

EXERCISES QUE TRABAJAREMOS EN LA CUE

- EXERCISE 1: SECURIZACIÓN MEDIANTE JWT, IMPLEMENTACIÓN EN EXPRESS

EXERCISE 1: SECURIZACIÓN MEDIANTE JWT, IMPLEMENTACIÓN EN EXPRESS

El objetivo del presente ejercicio es una guía paso a paso que realiza el proceso de autenticación y login creando desde el inicio un servicio API REST de autenticación de usuario utilizando Express, MongoDB y JWT.

Pasos principales que se realizaran para la visualizar los componentes de un JWT (To-Do):

1. Crear el directorio e inicializar el proyecto con npm
2. Creando los archivos y directorios
3. Instalación de las dependencias
4. Crear el servidor Node.js y conectarse a la base de datos
5. Crear el modelo de usuario y la ruta
6. implementar la funcionalidad del registro y login
7. Crear un middleware para la autenticación con jwt
8. Configurando los Cors

1. CREAR EL DIRECTORIO E INICIALIZAR EL PROYECTO CON NPM

Para comenzar, necesitaremos configurar el proyecto. Abra Visual Studio Code navegando a un directorio de su elección en su máquina o abriéndolo en la terminal.

```
1 mkdir api_jwt_token
2
3 cd jwt-project
4
5 npm init -y
6
7
8 {
```

```
9  "name": "api_jwt_token",
10 "version": "1.0.0",
11 "description": "",
12 "main": "index.js",
13 "scripts": {
14   "test": "echo \"Error: no test specified\" && exit 1"
15 },
16 "keywords": [],
17 "author": "",
18 "license": "ISC"
19 }
```

2. CREANDO LOS ARCHIVOS Y DIRECTORIOS

En el paso 1, se inicializa npm con el comando `npm init -y`, que crea automáticamente un `package.json`.

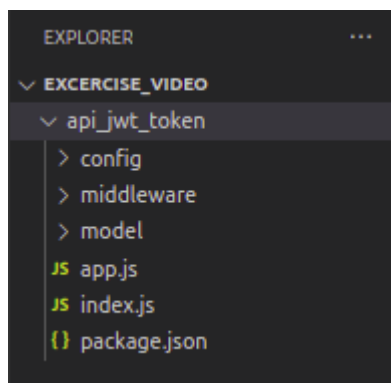
Necesitamos crear el **modelo**, el **middleware**, el directorio de **config** y sus archivos, por ejemplo, `user.js`, `auth.js`, `database.js` usando los comandos a continuación.

```
1 mkdir model middleware config
2
3 touch config/database.js middleware/auth.js model/user.js
```

Procedemos a crear el archivo `index.js` y `app.js` en el directorio raíz del proyecto:

```
1 touch app.js index.js
```

Con la estructura del proyecto de la siguiente manera:



3. INSTALACIÓN DE LAS DEPENDENCIAS

Procedemos a instalar varias dependencias como mongoose, jsonwebtoken, express dotenv bcryptjs y dependencias de desarrollo como nodemon para reiniciar el servidor a medida que hacemos cambios automáticamente.

Para efectos prácticos de este tutorial instalaremos mongoose porque se utilizará MongoDB. Se realizarán validaciones de las credenciales de usuario contra lo que tenemos en nuestra base de datos.

```
1 $ npm install mongoose express jsonwebtoken dotenv bcryptjs
2
3 $ npm install nodemon -D
```

Creamos el archivo app.js:

```
1 // Requerimiento para cargar las variables de entorno .env
2 require("dotenv").config();
3
4 const express = require("express");
5
6 const app = express();
7
8 app.use(express.json());
9
10 // Logic goes here
11
12 module.exports = app;
```

El archivo index.js

```
1 const http = require("http");
2 const app = require("./app");
3 const server = http.createServer(app);
4
5 const {
6   API_PORT
7 } = process.env;
8 const port = process.env.PORT || API_PORT;
9
10 // server listening
11 server.listen(port, () => {
12   console.log(`Servidor corriendo en ${port}`);
```

```
13 } ) ;
```

observamos que el archivo necesita algunas variables de entorno. Creamos el archivo `.env` y agregar sus variables antes de iniciar nuestra aplicación.

```
1 API_PORT=5001
```

