# JWT PROJECT

**2023-2024**

SUBMITTED TO | **Mehdi Tmimi**

**realised by:**
Mohammed Elias Lakhmiri

## SUMMARY

# INTRODUCTION

**This report aims to provide an overview of Representational State Transfer (REST) architectural style and JSON Web Token (JWT) technology. REST is a software architectural style that defines a set of constraints to be used when creating web services. JWT, on the other hand, is a compact, URL-safe means of representing claims to be transferred between two parties.**

**The report will begin by discussing the basics of REST. We will then move on to cover JWT, including its structure, how it is used for authentication, and its advantages over traditional methods of authentication.**

**The report will also cover the implementation details of using both REST and JWT in a server-client architecture. It will explain the usage of MongoDB as the database, and the usage of Mongoose library to perform CRUD operations, and the separation of routes for user and memo models.**

**Finally, the report will conclude by discussing the benefits of using REST and JWT in a web application and the importance of security in modern web development.**

**Please let me know if you need any further help with your report.**

REST APIs are a way to expose the functionality of a web service through a simple, consistent interface. They use the principles of the REST architectural style to provide a simple, easy-to-use interface for interacting with the service.

REST APIs are based on the HTTP protocol, which is the same protocol used by the World Wide Web. This makes them accessible to a wide variety of clients, including web browsers, mobile devices, and other servers. They can return data in different formats like JSON, XML, etc.

In summary, REST API is an architectural style that defines a set of constraints to be used when creating web services, it uses the HTTP protocol and it is based on the principles of REST architectural style.

## SERVER SIDE

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. JWT is a JSON object that is encoded as a string and can be digitally signed using JSON Web Signature (JWS) or encrypted using JSON Web Encryption (JWE).
The structure of a JWT typically includes three parts, separated by dots:

- The header, which contains information about how the JWT is encoded
- The payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata.
- The signature, which is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

Additionally, JWT's can be easily passed around different domains and systems, making them suitable for microservices architectures and Single Sign-On (SSO) solutions

## MONGODB SERVER SIDE

**In my project, I am using MongoDB as the database to store and retrieve data for the application. The Mongoose library is used to define the schema for the data and perform CRUD (Create, Read, Update, and Delete) operations on the data using Mongoose models.**

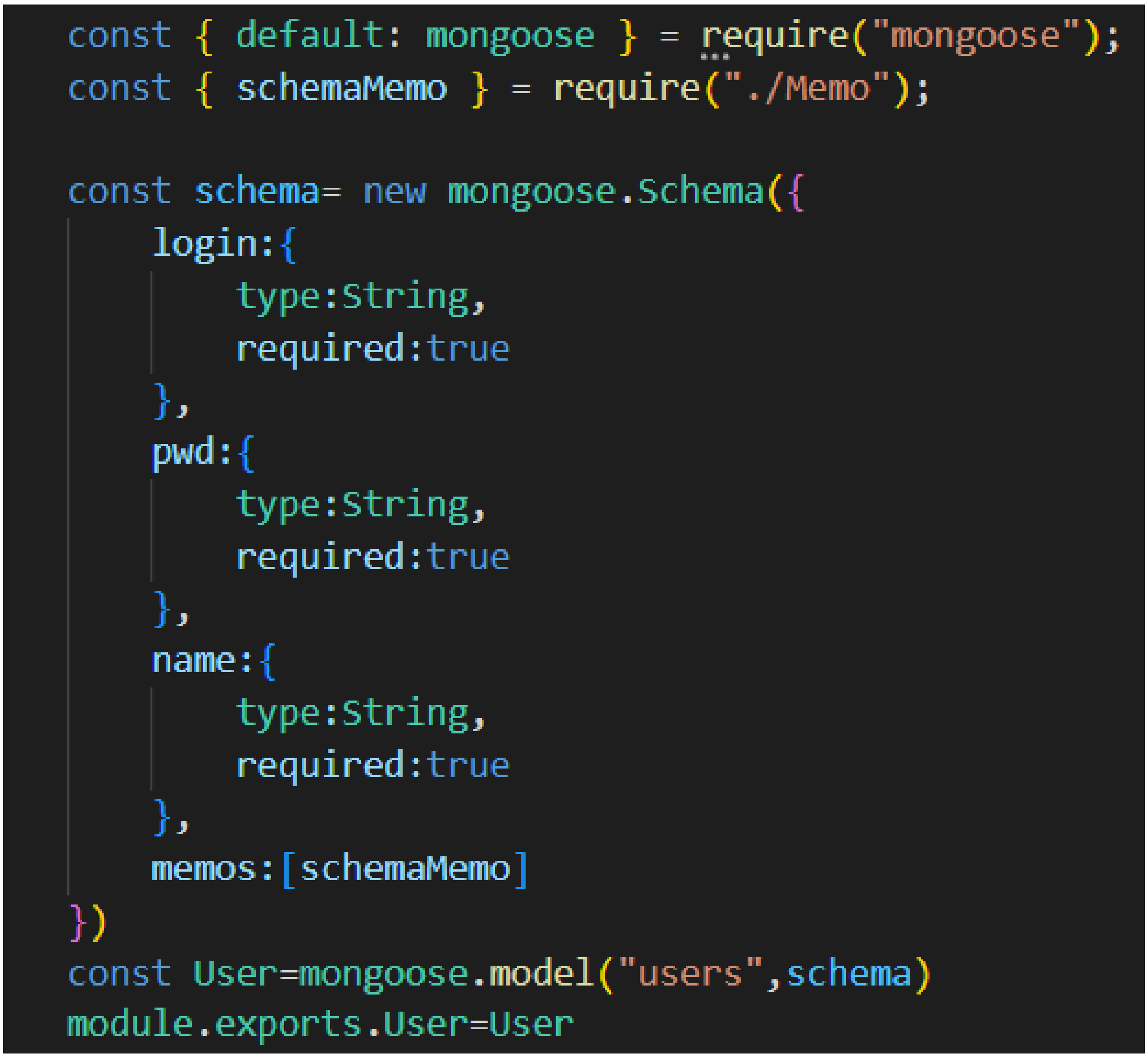
**I have defined two Mongoose schemas, one for users and another for memos. The user schema includes fields for login, password, name, and memos. The memo schema includes fields for date and content.**

**The user schema also includes an array of memos, this allows me to store multiple memos for each user, and I can easily retrieve the memos of a user by querying the memos field.**

**By using Mongoose schema, I can define validation rules, default values, and custom methods and then use mongoose functions like findOne, find, save, update, delete to perform CRUD operation on the data.**

**Overall, MongoDB is playing a crucial role in my project by providing an efficient way to store, retrieve and manage data and it's also providing the ability to scale the application to handle more traffic.**

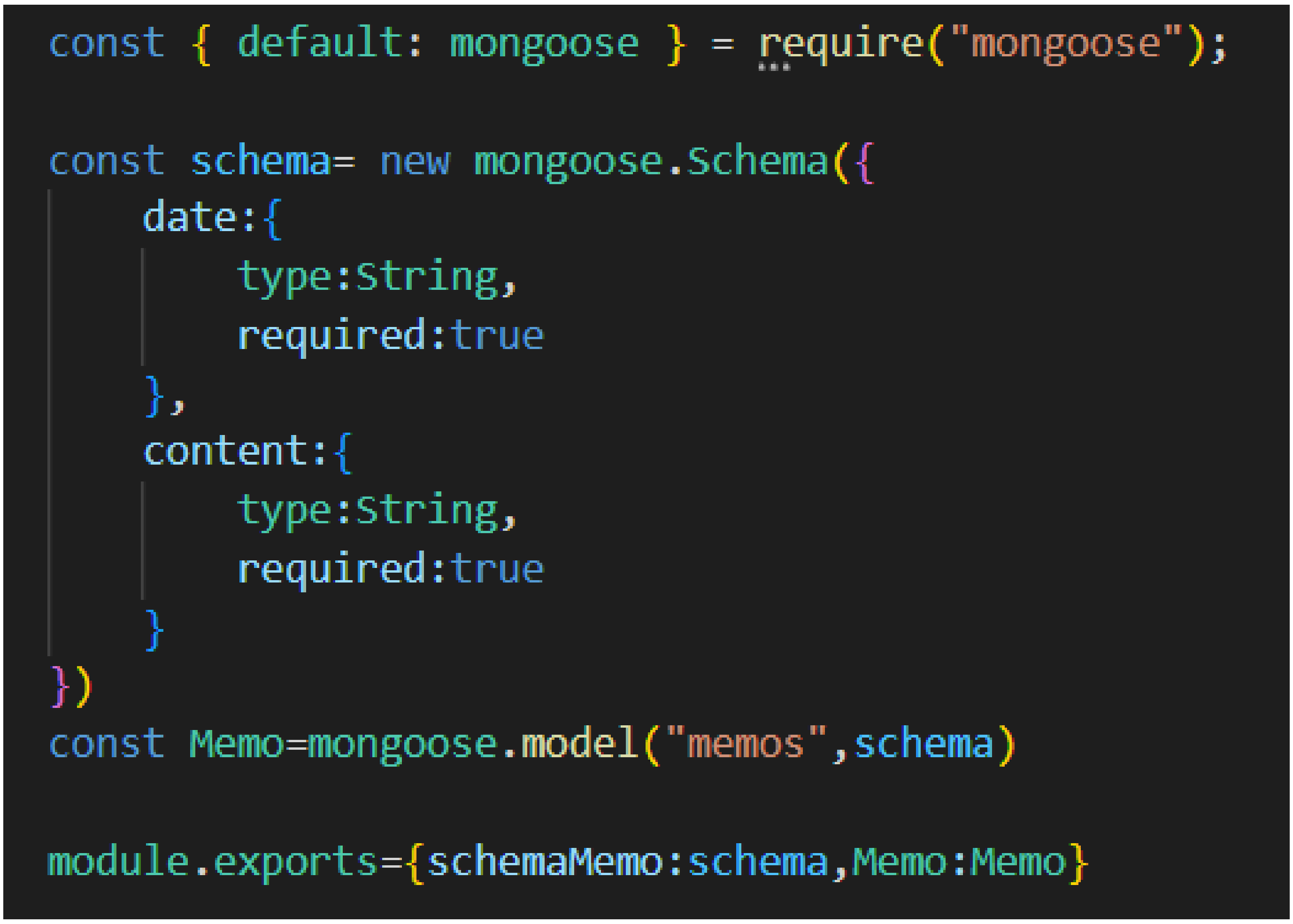
# USER MODEL SERVER SIDE



```
const { default: mongoose } = require("mongoose");
const { schemaMemo } = require("./Memo");

const schema= new mongoose.Schema({
    login:{
        type:String,
        required:true
    },
    pwd:{
        type:String,
        required:true
    },
    name:{
        type:String,
        required:true
    },
    memos:[schemaMemo]
})
const User=mongoose.model("users",schema)
module.exports.User=User
```

**First we creates a Mongoose schema for a user, which defines the structure of the data that will be stored in the MongoDB collection for users and creates a Mongoose model for this schema so it can be used to perform CRUD operations on the data.**

# MEMO MODEL SERVER SIDE



```
const { default: mongoose } = require("mongoose");

const schema= new mongoose.Schema({
    date:{
        type:String,
        required:true
    },
    content:{
        type:String,
        required:true
    }
})
const Memo=mongoose.model("memos",schema)

module.exports={schemaMemo:schema,Memo:Memo}
```

**Then we creates a Mongoose schema for a memo, which defines the structure of the data that will be stored in the MongoDB collection for memos, creates a Mongoose model for this schema so it can be used to perform CRUD operations on the data and exports the schema and the model so they can be used in other parts of the application.**

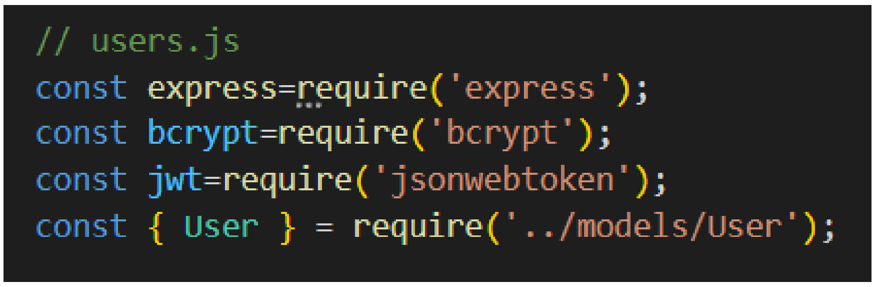
## ROUTES

In my project, I have two main routes files, "users.js" and "memos.js", each one is responsible for handling different endpoints of the application.
The "users.js" file, I define the routes and the logic for handling user-related requests such as creating a new user, logging in, and retrieving user information. I use the logic and Mongoose models that I have defined earlier to perform the CRUD operations on the data and handle the requests. I also use JWT for authentication and authorization on these routes.
The "memos.js" file, I define the routes and the logic for handling memo-related requests such as creating new memo, retrieving, updating and deleting a memo. I use the logic and Mongoose models that I have defined earlier to perform the CRUD operations on the data and handle the requests. I also use JWT for authentication and authorization on these routes.

