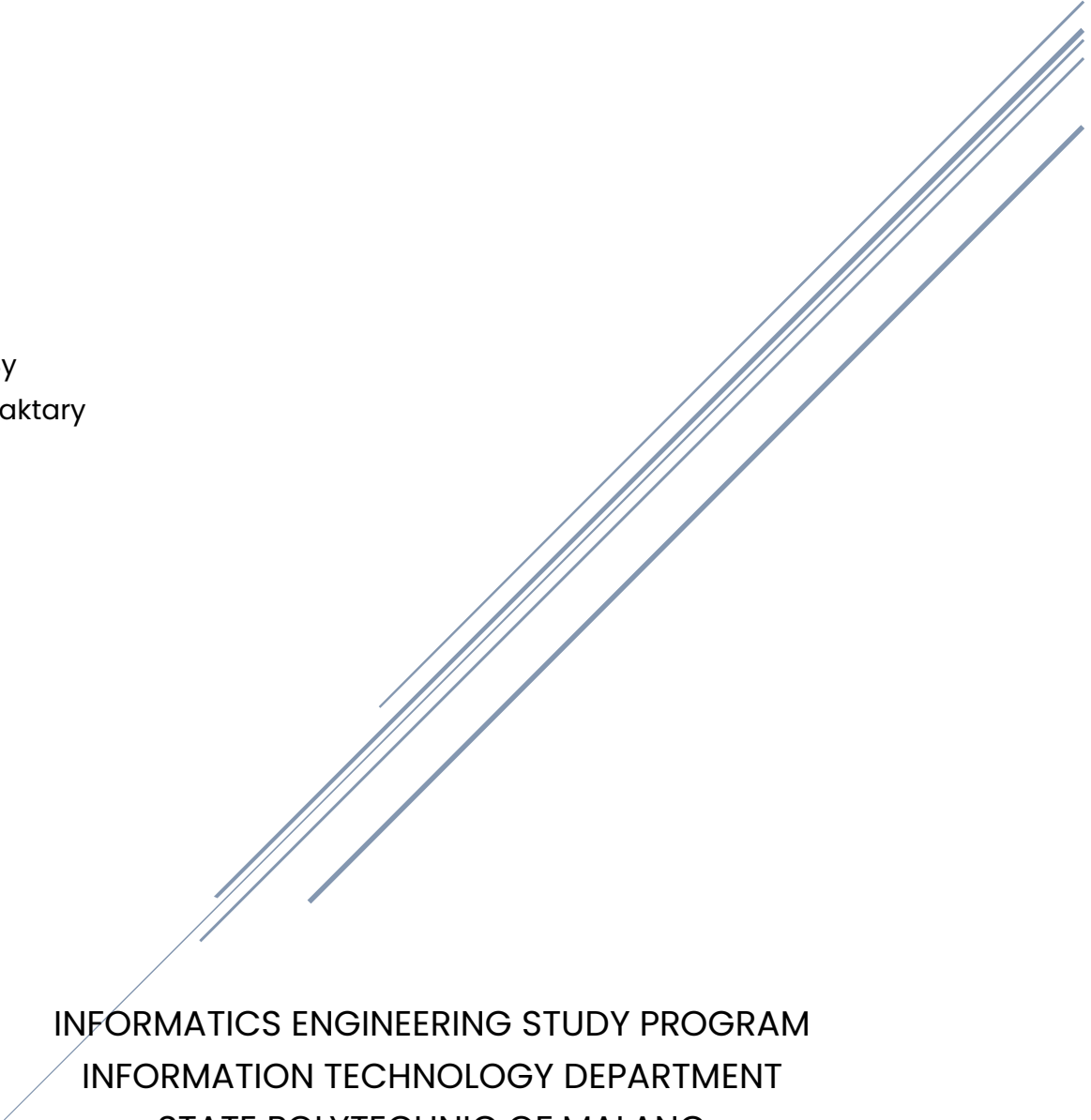


GUIDE B01

Introduction to Authentication on NodeJS

Arranged By
Omar Al-Maktary



INFORMATICS ENGINEERING STUDY PROGRAM
INFORMATION TECHNOLOGY DEPARTMENT
STATE POLYTECHNIC OF MALANG

2023

Contents

Objectives.....	1
Requirements.....	1
Hardware Specifications	1
Minimum Requirements.....	1
Recommended Requirements	1
Software required	2
NPM Packages	2
Resource.....	2
Task Description.....	3
Creating a New Project.....	3
Start Coding.....	7
Running The API Application.....	15
Testing The API Application.....	15
Via Console and Browser.....	15
Using Postman.....	16
Running The API Test File	18
Creating The Web Interface.....	20
Running and Testing The Web Interface.....	26
Results	29

Introduction to Authentication APIs

NodeJS

Objectives

1. Students understand how to create an authentication system in a NodeJS application.
2. Students can create a Register endpoint for users to register in the application using the POST method.
3. Students can create a web page for user registration.

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 B01
- Tests: api/testB01.test.js, web/testB01.test.js, web/images
- Supplements: .env.example, gitignore, main.ejs, main.css

Task Description

Students understand the concept of authentication in a NodeJS application and create a register endpoint and web interface for authentication purposes. The expected output from this document is a simple NodeJS application to handle basic authentication functions such as registration and login. These functions will be used in the API and web interface of the NodeJS application.

Creating a New Project

In this section, students should create a new folder named “auth-experiment” and then open the folder in VSCode from the “File” tab. To Create the NodeJS project follow these steps:

1. Create a new folder named “auth-experiment”.
2. Open the folder in VSCode.
3. Open the VSCode terminal from the Terminal tab by clicking “New Terminal”.
4. Type “npm init” to initialize a Node project and create a new “package.json” file.

In the terminal, answer the following questions as follows:

- a. package name: (auth-experiment): click “Enter” to keep the name of the package the same as the folder name.
- b. version: (1.0.0): click “Enter” to continue.
- c. description: add a description for the project. Example: “Test for express.js and mongoose auth application”.
- d. entry point: (index.js): change this to “server.js” by typing it in the terminal.
- e. test command: click “Enter” to continue.
- f. git repository: click “Enter” to continue.
- g. keywords: add keywords for the project for example “NodeJS, ExpressJS, API, MongoDB, Auth”.
- h. author: add your name
- i. license: (ISC): click “Enter” to continue.
- j. Finally, npm will generate a new “package.json” file and in the terminal, it will show a summary of the file, type “yes” to continue.

```

package.json x
package.json > {} scripts > test
1
2 {
3   "name": "auth-experiment",
4   "version": "1.0.0",
5   "description": "Test for express.js and mongoose auth application",
6   "main": "server.js",
7   "scripts": {
8     "test": "echo \\\"Error: no test specified\\\" && exit 1"
9   },
10  "repository": {
11    "type": "git",
12    "url": "git+https://github.com/Omar630603/auth-experiment.git"
13  },
14  "keywords": [
15    "express.js",
16    "mongoose",
17    "api",
18    "auth"
19  ],
20  "author": "Omar Al-Maktary",
21  "license": "ISC",
22  "bugs": {
23    "url": "https://github.com/Omar630603/auth-experiment/issues"
24  },
25  "homepage": "https://github.com/Omar630603/auth-experiment#readme"
26 }

```

Figure 1 "package.json" File Configuration

5. Install dependencies by following these steps:

There are dependencies and there are development dependencies. A dependency is an object that contains the library, which is required for a project's environment and functionalities. A development dependency is a dependency that is required for development purposes only.

