Lab 5 – Expanding the API

API Authentication

Introduction

There are many different methods for online user authentication. For RESTful APIs, we have seen that we must use a method that does not require sessions because a **RESTful API server does not maintain any session data about the client**. Therefore, **the client needs to send the auth data with every request**.

Basic Auth

One of the easiest methods to use is called **Basic Auth**. Basic auth involves sending a username and password coded inside a header. Note that this differs from sending the username and password inside the request body, either encrypted or not. The latter is not considered Basic Auth.

On a client such as a React App, the API request code will thus need to send the username and password with every request it makes, so the user authentication data is needed to be kept somewhere. Typically, this is done using **localStorage** in the browser, and we will look at this when covering in the Full-stack Web Development.

To send the auth data from the client to the server using the Basic Auth strategy format, the client headers sent in the request would look something like this:

```
headers: {
  'Accept': 'application/json',
  'Content-Type': 'application/json',
  'Authorization': 'Basic ' + window.btoa(username + ':' + password)
}
```

This tells the API that the client is requesting JSON and is sending JSON in the body (if any), and in line 4, we add the '**Authorization**' header. Its value indicates it will be Basic Auth and contains a base64 version of the username and password. This is created using the two-way base64 encoding function **btoa** (B to A), which will be discussed in the **Front-end part**.

The username and password are two variables you need to capture from a user. Once they are stored, that's it for the client. The application needs to remember that this authorization header should be included with every request it makes to the API server (for protected routes).

Authenticating on the Server

On the server, the actual authentication will take place. The server will need to do the following:

- Protect certain resources so only authenticated users can access them.
- Extract auth data from the request, whether they exist in the header as in Basic Auth (and some other strategies) or by using third-party auth strategies such as Facebook or Google.

- Validate whether the user is authentic by checking against the user data in the database – i.e. check the stored version of the credentials for this user.
- Grant or deny access to the resource by allowing or disallowing the next route handler in the sequence to be run – in practice, if access is denied, then the response is sent at this point.

Implementation with Basic Authentication

We use the existing project from Lab 4 to implement the login mechanism. In this stage, we also use the sample user data within the code to simplify the flow. **Note that we should always be mindful not to store the password with plain text anywhere in the program.**

Sample data

Let's create a sample user data in the **users.ts** under **models** folder:

```
interface User {
    id: number;
    username: string;
    password: string;
    firstname: string;
    lastname: string;
}

export const users: User[] = [
    {id: 1, username: 'admin', password: 'admin', firstname: 'Admin', lastname:
'Admin'},
    {id: 2, username: 'picard', password: '1234567890', firstname: 'Picard',
lastname: 'Jean-Luc'}
];
```

Set up the strategy in the controller

Next, we set up our first auth strategy. In the '**controllers**' folder of your project, add a file called '**authMiddleware.ts**'.

Inside it, we begin by implementing the following:

- import the auth strategy we want to use (basic auth)
- import the users' model so we can search for users who are trying to authenticate
- define a way to compare the password supplied with the one we have in persistent storage

To do these, add the following code to the '**authMiddleware.ts**' file:

```
import { Context } from "koa";
import { users } from "../models/users";
import { validationResults } from 'koa-req-validation';

export const basicAuthMiddleWare = async (ctx: Context, next: any) => {
    const authHeader = ctx.request.headers.authorization;
```

```
    if (!authHeader || !authHeader.startsWith('Basic ')) {
        ctx.status = 401;
        ctx.headers['WWW-Authenticate'] = 'Basic realm="Secure Area"';
        ctx.body = { msg: 'Authorization required' };
    } else {
        const auth = Buffer.from(authHeader.split(' ')[1], 'base64').toString();
        const [username, password] = auth.split(':');
        console.log(`${username} is trying to access`);
        if (validationCredentials(username, password)) {
            ctx.state.user = { username };
        } else {
            ctx.status = 401;
            ctx.body = { msg: 'Authorization failed' };
        }

    }
    await next();
}

const validationCredentials = (username: string, password: string):boolean => {
    let result = false;

    users.forEach((user) => {
        if (user.username === username && user.password === password) {
            result = true;
        }
    });
    return result;
}
```

We provide the **`BasicStrategy`** with a way of checking whether the user making the request is valid. To do this, we check the header to see if the request authorisation header is included. If there is no, a status code 401 will be returned.

Once the user is found in the database, the API will add the username in the content state. Otherwise, a status code 401 will be responded to.

