SWEN90007 Software Design and Architecture Part 2
Software Architecture Design Report

Team: Green Day

GitHub Release Tag: SWEN90007_2022_Part2_GreenDay

Quanchi Chen	quanchic@student.unimelb.edu.au	quanchic
Yijie Xie	yijxie@student.unimelb.edu.au	yijxie
Wenxuan Xie	wexie2@student.unimelb.edu.au	wexie2
Jingning Qian	jingningq@student.unimelb.edu.au	jingningq

Table of Contents

This document specifies the design and architecture of the music events system (which is referred to as MES in the rest of this report) developed by the team Green Day, explaining its logical view, the underlying database schema, and the employed enterprise application architecture patterns. This document also serves as the common understanding among all the team members and should help new developers quickly understand the design rationale behind the software system.

Logical View

The first section describes the logical view of MES by discussing the software layers and the class diagram. We structure the system by breaking it into three layers, i.e., presentation, domain, and data source. Table 1 lists their primary responsibilities and components.

Layer	**Responsibility**	**Components**
Presentation	Provide user interaction	Reusable React components
Domain	Implement business logic	Controllers (i.e., Servlets) accepting client requests, invoking services to process these requests, and sending the responses back; Services implementing business rules and logic; Domain objects representing meaningful domain models
Data Source	Data transfer between the domain layer and the underlying data store	Data mappers

Table 1 - The responsibilities and components of the three software layers

To increase the readability, we break the class diagram into four sub-diagrams, each dedicated to explaining a particular software layer.

Figure 1 depicts the class diagram of the controllers layer. The controllers are responsible for accepting client requests from the React application, invoking corresponding services to process these requests, and returning the responses to the client.

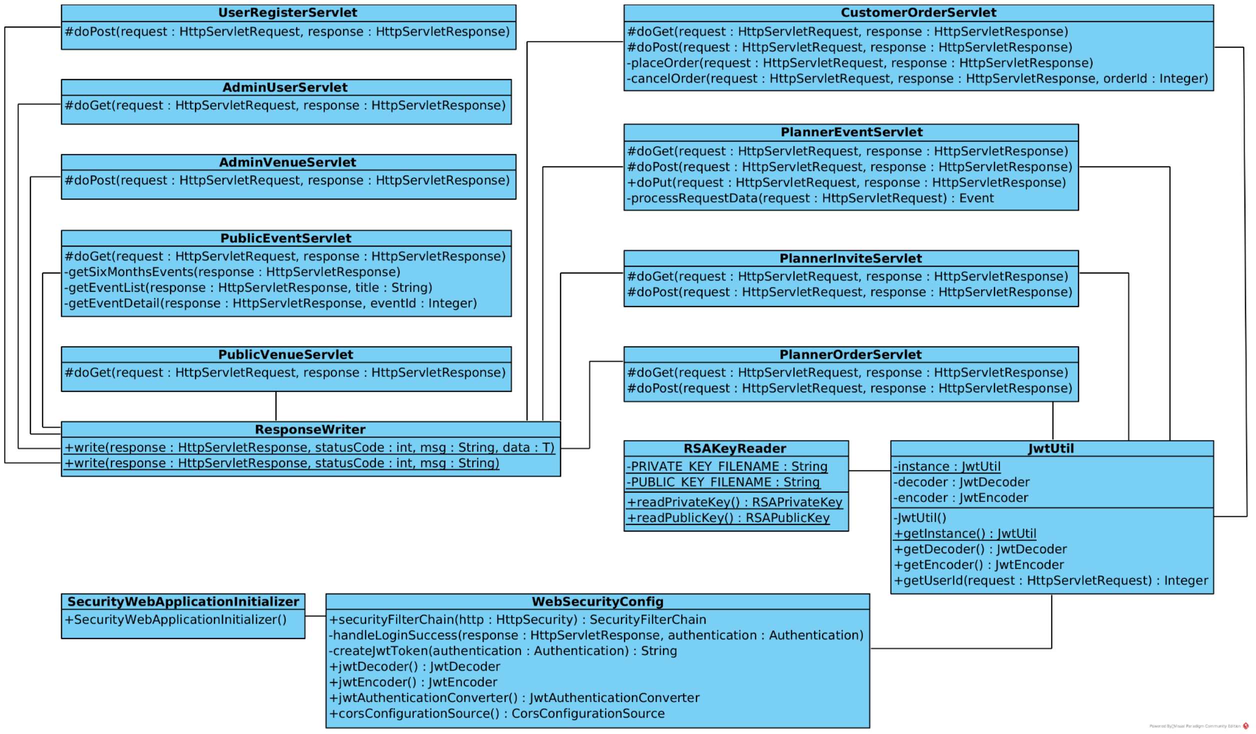
UserRegisterServlet
#doPost(request : HttpServletRequest, response : HttpServletResponse)
AdminUserServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
AdminVenueServlet
#doPost(request : HttpServletRequest, response : HttpServletResponse)
PublicEventServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
-getSixMonthsEvents(response : HttpServletResponse)
-getEventList(response : HttpServletResponse, title : String)
-getEventDetail(response : HttpServletResponse, eventId : Integer)
PublicVenueServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
ResponseWriter
+write(response : HttpServletResponse, statusCode : int, msg : String, data : T)
+write(response : HttpServletResponse, statusCode : int, msg : String)
CustomerOrderServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
#doPost(request : HttpServletRequest, response : HttpServletResponse)
-placeOrder(request : HttpServletRequest, response : HttpServletResponse)
-cancelOrder(request : HttpServletRequest, response : HttpServletResponse, orderId : Integer)
PlannerEventServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
#doPost(request : HttpServletRequest, response : HttpServletResponse)
+doPut(request : HttpServletRequest, response : HttpServletResponse)
-processRequestData(request : HttpServletRequest) : Event
PlannerInviteServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
#doPost(request : HttpServletRequest, response : HttpServletResponse)
PlannerOrderServlet
#doGet(request : HttpServletRequest, response : HttpServletResponse)
#doPost(request : HttpServletRequest, response : HttpServletResponse)
RSAKeyReader
-PRIVATE_KEY_FILENAME : String
-PUBLIC_KEY_FILENAME : String
+readPrivateKey() : RSAPrivateKey
+readPublicKey() : RSAPublicKey
JwtUtil
-instance : JwtUtil
-decoder : JwtDecoder
-encoder : JwtEncoder
-JwtUtil()
+getInstance() : JwtUtil
+getDecoder() : JwtDecoder
+getEncoder() : JwtEncoder
+getUserId(request : HttpServletRequest) : Integer
SecurityWebApplicationInitializer
+SecurityWebApplicationInitializer()
WebSecurityConfig
+securityFilterChain(http : HttpSecurity) : SecurityFilterChain
-handleLoginSuccess(response : HttpServletResponse, authentication : Authentication)
-createJwtToken(authentication : Authentication) : String
+jwtDecoder() : JwtDecoder
+jwtEncoder() : JwtEncoder
+jwtAuthenticationConverter() : JwtAuthenticationConverter
+corsConfigurationSource() : CorsConfigurationSource

