PLMS Assignment 2

PLMS - Building Microservices using NodeJS Assignments

Requirements:

1. MongoDB

Tasks:

1. Upgrade bank-service.
    a. Connect to MongoDB database.
    b. Create MongoDB Model
    c. Create a schema and a middleware to check it.
2. Create auth-service for user authentication.
    a. Set up the auth-service.
    b. Connecting to MongoDB.
    c. Creating schema for the routes.
    d. Create the 3 routes (also create a new middleware).
3. Create loan-service for loan management.
    a. Set up the loan-service.
    b. Creating schema for the routes.
    c. Create the 2 routes (create and fetch loans).

Task 1. Upgrade bank-service

In this task, we'll convert our existing bank-service from using in-memory storage to MongoDB and add proper validation.

1. Connect to MongoDB database:

Step 1. Install Additional Dependencies. Navigate to your `bank-service` directory and install the required packages:

```
cd bank-service
npm install mongoose zod
```

- **mongoose** is a library used to connect to a MongoDB server and perform CRUD operations.
- **zod** is a library for validating request data with various built-in validators.

Create a "`.env`" file at the service root with your MongoDB URI. Name it `MONGODB_URI` and assign it your MongoDB access URI. "`.env`" is used to store the environment variables for the project.

```
MONGODB_URI=<-MongoDB URI->/PLMS-Bank
// PLMS-Bank is to make sure that this is a bank cluster data
// but you can also use different MongoDB instances to separate
// the data and make sure that each service is using its own
// MongoDB Database.
```

Remember to use this URI in your project to connect to MongoDB servers.

```
await mongoose.connect(process.env.MONGODB_URI);

console.log("Connected to MongoDB :]");
```

Note: You must create a new async function to run the app and connect to a database as mongoose creates a promise which needs to be awaited before proceeding.

Since `mongoose.connect` returns a promise and hence we need to await the connect request. We cannot use "`await`" without using the async function, so we will create a new function called `start` and move our connect request along with the app listener into the `start` function.

2. Create MongoDB Model.

Step 1. We will create a model for our bank-service in `src/models/bank.js`. Define the Mongoose schema for this database so we can eliminate the use of wrong data structures to make requests. The model will consist of the following:

- bankName (String, required)
- logo (String, required)
- minLoanAmount (Number, required)
- maxLoanAmount (Number, required)
- minInterestRate (Number, required)
- maxInterestRate (Number, required)
- minCreditScore (Number, required)
- termLength (Number, required)
- processingFee (Number, required)
- rating (Number, required)

We will then modify the build method so that it removes any unnecessary data or transform the data into a more understandable format, like replacing the field `_id` with `id` or removing the pre-generated version number by MongoDB. Since most of you may not be familiar with MongoDB you can take this example to learn how to create a mongoose schema.

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

const bankSchema = new mongoose.Schema(
  {
    bankName: {
      type: String,
      required: true,
    },
    logo: {
      type: String,
      required: true,
    },
    minLoanAmount: {
      type: Number,
      required: true,
    },
    maxLoanAmount: {
      type: Number,
      required: true,
    },
    minInterestRate: {
      type: Number,
      required: true,
    },
```

```
      maxInterestRate: {
        type: Number,
        required: true,
      },
      minCreditScore: {
        type: Number,
        required: true,
      },
      termLength: {
        type: Number,
        required: true,
      },
      processingFee: {
        type: Number,
        required: true,
      },
      rating: {
        type: Number,
        required: true,
      },
    },
  {
      toJSON: {
        transform(doc, ret) {
          ret.id = ret._id;
          delete ret._id;
          delete ret.password;
          delete ret.__v;
        },
      },
    }
  );

  bankSchema.pre("save", async function (done) {

    done();
  });

  bankSchema.statics.build = (attrs) => {
    return new Bank(attrs);
  };

  const Bank = mongoose.model("Bank", bankSchema);

