Final Project: CodeCraftHub: Building Personalized Learning Platform for Developers

Skills Network

In this project, you'll leverage the power of Generative AI and a diverse array of technologies to transform your vision into a reality.

Learning objectives

After completing this lab, you will be able to perform the following tasks:

- Design and develop software applications using Generative AI
- Create documentation for the code with Generative AI
- Create test cases with Generative AI
- Deploy the deployable application designed and developed entirely with Generative AI

Prerequisites

1. You must be familiar with at least one programming language and know software architectures well.
2. You must have a GitHub account.
3. You must be comfortable using the IDE.
4. You must be familiar with using Postman.
5. You must be familiar with Docker applications and commands.

Setting up the AI classroom

In case you need help with the Interface/classroom, please click here

Gathering requirements for the development of the learning platform

Using GenAI, gather requirements for developing the server-side learning platform by asking the following questions:

- The effectiveness of the responses depends on the prompts provided.
- The prompts provided here are suggestions; you can use your discretion to change them.
- You should also use your subject matter expertise and judgment as a developer.
- It is your responsibility to check the correctness of the responses.

Type the following prompt to give the context and the objective:

> **Note:** Make sure you are using the ChatGPT Mini 40 model.

```
I want to create a personalized online learning platform, starting with the server side. Recommend an optimal design and architecture for the backend, and help me gather all the functional and non-fu
```

▼ Click here to view the sample response generated

Creating a personalized online learning platform involves careful consideration of both functional and non-functional requirements, as well as an optimal design and architecture for the backend. Below is a detailed breakdown of these aspects.

Recommended Design and Architecture

1. Architecture Style
    - Microservices Architecture: This allows for independent deployment and scaling of different components (e.g., user management, course management, payment processing).
2. Technology Stack
    - Programming Language: Node.js or Python (Django/Flask)
    - Database: PostgreSQL for relational data, MongoDB for unstructured data
    - API: RESTful API or GraphQL for communication between frontend and backend
    - Authentication: OAuth 2.0 or JWT (JSON Web Tokens) for secure user authentication
    - Containerization: Docker for easy deployment and scaling
    - Cloud Provider: AWS, Azure, or Google Cloud for hosting
3. Components
    - User Service: Handles user registration, authentication, and profile management.
    - Course Service: Manages courses, lessons, assignments, and quizzes.
    - Payment Service: Integrates with payment gateways for course purchases.
    - Notification Service: Sends emails or notifications for course updates and reminders.
    - Analytics Service: Tracks user progress and engagement metrics.

Functional Requirements

1. User Management
    - User registration and login
    - Profile management (update details, change password)
    - Role-based access control (admin, instructor, student)
2. Course Management
    - Create, update, and delete courses
    - Enroll users in courses
    - Upload and manage course materials (videos, documents)
    - Assign quizzes and track completion
3. Payment Processing
    - Integrate with payment gateways (e.g., Stripe, PayPal)
    - Handle subscriptions or one-time payments
    - Issue refunds if necessary
4. Notifications
    - Email notifications for course updates
    - Reminders for upcoming deadlines or quizzes
5. Analytics and Reporting
    - Track user progress through courses
    - Generate reports for instructors and admins

Non-Functional Requirements

1. Performance
    - The system should handle at least 1000 concurrent users without degradation of performance.
2. Scalability
    - Ability to scale services independently based on demand.
3. Security
    - Ensure data protection through encryption (SSL/TLS) and secure storage of user data.
4. Usability
    - The system should have an intuitive user interface that is easy to navigate.
5. Maintainability
    - Code should be well-documented and adhere to coding standards for easier maintenance.
6. Availability
    - The system should have an uptime of 99.9% to ensure accessibility for users.

By following this architecture and gathering these requirements, you can set a solid foundation for your personalized online learning platform.

Next Steps

Consider prototyping the user interface and defining the database schema based on the requirements outlined above.

Disclaimer: Your response might vary.

For the following exercise, microservices architecture is the recommended architecture.

Choosing the architecture and components

Type the following in the prompt to choose the microservices architecture and the appropriate server-side components.

```
I want to use a microservices architecture for the server side of a learning platform. The key services should include:
Personalized learning recommendations,
Interactive coding exercises
Real-time feedback to help developers improve their skills and knowledge.
Recommend the complete set of microservices and supporting server-side components I should include, along with their roles and how they should interact.
```

The response will comprise the recommended services.

