VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

Ho Chi Minh City University of Technology

Faculty of Computer Science and Engineering

DATABASE SYSTEMS (CO2013)

Assignment 2

BKinema
A Movie Ticket Booking System

Instructor: Le Duc Hoang Nam

Semester: 251
Class: CC06
Group: BKinema
Students: Nguyen Trung Nhan – 2352852
Le Pham Tien Long – 2352688
Ho Minh Nhat – 2352858
Nguyen Vinh Phu – 2352919
Pham Nam An – 2353016

HO CHI MINH CITY, DECEMBER 2025

Contents

1 Introduction

Building upon the conceptual database design established in Assignment 1, this report documents the practical implementation of the BKinema movie ticket booking system.
Following the Entity-Relationship Diagram (ERD) and relational schema designed in the previous assignment, this implementation addresses three critical aspects of database system development: **physical schema implementation**, **advanced database programming**, and **application connectivity**.
Application integration demonstrates how the database connects with a application stack. The implementation uses **NestJS** (a TypeScript-based Node.js framework) for the backend and **React.js** for the frontend, communicating with a **MySQL** database through stored procedure calls. This architecture ensures clear separation of concerns, improved security through parameterized queries, and better performance through server-side processing.
The remaining sections of this report present detailed documentation of each implementation component. Section 2 demonstrates the SQL statements for creating all database tables, including primary keys, foreign keys, and check constraints, along with sample data insertion. Section 3 documents the implementation of triggers for enforcing business rules and maintaining derived attributes. Section 4 presents stored procedures for insert, update, and delete operations with validation, as well as query procedures for data retrieval. Section 5 describes the implementation of functions with cursor operations and conditional logic. Section 6 illustrates the web application implementation, showing how frontend interfaces connect to database operations through stored procedure calls, accompanied by sequence diagrams and activity diagrams to visualize system workflows. Finally, Section 7 summarizes the achievements and lessons learned from this database implementation project.

2 Database Schema Implementation

The BKinema database consists of the following table categories:

- **User Management**: User, Customer, Staff, Membership (4 tables)
- **Theater Infrastructure**: Theater, Auditorium, Seat (3 tables)
- **Movie Information**: Movie, Director, Actor, Genre, Subtitle, Dubbing (6 tables)
- **Showtime and Bookings**: Showtime, Showtime_seat, Booking (3 tables)
- **Payments and Transactions**: Payment, Coupon, Refund (3 tables)
- **Food and Beverage**: FWB_Menu, FWB, Contains_item (3 tables)
- **Gift Relationships**: Give, Send_gift (2 tables)

2.1 Core Entity Tables

2.1.1 User Table

The User table stores information about all system users, serving as the base table for both customers and staff through a specialization relationship.
Key Constraints:

- Email uniqueness prevents duplicate accounts
- Password field stores bcrypt-hashed passwords (handled at application layer)
- `created_at` and `updated_at` provide audit trail
- Gender enum restricts values to valid options

```sql
CREATE TABLE "User" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "fname" varchar(255) NOT NULL,
  "minit" varchar(50) DEFAULT NULL,
  "lname" varchar(255) NOT NULL,
  "birthday" date DEFAULT NULL,
  "gender" enum('Male','Female','Other') NOT NULL DEFAULT 'Other',
  "email" varchar(255) NOT NULL,
  "password" varchar(255) NOT NULL,
  "district" varchar(255) DEFAULT NULL,
  "city" varchar(255) DEFAULT NULL,
  "created_at" datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" datetime NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY ("id"),
  UNIQUE KEY "uk_user_email" ("email")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 1: *User table structure from schema.sql*

2.1.2 Membership Table

Defines the loyalty program tiers with point rates for earning rewards.

```sql
CREATE TABLE "membership" (
  "tier_name" varchar(255) NOT NULL,
  "min_spent" decimal(12,2) NOT NULL DEFAULT '0.00',
  "box_office_point_rate" decimal(5,2) NOT NULL DEFAULT '0.00',
  "concession_point_rate" decimal(5,2) NOT NULL DEFAULT '0.00',
  PRIMARY KEY ("tier_name"),
  CONSTRAINT "chk_membership_box_rate" CHECK (
    ("box_office_point_rate" >= 0) AND
    ("box_office_point_rate" <= 100)
  ),
  CONSTRAINT "chk_membership_con_rate" CHECK (
    ("concession_point_rate" >= 0) AND
    ("concession_point_rate" <= 100)
  )
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 2: *Membership table structure*

Business Rules:

- Point rates are percentages (0-100) applied to spending amounts
- Separate rates for box office (tickets) and concessions (F&B)
- Membership tiers: Member, U22, VIP, VVIP

2.1.3 Customer Table

Extends the User table with customer-specific attributes including membership and points.

```sql
CREATE TABLE "customer" (
  "user_id" bigint NOT NULL,
  "accumulated_points" int NOT NULL DEFAULT '0',
  "total_spent" decimal(12,2) NOT NULL DEFAULT '0.00',
  "membership_name" varchar(50) DEFAULT NULL,
  "membership_valid_until" date DEFAULT NULL,
  PRIMARY KEY ("user_id"),
  KEY "fk_customer_membership" ("membership_name"),
  CONSTRAINT "fk_customer_membership" FOREIGN KEY ("membership_name")
    REFERENCES "membership" ("tier_name")
    ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT "fk_customer_user" FOREIGN KEY ("user_id")
    REFERENCES "User" ("id"),
  CONSTRAINT "customer_chk_1" CHECK (("accumulated_points" >= 0)),
  CONSTRAINT "customer_chk_2" CHECK (("total_spent" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 3: *Customer table structure*

Key Features:

- Automatically created via trigger when User is inserted
- Points and spending track customer loyalty
- Membership valid until date implements annual renewal cycle
- Check constraints prevent negative values

2.1.4 Staff Table

Extends the User table for staff members with role and shift information.

```sql
CREATE TABLE "staff" (
  "user_id" bigint NOT NULL,
  "shift" varchar(255) NOT NULL,
  "role" varchar(255) NOT NULL,
  "theater_id" bigint DEFAULT NULL,
  PRIMARY KEY ("user_id"),
  KEY "fk_staff_theater" ("theater_id"),
  CONSTRAINT "fk_staff_theater" FOREIGN KEY ("theater_id")
    REFERENCES "theater" ("id"),
  CONSTRAINT "fk_staff_user" FOREIGN KEY ("user_id")
    REFERENCES "User" ("id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 4: *Staff table structure*

2.2 Theater Infrastructure Tables

2.2.1 Theater Table

Represents cinema branches across different locations.

```sql
CREATE TABLE "theater" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "street" varchar(255) NOT NULL,
  "district" varchar(255) NOT NULL,
  "city" varchar(255) NOT NULL,
  "image" varchar(255) DEFAULT NULL,
  "description" text,
  PRIMARY KEY ("id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 5: *Theater table structure*

2.2.2 Auditorium Table

Represents screening rooms within theaters with different formats.

```sql
CREATE TABLE "auditorium" (
  "number" int NOT NULL,
  "theater_id" bigint NOT NULL,
  "type" enum('2D','ScreenX','IMAX','4DX') NOT NULL DEFAULT '2D',
  "capacity" int NOT NULL DEFAULT '0',
  "image" varchar(255) DEFAULT NULL,
  "description" text,
  PRIMARY KEY ("number","theater_id"),
  KEY "fk_auditorium_theater" ("theater_id"),
  CONSTRAINT "fk_auditorium_theater" FOREIGN KEY ("theater_id")
    REFERENCES "theater" ("id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 6: *Auditorium table structure*

Design Notes:

- Composite primary key (`number`, `theater_id`) allows same auditorium numbers across different theaters
- Capacity is automatically maintained by triggers when seats are added/removed
- Type enum supports various cinema formats

2.2.3 Seat Table

Individual seats within auditoriums with position and pricing information.
Seat Identification:

- `row_char` + `column_number` creates human-readable seat names (e.g., "A1", "B5")
- Composite PK ensures seat uniqueness within auditorium
- Triggers update auditorium capacity when seats are modified

```sql
CREATE TABLE "seat" (
  "id" bigint NOT NULL,
  "au_number" int NOT NULL,
  "au_theater_id" bigint NOT NULL,
  "row_char" varchar(10) NOT NULL,
  "column_number" int NOT NULL,
  "type" varchar(255) NOT NULL DEFAULT 'Standard',
  "price" decimal(10,2) NOT NULL,
  PRIMARY KEY ("id","au_number","au_theater_id"),
  KEY "fk_seat_auditorium" ("au_number","au_theater_id"),
  CONSTRAINT "fk_seat_auditorium" FOREIGN KEY
    ("au_number", "au_theater_id")
    REFERENCES "auditorium" ("number", "theater_id"),
  CONSTRAINT "seat_chk_1" CHECK (("price" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 7: *Seat table structure*

2.3 Movie and Showtime Tables

2.3.1 Movie Table

Core movie information with release scheduling.

