API Documentation

Configuration

> In this section we will discuss all the steps that enabled us to set up our development environment and the tools used.

Tools used :

- **NPM**
- **Azure Cosmos DB**
- **Visual Studio Code**
- **Docker**

Setting up the environement

```
const cors = require('cors');
const mongoose = require('mongoose');

const app = express();
app.use(cors({ origin: true, credentials: true }));

// Use the .env file to declare DB_CONNECT
mongoose.connect(process.env.DB_CONNECT, {useUnifiedTopology: true,
useNewUrlParser: true, useCreateIndex: true });
const connection = mongoose.connection;
connection.once('open', () => {
    console.log("MongoDB database connection established successfully");
});
```

With the dotenv package, we are going to configure an env variable that will allow us to pass secret parameters like credentials to make everything secure, we are going to use Mongoose because it will allow us to create models that will help us create our collections.

```
//Configure port
const port = process.env.PORT || 5000;

app.listen(port, () => {
console.log(`Server is running on port: ${port}`);
});
```

And finally we are going to tell our server to listen to a certain port in order to run the server and make it accessible through our browser.

Routing System

For that we need to create a folder routes which will map all the routes in order to make them accessible. For that we create a Router:

```
// Import routes
const Router = require('./routes/routes');
```

And we set-up all the routes in order to link with our controllers :

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

//Authentication
const authRoute = require('../controller/auth');
router.use('/api/user', authRoute);

module.exports = router;
```

Models

Models are fancy constructors compiled from Schema definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

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

const userSchema = new mongoose.Schema({
    name:{type: String, required: true, min:6},
    email:{type: String, required: true, max:255, min:10},
    password:{type: String, required: true, max:1024, min:8},
    date:{type: Date, default: Date.now}
});

module.exports = mongoose.model('User', userSchema);
```

We create a schema for users (kind of object) with attributes we want to store on our collection.

Controller

> It is the center of our operations, it will implement every feature

We will call the function `Router()` of express which will perform different types of requests, it will take into parameters our route, middlewares, the request and a response:

```
router.get('/',verify , (req, res) => {
     res.send(req.user);
});
```

Functionnalities

Authentication

Registration

> From User schema, we can create and manipulate users we have on our database with creating a new object from it.

That's why we require it in our controller, once users are going to register their credentials will pass our validation filter (using Joi) using an object :

```
const schema = Joi.object({
    name: Joi.string().min(6).required(),
    email: Joi.string().min(10).max(255)required().email(),
    password: Joi.string().min(8)required()
});
```

We automatically send a message error if the inputs didn't pass the validation, then we check if the email exists with the request :

```
User.findOne({email: req.body.email});
```

Then we hash the password in order to crypt our data using Bcrypt

And finally we can save our user within a try catch instruction

```
const savedUser = await user.save();
```

Log-in

> Our login system uses token-based solution, since user has a valid token he can carry out transactions within our website until the token is destroyed

Once user logging in he provides credentials, using validator we first check if the e-mail address is correct.

```
const user = await User.findOne({email: req.body.email});
    if(!user) {
        console.log("Email is wrong");
        return res.status(400).send("Email is wrong!");
}
```

Then we compare the password introduced with the one related to the user's e-mail that we have collected using a compare method from the bcrypt library :

```
bcrypt.compare(req.body.password, user.password);
```

if everything is correct we sign the user id with JsonWebToken library sign method, giving it the user and a secret paraphrase that will be our secret key, and everything is stored on a token variable that will be put onto user local storage.

Middlewares

> A middleware in our case is a system allowing to redirect an unauthorized user so that he cannot have access to some routes as he has no token, he is not identified.

```
function(req, res, next){
    const token = req.header('auth-token');
    if(!token) return res.status(401).send('Access denied');

    try {
        const verified = jwt.verify(token, process.env.TOKEN_SECRET);
        req.user = verified;
        next();
    } catch (err) {
        res.status(400).send('Invalid Token');
}
```

Posts

> In order to automate it, I chosed to make the interactions with the posts dynamic so that we can add and remove posts without it being a problem

Operations to create, delete, update and get posts.

