UNIVERSITÀ DEGLI STUDI FIRENZE

Domain Driven Design of a Digital Ticketing Validation System to Prevent Scalping

Caselli Niccolò, Cavichia Lautaro, Tinacci Lapo

31/01/2025

Project submitted for the final exam of the Software Engineering course - B003372

Professor:
Enrico Vicario

Department:
Department of Information Engineering

Program:
Bachelor's Degree in Computer Engineering

Table of Contents

List of Figures

List of Tables

1 Introduction

Ticket resale, also known as ticket scalping, is a practice that involves reselling tickets for admission to events like concerts through unlicensed sellers, often at inflated prices. This increasingly common procedure creates significant inconvenience for consumers and worsens the overall experience of attending an event. It negatively impacts the social lives of many people who cannot afford to pay higher prices to attend their favorite concerts or shows. Some applications, like DICE [1], allowing users to view their tickets only a few hours before the beginning of the event. While this approach certainly reduces the likelihood of scalping, it does not guarantee complete effectiveness. This work represents the Software Engineering Final Project of Caselli Niccolò, Cavichia Lautaro, and Tinacci Lapo. It aims to propose a simple yet effective architecture, designed using a domain-driven approach, to address the issue.

2 Requirements Analysis

This section outlines the key aspects of the system, including its objectives, core requirements, and the actors involved in the system's functionality.

2.1 Problem Statement

This project aims to develop innovative software to manage the event ticket verification process through a system designed from its foundation to make ticket scalping impossible. The application is designed as a mobile app, taking advantage of the phone's camera to scan QR codes efficiently and securely for ticket validation. Furthermore, the application incorporates comprehensive tools for event and staff management.

2.1.1 Involved Actors

The actors in the system are as follows:

- **Admins**: Admins are the event organizers with administrative control over their events. They can create and edit events, manage collaborators by adding or removing staff members, and customize event details, such as the maximum number of tickets and ticket prices.
- **Staff**: The Staff represents the collaborators, authorized by the Admins, who are responsible for performing ticket verification at the event entrance. They ensure that participants have a valid ticket that has not been used.
- **Guest**: The Guests are users who act as clients. They can view all available events, purchase one or more tickets for their desired event, and attend by validating their tickets.

2.2 Core Requirements

The anti-scalping system is designed to reverse the traditional responsibility in the ticket verification process: the usual validation flow of a ticket in a concert or any other kind of event consists of the entrance employee who scans the user ticket, which is in the form of a QR code. The problem with this approach is that the code contained in the ticket can be copied, shared, or sold illegally, and there are no mechanisms to prevent this. The approach proposed in this work relies on the fact that it is not the client who presents the QR code to the staff, but the staff member who shows a verification QR code to the guest containing a secret, unique code. The user scans it and doing so confirms the presence at the entrance of the event: this cannot be falsified and implies that the user can't see or exchange the ticket before arriving at the event. Once the user's presence and a valid ticket are confirmed, the system notifies the staff, completing the verification process securely and efficiently.

The project serves as a simple demonstration rather than a complete implementation. However, it includes functional JWT-based authentication, DTO validation, and database persistence, following best practices of software engineering and domain-driven design.

[1]DICE is an app for discovering and purchasing tickets for live events, offering personalized recommendations via Spotify and Apple Music. More info on `https://dice.fm/`.

2.3 Possible Drawbacks of The Approach

Here are some potential drawbacks of this approach to handling ticket validation.

- **No Ticket Printing**: Since the system relies on electronic ticket verification, users cannot print their tickets.
- **User Inconvenience**: The system requires the user to scan the QR code provided by the staff upon arrival, which may cause confusion, especially if the user is unfamiliar with the process.
- **Technology Dependency**: The system relies on the user's device and internet connectivity for ticket verification. Any issues with network access or device malfunction could prevent successful verification, thereby preventing the user's ability to attend the event.
- **Scalability**: The verification process could become slower as the number of attendees increases, particularly if staff is not properly trained or if users face difficulties during the verification process.

3 System Design

During the system design phase, several tools were utilized:

- **StarUML**: For the creation of use case diagrams and UML diagrams.
- **Draw.io**: For entity-relationship schemas.
- **Figma**: For mockups of user interfaces.

3.1 Use Case Diagrams

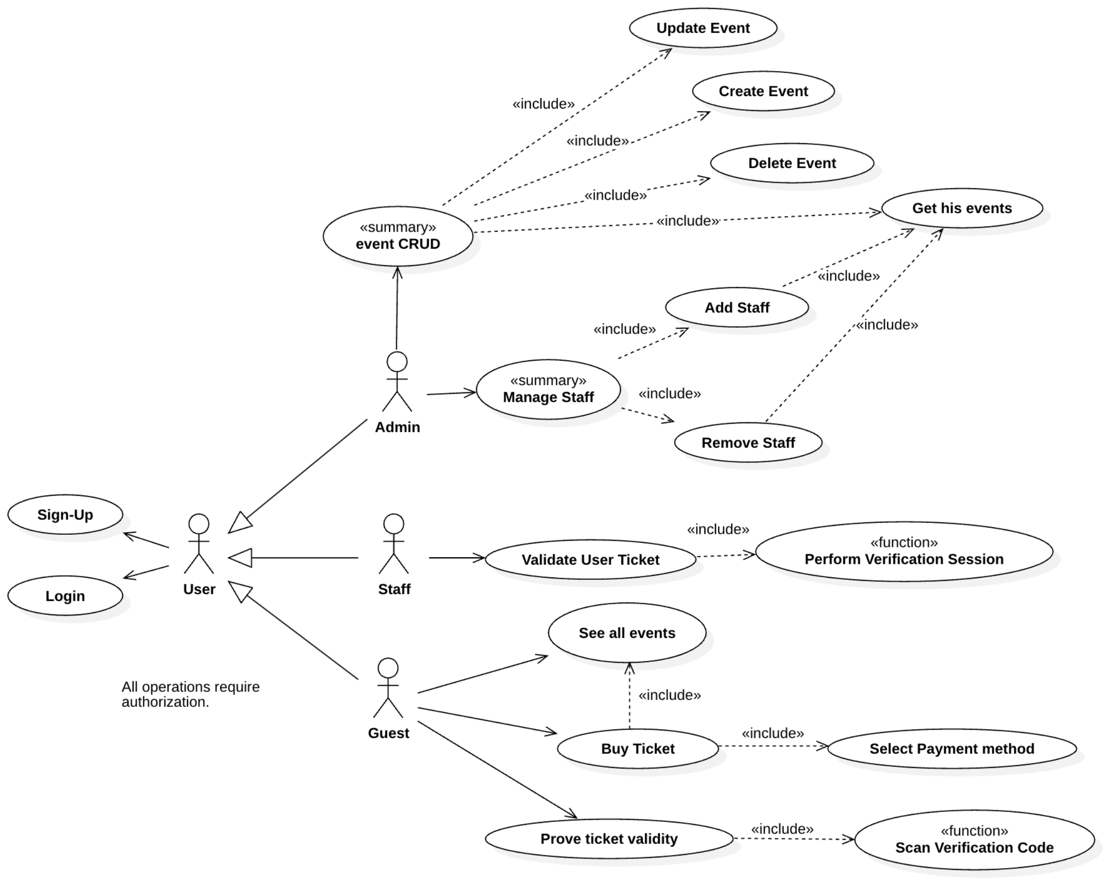


Figure 1: Use Case Diagram

3.2 Detailed Use Case Templates

The following templates outline some of the primary use cases identified in the project. During the design phase, the original use cases included only standard fields like `ID`, `Name`, `Level`, `Actors`, `Pre-conditions`, `Post-conditions`, `Basic Flow`, and `Alternative Flow`. However, after the integration phase, we decided to enhance the use case documentation by adding two additional fields: `Test` and `DTO`. These additions align the use cases with the code documentation. The `DTOs`, which represent the arguments for interacting with the controllers, are included in the template, as they may help in designing future API fields for potential implementations (e.g. REST API). The `Test` section was introduced to ensure that use cases are thoroughly tested with integration tests.

3.2.1 User Registration

Id	UC-1
Name	User Registration
Level	System Goal
Actors	User
Pre-conditions	The user is not authenticated in the system and does not have a profile.
Post-conditions	The user has a profile, is authenticated, and has access to the personalized system interface.
Basic flow	1. The user opens the registration page. (See mockup in Figure 3) 2. The user enters first name, last name, email, and password. 3. The system validates the fields. 4. The system creates a new account.
Alternative flow	• If the fields contain validation errors, the system displays an error message. • If there is an authentication system error, the system displays an error message and asks the user to try again later.
DTO	DTO 1
Test	Integration Tests 1, 2, 3

3.2.2 User Login

Id	UC-2
Name	User Login
Level	System Goal
Actors	User
Pre-conditions	The user has an account but is not yet authenticated in the system.
Post-conditions	The user is authenticated and has access to the personalized system interface.
Basic flow	1. The user opens the login page. (See mockup in Figure 2) 2. The user enters their email and password. 3. The system verifies that the credentials are correct. 4. The system logs the user in upon successful verification.
Alternative flow	• If the credentials are incorrect, the system displays an error message and prompts the user to try again. • If there is an authentication system error, the system displays an error message and asks the user to try again later.
DTO	DTO 2
Test	Integration Tests 4, 5, 6

3.2.3 Event Creation

Id	UC-3
Name	Event Creation
Level	User Goal
Actors	Admin
Pre-conditions	The admin is authenticated in the system.
Post-conditions	The new event is created and the creator is registered as an admin for the event. The event should appear in the lists of incoming events.
Basic flow	1. The admin opens the event creation page. (See Figure 4) 2. The admin fills in all the required fields (title, date, description, etc.). 3. The system performs the validation of the fields. 4. The system saves the event data in the database and confirms the creation to the organizer.
Alternative flow	• If the organizer does not provide all the required data or if the data does not meet the validation conditions, the system will display an error message and request that the missing fields be completed.
DTO	DTO 3
Test	Integration Tests 7

3.2.4 Add Staff Member

Id	UC-4
Name	Add Staff Member
Level	User Goal
Actors	Admin
Pre-conditions	The admin is authenticated in the system and should have access to an event along with the e-mail of the user they wish to add as staff.
Post-conditions	The new collaborator is added to the event.
Basic flow	1. The admin opens the staff management page. (See Figure 5) 2. The admin searches by e-mail for the desired user. 3. The system adds the user as a staff member of the event.
Alternative flow	• If the staff member is already in the event: do nothing. • If the entered email does not correspond to any user: throw an exception. • If an error occurs while saving the data in the system, the organizer receives an error notification.
DTO	DTO 6
Test	Integration Tests 11, 12, 13

3.2.5 Remove Staff Member

Id	UC-5
Name	Remove Staff Member
Level	User Goal
Actors	Admin
Pre-conditions	The admin is authenticated in the system and should have access to an event along with the e-mail of the user they wish to remove from the staff.
Post-conditions	The new collaborator is removed from the event.
Basic flow	1. The admin opens the staff management page. (See Figure 5) 2. The admin selects the desired user. 3. The system removes the user from the staff of the event.
Alternative flow	• If the user is not in the event staff: do nothing. • If an error occurs while saving the data in the system, the organizer receives an error notification.
DTO	DTO 7
Test	Integration Tests 14, 15, 16

3.2.6 Consultation of all available events

Id	UC-6
Name	Consultation of all available events
Level	User Goal
Actors	Guest
Pre-conditions	The guest must be authenticated.
Post-conditions	The user sees a list of available events with relevant information.
Basic flow	1. The guest opens the events page. 2. The system loads all the events that have not yet passed (Image 6).
Alternative flow	• If no events are available, the system shows a message informing the user that no events are available.
Test	Integration Tests 8

3.2.7 Ticket Purchase

Id	UC-7
Name	Ticket Purchase
Level	User Goal
Actors	Guest
Pre-conditions	The guest is authenticated in the system.
Post-conditions	The guest has completed the purchase, has been notified of the success of the process, and a new ticked is issued.
Basic flow	1. The user selects the event for which they wish to purchase tickets (see Mockup in Figure 6). 2. The user navigates to the event page (see Figure 7). 3. The user enters the number of tickets they wish to purchase. 4. The user selects the desired payment method. 5. The system processes the payment and issues the ticket. 6. The system confirms the purchase.
Alternative flow	• If the user is not in the event staff: do nothing. • If an error occurs while saving the data in the system, the organizer receives an error notification.
DTO	DTO 8
Test	Integration Tests 18, 19

3.2.8 Ticket Verification

Id	UC-8
Name	Ticket Verification
Level	User Goal
Actors	Guest, Staff
Pre-conditions	Both the guest and the staff are authenticated. The guest has a ticket for an event and is at the entrance of the event. The staff member is there to validate the guest's ticket.
Post-conditions	The ticket is successfully verified, marked as "used", and the guest can join the event.
Basic flow	1. The guest arrives at the event with a ticket. 2. The staff member starts the verification process and generates a unique QR code (see Figure 9). 3. The guest scans the QR code with the application on their phone (see Mockup 8). 4. The system verifies the authenticity of the code and confirms that the guest has a valid ticket. 5. The system registers the ticket as used. 6. The staff member receives a notification of success (see mockup in Figure 10).
Alternative flow	• If the guest attempts to fake the verification code, the system will raise an exception. • If the guest doesn't have a ticket (or it's used), the verification process fails (see mockup in Figure 11).
DTOs	DTOs 9, 10, 11
Test	Integration Tests 25, 26, 27, 28, 29, 30

3.3 Mockups for User Interfaces

The following presents potential mockups for some of the primary user interfaces intended for the actors within the project.