Para iniciar el servidor, edite el objeto scripts en nuestro package.json para que se vea como el que se muestra a continuación.

```
1 "scripts": {  
2   "start": "node index.js",  
3   "dev": "nodemon index.js",  
4   "test": "echo \"Error: no test specified\" && exit 1"  
5 }
```

y se ejecuta el comando:

```
1 npm run dev  
2  
3  
4 > api_jwt_token@1.0.0 dev  
5 > nodemon index.js  
6  
7 [nodemon] 2.0.15  
8 [nodemon] to restart at any time, enter `rs`  
9 [nodemon] watching path(s): *.*  
10 [nodemon] watching extensions: js,mjs,json  
11 [nodemon] starting `node index.js`  
12 Servidor corriendo en 5001
```

4. CREAR EL SERVIDOR NODE.JS Y CONECTARSE A LA BASE DE DATOS

Nos conectemos a la base de datos agregando los siguientes fragmentos a su `app.js`, `database.js` `.env` en ese orden.

```
1 config/database.js:  
2 const mongoose = require("mongoose");
```

```
3
4 const {
5     MONGO_URI
6 } = process.env;
7
8 exports.connect = () => {
9     // Connecting to the database
10    mongoose
11        .connect(MONGO_URI, {
12            useNewUrlParser: true,
13            useUnifiedTopology: true,
14            //useCreateIndex: true,
15            //useFindAndModify: false,
16        })
17        .then(() => {
18            console.log("Correctamente Conectado a la Bases de Datos");
19        })
20        .catch((error) => {
21            console.log("Fallo de conexión de a la base de datos");
22            console.error(error);
23            process.exit(1);
24        });
25};
```

En app.js se agrega:

```
1 // Conectando a la base de datos
2 require("../config/database").connect();
```

y en el .env

```
1 MONGO_URI=mongodb://localhost:27017/api-rest-jwt
```

5. CREAR EL MODELO DE USUARIO Y EL RUTA (ENDPOINT)

Se procede a desarrollar el esquema para los detalles del usuario cuando se registre por primera vez y los validamos con las credenciales guardadas al iniciar sesión.

Se adecua el archivo user.js la carpeta model

```
1 const mongoose = require("mongoose");
2
3 const userSchema = new mongoose.Schema({
4   first_name: {
5     type: String,
6     default: null
7   },
8   last_name: {
9     type: String,
10    default: null
11  },
12  email: {
13    type: String,
14    unique: true
15  },
16  password: {
17    type: String
18  },
19 });
20
21 module.exports = mongoose.model("user", userSchema);
```

Se procede a crear las rutas para registrarse e iniciar sesión, respectivamente.

En app.js en el directorio raíz, agregue el siguiente código para el registro y el inicio de sesión.

```
1 // Importando la logista del modelo
2 const User = require("../model/user");
3
4 // Registro
5 app.post("/registro", (req, res) => {
6   // lógica del registro
7 });
8
9 // Login
10 app.post("/login", (req, res) => {
11   // lógica del inicio de sesión
12 });
```

6. IMPLEMENTAR LA FUNCIONALIDAD DEL REGISTRO Y E INICIO DE SESIÓN (LOGIN)

Seguidamente se crearán dos rutas en la aplicación. Se usará JWT para firmar las credenciales y *bcrypt* para cifrar la contraseña antes de almacenarlas en nuestra base de datos.

En la ruta **/registro**, se realizará lo siguiente:

- Obtener los datos del usuario.
- Validar los datos del usuario.
- Validar si el usuario ya existe.
- Cifrar la contraseña de usuario.
- Crea un usuario en nuestra base de datos.
- Y finalmente, cree un token JWT firmado.

