GUIDE B02

Create an Endpoint for The Login

Arranged By
Omar Al-Maktary

INFORMATICS ENGINEERING STUDY PROGRAM
INFORMATION TECHNOLOGY DEPARTMENT
STATE POLYTECHNIC OF MALANG
2023

Contents

Create an Endpoint for The Login

Objectives

1. Students can create a Login endpoint for users of the application using the POST method.
2. Students understand the concept of JSON Web Tokens.
3. Students can create a web page for the login page.

Requirements

Having the correct hardware and software components is essential for ensuring the successful execution of the tasks outlined in this guide. The hardware configuration and software required for completing this guide tasks are as the following:

Hardware Specifications

The minimum hardware specifications for running a NodeJS API application on the Windows operating system and using software such as Postman and Visual Studio Code are the following:

Minimum Requirements

- Processor: Intel Core i3 or equivalent.
- RAM: 4 GB.
- Storage: 500 GB HDD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

Recommended Requirements

- Processor: Intel Core i5 or equivalent.
- RAM: 8 GB or more.
- Storage: 256 GB SSD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

Software required

It is important to have the correct software installed on your system to ensure that the application runs smoothly and meets performance expectations. The software required is as follows:

- Operating System: Windows 10 or later.
- NodeJS: Latest stable version installed.
- Visual Studio Code: Latest stable version installed.
- Postman: Latest stable version is installed.

NPM Packages

- nodemon: Automatically restarts Node application on file changes.
- cross-env: Sets environment variables in a cross-platform way.
- jest: Creates and executes tests.
- jest-expect-message: Enhances Jest assertions with custom error messages.
- jest-image-snapshot: Adds image snapshot testing to Jest.
- puppeteer: Node library to control a headless Chrome or Chromium browser.
- supertest: Makes HTTP queries to the application and checks results.
- dotenv: Simplifies management of environment variables.
- express: NodeJS framework for creating apps with routing and middleware.
- ejs: Embedded JavaScript templating.
- express-ejs-layouts: Layout support for EJS in Express.
- mongoose: MongoDB object modeling library for NodeJS.
- bcryptjs: NodeJS library for hashing passwords with the bcrypt algorithm.
- cookie-parser: Parses cookies attached to the incoming HTTP requests.
- cors: Middleware for enabling Cross-Origin Resource Sharing (CORS) in Express.
- express-unless: Defining exceptions to other middleware functions in Express.
- jsonwebtoken: Manage authentication by creating and verifying tokens.

Resource

- Documents: Guide B02
- Tests: api/testB02.test.js, web/testB02.test.js

Task Description

Students understand how to create an endpoint for logging in users. This endpoint will return the user data and a token for accessing routes that only registered users can use. Students should also understand how to create a web interface that handles the login process. This process will save the access token in a browser cookie which will be used to authenticate users' access.

Start Coding

To create an endpoint for users' login process, students should understand the following table which outlines the structure and goals of the endpoint.

POST "/api/v1/login" ENDPOINT STRUCTURE			
API Endpoint Path	*Request Method*	*Response Format*	*Description*
"/api/v1/login"	POST	JSON	Authenticate the user and return an access token.
Request Parameters			
Parameter	*Type*		*Description*
"username"	String		The user's email or username.
"password"	String		The user's password.
Response Parameters			
Parameter	*Type*		*Description*
"user"	Object		An object for the user data without the password.
"token"	String		A string for the access token.
"message"	String		A message indicating the status of the response.
Success Responses			
HTTP Status Code	*Response*		

200	{
	" user": {
	"name": "John Doe",
	"username": "johndoe",
	"email": "johndoe@gmail.com",
	"createdAt": "2023-02-20T07:32:14.786Z",
	"updatedAt": "2023-02-20T07:32:14.786Z",
	"id": "6424370fe2a9f3e77c1573ee"
	},
	"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..... ",
	"message": "Logged In User Successfully"		

	}
Error Responses	
HTTP Status Code	***Response***
400	{ "message": "Incorrect Username or Password" }
400	{ValidationError}
500	{ error object }

Table 1 POST "/api/v1/login" ENDPOINT STRUCTURE

Follow the steps below to complete the code for this guide document:

1. In the "auth.service.js" file, import the library "bcryptjs" in a constant named "bcrypt".
2. Copy and complete the following code in the "auth.service.js".

```
async function login(username, password) {
  // Find the user by username or email using the findOne() method
  // Use the $or operator to search for multiple fields
  // $or: [{ username: username }, { email: username }]
  const user = // Write your code here...
  // Check if the user exists and the password is correct
  if (user && bcrypt.compareSync(password, user.password)) {
    // Generate access token using the generateAccessToken() method
    const token = // Write your code here...
    // Return user using the toJSON() method,
    // token and message as an object {}
    return {
      // Write your code here...
    };
    // If the user does not exist or the password is incorrect,
    // throw an error
  } else throw "Incorrect Username or Password";
}
```

Figure 1 Login Service Function Code

3. Export the "login" function at the end of the "auth.service.js" file.
4. In the "controllers/api/auth.controller.js" file, copy and complete the following code.

```
function login(req, res, next) {
  // Validate request body fields using the validateData function
  // The data required is a username and password
  const requiredFields = // Write your code here...
  // validateData(Write your code here...)
```

```
  // Call the login function from the authServices
  // The login function takes two parameters: username and password
  // The login function returns a promise
  // If the promise is resolved, return a 200 status code, and the results
  // If the promise is rejected, call the next function with the error
  // Write your code here...
}
```

Figure 2 Login Controller Function Code

5. Export the "login" function at the end of the "controllers/api/auth.controller.js" file.
6. In the "routes/api/auth.routes.js" file, import the login function from the API controller.
7. Create a new POST route with the "/login" path.
8. Finally, in the "app.js" file, update the "authenticateToken.unless" function to include the following route "{ url: "/api/v1/login", methods: ["POST"] }"

Running The API Application

For this guide and development purposes the command "npm run dev" is used to execute the command "nodemon server.js" which will run the "server.js" using the nodemon package. This package allows the server to reload if any changes occur in the code of the application.

Run the development command "npm run dev" in the terminal and notice the console message.

Testing The API Application

In this section, several tests in different ways will be explored to verify the results of the student's work on this document.

Using Postman

To test results from this guide on Postman, follow these steps:

1. In the "auth-experiment" collection, create a POST request with the name "POST /api/v1/login".
2. Make sure that the environment created is being used by selecting it from the top right option and then fill in the URL in the POST request as the following: "{{protocol}}{{host}}{{port}}{{version}}/login"

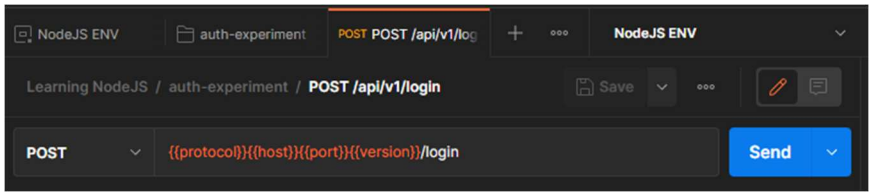

Figure 3 Postman Request Configuration Bar

3. In the request body, choose the "x-www-form-urlencoded" and fill in the following parameters:
    a. username
    b. password
4. Click "Send" and wait for the response. Postman should show results as the following if everything is working correctly:

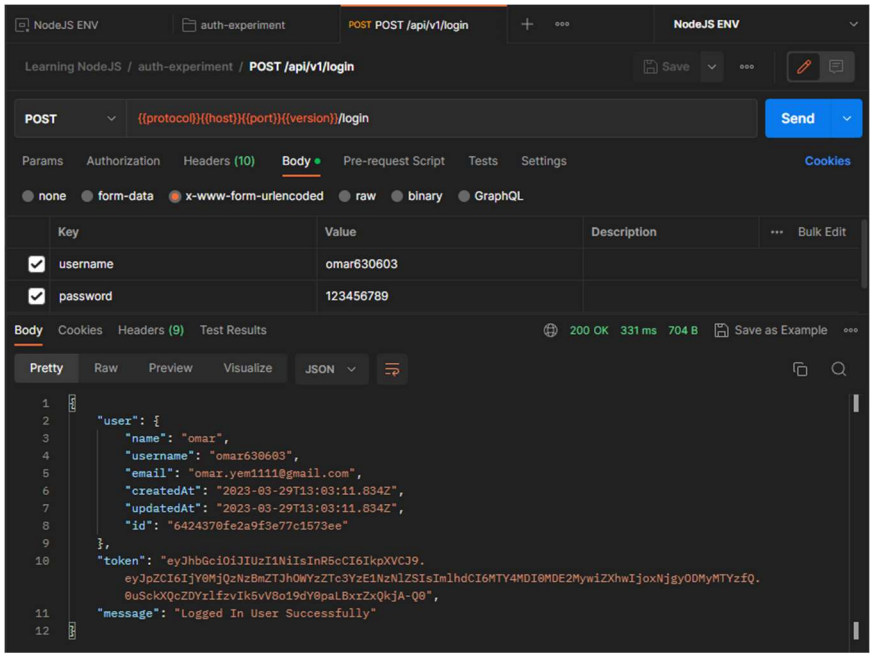

Figure 4 Postman Request Results

Running The API Test File

Note: Sometimes the test will have an error of time limit, try to re-run the test or increase the testTimeout in scripts of the "package.json" file.

Verify results by following these steps:

1. Copy the file "testB02.test.js" from the "api" folder within the "tests" folder for this material to the "tests/api" folder of your project base directory.
1. Run the fill in the VSCode integrated terminal by running this command "npm run api-testB02" and then wait for results.

2. If everything is correct and working well the results in the terminal should look as the following:

```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/auth-experiment
$ npm run api-testB02

> auth-experiment@1.0.0 api-testB02
> cross-env NODE_ENV=test jest -i tests/api/testB02.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testB02.test.js:31:15)

 PASS  tests/api/testB02.test.js
  Testing POST /api/v1/login
    √ should register a new user (638 ms)
    √ should login a user with username (138 ms)
    √ should login a user with email (130 ms)
    √ should not login a user with wrong username (96 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        3.814 s, estimated 4 s
Ran all test suites matching /tests\\api\\testB02.test.js/i.
```

Figure 5 Successful Test Results

3. If the test failed and it shows an error similar to the following figure, the error shows feedback for the cause of the error:

```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/auth-experiment
$ npm run api-testB02

> auth-experiment@1.0.0 api-testB02
> cross-env NODE_ENV=test jest -i tests/api/testB02.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testB02.test.js:31:15)

 FAIL  tests/api/testB02.test.js
  Testing POST /api/v1/login
    √ should register a new user (572 ms)
    √ should login a user with username (144 ms)
    √ should login a user with email (136 ms)
    × should not login a user with wrong username (57 ms)

  ● Testing POST /api/v1/login › should not login a user with wrong username

    The message returned in the response "Incorrect Username" is not correct, it should be "Invalid Username or Password",
 change the response body in the function that handles the POST /api/v1/login route

    expect(received).toBe(expected) // Object.is equality

    Expected: "Incorrect Username or Password"
    Received: "Incorrect Username"

Test Suites: 1 failed, 1 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        3.759 s, estimated 4 s
Ran all test suites matching /tests\\api\\testB02.test.js/i.
```

Figure 6 Failed Test Results

Try to find out why the test failed and fix it until the test result shows successful results.

Creating The Web Interface

In this section, the web interface for the login page will be created. The same basic endpoint explained in the previous sections will be implemented in a web page that can authenticate users and return an access token.

To start working on the web interface, follow these steps:

1. In the "controllers/web/auth.controller.js" file, copy and complete the following code:

```
function login(req, res, next) {
 if (req.method === "GET") {
  // Return the login page with the title and message
  // The title is "Auth-Experiment | Login"
  // The message is req.body.message
  // The message is used to display errors
  // The Login page is located at web/views/auth/login.ejs
 } else {
  // if the request method is not GET, then it is POST
  // Validate the request body
  // The required fields are "username" and "password"
  const requiredFields = // Write your code here...
  // Call the validateData function to validate the request body
  // If the error is not empty, then set the request method to GET
  // Set the message in the request body to the error
  // Call the login function again with the request, response, and next
  const error = // validateData(req.body, requiredFields);
  if (error !== "") {
   // Write your code here...
  }
  // Call the login function in the authService
  // The username is req.body.username
  // The password is req.body.password
  authServices
   .login(username, password)
   .then((results) => {
    // if the login is successful, then set the cookie with the token
    // The token is results.token and the maxAge is 60 * 60 * 1000
    // Redirect to the /profile route
    // Write your code here...
   })
   .catch((err) => {
```

```
      || if the login is not successful, then set the request method to GET
      || Set the message in the request body to the err
      || Call the login function again with the request, response, and next
    });
  }
}
```