# USERS.JS ROUTES

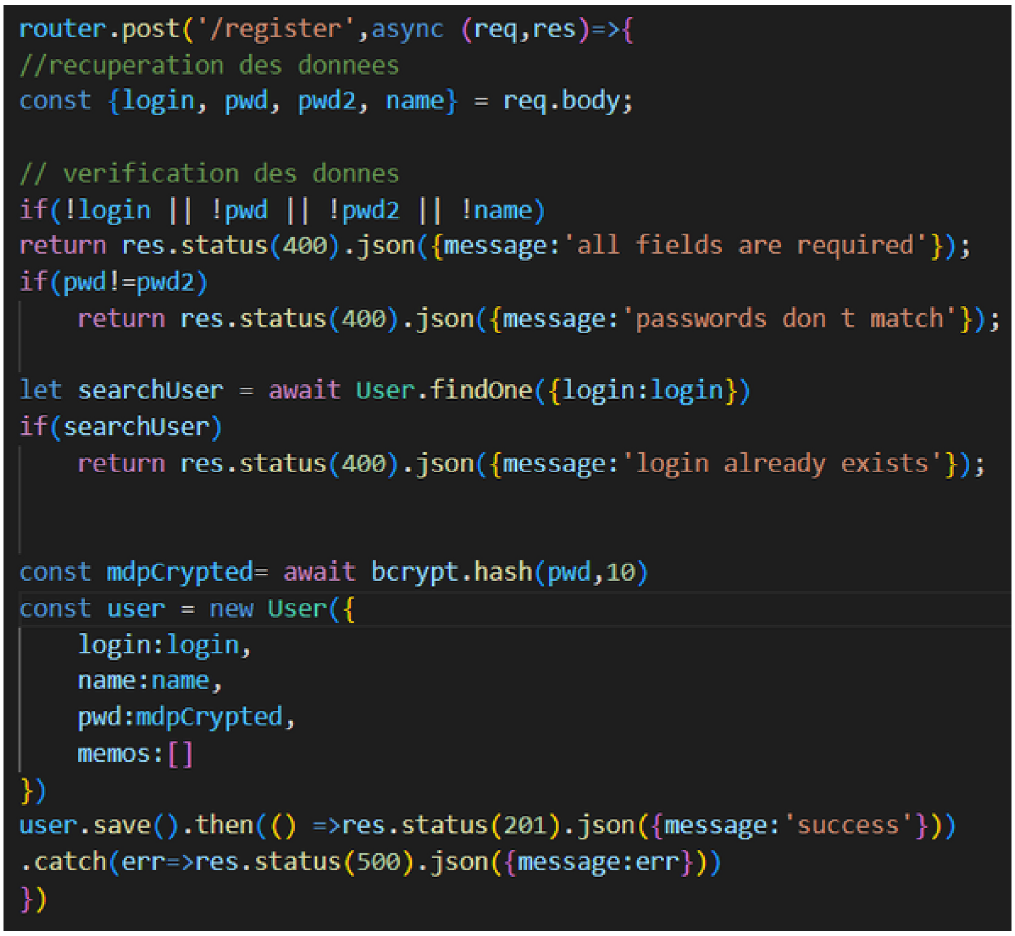
The users.js file is a part of the server-side code that handles user-related routes and logic. It uses the Express.js framework to define routes for creating new users, logging in, and handling user-related errors

First we should imports the necessary modules, such as the express module, bcrypt and jsonwebtoken, to handle routing, password encryption and token creation respectively. Also, it imports the User model which is defined in the User.js file.



```
// users.js
const express=require('express');
const bcrypt=require('bcrypt');
const jwt=require('jsonwebtoken');
const { User } = require('../models/User');
```

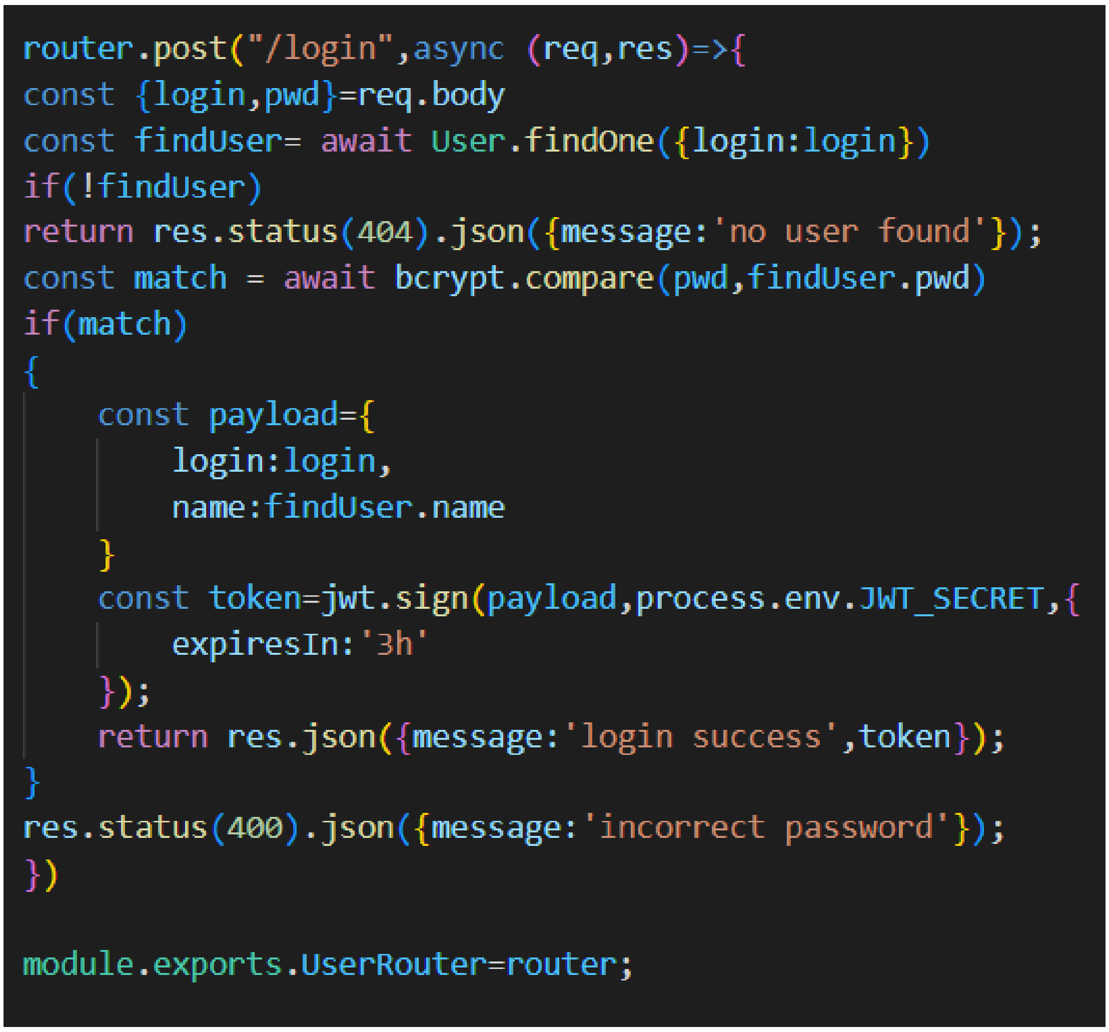
Then I creates an instance of the express Router, which allows you to define routes. The first route is "/register" is used to handle user registration. It takes the data sent in the request body and checks if all the fields are filled, if the passwords match, and if the login already exists. If everything is fine, it encrypts the user's password, creates a new user object, and saves it in the database.



```
router.post('/register',async (req,res)=>{
//recuperation des donnees
const {login, pwd, pwd2, name} = req.body;

// verification des donnes
if(!login || !pwd || !pwd2 || !name)
return res.status(400).json({message:'all fields are required'});
if(pwd!=pwd2)
    return res.status(400).json({message:'passwords don t match'});

let searchUser = await User.findOne({login:login})
if(searchUser)
    return res.status(400).json({message:'login already exists'});


const mdpCrypted= await bcrypt.hash(pwd,10)
const user = new User({
    login:login,
    name:name,
    pwd:mdpCrypted,
    memos:[]
})
user.save().then(() =>res.status(201).json({message:'success'}))
.catch(err=>res.status(500).json({message:err}))
})
```

# USERS.JS ROUTES

**The second route is "/login" is used to handle the user login process. It takes the login and the password from the request body, then it checks if the user exists in the database. If the user exists, it compares the password that was sent with the one that is stored in the database. If the passwords match, it creates a JSON web token, signs it, and sends it back to the client.**



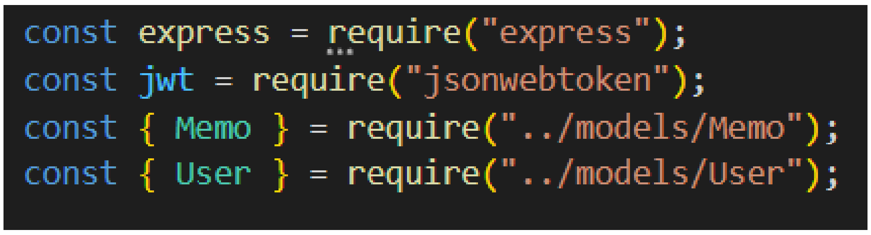
```
router.post("/login",async (req,res)=>{
const {login,pwd}=req.body
const findUser= await User.findOne({login:login})
if(!findUser)
return res.status(404).json({message:'no user found'});
const match = await bcrypt.compare(pwd,findUser.pwd)
if(match)
{
    const payload={
        login:login,
        name:findUser.name
    }
    const token=jwt.sign(payload,process.env.JWT_SECRET,{
        expiresIn:'3h'
    });
    return res.json({message:'login success',token});
}
res.status(400).json({message:'incorrect password'});
})

module.exports.UserRouter=router;
```

**Overall, this file contains the logic for handling user registration and authentication using a combination of MongoDB, bcrypt and jsonwebtoken.**

# MEMOS.JS ROUTES

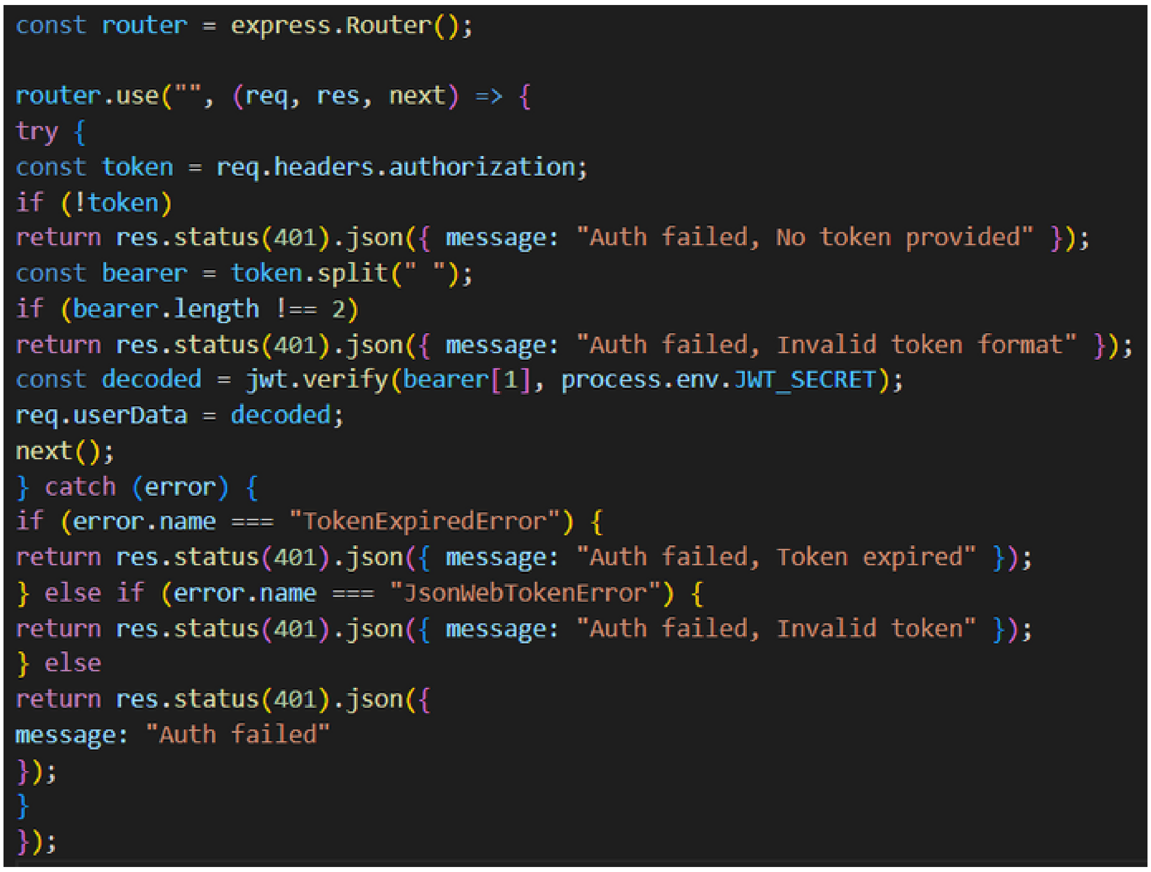
The memos.js file is a part of the server-side code that handles memo-related routes and logic. It uses the Express.js framework to define routes for creating new memos, retrieving, updating and deleting memos.

The file imports the necessary modules, such as the express module and jsonwebtoken, to handle routing and token validation respectively. Also, it imports the Memo model and the User model which are defined in the Memo.js and User.js files respectively.



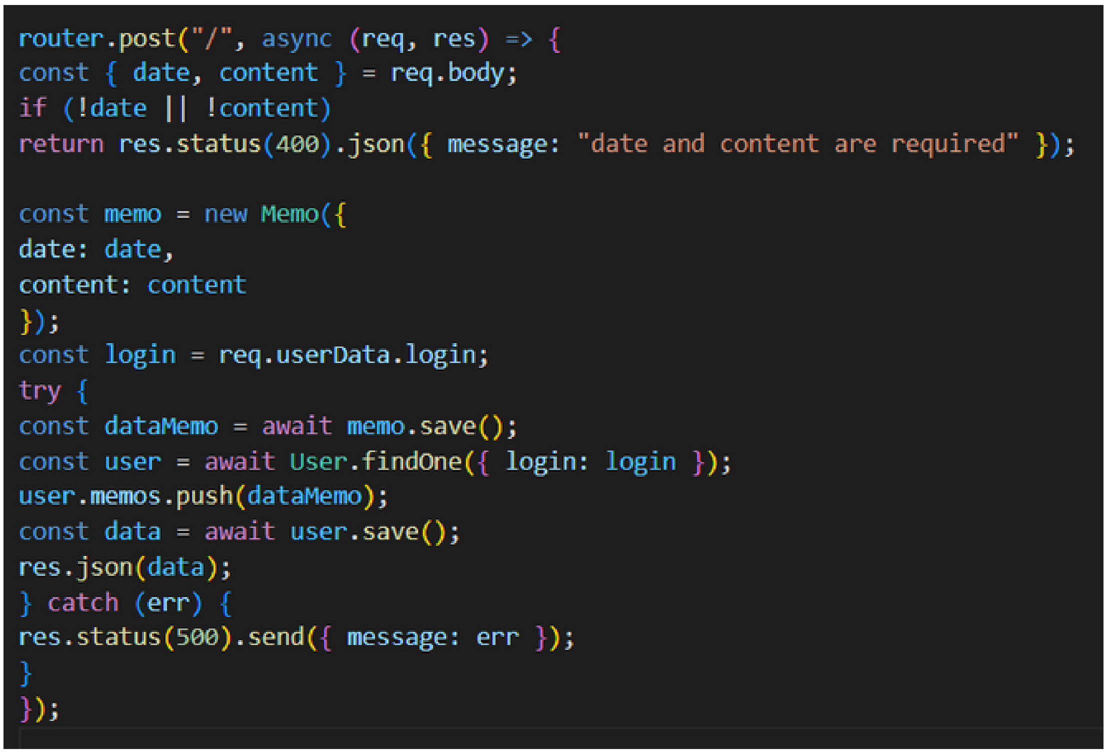
```
const express = require("express");
const jwt = require("jsonwebtoken");
const { Memo } = require("../models/Memo");
const { User } = require("../models/User");
```

It also creates an instance of the express Router, which allows you to define routes. The first route is a middleware that verifies the token sent in the request header, and it checks if the token is valid and not expired. If the token is invalid, expired or not sent, it sends an error message.