```sql
CREATE TABLE "movie" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "duration" int NOT NULL,
  "language" varchar(255) DEFAULT NULL,
  "release_date" date DEFAULT NULL,
  "end_date" date DEFAULT NULL,
  "age_rating" varchar(3) NOT NULL,
  "poster_file" varchar(255) DEFAULT NULL,
  "url_slug" varchar(255) DEFAULT NULL,
  "description" text,
  "trailer_url" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id"),
  UNIQUE KEY "uk_movie_url_slug" ("url_slug"),
  CONSTRAINT "movie_chk_1" CHECK (("duration" > 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 8: *Movie table structure*

2.3.2 Movie Multivalued Attributes

The following tables implement multivalued attributes for movies:

```sql
CREATE TABLE "director" (
  "movie_id" bigint NOT NULL,
  "director" varchar(255) NOT NULL,
  PRIMARY KEY ("movie_id","director"),
  CONSTRAINT "fk_director_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id")
);

CREATE TABLE "actor" (
  "movie_id" bigint NOT NULL,
  "actor" varchar(255) NOT NULL,
  "name" varchar(255) DEFAULT NULL,
  "age" int DEFAULT NULL,
  PRIMARY KEY ("movie_id","actor"),
  CONSTRAINT "fk_actor_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id"),
  CONSTRAINT "actor_chk_1" CHECK ((("age" IS NULL) OR ("age" >= 0)))
);

CREATE TABLE "genre" (
  "movie_id" bigint NOT NULL,
  "genre" varchar(255) NOT NULL,
  PRIMARY KEY ("movie_id","genre"),
  CONSTRAINT "fk_genre_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id")
);

CREATE TABLE "subtitle" (
  "movie_id" bigint NOT NULL,
  "subtitle" varchar(255) NOT NULL,
  PRIMARY KEY ("movie_id","subtitle"),
  CONSTRAINT "fk_subtitle_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id")
);

CREATE TABLE "dubbing" (
  "movie_id" bigint NOT NULL,
  "dubbing" varchar(255) NOT NULL,
  PRIMARY KEY ("movie_id","dubbing"),
  CONSTRAINT "fk_dubbing_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id")
);
```

Listing 9: *Director table (multivalued attribute)*

2.3.3 Showtime Table

Links movies to specific auditoriums at scheduled times.
Important Business Logic:

- Triggers prevent overlapping showtimes in same auditorium
- 15-minute gap enforced between consecutive showtimes
- Booking cutoff time validated by trigger (15 minutes before start for online)

```sql
CREATE TABLE "showtime" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "date" date NOT NULL,
  "start_time" time NOT NULL,
  "end_time" time NOT NULL,
  "movie_id" bigint NOT NULL,
  "au_number" int NOT NULL,
  "au_theater_id" bigint NOT NULL,
  PRIMARY KEY ("id"),
  KEY "fk_showtime_movie" ("movie_id"),
  KEY "fk_showtime_auditorium" ("au_number","au_theater_id"),
  CONSTRAINT "fk_showtime_auditorium" FOREIGN KEY
    ("au_number", "au_theater_id")
    REFERENCES "auditorium" ("number", "theater_id"),
  CONSTRAINT "fk_showtime_movie" FOREIGN KEY ("movie_id")
    REFERENCES "movie" ("id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 10: *Showtime table structure*

2.4 Booking and Transaction Tables

2.4.1 Booking Table

Central transaction hub tracking customer bookings.

```sql
CREATE TABLE "booking" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "created_time_at" datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "status" varchar(255) NOT NULL,
  "booking_method" varchar(255) NOT NULL,
  "is_gift" tinyint(1) NOT NULL DEFAULT '0',
  "live_direction" varchar(255) DEFAULT NULL,
  "points_earned" int NOT NULL DEFAULT '0',
  "points_used" int NOT NULL DEFAULT '0',
  "customer_id" bigint DEFAULT NULL,
  "staff_id" bigint DEFAULT NULL,
  PRIMARY KEY ("id"),
  KEY "fk_booking_staff" ("staff_id"),
  KEY "idx_booking_status_created" ("status","created_time_at"),
  KEY "idx_booking_customer" ("customer_id"),
  CONSTRAINT "fk_booking_customer" FOREIGN KEY ("customer_id")
    REFERENCES "customer" ("user_id"),
  CONSTRAINT "fk_booking_staff" FOREIGN KEY ("staff_id")
    REFERENCES "staff" ("user_id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 11: *Booking table structure*

Status Values: `Pending`, `Paid`, `Cancelled`, `Refunded`
Booking Method: `Online`, `Counter`

2.4.2 Showtime_seat Table

Represents individual ticket bookings linking seats to showtimes.

```sql
CREATE TABLE "showtime_seat" (
  "ticketid" bigint NOT NULL AUTO_INCREMENT,
  "st_id" bigint NOT NULL,
  "seat_id" bigint NOT NULL,
  "seat_au_number" int NOT NULL,
  "seat_au_theater_id" bigint NOT NULL,
  "status" varchar(255) NOT NULL DEFAULT 'Valid',
  "price" decimal(10,2) NOT NULL,
  "booking_id" bigint DEFAULT NULL,
  PRIMARY KEY ("ticketid"),
  KEY "fk_showtimeseat_seat"
    ("seat_id","seat_au_number","seat_au_theater_id"),
  KEY "idx_ss_stid_seatid_status" ("st_id","seat_id","status"),
  KEY "idx_ss_booking_status" ("booking_id","status"),
  CONSTRAINT "fk_showtimeseat_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id"),
  CONSTRAINT "fk_showtimeseat_seat" FOREIGN KEY
    ("seat_id", "seat_au_number", "seat_au_theater_id")
    REFERENCES "seat" ("id", "au_number", "au_theater_id"),
  CONSTRAINT "fk_showtimeseat_showtime" FOREIGN KEY ("st_id")
    REFERENCES "showtime" ("id"),
  CONSTRAINT "showtime_seat_chk_1" CHECK (("price" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 12: *Showtime_seat table structure*

Status Lifecycle: `Available` → `Held` → `Booked` → `Refunded`

2.4.3 Payment Table

Records payment transactions for bookings.

```sql
CREATE TABLE "payment" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "payment_method" varchar(255) NOT NULL,
  "status" varchar(255) NOT NULL,
  "created_time_at" datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "transaction_id" varchar(255) NOT NULL,
  "expired_time_at" datetime DEFAULT NULL,
  "duration" int DEFAULT NULL,
  "booking_id" bigint NOT NULL,
  PRIMARY KEY ("id"),
  UNIQUE KEY "uk_payment_transaction" ("transaction_id"),
  UNIQUE KEY "uk_payment_booking" ("booking_id"),
  KEY "idx_payment_booking" ("booking_id"),
  CONSTRAINT "fk_payment_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id")
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 13: *Payment table structure*

2.4.4 Coupon Table

Manages promotional coupons and gift cards.

```sql
CREATE TABLE "coupon" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "gift" tinyint(1) NOT NULL DEFAULT '0',
  "balance" decimal(10,2) NOT NULL DEFAULT '0.00',
  "coupon_type" enum('Amount','Percent','GiftCard') NOT NULL,
  "date_expired" date DEFAULT NULL,
  "booking_id" bigint DEFAULT NULL,
  "customer_id" bigint DEFAULT NULL,
  "discount_amount" decimal(10,2) DEFAULT '0.00',
  PRIMARY KEY ("id"),
  KEY "fk_coupon_booking" ("booking_id"),
  KEY "fk_coupon_customer" ("customer_id"),
  CONSTRAINT "fk_coupon_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id"),
  CONSTRAINT "fk_coupon_customer" FOREIGN KEY ("customer_id")
    REFERENCES "customer" ("user_id"),
  CONSTRAINT "coupon_chk_1" CHECK (("balance" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 14: *Coupon table structure*

2.4.5 Refund Table

Tracks refund requests and processing with optional coupon compensation.

```sql
CREATE TABLE "refund" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "booking_id" bigint NOT NULL,
  "amount" decimal(10,2) NOT NULL,
  "reason" varchar(255) DEFAULT NULL,
  "status" varchar(255) NOT NULL DEFAULT 'Requested',
  "created_time_at" datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "processed_time_at" datetime DEFAULT NULL,
  "coupon_id" bigint DEFAULT NULL,
  PRIMARY KEY ("id","booking_id"),
  KEY "fk_refund_coupon" ("coupon_id"),
  KEY "idx_refund_booking_status" ("booking_id","status"),
  CONSTRAINT "fk_refund_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id"),
  CONSTRAINT "fk_refund_coupon" FOREIGN KEY ("coupon_id")
    REFERENCES "coupon" ("id"),
  CONSTRAINT "refund_chk_1" CHECK (("amount" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 15: *Refund table structure*

2.5 Food and Beverage Tables

2.5.1 FWB_Menu Table

Catalog of available food and beverage items.

```
CREATE TABLE "fwb_menu" (
  "id" bigint NOT NULL AUTO_INCREMENT,
  "name" varchar(255) NOT NULL,
  "description" text,
  "image" varchar(255) DEFAULT NULL,
  "price" decimal(10,2) NOT NULL,
  "category" varchar(255) NOT NULL,
  "capacity" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id"),
  CONSTRAINT "fwb_menu_chk_1" CHECK (("price" >= 0))
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Listing 16: *FWB_Menu table structure*

2.5.2 FWB and Contains_item Tables

These tables manage the N:M relationship between bookings and menu items.

```
CREATE TABLE "fwb" (
  "id" bigint NOT NULL,
  "booking_id" bigint NOT NULL,
  "quantity" int NOT NULL DEFAULT '1',
  "price" decimal(10,2) NOT NULL,
  PRIMARY KEY ("id","booking_id"),
  KEY "idx_fwb_booking" ("booking_id"),
  CONSTRAINT "fk_fwb_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id"),
  CONSTRAINT "fwb_chk_1" CHECK (("quantity" > 0)),
  CONSTRAINT "fwb_chk_2" CHECK (("price" >= 0))
);

CREATE TABLE "contains_item" (
  "fwb_menu_id" bigint NOT NULL,
  "fwb_id" bigint NOT NULL,
  "fwb_booking_id" bigint NOT NULL,
  "quantity" int NOT NULL DEFAULT '1',
  PRIMARY KEY ("fwb_menu_id","fwb_id","fwb_booking_id"),
  KEY "fk_contains_item_fwb" ("fwb_id","fwb_booking_id"),
  KEY "idx_contains_fwb_booking_menu" ("fwb_booking_id","fwb_menu_id"),
  CONSTRAINT "fk_contains_item_fwb" FOREIGN KEY
    ("fwb_id", "fwb_booking_id")
    REFERENCES "fwb" ("id", "booking_id"),
  CONSTRAINT "fk_contains_item_fwbmenu" FOREIGN KEY ("fwb_menu_id")
    REFERENCES "fwb_menu" ("id"),
  CONSTRAINT "contains_item_chk_1" CHECK (("quantity" > 0))
);
```

Listing 17: *FWB and Contains_item structures*

2.6 Gift Relationship Tables

These tables track coupon and booking gift transactions between customers.

