**94/135**

**Testing environment/rules**

* For this test, you may use your notes, textbook, reading materials, your completed lab assignments, and the internet as a resource for this Hands-On test.
* This Hands-On Test is due **Friday 11/3/23 at 12:00PM**
* The test must be submitted via a GitHub repository, and you must give collaborator/admin access to your instructor.
* This is an individual project, you are expected to complete all work by yourself without collaboration/input from other students.
* All code must be your own original work. Code found online, on websites like Stack Overflow, Medium, etc. is strictly prohibited. *(Example code found on* [*developer.mozilla.org*](https://developer.mozilla.org) *is permitted though.)*
* *This test is worth a total of 135 points (or 1.35 exam grades.)*

**How to submit your code**

As we have done with previousassignments, you need to create a new GitHub repository for this project.

Details are listed below:

* Use the **"awd1111-exam-3"** repository that you created for the last exam.

**Install additional dependencies**

You should already have these dependencies installed:

* **joi**
* **debug**
* **dotenv**
* **express**
* **mongodb**

For this project, you will need to install and use the following dependencies:

* **bcrypt**
* **cookie-parser**
* **jsonwebtoken**

**Best Practices**

To receive full credit for this assignment, you must consistently follow all of the below coding standards.

* Indent code using **2 spaces** per level, as per [industry standards.](https://google.github.io/styleguide/jsguide.html#formatting-block-indentation)
* Name all variables and functions using **camelCase.**
* Avoid declaring **global variables** where possible, prefer **local variables** and **function parameters** instead.
* **Don't use var. You will lose 5 points for every use of the var keyword!**
* Use **const** to define variables whenever possible. For example:

|  |
| --- |
| const milesDriven = req.body.milesDriven; |

* Use **let** to define variables only when **const** isn't possible. For example:

|  |
| --- |
| let x = 3; x += 5; |

* Prefer **arrow functions (=>)** for anonymous functions, over the traditional **function declaration** syntax. For example:

|  |
| --- |
| (x, y) => x + y |

* The **function declaration** syntax is permitted for creating **named functions.** For example:

|  |
| --- |
| function add(x, y) { return x + y; } |

* Use [**template strings**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) instead of string concatenation. For example:

|  |
| --- |
| const fullName = `${firstName} ${lastName}`; |

* Use [**async-await**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) to support asynchronous database operations.

|  |
| --- |
| async function findAllPets() {  const db = await connect();  const pets = await db.collection('pets').find({}).toArray();  return pets; } |

* Use [**try-catch**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch) to handle exceptions and promise rejections.

|  |
| --- |
| router.get('/list', async (req, res, next) => {  try {  const pets = await dbModule.findAllPets();  res.json(pets);  } catch (err) {  next(err);  } }); |

**Project Overview**

* Previously you built a small Products API.
* In this assignment you will implement the Authentication and Authorization mechanisms.
* You will also improve the /api/product/list to allow users to search for products.

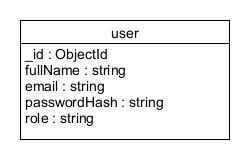
**Users API (75 pts)**

Create a Users API for an ecommerce store. The CRUD routes described below implement the basic functionality needed to manage users and support logins.

A UML diagram of the user document is shown on the right.

Implement this API as a route module, located at **/routes/api/user.js**

*The role attribute can optionally be an array of strings*

* Use **async-await** to implement asynchronous database operations.
* Use **try-catch** to handle database errors and promise rejections.
* Use **joi** to validate the request data.
* Send all data and messages back as **JSON.**
* Use the **200, 400, 404,** and **500** status codes appropriately.

***Test each route as you implement it!***

**GET /api/user/list** \_\_\_\_\_/ 5pts

* Returns all of the users in the database as a **JSON array.**
* *You do not need to implement searching, filtering, sorting, or paging for this route.*

**GET /api/user/:userId** \_\_\_\_\_/ 5pts

* Returns a single user from the database as a **JSON object.**
* Find the user based on the provided **ID.**
* If the ID is invalid or the user is not found, return a **404** response.

***Test each route as you implement it!***

**POST /api/user/register**

* **Inserts a new user into the database.** \_\_\_\_\_/ 5pts
* Hash passwords using bcrypt. \_\_\_\_\_/ 5pts
* Generate a new JWT token and store it in a cookie. \_\_\_\_\_/ 5pts
* Returns a JSON object containing a **message,** the new **userId,** and a new **token.**
* Accept the following fields via the **body** of the request:
  + fullName : string
  + email : string
  + password : string
* Validate the body of the request using **Joi.** \_\_\_\_\_/ 5pts

**POST /api/user/login**

* **Checks the user's credentials against the database**. \_\_\_\_\_/ 5pts
* Compare passwords using bcrypt. \_\_\_\_\_/ 5pts
* Generate a new JWT token and store it in a cookie. \_\_\_\_\_/ 5pts
* Returns a JSON object containing a **message,** the **userId,** and a new **token.**
* Accept the following fields via the **body** of the request:
  + email : string
  + password : string
* Validate the body of the request using **Joi.** \_\_\_\_\_/ 5pts

**GET /api/user/me** \_\_\_\_\_/ 5pts

* Returns the current user's info from the database as a **JSON object.**
* Find the user based on the **ID** of the logged in user.
* If the user is not logged in, return a **401** response.
* If the user is not found, return a **404** response.

