Aplicación de inicio de sesión/Autenticación

Aplicación MERN Stack con autenticación de usuario a través de passport y JWT.

Crearemos una aplicación mínima de inicio de sesión/autorización utilizando MERN (MongoDB para nuestra base de datos, Express y Node para nuestro backend y React para nuestro frontend). También integraremos Redux para la administración del estado de nuestros componentes React.

Nuestra aplicación permitirá a los usuarios:

- Registrarse
- Iniciar sesión
- Acceder a páginas protegidas solo accesibles para usuarios registrados
- Mantenerse logueado cuando cierren la aplicación o actualicen la página
- Cerrar sesión

En esta parte:

- Se Inicializará nuestro backend usando npm e instalando los paquetes necesarios
- Configurar una base de datos MongoDB usando mLab
- Configurar un servidor con Node.js y Express
- Cree un esquema de base de datos para definir un usuario para fines de registro e inicio de sesión
- Configurar dos rutas API, registro e inicie sesión, usando passport + jsonwebtokens para autenticación y validador para validación de entrada
- Probar nuestras rutas API usando Postman

Prerrequisitos

Tener instalado en el pc lo siguiente:

- Editor VSCode.

- Última versión de Node.js (usaremos npm, o "Node Package Manager", para instalar dependencias, como pip para Python o gems para Ruby). Para actualizar NPM utilizar el siguiente comando:

NPM

```
npm install npm -g
```

- MongoDB (instalación rápida: instale Homebrew y ejecute brew update && brew install mongodb)
- Postman para pruebas de API
- Prettier, para formatear sin problemas nuestro Javascript

Parte 1: Creando nuestro backend

I. Inicializando nuestro proyecto

Establezca el directorio actual en el lugar donde desea desarrollar su proyecto:

```
mkdir mern-auth
```

Cambie a ese directorio:

```
cd mern-auth
```

Ejecute:

```
npm init
```

Después de ejecutar el comando, una utilidad lo guiará a través de la creación de un archivo package.json.

Puede ingresar a través de la mayoría de estos de manera segura, pero continúe y configure el punto de entrada en server.js en lugar del index.js predeterminado cuando se le solicite (puede hacerlo más adelante en nuestro package.json).

II. Configurando nuestro package.json

1. Establezca el punto de entrada "main" en "server.js" en lugar del "index.js" predeterminado, si aún no lo ha hecho (para fines convencionales)

2. Instale las siguientes dependencias usando npm

```
npm i bcryptjs body-parser concurrently express is-empty jsonwebtoken
mongoose passport passport-jwt validator
```

Una breve descripción de cada paquete y la función que cumplirá.

- bcryptjs: utilizado para hash de contraseñas antes de almacenarlas en nuestra base de datos
- body-parser: se utiliza para analizar los cuerpos de las solicitudes entrantes en un middleware
- concurrently: nos permite ejecutar nuestro backend y frontend al mismo tiempo y en diferentes puertos
- express: se ubica en la parte superior de Node para hacer que el enrutamiento, el manejo de solicitudes y la respuesta sean más fáciles de escribir
- is-empty: función global que será útil cuando usemos el validador
- jsonwebtoken: utilizado para autorización
- mongoose: utilizado para interactuar con MongoDB
- passport: se utiliza para autenticar solicitudes, lo que hace a través de un conjunto extensible de complementos conocidos como estrategias
- passport-jwt: estrategia de passport para autenticarse con un JSON Web Token (JWT); le permite autenticar puntos finales usando un JWT
- validator: se utiliza para validar las entradas (por ejemplo, verificar el formato de correo electrónico válido, confirmar que las contraseñas coinciden)

3. Instale nodemon devDependency (-D) usando npm

```
npm i -D nodemon
```

Nodemon es una utilidad que monitoreará cualquier cambio en su código y reiniciará automáticamente su servidor, lo cual es perfecto para el desarrollo. La alternativa sería tener que desactivar su servidor (Ctrl + C) y volver a ponerlo en funcionamiento cada vez que realice un cambio. No es ideal.
Asegúrese de usar nodemon en lugar de node cuando ejecute su código con fines de desarrollo.

4. Cambie el objeto "scripts" del archivo “package.json” a lo siguiente

```
"scripts": {
  "start": "node server.js",
  "server": "nodemon server.js",
},
```

Más adelante, usaremos el servidor de ejecución de nodemon para ejecutar nuestro servidor de desarrollo.

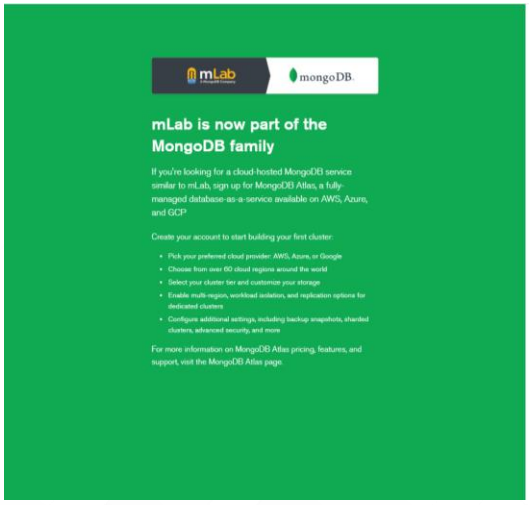
Su archivo package.json debería tener el siguiente aspecto en esta etapa.

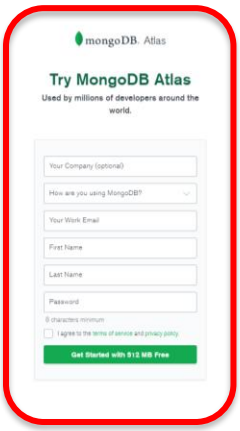
```
{
  "name": "mern-auth",
  "version": "1.0.0",
  "description": "Mern Auth Example",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "server": "nodemon server.js"
  },
  "author": "",
  "license": "MIT",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "body-parser": "^1.18.3",
    "concurrently": "^4.0.1",
    "express": "^4.16.4",
    "is-empty": "^1.2.0",
    "jsonwebtoken": "^8.3.0",
    "mongoose": "^5.3.11",
    "passport": "^0.4.0",
    "passport-jwt": "^4.0.0",
    "validator": "^10.9.0"
  }
}
```

