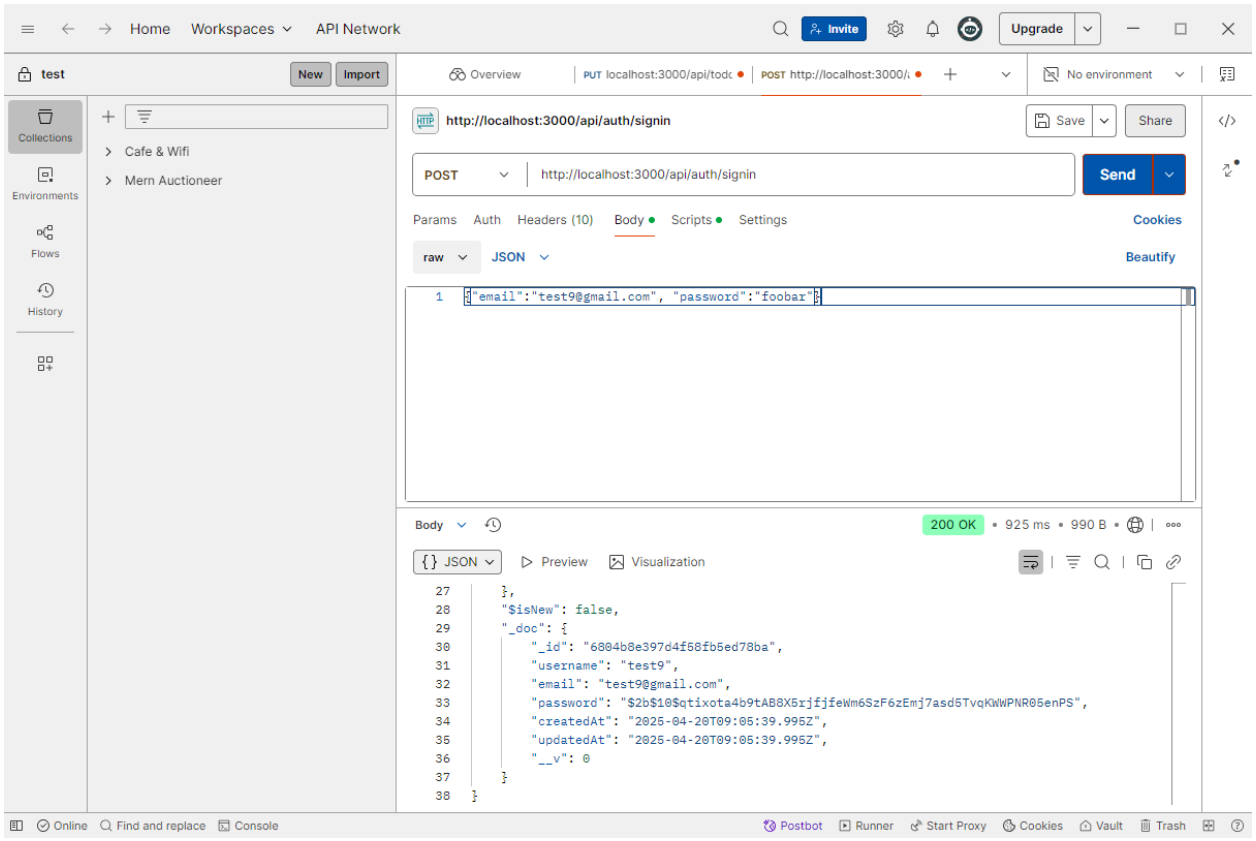
We can test the sign in using postman sending just email and password and see we get a response with or valid user info. But the valid user contains the hashed password and we don't want to send the password even if it is hashed as this is bad practice.



We can remove the password by destructuring the valid user before we send it

```
if(!validPassword) return next(errorHandler(401, 'Invalid
credentials'));
        const token = jwt.sign({ id: validUser._id },
process.env.JWT_SECRET);
        // destructure the user so we can remove the password
        const { password: pass, ...rest } = validUser;
        // store the cookie in the browser as a session
        res
        .cookie('access_token', token, {httpOnly: true})
        .status(200)
        .json(rest);
```

But if we test it using postman we are still sending the password



As you can see we need to destructure the validUser._doc in our code to get the correct response.

```
// destructure the user so we can remove the password
        const { password: pass, ...rest } = validUser._doc;
```

If we test it with postman again we see the password has been removed.

Create the sign in page

Just copy all the sign up page code and paste into signin.jsx. Change these lines to suit the sign in page.

```
export default function SignIn() {
```

```
const res = await fetch("/api/auth/signin", {
```