Modifique la estructura de ruta **/registro** que se creo anteriormente con el siguiente código en el archivo **app.js**

```
1 // Registro
2 app.post("/registro", async (req, res) => {
3   // logica del registro
4   try {
5     // obteniendo los valores de entrada
6     const {
7       first_name,
8       last_name,
9       email,
10      password
11    } = req.body;
12
13    // Validar los datos de entrada
14    if (!(email && password && first_name && last_name)) {
15      res.status(400).send("Todos los campos son requeridos");
16    }
17
18    // Chequeando si el usuario existe
19    // Validar si el usuario existe en la bases de datos
20    const oldUser = await User.findOne({
21      email
22    });
```

```
23     if (oldUser) {
24         return res.status(409).send("Actualmente el usuario existe,
25 inicie login en http://localhost:5001/login");
26     }
27
28     // Encriptando la contraseña del usuario
29     encryptedPassword = await bcrypt.hash(password, 10);
30
31     // Password encriptado
32     console.log("\nPassword encriptado: " + encryptedPassword);
33
34     // Creando el usuario en la bases de datos
35     const user = await User.create({
36         first_name,
37         last_name,
38         email: email.toLowerCase(), // Convertimos a minuscula
39         password: encryptedPassword,
40     });
41
42     // Creación del Token
43     const token = jwt.sign({
44         user_id: user._id,
45         email
46     },
47     process.env.TOKEN_KEY, {
48         expiresIn: "1h",
49     }
50 );
51
52     // Token Generado
53     console.log("\nToken Generdo: " + token);
54
55     // retornamos el nuevo usuario
56     return res.status(201).json(user);
57 } catch (err) {
58     console.log(err);
59 }
60
61 });
```

Usando Postman para probar el punto final, obtendremos la respuesta de errores por falta de paquetes que no se encuentran definidos en el scripts, reflejando lo siguiente:

```
[nodemon] starting `node index.js`
Servidor corriendo en 5001
Correctamente Conectado a la Bases de Datos
ReferenceError: bcrypt is not defined
```


Agregamos las siguientes librerías en la parte inicial del **app.js** luego de la conexión a la base de datos:

```
1 // Librerías de encriptacion de jsonwebtoken
2 const bcrypt = require("bcryptjs");
3 const jwt = require("jsonwebtoken")
```

Ejecutamos nuevamente Postman y no refleja que no existe respuesta del servidor por falta de emitir y generar la clave y firma del token, revisamos la terminal y observamos:

```
[nodemon] starting node index.js
Servidor corriendo en 5001
Correctamente Conectado a la Bases de Datos
Error: secretOrPrivateKey must have a value
```