**PUT /api/user/me**

* **Provides a self-service route for user's to update their own information.** \_\_\_\_\_/ 5pts
* Hash passwords using bcrypt. \_\_\_\_\_/ 5pts
* Generate a new JWT token and store it in a cookie. \_\_\_\_\_/ 5pts
* Returns a JSON object containing a **message,** the **userId,** and a new **token.**
* Accept the following fields via the **body** of the request:
  + fullName : string
  + email : string
  + password : string
* Validate the body of the request using **Joi.** \_\_\_\_\_/ 5pts
* If the user is not logged in, return a **401** response.
* If the user is not found, return a **404** response.

**Implement Middleware (20 pts)**

**validId**  \_\_\_\_\_/ 2.5pts

* **Checks if a route parameter exists and is a valid ObjectId.**

**validBody**  \_\_\_\_\_/ 2.5pts

* **Checks if the request body is valid using a Joi object schema.**

**isLoggedIn**  \_\_\_\_\_/ 5pts

* **Checks if the user is currently logged in.**

**hasRole**  \_\_\_\_\_/ 5pts

* **Checks if the user has a role of administrator**

**Install Middleware (20 pts)**

Now that you have implemented the middleware modules described above, we need to install them.

Install the following middleware functions in **server.js,** so that they are executed for all requests. \_\_\_\_\_/ 10pts

* cookieParser() from 'cookie-parser'
* authMiddleware() from '@merlin4/express-auth'

Install the middlewares listed in the table below for the routes listed. \_\_\_\_\_/ 12pts **-6pts**

|  |  |
| --- | --- |
| **Route** | **Middleware** |
| GET /api/user/list | **hasRole(‘admin’)** |
| GET /api/user/:userId | **hasRole(‘admin’), validId('userId')** |
| POST /api/user/register | **validBody(registerSchema)** |
| POST /api/user/login | **validBody(loginSchema)** |
| GET /api/user/me | **isLoggedIn()** |
| PUT /api/user/me | **isLoggedIn(), validBody(updateSelfSchema)** |
|  |  |
| GET /api/product/list | *none, this route allows anonymous users* |
| GET /api/product/id/:productId | **validId('productId')** |
| GET /api/product/name/:productName | *no middleware needed* |
| PUT /api/product/new | **hasRole(‘admin’), validBody(newProductSchema)** |
| PUT /api/product/:productId | **hasRole(‘admin’), validId('productId'), validBody(updateProductSchema)** |
| DELETE /api/product/:productId | **hasRole(‘admin’), validId('productId')** |
|  |  |

***Test each route as you make these changes!***

**Product Search (20 pts)**

Now that our Authentication and Authorization mechanisms are in place, we want to implement tools for the customer to find the product that they are looking for.

**GET /api/product/list**

* Upgrade this route to provide search functionality.
* The following data **may** be provided via **query parameters,** and read from **req.query:**
  + **keywords** \_\_\_\_\_/ 4pts
    1. This parameter is optional, do nothing if it is falsy.
    2. If it is not falsy, then use the [**$text**](https://docs.mongodb.com/manual/text-search/) operator to search for users containing the provided string.
    3. Add a [**wildcard text index**](https://docs.mongodb.com/manual/core/index-text/) using the MongoDB shell to support these queries.
  + **category** \_\_\_\_\_/ 2pts
    1. This parameter is optional, do nothing if it is falsy.
    2. If it is truthy, then show products only from this category.
  + **maxPrice** \_\_\_\_\_/ 2pts
    1. This parameter is optional, do nothing if it is falsy.
    2. If it is truthy, then show only products that have a price of maxPrice or greater.
  + **minPrice** \_\_\_\_\_/ 2pts
    1. This parameter is optional, do nothing if it is falsy.
    2. If it is truthy, then show only products that have a price of minPrice or less.
  + **sortBy** \_\_\_\_\_/ 4pts
    1. Sort the results based on this selection.
    2. Allowed Options: **"name", "category", "lowestPrice", "newest"**
    3. Default to sorting by the **"name"** mode as described below.
    4. These options must sort the results as described in the list below:
       - **name:** name ascending
       - **category:** category ascending, name ascending
       - **lowestPrice:** price ascending, name ascending
       - **newest:** created date descending, name ascending
    5. Implement this logic with either a **switch** statement or **if** statements.
    6. *The extra fields listed above are to make the sort order stable.*
    7. *These sort orders presume that the product names are unique. Add a unique index for the name.*
  + **pageSize** \_\_\_\_\_/ 2pts
    1. Specifies the number of items per page to be displayed.
    2. Default to **5,** if not provided.
    3. Use **.limit()** or **$limit** to limit the number of results returned to the page size.
  + **pageNumber** \_\_\_\_\_/ 2pts
    1. Specifies which page of results that the client wants.
    2. The first page is page **1.**
    3. Default to **1,** if not provided.
    4. Use **.skip()** or **$skip** to skip over the required number of items. *(Remember to consider the pageSize.)*
* **All of the above query parameters are optional. All combinations of these query parameters must be supported!**