CoSpace
Architecture Notebook

1. Purpose

Architectural design is a critical stage of the software design process because of its variety of benefits. Software architecture represents an abstract view of the whole system. This means that it does not include any detail about the implementation of the components or the way of representing the data, it is just concentrating on the behavior of the system components as a black box. Owing to this, the stakeholders will have a mutual understanding of the system, and they can communicate with each other easily. Moreover, designing the architecture at an early stage of the system development requires analysis, and this can be useful for deciding that the system will meet the critical requirements such as performance, reliability and maintainability.

The purpose of this document is to show the various aspects of the architectural design of the *CoSpace.* The rest of the documents explain architectural patterns that will be used in the system, non-functional requirements because of the close relationship between the architecture, such as performance, security, and maintainability etc. Also the assumptions, dependencies, constraint and justifications will be examined.

2. Architectural goals and philosophy

For evaluating whether the system will be able to exhibit its desired functionality by its architecture, there is a need for setting architectural goals. Architectural goals might query whether the variety of different requirements of the system will be fulfilled or not, if it will be easy to adapt the system to legacy systems or the architecture comes with different issues such as performance or modifiability. The first obvious goal of developing the architecture is that it helps the team members understand the motivations behind architectural decisions so those decisions can be robustly implemented. The other architectural goals for the *CoSpace* system are as follows:

- The system must be maintabile and because of this, the system architecture should be designed using fine-grain. And make the system to be easy to modify, the elements of the system should have concrete responsibilities such that changes to the system do not have far-reaching consequences.
- The architecture should make possible the separation of presentation and interaction from the system data.
- The architecture should support the incremental development of the system, and also it should be portable. The operating system or browser that the system is running in should not be an issue, because the system should be independent of any of these.
- The system needs to work without any trouble under the unusual conditions. (e.g., when there are too many active users in the system, short-term internet connection problems etc.)

3. Assumptions and dependencies

- Each team member available at least 8 hours per week
- Almost each team member has an experience in a project of this size
- No budget limitations
- No dependencies on legacy interfaces

4. Architecturally significant requirements

By definition of Architecturally Significant Requirements, as they are a subset of all of the functional and non-functional requirements, in this section, we have chosen the requirements that have a wide effect, target trade-off points, are strict and assumption breaking as architecturally significant.

The system should respond to the requests as quickly as possible. At Software Requirements Specification (SRS), our provided response time is a maximum of 400 ms, excluding the internet response time of the user[1].

All of the passwords that are provided by users to the system should not be stored as plain text. Instead, they should be encrypted using known encryption methods with salt. Here, we have chosen the SHA256 algorithm[2].

Backend is handled by Spring. Hence, the user interactions are handled by Spring Security, and requests are handled by a RESTful web service which is created using Spring Boot[3]. A system should provide a good percent of availability. As stated at SRS, the system is expected to provide at least 99% uptime for availability[4]. Furthermore, if the system is down, it should be recoverable as quickly as possible.

The system should provide scalability. Scalability is the measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands[5].

5. Decisions, constraints, and justifications

In our project, the decisions and constraints are selected carefully; and in this section, you can see our decisions and constraints with their justifications and restrictions.

We have chosen the response time for the search functionality as 400 ms, real-time messaging as 250 ms, and other opening-closing activities as 250 ms. These numbers are coming from Doherty Threshold. As the Doherty Threshold dictates, an immediate interaction between user and computer will heighten the experience and result in a much simpler and more enjoyable experience and it says that the user experience turns from painful to addictive after the system feedback time drops below 400ms. Hence, a response time should not exceed 400 milliseconds.

