



**CICLO 4a**

[FORMACIÓN POR CICLOS]

# Desarrollo de **APLICACIONES WEB**

## Tutorial de aplicación MERN



**UNIVERSIDAD  
DE ANTIOQUIA**

Facultad de Ingeniería



## Tutorial

Con este tutorial podrás utilizar una aplicación usando Stack MERN. Con esta aplicación los usuarios podrán registrarse, iniciar sesión y cerrar sesión.

Inicializaremos nuestro Backend usando node, configuraremos la base de datos en MongoDB usando mLab, configuraremos el servidor con node.js y express y generaremos dos rutas para la API.

Este tutorial se desarrolló con el editor VSCode y la última versión de Node.js.



## Mongo

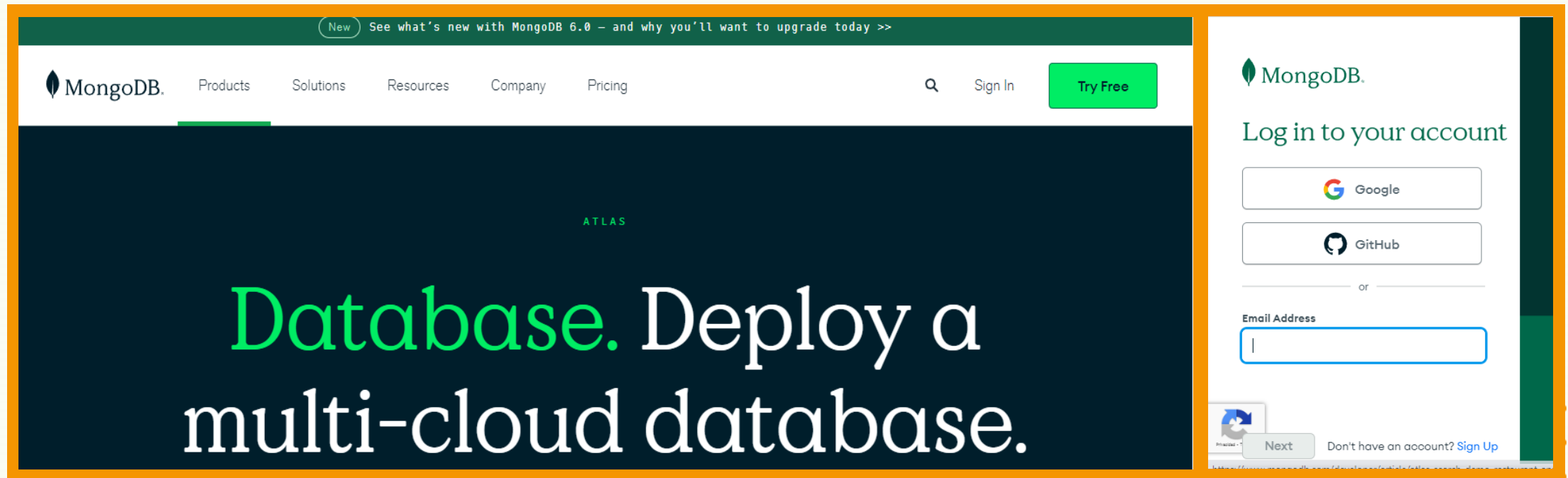
Trabajaremos nuestra base de datos desde mlab con la intención de ver una forma de usar base de datos Mongo desde un servicio en nube, y daremos las indicaciones para que tomes el servicio gratuito.



Sigue los pasos que se indican a continuación:

1

Ingresa a <https://www.mongodb.com/atlas/database> o a <https://mlab.com/>, crea una cuenta si aún no tienes una haciendo clic en Try Free o, si ya tienes una cuenta, ingresa haciendo clic en Sign in.

