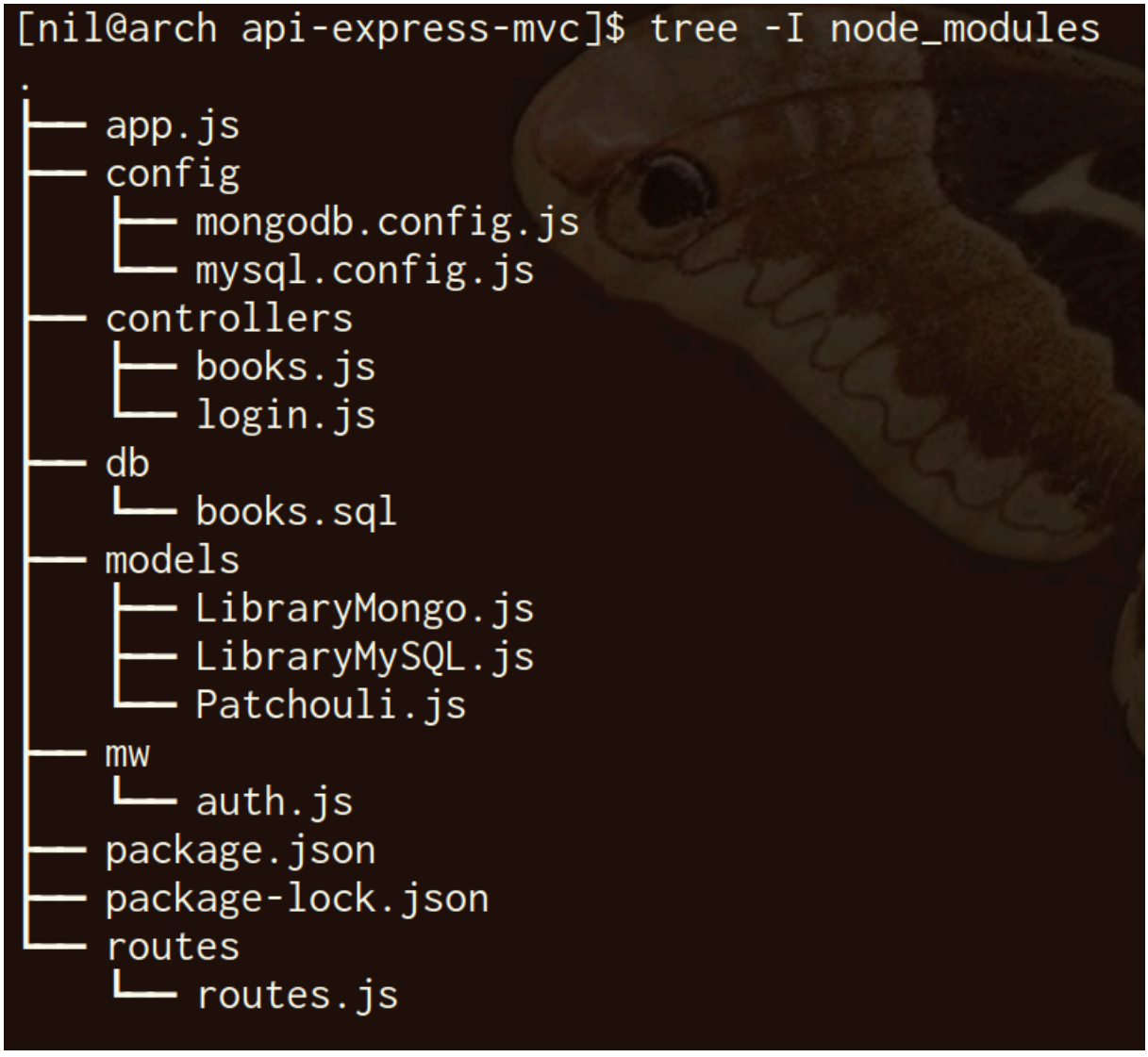
MP08 - Final project

Nil Jimeno - ETP Xavier

Structure

The project is divided in 2 main sections: the client, which is just a GUI client to access the api remotely, and the api.



The client

The client consists of an html file, a css file and a js file. It has a bare bones structure.

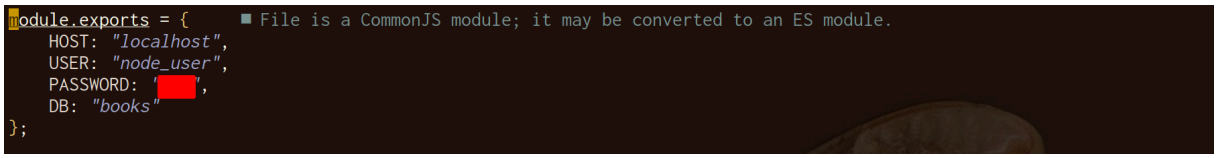
Backend (API)

The API attempts to replicate an MVC kind of structure. It starts at app.js, which is the main file, and then the routes file, which divides its branches to other folders such as the models and the controllers.