Using Multer to store an image on a an upload directory and saving path to the database:

```
const storage = multer.diskStorage({
     destination: function(req, file, cb) {
```

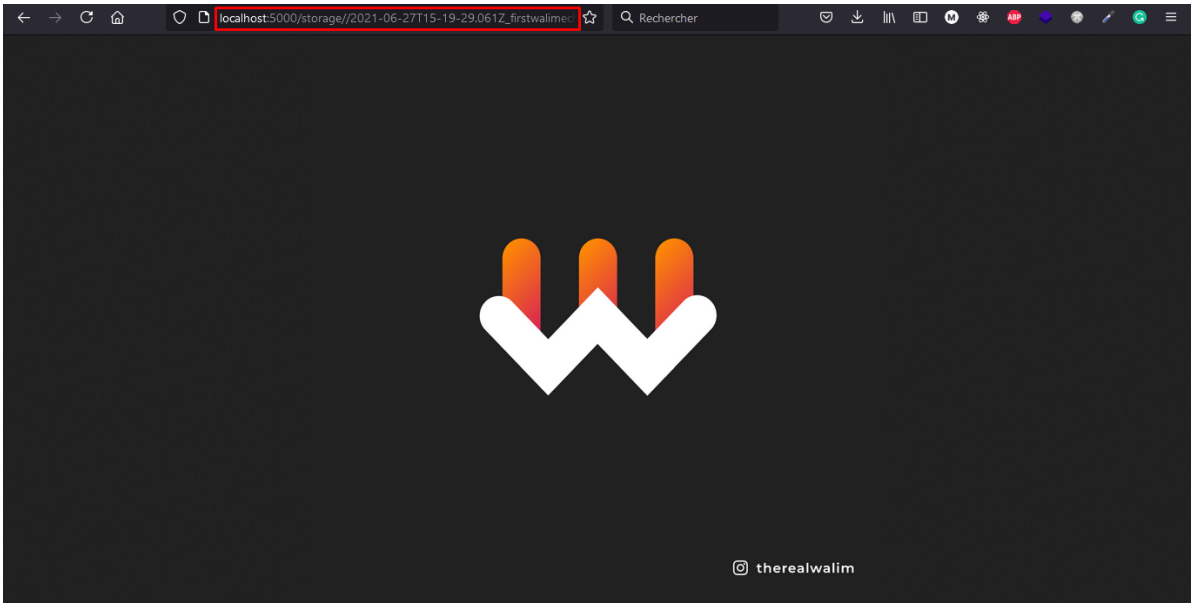
```
        cb(null, './storage/');
      },
      filename: function(req, file, cb) {
        cb(null, new Date().toISOString().replace(/:/g, "-") + "_" +
file.originalname);
      }
    });

    const fileFilter = (req, file, cb) => {
      // reject a file
      if (file.mimetype === 'image/jpeg' || file.mimetype === 'image/png') {
        cb(null, true);
      } else {
        cb(null, false);
      }
    };

    const store = multer({
      storage: storage,
      limits: {
        fileSize: 1024 * 1024 * 5
      },
      fileFilter: fileFilter
    });
```

Because it's a bad practice to save files directly on a database, the best way is to store it's path and DB and create a file to store it directly on our server.