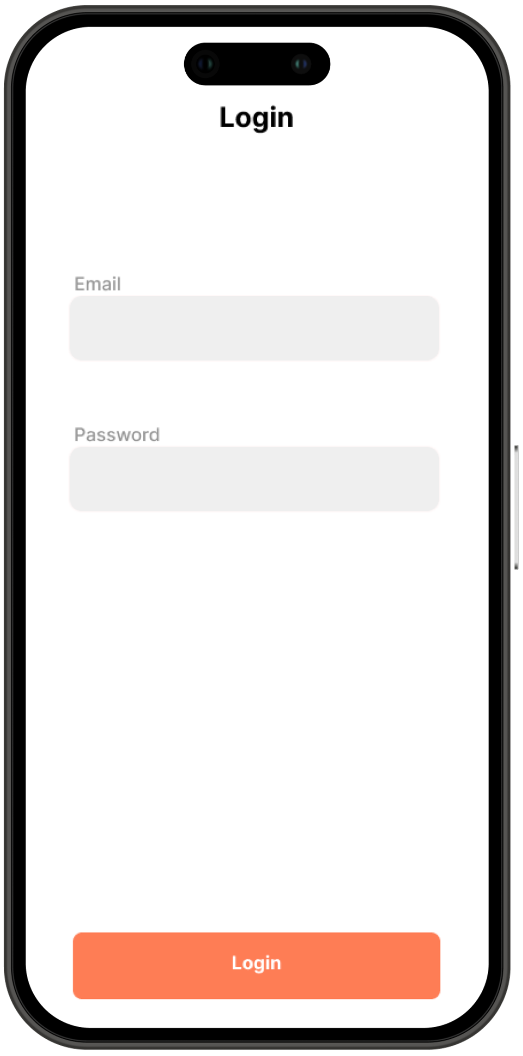


Figure 2: Login Screen Mockup

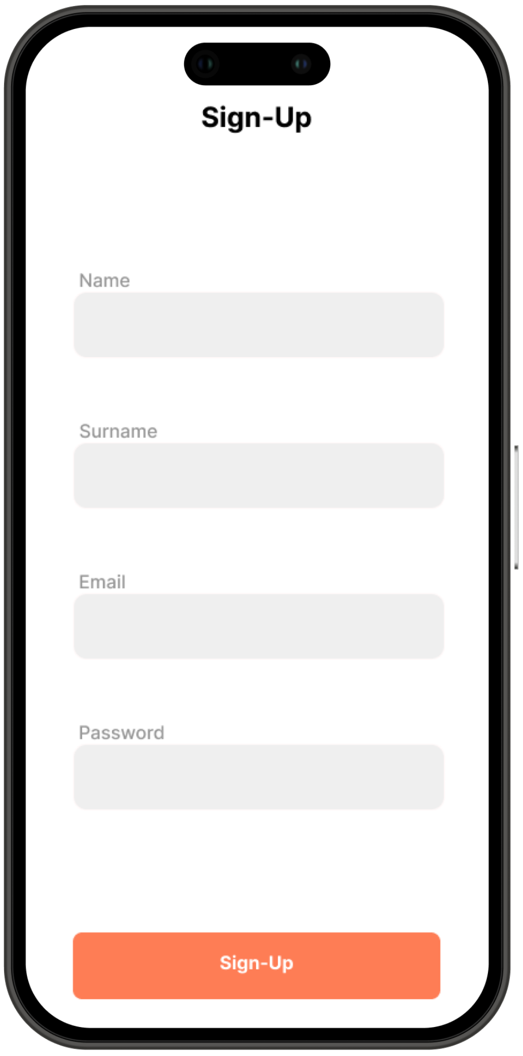


Figure 3: Sign-Up Screen Mockup

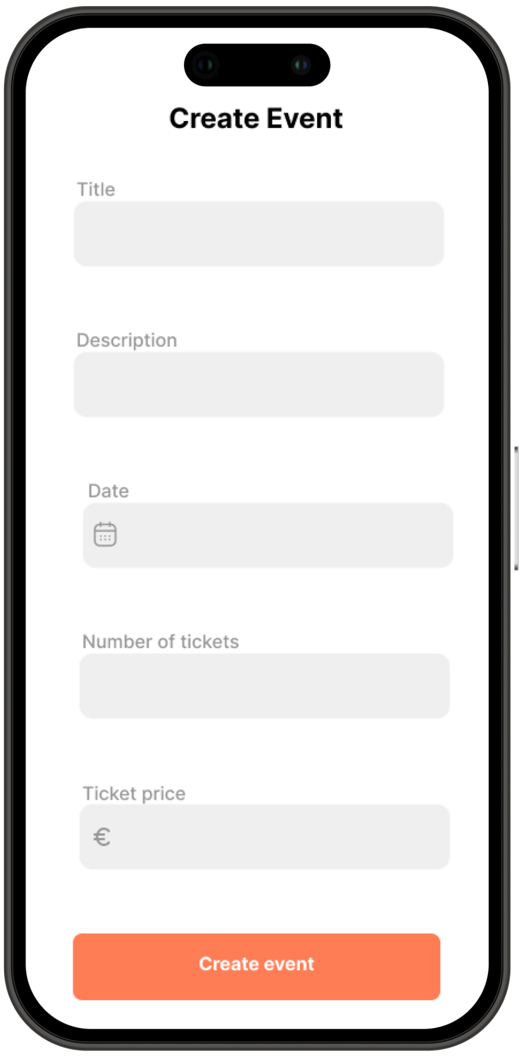


Figure 4: Create Event Screen Mockup

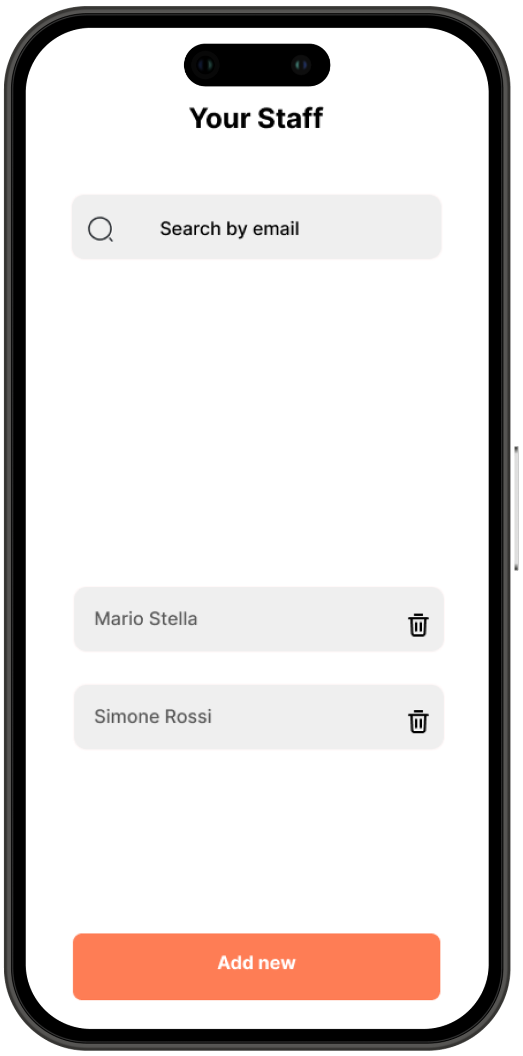


Figure 5: Staff Management Screen Mockup

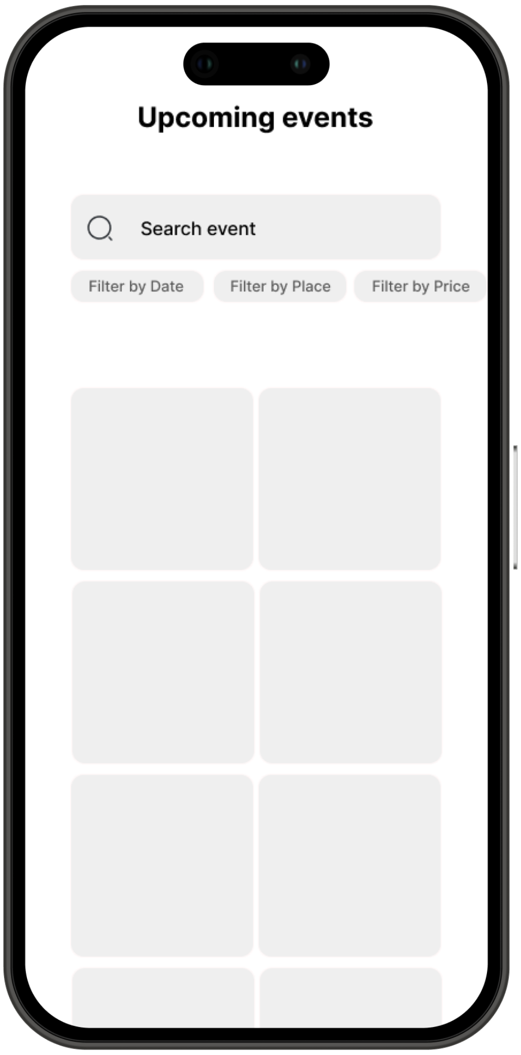
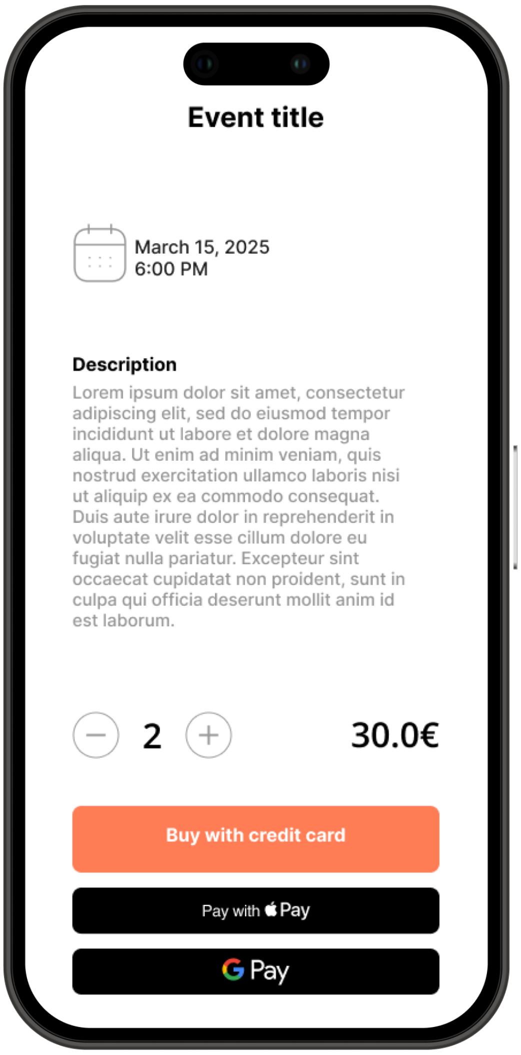


Figure 6: Events Screen Mockup



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 7: Event and Payment Screen Mockup

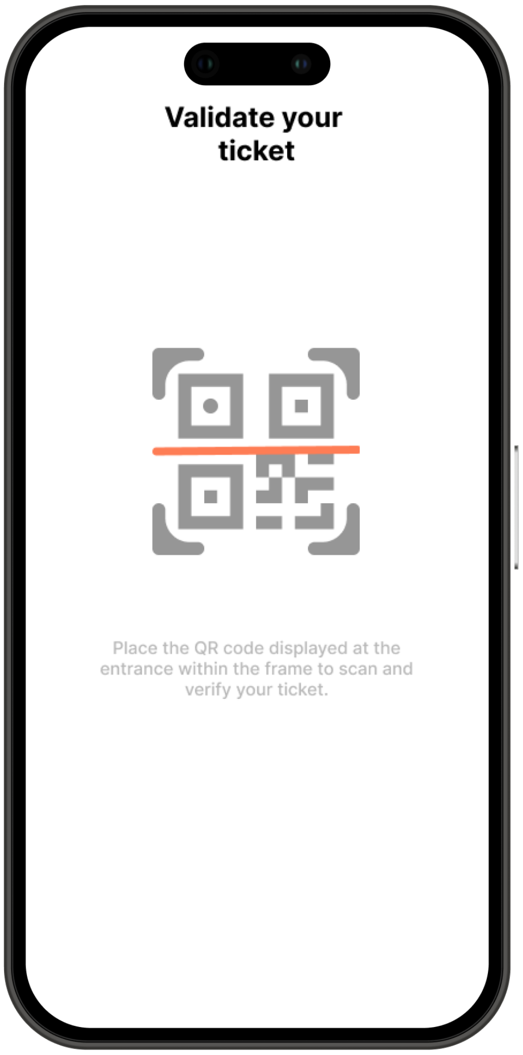

Validate your ticket
Place the QR code displayed at the entrance within the frame to scan and verify your ticket.

Figure 8: Mockup of Ticket Validation Screen for Guests

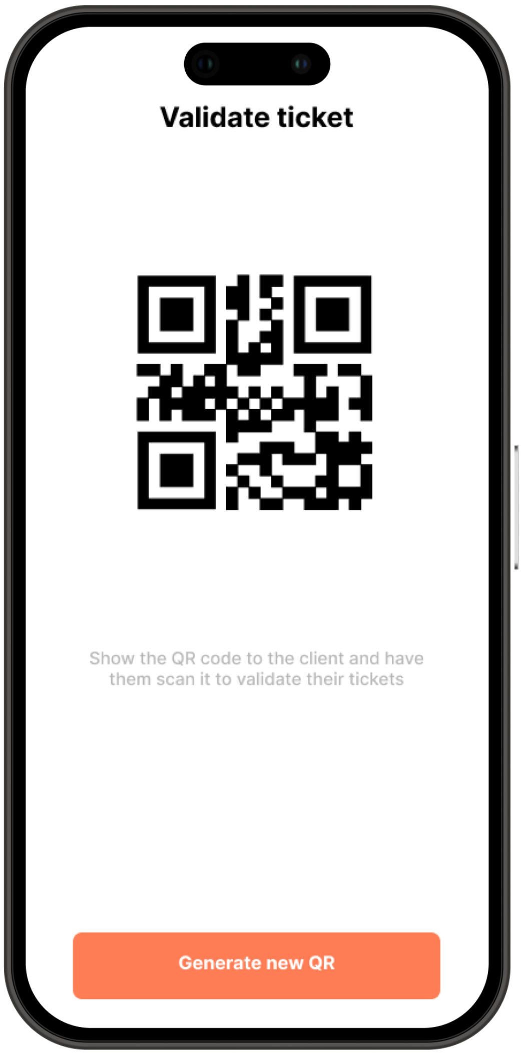

Validate ticket
Show the QR code to the client and have them scan it to validate their tickets
Generate new QR