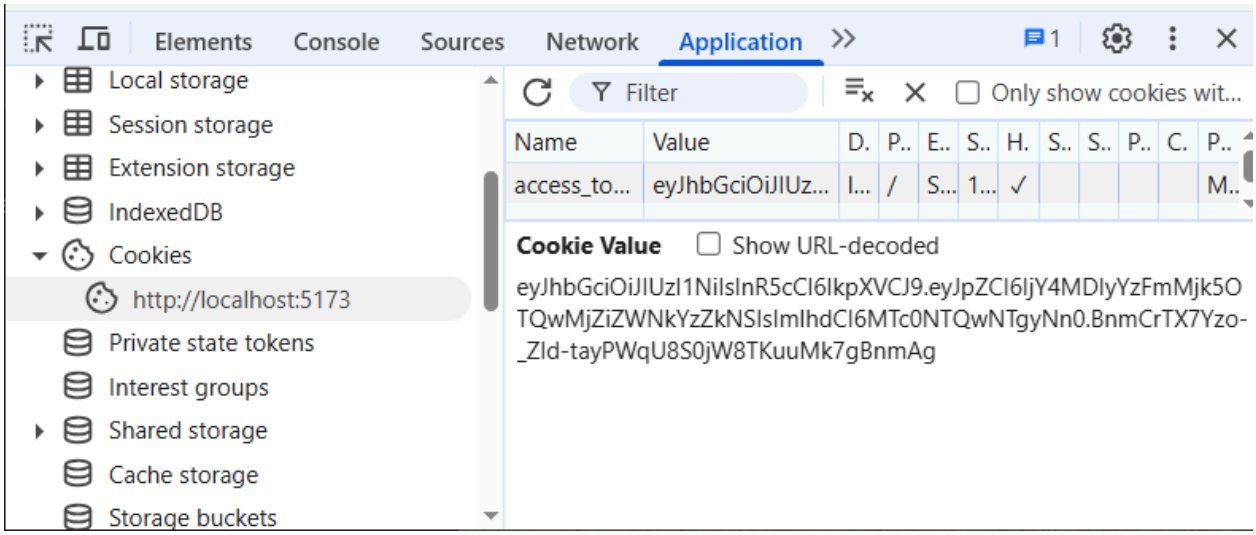
Remove the input for username and change the button name to

```
{loading ? "Loading..." : "Sign In"}
```

Change the message at the bottom of the page to

```
<div className="flex gap-2 mt-5">
        <p>Dont have an account?</p>
        <Link to={"/sign-up"}>
          <span className="text-blue-700">Sign Up</span>
        </Link>
```

Now sign in with a user you created earlier and check to see if your are sent to the home page and it has the cookie with a name of access_token is in the browser under the Application tab of the console



Add Redux Toolkit to our App

In order to keep our user credentials across all the pages on our app we need to install redux-toolkit on the client side. Go to https://redux-toolkit.js.org/tutorials/quick-start
npm install @reduxjs/toolkit react-redux

Create a new folder and file in the client src/redux/store.js

```
import { configureStore } from '@reduxjs/toolkit'


export const store = configureStore({
  reducer: {},
  // set serializableCheck to false so we don't get error for
non
  // serialized variables
  middleware: (getDefaultMiddleware) => getDefaultMiddleware({
```

```
    serializableCheck: false,
  }),
})
```

Now we need to create a Redux State Slice so create a new file in src/redux/user/userSlice.js

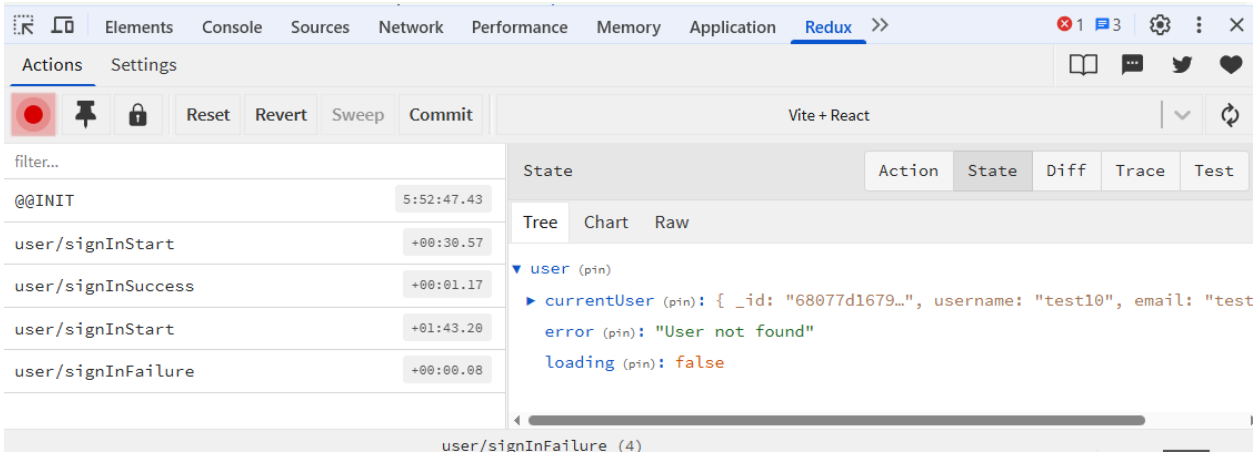
```
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
    currentUser: null,
    error: null,
    loading: false,
};

const userSlice = createSlice({
    name: 'user',
    initialState,
    reducers: {
        signInStart: (state) => {
            state.loading = true;
        },
        signInSuccess: (state, action) => {
            state.currentUser = action.payload;
            state.loading = false;
            state.error = null;
        },
        signInFailure: (state, action) => {
            state.error = action.payload;
            state.loading = false;
        }
    }
});

export const {signInStart, signInSuccess, signInFailure} =
export default userSlice.reducer;
```

