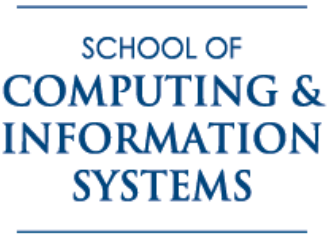
Architecture Document

SWEN90007

Music Events System

Team: GGBond

In Charge of
Linjing Bi (1369370)
Jie Zhou (1442449)
Baorui Chen (1320469)
Liuming Teng (1292608)



Revision History

Date	Version	Description	Author
04/09/2023	01.00-D01	Initial draft	LiumingTeng JieZhou BaoruiChen LInjingBi
05/09/2023	01.00-D02	Added Section 3 details to the document	LiumingTeng JieZhou BaoruiChen LInjingBi
08/09/2018	01.00-D03	Added Section 4 and 5 details to the document	LiumingTeng JieZhou BaoruiChen LInjingBi
12/09/2023	01.00	First version of the document	LiumingTeng JieZhou BaoruiChen LInjingBi
13/09/2023	02.00-D01	Adjust document to SWEN90007 subject	LiumingTeng JieZhou BaoruiChen LInjingBi
17/09/2023	02.00-D02	Simplify the document to SWEN90007 project	LiumingTeng JieZhou BaoruiChen LInjingBi
18/09/2023	02.00	Final review and improvements on the document	LiumingTeng JieZhou BaoruiChen LInjingBi

Contents

1. Introduction

This document specifies the system's architecture Music Events System, describing its main standards, module, components, *frameworks* and integrations.

1.1 Proposal

The purpose of this document is to give, in a high-level overview, a technical solution to be followed, emphasizing the components and *frameworks* that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target Users

This document is aimed at the project team, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	Description
Component	Reusable and independent software element with a well-defined public interface, which encapsulates numerous functionalities and can be easily integrated with other components.
Module	Logical grouping of functionalities to facilitate the division and understanding of software.
Venue	A place where music events, concerts, or festivals are held. It can range from small clubs to large stadiums.
Venue Capacity	The maximum number of attendees or audience that a particular venue can accommodate for an event.
User Dashboard	A personalized page for registered users that displays their profile, booked events, wish list, and other user-specific details.
Booking Confirmation	A digital or electronic receipt provided to the user upon successful reservation of tickets for a music event.
Event Page	A dedicated page within the system that provides comprehensive details about a specific music event, including date, time, venue, ticket prices, and artist lineup.

1.4 Actors

Actor	Description
Administrator	Manage the system, all users and venues are controlled by the admin.
Event Planner	Users who manage the events' information.
Customer	Users who want to buy or cancel the tickets.

2. Architectural representation

The specification of the system's architecture ***Music Event System*** follows the *framework* "4+1" [1], which defines a set of views, as shown in Figure 1. Each of these views approaches aspects of architectural relevance from different perspectives:

- The **logical view** shows the significant elements of the project for the adopted architecture and the relationship between them. Between the main elements are modules, components, packages and the application's main classes;
- The **process view** shows the concurrency and synchronization aspects of the system, mapping the elements of the logical view to processes, *threads* and execution tasks;
- The **development view** focuses on aspects relating to the organization of the system's source code, architectural patterns used and orientations and the norms for the system's development;
- The **physical view** shows the hardware involved and the mapping of the software elements to the hardware elements in the system's environment.
- The **scenarios** show a subset of the architecturally significant use cases of the system.

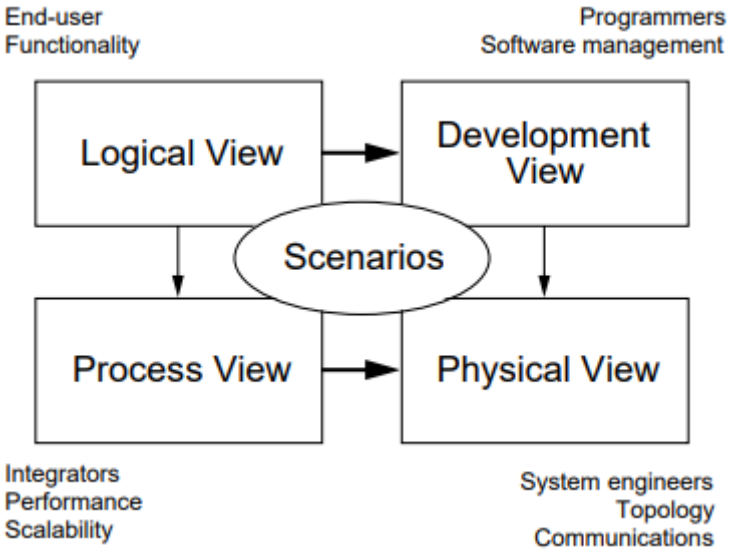


Figure 2.1 *Views of framework "4+1"*

source: Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE software*, *12*(6), 42-50.

In regards to our ***Music Event System***, we focus on its **logical view** and **development view**, as well as key usage **scenarios**. However, we will try to implement the process view to show the concurrency and synchronization aspects of the system in the coming assignments.

3. Architectural Objectives and Restrictions

The defined architecture's main objective is to make the system...

Objectives:

Portability: The system should be easily adaptable to different platforms and environments without requiring significant changes. This ensures that the system can cater to a diverse user base across various devices and operating systems.
Performance: The architecture aims to provide fast response times and efficient processing to ensure a smooth user experience. Performance optimizations are prioritized to handle high user traffic and data processing.
Scalability: As the user base grows, the system should be able to scale seamlessly. The architecture is designed to handle increased loads and traffic without compromising on performance.
Tolerance to Imperfections: The system should be resilient and capable of handling unexpected errors or issues without crashing. It should provide meaningful error messages and have mechanisms in place for quick recovery.
Integration: Given the potential need to integrate with other systems or third-party services in the future, the architecture is designed with flexibility in mind to allow for easy integrations.

Restrictions:

Memory Limitation: The system might be deployed in environments with limited memory, requiring efficient memory management and optimization techniques.
Bandwidth Limitation: The system should be optimized for environments with limited bandwidth, ensuring data is transferred efficiently and unnecessary data transfers are minimized.
Development Team Profile: The current development team has expertise in specific technologies and languages. The architecture should align with this expertise to ensure efficient development and maintenance.
Production Environment: The system will be deployed in specific server environments, which might have their own set of restrictions. The architecture should be compatible with these environments.
Integration with Legacy Systems: If the system needs to integrate with older, legacy systems, it might face challenges due to outdated technologies or protocols. The architecture should be flexible enough to handle such integrations.

Impacting Requirements:

Several non-functional requirements have a direct impact on the definition of the architecture:

- **Reliability:** The system should have a high uptime and be available to users whenever they need it.
- **Security:** User data should be protected, and all transactions should be secure.
- **Maintainability:** The system should be easy to update and maintain, allowing for the addition of new features or fixing of bugs without major overhauls.
- **Usability:** The user interface should be intuitive and user-friendly, ensuring users can easily navigate and use the system.

3.1 Requirements of Architectural Relevance

This section lists the requirements that have impact on the system's architecture and the treatment given to each of them.

Requirement	Impact	Treatment
The system will have multiple types of users.	Requires a robust authentication and authorization mechanism.	Utilizes AuthenticationClass and AuthorizationClass to manage different types of users and permissions. Role-based access control is implemented and stored in sessions.
The system should have reusable components to reduce development complexity.	Impacts the overall design pattern and architecture of the system.	Adopts the MVC (Model-View-Controller) model, which is modular and makes it convenient for reusing similar functionalities.
The system should be easily maintainable and upgradable.	Calls for a modular and well-documented architecture.	Utilizes Docker for containerization. Designed to be modular and well-documented. Each servlet and class has specific responsibilities, making it easier to update or replace individual components.
The system needs high availability and reliability.	Requires a robust error-handling and exception management mechanism.	Utilizes the CustomExceptionFilter class to catch exceptions and convert them into user-friendly error responses.
The system needs fast responsiveness.	Requires optimization of database queries and data loading.	Employs Lazy Loading and Unit of Work patterns to optimize database interactions.

The system needs to support multiple languages.	Requires a flexible internationalization and localization mechanism.	Utilizes Java's internationalization support, managing text in different languages through Resource Bundles.
The system needs to support multi-device access.	Requires a responsive front-end design.	Utilizes Bootstrap or other responsive frameworks to ensure the front-end design adapts to various screen sizes.
The system needs to support high concurrency from a large number of users.	Requires a high-performance backend architecture and database design.	Employs load balancing and database sharding to enhance the system's ability to handle concurrent access.

4. Logical View

Layer 1 : Presentation

This layer is responsible for handling user interactions with the system and providing the user interface and user experience.

- Controllers Component: Manages user requests and responses, routes requests to the appropriate services, and sends responses back to the user. They serve as the interface between the user interface and the backend services.
- Web Resources Component: Includes frontend resources such as CSS, JavaScript, images, etc., for rendering the web interface and enhancing the user experience. It is responsible for front-end interface rendering and interaction.

Layer 2: Service

This layer contains the core business logic, handling user requests, executing business functions, and coordinating different services and components.

- Service Component: Contains the primary business logic of the application, handling user requests, order processing, concert management, etc.
- User Authentication and Authorization Component: Manages user authentication and authorization, ensuring that only authorized users can access resources and functions.

Layer 3: Data Access

This layer deals with interactions with data storage, including database connections, data retrieval, and updates.

- Datasource Component: Manages connections and interactions with the database, responsible for communication with the data source.
- Model Component: Represents the main data structures and entities of the application, used for mapping data to application objects.
- Data Transfer Object (DTO) Component: Facilitates data transfer between different layers, passing data from the data access layer to the service layer or presentation layer.