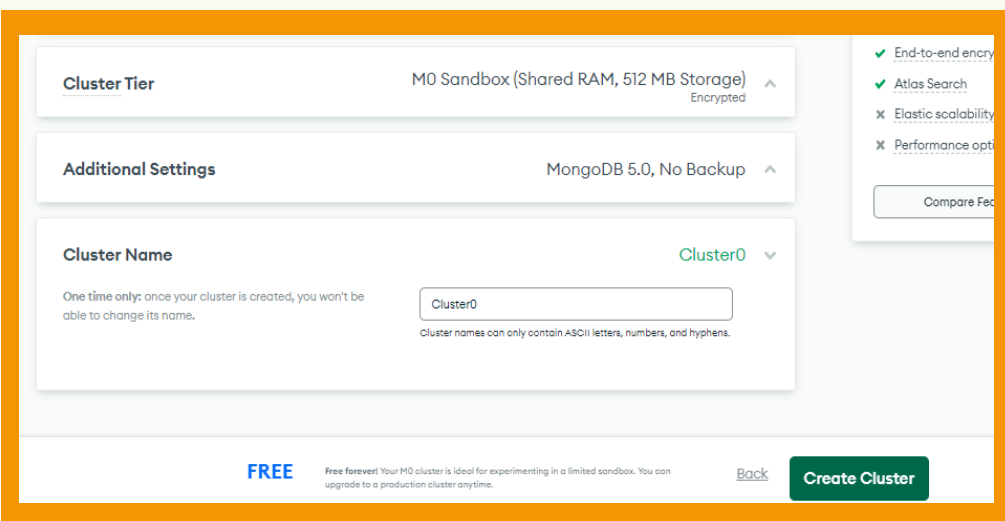


2

Crea una implementación nueva de MongoDB haciendo clic en el botón Build a Database y genera la siguiente configuración:

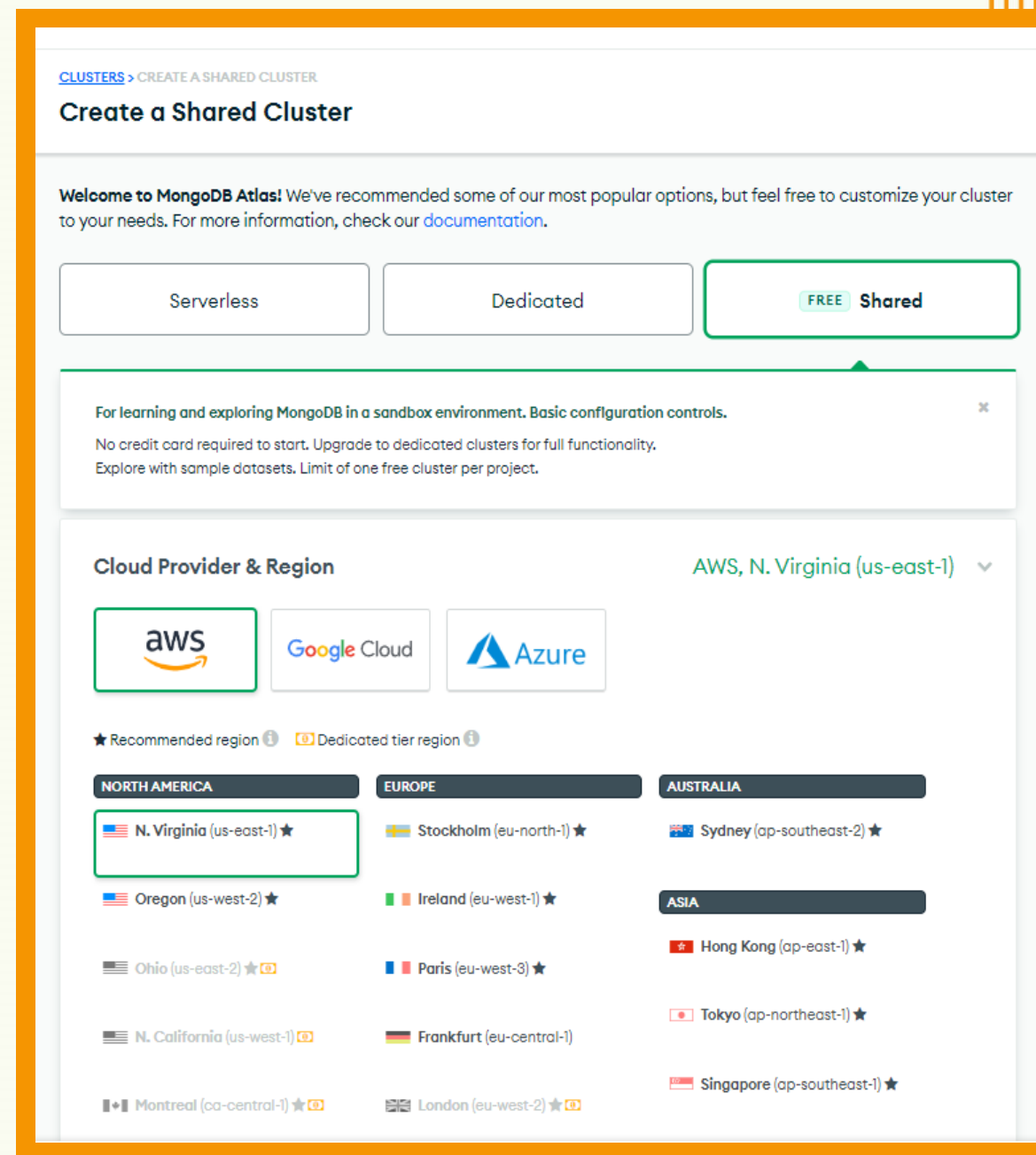
The screenshot displays the MongoDB Atlas interface for 'MARCELA'S ORG - 2022-08-31 > PROJECT 0'. The left sidebar contains a 'DEPLOYMENT' section with 'Database' selected, and other sections like 'Data Lake', 'DATA SERVICES', 'SECURITY', and 'New On Atlas'. The main area is titled 'Database Deployments' and shows a 'Build a Database' button circled in red. A yellow warning banner at the top of the main content area states: 'Current IP Address not added. You will not be able to connect to databases from this address.' with a close button and an 'Add Current IP Address' button. The top navigation bar includes 'MARCELA'S ...', 'Access Manager', 'Billing', 'All Clusters', 'Get Help', and 'MARCELA'.