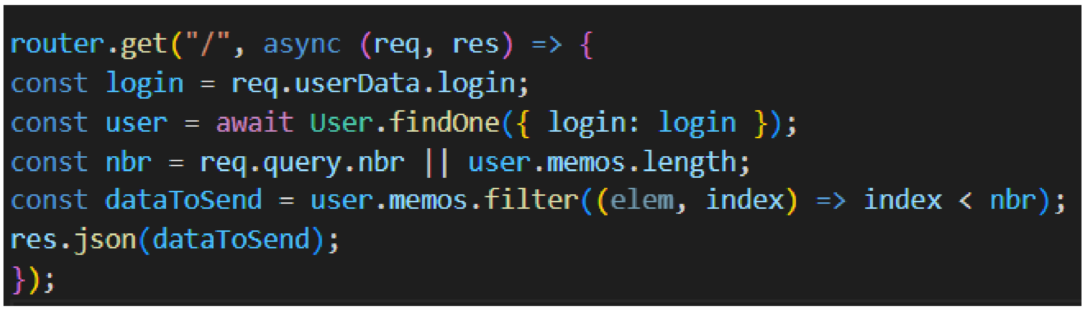
```
const router = express.Router();

router.use("", (req, res, next) => {
try {
const token = req.headers.authorization;
if (!token)
return res.status(401).json({ message: "Auth failed, No token provided" });
const bearer = token.split(" ");
if (bearer.length !== 2)
return res.status(401).json({ message: "Auth failed, Invalid token format" });
const decoded = jwt.verify(bearer[1], process.env.JWT_SECRET);
req.userData = decoded;
next();
} catch (error) {
if (error.name === "TokenExpiredError") {
return res.status(401).json({ message: "Auth failed, Token expired" });
} else if (error.name === "JsonWebTokenError") {
return res.status(401).json({ message: "Auth failed, Invalid token" });
} else
return res.status(401).json({
message: "Auth failed"
});
}
});
```

The second route is a post route that is used to create a new memo. It takes the date and content from the request body and checks if they are filled. If everything is fine, it creates a new memo object, saves it in the database and links it to the user who sent the request.



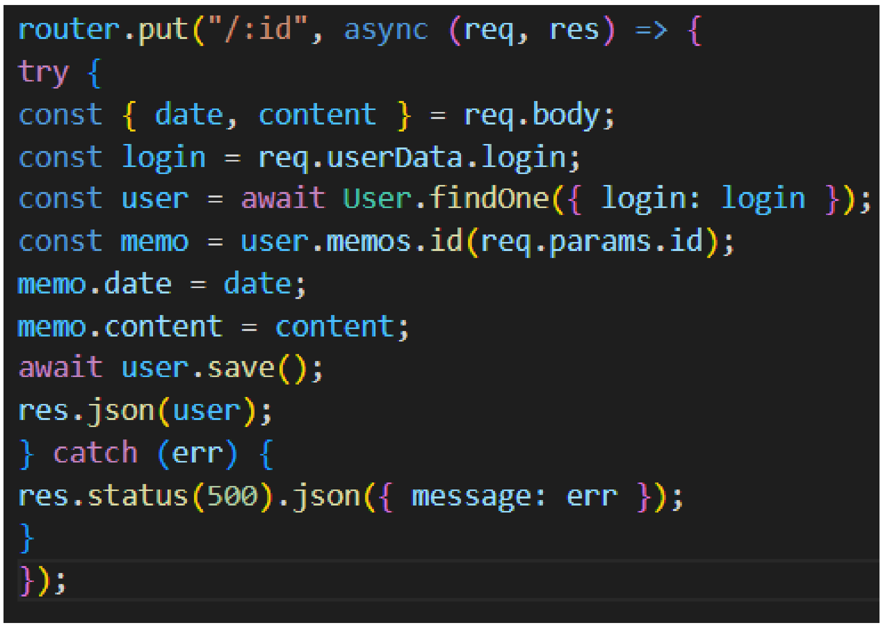
```
router.post("/", async (req, res) => {
const { date, content } = req.body;
if (!date || !content)
return res.status(400).json({ message: "date and content are required" });

const memo = new Memo({
date: date,
content: content
});
const login = req.userData.login;
try {
const dataMemo = await memo.save();
const user = await User.findOne({ login: login });
user.memos.push(dataMemo);
const data = await user.save();
res.json(data);
} catch (err) {
res.status(500).send({ message: err });
}
});
```

The third route is a get route that is used to retrieve a memo. It takes the token from the request header, verifies it and gets the user who sent the request. Then it filters the memos of the user to send the number of memos that the user wants, the default value is all the memos of the user.



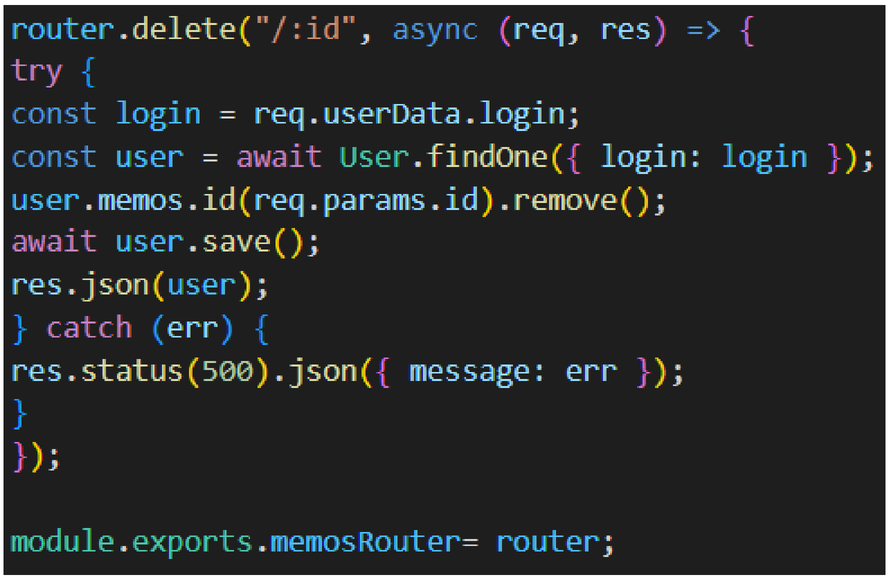
```
router.get("/", async (req, res) => {
const login = req.userData.login;
const user = await User.findOne({ login: login });
const nbr = req.query.nbr || user.memos.length;
const dataToSend = user.memos.filter((elem, index) => index < nbr);
res.json(dataToSend);
});
```

The fourth route is a put route that is used to update a memo. It takes the token from the request header, verifies it and gets the user who sent the request. Then it updates the memo with the new date and content sent in the request body.



```
router.put("/:id", async (req, res) => {
try {
const { date, content } = req.body;
const login = req.userData.login;
const user = await User.findOne({ login: login });
const memo = user.memos.id(req.params.id);
memo.date = date;
memo.content = content;
await user.save();
res.json(user);
} catch (err) {
res.status(500).json({ message: err });
}
});
```

The fifth route is a delete route that is used to delete a memo. It takes the token from the request header, verifies it and gets the user who sent the request. Then it deletes the memo.



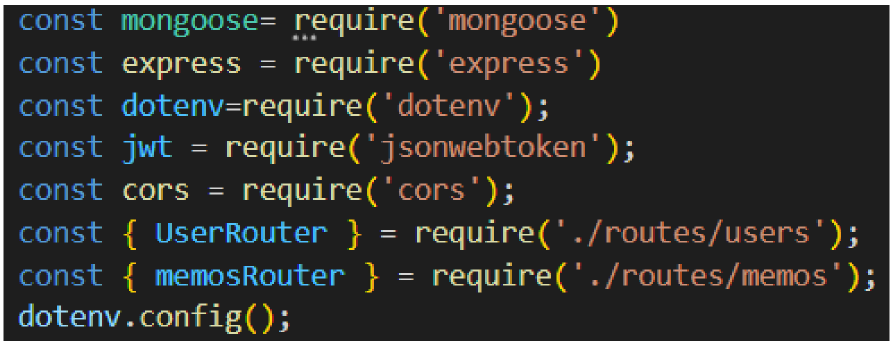
```
router.delete("/:id", async (req, res) => {
try {
const login = req.userData.login;
const user = await User.findOne({ login: login });
user.memos.id(req.params.id).remove();
await user.save();
res.json(user);
} catch (err) {
res.status(500).json({ message: err });
}
});

module.exports.memosRouter= router;
```

The last line exports the router so it can be used in the app.js file.

# APP.JS

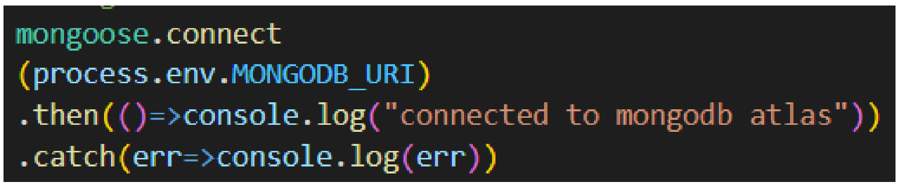
**The app.js file is the entry point of the application and it's the main logic and configurations of the application .**

**It starts by importing the necessary modules such as mongoose, express, dotenv, jsonwebtoken and the two routes files, "users.js" and "memos.js"**



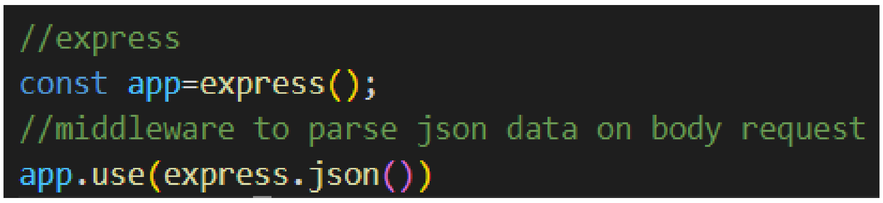
```
const mongoose= require('mongoose')
const express = require('express')
const dotenv=require('dotenv');
const jwt = require('jsonwebtoken');
const cors = require('cors');
const { UserRouter } = require('./routes/users');
const { memosRouter } = require('./routes/memos');
dotenv.config();
```

**It uses mongoose to connect to the MongoDB Atlas cluster and handle the database operations.**



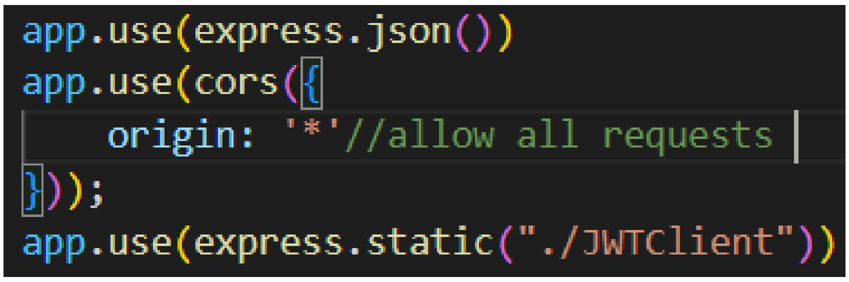
```
mongoose.connect
(process.env.MONGODB_URI)
.then(()=>console.log("connected to mongodb atlas"))
.catch(err=>console.log(err))
```

**It creates an instance of the express application, and sets up middleware to parse json data from the body of incoming requests.**



```
//express
const app=express();
//middleware to parse json data on body request
app.use(express.json())
```

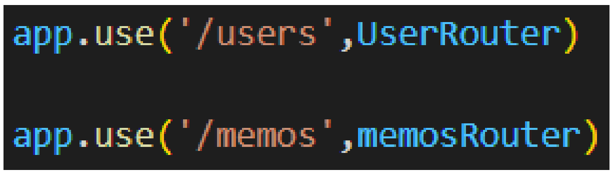
- **The first app.use call adds the CORS (Cross-Origin Resource Sharing) middleware to the server, which allows all incoming requests from any origin (origin: '*').**
- **The second app.use call adds the express.static middleware, which serves static files from the directory ./JWTClient.**



```
app.use(express.json())
app.use(cors({
    origin: '*'//allow all requests
}));
app.use(express.static("./JWTClient"))
```

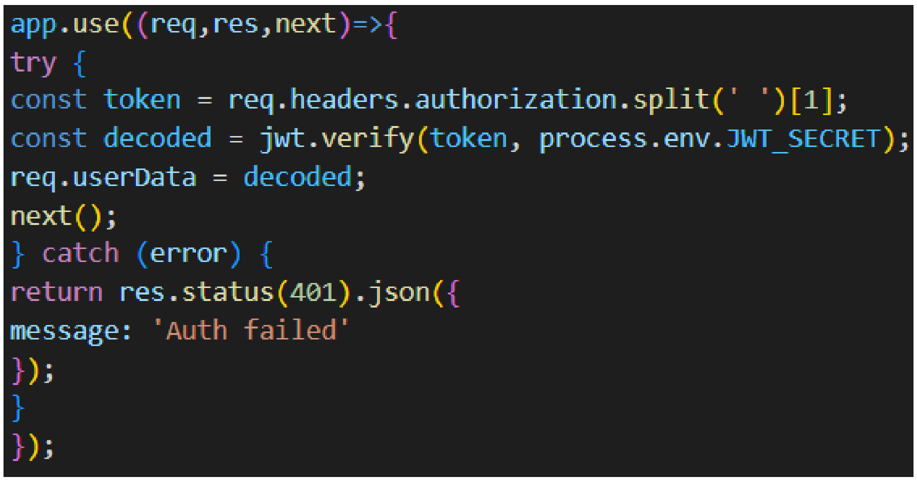
## APP.JS

It uses the two routes files, "users.js" and "memos.js" to handle the different endpoints of the application and perform CRUD operations.



