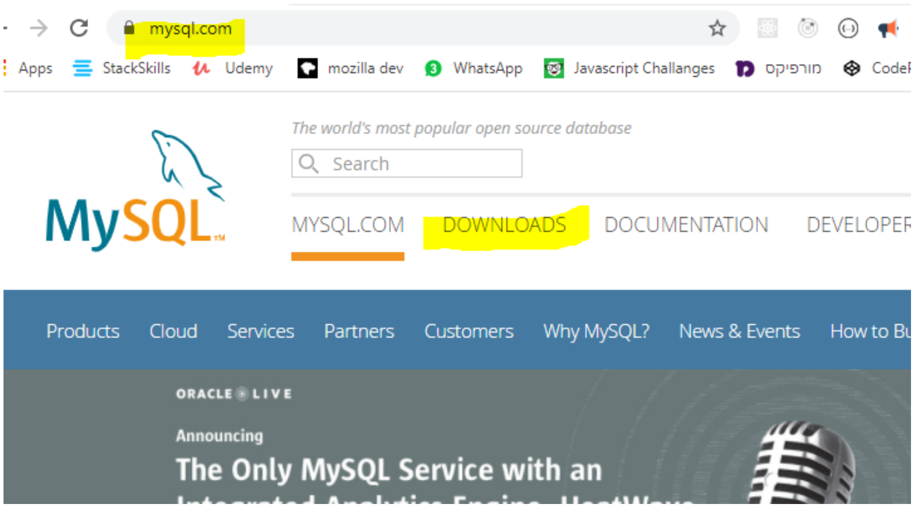
Derma-Detect home assignment

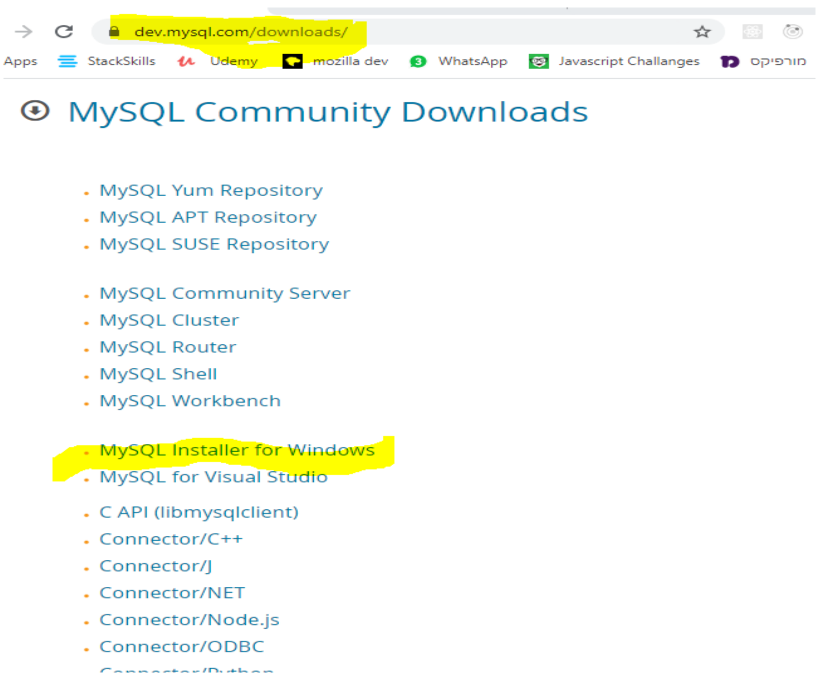
Let’s create a small CRUD service with MySQL and Token based authentication

Make sure you download and install MySQL on your system:

Setup instructions- https://www.youtube.com/watch?v=OM4aZJW_Ojs



Look for the community installer link, and then the windows link:


dev.mysql.com/downloads/

MySQL Community Downloads

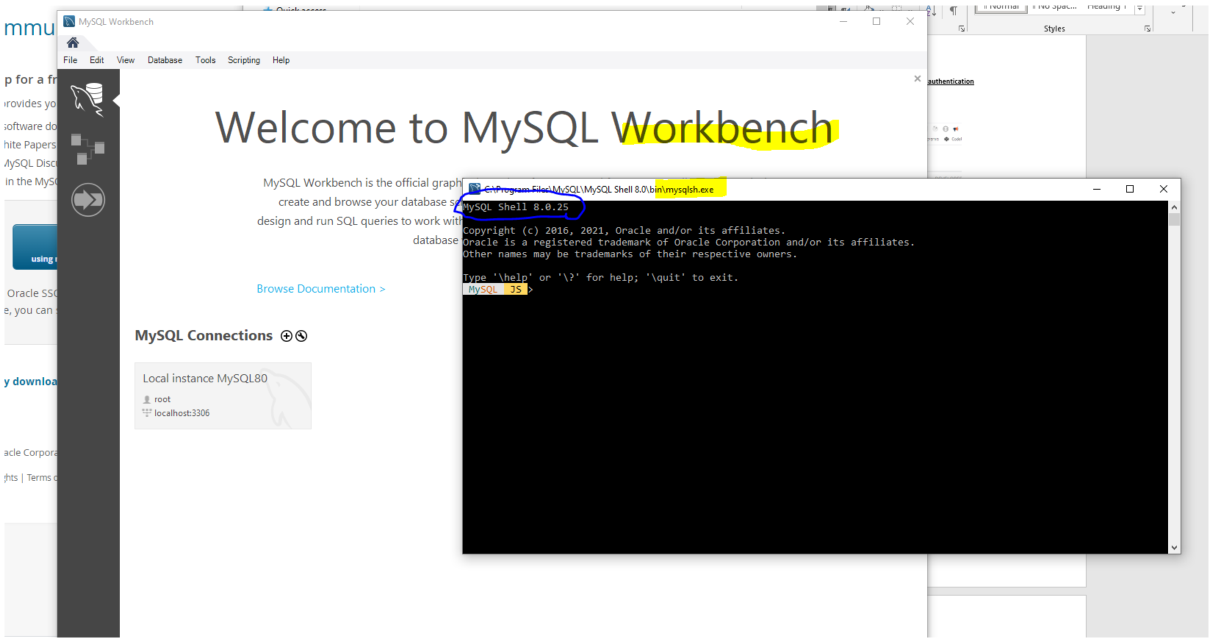
- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository

- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench

- MySQL Installer for Windows
- MySQL for Visual Studio

- C API (libmysqlclient)
- Connector/C++
- Connector/J
- Connector/NET
- Connector/Node.js
- Connector/ODBC

Once you finished the setup the MySQL workbench and shell will be opend:

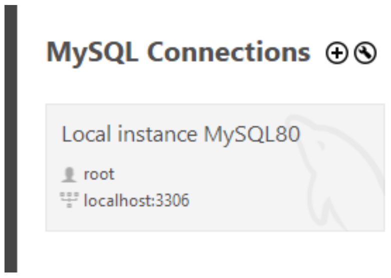

MySQL Workbench
File Edit View Database Tools Scripting Help
Welcome to MySQL Workbench
Browse Documentation >
MySQL Connections
Local instance MySQL80
root
localhost:3306
C:\Program Files\MySQL\MySQL Shell 8.0\bin\mysqlsh.exe

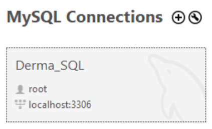
```
MySQL Shell 8.0.25

Copyright (c) 2016, 2021, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.
 MySQL  JS >
```