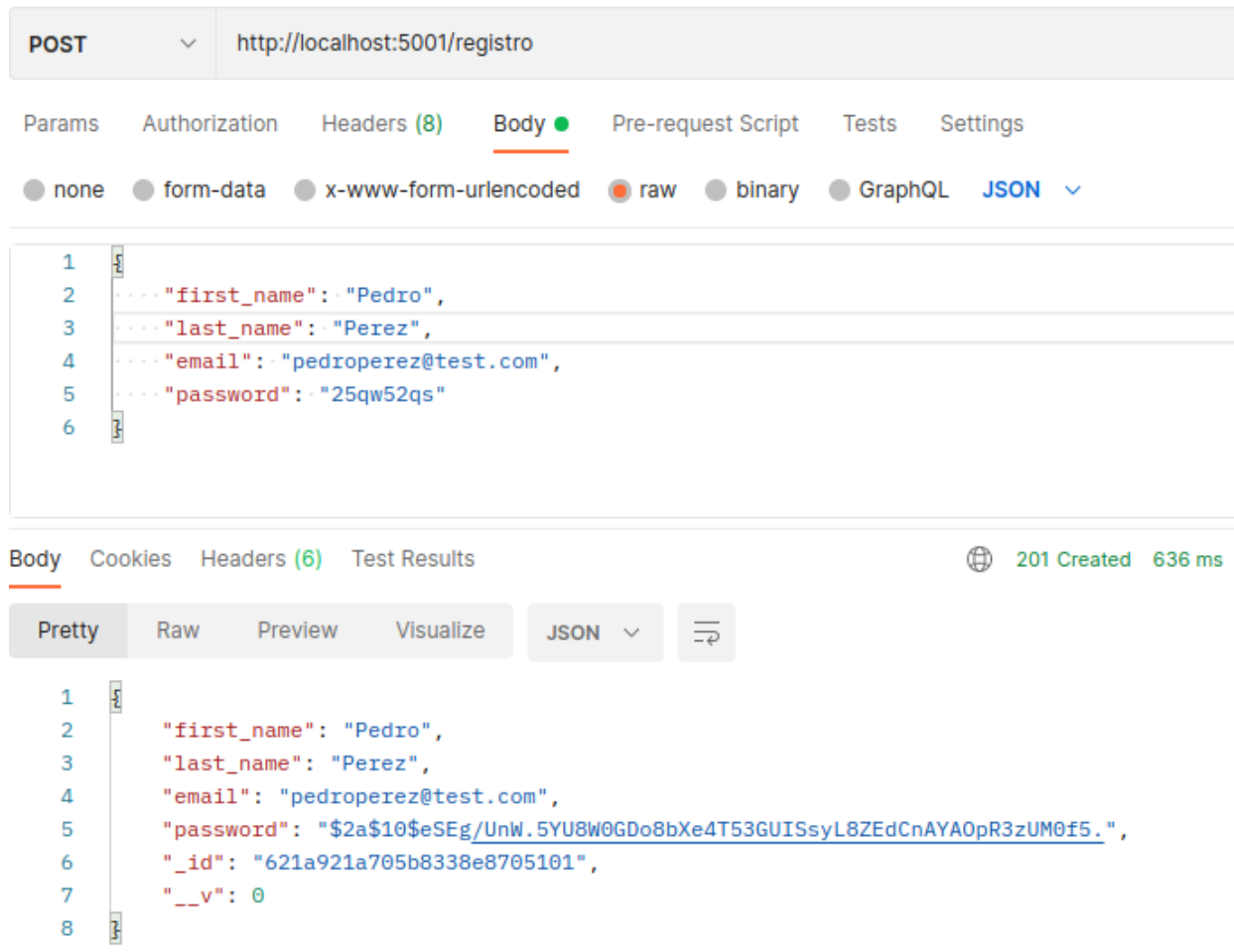
El error proviene de **TOKEN_KEY**. Es importante mencionar que el token necesita de una clave secreta, la que colocaremos dentro de nuestras variables de entorno con nombre de **TOKEN_KEY**. Esa es la magia de un token, solo si el token trae la misma clave secreta podrá ser validada por el backend. Para generar el mismo aleatoriamente por la terminal, o podemos colocar cualquier clave o frase secreta para el mismo, en este caso lo generamos:

```
1 api_jwt_token$ node
2 Welcome to Node.js v12.16.2.
3 Type ".help" for more information.
4 > require("crypto").randomBytes(64).toString("hex")
5 'd6699170151a914e2b5a67c3b913401e971f28e66427286d7e3af04194d764fc70b1466686
6 8121
7 c515188a9aff1fc3550d7a25d361dbd779863ee6cb9a2d7abc'
8 >
```

Nuestro token generado lo agregamos al archivo **.env**:

```
1 TOKEN_KEY=d6699170151a914e2b5a67c3b913401e971f28e66427286d7e3af04194d764fc7
2 0b
3 14666868121c515188a9aff1fc3550d7a25d361dbd779863ee6cb9a2d7abc
```

Ejecutando nuevamente el Postman obtenemos la transacción exitosa



POST ▼ http://localhost:5001/registro

Params Authorization Headers (8) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   ... "first_name": "Pedro",
3   ... "last_name": "Perez",
4   ... "email": "pedroperez@test.com",
5   ... "password": "25qw52qs"
6 }
```

Body Cookies Headers (6) Test Results 🌐 201 Created 636 ms

Pretty Raw Preview Visualize **JSON** ▼ ≡

```
1 {
2   "first_name": "Pedro",
3   "last_name": "Perez",
4   "email": "pedroperez@test.com",
5   "password": "$2a$10$eSEg/UnW.5YU8W0GDo8bXe4T53GUISSyL8ZEdCnAYA0pR3zUM0f5.",
6   "_id": "621a921a705b8338e8705101",
7   "__v": 0
8 }
```

Es importante mencionar que jamás se almacena la contraseña en formato de texto plano dentro de la base de datos. La contraseña debe ser encriptada, de tal forma que muy pocas personas puedan acceder a ella y ver la contraseña de los usuarios. Para hacer esto utilizaremos un algoritmo de Hash mediante la dependencia que instalamos anteriormente llamada `bcrypt` dentro del registro de usuario.

Usar sólo un algoritmo de hash no es suficiente, ya que un input igual nos devolverá un output igual, así que lo que necesitamos es realizar el hash utilizando un número aleatorio, de esta manera si un atacante tiene acceso a la base de datos de las contraseñas, no podrá descifrar las contraseñas. Por otro lado, el número aleatorio va incluido en el hash por lo que no solo bastará el hash para descifrar la contraseña. Este número aleatorio es entregado por la dependencia `bcrypt` y es conocido como `salt`.

En nuestro caso el salt le colocamos en numero 10

