PLMS Assignment 1

PLMS – Building Microservices using NodeJS Assignments

Requirements:

1. IDE of your choice
2. NodeJS (and NPM)
3. POSTMAN

Tasks:

1. Create and set up your project environment.
2. Create bank-service.
    a. Basic set up to run the project.
    b. Create a basic route to test the service.
3. Create boiler plates for all the services.
    a. Creating common error classes.
    b. Creating common middlewares.
4. Create route handlers.
    a. Handle adding data.
    b. Handle fetching data.

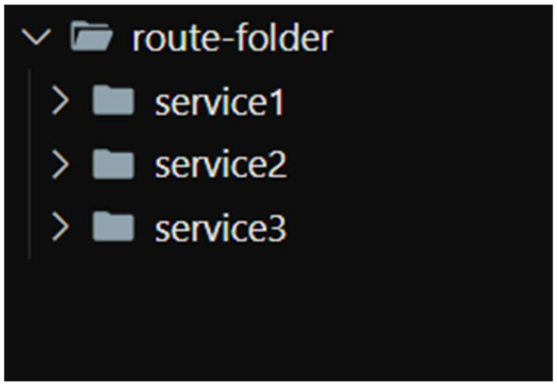
Task 1. Create and set up your project working environment

In this task, we are going to set up a folder where all your services are going to be stored. This folder structure is important because we are going to access all these services in one place which makes it easier to find all services inside of one workspace easily.

Step 1. Create project folder in your filesystem, make a directory named PLMS-microservices (or a name of your choice). Then open this folder in your IDE.

Step 2. Open your terminal with the same path using the command line

```
cd %PATHNAME%/PLMS-microservices
```



```
route-folder
  service1
  service2
  service3
```

__Note:__ Keep `%PATHNAME%` *consistent for all services to simplify workspace navigation. "*`route-folder`*" would be* `PLMS-microservices` *or the name of your choosing and* `services` *would be something we will be working on this project.*

Task 2. Create bank-service

In this task, we are going to work on our first service called bank-service.

1. Basic set up to run the project

Step 1. Create service folder called bank-service change your directory to bank-service.

```
mkdir bank-service
cd bank-service
```

Step 2. Let us now initialize `package.json` file.

```
npm init
```

Fill in details what this service is going to be about.

```
package name: (bank-service) bank-service
version: (1.0.0)
description: A service to add and fetch bank data.
entry point: (index.js)
test command:
git repository:
keywords:
git repository:
keywords:
keywords:
author:
license: (ISC)
type: (commonjs)
```

Step 3. Install required dependencies

a) express
b) cors
c) nodemon
d) dotenv

```
npm install express cors nodemon dotenv
```

Step 4. Now we need to configure start script

```
"scripts": {
  "start": "nodemon src/index.js"
},
```

Note: Create a `src` *directory and place* `index.js` *inside it.*

2. Create a basic route to test the service

Step 1. We now need to establish a folder structure

```
bank-service
├── src
│   ├── controllers      (for service controllers functions)
│   ├── errors           (for error classes)
│   ├── middlewares      (for necessary functions before controllers)
│   ├── routes           (for defining API endpoints)
│   └── index.js         (app of our service)
```

Note: The best practices are subject to change and are also different for different people. Here we are going to follow the routes/middleware/controller.

Step 2. Now we implement our test route called `helloRouter`.

Define a `GET` route at ("`/api/hello`") that responds with:

```
{ "message": "Hello World!" }
```

Note: This is only a test route you can delete after seeing it work properly.

Step 3. Wire up the Express app

The PORT we are going to use in this service we

```
const express = require("express");
const { helloRouter } = require("./routes/hello-route");

const app = express();
app.use(express.json());
app.use(helloRouter);

// Optional: enable CORS for a frontend at http://localhost:5173
const cors = require("cors");
app.use(cors({ origin: "http://localhost:5173" }));

const PORT = 8002;
app.listen(PORT, () => console.log(`bank-service running on ${PORT}`));
```

(Optional) If you are going frontend for testing the app then using cors set the origin to "`http://localhost:5173`". So that it does not have a communication error for Cross-Origin Resource Sharing with the frontend.

Note: the set-up for cors needs to be done with all the services, so these steps are to be repeated with all the services.

Step 4. Test your API using Postman

Send a `GET` request to `http://localhost:8002/api/hello`. If you can see the response as shown in the example given above then congratulations, your end point is ready and functional.

Note: This service cannot be tested on the frontend because we don't have an authentication-service

Task 3. Creating Boilerplates for all services

In this task, we will first create custom error classes. This way we do not have messy error messages that are different and not understandable by the user (in this case the front end). Then we will work on each specific error, they would all extend to this custom error class.

1. Creating common error classes.

Step 1. Create CustomError base class. This class is going to make sure that we have serialized error messages throughout all the services.