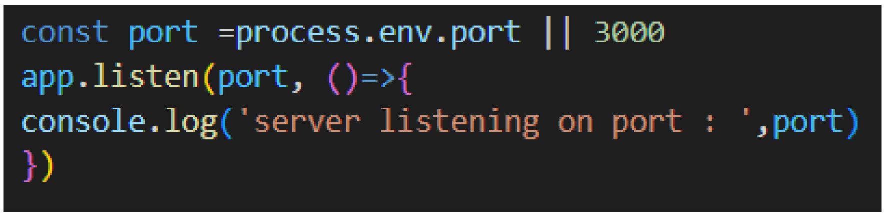
```
app.use('/users',UserRouter)

app.use('/memos',memosRouter)
```

It also has a check authentication middleware that verifies the token sent in the request header and decode it, it checks if the token is valid and not expired. If the token is invalid, expired or not sent, it sends an error message.



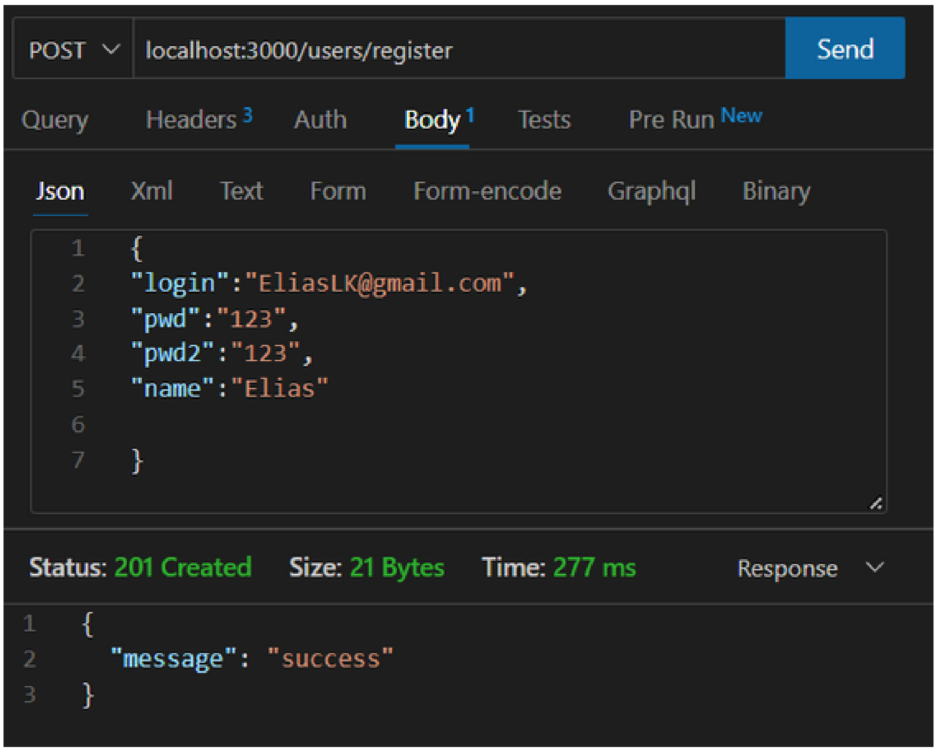
```
app.use((req,res,next)=>{
try {
const token = req.headers.authorization.split(' ')[1];
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.userData = decoded;
next();
} catch (error) {
return res.status(401).json({
message: 'Auth failed'
});
}
});
```

Finally, it starts the server on the port defined in the environment variable or 3000 if it's not defined and logs a message to the console when the server is up and running.



```
const port =process.env.port || 3000
app.listen(port, ()=>{
console.log('server listening on port : ',port)
})
```

## We should first register



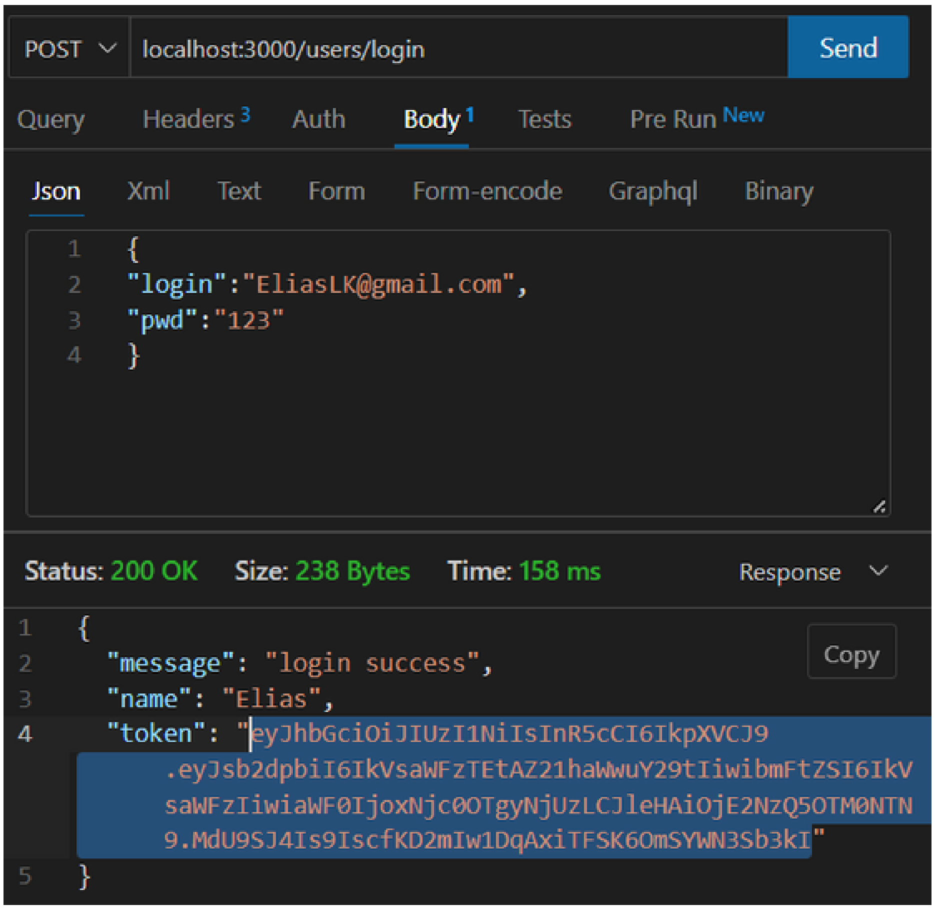
localhost:3000/users/register

```
{
"login":"EliasLK@gmail.com",
"pwd":"123",
"pwd2":"123",
"name":"Elias"

}
```

Status: 201 Created Size: 21 Bytes Time: 277 ms

```
{
  "message": "success"
}
```

## The login



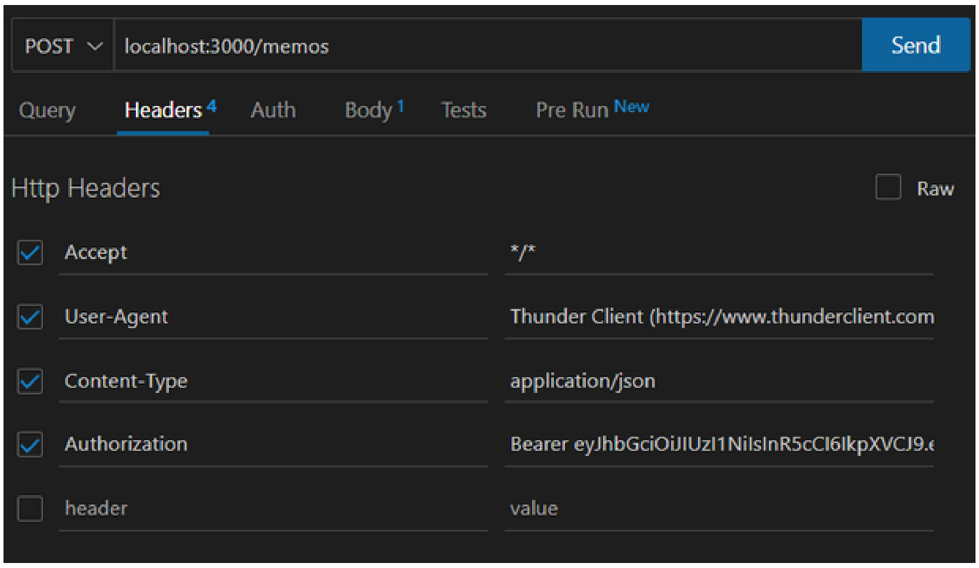
localhost:3000/users/login

```
{
"login":"EliasLK@gmail.com",
"pwd":"123"
}
```

Status: 200 OK Size: 238 Bytes Time: 158 ms

```
{
  "message": "login success",
  "name": "Elias",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
      .eyJsb2dpbiI6IkVsaWFzTEtAZ21haWwuY29tIiwibmFtZSI6IkV
      saWFzIiwiaWF0IjoxNjc0OTgyNjUzLCJleHAiOjE2NzQ5OTM0NTN
      9.MdU9SJ4Is9IscfKD2mIw1DqAxiTFSK6OmSYWN3Sb3kI"
}
```

**Now befor posting any memos we should first adding our token in the Authorization field in the headers**

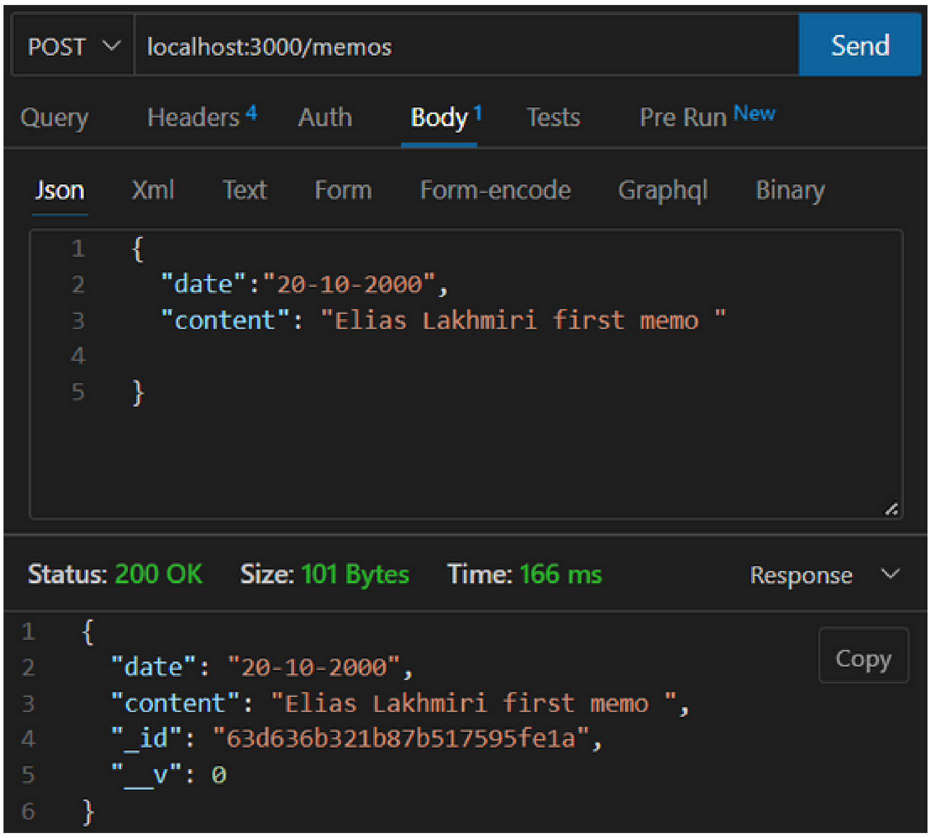


POST localhost:3000/memos

Http Headers

| | |
|---|---|
| Accept | */* |
| User-Agent | Thunder Client (https://www.thunderclient.com |
| Content-Type | application/json |
| Authorization | Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ε |
| header | value |

**We can now Post memos to this user which have this token**



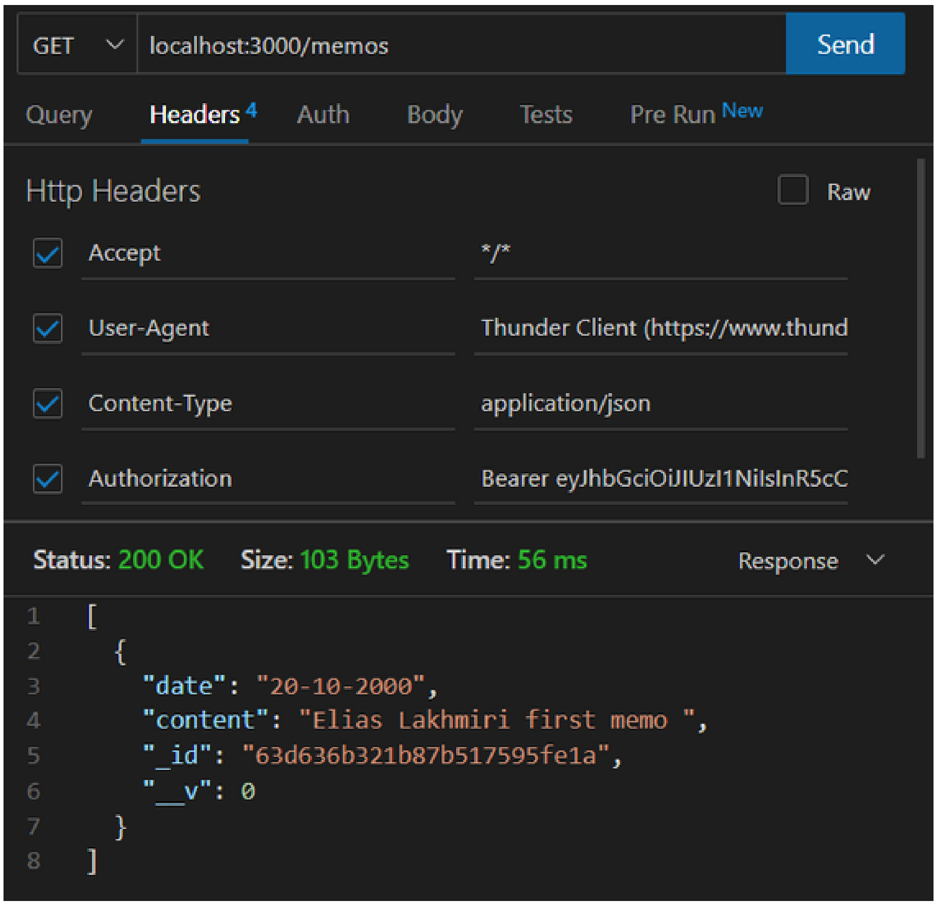
POST localhost:3000/memos

```
{
  "date":"20-10-2000",
  "content": "Elias Lakhmiri first memo "

}
```