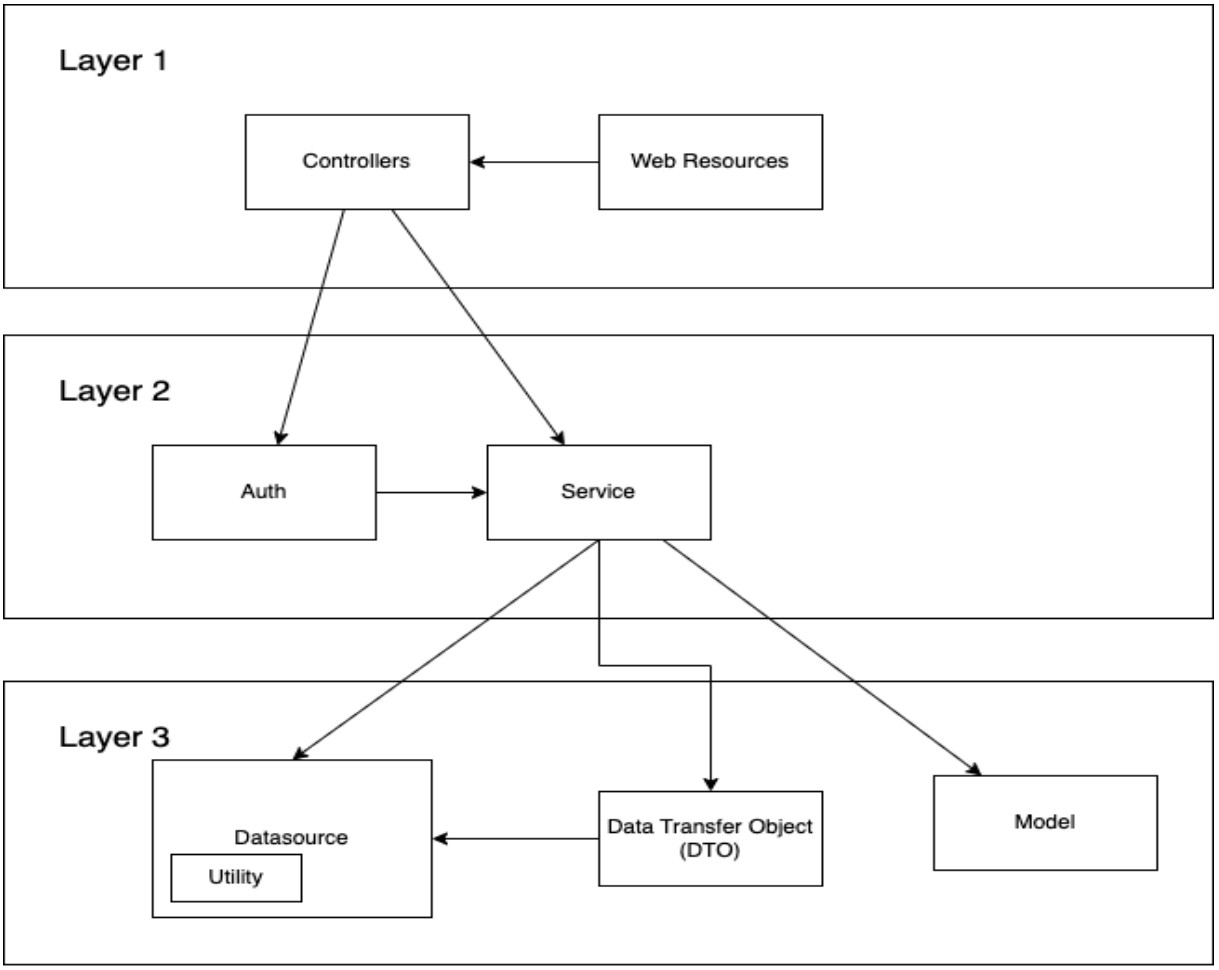


Figure 4.1 *General view of the architecture*

4.1 Components

4.1.1 Controllers Component

Responsibilities	The Controllers Component is responsible for receiving and processing user requests, routing them to the appropriate services, and sending responses to the user.
Handled Requirements	Handling user requests and directing them to the appropriate services, as well as sending responses to the user.
Justification	**The only option:**The choice of the Controllers Component is justified as the only option because it has been identified as the sole component capable of handling the requirement effectively.
Will it be reused?	No. Since controllers are typically specific to the application, they are less likely to be reused.

Source	Developed by the project.
Will it be reusable?	No. Controllers are specific to the application and not suitable for reuse by other projects.

4.1.2 Datasource Component

Responsibilities	The Datasource Component is responsible for managing connections and interactions with databases or other data sources.
Handled Requirements	This component deals with the requirements related to data source connections and data interactions.
Justification	**CIS Standard :** The choice of the Datasource Component is based on the CIS Standard as it aligns with established standards and practices in data source management.
Will it be reused?	Yes. If this is a generic data connection component, it may be reusable in other projects.
Source	Developed by the project or possibly retrieved from third-party libraries.
Will it be reusable?	Yes. Justification for developing a component that can be reused by other projects: If this is a generic data connection component, it can be reused in other projects.

4.1.3 Data Transfer Object (DTO) Component

Responsibilities	The Data Transfer Object (DTO) Component facilitates the transfer of data within the system, especially between different layers.
Handled Requirements	This component addresses the requirements related to data transfer between different layers.
Justification	**The only option:** This component is selected because it has been identified as the most suitable and necessary option to meet the specific data transfer requirements of the application.
Will it be reused?	No. DTOs are typically specific to the application and are less likely to be reused in other projects.
Source	Developed by the project.
Will it be reusable?	No. DTOs are usually tailored to the specific needs of the application, making them less suitable for reuse in other projects.

4.1.4 Model Component

Responsibilities	The Model Component represents the main data structures and entities of the application.
Handled Requirements	This component addresses the requirements related to defining and managing the data structures and entities used in the application.
Justification	**The only option:** This component is selected because it has been identified as the most suitable and necessary option to represent the specific data structures and entities required by the application.
Will it be reused?	No. Models are usually specific to the application and are less suitable for reuse in other projects.
Source	Developed by the project.
Will it be reusable?	No. Models are designed to meet the specific data needs of the application and are not intended for reuse in other projects.

4.1.5 Service Component

Responsibilities	The Service Component contains the critical business logic of the application, encompassing the core functionalities and operations.
Handled Requirements	This component effectively addresses the requirements related to implementing and managing the essential business logic of the application.
Justification	**The only option:** This component is selected because it has been identified as the most suitable and necessary option to encapsulate and execute the specific business logic required by the application.
Will it be reused?	Yes. Services might consider reuse, especially if they offer generic functionalities that could be beneficial in other projects.
Source	Developed by the project.
Will it be reusable?	Yes. There is potential for reuse if the services contain generic functionalities that can be applied to other projects.

4.1.6 Utility Component

Responsibilities	The Utility Component provides utility functions and tools that can be employed throughout the application, aiding in various tasks.
Handled Requirements	Provides utility functions and tools that can be used throughout the application.

Justification	**The only option:** This component is selected because it has been identified as the most suitable and necessary option to provide essential utility functions and tools required by the application.
Will it be reused?	Yes. Utilities are typically designed for reuse and can be valuable in other projects.
Source	Developed by the project.
Will it be reusable?	Yes. Utility functions and tools are typically designed for reuse and can be applied in other projects where similar utility functions are needed.

4.1.7 Web Resources Component

Responsibilities	The Web Resources Component contains frontend resources such as CSS, JS, and images, which are crucial for the presentation and user interface of the application.
Handled Requirements	Contains frontend resources such as CSS, JS, and images. Reuse: Called when a permission request is received.
Justification	**The only option:** This component is selected because it has been identified as the most suitable and necessary option to manage and serve the frontend resources required for the application's user interface.
Will it be reused?	Yes. Web Resources, especially common styles or scripts, might be reused in other projects.
Source	Developed by the project.
Will it be reusable?	No. Controllers are specific to the application and not suitable for reuse by other projects.

4.1.8 Authorization Component

Responsibilities	The Web Resources Component contains frontend resources such as CSS, JS, and images, which are crucial for the presentation and user interface of the application.
Handled Requirements	Determine the role of the user, if the user does not have permission to deny access.
Justification	**The only option:** This component is selected because it is the primary means of defining and structuring the core data entities required by the application to implement use cases such as "Manage Account" and "View Event."
Will it be reused?	No.

	Since models are typically specific to the application's data structure and entities, they are less likely to be reused in other projects.
Source	Developed by the project.
Will it be reusable?	No. Models are usually tailored to the specific data structure and entities of the application, making them less suitable for reuse in other projects.

5. Process Architectural representation

NA

6. Development View

This section provides orientations to the project and system implementation in accordance with the established architecture. Figure 6 illustrates the implementation view of system architecture.

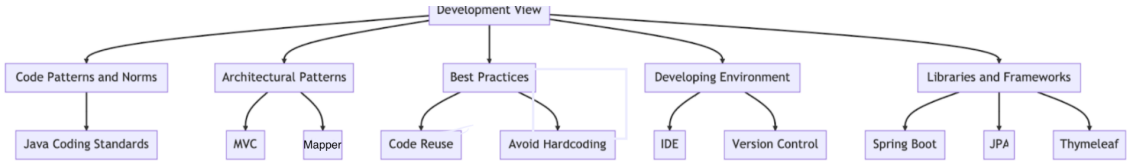


Figure 6. *Development view of system architecture*

6.1 Architectural Patterns

6.1.1 Domain Model

- Justification

This project involves multiple user roles, including administrators, event planners, and customers. The relationships between them are complex, involving event creation and ticket booking. To streamline these intricacies and streamline the business logic, domain models are implemented. These models contain the logic within corresponding domain objects, resulting in more efficient code management. Furthermore, domain models facilitate easy unit testing and future system expansion.

Although utilizing a table model can be advantageous in database operations, its capabilities are limited in managing intricate business logic and multi-role interactions. In contrast, transaction scripts

are more fitting for straightforward, sequential procedures. Nonetheless, for a venture that encompasses various roles and intricate interactions, like this one, the drawbacks of transaction scripts become apparent, particularly in regards to the testability and maintainability of the code.

In general, due to the business complexity of the project, the participation of multiple roles, and the need for future scalability, the domain model became the most appropriate choice. Compared with table models and transaction scripts, it has obvious advantages in organizing complex business logic and ensuring code quality.

