**Project Management System**

**Problem Statement:**The main problem I focused on in my project was how difficult it is to manage software projects without a proper system. Teams often struggle to keep track of tasks, deadlines, and communication. This can cause confusion, delays, and low productivity. My project aims to solve this by providing a simple and organized way to manage everything in one place

**Role & Responsibilities:**I worked on the project by myself and took care of both the frontend and backend. I built all the features and made sure everything worked well together

**Tech Stack & Architecture:**The system is built using a **microservices architecture**, where each core feature—such as task management, user roles, notifications, and calendar—is developed as an independent service. This modular approach allows for easier development, deployment, and scaling of individual components.

* **Backend:** I used Spring Boot to rapidly develop robust, production-ready microservices. For efficient data querying and real-time updates, I integrated GraphQL, which allows clients to request precisely the data they need, solving common REST API issues like overfetching or underfetching. MYSQL serves as the primary relational database, handling structured data such as tasks, subtasks, and user roles.
* **Inter-service communication:** To maintain asynchronous, event-driven communication between microservices, I implemented Apache Kafka. For instance, when a new task is created, the task service produces an event to a Kafka topic, which the calendar service consumes to update deadlines and task progress accordingly.
* **Frontend:** I built the UI using React.js. Its component-based structure helped me easily manage dynamic features like task lists, user management, and notifications. I used React’s Context API for state management to keep the app lightweight and easy to maintain.
* **Real-time updates:** I used GraphQL subscriptions to send instant notifications without needing a page refresh. This allowed users to get live updates on task changes, comments, and assignments, which made the app more responsive and engaging

**Detailed Explanation:**

My Project Management System is a comprehensive platform designed to streamline team collaboration by breaking down complex projects into manageable hierarchical components: projects, epics, stories, tasks, subtasks, and bugs. This multi-level structure mirrors real-world software development workflows, allowing teams to organize and prioritize work effectively.

Users start by creating a project, which acts as a container for all related work items. Inside a project, users can define epics—large bodies of work that break down into smaller user stories. Each story contains tasks and subtasks, facilitating detailed tracking of progress. Crucially, the system supports bug tracking, enabling testers or team members to report bugs immediately and assign them to the appropriate developers, accelerating resolution.

Every item—be it a task, bug, or story—can be assigned to specific team members, providing clear ownership and accountability. When a bug is created, for example, the tester assigns it directly to the developer responsible, fostering quick and transparent collaboration.

Beyond assignments, users can customize notification preferences to control how and when they receive updates. Notifications are delivered in real-time via GraphQL subscriptions, ensuring users stay informed without needing to refresh the page. Users receive alerts instantly for important events such as task updates, new comments, or bug assignments.

The backend’s microservices architecture provides scalability and maintainability. Each service—task management, bug tracking, notifications, calendar—runs independently but communicates asynchronously through Kafka. For example, when the task service creates a new task, it emits an event that the calendar service consumes to update deadlines and progress on a visual calendar, offering users a clear overview of timelines and priorities.

Users can update task statuses, add live comments, and edit or delete those comments as needed. This fosters transparent communication and easy tracking of discussions and decisions. The system also allows archiving of completed or inactive tasks and stories, maintaining a clean and focused workspace while preserving historical data.

Email notifications integrated through Gmail SMTP ensure users receive updates even outside the platform. When a user is assigned or unassigned from a task, or when task details change, an email alert is sent to keep them informed at all times.

Security is a core focus. I implemented JWT (JSON Web Token) authentication to provide secure, stateless user sessions. Additionally, I built session management to restrict accounts to a single active login session, preventing unauthorized access from multiple devices simultaneously. Users can securely update their profiles within the app.

The system also includes partial Git integration, linking commits or branches to specific tasks or stories. Though not fully complete, this feature demonstrates the potential for seamless DevOps workflows and version control integration in future enhancements.

