Architecture Document

SWEN90007

Music Events System

Team: GGBond

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

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

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. User Story View.

Use Case 1: Manage Account

Actors

All Users

Basic Flow

All users log into their Music System account by email or account name and password. After logging in, they browse the Account page where they can manage their account, including change email, change phone number, reset password and delete account.

Use Case 2: Change Email

Actors

All Users

Basic Flow

All users navigate to the Account page, they click on the Change Email button, and then they go to the new window to enter the new email account and confirm the email address. After they successfully change the email address, the new one is displayed on the Account page.

Use Case 3: Change Phone Number

Actors

All Users

Basic Flow

All users navigate to the Account page, they click the Change Phone Number button, then they go to the new window to enter the new phone number and confirm the phone number by receiving the message from the new phone number. After they successfully change the phone number, the new phone number is displayed on the Account page.

Use Case 4: Reset Password

Actors

All Users

Basic Flow

On the Account page, they choose the Reset Password option, then they enter the new password and confirm it through a new window. They also need to perform security checks via email or phone number. After successfully resetting the password, the system automatically logs them out, and they need to log in with the new password.

Use Case 5: Delete Account

Actors

All Users

Basic Flow

On the Account page, users choose the Delete Account option, then a new window pops up asking them “Are you sure that you want to delete this account?”, if users choose “Yes”, their account

information will be cleared and the system will automatically log them out. If they choose “No”, they will stay on the Account page without any changes.

Use Case 6: View Event

Actors

All Users

Basic Flow

All users log into the system and navigate to the Home page where they can view all events across Australia. Then they click on one event which they are interested in, they will go to another page which includes the details of this event, such as date, time, location, and associated event planners.

Use Case 7: View users

Actors

Administrator

Basic Flow

The administrator logs into the Music Event System. Then the admin views a list of users on the User Management page which contains all users of the system. After the administrator clicks on one user, the information of the user shows up on the new page, including user name, email, phone number and other information associated with the user.

Use Case 8: View Customers Details

Actors

Administrator

Basic Flow

In the User Management page, the administrator selects one customer and the detailed information of the customer is displayed on the new page. It shows the customer’s name, email, phone number and ticket purchase history.

Use Case 9: Manage venue

Actors

Administrator

Basic Flow

The administrator logs into the system and navigates to the Venue Management page, where it displays all the venues that were created before. Then the administrator clicks on one venue and the detailed information of the venue shows up in a new window which includes the name, sections and the capacity of each section. On the Venue Management page, the administrator also has Create, Delete and Modify Venues options.

Use Case 10: Create venue

Actors

Administrator

Basic Flow

On the Venue Management page, the admin chooses the "Venue Creation" option, then the admin inputs the necessary information such as venue name, sections (mosh, standing, seated, VIP, etc.), and the capacity of each section into a form. After confirming the details, the admin submits the form, creating a new venue in the system.

Use Case 11: Modify Venue

Actors

Administrator

Basic Flow

On the Venue Management page, the administrator selects the “Modify Venue” option next to the venue. Once the venue is selected, editable venue information appears. The administrator then adjusts the details and confirms the changes and the venue information is updated.

Use Case 12: Delete venue

Actors

Administrator

Basic Flow

The Administrator logs into the system and goes to the Venue Management section. The admin selects the venue he wants to delete and clicks on the "Delete Venue" option next to the venue. After confirming the deletion, the venue is removed from the system.

Use Case 13: View Ticket Details

Actors

Administrator

Basic Flow

The administrator logs into the system using their unique administrator credentials. Once logged in, the administrator is presented with a dashboard that provides an overview of various events. Then, the administrator navigates to one of the events. On this page, there’s a section called “Ticket Details”. The administrator clicks on the “Ticket Details” tab. Here the admin sees a list of all customers who purchased tickets for this event, ticket types, quantities and price.

Use Case 14: Create Account

Actors

Normal User

Basic Flow

Normal users open the Music Event system, and on the Login page, they choose the “Create Account” option at the bottom of the page. Then they enter their personal information including, user name, email, contact number and set password for this system. After confirming this information they enter, they log into the system with their credentials.

Use Case 15: Upload Identity

Actors

Normal User

Basic Flow

When normal users create the account, there is an option for event planners uploading their identity on the page that they input their information. Event planners click on the "Upload Identity" option, they choose the picture of their work card and upload. After confirming the identity, event planners login with their name and password and have rights to manage events associated with them.