▼ Click here to view the sample response generated

To implement a microservices architecture for your online learning platform with the specified services, you will need to design various components that work together seamlessly. Here's a breakdown of the necessary components for each service you mentioned, along with some additional considerations:

1. **Personalized Learning Recommendations Service**
    - **Recommendation Engine**: This component uses algorithms to analyze user behavior, preferences, and learning history to suggest relevant courses and content.
        - **Data Sources**: Collect data from user interactions, course completions, and feedback.
        - **Machine Learning Models**: Implement models (e.g., collaborative filtering, content-based filtering) to enhance recommendations.
    - **User Profile Management**: Store user profiles, including learning preferences and progress.
    - **API Gateway**: Facilitate communication between the recommendation service and other services (like user management and course management).
2. **Interactive Coding Exercises Service**
    - **Exercise Repository**: A database or storage system to manage coding exercises, including problem statements, solutions, and metadata (difficulty level, tags).
    - **Execution Environment**: A sandboxed environment to run user-submitted code securely. This could be implemented using containerization (e.g., Docker) to isolate execution.
    - **Assessment Engine**: Automatically evaluate user submissions against predefined test cases and provide feedback.
    - **User Interface**: A simple web interface for users to interact with coding exercises, submit solutions, and view feedback.
3. **Real-time Feedback Service**
    - **WebSocket Server**: Implement a WebSocket server to provide real-time communication between users and the feedback service, allowing for instant feedback on coding exercises.
    - **Feedback Engine**: Analyze user submissions and provide constructive feedback based on coding standards, best practices, and performance metrics.
    - **Notification System**: Notify users about feedback, updates, or new exercises via real-time alerts.
4. **Common Components Across Services**
    - **API Gateway**: An API gateway can manage requests to different microservices, handle load balancing, and provide a unified entry point for the frontend.
    - **Service Registry and Discovery**: Use tools like Consul or Eureka to keep track of service instances and their locations, enabling dynamic discovery of services.
    - **Database Management**: Decide whether each service will have its own database (which is often recommended in microservices) or if some services will share a database. This will depend on the specific data requirements.
    - **Authentication and Authorization**: Implement a centralized authentication service (e.g., using OAuth 2.0 or JWT) to manage user sessions and permissions across all services.
    - **Logging and Monitoring**: Use tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Prometheus and Grafana for logging, monitoring, and visualizing application performance and health.
5. **Deployment and Infrastructure**
    - **Containerization**: Use Docker to containerize each microservice, which simplifies deployment and scaling.
    - **Orchestration**: Use Kubernetes or Docker Swarm to manage the deployment, scaling, and operation of your containerized applications.
    - **Cloud Services**: Consider using cloud platforms (AWS, Google Cloud, Azure) for hosting your services, leveraging their managed services for databases, storage, and networking.
6. **Security Measures**
    - **Input Validation**: Ensure all user inputs are validated and sanitized to prevent code injection attacks.
    - **Rate Limiting**: Implement rate limiting on APIs to prevent abuse and ensure fair usage.
    - **Data Encryption**: Encrypt sensitive data both in transit (using HTTPS) and at rest.

Conclusion
By structuring your online learning platform with these microservices and components, you can achieve a scalable and maintainable architecture that supports personalized learning recommendations, interactive coding exercises, and real-time feedback. Focus on clear interfaces between services, robust data management, and security practices to ensure a seamless user experience.

Disclaimer: Your response might vary.

Create the user service

User Management Service are a pivotal service. You will create that service using Node.js and MongoDB.

Type the following in the prompt:

```
I want to create a User Management Service for my learning platform. I will use Node.js and MongoDB. Please recommend an optimal project structure, including key folders, files, and configurations, f
```

This prompt's response will be similar to the following description.

▼ Click here to view the sample response generated

This project structure for the User Management Service in a learning platform using Node.js and MongoDB follows best practices for scalability, maintainability, and security. Key components include:

- **src/:** Contains the main source code, organized into subfolders for configuration, controllers, models, routes, middlewares, services, and utilities.
- **config/:** Configuration files for database connections and server settings.
- **controllers/:** Contains the business logic for user-related operations.
- **models/:** Defines the MongoDB schema for users.
- **routes/:** API endpoints for user management.
- **middlewares/:** Custom middleware for authentication and authorization.
- **services/:** Contains service layer logic to separate business logic from route handling.
- **utils/:** Utility functions for logging and error handling.
- **tests/:** Contains unit tests for the controllers and services.
- **.env:** Holds environment variables for sensitive configurations.
- **README.md:** Provides documentation for setup and usage.

```
user-management-service/
├── src/
│   ├── config/
│   │   ├── db.js              # Database connection configuration
│   │   ├── server.js          # Server configuration and initialization
│   │   └── env.js             # Environment variable configurations
│   ├── controllers/
│   │   ├── userController.js   # User-related business logic
│   ├── models/
│   │   ├── userModel.js       # User schema and model definition
│   ├── routes/
│   │   ├── userRoutes.js      # API routes for user management
│   ├── middlewares/
│   │   ├── authMiddleware.js    # Middleware for authentication and authorization
│   ├── services/
│   │   ├── userService.js       # Service layer for user-related operations
│   ├── utils/
│   │   ├── logger.js            # Logging utility
│   │   └── errorHandler.js       # Centralized error handling
│   └── app.js                   # Main application file
├── tests/
│   ├── userController.test.js   # Unit tests for user controller
│   ├── userService.test.js       # Unit tests for user service
├── .env                          # Environment variables
├── .gitignore                    # Files/directories to ignore in git
├── package.json                  # Project dependencies and scripts
└── README.md                     # Project documentation and setup instructions
```

Disclaimer: Your response might vary.

