# Architecture Document

**SWEN90007**

**Music Events System**

Team: SDA1

In charge of:

| Name | Student id | Email | Unimelb username | Github username |
|---|---|---|---|---|
| Chengyi Huang | 1173994 | chengyih@student.unimelb.edu.a | chengyih | Kind0fblue |
| Ziqiang Li | 1173898 | ziqiangl1@student.unimelb.edu.au | ziqiangl1 | johnnylild |
| Yuxin Liu | 1408760 | yuxliu20@student.unimelb.edu.au | YUXLIU20 | YuxinLiu-unimelb |
| Hanming Mao | 1257522 | hanmingm@student.unimelb.edu.au | HanmingM | KkkellyM |

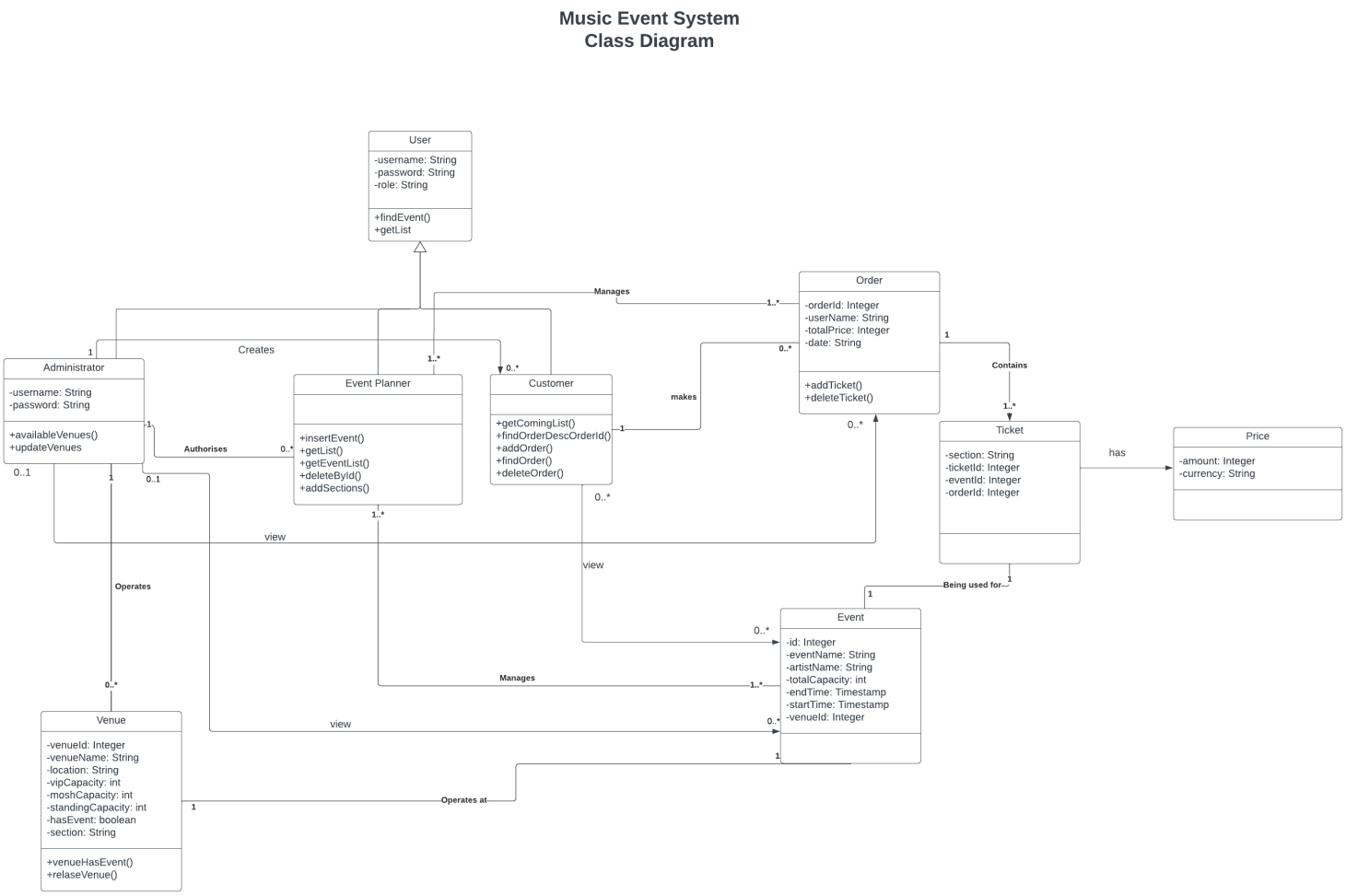

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

**Revision History**

| Date | Version | Description | Author |
|---|---|---|---|
| 15/09/2023 | 01.00 | Initial draft | Hanming Mao |
| 18/09/2023 | 02.00 | Modify Authentication | Yuxin Liu |
| 19/09/2023 | 03.00 | Add class diagram | Chengyi Huang |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

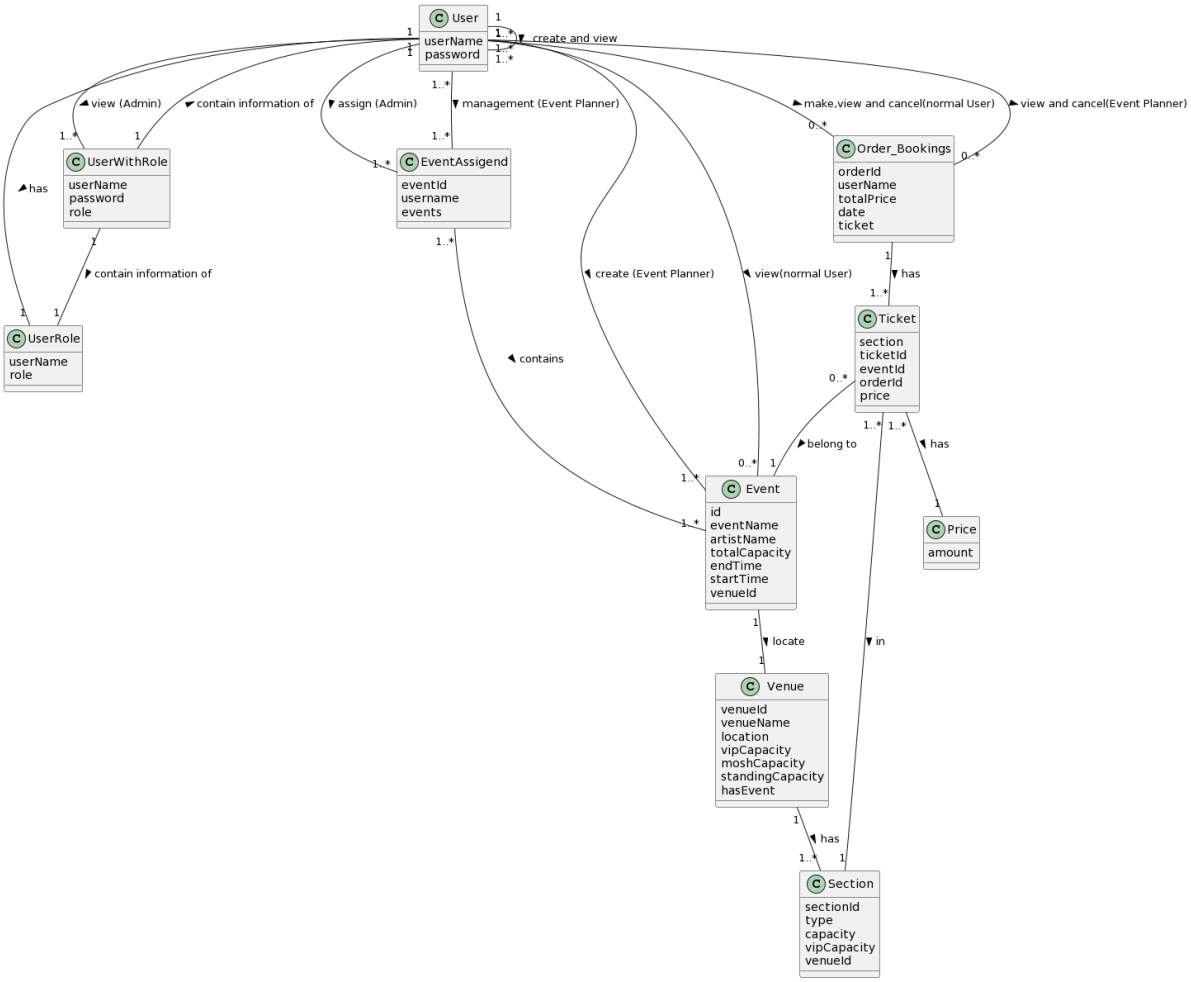
## Contents

# 1. Class diagram



# 2. Patterns used

## 2.1 Domain model




As described in the specification, we have three types of users: administrator, event planners and customers (in our role table it set as User) so as shown in the diagram, a user has a uerRole. To better develop view user function, we created a UserWithRole class which contains both user information and user role information. Each user has only one record in UserWithRole which means each record in UserRole is related to only one record in UserWithRole.

The system only has one Administrator and the Administrator can add multiple users and view all users. The Administrator can also assign multiple events to multiple Event Planners. EventAssigned class is a class contains the information of Event Planners' username and the events assigned to them. Each record contains a Event Planner's username and a event's id so in this case this class contains multiple user and multiple events information.

For Event Planners, each of them can create multiple new events and manage multiple existed events. The same event can be managed (modify, cancel and view) by multiple event planners (as assigned in the EventAssigned class) but can only be added by one Event Planner. Event Planners can view and cancel multiple orders (which are bookings in the specification) from their assigned events.