Use Case 16: Manage Event

Actors

Event Planner

Basic Flow

Event planners log into their accounts, after successfully login, they navigate to the My Event page where they can manage their event. On that page they have many options, including create, modify, delete and share events.

Use Case 17: Create Event

Actors

Event Planner

Basic Flow

Event Planners, after logging into the system, go to the My Event page. They select the "Create Venue" option, Then they set the date, and time, and define the ticket prices for each section of the venue on a new page. They confirm the details and submit the form, after that, a new event shows on the page which shows a list of events.

Use Case 18: Modify Event

Actors

Event Planner

Basic Flow

Event Planners login and navigate to My Event page. They select an event that they want to modify. They can change the date, time, ticket prices, or even cancel the event entirely. After making the necessary modifications, they confirm the changes and update the event.

Use Case 19: Change Time

Actors

Event Planner

Basic Flow

On the My Event page, event planners choose one event that they want to change the time. They click on the time section and then they choose a different time throughout a calendar. After confirming the new time, the event information is updated.

Use Case 20: Change Price of Tickets

Actors

Event Planner

Basic Flow

On the My Event page, event planners choose one event that they want to change the price of the tickets. They view the different sections (mosh, standing, seated, VIP, etc.) of the tickets and click one section that they need to adjust the price of. After successfully adjusting the price, the new price is shown on the event information page.

Use Case 21: Delete Event

Actors

Event Planner

Basic Flow

On the My Event page, event planners choose one event that they want to remove from the page. They click on the event and choose the “Delete option” on the right top of the page. Then they need to confirm the deletion. After completing the deletion, the event disappears on the My Event page.

Use Case 22: Share Event

Actors

Event Planner

Basic Flow

Event Planners log into the system and select an event from their events list. They click on the "Share Event" option, and the system provides sharing options, such as sharing a link via email or social media. The Event Planner selects the preferred method and shares the event.

Use Case 23: Manage Orders

Actors

Customer

Basic Flow

Customers log into the system and navigate to the Home page. They can view the events and select the tickles that they want. Based on the selected tickets, customers can create an order, the order contains the details about the tickets and customer. Then customers can manage their orders in their My Order page, including view order and cancel order.

Use Case 24: Book Tickets

Actors

Customer

Basic Flow

After logging into the system and selecting an event, customers choose their preferred section at the venue and specify the number of tickets they want to purchase. They proceed to checkout, where they confirm the details of their order and make payment. Once the payment is confirmed, the tickets are booked.

Use Case 25: View Order

Customer

Basic Flow

After logging into the system, the customer navigates to their order history. Here, they can view details of their previous and current orders, including event details, ticket type, number of tickets purchased, total cost, and the status of each order.

Use Case 26: Cancel Order

Actors

Customer

Basic Flow

Customers log into the system and navigate to their reservations. They select the reservation they want to cancel and click on "Cancel Reservation". After confirming the cancellation, the system processes the request, and handles refunds or credits as appropriate.

Use Case 27: Search Events

Actors

Customer

Basic Flow

Customers log into the system. They can use the search function to search music shows by name, or browse through a calendar view of all upcoming shows within the next several months.

Use Case 28: Search By Name

Actors

Customer

Basic Flow

Customers log into the system. They want to search for music shows by name. Enter the name of the music exhibition in the search bar and click the search button, then the eligible music exhibition will be displayed.

Use Case 29: Search By Time

Actors

Customer

Basic Flow

Customers log into the system. They want to search for music shows by time. Enter the time of the music exhibition in the search bar and click the search button, then the eligible music exhibition will be displayed.