Figure 7 "controllers/web/auth.controller.js" Login Function Code

2. Export the "login" function at the end of the "controllers/web/auth.controller.js" file.
3. In the "routes/web/auth.routes.js" file, import the "login" function from the web controller.
4. Add two new routes as the following.

```
router.get("/login", isLoggedIn, login);
router.post("/login", isLoggedIn, login);
```

Figure 8 "routes/web/auth.routes.js" New Routes

5. In the "app.js" file, update the "authenticateToken.unless" function to include the following route "{ url: "/login", methods: ["POST"] }"
6. In the "web/view/auth" folder create a new file named "login.ejs".
7. In the "login.ejs", copy the following code.

```
<% if (typeof message !== 'undefined') { %>
<div class="alert">
  <p class="message"><%= message %></p>
</div>
<% } %>
<div class="container">
  <h1 class="title">Login</h1>
  <p class="description">Fill the form below to log in</p>
</div>
<div class="container">
  <form action="/login" method="POST">
    <div class="form-group">
      <label for="username">Username or Email</label>
      <input type="text" name="username" id="username" class="form-control" />
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input
        type="password"
        name="password"
```

```
      id="password"
      class="form-control"
     />
   </div>
   <button type="submit" class="btn btn-primary">Login</button>
  </form>
</div>
```

Figure 9 "login.ejs" View Code

Note that this code has a form with an action attribute that request the "/login" route with a POST method. This form has two inputs one for the username or email and the other for the password. There is a button within the form with the "submit" type which when clicked the form will request the route "/login".

Running and Testing The Web Interface

If the application still running from the previous exercise then try to visit the following link http://localhost:8080/login. If the application is not running in the terminal then use the command "npm run dev" to start the app.

If the user is still logged in from the previous material, "Registration", then clear the browser cookies and restart the browser.

Upon visiting the URL, the web interface should look similar to the following:

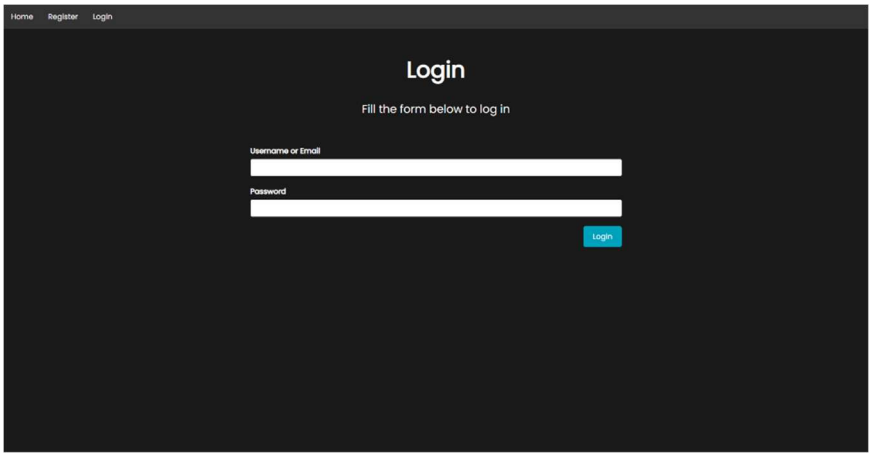

Figure 10 The Login Page

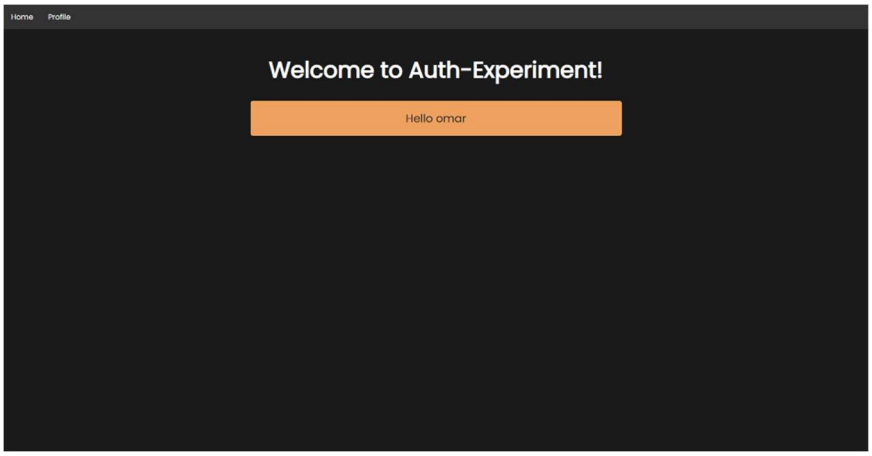

Figure 11 The Home Page After Login

Copy the file “testB02.test.js” and paste it to the folder “/tests/web” in your project directory. After that, run the command “npm run web-testB02” and notice the results.

If everything is correct and working well the results in the terminal should look as the following:

```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/auth-experiment
$ npm run web-testB02

> auth-experiment@1.0.0 web-testB02
> cross-env NODE_ENV=test jest -i tests/web/testB02.test.js --testTimeout=20000

 PASS  tests/web/testB02.test.js (7.699 s)
  Testing the login page
    √ should have the right title (128 ms)
    √ should have a form with 2 inputs and 1 button (114 ms)
    √ should login a user (514 ms)
    √ should have the name of the user in the index page (197 ms)
  Testing the login page image snapshots
    √ matches the expected styling for the login page (2637 ms)
    √ matches the expected styling for the login page with error (683 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   2 passed, 2 total
Time:        7.774 s, estimated 9 s
Ran all test suites matching /tests\\web\\testB02.test.js/i.
```

Figure 12 Successful Web Test Results

```
OMAR@LAPTOP-N1SUCSAB MINGW64 ~/Desktop/auth-experiment
$ npm run web-test002

> auth-experiment@1.0.0 web-test002
> cross-env NODE_ENV=test jest -i tests/web/test002.test.js --testTimeout=20000

 FAIL  tests/web/test002.test.js (7.437 s)
  Testing the login page
    × should have a form with 2 inputs and 1 button (79 ms)
    √ should login a user (328 ms)
    √ should have the name of the user in the index page (235 ms)
  Testing the login page image snapshots
    × matches the expected styling for the login page (2595 ms)
    × matches the expected styling for the login page with error (733 ms)

  ● Testing the login page › should have a form with 2 inputs and 1 button

    The form should contain a label with the text "Username or Email" and the attribute "for" with the value "username" Check the "login.ejs" file in the "web/views/auth" folder to find the form

    expect(received).toEqual(expected) // deep equality

    - Expected  - 1
    + Received  + 1

      Object {
    -   "innerText": "Username or Email",
    +   "innerText": "Username",
      }

  ● Testing the login page image snapshots › matches the expected styling for the login page

    The web styling for the login page is not correct check the file "tests/web/images/__diff_output__/login-page-diff.png" to find the difference

    Expected image to match or be a close match to snapshot but was 0.042851765950520 83% different from snapshot (337 differing pixels).
    See diff for details: tests\web\images\__diff_output__\login-page-diff.png

  ● Testing the login page image snapshots › matches the expected styling for the login page with error

    The web styling for the login page with error is not correct check the file "tests/web/images/__diff_output__/login-page-with-error-diff.png" to find the difference

    Expected image to match or be a close match to snapshot but was 0.042851765950520 83% different from snapshot (337 differing pixels).
    See diff for details: tests\web\images\__diff_output__\login-page-with-error-diff.png

 › 2 snapshots failed.
Snapshot Summary
 › 2 snapshots failed from 1 test suite. Inspect your code changes or run `npm run web-test002 -- -u` to update them.

Test Suites: 1 failed, 1 total
Tests:       3 failed, 3 passed, 6 total
Snapshots:   2 failed, 2 total
```

Figure 13 Failed Web Test Results

If you face a similar error try to figure out the reason for the problem until the test shows successful results.

Results

This document outlines the intended outcomes of the second meeting for the second material on the topic of web programming using NodeJS. Students should be introduced to various topics regarding authentication. Students should learn how to create an API endpoint for authenticating users and returning access tokens. The students should also learn how to create a new web interface to deal with the login process.