UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

CIENCIAS DE LA COMPUTACIÓN

INGENIERÍA DE SOFTWARE

INFORME DE LABORATORIO 1.1

Desarrollo Web Avanzado

NRC 2305

Integrantes:

Rubén Benavides

Joan Cobeña

Juan Pasquel

Damián Verdesoto

17 de Noviembre del 2024

2. Índice

3. Resumen

Este informe describe el desarrollo de una aplicación web utilizando el modelo de arquitectura MVC. El objetivo principal del proyecto es crear una API REST con Node.js para gestionar el registro y la autenticación de usuarios, y un frontend funcional desarrollado en React.js. En el backend, se implementaron tecnologías como Express, MongoDB, bcryptjs y jsonwebtoken para garantizar la seguridad y eficiencia. Postman fue empleado para realizar pruebas exhaustivas de los endpoints. Este proyecto tiene como finalidad demostrar la implementación de principios avanzados de desarrollo web en un entorno práctico.

4. Introducción

4.1 Objetivo del informe

El propósito de este informe es documentar el desarrollo de una aplicación web basada en el modelo MVC (Modelo-Vista-Controlador). Este documento detalla los procesos, tecnologías y metodologías empleadas, además de los resultados obtenidos al implementar una solución web funcional y segura.

4.1.1 Link de github

https://github.com/dr00p3r/Web-Avanzada

4.2 Contexto

En el ámbito del desarrollo web avanzado, la implementación de aplicaciones seguras y bien estructuradas es fundamental. El modelo MVC facilita la separación de responsabilidades, mejorando la mantenibilidad y escalabilidad del código. Este laboratorio se enmarca dentro de los contenidos de la asignatura "Desarrollo Web Avanzado" y busca integrar conceptos teóricos con prácticas concretas, como el uso de Node.js para el backend y React.js para el frontend.

4.3 Alcance y limitaciones

El proyecto se enfoca en la creación de un sistema básico de autenticación y registro de usuarios. Los límites del desarrollo incluyen:

- **Alcance:** Construcción de una API REST funcional, implementación de un frontend básico, y pruebas con Postman.
- **Limitaciones:** No se aborda la implementación de despliegue en servidores de producción profesional ni la integración con servicios externos.

5. Objetivos

5.1 General

Diseñar y desarrollar una aplicación web utilizando el modelo MVC, integrando tecnologías modernas de desarrollo web para crear una API REST funcional y un frontend dinámico.

5.2 Específicos

Implementar una API REST en Node.js para gestionar el registro y la autenticación de usuarios.

- Utilizar MongoDB para almacenar y gestionar datos de usuarios, garantizando la seguridad de contraseñas con bcryptjs.
- Implementar la autenticación de usuarios mediante tokens JWT.
- Diseñar y desarrollar un frontend con React.js que permita la interacción con los servicios de la API.
- Realizar pruebas de los endpoints de la API utilizando Postman para garantizar su funcionalidad.

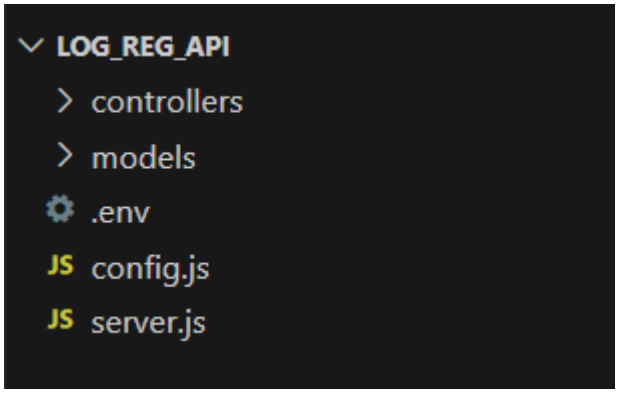
6. Desarrollo o Cuerpo del Informe

El proyecto fue dividido en distintos proyectos.

6.1 API

Creamos un proyecto para la API en Visual Studio Code. Creamos nuestro archivo principal **server.js**, además de nuestras carpetas **models** y **controllers** para aplicar el MVC. La razón de no crear la carpeta Views, es porque decidimos hacerlo desde el otro proyecto, dado que precisamente es la API.

Adicionalmente, creamos nuestro archivo **config.js**, cuyo propósito es manejar la configuración a la base de datos. Creamos un **.env** para nuestras credenciales de base de datos y del token generado.



Ahora, descargamos las librerías que vamos a implementar.

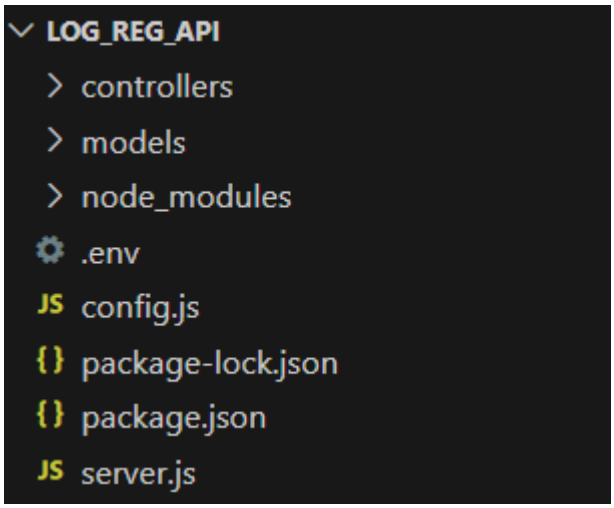
- **dotenv**: Para utilizar variables de entorno.
- **mongoose**: Para la conexión y uso de la base de datos MongoDB.
- **bcryptjs**: Para la encriptación de contraseñas.
- **jsonwebtoken**: Para la creación de tokens.
- **express**: Para el servidor levantado.
- **cookie-parser**: Para el alojamiento de cookies con los tokens.
- **cors:** Para permitir el origen cruzado de distintos recursos.

