

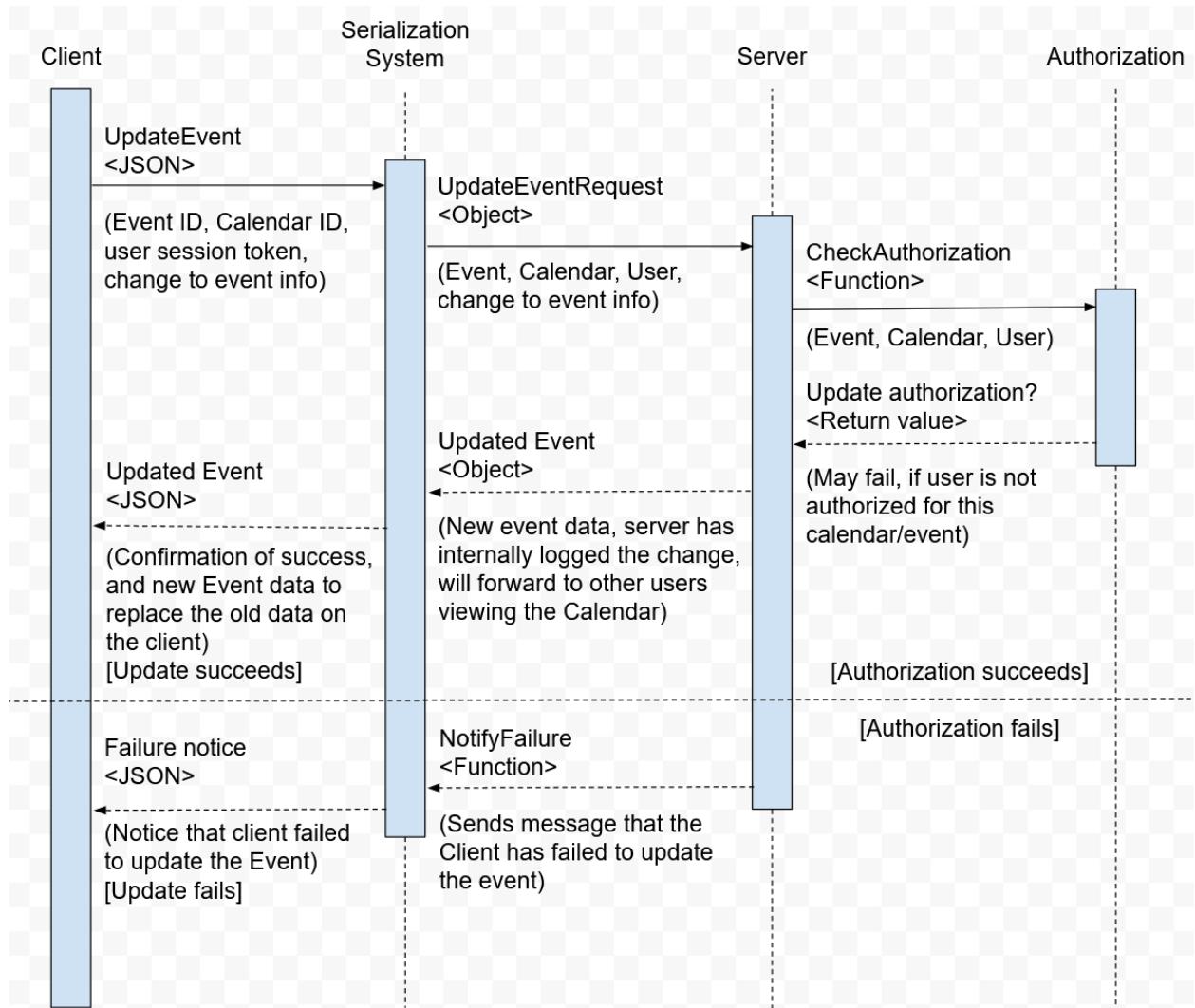
## Process Model

[The diagram for this part is quite large and does not properly fit in this document. It is instead included as an image in the root directory named "CS3354 Group 15 Deliverable 2 Process Model Diagram.png".]

