Contact Management System

Project Description

The Contact Management System is a full-stack application that enables users to manage contact information seamlessly. It includes features to add, view, update, and delete contacts, making it ideal for businesses to maintain relationships with clients and customers. The frontend is developed using ReactJS and Material UI (MUI) for an elegant user interface, while the backend utilizes Django REST framework for handling data and API endpoints. MySQL is used as the database for its robust data management capabilities.

Major Technical Decisions and How the App Works

1. Backend Development with Django REST Framework

Overview: The backend is responsible for processing API requests, interacting with the database, and returning responses to the frontend.

Key Steps:

- Model Creation: Defined a Contact model with fields such as first name, last name, email, phone number, company, and job title.
- API Endpoints: Created API endpoints using Django REST framework for each
    - CRUD operation:
    - POST /contacts: Accepts new contact data and saves it to the database.
    - GET /contacts: Retrieves a list of all contacts to display in the frontend.
    - PUT /contacts/ : : Updates contact information based on the provided ID.
    - DELETE /contacts/: : Deletes a contact from the database by ID.
- Validation and Error Handling: Implemented input validation to ensure required fields are filled and email addresses are unique. Proper error responses are returned for invalid data.

Technical Decision: Using Django REST framework simplifies building RESTful APIs with built-in tools for serialization, authentication, and CRUD operations.

2. Frontend Development with ReactJS and Material UI

Overview: The frontend is designed for an intuitive user experience where users can interact with contact information seamlessly.

Key Steps:

- Form Creation: A form component was built using MUI components, allowing users to input details like first name, last name, email, phone number, company, and job title.

- Contacts Table: Displayed a table with MUI that lists all contacts, complete with sorting and pagination features for easy browsing of large datasets.
- Interactivity:
    - Add Contact: When users submit the form, an API call sends the data to the backend, where it's stored in the database.
    - Edit Contact: Users can click an edit button, pre-fill the form with existing contact data, make changes, and submit to update the backend via an API call.
    - Delete Contact: A delete button removes a contact entry by sending a delete request to the backend.
- Error Handling: Handled errors gracefully with feedback to the user if any required fields are missing or data is invalid.

Technical Decision: MUI was chosen for its pre-built, customizable React components that align with modern design principles, ensuring a consistent and attractive UI.

3. Database Choice: MySQL

Overview: MySQL was selected for its reliability and efficiency in handling relational data, making it suitable for structured contact records.

Key Steps:

- Database Schema: Designed to match the Contact model, allowing the backend to perform seamless read and write operations.
- Integration: Connected the Django backend to the MySQL database using Django's built-in database management capabilities.
- Technical Decision: MySQL's support for scalability and structured data made it an ideal fit for this project, given the potential for a large number of contact entries.

How Each Part of the App Works

- User Interaction: Users interact with the React frontend, inputting and managing contact data via forms and tables.
- API Communication: The frontend sends API requests to the Django backend, which processes these requests, interacts with the MySQL database, and sends responses back to the frontend.
- Data Flow:
    - Create: User submits a new contact -> React form sends a POST request -> Django validates and saves data to MySQL -> Response sent back confirming success.
    - Read: User navigates to view contacts -> React sends a GET request -> Django fetches data from MySQL -> Contacts displayed in an MUI table.
    - Update: User edits a contact -> React sends a PUT request -> Django updates the record in MySQL -> Confirmation sent back.
    - Delete: User clicks delete -> React sends a DELETE request -> Django removes the record from MySQL -> Response confirming deletion.

Challenges and Solutions

Challenge 1: Handling Pagination and Sorting in the Contacts Table

Solution: Integrating pagination and sorting using Material UI (MUI) initially posed a challenge due to the complexity involved in managing state and data. To overcome this, I researched online tutorials, MUI documentation, and community forums to gain a better understanding of these features. Through this research, I learned how to implement MUI's table components effectively, customize sorting behavior, and enable pagination. With these insights, I successfully integrated a seamless user experience for navigating large contact lists, allowing users to view data in a structured and manageable way.

Challenge 2: Testing Endpoints with Postman

Solution: Testing the API endpoints using Postman took longer than anticipated, particularly when diagnosing unexpected responses or errors in the request/response cycle. To address these issues, I carefully examined the backend code, validated request payloads, and ensured proper status code returns for various scenarios. This step-by-step debugging process helped identify and rectify errors, ensuring that the GET, POST, PUT, and DELETE endpoints functioned correctly and consistently. The thorough testing approach provided confidence in the stability and reliability of the backend.

Challenge 3: Connecting the React Frontend with the Django Backend

Solution: Integrating React and Django required special attention, particularly with handling cross-origin requests and ensuring smooth data exchange. I used Axios to facilitate HTTP requests between the React frontend and the Django REST API. During integration, I encountered minor issues such as CORS errors and misconfigured request headers. By configuring the django-cors-headers package in the Django project and adjusting Axios request settings, I resolved these connectivity issues. Additionally, I double-checked for typos, path mismatches, and response handling logic to ensure consistent communication between the frontend and backend.