3. Development View

3.1 Architectural Patterns

3.1.1 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

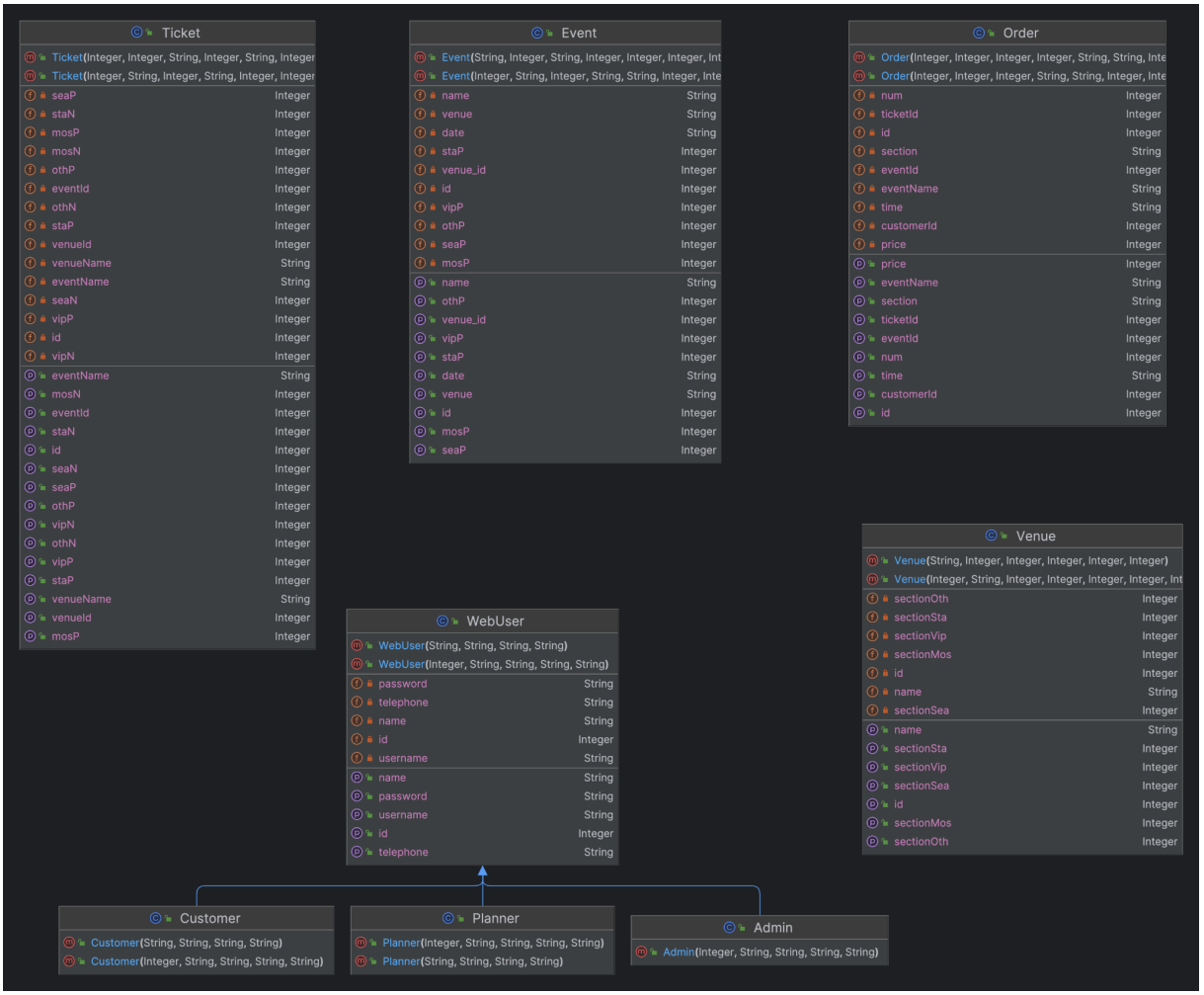


Figure 6.1.1 *Class Diagram*

3.1.2 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

3.1.3 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

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.

Steps:

Steps:

Customer Chooses to Purchase Tickets:

- The customer initiates the ticket purchase process, triggering the `Purchase.list` method.

Fetching Event Information:

- The `list` method calls the `Event.list` method to obtain information about the music event.
- At this stage, `EventMapper` is also invoked to query the `Event` table in the database.

Fetching Ticket Information:

- The `list` method continues by calling another method (possibly `TicketService.list()`) to fetch the available ticket quantities.
- This step also involves invoking `TicketMapper` to query the `Tickets` table in the database.

Creating the Purchase Object:
- The `list` method compiles the fetched event and ticket information into a `Purchase` object.
- This `Purchase` object serves as a Unit of Work, encapsulating all the changes that need to be made.

Generating Order Attributes:
- The `Purchase` object is used to generate new `Order` attributes, such as the order ID, customer ID, and total price.