Para aquello utilizamos el gestor de paquetes **npm** de node.js.

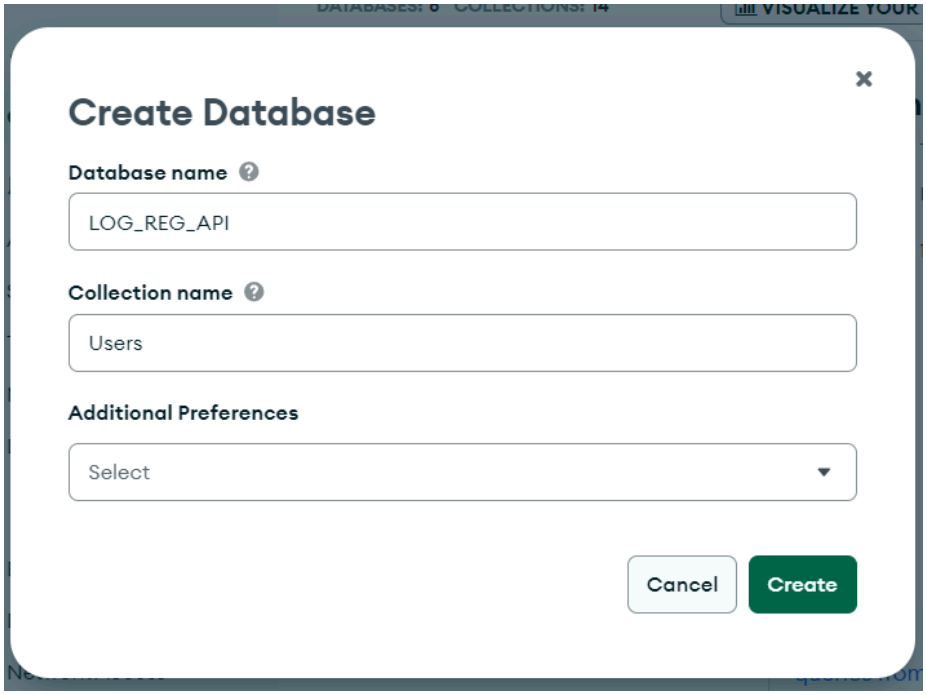
npm i dotenv mongoose bcryptjs jsonwebtoken express cookie-parseer cors

```
PS D:\ESPE\QUINTO\web_avanzado\p1\tareas\log_reg_API> npm i dotenv mongoose bcryp
tjs jsonwebtoken express cookie-parser cors

added 105 packages in 10s

15 packages are looking for funding
  run `npm fund` for details
```

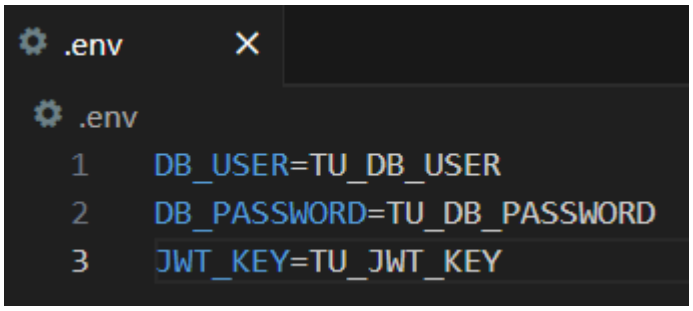
Nuestro proyecto debería quedar así:



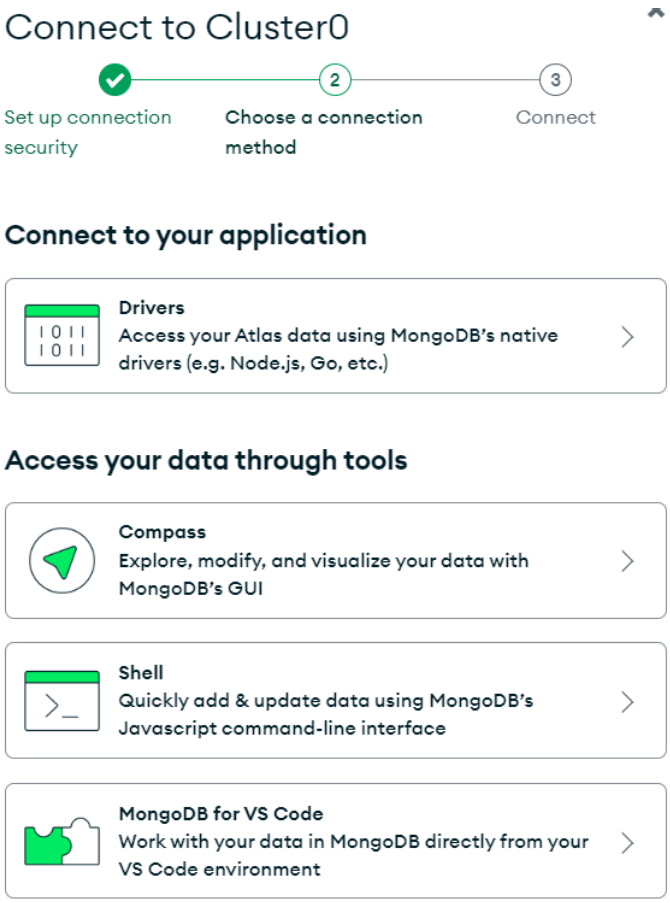
Ahora, nosotros ya hemos creado nuestra cuenta en MongoDB con nuestro cluster, creamos la base de datos que vayamos a utilizar y la colección.