Figure 1 - Class Diagram (Controllers/Servlets)

Figure 2 depicts the class diagram of the services layer. The services belong to the domain layer and are responsible for implementing the core business logic and rules. Separating the data and operations in the domain model pattern improves loose coupling. The services are also responsible for registering any changed objects to UnitOfWork instances, i.e., caller registration.

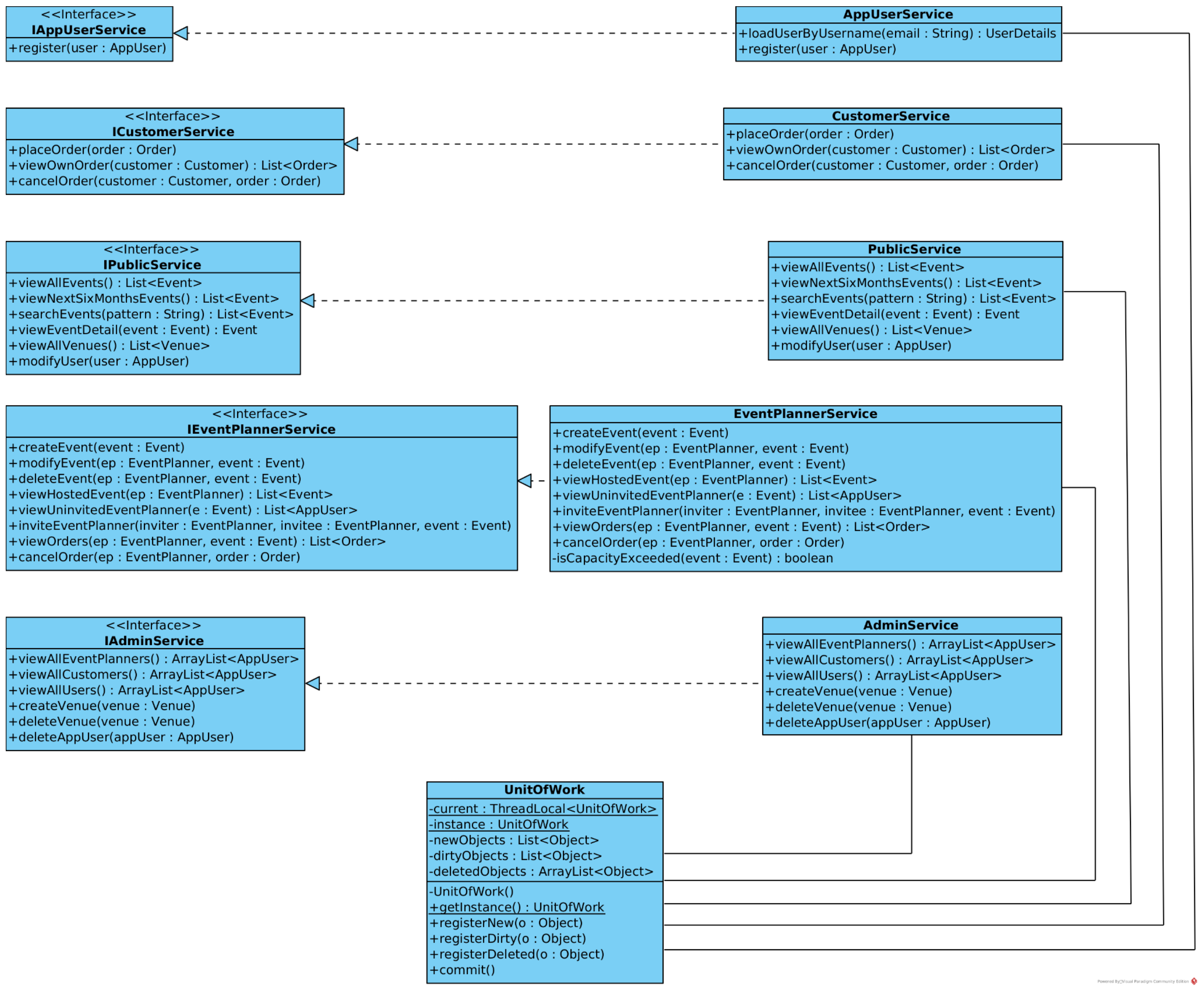
<<Interface>>
IAppUserService
+register(user : AppUser)
AppUserService
+loadUserByUsername(email : String) : UserDetails
+register(user : AppUser)
<<Interface>>
ICustomerService
+placeOrder(order : Order)
+viewOwnOrder(customer : Customer) : List<Order>
+cancelOrder(customer : Customer, order : Order)
CustomerService
+placeOrder(order : Order)
+viewOwnOrder(customer : Customer) : List<Order>
+cancelOrder(customer : Customer, order : Order)
<<Interface>>
IPublicService
+viewAllEvents() : List<Event>
+viewNextSixMonthsEvents() : List<Event>
+searchEvents(pattern : String) : List<Event>
+viewEventDetail(event : Event) : Event
+viewAllVenues() : List<Venue>
+modifyUser(user : AppUser)
PublicService
+viewAllEvents() : List<Event>
+viewNextSixMonthsEvents() : List<Event>
+searchEvents(pattern : String) : List<Event>
+viewEventDetail(event : Event) : Event
+viewAllVenues() : List<Venue>
+modifyUser(user : AppUser)
<<Interface>>
IEventPlannerService
+createEvent(event : Event)
+modifyEvent(ep : EventPlanner, event : Event)
+deleteEvent(ep : EventPlanner, event : Event)
