22 SEPTEMBER 2024

A-LEVEL MAJOR PROJECT REPORT

E-COMMERCE PROGRASSIVE WEB APP

Technical

STACK

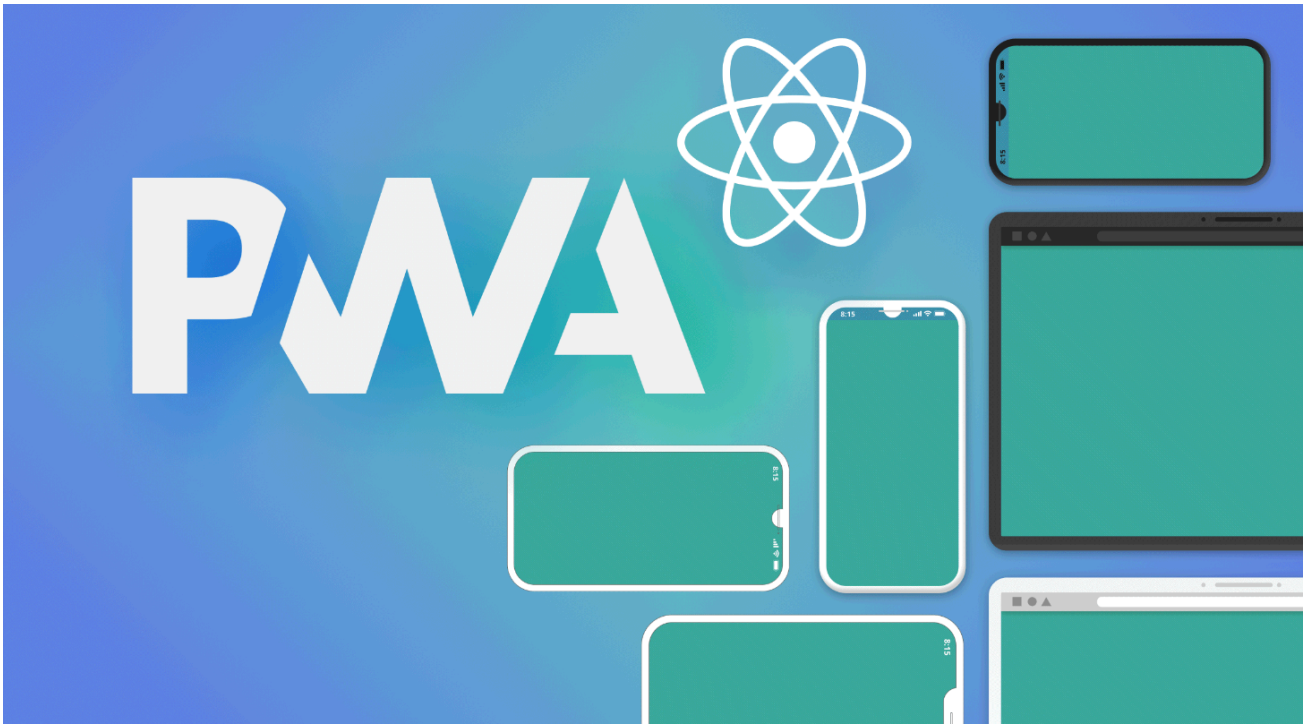
NODEJS

REACT

MYSQL

Introduction

In this project, I have developed a web application using **React** for the frontend, **Node.js** for the backend, **MySQL** as the database, **GitHub** for version control, **Render** for hosting the backend, **GitHub Pages** for hosting the frontend, and **Cloudinary** as a service for image hosting. This stack allows for seamless development, scalability, and efficient deployment. Each technology plays a significant role in the overall architecture, ensuring that the application is reliable, scalable, and easy to maintain.



PROGRESSIVE WEB APPS

Progressive Web Apps (PWAs) represent a significant evolution in the way web applications are built and deployed. They combine the best features of web and mobile applications, providing users with a seamless experience that is fast, reliable, and engaging. By leveraging modern web capabilities, PWAs offer a native app-like experience directly through the browser, enabling access on any device without the need for installation from app stores.

Key Features of PWAs

1. **Responsive Design**: PWAs are designed to work on any device, whether it's a desktop, tablet, or smartphone. The layout adjusts dynamically, ensuring a consistent user experience across various screen sizes.
2. **Offline Access**: One of the standout features of PWAs is their ability to function offline or in low-network conditions. Using service workers, they

can cache resources and data, allowing users to continue accessing the app even without an internet connection.

3. **Performance**: PWAs are built to be fast. They load quickly, even on slow networks, by pre-caching key resources. This not only enhances the user experience but also reduces bounce rates.
4. **App-Like Experience**: PWAs mimic the look and feel of native applications, complete with smooth transitions, animations, and interactions. Users can add them to their home screens, and they often have a full-screen mode that hides browser chrome.
5. **Automatic Updates**: Unlike traditional apps that require users to manually update, PWAs automatically refresh in the background, ensuring that users always have access to the latest version without any intervention.
6. **Secure**: PWAs are served over HTTPS, providing a secure connection that protects user data and ensures the integrity of the application.
7. **Engagement**: Features like push notifications allow PWAs to engage users in real-time, sending updates and alerts similar to native applications, which can significantly improve user retention.
8. **Discoverability**: Since PWAs are web-based, they are easily discoverable through search engines. This increases the chances of attracting new users compared to traditional apps that require installation.
9. **Cost-Effectiveness**: Developing a PWA can be more cost-effective than creating separate apps for different platforms (iOS, Android). A single codebase can serve all users, reducing development and maintenance efforts.
10. **Analytics and Tracking**: PWAs can utilize web analytics tools to track user engagement and behavior, providing valuable insights that can inform future development and marketing strategies.

Advantages of PWAs

1. **Broader Reach**: PWAs can be accessed by anyone with a web browser, increasing potential user base significantly.
2. **Lower Development Costs**: With a single codebase, businesses can save on development and maintenance costs, compared to managing multiple native apps.
3. **Enhanced User Engagement**: Features like push notifications and offline capabilities can lead to higher user engagement and retention rates.

In summary, Progressive Web Apps represent a revolutionary approach to application development, combining the advantages of both web and mobile applications. By providing a fast, reliable, and engaging user experience, PWAs can help businesses reach broader audiences and enhance user engagement. However, it is essential to consider the limitations and ensure that the specific needs of the application align with the capabilities of PWAs. As technology continues to evolve, PWAs will likely become an increasingly integral part of the digital landscape, shaping the future of web applications.

PROJECT ARCHITECTURE

The architecture of the system is divided into three layers:

1. **Backend (Node.js & Express.js)**: This layer handles the business logic, API endpoints, and database interactions.
2. **Database (MySQL)**: Stores order details, customer information, and item-related data.
3. **Frontend (React.js)**: Provides the user interface for customers to view and manage orders. It interacts with the backend via REST APIs.

4. **GitHub for Version Control**: GitHub is used to manage the project's version control, allowing for efficient collaboration, tracking changes, and maintaining a history of updates and features.
5. **Render for Hosting the Backend**: Render is a cloud service used to deploy and manage the backend server. It provides a scalable, easily manageable environment for running the Node.js backend in a production setting.
6. **GitHub Pages for Hosting the Frontend**: GitHub Pages will host the React web application. This is a cost-effective solution that allows the app to be accessible online with minimal effort for deployment and management.
7. **Aiven for storing users data:** For storing MySQL data, I chose **Aiven** as the managed service provider. Aiven offers fully managed cloud services, including MySQL, which provides several benefits such as scalability, security etc.

Why React, Node.js, and MySQL?

React

React is a powerful JavaScript library for building user interfaces, particularly for single-page applications (SPAs). It allows developers to create large web applications where data can change without reloading the page, enhancing the user experience. React's component-based structure promotes code reusability and faster development cycles. The virtual DOM implementation makes the rendering process efficient, ensuring that updates are applied quickly without reloading the entire page. Moreover, React's support for hooks and state management helps in building interactive and dynamic UIs with ease.

Node.js

Node.js is a runtime environment built on Chrome's V8 JavaScript engine, designed for building fast and scalable network applications. With its non-blocking, event-driven architecture, Node.js is ideal for I/O-heavy tasks like API calls, file handling, and database queries. Since both the frontend and backend can be written in JavaScript, Node.js allows for a unified language across the stack, simplifying the development process. Furthermore, its asynchronous capabilities enable it to handle numerous concurrent requests, making it highly suitable for real-time applications, including those that require constant data updates, like an e-commerce order system.

MySQL

MySQL is one of the most popular open-source relational database management systems (RDBMS). It is known for its reliability, robustness, and ease of use. The relational nature of MySQL makes it well-suited for managing structured data, like orders, customers, and items in an e-commerce system. Its support for ACID (Atomicity, Consistency, Isolation, Durability) ensures the integrity of transactions, which is crucial for handling payment and order processing in an e-commerce platform. MySQL also offers efficient indexing and querying capabilities, ensuring that data retrieval remains fast and scalable, even as the database grows.

Version Control with GitHub

For version control, I used **GitHub**. Version control is critical in modern development to track changes, collaborate with others, and ensure that code is versioned properly. Some key reasons why GitHub was used:

- **Collaboration**: GitHub makes it easy to collaborate with other developers by allowing code to be shared, reviewed, and merged through pull requests.
- **Branching**: GitHub's branching features allowed for a smooth development process. Feature branches were created for each task, and once completed and tested, they were merged into the main branch.
- **Code History and Rollbacks**: With GitHub, every change is logged, and past versions of the project can be accessed if something goes wrong. This ensures that if any bugs arise, I can easily revert to a previous state of the code.

GitHub served as the central repository for this project, ensuring that the code was safely stored and could be reverted to previous versions in case of errors. Regular commits and branching made it easy to track progress and test features separately.

Hosting Backend on Render

For backend hosting, I used **Render**, a cloud platform that allows developers to deploy web applications, APIs, and other services seamlessly. Render was chosen because of:

- **Easy Deployment**: Render simplifies the deployment process with continuous deployment from GitHub. Any changes pushed to the GitHub repository are automatically deployed to the live server.
- **Scalability**: Render allows for scalable applications, ensuring that as the number of users grows, the backend can handle more requests without any significant degradation in performance.
- **Managed Infrastructure**: Render manages the infrastructure, freeing up developers from having to manage servers, networking, and configurations.

In this project, Render is used to host the Node.js backend, making it accessible through API endpoints. The service was configured to automatically redeploy the application whenever changes were made to the code on GitHub.

Frontend Hosting on GitHub Pages

The React frontend is hosted on **GitHub Pages**, which is a static site hosting service provided by GitHub. Some reasons for choosing GitHub Pages include:

- **Cost-Effective**: GitHub Pages is free to use, making it an excellent choice for deploying frontend applications.
- **Seamless Integration with GitHub**: Since the project was already stored in a GitHub repository, deploying the frontend to GitHub Pages was quick and easy.
- **Automatic Updates**: Whenever changes are pushed to the main branch, GitHub Pages automatically reflects those changes on the live site.

By hosting the React frontend on GitHub Pages, users can access the application directly from their browser without needing to set up a local environment.

Cloudinary for Image Hosting

For hosting images, I used **Cloudinary**, a cloud-based image and video management platform. Some reasons for choosing Cloudinary include:

- **Global CDN**: Cloudinary serves images via a content delivery network (CDN), ensuring fast loading times for users no matter their location.
- **Image Manipulation**: Cloudinary provides image manipulation capabilities, allowing for resizing, cropping, and format conversion on the fly.
- **Storage Management**: Cloudinary offers reliable cloud storage for images, ensuring that they are always available and can be accessed quickly.

In this project, Cloudinary is used to store and serve the images for items ordered by the users. The image URLs are fetched from Cloudinary and displayed in the frontend, ensuring high performance and availability.

Storing MySQL Data with Aiven

For storing MySQL data, I chose **Aiven** as the managed service provider. Aiven offers fully managed cloud services, including MySQL, which provides several benefits such as scalability, security, and ease of use. Aiven's platform enables the seamless deployment and management of MySQL databases on various cloud platforms like AWS, Google Cloud, and Azure.

Key advantages of using Aiven for MySQL storage:

- **Fully Managed**: Aiven handles all aspects of managing MySQL databases, including setup, scaling, maintenance, backups, and security updates. This removes the burden of managing these tasks manually, allowing me to focus on the development of the application.
- **Scalability**: Aiven allows for easy scaling of database resources, which is crucial as the application grows. MySQL databases can be resized without causing any downtime, ensuring the application can handle more users and data as needed.

- **High Availability**: With Aiven, I can deploy MySQL in a highly available configuration with automated failovers and backups, ensuring that the data is always available and protected against data loss.
- **Security**: Aiven takes care of security by providing built-in encryption for data both at rest and in transit. It also offers role-based access control and integrates with various authentication mechanisms, ensuring that only authorized users can access the data.

Using Aiven for MySQL storage significantly streamlined the data management process. It offers a reliable and scalable environment for storing user data, order information, and product details without the overhead of managing the infrastructure myself. This allowed me to focus more on building the core functionalities of the application while knowing the data is secure and readily available.

Advantages:

1. **Scalability**: The combination of React, Node.js, and MySQL enables the application to scale both horizontally and vertically, handling more users and orders as the system grows.
2. **Component Reusability**: React's component-based architecture allows for reusing UI elements across different parts of the application, speeding up development and maintaining consistency.
3. **High Performance**: React's virtual DOM and Node.js' event-driven architecture ensure the application performs efficiently, handling multiple requests and updates without lag.

4. **Separation of Concerns**: Using React for the frontend, Node.js for the backend, and MySQL for the database ensures a clear separation of concerns, making the codebase easier to maintain and develop.
5. **Asynchronous Processing**: Node.js excels at handling asynchronous operations, such as database queries and API calls, improving the app's performance and user experience.
6. **Cross-Platform Compatibility**: The system can run on multiple platforms (Linux, Windows, Mac), providing flexibility in deployment environments.
7. **Extensibility**: React, Node.js, and MySQL are supported by vast ecosystems of libraries and tools, allowing the application to be extended with additional features, such as real-time updates, analytics, and third-party integrations.
8. **Developer Community and Support**: All three technologies have large communities and extensive documentation, ensuring quick solutions to issues and access to numerous tutorials and best practices.
9. **Easy Data Handling**: With MySQL's relational database management, handling complex relationships between tables (like orders and items) becomes easier and more efficient.
10. **Cost-Effective**: Open-source nature of React, Node.js, and MySQL significantly reduces the cost of development as no proprietary licenses are required.

Disadvantages:

1. **Learning Curve**: While React, Node.js, and MySQL have great advantages, developers may face a steep learning curve if they are not familiar with JavaScript, asynchronous programming, or SQL queries.

2. **Server-Side Performance Limits**: Although Node.js is non-blocking and event-driven, it is single-threaded, which could lead to performance bottlenecks in CPU-intensive tasks.
3. **Database Scalability**: MySQL, being a traditional relational database, might face challenges when scaling horizontally, especially with large datasets or highly concurrent writes.

Development Environment: VS Code, Postman, and MySQL Workbench

Visual Studio Code (VS Code)

For the development of both the frontend and backend, I used **Visual Studio Code (VS Code)** as my Integrated Development Environment (IDE). VS Code is a lightweight yet powerful code editor that supports a wide variety of programming languages and frameworks, including JavaScript, Node.js, and React.

Key features that made VS Code the preferred choice for this project:

- **Extensibility**: VS Code has a vast marketplace of extensions, which made it easy to add support for React, Node.js, and MySQL. Extensions like ESLint and Prettier helped maintain code quality by enforcing coding standards and formatting rules.
- **Integrated Terminal**: VS Code's built-in terminal allowed me to run Node.js servers, execute database commands, and use Git for version control, all within a single interface.
- **Debugging**: The built-in debugging tools in VS Code made it easy to troubleshoot both the frontend and backend code. I could set breakpoints,

inspect variables, and monitor the call stack, which sped up the process of identifying and resolving issues.
- **Git Integration**: The seamless Git integration in VS Code made version control effortless. I could stage changes, commit code, and push updates to GitHub directly from the editor, without needing a separate Git client.

VS Code's flexibility, extensive plugin support, and strong integration with other tools in my stack made it an indispensable tool throughout the development process.

Postman

For testing and debugging the backend APIs, I used **Postman**, a powerful tool for API development. Postman made it easy to test the API endpoints by sending HTTP requests and inspecting the responses.