Figure 9: Mockup Screen for Staff Ticket Validation

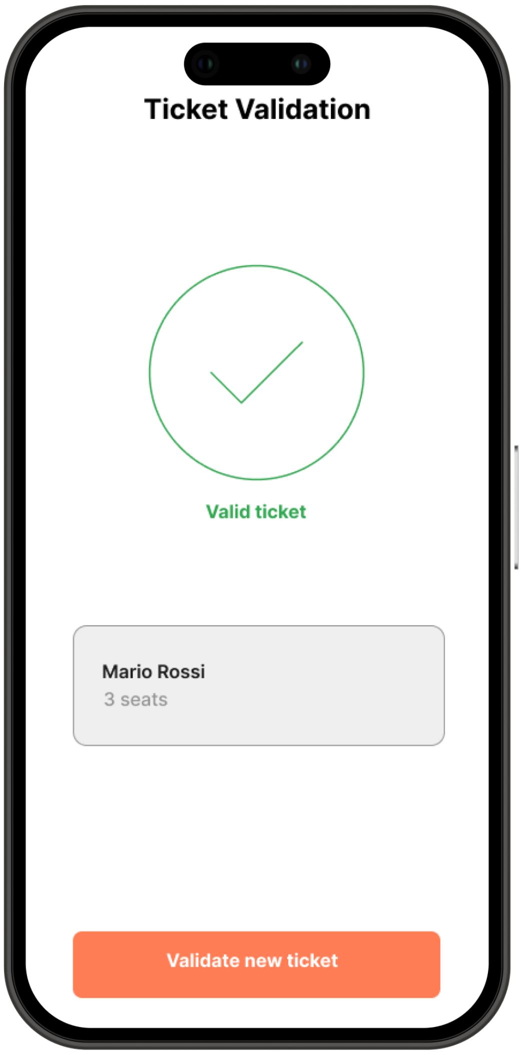


Figure 10: Ticket Validation Success Screen

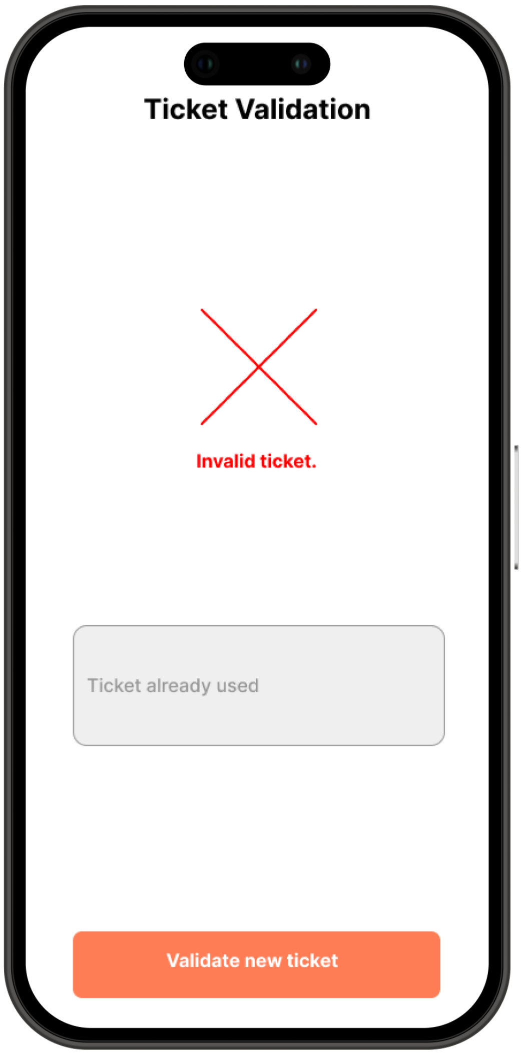


Figure 11: Ticket Validation Failure Screen

3.4 Database conceptual data modeling

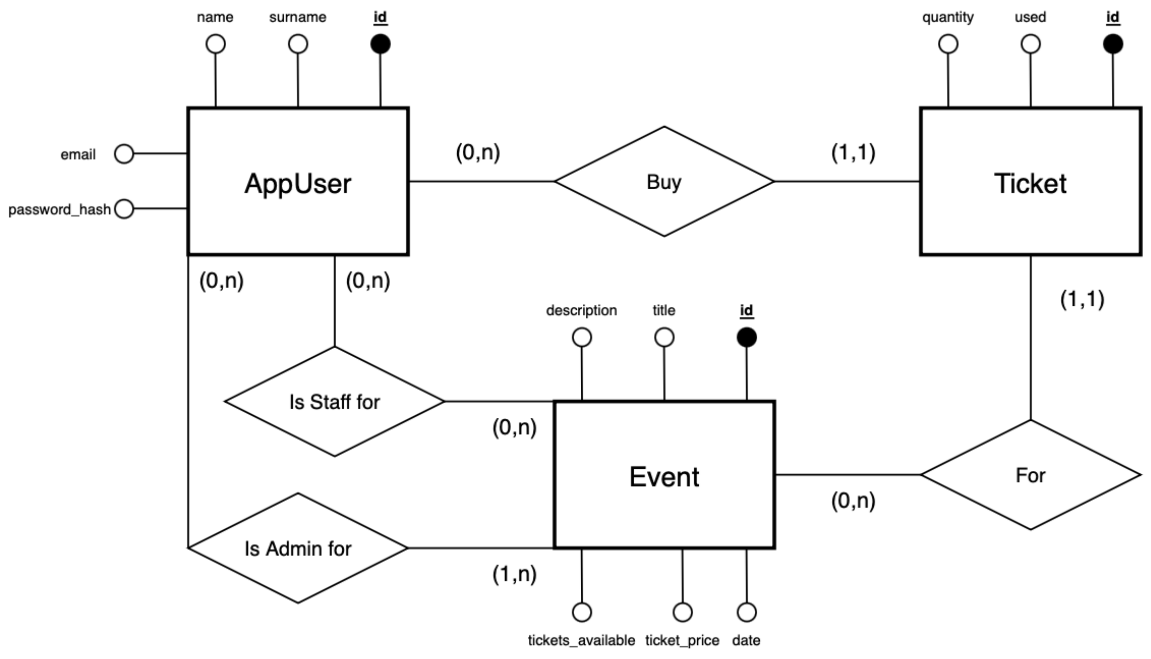


Figure 12: Entity-Relationship Model

The Entity-Relationship (ER) model shown in Figure 12 represents the conceptual design of the database. This specific design is not limited to a particular logical model; the implementation of the schema in the relational model will be discussed later (see Subsection 4.4). The design includes three main entities: `Event`, `AppUser`, and `Ticket`. The `Event` entity stores details about events, such as the date, title, description, number of available tickets, and ticket price. The `AppUser` entity represents users with attributes including a unique identifier, email, password (hashed), name, and surname. The `Ticket` entity stores information about purchased tickets. Two many-to-many relationships are represented between `AppUser` and `Event`, capturing the roles of staff and administrators responsible for managing specific events.

4 Implementation Details

During the implementation phase of the project, the use cases were brought to life, starting with the design of the domain model. Next, the business logic was designed while considering the overall architecture. Only after that, coding was implemented using the Java programming language.

4.1 Project Dependencies

The project relies on several dependencies, managed through *Maven* and defined in the `pom.xml` file. Below is an overview of the main dependencies and their respective versions:

- **Environment Variable Management:** The `dotenv-java` library (version 3.1.0) is employed to securely handle configuration variables, such as credentials and sensitive parameters, using `.env` files.
- **JWT Authentication:** The `jjwt` library (version 0.12.2), divided into `jjwt-api`, `jjwt-impl`, and `jjwt-jackson` modules, is used for generating and verifying JSON Web Tokens (JWT) to authenticate users and share information during the ticket verification process.
- **Database Integration:** The PostgreSQL JDBC driver (`org.postgresql`, version 42.5.0) enables the interaction with the relational database.
- **Data Validation:** The project employs `jakarta.validation-api` (version 3.1.0), `jakarta.el` library (`org.glassfish`, version 4.0.2) and `hibernate-validator` (version 8.0.0.Final) to ensure the validation of Data Transfer Objects (DTO).
- **Password Hashing:** The `jbcrypt` library (`org.mindrot`, version 0.4) is used to securely store passwords in the database by computing their hashes.

- **Testing:** `junit-jupiter` (version 5.11.4) and `mockito-core` (version 4.2.0) are utilized for testing and mocking components. The `jacoco-maven-plugin` (version 0.8.12) is configured to monitor test coverage.

The project is configured to use Java 17 as the compilation target.

4.2 Overview of the System Architecture

An overview of the system architecture is presented in Figure 13, which offers a comprehensive visual representation of the entire system. The following sections will delve into each component of the architecture in detail, examining their individual roles and the interactions between them.

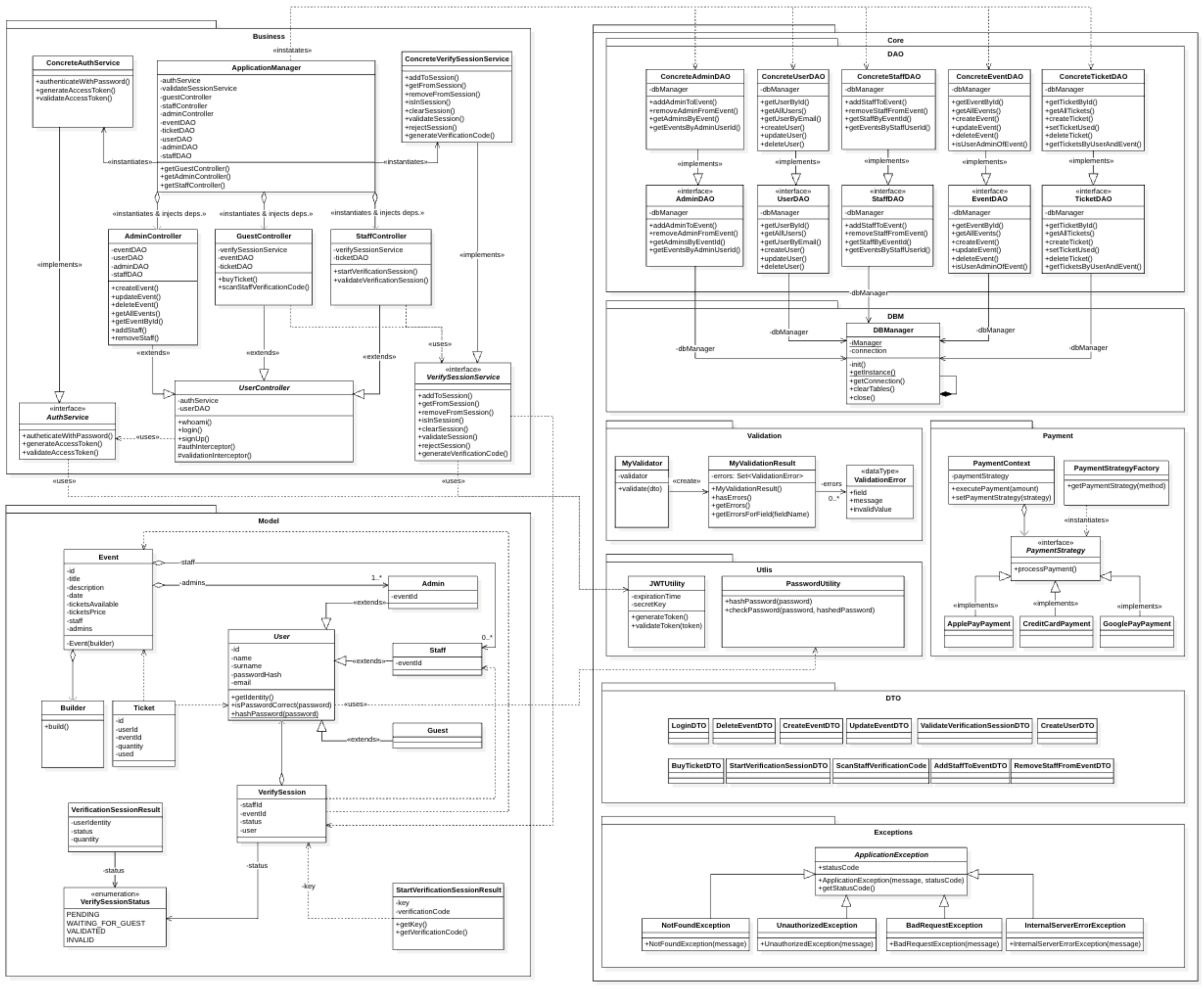


Figure 13: System Architecture

4.3 Codebase Architecture and Implementation

Below is the structured layout of the project's directory, following best practices in Java for package separation and layering.

```
src
├── it
│   ├── java
│   │   └── org.swe
│   └── resources
├── main
│   ├── java
│   │   ├── org.swe
│   │   │   ├── business
│   │   │   ├── core
│   │   │   │   ├── DAO (Data Access Objects)
│   │   │   │   ├── DBM (Database Manager)
│   │   │   │   └── DTO (Data Transfer Objects)
│   │   │   ├── exceptions
│   │   │   ├── payment
│   │   │   ├── utils
│   │   │   ├── validation
│   │   │   └── Config (Configuration File)
│   │   └── model
│   │       ├── Config (Configuration File)
│   │       └── Main (Entry Point)
│   └── resources
├── test
│   └── java
│       └── org.swe
│           ├── business
│           ├── core
│           │   ├── utils
│           │   └── validation
│           └── model
└── .env (Environment Variables File)
```

The `test` directory mirrors the `main` structure to keep unit tests organized (as discussed in section 5). Following these practices ensures a clean, structured, and easily maintainable project, while also avoiding issues related to package visibility.

Figure 14 shows the interaction between different parts in the system, designed to ensure a clear separation of concerns and to promote modularity. Each package interacts with other packages to perform specific tasks. This structured approach to package interaction ensures that each component of the system has a well-defined responsibility, making the codebase easier to navigate, understand, and extend. It also facilitates collaboration among team members, as each package can be developed and tested independently.