Selecciona Shared como tipo de plan y AWS como proveedor de nube, dejando la región de AWS por defecto. Por último, asigna un nombre a tu base de datos y dale "Create Cluster" (no te preocupes, es gratis):



The screenshot shows the 'Create Cluster' form in MongoDB Atlas. It includes the following sections:

- Cluster Tier:** M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted.
- Additional Settings:** MongoDB 5.0, No Backup.
- Cluster Name:** Cluster0. A note states: "One time only: once your cluster is created, you won't be able to change its name. Cluster names can only contain ASCII letters, numbers, and hyphens."
- Features:** A list of features with checkboxes: End-to-end encryption (checked), Atlas Search (checked), Elastic scalability (unchecked), and Performance optimization (unchecked).
- Footer:** A 'FREE' badge, a note about the M0 cluster being for experimentation, a 'Back' link, and a 'Create Cluster' button.



The screenshot shows the 'Create a Shared Cluster' page in MongoDB Atlas. It includes the following sections:

- Header:** 'CLUSTERS > CREATE A SHARED CLUSTER' and 'Create a Shared Cluster'.
- Welcome Message:** "Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our [documentation](#)."
- Plan Selection:** Three buttons: 'Serverless', 'Dedicated', and 'FREE Shared' (highlighted with a green border).
- Information Box:** "For learning and exploring MongoDB in a sandbox environment. Basic configuration controls. No credit card required to start. Upgrade to dedicated clusters for full functionality. Explore with sample datasets. Limit of one free cluster per project."
- Cloud Provider & Region:** A section with a dropdown menu set to 'AWS, N. Virginia (us-east-1)'. Below are three buttons for 'aws', 'Google Cloud', and 'Azure' (the 'aws' button is highlighted with a green border).
- Region Selection:** A grid of regions categorized by continent: NORTH AMERICA, EUROPE, AUSTRALIA, and ASIA. The 'N. Virginia (us-east-1)' region is highlighted with a green border.

3

Ahora vamos a crear nuestro usuario para acceder a nuestras bases de datos. Ubícate en la pestaña “Database Access”, haz clic en “Add new database user”, indica su usuario y contraseña, haz clic en “add built role” para dar los permisos de lectura y escritura, y haz clic en “add user”:

Billing

App Services Charts

**Authentication Method**

Password Certificate AWS IAM (MongoDB 4.4 and up)

MongoDB uses [SCRAM](#) as its default authentication method.

**Password Authentication**

jmoreno

..... SHOW

Autogenerate Secure Password Copy

**Database User Privileges**

Configure role based access control by assigning database users a mix of one built-in role, multiple custom roles, and multiple specific privileges. A user will gain access to all actions within the roles assigned to them, not just the actions those roles share in common. **You must choose at least one role or privilege.** [Learn more about roles.](#)

**Built-in Role**  
Select one [built-in role](#) for this user. 0 SELECTED

Add Built In Role

**Custom Roles**  
Select your [pre-defined custom role\(s\)](#). Create a custom role in the [Custom Roles](#) tab.

**Built-in Role** 1 SELECTED

Select one [built-in role](#) for this user.

Read and write to any database

**Custom Roles**  
Select your [pre-defined custom role\(s\)](#). Create a custom role in the [Custom Roles](#) tab.

**Specific Privileges**  
Select multiple privileges and what database and collection they are associated with. Leaving collection blank will grant this role for all collections in the database.

**Restrict Access to Specific Clusters/Federated Database Instances**  
Enable to specify the resources this user can access. By default, all resources in this project are accessible. ☐

**Temporary User**  
This user is temporary and will be deleted after your specified duration of 6 hours, 1 day, or 1 week. ☐

Cancel Add User

4

Ahora agrega una IP address para tu lista de acceso haciendo clic en “Network Access” y luego clic en la opción “ALLOW ACCESS FROM ANYWHERE”. Asegúrate de que luzca de la siguiente forma antes de confirmar:

The screenshot displays the Databricks user interface for configuring network access. The background shows the 'Network Access' section with tabs for 'IP Access List', 'Peering', and 'Private Endpoints'. A modal dialog titled 'Add IP Access List Entry' is open in the foreground. The dialog contains the following elements:

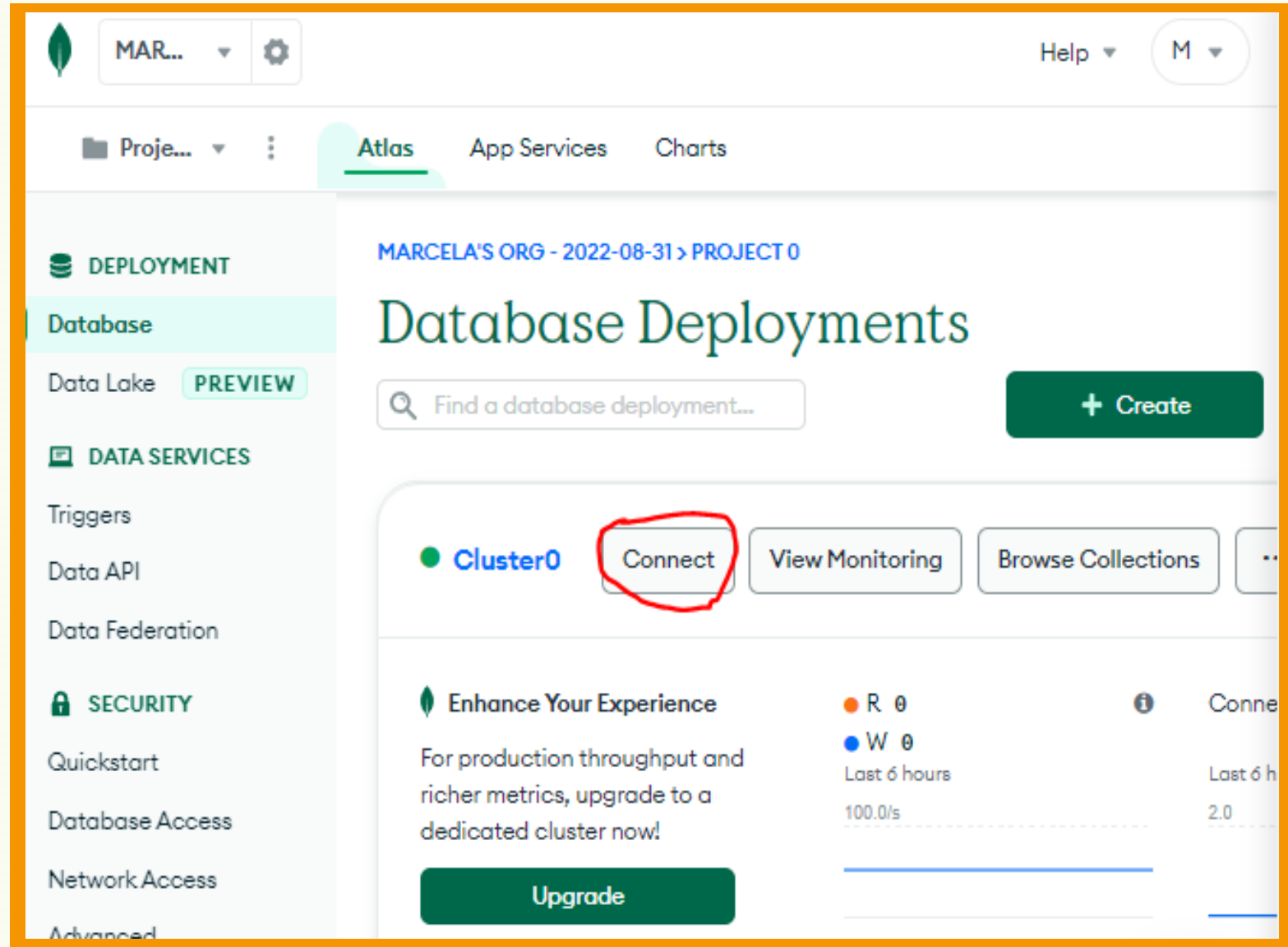
- Header:** 'Add IP Access List Entry' with a close button (X).
- Text:** 'Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. [Learn more.](#)'
- Buttons:** Two buttons at the top: 'ADD CURRENT IP ADDRESS' and 'ALLOW ACCESS FROM ANYWHERE' (which is highlighted with a blue border).
- Form Fields:**
  - Access List Entry:** A text input field containing '0.0.0.0/0'.
  - Comment:** A text input field containing 'Optional comment describing this entry'.
- Footer:**
  - A toggle switch labeled 'This entry is temporary and will be deleted in' followed by a dropdown menu set to '6 hours'.
  - 'Cancel' and 'Confirm' buttons.

Below the dialog, the main interface shows a section titled 'Add an IP address' with the subtitle 'Configure which IP addresses can access your cluster.' and a large green 'Add IP Address' button.