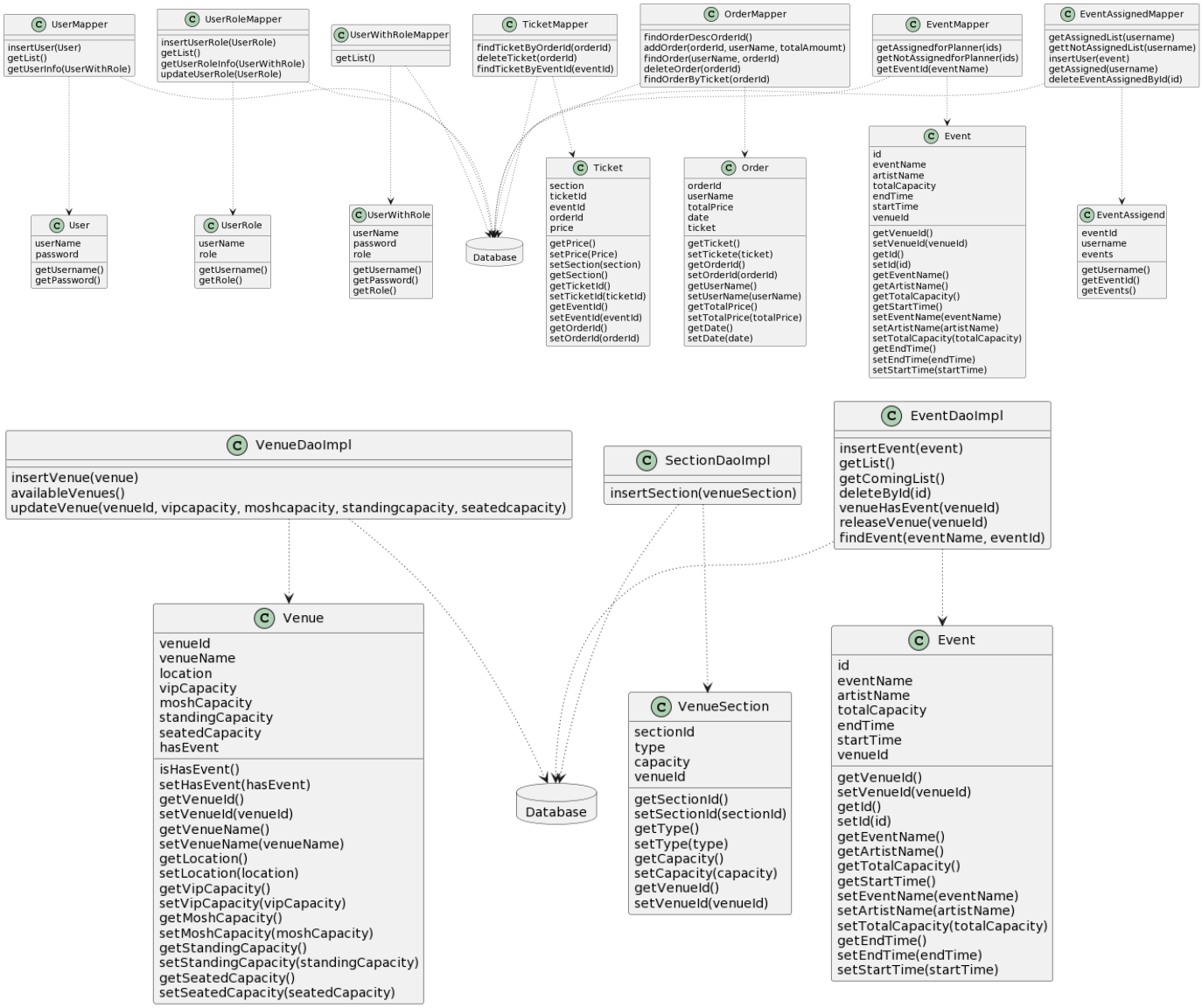
For normal Users (which are customers in the specification), they can view all existing events and make bookings for these events. They can view all their orders and cancel multiple orders as they want.

An event should located in a venue and a venue should contain multiple sections.

An order can contain multiple tickets. A ticket should belongs to an event and has its own price and section information while a section can have multiple tickets and multiple tickets can be the same price.

## 2.2 Data mapper

### 2.2.1 All mapper class diagram

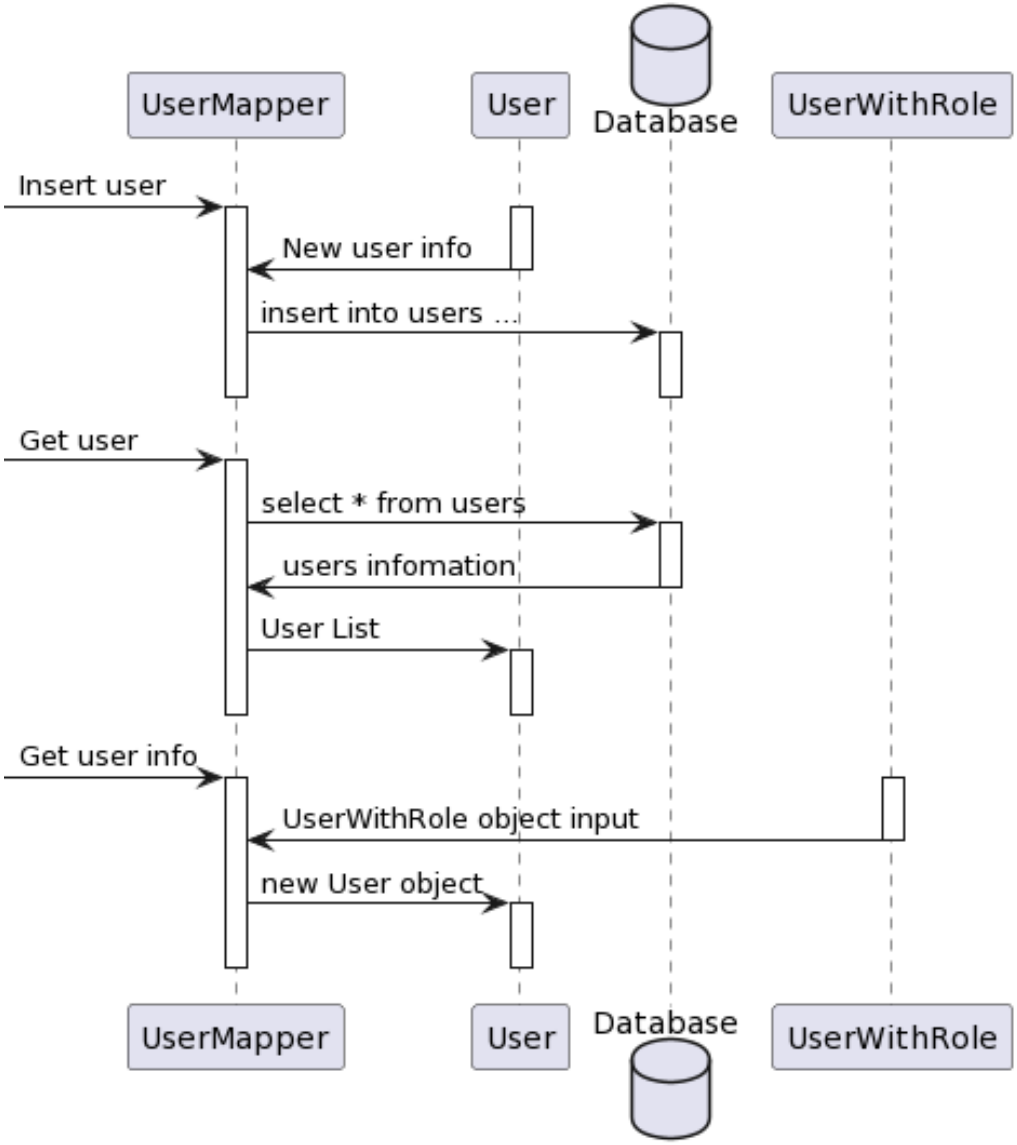



We put all our data types' get and modify methods into different database interaction layers. This means all our system's objects and in servlet functions only focus on inter java data communication and all data communication among database and system is organized by different mappers or DaoImpl.

To be clear here, we have mapper files named as …Mapper but also other database interact controller files named as …DaoImpl. This difference in name is caused by our divided developing process as we name similar thing in different way depends on personal preference. However, all the functions in …Mapper and …DaoImpl are both related to the interaction between database and Java objects. In rare cases, these files may include interaction between different objects.

Details for each class with diagrams and specifications are listed below. These diagrams and specification is only about those files named as …Mapper, as …DaoImpl is used in a different way.