This workflow captures the essence of our system and how we structured it as it shows all the important possible actions the user is able to conduct while on this website. This highlights the main purpose of this system, for user interaction with the calendar to create, edit, and modify any events and work with other individuals if needed. This is seen especially when after logging/signing up, there is a decision node in the user's swim lane that has, "delete event", "edit event", "create event", "save calendar", "load calendar", "share", "view events", "next month", "previous month", and "exit". These are all the primary functions the user is able to do with the calendar as it goes with our function requirements. For instance, the user is able to create, edit, and delete events in requirement FR-1, or save and load calendars from FR-6. Thus, showing the essence of how our calendar system will work when a user uses it whether it's their first time or not.

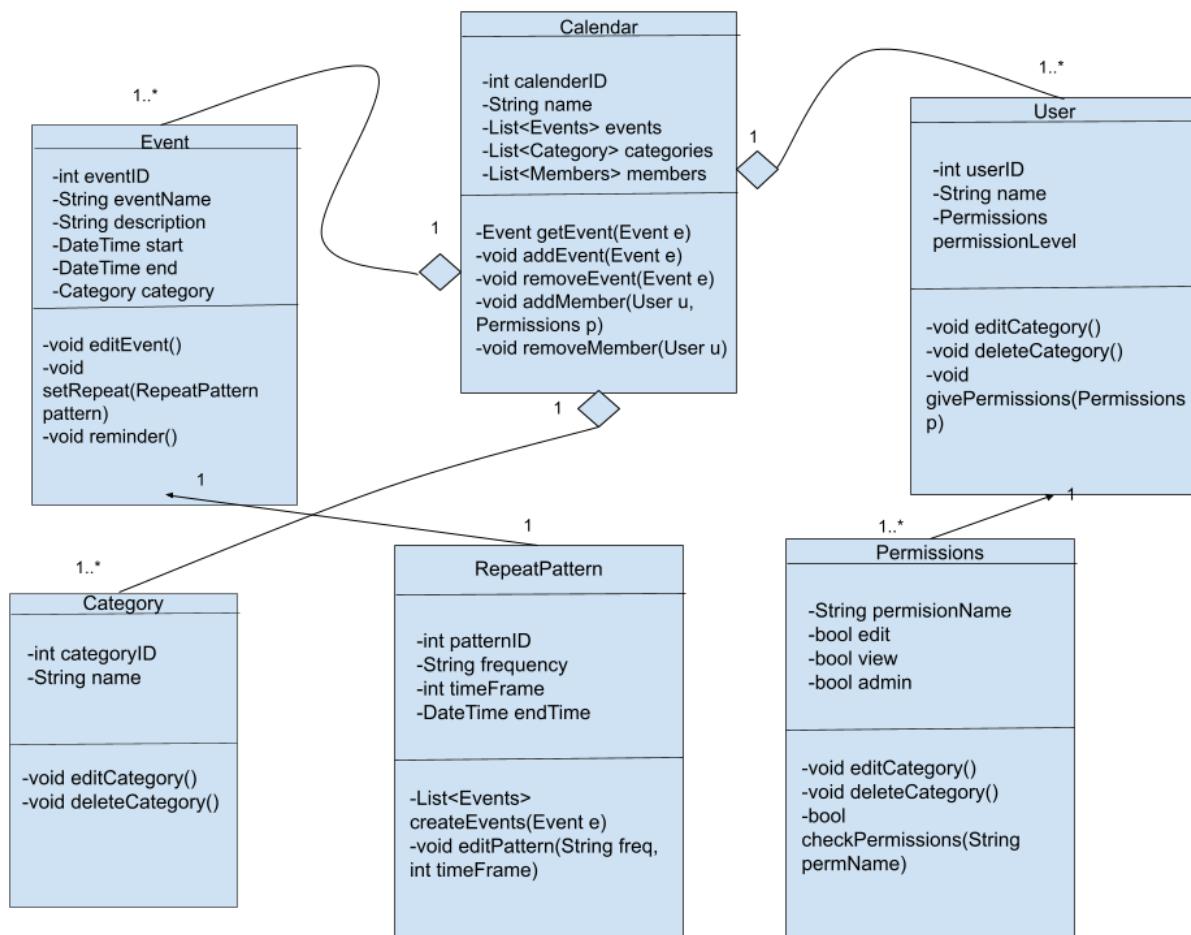
For this activity diagram, we structured it using swim lanes, more specifically 3. These three swim lanes are "User", "System", and "Storage". The user is the end user using the system, such as groups or individuals who want to stay organized and collaborate with others. The system is the whole system itself (the website). It displays any new pages, popups, and even communicates between the user and the storage. It handles the logic for every functional requirement and handles any request from the user. The storage is where the calendars are saved and loaded from. This helps the users collaborate on different calendars while still being able to save their own and not lose any important events, or progress. As a result, this diagram clearly defines our system as it maintains an interactive loop of activities that the user is able to conduct while on the website.