```
1 // Encriptando la contraseña del usuario
2 encryptedPassword = await bcrypt.hash(password, 10);
```

Lo podemos generar y realizarlo mas seguro de una manera aleatoria, para ello generamos el salt con el metodo `genSalt()`, esto es:

```
1 //Generamos aleatoriamente el salt
2 const salt = await bcrypt.genSalt(10)
3 console.log("Salt generado: " + salt);
4
5 // Encriptando la contraseña del usuario
6 encryptedPassword = await bcrypt.hash(password, salt);
7 console.log("\nPassword encriptado: " + encryptedPassword);
```

Al registrar un nuevo usuario tenemos:

```
1 [nodemon] starting `node index.js`
2 Servidor corriendo en 5001
3 Correctamente Conectado a la Bases de Datos
4 [nodemon] restarting due to changes...
5 [nodemon] starting `node index.js`
6 Servidor corriendo en 5001
7 Correctamente Conectado a la Bases de Datos
```

SALT GENERADO:

```
1 $2a$10$BU6DGD3WfZqRJPX4xN6uDO
```

PASSWORD ENCRYPTADO:

```
1 $2a$10$BU6DGD3WfZqRJPX4xN6uDO1bsnsYGmhQVWsTjFqEBHi5BHXtJxvQC
```

TOKEN GENERADO:

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiaWJxYjhmMTFlYzU2ODMzM
2 TJhMGM1ODBiIiwiaWZlhaWwiOiJwZWRYc3NvcGVyZXpAdGVzdC5jb20iLCJpYXQiOiJlNDU5NzM
3 yNjUsImV4cCI6MTY0NTk3Njg2NX0.zw8QRi266SWU1txx5ybqplag_qVURvoRsBiFe0tHAVA
```

Generando la lógica para el inicio de sesión o endpoint **/login**

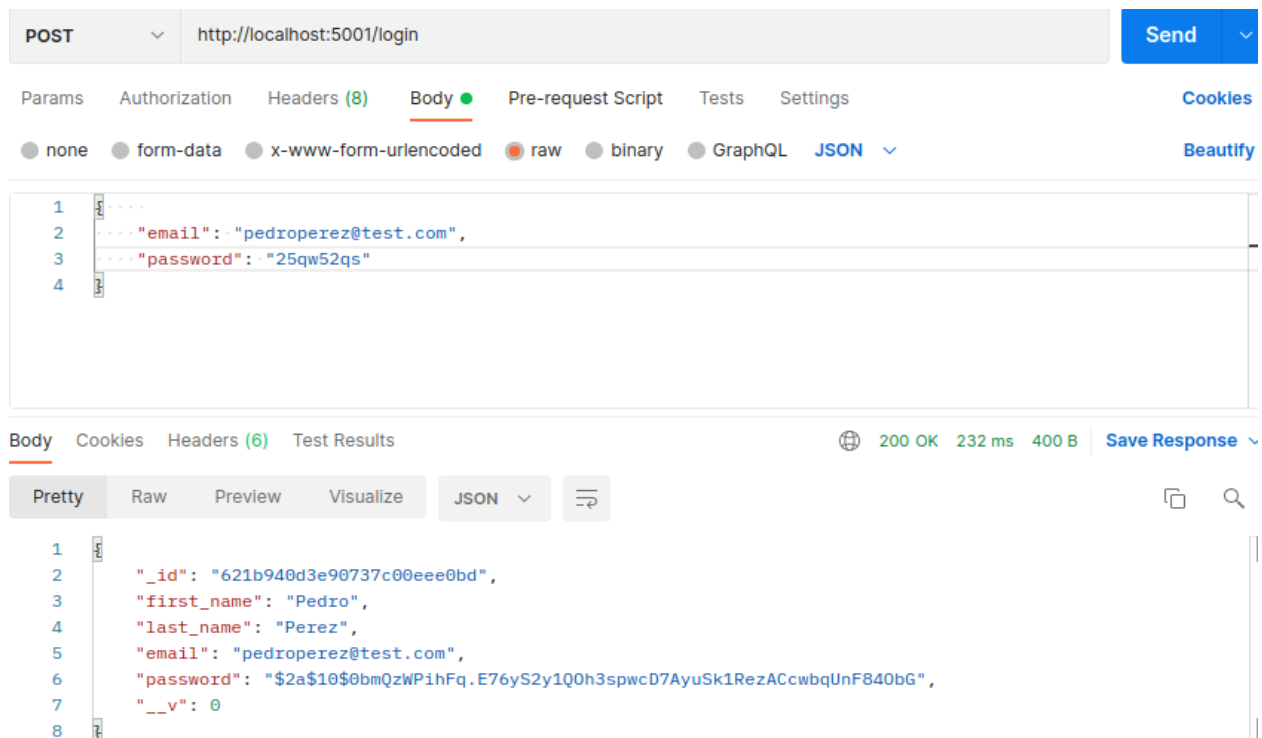
- Lectura de los datos de entrada
- Validar los datos de entrada
- Validar la existencia del usuario en la DB
- Verificación de la contraseña del usuario con la contraseña en la DB
- Se crea un Token JWT firmado