5

Finalmente, vamos a conectarnos a la base de datos. Haz clic en el menú de la parte izquierda “Database”, y a la derecha ubica el botón “connect”:






Posteriormente selecciona “connect your application”:  
Selecciona el driver y la versión; en nuestro caso será el mismo que aparece por defecto, y copia la URI, a saber:


### Connect to Cluster0


✓ Setup connection security > Choose a connection method > Connect


Choose a connection method [View documentation](#)

Get your pre-formatted connection string by selecting your tool below.

**Connect with the MongoDB Shell**  
Interact with your cluster using MongoDB's interactive Javascript interface >

**Connect your application**  
Connect your application to your cluster using MongoDB's native drivers >

**Connect using MongoDB Compass**  
Explore, modify, and visualize your data with MongoDB's GUI >

**Connect using VS Code**  
Connect to a MongoDB host in Visual Studio Code >

Go Back

Close

### Connect to Cluster0

✓ Setup connection security > ✓ Choose a connection method > Connect

- Select your driver and version  
DRIVER: Node.js  
VERSION: 4.1 or later
- Add your connection string into your application code  
☐ Include full driver code example  

```
mongodb+srv://jmoreno:<password>@cluster0.8spvymv.mongodb.net/?retryWrites=true&w=majority
```

Replace **<password>** with the password for the **jmoreno** user. Ensure any option params are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back

Close

Reemplaza <password> con la contraseña que indicaste cuando creaste tu usuario.

# Express

Vamos inicialmente a preparar nuestro proyecto. Creamos una carpeta con el nombre de tu proyecto, y dentro vamos a hacer el primer microservicio que será la carpeta denominada auth:

- 1 Instalamos node con la línea `npm install npm -g`,
- 2 Para crear nuestro backend inicializamos nuestro proyecto con la línea `npm init`. Una vez iniciado debemos visualizar en nuestra carpeta del proyecto el archivo `package.json`.
- 3 Instala las siguientes dependencias con npm en tu terminal:  
`npm i bcryptjs body-parser concurrently express is-empty jsonwebtoken mongoose passport passport-jwt validator`

Verifica que tu  
package.json luzca  
como este:



```
{
  "name": "auth",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" &&
exit 1",
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "body-parser": "^1.20.0",
    "concurrently": "^7.3.0",
    "express": "^4.18.1",
    "is-empty": "^1.2.0",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^6.5.4",
    "passport": "^0.6.0",
    "passport-jwt": "^4.0.0",
    "validator": "^13.7.0"
  }
}
```

- 

```
module.exports = {
  mongoURI:
    "mongodb+srv://jmoreno:Jmoreno123@cluster0.8spvymv.mongodb.net/?ret
ryWrites=true&w=majority"
};
```

- 5** En la raíz del proyecto crea un archivo `server.js` y ubica allí el siguiente código:

```
1. const express = require("express");
2. const mongoose = require("mongoose");
3. const bodyParser = require("body-parser");
4. const passport = require("passport");
5. const users = require("../routes/api/users");
6.
7. const app = express();
8. // Bodyparser middleware
9. app.use(
```

Allí estamos ingresando las dependencias express, mongoose y bodyParser. Se inicializa la aplicación usando express(), aplicamos la función del middleware para bodyParser y para extraer nuestro MongoDB que ubicamos en el archivo keys.js. Finalmente, configuramos nuestro puerto.

Verificamos la funcionalidad de nuestra aplicación poniendo en el terminal el código node server.js. Debes visualizar tu terminal así:

```
1.  bodyParser.urlencoded({
2.    extended: false
3.  })
4. );
5. app.use(bodyParser.json());
6. // DB Config
7. const db = require("../config/keys").mongoURI;
8. // Connect to MongoDB
9. mongoose
10. .connect(
11.  db,
12.  { useNewUrlParser: true }
13. )
14. .then(() => console.log("MongoDB successfully
    connected"))
15. .catch(err => console.log(err));
16.
17. // Passport middleware
18. app.use(passport.initialize());
19. // Passport config
20. require("../config/passport")(passport);
21. // Routes
22. app.use("/api/users", users);
23.
24. const port = process.env.PORT || 5000;
25. app.listen(port, () => console.log(`Server up and
    running on port ${port} !`));
```

```
PS D:\Docencia\UDEA\Desarrollo web\Semana 6\MERN\auth> node server.js
Server up and running on port 5000 !
MongoDB successfully connected
█
```

# Configurar base de datos en el proyecto

- 1 Creamos la carpeta models y establecemos allí el archivo denominado user.js. Allí extraemos nuestras dependencias y creamos el esquema para representar a un usuario definiendo los objetos con sus respectivos campos y tipos. Finalmente, exportamos el modelo para poder accederlo desde fuera:

```
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);
```

2

Para configurar la validación de formulario crearemos un archivo llamado register.js y login.js.