In this system, user logins and logouts are happening but without any authentication mechanism, it will not be possible. Here, we should use a mechanism that provides the authentication of the users as securely as possible, and Spring used for our system's backend, Spring Security is the most viable option for our case. Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

Storing user passwords is a critical component for any web application. When you store a user’s password, you must ensure that you have it secured in such a way that if your data is compromised, you don’t expose your user’s password. There have been many high-profile cases of websites and web applications that have had their user details compromised. This is made even worse when the developers of the website have not securely stored the user’s password. If you are developing a website or web application that needs to store user data, it is incredibly important that you take the correct precautions should your data become exposed. When you store a password in a database, you never want to store it in plain text. Storing a password in plain text would mean that anyone who looked through the database would be able to just read the user’s passwords. If you do not take the right precautions when storing passwords, you could expose your user passwords to an attacker. "Hashing" passwords is the common approach to storing passwords securely. Hashes are impossible to convert back into plain text, but you don’t need to convert them back to break them. Once you know that a certain string converts to a certain hash, you know that any instance of that hash represents that string. Hashing a password is good because it is quick and it is easy to store. Instead of storing the user’s password as plain text, which is open for anyone to read, it is stored as a hash which is impossible

[1] https://www.sciencedirect.com/topics/computer-science/system-response-time
[2] https://en.wikipedia.org/wiki/SHA-2
[3] https://spring.io/projects/spring-security
[4] https://en.wikipedia.org/wiki/Availability_(system)
[5] https://en.wikipedia.org/wiki/Scalability

for a human to read. Here, we choose the SHA256 algorithm to do hashing. SHA256 algorithm generates an almost-unique, fixed-size 256-bit (32-byte) hash. Furthermore, you also need to salt the passwords because a hacker can eliminate the advantages of just hashing the passwords by some techniques like the Rainbow Table attack. So, we have to salt these. Salting is where you add an extra bit of data to the password before you hash it, something like you would append every password with a string before hashing. This would mean the string before hashing would be longer and therefore harder to find a match.

In our backend, Spring Boot Application architecture is used to eliminate redundancy over implementations. It allows us to build an application with minimal or zero configurations. Hence, the architecture provides quick implementations. Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it. There are four layers in Spring Boot are as follows: Presentation Layer, Business Layer, Persistence Layer, Database Layer. The presentation layer handles the HTTP requests, translates the JSON parameter to the object, and authenticates the request, and transfers it to the business layer. The business layer handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorization and validation. The persistence layer contains all the storage logic and translates business objects from and to database rows. In the database layer, CRUD operations are performed.

We want our system as available to the users and stakeholders as possible. It is a metric that measures the probability that a system is not failed or undergoing a repair action when it needs to be used. For high availability, we choose 99% uptime as the most optimal for our case. We cannot choose any higher, because of higher cost.

6. Architectural Mechanisms

MVC Pattern

In our project, we have precisely used the MVC pattern. We have Controller, Model and View separated. All of the requests created by the user are coming first to the controller, even though we have separate classes for controllers for cases i.e. authorization, club, post, etc. in the same package. Controller passes the related information to the View or Model components, gets the results from these, and responds to the user. In that way, we have an architecture that is not cluttered.

Layered Architecture

As said in previous sections, we have used Spring Framework for our backend. Thus, it is natural that we also used layered architecture. Our View and Controller stays in the Presentation Layer. All of the other layers as Business Layer and Persistence Layer are in the Model. In Presentation Layer, the requests made by a user (it may be a login request, post request, create club request, logout request, etc.) are handled and converted for other layers (Model) if needed. In Business Layer, the business logic is handled and all of the authorization and validation is performed. In the Persistence Layer, all of the storage logic is stored and handled. Also, CRUD operations are performed there. In that way, the code is going to be easier to maintain, compact, and easily movable.

7. Key abstractions

We abstract the components which we have with a layered pattern. Presentation layer, business layer, persistence layer, and database layer are main objects of the layered pattern. We aimed to satisfy high cohesion and low coupling with modularity of our project. We divided into parts our classes instead of singular class structure.