Use a constructor that will make sure that each error class has a status code and will return the error message that are easily understood by the front-end app.

Step 2. Let us create specialized error classes that extends the CustomError class. The few error classes that we are going to work on are listed below.

a) Bad Request Error
b) Not Found Error
c) and more error classes in the future assignments.

Each of the errors should follow this standard structure for the error message. So that the data is better parsed by the front-end without any complex logic involved. If we use complex logic here then we will not be able to get the user a satisfying experience.

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

2. Creating common middlewares.

Step 1. Now we will work on the Error-handling middleware. This error handler will send the response as intended or will send out the message as intended and if some new instance of error occurs then that would be handled as well using this middleware.

```
const errorHandler = (err, req, res, next) => {
  if (err instanceof CustomError) {
    return res.status(err.statusCode).send({
      errors: err.serializeErrors(),
    });
  }

  res.status(400).send({
    errors: [
      {
        message: err.message,
      },
    ],
  });
};
```

Step 3. Now we will wire up the middle ware and the not-found error to the express app. The wiring of these two middleware and error classes must be done separately as the errors handled by the middleware are errors that occur during a request while the not-found error is for the request as the request is being sent to a path which is not defined.

```
app.use(async (req, res, next) => {
  next(new NotFoundError());
});

app.use(errorHandler);
```

Note: Remember to wire the error handlers after the routes themselves, otherwise all the routes would start saying not-found error.

Task 4. Create route handlers

In this task, we will start to build a route and add a controller function to check the values and then save them into a variable and then another route to show those data.

1. Handle adding data

Step 1. Under `src/routes`, create `add-bank-route.js`. Define `POST` (“`/api/bank/add`”).

Create a controller for this route where we use the logic to process the data and then store the data. Add a logic that will validate input fields (e.g., ensure minLoanAmount < maxLoanAmount) and then store bank entries in a global array which needs to be defined in `index.js`. Make sure you send a status code 200 “`OK`” message back to the user.

```
global.bank = [];
```

Step 2. Time to test out our API.

Send a `POST` request to `http://localhost:8002/api/bank/add` along with the dummy data given below. If you can see the response with status code 200 and a message that says “`OK`” then congratulations, your end point is ready and functional.

```
        {
            "bankName": "Bank of America",
            "logo": "https://purepng.com/public/uploads/large/purepng.com-bank-of-america-logologobrand-logoiconslogos-251519939688snesl.png",
            "minLoanAmount": 5000,
            "maxLoanAmount": 100000,
            "minInterestRate": 4.99,
            "maxInterestRate": 15.69,
            "minCreditScore": 660,
            "termLength": 48,
            "processingFee": 80,
            "rating": 4.9
        },
        {
            "bankName": "Chase",
            "logo": "https://logos-world.net/wp-content/uploads/2020/11/Chase-
Logo-2005-present.jpg",
            "minLoanAmount": 4000,
            "maxLoanAmount": 50000,
            "minInterestRate": 5.99,
            "maxInterestRate": 18.69,
            "minCreditScore": 680,
            "termLength": 60,
            "processingFee": 90,
            "rating": 4.5
        },
        {
            "bankName": "State Farm",
            "logo": "https://logos-world.net/wp-content/uploads/2021/10/State-
Farm-Logo-700x394.png",
            "minLoanAmount": 3500,
            "maxLoanAmount": 40000,
            "minInterestRate": 6.99,
            "maxInterestRate": 25.69,
            "minCreditScore": 660,
            "termLength": 60,
            "processingFee": 100,
            "rating": 4.7
        },
        {
            "bankName": "US Bank",
            "logo": "https://purepng.com/public/uploads/large/purepng.com-us-
bank-logologobrand-logoiconslogos-251519940417jbyiu.png",
            "minLoanAmount": 2000,
            "maxLoanAmount": 35000,
            "minInterestRate": 9.55,
            "maxInterestRate": 15.69,
            "minCreditScore": 580,
            "termLength": 48,
            "processingFee": 50,
            "rating": 4
        }
```

Note: This is just dummy data, and you can always edit this data to test out different functionalities of your express app.

2. Handle fetching data.

Step 1. Under `src/routes`, create `get-bank-route.js`. Define `GET` (“`/api/bank/getbank`”).

Create a controller for this route which does not need to process anything as this data/service is public data and hence we want it to be accessible by all the users. This controller will then send back a response with the bank details all in an array.

```
{
    "banks": [
        {
            // bank 1
        },
        {
            // bank 2
        },
        {
            // bank 3
        },
    ]
}
```

Step 2. Time to test out our API.

Send a `GET` request to `http://localhost:8002/api/bank/getbank`. If you can see the response as shown in the example given above then congratulations, your end point is ready and functional.

Congrats! You have successfully finished assignment 1. Have some good rest and see you tomorrow!