- a. In this guide, development dependencies are:
 - i. nodemon: Reload the application after each change in the code.
 - ii. cross-env: Allows developers to set environment variables.
 - iii. jest: Executing tests.
 - iv. supertest: Send HTTP requests.
 - v. jest-expect-message: Jest with custom error messages.
 - vi. jest-image-snapshot: Adds image snapshot testing to Jest.
 - vii. puppeteer: Headless Chrome or Chromium browser.
 - viii. supertest: Makes HTTP requests to a node app.

To install these packages, type the following command:

```
npm i nodemon cross-env jest supertest jest-expect-message jest-image-snapshot puppeteer supertest --save-dev
```

"npm": a package manager for installing the above dependencies.

"I": is short for install.

"--save": to write the dependencies in the "package.json" file.

"--dev": to specify that those are development dependencies.

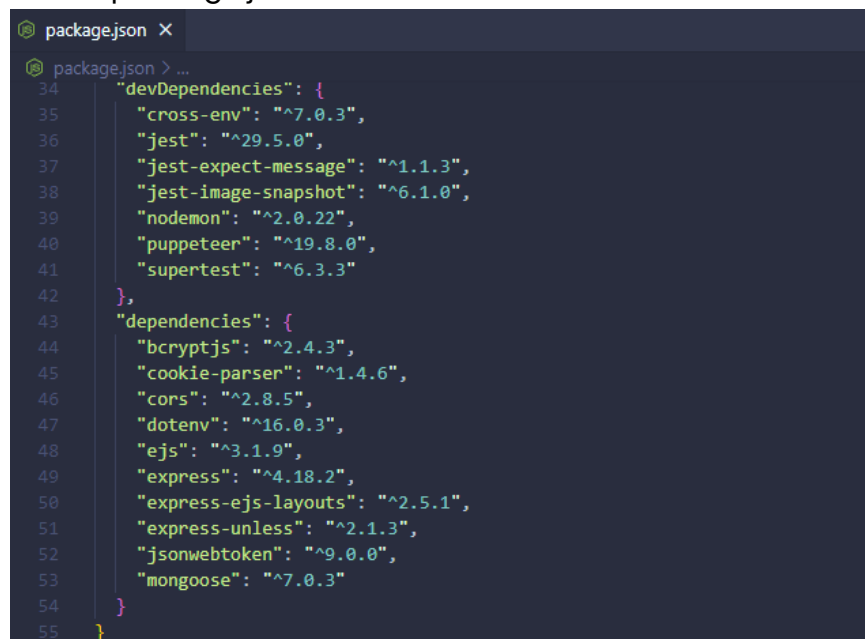
- b. In this guide, production dependencies are:
- i. dotenv: Manage environment variables.
 - ii. express: Node lightweight framework.
 - iii. ejs: Embedded JavaScript templating.
 - iv. express-ejs-layouts: Layout support for EJS in Express.
 - v. mongoose: MongoDB object modeling library for NodeJS.
 - vi. bcryptjs: Library for hashing passwords with the bcrypt algorithm.
 - vii. cookie-parser: Parses cookies for incoming HTTP requests.
 - viii. cors: Middleware for enabling Cross-Origin Resources.
 - ix. express-unless: Define exceptions to other middleware functions.
 - x. jsonwebtoken: Creating and verifying authentication tokens.

To install these packages, type the following command:

```
npm i dotenv express ejs express-ejs-layouts mongoose bcryptjs
cookie-parser cors express-unless jsonwebtoken --save
```

- c. Once the installation is completed, there will be a folder name "node_modules" which will contain all the dependencies files, and a file named "package-lock.json" which will keep information about the dependencies like version and repositories. For this experiment, students must not change those files.

It should be noted that the names of the dependencies are listed in the "package.json" file.



```
34  "devDependencies": {
35    "cross-env": "^7.0.3",
36    "jest": "^29.5.0",
37    "jest-expect-message": "^1.1.3",
38    "jest-image-snapshot": "^6.1.0",
39    "nodemon": "^2.0.22",
40    "puppeteer": "^19.8.0",
41    "supertest": "^6.3.3"
42  },
43  "dependencies": {
44    "bcryptjs": "^2.4.3",
45    "cookie-parser": "^1.4.6",
46    "cors": "^2.8.5",
47    "dotenv": "^16.0.3",
48    "ejs": "^3.1.9",
49    "express": "^4.18.2",
50    "express-ejs-layouts": "^2.5.1",
51    "express-unless": "^2.1.3",
52    "jsonwebtoken": "^9.0.0",
53    "mongoose": "^7.0.3"
54  }
55 }
```

Figure 2 Dependencies Installation

6. Copy the supplement “.env.example” and “gitignore” files to the base directory of the project.

The file “gitignore” is used to prevent specific files or folders, like “node_modules”, from being committed when using version control like git. This file is helpful to not push huge files and secret files such as “.env” which contains values like the URL to the database.

7. Create a new file named “.env”
8. Copy the content of the file “.env.example” to the file “.env”
9. Change the values of the variables inside “.env” to the correct values according to the instructions in the “.env.example” file.

The variables in the “.env” file are:

- a. MONGODB_URI: Change the <username>, <password>, and <cluster_name> according to the link provided by MongoDB Atlas. Change the <database_name> to “auth-experiment”
- b. MONGODB_URI_TEST: Fill in with the same value of the previous variable but the <database_name-test> should be “auth-experiment-test”. This will ensure tests will run in a different database.
- c. PORT: Use port 8080.
- d. JWT_SECRET: Fill in a string value. Example: “SecretWord”.
- e. JWT_EXPIRE: Fill in a period in days for token expiration. Example “30d”.