- Implementation

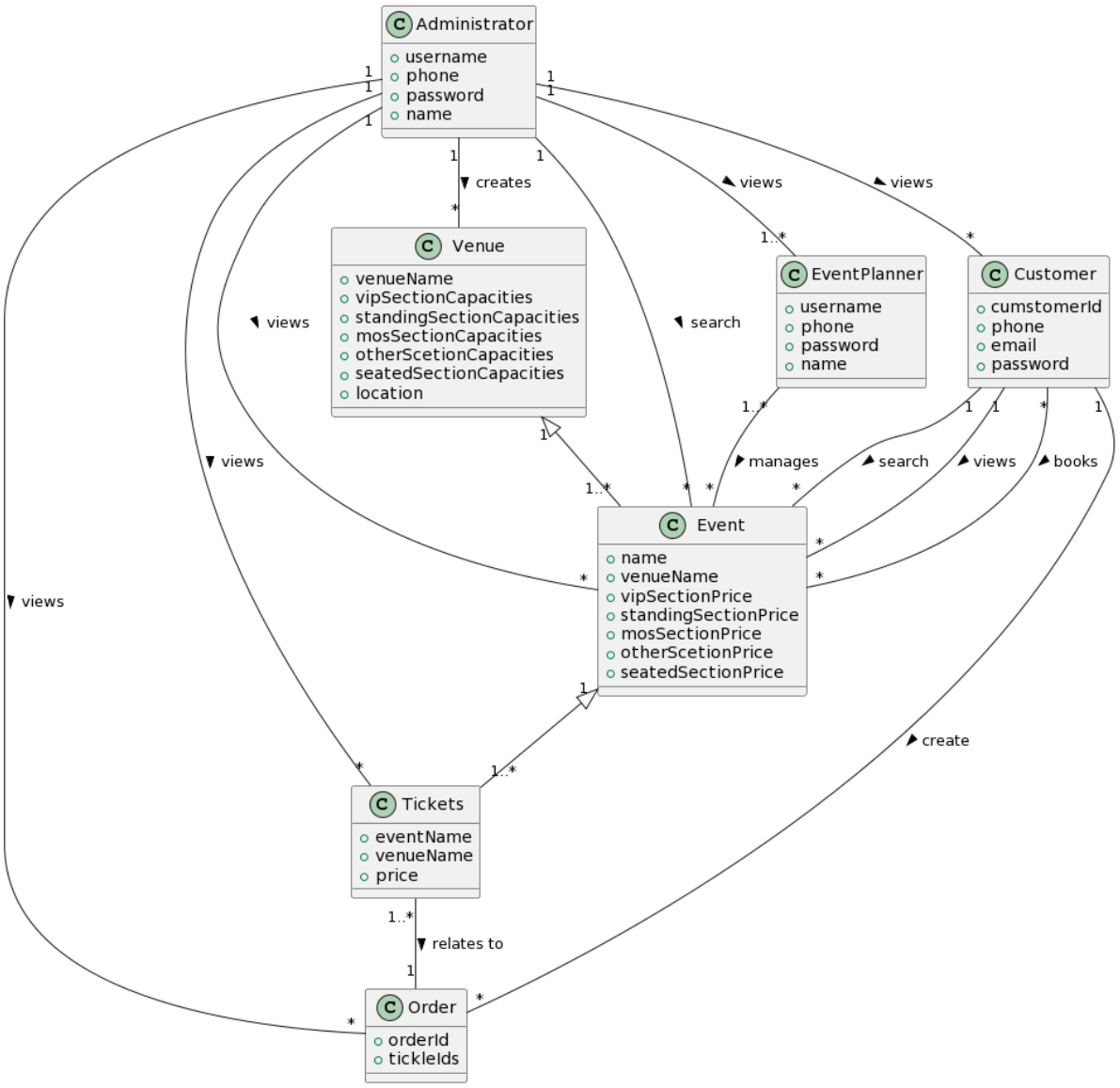


Figure 6.1.1.1 *Domain Model*

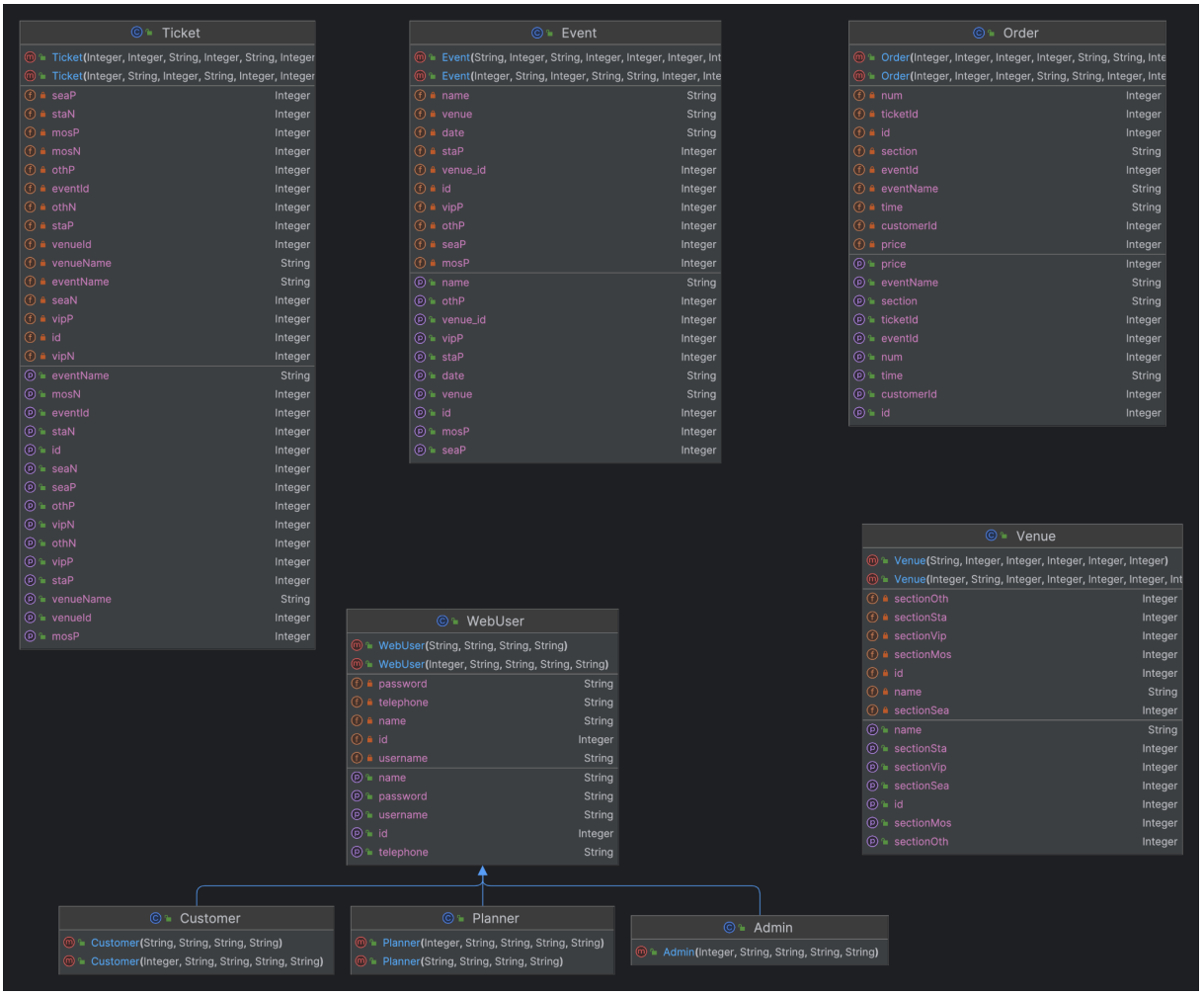


Figure 6.1.1.2 *Class Diagram*

6.1.2 Data Mapper

- Justification

The project has multiple different user roles and complex business logic, including event booking, ticket management, multi-role permissions, etc. The data mapper pattern can isolate the data interaction logic between objects and databases so that the business logic layer can focus more on processing business rules without caring about the details of data storage. This also makes unit testing easier while providing greater scalability.

Although Table Data Gateway and Row Data Gateway are simple, they are closer to database operations and are not suitable for handling complex business logic. The Active Record pattern combines data access logic and business logic in a single object, which may make the code difficult to manage in scenarios like this project that require multiple roles and complex interactions. Data mappers provide greater flexibility and maintainability by separating concerns.

- Implementation

In this project, we employed a consistent mapper pattern across all tables, as detailed below. The majority of these mappers feature a set of standard functions, including *list()*, *search()*, *update()*, *save()*, and *delete()*, among others. These mappers are all housed within the "***datasource***" package, ensuring that only this layer interacts directly with the database. As a result, other layers in the application can remain agnostic to the complexities of data handling.

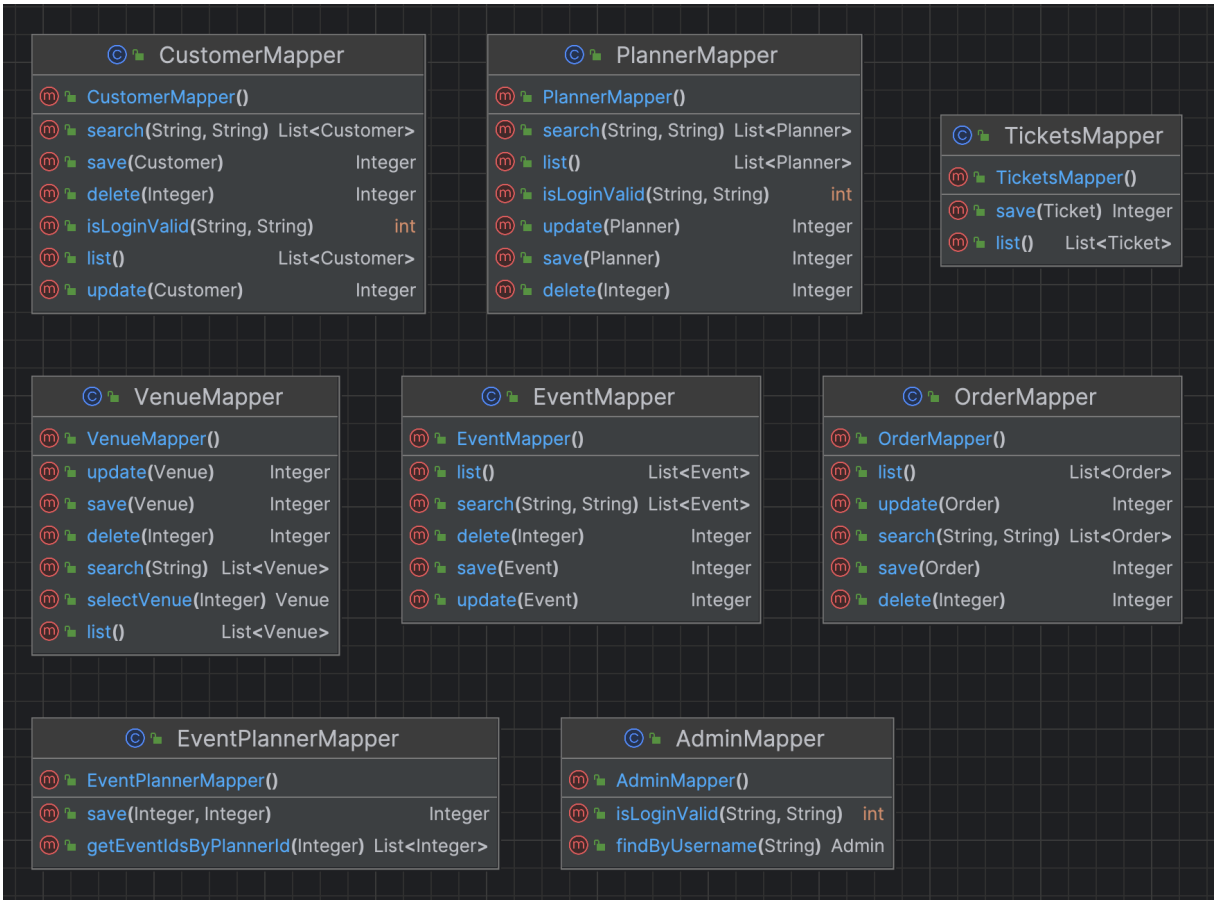


Figure 6.1.2.1 *Data Mapper Classes*

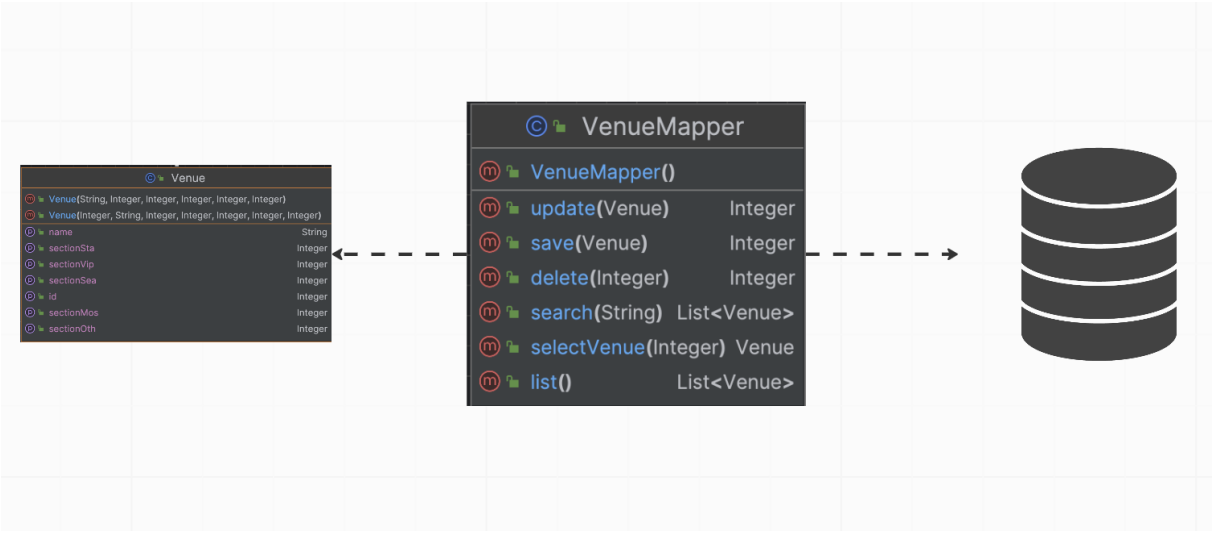


Figure 6.1.2.2 *Data flow of VenueMapper Implementation*

I will showcase the VenueMapper as an illustrative example to elucidate the process of leveraging the data mapper pattern to retrieve venue-related information. The following sequence diagrams illustrate the general functionality of some methods that are commonly used among each DataMapper.