Committing Changes via Unit of Work:
- The `UnitOfWork` class is invoked to manage the transaction.
- It takes the `Purchase` object and commits the changes to the `Tickets` and `Orders` tables.
- This involves calling the respective mappers (`TicketMapper` and `OrderMapper`) to execute the database operations.

Transaction Completion:
- Once all operations are successfully executed, the `UnitOfWork` commits the transaction, ensuring data integrity.

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.

Steps:

Customer Clicks to Purchase:
- The customer clicks the "Purchase" button, triggering the `Purchase.save` method.

Fetching Customer Information:
- The `save` method calls `Customer.search` to obtain the customer's information.
- This step may involve querying the `Customer` table in the database.

Fetching Ticket and Event Information:
- The `save` method also fetches information from the `Tickets` and `Event` tables.
- This is done by calling methods that interact with `TicketMapper` and `EventMapper` to query the respective tables in the database.

Compiling Purchase Data:
- The `save` method compiles the fetched customer, ticket, and event information.
- It also includes the section and number of tickets specified by the customer.

Creating the Order Object:

- The compiled information is used to generate a new `Order` object with attributes like order ID, customer ID, total price, section, and number of tickets.

Saving the Order:

- The `Order.save` method is called to save the new order data to the `Orders` table.
- This involves invoking the `OrderMapper` to execute the database operation.

Committing Changes via Unit of Work:

- The `UnitOfWork` class is invoked to manage this transaction.
- It takes the newly created `Order` object and commits the changes to the `Tickets` and `Orders` tables.

Transaction Completion:

- Once all operations are successfully executed, the `UnitOfWork` commits the transaction, ensuring data integrity.

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.

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

Participants:

Planner: The user who interacts with the system.

EventServlet: The servlet that handles incoming requests.

list() Method: The method responsible for listing events.

lazyLoad() Class: The class that implements lazy loading.

EventMapper: The data mapper for event objects.

Database: The database where event data is stored.

Scenario 1: Initial List View

Step 1: The Planner initiates a request to view a list of events by calling requestViewEvents() on the EventServlet.

Step 2: The EventServlet calls the list() method to fetch the list of events.

Step 3: The list() method invokes the lazyGetPartial() method of the lazyLoad() class to fetch partial attributes of the events.

Step 4: The lazyLoad() class calls getPartialEventAttributes() on the EventMapper.

Step 5: The EventMapper queries the Database for partial data using queryPartialData().

Step 6: The Database returns the partial data to the EventMapper.

Step 7: The EventMapper returns a partial ghost object to the lazyLoad() class.

Step 8: The lazyLoad() class returns the partial ghost object to the list() method.

Step 9: The list() method returns the list of events to the EventServlet.

Step 10: The EventServlet displays the list of events to the Planner.

Scenario 2: Clicking an Event for Details

Step 1: The Planner clicks on an event, triggering the clickEvent() method in the EventServlet.

Step 2: The EventServlet calls the lazyGetComplete() method of the lazyLoad() class to fetch all attributes of the clicked event.

Step 3: The lazyLoad() class calls getCompleteEventAttributes() on the EventMapper.

Step 4: The EventMapper queries the Database for complete data using queryCompleteData().