Now we can use the reducer in SignIn.jsx by importing useDispatch from react and our actions and refactor signin.jsx to use our userSlice actions anywhere we had setError or setLoading

```
import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { useDispatch, useSelector } from "react-redux";
import {
  signInStart,
  signInSuccess,
  signInFailure,
} from "../redux/user/userSlice.js";

export default function SignIn() {
  const [formData, setFormData] = useState({});
  // const [error, setError] = useState(null);
  // const [loading, setLoading] = useState(false);
  const { loading, error } = useSelector((state) => state.user);
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const handleChange = (e) => {
    setFormData({
      ...formData,
      // whatever is changing set to that value
      [e.target.id]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      // setLoading(true);
      dispatch(signInStart());
      // in order to use the url '/api/auth/signup' from the
client side
      // to the api side we need to use a proxy configured in
vite.config.js
```

```
      // "/api/auth/signup" comes from auth.route.js in
api/routes
      const res = await fetch("/api/auth/signin", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(formData),
      });

      const data = await res.json();
      // api/index.js sets success to false in error
      // handling middleware
      if (data.success === false) {
        // setError(data.message);
        // setLoading(false);
        dispatch(signInFailure(data.message));
        return;
      }
      // setLoading(false); // loading done
      // setError(null);
      dispatch(signInSuccess(data));
      navigate("/");
    } catch (error) {
      // setLoading(false);
      // setError(error.message);
      dispatch(signInFailure(error.message));
    }
  };
  return (
```

Now we can use Redux developer tool in the chrome browser to see what our state is when signing in.



Adding redux persist to our app

Although the redux toolkit is now storing the state in the browser, if we refresh the page we lose all this data. We have to store this data in the local storage using redux persist.
Install redux-persist in the client folder ***npm i redux-persist*** and refactor store.js to combine the reducers instead of having a single userReducer and then persist this reducer using redux-persist

```
import { combineReducers, configureStore } from
'@reduxjs/toolkit';
import userReducer from './user/userSlice';
import {persistReducer} from 'redux-persist';
import storage from 'redux-persist/lib/storage'
import persistStore from 'redux-persist/es/persistStore';

const rootReducer = combineReducers({user: userReducer});

const persistConfig ={
    key: 'root',
    storage,
    version: 1,
}

const persistedReducer = persistReducer(persistConfig,
rootReducer)
export const store = configureStore({
```

```
    reducer: persistedReducer,
    // set serializableCheck to false so we don't get error for
non
  // serialized variables
  middleware: (getDefaultMiddleware) => getDefaultMiddleware({
    serializableCheck: false,
  }),
});

// make the store persist
export const persistor = persistStore(store);
```

Then we go to the main.jsx file and wrap everything in the persistor using the PersistGate

```
import { createRoot } from "react-dom/client";
import "./index.css";
import App from "./App.jsx";
import { persistor, store } from "./redux/store.js";
import { Provider } from "react-redux";
import { PersistGate } from "redux-persist/integration/react";

createRoot(document.getElementById("root")).render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>
);
```

When we refresh the browser now our current user data will persist so we can browse the app without having to log in again and again.The data is stored in the browser under application - storage - local storage where you will see a key of root and all the user data.

Add Google authentication to our app

Add a new component file called OAuth.jsx in client/src/components/OAuth.jsx. To prevent the Continue with google button from submitting the whole form as the button is inside the form we need to change the type to type=”button” as buttons in a form are submit by default.

```
import React from "react";

export default function OAuth() {
    const handleGoogleClick = async () => {

    }
  return (
    <button
      onClick={handleGoogleClick}
      type="button"
      className="bg-red-700 text-white p-3 rounded-lg
     uppercase hover:opacity-85"
    >
      Continue with google
    </button>
  );
}
```

Now add this to both the Signin and Signup pages under the button element

```
 {loading ? "Loading..." : "Sign In"}
        </button>
        <OAuth />
      </form>
```

We will use firebase to set up our authentication so go to firebase.google.com and log in and go to the console.to set up a new project. Follow the prompts to set up a project and once done you need to install firebase in the client

`npm install firebase`

`Create a new file in client/src/firebase.js and copy the code that includes your api key into it.`

