INSTITUTO TECNOLÓGICO SUPERIOR DE GUASAVE

TITULO MANUAL DE PRACTICAS PARA DESARROLLO DE BACK END USANDO TECNOLOGIAS WEB.

Materia: PROGRAMACIÓN WEB.

FRANCISCO JAVIER ARCE CARDENAS

ELABORADO ENERO- JUNIO DEL 2025

---

INTRODUCCION

---

Express.js es un framework web para Node.js que se utiliza para crear aplicaciones web y APIs. Es el framework web más popular de Node.js. Características de Express.js Es un framework de código abierto y gratuito Permite crear aplicaciones web de una sola página, multipágina e híbridas Simplifica el desarrollo de aplicaciones web y APIs Proporciona funcionalidades como enrutamiento, gestión de sesiones y cookies Tiene una curva de aprendizaje baja, especialmente para quienes ya conocen JavaScript Es altamente compatible con otras tecnologías Cómo se utiliza Express.js Se utiliza junto con Node.js como plataforma de tiempo de ejecución Se puede utilizar para construir aplicaciones web y APIs Se puede utilizar para crear una API sólida Se puede utilizar para gestionar rutas y peticiones HTTP

Para iniciar se crea el archivo de package.json dentro de la carpeta de trabajo con:

```
npm init -y
```

Después se tiene que instalar express:

```
Npm i express
```

Una vez instalado se crea la carpeta src y el archivo app.js dentro de ella y posteriormente dentro del archivo se importa la librería express con la siguiente línea Import express from ‘express’ Adicionalmente se crea la variable app instanciada de express y se puede levantar el servicio con la instrucción listen mas el puerto que en este caso es 3000 como lo podemos observar acontinuacion.

```
import express from 'express'
//se cre la app
const app = express()
// se lanza con el puerto 300 con:
app.listen(3000)
console.log('Server on port', 3000)
```

para lanzarlo se usa el comando:

```
node src/app.js
```

Pero hasta aquí ocurre un error ya que por defecto js no reconoce los imports, para reparar este problema hay que ir al archivo 'package.json' y agregar el atributo "type":"module", para que reconozca los imports. Si se corre de nuevo `node src/app.js` ya tendríamos response, mal pero ya respondería el servicio.

Y eso es porque aún no hay contenido para mostrar de la página, para tener se crea la carpeta `routes` dentro de `src` para poder especificar rutas de archivos. También se creará la carpeta `controllers` para almacenar funciones de nuestra página como el gestor del login y otras. Adicionalmente hay que crear una carpeta llamada models para guardar los modelos de datos. La carpeta `middlewares` se necesita para gestionar contenido a nivel usuario (permisos). Necesitamos la carpeta para validaciones de errores que llamaremos `schemas`. También necesitamos la carpeta `libs` tendrá código reutilizable. Y por último unos archivos adicionales también dentro de src:

```
Db.js (gestión de la conexión de la base de datos).
Config.js (se genera para reutilizar configuraciones como configuraciones
globales).
Index.js(a donde se dirigirán inicialmente los contenidos).
```

Una vez creado todo el conjunto de carpetas y archivos proseguimos con algunas configuraciones. Los primero es ir a app.js y cortar las ultimas 2 lineas y llevaras a index.js, una vez eliminadas sustituir esas 2 lineas:

```
app.listen(3000)
console.log('Server on port', 3000)
```

(con la siguiente 'express defautl app"; esto permitirá exportar la app a otros archivos, lo siguiente es ir a `index.js` y agregar la siguiente línea:

```
import app from './app.js'
```

quedando los archivos de la siguiente manera:

`index.js`

```
import app from './app.js'

app.listen(3000)
console.log('Server on port', 3000)
app.js
import express from 'express'
//se crea la app
const app = express()
```

```
// se lanza con el puerto 3000 con:
export default app;
```

Se levanta el servicio ahora con node `src/index.js` y seguirá mandando el error cant get pero ya está creada la estructura, adicionalmente podemos instalar otra dependencia para que no tengamos que estar bajando y subiendo el servicio cada que hagamos algún cambio `npm i nodemon`, una vez instalado vamos al archivo package json y en el elemento “scripts” agregamos el elemento “dev” de la siguiente manera.

```
"dev" : "nodemon src/index.js"
```

Una vez hecho esto solo iniciamos el server con la instrucción

`npm run dev`

Nos falta un logger para ver la actividad de peticiones y respuestas del servidor, para eso agregaremos el administrador de paquetes Morgan desde la terminal con la instrucción `npm i Morgan` una vez instalado hay que instanciarlo en el archivo app.js y quedaría asi:

```
import express from 'express'
import morgan from 'morgan'

const app = express();
app.use(morgan('dev'));
export default app;
```

---

BASE DE DATOS

---

Hasta ese punto ya debemos tener instalado mongodb comunnity para continuar.

Si ya esta necesitamos el paquete mongoose para poder conectarnos con mongodb, lo instalamos con la siguiente instrucción.

`Npm i mongoose`

Una vez instalado procederemos a instanciar el driver de mongo en el archivo `db.js` en el cual manejaremos todo lo referente a la base de datos. Lo primero que hay que hacer es importar mongoose.

```
import mongoose from "mongoose";
```

Y para poder realizar la conexión crearemos una función llamada connectDB que será asíncrona la sintaxis es la siguiente:

```
export const connectDB = async () => {
    try {
        await mongoose.connect('mongodb://localhost/merndb')
        console.log(">>> DB conectada");
    } catch (error){
        console.log(error);
    }


};
```

Esta función incluye un try catch para verificar que la conexión sea exitosa, la línea del await contiene la cadena de conexión a la base de datos que queramos. Se le agrega un consolelog nomas para notificar en la consola que, si se realizó la conexión, hasta aquí ya está la preparación de la base de datos. Lo siguiente es consumir la base de datos y para ello iremos al archivo `index.js` Donde importaremos e invocaremos la función connectDB

```
import app from './app.js'
import { connectDB } from './db.js'

connectDB();
app.listen(3000)
console.log('Server on port', 3000);
```

al guardar en el console log podemos ver si se conectó o no la base de datos.

```
[nodemon] restarting due to changes...
[nodemon] starting `node src/index.js`
Server on port 3000
>>> DB conectada
```

Lo siguiente es crear un archivo en models llamado `user.models.js`, dentro importaremos mongoose para utilizar su esquema con datos para almacenar con algunos atributos.

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
    username:{
        type:String,
        required : true,
        trim : true,
    },
    email :{
        type:String,
        required : true,
        trim : true,
```

```
        unique : true
    },
    password :{
        type:String,
        required : true,
        trim : true,
    },

})
export default mongoose.model('User','userSchema')
```

la función me dice que datos voy a manipular y el export me da la interacción con la base de datos desde otras ubicaciones.

Lo siguiente es configurar el archivo de rutas(`src/routes/auth.routes.js`) para los componentes del sitio con 2 rutas una de registro y la del login:

```
import { Router } from "express";

const router = Router()

router.post('/register')
router.post('/login')
export default router
```

después iremos a la carpeta controllers y generaremos el archivo `auth.controller.js` que tendrá funciones para procesar peticiones:

```
export const register = (req, res) => res.send("registra");
export const login = (req, res) => res.send("logiado");
```

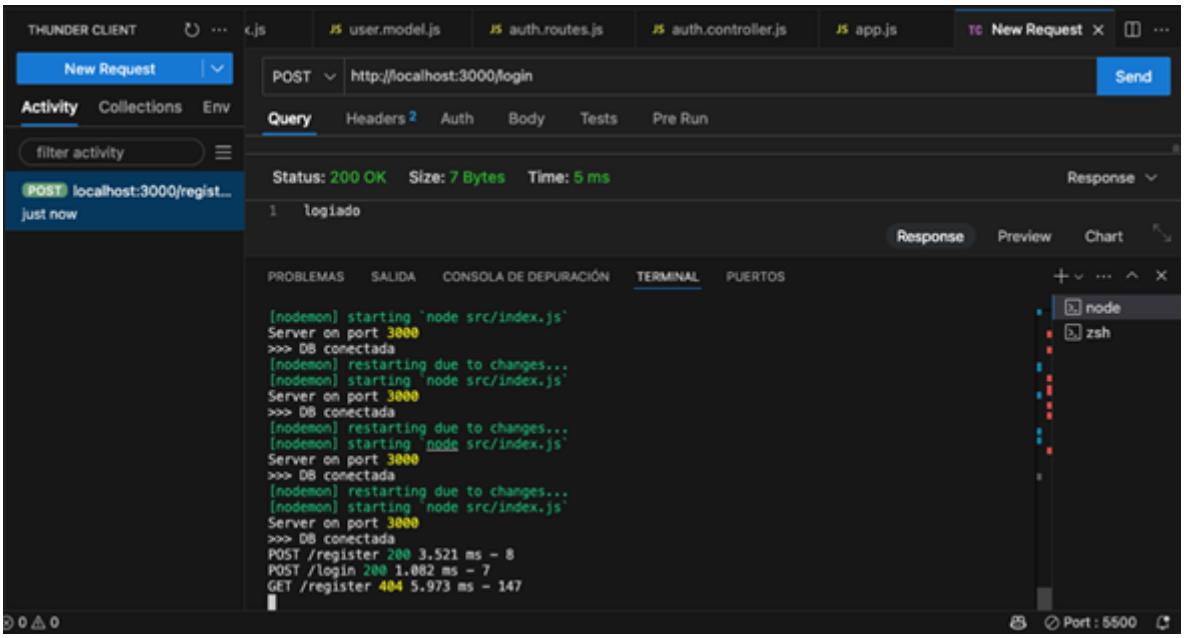
De aquí volvemos al archivo que creamos en routes para actualizarlo relacionándolo con las peticiones de register y login:

```
import { Router } from "express";
import {login, register} from '../controllers/auth.controller.js'
const router = Router()

router.post('/register', register)
router.post('/login', login)
export default router
```

de aquí nos vamos al archivo `app.js` y le agregamos las siguientes líneas para usar authRouthes que serian la línea 3 y la línea 6