Log in into the root account with the password you set during the setup…

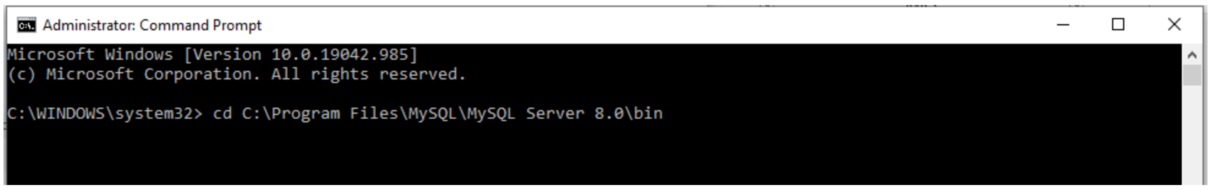
Click it and login:

(you can create a new DB using the "+" sign) or edit the current, just give it a name you will remember to use later)



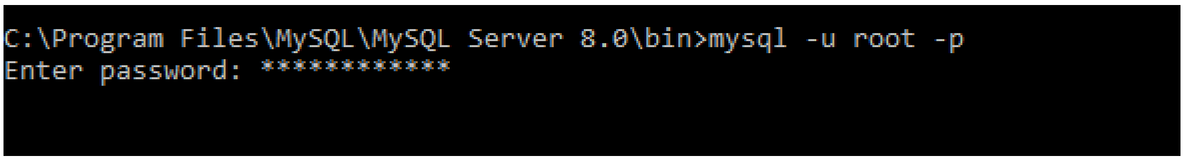


Open CMD as an admin (by right clicking it, and enter the path to the my sql server bin folder:



```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32> cd C:\Program Files\MySQL\MySQL Server 8.0\bin
```

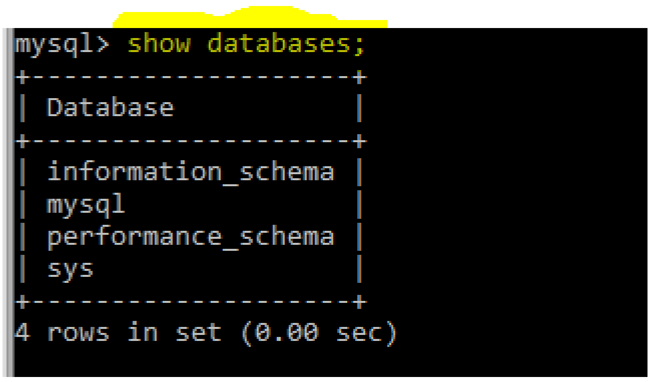
Then : write- mysql -u root -p and enter your DB root user password:



```
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u root -p
Enter password: ************
```

Write >> show databases;

And you will see the defaul DB



```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)
```

Let’s create a user with all privleges & a database we will work with in this app:

In root user:

1. mysql> CREATE USER 'db_user'@'localhost' IDENTIFIED BY '(leave it empty mening no password)';
2. mysql> GRANT ALL PRIVILEGES ON *.* TO 'db_user'@'localhost';
3. mysql> \q →log out of your root user
4. log in into db_user →
   C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u db_user -p
   Enter password: (press enter because we have no password)
5. mysql> Show grants; ➔ will show the db_user privleges
6. mysql> CREATE DATABASE database_development; ➔ will create the DB we will work with localy in dev mode.

<u>Setup the project</u>

<u>Make sure you have Nodejs installed on your machine as well.</u>

- **Create a folder for your project**
- **Open terminal in this folder**
- **>>npm init -y**
- **Install express using npm : >>npm i –save express**