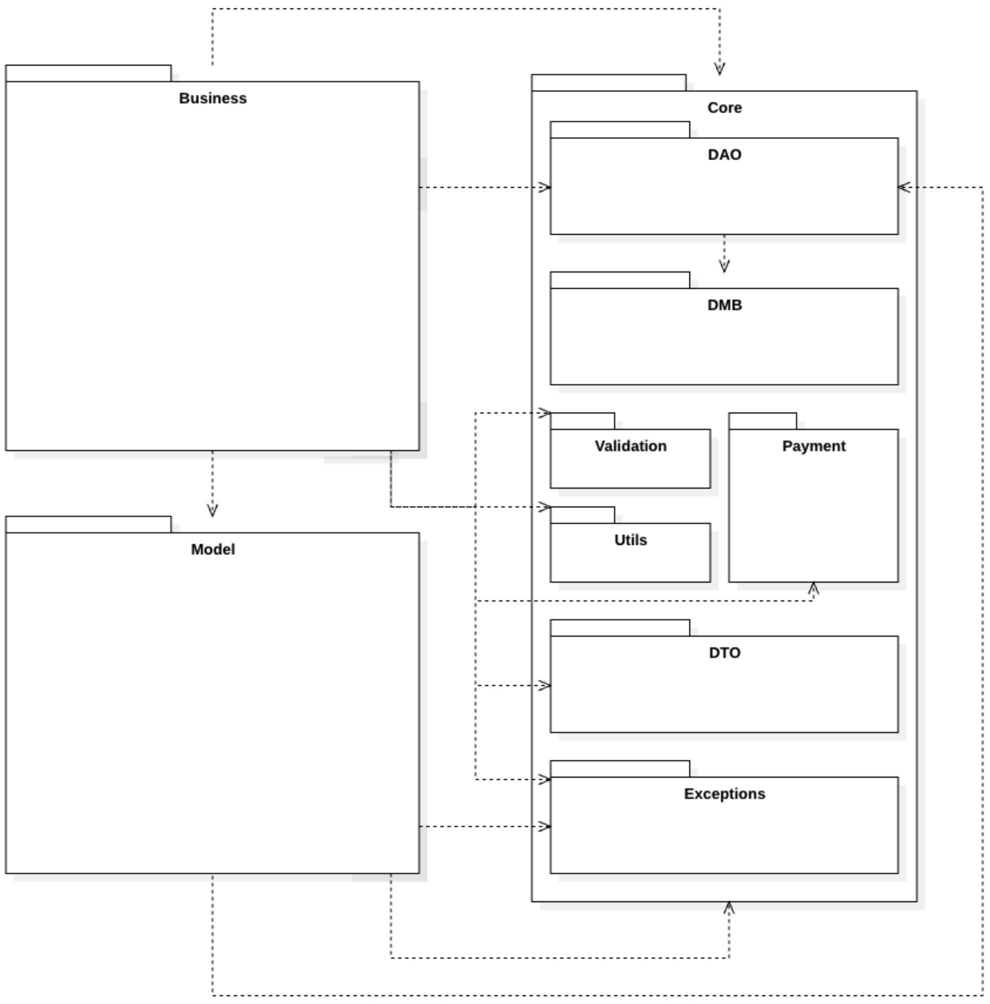


Figure 14: Interactions Between Packages

4.3.1 Domain Model

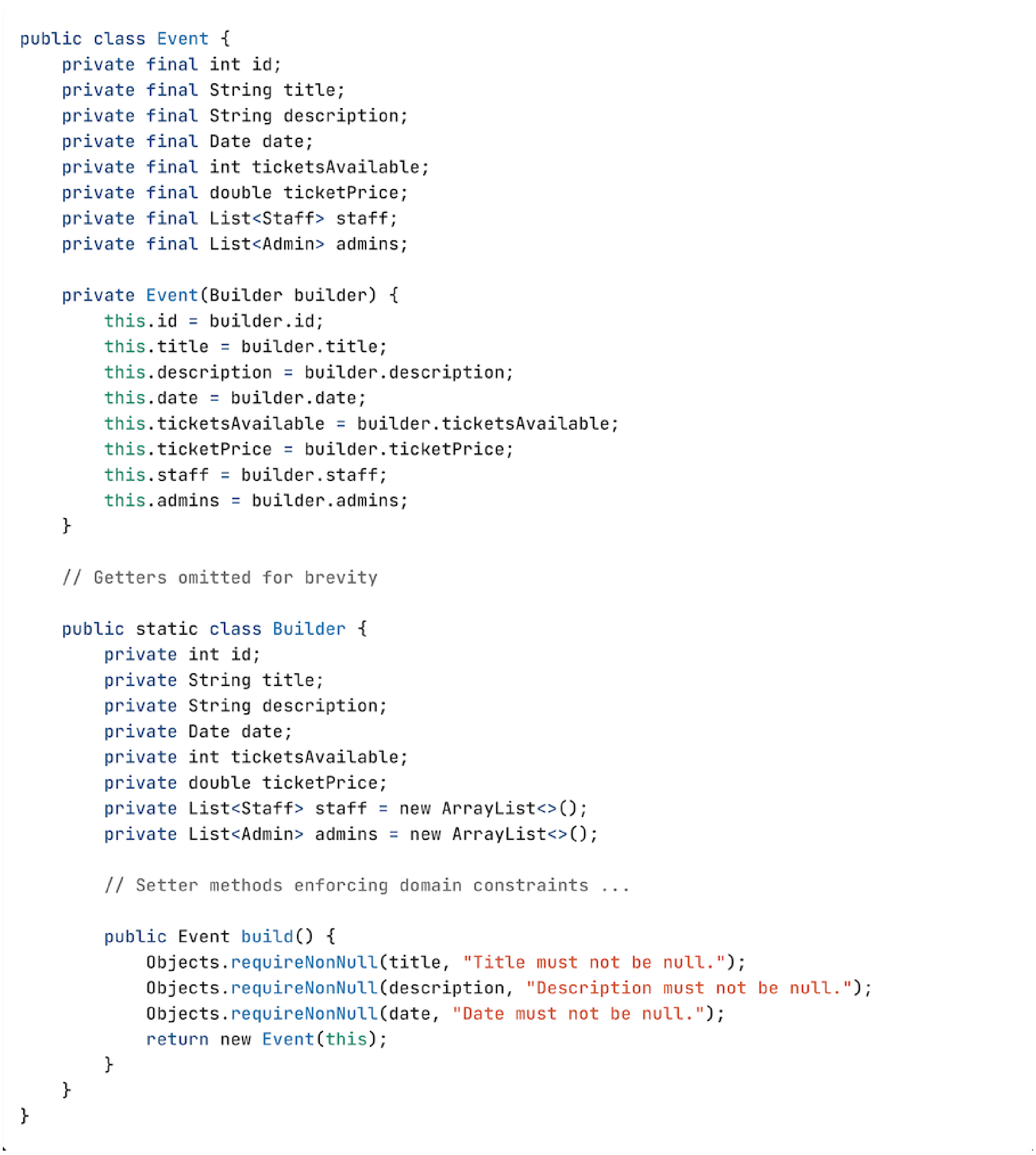
Figure 16 illustrates the key entities and relationships within the domain layer. In keeping with *Domain-Driven Design*, each class encapsulates specific invariants and behaviors that model real-world scenarios.

- **Event and Its Builder**

  `Event` is the central entity responsible for modeling a scheduled occasion. It holds core attributes such as the `id`, `title`, `description`, `date`, `ticketsAvailable`, and `ticketPrice`. From a domain perspective, this class ensures the integrity of its invariants, including non-empty titles/descriptions, valid (future) dates, and non-negative ticket counts or prices.

- **Builder Pattern:** To avoid the "telescoping constructor" anti-pattern, `Event` employs a static `Builder` class. The builder enforces domain constraints at construction time, and once fields are validated, `build()` produces a correctly initialized `Event` instance.

Below is a simplified code snippet:



```java
public class Event {
    private final int id;
    private final String title;
    private final String description;
    private final Date date;
    private final int ticketsAvailable;
    private final double ticketPrice;
    private final List<Staff> staff;
    private final List<Admin> admins;

    private Event(Builder builder) {
        this.id = builder.id;
        this.title = builder.title;
        this.description = builder.description;
        this.date = builder.date;
        this.ticketsAvailable = builder.ticketsAvailable;
        this.ticketPrice = builder.ticketPrice;
        this.staff = builder.staff;
        this.admins = builder.admins;
    }

    // Getters omitted for brevity

    public static class Builder {
        private int id;
        private String title;
        private String description;
        private Date date;
        private int ticketsAvailable;
        private double ticketPrice;
        private List<Staff> staff = new ArrayList<>();
        private List<Admin> admins = new ArrayList<>();

        // Setter methods enforcing domain constraints ...

        public Event build() {
            Objects.requireNonNull(title, "Title must not be null.");
            Objects.requireNonNull(description, "Description must not be null.");
            Objects.requireNonNull(date, "Date must not be null.");
            return new Event(this);
        }
    }
}
```

Figure 15: Event Builder

- **Associations with Admin and Staff:** As illustrated in Figure 16, an `Event` can be associated with multiple `Admin` entities (1..∗ cardinality) and zero or more `Staff` entities (0..∗ cardinality). In code, these relationships manifest as:
  - `List<Admin> admins` – Links the event to the administrators responsible for its oversight.
  - `List<Staff> staff` – Refers to staff members who have operational duties at the event.

  By storing these lists directly in `Event`, we honor the domain requirement that an `Event` can track which admins and staff are currently attached. Additional role-specific logic is delegated to the specialized classes (`Admin`, `Staff`).
- **User and Role Specializations:**
  - `User`: The base class with `id`, `name`, `surname`, `passwordHash`, and `email`.
  - `Admin`, `Staff`, and `Guest` extend `User`, each adding role-specific associations. For instance, `Staff` and `Admin` have an `eventId` reference to the event where the staff member can perform verifications, whereas the admin has privileges to create, update, or delete an event.

Thanks to this approach we prevent other layers (e.g., controllers) from introducing domain-specific rules. For example, `Event` is in charge of tracking ticket availability, `Ticket` ensures correct usage flags, and `VerifySession` coordinates states for the validation process. Each class encapsulates its own invariants (like non-negative ticket counts), ensuring consistency is preserved no matter which application services or controllers use the operations on these objects.

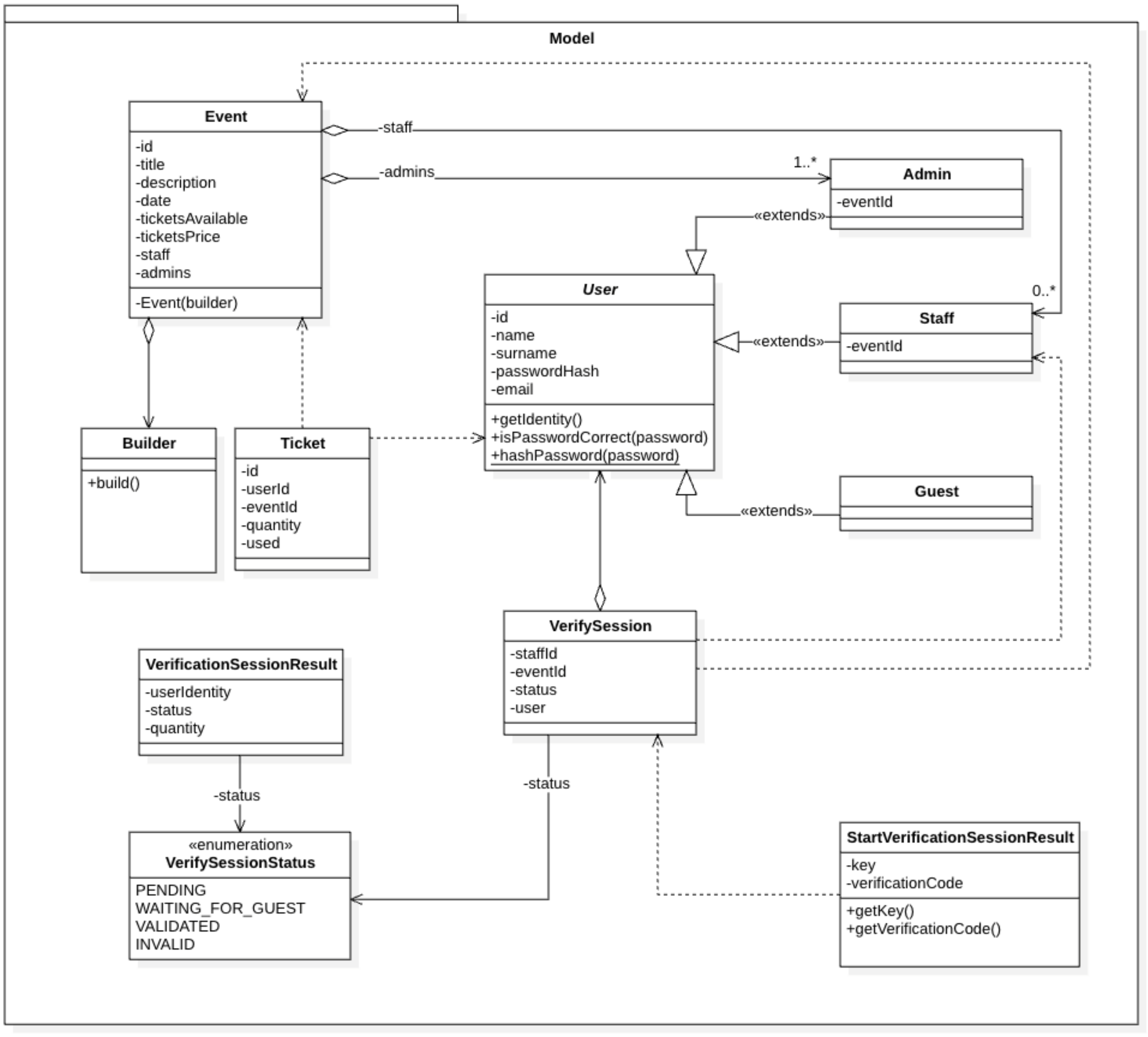


Figure 16: Domain Model

4.3.2 Business Logic Implementation

The business logic of the system adheres to the same approach in which each primary actor `Admin`, `Staff`, `Guest`, is served by a dedicated controller. As illustrated in Figure 19, the `ApplicationManager` class takes on the role of a centralized *injector*: it instantiates the required *services* and *data access objects* (DAOs) at runtime and provides these dependencies to the corresponding controllers. This design has two main motivations:

- **One-Controller-Per-Actor:** Dividing the controllers by actor reflects the principle of DDD. Each actor, such as `Admin` or `Guest`, has a distinct set of responsibilities and interactions with the domain (for example, creating an event vs. buying a ticket). Keeping these domains in separate classes helps ensure that each controller has a well-defined scope, making the code easier to understand, maintain, and extend.
- **Centralized Dependency Injection:** The `ApplicationManager` centralizes the creation of services (e.g., `AuthService`, `VerifySessionService`) and DAOs, then injects them into controllers. This arrangement ensures that controllers do not instantiate their own dependencies; instead, they receive *already-constructed* objects with clearly defined interfaces.