```
import express from 'express'
    import morgan from 'morgan'
    import authRoutes from "./routes/auth.routes.js"

    const app = express();

    app.use(morgan('dev'));
    app.use(authRoutes)
    export default app;
```

Para poder hacer una petición de prueba instalaremos la extensión Thunder Client para hacerle pruebas.




Por motivos de formato u organización de ulrs se agrega par a la petición de registro el subfijo api se puede hacer en routes en el archivo que se quiere hacer el cambio en el router post por esta línea:

```
import { Router } from "express";
import {login, register} from '../controllers/auth.controller.js'
const router = Router()
router.post('/api/register', register)
router.post('/api/login', login)
export default router
```

o bien se puede hacer en el archivo `app.js` en la línea de app.use de authRoutes:

```
app.use(morgan('dev'));
app.use("/api",authRoutes)
export default app;
```

Con esto las peticiones a :

```
localhost:3000/register
```

o

```
localhost:3000/login
```

Ya no funcionaran ahora tendríamos que hacer la petición asi:

```
localhost:3000/api/register
```

Para evitar redundancia se optara por el método de app.js.

Una vez hecho esto iremos al archivo `auth.controller.js` a especificar que en register en vez de contestar con “registra” un post ahora va recibir dato lo primero es actualizar estas líneas

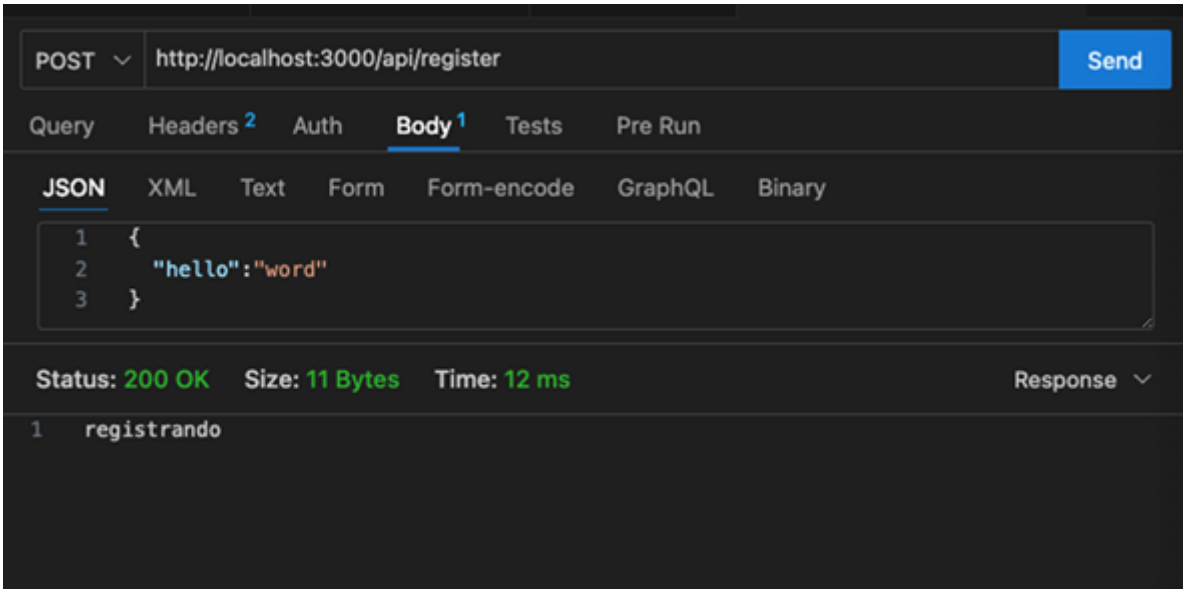
```
export const register = (req, res) => res.send("registra");
export const login = (req, res) => res.send("logiado");
```

por esto

```
export const register = (req, res) => {
    console.log(req.body);
    res.send('registrando')
}
```

Donde en el `console.log` mostraremos el body del request que es un json y la respuesta del request seria `res.send(‘registrando’)` lo siguiente seria hacer un post para ver que responde en la consola pero habría que enviarle un dato por thunder client ese dato es un json que lo pondremos en la sección body como se

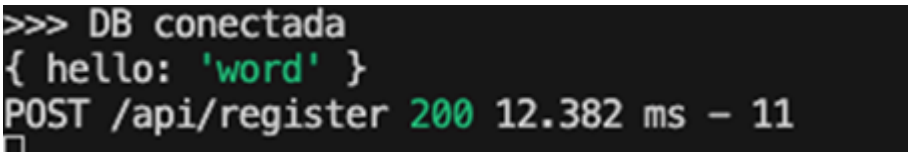
puede apreciar en la imagen de abajo.




Al ejecutarlo en la consola obtenemos un “undefined” porque actualmente nodejs no es capaz de leer un json nativamente, tenemos que agregar una app en el archivo app.js que habilite la lectura de jsons entonces nos movemos a app.js y debajo de Morgan agregamos:

```
import express from 'express'
import morgan from 'morgan'
import authRoutes from "./routes/auth.routes.js"

const app = express();

app.use(morgan('dev'));
app.use(express.json());
app.use("/api",authRoutes);
export default app;
```

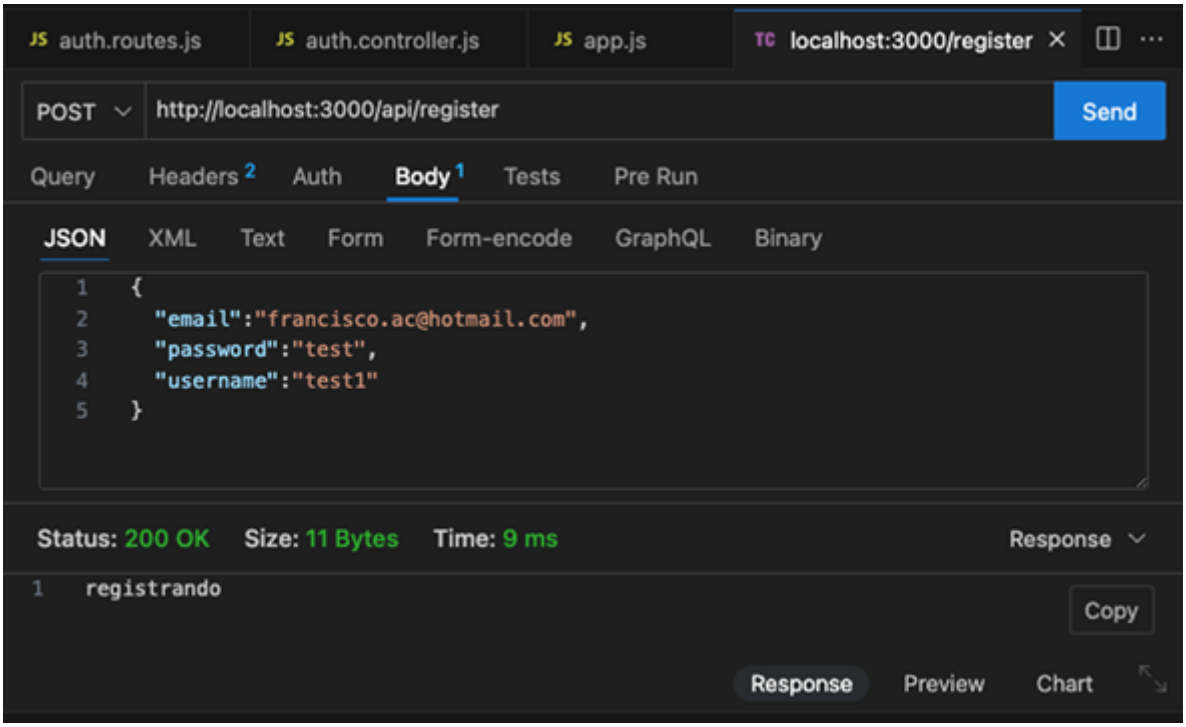
Volvemos a mandar el post y ahora si en el consolé saldrá el json que mandamos.



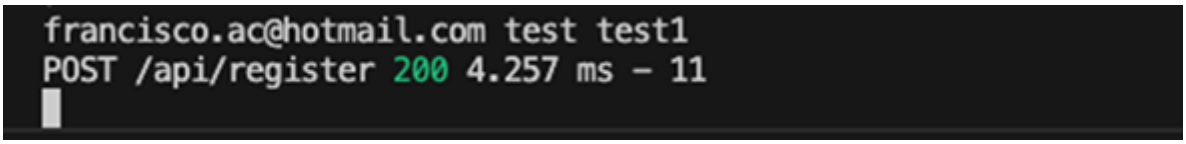

Para probar que podemos enviar en el body los 3 datos de usuarios que creamos en el modelo de datos de la carpeta models que son email usuario y password modificaremos el archivo que con la línea 2 cache los datos que vienen de la petición y los muestre por consola quedaría asi el archivo auth

```
export const register = (req, res) => {
    const {email,password,username} = req.body
    console.log(email,password,username)
    res.send('registrando')
}
export const login = (req, res) => res.send("logiado");
```

Al hacer el un post con el body modificado con los datos del username, password y email por consola obtendríamos el request de esta forma.




Con esta consola




Una vez confirmado que podemos cachar información procederemos a llamar al modelo de datos para utiliarlo. Primero nos vamos al archivo `auth.controller` e importamos el modelo con

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

eliminamos el console.log que teníamos y usamos el método créate para crear el usuario con los datos que cachamos de request body todo esto en `register`.