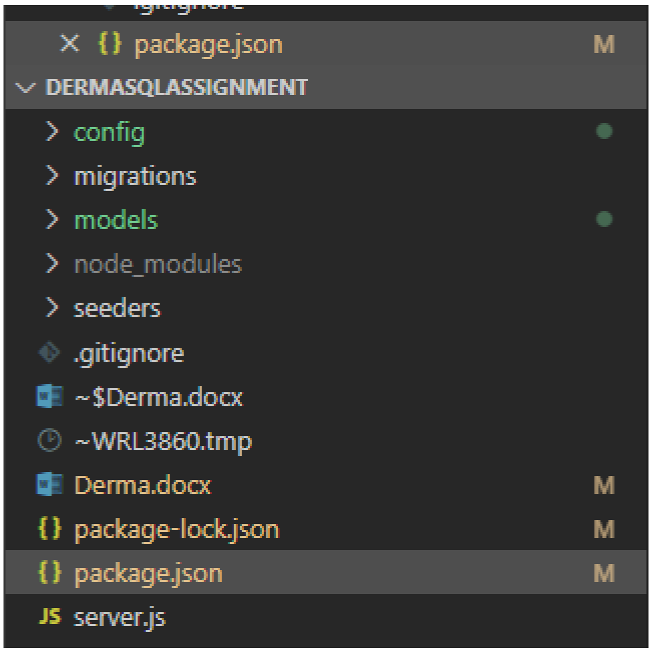
<u>Install Sequelize- </u>a library that allow us to connect via Node.js to our MySQL DB and operate in it, without writing SQL queries (handle it with JS syntax and operate in it as if we are handling objects).

In the root of the project

>> npm install –save sequelize

>> npm install --save sequelize-cli

>>npx sequelize init ➔ will create folders and files in your project to hanle the SQL DB



```
  × {} package.json                M
∨ DERMASQLASSIGNMENT
  > config                         ●
  > migrations
  > models                         ●
  > node_modules
  > seeders
  .gitignore
  ~$Derma.docx
  ~WRL3860.tmp
  Derma.docx                       M
  {} package-lock.json             M
  {} package.json                  M
  JS server.js
```

Install .env :

dotenv - an npm package that loads environment variables from a `.env` file into `process.env`. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

Lets install it:

At root level-

> npm i dotenv

CREATE A NEW .env FILE AT ROOT LEVEL:

IN IT SET TWO ENV VARIABLES, PORT and NODE_ENV

ROOT-> .env ➔

**make sure it's in the .gitignore file as well!!!

Creating the server and app entry point:

Create a new file in the root of the project → app.js

```
const express  = require( 'express' );

const app = express();

app.use(express.json())

app.get('/', (req, res)=>{
    console.log('hi there,you got to / route');
    res.status(200).send('you got to / route');
});

module.exports = app
```

Create a new file in the root of the project → server.js

```
const http = require('http');
const dotenv = require('dotenv')
const app = require('./app');

dotenv.config();
const PORT = process.env.PORT || 3000;
const mode = process.env.NODE_ENV;
const server = http.createServer(app);
server.listen(PORT, ()=>{
    console.log(`you are listenning to PORT: ${PORT}, in ${mode} environment`)
});
```

GIT- version control

Create a **.gitignore** file

We will make sure you add =>

```
# dependencies
node_modules
```

```
# misc.
.env
```

.env ->a file in which we will store some global variables that may contain sensitive information such as API keys we don't want GitHub users to see.

Initializing GitHub Repository:

In the terminal :

We want to be on the root folder ->

>> git init

>> git add . ➔ stage all files!

>> git status ➔ check which files got staged

>> git commit -m 'Project first setup'

- ➔ Go to GitHub and open a new reposetory

>> git remote add origin "repo URL…"

>>git push -u origin master

Routes & controllers

Create folder at root level → controllers

Create userController.js file

Create folder at root level→ routes.

Create userRoutes.js file.

We will separate the request's route and functionality into 2 separate files,

To fit mvc structure.

We will come back here later…

<u>for now lets install MySQL into our project using mysql2 npm package:</u>

https://www.npmjs.com/package/mysql2

terminal at root level: >>npm i –save mysql2

now we will create a user model, it will include this data: userId, name, email, password.

This model will correspond with the user table which will be formatted with the same columns.

>>npx sequelize model:generate –name User –attributes name:string, email:string, password:string

This will create a user.js model file in model folder and a migration file which allows us to add more restriction for each column.

Id column is auto generated.