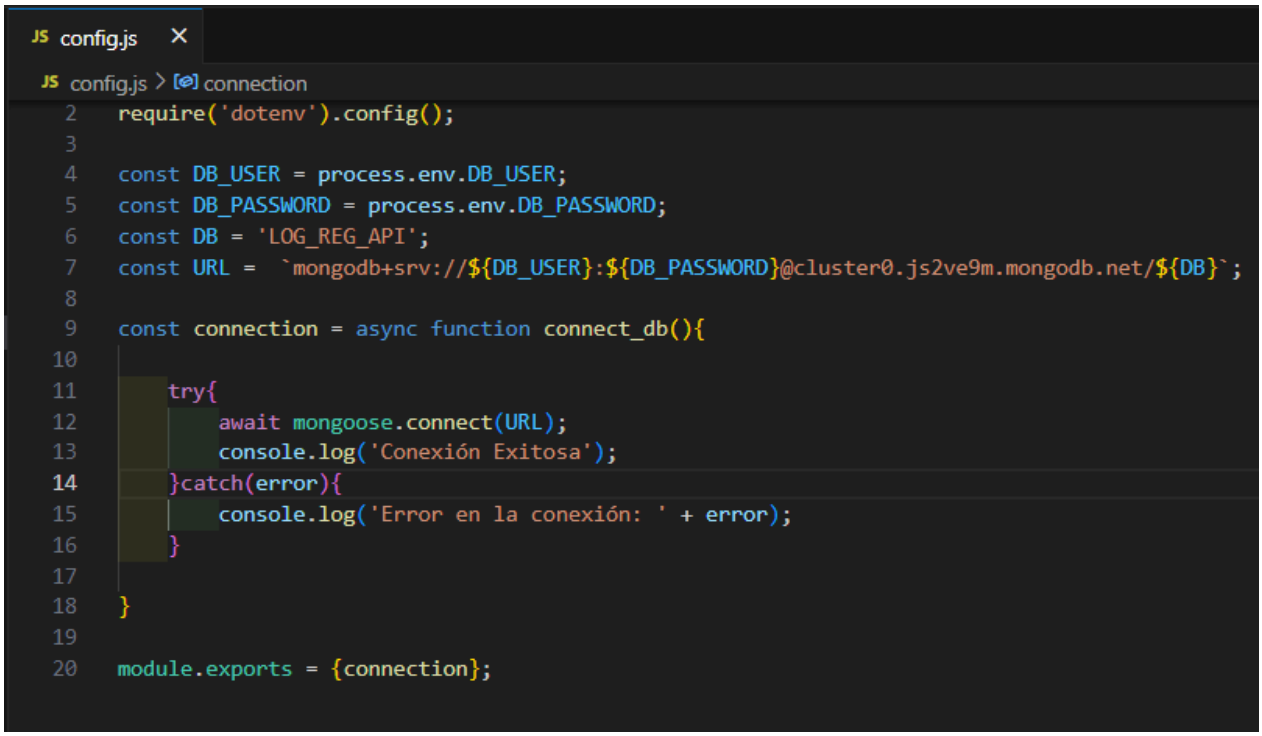
Ahora, en el .env alojamos nuestras credenciales.



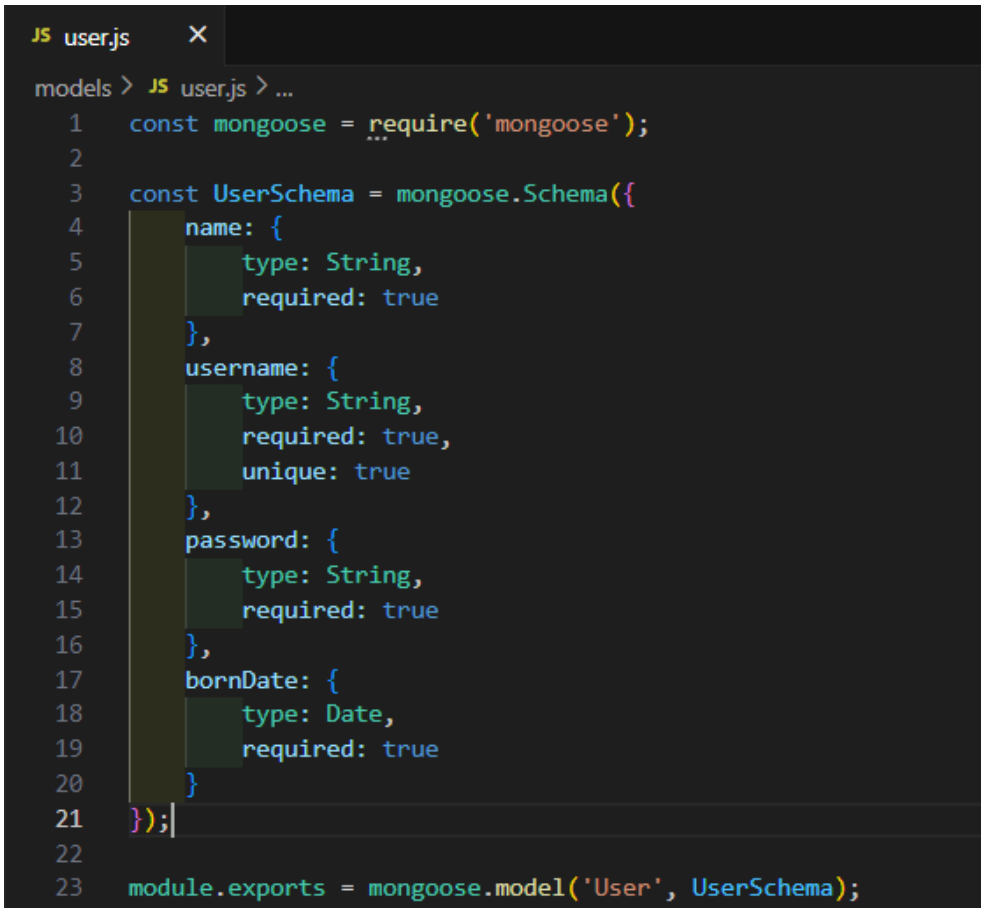
Ahora, procedemos con nuestra configuración de la base de datos.
Extraemos nuestras credenciales.
Preparamos nuestra URL con la base de datos. Puedes hallarla en tu Cluster > Connect.