+viewHostedEvent(ep : EventPlanner) : List<Event>
+viewUninvitedEventPlanner(e : Event) : List<AppUser>
+inviteEventPlanner(inviter : EventPlanner, invitee : EventPlanner, event : Event)
+viewOrders(ep : EventPlanner, event : Event) : List<Order>
+cancelOrder(ep : EventPlanner, order : Order)
EventPlannerService
+createEvent(event : Event)
+modifyEvent(ep : EventPlanner, event : Event)
+deleteEvent(ep : EventPlanner, event : Event)
+viewHostedEvent(ep : EventPlanner) : List<Event>
+viewUninvitedEventPlanner(e : Event) : List<AppUser>
+inviteEventPlanner(inviter : EventPlanner, invitee : EventPlanner, event : Event)
+viewOrders(ep : EventPlanner, event : Event) : List<Order>
+cancelOrder(ep : EventPlanner, order : Order)
-isCapacityExceeded(event : Event) : boolean
<<Interface>>
IAdminService
+viewAllEventPlanners() : ArrayList<AppUser>
+viewAllCustomers() : ArrayList<AppUser>
+viewAllUsers() : ArrayList<AppUser>
+createVenue(venue : Venue)
+deleteVenue(venue : Venue)
+deleteAppUser(appUser : AppUser)
AdminService
+viewAllEventPlanners() : ArrayList<AppUser>
+viewAllCustomers() : ArrayList<AppUser>
+viewAllUsers() : ArrayList<AppUser>
+createVenue(venue : Venue)
+deleteVenue(venue : Venue)
+deleteAppUser(appUser : AppUser)
UnitOfWork
-current : ThreadLocal<UnitOfWork>
-instance : UnitOfWork
-newObjects : List<Object>
-dirtyObjects : List<Object>
-deletedObjects : ArrayList<Object>
-UnitOfWork()
+getInstance() : UnitOfWork
+registerNew(o : Object)
+registerDirty(o : Object)
+registerDeleted(o : Object)
+commit()