Connect our application to a DB:

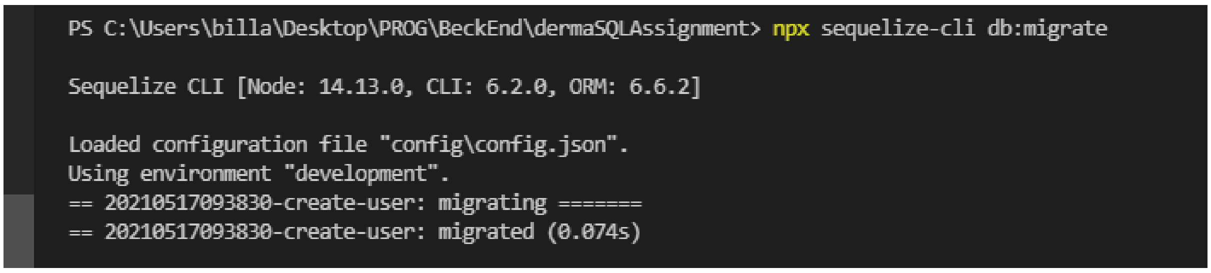
Our DB name is: database_development

Go to config folder→ config.js →change the development DB to our local DB name:

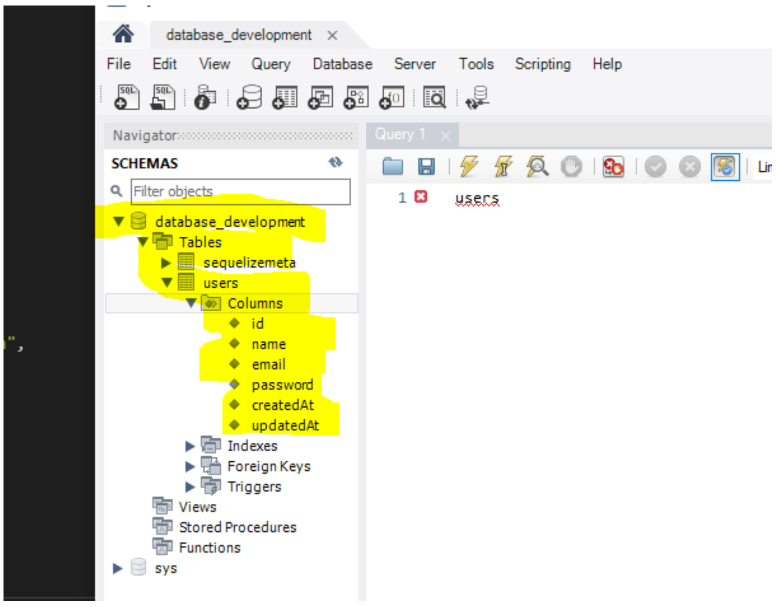
```
{
  "development": {
    "username": "db_user",
    "password": “”,
    "database": " database_development”,
    "host": "localhost",
    "dialect": "mysql"
  },
  "test": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "production": {
    "username": "root",
    "password": null,
    "database": "database_production",
    "host": "127.0.0.1",
    "dialect": "mysql"
  }
}
```

In our terminal at root level: >>npx sequelize db:migrate

This will create tables in our local db according to the models we previously created!



```
PS C:\Users\billa\Desktop\PROG\BeckEnd\dermaSQLAssignment> npx sequelize-cli db:migrate

Sequelize CLI [Node: 14.13.0, CLI: 6.2.0, ORM: 6.6.2]

Loaded configuration file "config\config.json".
Using environment "development".
== 20210517093830-create-user: migrating =======
== 20210517093830-create-user: migrated (0.074s)
```



```
database_development
  Tables
    sequelizemeta
    users
      Columns
        id
        name
        email
        password
        createdAt
        updatedAt
      Indexes
      Foreign Keys
      Triggers
  Views
  Stored Procedures
  Functions
sys
```

```
1 users
```

Great we have created the tables as we can see in workbench!