III. Configurando nuestra base de datos

1. Dirígete a mLab y crea una cuenta si aún no tienes una.

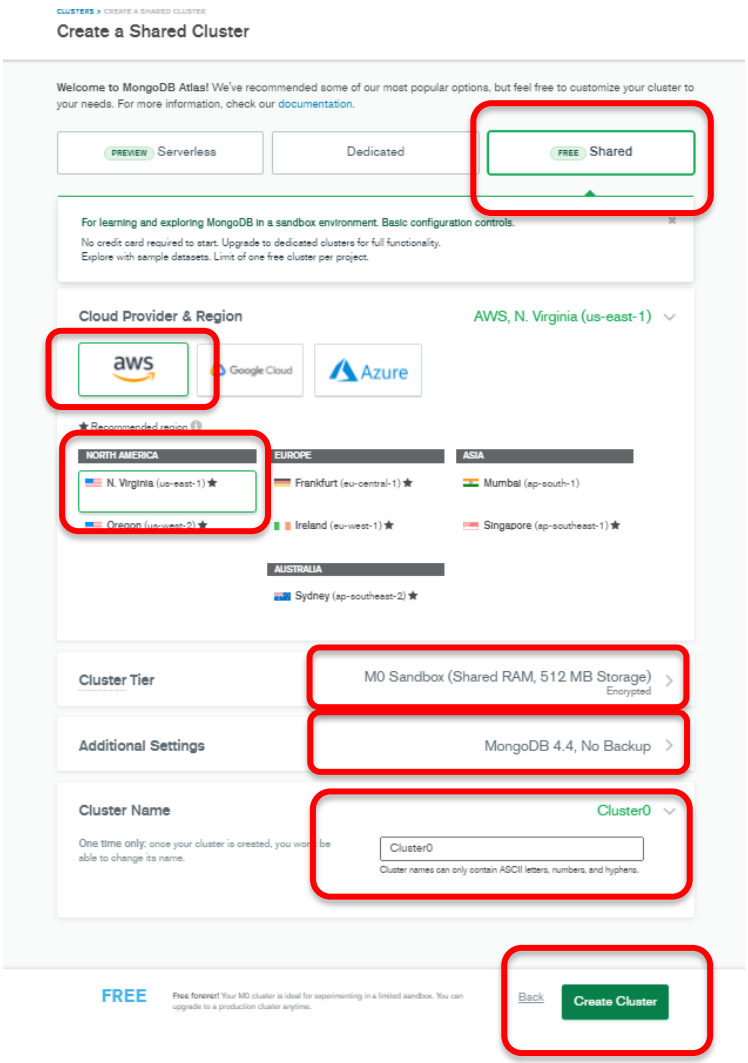
https://mlab.com/





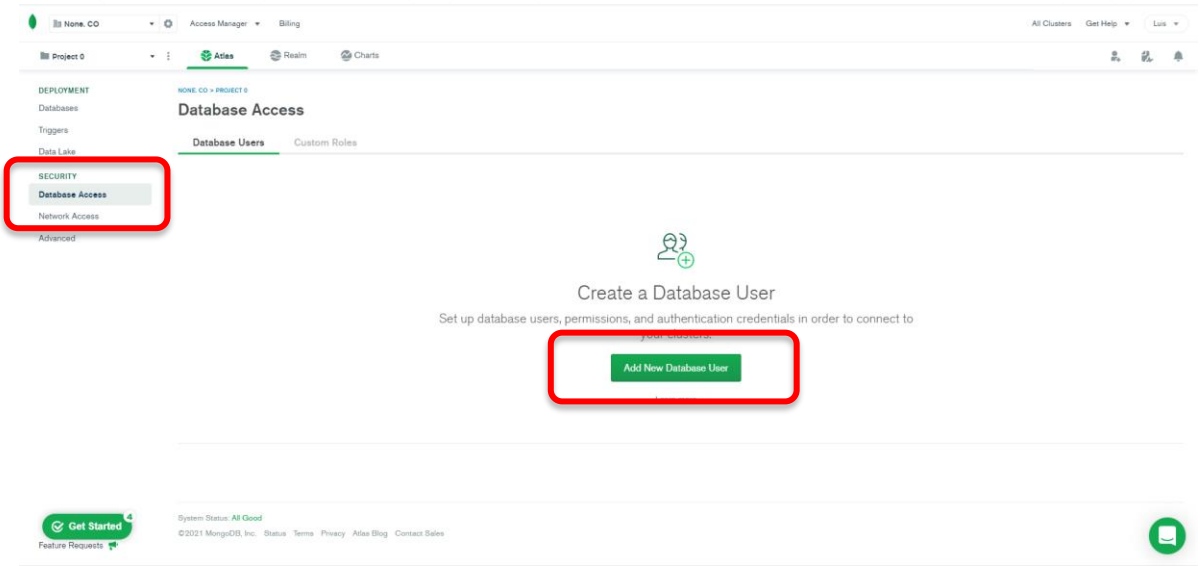
2. Cree una nueva implementación de MongoDB

Seleccione **Shared** como su tipo de plan, **AWS** como su proveedor de nube, dejando la región de AWS por defecto. Por último, asigne un nombre a su base de datos y envíe su pedido (no se preocupe, es gratis).