`We need to hide our api key in firebase.js file so replace the apiKey:`

With

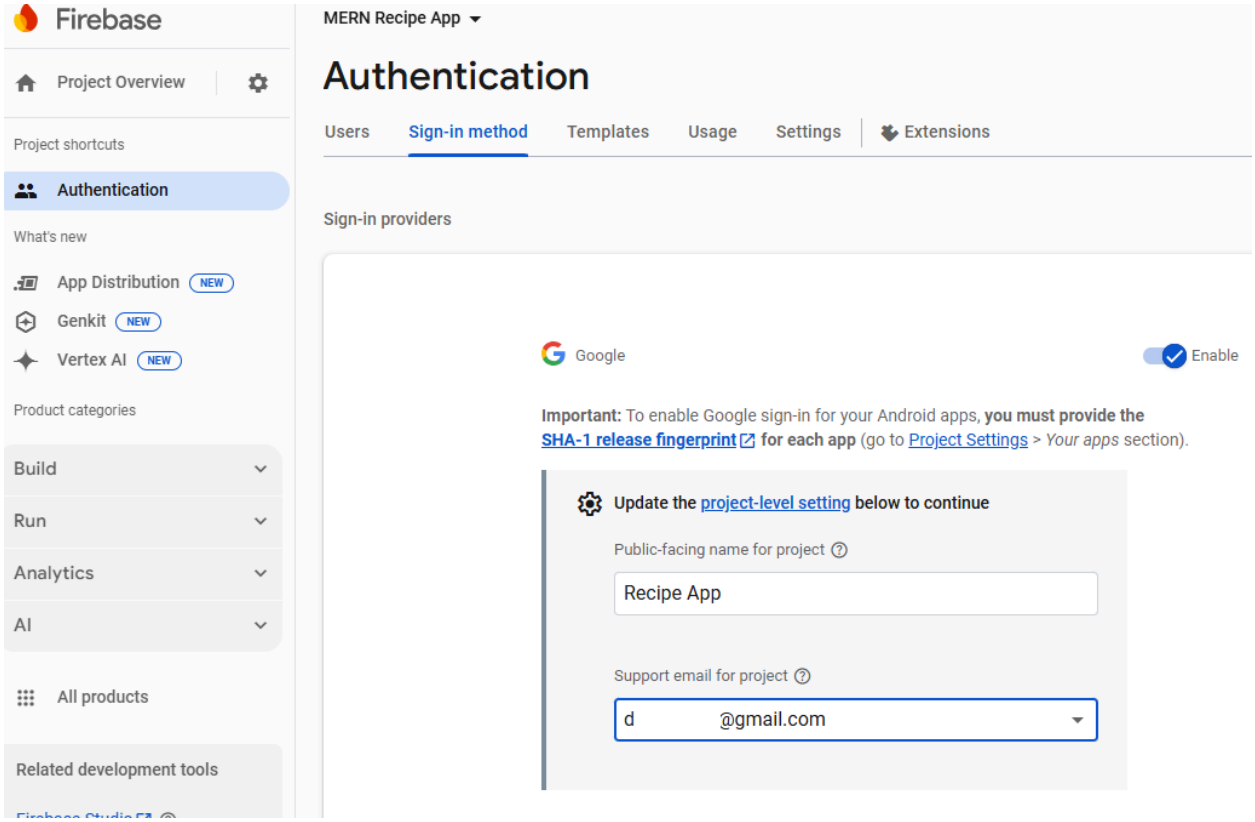
```
import.meta.env.VITE_FIREBASE_API_KEY,
```

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// import { getAnalytics } from "firebase/analytics";
// TODO: Add SDKs for Firebase products that you want to use
//
https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is
optional
const firebaseConfig = {
  apiKey: import.meta.env.VITE_FIREBASE_API_KEY,
  authDomain: "mern-recipe-app-ce8c5.firebaseapp.com",
  projectId: "mern-recipe-app-ce8c5",
  storageBucket: "mern-recipe-app-ce8c5.firebasestorage.app",
  messagingSenderId: "819642081667",
  appId: "1:819642081667:web:5d201b9d0036204ce7d6b6",
  measurementId: "G-C32YTY369L",
};
// Initialize Firebase
export const app = initializeApp(firebaseConfig);
// const analytics = getAnalytics(app);
```

Then create a .env file in the client folder and save your firebase key there, make sure you have a .gitignore file with .env in the list of files to ignore so as your git repository will not save your keys. Now save a env.example file to give people know what is expected.

```
VITE_FIREBASE_API_KEY = "your api key here"
```

In firebase go to authentication and select the authentication you want to use. In our case we will use google. Click enable and choose a name and email account and save



Once we see it is enabled we can use google OAuth in our app. Update your OAuth.jsx handleGoogleClick method with provider, auth and await the result from google.

```
import { getAuth, GoogleAuthProvider, signInWithPopup } from
"firebase/auth";
import { app } from "../firebase";
import { useDispatch } from "react-redux";
import { signInSuccess } from "../redux/user/userSlice";

export default function OAuth() {
  const dispatch = useDispatch();
  const handleGoogleClick = async () => {
    try {
      const provider = new GoogleAuthProvider();
      // pass our firebase config to getAuth
      const auth = getAuth(app);

      // use signup with popup
```

```
      const result = await signInWithPopup(auth, provider);

      // send info from google to the backend
      const res = await fetch("/api/auth/google", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          name: result.user.displayName,
          email: result.user.email,
          photo: result.user.photoURL,
        }),
      });
      // when we get the result convert to json
      const data = await res.json();
      dispatch(signInSuccess(data));
    } catch (error) {
      console.log("Could not sign in with google ", error);
    }
  };
  return (
    <button
      onClick={handleGoogleClick}
      type="button"
      className="bg-red-700 text-white p-3 rounded-lg
     uppercase hover:opacity-85"
    >
      Continue with google
    </button>
  );
}
```

If we console.log our data before dispatch(signInSuccess) we can click the continue with google button on your signin page and you should see your info from google in the console log. We need user.displayName, user.email, and user.photoURL for our app.

That's the frontend completed for OAuth now we need to do the backend. Create the route in api/routes/auth.route.js and add a route for google

```
router.post('/signup', signup);
router.post('/signin', signin);
router.post('/google', google);
```

Next create a function for the controller in api/controllers/auth.controller.js