Figure 2 - Class Diagrams (Services)

Figure 3 depicts the class diagram of the domain objects belonging to the domain layer. Each domain object is a reasonable representation of a business entity.

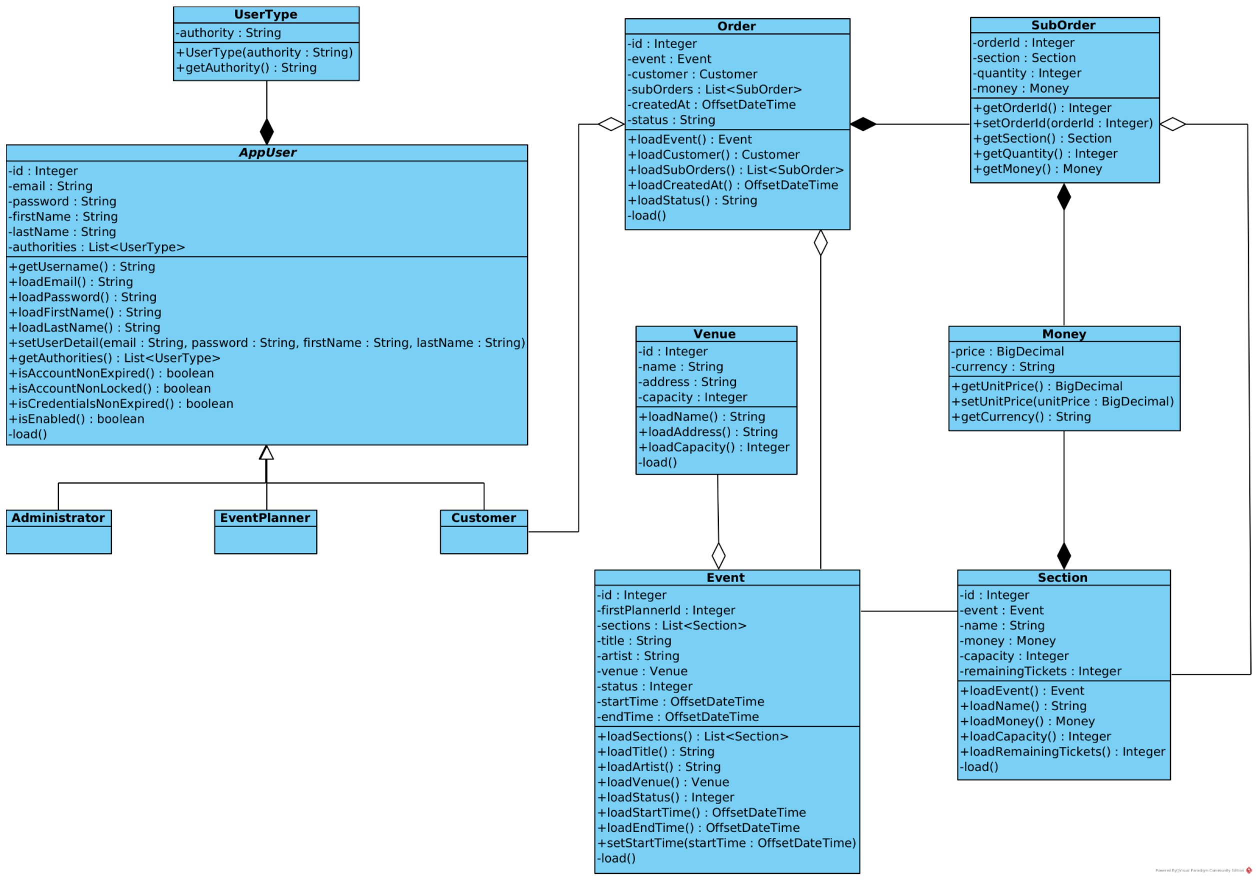

Figure 3 - Class Diagram (Domain Objects)