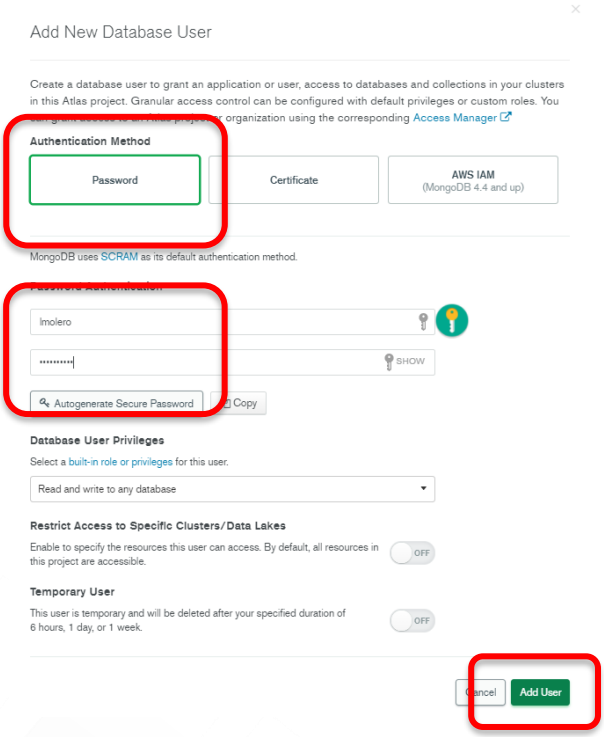


3. Crea tu usuario para acceder a tus bases de datos

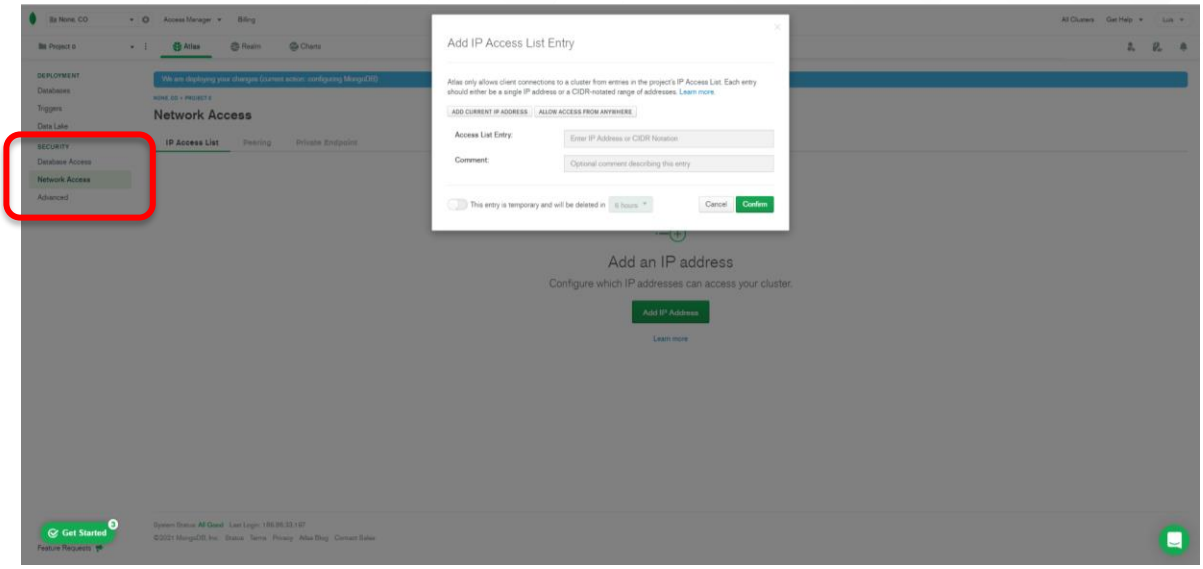
Vaya a la pestaña **Database Access**, haga clic en Agregar usuario de base de datos y cree un usuario de base de datos. Su base de datos necesita al menos un usuario para poder utilizarla.



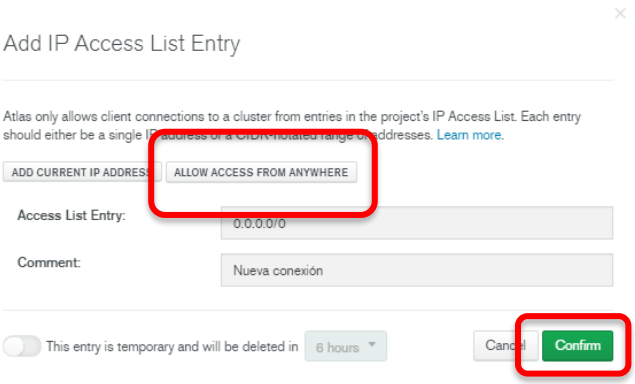
Coloque su nombre de usuario y contraseña y presione el botón “Add User”



Ahora, agregue una IP address para tu lista de acceso, de la forma:

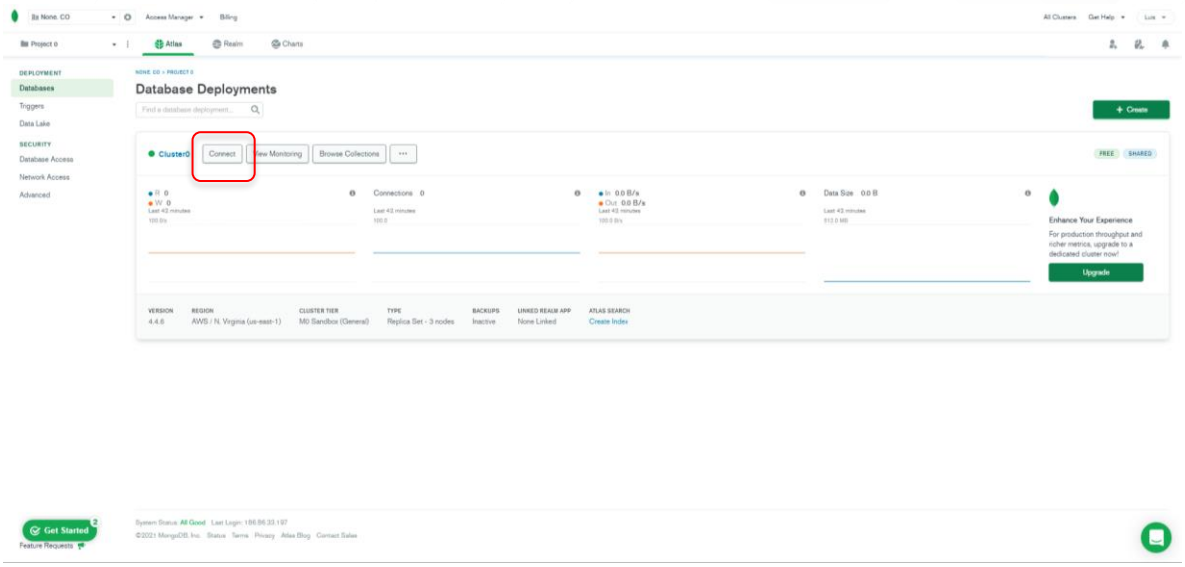


Haga click en la opción “ALLOW ACCESS FROM ANYWHERE”. Asegúrese que quede de la siguiente forma:

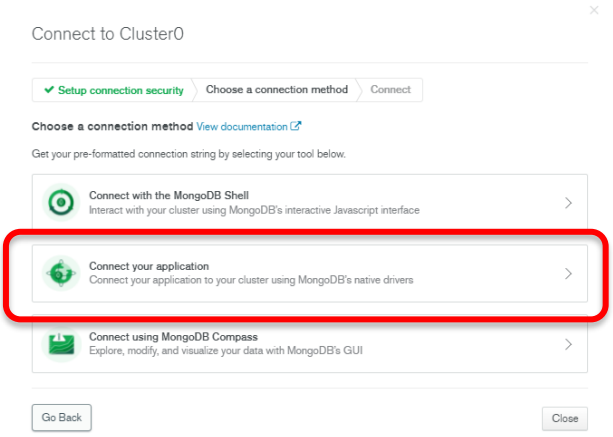


Conectarse a la base de datos

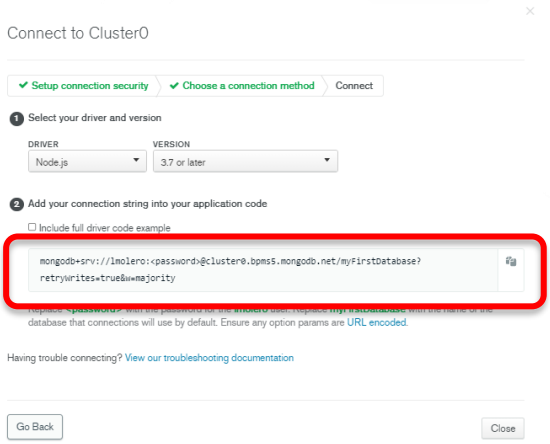
Presione el botón “Connect” ubicado al lado derecho del cluster