If the provided response doesn't align with the expected project structure, consider refining your prompt by incorporating more specific questions.

In the IDE, create the recommended directory structure and add the files as necessary.

Insert code into each file

Please include the following statement in the prompt:

```
I have set up the User Management Service project using Node.js and MongoDB based on the recommended folder structure. Please give me the code that is to be included in each of the files, ensuring it
```

The goal is to leverage Generative AI for generating the entire code. After manually setting up the files in the IDE based on the previous instructions, you can now include the provided code. Make sure to prompt it to provide the intended fields.

▼ Click here to view the sample response generated

This code provides a complete User Management Service structure using Node.js and MongoDB. It includes configurations for connecting to MongoDB, setting up the server, defining user models, controllers for user registration and login, routes for handling API requests, middleware for authentication, utility functions for logging and error handling, and the main application file to tie everything together.

```
// src/config/db.js
const mongoose = require('mongoose');
const connectDB = async () => {
    try {
        await mongoose.connect(process.env.MONGO_URI, {
            useNewUrlParser: true,
            useUnifiedTopology: true
        });
        console.log('MongoDB connected successfully.');
    } catch (error) {
        console.error('MongoDB connection failed:', error);
        process.exit(1);
    }
};
module.exports = connectDB;
// src/config/server.js
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const initServer = () => {
    const app = express();
    app.use(cors());
    app.use(bodyParser.json());
    return app;
};
module.exports = initServer;
// src/config/env.js
require('dotenv').config();
// src/models/userModel.js
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
```

```
    email: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    },
    role: {
        type: String,
        enum: ['student', 'instructor', 'admin'],
        default: 'student'
    },
    createdAt: {
        type: Date,
        default: Date.now
    }
});
const User = mongoose.model('User', userSchema);
module.exports = User;
// src/controllers/userController.js
const User = require('../models/userModel');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
// Register a new user
exports.registerUser = async (req, res) => {
    const { username, email, password } = req.body;
    try {
        const hashedPassword = await bcrypt.hash(password, 10);
        const newUser = new User({ username, email, password: hashedPassword });
        await newUser.save();
        res.status(201).json({ message: 'User registered successfully.' });
    } catch (error) {
        res.status(500).json({ error: 'Registration failed.' });
    }
};
// Login user
exports.loginUser = async (req, res) => {
    const { email, password } = req.body;
    try {
        const user = await User.findOne({ email });
        if (!user) return res.status(404).json({ error: 'User not found.' });
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) return res.status(401).json({ error: 'Invalid credentials.' });
        const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET || 'secret', { expiresIn: '1h' });
        res.status(200).json({ token });
    } catch (error) {
        res.status(500).json({ error: 'Login failed.' });
    }
};
// src/routes/userRoutes.js
const express = require('express');
const { registerUser, loginUser } = require('../controllers/userController');
const router = express.Router();
router.post('/register', registerUser);
router.post('/login', loginUser);
module.exports = router;
// src/middlewares/authMiddleware.js
const jwt = require('jsonwebtoken');
const authMiddleware = (req, res, next) => {
    const token = req.header('Authorization')?.split(' ')[1];
    if (!token) return res.status(401).json({ error: 'Access denied.' });
    try {
        const verified = jwt.verify(token, process.env.JWT_SECRET);
        req.user = verified;
        next();
    } catch (error) {
        res.status(400).json({ error: 'Invalid token.' });
    }
};
module.exports = authMiddleware;
// src/services/userService.js
const User = require('../models/userModel');
```

```
// Function to find user by ID
exports.findUserById = async (userId) => {
    return await User.findById(userId);
};
// src/utils/logger.js
const winston = require('winston');
const logger = winston.createLogger({
    level: 'info',
    format: winston.format.json(),
    transports: [
        new winston.transports.File({ filename: 'error.log', level: 'error' }),
        new winston.transports.Console()
    ]
});
module.exports = logger;
// src/utils/errorHandler.js
const logger = require('./logger');
const errorHandler = (err, req, res, next) => {
    logger.error(err);
    res.status(500).json({ error: 'Something went wrong.' });
};
module.exports = errorHandler;
// src/app.js
require('dotenv').config();
const express = require('express');
const connectDB = require('./config/db');
const initServer = require('./config/server');
const userRoutes = require('./routes/userRoutes');
const errorHandler = require('./utils/errorHandler');
const app = initServer();
connectDB();
app.use('/api/users', userRoutes);
app.use(errorHandler);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
// .env
MONGO_URI=mongodb://root:<mongo_password>@<mongo_host>:27017/usermngtservice?authSource=admin
PORT=5000

// package.json
{
  "name": "user-management-service",
  "version": "1.0.0",
  "description": "A Node.js service for managing users with authentication and authorization.",
  "main": "src/app.js",
  "scripts": {
    "start": "node src/app.js",
    "dev": "nodemon src/app.js",
    "test": "jest --coverage"
  },
  "keywords": [
    "node",
    "express",
    "user",
    "authentication",
    "api"
  ],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "bcrypt": "^5.1.1",
    "cors": "^2.8.5",
    "dotenv": "^16.6.1",
    "express": "^4.19.2",
    "jsonwebtoken": "^9.0.2",
    "mongoose": "^8.3.1",
    "morgan": "^1.10.0",
    "winston": "^3.17.0"
  },
  "devDependencies": {
    "jest": "^29.7.0",
    "nodemon": "^3.1.4",
    "supertest": "^6.3.4"
  }
```

```
}
```