The file structure of the project should look like this:

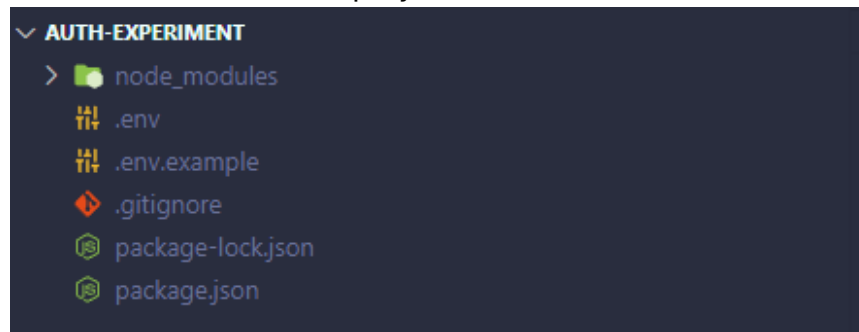


Figure 3 File Structure

Start Coding

The following table outlines the structure and goals of the register endpoint.

POST “/api/v1/register” ENDPOINT STRUCTURE				
API Endpoint Path		Request Method	Response Format	Description
“/api/v1/register”		POST	JSON	Create a new user and return an access token.
Request Parameters				
Parameter	Type	Description		
“name”	String	The user’s name.		
“username”	String	The user’s username.		
“email”	String	The user’s email.		
“password”	String	The user’s password.		
“confirmPassword”	String	The user’s password.		
Response Parameters				
Parameter	Type	Description		
“user”	Object	An object for the user data without the id and 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			
201	{ " 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": "Registered User Successfully" }			
Error Responses				
HTTP Status Code	Response			
400	{ "message": “Registration User Failed” }			
400	{ValidationError}			
500	{ error object }			

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

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

1. Create a folder in the base directory named "models". In this folder make a new file named "user.model.js".
2. Define a new schema in the "user.model.js" file for the "users" table as the following:

Field Name	name	username	email	password
Data Type	String	String	String	String
Required	Yes	Yes	Yes	Yes
Unique	-	Yes	Yes	-
Lowercase	-	Yes	Yes	-
Match	-	-	/^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*\\.\\w{2,3})+\$/	-
Validation Message	Required: "Please enter your name"	Required: "Please enter your username"	Required: "Please enter your email" Match: "Must be a valid email format"	Required: "Please enter your password"

Table 2 User Model Requirements

3. Include timestamps and the name of the collection which should be users.
4. Add the following function to the model before exporting the schema. This will set a function that will return the data of the user resource in json format without including the password.

```
userSchema.set("toJSON", {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString();
    delete returnedObject._id;
    delete returnedObject.__v;
    delete returnedObject.password;
  },
});
```

Figure 4 User toJson Function

5. Create a folder in the base directory named "helpers". In this folder, create two files namely "jsonwebtoken.helper.js" and "errorhandler.helper.js".
6. In the "jsonwebtoken.helper.js" file, two functions will be created namely "authenticateToken" and "generateAccessToken".
7. Copy the following code to the file "jsonwebtoken.helper.js".

```
const dotenv = require("dotenv");
```

```

const jwt = require("jsonwebtoken");
const { unless } = require("express-unless");
const User = require("../models/user.model");
dotenv.config();
function authenticateToken(req, res, next) {
  try {
    const authHeader = req.headers["authorization"];
    let token = authHeader && authHeader.split(" ")[1];
    if (token == null) token = req.cookies?.token;
    if (token == null) {
      return next();
    }
    jwt.verify(token, process.env.JWT_SECRET, async (err, user) => {
      if (err) return next(err);
      const userData = await User.findById(user.id).exec();
      req.user = user;
      if (userData) req.user.data = userData.toJSON();
      return next();
    });
  } catch (err) {
    next(err);
  }
}
function generateAccessToken(id) {
  try {
    const token = jwt.sign({ id: id }, process.env.JWT_SECRET, {
      expiresIn: process.env.JWT_EXPIRE,
    });
    return token;
  } catch (error) {
    throw error;
  }
}
authenticateToken.unless = unless;
module.exports = {
  authenticateToken,
  generateAccessToken,
};

```

Figure 5 "jsonwebtoken.helper.js" Code

Note that the first function is acquiring the value of the header and then saving it in a variable which will be later used to verify the token value based on the secret key in the ".env" file.

If the token is empty the function will check the value of the request cookie which will be used for the web interface. If the token is still empty it will carry on to the next middleware. If the token is verified then the id will be retrieved and saved in the request user property which can be used to fetch the user data.

The second function is used to generate tokens once the registration process is correct. The token will use the user id as a reference for the token hash.

8. Copy the following code to the "errorhandler.helper.js". file.

```
function errorHandler(err, req, res, next) {
  if (typeof err === "string") {
    return res.status(400).json({ message: err });
  }
  if (err.name === "MongoServerError" || err.name === "ValidationError") {
    return res.status(400).json({ message: err.message });
  }
  if (err.name === "JsonWebTokenError" || err.name === "TokenError") {
    return res.status(401).json({ message: "Unauthorized" });
  }
  return res.status(500).json({ message: err.message });
}
module.exports = {
  errorHandler,
};
```

Figure 6 "errorhandler.helper.js" Code

This file will be used to handle errors by returning custom messages based on the type of error.

9. Create a new folder in the base directory named "services" and in this folder create a new file named "auth.service.js". This file will be used to access the user model and the JWT functions for verifying and creating tokens. This function return data which will be handled in the "auth.controller.js" file.
10. Copy the following code to the "auth.service.js" file.

```
const User = require("../models/user.model");
const authJWT = require("../helpers/jsonwebtoken.helper");
async function register(params) {
  const user = new User(params);
  await user.save();
  if (user) {
```

```

const token = authJWT.generateAccessToken(user.id);
return {
  user: user.toJSON(),
  token: token,
  message: "Registered User Successfully",
};
} else throw "Registration User Failed";
}
module.exports = {
  register
};

```

Figure 7 "auth.service.js" Code

11. Create a folder in the base directory named "controllers". In this folder create another folder named "api". In the API folder create a file named "auth.controller.js" file.
12. In the "controllers/api/auth.controller.js" file, copy the following code into it. This file will handle requests immediately from the route file and forward the request to be handled by the auth service file. In the following code, note that there is a custom function for data validation. This function accepts three parameters. The first is the request's data, the second is the response object which will be used to return the results for the request in case the data is not the same as the reference which is the third parameter. The third parameter is an array of the expected variables in the request body data. In the registration function, there are additional steps to validate data specifically the password. If all the data is valid then the request will be sent to the auth service file which will handle the insertion of data into the users' collection.

```

const bcrypt = require("bcryptjs");
const authServices = require("../services/auth.service");
function register(req, res, next) {
  const requiredFields = [
    "name",
    "username",
    "email",
    "password",
    "confirmPassword",
  ];
  validateData(req.body, res, requiredFields);
  if (req.body.password !== req.body.confirmPassword) {
    return res.status(400).json({ message: "Passwords do not match" });
  }
}