```
export const google = async (req, res, next) => {
    try {
        const user = await User.findOne({email:
req.body.email});
        if (user) {
            const token = jwt.sign({ id: user._id },
process.env.JWT_SECRET);
            const { password: pass, ...rest } = user._doc;
            res
              .cookie('access_tooken', token, { httpOnly: true
})
              .status(200)
              .json(rest);
        } else {
const generatedPassword = Math.random().toString(36).slice(-8) +
Math.random().toString(36).slice(-8);
            const hashedPassword =
bcrypt.hashSync(generatedPassword, 10);
            const uniqueUserName = req.body.name.split("
").join("").toLowerCase() +
Math.random().toString(36).slice(-4);
            const newUser = new User({
                username: uniqueUserName,
                email: req.body.email,
```

```
                password: hashedPassword,
                avatar: req.body.photo
            });
            await newUser.save();
            const token = jwt.sign({ id: newUser._id },
process.env.JWT_SECRET);
            const { password: pass, ...rest } = newUser._doc;
            res
            .cookie('access_tooken', token, { httpOnly: true })
            .status(200)
            .json(rest);
        }
    } catch (error) {
        next(error);
    }
}
```

If the user exists, sign them in. If not, create a new user in the database with the info from google OAuth and sign them in.
We also need to add an avatar photo for our profile page and header, so open api/models/user.model.js and add the avatar to our userSchema

```
 avatar: {
        type: String,
        default:
"https://cdn.pixabay.com/photo/2016/11/08/15/21/user-1808597_640
.png"
    },
```

Now we can test it by signing in with a google account, make sure you have at least two google accounts or it will not pop up a sign in window for you to choose which account. Sign in and look at the network tab in chrome console and you should see your google response with status 200 and a json object with avatar, id, username, and email. Your username will be concatenated and have 4 random characters added to make it unique.
Also check redux state to see that the current user is the user from google.

Once we have a successful sign in we want to navigate to the home page, so in OAuth.jsx import useNavigate

```
import { useNavigate } from "react-router-dom";
```

And navigate to home page after we dispatch signInSuccess

```
 dispatch(signInSuccess(data));
      navigate("/");
```

Update our Header page with a profile picture and make the profile page private.

In Header.jsx we need to access the redux store to see the current user and add a profile picture to the header with a link to the profile page. In ***client/src/components/Header.jsx*** add the following code

```
import { useSelector } from "react-redux";

export default function Header() {
  // get just the currentUser property from state.user in redux
store
  const { currentUser } = useSelector((state) => state.user);
  return (
```

```
<Link to="/about">
            <li className="hidden sm:inline text-slate-700
hover:underline">
              About
            </li>
          </Link>
          <Link to="/profile">
            {currentUser ? (
              <img
                className="rounded-full h-7 w-7 object-cover"
                src={currentUser.avatar}
                alt="profile"
              />
            ) : (
              <li className="text-slate-700 hover:underline
">SignIn</li>
            )}
```

```
          </Link>
```

To protect the profile page from unauthorized access we need to create a new component called PrivateRoute.jsx. In client/arc/components/PrivateRoute.jsx add the code which returns the currentUser or redirects to the /sign-in page.

```
import { useSelector } from "react-redux";
import { Outlet, Navigate } from "react-router-dom";

export default function PrivateRoute() {
  const { currentUser } = useSelector((state) => state.user);
  return currentUser ? <Outlet /> : <Navigate to="/sign-in" />;
}
```

We also have to wrap our route in App.jsx with the PrivateRoute as below

```
<BrowserRouter>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route element={<PrivateRoute />}>
          <Route path="/profile" element={<Profile />} />
        </Route>
        <Route path="/sign-in" element={<SignIn />} />
        <Route path="/sign-up" element={<SignUp />} />
      </Routes>
    </BrowserRouter>
```

Now when a user is signed in and clicks on their profile picture they will go to a private profile page in which they can change their profile when we write the code for that next.

Complete profile page

For the UI add this code to the profile page

```
import { useSelector } from "react-redux";

export default function Profile() {
```

```
  const { currentUser } = useSelector((state) => state.user);
  return (
    <div className="p-3 max-w-lg mx-auto">
      <h1 className="text-3xl font-semibold text-center
my-7">Profile</h1>
      <form className="flex flex-col gap-4">
        <img
          src={currentUser.avatar}
          alt="profile"
          className="rounded-full h-24 w-24 object-cover
cursor-pointer self-center mt-2"
        />
        <input
          type="text"
          placeholder="username"
          id="username"
          className="border p-3 rounded-lg"
        />
        <input
          type="text"
          placeholder="email"
          id="email"
          className="border p-3 rounded-lg"
        />
        <input
          type="text"
          placeholder="password"
          id="password"
          className="border p-3 rounded-lg"
        />
        <button className="bg-slate-700 text-white rounded-lg
p-3 uppercase hover:opacity-90 disabled:opacity-80">
          Update
        </button>
```

```
      </form>
      <div className="flex justify-between mt-4">
        <span className="text-red-700 cursor-pointer">Delete
Account</span>
        <span className="text-red-700 cursor-pointer">Sign
Out</span>
      </div>
    </div>
  );
}
```

Complete the Image upload functionality

In order to be able to click on the profile image and select an image from our device we need to add an input file field and add a reference using useRef from react. Then add that reference to the onClick event on the image. Once the image and the file input are linked you can hide the input.