```
CREATE TABLE "give" (
  "coupon_id" bigint NOT NULL,
  "sender_id" bigint NOT NULL,
  "receiver_id" bigint NOT NULL,
  PRIMARY KEY ("coupon_id"),
  KEY "fk_give_sender" ("sender_id"),
  KEY "fk_give_receiver" ("receiver_id"),
  CONSTRAINT "fk_give_coupon" FOREIGN KEY ("coupon_id")
    REFERENCES "coupon" ("id"),
  CONSTRAINT "fk_give_receiver" FOREIGN KEY ("receiver_id")
    REFERENCES "customer" ("user_id"),
  CONSTRAINT "fk_give_sender" FOREIGN KEY ("sender_id")
    REFERENCES "customer" ("user_id")
);

CREATE TABLE "send_gift" (
  "booking_id" bigint NOT NULL,
  "sender_id" bigint NOT NULL,
  "receiver_id" bigint NOT NULL,
  PRIMARY KEY ("booking_id"),
  KEY "fk_sendgift_sender" ("sender_id"),
  KEY "fk_sendgift_receiver" ("receiver_id"),
  CONSTRAINT "fk_sendgift_booking" FOREIGN KEY ("booking_id")
    REFERENCES "booking" ("id"),
  CONSTRAINT "fk_sendgift_receiver" FOREIGN KEY ("receiver_id")
    REFERENCES "customer" ("user_id"),
  CONSTRAINT "fk_sendgift_sender" FOREIGN KEY ("sender_id")
    REFERENCES "customer" ("user_id")
);
```

Listing 18: *Gift relationship tables*

2.7 Indexing Strategy

The database implements a indexing strategy for query optimization:

- **Primary Keys**: All tables have primary key indexes (B-Tree, Unique)
- **Foreign Keys**: Automatic indexes on all foreign key columns
- **Unique Constraints**: `uk_user_email`, `uk_movie_url_slug`, `uk_payment_transaction`
- **Composite Indexes**:
  - `idx_booking_status_created` (`status`, `created_time_at`) for filtering
  - `idx_ss_stid_seatid_status` for seat availability queries
  - `idx_ss_booking_status` for ticket lookup
- **Performance Indexes**:
  - `idx_booking_customer` for customer booking history
  - `idx_payment_booking` for payment lookups
  - `idx_refund_booking_status` for refund queries

3 Triggers

3.1 User Management Triggers

3.1.1 Trigger: Auto-Create Customer Record

Purpose: Automatically creates a customer record when a new user is inserted, implementing the User-Customer specialization relationship from the ERD.
Timing: AFTER INSERT on `User` table
Implementation:

```
DELIMITER $$
CREATE TRIGGER "trg_after_user_insert"
AFTER INSERT ON "User"
FOR EACH ROW
BEGIN
    DECLARE v_membership VARCHAR(50);
    DECLARE v_age INT;
    DECLARE v_valid_until DATE;

    IF NOT EXISTS (SELECT 1 FROM staff WHERE user_id = NEW.id) THEN
        IF NEW.birthday IS NOT NULL THEN
            SET v_age = TIMESTAMPDIFF(YEAR, NEW.birthday, CURDATE());
        ELSE
            SET v_age = NULL;
        END IF;

        IF v_age IS NOT NULL AND v_age < 22 THEN
            SET v_membership = 'U22';
        ELSE
            SET v_membership = 'Member';
        END IF;

        IF MONTH(CURDATE()) < 6 OR
           (MONTH(CURDATE()) = 6 AND DAY(CURDATE()) = 1) THEN
            SET v_valid_until = DATE(CONCAT(YEAR(CURDATE()), '-06-01'));
        ELSE
            SET v_valid_until = DATE(CONCAT(YEAR(CURDATE()) + 1, '-06-01'));
        END IF;

        INSERT INTO customer (
            user_id, accumulated_points, total_spent,
            membership_name, membership_valid_until
        )
        VALUES (
            NEW.id, 0, 0, v_membership, v_valid_until
        );
    END IF;
END$$
DELIMITER ;
```

Listing 19: *Trigger: trg_after_user_insert*

Business Logic:

- Only creates customer record if user is not a staff member

- Determines initial membership based on age (U22 for users under 22, otherwise Member)
- Sets membership valid until next June 1st (annual renewal cycle)
- Initializes `accumulated_points` and `total_spent` to 0

Integration: This trigger works seamlessly with the `sp_register_user` stored procedure, eliminating the need for application code to manually create customer records.

3.1.2 Trigger: Validate Customer Updates

Purpose: Validates customer data before updates to prevent negative values and maintain data integrity.
Timing: BEFORE UPDATE on `customer` table
Implementation:

```
DELIMITER $$
CREATE TRIGGER "trg_before_customer_update"
BEFORE UPDATE ON "customer"
FOR EACH ROW
BEGIN
    IF NEW.accumulated_points < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Accumulated points cannot be negative';
    END IF;

    IF NEW.total_spent < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Total spent cannot be negative';
    END IF;
END$$
DELIMITER ;
```

Listing 20: *Trigger: trg_before_customer_update*

3.2 Seat Capacity Management Triggers

These triggers automatically maintain the auditorium capacity field as seats are added, removed, or moved between auditoriums.

3.2.1 Trigger: Update Capacity on Seat Insert

Purpose: Increments auditorium capacity when a new seat is added.
Timing: AFTER INSERT on `seat` table
Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_after_seat_insert"
AFTER INSERT ON "seat"
FOR EACH ROW
BEGIN
    UPDATE auditorium
    SET capacity = (
        SELECT COUNT(*)
        FROM seat
        WHERE au_number = NEW.au_number
        AND au_theater_id = NEW.au_theater_id
    )
    WHERE number = NEW.au_number
    AND theater_id = NEW.au_theater_id;
END$$
DELIMITER ;
```

Listing 21: *Trigger: trg_after_seat_insert*

3.2.2 Trigger: Update Capacity on Seat Delete

Purpose: Decrements auditorium capacity when a seat is removed.
Timing: AFTER DELETE on `seat` table
Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_after_seat_delete"
AFTER DELETE ON "seat"
FOR EACH ROW
BEGIN
    UPDATE auditorium
    SET capacity = (
        SELECT COUNT(*)
        FROM seat
        WHERE au_number = OLD.au_number
        AND au_theater_id = OLD.au_theater_id
    )
    WHERE number = OLD.au_number
    AND theater_id = OLD.au_theater_id;
END$$
DELIMITER ;
```

Listing 22: *Trigger: trg_after_seat_delete*

3.2.3 Trigger: Update Capacity on Seat Move

Purpose: Updates capacity for both source and destination auditoriums when a seat is moved.
Timing: AFTER UPDATE on `seat` table
Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_after_seat_update"
AFTER UPDATE ON "seat"
FOR EACH ROW
BEGIN
    IF OLD.au_number != NEW.au_number OR
       OLD.au_theater_id != NEW.au_theater_id THEN
        UPDATE auditorium
        SET capacity = (
            SELECT COUNT(*)
            FROM seat
            WHERE au_number = OLD.au_number
            AND au_theater_id = OLD.au_theater_id
        )
        WHERE number = OLD.au_number
        AND theater_id = OLD.au_theater_id;
    END IF;

    UPDATE auditorium
    SET capacity = (
        SELECT COUNT(*)
        FROM seat
        WHERE au_number = NEW.au_number
        AND au_theater_id = NEW.au_theater_id
    )
    WHERE number = NEW.au_number
    AND theater_id = NEW.au_theater_id;
END$$
DELIMITER ;
```

Listing 23: *Trigger: trg_after_seat_update*

3.3 Showtime Scheduling Triggers

3.3.1 Trigger: Prevent Overlapping Showtimes on Insert

Purpose: Enforces the business rule that showtimes in the same auditorium must not overlap and must have at least 15 minutes gap for cleaning.
Timing: BEFORE INSERT on `showtime` table
Business Rules:

- Two showtimes cannot overlap in the same auditorium on the same date
- Minimum 15-minute gap required between consecutive showtimes
- Validation uses ADDTIME/SUBTIME for buffer calculation

Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_before_showtime_insert"
BEFORE INSERT ON "showtime"
FOR EACH ROW
BEGIN
    DECLARE conflict_count INT;

    SELECT COUNT(*) INTO conflict_count
    FROM showtime
    WHERE au_number = NEW.au_number
      AND au_theater_id = NEW.au_theater_id
      AND date = NEW.date
      AND (
          (NEW.start_time < ADDTIME(end_time, '00:15:00')
           AND NEW.end_time > SUBTIME(start_time, '00:15:00'))
      );

