

# Final Project: CodeCraftHub: Building Personalized Learning Platform for Developers



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:

I want to create a personalized online learning platform. I want to start with the server side. Recommend a good design and architecture for the

▼ [Click here to view the sample response generated](#)

To create a personalized online learning platform with a robust server-side architecture, you should consider the following design and architecture components:

### 1. Architecture Pattern

- **Microservices Architecture:** This approach allows you to build your application as a collection of loosely coupled services, making it easier to scale and maintain. Each service can handle different functionalities such as user management, course management, and payment processing.
- **RESTful API:** Design your backend using RESTful APIs to allow communication between the client-side and server-side. This will enable you to create a flexible and scalable system.

### 2. Technology Stack

- **Programming Language:** Choose a language that fits well with your team's expertise. Popular choices include Node.js (JavaScript), Python (Django or Flask), Ruby (Ruby on Rails), or Java (Spring Boot).
- **Database:** For a personalized learning platform, consider using a relational database like PostgreSQL or MySQL for structured data, or a NoSQL database like MongoDB for flexibility in handling unstructured data.
- **Authentication:** Implement OAuth 2.0 or JWT (JSON Web Tokens) for secure user authentication and authorization.

### 3. Key Components

- **User Management:** Implement a user registration and profile management system, allowing users to create profiles, track progress, and manage their learning paths.

- **Course Management:** Create a service to handle course creation, modification, and deletion. This should also include the ability to upload and manage multimedia content (videos, PDFs, quizzes).
- **Personalization Engine:** Use algorithms to recommend courses and content based on user preferences, past behavior, and learning goals.
- **Analytics and Reporting:** Integrate analytics to track user engagement, course completion rates, and other metrics to improve the learning experience.

#### 4. Deployment and Scalability

- **Containerization:** Use Docker to package your services, making it easier to deploy and manage them in different environments.
- **Cloud Services:** Consider deploying your application on cloud platforms like AWS, Google Cloud, or Azure, which offer scalability and various services (like managed databases, serverless functions, etc.).
- **Load Balancing:** Implement load balancers to distribute traffic evenly across your services and ensure high availability.

#### 5. Security Considerations

- **Data Encryption:** Use HTTPS for secure data transmission and encrypt sensitive data in your database.
- **Regular Updates:** Keep your software and dependencies up to date to protect against vulnerabilities.

#### 6. Development Tools

- **Version Control:** Use Git for version control to manage changes to your codebase.
- **CI/CD Pipeline:** Set up a continuous integration/continuous deployment pipeline to automate testing and deployment processes.

#### 7. Documentation and Community

- **API Documentation:** Use tools like Swagger or Postman to document your APIs for easier integration and usage by front-end developers.
- **Community Support:** Engage with online communities (like Stack Overflow, GitHub, or relevant forums) for support and collaboration.

#### Conclusion

Starting with a microservices architecture, choosing the right technology stack, and focusing on key components such as user management and personalization will help you build a strong foundation for your online learning platform. Make sure to prioritize security and scalability as you develop your server-side architecture

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 would like to use a microservices architecture for the server side. These are the services I want to be able to provide.  
Personalized learning recommendations,  
Interactive coding exercises  
Real-time feedback to help developers improve their skills and knowledge.  
What are the various components I should have?

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.

Create the user service

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

Type the following in the prompt:

I would like to create the user service. I would like to use Node.js and MongoDB for this project. How do I create a project structure?

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

▼ Click here to view the sample response generated

Creating a user service using Node.js and MongoDB involves setting up a well-organized project structure that promotes maintainability and scalability. Below is a recommended project structure along with explanations for each component.