Some benefits of using Postman include:

- **API Testing**: Postman allows for sending various types of HTTP requests (GET, POST, PUT, DELETE) to test the API's functionality. I used it extensively to verify that the API endpoints for order management, user authentication, and data retrieval were working as expected.
- **Collection Management**: I could group related API requests into collections, which made it easy to organize and manage all the different API

endpoints involved in the project. This feature was particularly useful when I needed to repeatedly test a series of requests.
- **Environment Variables**: Postman supports environment variables, allowing me to store and reuse values like API keys, user IDs, and URLs. This saved time and reduced the chance of errors when switching between development and production environments.
- **Automated Testing**: Postman also allows for scripting and automated testing of APIs. Although I focused primarily on manual testing, the ability to add tests and assertions within Postman is a powerful feature for future use in automated testing scenarios.

Postman was critical for ensuring that the backend APIs were robust and worked as expected before integrating them with the frontend.

MySQL Workbench

For managing and interacting with the MySQL database, I used **MySQL Workbench**, a visual tool for database design and administration.

Key benefits of using MySQL Workbench include:

- **Database Design and Schema Management**: MySQL Workbench made it easy to visualize the database schema and relationships between tables. I could design tables, set up primary and foreign keys, and define relationships between data entities using its graphical interface.
- **SQL Query Execution**: MySQL Workbench includes a query editor where I could write and execute SQL queries to interact with the database. This was

useful for manually inserting, updating, or deleting data, as well as retrieving data for testing purposes.

- **Data Modeling**: The tool also provides data modeling features that allow for reverse engineering of the database schema, which helped ensure that the structure of the database aligned with the project's requirements.
- **Performance Monitoring**: MySQL Workbench includes monitoring tools to keep track of database performance. While this was not heavily utilized in the early stages of the project, it can be extremely valuable as the database scales and the number of users grows.

SETTING UP DEVELOPMENT ENVIRONMENT FOR WEB APPLICATIONS

Creating a robust development environment is crucial for building efficient web applications. This guide will walk you through the essential software and IDEs needed for a seamless development experience, focusing on Node.js, MySQL (with MySQL Workbench), GitHub, and Postman. Each step is designed to ensure that you have all the necessary tools to kickstart your development journey.

BACKEND

Step 1: Install Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, allowing you to run JavaScript on the server side. Here's how to set it up:

1. **Download Node.js**: Go to the Node.js official website and download the installer suitable for your operating system (Windows, macOS, or Linux).

2. **Run the Installer**: Execute the downloaded installer. Follow the prompts to accept the license agreement, select the installation path, and complete the installation.
3. **Verify Installation**: Open a command prompt (Windows) or terminal (macOS/Linux) and type the following command to verify the installation:
   -> node -v

This should display the installed version of Node.js.

4. **Install npm**: Node.js comes with npm (Node Package Manager) pre-installed. You can check its version with:
   -> npm -v

Step 2: Set Up MySQL and MySQL Workbench

MySQL is a popular relational database management system. MySQL Workbench provides a graphical interface to manage databases easily. Follow these steps to set them up:

1. **Download MySQL**: Visit the MySQL Community Downloads page and download the MySQL Installer for your operating system.
2. **Run the Installer**: Execute the installer and follow the setup wizard. Choose the server and any additional products you may need (like MySQL Workbench).
3. **Configure MySQL**: During the installation, you'll be prompted to configure the MySQL server:
   - Set a root password.
   - Choose the default server configuration (Development Machine is usually recommended).

4. **Install MySQL Workbench**: If you didn't select it during the MySQL installation, download MySQL Workbench from the MySQL Workbench page and install it.
5. **Verify Installation**: Open MySQL Workbench and connect to your local MySQL server using the root user and the password you set during installation.

Step 3: Set Up GitHub

GitHub is essential for version control and collaboration. To set it up:

1. **Create a GitHub Account**: Go to GitHub.com and sign up for a free account.
2. **Install Git**: Download and install Git from the official Git website. Follow the installation instructions for your operating system.
3. **Configure Git**: Open your command prompt or terminal and set your username and email (replace with your own):
   -> git config --global user.name "Your Name"
   -> git config --global user.email "your.email@example.com"
4. **Verify Installation**: Check the Git installation by running:
   -> git —version
5. **Create a Repository**: You can create a new repository on GitHub and clone it to your local machine using:
   -> git clone https://github.com/username/repository-name.git

Step 4: Install Postman

Postman is a powerful tool for testing APIs. Here's how to set it up:

1. **Download Postman**: Visit the Postman website and download the application for your operating system.
2. **Install Postman**: Run the downloaded installer and follow the on-screen instructions to complete the installation.
3. **Create an Account**: Open Postman and sign up for a free account if you wish to sync your work across devices.
4. **Familiarize Yourself**: Spend some time exploring Postman's interface. You can create collections for your API requests, which will help organize your testing.

Step 5: Setting Up Your Project

Now that you have all the necessary tools installed, you can set up your project:

1. **Create a Project Directory**: Use the command line to create a new directory for your project:
   -> mkdir my-web-app
   -> cd my-web-app
2. **Initialize a Node.js Project**: Run the following command to create a package.json file:
   -> npm init -y
4. **Create Your Project Files**: You can use VS Code (or any other IDE) to create your main application file (e.g., app.js) and start coding.
5. **Version Control Your Project**: Use Git to track changes in your project. Commit your changes regularly:
   -> git add .
   -> git commit -m "Initial commit"

Creating a structured project folder for Node.js backend and React frontend is an essential step in developing modern web applications. Below, is the process of setting up project, including initializing your Node.js backend and React frontend inside the AlevelProject folder.

Step 1: Create the Project Structure

AlevelProject/
├── Nodejs/
└── React/

Step 2: Setting Up the Node.js Backend

1. **Navigate to the Nodejs Folder**:

Open your terminal and navigate to the Nodejs directory:

```
npm init -y
```

3. **Install Required Packages**:

Depending on your project requirements, install the necessary packages. Common packages for a backend might include express, cors, and dotenv. Install them using:

```
npm install express cors dotenv
```

4. **Create Your Server File**:

Create a file named server.js in the Nodejs folder:

```
touch server.js
```

5. **Set Up Basic Server Code**:

Open server.js in your code editor (e.g., VS Code) and add the following basic server code:

```
const express = require('express');
const cors = require('cors');
const dotenv = require('dotenv');

dotenv.config();
const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());

app.get('/', (req, res) => {
    res.send('Hello from the Node.js backend!');
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

6. **Start the Server**:

In your terminal, run the server using:

```
node server.js
```

Creating an Account on Aiven Cloud for MySQL Database Storage

Aiven Cloud provides a fully managed service for various databases, including MySQL. Setting up an account on Aiven and obtaining your MySQL credentials is a straightforward process. This guide will walk you through the

steps needed to create an account, set up a MySQL instance, and retrieve your credentials for use in your applications.

Step 1: Sign Up for an Aiven Account

1. **Visit Aiven Website**: Go to the Aiven website.
2. **Click on "Get Started"**: On the homepage, you will find a "Get Started" button. Click on it to initiate the sign-up process.
3. **Fill Out the Registration Form**: You will be required to provide some basic information:
   - **Email Address**: Use a valid email address that you have access to.
   - **Password**: Create a strong password for your account.
   - **Company Name**: Enter the name of your organization or personal project.
   - **Country**: Select your country from the dropdown menu.
4. **Accept the Terms of Service**: Review Aiven's terms of service and privacy policy, then check the box to accept them.
5. **Click "Create Account"**: After filling in all the required fields, click the "Create Account" button to proceed.
6. **Verify Your Email**: Aiven will send a verification email to the address you provided. Open the email and click on the verification link to confirm your account.

Step 2: Log In to Your Aiven Account

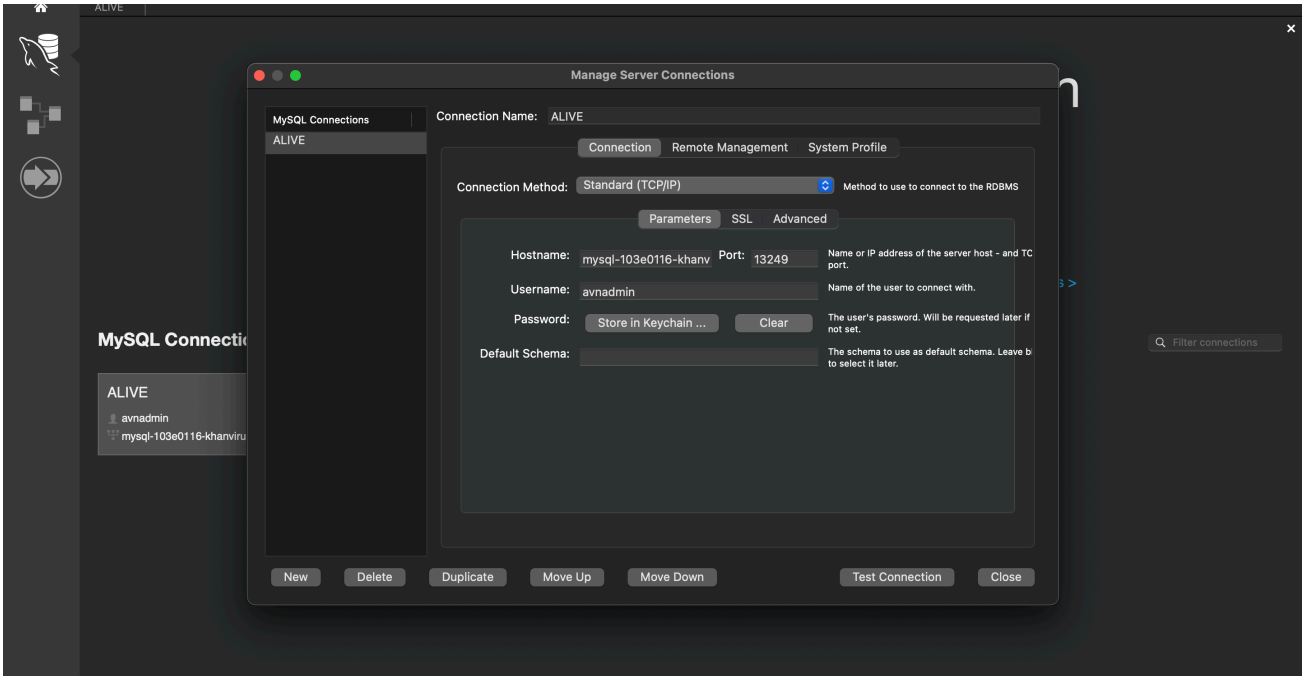
1. **Go to the Aiven Login Page**: Once your email is verified, return to the Aiven login page.
2. **Enter Your Credentials**: Use your email address and the password you created to log in.

Step 3: Create a MySQL Service

1. **Access the Aiven Console**: After logging in, you will be directed to the Aiven console dashboard.
2. **Click on "Create Service"**: Find and click the "Create Service" button, usually located at the top right of the dashboard.
3. **Choose MySQL**: From the list of available services, select "MySQL" as your database option.
4. **Fill Out Service Details**:
   - **Service Name**: Provide a unique name for your MySQL service.
   - **Project**: Select or create a project under which the service will be categorized.
   - **Cloud Provider**: Choose your preferred cloud provider (e.g., AWS, Google Cloud, Azure) and the region where you want the database to be hosted.
   - **Plan**: Select a plan based on your performance and storage needs. Aiven offers various plans, including free trials for new users.
5. **Configure Additional Settings**: You can set additional options like backups, high availability, and scaling as per your requirements.
6. **Create the Service**: Once all fields are filled out, click the "Create Service" button. Aiven will take a few moments to set up your MySQL instance.

Step 4: Retrieve MySQL Credentials

1. **Access Your Service**: After the MySQL service is created, navigate back to your Aiven console dashboard.
2. **Select Your MySQL Service**: Click on the service name you just created to access its details.



3. **Find Credentials**: In the service details page, you will see a section labeled “Service credentials.” Here, you will find:
   - **Host**: The address of your MySQL server.
   - **Port**: The port number through which you can connect (default for MySQL is 3306).
   - **Username**: The default username (usually “avnadmin” or similar).
   - **Password**: The auto-generated password for your database user.
4. **Download Credentials**: You can download the credentials as a JSON file or copy them manually for use in your application.

Step 5: Connecting to Your MySQL Database

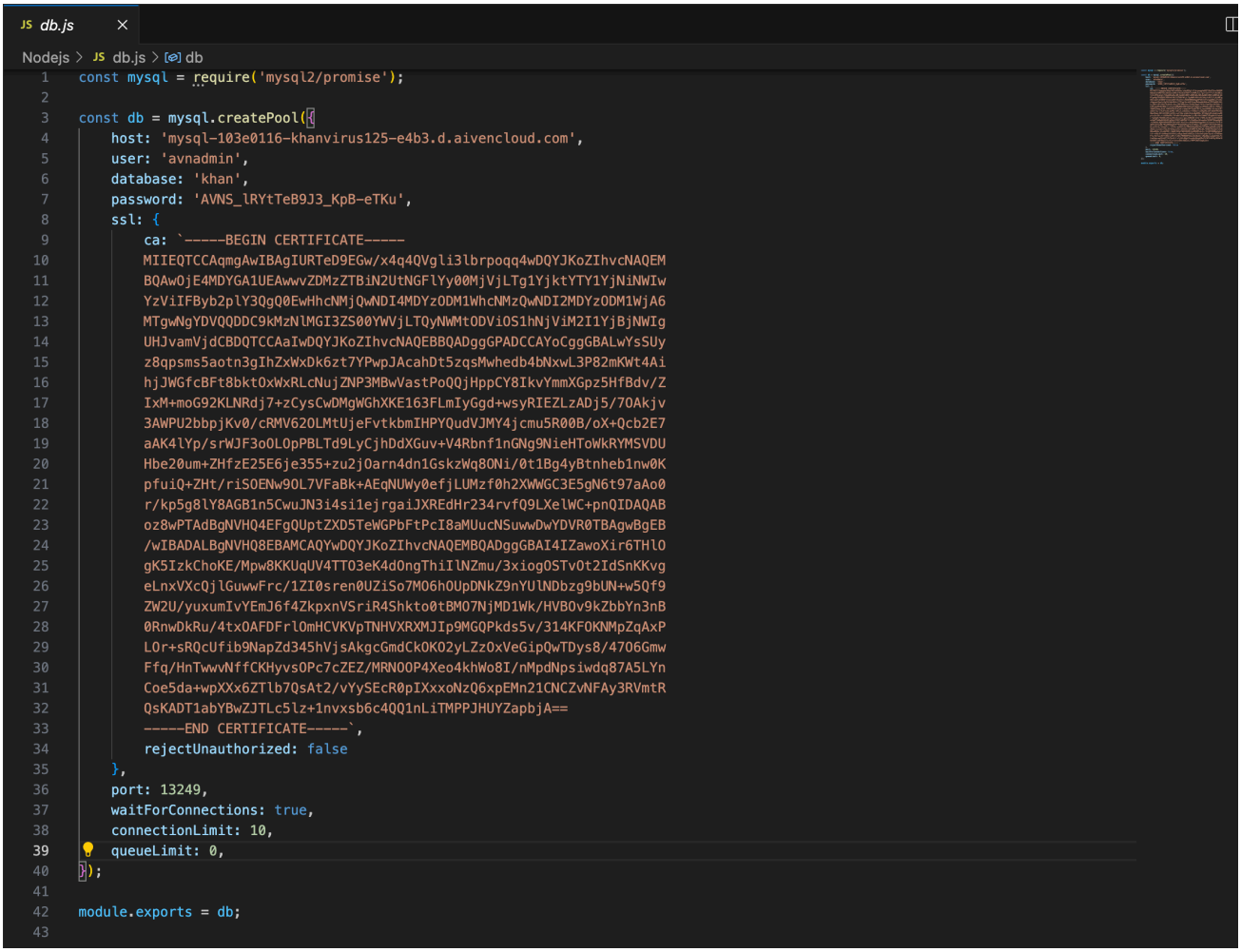
Now that you have your MySQL credentials, you can connect to your Aiven MySQL instance from your application or MySQL Workbench. Here’s how:

1. **Using MySQL Workbench**:
   - Open MySQL Workbench.
   - Click on “New Connection.”

- Enter a connection name.
- For “Hostname,” use the host value from Aiven.
- Set the port to the value provided (usually 3306).
- Enter the username and password.
- Click “Test Connection” to verify if everything is set up correctly.

2. **Using Node.js**:

In your Node.js application, use the MySQL client (like mysql or mysql2) to connect using the retrieved credentials:



```
db.js
Nodejs > db.js > db
const mysql = require('mysql2/promise');

const db = mysql.createPool({
    host: 'mysql-103e0116-khanvirus125-e4b3.d.aivencloud.com',
    user: 'avnadmin',
    database: 'khan',
    password: 'AVNS_lRYtTeB9J3_KpB-eTKu',
    ssl: {
        ca: `-----BEGIN CERTIFICATE-----
        MIIEQTCCAqmgAwIBAgIURTeD9EGw/x4q4QVgli3lbrpoqq4wDQYJKoZIhvcNAQEM
        BQAwOjE4MDYGA1UEAwwvZDMzZTBiN2UtNGFlYy00MjVjLTg1YjktYTY1YjNiNWIw
        YzViIFByb2plY3QgQ0EwHhcNMjQwNDI4MDYzODM1WhcNMzQwNDI2MDYzODM1WjA6
        MTgwNgYDVQQDDC9kMzNlMGI3ZS00YWVjLTQyNWMtODViOS1hNjViM2I1YjBjNWIg
        UHJvamVjdCBDQTCCAaIwDQYJKoZIhvcNAQEBBQADggGPADCCAYoCggGBALwYsSUy
        z8qpsms5aotn3gIhZxWxDk6zt7YPwpJAcahDt5zqsMwhedb4bNxwL3P82mKWt4Ai
        hjJWGfcBFt8bktOxWxRLcNujZNP3MBwVastPoQQjHppCY8IkvYmmXGpz5HfBdv/Z
        IxM+moG92KLNRdj7+zCysCwDMgWGhXKE163FLmIyGgd+wsyRIEZLzADj5/7OAkjv
        3AWPU2bbpjKv0/cRMV62OLMtUjeFvtkbmIHPYQudVJMY4jcmu5R00B/oX+Qcb2E7
        aAK4lYp/srWJF3oOLOpPBLTd9LyCjhDdXGuv+V4Rbnf1nGNg9NieHToWkRYMSVDU
        Hbe20um+ZHfzE25E6je355+zu2jOarn4dn1GskzWq8ONi/0t1Bg4yBtnheb1nw0K
        pfuiQ+ZHt/riSOENw9OL7VFaBk+AEqNUWy0efjLUMzf0h2XWWGC3E5gN6t97aAo0
        r/kp5g8lY8AGB1n5CwuJN3i4si1ejrgaiJXREdHr234rvfQ9LXelWC+pnQIDAQAB
        oz8wPTAdBgNVHQ4EFgQUptZXD5TeWGPbFtPcI8aMUucNSuwwDwYDVR0TBAgwBgEB
        /wIBADALBgNVHQ8EBAMCAQYwDQYJKoZIhvcNAQEMBQADggGBAI4IZawoXir6THlO
        gK5IzkChoKE/Mpw8KKUqUV4TTO3eK4dOngThiIlNZmu/3xiogOSTvOt2IdSnKKvg
        eLnxVXcQjlGuwwFrc/1ZI0sren0UZiSo7MO6hOUpDNkZ9nYUlNDbzg9bUN+w5Qf9
        ZW2U/yuxumIvYEmJ6f4ZkpxnVSriR4Shkto0tBMO7NjMD1Wk/HVBOv9kZbbYn3nB
        0RnwDkRu/4txOAFDFrlOmHCVKVpTNHVXRXMJIp9MGQPkds5v/314KFOKNMpZqAxP
        LOr+sRQcUfib9NapZd345hVjsAkgcGmdCkOKO2yLZzOxVeGipQwTDys8/47O6Gmw
        Ffq/HnTwwvNffCKHyvsOPc7cZEZ/MRNOOP4Xeo4khWo8I/nMpdNpsiwdq87A5LYn
        Coe5da+wpXXx6ZTlb7QsAt2/vYySEcR0pIXxxoNzQ6xpEMn21CNCZvNFAy3RVmtR
        QsKADT1abYBwZJTLc5lz+1nvxsb6c4QQ1nLiTMPPJHUYZapbjA==
        -----END CERTIFICATE-----`,
        rejectUnauthorized: false
    },
    port: 13249,
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0,
});

module.exports = db;
```

Understanding the MVC Pattern in a Node.js Backend Project

The Model-View-Controller (MVC) pattern is a widely used architectural design pattern in software development, particularly for building web applications. In a Node.js backend project, the MVC pattern helps to separate concerns, making the application easier to manage, scale, and maintain.

Overview of the MVC Components

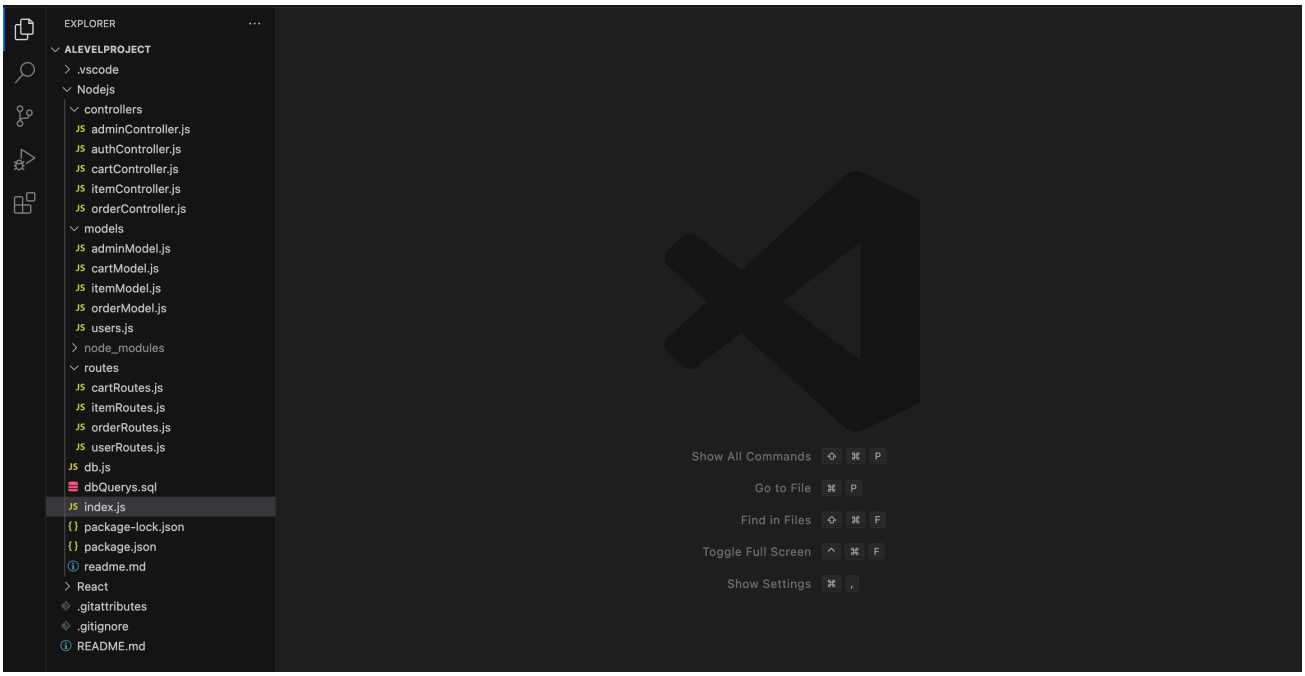
1. **Model**: The model is responsible for managing the data of the application. It defines the structure of the data, retrieves and saves data to the database, and contains business logic. In a Node.js application, models often use libraries like Mongoose (for MongoDB) or Sequelize (for SQL databases) to interact with the database.
2. **View**: In a typical MVC architecture, the view represents the user interface elements. However, in a Node.js backend context, views can refer to templates or JSON responses sent to the client. The server generates the response, which could be in the form of HTML (for traditional web applications) or JSON (for APIs).
3. **Controller**: The controller acts as an intermediary between the model and the view. It processes incoming requests, interacts with the model to fetch or modify data, and returns the appropriate view or response. Controllers contain the logic for handling user input and updating the model.

Folder Structure of an MVC Pattern in a Node.js Project

When implementing the MVC pattern in a Node.js project, organizing your folder structure is crucial for maintainability and clarity. Below is a recommended folder structure for an MVC-based Node.js backend project:

```
AlevelProject/
└─── Nodejs/
    ├─── controllers/
    |    ├─── orderController.js
    |    ├─── authController.js
    |    ├─── cartController.js
    |    ├─── adminController.js
    |    ├─── userController.js
    |    └─── itemController.js
    ├─── models/
    |    ├─── orderModel.js
    |    ├─── cartModel.js
    |    ├─── adminModel.js
    |    ├─── userModel.js
    |    └─── itemModel.js
    ├─── routes/
    |    ├─── orderRoutes.js
    |    ├─── userRoutes.js
    |    ├─── itemRoutes.js
    |    └─── cartRoutes.js
    ├─── config/
    |    ├─── db.js
    |    └─── dotenv.js
    ├─── server.js
```

└─── package.json



Required RESTful API's

RESTful API for an e-commerce application that handles various functionalities such as user authentication, admin management, order management, item management, and cart operations. Here's a detailed explanation of each API and its purpose within the project:

1. User Authentication APIs

- **POST /register:** This route allows new users to create an account. The authController.register method handles the registration process, which typically involves receiving user details (such as name, email, password, etc.), validating the input, and storing the user information in the database. It might also include hashing the password for security.
- **POST /login:** This route handles user login functionality. The authController.login method verifies the provided credentials (email and password) against the stored data in the database. Upon successful

authentication, a token (such as a JWT) is generated and returned to the client, allowing the user to make authenticated requests.

2. Admin Authentication APIs

- **POST /adminRegister:** This route is similar to the user registration route but is specifically for admin users. The adminController.register method handles admin-specific registration, ensuring only authorized users can become admins.
- **POST /adminLogin:** This route allows admins to log in to the system. The adminController.login method verifies the admin's credentials and grants access by issuing a token. Admins typically have different privileges than regular users, such as managing items or orders.

3. Order Management APIs

- **GET /orders:** This route retrieves all the orders in the system. The orderController.getAllOrders method fetches a list of all orders placed by users. Admins can use this endpoint to monitor or manage the orders.
- **GET /orders/:id:** This route retrieves a specific order based on the id provided in the URL. The orderController.getOrderById method fetches the order details, such as the items ordered, shipping details, and payment status.
- **POST /orders:** This route allows the creation of a new order. The orderController.createOrder method processes the request, typically receiving a list of items, user details, and other order-related data. This data is then saved in the database as a new order.
- **PUT /orders/:id:** This route updates an existing order based on the id. The orderController.updateOrder method is used to modify order details such as shipping address, item quantities, or order status.

- **DELETE /orders/:id:** This route deletes a specific order from the system. The orderController.deleteOrder method removes the order from the database, often used by admins to clear old or canceled orders.

4. Item Management APIs

- **GET /items:** This route retrieves all items available in the system. The itemController.getAllItems method fetches a list of products or services offered in the application. This could include details like item name, price, description, images, etc.
- **POST /items:** This route allows the creation of a new item. The itemController.createItem method receives the necessary item details (e.g., name, description, price, stock) and adds it to the database.
- **GET /items/:id:** This route retrieves a specific item based on the id. The itemController.getItemById method fetches the item's details, useful for displaying product information on a detailed view page.
- **PUT /items/:id:** This route updates a specific item. The itemController.updateItem method modifies the item's details such as price, stock, or description based on the provided data.
- **DELETE /items/:id:** This route deletes a specific item from the system. The itemController.deleteItem method removes the item from the database, often used by admins to manage the product catalog.

5. Cart Management APIs

- **GET /cart:** This route retrieves all the items in the user's cart. The cartController.getCartItems method fetches the items that a user has added to their shopping cart but hasn't purchased yet. It displays item names, quantities, and prices.

- **GET /cart/:id:** This route retrieves a specific item in the cart by its id. The cartController.getCartItemById method allows for viewing detailed information about a single cart item, such as its quantity or total cost.
- **POST /cart:** This route allows the addition of a new item to the user's cart. The cartController.addItemToCart method takes item details (like item ID, quantity, etc.) and adds the item to the cart in the database. The user can add multiple items.
- **PUT /cart/:id:** This route updates a specific item in the user's cart. The cartController.updateCartItem method modifies the quantity or other details of a cart item, such as removing a product or updating its count.
- **DELETE /cart/:id:** This route deletes an item from the user's cart. The cartController.deleteCartItem method removes the item from the cart, useful if the user no longer wants to purchase the item.

Setting Up Required Dependencies for API Implementation:

For implementing the mentioned APIs in a Node.js project, we will need certain dependencies to help with encryption, request handling, and database connections. Below is a detailed step-by-step guide along with explanations for each dependency.

1. Initialize a Node.js Project

First, you need to initialize a Node.js project, which will create a package.json file that manages your project's dependencies and scripts.

- Open your terminal and navigate to your project directory.
- Run the following command:

  -> npm init -y

This command will create a basic package.json file with default settings. It helps to track the project dependencies and other configurations.

2. Install Required Dependencies

Now, you need to install the required dependencies for implementing the APIs. You can install all dependencies at once using the following command:

-> npm install bcrypt body-parser cors express jsonwebtoken mysql2

Purpose:

> **bcrypt** is used for password hashing and security. It ensures that passwords are stored securely in the database by hashing them before saving. When a user logs in, the hashed password is compared with the stored hash.

> **body-parser** is middleware for Express that parses incoming request bodies in a middleware before the handlers, making it accessible under req.body. It's especially useful when handling form data or JSON requests in APIs.

> **cors** is middleware that allows your API to handle cross-origin requests (requests made from a different domain). This is essential when your frontend and backend are hosted on different servers.

> **express** is a fast and minimalist web framework for Node.js. It helps in building APIs by providing features like routing, middleware, and much more.

> **jsonwebtoken** is used for token-based authentication. It allows you to generate a secure token upon login, which can be sent back with future requests for authentication purposes.

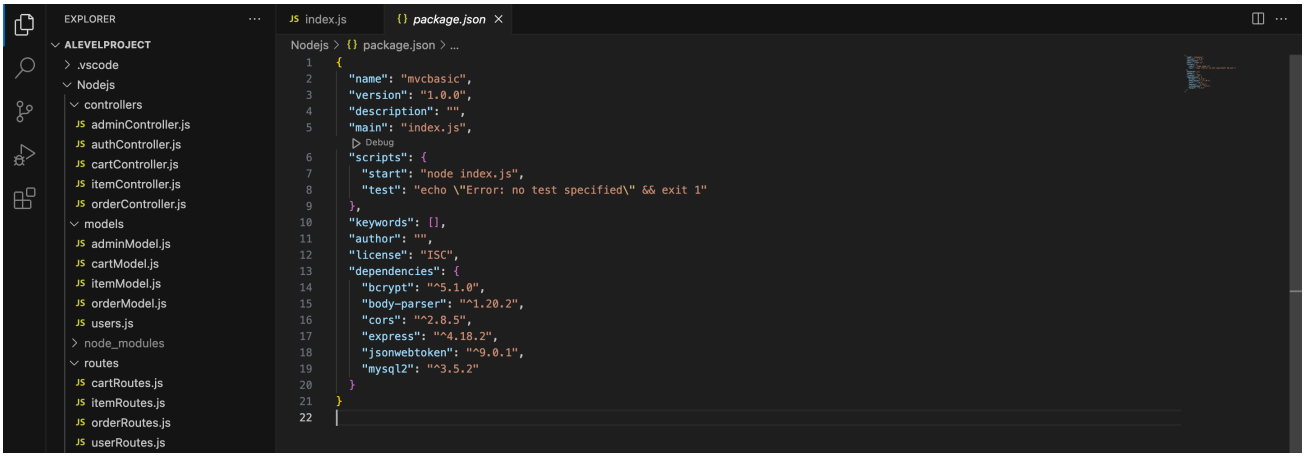
> **mysql2** is a Node.js driver for MySQL that allows you to interact with the MySQL database. It supports both promises and callback-based query execution, making it easier to work with the database.

When building a Node.js project, two critical files help manage dependencies and project configuration: **package.json** and **package-lock.json.**

1. package.json: The Heart of a Node.js Project

The package.json file is a central configuration file for any Node.js project. It defines the project's metadata, scripts, dependencies, and versioning. It's often referred to as the blueprint of the project. This file is automatically generated when you run the npm init command to initialize a Node.js project.

- **Dependencies**: Lists the modules/packages that your project depends on to run in production. These packages will be installed with npm install. For instance, in your case, you're using packages like express, mysql2, bcrypt, etc.



Why is package.json important?

- **Dependency Management**: This is where all the dependencies for your project are tracked. If you share the project with other developers, they can quickly install all the required modules by running npm install.
- **Versioning and Maintenance**: You can track the version of the project and the dependencies it requires, ensuring consistent builds across environments.

- **Scripts Automation**: You can define scripts to automate tasks like testing, building, and deploying the project, which can be executed with simple commands.

2. package-lock.json: Ensuring Consistent Builds

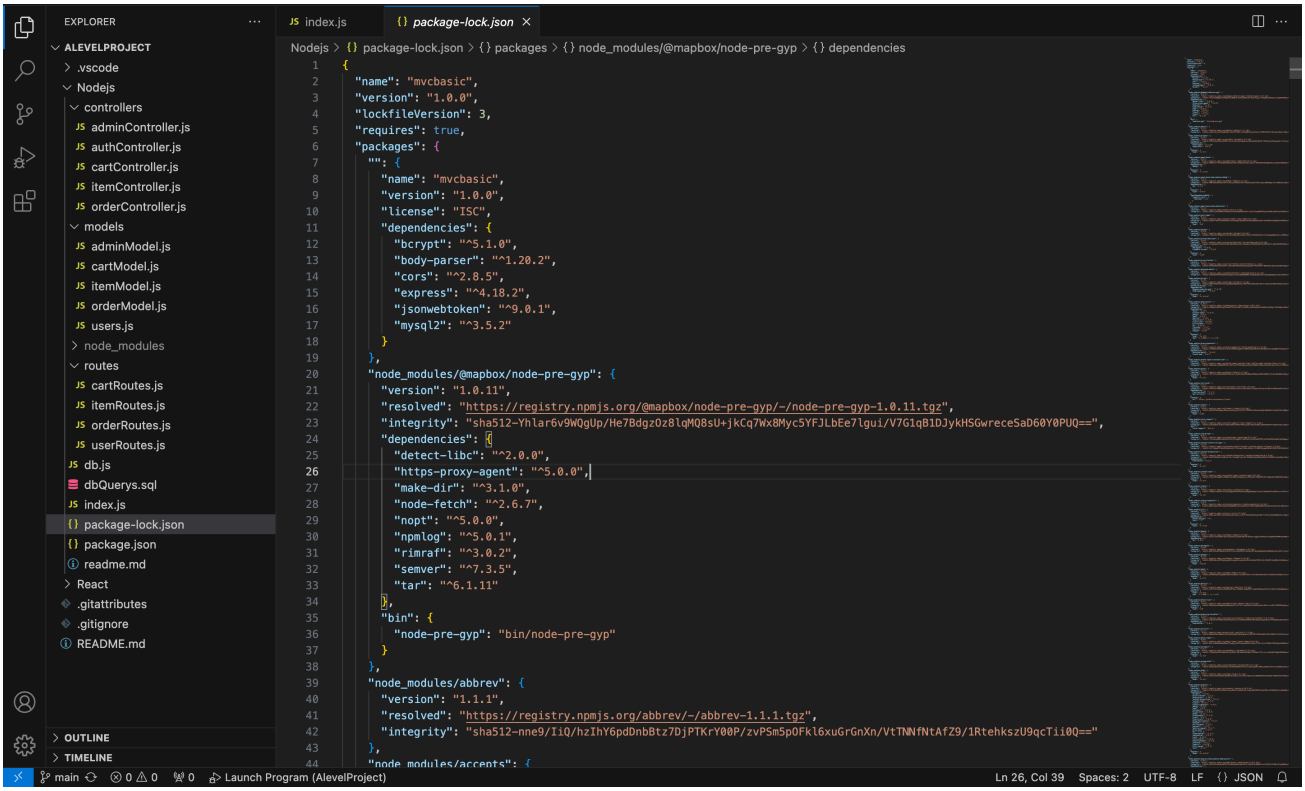
The package-lock.json file is automatically generated when you run npm install and is used to lock the versions of the dependencies (and their dependencies) installed. It helps ensure that the exact versions of packages are used across different environments, which reduces inconsistencies and bugs.

Key Characteristics of package-lock.json:

- **Dependency Tree**: Unlike package.json, which only lists the top-level dependencies of your project, package-lock.json contains the entire dependency tree, including the specific versions of each sub-dependency.
- **Version Locking**: package-lock.json locks the specific versions of every installed package, ensuring that the same versions are used whenever you or someone else installs the project in the future.
- **Exact Module Resolution**: The package-lock.json file maps each dependency to the exact version of the package, ensuring the same versions are used even if the main dependencies have flexible versioning in package.json (e.g., using the caret ^ symbol).

Example Structure of package-lock.json:
Why is package-lock.json important?

- **Consistent Builds**: By locking dependencies to a specific version, package-lock.json ensures that every developer or environment gets the exact same dependency versions, preventing potential issues caused by newer versions of a library introducing bugs.



- **Faster Installs**: Because it locks exact versions, npm can install dependencies faster since it knows exactly what versions to retrieve without checking for updates.
- **Security and Stability**: Locking dependencies helps avoid breaking changes in your project due to automatic updates in dependent packages that could introduce security vulnerabilities or instability.

Database Management:

Using MySQL to store order and item data, the backend retrieves, updates, and deletes entries in response to API calls.In this project, I have created a database named khan with several key tables that are used to store and manage different types of data essential to the application's functionality. Each table serves a specific purpose in relation to user authentication, product management, order processing.

1. Admin Table

The Admin table is designed to store information related to the administrative users of the application. This table contains data such as admin credentials (like username, password), roles, and other authentication-related fields that allow the admins to log in and manage the platform. Typically, only authorized personnel, such as site administrators or system managers, are stored here to maintain the security of the system.