En el register.js extraemos el validador y las dependencias. Posteriormente exportamos la función de validación de entrada, instanciamos el objeto a usar para errores, convertimos los campos vacíos en una cadena vacía antes de la comprobación, comprobamos si hay campos vacíos y devolvemos el objeto de error:

```
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)
    };
  };
};
```

3

El flujo para el login.js es idéntico al anterior, aunque con variables distintas:

```
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)
  };
};
```

# Configurar rutas API

Ya que se ha manejado la validación, creamos una nueva carpeta para las rutas y generamos el archivo users.js para el registro e inicio de sesión, y en él ubicamos el siguiente código:

```
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");
```

# Endpoint de registro

Para el endpoint de registro extraemos errores y variables de la función `validateRegisterInput` y verificamos la validación de entrada. Si la entrada es válida, se ingresa a Mongo para validar si existe el usuario. Si es un usuario nuevo, se completan los campos con los datos del cuerpo y se encripta la contraseña antes de almacenarla en la bd.

Ubiquemos el siguiente código en el archivo `user.s` debajo de la declaración de variables:

```
// @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))
16
  .catch(err => console.log(err));
});
});
}
});
});
```

# Configurar Passport

Vamos al archivo keys.js y agregamos la línea `secretOrKey: "secret"`:

```
module.exports = {  
  mongoURI:  
    "mongodb+srv://jmoreno:jmoreno123@cluster0.8spvymv.mongodb.net  
/?retryWrites=true&w=majority",  
  secretOrKey: "secret"  
};
```

Creamos ahora el archivo Passport.js y ubicamos el siguiente código allí:

```
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));
    })
  );
};
```



## Crear endpoint de inicio de sesión

En el código siguiente extraemos los errores y las variables `isValid` de nuestra función `validateLoginInput (req.body)` y verificamos la validación de entrada para ver si el usuario existe. De ser así, usa `bcryptjs` y compara la contraseña enviada con la contraseña hash en nuestra base de datos. Si coinciden, crea nuestra carga útil JWT, firma nuestro jwt, incluida nuestra carga útil, `keys.secretOrKey` de `keys.js`, y establece un tiempo `expiresIn` (en segundos). Si tienes éxito, agrega el token a una cadena `Bearer`. Finalmente, agrega 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 => {
18
```



```
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;
```

Finalmente, introducimos las rutas al archivo server.js, el cual debe actualizarse 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;
19
// 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} !`));
```

# React

Hemos finalizado la creación de nuestro backend. En este apartado vamos a configurar nuestra interfaz con React. Vamos a crear algunos componentes estáticos y usaremos una herramienta Redux para realizar la gestión de estado global y empatar con nuestro backend.

- 1 Crearemos nuestro proyecto react-app y lo vincularemos. Si no has instalado react, corre en tu terminal la siguiente línea: `npm i -g create-react-app`. Si ya está instalado, ve al paso 2.
- 2 En el directorio raíz ejecuta la línea `npx create-react-app client`.
- 3 Ahora, en el `package.json`, debajo del objeto `script` del cliente, agrega la línea:

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



4

El package.json del cliente debe lucir de la siguiente manera:

```
1. {
2.   "name": "client",
3.   "version": "0.1.0",
4.   "private": true,
5.   "dependencies": {
6.     "axios": "^0.18.0",
7.     "classnames": "^2.2.6",
8.     "jwt-decode": "^2.2.0",
9.     "react": "^16.6.3",
10.    "react-dom": "^16.6.3",
11.    "react-redux": "^5.1.1",
12.    "react-router-dom": "^4.3.1",
13.    "react-scripts": "2.1.1",
14.    "redux": "^4.0.1",
15.    "redux-thunk": "^2.3.0"
16.  },
17.  "scripts": {
18.    "start": "react-scripts start",
19.    "build": "react-scripts build",
20.    "test": "react-scripts test",
```



```
1.  "eject": "react-scripts eject"
2.  },
3.  "proxy": "http://localhost:5000",
4.  "eslintConfig": {
5.    "extends": "react-app"
6.  },
7.  "browserslist": [
8.    ">0.2%",
9.    "not dead",
10.   "not ie <= 11",
11.   "not op_mini all"
12. ]
13. }
```

**5** Ahora limpiaremos la aplicación React eliminando algunos archivos innecesarios:

- a.** Elimina logo.svg en client / src.
- b.** Elimina la importación de logo.svg en App.js.
- c.** Elimina todo el CSS en App.css (mantendremos la importación en App.js en caso de que se desee agregar el propio CSS global aquí).
- d.** Borra el contenido en el div principal en App.js y reemplázalo con un `<h1>` por ahora.