Recommended Project Structure

```
user-service/
├── src/
│   ├── config/
│   │   ├── db.js           # Database connection configuration
│   │   └── server.js       # Server configuration (port, middleware)
│   ├── controllers/
│   │   └── userController.js # User-related request handlers
│   ├── models/
│   │   └── userModel.js     # User schema and model definitions
│   ├── routes/
│   │   └── userRoutes.js    # API route definitions for user-related endpoints
│   ├── middleware/
│   │   └── authMiddleware.js # Authentication and authorization middleware
│   ├── services/
│   │   └── userService.js   # Business logic related to user operations
│   ├── utils/
│   │   └── validation.js    # Input validation functions
│   └── app.js               # Main application entry point
├── tests/
│   └── user.test.js         # Unit and integration tests for user service
├── .env                    # Environment variables (e.g., DB URI, JWT secret)
└── .gitignore              # Files and directories to ignore in Git
```

```
└─ package.json      # Project metadata and dependencies
  README.md          # Project documentation
```

## Explanation of Each Component

1. **src/**: This is the main source directory where all your application code resides.
  - **config/**: Contains configuration files for your application.
    - `db.js`: Responsible for connecting to MongoDB using Mongoose or the native MongoDB driver.
    - `server.js`: Configures the Express server, middleware, and routes.
  - **controllers/**: Contains the logic for handling incoming requests and sending responses.
    - `UserController.js`: Defines functions for user-related operations like registering, logging in, updating profiles, etc.
  - **models/**: Contains Mongoose schemas and models that define the structure of your data.
    - `userModel.js`: Defines the user schema (e.g., username, email, password, etc.) and exports the Mongoose model.
  - **routes/**: Defines the API endpoints for your user service.
    - `userRoutes.js`: Sets up routes for user-related actions (e.g., `POST /register`, `POST /login`).
  - **middleware/**: Contains middleware functions for handling requests.
    - `authMiddleware.js`: Implements authentication checks (e.g., verifying JWT tokens).
  - **services/**: Contains the business logic for user operations.
    - `userService.js`: Contains functions that interact with the user model (e.g., creating a user, finding a user by ID).
  - **utils/**: Contains utility functions that can be reused across the application.
    - `validation.js`: Functions for validating user input (e.g., checking email format, password strength).
  - **app.js**: The main entry point for your application, where you initialize the server, connect to the database, and set up middleware and routes.
2. **tests/**: Contains test files for unit and integration tests to ensure your user service works as expected.
3. **.env**: A file to store environment variables, such as your MongoDB URI and JWT secret. Use the `dotenv` package to load these variables in your application.
4. **.gitignore**: Specifies files and directories that should not be tracked by Git (e.g., `node_modules`, `.env`).
5. **package.json**: Contains metadata about your project, including dependencies and scripts. Use `npm init` to create this file and add dependencies like Express, Mongoose, and any testing libraries.
6. **README.md**: A documentation file that provides an overview of your project, installation instructions, and usage examples.

## Setting Up the Project

### 1. Initialize the Project:

```
mkdir user-service
cd user-service
npm init -y
```

### 2. Install Dependencies:

```
npm install express mongoose dotenv bcryptjs jsonwebtoken
```

### 3. Create the Directory Structure:

Use the structure outlined above to create directories and files.

#### 4. Example Code Snippets:

- **db.js:**

```
const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config();
const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (error) {
    console.error('MongoDB connection failed:', error.message);
    process.exit(1);
  }
};
module.exports = connectDB;
```

- **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 },
});
const User = mongoose.model('User', userSchema);
module.exports = User;
```

- **userController.js:**

```
const User = require('../models/userModel');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
// Register 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({ message: 'Error registering user', error });
  }
};
// Add more controller functions for login, etc.
```

#### 5. Run the Application:

In your `app.js`, set up the server and connect to the database:

```
const express = require('express');
const connectDB = require('./config/db');
const userRoutes = require('./routes/userRoutes');
const app = express();
app.use(express.json());
// Connect to the database
connectDB();
// Define routes
app.use('/api/users', userRoutes);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

### Conclusion

This project structure and setup will help you create a well-organized user service using Node.js and MongoDB. You can expand upon this foundation by adding more features, such as password reset functionality, email verification, and more robust error handling.