Disclaimer: Your response might vary.

▼ Click here to view the steps to push your work to GitHub

Remember, any modifications you make in the lab environment won't be saved. If you plan to step away, use the following steps to ensure your changes push to GitHub:

Ensure that you are currently inside the Project directory.

1. Navigate to the project directory by using: cd <project dir>.
2. Set up your Git configuration:
   - Run: `git config --global user.email "yourgithub@email.com"`
   - Run: `git config --global user.name "name"`
3. Add your changes to the staging area:
   - Run: `git add .`
4. Commit your changes with a descriptive message:
   - Run: `git commit -m "Adding temporary changes to Github"`
5. The first step is to generate an access token from GitHub.com. Follow directions from the lab Generate GitHub personal access token and copy the access token as a password in the upcoming exercises.
6. Push your changes to the Git repository:
   - Run: `git push`
7. A prompt in the terminal will prompt you to enter your GitHub username and password (your previously created Personal Access Token from Step 5).

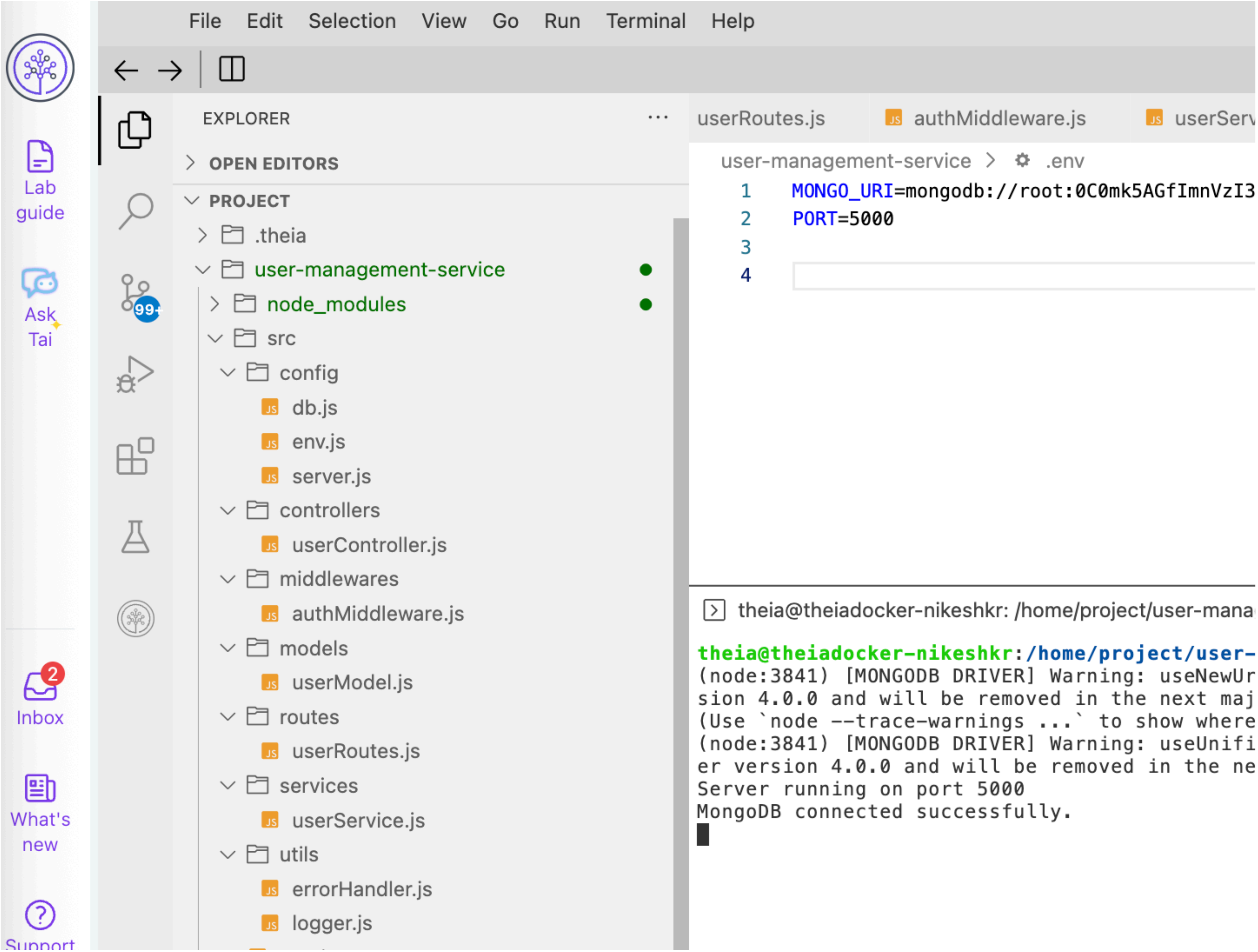
These steps ensure your work is safely stored in GitHub, allowing you to continue when you return to the lab environment.