Models

There are 2 main models in the application: LibraryMySQL and LibraryMongo, one for each database. These contain the functions that will interact with the desired database. Then there's Patchouli (whose name is a bad pun), that exports the functions from the appropiate database.

The configuration files for the models are not only outside the models folder, but also don't use a .env file to save credentials. Then there's also the middleware folder, which is called "mw" (not very self-explainatory) that is supposed to interact with the database from outside the models folder.



There is also a .sql file (done by me), which is quite bad. Ideally, you would use something like a shell script or the like.

Controllers

These manage the surface-level api logic and interact with both the model and the client. They process the values from the client and send them to the model (including the authentication model, which is not in the models folder (bad)), then wait for the response, process it and send it back to the client.

MongoDB adaptation

The MongoDB library effectively works the same as MongoDB, having the same commands to interact with the database. Most of that work was to replace the SQL code into MongoDB code directly, but there has also been some issues with concurrency since js does not allow waiting for asyncronous functions in a constructor class. To solve that, the program creates a connection at the beggining of each route, which is definitely not the best solution but it kind of works!

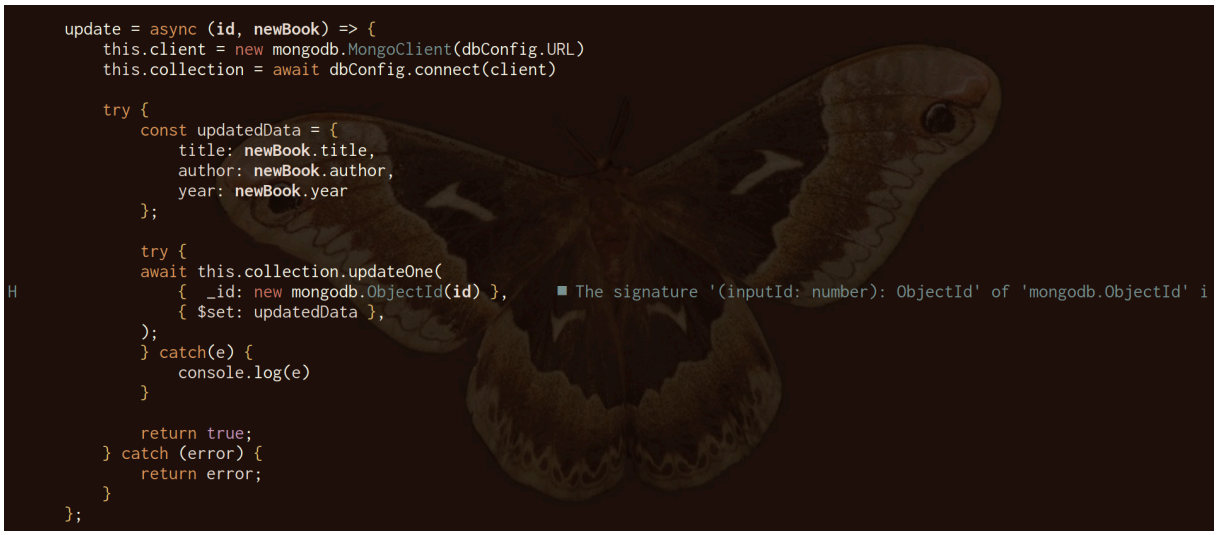
The database used can be changed in the .env file.