```
export const register = (req, res) => {
    const {email,password,username} = req.body
    User.create({
        username,
        email,
        password
    })

    res.send('registrando')
}
export const login = (req, res) => res.send("logiado");
```

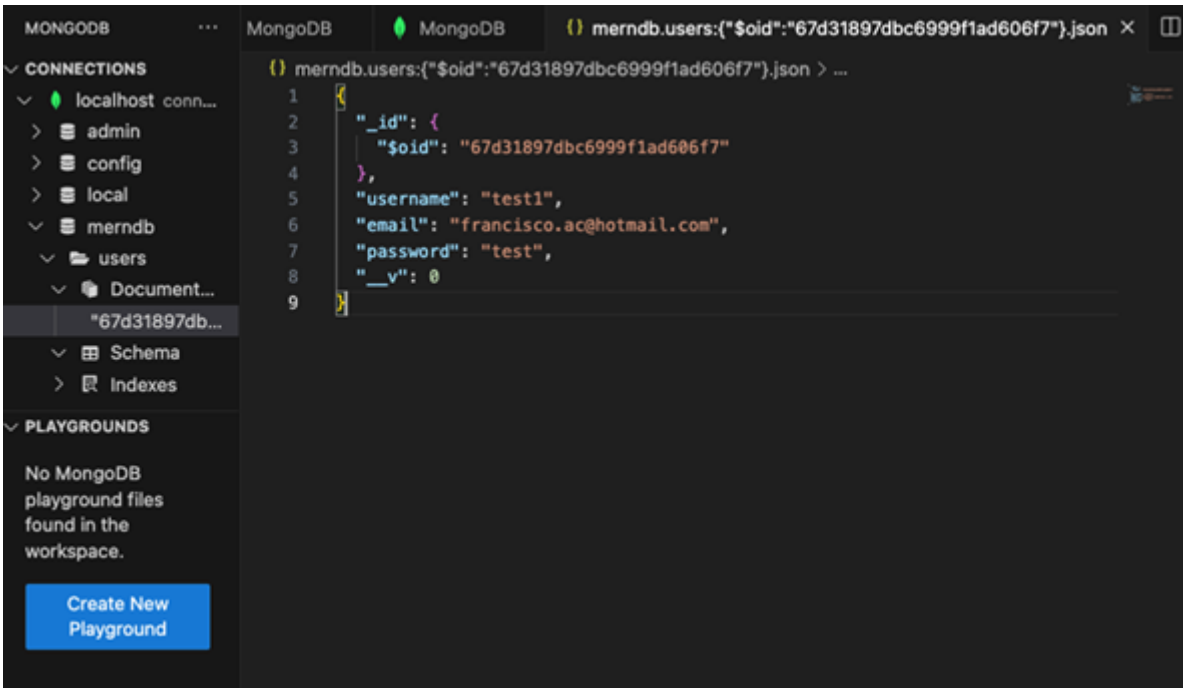
Pero es mas eficiente si los instanciamos a en el backend temporalmente antes de mandarlo a la base de datos por algunas cuestiones de validaciones o modificaciones. Quedaría de esta forma aun sin enviarlo a guardar, aun es temporal.

```
import User from '../models/user.model.js'
export const register = (req, res) => {
    const {email,password,username} = req.body
    const newUser = new User({
        username,
        email,
        password
    })
    console.log(newUser);
    res.send('registrando')
}
export const login = (req, res) => res.send("logiado");
```

Si hacemos una peticion con esto la consola nos mandaria el json que genero para mongo, para guárdarlo definitivamente como esta operación es asíncrona agregaremos lo que vimos en la conexión de la base de datos en `db.js` que seria agregarle el atributo `async` a la función `newUser` y en la línea de save agregar un `await` quedando finalmente de esta forma adiconalmente metiendo un `try catch`.

```
import User from '../models/user.model.js'
export const register = async (req, res) => {
    const {email,password,username} = req.body
  try {
    const newUser = new User({
        username,
        email,
        password
    });
    await newUser.save();
    res.send('registrando');
  }catch (error){
    console.log(error);
  }

}
export const login = (req, res) => res.send("logiado");
```

Al hacer el post ahora si se guardaría el dato en la base de datos, para verificarlo podemos hacerlo desde atlas o de algún manejador que tengamos instalado pero esto lo podemos hacer desde `mongodb for vsc`, instalamos la extensión y solo agregamos el localhost de nuestra comunnity mongodb (mongodb://localhost) o puede ser la url de la que tengamos en atlas. En la siguiente imagen veremos la captura que demuestra que ese dato ya esta en mongo.



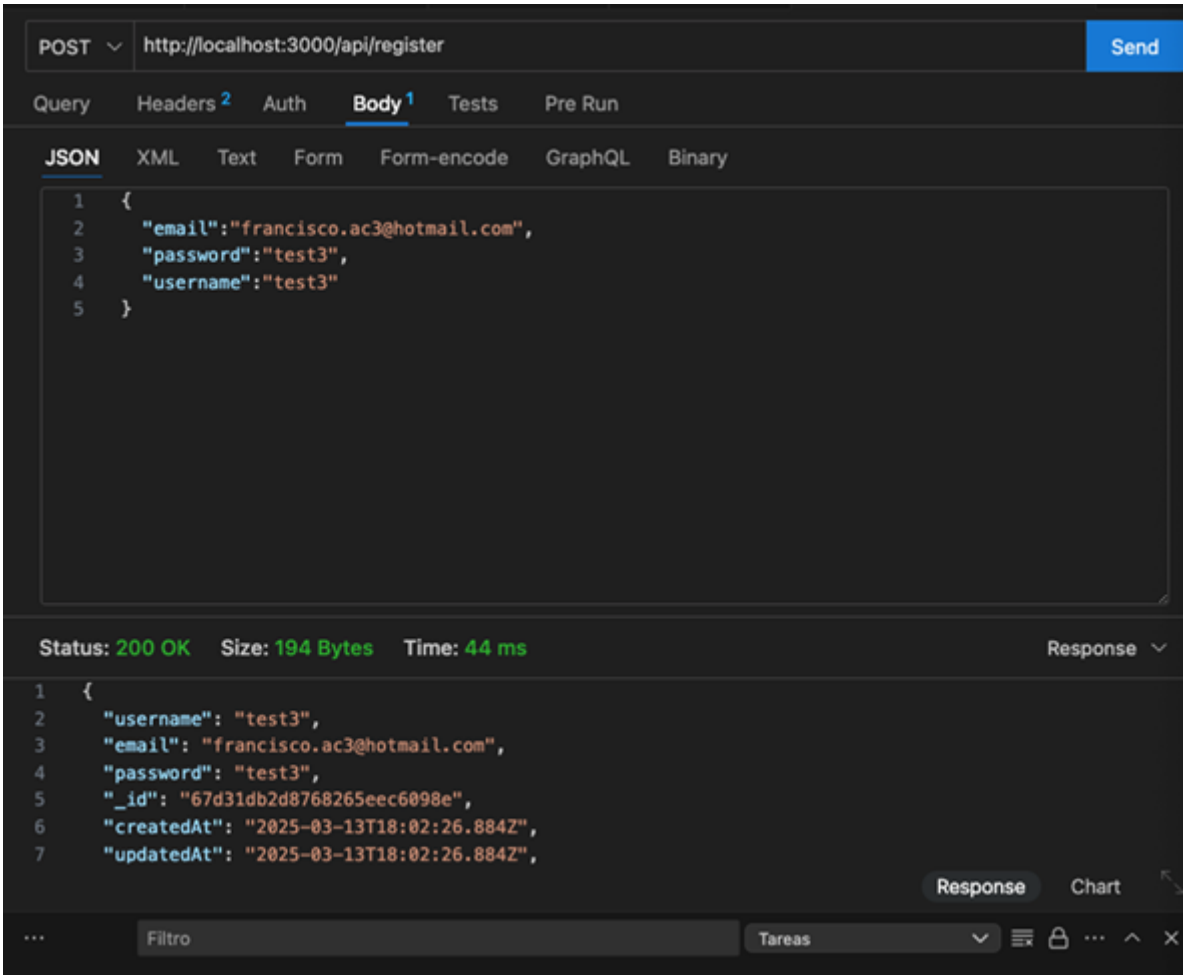

Lo siguiente será agregar un timestamp en el modelo de datos(`user.models.js`) justo después del ultimo registro para que nos autorregistre fecha y hora.

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
    username:{
        type:String,
        required : true,
        trim : true,
    },
    email :{
        type:String,
        required : true,
        trim : true,
        unique : true
    },
    password :{
        type:String,
        required : true,
        trim : true,
    }

},{
    timestamps:true
})
export default mongoose.model('User',userSchema)
```

Lo siguiente es notificar al frontend que los datos están correctos para eso vamos a modificar y agregar algunas líneas en el `auth.controller`, se puede regresar el `newUser` pero como por el modelo de datos se autocompleta con el ID y el timestamp si instanciamos el `newUser` a la variable `userSaved` podemos responderle a la petición todos los datos que se guardaron en mongo quedando de la siguiente manera.

```
import User from '../models/user.model.js'
export const register = async (req, res) => {
    const {email,password,username} = req.body
  try {
    const newUser = new User({
        username,
        email,
        password
    });
    const userSaved = await newUser.save();
    res.json(userSaved)
    res.send('registrando');
  }catch (error){
    console.log(error);
  }

}
export const login = (req, res) => res.send("logiado");
```

con esta actualización en el response de la petición tendríamos lo siguiente




---

CREACION DEL TOKEN

Vamos a mejorar la contraseña con encriptación.

Para lograr esto instalaremos un nuevo modulo llamado `bcryptjs` desde consola, detendremos temporalmente el servidor para hacer esto, y con `npm i bcryptjs` se instalara el modulo y lo utilizaremos en `auth.controller` iniciaremos importándolo:

```
import User from '../models/user.models.js';
import bcrypt from 'bcryptjs';
export const register = async (req, res) => {
    const { name, email, password } = req.body;
    //User.create({ name, email, password });
    try {
        const passwordHash = await bcrypt.hash(password, 10);
        const newUser = new User({ name, email, password : passwordHash });
        const userSaved = await newUser.save();
        res.json({
            id: userSaved._id,
            name: userSaved.name,
            email: userSaved.email
        });
    } catch (error) {
        console.log(error);
        return res.status(500).json({ message: 'Error al registrar el usuario' });
    }

};
export const login = (req, res) => {};
```

Lo que sucede aquí es que el password que creo el usuario ahora se almacenara con hash 10, pero al hacer la petición este regresara el hash lo que no es seguro por lo que tenemos que modificar que es lo que vamos a regresar que seria todo menos password.