In the presentation layer, we have controller classes handle the requests and forward them to the next layer if the requests have correct structure. Only simple controls are done in this layer. We aimed to make an abstraction between this layer and business layer through the controller classes. In the business layer, we have service classes which don’t have any access to databases without the next layer. This access limit creates an abstraction between business layer and persistence layer through the repository classes. The repository classes satisfy connection between business layer and the database. We used interfaces as repositories using JpaRepository interface.

8. Layers or architectural framework

We used Spring Framework for implementation of layered patterns. Spring Framework includes 4 layers which are communicated with only upper and lower layers for abstraction.

Presentation Layer

We have controller classes to handle requests and forward them to the next layer, business layer. This layer gets requests, check them according to request validation, and forward them to related service classes.

Business Layer

We have service classes to handle necessary operations for the requests coming from the presentation layer and forward them to the persistence layer. Also, the service classes communicate with repository classes in the persistence layer.

Persistence Layer

We have repository classes to make data operations without direct connection between the project and database. Also, repository classes communicate with the business layer and database. We implemented SpringDataJpa interface to use essential database CRUD functions such as find, save, findAll, findBy, etc. Also, we have PostgreSQL Driver which gives database connection interface us.

MVC Architecture

Model

Our database is PostgreSQL which is deployed in AWS servers. We selected this database, because we have experience with PostgreSQL. Therefore, this selection makes the implementation duration sorter because of our previous knowledge.

View

Our front-end architecture is based on ReactJS because of its benefits such as modularity, efficiency, etc. Also, we used Axios to establish a connection between view and controller layer.

Controller

We used Spring RestController annotation to build a Rest API infrastructure. Also, we have separate security configurations for each controller. We created GET, POST, PUT, and DELETE request handlers with the annotations which are provided by the Spring Framework.