```

```

}
if (req.body.password.length < 8) {
  return res
    .status(400)
    .json({ message: "Password must be at least 8 characters" });
}
const name = req.body.name;
const email = req.body.email;
const username = req.body.username;
const password = bcrypt.hashSync(req.body.password, bcrypt.genSaltSync(10));
authServices
  .register({ name, email, username, password })
  .then((results) => res.status(201).json(results))
  .catch((err) => next(err));
}
function validateData(data, res, requiredFields) {
  const missingFields = requiredFields.filter((field) => !(field in data));
  if (missingFields.length > 0) {
    const fieldNames = missingFields.join(", ");
    return res.status(400).json({
      message: `Please fill out the following required field(s): ${fieldNames}`,
    });
  }
}
module.exports = {
  register
};

```

Figure 8 "controllers/api/auth.controller.js" Code

13. Create a folder in the base directory named "routes". Create a folder named "api" in the "routes" folder. In this folder, create a new file named "auth.routes.js".
14. Copy the following code to the "routes/api/auth.routes.js" file.

```

const express = require("express");
const {
  register
} = require("../controllers/api/auth.controller");
const router = express.Router();
router.post("/register", register);
module.exports = router;

```

Figure 9 "routes/api/auth.routes.js" Code

15. Create two files in the base directory namely "app.js" and "server.js". Copy the following code to the "app.js" file.

```
const express = require("express");
const app = express();
const cors = require("cors");
const authJWT = require("./helpers/jsonwebtoken.helper");
const errors = require("./helpers/errorhandler.helper");
const authApiRoutes = require("./routes/api/auth.routes");
app.use(
  cors({
    origin: ["*", "http://127.0.0.1:5500"],
    methods: ["GET", "POST", "PATCH", "DELETE"],
    credentials: true,
  })
);
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(
  authJWT.authenticateToken.unless({
    path: [
      { url: "/api/v1/register", methods: ["POST"] },
    ],
  })
);
app.use("/api/v1", authApiRoutes);
app.use(errors.errorHandler);
app.get("/api/v1/test", (req, res) => {
  return res
    .status(200)
    .json({ message: "Welcome to the Auth-Experiment API" });
});
module.exports = app;
```

Figure 10 "app.js" file

Note the use of cors, CORS (Cross-Origin Resource Sharing) is being enabled to allow requests from any origin. This will allow requests from all origins by using the "*" code and allowing it for the GET, POST, PATCH, and DELETE request methods.

The app is utilizing a function called "authenticateToken". The "authenticateToken" function is a middleware to enforce authentication for all routes except for those specified in the "path" array. This means that any

request made to the server will first go through the "authenticateToken" function before proceeding to the relevant route handler. However, for the routes specified in the "path" array, the request will be allowed to bypass the authentication check and proceed directly to the route handler.

16. Copy the following code to the "server.js" file.

```
const mongoose = require("mongoose");
mongoose.set("strictQuery", true);
const app = require("./app");
require("dotenv").config();
const PORT = process.env.PORT || 5000;
mongoose
  .connect(process.env.MONGODB_URI)
  .then(() => {
    app.listen(PORT, console.log(`Server started on port ${PORT}`));
  })
  .catch((error) => {
    console.log(error);
  });
```

Figure 11 "server.js" code

This code is used to run the application after the connection to the database has been established.

17. Finally, update the "package.json" file. Change the scripts and jest configuration as the following:

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js",
  "api-testB01": "cross-env NODE_ENV=test jest -i tests/api/testB01.test.js --testTimeout=20000",
  "web-testB01": "cross-env NODE_ENV=test jest -i tests/web/testB01.test.js --testTimeout=20000",
  "api-testB02": "cross-env NODE_ENV=test jest -i tests/api/testB02.test.js --testTimeout=20000",
  "web-testB02": "cross-env NODE_ENV=test jest -i tests/web/testB02.test.js --testTimeout=20000",
  "api-testB03": "cross-env NODE_ENV=test jest -i tests/api/testB03.test.js --testTimeout=20000",
  "web-testB03": "cross-env NODE_ENV=test jest -i tests/web/testB03.test.js --testTimeout=20000",
```



```

    "api-testB04": "cross-env NODE_ENV=test jest -i tests/api/testB04.test.js --
testTimeout=20000",
    "web-testB04": "cross-env NODE_ENV=test jest -i tests/web/testB04.test.js --
testTimeout=20000",
    "api-testB05": "cross-env NODE_ENV=test jest -i tests/api/testB05.test.js --
testTimeout=20000",
    "web-testB05": "cross-env NODE_ENV=test jest -i tests/web/testB05.test.js --
testTimeout=20000",
    "testBA": "cross-env NODE_ENV=test jest --testTimeout=40000 --silent --
detectOpenHandles"
  },
  "jest": {
    "setupFilesAfterEnv": [
      "jest-expect-message"
    ],
    "noStackTrace": true,
    "silent": false
  },
}

```

Figure 12 "package.json" Configuration

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.

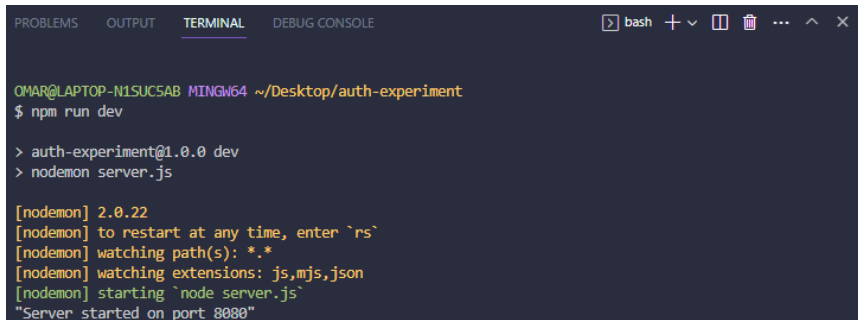
Run the development command "npm run dev" in the terminal and notice the console message.

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.

Via Console and Browser

After running the application, the console should show a message stating that the application is running on port 8080 similar to the following:



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

> auth-experiment@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
"Server started on port 8080"
```