Figure 4 depicts the class diagram of the data mappers responsible for efficiently transferring data between the services and the database.

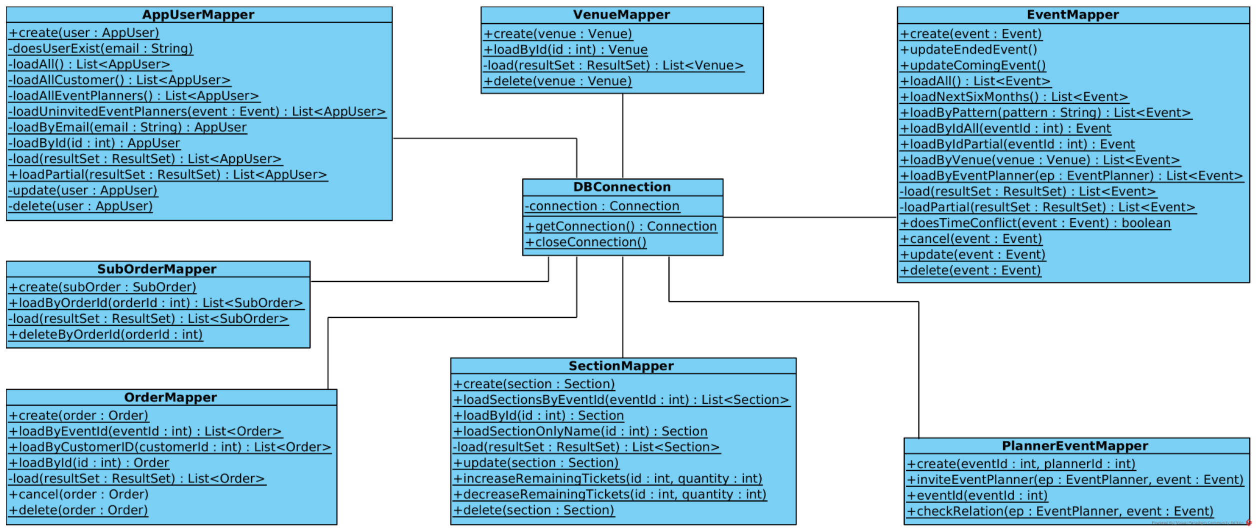
AppUserMapper
+create(user : AppUser)
-doesUserExist(email : String)
-loadAll() : List<AppUser>
-loadAllCustomer() : List<AppUser>
-loadAllEventPlanners() : List<AppUser>
-loadUninvitedEventPlanners(event : Event) : List<AppUser>
-loadByEmail(email : String) : AppUser
-loadById(id : int) : AppUser
-load(resultSet : ResultSet) : List<AppUser>
+loadPartial(resultSet : ResultSet) : List<AppUser>
-update(user : AppUser)
-delete(user : AppUser)
VenueMapper
+create(venue : Venue)
+loadById(id : int) : Venue
-load(resultSet : ResultSet) : List<Venue>
+delete(venue : Venue)
EventMapper
+create(event : Event)
+updateEndedEvent()
+updateComingEvent()
+loadAll() : List<Event>
+loadNextSixMonths() : List<Event>
+loadByPattern(pattern : String) : List<Event>
+loadByIdAll(eventId : int) : Event
+loadByIdPartial(eventId : int) : Event
+loadByVenue(venue : Venue) : List<Event>
+loadByEventPlanner(ep : EventPlanner) : List<Event>
-load(resultSet : ResultSet) : List<Event>
-loadPartial(resultSet : ResultSet) : List<Event>
+doesTimeConflict(event : Event) : boolean
+cancel(event : Event)
+update(event : Event)
+delete(event : Event)
DBConnection
-connection : Connection
+getConnection() : Connection
+closeConnection()
SubOrderMapper
+create(subOrder : SubOrder)
+loadByOrderId(orderId : int) : List<SubOrder>
-load(resultSet : ResultSet) : List<SubOrder>
+deleteByOrderId(orderId : int)
OrderMapper
+create(order : Order)
+loadByEventId(eventId : int) : List<Order>
+loadByCustomerID(customerId : int) : List<Order>
+loadById(id : int) : Order
-load(resultSet : ResultSet) : List<Order>
+cancel(order : Order)
+delete(order : Order)
SectionMapper
+create(section : Section)
+loadSectionsByEventId(eventId : int) : List<Section>
+loadById(id : int) : Section
+loadSectionOnlyName(id : int) : Section
-load(resultSet : ResultSet) : List<Section>
+update(section : Section)
+increaseRemainingTickets(id : int, quantity : int)
+decreaseRemainingTickets(id : int, quantity : int)
+delete(section : Section)
PlannerEventMapper
+create(eventId : int, plannerId : int)
+inviteEventPlanner(ep : EventPlanner, event : Event)
+eventId(eventId : int)
+checkRelation(ep : EventPlanner, event : Event)