Status: 200 OK Size: 101 Bytes Time: 166 ms

```
{
  "date": "20-10-2000",
  "content": "Elias Lakhmiri first memo ",
  "_id": "63d636b321b87b517595fe1a",
  "__v": 0
}
```

**Lets Post some random Memos to make some tests....after posting we can GET this memos as following:**

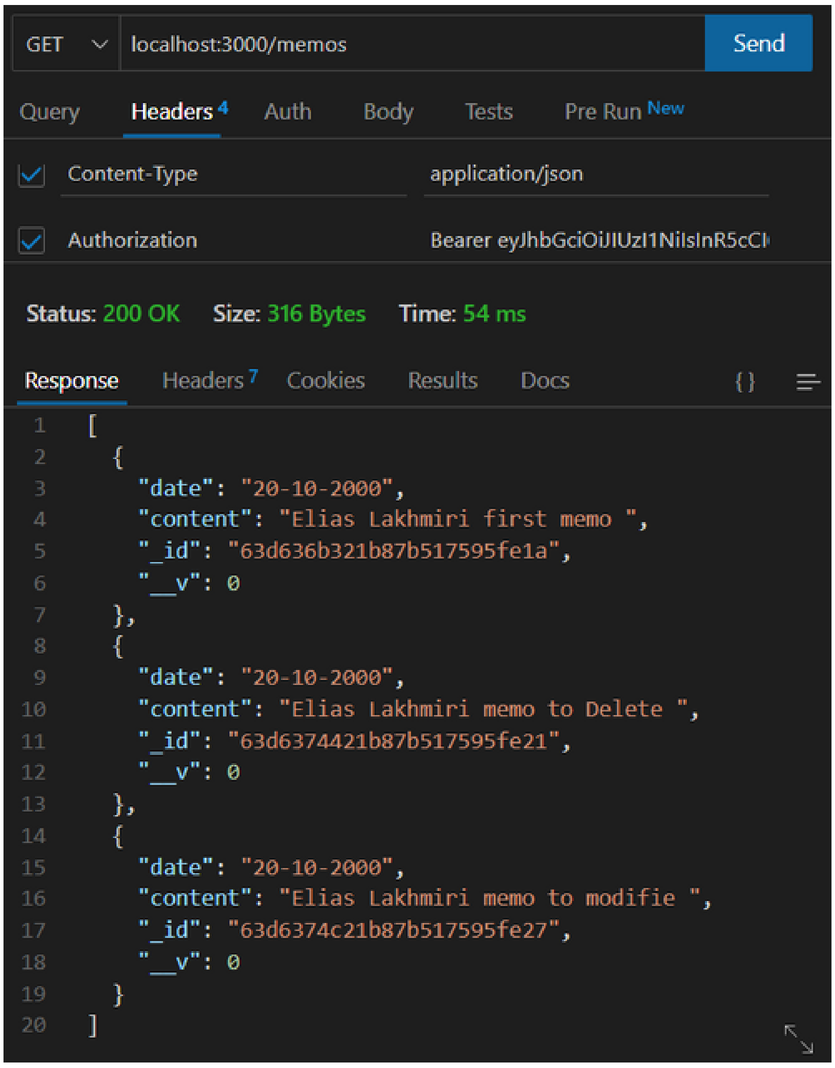


localhost:3000/memos

| | |
|---|---|
| Accept | */* |
| User-Agent | Thunder Client (https://www.thund |
| Content-Type | application/json |
| Authorization | Bearer eyJhbGciOiJIUzI1NiIsInR5cC |

Status: 200 OK Size: 103 Bytes Time: 56 ms

```
[
  {
    "date": "20-10-2000",
    "content": "Elias Lakhmiri first memo ",
    "_id": "63d636b321b87b517595fe1a",
    "__v": 0
  }
]
```



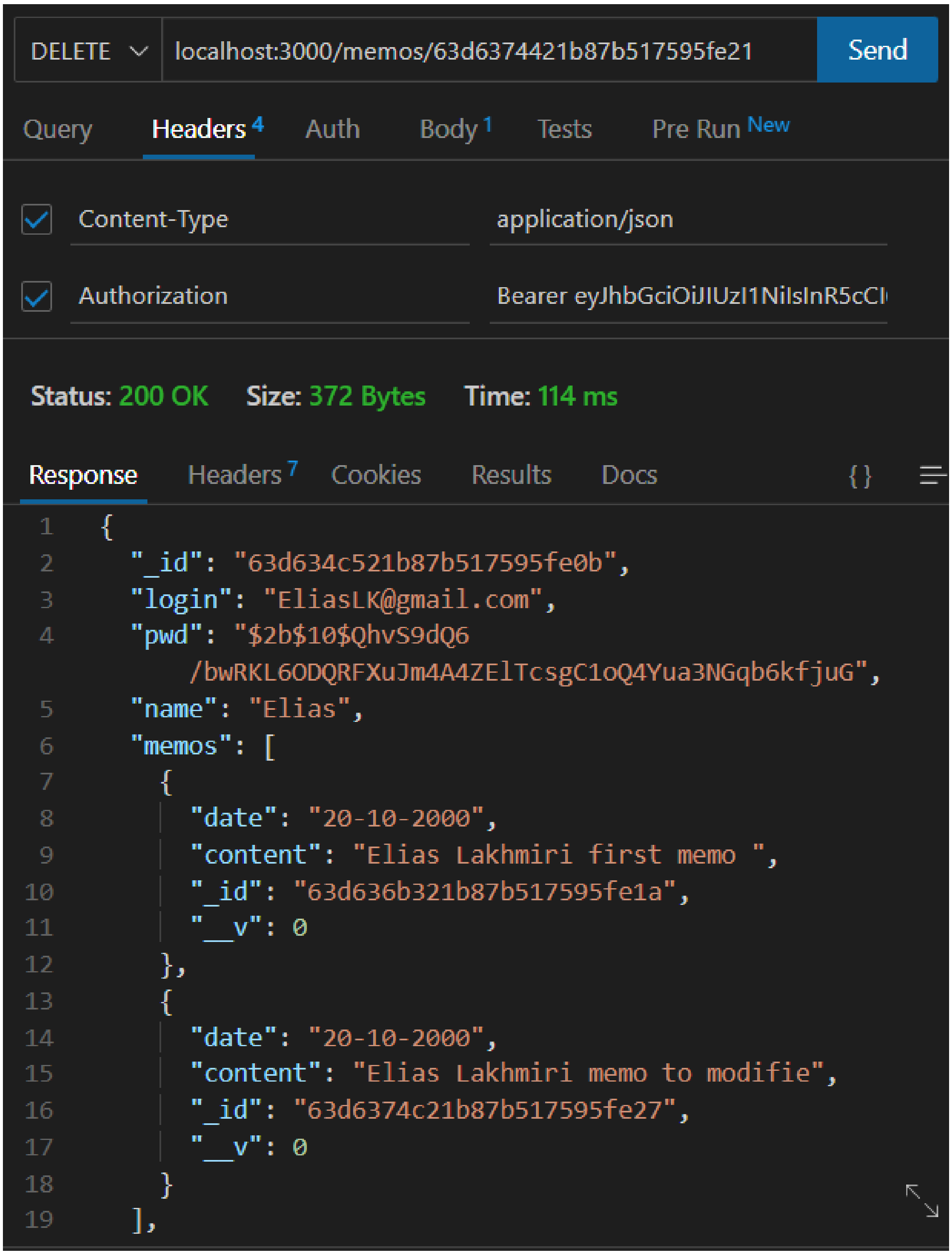
localhost:3000/memos

| | |
|---|---|
| Content-Type | application/json |
| Authorization | Bearer eyJhbGciOiJIUzI1NiIsInR5cCI |

Status: 200 OK Size: 316 Bytes Time: 54 ms

```
[
  {
    "date": "20-10-2000",
    "content": "Elias Lakhmiri first memo ",
    "_id": "63d636b321b87b517595fe1a",
    "__v": 0
  },
  {
    "date": "20-10-2000",
    "content": "Elias Lakhmiri memo to Delete ",
    "_id": "63d6374421b87b517595fe21",
    "__v": 0
  },
  {
    "date": "20-10-2000",
    "content": "Elias Lakhmiri memo to modifie ",
    "_id": "63d6374c21b87b517595fe27",
    "__v": 0
  }
]
```

now we can **Delete** a specific memo By ID



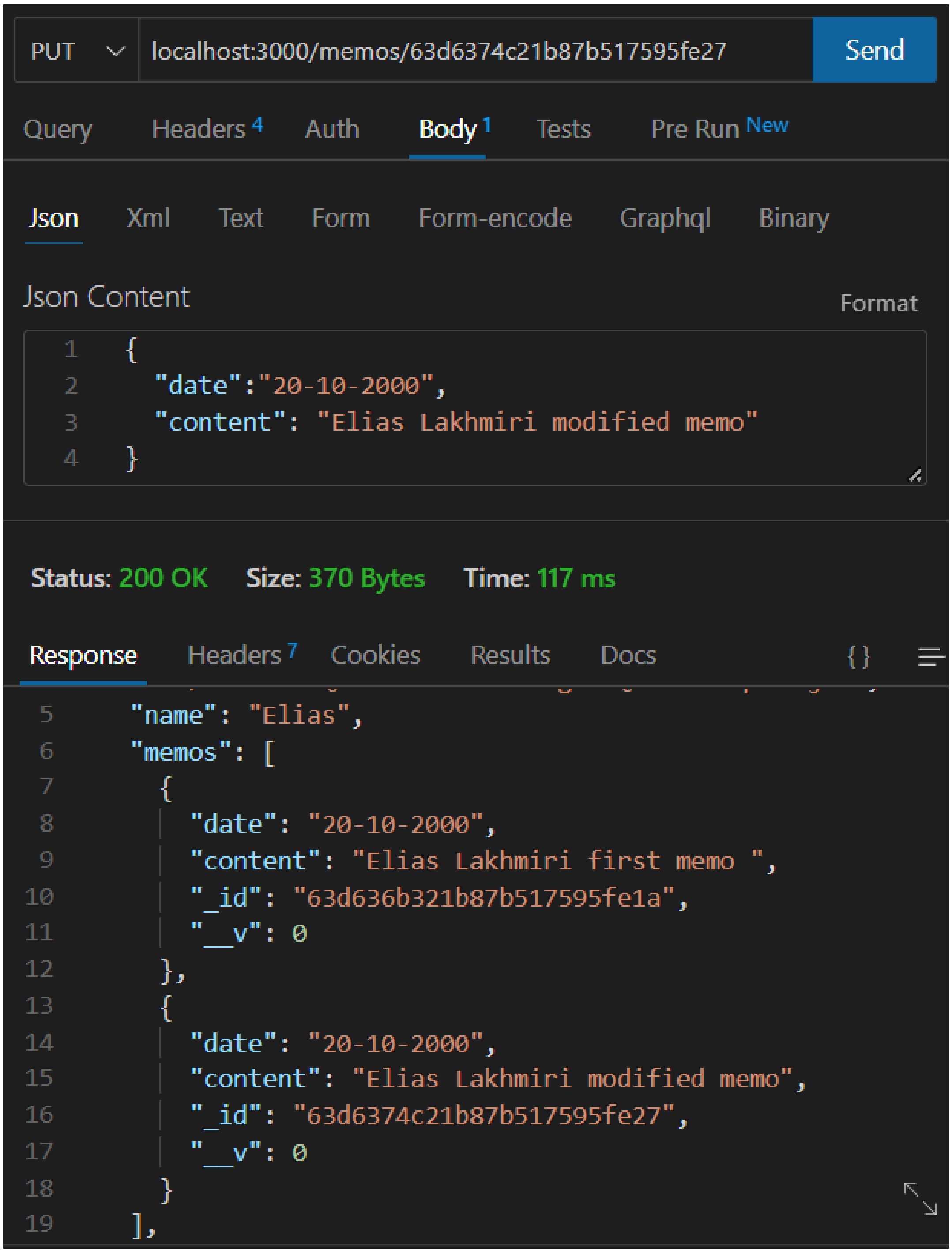
```
localhost:3000/memos/63d6374421b87b517595fe21
```

```
Content-Type    application/json
Authorization   Bearer eyJhbGciOiJIUzI1NiIsInR5cCI
```

Status: 200 OK    Size: 372 Bytes    Time: 114 ms

```
{
  "_id": "63d634c521b87b517595fe0b",
  "login": "EliasLK@gmail.com",
  "pwd": "$2b$10$QhvS9dQ6
      /bwRKL6ODQRFXuJm4A4ZElTcsgC1oQ4Yua3NGqb6kfjuG",
  "name": "Elias",
  "memos": [
    {
      "date": "20-10-2000",
      "content": "Elias Lakhmiri first memo ",
      "_id": "63d636b321b87b517595fe1a",
      "__v": 0
    },
    {
      "date": "20-10-2000",
      "content": "Elias Lakhmiri memo to modifie",
      "_id": "63d6374c21b87b517595fe27",
      "__v": 0
    }
  ],
```