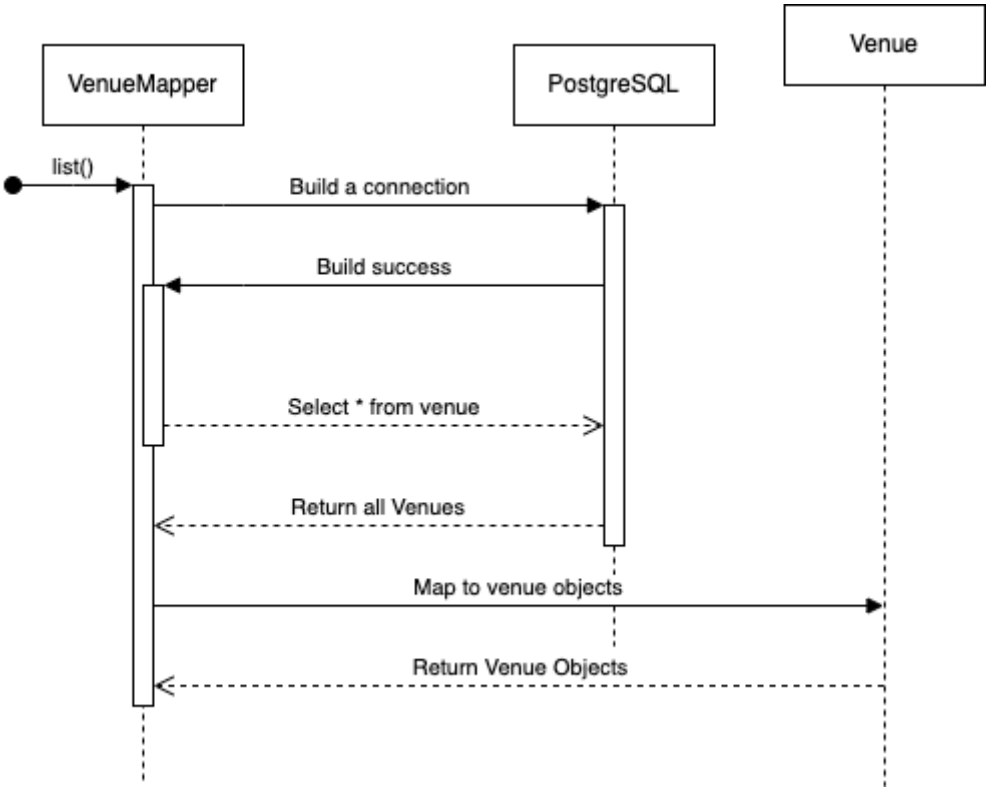


Figure 6.1.2.3 *VenueMapper lists all venues in the table*

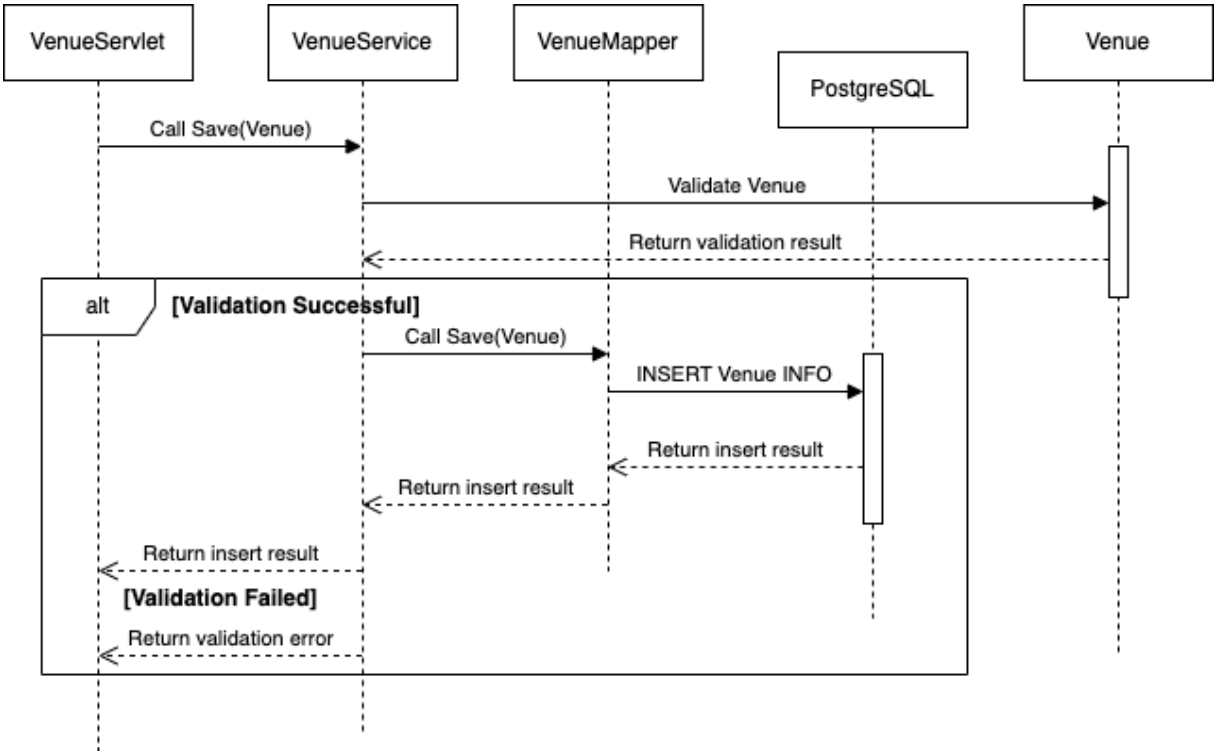


Figure 6.1.2.4 *VenueMapper inserts a new venue into the table*

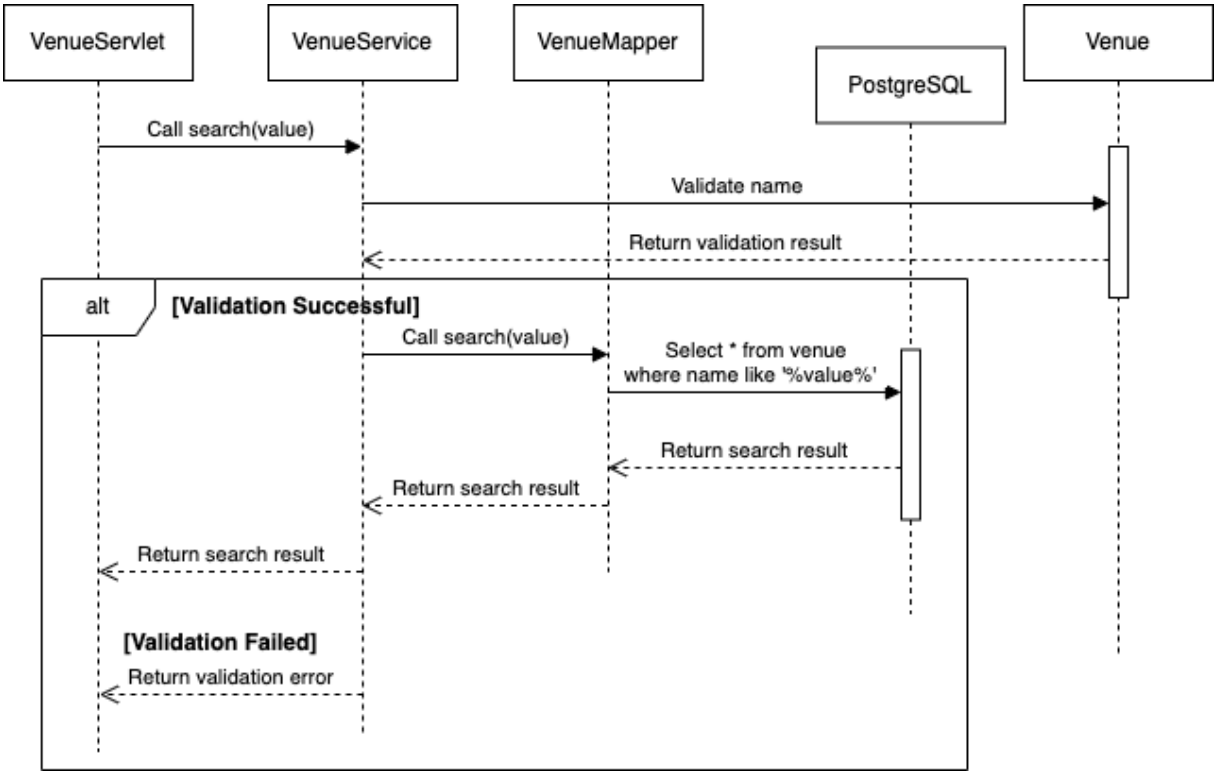


Figure 6.1.2.5 *VenueMapper searches venues by name*

6.1.3 Unit of Work

- Justification

The Unit of Work pattern keeps track of everything you do during a business transaction that can affect the database. This feature manages a collection of tasks that need to be executed together. By doing so, it guarantees that all database operations related to the tasks either succeed or fail as a unit. This ensures the consistency and accuracy of the data. This function is especially useful for this project, as actions such as creating a music event, booking tickets, and more may involve several database tables and operations.

While using database transactions directly is an option, the unit of work pattern provides a higher level of abstraction that makes the business logic clearer and maintainable. Database transactions are typically low-level and targeted at a single data source, while the unit of work model allows transactions across multiple objects and operations, providing greater flexibility.

Overall, the unit work model provides an efficient and consistent way to handle complex business logic and data operations. It not only ensures data consistency and integrity but also provides a higher level of abstraction and greater flexibility than underlying database transactions. This makes the code easier to understand and easier to maintain, which is perfect for this music event system.

- Implementation

1. Ticket Purchasing Process:

When a customer decides to purchase a music event ticket, several database operations are required. These operations include fetching data from the Event table, updating the Tickets table, and adding a new entry to the Orders table. To manage these operations as a single transaction, the Unit of Work pattern can be employed.

By using the Unit of Work pattern, we can efficiently manage multiple database operations as a single transaction, thereby ensuring data consistency and integrity for music event ticket purchases.

After the customer decides on a specific ticket to purchase, another set of database operations is triggered. These operations involve fetching customer information, updating the Tickets table, and adding a new entry to the Orders table. The Unit of Work pattern is again used to manage these operations as a single transaction.

By extending the Unit of Work pattern to the purchase phase, we can ensure that all database operations related to ticket purchasing are managed as a single, cohesive transaction, thereby maintaining data consistency and integrity.