    IF conflict_count > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT =
            'Showtime conflicts with existing schedule. ' ||
            'Showtimes must not overlap and must have at least ' ||
            '15 minutes gap between them.';
    END IF;
END$$
DELIMITER ;
```

Listing 24: *Trigger: trg_before_showtime_insert*

3.3.2 Trigger: Prevent Overlapping Showtimes on Update

Purpose: Enforces the same scheduling rules when modifying existing showtimes.
Timing: BEFORE UPDATE on `showtime` table
Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_before_showtime_update"
BEFORE UPDATE ON "showtime"
FOR EACH ROW
BEGIN
    DECLARE conflict_count INT;

    IF NEW.date != OLD.date
       OR NEW.start_time != OLD.start_time
       OR NEW.end_time != OLD.end_time
       OR NEW.au_number != OLD.au_number
       OR NEW.au_theater_id != OLD.au_theater_id THEN

        SELECT COUNT(*) INTO conflict_count
        FROM showtime
        WHERE id != NEW.id
          AND au_number = NEW.au_number
          AND au_theater_id = NEW.au_theater_id
          AND date = NEW.date
          AND (
              (NEW.start_time < ADDTIME(end_time, '00:15:00')
               AND NEW.end_time > SUBTIME(start_time, '00:15:00'))
          );

        IF conflict_count > 0 THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT =
                'Showtime conflicts with existing schedule.';
        END IF;
    END IF;
END$$
DELIMITER ;
```

Listing 25: *Trigger: trg_before_showtime_update*

3.4 Booking and Payment Triggers

3.4.1 Trigger: Validate Booking Time and Seat Availability

Purpose: Prevents booking tickets for showtimes that have already started or are too close to start time for online bookings.
Timing: BEFORE INSERT on `showtime_seat` table
Business Rules:

- Cannot book tickets for showtimes that have already started or passed
- Online bookings must be completed at least 15 minutes before showtime
- Staff counter bookings can be made closer to showtime (bypass 15-minute rule)
- Uses Vietnam timezone (GMT+7) for time comparisons

Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_before_showtime_seat_insert"
BEFORE INSERT ON "showtime_seat"
FOR EACH ROW
BEGIN
    DECLARE v_showtime_datetime DATETIME;
    DECLARE v_booking_method VARCHAR(255);
    DECLARE v_minutes_until_showtime INT;
    DECLARE v_current_datetime_vn DATETIME;

    SET v_current_datetime_vn = CONVERT_TZ(NOW(), '+00:00', '+07:00');

    SELECT TIMESTAMP(date, start_time) INTO v_showtime_datetime
    FROM showtime
    WHERE id = NEW.st_id;

    IF v_showtime_datetime <= v_current_datetime_vn THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT =
            'Cannot book tickets for showtimes that have ' ||
            'already started or passed.';
    END IF;

    IF NEW.status = 'Held' AND NEW.booking_id IS NOT NULL THEN
        SELECT booking_method INTO v_booking_method
        FROM booking
        WHERE id = NEW.booking_id;

        IF v_booking_method = 'Online' THEN
            SET v_minutes_until_showtime =
                TIMESTAMPDIFF(MINUTE, v_current_datetime_vn,
                              v_showtime_datetime);

            IF v_minutes_until_showtime < 15 THEN
                SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT =
                    'Online booking must be completed at least ' ||
                    '15 minutes before showtime start time.';
            END IF;
        END IF;
    END IF;
END$$
DELIMITER ;
```

Listing 26: *Trigger: trg_before_showtime_seat_insert*

3.4.2 Trigger: Set Booking Gift Flag

Purpose: Automatically sets the `is_gift` flag on a booking when a gift transaction is recorded.
Timing: AFTER INSERT on `send_gift` table
Implementation:

```sql
DELIMITER $$
CREATE TRIGGER "trg_sendgift_set_isgift"
AFTER INSERT ON "send_gift"
FOR EACH ROW
BEGIN
    UPDATE booking
    SET is_gift = 1
    WHERE id = NEW.booking_id;
END$$
DELIMITER ;
```

Listing 27: *Trigger: trg_sendgift_set_isgift*

4 Stored Procedures

4.1 Authentication and User Management Procedures

4.1.1 Procedure: sp_register_user

Implementation:

```sql
CREATE PROCEDURE sp_register_user(...)
BEGIN
    DECLARE v_email_valid BOOLEAN;
    DECLARE v_age_valid BOOLEAN;

    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        GET DIAGNOSTICS CONDITION 1 v_error_msg = MESSAGE_TEXT;
        ROLLBACK;
        SET p_success = FALSE;
        SET p_message = CONCAT('Database error: ', v_error_msg);
    END;

    START TRANSACTION;

    SET v_email_valid = fn_validate_email(p_email);
    IF NOT v_email_valid THEN
        SET p_success = FALSE;
        SET p_message = 'Invalid email format';
        ROLLBACK;
    ELSE
        INSERT INTO `User` (...) VALUES (...);
        SET p_user_id = LAST_INSERT_ID();
        COMMIT;
    END IF;
END;
```

Listing 28: *sp_register_user procedure*

Purpose: Registers a new user with validation.
Parameters:

- IN: `p_fname`, `p_minit`, `p_lname`, `p_email`, `p_password`, `p_birthday`, `p_gender`, `p_district`, `p_city`
- OUT: `p_user_id` (generated ID), `p_success` (boolean), `p_message` (status message)

Validation Logic:

- Email format validation using `fn_validate_email()`
- Email uniqueness check using `fn_check_email_exists()`
- Age validation (must be ≥ 13 years) using `fn_validate_age()`
- Password must be bcrypt-hashed before calling (handled by application)

4.1.2 Procedure: sp_login_user

Purpose: Retrieves user credentials for authentication.
Parameters:

- IN: `p_email`
- OUT: `p_user_id`, `p_password`, `p_fname`, `p_lname`, `p_success`, `p_message`

Note: Password verification using bcrypt is performed in the application layer (NestJS) after retrieving the hashed password.

4.1.3 Procedure: sp_update_user_profile

Purpose: Updates user profile information.
Business Rules:

- Only non-null parameters are updated (partial updates supported)
- `birthday` cannot be changed after registration (security requirement)
- `email` cannot be changed (would require separate email change with verification)

4.2 Showtime Management Procedures

4.2.1 Procedure: sp_delete_showtime

Purpose: Deletes a showtime from the system with proper authorization and validation checks.
Parameters:

- IN: `p_showtime_id` (ID of showtime to delete)
- IN: `p_staff_id` (ID of staff requesting deletion)

Business Rules:

1. If any customer has booked tickets for this showtime
2. If staff role is not Admin or Manager (e.g., regular Staff)
3. If showtime ID does not exist

Implementation:

```sql
CREATE PROCEDURE sp_delete_showtime(
    IN p_showtime_id BIGINT,
    IN p_staff_id BIGINT
)
BEGIN
    DECLARE v_showtime_exists INT;
    DECLARE v_ticket_count INT;
    DECLARE v_staff_role VARCHAR(255);

    SELECT `role` INTO v_staff_role
    FROM staff
    WHERE user_id = p_staff_id;

    IF v_staff_role IS NULL THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Staff not found or unauthorized';
    END IF;

    IF v_staff_role NOT IN ('Admin', 'Manager') THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT =
                'Only Admin or Manager staff can delete showtime';
    END IF;

    SELECT COUNT(*) INTO v_showtime_exists
    FROM showtime
    WHERE id = p_showtime_id;

    IF v_showtime_exists = 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Showtime not found';
    END IF;

    SELECT COUNT(*) INTO v_ticket_count
    FROM showtime_seat
    WHERE st_id = p_showtime_id
      AND booking_id IS NOT NULL;

    IF v_ticket_count > 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT =
                'Cannot delete showtime: ' ||
                'There are tickets already booked';
    END IF;

    DELETE FROM showtime_seat WHERE st_id = p_showtime_id;
    DELETE FROM showtime WHERE id = p_showtime_id;

END;
```

Listing 29: *sp_delete_showtime procedure*

Deletion Strategy:

- **Hard Delete**: Showtime is permanently removed from database

- **Cascade Deletion**: All associated `showtime_seat` records are deleted first
- **No Soft Delete**: System does not use status-based soft deletion for showtimes
- **Rationale**: Showtimes without bookings have no historical value and can be safely removed to maintain database cleanliness

Alternative Approach for Booked Showtimes:

- For showtimes with existing bookings, use `sp_cancel_booking` to process refunds
- After all bookings are cancelled and refunded, the showtime can be deleted
- This ensures customer rights are protected and refund policies are enforced

4.3 Booking Management Procedures

4.3.1 Procedure: sp_start_booking

Purpose: Creates a new booking with seat reservations and optional F&B items in a single transaction.
Parameters:

- IN: `p_customer_id`, `p_showtime_id`, `p_seat_ids_json` (array), `p_fwb_items_json` (array or NULL)
- OUT: `p_booking_id`

Transaction Flow:

1. Validate showtime exists and get auditorium info
2. Create booking record with `status='Pending'`
3. Loop through seat IDs from JSON array:
   - Verify seat exists in showtime's auditorium
   - Check seat is not already `Held` or `Booked`
   - Insert `showtime_seat` with `status='Held'`
4. If F&B items provided:
   - Create parent FWB record
   - Loop through items, validate menu IDs, insert `contains_item` records
   - Calculate total F&B price
5. COMMIT if all seats successfully held, ROLLBACK on any error

Implementation:

```sql
CREATE PROCEDURE sp_start_booking(...)
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        RESIGNAL;
    END;

    START TRANSACTION;

    INSERT INTO booking (created_time_at, status, booking_method, ...)
    VALUES (NOW(), 'Pending', 'Online', ...);
    SET p_booking_id = LAST_INSERT_ID();

    WHILE v_idx < v_len DO
        SET v_seat_id = CAST(JSON_UNQUOTE(
            JSON_EXTRACT(p_seat_ids_json, CONCAT('$[', v_idx, ']'))
        ) AS UNSIGNED);

        IF EXISTS (
            SELECT 1 FROM showtime_seat
            WHERE st_id = p_showtime_id
            AND seat_id = v_seat_id
            AND status IN ('Held','Booked')
        ) THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Seat already taken';
        END IF;

        INSERT INTO showtime_seat (...)
        VALUES (..., 'Held', ...);

        SET v_idx = v_idx + 1;
    END WHILE;