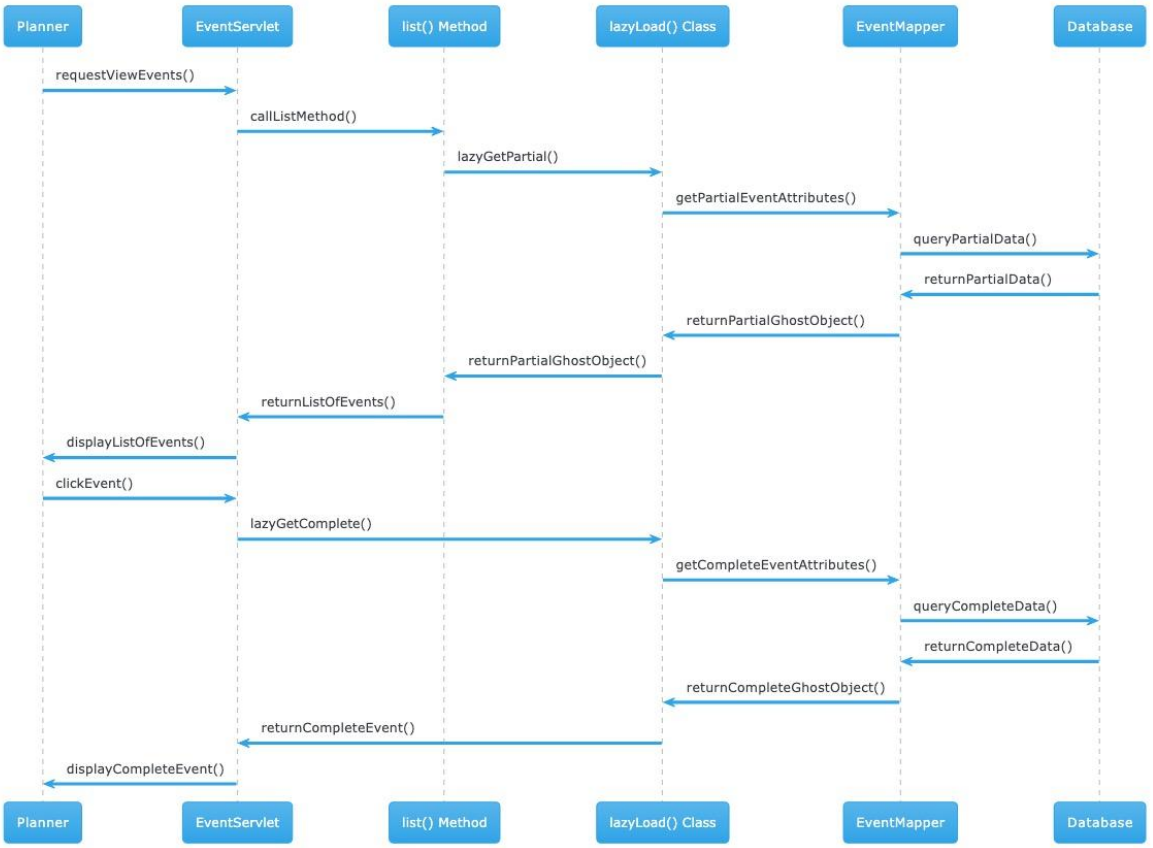
Step 5: The Database returns the complete data to the EventMapper.

Step 6: The EventMapper returns a complete ghost object to the lazyLoad() class.

Step 7: The lazyLoad() class returns the complete ghost object to the EventServlet.

Step 8: The EventServlet displays the complete event details to the Planner.

This sequence diagram effectively captures the lazy loading mechanism in the Music Event System, optimizing performance by only loading necessary data.



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.