This is our design but we still working on it and not yet finish the implementation of UinitOfWork on purchasing tickets.

2. Customer information updates deletes and inserts:

Our application greatly benefits from the Unit of Work pattern, which has been elegantly integrated into the customer service functionality. The simplicity of the pattern shines in our design; for example, in the "delete" function, initializing a new unit of work, registering objects for deletion, and

committing the transaction can be accomplished with just a few lines of code. This proves that the Unit of Work pattern provides a very simple and effective way to manage object lifecycle, even in complex applications.

The efficiency and high cohesion of the unit of work pattern are also evident in our implementation. With this pattern, we can make multiple object changes (such as add, modify, and delete) in a single transaction, thereby reducing the number of database round-trips and improving performance. All information about the object's state is centralized in the unit of work instance, making it easier to maintain and debug the code because changes are atomic and limited to a single location per thread. This high cohesion ensures that our applications are not only efficient but also robust and maintainable.

```
public void delete(Integer id) {
    UnitOfWork.newCurrent();
    try {
        Customer customerToDelete = customerMapper.findById(id);
        UnitOfWork.getCurrent().registerDeleted(customerToDelete);
        UnitOfWork.getCurrent().commit();
    } catch (Exception e) {
        throw new RuntimeException("Transaction failed", e);
    }
}
```

Figure 6.1.3.1 *UnitOfWork example of deleting a customer*

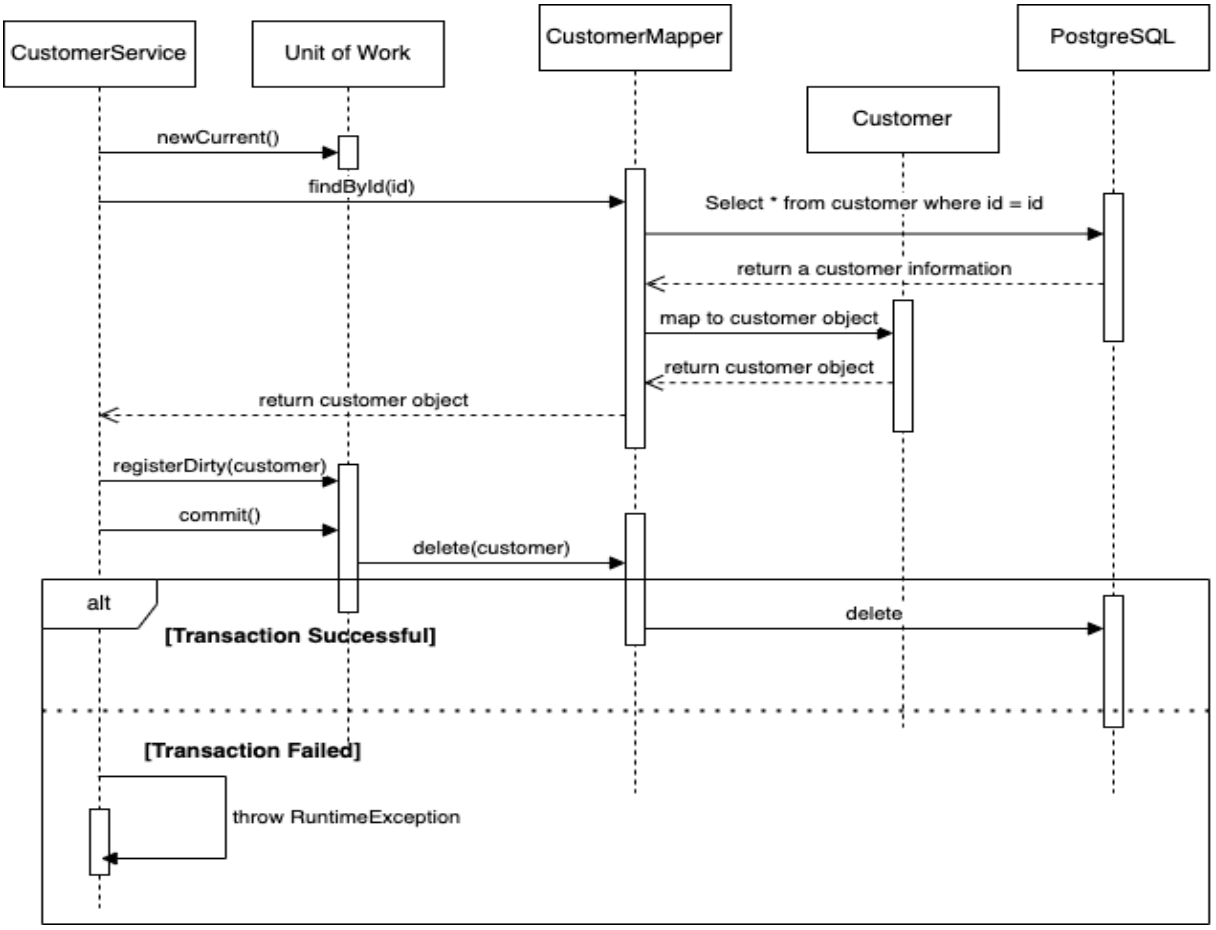


Figure 6.1.3.1 *Sequence Diagram for deleting a customer using UnitOfWork*

6.1.4 Lazy Load

- Justification

In this ***Music Event System*** project, we chose to use the lazy load pattern for several reasons. First, lazy loading can improve the performance and responsiveness of the system. Only when specific information is needed (such as details of an event or a user's booking history), the system will load this information from the database. This can reduce unnecessary database queries and consumption of system resources. Second, because our application needs to handle multiple user roles (such as administrators, event planners, and customers) and their different needs, lazy loading makes the system more flexible and can be optimized based on specific use cases or user behaviours.

In contrast, another common data loading strategy is "Eager Load", which loads all relevant data at the beginning. Although this approach has its advantages in some cases, such as reducing the need for subsequent queries, for this system, it may lead to performance degradation and resource waste during the initial load, especially when the user only needs part of the information or When performing simple operations.

- Implementation

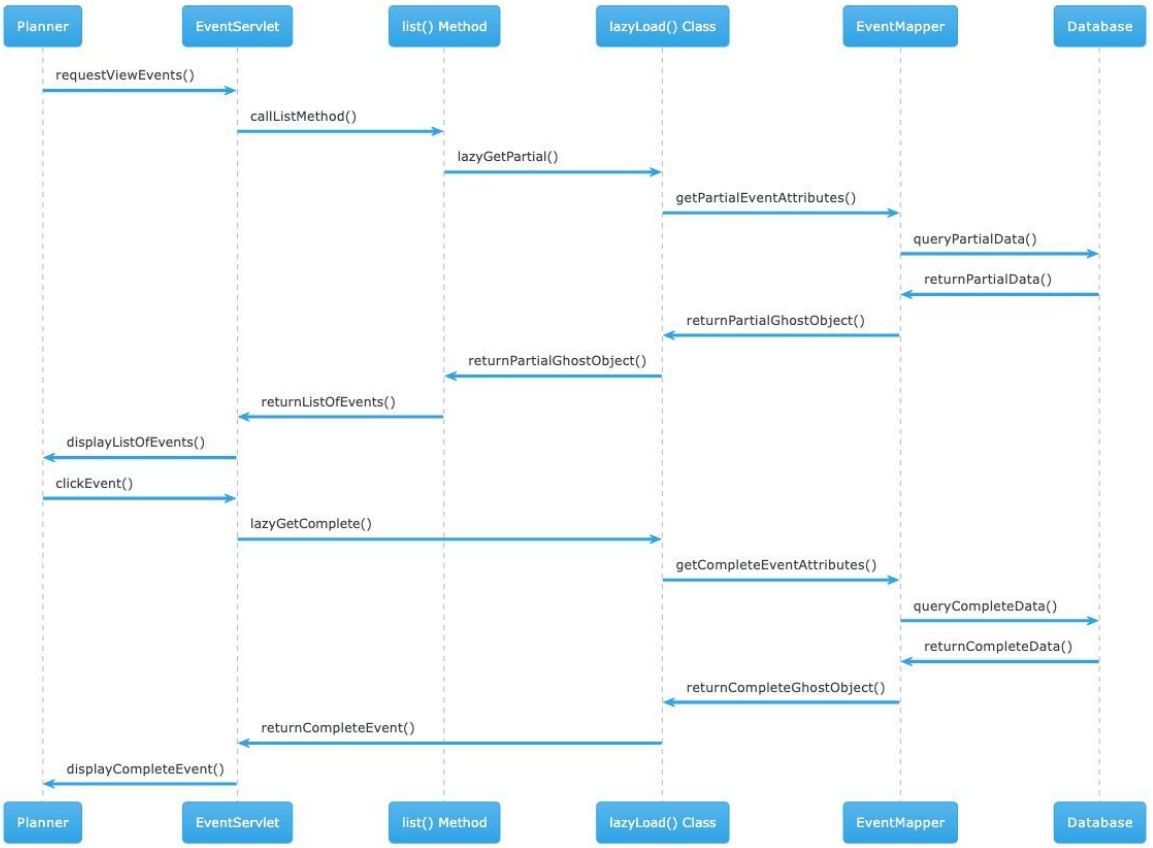


Figure 6.1.4.1 *Lazy Load*

In the Music Event System, the Planner interacts with the EventServlet to view a list of events. Initially, the EventServlet invokes the list() method, which in turn uses the lazyLoad class to fetch only partial attributes of each event from the Database via the EventMapper. This partial data is then displayed to the Planner. When the Planner clicks on a specific event for more details, the EventServlet calls the lazyLoad class again, but this time to fetch all attributes of the clicked event. The lazyLoad class queries the Database through the EventMapper to get the complete data, which is then displayed to the Planner. This approach effectively optimizes system performance by ensuring that only the necessary data is loaded at each step.

6.1.5 Identity Field

- Justification

For our music event system, we utilize the "Identity Field" pattern. This pattern allows for efficient identification and management of the mapping relationship between the database and object model. Each persistent object, such as a user, activity, or venue, has a unique identifier. This approach enhances data accuracy and consistency and simplifies object retrieval and update operations.

Compared with other identification methods such as "Natural Key" or "Composite Key", identification fields provide a simpler and more unified solution. Natural and composite keys often rely on business logic or multiple fields, which can lead to more complex queries and higher maintenance costs. The identification field is usually automatically generated by the database and has nothing to do with business logic, so it is more stable and efficient.

In summary, the "identity field" pattern can effectively simplify the complexity of object identification and data management.

- Implementation

When implementing the Identity Field pattern, we defined a unique identifier field named ID for each table in the database. Specifically, I used SERIAL PRIMARY KEY as the data type and constraints for this field. In this way, whenever a new record is inserted into the table, the database automatically generates a unique and self-increasing ID value for the record. The figure below displays the detailed information.

This implementation has several advantages:

1. Automatic management: The database is automatically responsible for the generation and maintenance of IDs, reducing the risk of errors.
2. Data consistency: PRIMARY KEY constraints ensure that the ID field in each table has a unique value, thereby ensuring data consistency.
3. Simplified queries: With unique identifiers, data retrieval and related queries become more straightforward and efficient.

By using ID SERIAL PRIMARY KEY, the Identity Field pattern is implemented the Identity Field pattern, making the data more consistent and easier to manage.