Exportamos una función asíncrona para realizar la conexión al momento de ejecutar nuestro **server.js**.

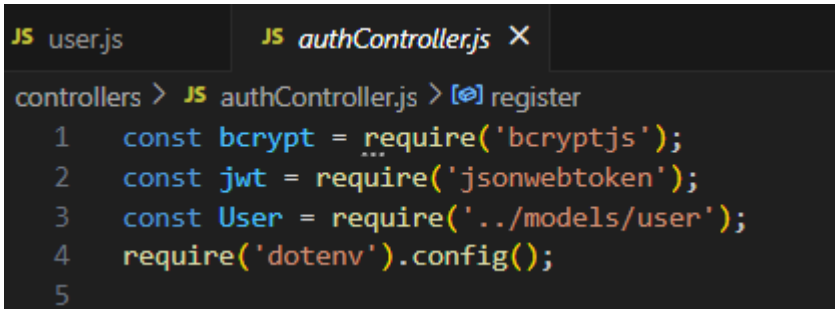


Antes de seguir con la aplicación, vamos a crear nuestro modelo para la base de datos. Por tanto, creamos un nuevo archivo en **models**, llamado **user.js**, y lo exportamos para usarlo en el controller.



Ahora, vamos a crear nuestro controlador, que se encargará de hacer las funciones, en este caso, de login y register. Así que creamos un archivo llamado **authController.js**.

En este archivo, se maneja también la lógica de tokens, encriptación y cookies, por tanto importamos las diferentes librerías instaladas anteriormente.



Se recupera el body enviado y se hashea la contraseña con bcrypt.
Se crea un nuevo user y se envía la respuesta para comprobar del otro lado.
Si no funciona, se validan los diferentes tipos de errores producidos.

```
6   const register = async (req, res) => {
7       const {name, username, password, bornDate} = req.body;
8       try{
9           const hashedPassword = await bcrypt.hash(password, 10);
10          const answer = await User.create({name, username, password: hashedPassword, bornDate
11          res.send(answer);
12      }catch(err){
13          if(err.code === 11000){
14              return res.status(400).json({
15                  error: 'El nombre de usuario ya está en uso'
16              });
17          }
18          return res.status(500).json({
19              error: 'Error del servidor'
20          })
21      }
22  }
```

Para el user, se intenta primeramente encontrar el usuario con la función del modelo y manejamos las respuestas. Luego comprobamos las contraseñas.
Una vez se valida correctamente, creamos nuestro token con los datos respectivos, en este caso el id del usuario, su username, la clave secreta para la firma y la expiración del mismo.
Guardamos el token en una cookie que es enviada al cliente con las configuraciones que se pueden observar.
Finalmente, enviamos que se pudo ingresar exitosamente.

```
24  const login = async(req, res) => {
25      const {username, password} = req.body;
26      try{
27          const user = await User.findOne({username});
28          if (!user) return res.status(400).json({ error: "Usuario no encontrado" });
29
30          const isMatch = await bcrypt.compare(password, user.password);
31          if (!isMatch) return res.status(400).json({ error: "Contraseña incorrecta" });
32
33          const token = jwt.sign({ id: user._id , username: user.username},
34              process.env.JWT_SECRET, { expiresIn: '3h' });
35
36          res.cookie('token', token, {
37              httpOnly: true,
38              secure: true,
39              sameSite: 'None',
40              maxAge: 3 * 60 * 60 * 1000,
41          });
42
43          res.status(200).json({message: "Inicio de sesión exitoso" });
44
45      }catch(err){
46          return res.status(500).json({
47              error: 'Error del servidor.'
48          })
49      }
50  }
```

Creamos otra función para verificar el token donde obtiene la firma de la cookie que se encuentre, así que verificamos que se pueda entregar la cookie.
Utilizamos el verify de jsonwebtoken para confirmar, y sí es posible verificamos el usuario alojando la información en la petición (req) y next() que permite pasar a otra solicitud, misma que se encarga de obtener la información del usuario.

```
const verifyToken = (req, res, next) => {
    const token = req.cookies.token;
    console.log(token);
    if (!token) return res.status(403).json({ error: "Acceso denegado" });
    try {
        const verified = jwt.verify(token, process.env.JWT_SECRET);
        req.user = verified;
        next();
    } catch (err) {
        res.status(401).json({ error: "Token inválido" });
    }
};
```

En la función de obtener la información del usuario verificamos que exista como tal la información, caso contrario, manejamos la respuesta.
Si es posible, enviamos la información pertinente.

```
65  const getUserInfo = (req, res) => {
66      try {
67          if (!req.user) {
68              return res.status(403).json({ error: "Usuario no autenticado" });
69          }
70
71          res.status(200).json({ username: req.user.username });
72      } catch (err) {
73          res.status(500).json({ error: "Error del servidor" });
74      }
75  };
```

Finalmente exportamos estas funciones anónimas.

```
77  module.exports = { register, login, verifyToken, getUserInfo };
```

Ahora, pasamos con el levantamiento del servidor.