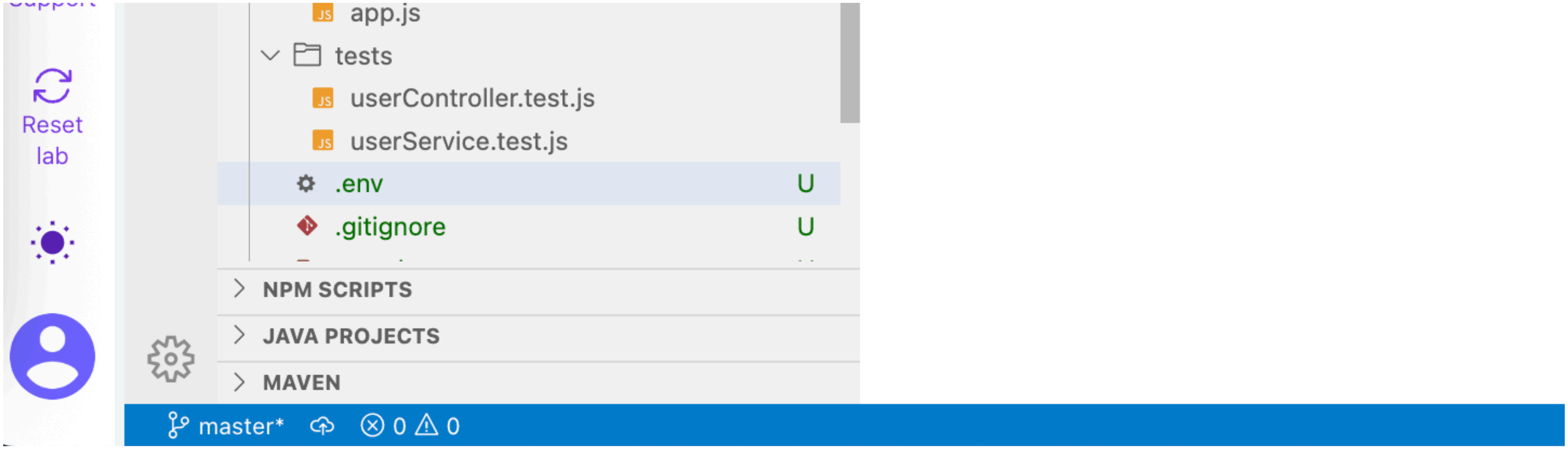
Test the application

To run and test your user management service, you can follow these steps:

1. To start your MongoDB server, use to the pre-work lab for the final project.
2. Start your Node.js server: In your project directory, open a new terminal or command prompt window and run the command `node src/app.js` to start your Node.js server.

   Note: Before running the command `node src/app.js`, first install your packages by running `npm install` or `npm i` to start your Node.js server successfully.


File Edit Selection View Go Run Terminal Help
Lab guide
Ask Tai
Inbox
What's new
Support
EXPLORER
OPEN EDITORS
PROJECT
.theia
user-management-service
node_modules
src
config
db.js
env.js
server.js
controllers
userController.js
middlewares
authMiddleware.js
models
userModel.js
routes
userRoutes.js
services
userService.js
utils
errorHandler.js
logger.js
userRoutes.js
authMiddleware.js
user-management-service > .env
1 MONGO_URI=mongodb://root:0C0mk5AGfImnVzI3
2 PORT=5000
3
4
theia@theiadocker-nikeshkr: /home/project/user-mana
theia@theiadocker-nikeshkr:/home/project/user-
(node:3841) [MONGODB DRIVER] Warning: useNewUr
sion 4.0.0 and will be removed in the next maj
(Use `node --trace-warnings ...` to show where
(node:3841) [MONGODB DRIVER] Warning: useUnifi
er version 4.0.0 and will be removed in the ne
Server running on port 5000
MongoDB connected successfully.