Esto nos servirá como credencial del usuario para tener sus datos en los lugares a los que pueda ir o no en la aplicación y para sustituir todos estos datos necesitamos un `webtoken` que lo puede manejar json, para utilizalo debemos instalarlo como ya lo hemos hecho con anteriormente con `npm i jsonwebtoken` y hay que importarlo en `auth.controller`:

```
import jwt from 'jsonwbtoken';
```

Ahora vamos a modificar un poco lo que ya tenemos, para empezar después del `userSaved` vamos a crear un webtoken con el valor sign lo vamos a relacionar con la sesión del usuario por lo pronto tendrá esta estructura.

```
jwt.sign({
      id:userSaved._id,
```

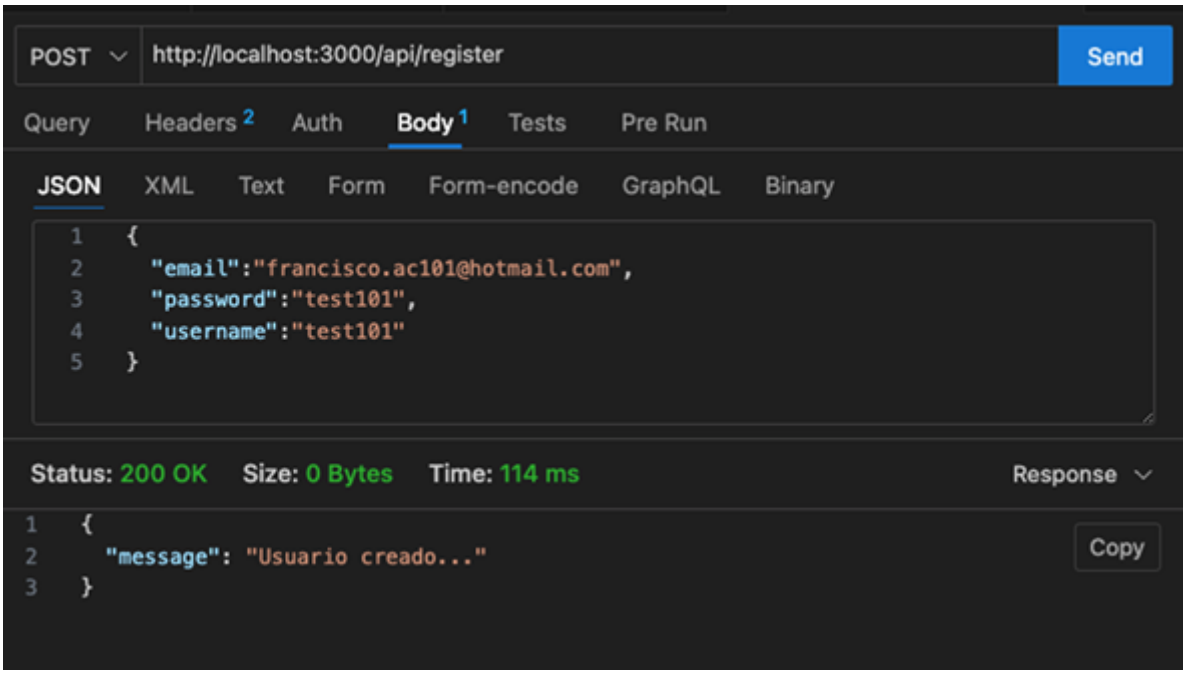
```
    },
    'secret123',
    {
      expiresIn:"1d",
    },
    (err,token) => {
      if (err) console.log(err);
      res.json({token});
    }
  );
```

Donde el payload sería el id que queremos que se lleve una palabra secreta, funciones adicionales como expiración, y un callback para hacer esta parte asíncrona.

En la siguiente sección crearemos un cookie para enviar la una cookie con el token y mandar un mensaje de respuesta de usuario creado:

```
const userSaved = await newUser.save();
    jwt.sign({
      id:userSaved._id,
    },
    "secret123",
    {
      expiresIn:"1d",
    },
    (err,token) => {
      if (err) console.log(err);
      //res.json({token}); inicialmente mandabamos un token
      res.cookie('token',token);// esta funcion es de express
      res.json({
        message : "Usuario creado...",
      })

    }
  );
```

Si creamos de nuevo un usuario obtendremos esto en la petición.




Para poder sacar los datos del token de la cookie, por cuestiones de orden se puede manejar los jwt desde su propio apartado, para ello crearemos en la carpeta libs, regresamos a auth.controller y sacamos los 2 res que creamos anteriormente justo debajo del primer paréntesis cerrado con ; que encontremos debajo:

```
(err,token) => {
      if (err) console.log(err);
      //res.json({token}); inicialmente mandabamos un token (Este)


    }
  );
  res.cookie('token',token);// esta funcion es de express
  res.json({
    message : "Usuario creado...",
  })

    // res.json({ (Este)
```

Todo lo que quede del `jwt.sign` nos lo llevamos a al archivo `jwt.sign.js` que creamos en libs y le creamos su propia función para poder ejecutarlo las veces que se necesite pero eliminaremos el diccionario de id y lo sustituimos por el argumento que estamos recibiendo de la función “payload”

```
function createAccesToken(payload){
jwt.sign(
payload,
    "secret123",
  {
    expiresIn:"1d",
  },
  (err,token) => {
```

```
    if (err) console.log(err);
    //res.json({token}); inicialmente mandabamos un token


  }
);
}
```

Para quitar el “secret del token podemos hacer lo siguiente, iremos a `config.js` y crearemos la variable TOKEN_SECRET para manipular la clave.

```
export const  TOKEN_SECRET = 'crea una clave secreta'
```

de aquí volvemos a `jwt.sign.js` y tenemos que importar la variable que creamos en `config.js` y sustituir el secret por la variable `TOKEN_SECRET`, por cuestiones de validaciones crearemos un `Promise` el cual es un objeto global que tiene node para resolver o rechazar una petición el cual contendrá el código del jwt, nota hay que importar la libreria jsonwebtoken.

```
import { TOKEN_SECRET } from "../config";
import jwt from 'jsonwebtoken';
export function createAccesToken(payload){
    return new Promise((resolve, reject) => {
        jwt.sign(payload,
            TOKEN_SECRET,
            {
              expiresIn:"1d",
            },
            (err,token) => {
              if (err) reject(err);
              resolve(token)
            });
    })

}
```

Dejado el archivo así nos vamos a authcontroller y vamos a importar la función de `jwt.sign.js` `createAccesToken` y lo invocaremos para instanciar el return en la variable token y reactivaremos el response el json que habíamos ignorado anteriormente para notificar al usuario si hay un error y vamos al catch un status 500 quedando el try catch de `auth.controller` asi :

```
import { createAccesToken } from '../libs/jwt.js';
export const register = async (req, res) => {
    const {email,password,username} = req.body
try {
    const passwordHash = await bcrypt.hash(password,10);
    const newUser = new User({
```

```
        username,
        email,
        password : passwordHash,
    });
  const userSaved = await newUser.save();
  const token = await createAccesToken({id : userSaved._id});
  res.cookie('token',token);// esta funcion es de express
  res.json({
      id:userSaved._id,
      username: userSaved.username,
      email : userSaved.email,
      timesst : userSaved.createdAt

    })

  }catch (error){
    res.status(500).json({ message : error.message })
  }
```

---

Siguiente sección creando la ruta del login

---

Para empezar, lo haremos en el `auth.controller` copiaremos por completo register y borramos la ultima línea que corresponde al `login` y pegaremos `register`, esta nos servirá para reutilizar algunas cosas y crear el nuevo `login`, primero iniciaremos modificándole el nombre a login, lo siguiente es que ya no se necesita el `username` del `req.body` asi que solo lo quitamos, dentro del `try` agregaremos una consulta a mongo para ver si el email existe.

```
const userFound = await User.findOne({email})
```

y lo manejamos con el siguiente if

```
if (!userFound) return res.status(400).json({ message : 'Usuario no encontrado'});
```

si lo encuentra ahora verifica si las credenciales son correctas

```
const isMatch = await bcrypt.compare(password,userFound.password);
  if (!isMatch) return res.status(400).json({message : 'Password o usuario
incorrectos'})
```

por ultimo asi como en el token del register tendremos que modificar todas las varibles de `userSaved` por `userFound`

Solo quedaría crear el controlador para logout

```
export const logout = (req,res) => {
  res.cookie('token',"",{expires:new Date(0)})
  return res.sendStatus(200)
}
```

Y se agrega al auth.routes.js

```
import { Router } from "express";
import {login, register, logout} from '../controllers/auth.controller.js'
const router = Router();

router.post('/register', register);
router.post('/login', login);
router.post('/logout', logout);

export default router
```

El auth.controller quedaría asi por completo

```
import User from '../models/user.model.js';
import bcrypt from 'bcryptjs';
import { createAccesToken } from '../libs/jwt.js';
export const register = async (req, res) => {
    const {email,password,username} = req.body
  try {
    const passwordHash = await bcrypt.hash(password,10);
    const newUser = new User({
        username,
        email,
        password : passwordHash,
    });
  const userSaved = await newUser.save();
  const token = await createAccesToken({id : userSaved._id});
  res.cookie('token',token);// esta funcion es de express
  res.json({
      id:userSaved._id,
      username: userSaved.username,
      email : userSaved.email,
      timesst : userSaved.createdAt

    })

  }catch (error){
    res.status(500).json({ message : error.message })
  }
```

```
}

//--------------------------------------------------
export const login = async (req, res) => {
  const {email,password } = req.body
try {
  const userFound = await User.findOne({email});
  if (!userFound) return res.status(400).json({ message : 'Usuario no
encontrado'});
  const isMatch = await bcrypt.compare(password,userFound.password);
  if (!isMatch) return res.status(400).json({message : 'Password o usuario
incorrectos'})



const token = await createAccesToken({id : userFound._id});
res.cookie('token',token);// esta funcion es de express
res.json({
    id:userFound._id,
    username: userFound.username,
    email : userFound.email,
    createdAt: userFound.createdAt
    updatedAt: userFound.updatedAt,



  })

}catch (error){
  res.status(500).json({ message : error.message })
}

}

export const logout = (req,res) => {
  res.cookie('token',"",{expires:new Date(0)})
  return res.sendStatus(200)
}
```

Solo queda que vayas a hacer post de login para verificar que si está borrando el token logout y que si se mantiene el token si haces login en thunder client.