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:

Please give me the code that is to be included in each of the files.

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](#)

Sure! Below is the complete code for each file in the user service project structure using Node.js and MongoDB.

#### 1. src/config/db.js

```
const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config();
const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (error) {
    console.error('MongoDB connection failed:', error.message);
    process.exit(1);
  }
};
module.exports = connectDB;
```

#### 2. 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 },
}, { timestamps: true });
const User = mongoose.model('User', userSchema);
module.exports = User;
```

#### 3. src/controllers/userController.js

```
const User = require('../models/userModel');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
// Register a new user
const registerUser = async (req, res) => {
  const { username, email, password } = req.body;
  try {
    const existingUser = await User.findOne({ email });
    if (existingUser) {
```

```

        return res.status(400).json({ message: 'User already exists' });
    }
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();
    res.status(201).json({ message: 'User created successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
// Login user
const loginUser = async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }
    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });
    res.json({ token });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
// Add more user-related functions (e.g., get user, update user, etc.)
module.exports = { registerUser, loginUser };

```

#### 4. 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);
// Add more routes for getting user, updating user, etc.
module.exports = router;

```

#### 5. src/app.js

```

const express = require('express');
const connectDB = require('./config/db');
const userRoutes = require('./routes/userRoutes');
const dotenv = require('dotenv');
dotenv.config();
const app = express();
app.use(express.json());
connectDB();
app.use('/api/users', userRoutes);
module.exports = app;

```

#### 6. src/server.js

```

const app = require('./app');
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});

```

## 7. Environment Variables in .env

Create a .env file in the root of your project:

```
MONGODB_URI=mongodb://localhost:27017/user-service
JWT_SECRET=your_jwt_secret
PORT=5000
```

## 8. tests/user.test.js (Example Test File)

You can create a test file to ensure your user service works as expected. Here's a simple example using Jest and Supertest:

```
const request = require('supertest');
const app = require('../src/app');
const mongoose = require('mongoose');
const User = require('../src/models/userModel');
beforeAll(async () => {
  await mongoose.connect(process.env.MONGODB_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  });
});
afterAll(async () => {
  await User.deleteMany({});
  await mongoose.connection.close();
});
describe('User Service', () => {
  it('should register a new user', async () => {
    const response = await request(app)
      .post('/api/users/register')
      .send({
        username: 'testuser',
        email: 'test@example.com',
        password: 'password123',
      });
    expect(response.statusCode).toBe(201);
    expect(response.body.message).toBe('User created successfully');
  });
  it('should login an existing user', async () => {
    await request(app)
      .post('/api/users/register')
      .send({
        username: 'testuser',
        email: 'test@example.com',
        password: 'password123',
      });
    const response = await request(app)
      .post('/api/users/login')
      .send({
        email: 'test@example.com',
        password: 'password123',
      });
    expect(response.statusCode).toBe(200);
    expect(response.body.token).toBeDefined();
  });
});
});
```

## Running the Service

1. Start your MongoDB server (ensure MongoDB is running).
2. Run the service using:

```
npx nodemon src/server.js
```

3. Run your tests using:

```
npx jest
```



### Conclusion

This complete code setup provides a solid foundation for your user service using Node.js and MongoDB. You can expand upon this by adding more features, such as user profile updates, password resets, and additional validations. If you have any further questions or need assistance with specific functionalities, feel free to ask!