Debería ahora estar sin errores y visualizarse de la siguiente manera:

```
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 Instala el estilo que deseas usar (en este caso usaremos materialize.css y editamos el index.html que se encuentra en client/public):

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

El index debe visualizarse 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-  
tofit=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">
```

24

```
<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>
```

- 





Allí mismo crearemos el archivo Landing.js y ubicaremos el siguiente código:

```
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>
          </div>
        </div>
      </div>
    );
  }
}
```



```
</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",
26
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;
```

8

Finalmente, importamos nuestros componentes Navbar y Landing en nuestro archivo App.js y los agregamos 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;
```

# Componente de autenticación

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

**1.** En el archivo Register.js ubicamos el siguiente código:

```
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>
28
</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
    accent3"
  >
    Sign up
  </button>
29
</div>
</form>
</div>
</div>
</div>
);
}
}
export default Register;
```

2. Debe visualizarse el componente de inicio de sesión similar al de registro. Ahora, en el archivo Loguin.js, ubicamos el siguiente código:

```
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}>
  30
  <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}
```



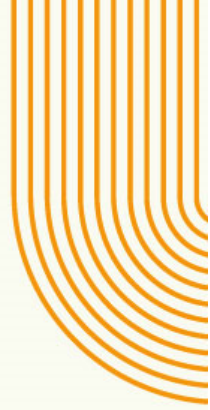
```

        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 accent3"
      >
        Login
      </button>
    </div>
  </form>
</div>
</div>
</div>
);
}
}
export default Login;

```

3. Configuraremos ahora en nuestra app el React Router, que permitirá definir las rutas de enrutamiento usando react-router-dom. Para ello pondremos lo siguiente en el archivo Apps.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 />
```



```
<Navbar />

    <Route exact path="/" component={Landing} />
    <Route exact path="/register" component={Register} />
    <Route exact path="/login" component={Login} />

</div>

    </Router>

);

}

}

export default App;
```



4. Configuramos redux para gestionar el estado, para lo cual editamos el archivo app.js de tal manera que luzca así:

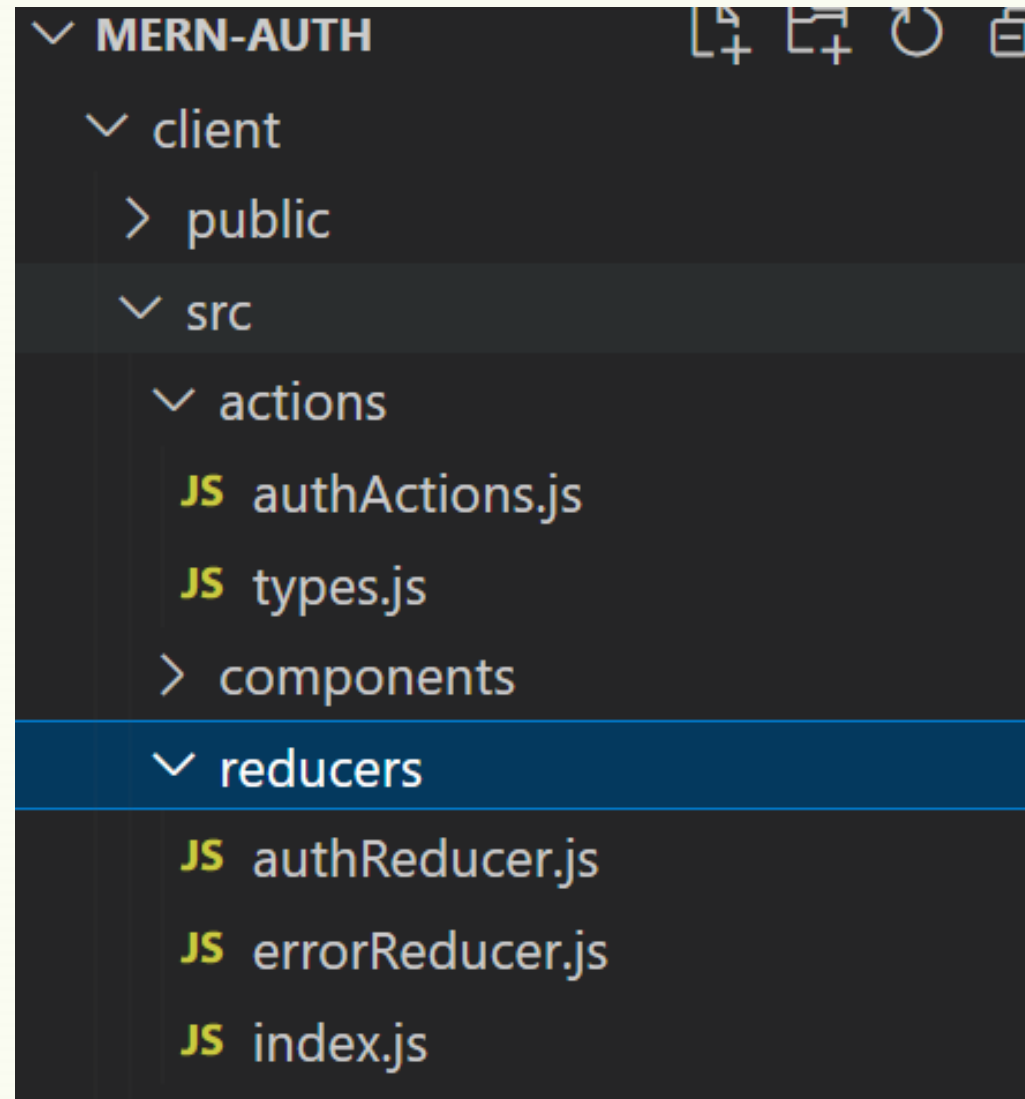
```
import React, { Component } from "react";
32
import { BrowserRouter as Router, Route } from "react-router-dom";
import { Provider } from "react-redux";
import store from "./store";
import Navbar from "./components/layout/Navbar";
import Landing from "./components/layout/Landing";
import Register 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;
```