  module.exports = { Bank };
```

Note: The change from session storage to MongoDB to have a better database management system and hence not losing data each time you restart the service.

Step 2. You will also need to update controllers to work with MongoDB, instead of using session data which gets reset each time you run the service.

3. Create a schema and a middleware to check it.

Step 1. Create Validation Schemas.

Create a new file in `src/schemas` and name it `validation-schemas.js`. This file will contain Zod schemas to validate incoming data. Each route in our service will have different validation schema.

The validation scheme for adding a bank will be:

- Bank name is required
- Logo must be a valid URL
- Loan amounts must be at least 1000
- Interest rates must be between 0.01% and 100%
- Minimum Credit score must be between 300 and 850
- Term length must be at least 1 month
- Processing fee must not be below $0
- Rating must be between 0 and 5

The validation scheme for getting the bank data will be empty as it is a get all request. We are not filtering the data why what has been requested.

This also makes us remove the complex logic we have implemented to make sure that the data is following the recommended schema.

Step 2. Create a middleware that will be overseeing the validation schema for each request and return an error message in the same serialized fashion as the other errorHandler.

```
{
  "errors" : [
    {
      "message" : "This is an error message!",
      "field" : "The field of error (optional)."
    }
  ]
}
```

This also brings us to create another error class because validationError is a very common error which will be used again multiple times. So, while we are at creating a new middleware let us also work on the new error class as well. Here, we can use the optional property field in our response message as the fields that came out wrong during the validation check in the middleware.

Step 3. Wire up all the middlewares to the route handler.

Adding middlewares is very easy and follows the following syntax:

```
router.method("/api/route", middlewares..., controller);
```

Note: The `[middlewares]` *is meant to represent multiple middlewares each separated by a comma* `(,)` *and not with triple dots* `(...)`*. They also follow the sequence they are written in. So, the middlewares is called before the controller.*

Step 4. Test your API using Postman

Try out all the previous test cases again but this time we have proper validation in place. We can finally try out all kinds of test cases to block malicious users from entering wrong data. If you can see the response as shown in the example given above then congratulations, your end point is ready and functional.

Task 2. Create auth-service for user authentication

Now we'll create a new authentication service to handle user registration and login.

1. Set up the auth-service

Step 1. Set up the service structure like our bank-service but this time for `auth-service`. Create a new folder called auth-service and initialize a `package.json` file and install all these packages:

a) dotenv
b) express
c) jsonwebtoken
d) cors
e) mongoose
f) nodemon
g) zod

Step 2. We have already created error classes and middleware; we can just copy them over to this new service as they created by keeping it reusable for other services too. Now we are going to set our basic express, cors. Set the PORT for this app to 8001. Remember to also go over the CORS as well if you are using the frontend.

2. Connecting to MongoDB

Step 1. Connect to MongoDB.

This time creating a new cluster or an entire new MongoDB server called "`PLMS-Auth`". Next we start working on the model. They should include the following:

- name (required)
- email (required)
- password (required, hashed password)
- role (required, default: "customer", enum: ["customer", "manager"])
- A pre-save hook to hash passwords

Step 2. Let us create a hashing system (If you know how to hash the password you can create a new JavaScript file with the hashing function). Here is a very basic hashing function for you to try.

```
const { scrypt, randomBytes } = require("crypto");
const { promisify } = require("util");

const scryptAsync = promisify(scrypt);

class Password {
  static async toHash(password) {
    const salt = randomBytes(8).toString("hex");
    const buf = await scryptAsync(password, salt, 64);
    return `${buf.toString("hex")}.${salt}`;
  }

  static async compare(storedPassword, suppliedPassword) {
    const [hashedPassword, salt] = storedPassword.split(".");
    const buf = await scryptAsync(suppliedPassword, salt, 64);
    return buf.toString("hex") === hashedPassword;
  }
}