En nuestro archivo server.js importamos las librerías de express para levantar el servidor, de cors para permitir el intercambio de recursos de origen cruzado, cookie-parser para la destrucción de cookies, nuestra conexión a la base de datos, y las funciones de authController.

Establecemos el puerto.
Configuramos los cors, el cookieparser y el uso de json.

```
8   const port = 5000;
9
10  const app = express();
11  app.use(cors({
12      origin: 'https://web-avanzada-eight.vercel.app',
13      credentials: true
14  }));
15  app.use(cookieParser());
16  app.use(express.json());
17  app.listen(port, ()=>{
18      console.log('Serv Levantado.');
19  });
20
```

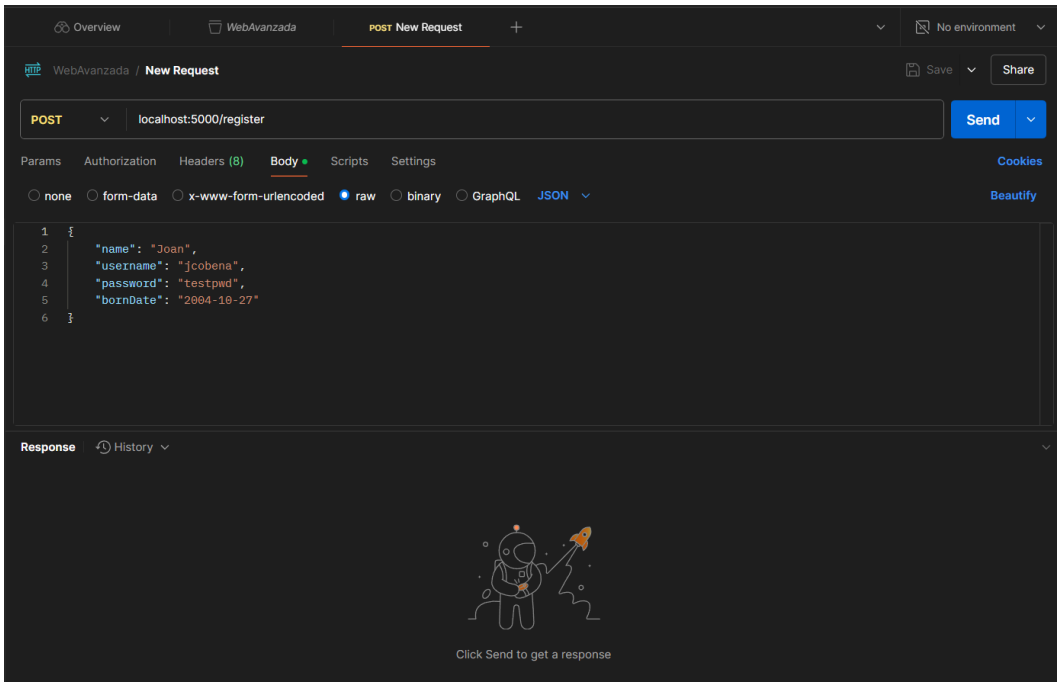
Conectamos a la base.
Establecemos las diferentes rutas.

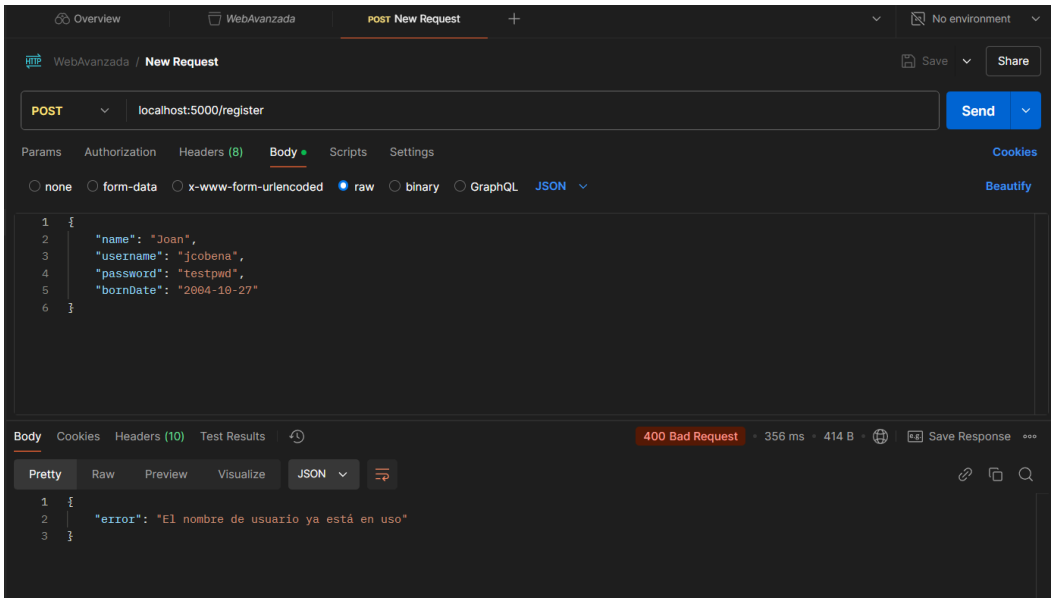
```
21  connection();
22
23  app.get("/", (req, res)=>{
24      res.send('Serv Levantado.');
25  });
26
27  app.post(`/login`, login);
28
29  app.post(`/register`, register);
30
31  app.get('/get-token', verifyToken, (req, res) => {
32      const token = req.cookies.token;
33      console.log(token);
34      if (!token) {
35          return res.status(404).json({ error: "Token no encontrado" });
36      }
37      res.status(200).json({ token });
38  });
39
40  app.post('/logout', (req, res) => {
41      res.clearCookie('token', {
42          httpOnly: true,
43          secure: true,
44          sameSite: 'None',
45      });
46      res.status(200).json({ message: 'Sesión cerrada exitosamente' });
47  });
48
49  app.get('/user-info', verifyToken, getUserInfo);
```

Verificamos las conexiones en server.js.

