Project Objectives and Planning

Our project at Trello was driven by the need to address critical pain points that were consistently highlighted through user feedback and internal performance analysis. The two primary concerns were the **slow loading times** experienced by users with large boards, and the **increasing need for robust data security** as our user base grew, particularly among enterprise clients.

Identifying Pain Points

- **User Feedback:** We regularly received feedback from users who were frustrated with the sluggish performance of Trello when working with large boards containing hundreds of cards. For example, users reported that opening a board with over 500 cards would lead to significant delays, negatively impacting their productivity.
- **Enterprise Client Concerns:** Our enterprise clients, particularly those in sectors like finance and healthcare, expressed concerns about the security of their data on Trello. They needed assurances that their sensitive information was protected against unauthorized access, especially given the collaborative nature of Trello where multiple users could access shared boards.

Project Planning

Based on these insights, we defined the following key objectives for the project:

1. **Reduce Load Times:** To tackle the issue of slow performance, our primary goal was to optimize how data was loaded and rendered in the application. This involved implementing techniques like **lazy loading** to defer the loading of non-essential components until they were needed. For example, rather than loading all cards when a board was opened, we decided to load only the visible cards and fetch additional cards as the user scrolled, thereby reducing initial load times.
2. **Enhance User Interface (UI):** The goal was to make the UI more intuitive and user-friendly, especially for users managing large boards. We aimed to redesign the navigation system to allow users quicker access to frequently used boards and cards. For instance, we planned to introduce a customizable sidebar where users could pin their most accessed boards, reducing the time spent navigating between different sections of the application.
3. **Strengthen Data Security:** Given the concerns around data security, we aimed to enhance the platform's security framework, focusing on **authentication** and **authorization**. We planned to implement **JWT-based authentication** to provide a more secure, scalable, and stateless solution. This was particularly crucial for our enterprise clients who required strict access controls to ensure that only authorized personnel could view or modify sensitive information.

Architecture Design:

Framework Selection: We chose **Angular version 15** for the frontend framework due to its enhanced performance features and compatibility with Angular Material. The decision was based on Angular 15's improved change detection and rendering engine, which were critical for handling Trello's complex, data-driven UI components. This version also supported better build optimizations, which was necessary to maintain the application's responsiveness even as it grew in complexity.

Folder Structure and Module-Based Architecture: To ensure modularity, maintainability, and scalability, we organized the project using a module-based architecture. The folder structure was divided into feature modules (e.g., boards, cards, users), shared modules (e.g., common components, pipes, directives), and core modules (e.g., services, guards, and interceptors). This separation allowed each module to be developed, tested, and maintained independently, reducing the risk of conflicts and making it easier to manage the codebase as the project expanded.

For example, the "Board" module included components and services specifically related to board management, while shared components like headers and footers were placed in the "Shared" module, making them reusable across different parts of the application.

Service-Based State Management:
For the majority of the application, especially where the complexity was moderate, we opted for a **service-based state management** approach rather than using NgRx. This decision was based on the specific needs of the project and the balance between complexity and maintainability. In this approach, Angular services were used to manage state and share data between components. This allowed us to keep the architecture simpler and more straightforward for areas of the application that didn't require the full power of a Redux-like state management library.

When NgRx Was Used:
However, in areas where more complex state management was required, particularly in handling real-time updates and ensuring consistency across multiple users interacting with shared data, we did utilize **NgRx**. This was primarily for features like real-time collaboration, where the benefits of a centralized, predictable state management system outweighed the added complexity.

Responsive Design: To ensure that the application was accessible on various devices, we implemented responsive design techniques using SCSS Flexbox and custom media queries. Angular's Flex Layout was used to create fluid grid layouts that automatically adjusted based on screen size. For instance, on mobile devices, the Trello boards would shift to a single-column layout, while on desktops, they would display multiple columns side by side. This approach

ensured that the application provided a consistent and user-friendly experience, regardless of the device being used.

Server-Side Rendering (SSR): We implemented **Angular Universal** to enable server-side rendering (SSR), which significantly improved the application's SEO and initial load times. With SSR, the application's pages were pre-rendered on the server before being sent to the client, allowing search engines to crawl and index fully rendered HTML content. This not only improved the visibility and ranking of the application in search engine results but also provided a faster user experience, especially for users on slower networks. For example, critical pages like the dashboard were rendered on the server, reducing the time it took for users to see content when they navigated to those pages.