module.exports = { Password };
```

After hashing the password, the saving/updating/deleting the data from the database will be handled by each route controller respectively.

Note: As for where the file is stored you can either save it under `src/controllers` *or create a new folder with name of your choice such as* `src/helper` *or* `src/services`.

3. Creating schema for the routes.

We will first create schemas for each route before working on the routes themselves.

Step 1. We will start working on creating the routes. This service is supposed to have three routes:

a) `signin-route` – This route is supposed to have the check for the email and password, which is hashed before checking using the function we had created.
b) `signup-route` – This route is supposed to create a new user on signup. It will have an additional check after validation (like if the user already exists) then creates a user.
c) `signout-route` – This route is supposed remove the user data from the response.

Step 2. Next we are going to create our 3 routes. First we are going to work with `signup-route`. This route would have the following required validated fields:

1) name (String with not special characters, required)
2) email (Email, required)
3) password (required)
4) role (only when creating a manager, otherwise defaults to customer)

Step 3. Now we are going to work on our `signin-route`. This route would have the following required validation field:

1) email (Email, required)
2) password (String, required)

Step 4. Now it is time for our last route in this service called `signout-route`. This route is made to remove the Authorization header from a request. So, for the schema of the route, we do not need any kind of scheme here.

Note: In Project that are in large scales we need a different approach from that signout which is more complex and since this is a very small-scale learning project, it does not require any additional complexity. Our focus on this project is to understand microservices and not how to solve complex problems.

4. Create the 3 routes.

Now that we have successfully created the schemas for the routes, we will work on the routes and their controllers.

Step 1. Creating our signin controller which involves the following

a) Pasing the request data (check if the user exists with the email).
b) Checking the password using the has check we created in the password controller.
c) Generate a new JWT token for the user.

Here is how you can sign a JWT token for the user. This token will grant access to the user if they wish to use any other services on our frontend.

```
const userJwt = jwt.sign(
  {
    id: existingUser.id,
    name: existingUser.name,
    email: existingUser.email,
    role: existingUser.role,
  },
  process.env.JWT_KEY,
  { expiresIn: "10d" } // expires in 10 days
);
```

Note: `existingUser` *here refers to the user found with the email from the database. The expiresIn variable is used to create a signed key for a specific duration, it is in the format "10m" for 10 minutes, "1h" for 1 hour, "120ms" for 120 milliseconds*

The response to the user should look like the json data below.

```
{
  "token" :
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4NjNmYTBhMDg5NWFiYjQyZDM2NGRlOCIsIm5hbWUiOiJUZXN0aW5nTmFtZSIsImVtYWlsIjoidGVzdGluZ0B0ZXN0LmNvbSIsInJvbGUiOiJjdXN0b21lciIsImlhdCI6MTc1MTM4MjUzOCwiZXhwIjoxNzUyMjQ2NTM4fQ.7vleoQBHHepw69SytMZNKZlprqaWySHaf5TM75yqouU"
}
```

Step 2. Creating our `signup` controller which involves the following

a) Parsing the request data (check for duplicate email).
b) Hashing the password.
c) Storing the data into MongoDB.
d) Create a new JWT token for the user to access the webpage.

You can use the same method to create JWT token in the similar fashion for this route controller as well.

Step 3. Before we go on to the `signout` controller let us first focus on creating new middleware which verifies if the user's provided token is valid or not.

This token will be used in the Authorization header in the request by the frontend (for testing without the frontend you would have to manually enter the Authorization header) with this token in the format given below. (i.e., adding "Bearer" followed by the token)