```
export default function Profile() {
  const fileRef = useRef(null);
```

```
return (
    <div className="p-3 max-w-lg mx-auto">
      <h1 className="text-3xl font-semibold text-center
my-7">Profile</h1>
      <form onSubmit={handleSubmit} className="flex flex-col
gap-4">
        <input
          onChange={(e) => setFile(e.target.files[0])}
          type="file"
          ref={fileRef}
          hidden
          accept="image/*"
        />
        <img
```

```
          onClick={() => fileRef.current.click()}
          src={formData.avatar || currentUser.avatar}
          alt="profile"
          className="rounded-full h-24 w-24 object-cover
cursor-pointer self-center mt-2"
        />
```

Add some state for the file and set it to undefined

```
 const [file, setFile] = useState(undefined);
```

Now add an onchange event listener to set the sate to the first file in the list (if user selects more that one file we will only take the first one)

```
<input
          onChange={(e) => setFile(e.target.files[0])}
          type="file"
          ref={fileRef}
          hidden
          accept="image/*"
        />
```

We will store our images in firebase, so go to your firebase project and select Build > Storage and follow the prompts to set up your storage for this project. Then in the rules section of the storage page edit the rules with this.

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read;
      allow write: if request.resource.size < 2 * 1024 * 1024
              && request.resource.contentType.matches('image/.*');
    }
  }
}
```

This only allows files less than 2MB and of type image.

We can use the useEffect hook to initiate the upload process when a file is selected. Import useEffect and use it by checking if the file changed and calling a function like so

```
useEffect(() => {
    if (file) {
      handleFileUpload(file);
```

```
    }
  }, [file]);
```

handleFileUpload will get a ref to our firebase storage using getStorage(app), we pass in our app which we exported from [firebase.js]

```
const handleFileUpload = (file) => {
    const storage = getStorage(app);
    const fileName = new Date().getTime() + file.name;
    // set up reference to firebase storage
    const storageRef = ref(storage, fileName);
    // use uploadTask to get the % of file uploaded
    const uploadTask = uploadBytesResumable(storageRef, file);

    uploadTask.on(
      "state_changed",
      (snapshot) => {
        // Progress handler
        const progress =
          (snapshot.bytesTransferred / snapshot.totalBytes) *
100;

        setFilePerc(Math.round(progress));
      },
      (error) => {
        // Error handler
        console.log(error);
        setFileUploadError(true);
      },
      () => {
        // Completion handler

getDownloadURL(uploadTask.snapshot.ref).then((downLoadURL) => {
          setFormData({ ...formData, avatar: downLoadURL });
        });
      }
```

```
    );
  };
```

Now if there is a fileUploadError we print it to the page using

```
p className="text-sm self-center">
          {fileUploadError ? (
            <span className="text-red-700">
              Error uploading Image(image must be less than 2mb)
            </span>
          ) : filePerc > 0 && filePerc < 100 ? (
            <span className="text-slate-700">{`Uploading
${filePerc}`}</span>
          ) : filePerc === 100 ? (
            <span className="text-green-700">Image uploaded
successfully</span>
          ) : (
            ""
          )}
        </p>
```

Once a new image is uploaded show it on the page using src attribute

```
<img
          onClick={() => fileRef.current.click()}
          src={formData.avatar || currentUser.avatar}
          alt="profile"
          className="rounded-full h-24 w-24 object-cover
cursor-pointer self-center mt-2"
        />
```

Adding Update and Delete User Functionality

Add the updateUser route and deleteUser route to api/routes/user.route.js.We sign in a user we create a token inside a cookie and we will use verifyToken to ensure only the user can update or delete their own profile.

```
import express from "express";
import { deleteUser, updateUser } from
"../controllers/user.controller.js";
import { verifyToken } from "../utils/verifyUser.js";

// create a router
const router = express.Router();

router.post("/update/:id", verifyToken, updateUser);
router.delete("/delete/:id", verifyToken, deleteUser);

export default router;
```

VerifyToken is a function we export from utils/verifyUser.js and uses jsonwebtoken to verify the token. In order to get data from the cookie we need to install a package called cookie-parser in the root directory which is where we install all packages needed for our api. To use it after install we need to initialise it in index.js

```
import cookieParser from 'cookie-parser';
app.use(cookieParser());
```

In utils/verifyUser.js we check to see if we have a token and if we have we use JWT with our JWT_SECRET to check if its a user from our server.

```
import { errorHandler } from "./error.js";
import jwt from "jsonwebtoken";

export const verifyToken = (req, res, next) => {
  // get token from the stored cookie
  const token = req.cookies.access_token;

  if (!token) return next(errorHandler(401, "Unauthorized"));

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return next(errorHandler(403, "Forbidden"));

    req.user = user;
```

```
    next();
  });
};
```

