**Short Write-Up: Task\_Management\_System**

**Overview :**

The **Task Management System** is designed as a full-stack web application, leveraging **React.js (frontend)**, **Node.js & Express.js (backend & API gateway)**, and **PostgreSQL (database)**. The goal is to provide a scalable, efficient, and user-friendly way to manage tasks while ensuring **data integrity, validation, and API security**.

**API & Database Design:**

The backend API is built using **Express.js and Sequelize ORM**, ensuring clean separation between **business logic and data handling**. PostgreSQL was chosen for its **robust ACID compliance and constraint handling**.

* **Unique constraints** on task titles prevent duplicate entries.
* **Due date validation** ensures tasks cannot be scheduled in the past.
* A **PostgreSQL trigger** automatically marks tasks as **"Overdue"** when their due date has passed.

This ensures that the **database remains clean and consistent** without requiring additional backend logic.

**Frontend Implementation:**

The frontend is developed using **React.js**, featuring:

* **State management (useState)** for handling tasks dynamically.
* **Custom hooks (useTasks.js)** for API interactions.
* **Filters for high-priority tasks & upcoming deadlines.**

This approach ensures a **responsive and interactive user experience**.

**Backend Implementation:**

* The backend is built using **Node.js with Express.js**, featuring:  
   **REST API with Sequelize ORM** to interact with PostgreSQL.
* **Middleware for error handling** to return **meaningful API responses**.
* **Database constraints (unique title, valid due dates)** to ensure data integrity.
* **API Gateway middleware** for **request validation and logging**.

These implementations ensure a **secure, scalable, and efficient** system

**API Gateway & Middleware :**

In my project, I implemented a Node.js API Gateway to enhance security and logging. This gateway includes request logging, which tracks the HTTP method, endpoint, and timestamp for better monitoring and debugging.

Additionally, I added validation middleware to reject tasks containing "test" in the title, ensuring data integrity and preventing unwanted entries. The gateway also handles request forwarding, securely routing API requests to the backend for processing.

By implementing these features, I improved the system's security, request monitoring, and overall, API reliability.

**Challenges & Solutions :**

1. I encountered a **Circular JSON Error** in React when trying to **stringify objects** for API requests or debugging. This happened because some objects contained **circular references**, meaning they **referenced themselves**, causing JSON.stringify() to fail.

To fix this issue, I ensured that only **valid objects** were passed to JSON.stringify(). I also used **structured cloning** or manually removed problematic references before converting objects to JSON.

By implementing these fixes, I prevented crashes and ensured **smooth API communication and state updates** in React.

1. I faced multiple **API issues** like **invalid requests, database constraints, and network failures**. To resolve this:

**Backend:** Added **Express.js middleware** to return clear error messages (e.g., **409 Conflict** for duplicate titles).  
**Frontend:** Implemented **UI alerts** to notify users of **failed API requests** or **invalid inputs**.

1. I initially planned to implement **PostgreSQL using Docker**, but faced configuration issues, including **networking and connection errors**. Due to these challenges, I decided to **install PostgreSQL manually** instead. My database is now stored in **pgAdmin 4**, which provides an intuitive interface for managing tables and queries. While **Docker would have been more scalable**, the manual installation allowed me to proceed efficiently. In the future, I plan to revisit **Docker implementation** for better deployment flexibility