---

Validación de Token y rutas protegidas

---

Crear una función que verifique que el usuario está conectado

Para eso necesitamos iniciar creando una ruta a la que llamaremos `profile` en `auth.routes.js`

```
router.post('/register', register);
router.post('/login', login);
router.post('/logout', logout);
router.get('/profile');
```

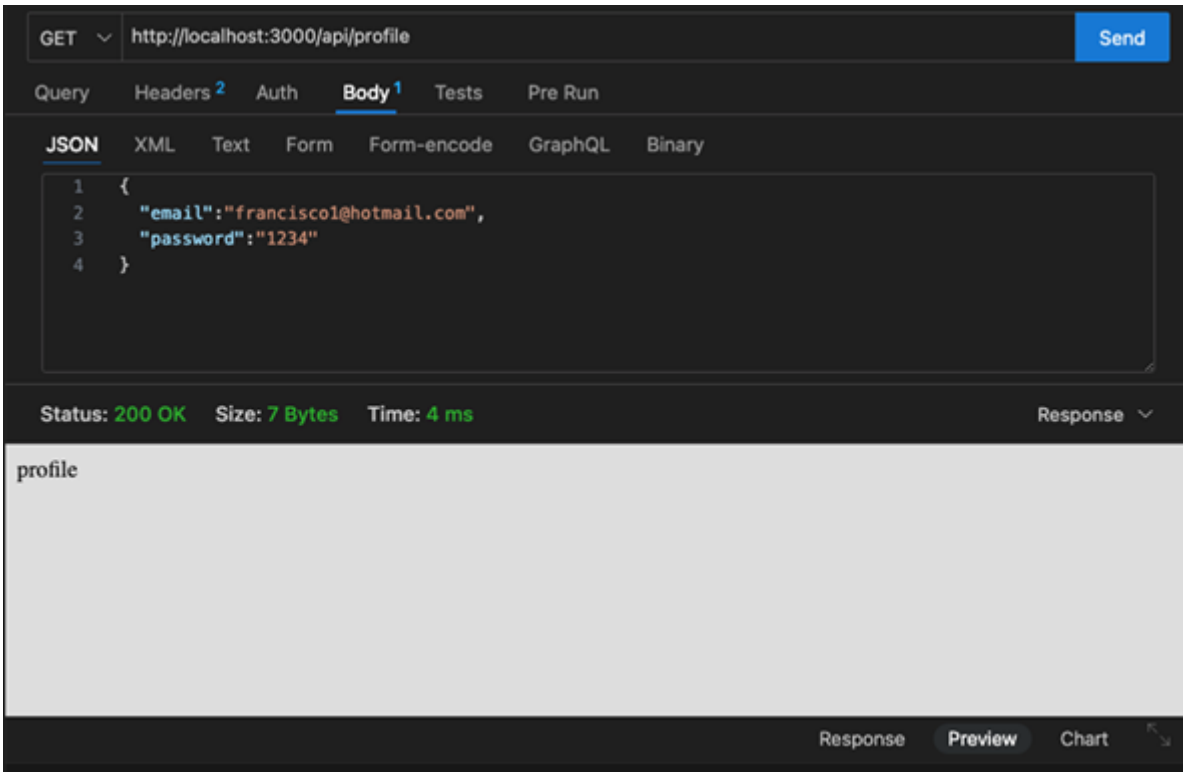
esto dejara listo el archivo para trabajar en el, ahora debemos ir a `auth.controller` para crear la función que va a interactuar con esta ruta.

```
export const profile = (req,res) => {
res.send("profile");
```

y regresamos a `routes` para importarlo.

```
import { Router } from "express";
import {
    login,
    register,
    logout,
    profile,
}
```

Si hacemos un get con tunder client a profile con el body de correo y password nos debe responder “Profile”



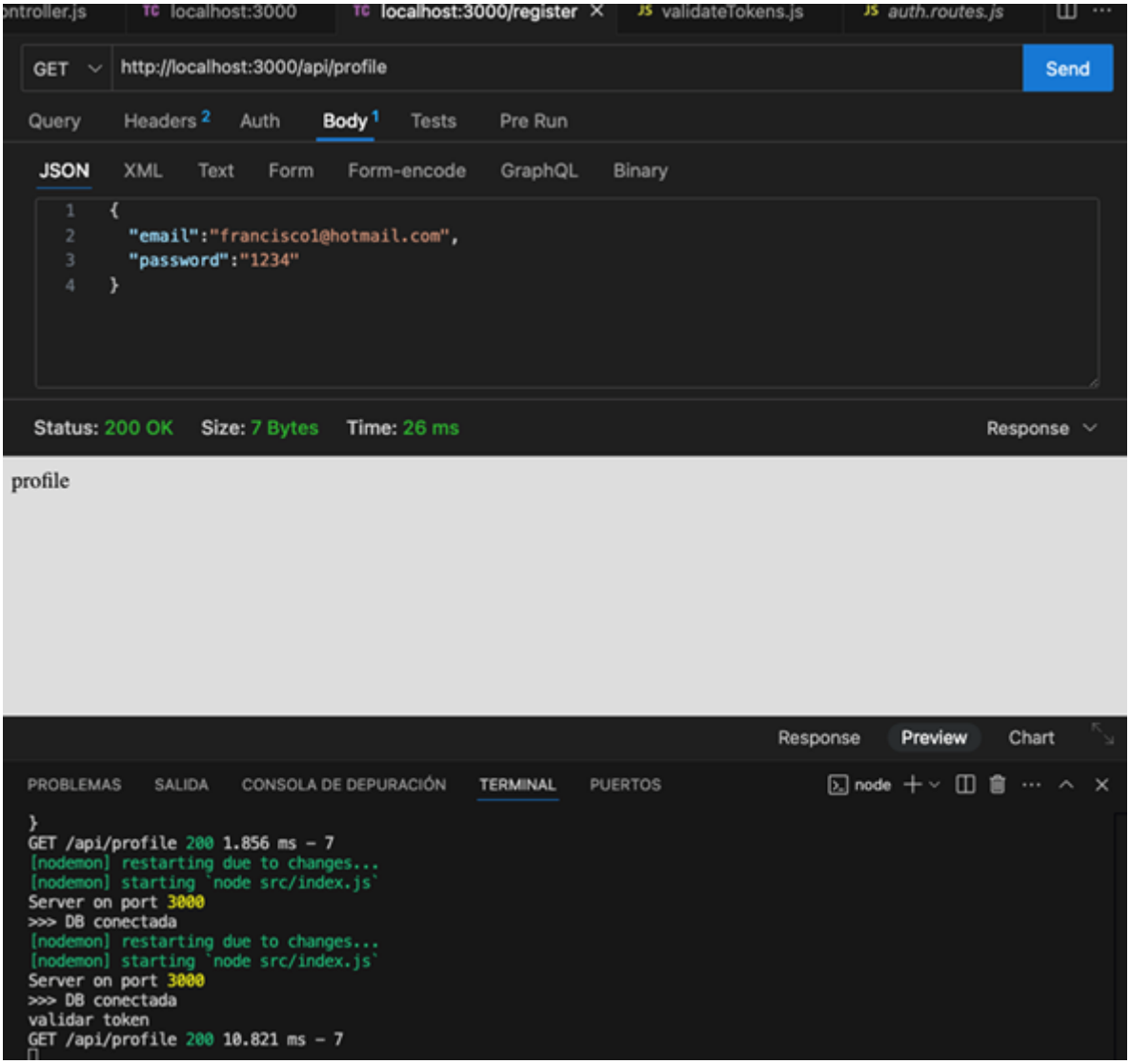

El objetivo de esto es en lugar de mostrar profile es traer los datos del usuario “logueado” Por lo tanto necesitamos una función que verifique que el usuario tenga ese estatus por lo pronto iremos a la carpeta `middlewares` y crearemos un archivo llamado `validateTokens.js` Con el siguiente contenido :

```
export const authRequired = (req,res,next) =>{
    console.log('validar token');
    next()
;}
```

Los middlewares son funciones que se ejecutan antes de cargar una ruta. Se ponen antes de la llamada a la ruta como en este ejemplo:

```
router.get('/profile',authRequired,profile);
```

Esto ejecuta la función “authRequired” antes de hacer la petición a profile. Si ahora hacemos un get de profile en consola debe contestarnos “Validar token” como vemos hasta debajo de la siguiente captura.




Ya lista la función ahora vamos a sacar los datos del token. Lo primero que podemos verificar es si en el header viene el token y lo podemos comprobar haciendo la petición y el lugar de mandar a consola “validar token” pondremos `req.headers` esto hará que por consola veamos todo lo que viene.


Si se fijan primero sale el json y una vez que pasa por ahí continua al get de profile, lo que significa que si está entrando a authRequired. Lo siguiente que haremos será cachear el token en una variable modificamos así el authRequired:

```
export const authRequired = (req,res,next) =>{
    const token = req.headers.cookie
    console.log(token);
    next();
;}
```

Y esto ya nos regresa el token. El problema de hacer esto es que la extracción de datos es un poco complicado porque saldrá la cadena completa pero para evitar esto traeremos directo la cookie.

```
export const authRequired = (req, res, next) => {
    const cookie = req.cookies;
    console.log(cookie);
    next();
};
```

Si hacemos un get a profile obtendremos un undefined por que node no puede leerlas, por lo tanto instalaremos un traductor:

```
npm i cookie-parser
```

(pueden hacerlo en otra terminal o bajar el server temporalmente para instalarlo) Después para eso iremos a `app.js` donde importamos cookie parser y lo usamos.

```
import express from 'express'
import morgan from 'morgan'
import cookieParser from 'cookie-parser'
import authRoutes from "./routes/auth.routes.js"

const app = express();

app.use(morgan('dev'));
app.use(express.json());
```

```
app.use(cookieParser());
app.use("/api",authRoutes);
export default app;
```

esto nos reportara el token con segmentaciones de puntos. { token: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4MGVlMmFmMTQyODQ2OGU5ZWQ0MDA1YiIsImlhdCI6MTc0NTgxMDAxNSwiZXhwIjoxNzQ1ODk2NDE1fQ.d0gTjLtfcqmbwoYwyrtBbNDeaukaJ7JyXavq2AgMt_Q' }

Aun si necesitamos validar que si exista el token en el contexto o en otras palabras logeado.