### 2.2.2 UserMapper sequence and entity relationship diagram




When

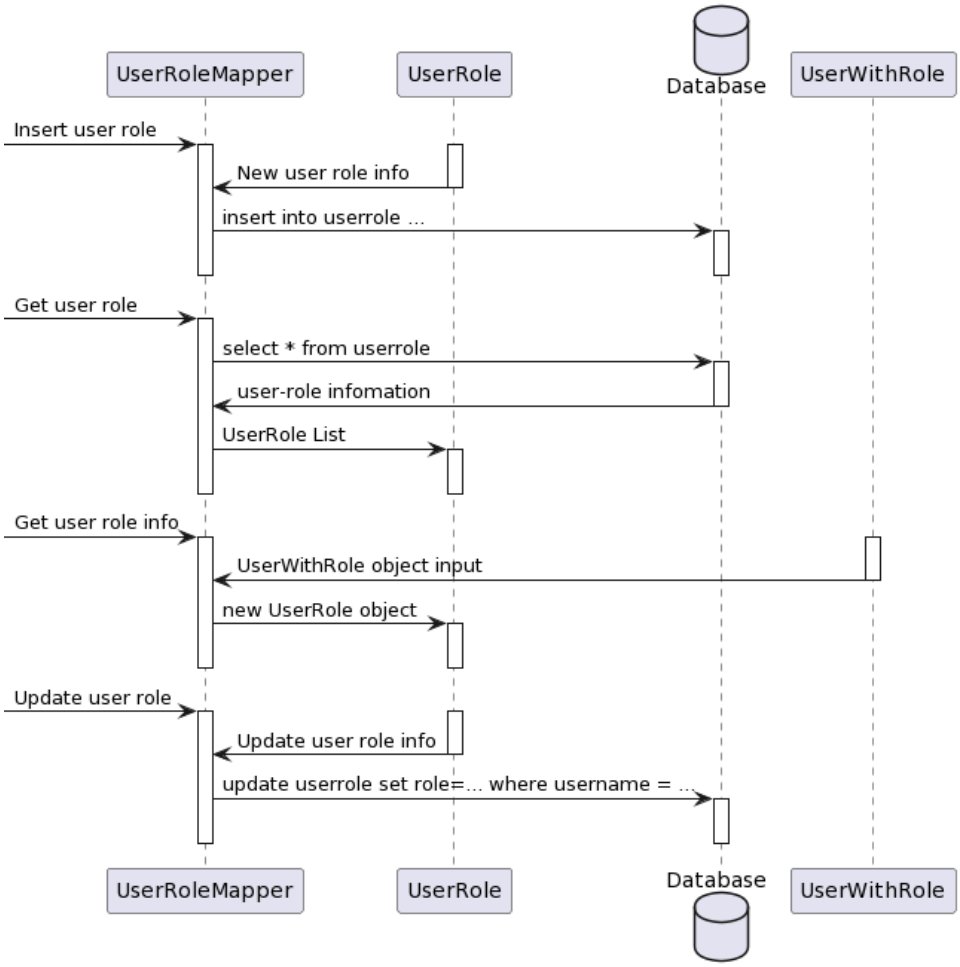
As shown here, all get and insert methods which will affect the database is included in this UserMapper. Besides, the communication between two independent objects: User and UserWithRole is also contained in this mapper.

When insert user, the mapper gets the new user information from User object and the insert it into database.

When get all users from data table, the mapper run get SQL query and output the result to multiple User object and return List<User>.

When get user information from UserWithRole object, the mapper get the user information from input UserWithRole object and output the related information to a new User object.

### 2.2.3 UserRoleMapper sequence and entity relationship diagram




As shown here, all get, update and insert methods which will affect the database is included in this UserRoleMapper. Besides, the communication between two independent objects: UserRole and UserWithRole is also contained in this mapper.

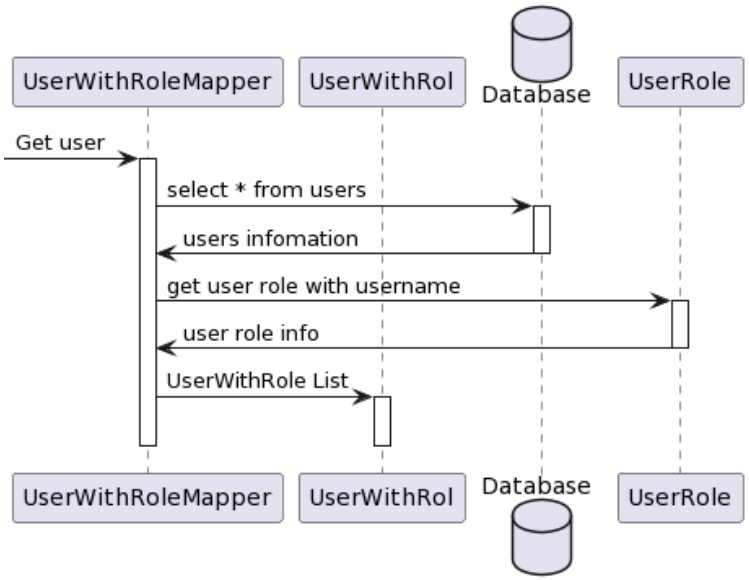
When insert new user-role connection information, the mapper gets the new connection information from UserRole object and the insert it into database.

When get all user-role connection information from data table, the mapper run get SQL query and output the result to multiple UserRole object and return List<UserRole>.

When get user information from UserWithRole object, the mapper get the user-role connection information from input UserWithRole object and output the related information to a new UserRole object.

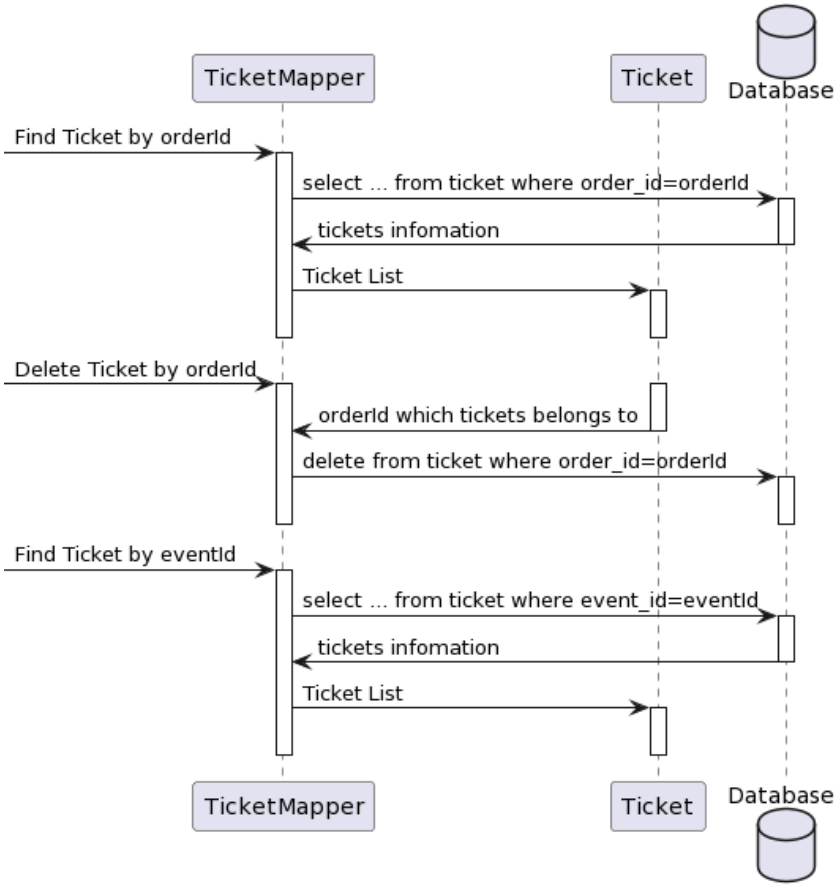
When update user-role connection, the mapper gets the new connection information from UserRole object and the update it into database depends on identity field username.

### 2.2.4 UserWithRoleMapper sequence and entity relationship diagram




For UserWithRole class, the system only run view method which mapper reads data from database and other object then mapper output the result to List of UserWIthRole object.

### 2.2.5 TicketMapper sequence and entity relationship diagram

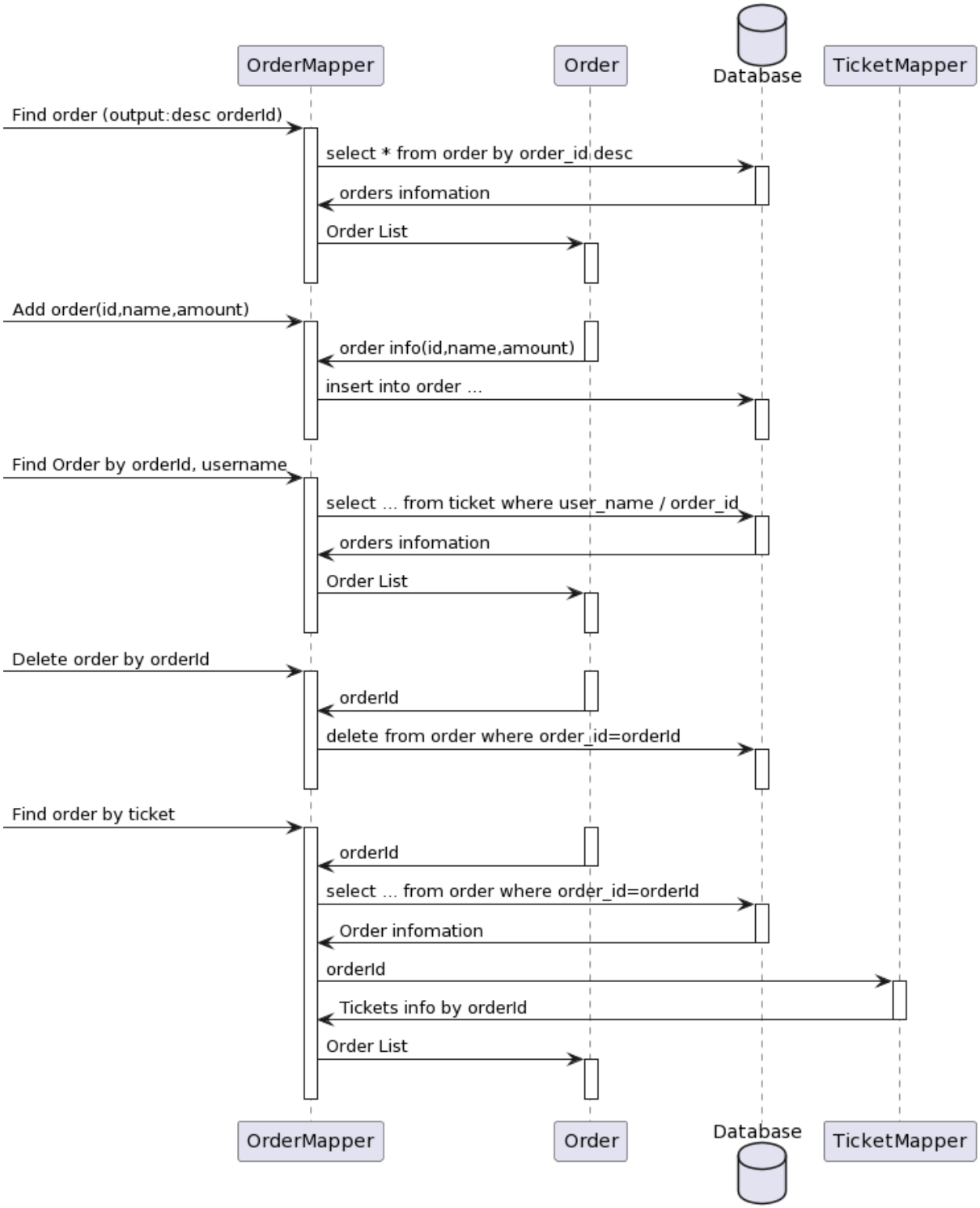



As shown here, all get and delete methods which will affect the database is included in this TicketMapper.