To access the photos you'll have to pick the link you got after submitting :

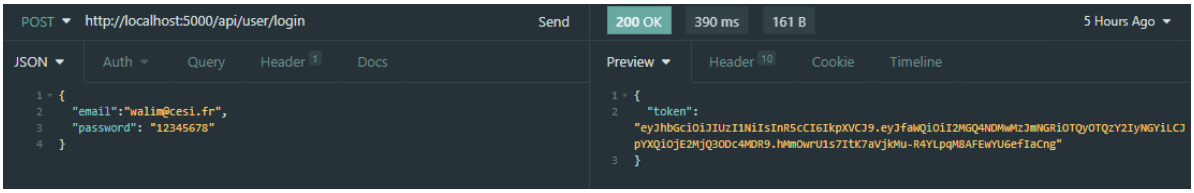

```
{
  "post": {
    "_id": "60d8970115e6863718f60bfd",
    "description": "My name is Walim and I like developing new applications using new
technologies",
    "content": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.",
    "created_by": "60d843032f4db942943cb24f",
    "thumbnail": "storage\\2021-06-27T15-19-29.061Z_firstwalimechaib.png",
    "created_at": "2021-06-27T15:19:29.090Z"
  }
}
```

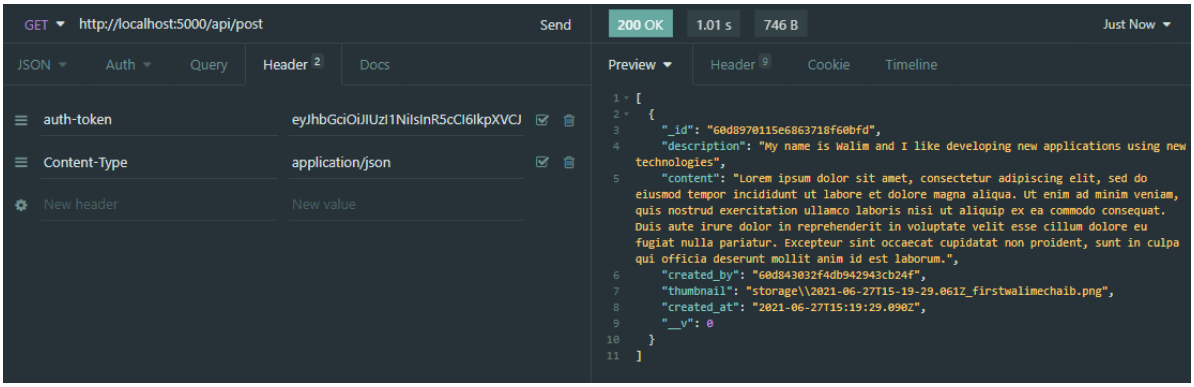

And put it to the root of the project like this :



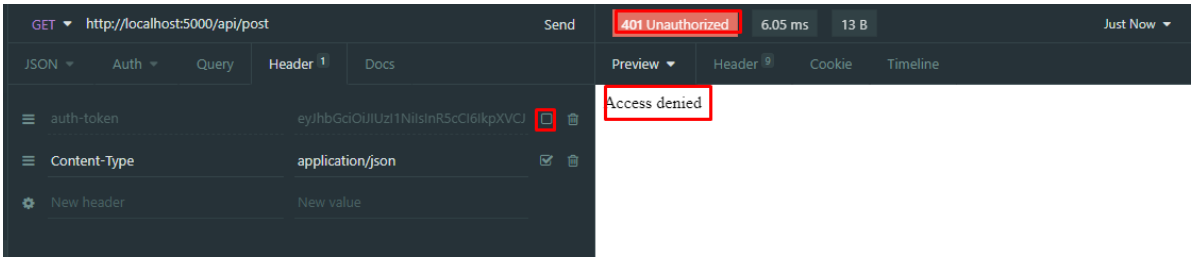
Token

In order to access posts functionalities, you'll have to add as header the token you got from login :





If you try to get posts without token (as an unauthenticated user) the access will be denied for you request:



Dockerfile

```
#  Dockerfile for Node Express Backend
FROM node:10.16-alpine
```

```
# Create App Directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install Dependencies in Silent Mode
COPY package*.json ./

RUN npm install --silent

# Copy app source code
COPY . .

# Exports By Defining Port
EXPOSE 5000

CMD ["npm","start"]
```

I used Typora to wrote this small report using Typora, it's a good tool to write reports displaying in live mode our markdown results

Walim ECHAIB ~ 2021