Luego, seleccione “Connect your application”



Seleccione el driver y la versión (En nuestro caso será el mismo que aparece por defecto y copie su URI, a saber:



Este URI, se usará para conectarnos a nuestra base de datos.

```
mongodb+srv://lmolero:<password>@cluster0.bpms5.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
```

Reemplace <contraseña> con la contraseña del usuario lmolero. Reemplace myFirstDatabase con el nombre de la base de datos que las conexiones usarán de forma predeterminada. Asegúrese de que los parámetros de las opciones estén codificados en URL.

4. Cree un directorio de configuración dentro de y dentro de él un archivo keys.js

```
mkdir config
```

Cambie a ese directorio:

```
cd config
```

Cree el archivo:

```
keys.js
```

Dentro de su archivo keys.js, coloquemos lo siguiente para facilitar el acceso fuera de este archivo.

```
module.exports = {
  mongoURI:
    "mongodb+srv://lmolero:1126254560@cluster0.bpms5.mongodb.net/myFirstDatabase?retryWrites=true&w=majority"
};
```

IV. Configurando nuestro servidor con Node.js y Express

El flujo básico para la configuración de nuestro servidor es el siguiente.

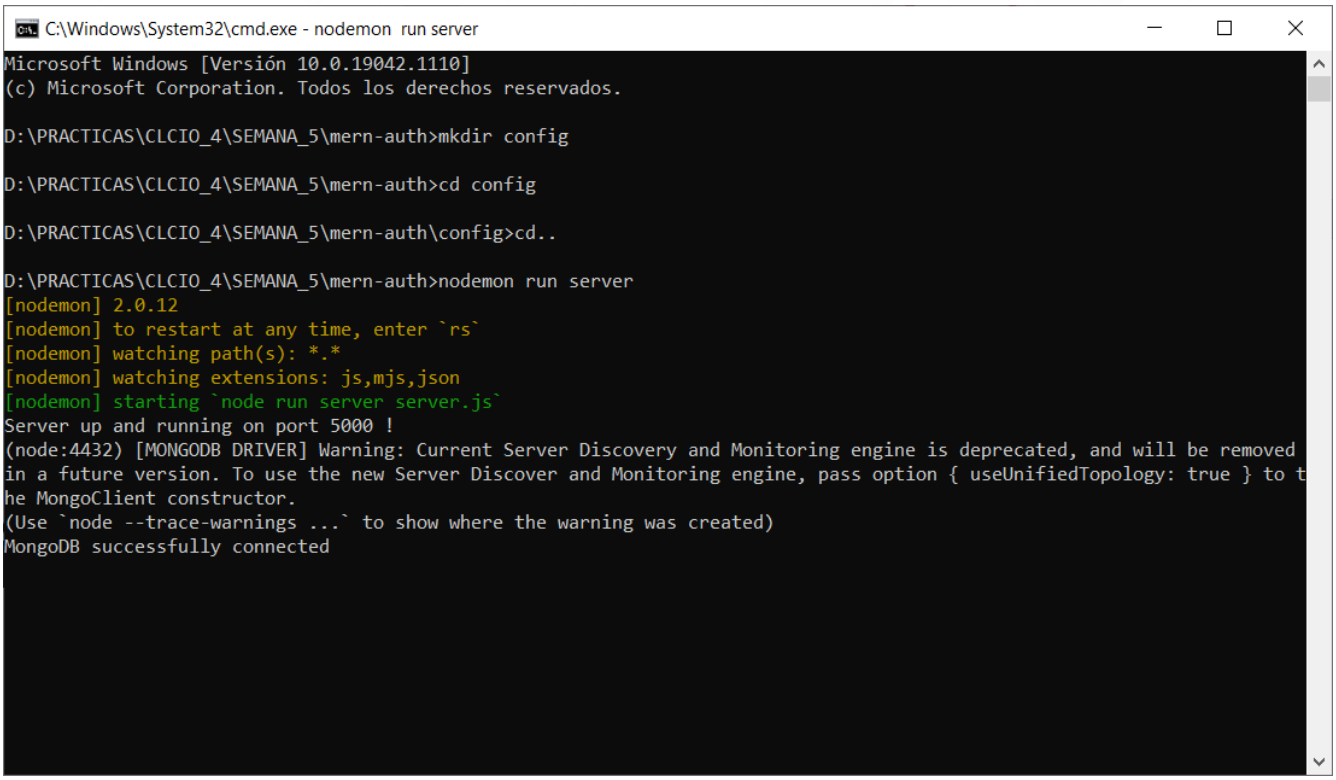
- Ingrese nuestras dependencias requeridas (a saber, express, mongoose y bodyParser)
- Inicialice nuestra aplicación usando express ()
- Aplicar la función de middleware para bodyparser para que podamos usarla
- Extraiga nuestro MongoURI de nuestro archivo keys.js y conéctese a nuestra base de datos MongoDB
- Configure el puerto para que se ejecute nuestro servidor y haga que nuestra aplicación escuche en este puerto

Creemos el archivo “server.js” en la raíz de nuestro proyecto y coloquemos el siguiente código:

```
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 you choose
to deploy the app there
app.listen(port, () => console.log(`Server up and running on port ${port} !`));
```

Ejecute nodemon run server en la raíz de nuestro proyecto y debería aparecer lo siguiente:

```
nodemon run server
```



Intente cambiar el mensaje "Servidor en funcionamiento ..." en su archivo, presione guardar y debería ver que su servidor se reinicia automáticamente.

V. Configurar nuestro esquema de base de datos

Creemos una carpeta “models” para definir nuestro esquema de usuario. Dentro de “models”, cree un archivo User.js, de la forma:

Cree el directorio “models”

```
mkdir models
```

Cambie a ese directorio:

```
cd models
```

Cree el archivo User.js y dentro del archivo:

- Extraiga nuestras dependencias requeridas
- Crear un esquema para representar a un usuario, definiendo campos y tipos como objetos del esquema.