/login

```
1 app.post("/login", async (req, res) => {
2   // logica del inicio de sesión
3   try {
4     // obteniendo los datos de entrada
5     const {
6       email,
7       password
8     } = req.body;
9
10    // Validar los datos de entrada
11    if (!(email && password)) {
12      res.status(400).send("Todos los datos son requeridos, email y
13password");
14    }
15
16    // Validando la existencia del usuario en la base de datos
17    const user = await User.findOne({
18      email
19    });
20
21    if (user && (await bcrypt.compare(password, user.password))) {
22      // Se genera el Token
23      const token = jwt.sign({
24        user_id: user._id,
25        email
26      },
27        process.env.TOKEN_KEY, {
28          expiresIn: "1h",
29        }
30      );
31
32      // Impresion por el terminal del Token generado para el
33usuario
34      console.log("Usuario: " + email + "\nToken: " + token);
```

```
35
36     // Retornando los datos del usuario
37     return res.status(200).json(user);
38   }
39   return res.status(400).send("Credenciales invalidas");
40 } catch (err) {
41   console.log(err);
42 }
43 });
```

Realizamos la verificación con Postman



POST http://localhost:5001/login

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "email": "pedroperez@test.com",
3   "password": "25qw52qs"
4 }
```

Body Cookies Headers (6) Test Results 200 OK 232 ms 400 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "621b940d3e90737c00eee0bd",
3   "first_name": "Pedro",
4   "last_name": "Perez",
5   "email": "pedroperez@test.com",
6   "password": "$2a$10$0bmQzWPihFq.E76yS2y1Q0h3spwcD7AyuSk1RezACcwbqUnF840bG",
7   "__v": 0
8 }
```

7. CREAR UN MIDDLEWARE PARA LA AUTENTICACIÓN CON JWT

En el backend hay endpoints protegidas, es decir, rutas a las que solo se puede acceder con el token JWT que obtenemos del inicio de sesión /login anteriormente. En este sentido éstas rutas primero se debe verificar el token del request de la petición. Para esto crearemos un middleware que nos permitirá hacer

esta verificación cuando accedemos a una determinada ruta, por ejemplo /inicio. Esto lo haremos dentro de un archivo nuevo llamado auth.js dentro de /middleware.

Actualizamos el auth.js.