Figure 4 - Class Diagram (Data Mappers)

Database Schema

This section describes the database schema by presenting a physical entity-relationship diagram (ERD) in Figure 5 and emphasises the assumptions mentioned in the Part 1A report. The ERD also serves as the basis for understanding the identity field, foreign key mapping, association table mapping, embedded values, and single table inheritance patterns described in the next section.

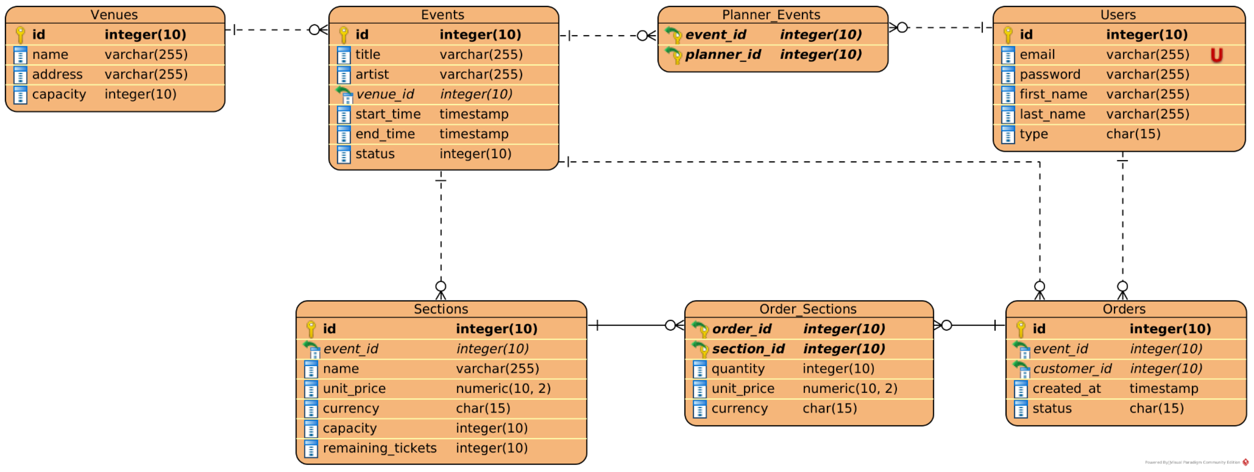

Figure 5 - Physical ERD

Below is the explanation of each of the tables and its columns.

Users

The table Users stores the data of all the active users in the system.

1. id: an ID that uniquely identifies a user
2. email: the user's email address
3. password: the user's password hash
4. first_name: the user's first name
5. last_name: the user's last name
6. type: the type of the user (either Administrator, EventPlanner, or Customer)

Venues

The table Venues stores the data associated with all the venues created by the administrator. An *assumption* is that the administrator can neither modify nor delete the existing event venues.

1. id: an ID that uniquely identifies a venue
2. name: the venue's name
3. address: the venue's address
4. capacity: the maximum number of people the venue can accommodate

Events

The table Events stores the data associated with all the events created by the event planners.

1. id: an ID that uniquely identifies a music event
2. title: the music event's title
3. artist: the music event's artist
4. venue_id: the ID of the venue in which the event will be held
5. start_time: the music event's start date and time
6. end_time: the music event's end date and time
7. status: the music event's status

Planner_Events

The association table Planner_Events maintains the many-to-many relationships among event planners and events.