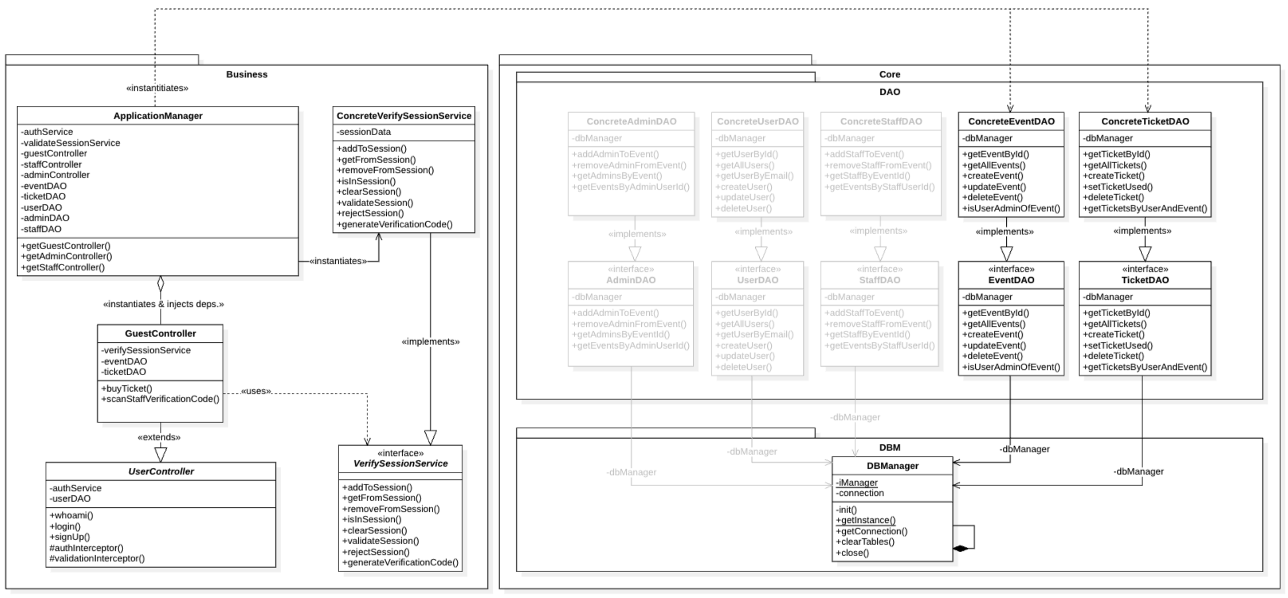
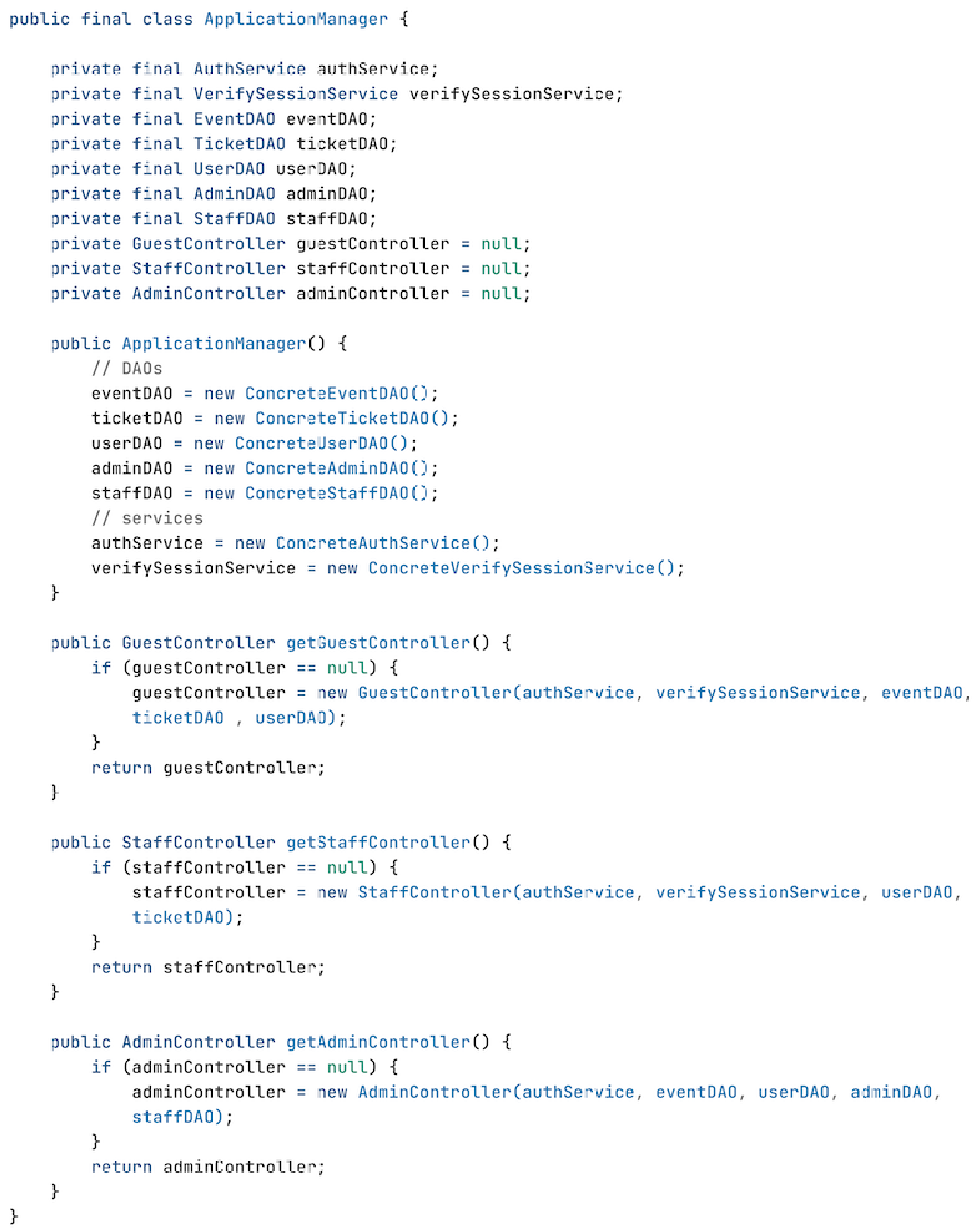


Figure 17: UML Example of Dependency Injection

Because each controller is **stateless**, it does not hold any in-memory data between method calls. However, the `ConcreteVerifySessionService` maintains a **stateful** data structure to keep track of ongoing verification sessions that link `Staff` and `Guest` information. By using `JWTUtility` internally, the `VerifySessionService` also generates secure verification codes, coupling these codes to a particular session. Consequently, whenever the `GuestController` scans a `Staff` QR code, both controllers interact with the *same* stateful service instance.



```
public final class ApplicationManager {

    private final AuthService authService;
    private final VerifySessionService verifySessionService;
    private final EventDAO eventDAO;
    private final TicketDAO ticketDAO;
    private final UserDAO userDAO;
    private final AdminDAO adminDAO;
    private final StaffDAO staffDAO;
    private GuestController guestController = null;
    private StaffController staffController = null;
    private AdminController adminController = null;

    public ApplicationManager() {
        // DAOs
        eventDAO = new ConcreteEventDAO();
        ticketDAO = new ConcreteTicketDAO();
        userDAO = new ConcreteUserDAO();
        adminDAO = new ConcreteAdminDAO();
        staffDAO = new ConcreteStaffDAO();
        // services
        authService = new ConcreteAuthService();
        verifySessionService = new ConcreteVerifySessionService();
    }

    public GuestController getGuestController() {
        if (guestController == null) {
            guestController = new GuestController(authService, verifySessionService, eventDAO,
            ticketDAO , userDAO);
        }
        return guestController;
    }

    public StaffController getStaffController() {
        if (staffController == null) {
            staffController = new StaffController(authService, verifySessionService, userDAO,
            ticketDAO);
        }
        return staffController;
    }

    public AdminController getAdminController() {
        if (adminController == null) {
            adminController = new AdminController(authService, eventDAO, userDAO, adminDAO,
            staffDAO);
        }
        return adminController;
    }
}
```

Figure 18: Application Manager Class

This centralized *dependency injection*, rather than creating a new service for each controller, constructs exactly one instance of the `ConcreteVerifySessionService` and injects it into all controllers that require it. As a result, each controller call references the same service instance and, by extension, the same session state.

Likewise, the `AuthService` (implemented by `ConcreteAuthService`) and the corresponding DAOs (as shown in Figure 17) are provided to all controllers through the `ApplicationManager` (Figure 18), tests can *replace* those real implementations with mock objects. This allows each business flow, such as event creation or ticket validation, to be tested in isolation, verifying that the controller's logic responds correctly to various service or DAO outcomes. (See section 5).

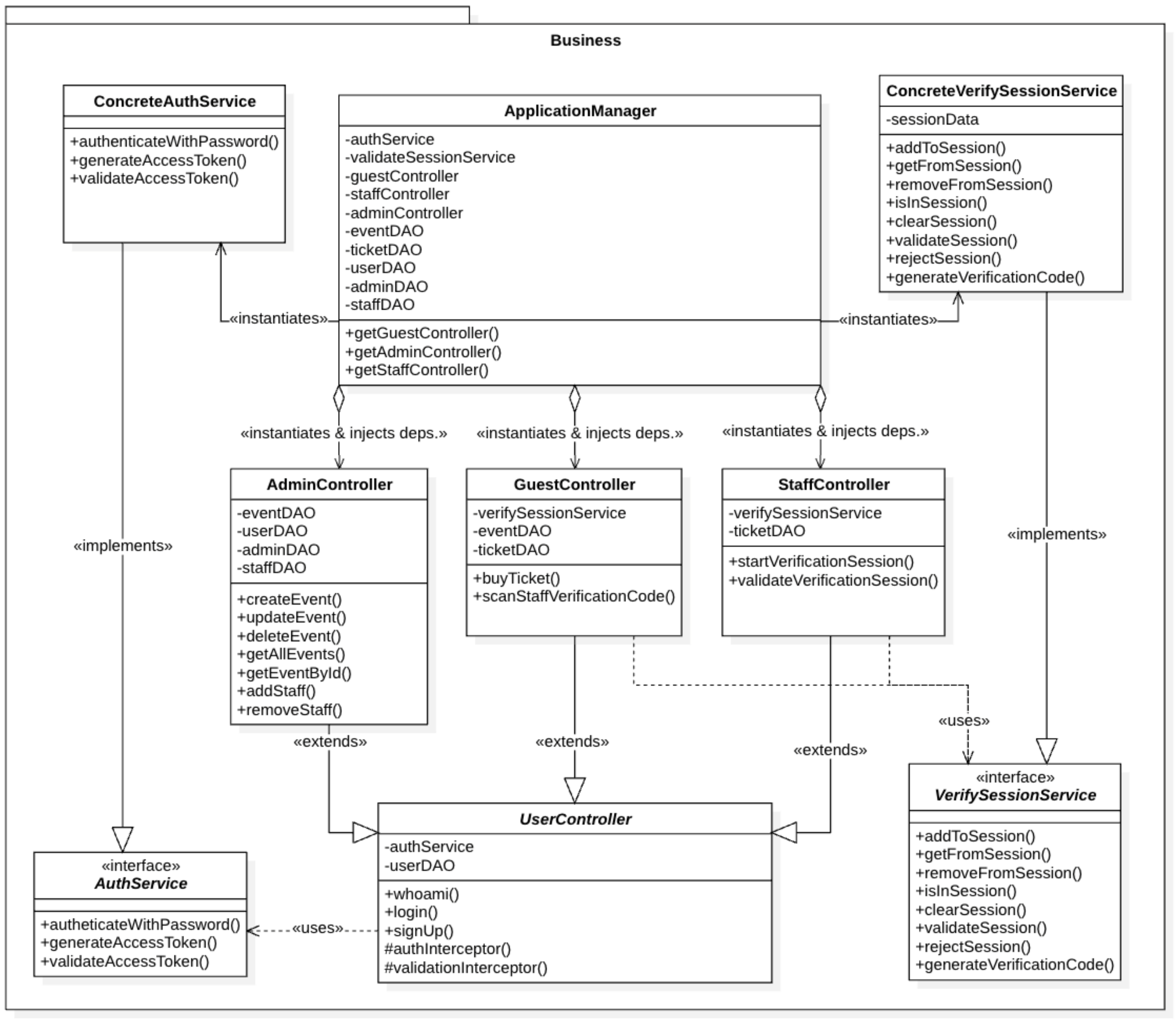


Figure 19: Business Logic

To enhance reusability and maintainability, the controllers are kept as thin as possible. Potentially reusable code that would traditionally reside in controllers has been extracted into dedicated services or directly into the `Core` package (see Figure 14). For example, common operations such as authentication, validation processing are encapsulated within services like `AuthService` and `ValidationService`, or payment that is encapsulated in a package for itself. This approach not only reduces code duplication but also ensures that the logic can be reused across different parts of the application, such as in different controllers or even in the model layer.

An important note about controllers in this implementation is that they are designed for an early prototype rather than a final, production-ready solution. As a result, some operations are intentionally simplified or not fully optimized. For instance, in the current implementation of ticket verification, the staff member must first initiate a verification session using the controller function `startVerificationSession`. Once the user scans their code, the staff must then manually retrieve the results by calling another controller method, `validateVerificationSession`. However, in a real-world system, using for instance an HTTP-based approach, this process could be implemented in multiple ways to improve efficiency. The current prototype follows a polling mechanism, where the staff continuously checks for verification results. Alternatively, a more robust solution could leverage **WebSocket** connections, allowing the staff to receive real-time success or failure events without needing to call another function or endpoint manually. This simplified approach was chosen to create a functional prototype.