### Note :

1. To obtain the secret key, execute the following command in the terminal.

```
node -e "console.log(require('crypto').randomBytes(32).toString('hex'))"
```

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`

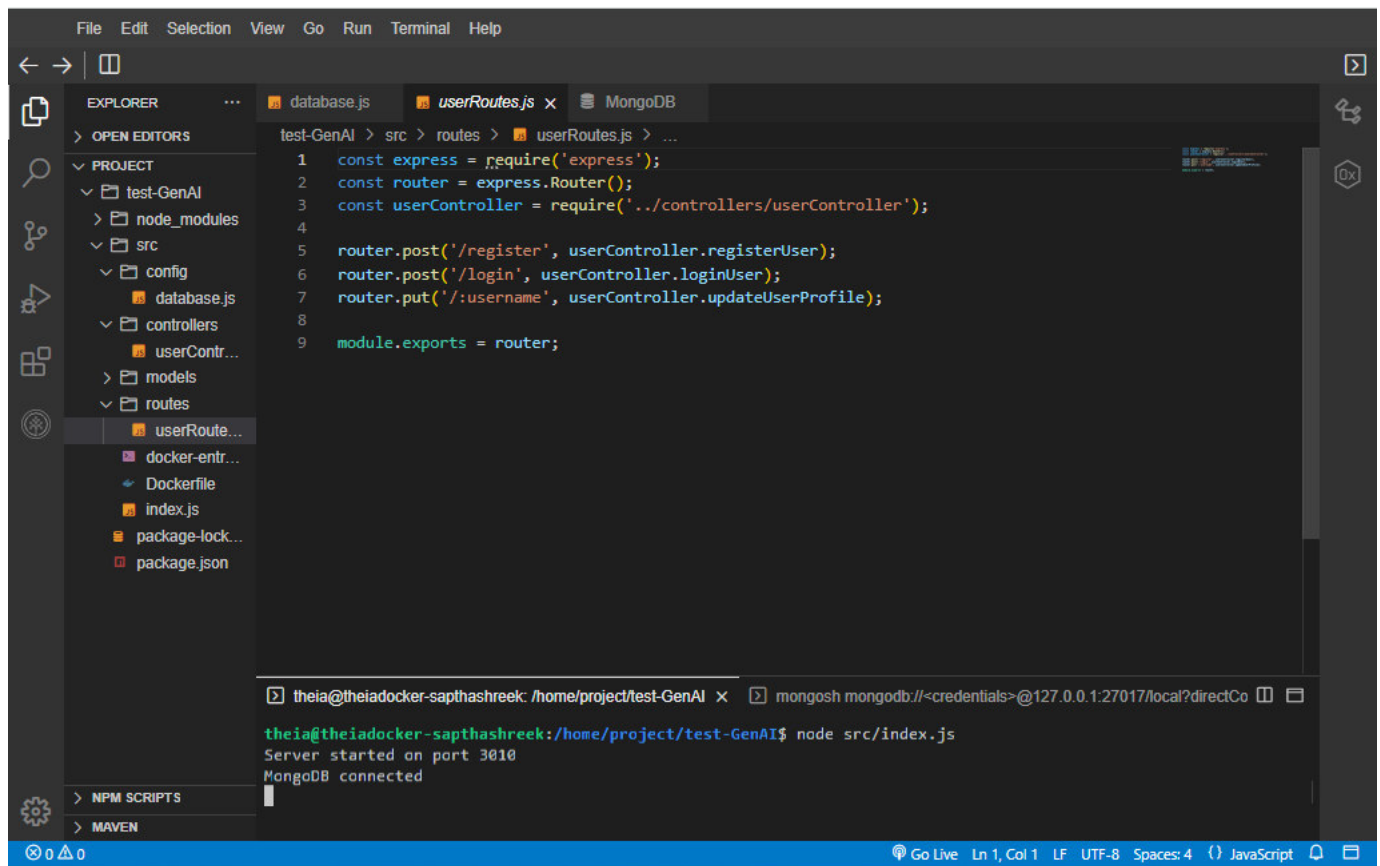
7. A prompt in the terminal will prompt you to enter your GitHub username and password (your previously created Personal Access Token from Step 5).

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, refer 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/index.js` to start your Node.js server.