1. event_id: the event ID
2. planner_id: the user ID

Sections

The table Sections stores the data associated with all the sections created by the event planners. An assumption is that all the seats in a section share the same unit price.

1. id: an ID that uniquely identifies a section
2. event_id: the ID of the event associated with the section
3. name: the section's name
4. unit_price: the unit price of the section seats
5. currency: the currency (i.e., AUD)
6. capacity: the maximum number of people this section can accommodate
7. remaining_tickets: the number of remaining tickets in the section

Orders

The table Orders stores metadata of customer orders. After an event planner cancels an event, the status of all the corresponding orders will turn to Cancelled.

1. id: the ID that uniquely identifies an order
2. event_id: the ID that uniquely identifies an event
3. customer_id: the ID of the customer placing this order
4. created_at: the created date and time
5. status: the order status (either Active or Cancelled)

Order_Sections

The association table Order_Sections maintains the many-to-many relationships among orders and sections. A customer can purchase tickets from more than one section within an order.

1. order_id: the ID of the order
2. section_id: the ID of the section
3. quantity: the number of purchased tickets within that section
4. unit_price: the unit price of the purchased tickets
5. currency: the currency

Note that this table also records the unit price and currency information since we *assume* the prices of sold tickets will remain the same even after event planners adjust the prices.

Pattern Description

This section describes all the employed patterns except for the unit of work and lazy load patterns. We will discuss these two patterns in detail in the next section.

Domain Model

We employ the domain model pattern to implement the domain logic layer. In addition, we introduce a service layer to extract all the behaviours related to the business rules and logic from the domain objects to achieve loose coupling. Thus, these domain objects only incorporate data but not operations.

According to the class diagram, MES contains the following interconnected objects: AppUser, Administrator, EventPlanner, Customer, Venue, Money, Section, Event, Order, and SubOrder. Below are brief explanations of all the domain objects.

1. An AppUser object represents a user in the system and serves as the blueprint for Administrator, EventPlanner and Customer objects. It contains the unique ID, email address, password, first name, and last name.
2. An Administrator object inherits all the public properties and operations from an AppUser object and represents the system administrator who creates venues and manages all other users.
3. An EventPlanner object inherits all the public properties and operations from an AppUser object and represents an event planner who plans and manages music events.
4. A Customer object inherits all the public properties and operations from an AppUser object and represents a customer who views music events and places orders.
5. A Venue object represents a venue created by the administrator and used by the music events. It contains the ID, the name, the address, and the total capacity.
6. A Money object represents the money of a particular section of seats. It contains the unit price and the currency.

7. A Section object represents a particular section of an event venue. It contains the unique ID, the associated event, the name, the money for the seat, the capacity, and the number of remaining seats. Note that all seats in the same section share the same unit price and concurrency.
8. An Event object represents a music event planned by one or more event planners. It contains the unique ID, the venue, the list of sections, the title, the artist, the status, the start time, and the end time.
9. An Order object represents an order placed by a customer. It contains the unique ID, the associated event, the associated customer, the list of sub-orders, the created time, and the status.
10. A SubOrder object represents a part of an order. It contains the order ID, the associated section, the number of purchased seats, and the money.

The controller layer (i.e., Servlets) will always directly invoke services necessary to complete the current business transactions.

Data Mapper

We employ the data mapper pattern to define the data access interface (i.e., the data source layer) between the domain logic layer and the underlying data store.

The motivation is that the object schema and the relational database schema are often different. For example, relational databases do not support the concept of inheritance and collections. We hope the in-memory domain objects are unaware of the relational database structure and vice versa to achieve loose coupling. Therefore, we introduce a layer of middleware that manages the data transfer between the domain model layer and the database and isolates these two components.

According to the class diagram, most domain objects have associated data mappers. The services will invoke the operations provided by these data mappers when necessary.

Identity Field

We employ the identity field pattern to establish the identity between an in-memory object and a database record by storing that record's primary key in the corresponding domain object. According to the class diagram, all the domain objects except for money hold the primary key of the related database record.

Foreign Key Mapping

We employ the foreign key mapping pattern to map an object reference in the domain model layer to a foreign key in a database table. For example, an Event object references a Venue object because an event must be held in a particular venue. Therefore, each record in the table Events must maintain a foreign key that references the ID of a venue record stored in the table Venues.

Association Table Mapping

We employ the association table mapping pattern to deal with the many-to-many relationship between two types of domain objects. For example, the many-to-many relationship exists between event planners and music events. An event planner can plan many music events, and a music event can have multiple planners. Thus, we create an association table, Planner_Events, which maintains a composite primary key consisting of an event ID and a planner ID.