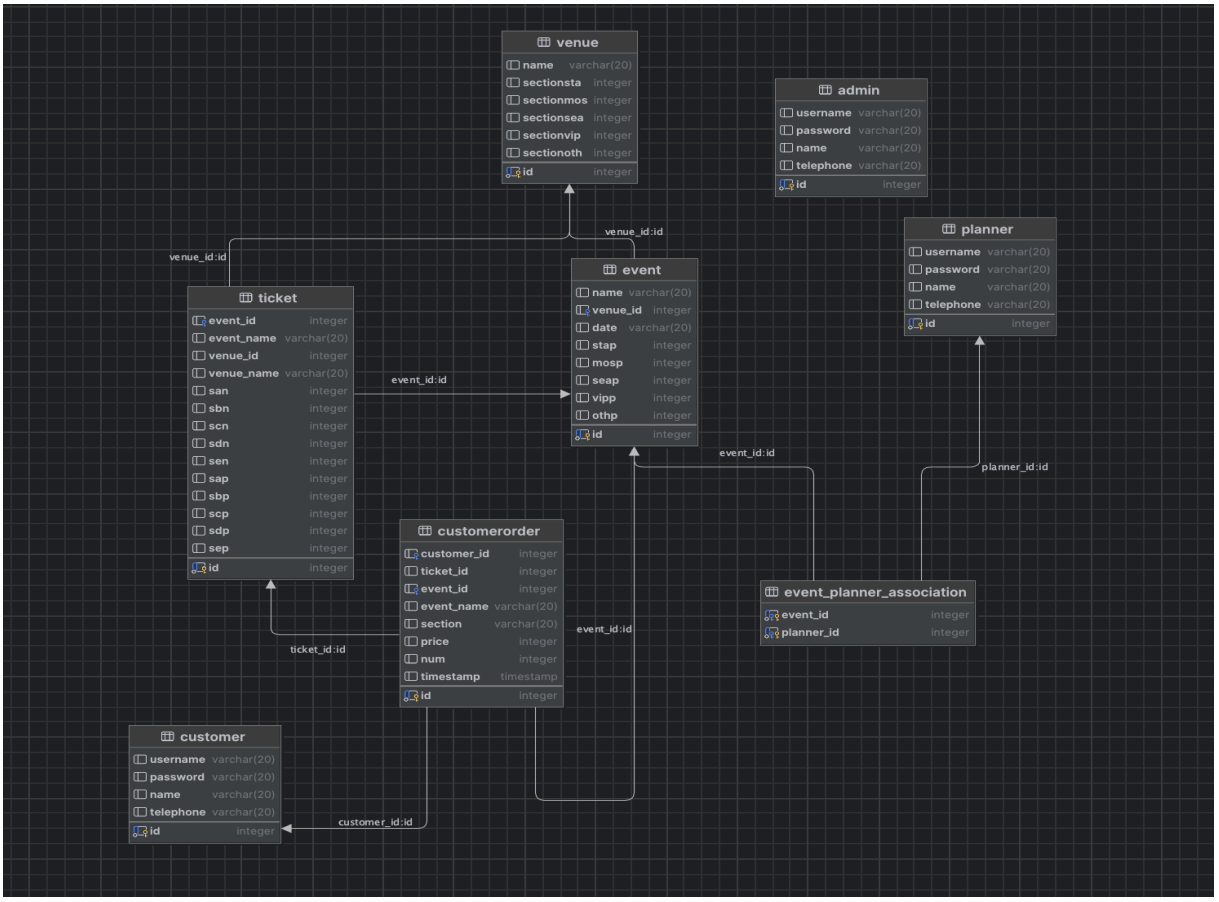


Figure 6.1.5.1 *Entity relationship diagram*

6.1.6 Foreign key Mapping

- Justification

It provides a structured and reliable way to represent the relationships between different tables in the database. For example, the Customer_ID field in the CustomerOrder table is a foreign key that references the id field in the customer table. The advantage of this is that it ensures data consistency and integrity because the database management system automatically performs referential integrity checks. In addition, foreign key mapping also makes joining queries and data associations simpler and more intuitive, thereby improving the efficiency and accuracy of data processing. Overall, by using foreign key mapping, we can more easily maintain and manipulate data models with complex relationships.

- Implementation

The detailed information is shown in figure 6.1.5

Table	Foreign Key	Reason
event	venue_ id	event can use venue_id to get the information of the venue
event_planner_association	event_id planner_id	It can save the information about “an event can be held by many planners”
ticket	event_id venue_id	ticket can get the information of the event and venue by their ids
custommerorder	event_id ticket_id customer_id	the oder table can save the information of the event, ticket and customer information by their ids

6.1.7 Association Table Mapping

- Justification

Use Association Table Mapping to handle many-to-many relationships, which is a method particularly suitable for complex relationship models. By using relational table mapping, we are not only able to represent complex data relationships more accurately but also provide greater flexibility when querying. This also helps keep the database model clean and consistent, making it easier to maintain and expand later. In general, association table mapping is an efficient and reliable way to process and represent complex data relationships involved in projects.

- Implementation

To accommodate the complex relationships between planners and events, wherein multiple planners may oversee numerous events concurrently, we have instituted a specialized table named `event_planner_association`. This table is specifically designed to store many-to-many relational data. Consequently, this structure allows for efficient querying to ascertain which planners are responsible for a given event, as well as to determine the scope of events managed by a single planner. This strategic database architecture not only enhances data retrieval efficiency but also facilitates scalable and robust event-planner relationship management.

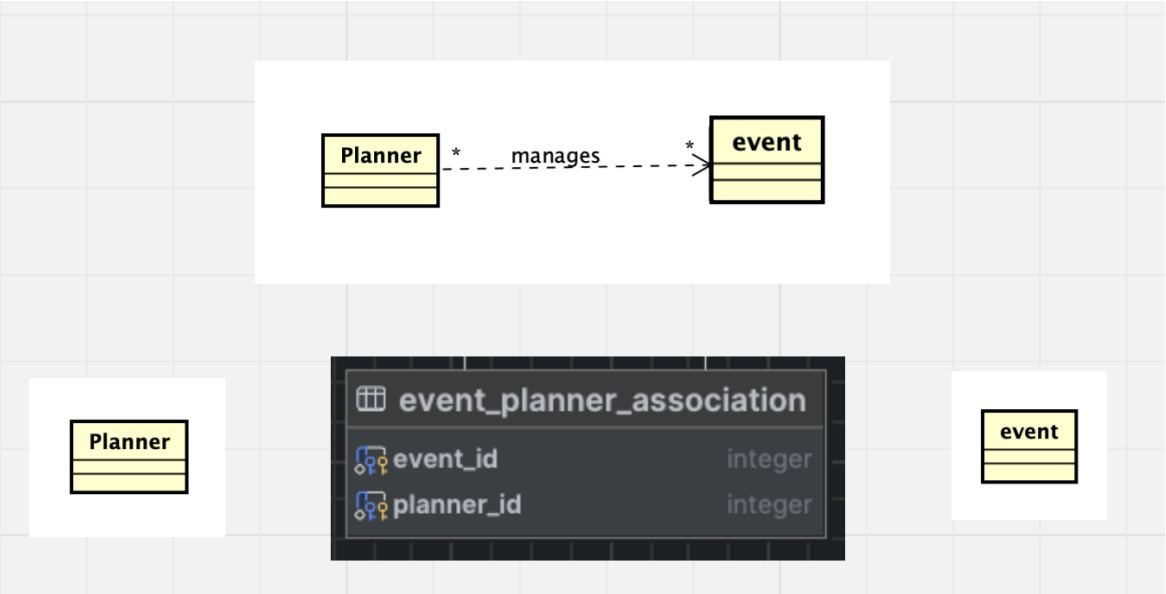


Figure 6.1.7.1 *Association table example*

6.1.8 Embedded Value

- Justification

Some objects have a set of value attributes that logically form an integral whole, such as an address. By using the embedded value pattern, we can encapsulate this set of properties into a single object and then embed this object into the larger object that owns it (such as a customer or venue).

Using the embedded value pattern helps improve the readability and maintainability of code. It makes it easier to reuse and modify this closely related set of properties, rather than dealing with them separately in multiple places. This schema also aids in data consistency because it enforces that this set of properties is treated as a single, indivisible unit. Overall, the embedded value pattern provides an efficient and organized way to manage and use these logically related collections of properties.

- Implementation

In ***venue*** table

1. Section_Standing int, -- Embedded value
2. Section_Mosh int, -- Embedded value
3. Section_Seated int, -- Embedded value
4. Section_VIP int, -- Embedded value
5. Section_Other int -- Embedded value

In ***event*** table

1. Section_Standing_Price int, -- Embedded value
2. Section_Mosh_Price int, -- Embedded value
3. Section_Seated_Price int, -- Embedded value
4. Section_VIP_Price int, -- Embedded value
5. Section_Other_Price int, -- Embedded value

In this project, we use the "Embedded Value" pattern to implement the complex data structures of venues (Venue) and activities (Event). In the `venue` table, we have embedded multiple capacity fields representing different areas (such as "stand area", "Mosh area", "seating area", "VIP area" and "other areas"), such as `Section_Standing` , `Section_Mosh`, `Section_Seated`, `Section_VIP` and `Section_Other`. These fields are of integer type and are used to store the seating capacity of their respective areas.

Similarly, in the `event` table, we have also embedded fields representing the ticket prices of each area, such as `Section_Standing_Price`, `Section_Mosh_Price`, `Section_Seated_Price`, `Section_VIP_Price` and `Section_Other_Price`. The advantage of this is that we can directly obtain or modify all the information related to a specific area in a single table, without having to go through multiple tables or complex queries.

This "Embedded Value" design greatly simplifies data query and modification operations, and improves the efficiency and maintainability of the system. At the same time, it also makes the code clearer and reduces the probability of errors. Overall, in this way, we successfully embedded multiple attributes related to venues and events into a single database table for more efficient and structured data management.

6.1.9 Concrete Inheritance Pattern

- Justification

We chose to design using the Concrete Inheritance pattern. This mode allows us to create a separate table in the database for each concrete subclass, which helps reduce data redundancy and improve query performance. Compared with other inheritance patterns, such as single-table inheritance or class-table inheritance, concrete inheritance can more directly reflect the unique properties and behaviours of each subclass.

Secondly, the specific inheritance model enables each subclass to have an independent data structure and business logic, which greatly improves the readability and maintainability of the code. It is also easier to support the independent evolution of each subclass because each subclass has its own independent database table.

Overall, the concrete inheritance pattern provides us with a flexible and efficient way to deal with polymorphism and inheritance relationships.

- Implementation

We use the Concrete Table Inheritance strategy where each table (e.g. `Admin`, `Planner`, `Customer`) stores all necessary fields independently.

This design pattern simplifies the process of reading and writing data by eliminating the need to traverse multiple tables to collect comprehensive user information. Given that this project does not require sharing a large number of fields between different user types, using single table inheritance might make it cumbersome to add foreign keys to other tables that would reference the IDs of different web users. On the other hand, using class table inheritance introduces additional complexity to the login process due to the additional search required. Therefore, after due consideration, concrete table inheritance emerged as the most suitable approach for this project.