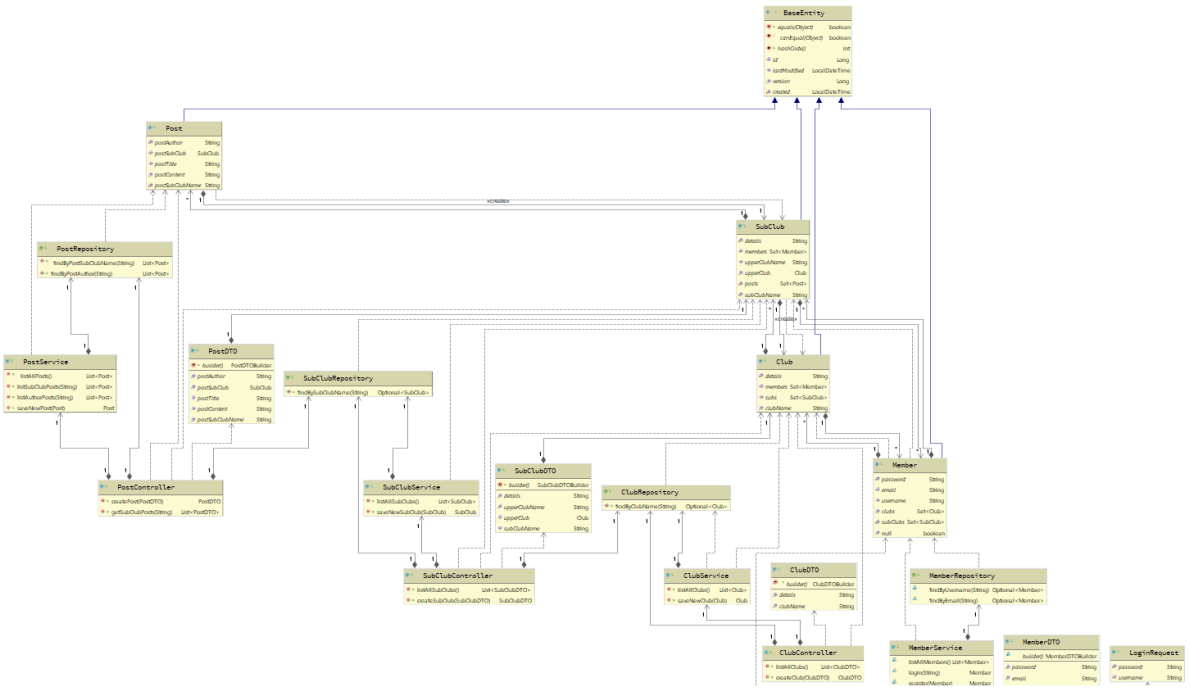
9. Architectural views

Recommended views

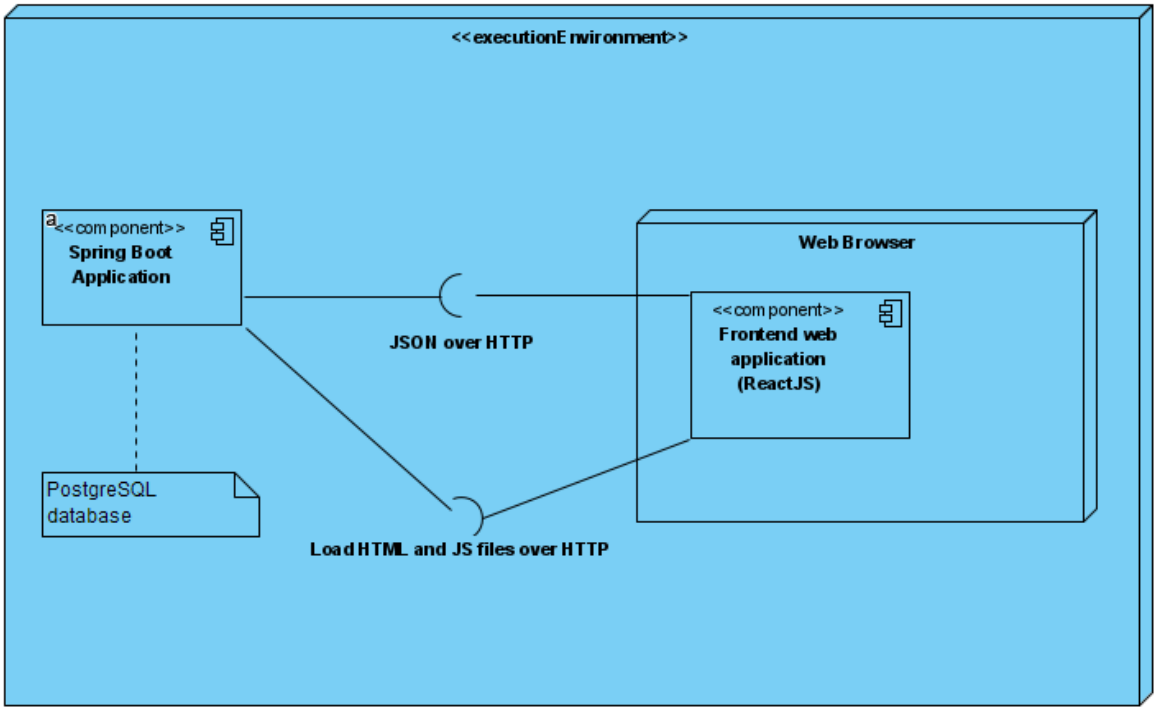
- **Logical:** Describes the structure and behavior of architecturally significant portions of the system. This might include the package structure, critical interfaces, important classes and subsystems, and the relationships between these elements. It also includes physical and logical views of persistent data, if persistence will be built into the system. This is a documented subset of the design.
- **Operational:** Describes the physical nodes of the system and the processes, threads, and components that run on those physical nodes. This view isn't necessary if the system runs in a single process and thread.
- **Use case:** A list or diagram of the use cases that contain architecturally significant requirements.