Generating a database to test the application

You have the code now but you have not created the database yet. You will now use Generative AI to populate the database.

```
Can you please provide the user data in JSON format?
```

▼ Click here to view the sample response generated

Here is an example of user data in JSON format:

```
[
  {
    "username": "john_doe",
    "email": "johndoe@example.com",
    "password": "Password123!"
  },
  {
    "username": "john_smith",
    "email": "johnsmith@example.com",
    "password": "password123!"
  }
]
```

You can modify this JSON structure to include additional fields or customize it according to your specific user data requirements.

Disclaimer: Your response might vary.

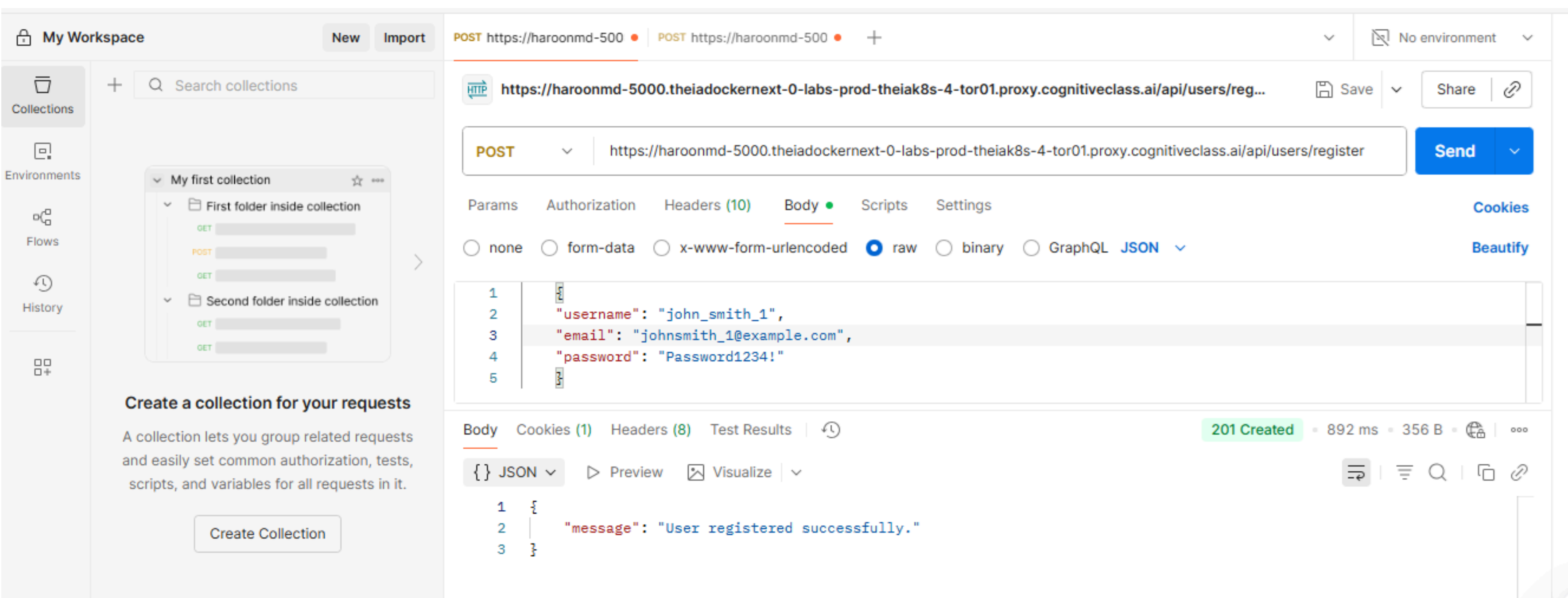
Test the API endpoints: You can use tools like [Postman]() or make API requests from your front-end application to test the API endpoints. Here are some example requests:

Note: The testing instructions provided are based on the response codes received in our example. Your implementation may produce different codes depending on your logic. Please adjust your testing steps accordingly to match your code's output.

1. User Registration: Send a POST request to `http://localhost:5000/api/users/register` with the following request body:

```
{
"username": "john_smith_1",
"email": "johnsmith_1@example.com",
"password": "Password1234!"
}
```

You should receive a response with status code **201** and the message "User registered successfully".