For get tickets' information by orderId or eventId from database, the mapper gets the orderId or eventide from Ticket class and then uses SQL query to get related tickets. After database return result, mapper outputs the result to a List of Ticket objects.

For delete ticket, the mapper gets the orderId from Ticket class and then uses SQL query to delete related tickets.

### 2.2.6 OrderMapper sequence and entity relationship diagram



As shown here, all get, insert and delete methods which will affect the database is included in this OrderMapper.

For find order method, the mapper gets all orders' information from database by SQL query then output the result to a List of Order object.

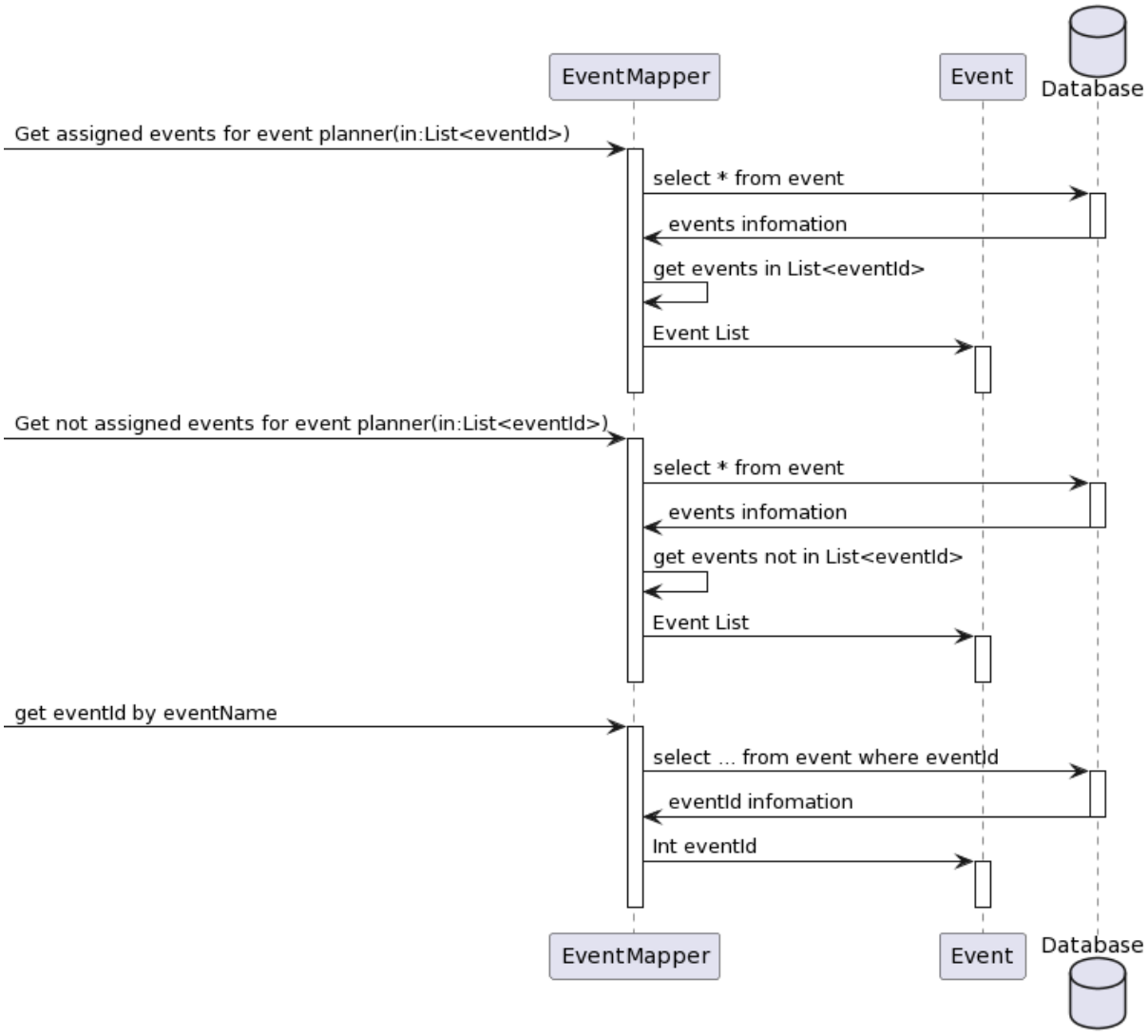
For add order method, the mapper gets the new order information from Order object and then execute SQL query to add the record to database.

For find order by username and orderId method, the mapper gets username or orderId from Order object and execute SQL query to get related order information from database. Once database returns the order information, the mapper output the result to a List of Order object.

For delete order method, the mapper gets orderId from Order object and execute SQL query to delete the record in database.

For find order by ticket method, the mapper gets orderId from Order object and execute SQL query to get related order information from database. Then the mapper used the orderId to get related tickets' information from TIcketMapper. After collecting all needed information, the mapper output the result to a List of Order object.

### 2.2.7 EventMapper sequence and entity relationship diagram

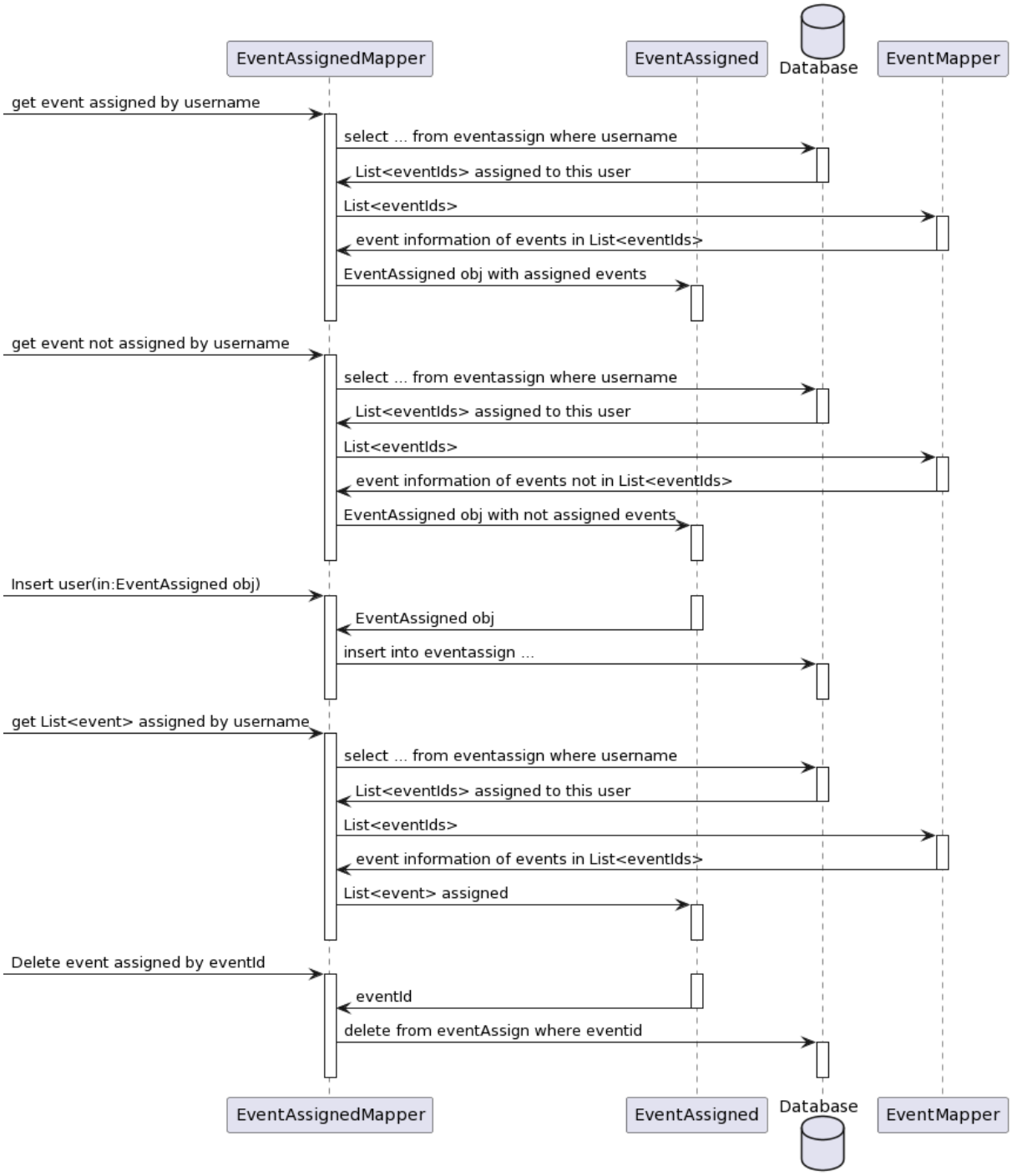


As shown here, all get methods which will affect the database is included in this EventMapper.

For get assigned or not assigned events method, the mapper get all events from database and get the result data based on the input eventId list. Then the mapper returns the result to a List of Event objects.