Screenshots

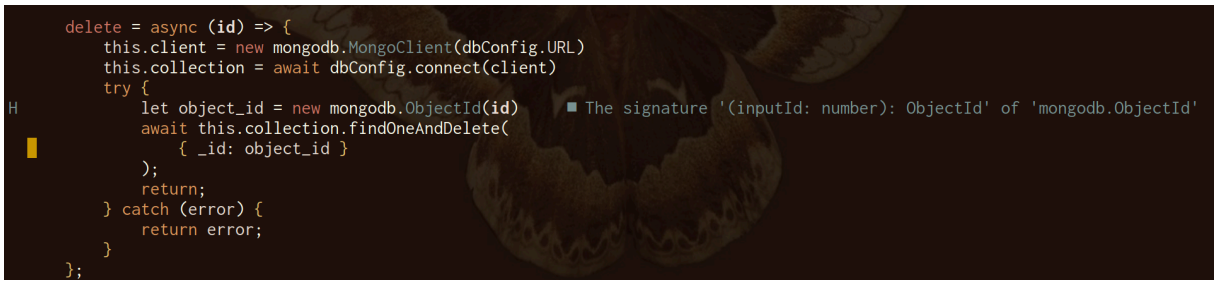
```
H const mongodb = require("mongodb");      ■ File is a CommonJS module; it may be

  require("mongodb").MongoClient
  const url = "mongodb://localhost:27017"

  client = new mongodb.MongoClient(url)

  async function connect(client) {
      console.log("Attempting to connect to database")

      try {
          await client.connect()
          console.log("Connected to database")
      } catch(err) {
          console.log("Connection failed", err)
      }

      let db = await client.db("nyanko")
      return await db.createCollection("books")
  }

  module.exports = {
      URL: "mongodb://localhost:27017",
      connect: connect,
  };
~
```


```
    listAll = async () => {
        this.client = new mongodb.MongoClient(dbConfig.URL)
        this.collection = await dbConfig.connect(client)

        try {
            let result = []
            await this.collection.find().forEach(a => {
                a.id = a._id.toString()
                result.push(a)
            });
            return result;
        } catch (error) {
            return error;
        }
    };
```

```
    create = async (newBook) => {
        this.client = new mongodb.MongoClient(dbConfig.URL)
        this.collection = await dbConfig.connect(client)

        try {
            await this.collection.insertOne(newBook);
            return true;
        } catch (error) {
            return error;
        }
    };
```



```
update = async (id, newBook) => {
    this.client = new mongodb.MongoClient(dbConfig.URL)
    this.collection = await dbConfig.connect(client)

    try {
        const updatedData = {
            title: newBook.title,
            author: newBook.author,
            year: newBook.year
        };

        try {
        await this.collection.updateOne(
            {  _id: new mongodb.ObjectId(id) },
            { $set: updatedData },
        );
        } catch(e) {
            console.log(e)
        }

        return true;
    } catch (error) {
        return error;
    }
};
```



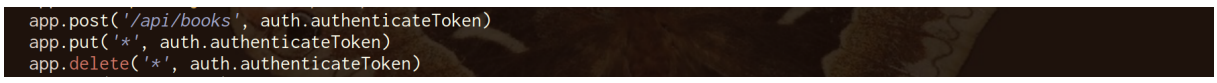
```
delete = async (id) => {
    this.client = new mongodb.MongoClient(dbConfig.URL)
    this.collection = await dbConfig.connect(client)
    try {
        let object_id = new mongodb.ObjectId(id)
        await this.collection.findOneAndDelete(
            { _id: object_id }
        );
        return;
    } catch (error) {
        return error;
    }
};
```

Why not just check the code yourself…

JWT

A middleware has been implemented in the server and is being called from app.js:

```
function authenticateToken(req, res, next) {
    let header = req.headers['authorization']
    let token =  header && header.split(' ')[1]
    console.log("token", token)

     if (token == null) return res.sendStatus(401)

    jwt.verify(token, process.env.TOKEN, (err) => {
        console.log(err)
        if (err) return res.sendStatus(403)
        next()
    })
}

module.exports = {
    login: login,
    authenticateToken: authenticateToken,
}
```



```
app.post('/api/books', auth.authenticateToken)
app.put('*', auth.authenticateToken)
app.delete('*', auth.authenticateToken)
```

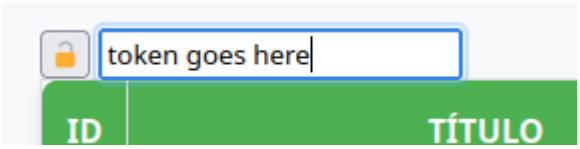
A new controller has been added for the login route, which gets the POST values to check if the user and password match.

The login implementation is scuffy to say the least: it doesn't connect to any database because 1. there is no way to register new users and 2. security has been completely disregarded since passwords have been saved outside of a dotenv, therefore it wouldn't make sense. Plus, it hasn't been specified anywhere so it's not a requirement.

```
function login(req) {
    if (!areCredentialsValid(req.body.user, req.body.password)) {
        return {
            access: `denied`,
            message: `access denied for ${req.body.user}`,
        }
    }
    return {
        access: "authorized",
        token: generateToken(req.body.user)
    }
}
```

As for the frontend of the page, no changes were required according to the instructions.

Insert your token in the input box and lock it with the lock button (careful with the extra spaces!!). Negated requests will not be notified since that wasn't specified anywhere, and for logins you still have to use an application like curlie or the like.




```
[nil@arch ~]$ curlie :5000/api/login user=user password=password
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 186
ETag: W/"ba-TIoGbrDx29EHgEXGbsR2Pl/RnkM"
Date: Sun, 23 Feb 2025 17:14:54 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
    "access": "authorized",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6In
vlCi-Yi3gs4ZuoiqbeYHYUlqCO6VoZ6xp2vUFO0"
```