**Future Scope:**

1. **Full Git Integration:** Expand to fully link commits, branches, and pull requests with tasks for better traceability and streamlined development workflows.
2. **Offline Mode and Sync:** Enable users to work offline and sync changes automatically when back online, improving usability in low-connectivity situations.
3. **Enhanced Collaboration & Tool Integration:** Add features like video calls and document editing within the app, plus integrate with popular tools like Slack, Jira, and Google Calendar to boost team collaboration and productivity.

**Challenges & Solutions:**

1. **Implementing Real-Time Notifications:  
   *Challenge:*** Delivering instant updates without overwhelming the server or relying on inefficient client-side polling.  
   **Solution:** I leveraged GraphQL subscriptions using WebSockets, enabling efficient event-driven communication that pushes updates only when necessary, minimizing server load and improving user experience.
2. **Performance Optimization in GraphQL:  
   *Challenge:*** Avoiding the N+1 query problem and preventing unnecessary data fetching which can slow down performance.  
   **Solution:** I also optimized GraphQL queries to fetch only the required fields, minimizing data transfer and improving response times.
3. **React State Management for Real-Time Data:  
   *Challenge:*** Managing frequent updates from subscriptions in React without causing UI glitches or excessive re-rendering.  
   **Solution:** I used React Context API to manage subscription data efficiently and implemented throttling to limit update frequency, ensuring a smooth and responsive user interface.

**Important Questions?**

**How did you break your application into microservices?**

I divided the system into several focused microservices based on distinct business capabilities and responsibilities to ensure modularity, independent scalability, and ease of maintenance. Specifically:

1. **User Service:**  
   Handles everything related to user management — authentication (using JWT), authorization, profile updates, and session control. This service is responsible for securing user data and managing login sessions, ensuring each user can only be logged in on one device at a time.
2. **Project Service:**  
   Manages projects, epics, and stories. It handles creating, updating, and archiving projects and their hierarchical components, keeping the core project structure isolated for better organization and control.
3. **Task Service:**  
   Focused on task, subtask, and bug management. It deals with task creation, status changes, assignment to users, live comments, and related business logic.
4. **Notification Service:**  
   Dedicated to managing all notifications, including real-time updates via GraphQL subscriptions and sending email alerts through Gmail SMTP. This separation ensures notifications can scale independently and are delivered reliably without impacting other services.
5. **Activity Service:**  
   Tracks user activities and changes in the system, like task updates, comments, and status changes, providing an audit trail for transparency and accountability.
6. **API Gateway & Discovery Server:**  
   The API Gateway serves as the single entry point for client requests, routing them to the appropriate microservices and handling cross-cutting concerns like authentication and rate limiting. The Discovery Server helps services find each other dynamically, supporting scalability and resilience.

**Why did you choose microservices over monolithic architecture?**

I chose microservices because they allow each feature or module to be developed, deployed, and scaled independently. This modularity improves maintainability and enables faster development cycles. In contrast, a monolithic app becomes harder to manage as it grows larger.

**How did you handle communication between microservices?**

I used asynchronous communication via Kafka, which allows services to send and receive events without waiting for each other. This event-driven approach reduces tight coupling and improves system resilience and scalability.

**How do you ensure data consistency across microservices?**  
Instead of immediate consistency, I implemented eventual consistency. Services update their own databases when they consume relevant Kafka events, which keeps data aligned across services over time without locking or direct synchronous calls.

**How did you ensure security between microservices?**

I implemented JWT-based authentication and authorization at the API Gateway level to ensure only authenticated requests reach the services. For inter-service communication, I used secure tokens and validated them to prevent unauthorized access.

**What strategies did you use to monitor and debug issues in a distributed system?**

I implemented centralized logging using tools like ELK Stack (Elasticsearch, Logstash, Kibana) to aggregate logs from all microservices, making it easier to trace requests across services. For monitoring, I used Prometheus and Grafana to track metrics like response times, error rates, and system health.