Figure 13 Console Message After Running The Application

In the browser, open the following link <http://localhost:8080/api/v1/test>. it should display the following message “{“message”:“ Welcome to the Auth-Experiment API”}”. Similar to the following figure:

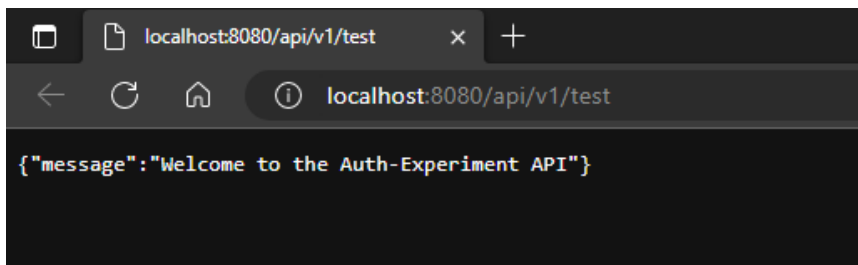


Figure 14 Browser After Running The Application

Using Postman

To test results from this guide and the next guides on Postman, follow these steps:

1. Open a workspace or create a new one from the Workspaces tab in the top navbar.
2. Create a new environment or use the same one as the previous guide. In the Environments tab on the left sidebar, fill in the following values:

	Variable	Type ⓘ	Initial value ⓘ	Current value ⓘ
<input checked="" type="checkbox"/>	protocol	default	http://	http://
<input checked="" type="checkbox"/>	host	default	localhost	localhost
<input checked="" type="checkbox"/>	port	default	:8080	:8080
<input checked="" type="checkbox"/>	version	default	/api/v1	/api/v1
<input checked="" type="checkbox"/>	token	default	accessToken	accessToken

Figure 15 Postman Environment Values

3. Create a new collection from the Collections tab.
4. In the created collection, create a folder named “auth-experiment”.
5. In the created folder create a GET request with the name “GET /api/v1/test”.

6. 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 GET request as the following:
“{{protocol}}{{host}}{{port}}{{version}}/test”

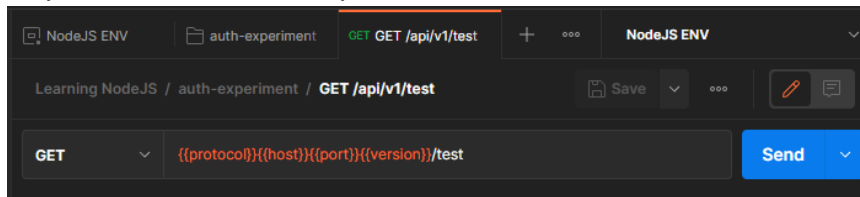


Figure 16 Postman Request Configuration Bar

7. Click “Send” and wait for the response. Postman should show results as the following if everything is working correctly:

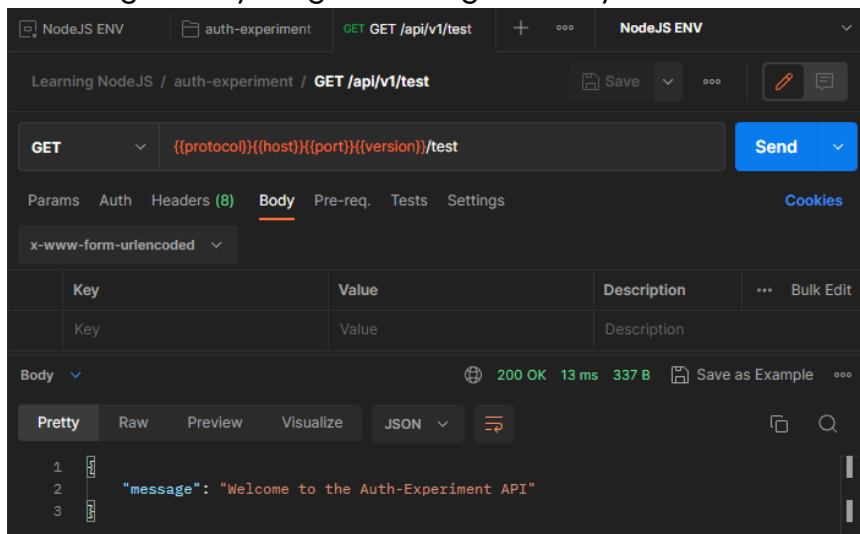


Figure 17 Postman Request Results

Notice that the results are in JSON format because the express JSON functionality is used in the application.

8. Create a POST request. This request has the following link:
“{{protocol}}{{host}}{{port}}{{version}}/register”
9. In the request body, choose the “x-www-form-urlencoded” and fill in the following parameters:
 - a. name
 - b. username
 - c. email
 - d. password
 - e. confirmPassword

10. Send the request and it should return the user data along with a message and access token indicating that the user has been added to the database. Check the database as well to check the results.

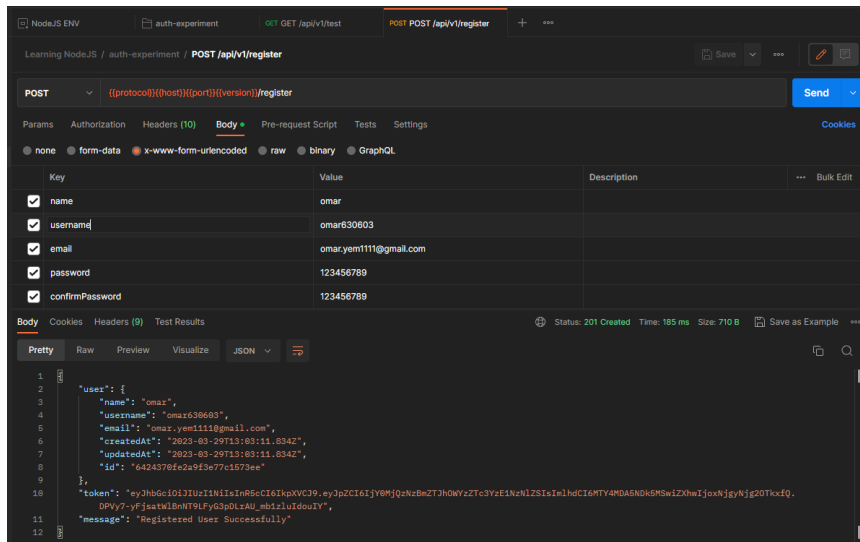


Figure 18 "api/v1/register" Request Results

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. Make a folder called "tests" in the main directory. Inside this folder, create a folder called "api". Copy the file "testB01.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-testB01" 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-testB01