**we can also Modifie a specific memo By ID**



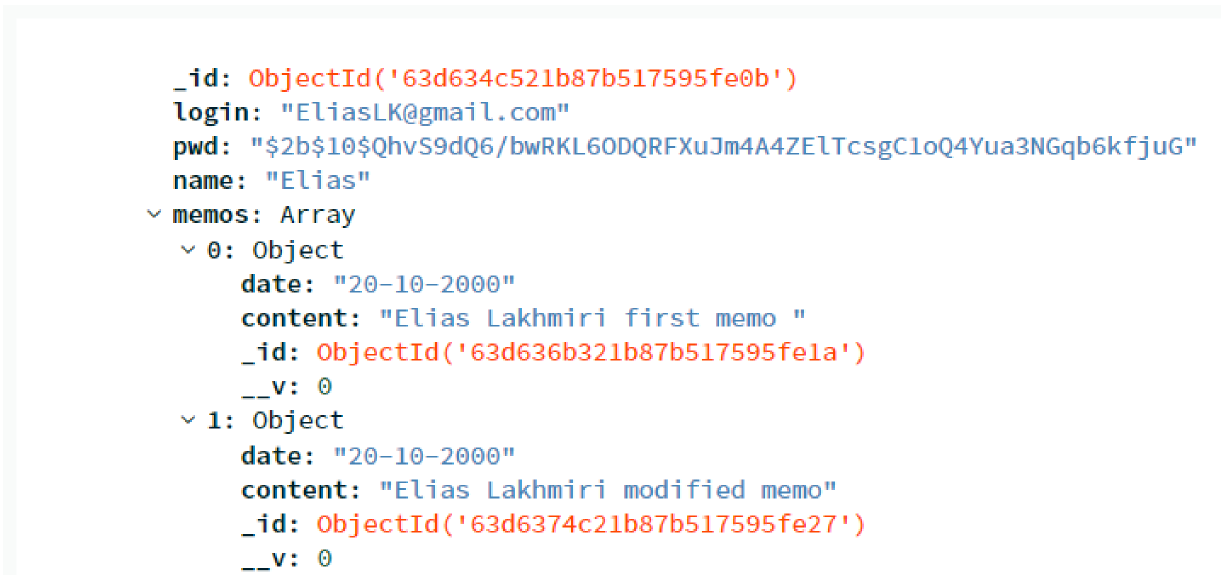
PUT localhost:3000/memos/63d6374c21b87b517595fe27

```
{
  "date":"20-10-2000",
  "content": "Elias Lakhmiri modified memo"
}
```

Status: 200 OK  Size: 370 Bytes  Time: 117 ms

```
    "name": "Elias",
    "memos": [
      {
        "date": "20-10-2000",
        "content": "Elias Lakhmiri first memo ",
        "_id": "63d636b321b87b517595fe1a",
        "__v": 0
      },
      {
        "date": "20-10-2000",
        "content": "Elias Lakhmiri modified memo",
        "_id": "63d6374c21b87b517595fe27",
        "__v": 0
      }
    ],
```

**We can see this Results in MongoDB Atlas**



```
_id: ObjectId('63d634c521b87b517595fe0b')
login: "EliasLK@gmail.com"
pwd: "$2b$10$QhvS9dQ6/bwRKL6ODQRFXuJm4A4ZElTcsgCloQ4Yua3NGqb6kfjuG"
name: "Elias"
memos: Array
  0: Object
    date: "20-10-2000"
    content: "Elias Lakhmiri first memo "
    _id: ObjectId('63d636b321b87b517595fe1a')
    __v: 0
  1: Object
    date: "20-10-2000"
    content: "Elias Lakhmiri modified memo"
    _id: ObjectId('63d6374c21b87b517595fe27')
    __v: 0
```

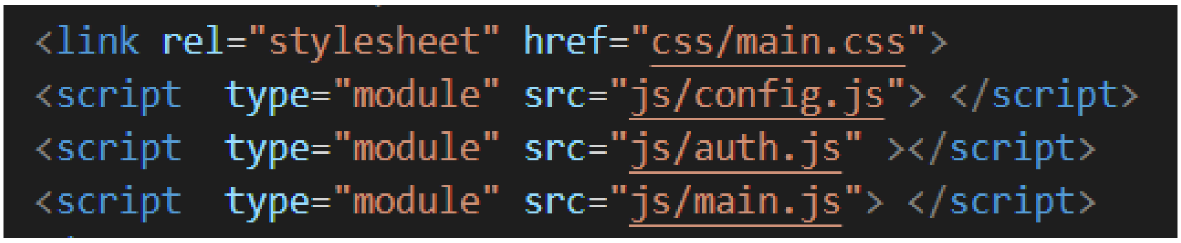
## CLIENT SIDE INDEX.HTML

1. The <link rel="stylesheet" href="css/main.css"> tag links the main CSS file for the page.

2.The three <script> tags load three JavaScript files:

- js/config.js
- js/auth.js
- js/main.js

These JavaScript files contain the JavaScript code for the page. The type="module" attribute specifies that these scripts are ES6 modules, which are a modern module system for JavaScript.

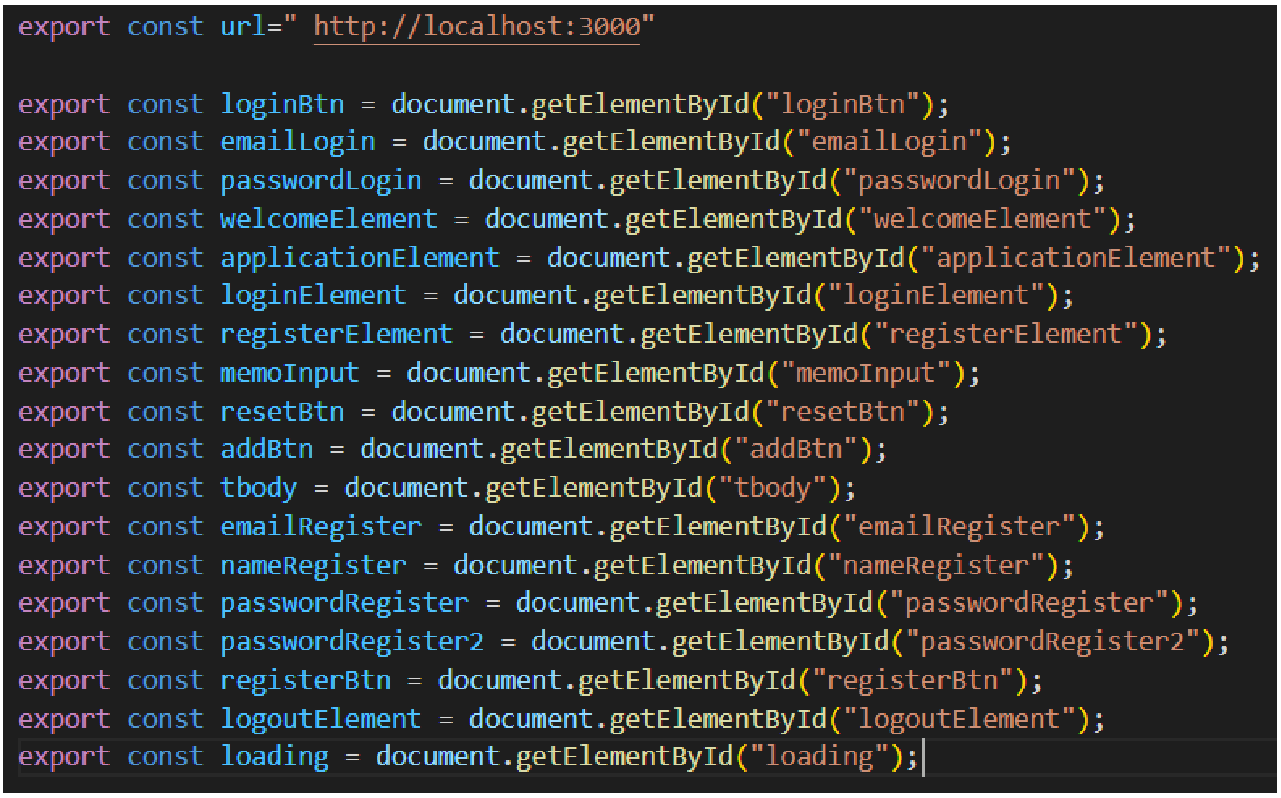


```html
<link rel="stylesheet" href="css/main.css">
<script  type="module" src="js/config.js"> </script>
<script  type="module" src="js/auth.js" ></script>
<script  type="module" src="js/main.js"> </script>
```

# CONFIG.JS

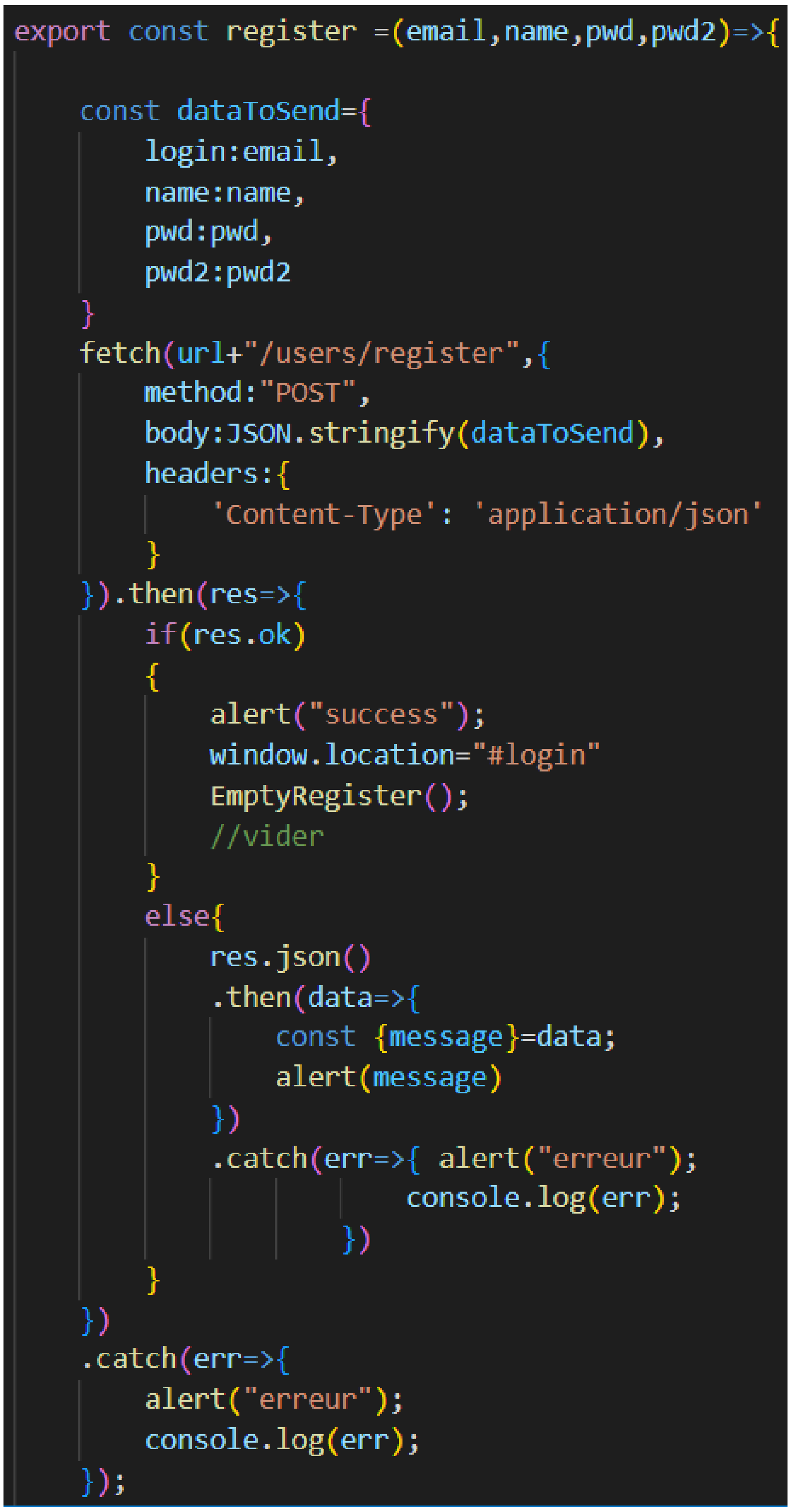
**exporting various DOM elements as constants to be used throughout the code**

**The elements include the login and register buttons, input fields for email, name, password, and password confirmation, various HTML elements for displaying the application and logging in/out, and a table for displaying memos**



```
export const url=" http://localhost:3000"

export const loginBtn = document.getElementById("loginBtn");
export const emailLogin = document.getElementById("emailLogin");
export const passwordLogin = document.getElementById("passwordLogin");
export const welcomeElement = document.getElementById("welcomeElement");
export const applicationElement = document.getElementById("applicationElement");
export const loginElement = document.getElementById("loginElement");
export const registerElement = document.getElementById("registerElement");
export const memoInput = document.getElementById("memoInput");
export const resetBtn = document.getElementById("resetBtn");
export const addBtn = document.getElementById("addBtn");
export const tbody = document.getElementById("tbody");
export const emailRegister = document.getElementById("emailRegister");
export const nameRegister = document.getElementById("nameRegister");
export const passwordRegister = document.getElementById("passwordRegister");
export const passwordRegister2 = document.getElementById("passwordRegister2");
export const registerBtn = document.getElementById("registerBtn");
export const logoutElement = document.getElementById("logoutElement");
export const loading = document.getElementById("loading");
```