```
CREATE TABLE "Admin" (
  "id" int NOT NULL AUTO_INCREMENT,
  "username" varchar(255) NOT NULL,
  "password" varchar(255) NOT NULL,
  "phone_number" varchar(15) NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "isVerified" int DEFAULT '0',
  PRIMARY KEY ("id"),
  UNIQUE KEY "phone_number" ("phone_number")
)
```

2. Cart Table

The cart table stores information about the shopping cart of individual users. It tracks the items that users add to their cart, allowing them to view, update, or remove products before placing an order. The table typically contains information such as the user's ID, the product's ID, quantity, and timestamps for when the items were added or modified.

- **Purpose**: Manage users' shopping cart items.

DDL for khan.cart

```
CREATE TABLE "cart" (
  "id" int NOT NULL AUTO_INCREMENT,
  "userId" varchar(255) NOT NULL,
  "name" varchar(255) NOT NULL,
  "description" text NOT NULL,
  "title" varchar(255) NOT NULL,
  "extra_info" text,
  "metadata" text,
  "qty" int NOT NULL,
  "price" decimal(10,2) NOT NULL,
  "discount" decimal(10,2) NOT NULL,
  "pic1" varchar(255) DEFAULT NULL,
  "pic2" varchar(255) DEFAULT NULL,
  "pic3" varchar(255) DEFAULT NULL,
  "pic4" varchar(255) DEFAULT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY ("id")
)
```

3. Items Table

The items table is responsible for storing all the products or items that are available in the application. Each record in this table represents a single product, containing details like the item's name, description, price, quantity,

and potentially images. This table is crucial for managing inventory and ensuring users have access to product data when browsing or making a purchase.

- **Purpose**: Store product or item information available in the application.

DDL for khan.items

```
CREATE TABLE "items" (
  "id" int NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "description" text,
  "title" varchar(255) DEFAULT NULL,
  "extra_info" text,
  "metadata" varchar(255) DEFAULT NULL COMMENT '                        ',
  "qty" int DEFAULT NULL,
  "price" decimal(10,2) DEFAULT NULL,
  "discount" decimal(10,2) DEFAULT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "pic1" text,
  "pic2" text,
  "pic3" text,
  "pic4" text,
  PRIMARY KEY ("id")
)
```

4. Orders Table

The orders table tracks all the orders placed by users. Each order represents a transaction and usually includes a reference to the user who placed the order, the total cost, order status (e.g., pending, shipped, delivered), and payment information. This table is essential for processing and managing user purchases, allowing administrators and users to view order histories, track shipments, and manage returns or cancellations.

- **Purpose**: Manage and track user orders.

DDL for khan.orders

```
CREATE TABLE "orders" (
  "id" int NOT NULL AUTO_INCREMENT,
  "fullname" varchar(255) NOT NULL,
  "email" varchar(255) NOT NULL,
  "mobile" varchar(15) NOT NULL,
  "address" text NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "itemlist" json NOT NULL,
  PRIMARY KEY ("id")
)
```

5. Users Table

The users table stores all the information related to the users of the application. This table typically contains details such as user credentials (username, password), contact information (email, phone), and other relevant personal data. The table allows the application to authenticate users, store preferences, and manage profiles, ensuring personalized experiences and secure access.

- **Purpose**: Store and manage user information for authentication and profiles.

DDL for khan.users

```
CREATE TABLE "users" (
  "id" int NOT NULL AUTO_INCREMENT,
  "username" varchar(255) NOT NULL,
  "password" varchar(255) NOT NULL,
  "phone_number" varchar(15) NOT NULL,
  "created_at" timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY ("id"),
  UNIQUE KEY "phone_number" ("phone_number")
)
```

Creating an Express Server and APIs:

index.js file is responsible for setting up an Express server and routing various API endpoints for the backend services of your project. Let's break it down step by step and explain the key elements involved in creating the server and connecting it to the APIs.

1. Importing Modules

```
//index.js
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors'); // Import the cors middleware
```

- express: This is the core framework for building web applications and APIs. It simplifies creating routes, handling requests, and sending responses.
- body-parser: This middleware is used to parse the body of incoming requests. In this case, we use bodyParser.json() to parse JSON payloads, which is essential for working with API requests and data sent by clients.
- cors: This middleware is used to enable CORS (Cross-Origin Resource Sharing), allowing your API to be accessed from different domains (e.g., when the frontend and backend are hosted on different servers).

2. Importing Routes

```
const itemRoutes = require('./routes/itemRoutes');
const userRoutes = require('./routes/userRoutes');
const orderRoutes = require('./routes/orderRoutes');
const cartRoutes = require('./routes/cartRoutes');
```

Here, importing the route files for different parts of the system:

- itemRoutes: Handles routes related to items (products) in the system.
- userRoutes: Handles user-related operations like registration and login.
- orderRoutes: Deals with managing orders, fetching orders, creating new orders, etc.
- cartRoutes: Manages cart-related operations like adding items to the cart, updating items, or deleting them.

3. Setting Up Middleware

```
// Middleware
const app = express();
app.use(cors()); // Use the cors middleware to enable CORS
app.use(bodyParser.json());
```

- app.use(cors()): This line enables CORS for the API. CORS is crucial when your frontend and backend are hosted on different domains or ports. By using the cors middleware, you allow your API to be accessed from the frontend even if they are hosted separately.

- app.use(bodyParser.json()): This middleware parses incoming requests with JSON payloads. It's essential for handling POST, PUT, and PATCH requests where the client sends data in the body as JSON.

4. Error Handling Middleware

```
// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong.' });
});
```

This is a custom error-handling middleware. If any unhandled error occurs during the request-response cycle, it will be caught here. It logs the error stack and sends a 500 status code (Internal Server Error) with a generic error message back to the client. This ensures the server doesn't crash due to unhandled exceptions and provides a useful error message for debugging.

5. Setting Up Routes

```
// Routes
// user Routes
app.use('/user', userRoutes);
// order routes
app.use('/api', orderRoutes);
// item routes
app.use('/api', itemRoutes);
// cart routes
app.use('/api', cartRoutes);
// order routes
app.use('/api', orderRoutes);
```

- app.use('/user', userRoutes): This line mounts all user-related routes (such as login and registration) under the /user path. Any request that hits this path will be handled by userRoutes.
- app.use('/api', orderRoutes): This mounts the order-related routes under the /api path. For example, /api/orders could fetch all orders or create a new one.
- app.use('/api', itemRoutes): All routes related to items (like fetching products, creating new items) are accessed under /api.

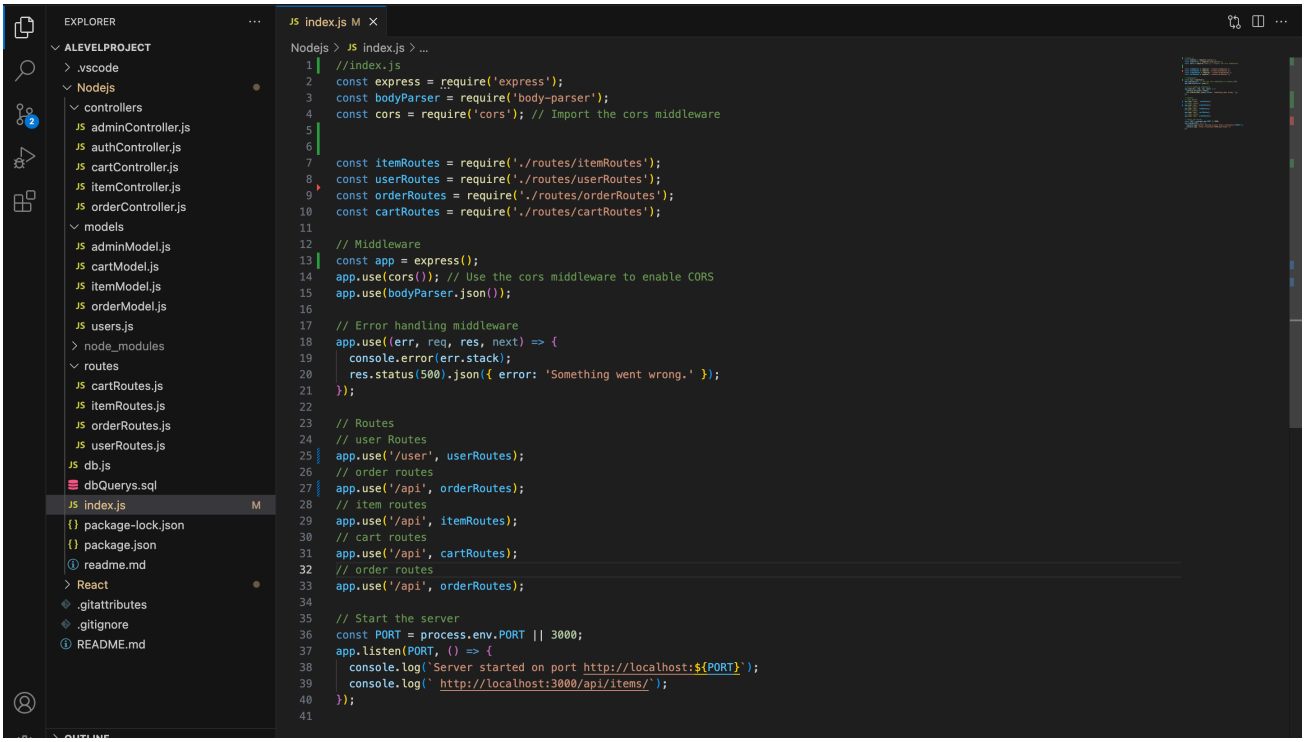
- app.use('/api', cartRoutes): Routes for cart-related operations are mounted here. For instance, /api/cart would handle adding items to the cart.

6. Starting the Server