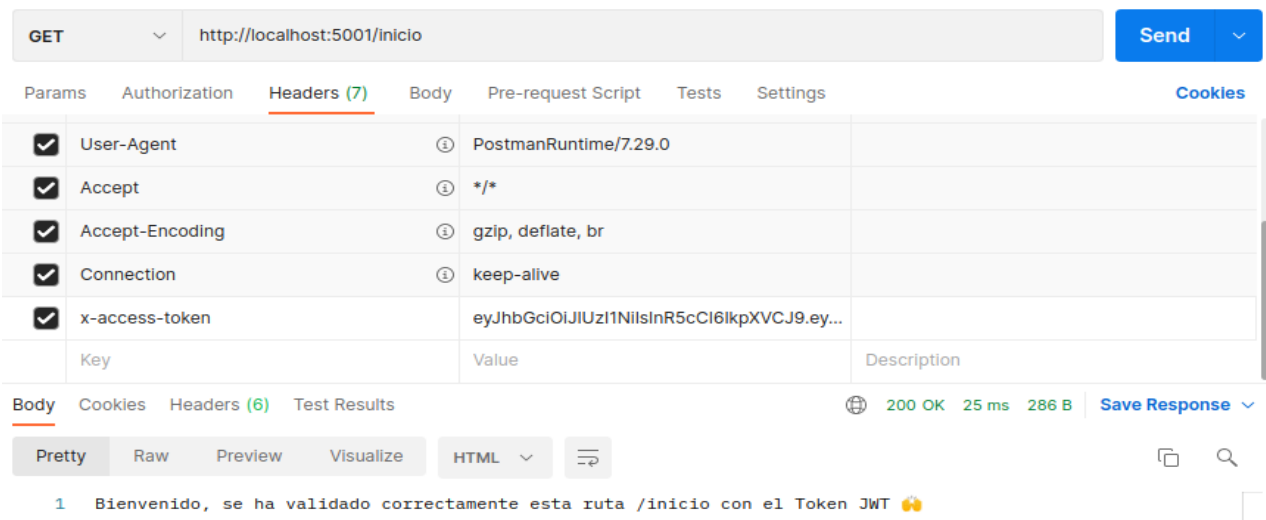
middleware/auth.js

```
1 const jwt = require("jsonwebtoken");
2 const config = process.env;
3
4 const verifyToken = (req, res, next) => {
5     // Obtenemos el token del header del request, del req.body o del
6     req.query
7     const token =
8         req.body.token || req.query.token || req.headers["x-access-
9     token"];
10
11     // Validamos si existe token en el req
12     if (!token) {
13         return res.status(403).send("Un token es requerido para la
14     autorización");
15     }
16     try {
17         // Verificamos el token usando la dependencia de jwt y el método
18     .verify
19         const decoded = jwt.verify(token, config.TOKEN_KEY);
20         // si el token es correcto nos devolvera los datos que colocamos
21     en el token
22         console.log(decoded);
23         req.user = decoded;
24         // next() indica que el req paso la prueba y continúe su camino
25         next()
26     } catch (err) {
27
28         return res.status(401).send("Token no valido, acceso denegado");
29     }
30     return next();
31 };
32
33 module.exports = verifyToken;
```

Procedemos a crear la ruta /inicio y actualizamos el app.js con el siguiente código para posteriormente realizar las pruebas del middleware.

```
1 // middleware que valida el Token JWT
2 const auth = require("../middleware/auth");
3
4 app.get("/inicio", auth, (req, res) => {
5
6     res.status(200).send("Bienvenido, se ha validado correctamente esta
7 ruta /inicio con el Token JWT 🤖 ");
8 });
```

Realizamos la pruebas con Postman pasando como token el generado al realizar login el usuario como headers.



The screenshot shows a Postman interface for a GET request to `http://localhost:5001/inicio`. The 'Headers' tab is active, displaying the following headers:

Key	Value	Description
User-Agent	PostmanRuntime/7.29.0	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...	

The 'Body' tab shows the response: `1 Bienvenido, se ha validado correctamente esta ruta /inicio con el Token JWT 🤖`. The status is `200 OK`, with a response time of `25 ms` and a size of `286 B`.

8. CONFIGURANDO LOS CORS

Los cors son necesarios ya que evitará que el navegador cierre la conexión entre cliente y servidor si es que estos dos se encuentran en diferentes lugares con IPs diferentes. Esto lo debemos configurar en `app.js` de la siguiente manera.

Instalamos la dependencia:

```
1 $ npm i cors
```

```
1 // Creamos la variable de configuración
```

```
2 var corsOpt = {  
3   origin: '*', // Se debe reemplazar el * por el dominio de nuestro front  
4   optionsSuccessStatus: 200 // Es necesario para navegadores antiguos o  
5   algunos SmartTVs  
6 }  
7 app.use(cors(corsOpt));
```