University Inventory Management System
FullStack Web Development Final Project Journal

By:
Mounisha Konduru Raju(mkonduru@pdx.edu)
Hanesh Kongati(hanesh@pdx.edu)

Github Repo link - https://github.com/hanesh16/FullStack/

Here is my link for the recording:
https://drive.google.com/file/d/1iRYmARkaNRGDpX5As1PKmuCBbecb2kEh/view?usp=sharing

Backend Development:

Session 1: Environment Setup and Package Installation

We set up VsCode and Docker. Vscode npm init allowed us install and manage our project's dependencies. Docker, on the other hand, provided containerization capabilities, allowing us to package our backend application and its dependencies into a single, portable unit.

We incorporated several essential packages for our backend development, including Apollo GraphQL. Apollo GraphQL is a powerful Mongoose ORM framework that simplifies the management of entities and database interactions. It offers a lightweight and efficient solution, enabling us to easily define and manipulate entities, establish relationships between them, and carry out various database operations. With the help of Mongoose ORM, our project benefits

from a streamlined approach to handling data entities and simplifies our database-related tasks..

Session 2: Entity Creation and Database Integration

We have used schema update for updating the schema in the database and schema drop to delete the database. Later we have set the configuration for the database.

In order to define the database schema for our university inventory management system, we have introduced the necessary entities, namely Client and Items. These items or data models are represented by these entities in our program. We were able to describe the necessary characteristics, relationships, and constraints for these entities using Mongoose ORM entity decorators. The entity classes' seamless mapping to the associated database tables was made possible by these decorators, which served as annotations on the entity classes. This method made it easier to create and manage our data models and ensured a seamless integration with the underlying database architecture.

Session 3: MongoDB Compass and GraphQL Integration

MongoDB Compass is a powerful GUI tool that provides a visual interface for interacting with MongoDB databases. With MongoDB Compass, users can easily view, query, and manipulate data, as well as perform administrative tasks such as managing indexes and analyzing query performance.To integrate MongoDB Compass with GraphQL in our University Inventory Management system, we can leverage the capabilities of Apollo Server, a GraphQL server implementation. Apollo Server acts as a middleware between our GraphQL API and the MongoDB database.

By utilizing Apollo Server's integration with MongoDB, we can define resolvers that handle GraphQL queries, mutations, and subscriptions. These resolvers communicate with the MongoDB database through the MongoDB Node.js driver, which enables us to perform CRUD operations and interact with the database using GraphQL.

With this integration, GraphQL queries and mutations from the client-side can be seamlessly translated into MongoDB Compass-compatible operations. This allows for efficient data retrieval, manipulation, and synchronization between the client and the database.By combining the strengths of MongoDB Compass's visual interface and the flexibility of GraphQL, we can simplify and enhance the process of managing and interacting with our University Inventory Management system's data.

Session 4: Route Implementation, Testing and Installing Plugins

To optimize our backend's functionality and efficiency, we incorporated various plugins, including GraphQL, a web framework designed for building scalable and performant server applications. GraphQL introduced essential features like route handling, request validation, error management, and more. These plugins streamlined our development workflow and significantly improved the overall backend performance.In our University Inventory Management system, we meticulously designed and implemented routes for each entity. These routes serve as API endpoints that enable clients to interact with the system. Leveraging GraphQL's query and mutation mechanisms, we defined URL paths, request methods, and associated handler functions for each endpoint. These route handlers interacted with the database using Mongoose ORM's querying and persistence methods to retrieve or modify data. We employed Postman to thoroughly test these routes, ensuring their proper functioning and adherence to requirements.

Upon creating the routes, it was crucial to register them to make them active and accessible. This registration step ensures that the routes are correctly linked and operational within the backend system, ready to handle client requests.

Session 5:Code Formatting and Refinement

We integrated ESLint, a popular code linter and formatter, to maintain code consistency and improve the overall quality of our codebase. Enforcing coding standards, spotting any mistakes or code difficulties, and formatting the code in accordance with specified criteria were all made possible by the use of ESLint. By using ESLint, we were able to make sure that our code followed best practices, promoting clear and understandable code throughout the development process.

Frontend Development:

Session 6: Project Initialization, Styling and Routing

We began frontend development by constructing a React component. Vite is a quick and lightweight build tool for modern web apps, while React is a popular JavaScript package for developing user interfaces. The basic project structure was scaffolded with the essential files and directories when the Reactapp was created.

We developed a number of components to handle various functions within our University Inventory Management system. These components are self-contained, reusable bits of code that encapsulate certain user interface elements. ItemCard, Header, ClientRow, and Spinner are a few examples of these components. Each component was created to control the display and interaction logic for its appropriate capability, ensuring system modularity and reusability.

We applied CSS styles into our components to ensure a visually pleasing and consistent user interface. Each component had its own CSS file in which we defined the desired styles using CSS selectors, classes, and properties. These styles included layout, colors, font, and other design features, creating a consistent and visually pleasant appearance across our components.

Session 7: API Integration and Database Connectivity

Using tools such as React Router, we integrated routing capabilities into our frontend. Based on user interactions and URL changes, this provided seamless navigating between different components. We created routes for distinct pages and components, such as Home, Item, EditItemForm, and AddClientForm, with each route corresponding to a separate component. We could access and interact with individual components manually or through a navbar that displayed accessible features as independent routes by altering the route. Furthermore, we

used ApolloClient to interface with the backend API endpoints, allowing us to retrieve and update data in real time.