Design Goals

Security: Our system employs JWT with randomized seeds and time limits for enhanced security. Tokens contain unique seeds, mitigate token-based attacks, and expire after a set period, reducing the risk of misuse.

Scalability: Adopting a layered approach, we prioritize code readability and modularity to facilitate scalability. Clear separation of concerns and concise, modular functions enhance flexibility, enabling seamless integration of new features.

User-Friendly UI: Leveraging Flutter's responsive and visually appealing UI capabilities, we ensure a user-friendly interface. Sleek, intuitive designs adapt seamlessly to various devices, enhancing usability and navigation. By prioritizing user satisfaction and engagement, we create an enjoyable experience for all stakeholders within the school system.

Flexibility: Leveraging Flutter's cross-platform nature, we deploy our software as native desktop or mobile applications. This approach streamlines development and offers versatility, reaching users across diverse devices and operating systems.

Reliability: Our system ensures consistent performance and uptime through robust architecture and industry-standard technologies. Node.js for server-side logic and MongoDB for data storage offer stability and scalability. Adopting an MVC approach enhances code maintainability and reduces the risk of failures. Deploying on a Linux server, JWT token authentication, and proactive monitoring prioritize uninterrupted access to our school system.

Connectors

Node.js Express: We chose Node.js with Express for our backend server due to its non-blocking, event-driven architecture, which allows for high concurrency and scalability. Node.js is well-suited for our real-time applications, such as chat and handling asynchronous requests, which is the mechanism by which we built our app.

Flutter HTTP Package: For communication between our Flutter frontend and backend server, we utilize the HTTP package provided by the Flutter framework. This package enables us to make HTTP requests to our Node.js server endpoints.

Socket.IO: In scenarios requiring real-time bidirectional communication, such as the chat part in our application and the notifications, we integrate Socket.IO with our Node.js server.

Mongoose ODM for MongoDB: To streamline data modeling and interaction with MongoDB in our Node.js backend, we employ Mongoose, an Object-Document Mapping (ODM) library. Our models are written in Node.js, and Mongoose maps our models to MongoDB, providing a useful abstraction.

Architectural Style

Distributed Architecture: With the use of Node.js for server-side logic and MongoDB for data storage and with Flutter build completely separate from each of them, our systems have the characteristics of a distributed system architecture to some extent. It is chosen since this way we can replace any of these elements without changing the others. For example, if we choose to move on from Flutter in the future, we will not have to change any part of the backend or the database.

Layered Architecture: Our system's emphasis on modularity, code readability, and separation of concerns indicates the use of layered architecture. It adopted an extended version of MVC, with two extra layers of Routers and Middlewares, along with traditional Models, Views, and Controllers. This ensures the separation of logic, view, and database objects, making writing and reading code easy.