> auth-experiment@1.0.0 api-testB01
> cross-env NODE_ENV=test jest -i tests/api/testB01.test.js --testTimeout=20000

console.log
  Database connected successfully
    at log (tests/api/testB01.test.js:20:15)

PASS tests/api/testB01.test.js (7.475 s)
  Testing application configuration
    ✓ should have the necessary development packages (13 ms)
    ✓ should have the necessary production packages (7 ms)
    ✓ should have the right name and packages (2 ms)
    ✓ should have the right environment variables (5 ms)
    ✓ should have the right database connection (1 ms)
    ✓ should be using json format and express framework (15 ms)
  Testing GET /api/v1/test
    ✓ should return message from the testing endpoint (59 ms)
  Testing POST /api/v1/auth/register
    ✓ should register a new user (834 ms)
    ✓ should return an error if the user does not provide all the required fields (171 ms)
    ✓ should return an error if the user provides an invalid email (122 ms)
    ✓ should return an error if the user provides an invalid password (9 ms)
    ✓ should return an error if the user provides an invalid password (7 ms)
    ✓ should return an error if the user provides an email or username that already exists (196 ms)

Test Suites: 1 passed, 1 total
Tests: 13 passed, 13 total
Snapshots: 0 total
Time: 7.622 s, estimated 8 s
Ran all test suites matching /tests\\api\\testB01.test.js/i.

```

Figure 19 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-testB01

> auth-experiment@1.0.0 api-testB01
> cross-env NODE_ENV=test jest -i tests/api/testB01.test.js --testTimeout=20000

console.log
  Database connected successfully
    at log (tests/api/testB01.test.js:20:15)

FAIL tests/api/testB01.test.js (5.536 s)
  Testing application configuration
    ✓ should have the necessary development packages (6 ms)
    ✓ should have the necessary production packages (5 ms)
    ✓ should have the right name and packages
    ✓ should have the right environment variables (3 ms)
    ✓ should have the right database connection (3 ms)
    ✓ should be using json format and express framework (6 ms)
  Testing GET /api/v1/test
    ✓ should return message from the testing endpoint (51 ms)
  Testing POST /api/v1/auth/register
    ✗ should register a new user (794 ms)
    ✓ should return an error if the user does not provide all the required fields (127 ms)
    ✓ should return an error if the user provides an invalid email (119 ms)
    ✓ should return an error if the user provides an invalid password (8 ms)
    ✓ should return an error if the user provides an invalid confirm password (8 ms)
    ✓ should return an error if the user provides an email or username that already exists (194 ms)

  • Testing POST /api/v1/auth/register > should register a new user

  The response should be {message: 'Registered User Successfully'}, but it is 'User Registered Successfully', change the response in the function that handles the POST /api/v1/register route to return {message: 'Registered User Successfully'}

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

  Expected: "Registered User Successfully"
  Received: "User Registered Successfully"

Test Suites: 1 failed, 1 total
Tests: 1 failed, 12 passed, 13 total
Snapshots: 0 total
Time: 5.662 s, estimated 7 s
Ran all test suites matching /tests\\api\\testB01.test.js/i.

```

Figure 20 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 this application will be created. The same basic endpoint explained in the previous sections will be implemented in a web page that can register users into the database.

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

1. Update the file "app.js". Add the following code to the file to set up a view engine using EJS and configure the application page layout directory.

Add the following lines to the first lines of the page. These lines import the libraries for the view engine and lay-outing. The following code is also importing the "cookie-parser" for handling web cookies.

```
const path = require("path");
const ejsLayouts = require("express-ejs-layouts");
const authWebRoutes = require("./routes/web/auth.routes");
const cookieParser = require("cookie-parser");
```

Figure 21 "app.js" Updates for The Web Interface 1

Add the following lines after this line "app.use(express.urlencoded({ extended: true }));":

These lines are for configuring the application's view engine and specifying the directory of the view file and the layout file which will be used for rendering the index page for the application.

```
app.use(cookieParser());
app.use(express.static(path.join(__dirname, "web")));
app.use(ejsLayouts);
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "web", "views"));
app.set("layout", path.join(__dirname, "web", "layouts", "main"));
```

Figure 22 "app.js" Updates for The Web Interface 2

Add the following path to the "authenticateToken.unless" function. This route will be excluded from the function meaning when there is a request to the registration route, the application won't run the request through the authentication middleware.

```
{ url: "/register", methods: ["POST"] },
```

Figure 23 "app.js" Updates for The Web Interface 3

Add the following function to render the index page with the user's data if the user has been authenticated. This function will handle the request for the entry route for the application.

```
app.get("/", (req, res) => {  
  if (req.user) {  
    return res.render("index", {  
      title: "Auth-Experiment | Home",  
      user: req.user.data,  
    });  
  } else {  
    return res.render("index", {  
      title: "Auth-Experiment | Home",  
      user: req.user,  
    });  
  }  
});
```

Figure 24 "app.js" Updates for The Web Interface 4

Add the following line to use the web routes. Place the line after the previous function.

```
app.use("/", authWebRoutes);
```

Figure 25 "app.js" Updates for The Web Interface 5

Add the following lines at the end of the application before the last line in which the app instance is exported by the model object. These lines are used to respond to any request which has not been listed in the application as a URL. This function will render the error page with a status of 404 indicating that the page or the request was not found.

```
app.use((req, res, next) => {  
  const error = { status: 404, message: "NOT FOUND" };  
  return res.render("error", {  
    title: "Auth-Experiment | Error",  
    error,  
    user: req.user,  
  });  
});
```

Figure 26 "app.js" Updates for The Web Interface 6

2. Create a folder named "web" in the "controllers" folder. In this folder, create a folder named "auth.controller.js".

3. Add the following lines to the "controllers/web/auth.controller.js" file.

```
const bcrypt = require("bcryptjs");
const authServices = require("../services/auth.service");
function register(req, res, next) {
  if (req.method === "GET") {
    return res.render("auth/register", {
      title: "Auth-Experiment | Register",
      message: req.body.message,
    });
  } else {
    const requiredFields = ["name", "username", "email", "password", "confirmPassword"];
    const error = validateData(req.body, requiredFields);
    if (error !== "") {
      req.method = "GET";
      req.body.message = error;
      return register(req, res, next);
    }
    if (req.body.password !== req.body.confirmPassword) {
      req.method = "GET";
      req.body.message = "Passwords do not match";
      return register(req, res, next);
    }
    if (req.body.password.length < 8) {
      req.method = "GET";
      req.body.message = "Password must be at least 8 characters";
      return register(req, res, next);
    }
    const name = req.body.name;
    const email = req.body.email;
    const username = req.body.username;
    const password = bcrypt.hashSync(req.body.password, bcrypt.genSaltSync(10));
    authServices
      .register({ name, email, username, password })
      .then((results) => {
        res.cookie("token", results.token, { maxAge: 60 * 60 * 1000 });
        return res.redirect("/profile");
      })
      .catch((err) => {
        req.method = "GET";
        req.body.message = err;
        return register(req, res, next);
      });
  }
}
```



```

    }
  }
}
function validateData(data, requiredFields) {
  const missingFields = requiredFields.filter((field) => {
    return !(field in data) || data[field].trim() === "";
  });
  let message = "";
  if (missingFields.length > 0) {
    const fieldNames = missingFields.join(", ");
    message = `Please fill out the following required field(s): ${fieldNames}`;
  }
  return message;
}
module.exports = {
  register,
};