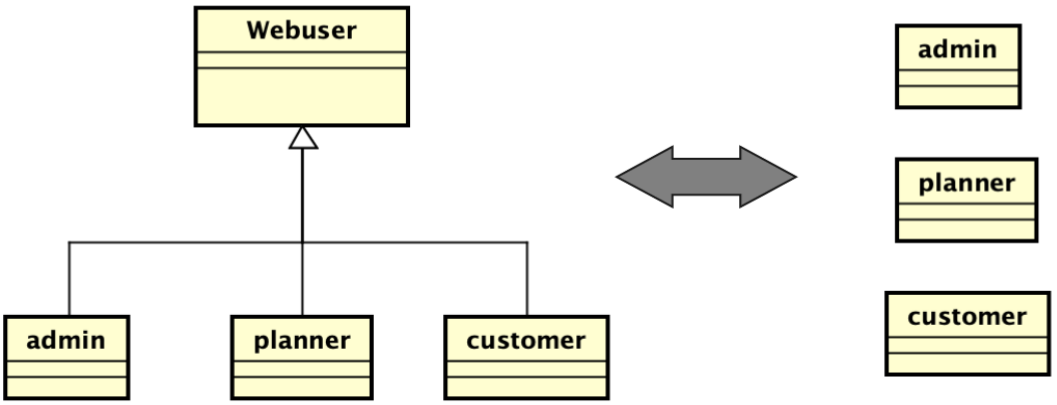


Figure 6.1.9.1 *Concrete Table Inheritance*

6.1.10 Authentication and Authorization

- Justification

Implementing robust authentication and authorization mechanisms is crucial for any system or application, ensuring that only legitimate users can access the system and that they can only perform actions they are permitted to. This prevents unauthorized data access or malicious actions. Additionally, it provides an audit trail, logging who did what and when, which is vital for accountability and compliance. Incorporating these strong processes not only safeguards sensitive data and functionalities but also fosters trust with users, ensuring their security and confidence while using the system or application.

- Implementation

Authentication Process:

In the login interface, users can select the type of account they wish to log in with. The AuthenticationClass then uses the isLoginValid method to verify whether the username and password are correct. If they are, the server sets the corresponding roleType for the user and stores it in the session.

Authorization Process:

User permissions are managed through the AuthorizationClass, which maintains a permission list for each user and provides a checkPermission method to check permissions.

After a user logs in and attempts to access a specific page, such as the Event page, the EventServlet is invoked for processing. At this point, the system retrieves the user's authentication result from the session and uses the checkPermission method in the AuthorizationClass to verify whether the user has

the necessary permissions in their permission list. If they don't, the user is redirected to a "No Permission" page with an option to return to the previous page. If they do have the required permissions, the process continues as expected.

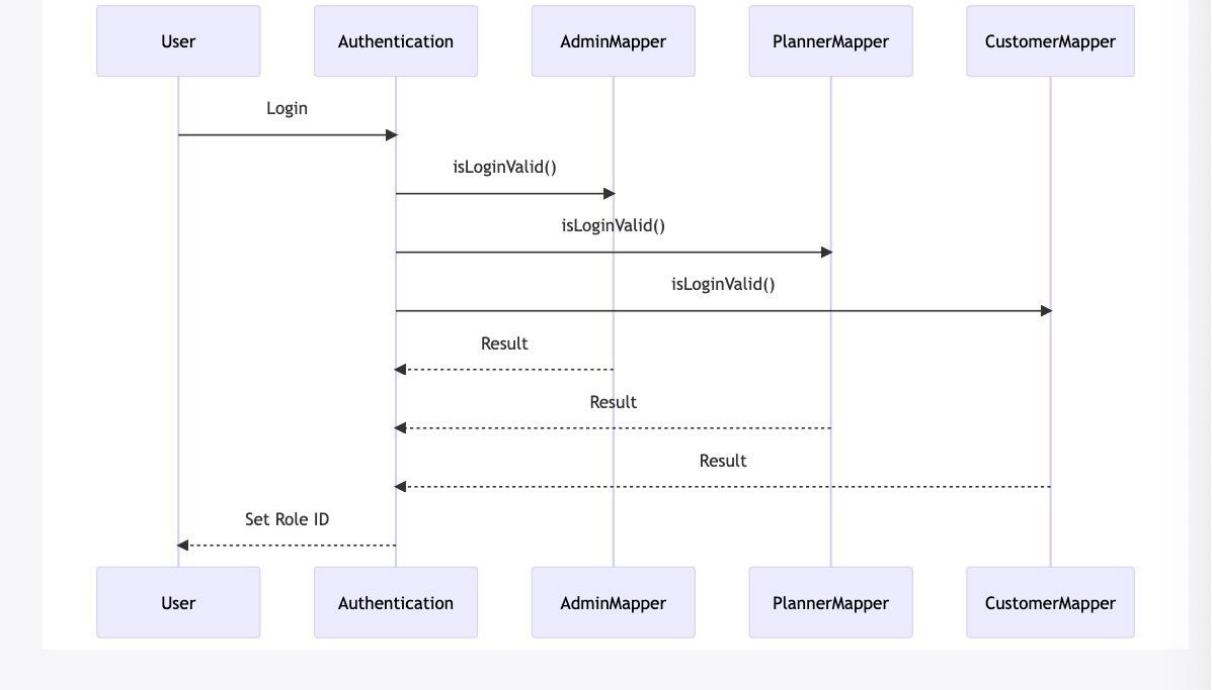


Figure 6.1.10.1 *Authentication and Authorization*

Process Steps:

1. The user attempts to log in.
2. The Authentication class calls the isLoginValid() method of the corresponding Mapper (AdminMapper, PlannerMapper, CustomerMapper) based on the user type.
3. The Mapper returns the validation result.
4. The Authentication class sets the user role ID based on the validation result.

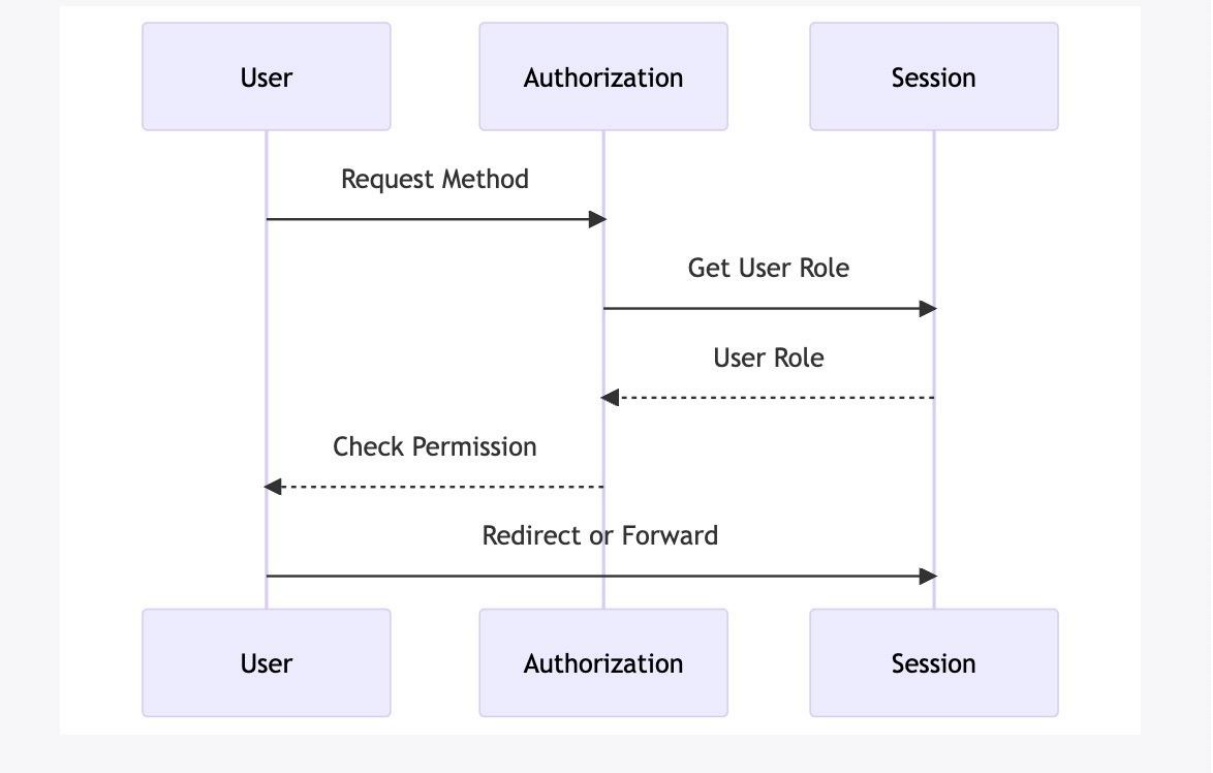


Figure 6.1.10.2 *Authorization Process*

Process Steps:

1. The user requests to perform a certain method (e.g., list all customers).
2. The Authorization class retrieves the user role from the Session.
3. The Authorization class checks if the user has permission to perform the method.
4. Based on the permission check, the user is either redirected or the request is forwarded.

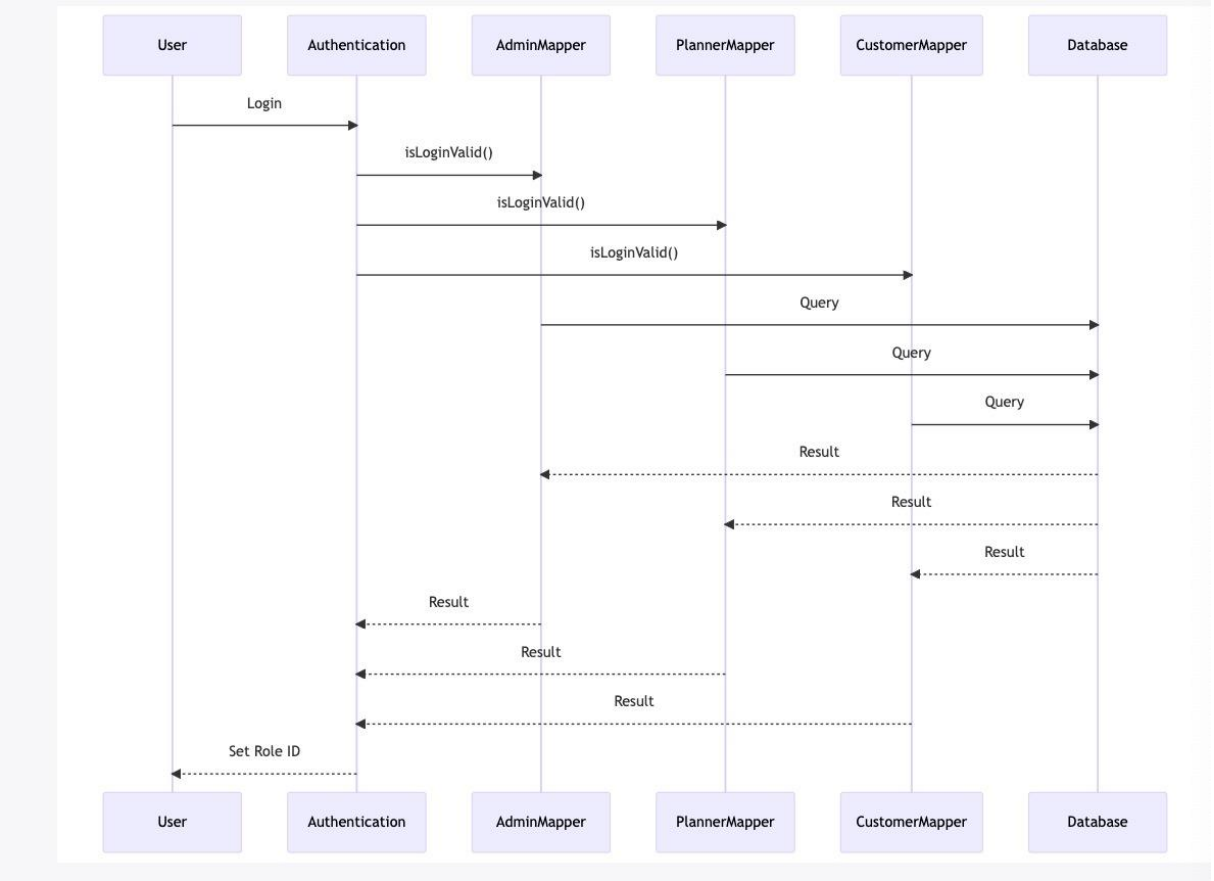


Figure 6.1.10.3 *Authentication Process with Database and Mapper Interaction*

Process Steps:

1. The user attempts to log in.
2. The Authentication class calls the isLoginValid() method of the corresponding Mapper based on the user type.
3. The Mapper sends a query request to the database.
4. The database returns the query result to the Mapper.
5. The Mapper returns the validation result to the Authentication class.
6. The Authentication class sets the user role ID based on the validation result.

