PLMS Assignment 4

PLMS - Building Microservices using NodeJS Assignments

Tasks:

1. Creating notification-service.
    a. Setting up the project.
    b. Connecting to MongoDB and RabbitMQ.
    c. Creating schemas for the routes.
    d. Creating the two routes.

2. Implement logging into your services.

Task 1. Creating notification-service

In this task, we will be working on a new service entirely for user notification. It will listen to both the queues together and then save them into the database.

1. Set up the auth-service

Step 1. Set up the service structure like all our previous services and this time we will do the same for `review-service`. Create a new folder called `review-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
h) amqplib

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 8005. Remember to also go over the CORS as well if you are using the frontend.

2. Connecting to MongoDB and RabbitMQ

Step 1. Connect to MongoDB.

This time create a new cluster or an entire new MongoDB server and call it “`PLMS-Review`”. Next we start working on the model. They should include the following:

- userid (string, required)
- loanId (string, required)
- role (enum:[“customer”, “manager”], required)
- type (enum:[“loan_approved”, “loan_rejected”, “loan_created”], required)
- title (string, required)
- message (string, required)

- isRead (Boolean, required)
- createdAt (Date, required)

Now remember to also connect RabbitMQ to your app using the same setup as that from the previous services. Wire the listeners to the app so that the queue gets updated when a new message has been sent.

3. Creating schemas 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) get-notifications-route – to fetch notifications of user.
b) mark-read-route – which marks the notification as read.

There would be no scheme for request to get-notifications-route because this route would directly use the JWT token to fetch the user's notifications.

As for the other route mark-read-route this will involve the marking the notification as read. Again, we do not need any kind of scheme for this route as we will be using the dynamic route to find the notification and then simply mark them as read after checking if the notification was marked by the right person.

4. Creating two routes.

Step 1. Creating get-loan-route.

Under `src/routes`, create `get-notifications-route.js`. Define `GET` request at ("`/api/notifications/all`").

Create a controller for this route where you need to fetch all notification 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 role, and we proceed to fetch all notification for the user.

Step 2. Creating mark-read-route

Under `src/routes`, create `mark-read-route.js`. Define `POST` request at ("`/api/notifications/read/:id`").

Create a controller for this route where you need to mark the user’s notification as read. Here we will first get user’s id and role from the JWT token then divide the customers and manager separately. While all the managers get the same notifications, we will only show the customers’ notifications to their respective customers, and we then send out the customer details.

Once you have successfully implemented everything then test it out on Postman and verify your results.

Task 2. Implement logging into your services

In this task you will be responsible for logging the data. Previously you may have used `console.log` to check where you have an error, which is okay in development environment but not in deployment environment, we simply cannot use `console.log` to make our logger system work.

Step 1. Set up Winston

Install "winston" using npm. We will be using this to log the data properly.

```
npm install winston
```

Step 2. We will now work on a utility for our logger.

Create a new file under `src/untils`, called `logger.js`. In this file, we will be making guidelines for our logger. These guidelines includes level, colors and other standards for logging.

