My Task Management System is designed with a focus on intuitive, scalable, and collaborative task management. At its core, the system uses a relational database with three main tables. The Users table stores essential information—usernames, emails, passwords, and unique IDs—to ensure every user has a secure and personalized experience. TaskItems holds all the details about each task, such as titles, descriptions, due dates, statuses, and even dynamically calculated priorities. Finally, the TaskSubscriptions table links users and tasks together, enabling multiple users to follow and update the same task in real time, which helps maintain data integrity and smooth interaction across the application.
On the backend, our system relies on several controllers and services to manage different aspects of task management. The TasksController is responsible for creating, updating, retrieving, and deleting tasks. When a task is modified, updates go straight to the database, and any changes are broadcast to connected clients to keep everyone on the same page. One of the more interesting parts of our system is the Task Priority Service. This service calculates the urgency of a task by combining the category's inherent priority, a user-assigned priority, and the proximity of its due date, using a weighted average and logarithmic time adjustments to ensure tasks near their deadlines stand out. Other controllers, like the UsersController and CategoriesController, handle user registrations, authentication, and category management, ensuring that both individual and organizational needs are addressed.
A standout feature of our solution is its real-time update capability, enabled by SignalR. By establishing a continuous, two-way connection between client and server, SignalR makes sure that any change made to a task is instantly communicated to all interested users. The SignalR connection is managed by our client-side script (signalr-connection.js), which uses WebSockets and automatic reconnection strategies to keep the connection stable. On the server side, our TaskHub groups client connections based on their task subscriptions. For instance, when a user updates a task via the UpdateTaskModal, the updated data is sent to the server, and TaskHub then notifies all connected clients in that task group so that everyone's view stays up to date without needing to refresh their page.
The front end, built with React, is structured into modular components like myTasks, TaskHub, and a comprehensive Dashboard. These components interact with a global state managed by Recoil (supported by local storage) to ensure session persistence, even after a page reload. This design makes it easy for users to create, update, and manage tasks, subscribe for real-time updates, and stay organized via an interface that's responsive and user-friendly.
However, there are some limitations to our approach. We currently rely on session-based authentication using Recoil state and local storage rather than a

token-based approach like JSON Web Tokens (JWTs). Although sessions are easier to manage in a single-server setup, they can become challenging to scale horizontally and might expose the application to security risks like XSS attacks. A token-based system would offer stateless authentication with cryptographic signatures and short expiration times, but it would also add extra complexity. Additionally, our task priority calculation uses a weighted average approach that combines category priority, user-assigned priority, and a logarithmic time factor. While this is simple to understand and implement, it depends heavily on heuristically determined weights and can be sensitive to outliers. This might result in inaccurate task ordering if the weights or due dates aren't calibrated perfectly.
There's also a limitation with our real-time notifications: the SignalR-based TaskHub only broadcasts updates for tasks that are marked as "Incomplete" or "InProgress." While this exclusion of completed tasks helps reduce clutter, it may overlook other statuses—like "Delayed" or "Paused"—that could be important in more complex scenarios. Lastly, the current implementation in our React components lacks comprehensive runtime input validation. This means that user input might not always be thoroughly checked, leading to potential data errors and impacting overall reliability. As the system grows, addressing these limitations could significantly enhance both its performance and security.