## 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:

```
[
  {
    "name": "John Doe",
    "password": "password123"
  },
  {
    "name": "Jane Smith",
    "password": "password456"
  }
]
```

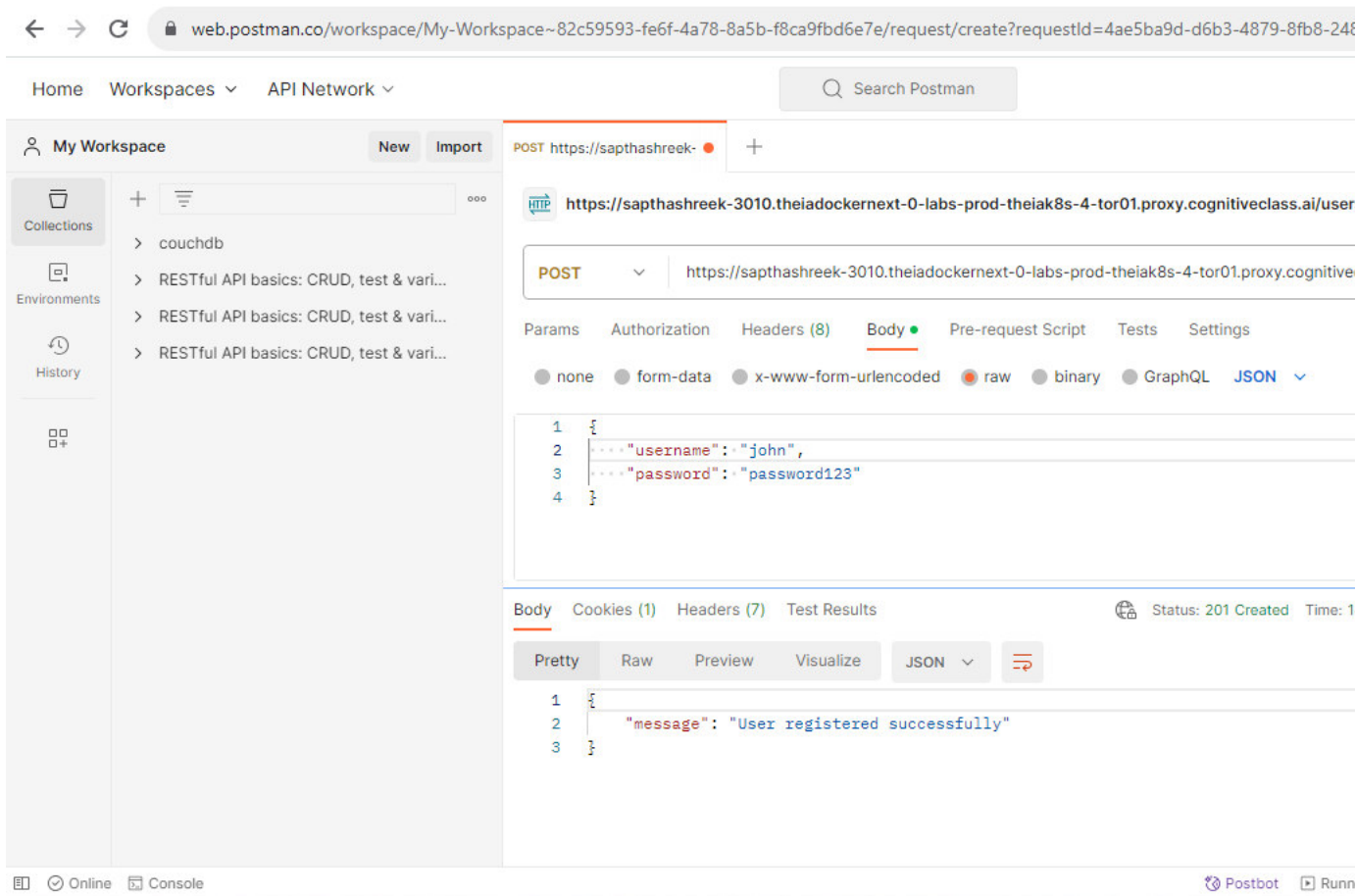
You can modify this JSON structure to include additional fields or customize it according to your specific user data requirements.