```
// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server started on port http://localhost:${PORT}`);
  console.log(` http://localhost:3000/api/items/`);
});
```

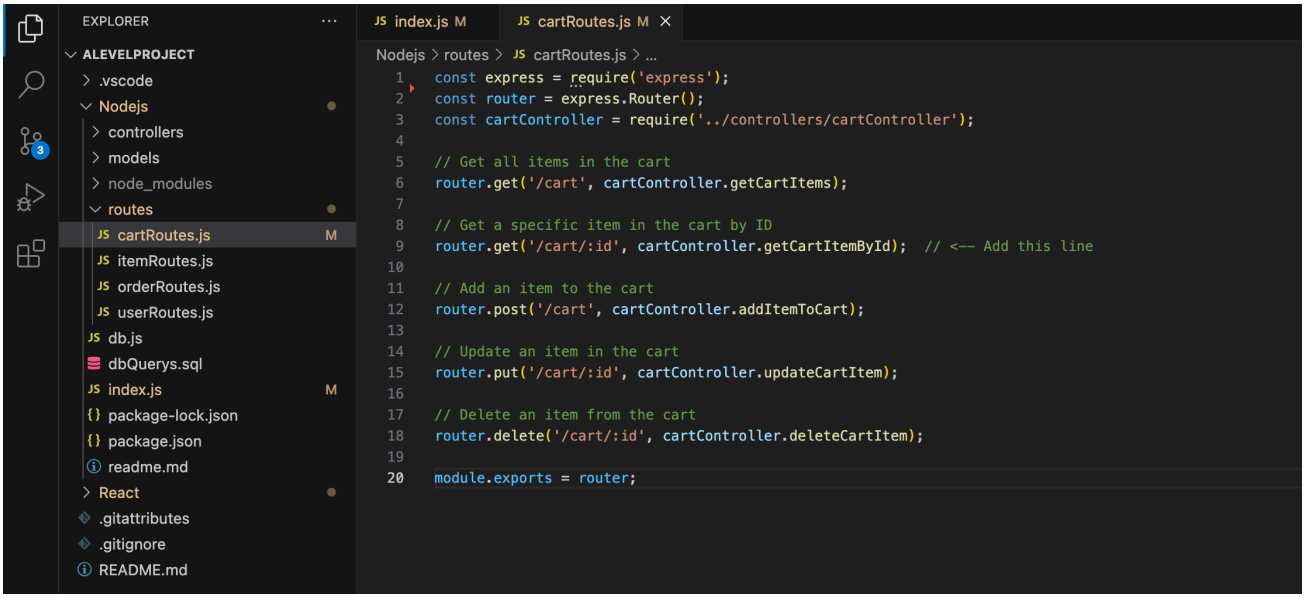
- PORT = process.env.PORT II 3000: The PORT variable is defined to allow the server to run on a custom port if specified in the environment (process.env.PORT), otherwise, it defaults to port 3000.
- app.listen(PORT, () => { ... }): This function starts the server and listens for incoming requests on the specified port.