## 5.

Para configurar la estructura de archivos Redux en el src deben crearse los archivos y carpetas que se visualizan en la siguiente imagen:



6. Crea en la carpeta src el archivo store.js y ubica el siguiente código:

```
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;
```

7. 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" ponemos 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";
```

8. Creamos ahora el archivo `authReducer.js`, en el que configuraremos el cambio de estado de la aplicación. Ubica en él el siguiente código:

```
import {
  SET_CURRENT_USER,
  USER_LOADING
} from "../actions/types";
const isEmpty = require("is-empty");
34
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;  
  }  
}
```



9. En el archivo errorReducer.js ubica el siguiente código:

```
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;
  }
}
```



- 10.** El rootReducer se configura en el index.js. Agrega el siguiente código al index:

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

# Configuración de auth token

Para configurar los eventos es necesario crear un directorio en src denominado utils y, en él, el archivo setAuthToken.js:

```
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;
```



El flujo a tener en cuenta en actions será importar dependencias y definiciones de acciones types.js. Usaremos axios para hacer HTTPRequests dentro de cada acción y finalmente utilizaremos el envío de acciones a los reductores que creamos en el paso anterior.

- 1 Creamos el archivo authActions.js en la carpeta actions y ubicamos el siguiente código:

```
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
36
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({}));  
};
```

2

Ubica el siguiente código en el archivo store.js de la 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;
```

- 3 Es necesario modificar nuestro archivo Register.js para vincular el componente Redux. Ubica lo siguiente en la parte inferior:

```
export default connect(  
  mapStateToProps,  
  { registerUser }  
) (withRouter(Register));  
const mapStateToProps = state => ({  
  auth: state.auth,  
  errors: state.errors  
});
```



4

Ya que no es posible definir tipos en el constructor, se considera buena convención hacerlo usando el paquete prop-types. Para ello, en el Register.js en la parte final agrega el siguiente código:

```
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));
```

- 5 Creación del componente tablero para el inicio de sesión. Para ello, en la carpeta components ubicada en el src, creamos el archivo Dashboard.js y en él ubicamos el siguiente código:

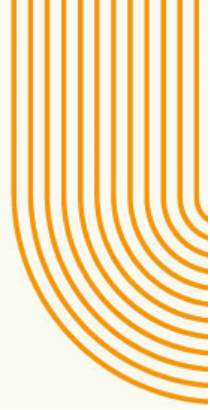
```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { connect } from "react-redux";
import { logoutUser } from
"../../actions/authActions";
class Dashboard extends Component {
  onLogoutClick = (e) => {
    e.preventDefault();
    this.props.logoutUser();
  };
  render() {
    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"
        >

```



```
45
    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);
```

6

Para generar rutas protegidas 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);
```

Si deseas ampliar la información acerca del uso de rutas protegidas, puedes hacerlo consultando el siguiente enlace:

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

7

Para unir los últimos archivos creados es preciso modificar el archivo Apps.js de la siguiente manera:

```
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) {
  47
  // 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;
```



# Node

Desde el inicio hemos estado trabajando Node; sin embargo, es importante realizar una configuración adicional:

- 1 Edita el package.json con el siguiente código. Nodemon nos permitirá correr el programa más fácilmente; no obstante, es opcional usarlo:

```
"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\"",  
},
```

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

- 2 Para lograr esto, agrega el siguiente método de ciclo de vida debajo del constructor en Register.js en la ruta "mern-auth/client/src/components/auth/Register.js/" y también en la ruta "mern-auth/client/src/components/auth/Login.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");  
  }  
}
```

Hemos terminado. Ya puedes desplegar el proyecto con la siguiente línea en la terminal npm run dev.