- Exportar el modelo para que podamos acceder a él fuera de este archivo

Coloquemos lo siguiente en nuestro archivo 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
  },
  password: {
    type: String,
    required: true
  },
  date: {
    type: Date,
    default: Date.now
  }
});
module.exports = User = mongoose.model("users", UserSchema);
```

VI. Configurar la validación de formularios

Antes de configurar nuestras rutas, creemos un directorio para la validación de entrada y creemos un archivo register.js y login.js para la validación de cada ruta

Cree el directorio “validation” en la raíz de nuestro proyecto

```
mkdir validation
```

Cambie a ese directorio:

```
cd validation
```

Nuestro flujo de validación para nuestro archivo register.js será el siguiente:

- Extraiga el validador y las dependencias vacías
- Exporte la función validateRegisterInput, que toma datos como un parámetro.
- Instancia de nuestro objeto de errores
- Convierta todos los campos vacíos en una cadena vacía antes de ejecutar las comprobaciones de validación (el validador solo funciona con cadenas)

- Compruebe si hay campos vacíos, formatos de correo electrónico válidos, requisitos de contraseña y confirme la igualdad de contraseña mediante las funciones de validación
- Devuelve nuestro objeto de errores con todos y cada uno de los errores contenidos, así como un booleano isValid que verifica si tenemos algún error.

Dentro de la carpeta “validation” creemos el archivo “register.js” y coloquemos lo siguiente:

```
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.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";
  }
// 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)
  };
};
```

Nuestra validación para nuestro login.js sigue un flujo idéntico al anterior, aunque con diferentes campos.

Ahora, dentro de la carpeta “validation” creemos el archivo “login.js” y coloquemos lo siguiente:

```
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)
  };
};
```

VII. Configurando nuestras rutas API

Ahora que tenemos la validación manejada, creemos una nueva carpeta para nuestras rutas API y creemos un archivo users.js para el registro e inicio de sesión.

Cree el directorio “routes” en la raíz del proyecto

```
mkdir routes
```

Cambie a ese directorio:

```
cd routes
```

Cree el directorio “api”

```
mkdir api
```

Cambie a ese directorio:

```
cd api
```

Ahora, creamos el archivo “users.js” y en su parte superior ingresemos nuestras dependencias requeridas y carguemos nuestras validaciones de entrada y modelo de usuario, de la forma:

```
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");
```

Crear endpoint de registro

Para nuestro punto final de registro

- Extraiga los errores y las variables isValid de nuestra función validateRegisterInput (req.body) y verifique la validación de entrada
- Si la entrada es válida, use User.findOne () de MongoDB para ver si el usuario ya existe
- Si el usuario es un nuevo usuario, complete los campos (nombre, correo electrónico, contraseña) con los datos enviados en el cuerpo de la solicitud.
- Utilice bcryptjs para codificar la contraseña antes de almacenarla en su base de datos

Coloquemos lo siguiente en nuestro archivo users.js para nuestra ruta de registro.

```
// @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,
        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));
        });
      });
    }
  });
});
```

Configurar passport

En su directorio “config” cree el archivo passport.js.

Antes de configurar el passport, agreguemos lo siguiente a nuestro archivo keys.js.

```
module.exports = {
  mongoURI:
    "mongodb+srv://lmolero:<password>@cluster0.bpms5.mongodb.net/myFirstDatabase?retryWrites=true&w=majority",
  secretOrKey: "secret",
};
```

Volver a passport.js. Puede leer más sobre la estrategia passport-jwt en el enlace a continuación.

http://www.passportjs.org/packages/passport-jwt/

Hace un gran trabajo desglosando cómo se construye la estrategia de autenticación JWT, explicando los parámetros requeridos, variables y funciones como opciones, secretOrKey, jwtFromRequest, verify y jwt_payload.

Coloquemos lo siguiente en nuestro archivo 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));
    })
  );
};
```

Además, tenga en cuenta que jwt_payload se enviará a través de nuestro endpoint de inicio de sesión a continuación.

Crear el punto final de inicio de sesión

Para nuestro punto final de inicio de sesión,

- Extraiga los errores y las variables isValid de nuestra función validateLoginInput (req.body) y verifique la validación de entrada
- Si la entrada es válida, use User.findOne () de MongoDB para ver si el usuario existe
- Si el usuario existe, use bcryptjs para comparar la contraseña enviada con la contraseña hash en nuestra base de datos
- Si las contraseñas coinciden, cree nuestra carga útil JWT
- Firme nuestro jwt, incluida nuestra carga útil, keys.secretOrKey de keys.js, y establezca un tiempo expiresIn (en segundos)
- Si tiene éxito, agregue el token a una cadena Bearer (recuerde que en nuestro archivo passport.js, setopts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken ();)

Ahora, agreguemos lo siguiente en nuestro archivo “users.js” ubicado en la carpeta “routes/api” para nuestra ruta de inicio de sesión.

```
// @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;
```

Introduciendo nuestras rutas en nuestro archivo server.js

Actualice el archivo “server.js” con el siguiente código

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

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

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));

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

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

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

const port = process.env.PORT || 5000;
app.listen(port, () => console.log(`Server up and running on port ${port} !`));
```

VIII. Probando nuestras rutas API usando Postman.

Ver “03_Testing_API_Postman_CRUD_3”

Con esto, finalizamos nuestro BackEnd.

Creando nuestra interfaz y configurando Redux

En esta parte

- Configura nuestra interfaz usando create-react-app
- Cree componentes estáticos para nuestras páginas Navbar, Landing, Login y Register
- Configurar Redux para la gestión del estado global

Instale las extensiones de Chrome React Developer Tools y Redux DevTools

A continuación, los enlaces para instalar las extensiones de React y Redux en Google Chrome:

React:

https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

Redux

https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd/related?hl=en

I. Configurando nuestra interfaz

1. Edite el archivo root package.json.

Edite el objeto "scripts" a lo siguiente en el package.json de nuestro servidor.

```
"scripts": {
    "client-install": "npm install --prefix client",
    "start": "node server.js",
    "server": "nodemon server.js",
    "client": "npm start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\""
  },
```

Usaremos simultáneamente para ejecutar nuestro backend y frontend (cliente) al mismo tiempo. Usaremos `npm run dev` para ejecutar este comando más adelante.

2. Aplique andamiaje a nuestro cliente con create-react-app

Usaremos create-react-app para configurar nuestro cliente. Primero, si aún no lo ha instalado, ejecute el siguiente comando para instalar create-react-app globalmente.