- **Console Logs**: After starting, it logs the port where the server is running (in this case, http://localhost:3000). This helps confirm that the server has successfully started.

Index.js file

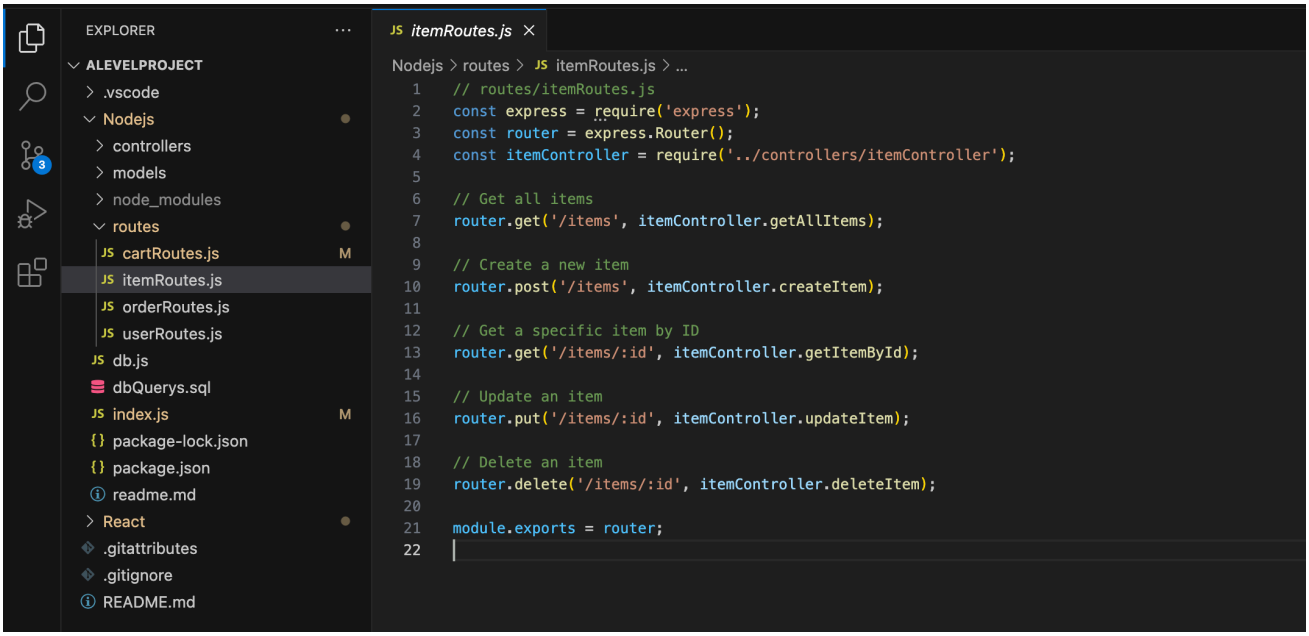


Routes Folder

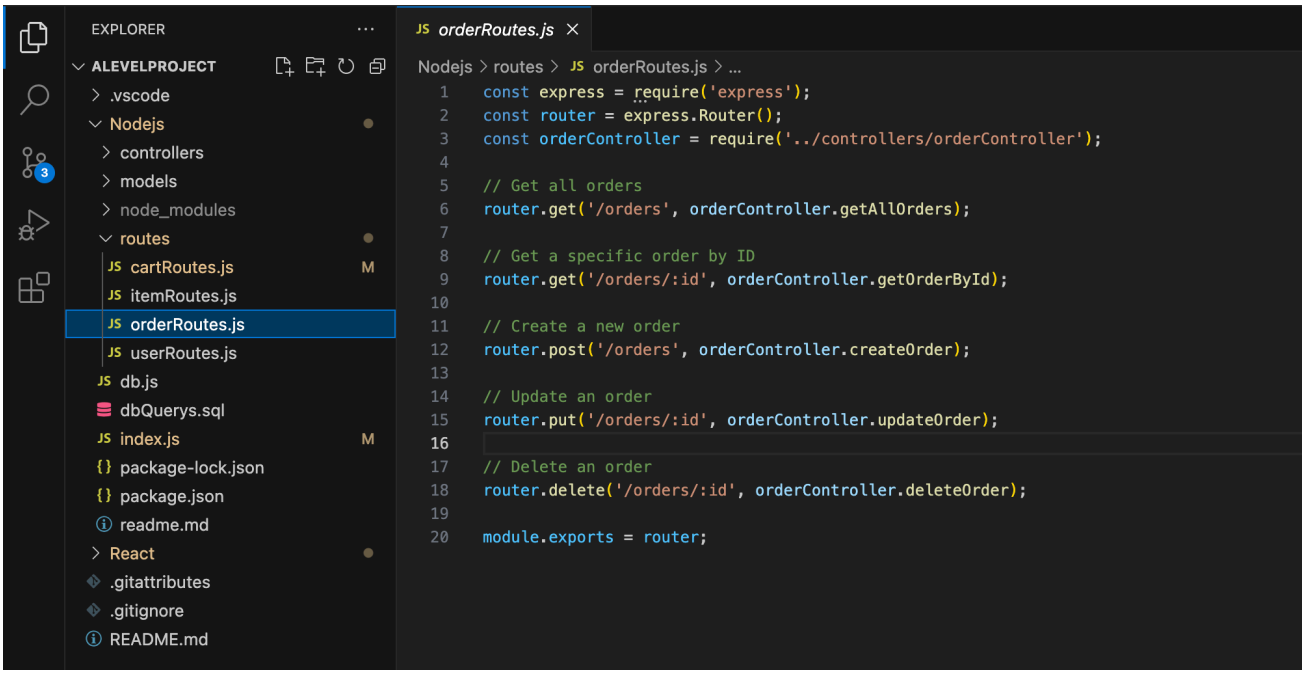
cartRoutes.js file



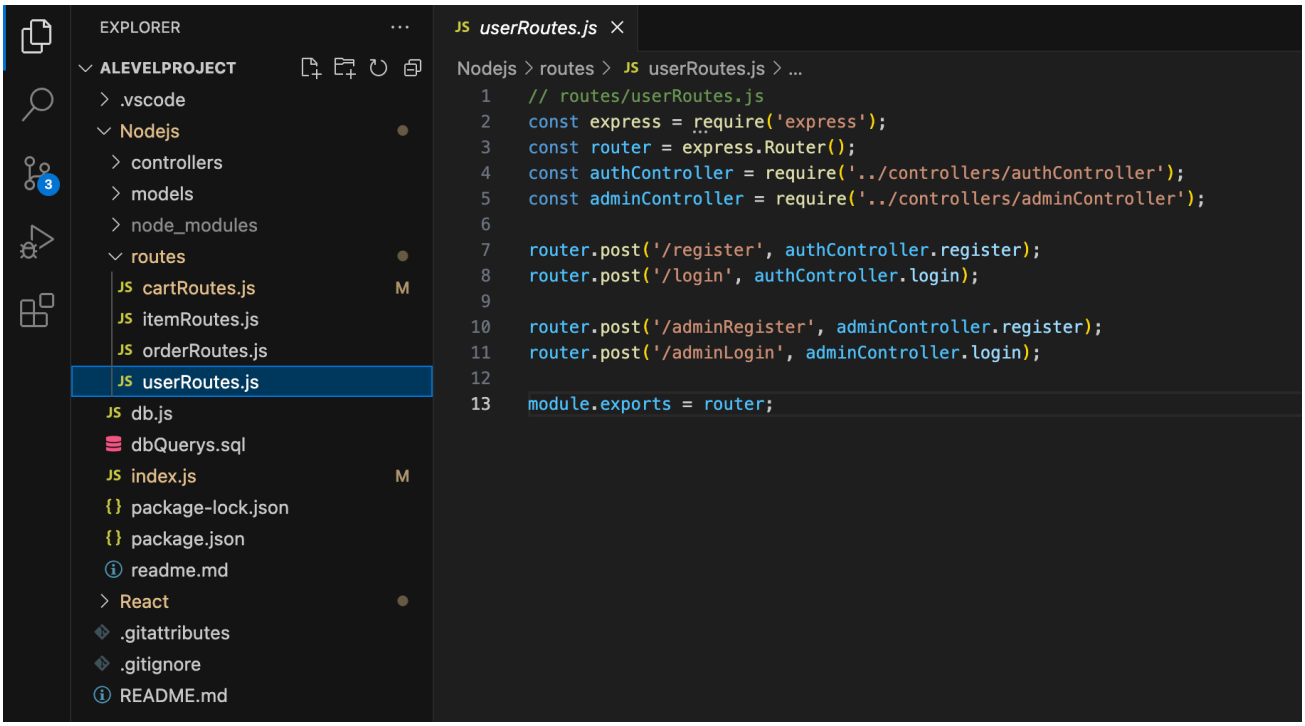
itemRoutes.js



orderRoutes.js

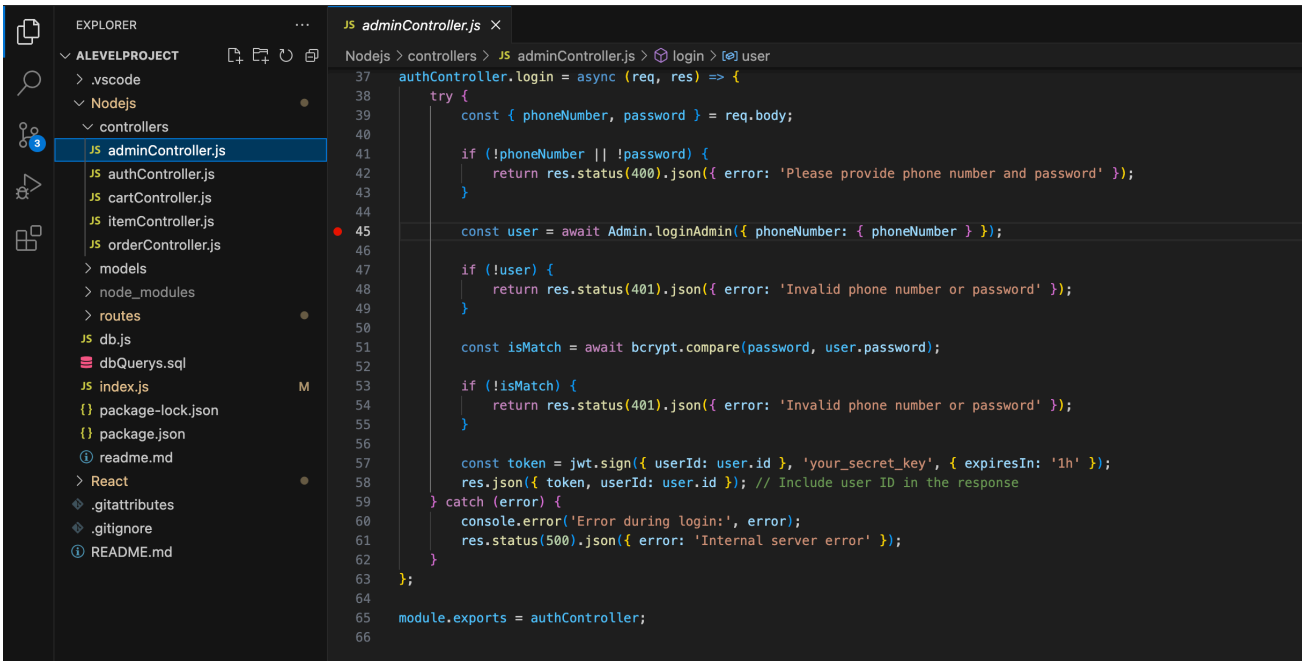


userRoutes.js

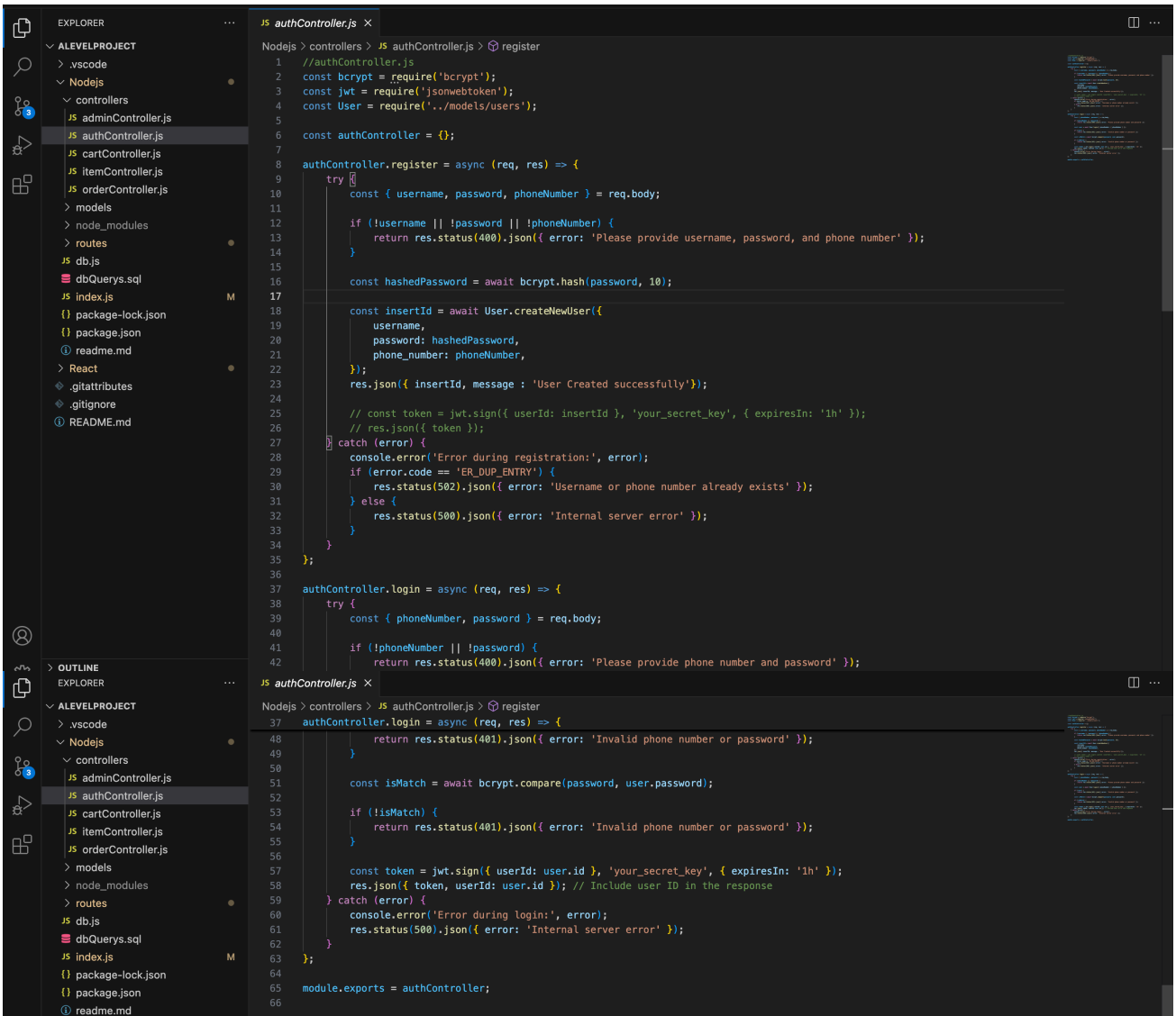


Controllers Folder

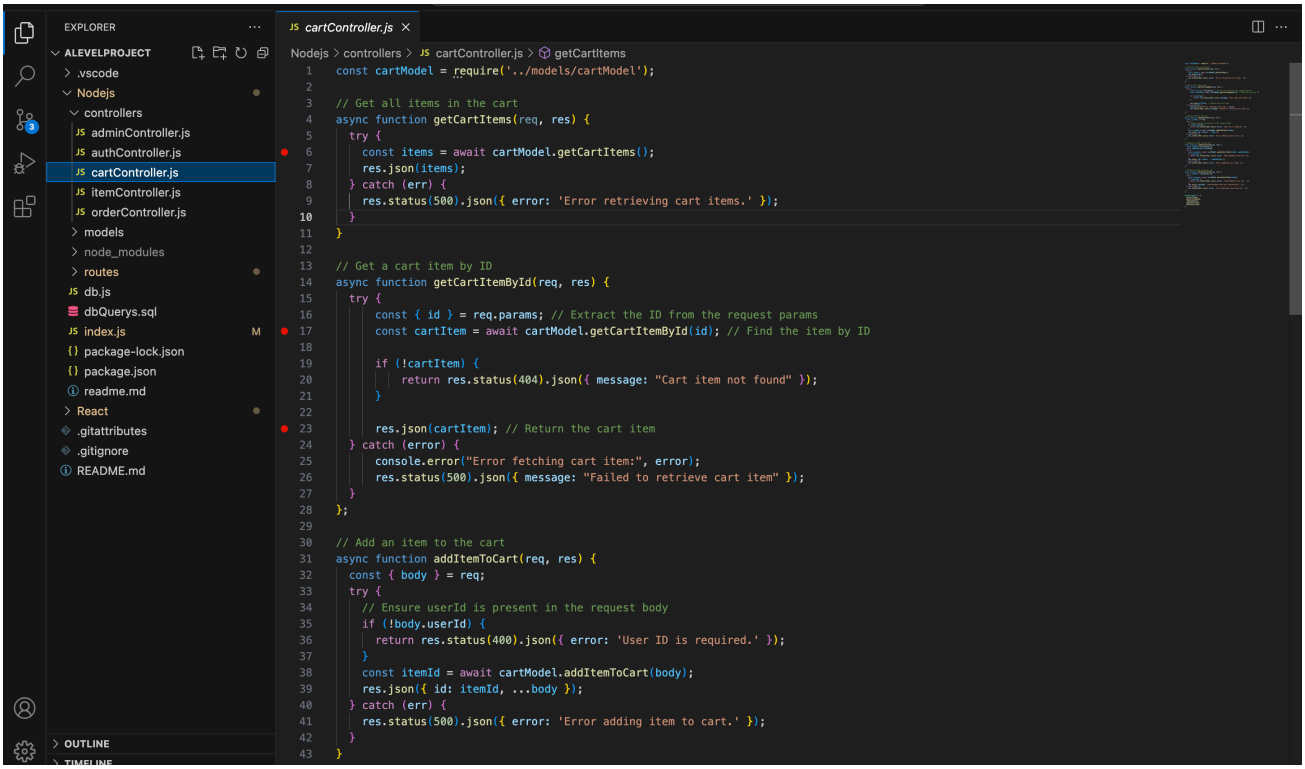
adminController.js

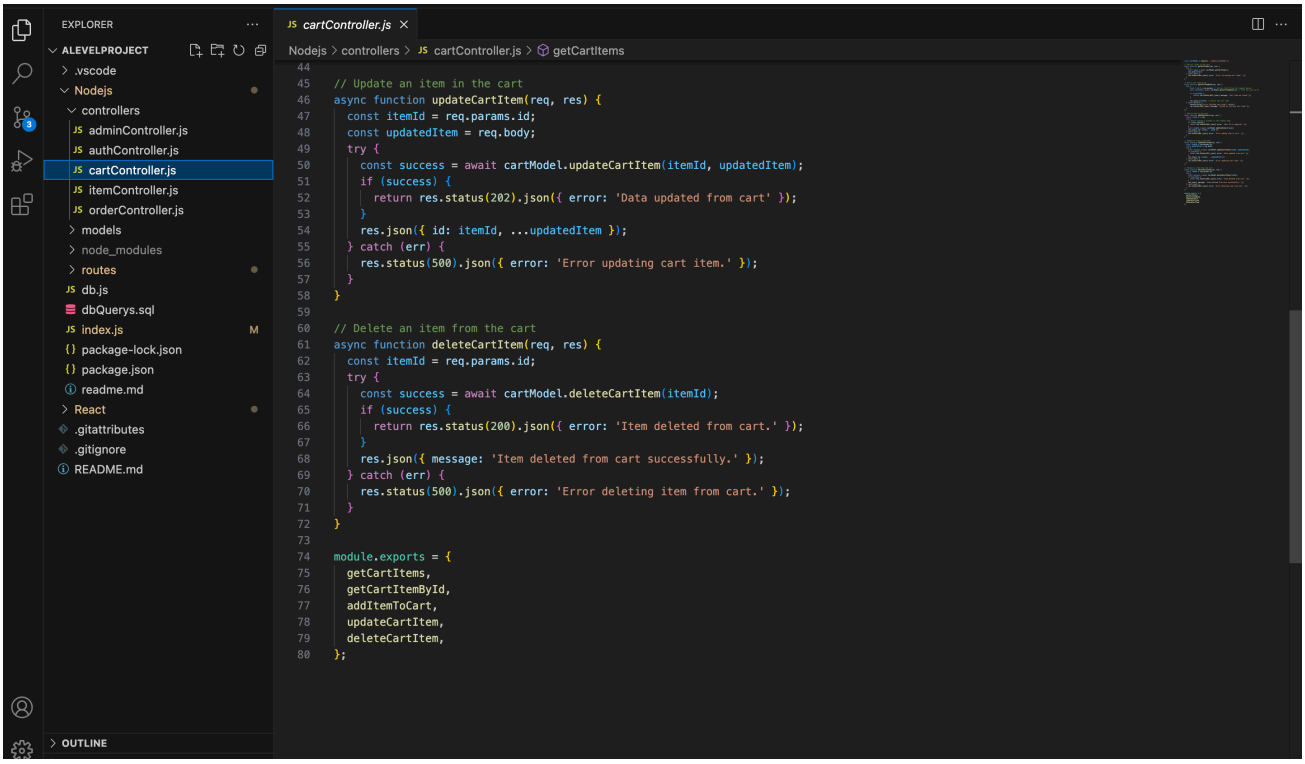


authController.js