Volvemos a `validateTokens.js` primero debemos verificar si hay token de esta manera.

```
export const authRequired = (req,res,next) =>{
    const {token} = req.cookies;

    if(!token)
        return res.status(401).json({message: "Sin token, Autorizacion denegada"})

    next();
;}
```

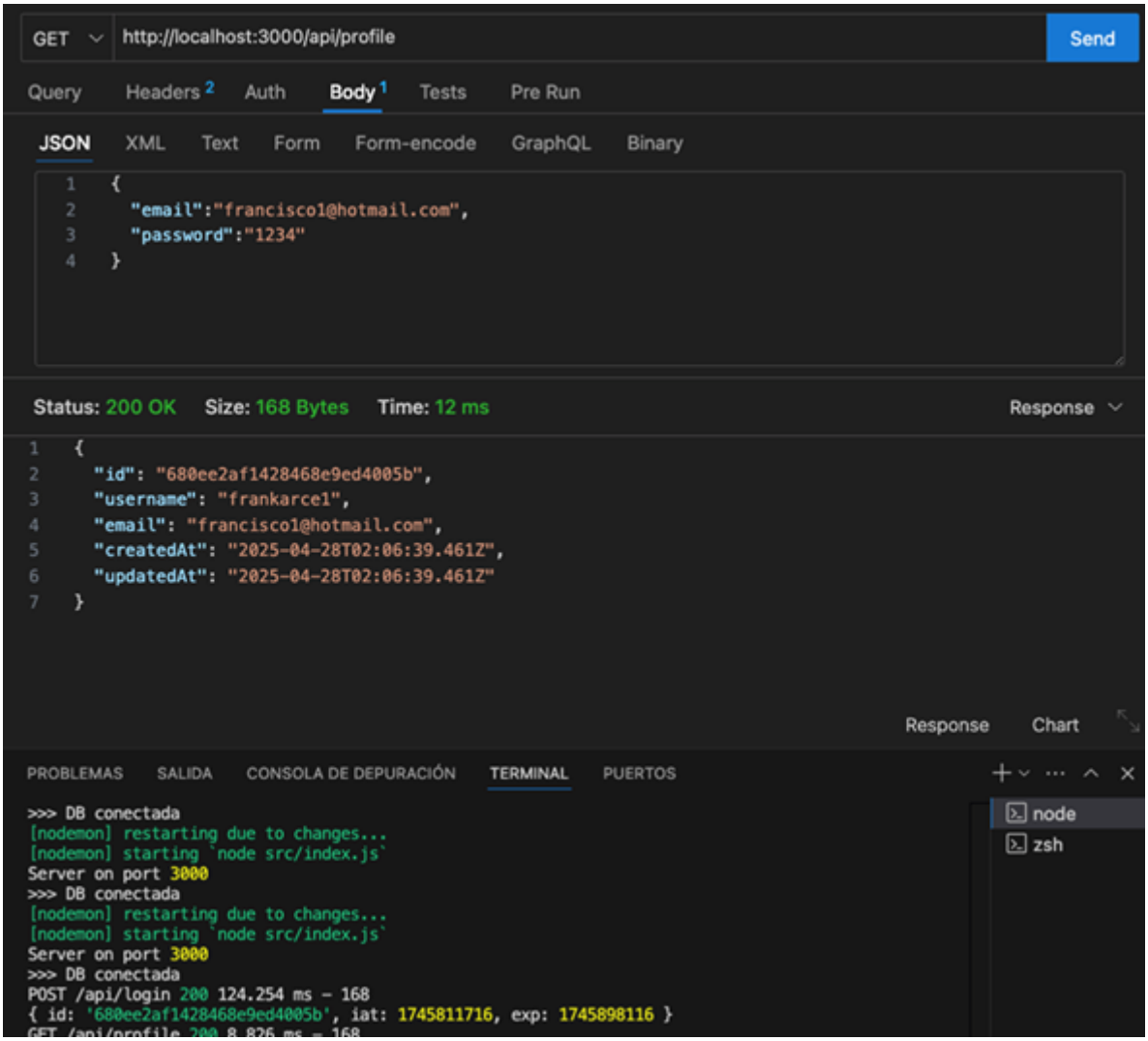
Después en caso de existir un token debemos verificar que sea generado por nosotros. Para eso importamos `jwt` y usamos la propiedad de `verify` y nos traempos el `TOKEN_SECRET` que creamos en `config.js`

```
import jwt from 'jsonwebtoken'
import { TOKEN_SECRET } from '../config.js';
export const authRequired = (req,res,next) =>{
    const {token} = req.cookies;

    if(!token)
        return res.status(401).json({message: "Sin token, Autorizacion
denegada"});
    jwt.verify(token, TOKEN_SECRET, (err, user) =>{
        if (err) return res.status(403).json({message:"Token Invalido"});
        console.log(user);
        req.user=user;

        next();
    })


;}
```

De aquí nos mudamos a `auth.controller` para los pasos finales de la verificación. En la función de profile ya no ocupamos el `console.log` ni el `res.send` lo primero que haremos es hacer una consulta para ver si existe el usuario que hizo la petición, si no lo encuentra mandamos el error 400 y le decimos “usuario no encontrado”, en caso contrario le regresamos los datos.

```
export const profile = async (req,res) => {
  const userFound = await User.findById(req.user.id)
  if (!userFound ) return res.status(400).json({ message : "Usuario no
encontrado"});

  return res.json({
    id: userFound._id,
    username : userFound.username,
    email : userFound.email,
    createdAt: userFound.createdAt,
    updatedAt: userFound.updatedAt
  })
  //console.log(req.user);
  //res.send("profile");
}
```

Con esto ya deberíamos tener una respuesta de profile de la siguiente manera.




---

TAREA CRUD.

---

Después de terminar con la autenticación lo siguiente es trabajar con las tareas del crud. Por lo tanto ahora vamos a crear un archivo de rutas para las tareas lo llamaremos tasks.routes.js.

```
import { Router } from "express";
import { authRequired } from "../middlewares/validateTokens.js";
import { getTasks, getTask, createTask, deleteTask, updateTask } from
"../controllers/task.controller.js";

const router = Router()

router.get('/tasks',authRequired,getTasks)
router.get('/tasks/:id',authRequired,getTask)
router.post('/tasks',authRequired,createTask)
router.delete('/tasks/:id',authRequired,deleteTask)
router.put('/tasks/:id',authRequired,updateTask)

export default router
```

tiene el mismo formato que auth.routes lo que cambiara serán las rutas solicitadas. Después de configurarlo vamos a app.js para registrarlo.

```
import express from 'express';
import morgan from 'morgan';
import cookieParser from 'cookie-parser';
import authRoutes from './routes/auth.routes.js';
import taskroutes from './routes/task.routes.js';
const app = express();
app.use(morgan('dev'));
app.use(express.json());
app.use(cookieParser());
app.use("/api",authRoutes);
app.use("/api",taskroutes);

export default app;
```

una vez hecho el registro de las rutas de task necesitamos los controles para las tasks nos vamos a la carpeta controllers y crearemos el archivo task controller para poner cada una de las tareas que ya registramos en rutas y quedaría asi.

```
import Task from "../models/task.model.js";

export const getTasks = async (req, res) =>{
    const tasks = await Task.find();
    res.json(tasks);
}


export const createTask = async (req, res) => {
```

```
    const { title, description, date } = req.body;
    const newTask = new Task({ title, description, date });
    const savedTask =  await newTask.save();
    res.json(savedTask);
}

export const getTask = async (req, res) => {
   const task = await Task.findById(req.params.id);
   if (!task) return res.status(404).json({ message: 'Tarea no encontrada' });
   res.json(task);
}

export const updateTask = async (req, res) => {
    const task = await Task.findByIdAndUpdate(req.params.id, req.body,{ new:
true});
    if (!task) return res.status(404).json({ message: 'Tarea no encontrada' });
    res.json({ message: 'Tarea actualizada correctamente' });
}

export const deleteTask = async (req, res) => {
   const task = await Task.findByIdAndDelete(req.params.id);
   if (!task) return res.status(404).json({ message: 'Tarea no encontrada' });
   res.json({ message: 'Tarea eliminada correctamente' });
}
```

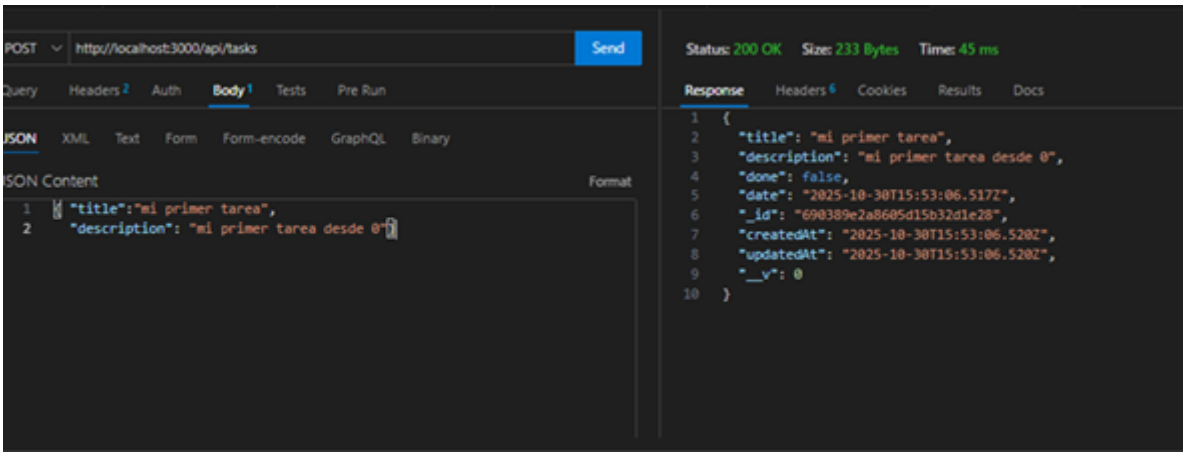
Como tal tenemos que interactuar con mongodb para poder realizar los movimientos de este crud y por lo tanto necesitarmos un modelo especifico para este modulo, nos iremos a la carpeta `models` y crearemos el archivo `task.model.js` y lo llenarmos con el siguiente esquema.

```
import mongoose  from "mongoose";
const taskSchema = new mongoose.Schema({

    title: {type: String, required: true},
    description: {type: String, required: true},
    date: {type: Date, default: Date.now},},
    {timestamps: true}
)

export default mongoose.model('Task', taskSchema);
```

para poder ver si funciona debemos ir al thunderclient tener un token valido disponible (un usuario logueado) y mandarle una petición a post a task http://localhost:3000/api/task con este body

```
{ "title":"mi primer tarea",
  "description": "mi primer tarea desde 0"}
```