For get eventide by name method, the mapper get eventId by the input event name from database and then output the eventId information.

### 2.2.8 EventAssignedMapper sequence and entity relationship diagram




As shown here, all get, insert and delete methods which will affect the database is included in this EventAssignedMapper.

For get assigned or not assigned events method by username, the mapper gets all events assigned to the use by username from database and outputs a List of assigned eventId. Then this mapper delivers the List of assigned eventId to EventMapper's getAssignedEvent/ getNotAssignedEvent for event planner method to get target List<events> information and put username and List<events> into a new EventAssigend object as output.

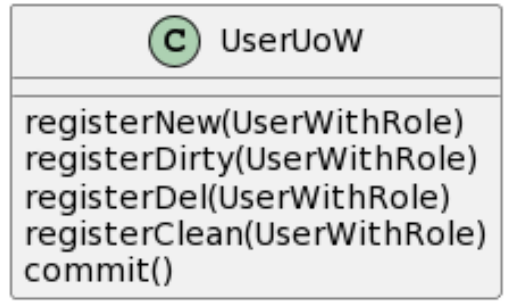
For get List<event> assigned method by username, the mapper gets all events assigned to the use by username from database and outputs a List of assigned eventId. Then this mapper delivers the List of assigned eventId to EventMapper's getAssignedEvent for the event planner method to get target List<events> information and uses List<events> as output.

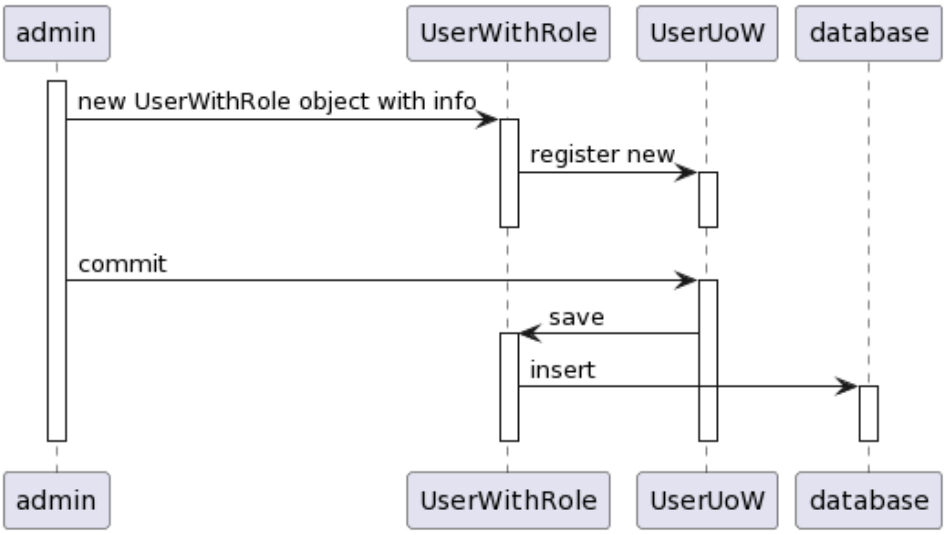
For insert user method, the mapper gets input by an EventAssigend object and insert the record into database.

For delete user method, the mapper gets input by eventId and run SQL query to delete all records related to this eventId in the database.

## 2.3 Unit of work

### 2.3.1 UserUoW diagrams



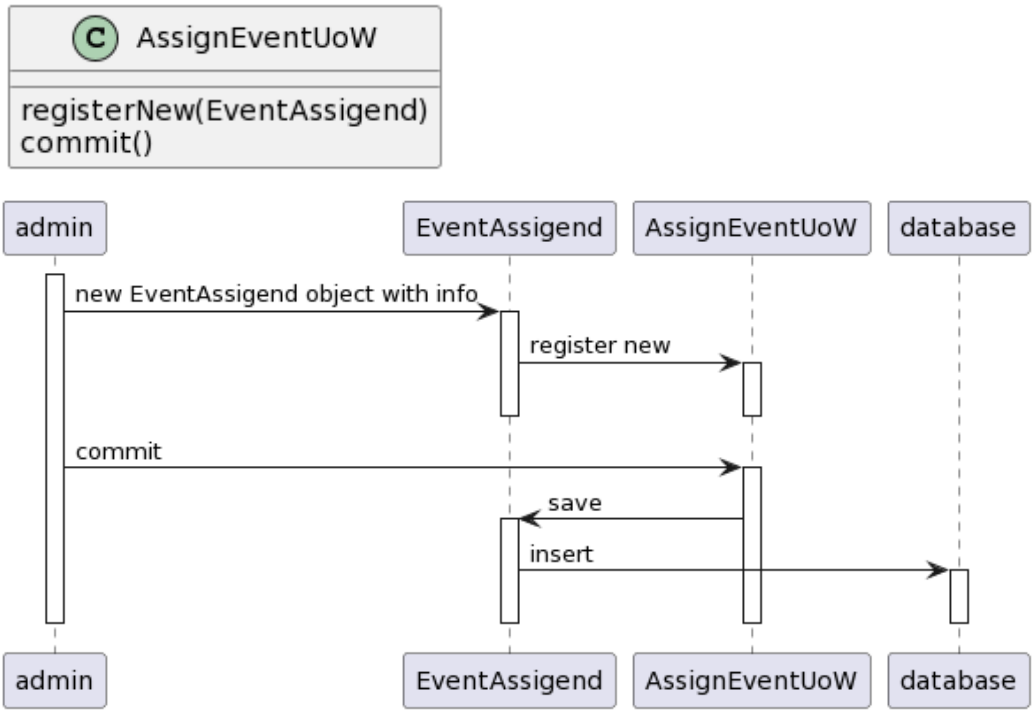

In our system, UserUoW is used when Admin tries to create new user.

Firstly, Admin inputs all need new user information (username, password and user role) and the Backend servlet get all these parameters from the Frontend.

Backend then create a new UserUoW object and a new UserWithRole object with all information input. The UserWithRole object's constructor has a method to register new UserWithRole object if UserUoW object exists.

After create these objects, the servlet called commit() to insert the new user information to the database.

### 2.3.2 AssignEventUoW diagrams




In our system, AssignEventUoW is used when Admin tries to assign new event to Event Planner.

Firstly, Admin select the specific Event Planner and the event to assign and the Backend servlet get all these parameters from the Frontend.

Backend then create a new AssignEventUoW object and a new EventAssigned object with all information input. The EventAssigned object's constructor has a method to register new EventAssigned object if UserUoW object exists.

After create these objects, the servlet called commit() to insert the new user information to the database.

### 2.3.3 Design rationale

As described before, both two Unit of work is used by admin to create related records in the database. For the event assign and user functions of admin, there is no delete or update user cases so all these methods are related to create.

As we know here, we only have one admin account for this system. However, we assume that in real-life use situation, there will be more than users logging in as admin to manage the system as the company definitely need more staff to organize the system to be more efficient. This may lead to severe concurrency problems as a small group of staff can all log in as admin and add users or assign events at the same time using the same account.

Unit of work is a design pattern that records all the related changes needed for the database and can organize multiple actions at a time by itself. By using this, the system can better track all the admin create and insert actions even when the same account is used by many different staff to avoid duplicated actions and later we can easily involve concurrency checks for these Unit of Work as planned.

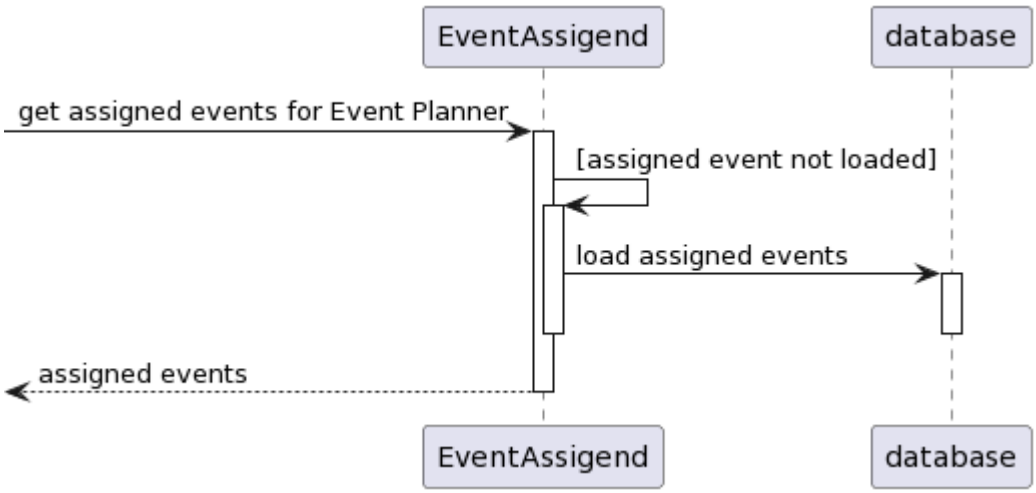
Also, Admin controls all the user information and all event information so that there will be huge amounts of data for all these functions when the system is used in real life so it's better to use Unit of Work pattern to keep all information at the same place to better track and manage the system progress.

Besides, for these two functions, there is not much code affected and all the method is in only a few related files so when using Unit of Work there's very little chance to forget using caller or object registration. Thus, the only disadvantage of Unit of Work pattern is solved here and it is the best choice for these functions.

## 2.4 Lazy load

### 2.4.1 EventAssigned

#### 2.4.1.1 Diagrams




In our system, the class EventAssigned contains three attributes: eventId, username and List< Event>. Our database table eventassigned only contains eventId and username. This means for each event and an event planner's assignment relationship, there is one record in the database. Thus, if select all assigned events for a event planner the output would be a list of eventId. The attribute List< Event > is added here to simplify some of the assign and manage event programming methods by recording all the assigned events information in it when needed.

