DOCUMENTATION - Building a Full-Stack To-Do List Application with FastAPI

This project is all about building a To-Do List application that connects a smooth, user-friendly front-end with a powerful, efficient back-end. On the front-end, we are tasked to utilize a **React** to create the interactive interface that users see and interact with in their browsers. React is great for building responsive and dynamic user experiences, making it easy for users to quickly add, complete, and manage their tasks in real time.

On the back-end, we're using **FastAPI**, which is known for its speed and modern features. FastAPI will handle all the data-related tasks behind the scenes—like storing to-do items and managing actions such as creating, updating, or deleting them. It also defines what each to-do item looks like (for example, the title and whether it's completed), and sets up API endpoints so the front-end can talk to the back-end. An added feature is task filtering, allowing users to view all tasks, only the completed ones, or just the pending ones—making it easier to stay organized.

The **React front-end** will be responsible for how the app looks and feels. It will include interactive elements like buttons and lists so users can easily add, edit, delete, and view tasks. There will also be a **dark mode toggle** for a more comfortable experience in low-light settings. All user actions on the front-end will send requests to the FastAPI back-end to make sure everything is saved and updated properly.

A key focus of this project is making sure the front-end and back-end work seamlessly together. This means setting up smooth communication between the two—making sure data is passed accurately, and that what the user sees always reflects what's stored in the back-end.

This project also includes documenting the code, comparing **FastAPI** with **Django REST Framework**, and deploying the app online so it can be accessed from anywhere.

Backend Documentation (FastAPI)

This documentation provides an overview of the FastAPI backend, which serves as a RESTful API for managing todo tasks. The backend utilizes SQLAlchemy for Object-Relational Mapping (ORM) and SQLite as the database. It supports CRUD operations on todo items and incorporates Cross-Origin Resource Sharing (CORS) to facilitate communication with the frontend.

Overview

The FastAPI application provides a RESTful API for managing todo tasks. It uses:

- **SQLAlchemy for ORM (Object-Relational Mapping).**
- **SQLite as the database.**
- **CORS middleware to allow requests from specified origins.**

The backend supports CRUD operations on todo items.

Database Setup

The database configuration is located in database.py.

- **SQLAlchemy is used to interact with the database.**
- **test.db is the SQLite database file.**
- **SessionLocal acts as a session factory, creating database sessions.**
- **The Base class, defined in base.py, serves as the foundation for database models.**
- **Base.metadata.create_all(bind=engine) ensures that the database tables are created automatically when the application starts.**

Models (models.py)

The models.py file defines the structure of the database tables using SQLAlchemy.

- **Book: Represents a book with the following attributes:**
  - **id: Integer, primary key.**
  - **title: String, title of the book.**
  - **author: String, author of the book.**
- **TodoItem: Represents a todo task with the following attributes:**
  - **id: Integer, primary key.**
  - **title: String, title of the task.**
  - **completed: Boolean, indicates whether the task is completed (defaults to**

False).

Schemas (schemas.py)

The schemas.py file defines Pydantic models used for data validation and serialization. These schemas ensure that the data exchanged between the client and server adheres to a specific structure.

- TodoItemBase:
    - Base Pydantic model for todo items.
    - Attributes:
        - title: String, task title.
        - completed: Boolean, task completion status (defaults to False).
- TodoItemCreate:
    - Pydantic model used for creating new todo items.
    - Inherits from TodoItemBase.
    - Attributes:
        - title: String, task title.
        - completed: Boolean, task completion status (defaults to False).
- TodoItemUpdate:
    - Pydantic model used for updating existing todo items. Allows partial updates.
    - Attributes:
        - title: String, task title (optional).
        - completed: Boolean, task completion status (optional).
- TodoItemResponse:
    - Pydantic model used for representing todo items in API responses.
    - Extends TodoItemBase to include the id.
    - Attributes:
        - id: Integer, task ID.
        - title: String, task title.
        - completed: Boolean, task completion status.

API Endpoints (main.py)

The main.py file defines the API endpoints using FastAPI. Each endpoint handles specific HTTP requests and interacts with the database.

- GET /:
  - Returns a welcome message (e.g., "Hello, world!").
  - Operation ID: N/A
- POST /tasks/:
  - Creates a new todo task.
  - Expects a request body conforming to the TodoItemCreate schema.
  - Returns the created TodoItem with its ID.
  - Operation ID: create_task
- GET /tasks/:
  - Retrieves a list of all todo tasks.
  - Accepts an optional query parameter completed to filter tasks by their completion status (e.g., /tasks/?completed=true or /tasks/?completed=false).
  - Returns a list of TodoItemResponse objects.
  - Operation ID: get_tasks
- GET /tasks/{task_id}:
  - Retrieves a single todo task by its ID.
  - task_id is a path parameter.
  - Returns the TodoItemResponse if found.
  - Returns a 404 HTTP error if the task with the given ID does not exist.
  - Operation ID: get_task_by_id
- PUT /tasks/{task_id}:
  - Updates an existing todo task.
  - task_id is a path parameter.
  - Expects a request body conforming to the TodoItemUpdate schema (allows partial updates).
  - Returns the updated TodoItemResponse.
  - Returns a 404 HTTP error if the task with the given ID does not exist.
  - Operation ID: update_task
- DELETE /tasks/{task_id}:
  - Deletes a todo task by its ID.