```
"Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4NjNmYTBhMDg5NWFiYjQyZDM2NGRlOCIsIm5hbWUiOiJUZXN0aW5nTmFtZSIsImVtYWlsIjoidGVzdGluZ0B0ZXN0LmNvbSIsInJvbGUiOiJjdXN0b21lciIsImlhdCI6MTc1MTM4MjUzOCwiZXhwIjoxNzUyMjQ2NTM4fQ.7vleoQBHHepw69SytMZNKZlprqaWySHaf5TM75yqouU"
```

Since this is how we are going to receive requests from the user. We need to create a middleware that first checks if the Authorization header has a token. Then if the token is present then verify the token. When verified it passes the request to the next function for the route.

Step 4. Creating our `signout` controller, this controller does not have much functionality and hence will only resend in a blank response. Since we worked on the new middleware which checks for the `current-user`, we can try out our middleware here (we don't need to use it here, but we will be using it very often while we work on other services).

Task 3. Create loan-service for loan management

In this task, we are going to create another new service called loan-service. The structure of this service is going to be like the previous services we have worked on.

1. Set up the loan-service.

Step 1. Set up the service structure like our bank-service but this time for loan-service. Create a new folder called loan-service and initialize a package.json file and install all these packages:

a) dotenv
b) express
c) jsonwebtoken
d) cors
e) mongoose
f) nodemon
g) zod

Step 2. We have already created error classes and middleware; we can just copy them over to this new service as they created by keeping it reusable for other services too. Now we are going to set our basic express, cors. Set the PORT for this app to 8001. Remember to also go over the CORS as well if you are using the frontend.

2. Connecting to MongoDB

Step 1. Connect to MongoDB.

This time creating a new cluster or an entire new MongoDB server called "`PLMS-Auth`". Next we start working on the model. They should include the following:

- userid (String, required)
- updatedUserData (object with user details)
    a) name (String, required)
    b) dob (Date, required)
    c) profileURL (URL, required)
    d) gender (enum: ["male", "female", "prefer not to say"], required)
    e) employer (String, required)
    f) monthlySalary (Number, required)
    g) creditScore (Number, required)
- bankId, bankName (String, required)
- bankLogo (URL, required)
- appliedLoanAmount, interestRate, termLength (Number, required)
- dateOfApplication (default: current date)
- status (enum: ["pending", "approved", "rejected"], default: "pending")

3. Creating schema for the routes.

Now we are going to work on the schema for the routes. There are a total of 2 routes for this service, and they are as follows:

a) getloan-route – to fetch each user's loan data.
b) createloan-route – to create a loan application.

The scheme for create-loan-route is going to include:

1) userid (String, required)
2) updatedUserData (object, required)
    a. name (String, required)
    b. dob (Number, required)
    c. profileURL (Url, required)
    d. gender (emun: ["male", "female", "prefer not to say"], required)
    e. employer (String, required)
    f. monthlySalary (Number, required)
    g. creditScore (Number, required)
3) bankId, bankName (String, required)
4) bankLogo (Url, required)
5) appliedLoanAmount (Number, required, needs to be between min/max value)
6) termLength (Number, required)
7) bankDetails (object, required)
    a. minLoanAmount
    b. maxLoanAmount
    c. minInterestRate
    d. maxInterestRate
    e. minCreditScore

While there would be no scheme required for get-loan-route as this route will only rely on the JWT Token embedded in the requests.

4. Creating 3 routes.

Step 1. Creating get-loan-route.

Under `src/routes`, create `get-loan-route.js`. Define `GET` request at (“`/api/loan/applications`”).

Create a controller for this route where you need to fetch their respective loan data from the database. This will not be done using any data but rather the JWT Token that we will attach to the Authorization header of any request. This route we will have checks for current user id and if the token he provides matches his own user id, we proceed to fetch their application.

Step 2. Creating create-loan-route

Under `src/routes`, create `get-loan-route.js`. Define `GET` request at (“`/api/loan/applications`”).

Create a controller for this route where they need to send in data which the controller will process it then save it into the database. The controller will also check if another loan active loan already exists for the same user, if yes then they cannot apply to the same bank until their loan has been approved already. This route we will have similar checks for current user id and if the token he provides matches his own user id, we proceed to create his application.

Now that most of the check have happened we can finally build something new now, a function that is going to calculate the interest rate based on the maximum and minimum loan amount and interest rates provided by the bank and the user applied loan amount.

```
      const calculateInterestRate = (amount, loan) => {
        const amt = Number(amount);
        const { minLoanAmount, maxLoanAmount, minInterestRate,
maxInterestRate } = loan;
        if (amt < minLoanAmount || amt > maxLoanAmount) return "";
        const fraction = (amt - minLoanAmount) / (maxLoanAmount -
minLoanAmount);
        return (
          minInterestRate +
          fraction * (maxInterestRate - minInterestRate)
        ).toFixed(2);
      };
```

Now we test out all the APIs using Postman and check if we are getting a desired output.

The service for auth would give us JWT tokens while the service for loans is to create loans and fetch only their own loans.

Additionally, add a new functionality to the app and not allow managers to access loan-service.

Congrats! You have successfully finished assignment 2. Have some good rest and see you tomorrow!