4.3.3 Core Package

The Core package (illustrated in Figure 20) contains components that do not belong to the Business Logic or the Domain Model. It includes the data access layer, all Data Transfer Objects (DTOs) used for client-server

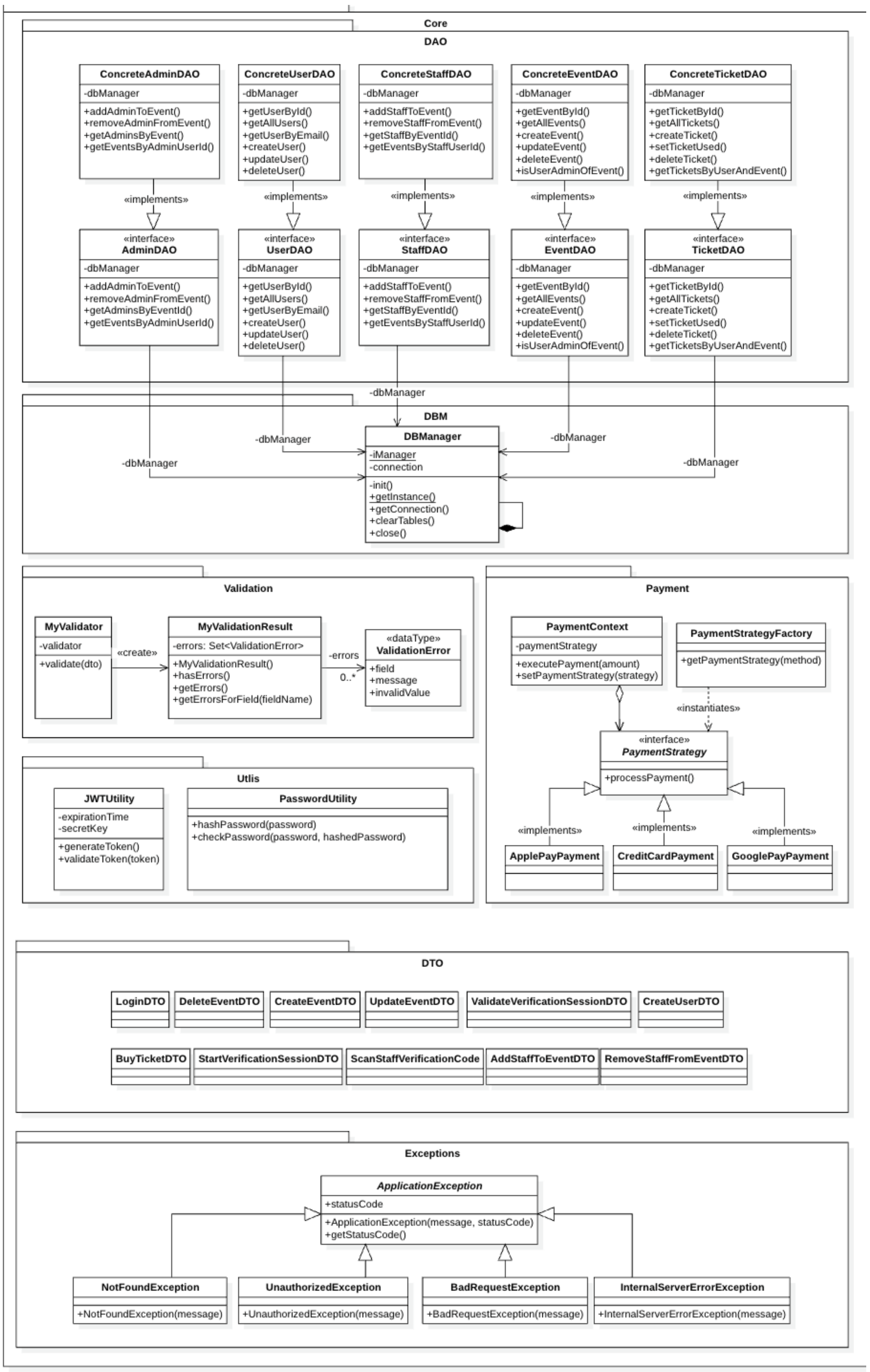


Figure 20: Core Package

interactions, and a validator module for verifying the integrity of DTOs. Additionally, the Core package provides utility functions for tasks such as generating password hashes and JSON Web Tokens (JWS, along with commonly used exception objects. A dedicated Payment subsystem contains the logic structure (currently not implemented) required to process payments. The **Strategy design pattern** is utilized to support multiple payment methods, including Google Pay, Apple Pay, and Credit Card payments. The creation of a specific `PaymentStrategy` instance is delegated to a **Simple Factory**, which generates the appropriate strategy object based on a string identifier passed via the controller DTO. This strategy instance is then used within a `PaymentContext` to manage the payment process in the `GuestController`.

4.3.4 Data Access Layer

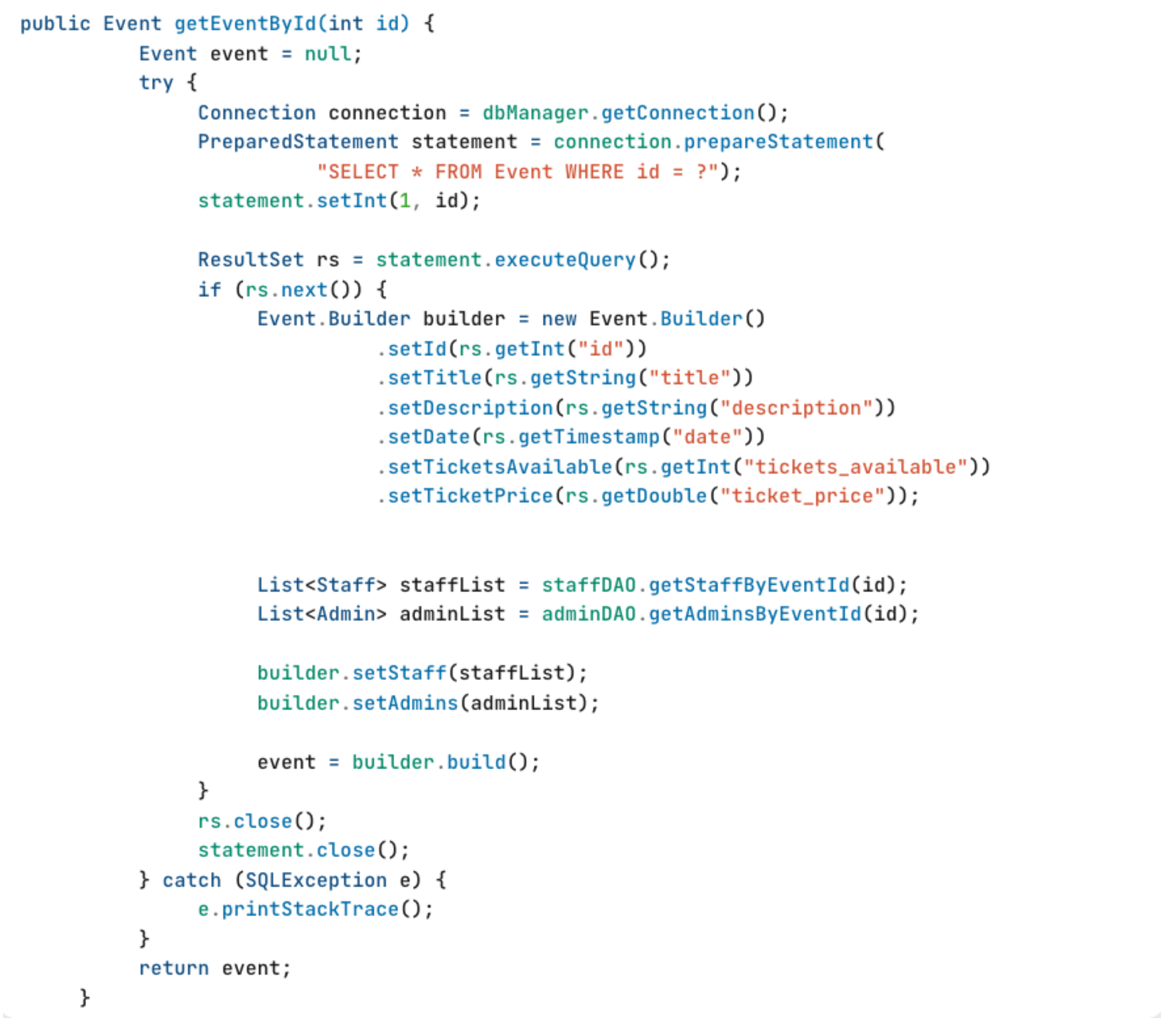
The **DAO pattern** (Data Access Object) is a structural design approach that provides an abstraction layer for database interactions, effectively decoupling the application's business logic from its persistence layer. By encapsulating database access logic in dedicated DAO classes, this pattern ensures centralized and modular handling of database operations, resulting in cleaner, more maintainable, and easily testable code. Furthermore, it reduces the coupling between the business and data access layers, promoting scalability and flexibility in application design.

In this implementation, the DAO pattern works in conjunction with JDBC (Java Database Connectivity), a Java-based API that provides a standardized interface for executing SQL queries, retrieving results, and managing database connections. JDBC bridges the application and the relational database, ensuring efficient database interactions while supporting performance and scalability.

Initially, the project used a schema that directly assigned each `DAO` to a corresponding database entity. Although this approach appeared straightforward, it revealed significant limitations in terms of flexibility and alignment with the application's business requirements. The rigid mapping to the database schema restricted the DAOs' ability to address the distinct needs of the system's user roles and responsibilities. Consequently, the DAO layer was refactored to better align with the application's domain model, focusing on the responsibilities and interactions of its classes rather than the database structure. This redesign not only improved flexibility and maintainability but also ensured that the DAOs better reflected the application's business logic.

Each class implements the core CRUD operations—create, read, update, and delete—and relies on a shared `DBManager` for database connection management. This ensures a clear separation of concerns, as each DAO is responsible for its domain-specific SQL logic while leveraging a centralized access point for connections. The DAO's used are:

- **EventDAO**: Manages operations related to the Event entity in the database including creation, retrieval, updating, and deletion. Additionally, utilizes composition with the `AdminDAO` and `StaffDAO` (Figure 21), delegating responsibilities like staff and admin management. This approach promotes code reuse and ensures a modular and maintainable structure.



```
public Event getEventById(int id) {
          Event event = null;
          try {
               Connection connection = dbManager.getConnection();
               PreparedStatement statement = connection.prepareStatement(
                         "SELECT * FROM Event WHERE id = ?");
               statement.setInt(1, id);

               ResultSet rs = statement.executeQuery();
               if (rs.next()) {
                    Event.Builder builder = new Event.Builder()
                              .setId(rs.getInt("id"))
                              .setTitle(rs.getString("title"))
                              .setDescription(rs.getString("description"))
                              .setDate(rs.getTimestamp("date"))
                              .setTicketsAvailable(rs.getInt("tickets_available"))
                              .setTicketPrice(rs.getDouble("ticket_price"));


                    List<Staff> staffList = staffDAO.getStaffByEventId(id);
                    List<Admin> adminList = adminDAO.getAdminsByEventId(id);

                    builder.setStaff(staffList);
                    builder.setAdmins(adminList);

                    event = builder.build();
               }
               rs.close();
               statement.close();
          } catch (SQLException e) {
               e.printStackTrace();
          }
          return event;
     }
```

Figure 21: Example of DAO composition in getEventById method

- **StaffDAO**: Dedicated to managing the relationship between staff members and events, provides methods for adding, removing, and retrieving staff associations. I also enables bidirectional mapping with methods like `getStaffByEventId` and `getEventsByStaffUserId`.
- **AdminDAO**: Handles interactions related to admin users and their privileges over events, central to granting administrative privileges to users. Additionally to providing methods for adding and removing admin members, the `getAdminsByEventId` method retrieves all administrators for a given event.
- **UserDAO**: Handles core user operations, including authentication and retrieval by Id or email. It integrates with `AdminDAO` and `StaffDAO` for role assignments, ensuring a unified and modular approach.
- **TicketDAO**: Manages ticket lifecycle operations, from creation to validation. It maintains data consistency by ensuring tickets are correctly associated with users and events. The `setTicketUsed` method tracks ticket usage, supporting event access control and fraud prevention.

4.4 Database Implementation

The Entity-Relationship (ER) model presented in Figure 12 has been implemented using the PostgreSQL Database Management System (DBMS). Entities were transformed into tables, and two additional tables (`admin` and `staff`) were created to represent the many-to-many relationships between events and users. Figure 12 illustrates the schema of the final relational model, highlighting the foreign key constraints, primary keys, and the domain of each attribute. All the tables are in Boyce–Codd Normal Form (BCNF); no further normalization is required. Regarding indexing, the email field is a suitable candidate as it is used for both login purposes and staff management to retrieve users. However, since it is set as UNIQUE, the DBMS automatically creates an index. Therefore, the only additional index created is:

Code 1: Indexes

```
CREATE INDEX idx_ticket_event_user ON Ticket(event_id, user_id);
```

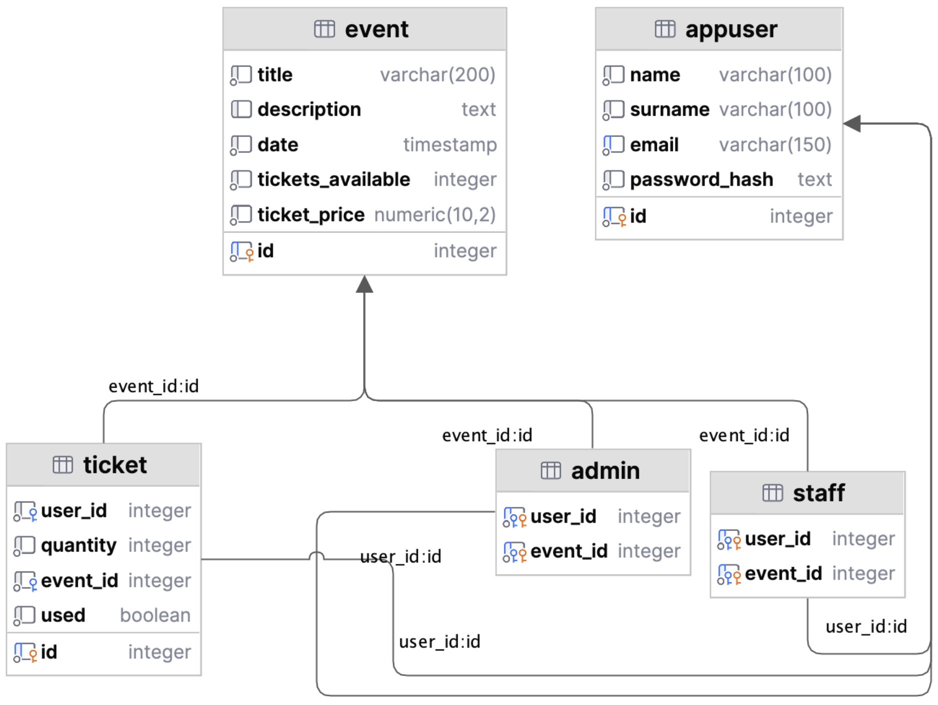


Figure 22: Database diagram

4.4.1 Database Connection

The database connection in this application is managed using the **Singleton pattern**, a design choice made after evaluating various alternatives during the initial phases of the project. This pattern ensures that only one instance of the database connection exists throughout the application lifecycle, providing a centralized and globally accessible access point. This is achieved through a private constructor, which prevents the creation of multiple instances, and a static method that returns the single instance of the object. Its simplicity and alignment with the application requirements made it the most suitable choice.