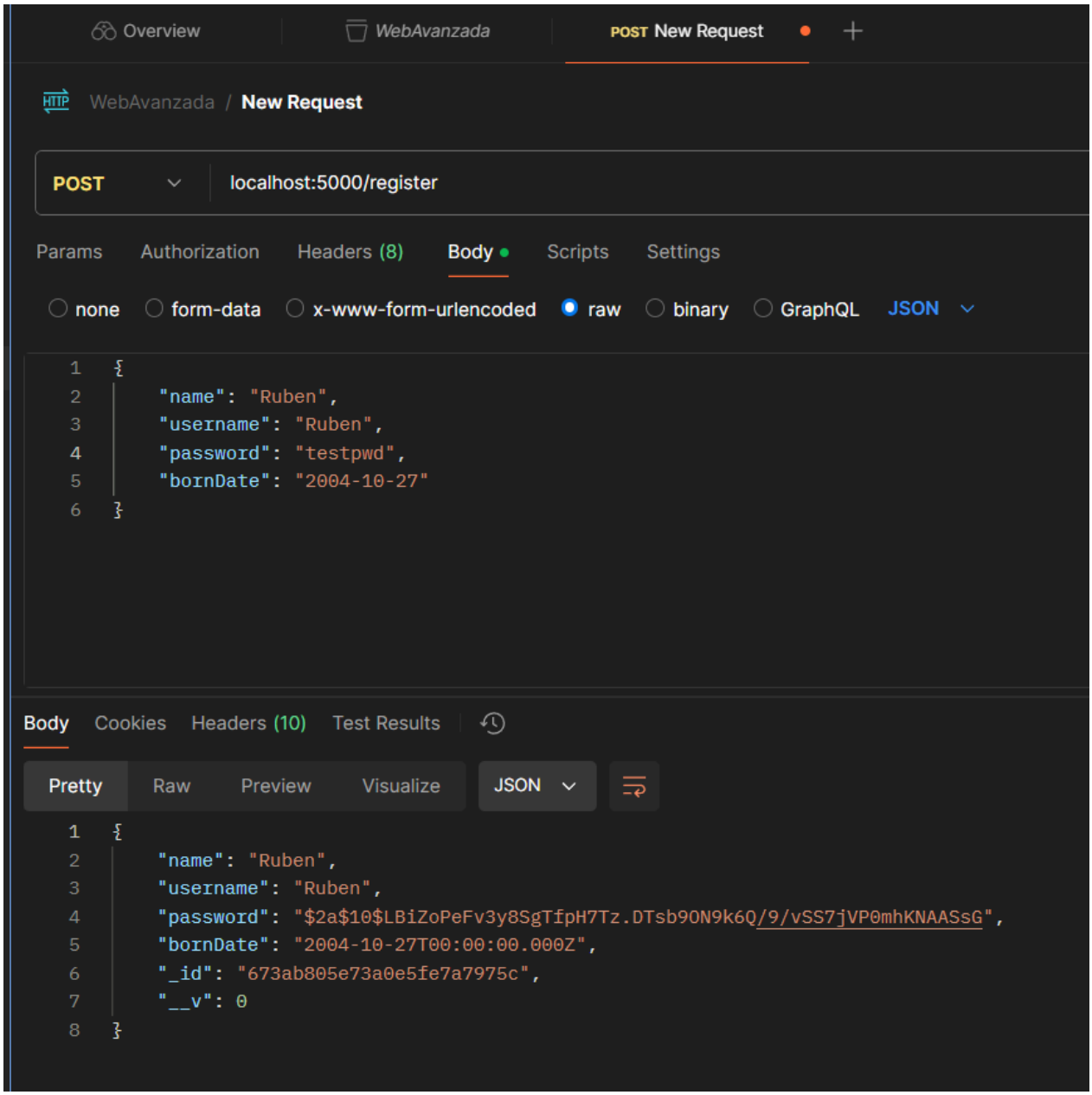
```
PS D:\ESPE\QUINTO\web_avanzado\p1\tareas\log_reg_API> node server.js
Serv Levantado.
Conexión Exitosa
```

Verificamos en postman la petición a register.





Ya habíamos puesto este usuario, pero podemos observar los distintos tipos de respuesta.



6.2 Frontend, para las peticiones.

6.2.1 Formulario de Reutilizable

Utilizamos un componente de API_FORM para un formulario reutilizable diseñado para envolver otros componentes y manejar su envío.

Utilizamos **children** para que el contenido del formulario sea renderizado entre las etiquetas y **data** como un objeto con las propiedades **data.onSubmit** y **data.msg** para configurar el formulario con la función y el mensaje que aparecerá en el botón de envío.

Damos propiedades adicionales para el diseño visual del formulario con **card** y **shadow-sm** para bordes y sombras, dentro de este segmento utilizamos **children** para renderizar los componentes que se pasen dentro de las etiquetas de **API_FORM**.