```
npm i -g create-react-app
```

Ahora, en el directorio raíz del proyecto ejecute create-react-app dentro de él.

```
npx create-react-app client
```

3. Cambie nuestro package.json dentro de nuestro directorio de clientes

Cuando hacemos solicitudes desde React con axios, queremos poder hacer lo siguiente:

```
axios.post('/api/users/register');
```

Para lograr esto, agregue lo siguiente debajo del objeto "scripts" en el package.json de nuestro cliente.

```
"proxy": "http://localhost:5000",
```

4. Dentro de la carpeta “client”, instale las siguientes dependencias usando npm

```
npm i axios classnames jwt-decode react-redux react-router-dom redux redux-thunk
```

Una breve descripción de cada paquete y la función que cumplirá.

- axios: cliente HTTP basado en promesas para realizar solicitudes a nuestro backend
- classnames: utilizado para clases condicionales en nuestro JSX
- jwt-decode: se usa para decodificar nuestro jwt para que podamos obtener datos de usuario de él
- react-redux: nos permite usar Redux con React
- react-router-dom: se utiliza con fines de enrutamiento
- redux: se usa para administrar el estado entre componentes (se puede usar con React o cualquier otra biblioteca de vista)
- redux-thunk: middleware para Redux que nos permite acceder directamente al método de despacho para realizar llamadas asincrónicas desde nuestras acciones

El package.json de su cliente debería verse así.

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "axios": "^0.18.0",
    "classnames": "^2.2.6",
    "jwt-decode": "^2.2.0",
    "react": "^16.6.3",
    "react-dom": "^16.6.3",
    "react-redux": "^5.1.1",
    "react-router-dom": "^4.3.1",
    "react-scripts": "2.1.1",
    "redux": "^4.0.1",
    "redux-thunk": "^2.3.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "proxy": "http://localhost:5000",
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": [
    ">0.2%",
    "not dead",
    "not ie <= 11",
    "not op_mini all"
  ]
}
```

Ejecute el siguiente comando en la raíz del proyecto y compruebe si tanto el servidor como el cliente se ejecutan al mismo tiempo y con éxito.

```
npm run dev
```

5. Limpie nuestra aplicación React eliminando archivos y códigos innecesarios

- Eliminar logo.svg en client / src
- Elimina la importación de logo.svg en App.js
- Elimine todo el CSS en App.css (mantendremos la importación en App.js en caso de que desee agregar su propio CSS global aquí)
- Borre el contenido en el div principal en App.js y reemplácelo con un <h1> por ahora

No debería tener errores y su App.js debería verse así en este momento.

```
import React, { Component } from "react";
import "./App.css";
class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hello</h1>
      </div>
    );
  }
}
export default App;
```

6. Instale Materialize.css editando nuestro index.html en client/public

A continuación, en el siguiente enlace se encuentra el instalador de Materialize.css

https://materializecss.com/getting-started.html

Navegue a la parte CDN y tome las etiquetas CSS y Javascript.
En client/public/index.html, agregue la etiqueta CSS encima de la etiqueta <head> y el script JS justo encima de la etiqueta </body>. Cambiemos el <title> de "React App" al nombre de su aplicación mientras estemos aquí también (esto es lo que se muestra en la barra de herramientas cuando la aplicación se está ejecutando).
También agreguemos la siguiente etiqueta CSS debajo de nuestra etiqueta Materialise para acceder a los íconos de materiales de Google.

Su index.html ahora debería verse así.

```
<!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/materializ
e/1.0.0/css/materialize.min.css">
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="styl
esheet">
```

```
    <title>MERN Auth App</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/mater
ialize.min.js"></script>
  </body>
</html>
```

II. Creando nuestros componentes estáticos

En nuestro directorio src, creemos una carpeta para nuestros componentes y creemos una carpeta de diseño dentro de ella para alojar nuestros componentes de " layout " compartidos en toda la aplicación (p. Ej., Página de destino, barra de navegación).

Ahora, cree el directorio “components” dentro de la carpeta /client/src

```
mkdir components
```

Cambie a ese directorio:

```
cd components
```

Ahora, cree el directorio “layout”

```
mkdir layout
```

Cambie a ese directorio:

```
cd layout
```

1. Componentes comunes: barra de navegación y aterrizaje

Ahora, crearemos el “Navbar.js” dentro del directorio “/client/src/components/layout/”, coloque lo siguiente:

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
class Navbar extends Component {
  render() {
    return (
      <div className="navbar-fixed">
        <nav className="z-depth-0">
          <div className="nav-wrapper white">
            <Link
              to="/"
```

```
              style={{
                fontFamily: "monospace"
              }}
              className="col s5 brand-logo center black-text"
            >
              <i className="material-icons">code</i>
              MERN
            </Link>
          </div>
        </nav>
      </div>
    );
  }
}
export default Navbar;
```

Crearemos el “Landing.js”, coloque lo siguiente:

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
class Landing extends Component {
  render() {
    return (
      <div style={{ height: "75vh" }} className="container valign-wrapper">
        <div className="row">
          <div className="col s12 center-align">
            <h4>
              <b>Build</b> a login/auth app with the{" "}
              <span style={{ fontFamily: "monospace" }}>MERN</span> stack from
              scratch
            </h4>
            <p className="flow-text grey-text text-darken-1">
              Create a (minimal) full-stack app with user authentication via
              passport and JWTs
            </p>
            <br />
            <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-3"
              >
                Register
              </Link>
            </div>
            <div className="col s6">
              <Link
                to="/login"
                style={{
                  width: "140px",
```

```
                  borderRadius: "3px",
                  letterSpacing: "1.5px",
                }}
                className="btn btn-large btn-flat waves-effect white black-text"
              >
                Log In
              </Link>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
export default Landing;
```

Finalmente, importemos nuestros componentes Navbar y Landing en nuestro archivo App.js y agréguelos a nuestro render ().

```
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;
```

2. Componentes de autenticación: registrarse e iniciar sesión

Ahora, creemos el directorio “auth” en la carpeta “mern-auth/client/src/components/” para nuestros componentes de autenticación y creemos los archivos Login.js y Register.js dentro de él.

Ahora, cree el directorio “auth”.

```
mkdir auth
```

Cambie a ese directorio:

```
cd auth
```

Los formularios funcionan de manera un poco diferente en React. Cada elemento de formulario tiene un evento onChange que vincula su valor al estado de nuestros componentes En nuestro evento onSubmit, usaremos e.preventDefault () para evitar que la página se vuelva a cargar cuando se haga clic en el botón Enviar.
Una nota al margen sobre la desestructuración en React: const {errors} = this.state; es lo mismo que hacer const errors = this.state.errors ;. Es menos detallado y se ve más limpio.