updateUser and deleteUser functions will be in controller/user.controller.js

```
import User from "../models/user.model.js";
import { errorHandler } from "../utils/error.js";
import bcryptjs from "bcryptjs";

export const updateUser = async (req, res, next) => {
  if (req.user.id !== req.params.id)
    return next(errorHandler(401, "You can only update your own
profile"));
  try {
    // Only process fields that are provided
    const updateFields = {};

    if (req.body.username) updateFields.username =
req.body.username;
    if (req.body.email) updateFields.email = req.body.email;
    if (req.body.avatar) updateFields.avatar = req.body.avatar;

    // Handle password separately for hashing
    if (req.body.password) {
      if (req.body.password.length < 6) {
        return next(
          errorHandler(400, "Password must be at least 6
characters long")
        );
      }
      updateFields.password =
bcryptjs.hashSync(req.body.password, 10);
    }
```

```
    // Only proceed if there are fields to update
    if (Object.keys(updateFields).length === 0) {
      return next(errorHandler(400, "No valid update fields
provided"));
    }

    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      { $set: updateFields },
      { new: true }
    );

    if (!updatedUser) {
      return next(errorHandler(404, "User not found"));
    }

    // Separate password from other data
    const { password, ...rest } = updatedUser._doc;
    res.status(200).json(rest);


    // res.status(200).json(rest);
  } catch (error) {
    if (error.code === 11000) {
      // MongoDB duplicate key error
      return next(errorHandler(400, "Username or email already
exists"));
    }
    next(error);
  }
};

export const deleteUser = async (req, res, next) => {
  if (req.user.id !== req.params.id)
```

```
    return next(errorHandler(401, "You can only delete your own
profile"));
  try {
    await User.findByIdAndDelete(req.params.id);
    res.clearCookie("access_token");
    res.status(200).json("Your profile has been deleted");
  } catch (error) {
    next(error);
  }
};
```

Complete user update and delete functionality

We need to update the profile.jsx in the client to show the user its current values for the input fields. We can do this by giving each a defaultValue. We also need to add an onChange event listener to call a function called handleChange if anything changes in these input fields.

```
<input
          type="text"
          placeholder="username"
          defaultValue={currentUser.username}
          id="username"
          className="border p-3 rounded-lg"
          onChange={handleChange}
        />
        <input
          type="text"
          placeholder="email"
          defaultValue={currentUser.email}
          id="email"
          className="border p-3 rounded-lg"
          onChange={handleChange}
        />
```

The handleChange function takes the event and changes the formdata based on the id of the input tag.

```
const handleChange = (e) => {
```

```
    // change formData based on id of the input tag
    setFormData({ ...formData, [e.target.id]: e.target.value });
  };
```

The handleSubmit takes care of the update so we have to add it to the form first

```
<form onSubmit={handleSubmit}
```

We are using useDispatch inside the handleSubmit functions so we need to add the reducers in client/src/redux/user/userSlice.js

```
// For more info on Redux State Slice see
// https://redux-toolkit.js.org/tutorials/quick-start

import { createSlice } from "@reduxjs/toolkit";

const initialState = {
  currentUser: null,
  error: null,
  loading: false,
};

const userSlice = createSlice({
  name: "user",
  initialState,
  reducers: {
    signInStart: (state) => {
      state.loading = true;
    },
    signInSuccess: (state, action) => {
      state.currentUser = action.payload;
      state.loading = false;
      state.error = null;
    },
    signInFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
```

```
    updateUserStart: (state) => {
      state.loading = true;
    },
    updateUserSuccess: (state, action) => {
      state.currentUser = action.payload;
      state.loading = false;
      state.error = null;
    },
    updateUserFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
    deleteUserStart: (state) => {
      state.loading = true;
    },
    deleteUserSuccess: (state) => {
      state.currentUser = null;
      state.error = null;
      state.loading = false;
    },
    deleteUserFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
  },
});

export const {
  signInStart,
  signInSuccess,
  signInFailure,
  updateUserStart,
  updateUserSuccess,
  updateUserFailure,
```

```
  deleteUserStart,
  deleteUserSuccess,
  deleteUserFailure,
} = userSlice.actions;

export default userSlice.reducer;
```

Now we need to import our reducers and userDispatch in profile.jsx and initialise useDispatch

```
import {
  updateUserStart,
  updateUserSuccess,
  updateUserFailure,
  deleteUserFailure,
  deleteUserStart,
  deleteUserSuccess,
} from "../redux/user/userSlice";
import { useDispatch } from "react-redux";
```

```
const dispatch = useDispatch();
```

And inside we use a fetch to send our formData to the api/user/update route with userId

```
const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      dispatch(updateUserStart());
      const res = await
fetch(`/api/user/update/${currentUser._id}`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(formData),
      });
      const data = await res.json();
      if (data.success === false) {
        dispatch(updateUserFailure(data.message));
        return;
```

```
      }

      dispatch(updateUserSuccess(data));
      setUpdateSuccess(true);
    } catch (error) {
      // dispatch the error using reducer
redux/user/userSlice.js
      dispatch(updateUserFailure(error.message));
    }
  };
```

The user routes in api/routes/user.route.js are

```
import express from "express";
import { deleteUser, updateUser } from
"../controllers/user.controller.js";
import { verifyToken } from "../utils/verifyUser.js";

// create a router
const router = express.Router();

router.post("/update/:id", verifyToken, updateUser);
router.delete("/delete/:id", verifyToken, deleteUser);

export default router;
```