Class Diagram

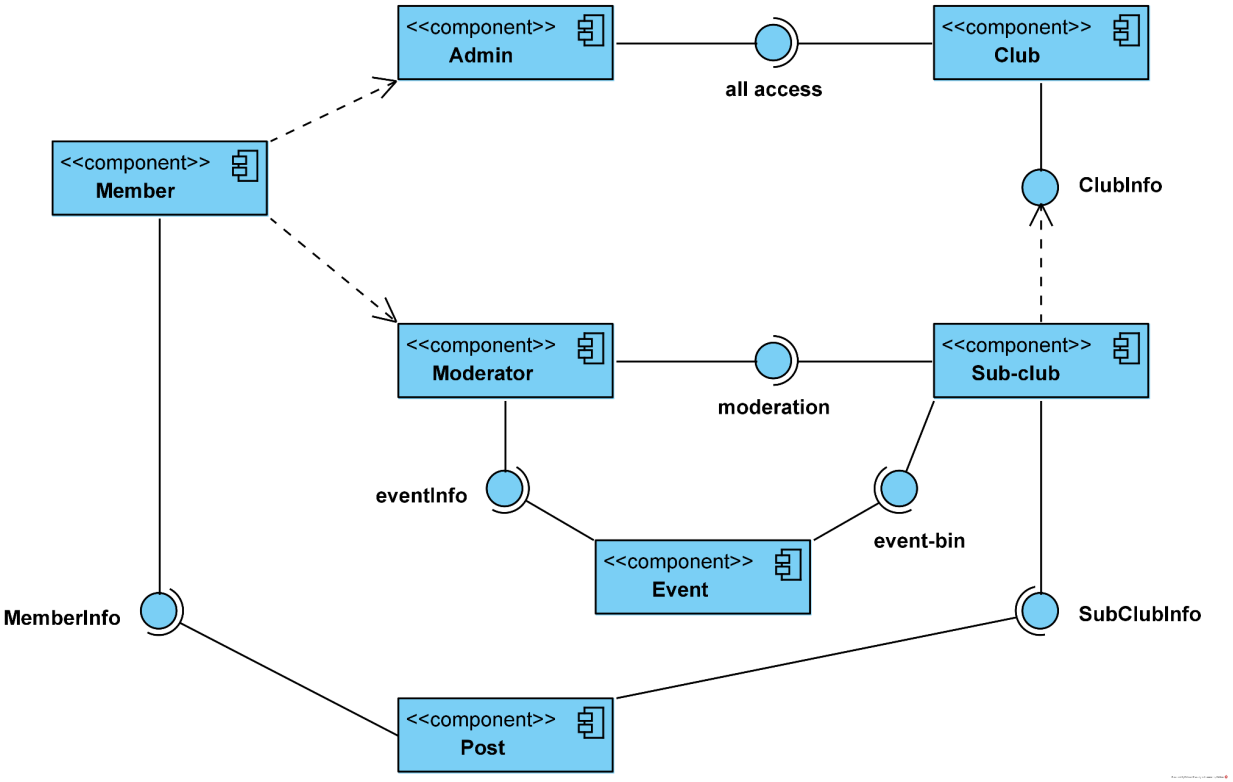
Some classes are hidden because of limited area in this document. Also, the methods of the classes are hidden, implementation details will be shown in the DEL4 document.