We used Lazy initialization for the EventAssigned class's List< eventId> and the getEvents() method in this class can get the event list when needed(As shown in the picture).

```
public List<Event> getEvents() {
    if(this.events==null){
        events= EventAssignedMapper.getAssigned(this.username);
    }
    return events;
}
```

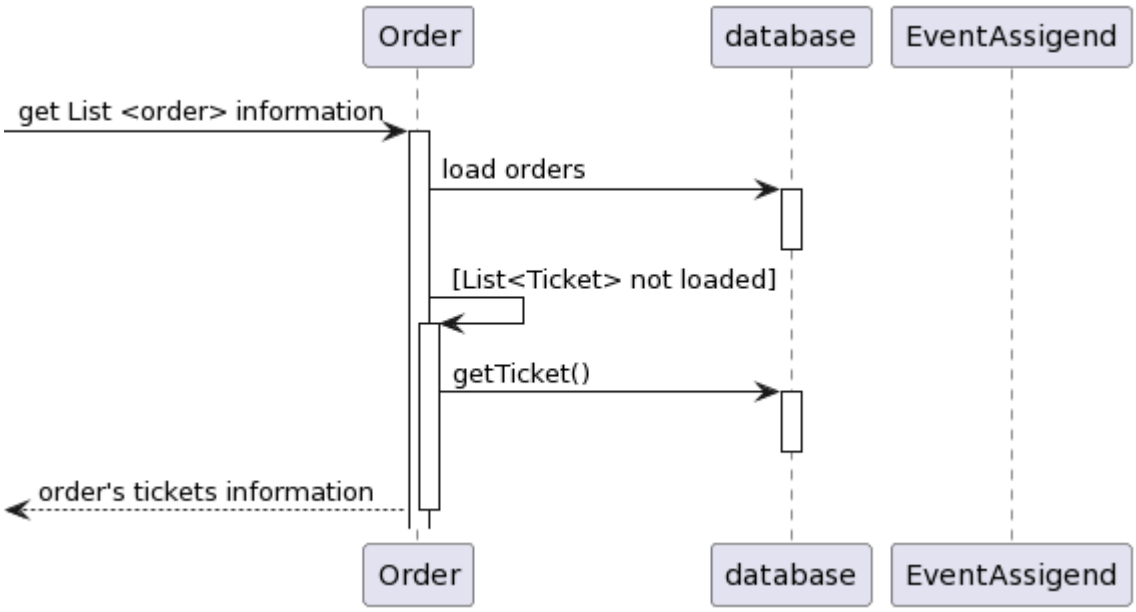
When the Admin navigates to the assign events page, even though a new EventAssigned object is created, it doesn't have any List< Event> information but when the page has to display all the events assigned to this event planner, the servlet calls getEvents() method and all events assigned can successfully displayed on the page.

#### 2.4.1.2 Design rationale

Due to our data structure design and database table design, when we try to get events assigned to an event planner, there will be at least two select actions to database and two while loops run by the system. Based on that, it is better that we do not get this information when not needed to improve our system's performance which means we should use lazy load to only load List< Event> data when needed.

### 2.4.2 Ticket

#### 2.4.2.1 Diagrams




In our system, the class Order contains a attribute List<Ticket> ticket except for its own fields. Our database table Order only contains it own information and for each order we need to query the ticket table and get related records. Thus, if select all tickets for an order the output would be a list of tickets. The attribute List< Ticket > is added here to simplify some of get order detail programming methods by recording all the tickets information in it when needed.

We used Lazy initialization for the Order class's List< Ticket > and the getTicket() method in this class can get the ticket list when needed(As shown in the picture).

```
public List<Ticket> getTicket() {
    if(ticket==null){
        TicketMapper ticketMapper=new TicketMapper();
        ticket = ticketMapper.findTicketByOrderId(orderId);
    }
    return ticket;
}
```

When the user navigates to their view order page, even though all orders related is read from the database, it doesn't have any List< Ticket > information but when the users try to see order detail page all the tickets of this order has to display, the servlet calls getTicket() method and all tickets can successfully displayed on the page. (As shown in the picture)

```
}else if (method.equals("admin")) {
    List<Order> orderList = orderService.findOrder( userName: null,  orderId: null);
    req.setAttribute( s: "orderList", orderList);
    req.getRequestDispatcher( s: "OrderView.jsp").forward(req, resp);
}
```

```
    } else if (method.equals("findOrderByOrderId")) {
        Integer orderId = Integer.valueOf(req.getParameter( s: "orderId"));
        List<Order> orderList = orderService.findOrder( userName: null, orderId);
        req.setAttribute( s: "tickets", orderList.get(0).getTicket());
        req.getRequestDispatcher( s: "TicketView.jsp").forward(req, resp);
    } else if (method.equals("deleteOrderByOrderId")) {
```

#### 2.4.2.2 Design rationale

In our system, the ticket and the order are stored in different data tables to avoid duplicated records. However, for all users, they need to view multiple order and multiple tickets. This is especially the case that Event Planner and Admin can see orders of different events, so they open the view order page, it takes a long time so load if we load the ticket information at the same time.

However, ticket information is not used before the user clicks into the order detail page and Users can sometime open order page to check order and never navigate to tickets page. Besides, if we try to load tickets with all orders, we need to join a large amount of data and go through huge amount of loops but if we only load tickets when the user queries a specific order, we can only search tickets for one order which large saves the resource and improve the system efficiency.

Thus, there is no need to load the tickets when the order loads and we use lazy load for Ticket attribute in Order object. As tested, the lazy load pattern has greatly shortened the time takes to load our order page especially for Admin and Event Planners.

## 2.5 Identity field

### 2.5.1 Users

For this table, we assume that a user cannot be added when the name is duplicated so we use a meaningful, simple and table-unique key: username. By using this, it is easier to identify the user when doing actions both for Frontend and Backend.

In our code, the User object contains an attribute username to save the database identity field username.

### 2.5.2 UserRole

We already knew that the user name is unique for each user and a user can only be one role so we chose the username field as the key to ensure each user can only have one record in this table.

In our code, the UserRole object contains an attribute username to save the database identity field username.

### 2.5.3 Event

For this table, we assume that sometimes the event name can duplicated as a company can host a similar event at a different time. If we use compound keys like eventName, location and start and end time, the table is too complicated as more than half of the fields is included in keys so we chose to a meaningless, simple and table-unique key: id to better identify events when we try to modify and manage it and simplify the event related actions by only search by the id field for certain records.

In our code, the Event object contains an attribute id to save the database identity field eventId.

### 2.5.4 EventAssigend

We already knew that the user name is unique for each user and the eventId is the identity key of the events table. This table is to save records of the assignment relationship of Event Planner user and events so this table's key is designed as meaningful and compound keys. This table use username and the eventId fields to be keys to ensure an event can only be assigned to an Event Planner once.

In our code, the EventAssigned object contains an attribute username and eventId to save the database identity field username and eventId.

### 2.5.5 Order

For this table we use a meaningless, simple and table-unique key: order_id as for orders the field username, date and total_price can be duplicated. Using meaningless field id is the simplest and quickest way to identify an order and to relate the order to other table records.

In our code, the Order object contains an attribute orderId to save the database identity field order_id.

### 2.5.6 Section

*Note: Even though we have this table and related classes, this table isn't used in the whole system logic.

For sections, all fields (venueId, capacity and type) can be duplicated to exclusively identify a record we will have to use compound keys with at least two fields so to be clear when the program tries to manage the section, we use a meaningless, simple and table-unique key: id to identify each record.

In our code, the VenueSection object contains an attribute sectionId to save the database identity field id.

### 2.5.7 Ticket

For Tcket, we choose a meaningless, simple and table-unique key: ticket_id as ticket's most field can be duplicated and a user or order can have multiple tickets. It is easier for the order or user to get their ticket by id than multiple fields.

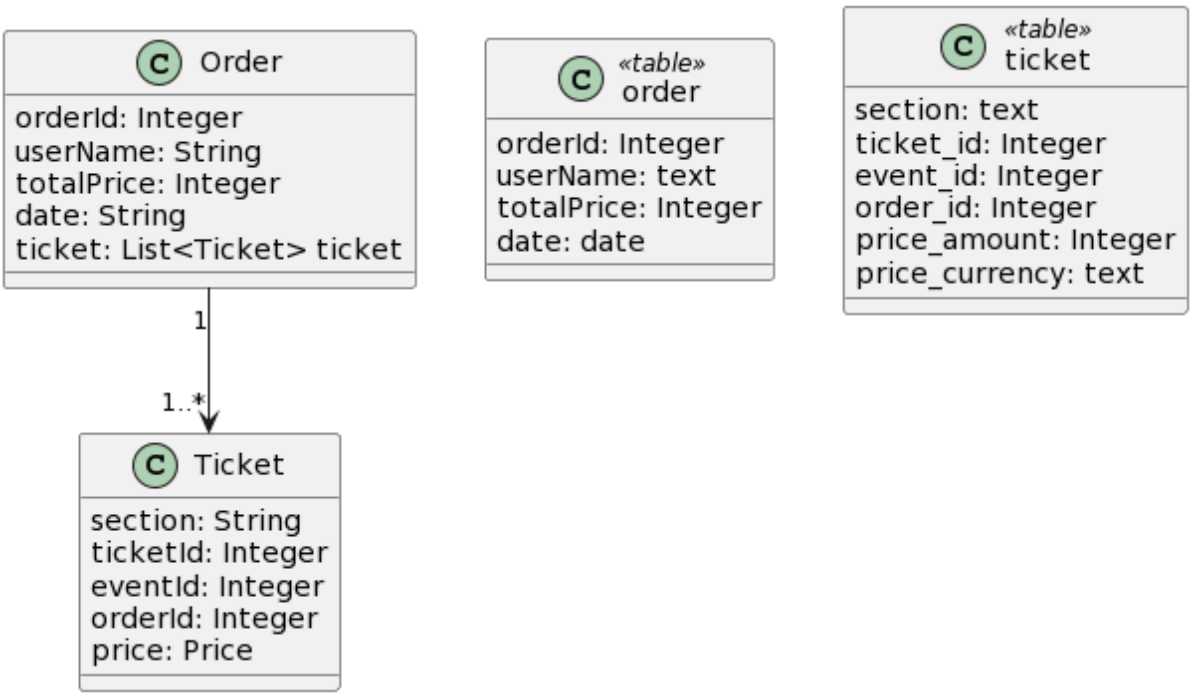
In our code, the Ticket object contains an attribute ticketId to save the database identity field ticket_id.