Client-Side Security: Client-side security was a top priority, particularly given the sensitive nature of the data managed by Trello. We used Angular's **DomSanitizer** service to sanitize potentially dangerous HTML content, preventing XSS (Cross-Site Scripting) attacks. Additionally, HTTP interceptors were implemented to attach JWT tokens to outgoing requests, ensuring that all API calls were authenticated. The interceptors also handled global error responses, such as unauthorized access attempts, by redirecting users to the login page or displaying appropriate error messages.

Moreover, we implemented Content Security Policy (CSP) headers to further protect against XSS and data injection attacks by restricting the sources from which content could be loaded. For instance, we allowed scripts to be loaded only from trusted domains and blocked inline script execution, adding an extra layer of security to the client-side.

FrontEnd Development:

On the frontend, I implemented lazy loading by modularizing the application. For instance, instead of loading all components at once, we configured the Angular router to load modules like the "Reports" section only when the user navigated to it. This was achieved by using Angular's loadChildren property in the routing configuration, which deferred the loading of specific modules until they were actually needed. We combined this with tree shaking to eliminate unused code and code splitting to ensure that only necessary components were loaded, further optimizing performance.

For form handling and validation, I used Angular's reactive forms to build complex forms with dynamic validations. For example, in the user profile section, I created a form where users could update their personal information. This form included custom validators for fields like email and password. For the password field, I implemented a validator using Angular's ValidatorFn interface that enforced criteria like the inclusion of uppercase and lowercase letters, numbers, and special characters. This ensured that all user inputs were validated before submission, maintaining data integrity.

Backend Development:

On the backend, we transitioned from a monolithic architecture to a microservices architecture to improve scalability and performance. Each service was designed to handle specific business logic independently, such as user management, board management, and notification services. This allowed us to scale individual services based on demand without impacting the entire system. For instance, the user service was scaled independently during peak times when user authentication and profile management requests spiked.

For security, we implemented JWT-based authentication using Node.js. JWT tokens were generated upon user login and included encoded user information and permissions. We utilized Express.js middleware to verify these tokens on each request, ensuring that only authenticated users could access specific routes. Additionally, we implemented role-based access controls by decoding the JWT token to check user roles, allowing or restricting access to certain API endpoints based on the user's role.

We also integrated MongoDB with Mongoose for data storage. I optimized data handling by creating compound indexes on frequently queried fields, which significantly improved query performance. We used aggregation pipelines to process and transform data directly within MongoDB, reducing the amount of data transferred between the database and application servers. For large datasets, we implemented sharding and replication to ensure data was distributed and replicated across multiple nodes, enhancing both performance and data reliability.

Documentation:

Throughout the project, I maintained comprehensive documentation using Confluence. This included API specifications, architectural diagrams, and user stories. I made sure that the documentation was continuously updated to reflect changes in the application's architecture, making it easier for team members to onboard and understand the system. Detailed API documentation was also provided using tools like Swagger, ensuring that frontend developers and third-party integrators could easily consume the APIs we developed.

Testing and Quality Assurance:

Testing was a crucial part of our process to ensure the quality and security of the enhancements. On the frontend, I wrote unit tests for Angular components using Jasmine and Karma, covering various scenarios to ensure the components behaved as expected. For the backend, we used Mocha and Chai to test API endpoints, ensuring they handled different types of requests correctly. Penetration testing was also conducted to identify potential security vulnerabilities, particularly in our JWT-based authentication and encryption methods. This testing was crucial in ensuring that our security implementations could withstand potential threats.

Deployment and Post-Release:

As we approached deployment, we containerized the application using Docker, allowing us to deploy updates consistently across different environments. This involved creating Docker images for each microservice, ensuring that each service had the necessary dependencies and configuration to run independently. We hosted the application on AWS, utilizing EC2 instances for scalable processing power and S3 for storing static assets like images and JavaScript bundles. The deployment process involved using AWS Lambda for serverless operations, particularly for tasks like image processing, which didn't require a persistent server.

We employed a staged rollout strategy, where updates were initially released to 10% of our user base. This allowed us to monitor performance and gather feedback before expanding the rollout to all users. Continuous monitoring with tools like AWS CloudWatch and New Relic was essential in detecting any issues post-release. For example, we detected a spike in response times during peak hours and quickly identified a bottleneck in one of our microservices. We resolved this by optimizing the service's database queries, which involved restructuring the queries and adding additional indexes.

Conclusion:

Overall, the project was a success. We met all the objectives within the planned timeline, resulting in faster load times, a more intuitive user interface, and enhanced security. These improvements led to increased user satisfaction and adoption, particularly in sectors like finance and healthcare, where data security and efficient workflows are critical.