    - task_id is a path parameter.
    - Returns a confirmation message upon successful deletion.
    - Returns a 404 HTTP error if the task with the given ID does not exist.
    - Operation ID: delete_task

Middleware

- CORS (Cross-Origin Resource Sharing):
    - CORS middleware is enabled to allow requests from specific origins. This is crucial for enabling communication between the frontend and backend, which might be served from different domains during development.
    - The application is configured to allow requests from the following origins:
        - http://localhost:5174
        - http://127.0.0.1:5174
        - http://RACHEL18ANN.github.io
    - The configuration uses allow_origins=["*"]. This allows all origins. For production, you should replace this with a list of specific, trusted origins to enhance security.
    - Other CORS settings:
        - allow_credentials=True allows cookies to be sent in cross-origin requests.
        - allow_methods=["*"] allows all HTTP methods (e.g., GET, POST, PUT, DELETE). This should be restricted in a production environment.
        - allow_headers=["*"] allows all HTTP headers. This should also be restricted in production.

Dependency Injection

- **get_db():**
    - **This function is used as a dependency to provide a database session to the route handlers.**
    - **It ensures that a new database session is created for each request and that the session is properly closed after the request is processed. This prevents potential database connection leaks.**

Documentation for frontend/todo-app/src files

This documentation provides an overview of the purpose and functionality of the main files in the frontend/todo-app/src directory.

main.jsx

This is the entry point for the React application. Its primary responsibilities include:

- **Imports:**
    - Imports StrictMode from React to enable stricter checks during development.
    - Imports createRoot from react-dom/client for managing the React application's root node.
    - Imports global CSS styles from index.css to provide a consistent look and feel.
    - Imports the main application component, App, from App.jsx.
- **Rendering:**
    - Uses createRoot to create a root node attached to the DOM element with the ID 'root'. This is where the React application will be rendered.
    - Renders the App component within a StrictMode wrapper. StrictMode helps highlight potential problems in the application during development, such as:
        - Identifying unsafe lifecycles.
        - Warning about legacy string ref usage.
        - Detecting unexpected side effects.
        - Detecting legacy context API usage.

In essence, main.jsx sets up the React application, applies global styles, and mounts the main component (App) to the specified DOM element, enabling React's component-based rendering.

TaskList.js

This file defines the TaskList React component, which manages and displays a list of tasks, including fetching, adding, and editing them, often with real-time updates.

Key aspects:

- **State Management:**
    - Uses the useState hook to manage the list of tasks.
    - Uses the useEffect hook to perform side effects, such as fetching data and setting up WebSocket connections.
- **Real-time Communication:**
    - Establishes a WebSocket connection to a server (specifically, http://localhost:5173/fastapi_todoapp/) using the socket.io-client library. This connection is used for real-time updates.
    - Listens for WebSocket events:
        - taskAdded: When a new task is added on the server, the component updates its local task list.
        - taskUpdated: When an existing task is modified on the server, the component updates the corresponding task in its local list.
- **Data Fetching:**
    - On component mount (using useEffect), it fetches the initial list of tasks from the /tasks endpoint, likely using a fetch or similar API call.
- **Task Management Functions:**
    - addTask(newTaskTitle):
        - Implements optimistic UI updates:
            - Immediately adds a temporary task to the local state with a unique ID. This makes the UI feel responsive.
        - Sends a POST request to the server's /tasks endpoint to persist the new task.
        - Handles the server response:
            - If the server confirms the addition, the temporary task is updated

        with the server's data (if different).
        - If an error occurs, the temporary task is removed from the local state, rolling back the optimistic update.
    - editTask(taskId, newTitle):
        - Implements optimistic UI updates:
            - Immediately updates the task's title in the local state.
        - Sends a PUT request to the server's /tasks/{taskId} endpoint to update the task.
        - May include error handling to rollback the local update if the server update fails.
- **Cleanup:**
    - In the useEffect hook, a cleanup function is used to disconnect from the WebSocket server and remove event listeners when the component unmounts. This prevents memory leaks.
- **Rendering:**
    - The component is responsible for rendering the list of tasks, including displaying task titles and providing UI elements for adding and editing tasks. (The specific UI rendering code is not provided in the original snippet, but would typically involve mapping over the task list and displaying each task.)

In summary, TaskList.js manages the presentation and manipulation of a list of tasks, coordinating between the local UI state and a remote server, often using WebSockets for real-time synchronization.