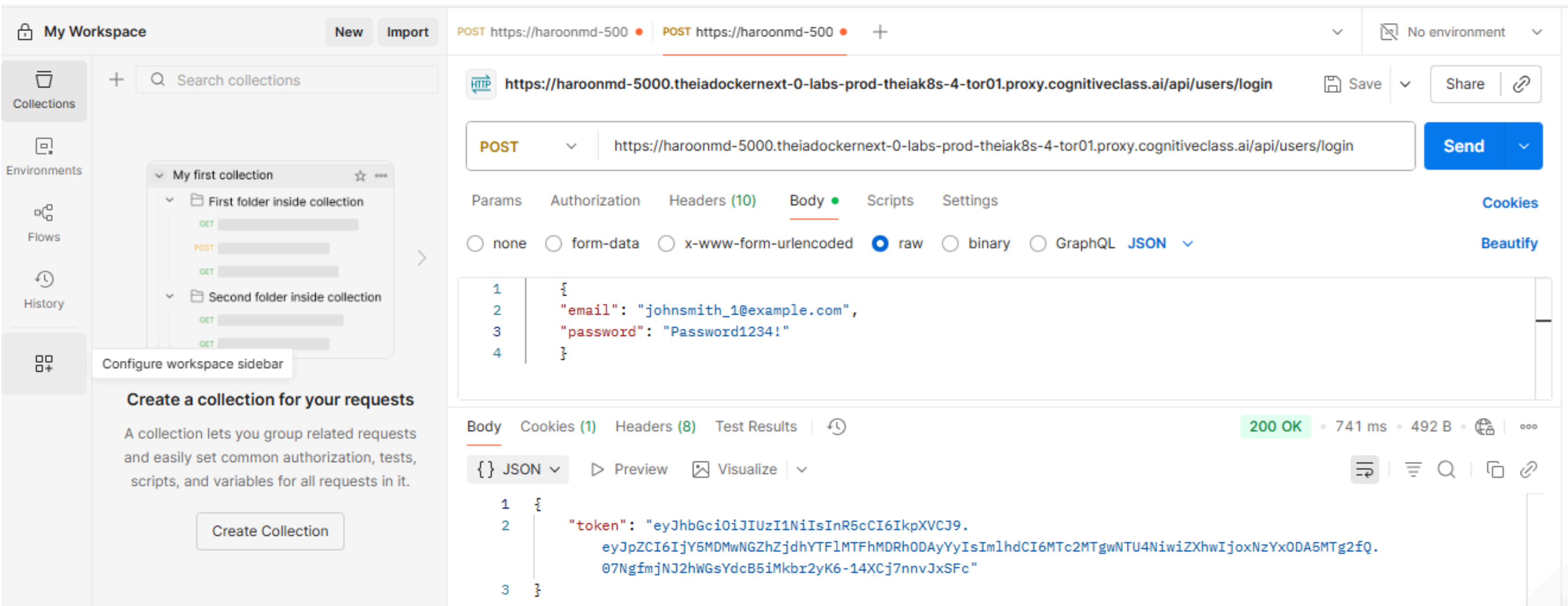
You can verify the endpoint using a curl command in the terminal window.

```
curl -X POST -H "Content-Type: application/json" -d '{"username": "john_smith_1", "email": "johnsmith_1@example.com", "password": "password1234!"}' http://localhost:5000/api/users/register
```

2. User Login: Send a POST request to `http://localhost:5000/api/users/login` with the following request body:

```
{
"email": "johnsmith_1@example.com",
"password": "Password1234!"
}
```

You should receive a response with status code **200** and a **JSON Web Token (JWT)** in the response body.



You can also verify the endpoint by using a curl command in the terminal window.

```
curl -X POST -H "Content-Type: application/json" -d '{"email": "johnsmith_1@example.com", "password": "Password1234!"}' http://localhost:5000/api/users/login
```

Verify the data in the MongoDB database: You can use a MongoDB client or run MongoDB queries to verify the data was added to the database.

Remember to handle errors, implement additional API endpoints, and thoroughly test your User Management Service to ensure it meets your requirements.

Note: Ensure all your work pushes to your GitHub repository. Click here for detailed steps.

▼ Click here to view the steps to push your work to GitHub

Remember, any modifications you make in the lab environment won't be saved. If you plan to step away, use the following steps to ensure your changes are pushed to GitHub:

Ensure that you are currently inside the project directory.

1. Navigate to the project directory by performing cd <project dir>.
2. Set up your Git configuration:
    - Run: `git config --global user.email "yourgithub@email.com"`
    - Run: `git config --global user.name "name"`
3. Add your changes to the staging area:
    - Run: `git add .`
4. Commit your changes with a descriptive message:
    - Run: `git commit -m "Adding temporary changes to Github"`
5. The first step is to generate an access token from GitHub.com. Follow the lab named Generate GitHub personal access token and copy the access token as a password in the upcoming exercises.
6. Push your changes to the Git repository:
    - Run: `git push`
7. A prompt in the terminal will prompt you to enter your GitHub username and password (your previously created Personal Access Token from Step 5).

These steps ensure your work is safely stored in GitHub, allowing you to continue when you return to the lab environment.

Code review

You must provide the code in each file you created and get them reviewed.

```
Can you review the code below?
```

And then paste the code that you want to get reviewed.

▼ Click here to view the sample response generated

The provided code sets up the user routes for the User Management Service using Express. It imports the necessary controllers for handling user registration and login. The syntax for destructuring the imported functions is corrected to ensure proper function assignment. Additionally, comments are added to clarify the purpose of each route.

```
// src/routes/userRoutes.js
const express = require('express');
const { registerUser, loginUser } = require('../controllers/userController');
const router = express.Router();
// Route for user registration
router.post('/register', registerUser);
// Route for user login
router.post('/login', loginUser);
module.exports = router;
```

Disclaimer: Your response might vary.