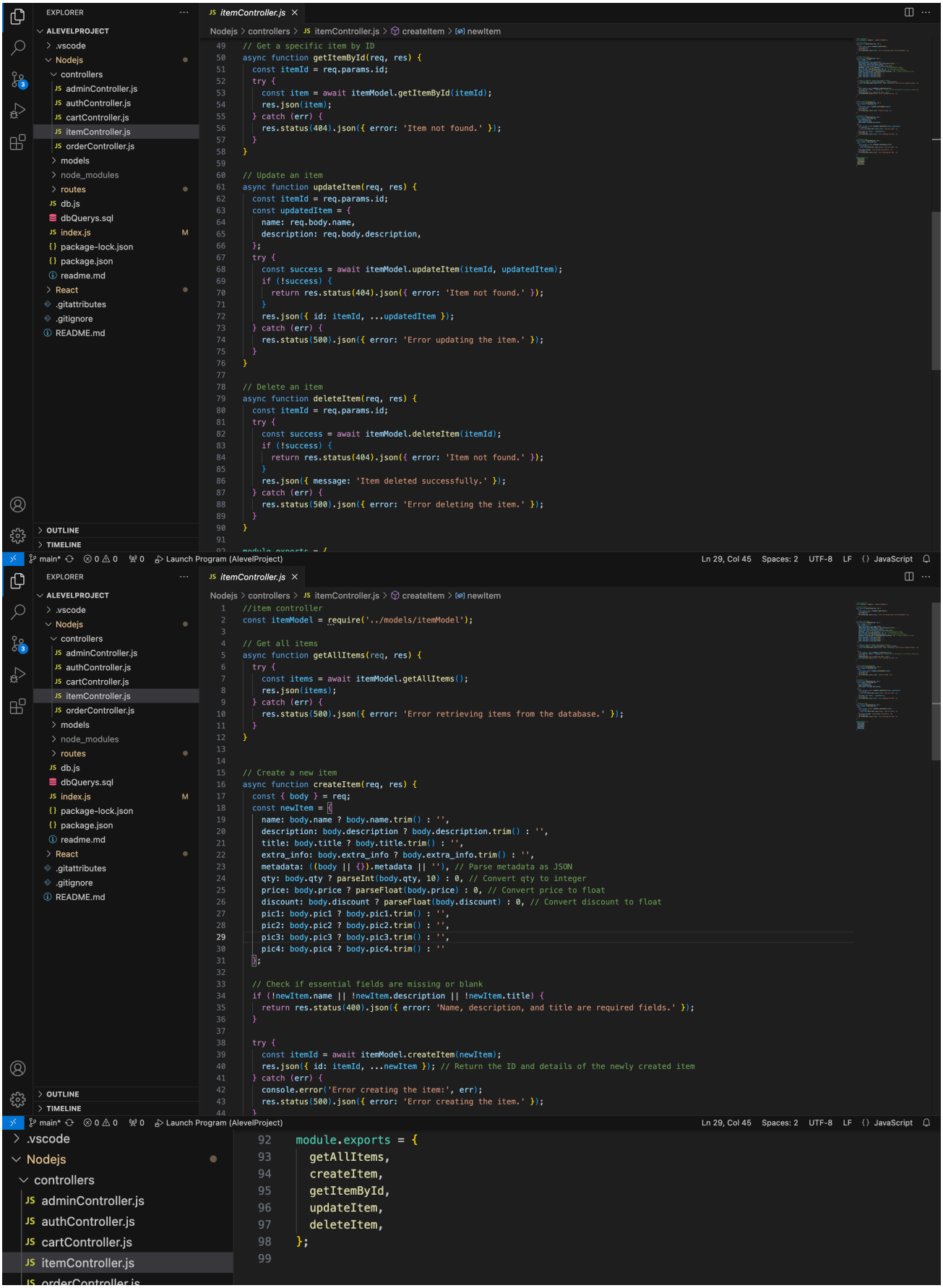


cartController.js

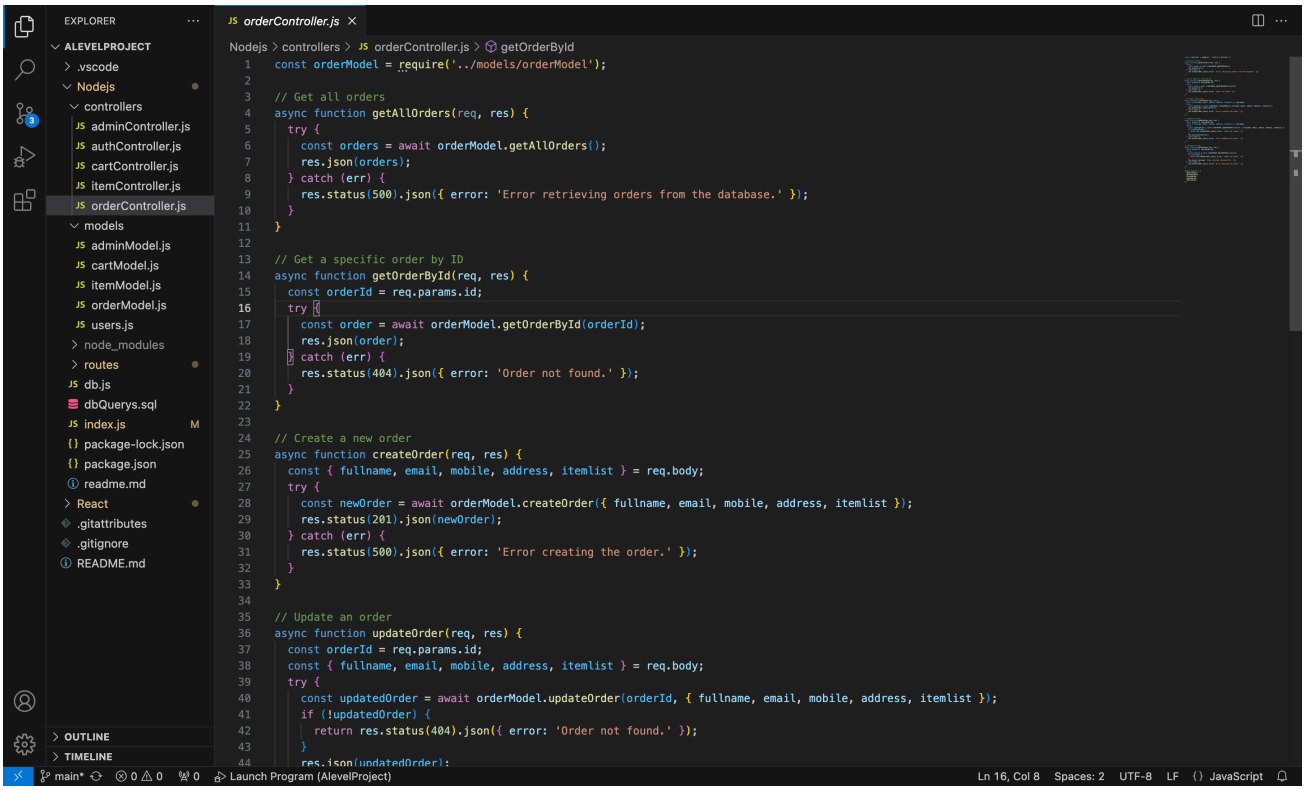


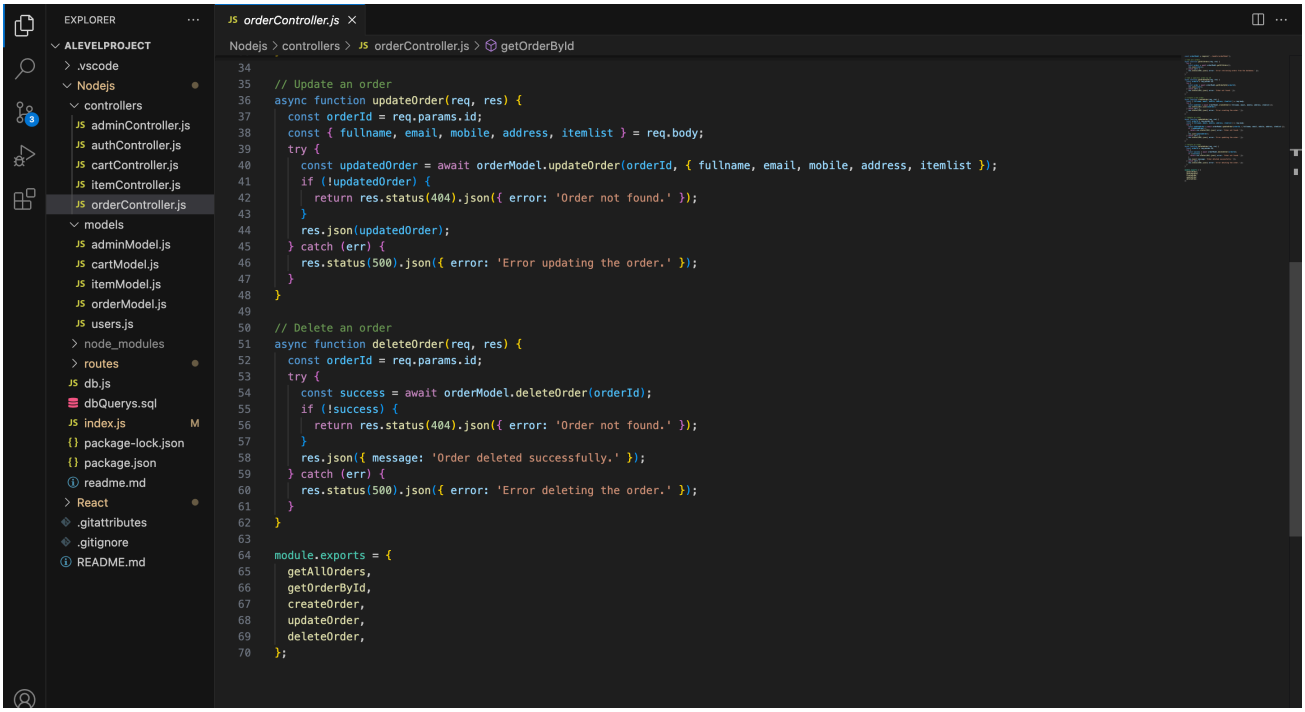


itemController.js



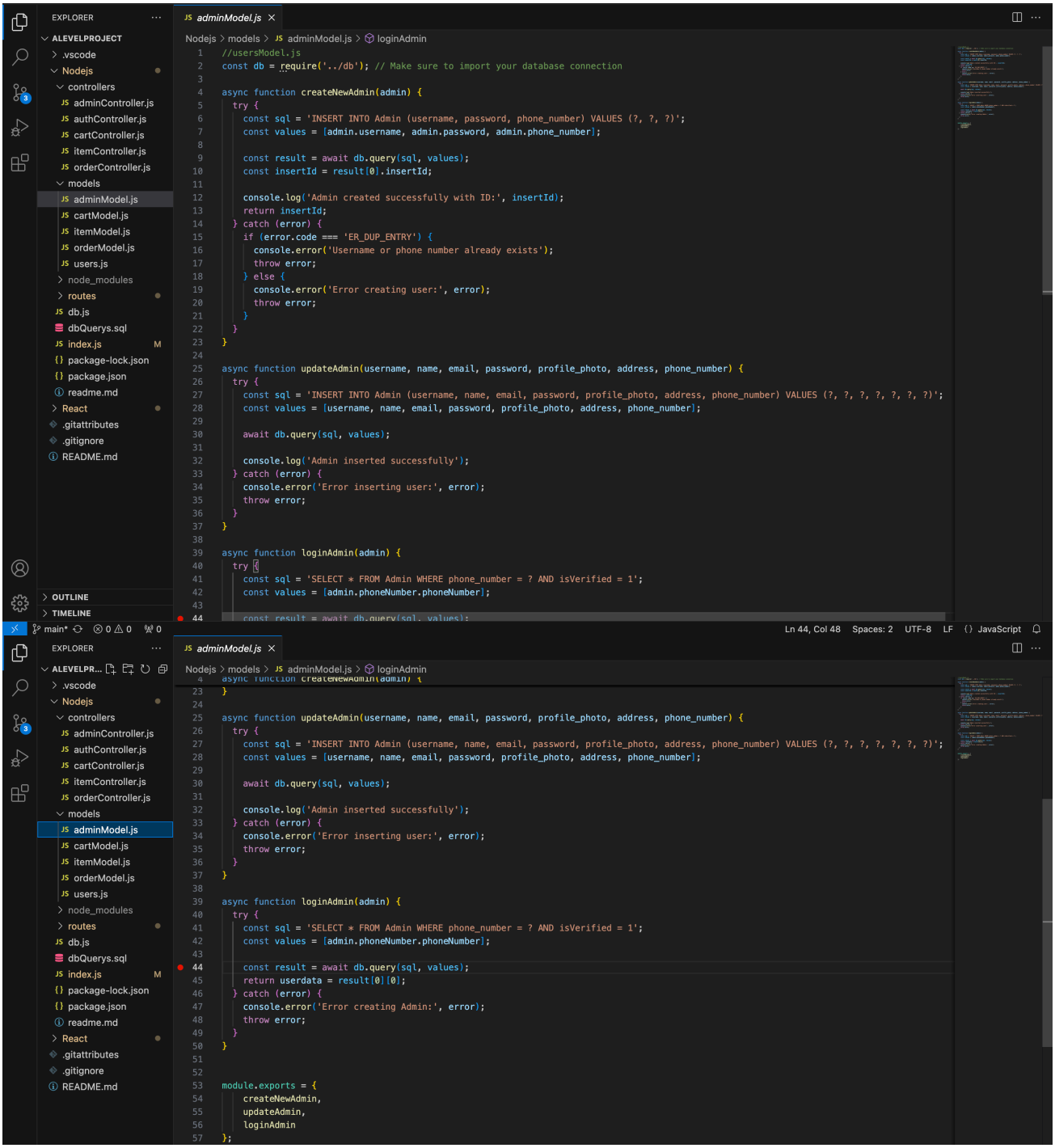
orderController.js



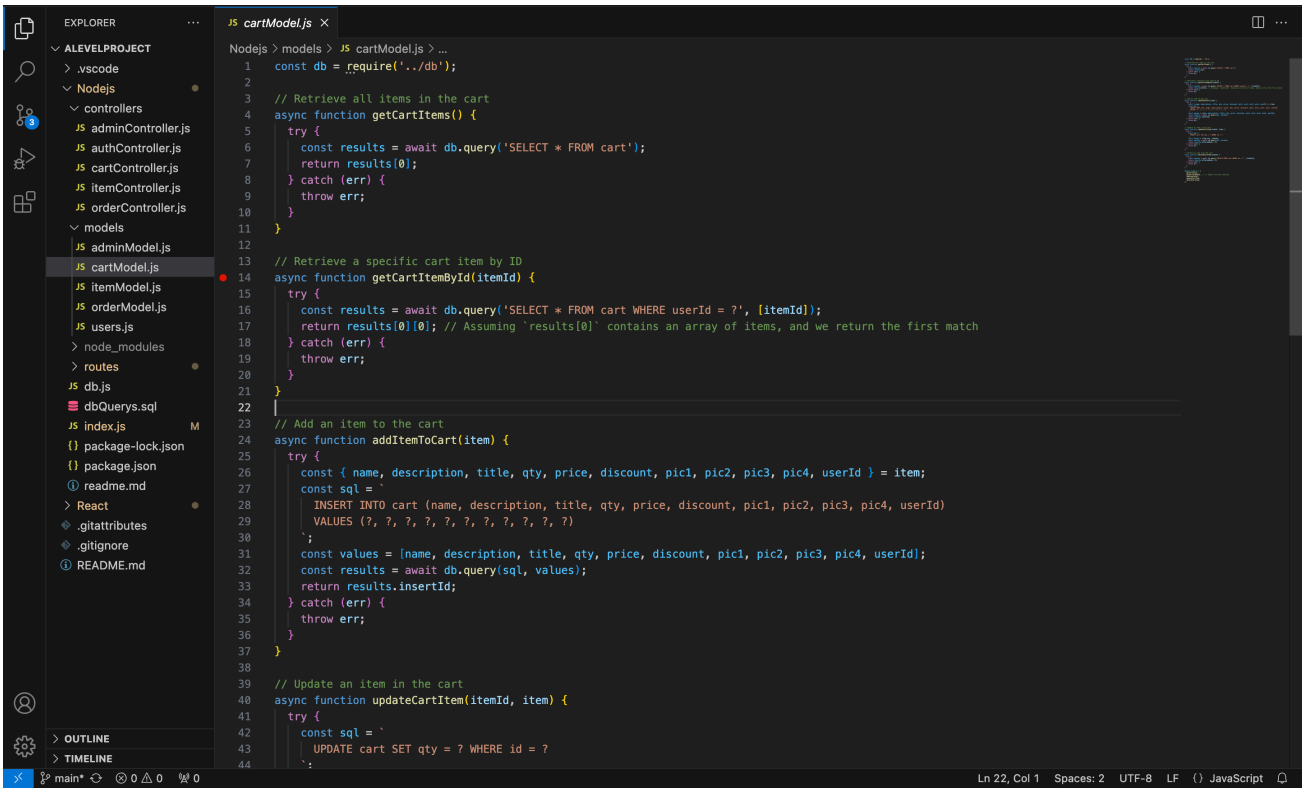


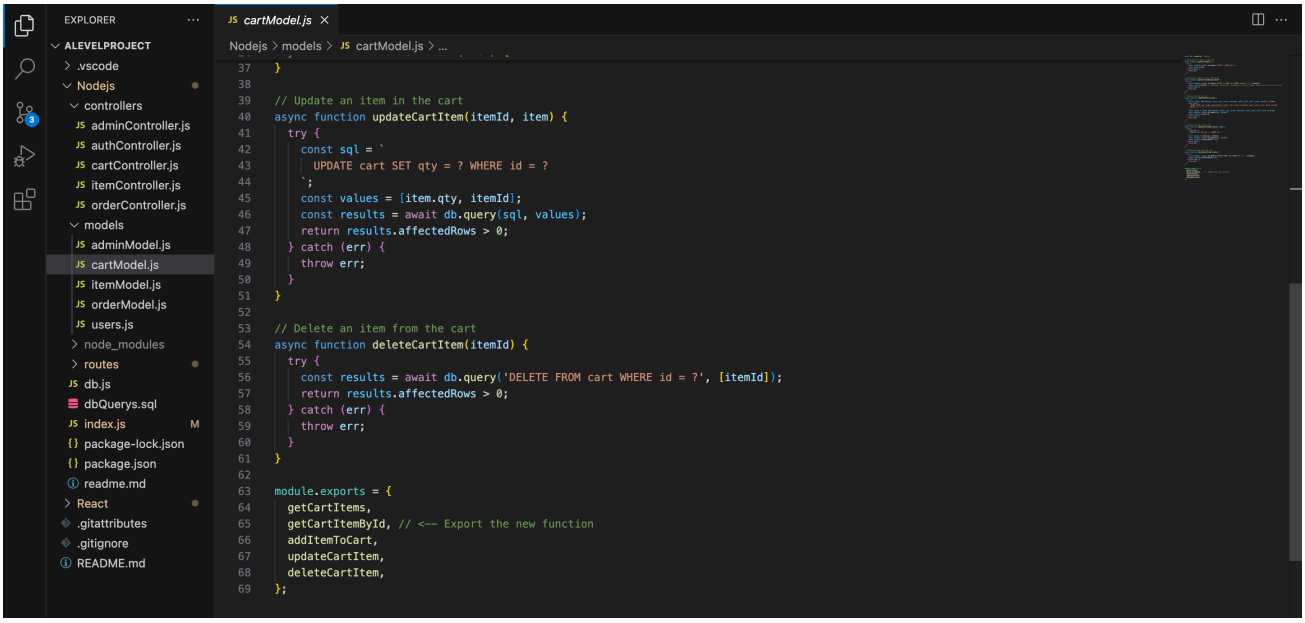
Models Folder

adminModel.js

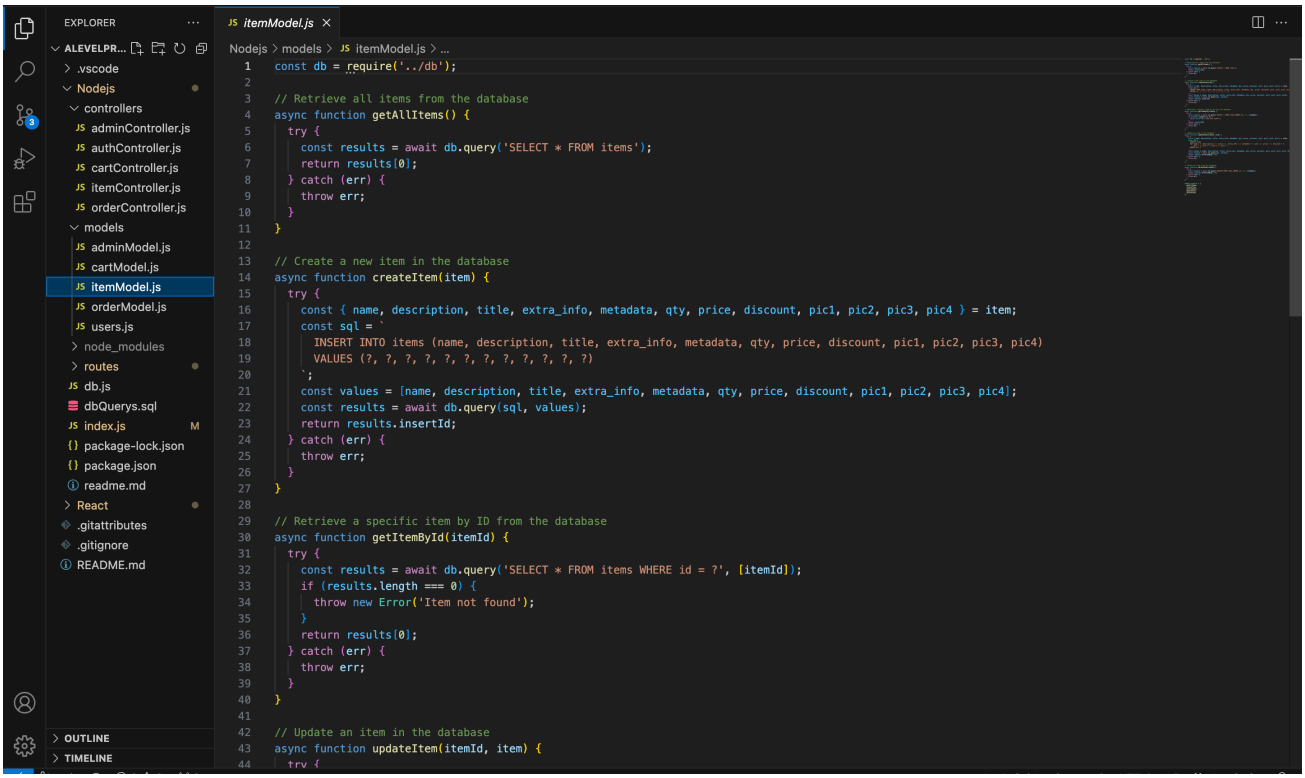


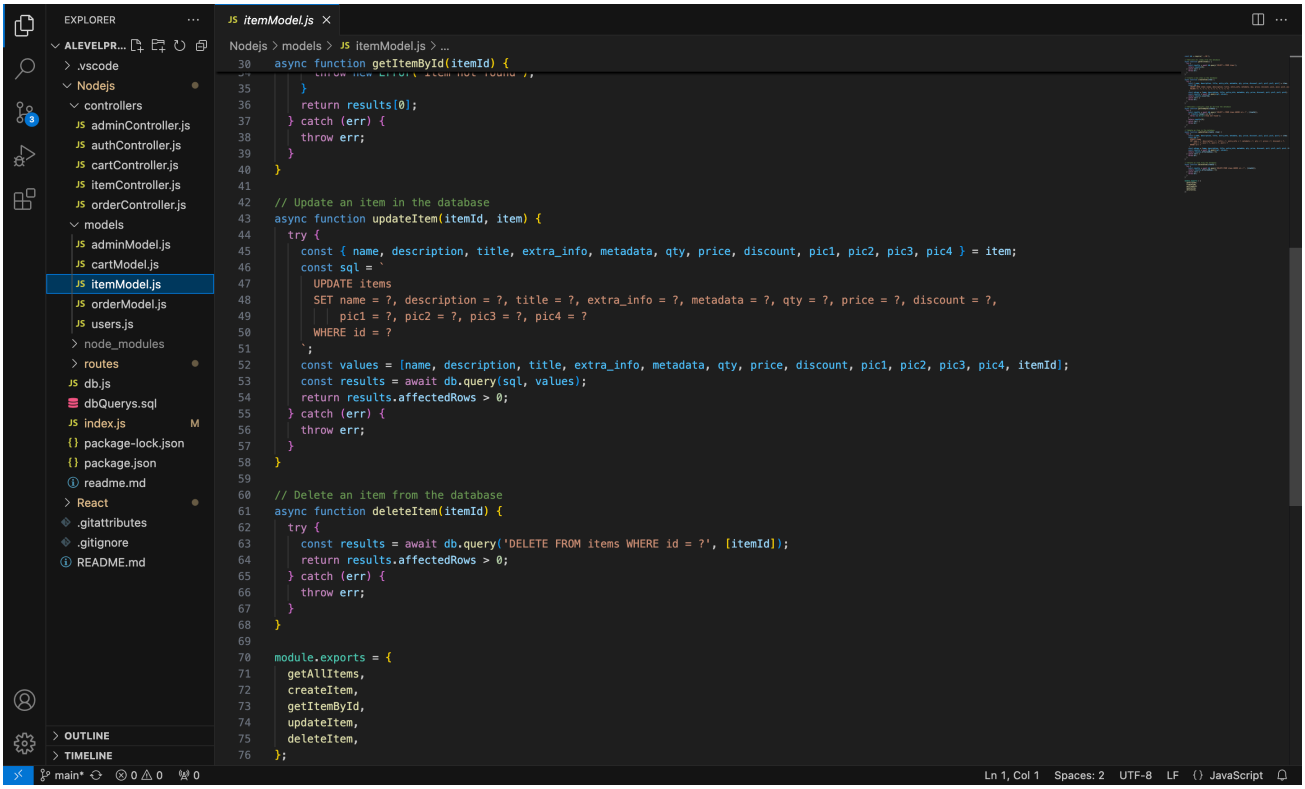
cartModel.js



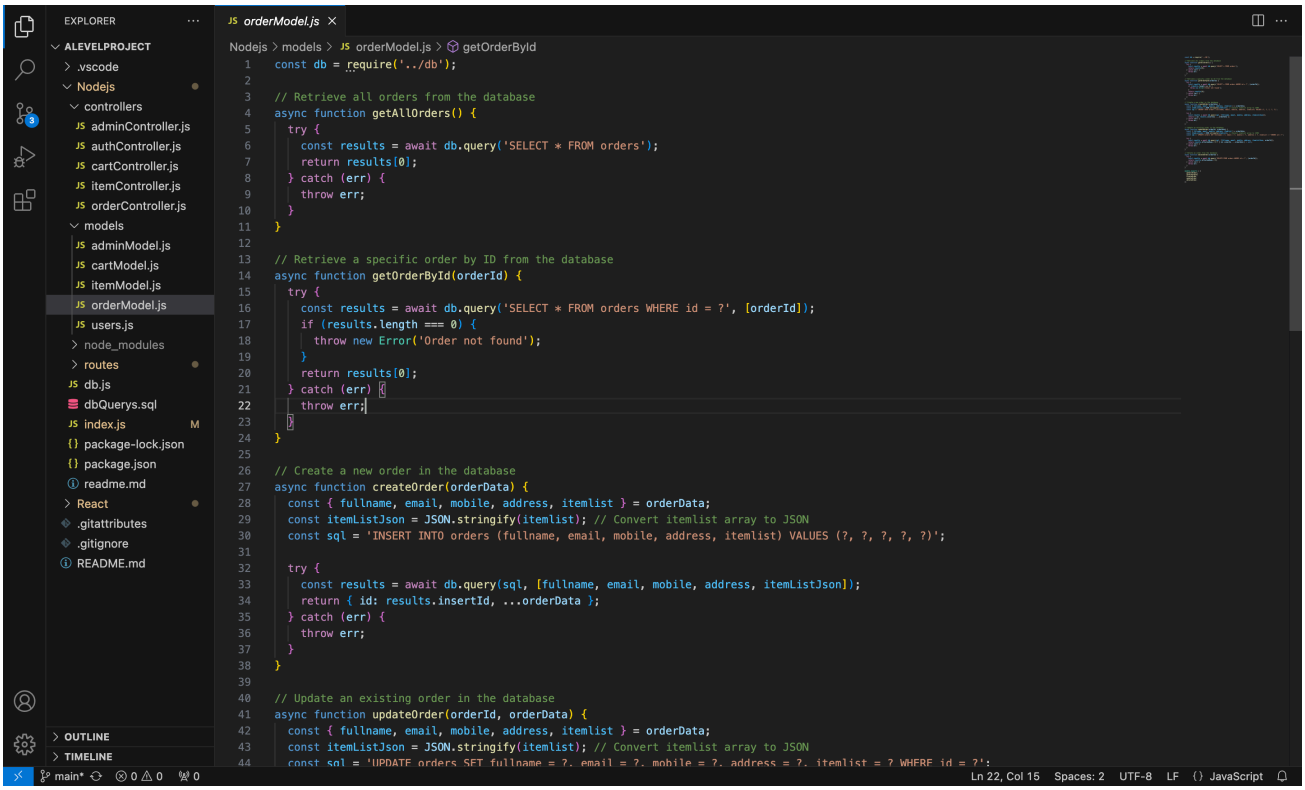


itemModel.js

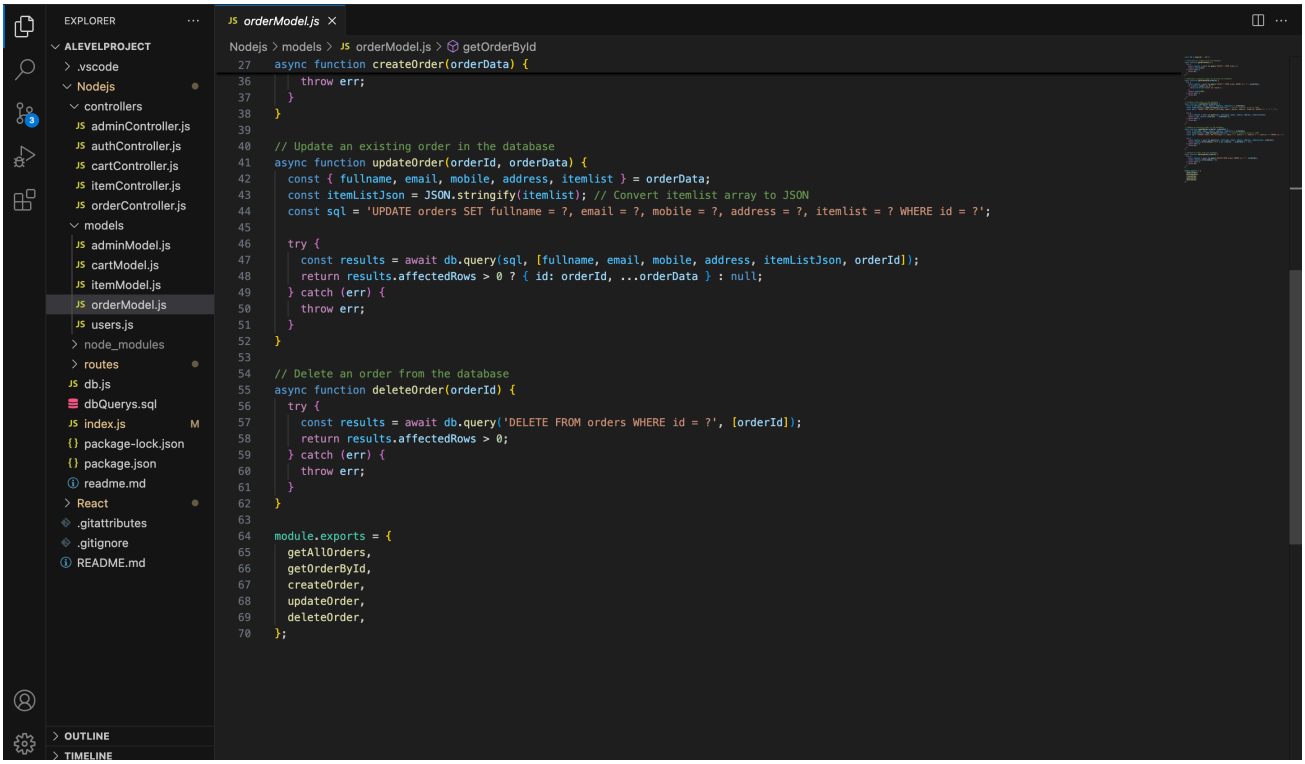




orderModel.js

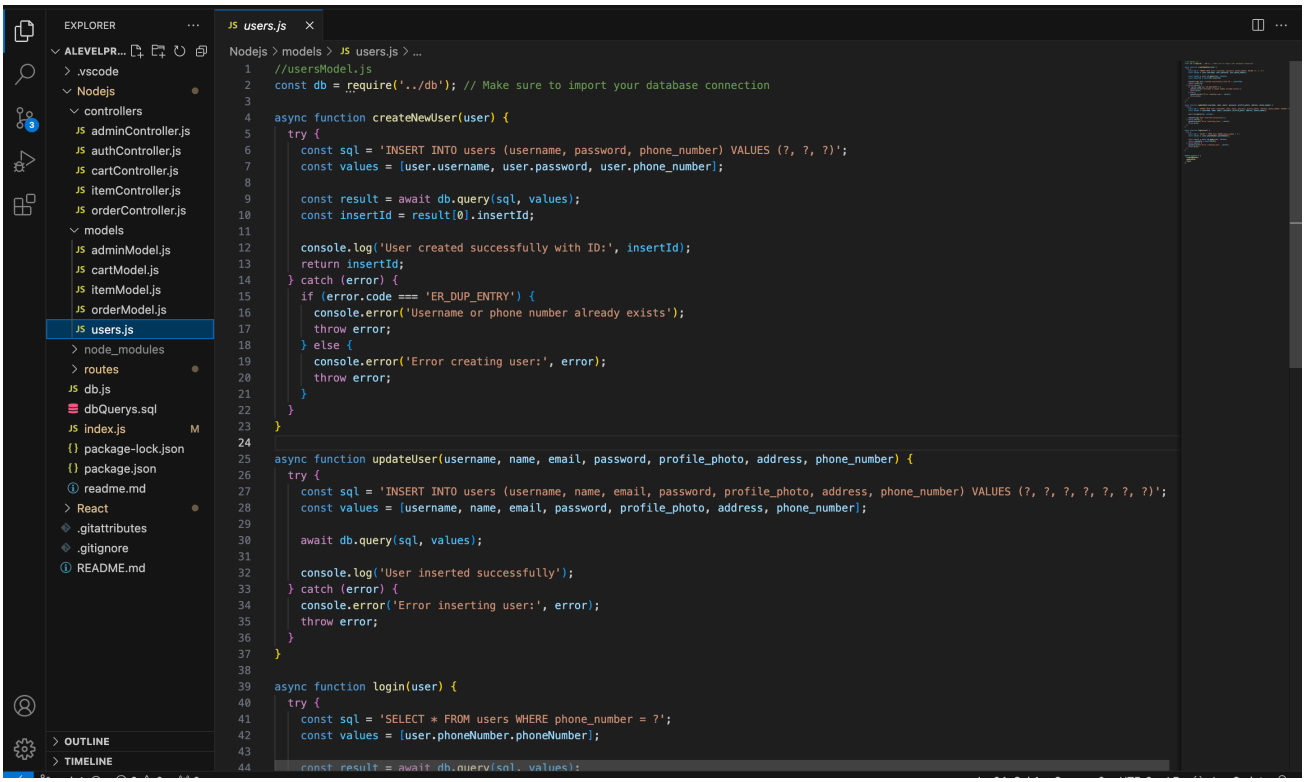