**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:

- User registration: Send a POST request to `http://localhost:3000/users/register` with the following request body:

```
{
  "username": "john",
  "password": "password123"
}
```

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", "password": "password123"}' http://localhost:3000/users/register
```

- User login: Send a POST request to `http://localhost:3000/users/login` with the following request body:

```
{
  "username": "john",
  "password": "password123"
}
```

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

The screenshot shows the Postman interface with a workspace named 'My Workspace'. A POST request is configured to the endpoint `https://saphashreek-3010.theiadockernext-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user`. The request body is a JSON object: `{ "username": "john", "password": "password123" }`. The response is displayed in the 'Body' tab, showing a JSON object with a 'token' field: `{ "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFvaG4iLCJpYXQiOiJlZ3MDI2Mjc0MDgsImV4cCI6MTcwMjYzMTAwOH0.UCFqzX9C7x6eyZa32KMT08232hJYQFhCoR6a5YsRDPc" }`. The status is 200 OK.

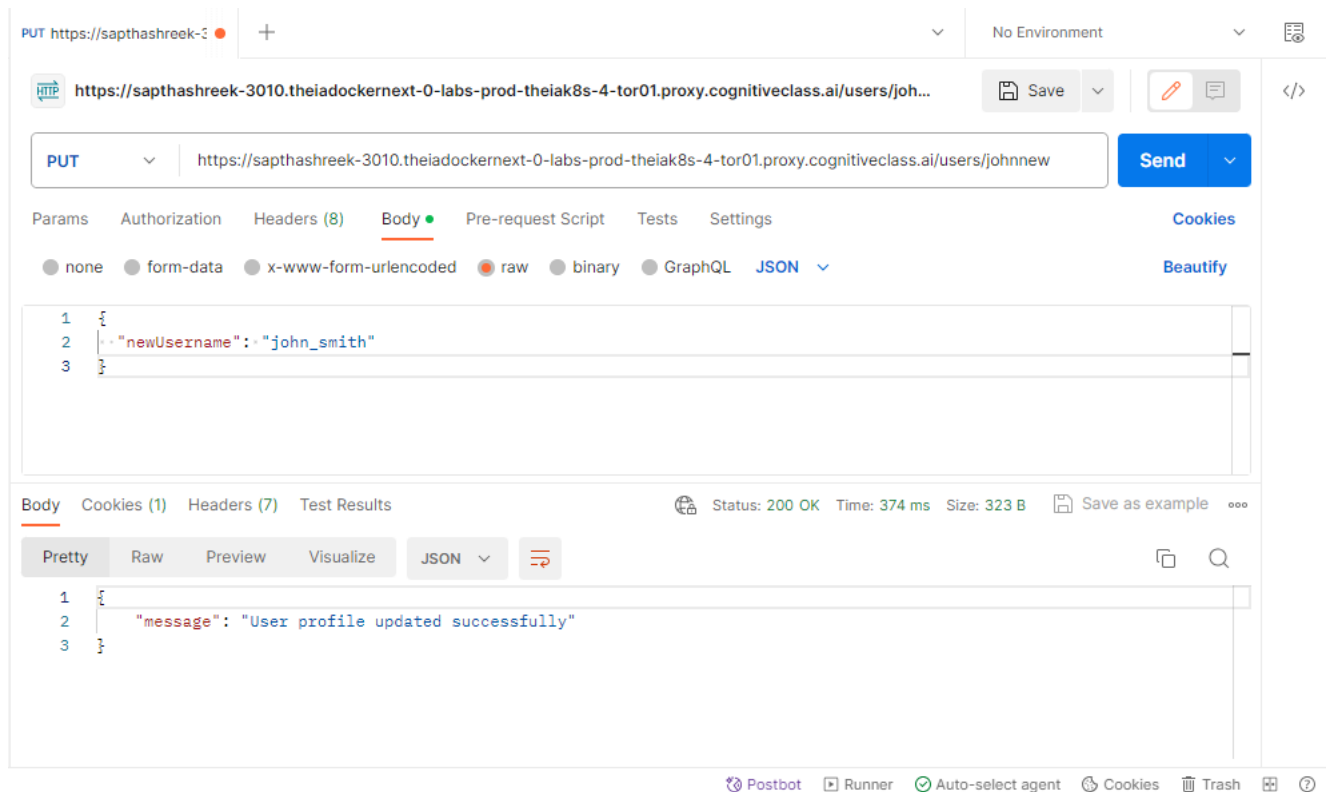
You can also verify the endpoint by using a curl command in the terminal window.