```

Figure 27 "controllers/web/auth.controller.js" Code

Note that there are two functions one to handle the request for the route `"/register"` and the other to validate data. In the first function, the request will be checked to see if the method is `"GET"` and then the function will render the register page. If the request has a `"POST"` method then the function will process the data to register users to the database and then the function will redirect the users to the `"/profile"` route. The `"/profile"` route will be worked on in the 3rd guide of this material. Note that the `"register"` function is using the same code from the `"auth.service.js"` file. After the registration process is done, the application should save the token into a cookie that can be used in the authentication process for the web interface.

4. Create a folder named `"web"` in the `"routes"` folder. In this folder, create a folder named `"auth.routes.js"`.
5. Add the following lines to the `"routes/web/auth.routes.js"` file.

```

const express = require("express");
const {
  register,
} = require("../controllers/web/auth.controller");
const router = express.Router();
function isLoggedIn(req, res, next) {
  if (req.user) {
    if (req.path === "/login" || req.path === "/register") return res.redirect("/");
    return next();
  }
}

```

```

    } else {
      if (req.path === "/login" || req.path === "/register") return next();
      return res.redirect("/login");
    }
  }
}
router.get("/register", isLoggedIn, register);
router.post("/register", isLoggedIn, register);
module.exports = router;

```

Figure 28 "routes/web/auth.routes.js" Code

The function "isLoggedIn" is used to redirect users to the login route if they are not logged in. The route "login" will be worked on in the next guide file.

6. Create a folder named "web" in the base directory. This folder will contain three folders namely "layouts", "styles", and "views".
7. Copy the file "main.ejs" from the supplements folder to the newly created folder "/web/layouts" and copy the file "main.css" to the folder "/web/styles".
8. Create two files namely "index.ejs" and "error.ejs" in the "/web/views" folder.
9. Copy the following code to the "index.ejs" file:

```

<div class="container">
  <h1 class="title">Welcome to Auth-Experiment!</h1>
  <% if (typeof user === 'undefined') { %>
    <div class="alert alert-warning">
      <p class="message">Hello</p>
    </div>
  <% } else { %>
    <div class="alert alert-success">
      <p class="message">Hello <%= user.name %></p>
    </div>
  <% } %>
</div>

```

Figure 29 "/web/views/index.ejs" File Structure

10. Copy the following code to the "error.ejs" file:

```

<% if (typeof error !== 'undefined') { %>
  <div class="alert alert-error">
    <h1 class="title"><%= error.status %></h1>
    <p class="message"><%= error.message %></p>
  </div>
<% } %>

```

Figure 30 "/web/views/error.ejs" File Structure

This code will be the template for error messages in the application. Mostly the error of "Not Found" can happen if a request is made to an unknown route.

11. Create a folder named "auth" in the "web/views" folder. In this folder, create a file named "register.ejs".
12. Copy the following code to the "web/views/auth/register.ejs" file.

```
<% if (typeof message !== 'undefined') { %>
<div class="alert">
  <p class="message"><%= message %></p>
</div>
<% } %>
<div class="container">
  <h1 class="title">Register</h1>
  <p class="description">Fill the form below to create a new account</p>
</div>
<div class="container">
  <form action="/register" method="POST">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" name="name" id="name" class="form-control" />
    </div>
    <div class="form-group">
      <label for="username">Username</label>
      <input type="text" name="username" id="username" class="form-control" />
    </div>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="email" name="email" id="email" class="form-control" />
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input
        type="password"
        name="password"
        id="password"
        class="form-control"
      />
    </div>
    <div class="form-group">
      <label for="confirmPassword">Confirm Password</label>
      <input
```

```

    type="password"
    name="confirmPassword"
    id="confirmPassword"
    class="form-control"
  />
</div>
<button type="submit" class="btn btn-primary">Register</button>
</form>
</div>

```

Figure 31 "web/views/auth/register.ejs" Code

This code is a template for the registration form which has 2 text inputs for the name, and username. One input with type email for the email. The form also has 2 password inputs for the password and password confirmation.

At the top of the code, there is an if condition for the message which will be used to show errors if they happen while processing the registration request.

After finishing all the steps above, the web interface should be running and has achieved all the goals for this document.

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/>. If the application is not running in the terminal then use the command "npm run dev" to start the app.