Y nos responderá:




Aquí falta un detalle, la tarea se guardo pero en la base de datos no se registra que usuario la creo por lo tanto hay que agregar ese campo en `task.models` y quedaría de esta forma el `user` justo debajo de date:

```
import mongoose  from "mongoose";
const taskSchema = new mongoose.Schema({

    title: {type: String, required: true},
    description: {type: String, required: true},
    date: {type: Date, default: Date.now},
    user: {type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true}
}, {timestamps: true}
)

export default mongoose.model('Task', taskSchema);
```

solo que lleva argumentos adicionales `mongoose.Schema.Types.ObjectId` esto saca el user id del usario y el ref hace una referencia al modelo user.

Si intentamos hacer de nuevo una petición a tasks la que ya nos había respondido con los datos de creación de la tarea ahora nos mandara un error por que falta agregar el path del usuario saldría esto:

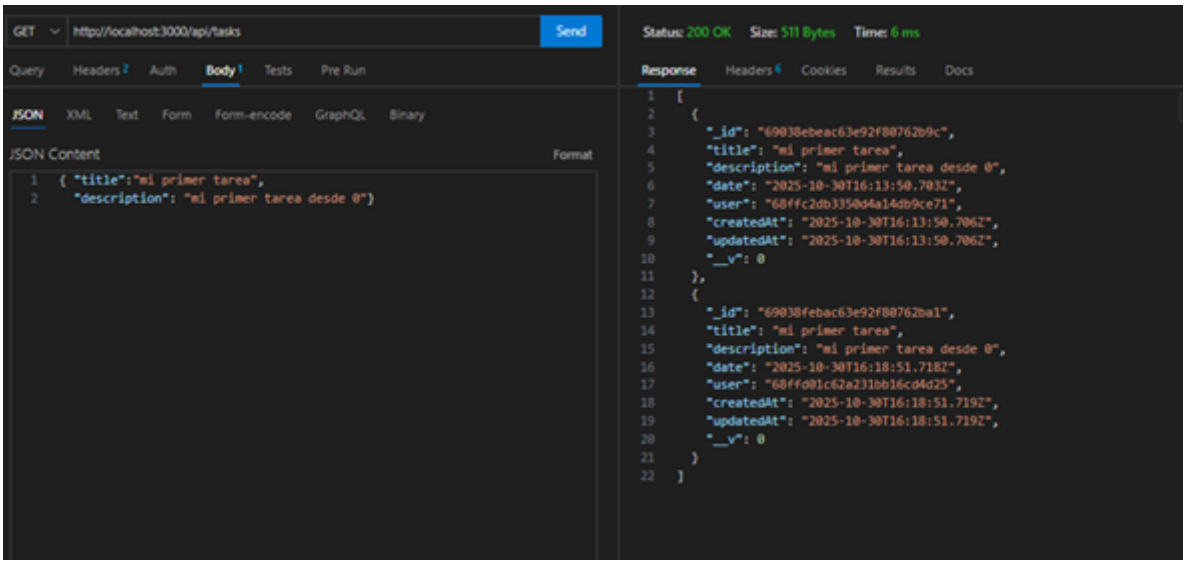
```
ValidationError: Task validation failed: user: Path `user` is required.
    at Document.invalidate
(C:\Users\franc\express\express\node_modules\mongoose\lib\document.js:3362:32)
    at
C:\Users\franc\express\express\node_modules\mongoose\lib\document.js:3123:17
    at
C:\Users\franc\express\express\node_modules\mongoose\lib\schemaType.js:1417:9
    at process.processTicksAndRejections (node:internal/process/task_queues:85:11)
```

para solucionar este error nos vamos al `task.controller` especificmente al control createTask

```
export const createTask = async (req, res) => {
    console.log(req.user);
    const { title, description, date } = req.body;
```

```
    const newTask = new Task({ title, description, date, user: req.user.id });
    const savedTask =  await newTask.save();
    res.json(savedTask);
}
```

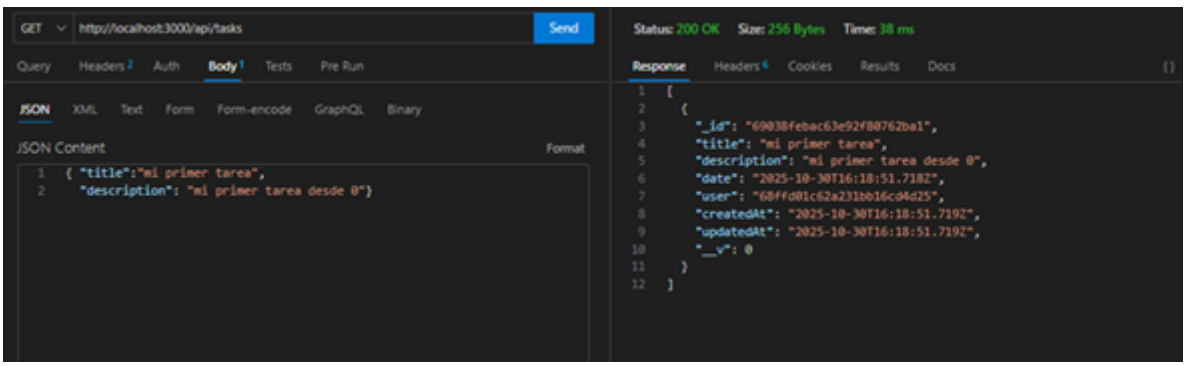
Y crea una tarea con 2 usuarios para iniciar un problema de permisos. Lo siguiente es manejar las tareas que ya existen para eso solo cambiamos el método de la petición a get siempre y cuando tengamos un `login` activo, si hacemos la petición tal como la tenemos nos mostrara todas las tareas que hay y eso no esta bien por que se supone que el usuario solo debe de ver sus tareas no las de los demás. Lo que se muestra a continuación es el get que me trae las tareas de 2 usuarios.




Para arreglar este problema volvemos a el control de getTasks y vamos a modificar la consulta

```
export const getTasks = async (req, res) =>{
    const tasks = await Task.find(
        { user: req.user.id }
    );
    res.json(tasks);
}
```

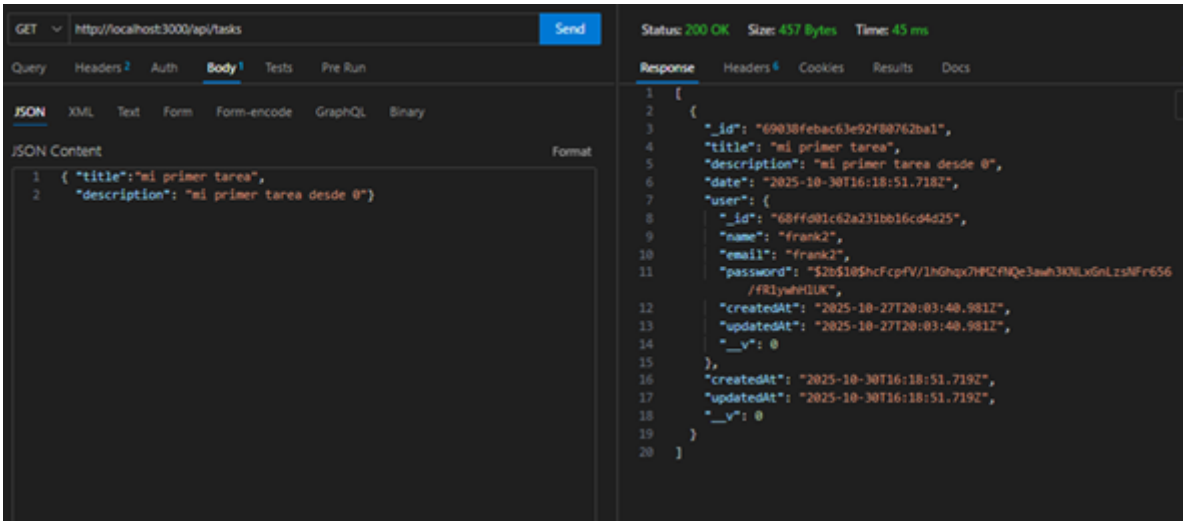
Ya con esto nos debe regresar solo la tarea de ese usuario pueden mandar al console log el user.id para corroborar que si es ese.