    COMMIT;
END;
```

Listing 30: *sp_start_booking - seat locking logic*

4.3.2 Procedure: sp_confirm_payment

Purpose: Processes payment confirmation, updates booking status, converts held seats to booked, and awards loyalty points.
Parameters:

- IN: `p_booking_id`, `p_payment_method`, `p_transaction_id`, `p_total_amount`, `p_duration`
- OUT: `p_payment_id`

Complex Business Logic:

1. Validate booking is in `'Pending'` status
2. Deduct used points from customer `accumulated_points`
3. Insert payment record with `status='Success'`

4. Update booking `status` to `'Paid'`
5. Update `showtime_seat` status from `'Held'` to `'Booked'`
6. Calculate final amount paid (after discounts)
7. Calculate points earned based on membership tier rates
8. Update customer `total_spent` and `accumulated_points`
9. Store `points_earned` in booking record

Points Calculation Logic:

```
CALL sp_calculate_final_amount(p_booking_id,
    v_base_seat_price, v_fwb_price, v_subtotal,
    v_coupon_discount, v_box_office_discount,
    v_concession_discount, v_points_discount,
    v_membership_tier, v_coupon_type, v_final_amount);

SELECT m.box_office_point_rate, m.concession_point_rate
INTO v_ticket_rate, v_fnb_rate
FROM customer c
JOIN membership m ON m.tier_name = c.membership_name
WHERE c.user_id = v_customer_id;

SET v_ticket_amount = v_final_amount * (v_base_seat_price / v_subtotal);
SET v_fnb_amount = v_final_amount * (v_fwb_price / v_subtotal);

SET v_ticket_points = fn_calculate_points(v_ticket_amount, v_ticket_rate);
SET v_fnb_points = fn_calculate_points(v_fnb_amount, v_fnb_rate);
SET v_total_points = v_ticket_points + v_fnb_points;

UPDATE customer
SET total_spent = total_spent + v_final_amount,
    accumulated_points = accumulated_points + v_total_points
WHERE user_id = v_customer_id;
```

Listing 31: *sp_confirm_payment - points calculation*

4.3.3 Procedure: sp_cancel_expired_bookings

Purpose: Automatically cancels bookings that have exceeded the payment timeout period (batch operation for cleanup job).
Parameters:

- IN: `p_timeout_minutes` (e.g., 15 minutes)
- OUT: `p_cancelled_count`, `p_failed_count`

Implementation Strategy:

- Uses CURSOR to iterate through expired pending bookings
- Each booking processed in separate transaction to prevent all-or-nothing rollback
- Creates cancelled payment record for audit trail
- Releases held seats back to `Available` status
- Continues processing even if individual cancellations fail
- Returns success and failure counts

4.4 Payment and Refund Procedures

4.4.1 Procedure: sp_apply_coupon

Purpose: Applies a coupon to a booking with validation.
Validation Rules:

- Coupon must belong to customer
- Coupon must not be already used (`booking_id IS NULL`)
- Coupon must not be expired
- Coupon must have positive `balance`

Key Implementation Detail:

```
UPDATE coupon
SET booking_id = p_booking_id,
    discount_amount = balance,
    balance = 0
WHERE id = p_coupon_id;
```

Listing 32: *sp_apply_coupon - discount tracking*

4.4.2 Procedure: sp_apply_points

Purpose: Applies loyalty points to a booking for payment discount.
Business Rules:

- Minimum 10 points required for redemption
- Points can cover up to 90% of transaction value
- 1 point = 1,000 VND
- Points are only deducted from customer when payment is confirmed (not when applied)

4.4.3 Procedure: sp_create_refund_coupon

Purpose: Creates a refund and issues a compensation coupon valid for 1 year.
Transaction Flow:

1. Verify booking ownership and status
2. Validate refund amount > 0
3. Create refund record with `status='Completed'`
4. Create compensation coupon (`type=GiftCard`, valid 1 year)
5. Link coupon to refund
6. Update booking `status` to `'Cancelled'`

4.5 Query and Reporting Procedures

4.5.1 Procedure: sp_get_movies_with_details

Purpose: Optimized query that returns movies with ALL related data in a single database call (eliminates N+1 query problem).
Features:

- Joins movie, director, actor, genre, subtitle, dubbing tables
- Uses `GROUP_CONCAT` to aggregate multivalued attributes
- Filters by `status` (now, upcoming, ended, all)
- Supports pagination with `LIMIT` and `OFFSET`
- Returns denormalized result set for efficient frontend rendering

Implementation:

```sql
CREATE PROCEDURE sp_get_movies_with_details(
    IN p_status VARCHAR(20),
    IN p_limit INT,
    IN p_offset INT
)
BEGIN
    SELECT
        m.id, m.name, m.duration, m.language, m.release_date,
        m.end_date, m.age_rating, m.poster_file, m.url_slug,
        m.description, m.trailer_url,
        GROUP_CONCAT(DISTINCT d.director ORDER BY d.director
            SEPARATOR '||||') AS directors,
        GROUP_CONCAT(DISTINCT a.actor ORDER BY a.actor
            SEPARATOR '||||') AS actors,
        GROUP_CONCAT(DISTINCT g.genre ORDER BY g.genre
            SEPARATOR '||||') AS genres,
        GROUP_CONCAT(DISTINCT s.subtitle ORDER BY s.subtitle
            SEPARATOR '||||') AS subtitles,
        GROUP_CONCAT(DISTINCT db.dubbing ORDER BY db.dubbing
            SEPARATOR '||||') AS dubbing_options
    FROM movie m
    LEFT JOIN director d ON m.id = d.movie_id
    LEFT JOIN actor a ON m.id = a.movie_id
    LEFT JOIN genre g ON m.id = g.movie_id
    LEFT JOIN subtitle s ON m.id = s.movie_id
    LEFT JOIN dubbing db ON m.id = db.movie_id
    WHERE (
        p_status = 'all'
        OR (p_status = 'now' AND m.release_date <= CURDATE()
            AND (m.end_date IS NULL OR m.end_date >= CURDATE()))
        OR (p_status = 'upcoming' AND m.release_date > CURDATE())
        OR (p_status = 'ended' AND m.end_date < CURDATE())
    )
    GROUP BY m.id, ...
    ORDER BY m.release_date DESC
    LIMIT p_limit OFFSET p_offset;
END;
```

Listing 33: *sp_get_movies_with_details - optimized query*

4.5.2 Procedure: sp_get_customer_bookings

Purpose: Retrieves customer's booking history with complete details including tickets, F&B, and calculated totals.
Return Values:

- Result Set 1: Array of bookings with nested details (JSON for F&B items)
- Result Set 2: Pagination metadata (`totalCount`, `limit`, `offset`, `hasMore`)

Complex Calculation:

- Uses temporary table to store calculated final amounts per booking
- Calls `sp_calculate_final_amount` for each booking
- Converts UTC timestamps to Vietnam timezone (GMT+7)
- Aggregates seat names into comma-separated list
- Returns F&B items as JSON array

4.5.3 Procedure: sp_calculate_final_amount

Purpose: Centralized calculation of booking total with all discounts applied.
Parameters:

- IN: `p_booking_id`
- OUT: `p_base_seat_price`, `p_fwb_price`, `p_subtotal`, `p_membership_tier`, `p_coupon_discount`, `p_box_office_discount`, `p_concession_discount`, `p_points_discount`, `p_membership_tier`, `p_final_amount`

Calculation Logic:

1. Calculate base ticket price (sum of `showtime_seat` prices)
2. Calculate F&B total (sum of fwb `price` $\times$ `quantity`)
3. Get coupon discount (from `coupon.discount_amount` where `booking_id` matches)
4. Calculate points discount (`points_used` $\times$ 1000)
5. Final amount = `subtotal` - `coupon_discount` - `points_discount`
6. Ensure final amount ≥ 0

Note: Current implementation does not apply membership percentage discounts (set to 0), only tracks tier for display purposes.

4.5.4 Procedure: sp_generate_sales_report

Purpose: Generates sales report for a theater within a date range.
Features:

- Uses CURSOR to iterate through showtimes
- Calculates tickets sold and revenue per showtime
- Includes F&B revenue linked to showtime bookings
- Creates temporary table for results
- Returns aggregated data ordered by date

Parameters Validation:

- Theater ID required and must exist
- Start date and end date required
- Start date must be $\leq$ end date
- Returns meaningful error messages via SIGNAL

4.6 Customer Dashboard Procedures

4.6.1 Procedure: sp_get_customer_dashboard

Purpose: Retrieves summary statistics and recent activities for customer dashboard.
Return Values:

- Result Set 1: Stats (`totalPoints`, `totalGiftCards`, `totalVouchers`, `totalBookings`)
- Result Set 2: Recent activities (last 10 activities combining bookings and coupons)

4.6.2 Procedure: sp_get_customer_points

Purpose: Retrieves customer's point balance and transaction history with pagination.
Point History Categories:

- `EARNED`: Points earned from completed payments
- `USED`: Points applied for payment discounts
- `REFUNDED`: Points deducted when booking is refunded

Implementation Note: Uses UNION ALL to combine point transactions from different sources, then applies pagination.

4.6.3 Procedure: sp_get_membership_card

Purpose: Generates membership card information for display.
Unique Card Number Generation:

```
CONCAT(
    LPAD(c.user_id, 4, '0'), '-',
    LPAD(MOD(c.user_id * 7919, 10000), 4, '0'), '-',
    LPAD(MOD(c.user_id * 4999 + 1000, 10000), 4, '0'), '-',
    LPAD(MOD(c.user_id * 9973 + 5000, 10000), 4, '0')
) AS card_number
```

Listing 34: *Membership card number generation*

4.7 Gift Transaction Procedures

4.7.1 Procedure: sp_gift_coupon

Purpose: Transfers coupon ownership from sender to receiver.
Business Rules:

- Cannot gift to yourself
- Receiver must exist and be a customer
- Sender must own the coupon
- Coupon cannot be already applied to booking

- Coupon cannot be expired
- Coupon can only be gifted once (`gift` flag check)

Transaction Safety:

- Uses row-level locking (`FOR UPDATE`) to prevent race conditions
- Checks `ROW_COUNT()` after `UPDATE` to detect concurrent modifications
- Records transaction in `give` table for audit trail

4.7.2 Procedure: sp_send_gift

Purpose: Transfers booking ownership as a gift.
Validation and Process:

- Validates booking `status` (must be `Confirmed` or `Paid`)
- Checks booking hasn't been previously gifted
- Updates booking `customer_id` to receiver
- Sets `is_gift` flag to 1 (via trigger)
- Records transaction in `send_gift` table

5 Functions

Functions in MySQL provide reusable computational logic that can be called within SQL statements, stored procedures, or triggers. The BKinema system implements 9 functions that handle validation, calculation, data transformation, and revenue analysis tasks. This section documents each function with its purpose, implementation, and integration with other database objects.

5.1 Validation Functions

These functions perform data validation and are used extensively by stored procedures and triggers to enforce business rules.

5.1.1 Function: fn_validate_email

Purpose: Validates email format using regex pattern matching.
Returns: BOOLEAN (1 if valid, 0 if invalid)
Implementation:

```
CREATE FUNCTION fn_validate_email(p_email VARCHAR(255))
RETURNS BOOLEAN
DETERMINISTIC
BEGIN
    RETURN p_email REGEXP
        '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$';
END;
```

Listing 35: *fn_validate_email function*

Usage: Called by `sp_register_user` to validate email format before account creation.
Regex Pattern Explanation:

- `[A-Za-z0-9._%+-]+`: Local part (before @) allows alphanumeric and common special characters

- `@`: Literal @ symbol
- `[A-Za-z0-9.-]+`: Domain name allows alphanumeric, hyphens, dots
- `.[A-Za-z]{2,}`: Top-level domain (.com, .net, etc.) requires at least 2 letters

5.1.2 Function: fn_check_email_exists

Purpose: Checks if an email is already registered in the system.
Returns: BOOLEAN (1 if exists, 0 if not)
Implementation:

```
CREATE FUNCTION fn_check_email_exists(p_email VARCHAR(255))
RETURNS TINYINT(1)
READS SQL DATA
BEGIN
    DECLARE email_count INT;
    SELECT COUNT(*) INTO email_count
    FROM `User`
    WHERE email = p_email;
    RETURN email_count > 0;
END;
```

Listing 36: *fn_check_email_exists function*

Usage: Called by `sp_register_user` to prevent duplicate email registrations.
Declaration Note: `READS SQL DATA` indicates the function queries the database but doesn't modify data.

5.1.3 Function: fn_validate_age

Purpose: Validates that user's age meets minimum requirement (13 years).
Returns: BOOLEAN (1 if valid, 0 if invalid)
Implementation:

```
CREATE FUNCTION fn_validate_age(p_birthday DATE)
RETURNS BOOLEAN
DETERMINISTIC
BEGIN
    RETURN TIMESTAMPDIFF(YEAR, p_birthday, CURDATE()) >= 13;
END;
```

Listing 37: *fn_validate_age function*

Usage: Called by `sp_register_user` and user update procedures to enforce minimum age policy.
TIMESTAMPDIFF Note: Calculates complete years between birthday and current date, properly handling leap years and varying month lengths.

5.1.4 Function: fn_validate_password_hash

Purpose: Validates that password is properly bcrypt-hashed.
Returns: BOOLEAN (1 if valid hash, 0 if not)
Implementation:

```sql
CREATE FUNCTION fn_validate_password_hash(p_password VARCHAR(255))
RETURNS TINYINT(1)
DETERMINISTIC
BEGIN
    RETURN p_password IS NOT NULL
           AND LENGTH(p_password) = 60
           AND p_password LIKE '$2%';
END;
```

Listing 38: *fn_validate_password_hash function*

Validation Criteria:

- Password must not be NULL
- Length must be exactly 60 characters (bcrypt hash format)
- Must start with $2 (bcrypt algorithm identifier: $2a$, $2b$, $2y$)

Security Note: Password hashing is performed in the application layer (NestJS with bcrypt library) before calling stored procedures. This function only validates the format.

5.2 Calculation Functions

5.2.1 Function: fn_calculate_points

Purpose: Calculates loyalty points earned from spending amount based on membership tier rate.
Parameters:

- p_amount: Spending amount in VND
- p_rate: Point earning rate as percentage (0-100)

Returns: INT (points earned, rounded to nearest integer)
Implementation:

```sql
CREATE FUNCTION fn_calculate_points(
    p_amount DECIMAL(10,2),
    p_rate DECIMAL(5,2)
)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE v_raw_points DECIMAL(10,2);
    DECLARE v_points INT;

    SET v_raw_points = (p_amount * p_rate / 100) / 1000;
    SET v_points = ROUND(v_raw_points, 0);

    RETURN v_points;
END;
```

Listing 39: *fn_calculate_points function*

Calculation Example:

- Spending: 100,000 VND

- Member rate: 5% box office, 3% concession
- Ticket points: (100,000 × 5 / 100) / 1000 = 5 points
- F&B points: (50,000 × 3 / 100) / 1000 = 1.5 → 2 points (rounded)

Usage: Called by `sp_confirm_payment` and `sp_update_customer_membership` to calculate points earned from completed transactions.
Design Rationale:

- Centralized calculation ensures consistency across all point-earning scenarios
- Rounding applied after rate calculation prevents accumulation of rounding errors
- Different rates for box office vs. concessions supported through separate function calls

5.2.2 Function: fn_count_showtimes_in_range

Purpose: Counts total number of showtimes for a theater within a date range.
Implementation:

```sql
CREATE FUNCTION "fn_count_showtimes_in_range"(
    p_theater_id INT,
    p_start_date DATE,
    p_end_date DATE
) RETURNS int
    READS SQL DATA
BEGIN
    DECLARE v_total INT DEFAULT 0;
    DECLARE v_current_date DATE;

    IF p_theater_id IS NULL OR p_theater_id <= 0 THEN RETURN -1; END IF;
    IF p_start_date IS NULL OR p_end_date IS NULL THEN RETURN -2; END IF;
    IF p_start_date > p_end_date THEN RETURN -3; END IF;

    SET v_current_date = p_start_date;

    date_loop: LOOP
        IF v_current_date > p_end_date THEN
            LEAVE date_loop;
        END IF;

        SET v_total = v_total + (
            SELECT COUNT(*)
            FROM showtime
            WHERE au_theater_id = p_theater_id
              AND date = v_current_date
        );

        SET v_current_date = v_current_date + INTERVAL 1 DAY;
    END LOOP;

    RETURN v_total;
END;
```

Listing 40: *fn_count_showtimes_in_range function*

Parameters:

- p_theater_id: Theater ID to query
- p_start_date: Range start date
- p_end_date: Range end date

Returns: INT (total showtime count, or negative error code)
Error Codes:

- -1: Invalid theater ID (NULL or ≤ 0)
- -2: NULL date parameters
- -3: Start date after end date

5.3 Membership Management Functions

5.3.1 Function: fn_get_default_membership

Purpose: Returns the default membership tier for new customers.
Returns: VARCHAR(50) (tier name)
Implementation:

```
CREATE FUNCTION fn_get_default_membership()
RETURNS VARCHAR(50) CHARSET utf8mb4 COLLATE utf8mb4_unicode_ci
READS SQL DATA
DETERMINISTIC
BEGIN
    RETURN 'Member';
END;
```

Listing 41: *fn_get_default_membership function*

Usage: Called by `trg_after_user_insert` trigger when creating customer records for users who don't qualify for U22 (based on age).
Design Note: This function encapsulates the default tier logic, allowing future changes to default tier assignment without modifying trigger code.

5.4 Revenue Analysis Functions

These functions implement complex business analytics using cursors and loops to calculate revenue metrics that support management decision-making and performance analysis.

5.4.1 Function: fn_calculate_avg_booking_value_by_month

Purpose: Calculates the average booking value (tickets + F&B) for all paid bookings within a specific month and year. This metric helps management track booking trends and revenue performance over time.
Parameters:

- `p_year`: Year to analyze (INT)
- `p_month`: Month to analyze (1-12)

Returns: DECIMAL(10,2) (average booking value in VND, or negative error code)
Implementation:

```sql
CREATE FUNCTION fn_calculate_avg_booking_value_by_month(p_year INT, p_month INT)
RETURNS DECIMAL(10,2)
READS SQL DATA
BEGIN
    DECLARE v_total_amount DECIMAL(12,2) DEFAULT 0.00;
    DECLARE v_booking_id BIGINT;
    DECLARE v_booking_value DECIMAL(12,2);
    DECLARE v_booking_status VARCHAR(255);
    DECLARE v_booking_count INT DEFAULT 0;
    DECLARE v_done INT DEFAULT 0;

    DECLARE booking_cursor CURSOR FOR
        SELECT b.id, b.status
        FROM booking b
        WHERE YEAR(b.created_time_at) = p_year AND MONTH(b.created_time_at) =
        ↪  p_month;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = 1;

    IF p_year IS NULL OR p_year < 2000 OR p_year > 2100 THEN RETURN -1; END IF;
    IF p_month IS NULL OR p_month < 1 OR p_month > 12 THEN RETURN -2; END IF;