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

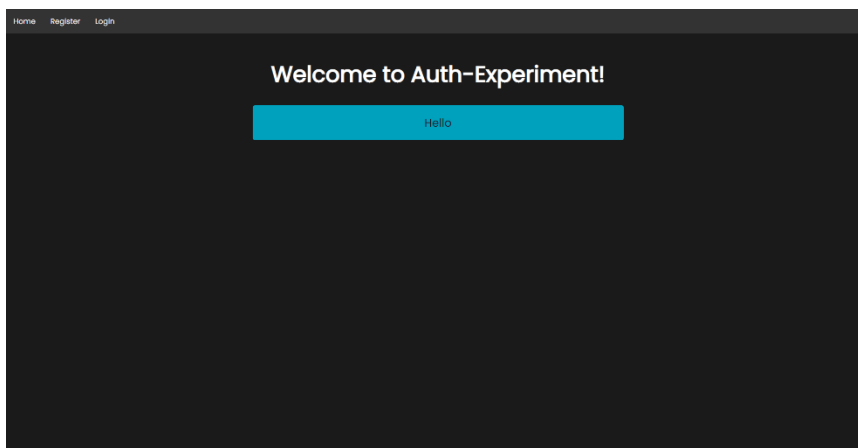


Figure 32 The Welcome Page

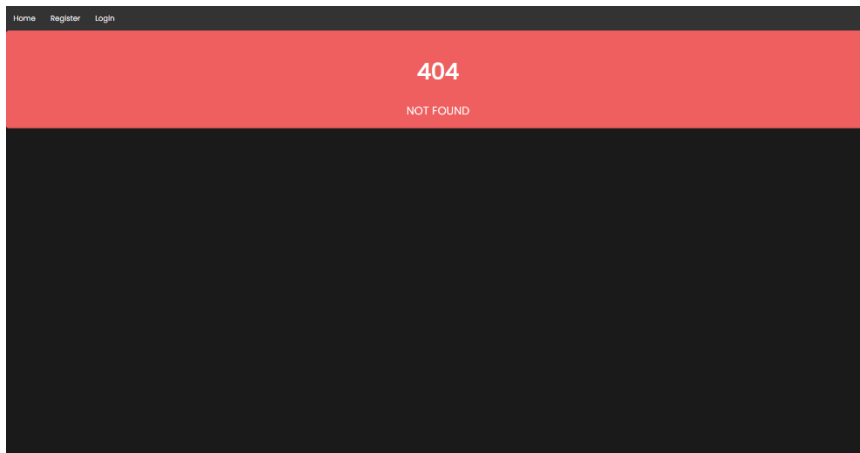


Figure 33 The Error Page

Figure 34 The Register Page

To test the application, create a folder named “web” in the “tests” folder and then copy the file “testB01.test.js” and the folder “images” from the “/tests/web” folder and paste it to the folder “/tests/web” in your project directory. After that, run the command “npm run web-testB01” and notice the results.

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

```

OWAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/auth-experiment
$ npm run web-testB01

> auth-experiment@1.0.0 web-testB01
> cross-env NODE_ENV=test jest -i tests/web/testB01.test.js --testTimeout=40000

PASS tests/web/testB01.test.js (8.759 s)
  Testing the index page title and content
    ✓ should have the right title (186 ms)
    ✓ should have nav bar with 3 links (75 ms)
  Testing the error 'Not Found' page
    ✓ should have the right title (80 ms)
    ✓ should have a status code of 404 (72 ms)
    ✓ should have a message saying 'NOT FOUND' (83 ms)
  Testing the register page
    ✓ should have the right title (76 ms)
    ✓ should have a form with 5 inputs and 1 button (127 ms)
    ✓ should register a new user (486 ms)
    ✓ should have the name of the user in the index page (93 ms)
  Testing the index page, register page, and error 'Not Found' page image snapshots
    ✓ matches the expected styling for the index page after register (676 ms)
    ✓ matches the expected styling for the index page before register (2890 ms)
    ✓ matches the expected styling for the register page (465 ms)
    ✓ matches the expected styling for the error page (483 ms)

Test Suites: 1 passed, 1 total
Tests: 13 passed, 13 total
Snapshots: 4 passed, 4 total
Time: 8.86 s, estimated 10 s
Ran all test suites matching /tests\\web\\testB01.test.js/i.

```

Figure 35 Successful Web Test Results

```

OWAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/auth-experiment
$ npm run web-testB01

> auth-experiment@1.0.0 web-testB01
> cross-env NODE_ENV=test jest -i tests/web/testB01.test.js --testTimeout=40000

FAIL tests/web/testB01.test.js (9.397 s)
  Testing the index page title and content
    ✓ should have the right title (215 ms)
    ✓ should have nav bar with 3 links (72 ms)
  Testing the error 'Not Found' page
    ✓ should have the right title (77 ms)
    ✓ should have a status code of 404 (89 ms)
    ✓ should have a message saying 'NOT FOUND' (110 ms)
  Testing the register page
    ✓ should have the right title (57 ms)
    ✓ should have a form with 5 inputs and 1 button (121 ms)
    ✓ should register a new user (455 ms)
    ✓ should have the name of the user in the index page (97 ms)
  Testing the index page, register page, and error 'Not Found' page image snapshots
    ✓ matches the expected styling for the index page after register (789 ms)
    ✓ matches the expected styling for the index page before register (2175 ms)
    ✗ matches the expected styling for the register page (623 ms)
    ✓ matches the expected styling for the error page (580 ms)

  • Testing the index page, register page, and error 'Not Found' page image snapshots › matches the expected styling for the register page

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

    Expected image to match or be a close match to snapshot but was 0.5696696017908273% different from snapshot (4486 differing pixels).
    See diff for details: tests/web/images/___diff_output___register-page-diff.png

  › 1 snapshot failed.
  Snapshot Summary
  › 1 snapshot failed from 1 test suite. Inspect your code changes or run 'npm run web-testB01 -- -u' to update them.

Test Suites: 1 failed, 1 total
Tests: 1 failed, 12 passed, 13 total
Snapshots: 1 failed, 3 passed, 4 total
Time: 9.498 s
Ran all test suites matching /tests\\web\\testB01.test.js/i.

```

Figure 36 Failed Web Test Results

The terminal displays feedback that compares your project screenshot to the reference image. Differences between the two images are located in the `"/test/web/images/___diff_output___"` folder. If there are no differences, the images will disappear. The different images are useful for identifying UI differences. The reference image is used as a guide for how the webpage should look.

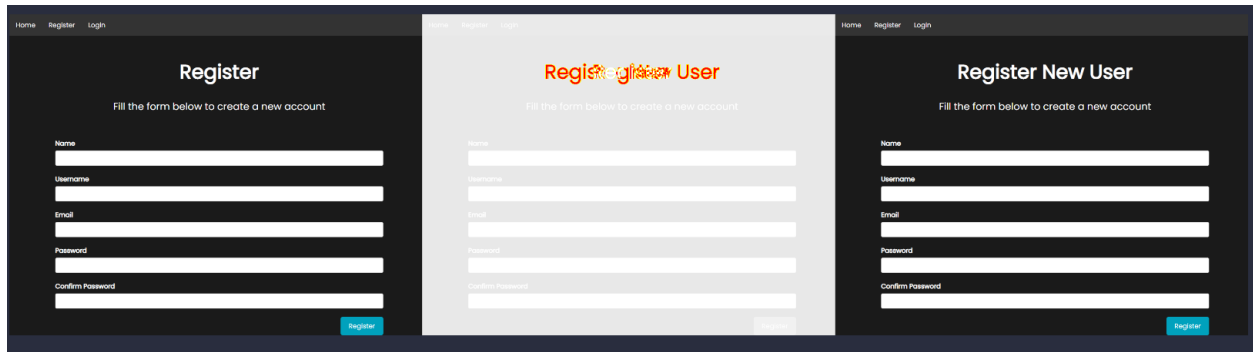


Figure 37 "__diff_output__" Image Example

The image on the left is the reference image and the one on the right is the project's current image. The middle image shows the differences in red color indicating that the difference is in the message area.

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 first 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 registering users and saving their data in a MongoDB instance. The students should also learn how to create a new web interface to deal with the registration process.