Finalmente se dan estilos básicos al botón.

```
function API_FORM({children, data}){

    return(
        <form className="form container mt-5" onSubmit={data.onSubmit}>
            <div className='form-container card p-4 shadow-sm'>
                {children}
                <button className='button-submit btn-primary w-100 mt-3'>{data.msg}</button>
            </div>
        </form>
    );
}

export default API_FORM;
```

6.2.2 Página de inicio

Importamos los componentes **frmLogin** y **frmRegister** que son los formularios de inicio de sesión y registro de los archivos **login** y **reg_form.**

La función home recibe el objeto data como prop y verifica si el usuario está logueado, si es que el usuario lo está imprime un mensaje de bienvenida con el nombre del usuario, caso contrario se muestran los botones para registrarse que renderizan el formulario de registro y de inicio de sesión mediante un evento **onClick.**

```
import frmLogin from './login';
import frmRegister from './reg_form';

export default function Home({data}){
    if(data.logIn){
        return(
            <h1 className="container text-center mt-5">HOLIII {data.username}!</h1>
        );
    }else{
        return(
            <div className="container text-center mt-5">
                <h1>HOLIII</h1>
                <div className="mt-4">
                    <div className="mb-3">
                        <button onClick={<frmRegister></frmRegister>} className="btn btn-primary btn-lg"> Registrarse </button>
                    </div>
                    <div className="mb-3">
                        <button onClick={<frmLogin></frmLogin>} className="btn btn-secondary btn-lg"> Iniciar Sesión </button>
                    </div>
                </div>
            </div>
        );
    }
}
```

6.2.3 Componentes Reutilizables

Se definen tres componentes reutilizables para generar campos de entrada para el formulario: **txtInput**, **dateInput** y **passwordInput.** Cada uno representa un tipo de campo específico.

Para **txtInput** utilizamos campos como **nombreInput** para nombre e ID del campo, **label** para la etiqueta asociada al campo, **value** para el valor del campo y **pattern** expresión regular para validar el contenido. Dejamos el caso base para el tipo y el nombre de la clase del input y modificamos los demás atributos dependiendo de la información ingresada en el llamado a la función.

```
function txtInput(nombreInput, label, value, pattern='') {
    return (
        <div className="form-group">
        <label htmlFor={nombreInput}>{label}</label>
        <input
            type="text"
            className="form-control"
            id={nombreInput}
            name={nombreInput}
            value={value}
            pattern={pattern}
            required
        />
        </div>
    );
}
```

passwordInput es similar a **txtInput**, pero se cambia el tipo con *type="password"* para ocultar los caracteres ingresados.

```
function passwordInput(nombreInput, label, value, pattern='') {
    return (
        <div className="form-group">
        <label htmlFor={nombreInput}>{label}</label>
        <input
            type="password"
            className="form-control"
            id={nombreInput}
            name={nombreInput}
            value={value}
            pattern={pattern}
            required
        />
        </div>
    );
}
```

dateInput es igual, el único cambio que se evidencia es que el tipo cambia, *type="date"*.

```
function dateInput(nombreInput, label, value, max) {
    return (
        <div className="form-group">
        <label htmlFor={nombreInput}>{label}</label>
        <input
            type="date"
            className="form-control"
            id={nombreInput}
            name={nombreInput}
            value={value}
            max={max}
            required
        />
        </div>
    );
}

export default {txtInput, dateInput, passwordInput};
```

6.2.4 Formulario de Registro

En este formulario se utiliza **AXIOS** para realizar solicitudes HTTP, utilizamos también el formulario reutilizable e importamos **express** mediante **e** para ser utilizado y finalmente importamos **txtInput**, **dateInput** y **passwordInput** del archivo **inputs** como componentes reutilizables.

```
const AXIOS = require('axios');
import API_FORM from './form';
import e from 'express';
import {txtInput, dateInput, passwordInput} from './inputs';
```

Inicializamos **actualDate** con la fecha actual.

La función register previene el comportamiento por defecto del formulario con el método **preventDefault**, utilizamos un **try-catch** para enviar los datos del formulario esperando la respuesta de **AXIOS**, si axios no encuentra respuesta o si **formData** no está declarado ni definido ocurrirá un error que se muestra en consola.

```
export default function frmLogin() {
    actualDate = new Date();

    const login = async (e) => {
        e.preventDefault();
        try {
            const response = await AXIOS.post(
            'https://web-avanzada-1.onrender.com/login',
            formData
            );
        }
        catch (error) {
            console.error('Error al iniciar sesión:', error);
        }
    };
```

Si pasa la verificación, la función retornará un formulario renderizado con las propiedades **onSubmit** que ejecuta la función **register** al enviar el formulario y **msg** que cambia el contenido del botón de envío. Contiene campos para Nombre, Usuario, Contraseña y Fecha de nacimiento.

```
return (
    <API_FORM data={{
        onSubmit: (e) => login(e),
        msg: 'Iniciar Sesión'
    }}>
        <txtInput name={'username'} label={'Usuario'} value={formData.username}></txtInput>
        <passwordInput name={'password'} label={'Contraseña'} value={formData.password}></passwordInput>
    </API_FORM>
);
```