    OPEN booking_cursor;
    booking_loop: LOOP
        FETCH booking_cursor INTO v_booking_id, v_booking_status;
        IF v_done = 1 THEN LEAVE booking_loop; END IF;
        IF v_booking_status = 'Paid' THEN
            SET v_booking_value = (
                SELECT COALESCE(SUM(ss.price), 0)
                FROM showtime_seat ss
                WHERE ss.booking_id = v_booking_id AND ss.status IN ('Held',
                ↪  'Booked')
            ) + (
                SELECT COALESCE(SUM(f.price * f.quantity), 0)
                FROM fwb f
                WHERE f.booking_id = v_booking_id
            );
            SET v_total_amount = v_total_amount + v_booking_value;
            SET v_booking_count = v_booking_count + 1;
        END IF;
    END LOOP;
    CLOSE booking_cursor;

    IF v_booking_count = 0 THEN RETURN 0.00; END IF;

    RETURN ROUND(v_total_amount / v_booking_count, 2);
END;
```

Listing 42: *fn_calculate_avg_booking_value_by_month function*

5.4.2 Function: fn_calculate_total_revenue_by_customer

Purpose: Calculates the total lifetime revenue generated by a specific customer across all their paid bookings, including both ticket sales and F&B purchases. This function supports customer value analysis and VIP tier qualification.

Implementation:

```sql
CREATE FUNCTION fn_calculate_total_revenue_by_customer(p_customer_id BIGINT)
RETURNS DECIMAL(12,2)
READS SQL DATA
BEGIN
    DECLARE v_total_revenue DECIMAL(12,2) DEFAULT 0.00;
    DECLARE v_item_price DECIMAL(10,2);
    DECLARE v_done INT DEFAULT 0;

    DECLARE ticket_cursor CURSOR FOR
        SELECT ss.price
        FROM showtime_seat ss
        INNER JOIN booking b ON ss.booking_id = b.id
        WHERE b.customer_id = p_customer_id AND b.status = 'Paid' AND ss.status IN
        ↪  ('Held', 'Booked');

    DECLARE fwb_cursor CURSOR FOR
        SELECT f.price * f.quantity
        FROM fwb f
        INNER JOIN booking b ON f.booking_id = b.id
        WHERE b.customer_id = p_customer_id AND b.status = 'Paid';

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = 1;

    IF p_customer_id IS NULL OR p_customer_id <= 0 THEN RETURN -1; END IF;

    IF NOT EXISTS (SELECT 1 FROM customer WHERE user_id = p_customer_id)
    THEN RETURN -2;
    END IF;

    OPEN ticket_cursor;
    ticket_loop: LOOP
        FETCH ticket_cursor INTO v_item_price;
        IF v_done = 1 THEN LEAVE ticket_loop; END IF;
        SET v_total_revenue = v_total_revenue + v_item_price;
    END LOOP;
    CLOSE ticket_cursor;

    SET v_done = 0;
    OPEN fwb_cursor;
    fwb_loop: LOOP
        FETCH fwb_cursor INTO v_item_price;
        IF v_done = 1 THEN LEAVE fwb_loop; END IF;
        SET v_total_revenue = v_total_revenue + v_item_price;
    END LOOP;
    CLOSE fwb_cursor;

    RETURN ROUND(v_total_revenue, 2);
END;
```

Listing 43: *fn_calculate_total_revenue_by_customer function*

Parameters:

- `p_customer_id`: Customer user ID to analyze (BIGINT)

Returns: DECIMAL(12,2) (total revenue in VND, or negative error code)
Business Use Cases:

- Customer lifetime value (CLV) calculation
- VIP and VVIP tier qualification based on total spending
- Identification of high-value customers for targeted marketing
- Customer segmentation for personalized offers and promotions
- Revenue attribution per customer for financial reporting

5.5 Function Integration with Database Objects

Usage in Stored Procedures:

- `sp_register_user`: Uses `fn_validate_email`, `fn_check_email_exists`, `fn_validate_age`
- `sp_confirm_payment`: Uses `fn_calculate_points` for loyalty point calculation
- `sp_update_customer_membership`: Uses `fn_calculate_points` for separate box office and concession calculations

Usage in Triggers:

- `trg_after_user_insert`: Uses `fn_get_default_membership` to assign initial tier

5.6 Function Call Examples

Example 1: Email Validation in Registration

```
SET v_email_valid = fn_validate_email('user@example.com');
IF NOT v_email_valid THEN
    SET p_message = 'Invalid email format';
    ROLLBACK;
END IF;
```

Listing 44: *Email validation usage example*

Example 2: Points Calculation with Different Rates

```
-- Member tier: 5% box office, 3% concession
SET v_ticket_points = fn_calculate_points(100000, 5.00);  -- 5 points
SET v_fnb_points = fn_calculate_points(50000, 3.00);      -- 1.5 -> 2 points
SET v_total_points = v_ticket_points + v_fnb_points;      -- 7 points total

-- VIP tier: 8% box office, 6% concession
SET v_ticket_points = fn_calculate_points(100000, 8.00);  -- 8 points
SET v_fnb_points = fn_calculate_points(50000, 6.00);      -- 3 points
SET v_total_points = v_ticket_points + v_fnb_points;      -- 11 points total
```

Listing 45: *Points calculation usage example*

Example 3: Showtime Count for Reporting

```sql
-- Count showtimes for theater 1 in November 2025
SELECT fn_count_showtimes_in_range(1, '2025-11-01', '2025-11-30')
AS november_showtimes;

-- Handle error codes
SET v_count = fn_count_showtimes_in_range(p_theater_id, p_start, p_end);
IF v_count < 0 THEN
    CASE v_count
        WHEN -1 THEN SET v_error = 'Invalid theater ID';
        WHEN -2 THEN SET v_error = 'Missing date parameters';
        WHEN -3 THEN SET v_error = 'Invalid date range';
    END CASE;
END IF;
```

Listing 46: *Showtime count usage example*

Example 4: Average Booking Value Analysis

```sql
-- Calculate average booking value for November 2025
SELECT fn_calculate_avg_booking_value_by_month(2025, 11)
AS avg_booking_value;
-- Result: 250000.00 (VND)

-- Compare Q4 2025 monthly averages
SELECT
    'October' AS month,
    fn_calculate_avg_booking_value_by_month(2025, 10) AS avg_value
UNION ALL
SELECT 'November', fn_calculate_avg_booking_value_by_month(2025, 11)
UNION ALL
SELECT 'December', fn_calculate_avg_booking_value_by_month(2025, 12);

-- Handle error cases
SET v_avg = fn_calculate_avg_booking_value_by_month(2025, 13);
IF v_avg = -2 THEN
    SET v_error = 'Invalid month parameter';
END IF;
```

Listing 47: *Average booking value calculation usage example*

Example 5: Customer Lifetime Value Calculation

```sql
-- Calculate total revenue from customer ID 1001
SELECT fn_calculate_total_revenue_by_customer(1001)
AS customer_lifetime_value;
-- Result: 5250000.00 (VND)

-- Identify top 10 customers by revenue
SELECT
    c.user_id,
    CONCAT(u.fname, ' ', u.lname) AS customer_name,
    fn_calculate_total_revenue_by_customer(c.user_id) AS total_revenue
FROM customer c
JOIN User u ON c.user_id = u.id
ORDER BY total_revenue DESC
LIMIT 10;

-- Check customer qualification for VIP tier (>= 2M VND)
SET v_revenue = fn_calculate_total_revenue_by_customer(p_customer_id);
IF v_revenue >= 2000000 THEN
    SET v_tier = 'VIP';
ELSEIF v_revenue >= 1000000 THEN
    SET v_tier = 'Member';
END IF;

-- Handle error cases
IF v_revenue = -1 THEN
    SET v_error = 'Invalid customer ID';
ELSEIF v_revenue = -2 THEN
    SET v_error = 'Customer not found';
END IF;
```

Listing 48: *Customer total revenue calculation usage example*

These 9 functions work together with stored procedures (Section 4) and triggers (Section 3) to create a comprehensive database programming layer for the BKinema system.

6 Application Implementation

The backend is implemented using NestJS, a modular Node.js framework that supports capabilities such as dependency injection, middleware, and robust error handling. Its native integration with TypeORM facilitates database operations while maintaining compatibility with stored procedures and triggers.
The frontend is developed with React.js, employing a component-based structure that enhances maintainability and rendering performance through the virtual DOM–particularly beneficial for dynamic interfaces such as seat selection and real-time availability updates. React Router manages client-side navigation, the Context API handles global state management, and Axios enables communication with backend services. Authentication is implemented using JSON Web Tokens (JWT) in conjunction with Passport middleware, and user passwords are secured through bcrypt hashing. API documentation is generated automatically via Swagger/OpenAPI specifications.

6.1 Backend Implementation

6.1.1 Authentication Service Implementation

```
class AuthService {
  constructor(
    @InjectDataSource() private dataSource: DataSource,
    private jwtService: JwtService,
  ) {}

  async register(registerDto: RegisterDto): Promise<AuthResponseDto> {
    const { fname, minit, lname, email, password, birthday,
            gender, district, city } = registerDto;

    const hashedPassword = await bcrypt.hash(password, 10);

    await this.dataSource.query(
      `CALL sp_register_user(?, ?, ?, ?, ?, ?, ?, ?, ?,
                             @user_id, @success, @message)`,
      [fname, minit || null, lname, email, hashedPassword,
       birthday || null, gender || 'Other',
       district || null, city || null],
    );

    const [result] = await this.dataSource.query(
      `SELECT @user_id as userId, @success as success,
              @message as message`
    );

    if (!result.success) {
      throw new ConflictException(result.message);
    }

    const payload = { email, sub: result.userId, role: 'customer' };
    return {
      access_token: this.jwtService.sign(payload),
      user: { id: result.userId, email, fname, lname }
    };
  }
}
```

Listing 49: *Authentication service - user registration*

User authentication is managed through a dedicated service that interfaces with the database's stored procedures for registration and login operations. The authentication flow validates credentials, generates JWT tokens, and maintains session security. Listing 49 presents the core authentication logic for user registration.

The registration process leverages the `sp_register_user` stored procedure, which handles database insertion and triggers the automatic creation of a customer record and membership initialization. By delegating business logic to stored procedures, the application layer remains thin and focused on request orchestration and response formatting.

6.1.2 Booking Management Controller

The booking controller exposes RESTful endpoints for initiating bookings, updating food and beverage selections, and retrieving booking history. The implementation coordinates between multiple services to validate seat availability, hold reservations, and manage booking lifecycles. Listing 50 demonstrates the booking endpoint implementations.