Deployment Diagram

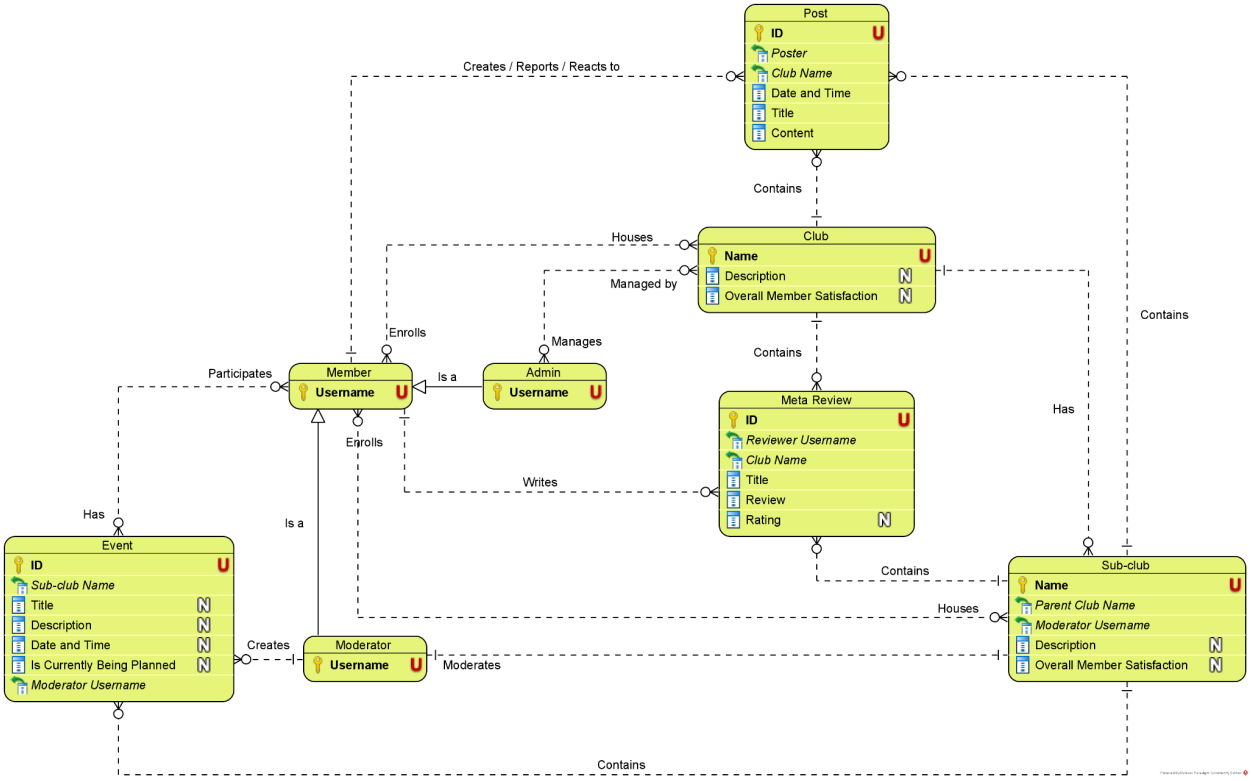


Component Diagram

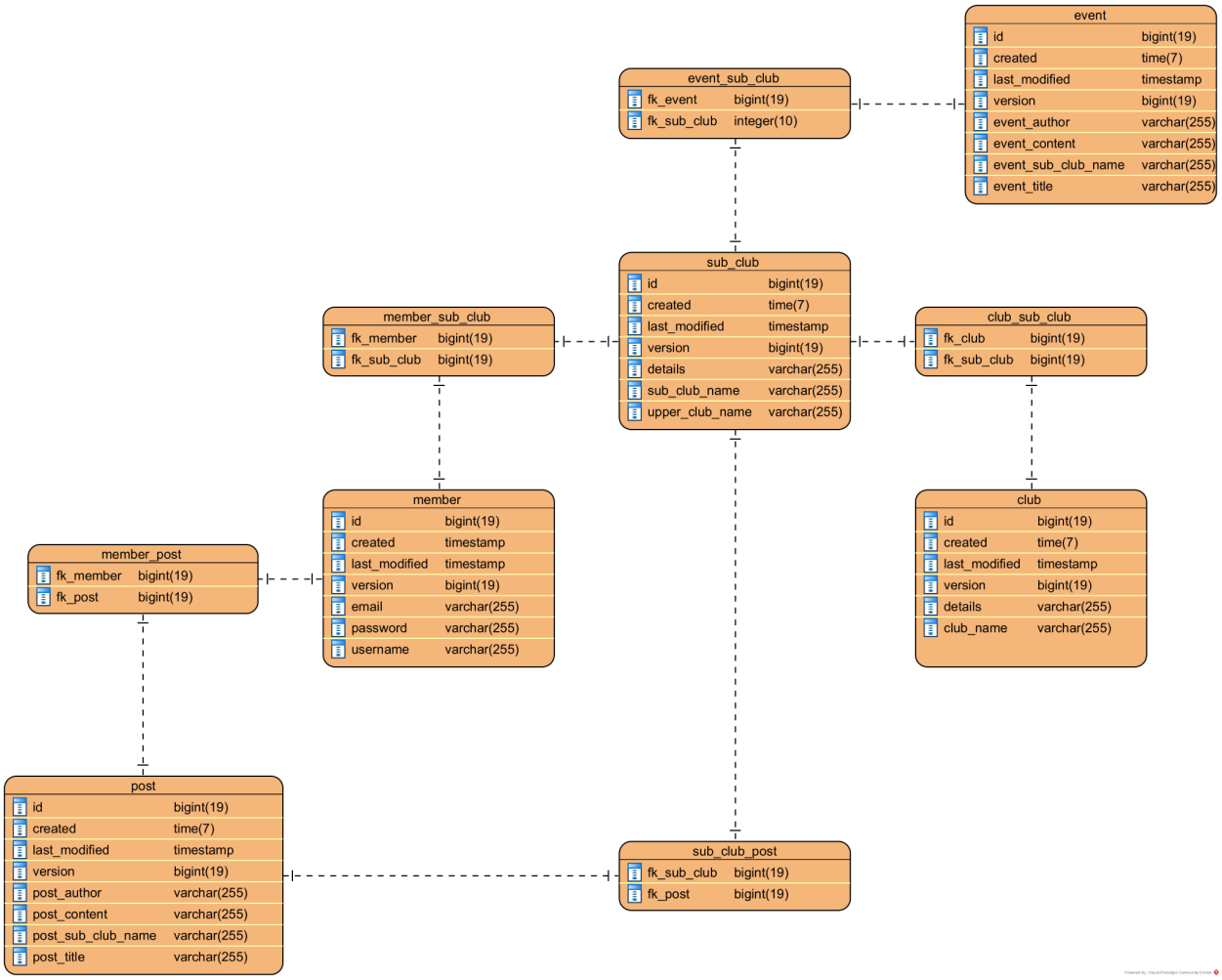


Entity-Relationship Diagram

Conceptual



Logical


event
id bigint(19)
created time(7)
last_modified timestamp
version bigint(19)
event_author varchar(255)
event_content varchar(255)
event_sub_club_name varchar(255)
event_title varchar(255)
event_sub_club
fk_event bigint(19)
fk_sub_club integer(10)
sub_club
id bigint(19)
created time(7)
last_modified timestamp
version bigint(19)
details varchar(255)
sub_club_name varchar(255)
upper_club_name varchar(255)
member_sub_club
fk_member bigint(19)
fk_sub_club bigint(19)
club_sub_club
fk_club bigint(19)
fk_sub_club bigint(19)
member
id bigint(19)
created timestamp
last_modified timestamp
version bigint(19)
email varchar(255)
password varchar(255)
username varchar(255)
club
id bigint(19)
created time(7)
last_modified timestamp
version bigint(19)
details varchar(255)
club_name varchar(255)
member_post
fk_member bigint(19)
fk_post bigint(19)
post
id bigint(19)
created time(7)
last_modified timestamp
version bigint(19)
post_author varchar(255)
post_content varchar(255)
post_sub_club_name varchar(255)
post_title varchar(255)
sub_club_post
fk_sub_club bigint(19)
fk_post bigint(19)

Package Diagram