Documentation

You need to provide documentation and comments for all the code written.

```
Can you provide documentation and comments for the code to make it readable?
```

You will use the prompt iteratively with the content of each file.

Dockerfile

You need to bundle the application in Docker. Type the following prompt to create a Dockerfile that bundles the application and MongoDB server in a container.

```
Can you provide the docker file to bundle the application and the MongoDB server in a container?
```

▼ Click here to view the sample response generated

Certainly! To bundle your Node.js application along with a MongoDB server in a Docker container, you'll need to create a `Dockerfile` for your Node.js application and a `docker-compose.yml` file to manage both services (the Node.js app and MongoDB). Below are the steps and code snippets to help you set this up.

Step 1: Create a `Dockerfile` for the Node.js Application

Create a file named `Dockerfile` in the root of your project directory (where your `package.json` is located):

```
# Use the official Node.js image as a base
FROM node:16
# Set the working directory inside the container
WORKDIR /usr/src/app
# Copy package.json and package-lock.json
COPY package*.json ./
# Install dependencies
RUN npm install
# Copy the rest of the application code
COPY . .
# Expose the application port
EXPOSE 5000
# Command to run the application
CMD ["node", "src/server.js"]
```

Step 2: Create a `docker-compose.yml` File

Now, create a `docker-compose.yml` file in the root of your project directory to define the services:

```
version: '3.8'
services:
  user-service:
    build:
      context: .  # Path to the Dockerfile
      dockerfile: Dockerfile
    ports:
      - "5000:5000"  # Map port 5000 of the container to port 5000 on the host
    environment:
      MONGODB_URI: mongodb://mongo:27017/user-service  # MongoDB connection string
      JWT_SECRET: your_jwt_secret  # Replace with your secret
      PORT: 5000
    depends_on:
      - mongo  # Wait for the MongoDB service to start
  mongo:
    image: mongo:latest  # Use the official MongoDB image
    ports:
      - "27017:27017"  # Map port 27017 of the container to port 27017 on the host
    volumes:
      - mongo-data:/data/db  # Persist MongoDB data
volumes:
  mongo-data:  # Named volume for MongoDB data persistence
```

Step 3: Building and Running the Containers

1. **Build the Docker Images**:
   Open a terminal in your project directory and run the following command to build your Docker images:

   ```
   docker-compose build
   ```

2. **Run the Containers**:
   After building, you can start the services with:

   ```
   docker-compose up
   ```

3. **Access the Application**:
   Your Node.js application should now be accessible at `http://localhost:5000`, and MongoDB will be running in the background.

Step 4: Stopping the Containers

To stop the running containers, you can use:

```
docker-compose down
```

Conclusion

With this setup, you have a Dockerized Node.js application running alongside a MongoDB server. The `docker-compose.yml` file simplifies the management of multiple services, allowing you to run everything with a single command. If you have any further questions or need assistance with specific configurations, feel free to ask!

Disclaimer: Your response might vary.

You may be prompted for a specific procedure if your response doesn't show how to deploy.

▼ Click here to view the steps to push your work to GitHub

Remember, any modifications you make in the lab environment won't be saved. If you plan to step away, use the following steps to ensure your changes are pushed to GitHub:

Ensure that you are currently inside the project directory.

1. Navigate to the project directory by performing cd <project dir>.
2. Set up your Git configuration:
   - Run: `git config --global user.email "yourgithub@email.com"`
   - Run: `git config --global user.name "name"`
3. Add your changes to the staging area:
   - Run: `git add .`
4. Commit your changes with a descriptive message:
   - Run: `git commit -m "Adding temporary changes to Github"`
5. The first step is to generate an access token from GitHub.com. Follow the lab named [Generate GitHub personal access token] and copy the access token as a password in the upcoming exercises.
6. Push your changes to the Git repository:
   - Run: `git push`
7. A prompt in the terminal will prompt you to enter your GitHub username and password (your previously created Personal Access Token from Step 5).

These steps ensure your work is safely stored in GitHub, allowing you to continue when you return to the lab environment.

Checklist

At this stage:

1. You now have a running application that offers CRUD microservices for the User Management Service.
2. The code has undergone a thorough review and is comprehensively documented.
3. The service has been successfully deployed within a Docker container.
4. Proceed to push all the code to your GitHub repository.

Summary

- You have successfully gathered requirements for a user management service of a programming-focused learning platform using Generative AI.
- You have explored vital aspects such as fundamental features, user-friendly design, interactive functionalities, and an efficient folder structure.

Subsequent actions involve:

- Employing MongoDB for user data
- Producing Node, Express, and Mongoose code
- Conducting a detailed code review with comprehensive documentation

Congratulations! You have successfully leveraged Generative AI to build a learning platform by choosing Microservices Architecture, Node.js, and MongoDB.

Author(s)

Sapthashree K S