6.1.11 Model View Controller (MVC)

- Justification

The Model View Controller (MVC) design pattern is a cornerstone in software development, emphasizing a clear separation of data (Model), user interface (View), and control flow (Controller). This separation fosters modularity, allowing for parallel development and efficient testing. By ensuring components are reusable and adaptable, MVC enhances the flexibility and maintainability of applications. It streamlines the development process, making it easier to accommodate changes, whether in the user interface or underlying data handling. Furthermore, MVC promotes interactivity, leading to a more responsive user experience. In essence, adopting MVC provides a structured and efficient approach to building scalable and maintainable software applications.

- Implementation

Music Event System followed the MVC architecture pattern which contains a model, view and controller.

Model

The model in the Music Event System represents the data of the domain. The model contains domain mode. The design rationale and implementation for the domain model are discussed in **sections 6.1.2**.

View

The Music Event System chose to adopt the template view design pattern, incorporating JavaScript markers into static HTML pages. This choice prioritized system architecture over UI design, favoring simplicity and ease of use. The implementation utilized JSP server pages for view development, while styling was achieved using Bootstrap. This approach streamlined the development process and allowed the team to focus on architectural robustness while maintaining a user-friendly interface.

Controller

For the implementation of the controller aspect in the MVC (Model-View-Controller) architecture, the Music Event team adopted the page controller pattern. In this design pattern, a single controller object is responsible for handling requests specific to a particular webpage or a specific action on a website. These controllers are invoked by clients through API calls, such as GET or POST requests, and they extract user data either from form submissions or URL parameters. Once the data is obtained, the controller follows predefined behavior associated with the invoked method. It communicates with mappers and the domain model to request data and process the user's query. Finally, the controller serves the user with a content page generated by the view.

The advantages of using page controllers include their simplicity and ease of comprehension. Each controller has a well-defined responsibility for both the model and view aspects, making it straightforward to debug during development and reducing coupling within the system.

The team's choice to employ the page controller pattern was driven by its ease of understanding and its direct alignment with front-end page development. Each controller was designed to handle specific tasks, such as calling relevant model methods, processing user queries, and presenting the corresponding view.

This approach ensured clear separation of concerns and facilitated efficient development and maintenance of the Music Event system.

Given below (figure) is an example where creation of an event with an associated planner on Music Event system. The eventServlet controller responds to an initial GET request to display the content defined in manageevent.jsp. When the has entered details and submits to the controller as a POST method, the eventServlet controller receives the data from the user, upon which, if the data is valid, the controller calls the relevant mappers to create new data in the database, redirecting the user when the actions have completed.

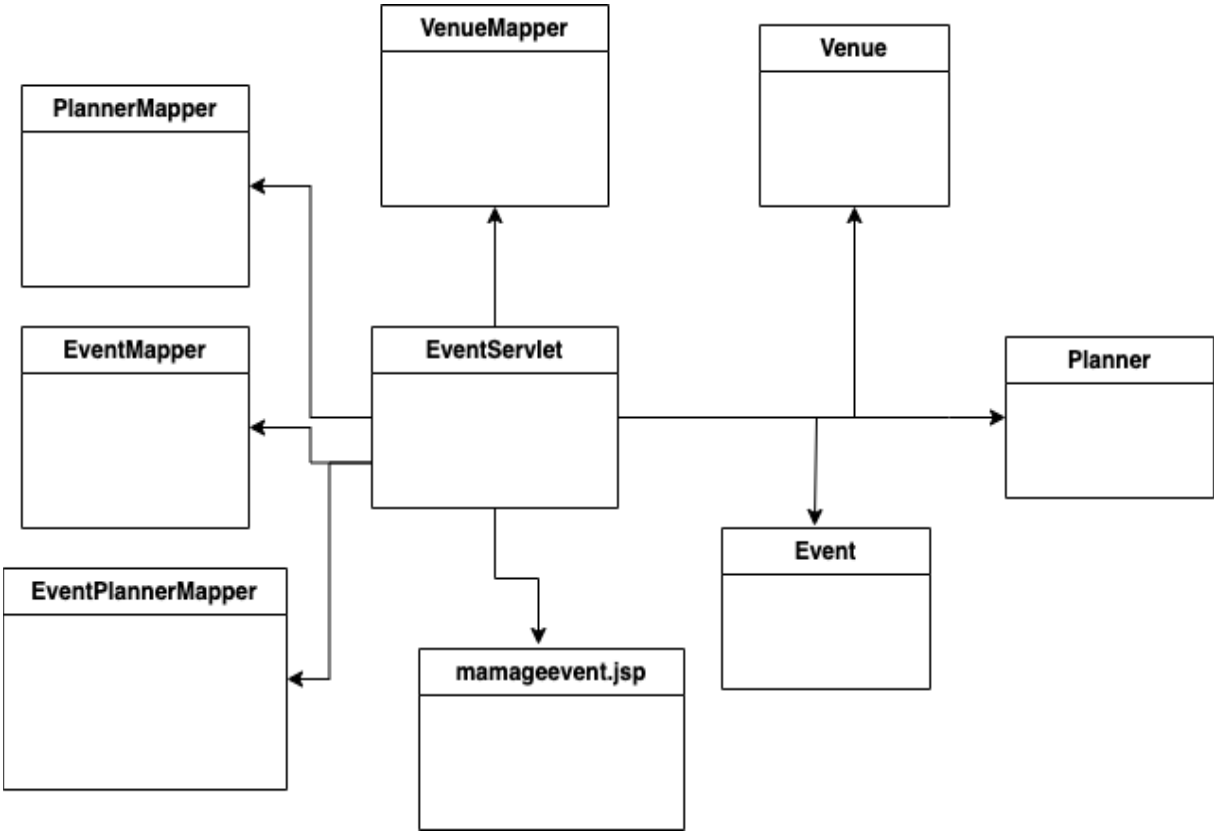


Figure 6.1.11 *Model View Controller (MVC)*

6.2. Source Code Directories Structure

For the MusicEventSystem project, the source code directory structure is as follows:

```
├── src
│   └── main
│       ├── java
│       │   └── com
│       │       └── example
│       │           └── musiceventsystem
│       │               ├── auth
│       │               ├── controllers
│       │               ├── datasource
│       │               ├── dto
│       │               ├── model
│       │               ├── service
│       │               └── util
│       └── webapp
```

6.3. Libraries and *Frameworks*

This section describes libraries and frameworks used by MusicEventSystem.

Library / Framework	Version	Environment
jakarta.servlet	2.0.0	Production and Development
Bootstrap	5	Production and Development
postgresql	42.6.0	Production and Development

Library / Framework	Version	Environment	Library / Framework
jakarta.servlet-api	The only option: Essential for building Java-based web applications.	5.0.0	All
junit-jupiter-api	CIS Standard: Widely used for unit testing in Java applications.	5.9.2	Test
junit-jupiter-engine	CIS Standard: Required to run JUnit 5 tests.	5.9.2	Test
jakarta.servlet.jsp.jstl	The only option: Provides support for JSP Tag Libraries, enhancing JSP functionality.	2.0.0	All
taglibs-standard-spec	Decision Making: Enhances JSP functionality with standard tags.	1.2.5	All
taglibs-standard-impl	Decision Making: Implementation of the standard tag library.	1.2.5	All
postgresql	Decision Making: JDBC driver for PostgreSQL, chosen for database connectivity.	42.6.0	All

6.4. Development Environment

Maven: Used for project management and building. It assists in managing project dependencies, executing build lifecycle tasks, and packaging and deploying the application.

Java JDK 11: Java JDK 11 or upper vision until 20 is required to compile and run the application.

JUnit 5: Used for unit testing. The project utilizes JUnit 5 for testing purposes.

Jakarta Servlet API: Essential for building Java web applications. It forms the foundation of the web application.

Spring Security: Used for application security. The code includes dependencies for Spring Security's Web and Config, indicating that the project uses Spring Security to enhance application security.

PostgreSQL JDBC Driver: Used for connecting to the PostgreSQL database. This suggests that the project might be using PostgreSQL as its database.

JSP and Tag Libraries: The project uses JSP as its view technology and employs tag libraries to enhance the functionality of JSP pages.

Integrated Development Environment (IDE): The project is developed in IntelliJ IDEA.

Version Control: As the project is hosted on GitHub, Git is used as the version control tool.

7. Physical View

NA

8. Scenarios

NA

9. References

- Cabibbo, L. (2009). On keys, foreign keys and nullable attributes in relational mapping systems. https://doi.org/10.1145/1516360.1516392
- Carreira, P., Galhardas, H., Pereira, J., & Lopes, A. (2005). Data Mapper: An Operator for Expressing One-to-Many Data Transformations. Lecture Notes in Computer Science, 136–145. https://doi.org/10.1007/11546849_14
- Conklin, A., Dietrich, G., & Walz, D. (2004). Password-based authentication: a system perspective. 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of The. https://doi.org/10.1109/hicss.2004.1265412
- Coutinho, J. C. S., Andrade, W. L., & Machado, P. D. L. (2019). Requirements Engineering and Software Testing in Agile Methodologies. Proceedings of the XXXIII Brazilian Symposium on Software Engineering - SBES 2019. https://doi.org/10.1145/3350768.3352584
- Ebert, C., & Jones, C. (2009). Embedded Software: Facts, Figures, and Future. Computer, 42(4), 42–52. https://doi.org/10.1109/mc.2009.118

- Niculescu, M., Ph. Dugerdil, & Canedo, B. M. (2015). Measuring Inheritance Patterns in Object Oriented Systems. ArODES (HES-so (Https://Www.hes-So.ch/)). https://doi.org/10.1145/2723742.2723755

- Richards, M. (2015). Software Architecture Patterns. 146.201. http://hdl.handle.net/1/5665

- Tuaycharoen, N., Prodpran, V., & Srithong, B. (2018, May 1). ClassSchedule: A web-based application for school class scheduling with real-time lazy loading. IEEE Xplore. https://doi.org/10.1109/ICBIR.2018.8391194