Embedded Value

We employ the embedded value pattern to map Money objects to several columns of the tables Sections and Order_Sections. A Money object contains the unit price and the currency. However, creating another table named Money in the relational database is unnecessary. Thus, the tables Sections and Order_Sections have two relevant columns, i.e., unit_price and currency, to record the data in the Money objects. Note that the Order_Sections table also holds such information since we assume the unit price of all tickets in an already placed order will remain the same even after the event planners modify the prices of venue sections.

Single Table Inheritance

We employ the single table inheritance pattern to map the user inheritance hierarchy to a single column of the table Users. According to the class diagram, the Administrator, EventPlanner, and Customer objects inherit from the AppUser object. Since the relational database does not support the concept of inheritance, and we wish to minimise the expensive join operations, we use a single column type in the table Users to reflect the actual user type of each record.

Authentication & Authorisation Enforcer

MES requires strict authentication and authorisation, as all the core application functionalities except for customer registration are accessible only to logged-in users. Moreover, MES has three types of end users, each with different authorities. For example, only event planners are authorised to create music events, and only administrators can view the account information of all non-admin users.

MES adopts the authentication and authorisation enforcer patterns, and we implement them with the built-in modules of Spring Security. All the authentication and authorisation settings are specified in the securityFilterCain Java Bean of the WebSecurityConfig class, and we customise the filter layers based on the Spring Security default configuration. The filter chain is enumerated in order as follows.

```
Security filter chain: [
  DisableEncodeUrlFilter
  WebAsyncManagerIntegrationFilter
  SecurityContextHolderFilter
  HeaderWriterFilter
  CorsFilter
  LogoutFilter
  UsernamePasswordAuthenticationFilter
  BearerTokenAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  AuthorizationFilter
]
```

Specifically, we customise the `UsernamePasswordAuthenticationFilter`, `BearerTokenAuthenticationFilter` and `AuthorizationFilter` to filter the requests on our protected API endpoints. We employ JWT for authentication and authorization, and the relevant logic is customised in the `BearerTokenAuthenticationFilter`. In addition to the Spring Security filters, we need further access control for some specific requests. For example, a customer can only cancel his own order and is not supposed to cancel others' orders. To enforce this, we need to extract the user ID from the authorisation token and utilise the `AppUserMapper` to verify the authority.

In summary, the authentication and authorisation enforcer patterns make MES highly maintainable and reusable, as all settings are centralised, encapsulated in one place, and flexibly configurable. Furthermore, we can rest assured that our application security is strongly guaranteed as Spring Security is well known for its safety and robustness with the minimum configuration.

Design Rationale

The final section describes the design rationale of the unit of work and lazy load patterns.

Unit of Work

We employ the unit of work pattern to track every change to the database during a business transaction and commit all these changes by the end of that transaction. Compared to issuing potentially many small database calls during a business transaction, the unit of work pattern is more efficient in writing the changes back to the database.

According to the class diagram, the UnitOfWork class maintains three lists of the new, dirty, and deleted objects. We employ the caller registration strategy, i.e., the services register any changed domain objects to a UnitOfWork instance. The commit method contains the logic of invoking the data mappers to alter the database as a business transaction's result.

Lazy Load

We employ the lazy load pattern to load data from the database efficiently. The motivation is that an object may not need to contain all its data to fulfil a system transaction as long as it knows how to load the remaining data.

For example, according to the class diagram, an Event object maintains a list of Section objects. When a customer browses the events on the home page, the system will not display the detailed section information of each event to avoid overwhelming the customer and improve the user experience. Thus, it is reasonable to not load all the instance variables except for the IDs of the Section objects maintained by each Event object. After the customer clicks a particular event, the system will load the complete data of the Section objects of the corresponding Event object according to their unique IDs.

Another example is that the system only loads each customer's ID, first name, and last name when an event planner views all orders associated with a particular event. In addition, when a customer browses all existing orders, the Customer object referenced by all the Order objects only loads that customer's ID, first name, and last name.

We adopt the ghost strategy, i.e., the first time calling a get operation of an object will load all that object's remaining empty fields since the lazy-loaded Section and Customer objects described in the above scenarios must provide all its fields to fulfil further business transactions.

Summary

In summary, we describe a high-level design and architecture of MES in this report, including all the enterprise application patterns used. After reading through this document, a new developer should be capable of understanding all the design rationales and being fully prepared to join the development team.