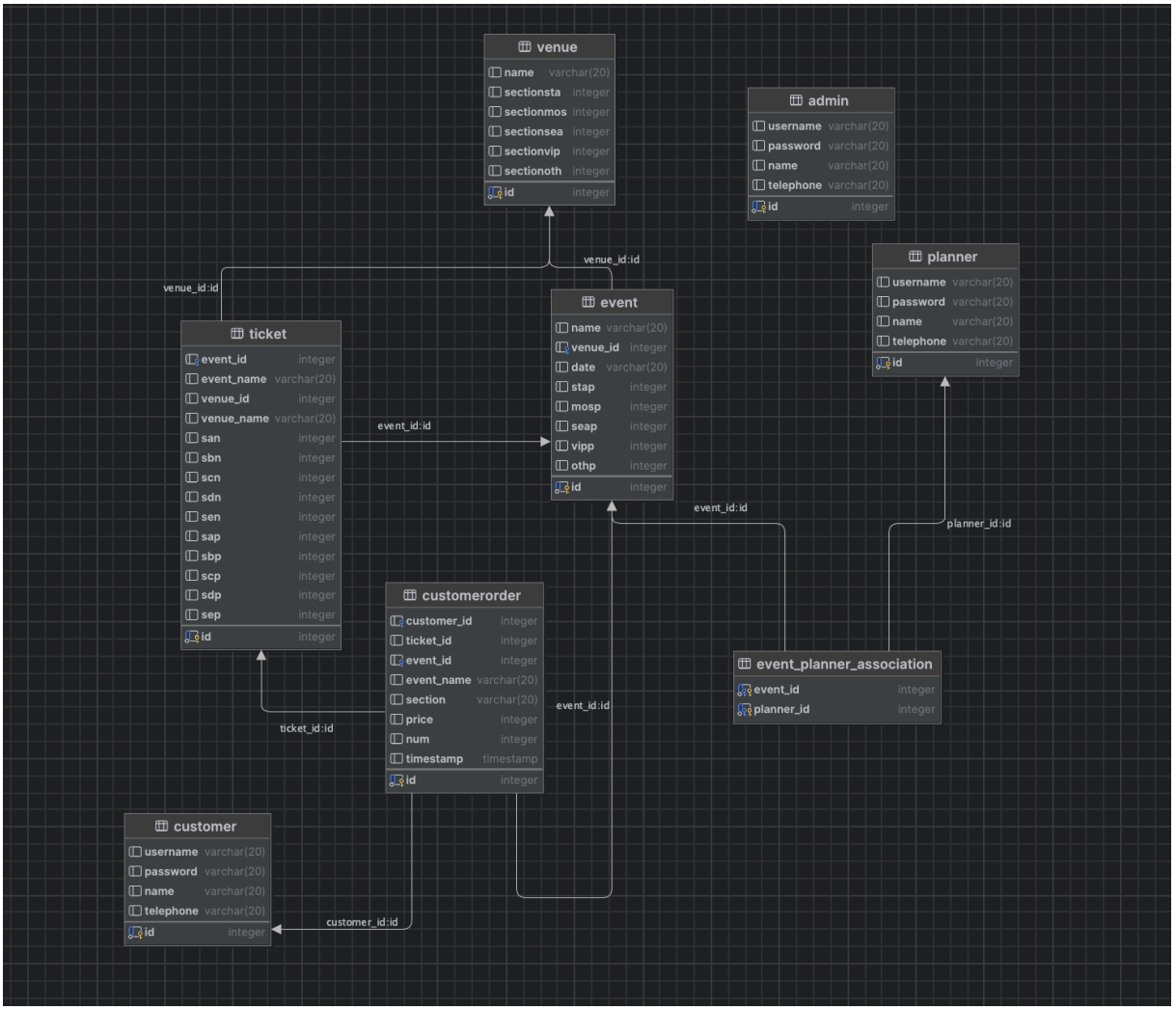


Figure 6.1.5 *Entity relationship diagram*

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.

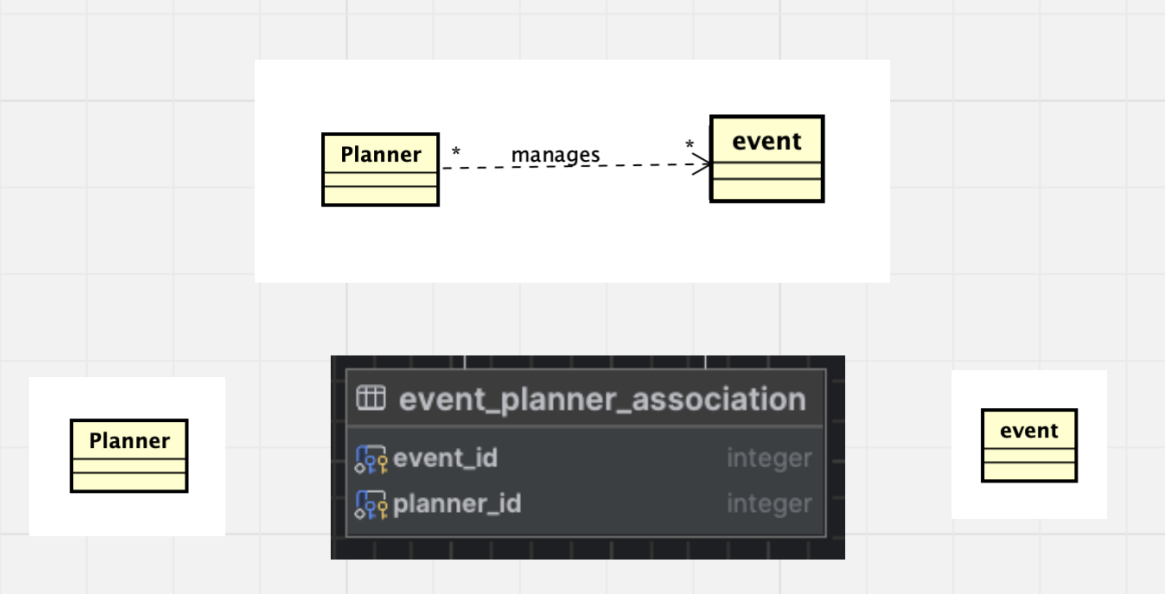


Figure 6.1.7 *Association table example*

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.