User registration

User Routes & controllers

app.js refactoring →

```
const express  = require( 'express' );

const app = express();

const userRoutes = require('./routes/userRoutes');

app.use(express.json())

app.use('/api/users/', userRoutes)

// app.get('/', (req, res)=>{
//     console.log('hi there,you got to / route');
//     res.status(200).send('you got to / route');
// });

module.exports = app;
```

User routes:

For now, let’s start working on the functionality of registering a new user in our DB:

```
const express = require ('express');

const router = express.Router()// api/users/

const { registerUser } = require ('../controllers/userControllers.js');

router.route('/register').post(registerUser);

module.exports = router;
```

user controllers:

```
const models = require('../models');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');


const registerUser = (req, res)=>{

    const {name, email, password} = req.body;

    //if userexists in DB...
    models.User.findOne({ where: { email: email } }).then( result =>{
        if(result){ //if user exists in db
            res.status(409).json({
                massage:'Cannot register, email exists in DB'
            });

        }else{ //user does not exists in db , proceed ...

            const newUser = {
                name,
                email,
                password
            };

            //hash the password before adding the user to the DB
            const salt = bcrypt.genSaltSync(10);
            newUser.password =  bcrypt.hashSync(password, salt);

            //email does not exists in the db, we shall now create it!
            models.User.create(newUser).then(result => {
                res.status(201).json({
                   message: 'user created!'
                })
            }).catch(error =>{
                res.status(400).json({
                    message: 'User could not be created- invalid data'
                })
            })
        }
        }).catch(error => {
            res.status(500).json({
                message: 'Cannot register, internal server error'
            })
        })
}
module.exports = {
    registerUser: registerUser,
}
```

**bcrypt & JWT & nodemon

In order to store the password of the user as a hashed string in the DB we will use this package!

Plus we will use JWT to create tokens- this will allow us to create a token based authentication api.

Plus lets install a hot reloader- nodemon.

>> npm i --save bcryptjs jsonwebtoken nodemon

Setting up nodemon→ in package.json, under scripts add:

```
"start": "nodemon server.js",
```

From now all you need to do to is to run the server once by : >>npm start

User login

User Routes & controllers

Controller : add this code…

```
const userLogin = (req, res)=>{
    const {name, email, password} = req.body;

    models.User.findOne({where:{email:email }}).then(user =>{
        if(!user){
            res.status(401).json({
                message: "Email does not exists in DB"
            })
        }else{
            //email exists, now compare the password the password we recieved
throug the body VS
            //the un-hashed password from the DB corrspondes with this email.
            bcrypt.compare(password, user.password, function(err, result){
                if(result){ //if they match
                    //provide the user a Token which includes a payload with d
ata from the DB, a secret and an exp date:
                    const token = jwt.sign({
                        email: user.email,
                        userId: user.id
                        },
                        process.env.JWT_SECRET,
                        {expiresIn:'30d'}
                    );
                    //provide the user with the token
                    res.status(200).json({
                        token: token
                    })

                }else{
                    res.status(400).json({
                        message: "wrong password..."
                    })

                }
            })
        }
    }).catch(error =>{
        res.status(500).json({
            message: "Something went wrong..."
        })
    });

}



module.exports = {
    registerUser: registerUser,
    userLogin: userLogin
}
```

User Route:

```
const express = require ('express');

const router = express.Router()// api/users/

const { registerUser, userLogin} = require ('../controllers/userControllers.js
');

router.route('/register').post(registerUser);

router.route('/login').post(userLogin)

module.exports = router;
```

.env → we should also configurate the secret sentence:

JWT_SECRET = justasecretsentence

Protect Routes

allow only authenticated users to enter specific routes.