```
const db = require('../db');

// Retrieve all orders from the database
async function getAllOrders() {
  try {
    const results = await db.query('SELECT * FROM orders');
    return results[0];
  } catch (err) {
    throw err;
  }
}

// Retrieve a specific order by ID from the database
async function getOrderById(orderId) {
  try {
    const results = await db.query('SELECT * FROM orders WHERE id = ?', [orderId]);
    if (results.length === 0) {
      throw new Error('Order not found');
    }
    return results[0];
  } catch (err) {
    throw err;
  }
}

// Create a new order in the database
async function createOrder(orderData) {
  const { fullname, email, mobile, address, itemlist } = orderData;
  const itemListJson = JSON.stringify(itemlist); // Convert itemlist array to JSON
  const sql = 'INSERT INTO orders (fullname, email, mobile, address, itemlist) VALUES (?, ?, ?, ?, ?)';

  try {
    const results = await db.query(sql, [fullname, email, mobile, address, itemListJson]);
    return { id: results.insertId, ...orderData };
  } catch (err) {
    throw err;
  }
}

// Update an existing order in the database
async function updateOrder(orderId, orderData) {
  const { fullname, email, mobile, address, itemlist } = orderData;
  const itemListJson = JSON.stringify(itemlist); // Convert itemlist array to JSON
  const sql = 'UPDATE orders SET fullname = ?, email = ?, mobile = ?, address = ?, itemlist = ? WHERE id = ?';
```



```
async function createOrder(orderData) {
    throw err;
  }
}

// Update an existing order in the database
async function updateOrder(orderId, orderData) {
  const { fullname, email, mobile, address, itemlist } = orderData;
  const itemListJson = JSON.stringify(itemlist); // Convert itemlist array to JSON
  const sql = 'UPDATE orders SET fullname = ?, email = ?, mobile = ?, address = ?, itemlist = ? WHERE id = ?';

  try {
    const results = await db.query(sql, [fullname, email, mobile, address, itemListJson, orderId]);
    return results.affectedRows > 0 ? { id: orderId, ...orderData } : null;
  } catch (err) {
    throw err;
  }
}

// Delete an order from the database
async function deleteOrder(orderId) {
  try {
    const results = await db.query('DELETE FROM orders WHERE id = ?', [orderId]);
    return results.affectedRows > 0;
  } catch (err) {
    throw err;
  }
}

module.exports = {
  getAllOrders,
  getOrderById,
  createOrder,
  updateOrder,
  deleteOrder,
};
```

users.js


EXPLORER
users.js
Nodejs > models > users.js > ...
//usersModel.js
const db = require('../db'); // Make sure to import your database connection
async function createNewUser(user) {
try {
const sql = 'INSERT INTO users (username, password, phone_number) VALUES (?, ?, ?)';
const values = [user.username, user.password, user.phone_number];
const result = await db.query(sql, values);
const insertId = result[0].insertId;
console.log('User created successfully with ID:', insertId);
return insertId;
} catch (error) {
if (error.code === 'ER_DUP_ENTRY') {
console.error('Username or phone number already exists');
throw error;
} else {
console.error('Error creating user:', error);
throw error;
}
}
}
async function updateUser(username, name, email, password, profile_photo, address, phone_number) {
try {
const sql = 'INSERT INTO users (username, name, email, password, profile_photo, address, phone_number) VALUES (?, ?, ?, ?, ?, ?, ?)';
const values = [username, name, email, password, profile_photo, address, phone_number];
await db.query(sql, values);
console.log('User inserted successfully');
} catch (error) {
console.error('Error inserting user:', error);
throw error;
}
}
async function login(user) {
try {
const sql = 'SELECT * FROM users WHERE phone_number = ?';
const values = [user.phoneNumber.phoneNumber];
ALEVELPR...
.vscode
Nodejs
controllers
adminController.js
authController.js
cartController.js
itemController.js
orderController.js
models
adminModel.js
cartModel.js
itemModel.js
orderModel.js
users.js
node_modules
routes
db.js
dbQuerys.sql
index.js
M
package-lock.json
package.json
readme.md
React
.gitattributes
.gitignore
README.md
OUTLINE
TIMELINE


EXPLORER
users.js
Nodejs > models > users.js > ...
async function updateUser(username, name, email, password, profile_photo, address, phone_number) {
throw error;
}
}
async function login(user) {
try {
const sql = 'SELECT * FROM users WHERE phone_number = ?';
const values = [user.phoneNumber.phoneNumber];
const result = await db.query(sql, values);
return userdata = result[0][0];
} catch (error) {
console.error('Error creating user:', error);
throw error;
}
}
module.exports = {
createNewUser,
updateUser,
login
};
ALEVELPR...
.vscode
Nodejs
controllers
adminController.js
authController.js
cartController.js
itemController.js
orderController.js
models
adminModel.js
cartModel.js
itemModel.js
orderModel.js
users.js
node_modules
routes
db.js
dbQuerys.sql
index.js
M
package-lock.json
package.json
readme.md
React
.gitattributes
.gitignore
README.md

For installing and starting the server.

*Open the terminal and navigate to the root directory and run this command.

-> npm install or npm i

*For starting the server run this command

-> npm start or node index.js