Set up a route handler that uses our strategy to authenticate API requests

Now the basic auth strategy is defined and exported from the '**controllers/authMiddleware.ts**' module. We need to let the API use it for authentication purposes. As with all other route handlers

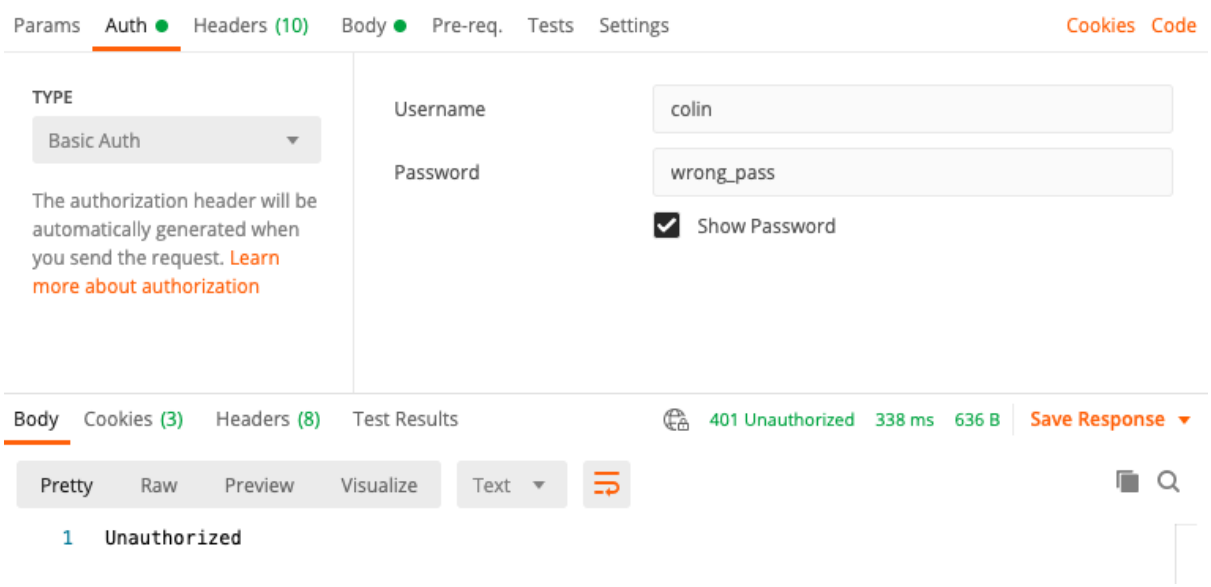
we have defined, we need an async function that takes **`(ctx, next)`** arguments and runs our strategy on the request in the **`ctx`** object.

Create a new **routes/users.ts** add the following code to handler authentication middleware in the login procedure:

```
import Router , { RouterContext } from 'koa-router';
import { users } from '../models/users';
import { basicAuthMiddleWare } from '../controllers/authMiddleware';
const router: Router = new Router({prefix: '/api/v1/users'});

const getUser = async(ctx: RouterContext, next: any) => {
    if(ctx.state.user===undefined){
        ctx.status = 401;
        ctx.body = {msg: 'Authorization failed'};
    } else {
        users.forEach((user) => {
            if(user.username===ctx.state.user.username){
                ctx.body = user;
            }
        });
    }

    await next();
}

router.get('/auth', basicAuthMiddleWare, getUser);

export { router };
```

Try accessing the public/protected endpoints:

Now, use Postman to test the endpoints. You can add authentication headers to a request sent from Postman using the 'Auth' tab under the URI address bar:



You have now protected an important/private endpoint using Basic Authentication.

Note the following:

1. Depending on the API's use, some API endpoints will not need any authentication. For example, no auth is required to create a new user if you want public registration. If you would like your cars to be publicly readable, then no auth is required for reading cars
2. Many different authentication methods exist, such as JSON Web Token (JWT), OAuth, HMAC and others. You will be required to investigate them independently if you wish to implement one (or more) for your coursework APIs. All can be achieved with exactly the same 'strategy' pattern as used for Basic auth above.

Handling MongoDB with Official Framework

In this stage, we will test to connect the API to the MongoDB (**Framework required: mongodb**). If you are not familiar with MongoDB, follow the instructions to create a **blogdb** database in MongoDB.

Define a module to connect the API to the database

To connect to the database to make use with CRUD operations, create a new file in '**helpers/dbhelpers.ts**' file, with the following code: (The URI of the MongoDB can be found in MongoDB Atlas if you are connecting there)

```
import {MongoClient, ServerApiVersion} from 'mongodb';

const uri =
"mongodb+srv://<dbusername>:<dbpassword>@cluster0.mtnbj.mongodb.net/
?retryWrites=true&w=majority";
```

```
const client = new MongoClient(uri,
    {
        serverApi: {
          version: ServerApiVersion.v1,
          strict: true,
          deprecationErrors: true,
        }
    }
);

export const find: any = async (collectionName: string, query: any)
=> {
    let values: any;
    try {
        await client.connect();
        const db = client.db("sampledbs");
        const collection = db.collection(collectionName);
        values = await collection.find(query).toArray();
    } catch (error) {
        values = error;
    } finally {
        await client.close();
    }
    return values;
}
```

Make use of find to return all articles record

Now go back to '**router/articles.ts**' file, update the source code make use with the database connection code (dbhelper.ts) and return all cars record as below:

```
// ... Other framework to be imported.
import * as db from "../helpers/dbhelper.ts";

// ...other codes go here

export const getAll = async (ctx: RouterContext, next: any) => {
  console.log('Return all articles');
  const articles = await db.find('blogdb', {});
  ctx.body = articles;
```

```
  await next();
}
// ...
```

Try again using Postman.

Basic Worksheet Tasks

Update all the source code and ensure you can read and write the data in the database.

Noted that:

All functions you have learnt in MongoDB can be adopted in Typescript, such as "findOne", "insertOne", "updateOne", "aggregate"…