The user update and delete functions for the api are in api/contorller/user.controller.js

```
import User from "../models/user.model.js";
import { errorHandler } from "../utils/error.js";
import bcryptjs from "bcryptjs";

export const updateUser = async (req, res, next) => {
  //compare id returned form verifyUser.js to the id in params
  if (req.user.id !== req.params.id)
```

```
    return next(errorHandler(401, "You can only update your own
profile"));
  try {
    // Only process fields that are provided
    const updateFields = {};

    if (req.body.username) updateFields.username =
req.body.username;
    if (req.body.email) updateFields.email = req.body.email;
    if (req.body.avatar) updateFields.avatar = req.body.avatar;

    // Handle password separately for hashing
    if (req.body.password) {
      if (req.body.password.length < 6) {
        return next(
          errorHandler(400, "Password must be at least 6
characters long")
        );
      }
      updateFields.password =
bcryptjs.hashSync(req.body.password, 10);
    }

    // Only proceed if there are fields to update
    if (Object.keys(updateFields).length === 0) {
      return next(errorHandler(400, "No valid update fields
provided"));
    }

    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      // $set ignores empty fields
      { $set: updateFields },
      { new: true } //returns updated user object
```

```
    );

    if (!updatedUser) {
      return next(errorHandler(404, "User not found"));
    }

    // Seperate password from the rest of data and send rest
    // as json response
    const { password, ...rest } = updatedUser._doc;
    res.status(200).json(rest);
  } catch (error) {
    if (error.code === 11000) {
      // MongoDB duplicate key error
      return next(errorHandler(400, "Username or email already
exists"));
    }
    next(error);
  }
};

export const deleteUser = async (req, res, next) => {
  // req.user.id comes from verifyUser JWT
  if (req.user.id !== req.params.id)
    return next(errorHandler(401, "You can only delete your own
profile"));
  try {
    await User.findByIdAndDelete(req.params.id);
    res.clearCookie("access_token");
    res.status(200).json("Your profile has been deleted");
  } catch (error) {
    next(error);
  }
};
```

The handleDeleteUser takes care of deleting a user using useDispatch

```
const handleDeleteUser = async () => {
    try {
      dispatch(deleteUserStart());
      const res = await
fetch(`/api/user/delete/${currentUser._id}`, {
        method: "DELETE",
      });
      const data = await res.json();
      if (data.success == false) {
        dispatch(deleteUserFailure(data.message));
      }
      dispatch(deleteUserSuccess(data));
    } catch (error) {
      dispatch(deleteUserFailure(error.message));
    }
  };
```

Update Profile.jsx to call handleDeleteUser when the Delete Account button is clicked.

```
<div className="flex justify-between mt-4">
        <span
          onClick={handleDeleteUser}
          className="text-red-700 cursor-pointer"
        >
          Delete Account
        </span>
```

Also use redux/userSlice.js to set currentUser to null.

```
deleteUserStart: (state) => {
      state.loading = true;
    },
```

```
    deleteUserSuccess: (state) => {
      state.currentUser = null;
      state.error = null;
      state.loading = false;
    },
    deleteUserFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
```

Update Profile.jsx to show errors updating and successful updates by adding state of updateSuccess and two p tags to the profile page. The error will come from the try catch statements in submit and delete handlers.

```
export default function Profile() {
  const fileRef = useRef(null);
  const { currentUser, loading, error } = useSelector((state) =>
state.user);
  const [file, setFile] = useState(undefined);
  const [filePerc, setFilePerc] = useState(0);
  const [fileUploadError, setFileUploadError] = useState(false);
  const [formData, setFormData] = useState({});
  const [updateSuccess, setUpdateSuccess] = useState(false);
  const dispatch = useDispatch();
```

```
<p className="text-red-700 mt-5">{error ? error : ""}</p>
      <p className="text-green-700 mt-5">
        {updateSuccess ? "User profile updated successfully" :
""}
      </p>
    </div>
```

Add Sign out functionality

First create the backend in authentication api/routes/auth.route.js

```
router.get("/signout", signout);
```

Create the controller in controllers/auth.controller.js to clear the cookie and return the status

```
export const signout = async (req, res, next) => {
  try {
    res.clearCookie("access_token");
    res.status(200).json("User has been logged out!");
  } catch (error) {
    next(error);
  }
};
```

Add the onClick to the Sign Out tag on Profile.jsx

```
<span onClick={handleSignOut} className="text-red-700
cursor-pointer">Sign Out</span>
```

Add the handleSignOut function to Profile.jsx

```
 const handleSignOut = async (req, res, next) => {
   try {
    const res = await fetch('/api/auth/signout');
    const data = await res.json();
    if(data.success === false) {
      return;
    }
   } catch (error) {
    next(error);
   }
  }
```

Create the signout redux reducers in redux/userSlice.js similar to delete

```
signOutUserStart: (state) => {
      state.loading = true;
    },
```

```
    signOutUserSuccess: (state) => {
      state.currentUser = null;
      state.error = null;
      state.loading = false;
    },
    signOutUserFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
```

Update handleSignOut in Profile.jsx with dispatch code

```
const handleSignOut = async (req, res, next) => {
    try {
      dispatch(signOutUserStart());
      const res = await fetch("/api/auth/signout");
      const data = await res.json();
      if (data.success === false) {
        dispatch(signOutUserFailure(data.message));
        return;
      }
      dispatch(signOutUserSuccess(data));
    } catch (error) {
      dispatch(signOutUserFailure(error));
      next(error);
    }
  };
```