## Behavioral Model



This diagram models a basic interaction in the system, that of a client trying to update an event on a calendar. It models two main systems: The client and the server, and two subsystems of the server: The serialization and authorization systems. The client is the basic point of interaction for the user. The server manages storing and updating calendar/event information, and synchronizing it between users. Serialization (for translating server objects to/from JSON so it can talk to the client website) and authorization (checking user permissions and logging users in/out) are both sufficiently complex to be worth noting as independent systems in the model.

# Structural Model

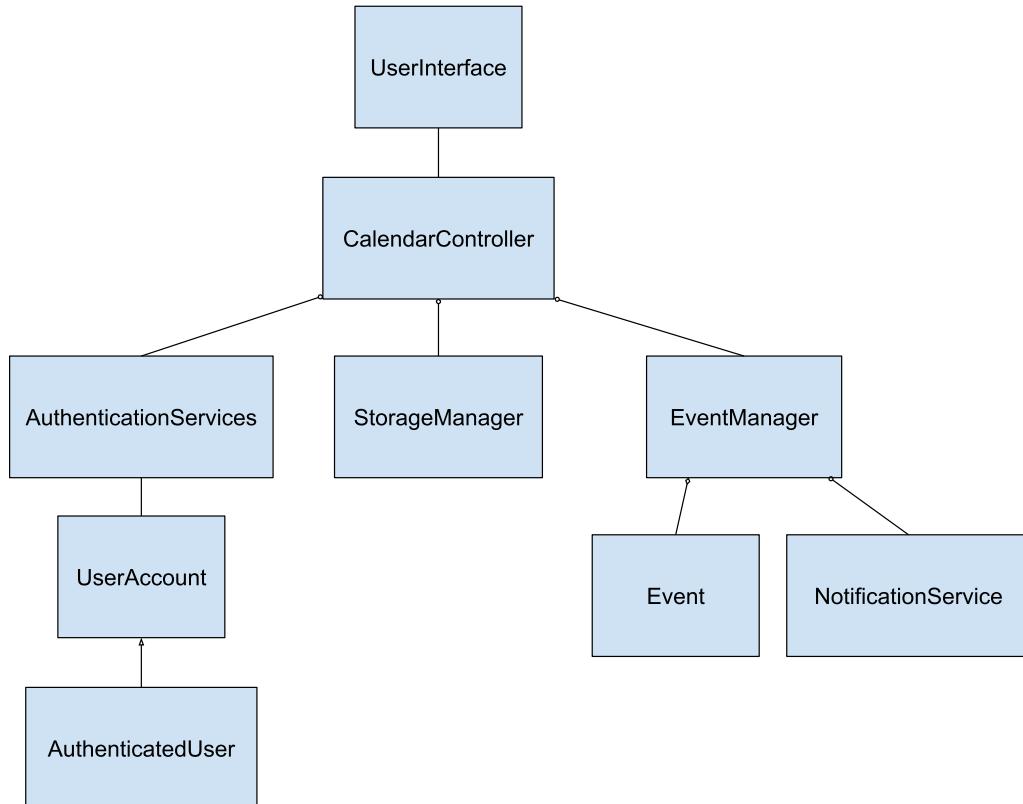


The core responsibilities of the calendar system were divided into 6 different classes to provide clarity throughout the software engineering process. The **Calendar** class is the main class, and it is composed of the **Event**, **User**, and **Category** classes. The **Event** class manages event details and is linked to the **RepeatPattern** class, which helps to handle recurring events in the system. The **User** class is critical for collaboration between calendars, and it is connected to the **Permissions** class, which ensures users have the proper authorization when using a calendar. Overall, this distribution of responsibilities will be used as the framework for our calendar system.