Although the Singleton pattern is effective, it does have potential limitations, particularly in scenarios requiring simultaneous interactions with the instance, which could lead to contention. However, this is not a concern for our application, as the design does not involve concurrent database operations.

During the design phase, we also considered using the Proxy pattern with a connection pool. This alternative involved creating an intermediary between the application and the database connections, managing a pool of connections that would allow efficient allocation and reuse of connections, along with features such as automatic reconnection. This approach is particularly advantageous in multi-threaded environments where concurrent access to the database is frequent. Ultimately, we decided against this approach. The primary reason was the lack of concurrent database access in our application, which rendered the connection pool unnecessary.

5 Testing Strategy

This section outlines the testing strategy adopted for the project. The development process followed **Test-Driven Development** (TDD), ensuring robust code throughout the implementation phase. In fact, this approach significantly reduced errors, such as versioning conflicts, which can arise during parallel development. The test suite consists of a total of **106 tests**, classified as follows:

- **Unit Tests**: 76 tests.
- **Integration Tests**: 30 tests.

5.1 Test Coverage

Test coverage metrics were generated to assess the effectiveness of the implemented tests. These metrics provide insights into the extent to which the test suite exercises the codebase. Tables 1 and 2 present the coverage percentages across **classes**, **methods**, and **lines of code**. As shown in Table 1, good results were achieved: **92.5% of all classes** and **77.8% of methods** are covered by tests. Furthermore, Table 2 provides a detailed breakdown of the the coverage metrics for each individual package in the project.

Table 1: Code Coverage Summary

Package	**Class %**	**Method %**	**Line %**
all classes	92.5% (49/53)	77.8% (193/248)	73.7% (750/1018)

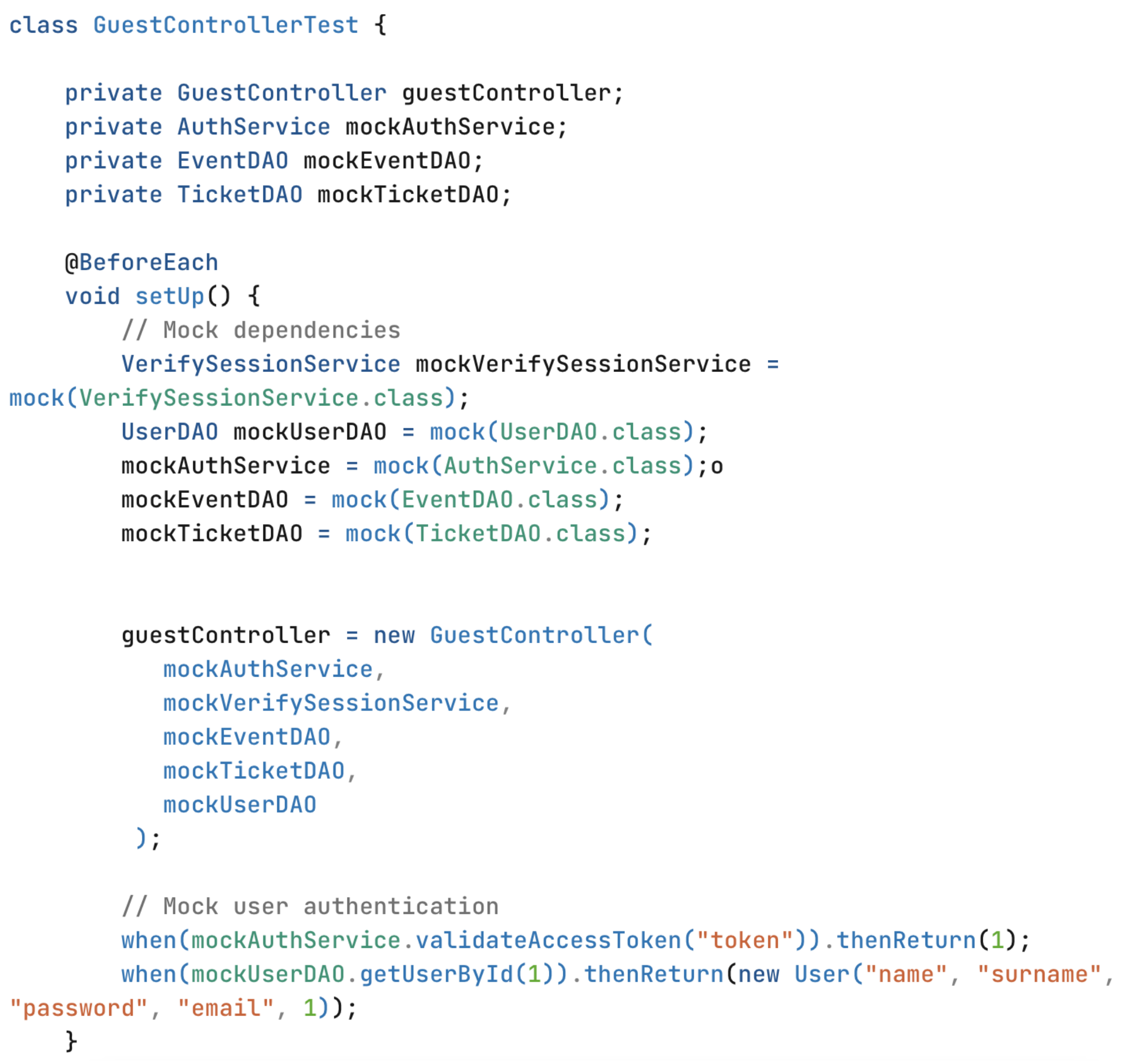
Table 2: Code Coverage Breakdown

Package	**Class %**	**Method %**	**Line %**
org.swe	50% (1/2)	25% (1/4)	57.1% (4/7)
org.swe.business	100% (7/7)	100% (38/38)	92.6% (225/243)
org.swe.core	100% (1/1)	50% (1/2)	57.1% (4/7)
org.swe.core.DAO	100% (5/5)	71% (22/31)	58.9% (251/426)
org.swe.core.DBM	100% (1/1)	85.7% (6/7)	51.3% (20/39)
org.swe.core.DTO	90.9% (10/11)	61.9% (39/63)	71.3% (62/87)
org.swe.core.exceptions	100% (5/5)	83.3% (5/6)	85.7% (6/7)
org.swe.core.payment	80% (4/5)	72.7% (8/11)	68.4% (13/19)
org.swe.core.utils	100% (2/2)	66.7% (4/6)	90.5% (19/21)
org.swe.core.validation	100% (3/3)	88.9% (8/9)	86.4% (19/22)
org.swe.model	90.9% (10/11)	85.9% (61/71)	90.7% (127/140)

5.2 Unit Tests

Unit tests were written for nearly all classes (49 out of 53) to verify their correct functionality. Since unit tests are typically numerous and must be executed frequently, it is crucial that they are both fast and deterministic. This requirement precludes the use of a real database connection, as it would significantly slow down the test suite and introduce potential failure points unrelated to the component being tested. Instead, unit tests were designed to isolate individual components, achieved through the use of mocking. The libraries `JUnit` and `Mockito` were employed for writing and managing these tests.

Dependency Injection, already discussed in the previous sections regarding the Business Logic, suits very well for this job; in fact, the adoption of this pattern was in part motivated by its suitability for testing. For example, in the unit tests of controllers, the real dependencies, DAOs and services, are mocked and directly injected in the constructors of the controllers. This separation of responsibilities offers a great flexibility that cannot be achieved with other solutions. Image 23, showing the setup method for the unit tests of `GuestController`, offers a representation of the technique just discussed. Before instantiating a new `GuestController` for the test, all the services (`VerifySessionService`, `AuthService`) and DAOs (`UserDAO`, `EventDAO`, and `TicketDAO`) are mocked using `Mockito`. This fakes the behavior of the individual methods of these classes, as shown in the same image for the user authentication. Then, using Dependency Injection, the mocked dependencies are passed to the new instance of the controller.



```
class GuestControllerTest {

    private GuestController guestController;
    private AuthService mockAuthService;
    private EventDAO mockEventDAO;
    private TicketDAO mockTicketDAO;

    @BeforeEach
    void setUp() {
        // Mock dependencies
        VerifySessionService mockVerifySessionService =
mock(VerifySessionService.class);
        UserDAO mockUserDAO = mock(UserDAO.class);
        mockAuthService = mock(AuthService.class);o
        mockEventDAO = mock(EventDAO.class);
        mockTicketDAO = mock(TicketDAO.class);


        guestController = new GuestController(
           mockAuthService,
           mockVerifySessionService,
           mockEventDAO,
           mockTicketDAO,
           mockUserDAO
         );

        // Mock user authentication
        when(mockAuthService.validateAccessToken("token")).thenReturn(1);
        when(mockUserDAO.getUserById(1)).thenReturn(new User("name", "surname",
"password", "email", 1));
    }
```

Figure 23: Example of injection of mocked dependencies

5.3 Integration Tests

Integration tests ensure the correct behavior of combined components by verifying the interactions between different system classes. Unlike unit tests, fast execution is not a priority, so the entire system was tested using a real database instance to simulate a production-like environment. Three main categories of integration tests were identified:

- **Authentication**: Verifies the entire process of user creation and login.
- **Event Management**: Tests the entire event life-cycle, including creation, updates, staff management, and tickets purchase.
- **Ticket Verification**: Verifies the correct functioning of the ticketing system.

To maintain an isolated environment between tests, all database relations are cleared during the setup and tear-down methods. In a production-ready implementation, it is crucial to have two separate database instances for production and testing.

6 Notes

6.1 Source Code

The complete source code for this project is publicly available on GitHub and can be accessed at the following repository: `https://github.com/NiccoloCase/ddd-secure-ticketing-system`.

6.2 Environment Configuration

To run this project successfully, a .env file must be created in the root directory with the required environment variables:

```
JWT_SECRET=<your_secret_key>
DB_URL=<your_database_url>
```

6.3 AI Assistance (LLM Usage) During Development

We have utilized AI-powered tools such as GitHub Copilot and large language models (LLMs) like ChatGPT and Claude as tools in our development and conceptual validation processes. GitHub Copilot has effectively automated repetitive coding tasks, such as generating getters and setters or refactoring code, enabling us to focus more on higher-order development activities. We also used LLMs to verify theoretical concepts, validate UML designs (through the PlantUML diagrams code), and refine our ideas. These tools have proven invaluable as extensions of our workflow, assisting us not only in coding, but also in critically evaluating and enhancing our design methodologies. That said, these tools have always been used in a controlled manner, with us maintaining a dominant and critical stance over the outputs, code in particular. The specific tools used varied according to individual preferences: some team members preferred ChatGPT (GPT-4o) for general tasks such as brainstorming, while Claude (Claude 3.5 Sonnet) performed better in coding-related tasks. DeepSeek (DeepSeek-R1) was also tested for those operations requiring deep analytical thinking (chain of thought), but it was not significantly adopted in our workflow.

6.3.1 UML-Driven Development vs. Code-Driven UML

A common misconception is that UML diagrams can be effectively generated from existing code. While tools like ChatGPT generating PlantUML code can extract class names and attributes—elements that are explicitly defined in the code—they often fail to accurately capture the relationships between classes and packages. Specifically, we observed that generated diagrams struggled to distinguish between association, aggregation, and composition, leading to misleading representations of system architecture. Figure 24 shows a UML diagram generated by a large language model (LLM) from the codebase of this project. While minor errors might be forgivable, the primary issue is that diagrams like this fail to provide useful insights.