Ahora, creamos el archivo “Register.js” y coloquemos lo siguiente:

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
class Register extends Component {
  constructor() {
    super();
    this.state = {
      name: "",
      email: "",
      password: "",
      password2: "",
      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,
      password: this.state.password,
      password2: this.state.password2
    };
console.log(newUser);
  };
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>
          <form noValidate onSubmit={this.onSubmit}>
            <div className="input-field col s12">
              <input
                onChange={this.onChange}
                value={this.state.name}
                error={errors.name}
                id="name"
                type="text"
              />
              <label htmlFor="name">Name</label>
            </div>
            <div className="input-field col s12">
              <input
                onChange={this.onChange}
                value={this.state.email}
                error={errors.email}
                id="email"
                type="email"
              />
              <label htmlFor="email">Email</label>
            </div>
            <div className="input-field col s12">
              <input
                onChange={this.onChange}
                value={this.state.password}
                error={errors.password}
                id="password"
                type="password"
              />
              <label htmlFor="password">Password</label>
            </div>
            <div className="input-field col s12">
              <input
                onChange={this.onChange}
                value={this.state.password2}
                error={errors.password2}
                id="password2"
                type="password"
              />
              <label htmlFor="password2">Confirm Password</label>
            </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-
3"
              >
                Sign up
              </button>
```

```
            </div>
          </form>
        </div>
      </div>
    </div>
  );
 }
}
export default Register;
```

Nuestro componente de inicio de sesión se verá similar a nuestro componente de registro. Ahora, creamos el archivo “Login.js” y coloquemos lo siguiente:

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
class Login extends Component {
  constructor() {
    super();
    this.state = {
      email: "",
      password: "",
      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
    };
console.log(userData);
  };
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"
                />
                <label htmlFor="email">Email</label>
              </div>
              <div className="input-field col s12">
                <input
                  onChange={this.onChange}
                  value={this.state.password}
                  error={errors.password}
                  id="password"
                  type="password"
                />
                <label htmlFor="password">Password</label>
              </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-
3"

                >
                  Login
                </button>
              </div>
            </form>
          </div>
        </div>
      </div>
    );
  }
}
export default Login;
```

Nuestros componentes no se mostrarán hasta que definamos nuestro inicio de sesión y registremos rutas en nuestro App.js usando react-router-dom.

3. Configuración de React Router en nuestra App.js

Definiremos nuestras rutas de enrutamiento usando react-router-dom. Agregue lo siguiente a su App.js. Asegúrese de envolver la etiqueta con una etiqueta de

enrutador de inicio y cierre. También incorporemos nuestros componentes de registro e inicio de sesión y creemos rutas para cada uno de ellos.

Escribamos lo siguiente en App.js.

```
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;
```

Todavía no hemos vinculado nuestra interfaz con nuestro backend, por lo que en realidad no estamos registrando ni iniciando sesión en usuarios, pero enviaremos estos objetos a Redux (y, a su vez, a nuestro backend) para completar esas acciones.

III. Configuración de Redux para la gestión del estado

La comunidad de desarrollo está de acuerdo en gran medida en que Redux es bastante necesario para cualquier aplicación a gran escala, ya que administrar el estado entre muchos componentes de React probablemente resultaría una pesadilla. En lugar de pasar el estado de un componente a otro, Redux proporciona una única fuente de verdad que puede enviar a cualquiera de sus componentes.

1. Realice las siguientes adiciones en negrita a App.js

Asegúrese de envolver toda su declaración de devolución con una etiqueta <Provider store = {store}> (y una etiqueta de cierre).

Editemos App.js con el siguiente código:

```
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;
```

Aún no hemos definido nuestro store

2. Configurar nuestra estructura de archivos Redux

En src,

- Crea un archivo store.js
- Crea directorios “actions” y “reducers”

En reducers

- Cree archivos index.js, authReducer.js y errorReducer.js

En actions,

- Cree archivos authActions.js y types.js

3. Montaje de nuestra tienda

createStore () crea una tienda Redux que contiene el árbol de estado completo de su aplicación. Solo debe haber una tienda en su aplicación.
Coloque lo siguiente en store.js. Pasaremos un rootReducer vacío por ahora como el primer parámetro para createStore () ya que aún no hemos creado nuestros reductores.

Agregue el siguiente código en store.js

```
import { createStore, applyMiddleware, compose } from "redux";
import thunk from "redux-thunk";
const initialState = {};
const middleware = [thunk];
const store = createStore(
  () => [],
  initialState,
  compose(
    applyMiddleware(...middleware),
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
  )
);
export default store;
```

4. Definición de nuestras actions

Una interacción (como un clic en un botón o el envío de un formulario) en nuestros componentes de React activará una acción y, a su vez, enviará una acción a nuestro store. En nuestra carpeta “actions”, coloquemos lo siguiente en types.js.

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

5. Creando nuestros reducers

Los “reducers“ son funciones puras que especifican cómo debe cambiar el estado de la aplicación en respuesta a una acción. Los “reducers“ responden con el nuevo estado, que se pasa a nuestro store y, a su vez, a nuestra interfaz de usuario.

Nuestro flujo de reductores será el siguiente.

- Importar todas nuestras acciones de nuestro archivo types.js
- Definir nuestro initialState
- Defina cómo debe cambiar el estado en función de las acciones con una declaración de cambio

I. Creando nuestro authReducer.js

Coloquemos lo siguiente en nuestro authReducer.js.

```
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;
  }
}
```

II. Creando nuestro errorReducer.js

Coloquemos lo siguiente en nuestro errorReducer.js.

```
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;
  }
}
```

III. Creando nuestro rootReducer en index.js

Usaremos MixedReducers de redux para combinar nuestro authReducer y errorReducer en un rootReducer.

Definamos nuestro rootReducer agregando lo siguiente a nuestro index.js.

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

```
});
```

6. Configuración de nuestro auth token

Antes de comenzar a crear nuestras acciones, creemos un directorio de utils dentro de src, y dentro de él, un archivo setAuthToken.js.

Ahora, cree el directorio “utils”.

```
mkdir utils
```

Cambie a ese directorio:

```
cd utils
```

Usaremos esto para configurar y eliminar el encabezado de Autorización para nuestras solicitudes axios dependiendo de si un usuario ha iniciado sesión o no.

