Week 16: Assignment Work - Phone Book (Backend & Full-Stack Sync)

Day 1: RESTful API Design for Contact Management

Theoretical Overview

The backend for the Phone Book assignment was built as a specialized RESTful API to handle persistent storage for the frontend application. Unlike the Gadget API, which was primarily read-heavy, the Phone Book API requires high-frequency Create, Update, and Delete operations to manage personal user data.

Endpoint Architecture

- **GET /api/contacts**: Retrieves the complete list of contacts from the MongoDB collection.
- **POST /api/contacts**: Validates and saves a new contact entry sent from the React frontend.
- **PUT /api/contacts/:id**: Finds a specific contact by its unique ID and updates fields like phone number or email.
- **DELETE /api/contacts/:id**: Permanently removes a contact from the database.

Day2: MongoDB Persistence and Schema Implementation

Structuring the "Contacts" Collection

Using the MongoDB Data Model, we established a "Contacts" collection within the database.

- **Document Structure**: Each document represents a single contact, utilizing the BSON format to store fields such as name (String), phone (String), email (String), and category (String).
- **Unique Identifiers**: MongoDB auto-generates an _id field for every document, which serves as the primary key for the React frontend to target specific contacts for deletion or editing.
- **Data Integrity**: We implemented server-side checks to ensure that mandatory fields, like the contact name, are present before the insertOne() operation is executed.

Day 3: Controller Logic and Error Handling

Backend Logic Separation

Following the "Separation of Concerns" principle practiced in the **Gadget API** , we moved the business logic into a dedicated controller file.

- **Search and Filter**: While the frontend handles real-time filtering, the backend supports query parameters to limit the data sent over the network, improving performance.
- **Error Middleware**: We implemented custom error-handling middleware to catch "404 Not Found" scenarios if a user tries to edit a contact that has already been deleted.
- **Response Codes**: The API was programmed to send a 201 Created status for successful additions and a 400 Bad Request if the data format is invalid.

Day 4: Full-Stack Synchronization and CORS

Connecting Frontend to Backend

To allow the React application (running on port 3000) to communicate with the Express server (running on port 5000), we had to address security protocols.

- **CORS (Cross-Origin Resource Sharing)**: We implemented the CORS middleware to authorize the frontend's origin, allowing it to perform POST and DELETE requests without being blocked by the browser.
- **Asynchronous Fetching**: The frontend useEffect hook was synchronized with the GET /api/contacts endpoint to ensure the UI is always up-to-date with the database upon loading.
- **State Reconciliation**: We developed a logic where the frontend state only updates *after* the backend confirms a successful database operation, ensuring data consistency.

Day 5: Final Testing and Debugging

End-to-End Validation

The final day was dedicated to testing the entire lifecycle of a contact.

- **CRUD Validation**: We verified that adding a contact in the UI correctly triggered the db.collection.insert() command in the background.
- **Network Tab Monitoring**: Used the Browser Developer Tools to inspect JSON payloads, ensuring that the req.body parsed by the backend correctly matched the state sent by React.
- **Using MongoDB from Node**, demonstrating our ability to build a fully persistent full-stack application.