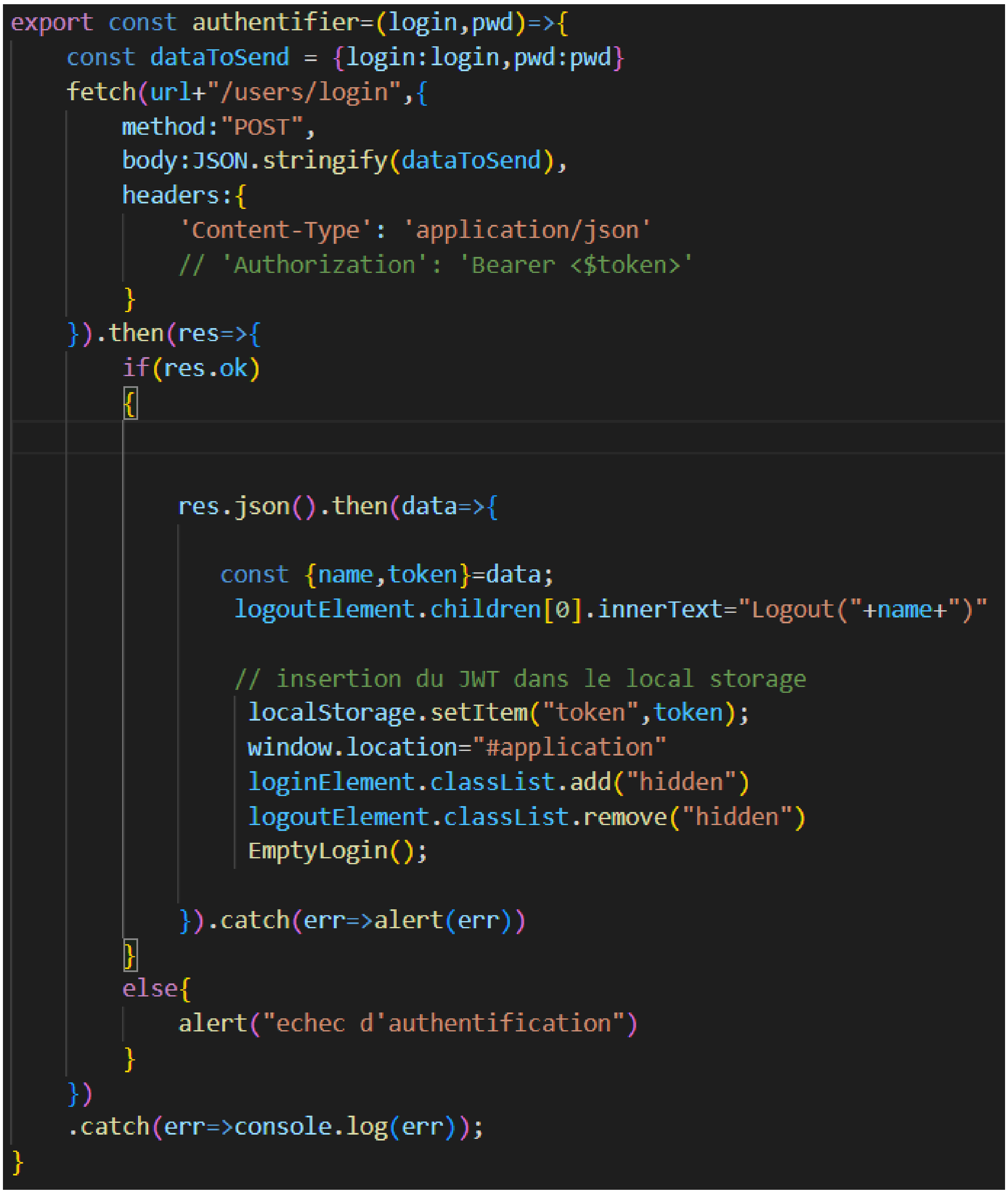
**The register** function takes an email, name, password and password confirmation as input and sends a POST request to the "/users/register" endpoint with the provided data. If the registration is successful, the user is redirected to the login page and the registration form is cleared.



```
export const register =(email,name,pwd,pwd2)=>{

    const dataToSend={
        login:email,
        name:name,
        pwd:pwd,
        pwd2:pwd2
    }
    fetch(url+"/users/register",{
        method:"POST",
        body:JSON.stringify(dataToSend),
        headers:{
            'Content-Type': 'application/json'
        }
    }).then(res=>{
        if(res.ok)
        {
            alert("success");
            window.location="#login"
            EmptyRegister();
            //vider
        }
        else{
            res.json()
            .then(data=>{
                const {message}=data;
                alert(message)
            })
            .catch(err=>{ alert("erreur");
                    console.log(err);
                })
        }
    })
    .catch(err=>{
        alert("erreur");
        console.log(err);
    });
```

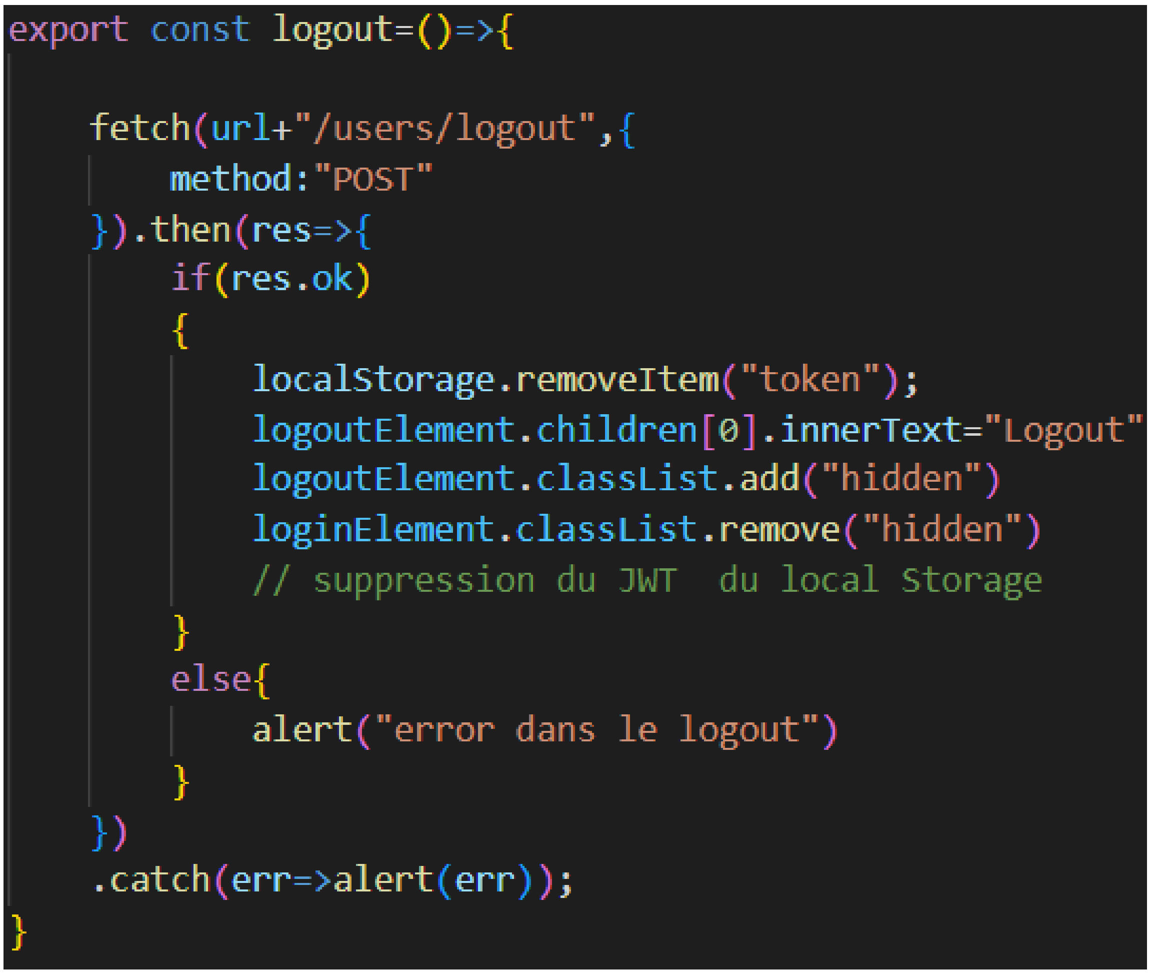
# CLIENT SIDE AUTH.JS

**This is a set of functions related to user authentication and registration using a JWT-based system. The authentifier function takes a username and password as input, then sends a POST request to the "/users/login" endpoint with the credentials as the request body. If the response is successful, the server returns a JSON object containing a user name and a token. The token is then stored in local storage and the user is redirected to the application page.**



```
export const authentifier=(login,pwd)=>{
    const dataToSend = {login:login,pwd:pwd}
    fetch(url+"/users/login",{
        method:"POST",
        body:JSON.stringify(dataToSend),
        headers:{
            'Content-Type': 'application/json'
            // 'Authorization': 'Bearer <$token>'
        }
    }).then(res=>{
        if(res.ok)
        {


            res.json().then(data=>{

               const {name,token}=data;
                logoutElement.children[0].innerText="Logout("+name+")"

                // insertion du JWT dans le local storage
                 localStorage.setItem("token",token);
                 window.location="#application"
                 loginElement.classList.add("hidden")
                 logoutElement.classList.remove("hidden")
                 EmptyLogin();

            }).catch(err=>alert(err))
        }
        else{
            alert("echec d'authentification")
        }
    })
    .catch(err=>console.log(err));
}
```

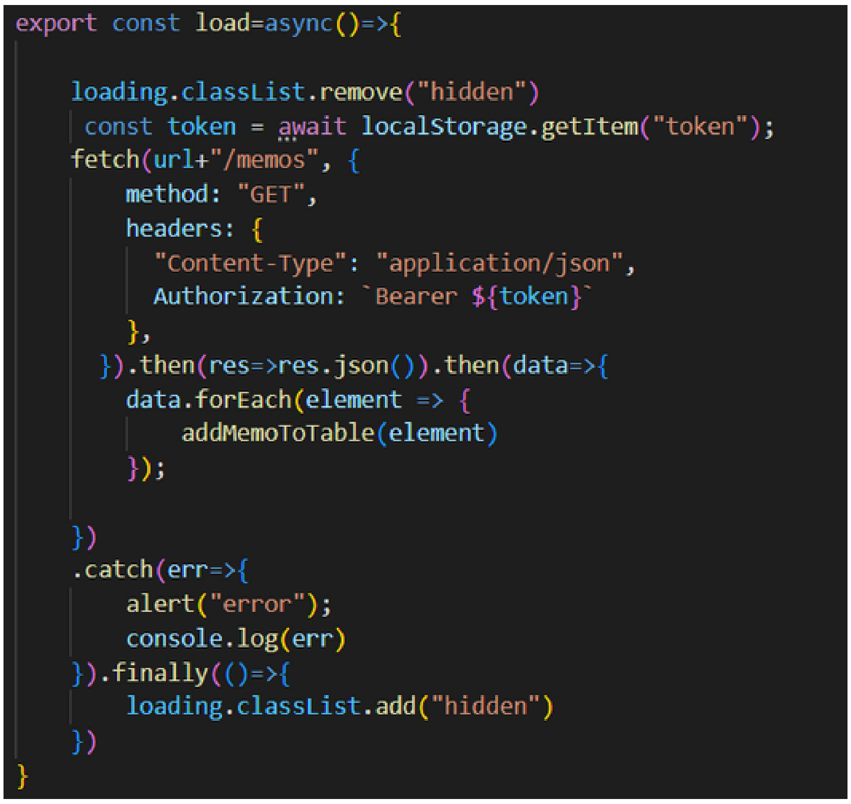
**The logout function sends a POST request to the "/users/logout" endpoint and removes the token from local storage.**



```
export const logout=()=>{

    fetch(url+"/users/logout",{
        method:"POST"
    }).then(res=>{
        if(res.ok)
        {
            localStorage.removeItem("token");
            logoutElement.children[0].innerText="Logout"
            logoutElement.classList.add("hidden")
            loginElement.classList.remove("hidden")
            // suppression du JWT  du local Storage
        }
        else{
            alert("error dans le logout")
        }
    })
    .catch(err=>alert(err));
}
```

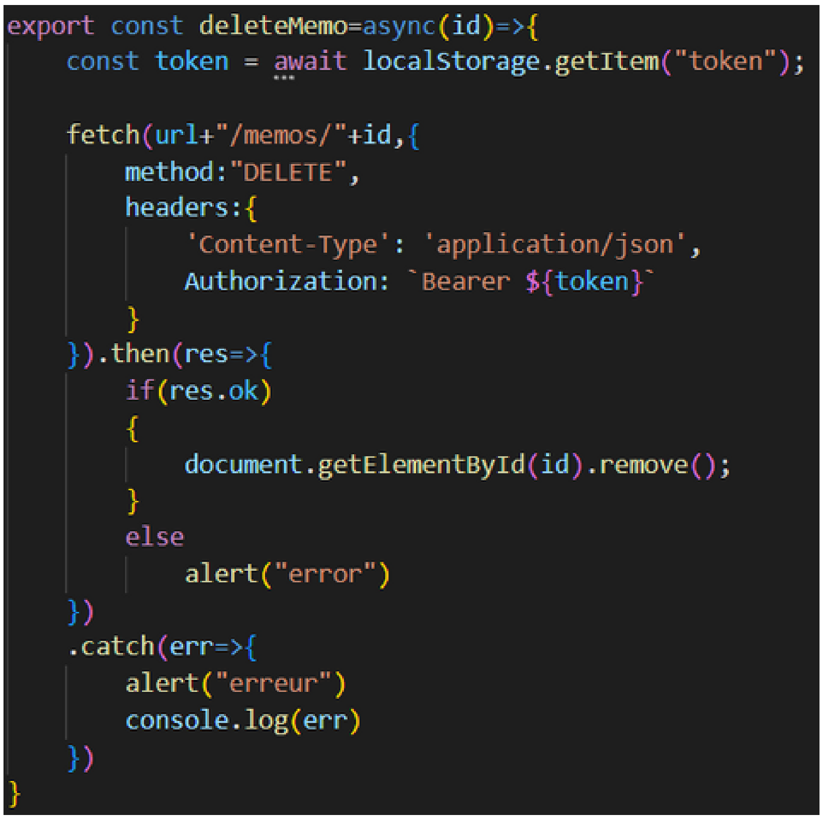
in "memos.js" we exports functions to interact with our memo application's REST API.

- **load**: fetches all the memos from the API and adds them to the table in the UI.