Ahora, creamos el archivo “setAuthToken.js” dentro de la carpeta “mern-auth/client/src/utils/” y coloquemos lo siguiente en

```
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;
```

7. Creando nuestras actions

Nuestro flujo general de nuestras acciones será el siguiente.

- Importar dependencias y definiciones de acciones de types.js
- Use axios para hacer HTTPRequests dentro de cierta acción
- Utilice el envío para enviar acciones a nuestros reductores

Coloquemos lo siguiente en el archivo “authActions.js” de nuestra carpeta “mern-auth/client/src/actions/”

```
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({}));
```

```
};
```

8. Colocando nuestro rootReducer en store.js

Coloque el siguiente código en “store.js” de nuestra carpeta “/src/”

```
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;
```

Vincular Redux con componentes de React

En esta parte final,

- Vincular Redux a nuestros componentes
- Mostrar errores de nuestro backend en nuestros formularios React
- Cree rutas protegidas (páginas a las que solo ciertos usuarios pueden acceder según su autenticación)
- Mantener a un usuario conectado cuando actualice o abandone la página (en otras palabras, hasta que cierre la sesión o caduque el jwt)

1. Vinculando Redux a nuestro componente de registro y mostrando errores en nuestro formulario

I. Usando connect () de react-redux

connect () hace precisamente eso; conecta nuestros componentes React a nuestro Redux store proporcionada por el componente Provider.

Tenemos que modificar nuestro export default Register; en la parte inferior de Register.js:

```
export default connect(
  mapStateToProps,
  { registerUser }
)(withRouter(Register));
```

También puede notar que envolvemos nuestro Registro con un withRouter (). Si bien es fácil redirigir dentro de un componente (simplemente puede decir this.props.history.push ('/ dashboard') por ejemplo), no podemos hacerlo de forma predeterminada dentro de una acción. Para permitirnos redirigir dentro de una acción:

- Usar conRouter de react-router-dom, envolviendo nuestro componente en nuestra exportación conRouter ()
- Agregará un parámetro a this.props.history dentro de nuestra llamada a this.props.registerUser (newUser, this.props.history) en nuestro evento onSubmit para que podamos acceder fácilmente a él dentro de nuestra acción (paso iv a continuación)

II. mapStateToProps

mapStateToProps nos permite obtener nuestro estado de Redux y asignarlo a accesorios que podemos usar dentro de los componentes.

Agregaremos lo siguiente sobre nuestra exportación en la parte inferior de Register.js.

```
const mapStateToProps = state => ({
  auth: state.auth,
  errors: state.errors
});
```

Esto nos permite llamar a this.props.auth o this.props.errors dentro de nuestro componente de registro.

III. Definición de propTypes

Dado que no podemos definir tipos en nuestro constructor, se considera una buena convención hacerlo utilizando el paquete prop-types.

```
Register.propTypes = {
  registerUser: PropTypes.func.isRequired,
  auth: PropTypes.object.isRequired,
  errors: PropTypes.object.isRequired
};
```

IV. Atarlo todo junto

A continuación, ubiquemos todo en el archivo Register.js de la ruta “mern-auth\client\src\components\auth\”

```
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: "",
      password: "",
      password2: "",
      errors: {},
    };
  }
  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,
      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>
            <form noValidate onSubmit={this.onSubmit}>
              <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>
        <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>
        <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>
        <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>
        <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-
3"
                >
                  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));
```

V. Vinculando Redux a nuestro componente de inicio de sesión y mostrando errores en nuestro formulario

Ahora, actualicemos el siguiente código en nuestro archivo Login.js.

```
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: {},
    };
  }
  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 component,
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-
3"
                >
                  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);
```

En este momento, cuando el usuario inicia sesión, la aplicación nos redirige a una página en blanco "/panel de control" según la primera declaración condicional de nuestro método de

ciclo de vida componentWillReceiveProps (nextProps). A continuación, crearemos nuestro componente Panel y lo convertiremos en una ruta privada para que solo un usuario que haya iniciado sesión pueda verlo.

Creación de nuestro componente Tablero para cuando los usuarios inician sesión

En nuestro directorio de “mern-auth/client/src/components/”, creemos un directorio “dashboard” y dentro de él, un archivo Dashboard.js.

Cree el directorio “dashboard” dentro de la carpeta “component”:

```
mkdir dashboard
```

Cambie a ese directorio:

```
cd dashboard
```

Ahora, creemos el archivo Dashboard.js y dentro de él, coloquemos el siguiente código:

```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { logoutUser } from "../../actions/authActions";
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 center-align">
            <h4>
              <b>Hey there,</b> {user.name.split(" ")[0]}
              <p className="flow-text grey-text text-darken-1">
                You are logged into a full-stack{" "}
                <span style={{ fontFamily: "monospace" }}>MERN</span> app 👏
              </p>
            </h4>
            <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-3"
```

```
            >
              Logout
            </button>
          </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);
```

Creación de rutas protegidas

No existe una forma estándar de crear rutas protegidas en React. Usaremos la lógica descrita en la siguiente publicación de Tyler McGinnis para crear rutas autenticadas (rutas a las que solo ciertos usuarios pueden acceder en función de su estado de autenticación).

Para mayor información, visita el siguiente enlace:

https://ui.dev/react-router-v4-protected-routes-authentication/

En nuestro directorio de “mern-auth/client/src/components/”, creemos un directorio y un archivo para nuestra ruta privada.

Cree el directorio “private-route” dentro de la carpeta “component”:

```
mkdir private-route
```

Cambie a ese directorio:

```
cd private-route
```

Ahora, creamos el archivo “PrivateRoute.js” y agregamos el siguiente código:

```
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);
```

Uniéndolo todo en App.js
En esto, lo que haremos

- Verifique localStorage en busca de un token para mantener al usuario conectado incluso si cierra o actualiza la aplicación (por ejemplo, hasta que cierre la sesión o el token caduque)
- Ingrese nuestro componente Dashboard y defínalo como PrivateRoute

Ahora, actualice App.js. con el siguiente código:

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

¡Un último paso!

No tendría sentido que los usuarios registrados pudieran acceder a las páginas /login y /register. Si un usuario que ha iniciado sesión navega a cualquiera de estos, debemos redirigirlo inmediatamente al panel de control.

Para lograr esto, agregue el siguiente método de ciclo de vida debajo del constructor en Register.js en la ruta “mern-auth/client/src/components/auth/Register.js/”

```
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");
  }
}
```

Y agregue el mismo método de ciclo de vida debajo del constructor en Login.js en la ruta “mern-auth/client/src/components/auth/Login.js/”

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

Para desplegar todo el proyecto:

Ejecute el siguiente comando en la raíz del proyecto

```
npm run dev
```