6.2.5 Formulario de Inicio de Sesión

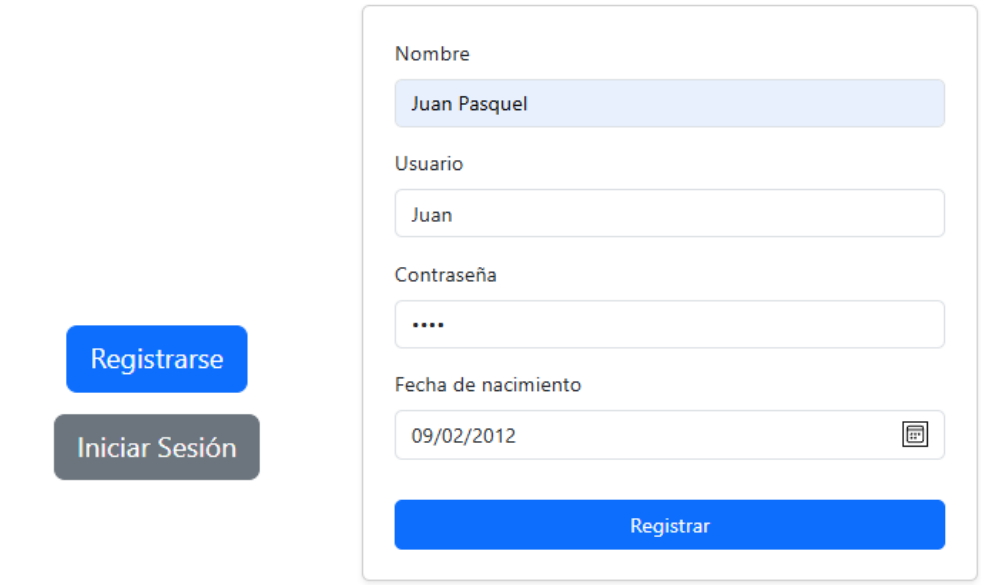
Aquí definimos el formulario de inicio de sesión, con una estructura similar al de registro pero con la diferencia que solo vamos a utilizar los campos **txtInput** y **passwordInput** del archivo inputs.

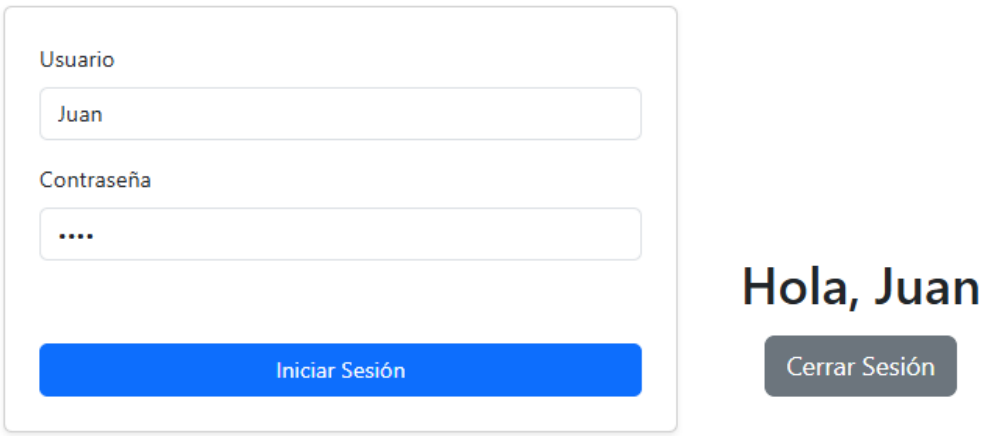
El formulario de inicio de sesión envía una solicitud HTTP POST al servidor con los datos de inicio de sesión de **formData**, e imprime un mensaje de error si falla la solicitud.

El formulario se renderiza a través de **API_FORM** que renderiza los campos para ingresar el nombre de usuario y la contraseña.

```jsx
const AXIOS = require('axios');
import API_FORM from './form';
import e from 'express';
import {txtInput, dateInput, passwordInput} from './inputs';

export default function frmLogin() {
    actualDate = new Date();

    const login = async (e) => {
        e.preventDefault();
        try {
            const response = await AXIOS.post(
            'https://web-avanzada-1.onrender.com/login',
            formData
            );
        }
        catch (error) {
            console.error('Error al iniciar sesión:', error);
        }
    };

    return (
        <API_FORM data={{
            onSubmit: (e) => login(e),
            msg: 'Iniciar Sesión'
        }}>
            <txtInput name={'username'} label={'Usuario'} value={formData.username}></txtInput>
            <passwordInput name={'password'} label={'Contraseña'} value={formData.password}></passwordInput>
        </API_FORM>
    );
}
```

7. Resultados





8. Conclusiones

- Presenta un resumen de los hallazgos principales y analiza si se cumplieron los objetivos del informe.

- Menciona las implicaciones de los resultados y lo que estos significan en el contexto del tema investigado.

- Utilizar herramientas como Docker para evitar discrepancias de configuración entre entornos locales y de producción.

9. Recomendaciones

- Para tener una documentación más detallada, podemos crear un archivo README que incluya instrucciones para configurar cookies y realizar el despliegue en plataformas específicas.
- Podemos tener capacitaciones adicionales para ampliar los conocimientos sobre cookies y despliegue en diferentes plataformas mediante talleres o cursos.

- Utilizar herramientas como Docker para evitar discrepancias de configuración entre entornos locales y de producción.

10. Bibliografía o Referencias

[1] Express.js. (n.d.). *Routing*. Retrieved November 17, 2024, from https://expressjs.com/en/guide/routing.html

[2] MongoDB. (n.d.). *Node.js driver documentation*. Retrieved November 17, 2024, from https://www.mongodb.com/docs/drivers/node/current/

[3] Mongoose. (n.d.). *User guide*. Retrieved November 17, 2024, from https://mongoosejs.com/docs/guide.html

[4] npm. (n.d.). *bcryptjs*. Retrieved November 17, 2024, from https://www.npmjs.com/package/bcryptjs

[5] npm. (n.d.). *jsonwebtoken*. Retrieved November 17, 2024, from https://www.npmjs.com/package/jsonwebtoken

[6] Postman. (n.d.). *Postman learning center: Overview*. Retrieved November 17, 2024, from https://learning.postman.com/docs/introduction/overview/

[7] React. (n.d.). *React docs: Learn React*. Retrieved November 17, 2024, from https://react.dev/learn