Create a new folder→ middleware → and a new file in it authMiddleware.js :

```
const jwt = require ('jsonwebtoken');
const models = require('../models');

const authUser = async (req,res,next) =>{
    //if the request headers doesn't include the proper data....
    if(req.headers.authorization &&
        req.headers.authorization.startsWith('Bearer')){
        try{
            const token = req.headers.authorization.split( ' ' )[1];   //we ne
ed to extract from the headers only the token string excluding the space and '
Bearer' so we will split the string at the space between the token and Bearer
and take the secont part (the string).
            const decoded = jwt.verify(token, process.env.JWT_SECRET);

            //so now we can gain access to this user's id as it appears in the
 DB
            req.userData = await  models.User.findOne({ where: { id: decoded.i
d } }) //user data is stored in this variable!
            next() //passed the middleware access granted!
        }catch(error){
                    res.status(401).json({
                    message: 'Not authrized, token faild verification',
                    error: error
                    })
        }
    }else{
        res.status(401).json({
            message: 'faild verification'
            })
    }

}

module.exports = {
    authUser: authUser
}
```

<u>Implementing protected routes:</u>

In userRoutes we will create a /profile route, where only authenticated user can approach this route to get his data and update it. (.get / .put methods).

When updating data , we will generate a new token and send it off the user for him to store it in his local storage.

The routes:

```
const express = require ('express');


const router = express.Router()// api/users/

const { registerUser, userLogin, getUserProfile, updateUserProfile} = require
('../controllers/userControllers.js');
const {authUser} = require('../middleware/authMiddleware.js')  //routes protec
ting middleware

router.route('/register').post(registerUser);

router.route('/login').post(userLogin);

router.route('/profile').get(authUser, getUserProfile).put(authUser,updateUser
Profile) //these routes can be accessed only if user is authenticated!

module.exports = router;
```

the controllers we added:

```
//this is a GET request to the protected route: api/users/profile
//return a specific user data.
const getUserProfile = async (req,res)=>{

    //in authMiddleware.js we store all user data that's passed the authentica
tion and authorization proccess in req.userData
    const user = await models.User.findOne({where: {id: req.userData.id}})  //
 req.userData.id refers to the logged in user

   if(user){
     res.json({
        id: user.id,
        name: user.name,
        email: user.email,
        password: user.password
     })
   }else{

       res.status(404).json({
            message:'user not found'
       })
   }

 }


//this is a PUT request to the protected route: api/users/profile
//enables edit the specific user's data.
const updateUserProfile = async (req,res)=>{

   //in authMiddleware.js we store all user data that's passed the authenticat
ion and authorization proccess in req.user (excluding his password)
   const user = await models.User.findOne({where: {id: req.userData.id}})// re
q.userData.id refers to the logged in user

   if(user){
         user.name = req.body.name || user.name
         user.email = req.body.email || user.email
         if(req.body.password){
            //hash the password before adding the user to the DB
            const salt = bcrypt.genSaltSync(10);
            user.password = bcrypt.hashSync(req.body.password, salt);

         }

         try{

             const updatedUser = await models.User.update( {
                id: user.id,
                name: user.name,
                email:  user.email,
                password: user.password
                },
                 {where: {id: req.userData.id}}
               );
              //generate a token with the new data! and we will send it to the
 user! in the response
               const token = jwt.sign({
                 id: user.id,
                 name: user.name,
                 email: user.email,
                 password: user.password
                 },
                 process.env.JWT_SECRET,
                 {expiresIn:'30d'}
             );
              console.log(updatedUser)
              res.json({
                id: user.id,
                name: user.name,
                email: user.email,
                password: user.password,

token: token
              })
         }catch(error){
            res.status(500).json({
                message: 'could not update user'
            })
         }


   }else{
       res.status(404).json({
           message: 'user not found'
       })

   }

 }



module.exports = {
    registerUser: registerUser,
    userLogin: userLogin,
    getUserProfile: getUserProfile,
    updateUserProfile : updateUserProfile
```

And that's it more or less,

We can use this server to register (while storing a hashed password in the DB), login (while generating & sending a token in the respons).

With the token being added to the header we gain access to protected routes where we can get access to our data and manipulate it (while generating & sending a new token).

It was fun learning and playing with MySQL DB.

There are endless features I can add to make this app a more production level app such as:

- **Adding an admin key in the User model so maybe he will have a broader access to routes then a normal user.**
- **Adding more routes.**
- **Adding data validations to body inputs.**

If you have any questions feel free to contact me !

Thanks for your time,

Ori S.