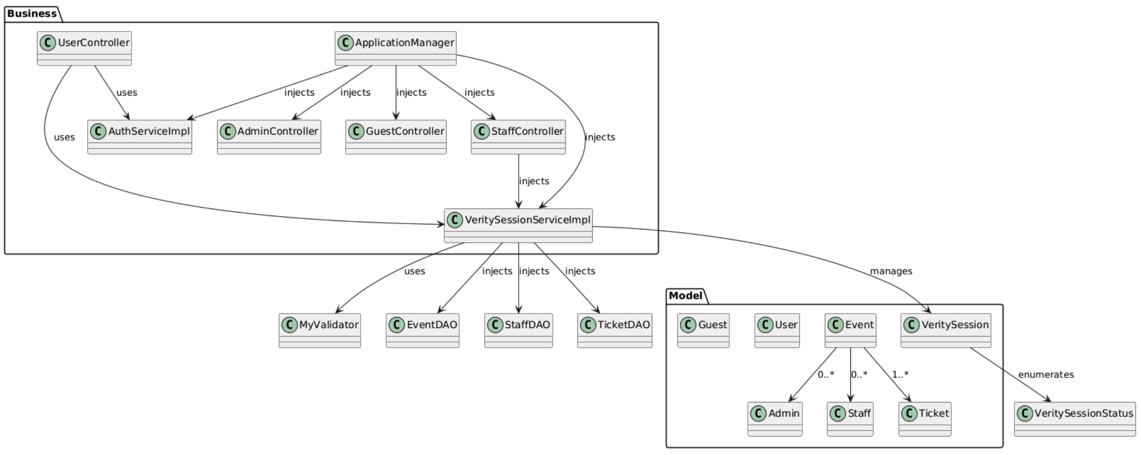
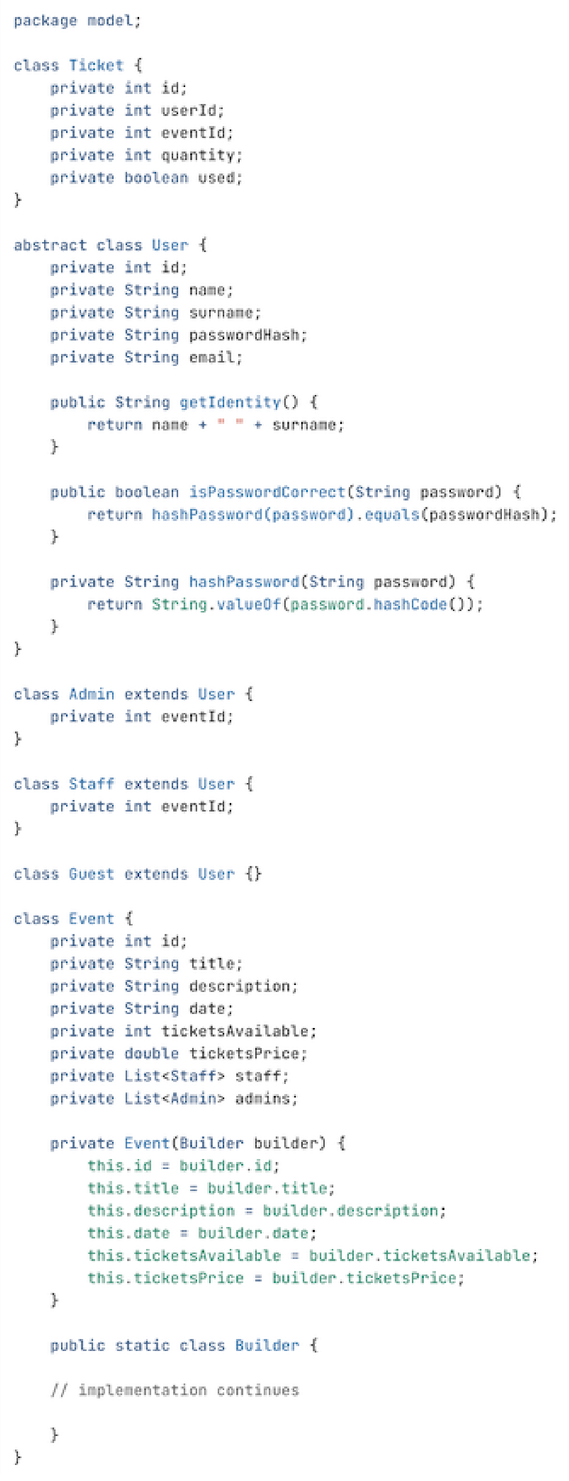


Figure 24: Wrong UML diagram generated from existing code.

Prompt: "This are the business and model packages containing all class implementations (code). Generate a UML diagram representing the relationships and structure based on the provided code."

Response: "To provide you with the requested PlantUML diagram, I need to translate your class diagram's connections and layout into PlantUML syntax. Let me create this for you."



```java
package model;

class Ticket {
    private int id;
    private int userId;
    private int eventId;
    private int quantity;
    private boolean used;
}

abstract class User {
    private int id;
    private String name;
    private String surname;
    private String passwordHash;
    private String email;

    public String getIdentity() {
        return name + " " + surname;
    }

    public boolean isPasswordCorrect(String password) {
        return hashPassword(password).equals(passwordHash);
    }

    private String hashPassword(String password) {
        return String.valueOf(password.hashCode());
    }
}

class Admin extends User {
    private int eventId;
}

class Staff extends User {
    private int eventId;
}

class Guest extends User {}

class Event {
    private int id;
    private String title;
    private String description;
    private String date;
    private int ticketsAvailable;
    private double ticketsPrice;
    private List<Staff> staff;
    private List<Admin> admins;

    private Event(Builder builder) {
        this.id = builder.id;
        this.title = builder.title;
        this.description = builder.description;
        this.date = builder.date;
        this.ticketsAvailable = builder.ticketsAvailable;
        this.ticketsPrice = builder.ticketsPrice;
    }

    public static class Builder {

    // implementation continues

    }
}
```

Figure 25: Code generated from UML diagram.

Prompt: "This is the UML diagram of the Model package (image) and the system's core statement (text) Generate the corresponding code, ensuring that the class relationships align with the given UML structure."

Response: "Based on the provided UML diagram and system description, here is the generated code for the Model package:"

Nevertheless, starting with a well-defined UML diagram provides a structured blueprint that ensures clarity and consistency throughout development. By designing the system before implementation, we were able to validate our approach and make early adjustments.

7 Documentation

7.1 Data Transfer Objects

DTO 1: `CreateUserDTO`

Arguments:
- `name: String`
- `surname: String`
- `email: String`
- `password: String`

DTO 2: `LoginDTO`

Arguments:
- `email: String`
- `password: String`

DTO 3: `CreateEventDTO`

Arguments:
- `title: String`
- `description: String`
- `date: Date`
- `ticketsAvailable: int`
- `ticketPrice: double`

DTO 4: `DeleteEventDTO`

Arguments:
- `eventId: Integer`

DTO 5: `UpdateEventDTO`

Arguments:
- `eventId: Integer`
- `title: String`
- `description: String`
- `date: Date`
- `ticketsAvailable: int`
- `ticketPrice: double`

DTO 6: `AddStaffToEventDTO`

Arguments:
- `eventId: Integer`
- `staffEmail: String`

DTO 7: `RemoveStaffFromEventDTO`

Arguments:
- `eventId: Integer`
- `staffEmail: String`

DTO 8: `BuyTicketDTO`

Arguments:
- `eventId: Integer`
- `quantity: Integer`
- `paymentMethod: String`

DTO 9: `StartVerificationSessionDTO`

Arguments:
- `eventId: Integer`

DTO 10: `ScanStaffVerificationCodeDTO`

Arguments:
- `code: String`

DTO 11: `ValidateVerificationSessionDTO`
Arguments:
- `sessionKey: String`

7.2 Unit Tests

7.2.1 UserControllerTest

Unit test 1: `loginShouldReturnTokenWhenValidCredentials`

Unit test 2: `loginShouldThrowBadRequestExceptionWhenUserNotFound`

Unit test 3: `signupShouldReturnTokenWhenUserIsCreated`

Unit test 4: `signupShouldThrowBadRequestExceptionWhenUserAlreadyExists`

Unit test 5: `signupShouldThrowBadRequestExceptionWhenEmailFormatIsInvalid`

7.2.2 StaffControllerTest

Unit test 6: `startVerificationSessionShouldReturnSessionResponseIfValid`

Unit test 7: `startVerificationSessionShouldThrowUnauthorizedExceptionIfTokenIsInvalid`

Unit test 8: `validateVerificationSessionShouldThrowIfSessionDoesNotExist`

Unit test 9: `validateVerificationSessionShouldThrowIfSessionBelongsToAnotherStaff`

Unit test 10: `validateVerificationSessionShouldThrowIfSessionIsAlreadyValidated`

Unit test 11: `validateVerificationSessionShouldThrowIfSessionIsPendingOrNoGuest`

Unit test 12: `validateVerificationSessionShouldThrowIfNoTicketsFound`

Unit test 13: `validateVerificationSessionShouldThrowIfAllTicketsUsed`

7.2.3 GuestControllerTest

Unit test 14: `buyTicketShouldReturnTicketIfPurchaseIsSuccessful`

Unit test 15: `buyTicketShouldThrowNotFoundExceptionIfEventDoesNotExist`

Unit test 16: `buyTicketShouldThrowBadRequestExceptionIfNotEnoughTicketsAvailable`

Unit test 17: `buyTicketShouldThrowBadRequestExceptionIfPaymentFails`

Unit test 18: `buyTicketShouldThrowBadRequestExceptionIfEventUpdateFails`

Unit test 19: `buyTicketShouldThrowExceptionIfTokenValidationFails`

Unit test 20: `buyTicketShouldThrowExceptionIfPaymentMethodIsInvalid`

Unit test 21: `buyTicketShouldThrowExceptionIfPaymentGoesWrong`

7.2.4 AdminControllerTest

Unit test 22: `createEventShouldReturnAnEventIfEventCreationIsSuccessful`

Unit test 23: `createEventShouldThrowInternalServerErrorExceptionIfEventCreationFails`

Unit test 24: `deleteEventShouldReturnTrueIfEventIsDeleted`

Unit test 25: `deleteEventShouldThrowUnauthorizedExceptionIfUserIsNotAdminOfEvent`

Unit test 26: `updateEventShouldReturnTrueIfEventIsUpdated`

Unit test 27: `updateEventShouldThrowUnauthorizedExceptionIfUserIsNotAdminOfEvent`

Unit test 28: `getAllEventsShouldReturnAListOfEvents`

Unit test 29: `addStaffShouldThrowBadRequestExceptionIfStaffEmailIsInvalid`

Unit test 30: `addStaffShouldThrowUnauthorizedExceptionIfUserIsNotAdminOfEvent`

Unit test 31: `addStaffShouldThrowNotFoundExceptionIfUserIsNotFound`

Unit test 32: `addStaffShouldThrowRuntimeExceptionIfStaffIsNotAddedToEvent`

Unit test 33: `addStaffShouldReturnWithoutExceptionIfStaffIsAddedToEvent`

Unit test 34: `removeStaffShouldThrowBadRequestExceptionIfStaffEmailIsInvalid`

Unit test 35: `removeStaffShouldThrowUnauthorizedExceptionIfUserIsNotAdminOfEvent`

Unit test 36: `removeStaffShouldThrowNotFoundExceptionIfUserIsNotFound`

Unit test 37: `removeStaffShouldThrowRuntimeExceptionIfStaffIsNotRemovedFromEvent`

Unit test 38: `removeStaffShouldReturnWithoutExceptionIfStaffIsRemovedFromEvent`

7.2.5 ConcreteVerifySessionServiceTest

Unit test 39: `testAddToSession`

Unit test 40: `testUniqueSessionKeys`

Unit test 41: `testGetFromSessionWhenNotExists`

Unit test 42: `testRemoveFromSession`

Unit test 43: `testIsInSession`

Unit test 44: `testClearSession`

Unit test 45: `testValidateSession`

Unit test 46: `testVerifySessionThrowsExceptionWhenSessionNotFound`

Unit test 47: `testRejectSession`

Unit test 48: `testRejectSessionThrowsExceptionWhenSessionNotFound`

Unit test 49: `testRejectSessionWithTicketId`

Unit test 50: `testRejectSessionWithTicketIdThrowsExceptionWhenSessionNotFound`

7.2.6 JWTUtilityTest

Unit test 51: `testGenerateToken`

Unit test 52: `testValidateToken_ValidToken`

Unit test 53: `testValidateToken_ExpiredToken`

Unit test 54: `testValidateToken_InvalidToken`

7.2.7 PasswordUtilityTest

Unit test 55: `testHashPassword`

Unit test 56: `testCheckPasswordCorrect`

Unit test 57: `testCheckPasswordIncorrect`

7.2.8 MyValidatorTest

Unit test 58: `testValidDTO`

Unit test 59: `testEmptyDTO`

Unit test 60: `testInvalidDTO`

7.2.9 EventTest

Unit test 61: `testEventBuilderAndGetters`

Unit test 62: `testInvalidIdThrowsException`

Unit test 63: `testInvalidTitleThrowsException`

Unit test 64: `testInvalidDateThrowsException`

Unit test 65: `testNegativeTicketsAvailableThrowsException`

Unit test 66: `testNegativeTicketPriceThrowsException`

Unit test 67: `testBuildWithoutTitleThrowsException`

7.2.10 UserTest

Unit test 68: `testUserConstructorAndGetters`

Unit test 69: `testSetters`

7.2.11 TicketTest

Unit test 70: `testTicketConstructor_InvalidQuantity`

Unit test 71: `testSetters_ValidData`

Unit test 72: `testSetters_InvalidId`

Unit test 73: `testSetters_InvalidUserId`

Unit test 74: `testSetters_InvalidEventId`

Unit test 75: `testSetters_InvalidQuantity`

Unit test 76: `testSetters_ValidUsedFlag`

7.3 Integration Tests

7.3.1 Authentication Integration Tests

Integration test 1: `signupShouldThrowsExceptionIfValidationFails`

Integration test 2: `signupShouldReturnTokenIfSuccess`

Integration test 3: `signupShouldFailIfEmailAlreadyExists`

Integration test 4: `loginShouldReturnTokenIfSuccess`

Integration test 5: `loginShouldFailIfUserDoesNotExist`

Integration test 6: `afterLoginWhoamiShouldReturnNameAndSurname`

7.3.2 Event Management Integration Tests

Integration test 7: `createEventShouldReturnTrue`

Integration test 8: `getAllEventsShouldReturnList`

Integration test 9: `updateEventShouldReturnTrue`

Integration test 10: `notAdminUpdateShouldThrowException`

Integration test 11: `notAdminAddStaffShouldThrowException`

Integration test 12: `addStaffShouldReturnTrue`

Integration test 13: `wrongEmailStaffShouldThrowException`

Integration test 14: `notAdminShouldNotRemoveStaff`

Integration test 15: `removeStaffShouldReturnTrue`

Integration test 16: `wrongEmailRemoveStaffShouldThrowException`

Integration test 17: `notAdminDeleteShouldThrowException`

Integration test 18: `eventPurchaseShouldDecrementAvailableTickets`

Integration test 19: `eventPurchaseShouldFailIsNotEnoughTickets`

Integration test 20: `getEventByIdsShouldPopulateAdminsAndStaff`

Integration test 21: `deleteEventShouldReturnTrue`

7.3.3 Ticket Verification Integration Tests

Integration test 22: `createAdminAndAddStaff`

Integration test 23: `createEvent`

Integration test 24: `createGuestAndBuyTicket`

Integration test 25: `staffStartsVerificationSession`

Integration test 26: `guestScansStaffCodeSuccessfully`

Integration test 27: `staffValidatesSession`

Integration test 28: `staffCannotValidateSessionBelongingToAnotherStaff`

Integration test 29: `staffCannotValidateAlreadyValidatedSession`

Integration test 30: `staffCannotValidateSessionIfUserHasNoTicketsForThatEvent`