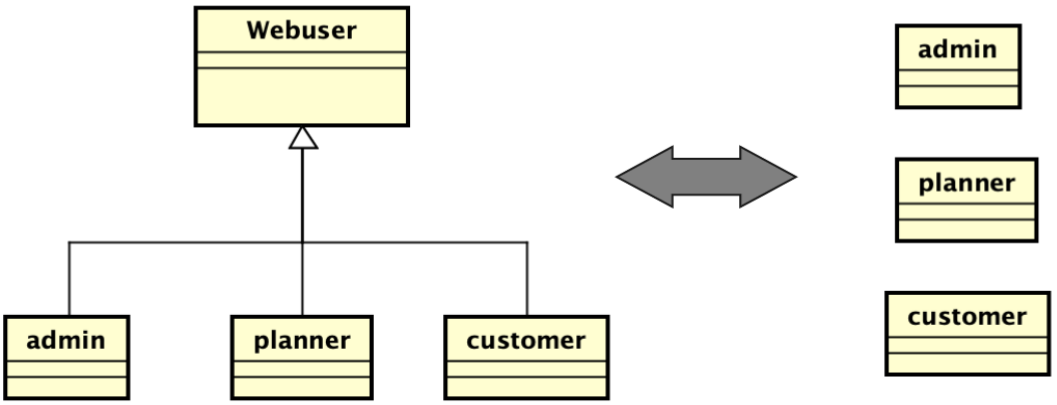


Figure 6.1.9 *Concrete Table Inheritance*

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

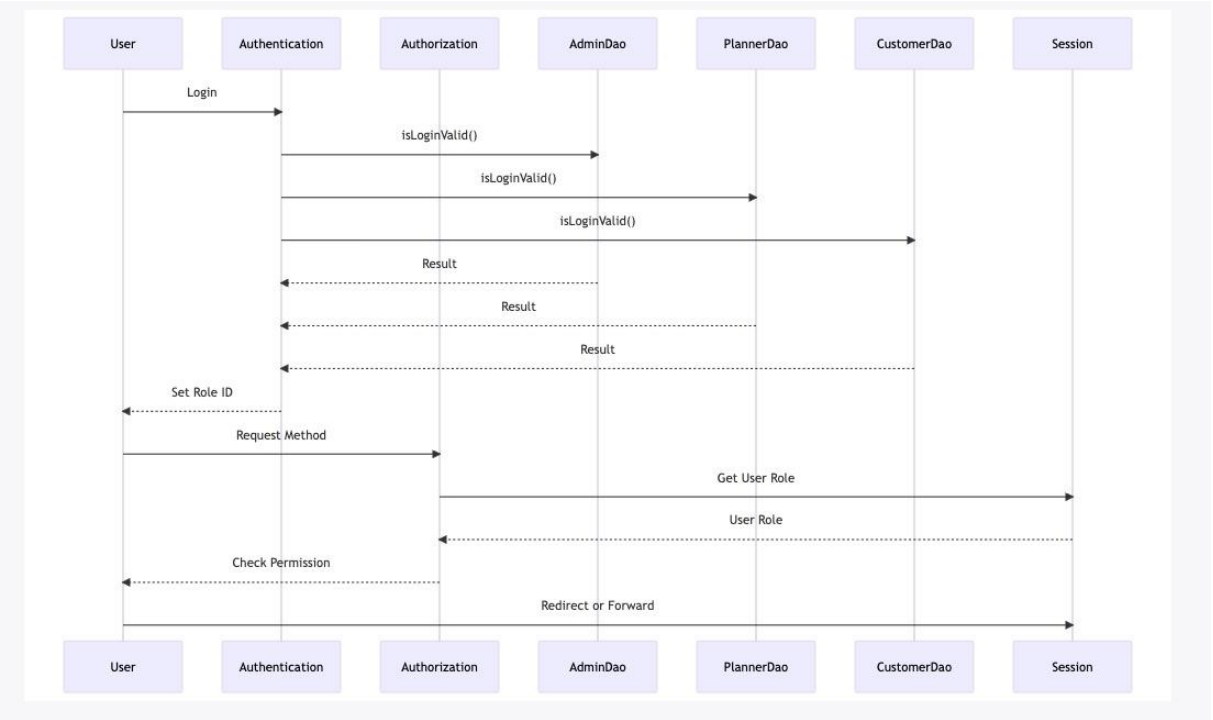
Authentication Process:

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.