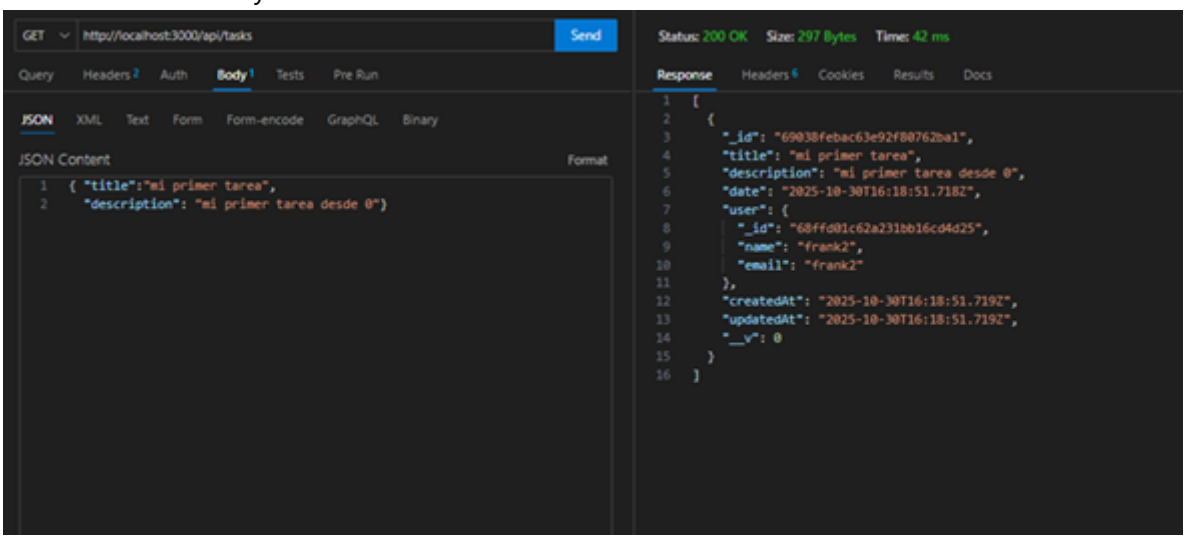
Pero si adicionalmente queremos datos del usuario podemos agregarle a la consulta la propiedad populate para que agregue todo el diccionario de los datos del usuario.

```
export const getTasks = async (req, res) =>{
    const tasks = await Task.find(
        { user: req.user.id }
    ).populate('user','name email');
    res.json(tasks);
}
```

Si dejan solo user les treaera todo hasta el password pero si solo quieren campos es especifico se pueden poner después de una coma. Este es con todo



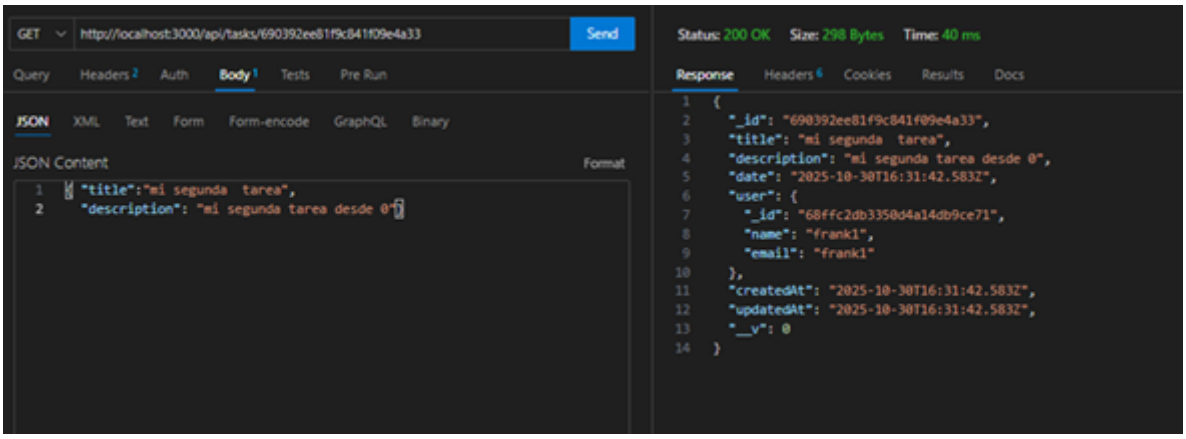

Este es con el name y el email




Ahora nos pasamos a pedir uan tarea especifica y en este caso iremos directo a thunderclient

Y haremos un get de una tarea especifica con su id para estos hacemos un get tasks para ver cuales hay y copiar su id

Y de ahí en el get le vamos a agregar ese id como parámetro solo separándolo de la url original con una /:




Ahora si cambiamos el método a delete necesita un id para eliminar como en el task de una tarea y en para actualizar también tenemos que mandarle el id pero en body los campos con los datos a actualizar y listo hasta aquí ya tenemos el crud como api, solo nos falta crear el frontend para deslindarnos de thunderclient.

Solo nos falta unas validaciones para los datos de entrada.

---

Siguiente sección Validar los datos de entrada

---

Para poder realizar estas validaciónes necesitamos crear esquemas de validación lo primer que hay que hacer si aun no lo hemos hecho es crear la carpeta `schemas` en `src` y reutilizaremos librerías de validación de datos. Existen muchas pero en este caso usaremos zod para instalarla usamos el

```
npm i zod
```

y nos teleportamos a la carpeta schemas a crear el archivo que validara los datos de autenticación, le vamos a poner `auth.schema.js`

Dentro vamos a importar zod como {z} por lo pronto solo agregaremos esquema para register y login que son los que ocupan ingresar datos.

```
import {z} from 'zod';

export const registerSchema = z.object({
    name: z.string({required_error: 'El nombre es obligatorio'}),
    email: z.string({required_error: 'El email es obligatorio'}).email('El email
no es válido'),
    password: z.string({required_error: 'La contraseña es obligatoria'}).min(6,
'La contraseña debe tener al menos 6 caracteres')
});

export const loginSchema = z.object({
    email: z.string({required_error: 'El email es obligatorio'}).email('El email
no es válido'),
```

```
    password: z.string({required_error: 'La contraseña es obligatoria'})
});
```

Ahora recordemos que estos archivos necesitan ser insertados en algún lugar ara establecerse como compuertas. Por lo tanto debemos crear un middleware como el que creamos antes donde se generaba el token de acceso.

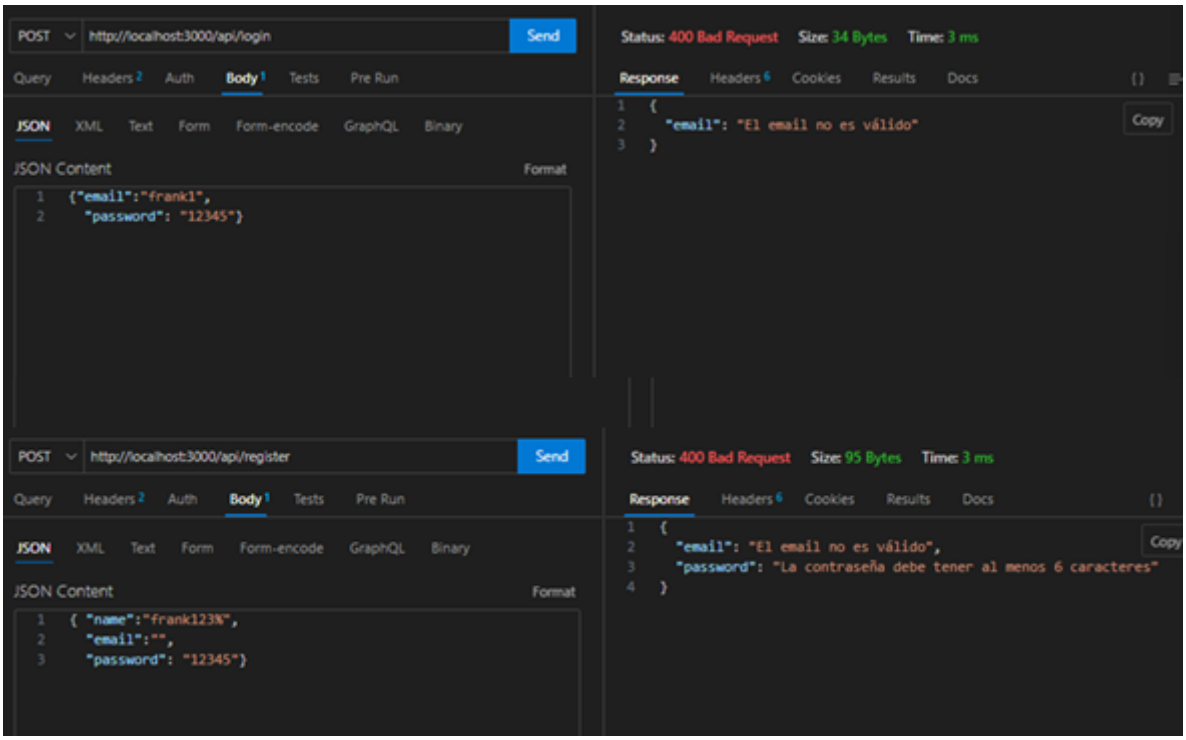
```
export const validateSchema = (schema) => (req, res, next) => {

    try {
            schema.parse(req.body);
            next();
    } catch (error) {
        // Creamos un objeto para mapear los errores
        const errorMessages = {};

        error.issues.forEach(issue => {
            // issue.path[0] será 'password' o 'email', etc.
            const field = issue.path[0];
            const message = issue.message;

            errorMessages[field] = message;
        });

        return res.status(400).json(errorMessages);
    }
}
```

Y por ultimo para que pueda estar como compuerta debemos ir a `auth.routes` para anteponerlo como lo hicimos con el `authRequired`

```
import { Router } from "express";

import { login, register, logout, profile } from
"../controllers/auth.controller.js";
import {authRequired} from '../middlewares/validateTokens.js';
import { validateSchema } from "../middlewares/validator.middleware.js";
import { loginSchema, registerSchema } from "../schemas/auth.schema.js";

const router = Router();
router.post('/login', validateSchema(loginSchema), login);
router.post('/register', validateSchema(registerSchema), register);
router.post('/logout', logout);
router.get('/profile', authRequired, profile);
export default router;
```

y ahora si podemos hacer peticiones a login o register cumpliendo o no con las reglas que establecimos para obtener errores. Pero nos regresa un diccionario con varias cosas y esto se debe pulir pero ya esta corregido

en el .




Para mostrar solo lo que queremos mostrar en el fron end

Tambien crearemos un esquema de validación para las tareas le pondremos task.schema.js

```
import {z} from 'zod';

export const createTaskSchema = z.object({
    title : z.string({ required_error: 'El título es obligatorio' }),
    description : z.string({required_error: 'La descripción es
obligatoria'}).optional(),
    date : z.string({ required_error: 'La fecha es obligatoria'
}).datetime().optional()
});
```