```
curl -X POST -H "Content-Type: application/json" -d '{"username": "john", "password": "password123"}' http://localhost:3000/users/login
```

- User profile update: Send a PUT request to `http://localhost:3000/users/john_smith` (Replace "john\_doe" with the actual username) with the following request body:

```
{
  "newUsername": "john_smith"
}
```

You should receive a response with status code **200** and the message "User profile updated successfully".



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

```
curl -X PUT -H "Content-Type: application/json" -d '{"newUsername": "john_smith"}' http://localhost:3000/users/john_smith
```

**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.

## 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.

## Code review

You have to iteratively give the code in each file you have created and get them reviewed.

```
Kindly review the userRoutes.js file code.
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
router.post('/register', userController.registerUser);
router.post('/login', userController.loginUser);
router.put('/:username', userController.updateUserProfile);
module.exports = router;
```

▼ [Click here to view the sample response generated](#)

Your userRoutes.js file looks mostly good, but there are a couple of things to consider for improvement and completeness. Here's a review along with some suggestions:

### Code Review

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
router.post('/register', userController.registerUser);
router.post('/login', userController.loginUser);
router.put('/:username', userController.updateUserProfile);
module.exports = router;
```

### Suggestions for Improvement

#### 1. Dynamic Route Parameter:

- Using `:username` as a route parameter is fine, but consider using a user ID (`:id`) instead. Usernames can change, whereas IDs are typically persistent and unique.
- Example: `router.put('/:id', userController.updateUserProfile);`

## 2. Middleware for Authentication:

- If you are implementing user profile updates, you might want to ensure that the user is authenticated before allowing them to update their profile. Consider adding an authentication middleware to protect this route.
- Example:

```
const authMiddleware = require('../middleware/authMiddleware');
router.put('/:id', authMiddleware, userController.updateUserProfile);
```

## 3. Error Handling:

- Make sure that your controller functions (e.g., `updateUserProfile`) handle errors properly and return appropriate status codes and messages.

## 4. Route Documentation:

- It's a good practice to document your API routes, either in comments or using tools like Swagger, to make it easier for other developers (or yourself in the future) to understand how to use the API.

## 5. Additional Routes:

- Depending on your requirements, you might want to add more routes, such as fetching user profiles, deleting users, etc.

Revised `userRoutes.js`

Here's a revised version of your `userRoutes.js` with the suggestions applied:

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
const authMiddleware = require('../middleware/authMiddleware'); // Optional: if using authentication
// Register a new user
router.post('/register', userController.registerUser);
// Login an existing user
router.post('/login', userController.loginUser);
// Update user profile (protected route)
router.put('/:id', authMiddleware, userController.updateUserProfile);
// Optional: Add more routes for getting user profile, deleting user, etc.
// router.get('/:id', authMiddleware, userController.getUserProfile);
// router.delete('/:id', authMiddleware, userController.deleteUser);
module.exports = router;
```

## Conclusion

Your `userRoutes.js` file is a good start for managing user-related routes. By implementing the suggested improvements, you can enhance the security and usability of your API. If you need further assistance with the controller or middleware implementations, feel free to ask!

# 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:



## 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!

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