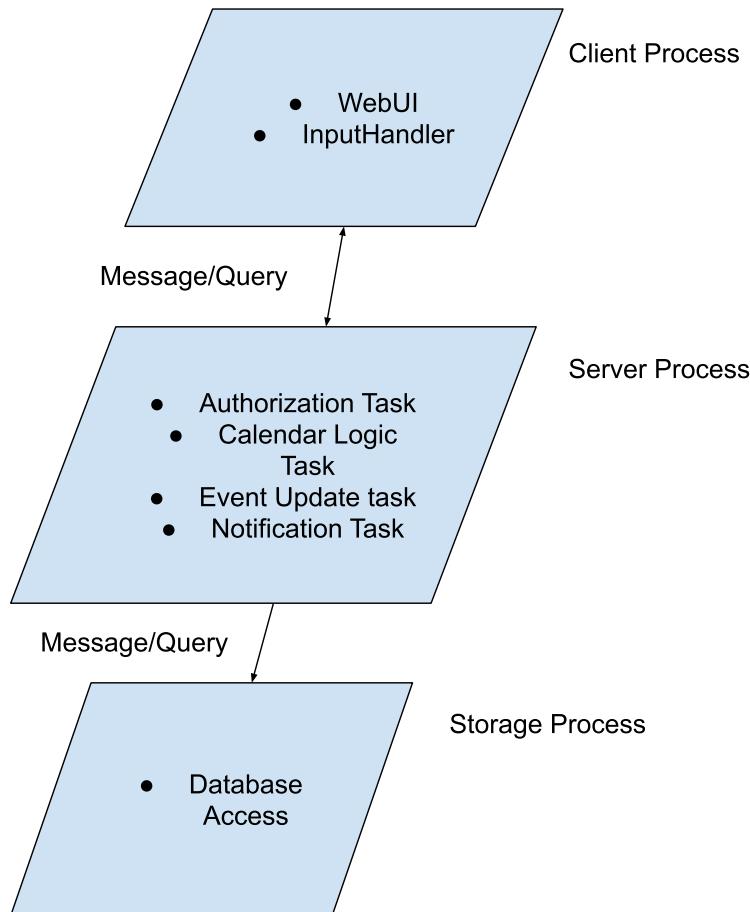
# Architectural Design Decisions

1. Non-Functional Requirement: Performance (The system shall respond to the users interaction within 3 seconds)
  - a. Design Decision: Implement server caching for the most frequently accessed data.
  - b. Justification: Without caching the user would need to repeatedly query the database for all data, no matter how frequently that data is needed. By having the common data come from memory rather than the database, we can more easily reach the 3-second response requirement, regardless of high user load.
2. Non-Functional Requirement: Security (The system shall store user data securely)
  - a. Design Decision: Implement end-to-end encryption for data in transit using HTTPS.
  - b. Justification: This ensures that all sensitive user data (login information, personal info, etc.) cannot be intercepted by third parties. Encryption allows the user the comfort of knowing that their data is safe in the hands of our software while also complying with data protection standards.

# High-Level Architecture



The logical view shows us the main classes and their interaction with each other. It can be seen that the User Interface interacts with the Calendar controller to do the logic and divide the tasks into the specialized classes. The three main specialized classes being the storage, which holds important data, the authentication service that deals with authentication and user safety, and finally the event manager that deals with the event logic.



The Process view model shows the interaction between major processes and tasks during runtime. The client process manages the users interactions, and communicates with the server processes to run multiple concurrent tasks at runtime. The storage process maintains the data and gives the corresponding data to the server processes when needed.

## Use of Architectural Patterns

Our project is primarily using a client-server model. It's a very textbook example of one: We have many clients (users on the website that displays the calendar and allows it to be modified), which must all be synchronized and have their data stored in some place when they're not using the app (the server).

A benefit is that this naturally compartmentalizes the main parts of the app: The display for the client, and the logic for the server. A downside is that the server must exist somewhere and the clients must be able to access it over some kind of connection, and the app doesn't work if either of those is not the case.