### 2.5.8 Venues

For Venues, we choose a meaningless, simple and table-unique key: venue_id as the venue's most fields can be duplicated. Even though mostly a venue should have a unique name, we still allow venues to have the same name in case of some rare situations. Besides, the venue is related to events and tickets so id would be faster for these functions to use.

In our code, the Venue object contains an attribute venueId to save the database identity field venue _id.

## 2.6 Foreign key mapping




As we designed in our system, a order can have multiple tickets and the foreign key of the ticket table is order_id which is primary key of the order table to nclusively identity a order. For a complete order information in the system, it should contain all the tickets in this order so the order class has an attribute List<Ticket> ticket.

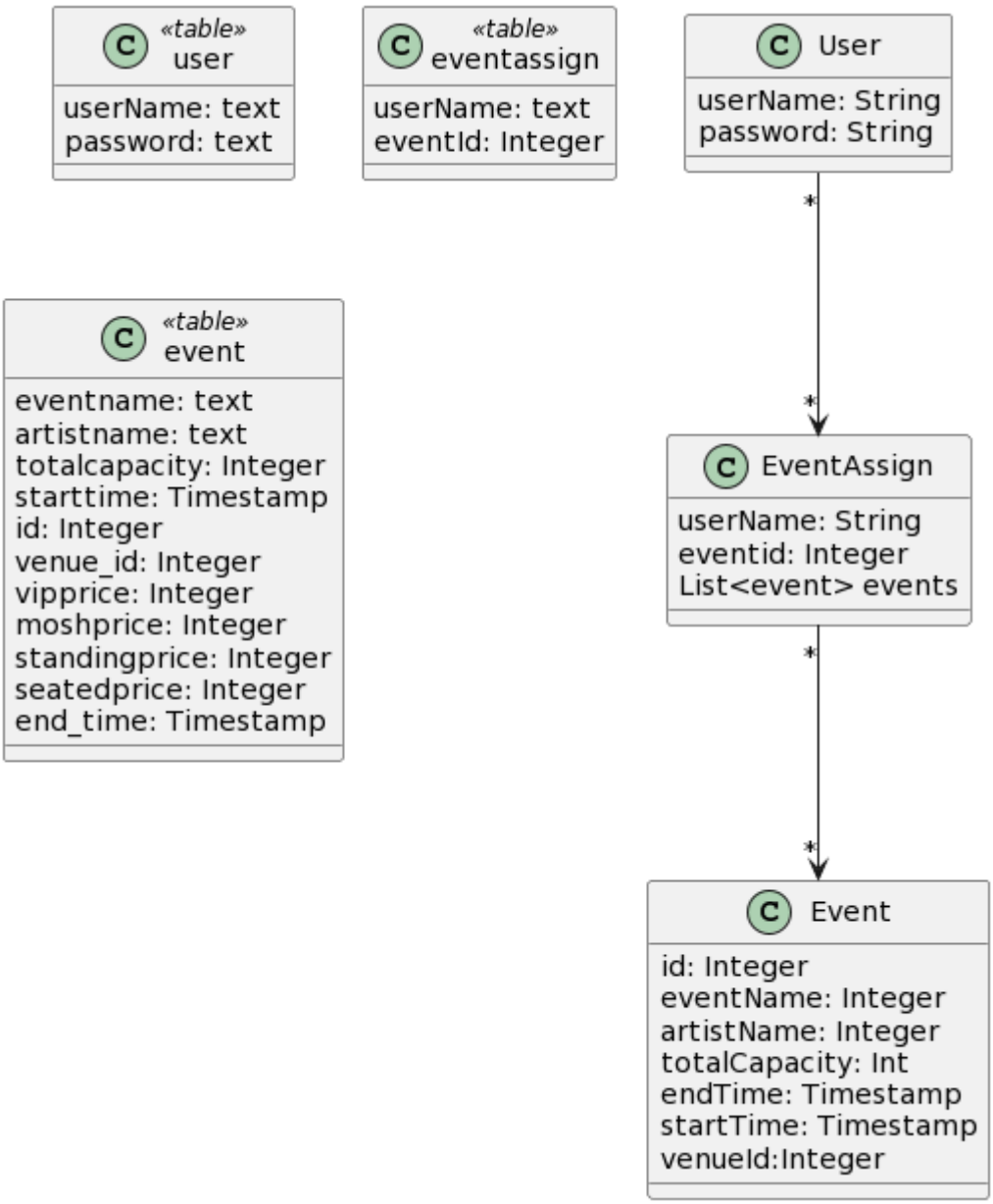
When we use find findorder(username, orderId) to find orders. In this method, it first runs an SQL query that selects the related order data and then calls the TicketMapper's findTicketByOrderId(orderId) method with the order_id return by the query. The method is shown below:

```
public List<Order> findOrder(String userName, Integer orderId) {
    Connection connection = JDBCtest.connectRender();
    String sql = "select order_id AS orderId, date,user_name AS userName,total_price AS totalPrice  from" + "\"" + "order" +
    if (Objects.nonNull(userName)) {
        sql += "WHERE user_name ='" + userName + "'";
    }
    if (Objects.nonNull(orderId)) {
        sql += "WHERE order_id =" + orderId;
    }
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    List<Order> resultList = new ArrayList<>();
    try {
        statement = connection.prepareStatement(sql);
        resultSet = statement.executeQuery();
        while (resultSet.next()) {
            orderId = resultSet.getInt( columnLabel: "orderId");
            userName = resultSet.getString( columnLabel: "userName");
            Integer totalPrice = resultSet.getInt( columnLabel: "totalPrice");
            String date = resultSet.getString( columnLabel: "date");
            List<Ticket> tickets = this.ticketMapper.findTicketByOrderId(orderId);
            resultList.add(new Order(orderId, userName, totalPrice, date, tickets));
        }
```

Inside the TicketMapper, findTicketByOrderId(orderId) runs a SQL query that returns a List of tickets which has the same orderId. This output is the attribute List<Ticket> for the Order object returned by findorder(username, orderId) method. The method is shown below:

```java
public List<Ticket> findTicketByOrderId(Integer orderId) {
    Connection connection = JDBCtest.connectRender();
    String sql = "select order_id AS orderId, price,section ,ticket_id AS ticketId ,event_id AS eventId from ticket ";
    if (Objects.nonNull(orderId)) {
        sql += "WHERE order_id =" + orderId;
    }
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    List<Ticket> resultList = new ArrayList<>();
    try {
        statement = connection.prepareStatement(sql);
        resultSet = statement.executeQuery();
        while (resultSet.next()) {
            orderId = resultSet.getInt( columnLabel: "orderId");
            Integer ticketId = resultSet.getInt( columnLabel: "ticketId");
            Integer eventId = resultSet.getInt( columnLabel: "eventId");
            String section = resultSet.getString( columnLabel: "section");
            /**********Embedded value***********/
            Price price = new Price(resultSet.getInt( columnLabel: "price"));
            resultList.add(new Ticket(price, section, ticketId, eventId, orderId));
            /**********Embedded value***********/
        }
```

## 2.7 Association table mapping




As shown here, the user (event planners) and the evet need to has a one-to-many mapping, as a event can have multiple event planners to manage it and one event planner can manage multiple events. We use three table in database to organize this relationship. User and event table is the original tables that are also used by other functions. To avoid adding multiple duplicated data records in these two frequently used tables we added a table called eventassign to record the relationship between event planners and the event they have been assigned. Each record in this table has a username field which is a foreign key related to the username in the user table and a eventId field which is a foreign key related to the id in the event table.

In actual code, we found that we do not use the password field when we try to create a new event assignment relationship so we did not map the user table. Instead we used the username in eventassign table and created a new EventAssign table which contains username, eventId and List<event> to do the mapping so that we do not have to query the user table to improve the system's efficiency.

When trying to get an event planner's associated events, we first run a query to search all related records of this user in eventassigned table and get a list of assigned events' id. Then we use these ids to query the event table to get these events' detailed information. The method is shown below:

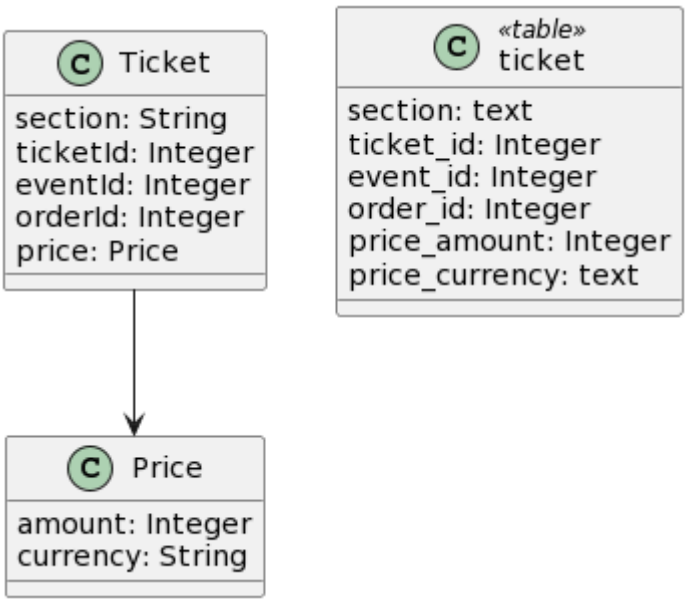
```
public class EventAssignedMapper {

    1 个用法  KkkellyM
    public static EventAssigend getAssignedList(String username) {
        Connection connection = JDBCtest.connectRender();
        String sql = "select * from eventassign where username = '"+username+"'";
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        EventAssigend result = null;
        String userName = null;
        List<Integer> resultList = new ArrayList<>();
        try {
            statement = connection.prepareStatement(sql);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                userName = resultSet.getString( columnIndex: 1);
                resultList.add(resultSet.getInt( columnIndex: 2));
            }
            EventMapper events =new EventMapper();
            result=new EventAssigend(userName, events.getAssignedforPlanner(resultList));
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
```

```
    2 个用法  KkkellyM +2
    public List<Event> getAssignedforPlanner(List<Integer> ids) {
        Connection connection = JDBCtest.connectRender();
        String sql = "select * from event";
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        List<Event> resultList = new ArrayList<>();
        try {
            statement = connection.prepareStatement(sql);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                int id = resultSet.getInt( columnIndex: 5);
                if(ids.contains(id)){
                    int venueId = resultSet.getInt( columnIndex: 6);
                    String eventName = resultSet.getString( columnIndex: 1);
                    String artistName = resultSet.getString( columnIndex: 2);
                    Integer totalCapacity = resultSet.getInt( columnIndex: 3);
                    Timestamp endTime = resultSet.getTimestamp( columnIndex: 11);
                    Timestamp startTime = resultSet.getTimestamp( columnIndex: 4);


                    resultList.add(new Event(id,eventName,artistName,totalCapacity,endTime,startTime,venueId));
                }
            }
        }
```

## 2.8 Embedded value




As shown, we have detailed price information for a ticket with both price number and currency. As said in the lecture slides, it does not make sense to have a corresponding table to just record the price but these two fields should be as a group so we use embedded value this.

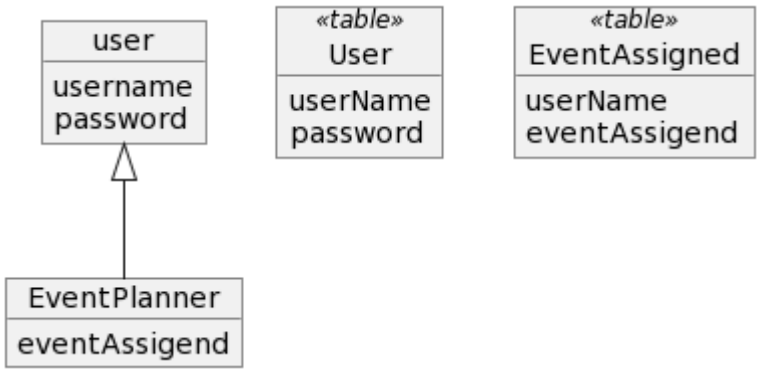
After read a ticket record from the database, beside put all other attributes to corresponding field, we put price and currency to a new Price object to show they are related.

The method is shown below:

```
public List<Ticket> findTicketByEventId(Integer eventId) {
    Connection connection = JDBCtest.connectRender();
    String sql = "select event_id AS eventId, price_amount As priceAmount, price_currency AS priceCurrency, section, ticket_
    if (Objects.nonNull(eventId)) {
        sql += "WHERE event_id =" + eventId;
    }
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    List<Ticket> resultList = new ArrayList<>();
    try {
        statement = connection.prepareStatement(sql);
        resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Integer orderId = resultSet.getInt( columnLabel: "orderId");
            Integer ticketId = resultSet.getInt( columnLabel: "ticketId");
            Integer id = resultSet.getInt( columnLabel: "eventId");
            String section = resultSet.getString( columnLabel: "section");
            Integer priceAmount = resultSet.getInt( columnLabel: "priceAmount");
            String priceCurrency = resultSet.getString( columnLabel: "priceCurrency");

            /**********Embedded value**********/
            Price priceObj = new Price(priceAmount, priceCurrency);
            resultList.add(new Ticket(priceObj, section, ticketId, eventId, orderId));
            /**********Embedded value**********/
```

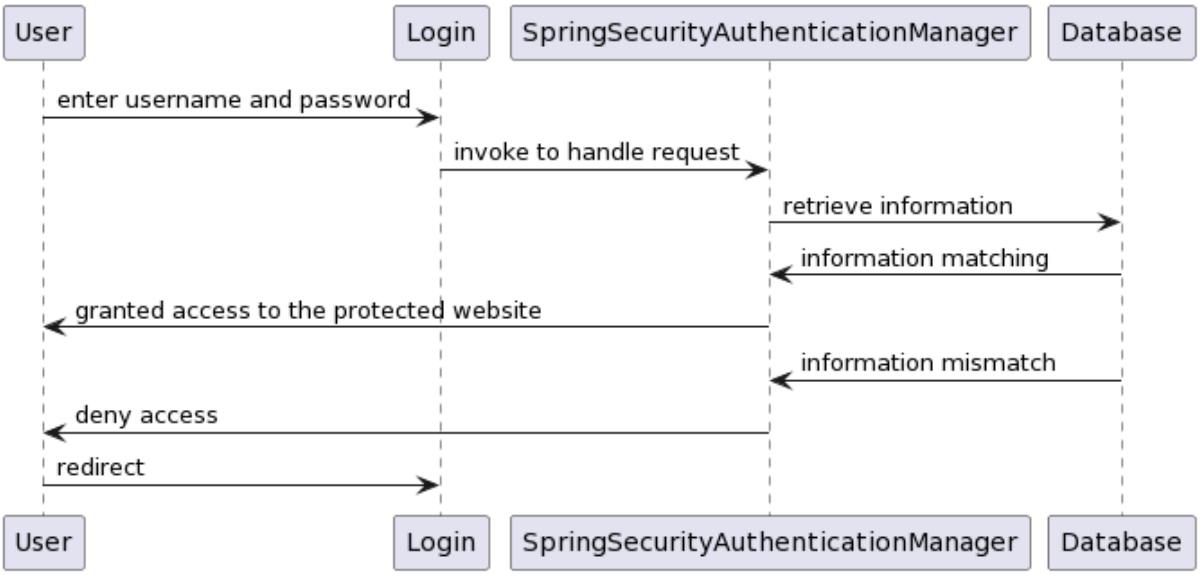
## 2.9 Inheritance patterns: Class table inheritance




As shown here, we use a table for user object and a table for EventPlanner which is named as EventAssigned object in our project.

Event Planners has assigned events which other user like Admin and customers do not have so for event planner information we not only need to read the information in the user table but also need to read the user table but also have to read the EventAssigned table to get all information needed for an Event Planner.

## 2.10 Authentication and Authorization




For this part, we use Spring Security to ensure that only valid users can access protected websites in the system and to restrict their access based on their roles..

We have configured the login page to be the root directory. On the login page, the user needs to provide their username and password. Then Spring Security Authentication Manager is invoked to look up the user's details. The loadUserByUsername method of the UserDetailsService implementation is called. This method retrieves user details from the database based on the username.

```java
@Override
public UserDetails  loadUserByUsername(String username) throws UsernameNotFoundException {
    UserWithRole user = findByUsername(username);

    if (user == null) {
        throw new UsernameNotFoundException(username);
    }

    UserBuilder builder = User.withUsername(user.getUsername());


    // Encode the password
    // Note there is no need to do this if the password is already encoded in the DB
    PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
    String encodedPassword = passwordEncoder.encode(user.getPassword());

    builder.password(encodedPassword);
    builder.roles(user.getRole());

    return builder.build();
}
```

If the user is not found or the password is incorrect, the user is considered unauthenticated and redirected to login page. Otherwise, when a user is authenticated and attempts to access a protected website, the Spring Security's filters intercept the user's request. It compares the user's roles and permissions against their requests to determine whether the user has enough privileges to access the website.

We add three roles: Admin, EventPlanner and User for user account stored in the database. And each role's privileges are shown as below.

```
.requestMatchers( ...patterns: "/viewuser*").hasRole("Admin")
.requestMatchers( ...patterns: "/createuser*").hasRole("Admin")
.requestMatchers( ...patterns: "/CreateUser*").hasRole("Admin")
.requestMatchers( ...patterns: "/CreateUser*").hasRole("Admin")
.requestMatchers( ...patterns: "/assignevents*").hasRole("Admin")
.requestMatchers( ...patterns: "/CreateVenue*").hasRole("Admin")
.requestMatchers( ...patterns: "/createvenue*").hasRole("Admin")
.requestMatchers( ...patterns: "/viewevents*").hasAnyRole( ...roles: "Admin","User")
.requestMatchers( ...patterns: "/OrderView*").hasAnyRole( ...roles: "Admin","User")
.requestMatchers( ...patterns: "/eventView*").hasAnyRole( ...roles: "Admin","User")
.requestMatchers( ...patterns: "/Ticket*").hasAnyRole( ...roles: "Admin","User")

.requestMatchers( ...patterns: "/updateEvent*").hasAnyRole( ...roles: "User","EventPlanner")
.requestMatchers( ...patterns: "/EventsView*").hasRole("User")
.requestMatchers( ...patterns: "/BuyEventsView*").hasRole("User")
.requestMatchers( ...patterns: "/TicketView*").hasRole("User")


.requestMatchers( ...patterns: "/eventplanner*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/eventPlannerCreateEvent*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/eventPlannerPage*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/AssignEvent*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/viewallbooking*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/viewbookings*").hasRole("EventPlanner")
.requestMatchers( ...patterns: "/viewticket*").hasRole("EventPlanner")

.requestMatchers( ...patterns: "/Navigate*").permitAll()
```

If the user has sufficient permissions, they are granted access to the protected website. Otherwise, they are denied access.

In summary, the Spring Security process involves credential submission, validation, user details retrieval and match, which ensures authentication and authorization.