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.

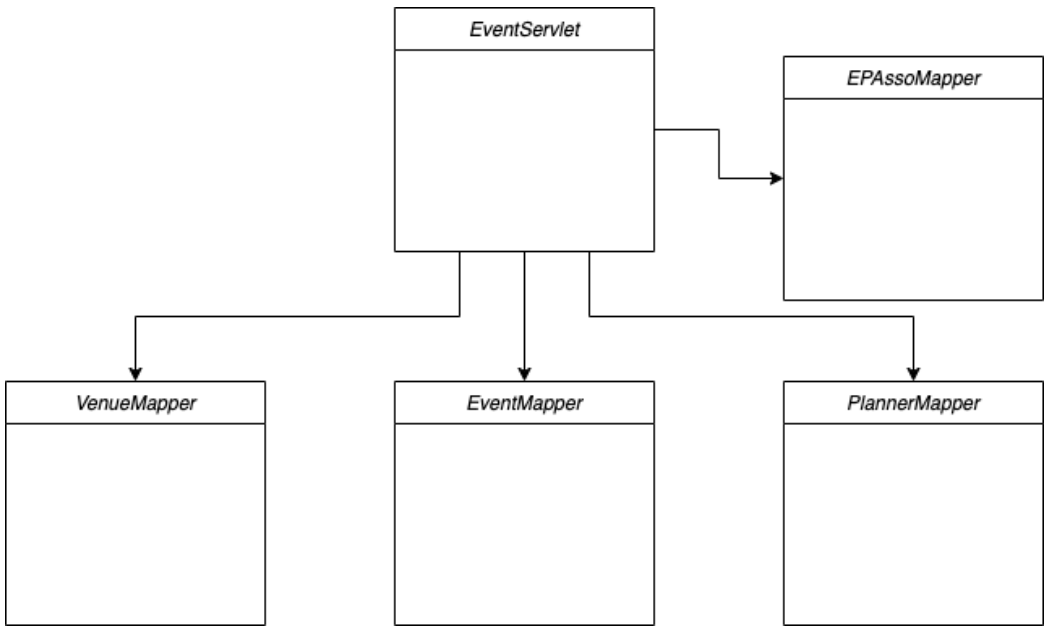
Controller

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.



6.2. Source Code Directories Structure

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

- src
  - main
    - java
      - com.example.musiceventsystem
        - controller: Contains the main controllers for handling web requests.
          - api: Public classes and interfaces for the controller component.
          - impl: Private classes and implementations for the controller component.
        - service: Contains service classes that handle business logic.
          - api: Public classes and interfaces for the service component.
          - impl: Private classes and implementations for the service component.
        - repository: Contains classes for data access and database operations.
          - api: Public classes and interfaces for the repository component.
          - impl: Private classes and implementations for the repository component.
        - model: Contains the main data models or entities of the application.
          - api: Public classes and interfaces for the model component.
          - impl: Private classes and implementations for the model component.
    - webapp: Contains web resources like CSS, JS, images, and other assets.
      - WEB-INF: Configuration files and resources for the web application.

For Java applications, the packaging of components is undertaken as follows:

Components in the MusicEventSystem project follow the naming convention org.unimelb.cis.musiceventsystem.<componentname>. Within each component package, there are two additional packages: api and impl. The api package groups the public classes of the component, while the impl package contains the private classes of the component.

6.3. Libraries and *Frameworks*This section describes libraries and frameworks used by MusicEventSystem.

6.4. Development Environment

Library / *Framework*	Reason	Version	Environment
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
spring-security-web	CIS Standard: Provides functionalities for web security in Spring applications.	(Managed by spring-security-bom)	All

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 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.

4. Physical View

NA

5. Scenarios

NA

6. References

< This section must provide a list of all related documents.>