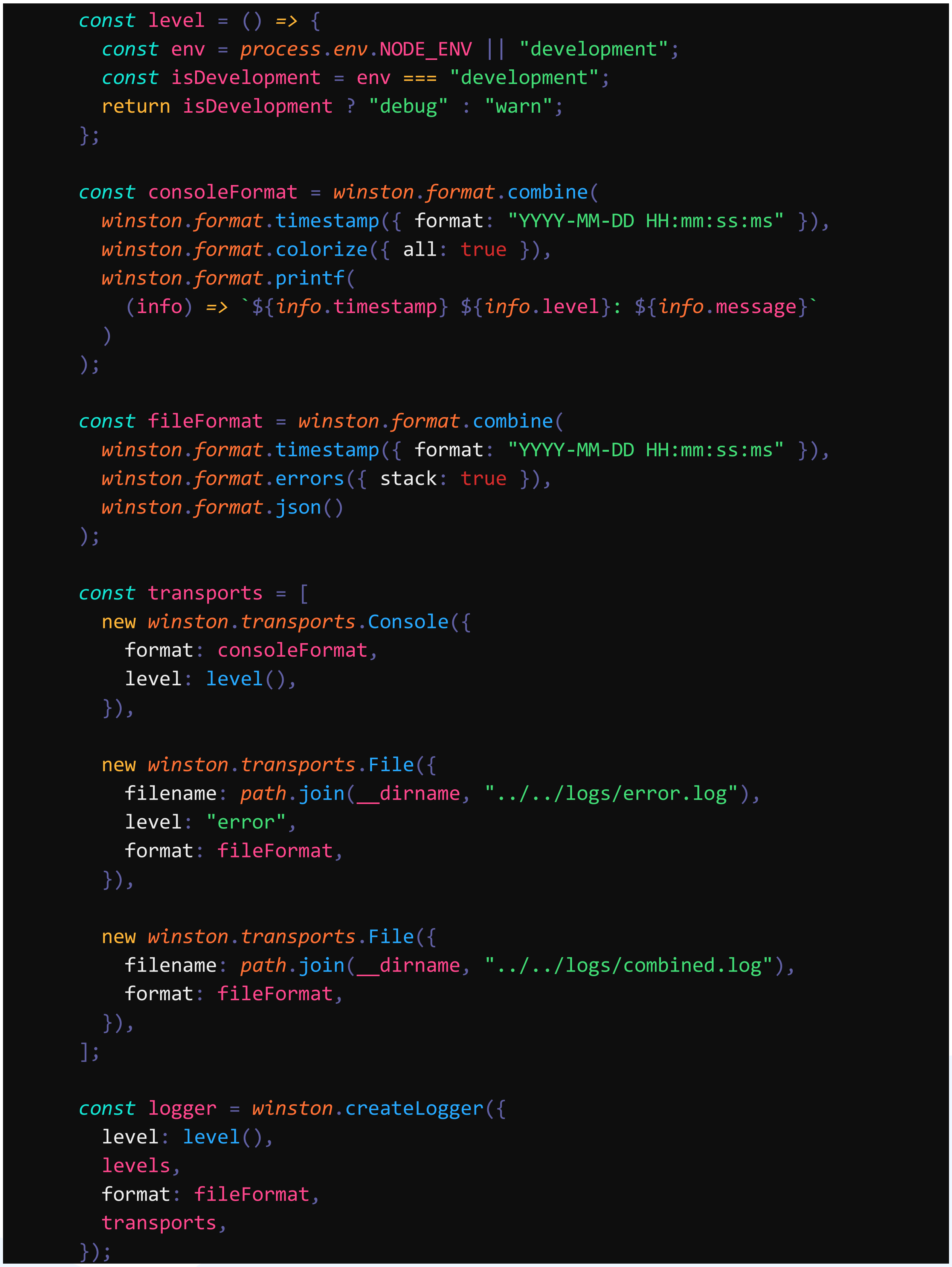
Here is the logger file that you can use:

```
const winston = require("winston");
const path = require("path");

const levels = {
  error: 0,
  warn: 1,
  info: 2,
  http: 3,
  debug: 4,
};

const colors = {
  error: "red",
  warn: "yellow",
  info: "green",
  http: "magenta",
  debug: "white",
};

winston.addColors(colors);
```



```javascript
const level = () => {
  const env = process.env.NODE_ENV || "development";
  const isDevelopment = env === "development";
  return isDevelopment ? "debug" : "warn";
};

const consoleFormat = winston.format.combine(
  winston.format.timestamp({ format: "YYYY-MM-DD HH:mm:ss:ms" }),
  winston.format.colorize({ all: true }),
  winston.format.printf(
    (info) => `${info.timestamp} ${info.level}: ${info.message}`
  )
);

const fileFormat = winston.format.combine(
  winston.format.timestamp({ format: "YYYY-MM-DD HH:mm:ss:ms" }),
  winston.format.errors({ stack: true }),
  winston.format.json()
);

const transports = [
  new winston.transports.Console({
    format: consoleFormat,
    level: level(),
  }),

  new winston.transports.File({
    filename: path.join(__dirname, "../../logs/error.log"),
    level: "error",
    format: fileFormat,
  }),

  new winston.transports.File({
    filename: path.join(__dirname, "../../logs/combined.log"),
    format: fileFormat,
  }),
];

const logger = winston.createLogger({
  level: level(),
  levels,
  format: fileFormat,
  transports,
});
```

```
module.exports = logger;
```

Use this file in all your services. After that you can start changing all your `console` function into logger functions to start logging each action and events that happen in all your service.

Note: *You need to do this for all the services we have worked on so far.*

Step 3. Once all these functionalities are built successfully then you run your service again and you see a new folder called `log`. There would be 2 different files one for all longs and one for only the errors. This way you can pinpoint the errors more effectively.

If you can see the logs in your log file then congratulations, you have successfully created a logger for your service. You can now use this log file to properly demonstrate any errors you get into your services.

Congrats! You have successfully finished assignment 4. You are almost there! Have some good rest and see you tomorrow!