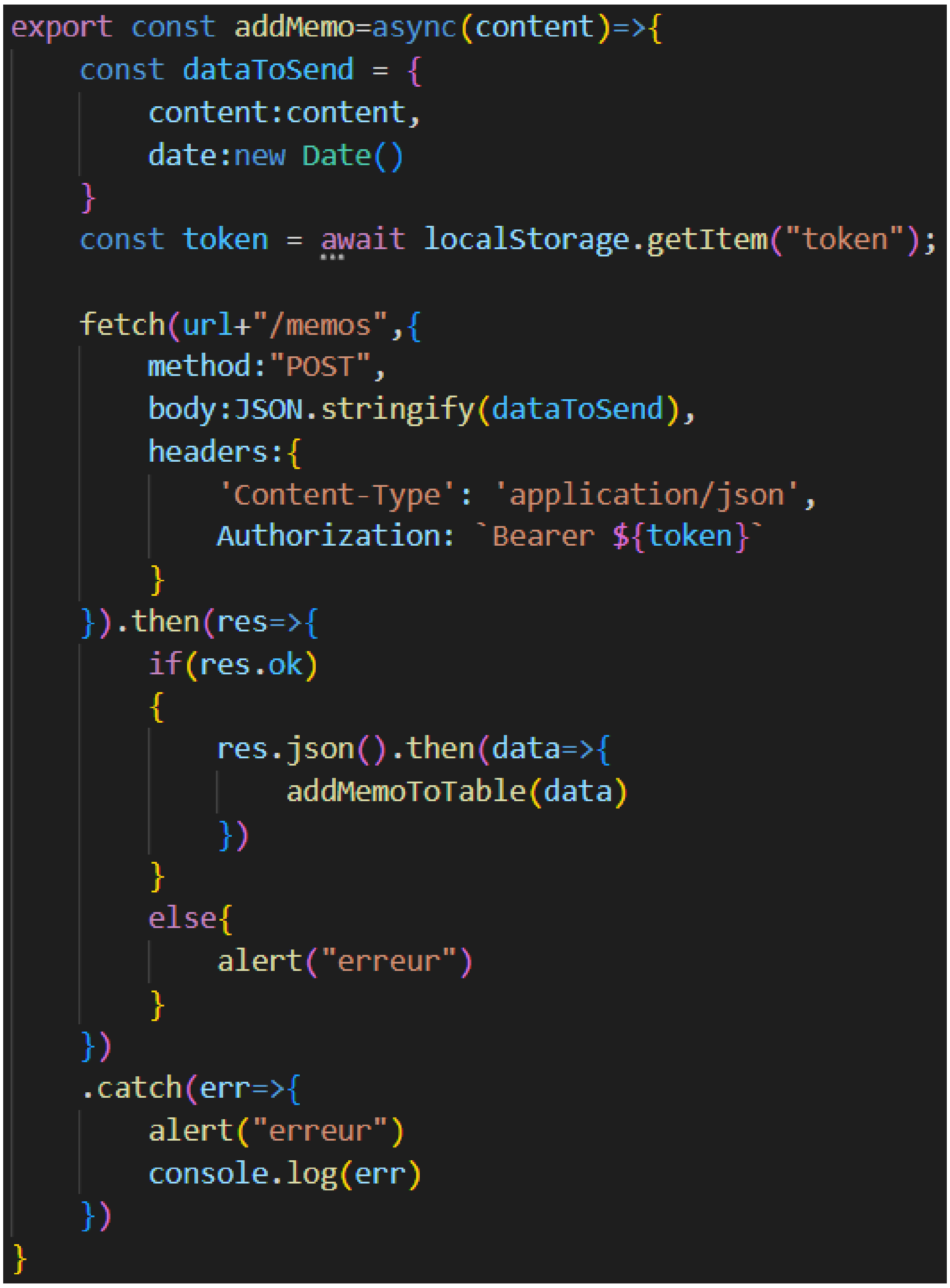
```
export const load=async()=>{

    loading.classList.remove("hidden")
     const token = await localStorage.getItem("token");
    fetch(url+"/memos", {
        method: "GET",
        headers: {
          "Content-Type": "application/json",
          Authorization: `Bearer ${token}`
        },
      }).then(res=>res.json()).then(data=>{
        data.forEach(element => {
            addMemoToTable(element)
        });

    })
    .catch(err=>{
        alert("error");
        console.log(err)
    }).finally(()=>{
        loading.classList.add("hidden")
    })
}
```

- **deleteMemo:** sends a request to delete a memo with a given ID



```
export const deleteMemo=async(id)=>{
    const token = await localStorage.getItem("token");

    fetch(url+"/memos/"+id,{
        method:"DELETE",
        headers:{
            'Content-Type': 'application/json',
            Authorization: `Bearer ${token}`
        }
    }).then(res=>{
        if(res.ok)
        {
            document.getElementById(id).remove();
        }
        else
            alert("error")
    })
    .catch(err=>{
        alert("erreur")
        console.log(err)
    })
}
```

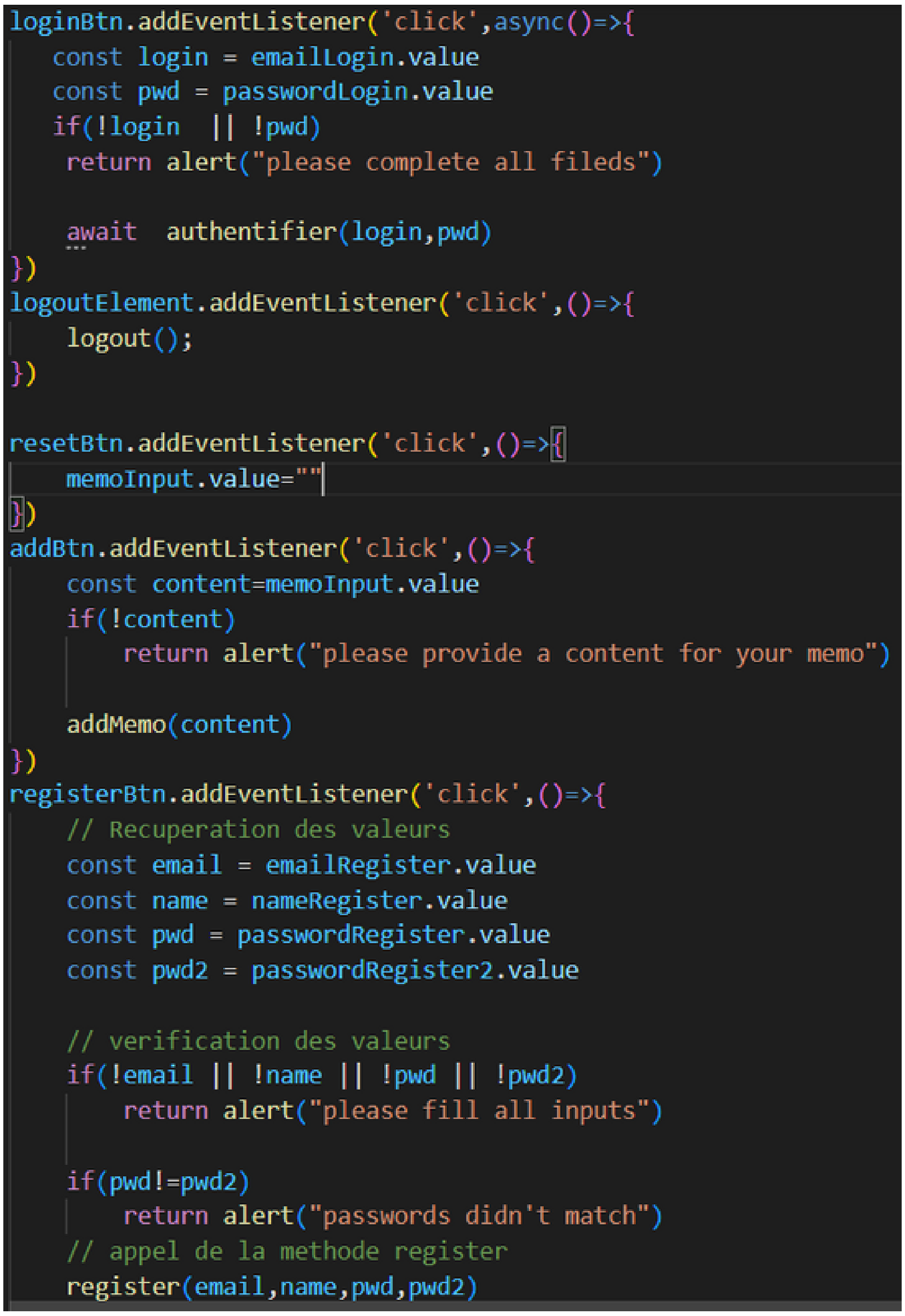
- **addMemo: sends a new memo to the API.**



```javascript
export const addMemo=async(content)=>{
    const dataToSend = {
        content:content,
        date:new Date()
    }
    const token = await localStorage.getItem("token");

    fetch(url+"/memos",{
        method:"POST",
        body:JSON.stringify(dataToSend),
        headers:{
            'Content-Type': 'application/json',
            Authorization: `Bearer ${token}`
        }
    }).then(res=>{
        if(res.ok)
        {
            res.json().then(data=>{
                addMemoToTable(data)
            })
        }
        else{
            alert("erreur")
        }
    })
    .catch(err=>{
        alert("erreur")
        console.log(err)
    })
}
```

# CLIENT SIDE MAIN.JS

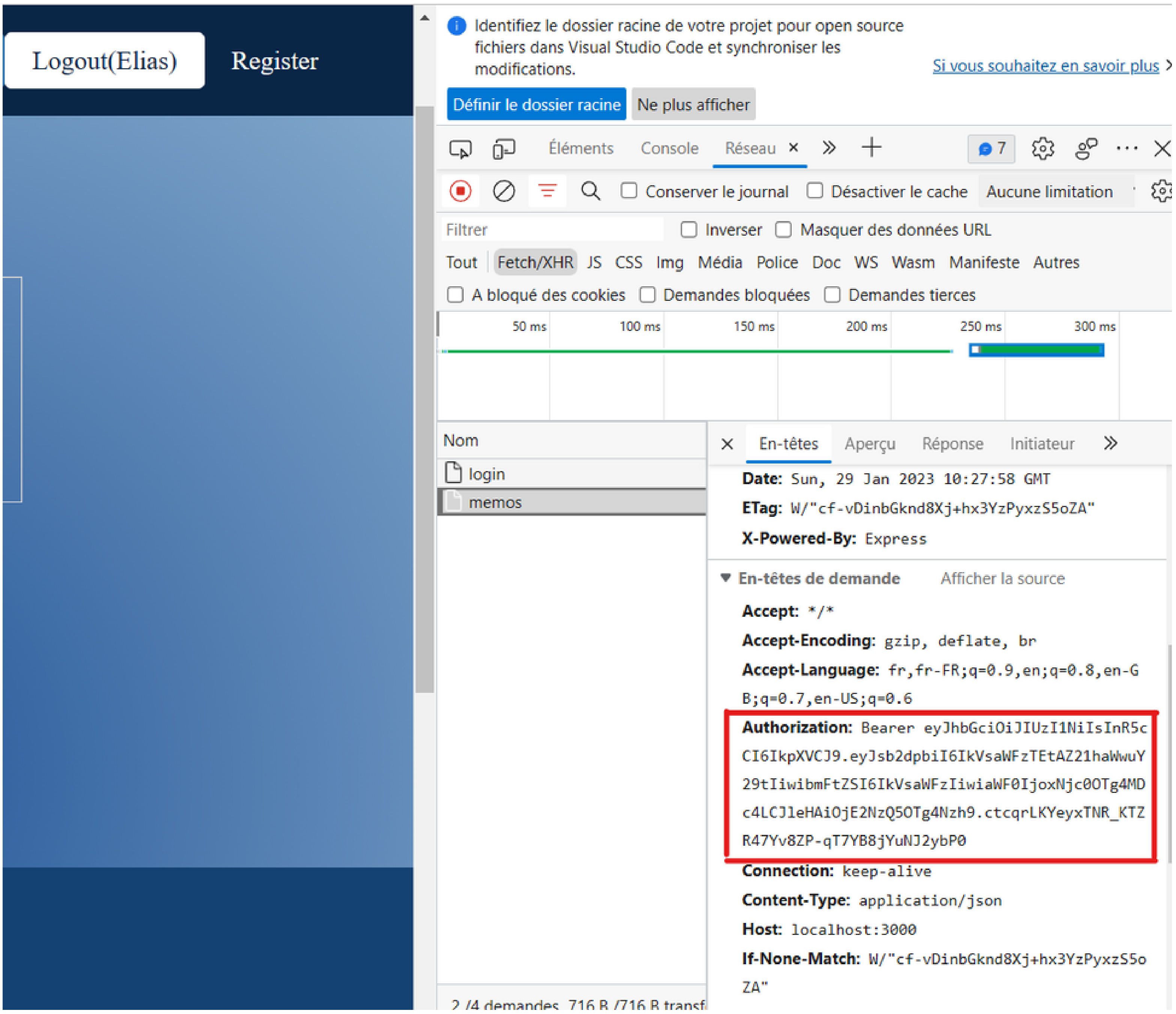
- **Now we sets up event listeners for different actions on the page, such as logging in, logging out, adding memos, and registering. It also has helper functions for clearing input fields for the login and for adding a memo to the table. The checkConnection function checks if the user is connected by checking the idUser in local storage.**



```
loginBtn.addEventListener('click',async()=>{
   const login = emailLogin.value
   const pwd = passwordLogin.value
   if(!login  || !pwd)
    return alert("please complete all fileds")

    await  authentifier(login,pwd)
})
logoutElement.addEventListener('click',()=>{
    logout();
})

resetBtn.addEventListener('click',()=>{
    memoInput.value=""
})
addBtn.addEventListener('click',()=>{
    const content=memoInput.value
    if(!content)
        return alert("please provide a content for your memo")

    addMemo(content)
})
registerBtn.addEventListener('click',()=>{
    // Recuperation des valeurs
    const email = emailRegister.value
    const name = nameRegister.value
    const pwd = passwordRegister.value
    const pwd2 = passwordRegister2.value

    // verification des valeurs
    if(!email || !name || !pwd || !pwd2)
        return alert("please fill all inputs")

    if(pwd!=pwd2)
        return alert("passwords didn't match")
    // appel de la methode register
    register(email,name,pwd,pwd2)
```

# Test on Utilisateur Interface

**As we can see our Token in the header of HTTP Request**



```
Date: Sun, 29 Jan 2023 10:27:58 GMT
ETag: W/"cf-vDinbGknd8Xj+hx3YzPyxzS5oZA"
X-Powered-By: Express
▼ En-têtes de demande        Afficher la source
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: fr,fr-FR;q=0.9,en;q=0.8,en-G
B;q=0.7,en-US;q=0.6
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5c
CI6IkpXVCJ9.eyJsb2dpbiI6IkVsaWFzTEtAZ21haWwuY
29tIiwibmFtZSI6IkVsaWFzIiwiaWF0IjoxNjc0OTg4MD
c4LCJleHAiOjE2NzQ5OTg4Nzh9.ctcqrLKYeyxTNR_KTZ
R47Yv8ZP-qT7YB8jYuNJ2ybP0
Connection: keep-alive
Content-Type: application/json
Host: localhost:3000
If-None-Match: W/"cf-vDinbGknd8Xj+hx3YzPyxzS5o
ZA"
```

2 /4 demandes 716 B /716 B transf

# CONCLUSION

**In conclusion, the implementation of the Node.js application with MongoDB Atlas and Express framework has provided a robust and secure solution for managing user data and memos. The implementation leverages the Model-View-Controller architecture for separation of concerns and efficient code organization. The implementation also implements security concepts such as JWT authentication and middleware to protect sensitive routes and data. The use of environment variables and the dotenv package enhances security by allowing sensitive information to be stored outside the codebase. With the integration of these concepts and best practices, the application is well-equipped to handle user data in a secure and efficient manner.**