```
class BookingController {
  constructor(private readonly bookingService: BookingService) {}

  @Post('start')
  @ApiOperation({ summary: 'Create booking and hold seats' })
  async startBooking(@Body() dto: StartBookingDto) {
    return this.bookingService.startBooking(dto);
  }

  @Post('fwb')
  @ApiOperation({ summary: 'Update food & beverage items' })
  async updateFwb(@Body() dto: UpdateFwbDto) {
    return this.bookingService.updateBookingFwb(dto);
  }

  @Get('my-bookings')
  @UseGuards(JwtAuthGuard)
  @ApiOperation({ summary: 'Get customer bookings with pagination' })
  async getMyBookings(
    @Request() req,
    @Query('limit') limit?: string,
    @Query('offset') offset?: string,
  ) {
    const parsedLimit = limit ? parseInt(limit) : 5;
    const parsedOffset = offset ? parseInt(offset) : 0;
    return this.bookingService.getMyBookings(
      req.user.userId, parsedLimit, parsedOffset
    );
  }
}
```

Listing 50: *Booking controller endpoints*

The `JwtAuthGuard` decorator ensures that only authenticated users can access protected endpoints such as booking history retrieval. The pagination parameters in the `getMyBookings` endpoint default to 5 items per page, aligning with the user interface design that displays bookings in manageable chunks.

6.2 Frontend Implementation

The frontend application is structured as a single-page application (SPA) utilizing React Router for navigation and component composition. The architecture emphasizes component reusability, state management through Context API, and responsive design principles to ensure optimal user experience across devices.

6.2.1 Booking Context State Management

The booking context implements a centralized state management solution for the multi-step booking process, maintaining data consistency as users progress through seat selection, combo selection, and payment. Listing 51 demonstrates the context implementation with state persistence.

```jsx
const BookingContext = createContext();

export function BookingProvider({ children }) {
  const [bookingData, setBookingData] = useState(() => {
    const saved = sessionStorage.getItem('bookingData');
    return saved ? JSON.parse(saved) : {
      selectedSeats: [],
      selectedCombos: [],
      showtimeId: null,
      theaterId: null,
      totalPrice: 0,
      comboTotal: 0,
    };
  });

  useEffect(() => {
    sessionStorage.setItem('bookingData',
                           JSON.stringify(bookingData));
  }, [bookingData]);

  const updateBookingData = (updates) => {
    setBookingData(prev => ({ ...prev, ...updates }));
  };

  const clearBookingData = () => {
    setBookingData({
      selectedSeats: [], selectedCombos: [],
      showtimeId: null, theaterId: null,
      totalPrice: 0, comboTotal: 0,
    });
    sessionStorage.removeItem('bookingData');
  };

  return (
    <BookingContext.Provider value={{
      bookingData, updateBookingData, clearBookingData
    }}>
      {children}
    <BookingContext.Provider>
  );
}

export const useBooking = () => useContext(BookingContext);
```

Listing 51: *Booking context for global state management*

The context utilizes `sessionStorage` for state persistence, ensuring that booking data survives page refreshes but is automatically cleared when the browser session ends. This approach balances user convenience with security, as sensitive payment information is never persisted beyond the current session.

6.3 Application Workflow and User Interface

6.3.1 User Registration and Authentication

The user onboarding process begins with the registration interface, which collects essential user information including name, email, contact details, and demographic data. Figure 1 displays the registration form with client-side validation to ensure data quality before submission.

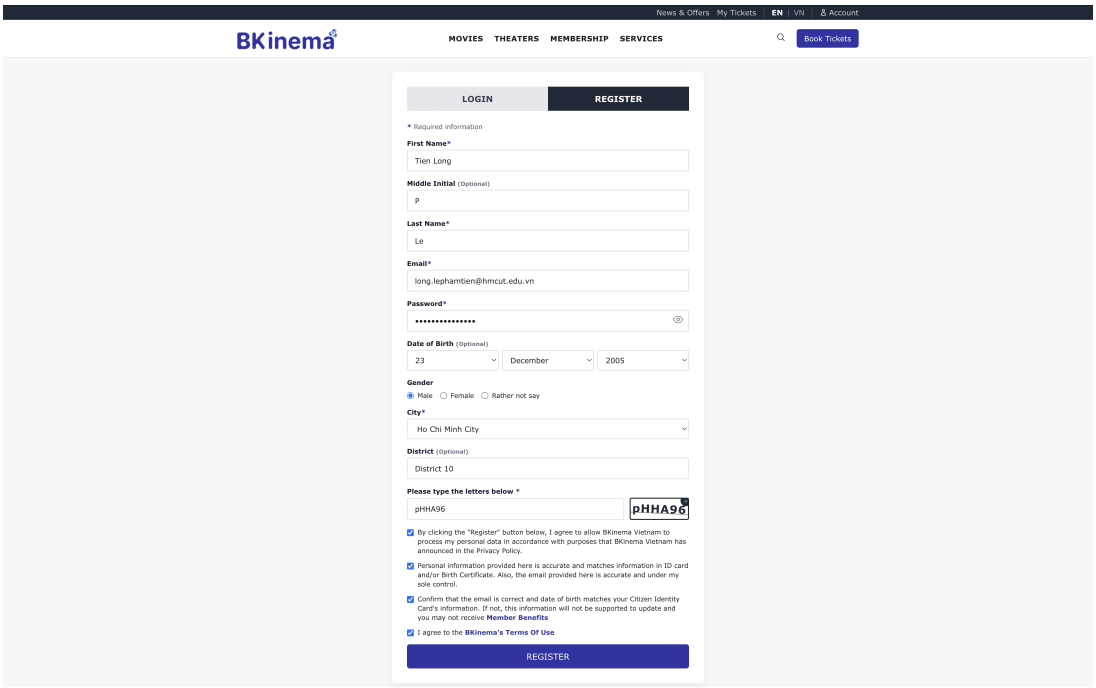

Figure 1: *User registration interface with form validation*

Upon successful registration, the system automatically creates a customer profile with a default membership tier and initializes the customer's point balance. The registration process invokes the `sp_register_user` stored procedure, which triggers the `trg_after_user_insert` trigger to establish the customer relationship and membership record as described in Section 3.

6.3.2 Home Page and Movie Discovery

The home page serves as the primary entry point, featuring promotional banners, highlighted movies, and quick access to currently showing films. The interface employs a carousel for featured content and grid layouts for movie browsing, as illustrated in Figure 2. The home page dynamically loads movie data from the backend API, displaying poster images, titles, ratings, and formats. Users can quickly navigate to detailed movie information or initiate the booking process directly from this interface.

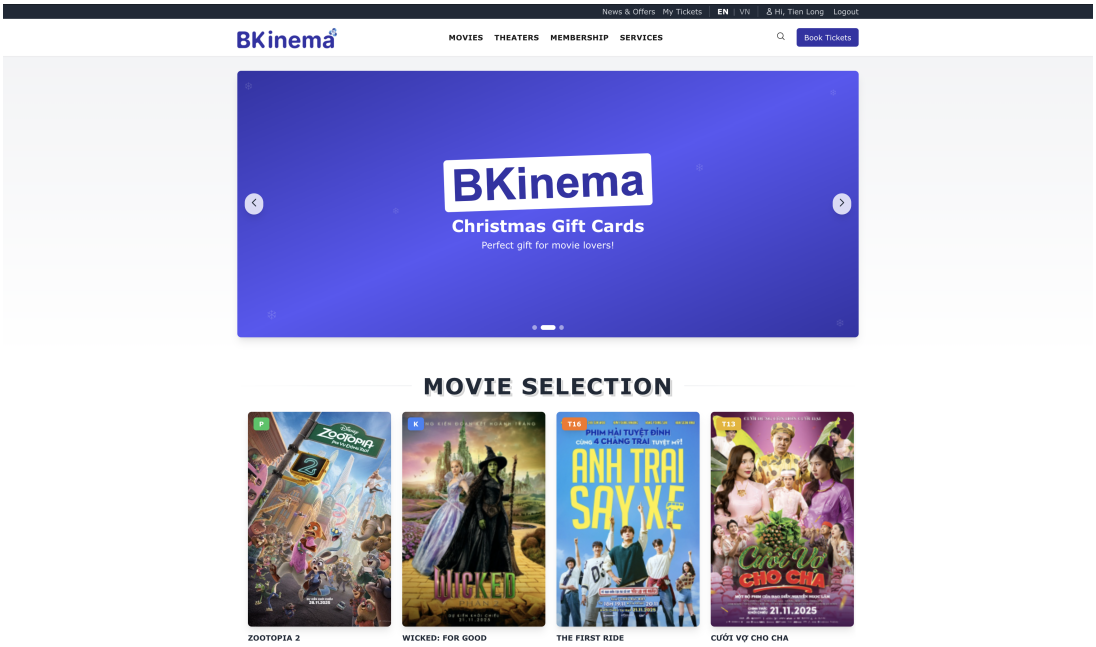

Figure 2: *Home page with featured banners and movie highlights*

6.3.3 Movie Catalog and Filtering

The movies page presents a comprehensive catalog of all currently showing films with filtering and sorting capabilities. Figure 3 demonstrates the movie listing interface with visual cards displaying essential information including poster, title, genre, duration, and rating.

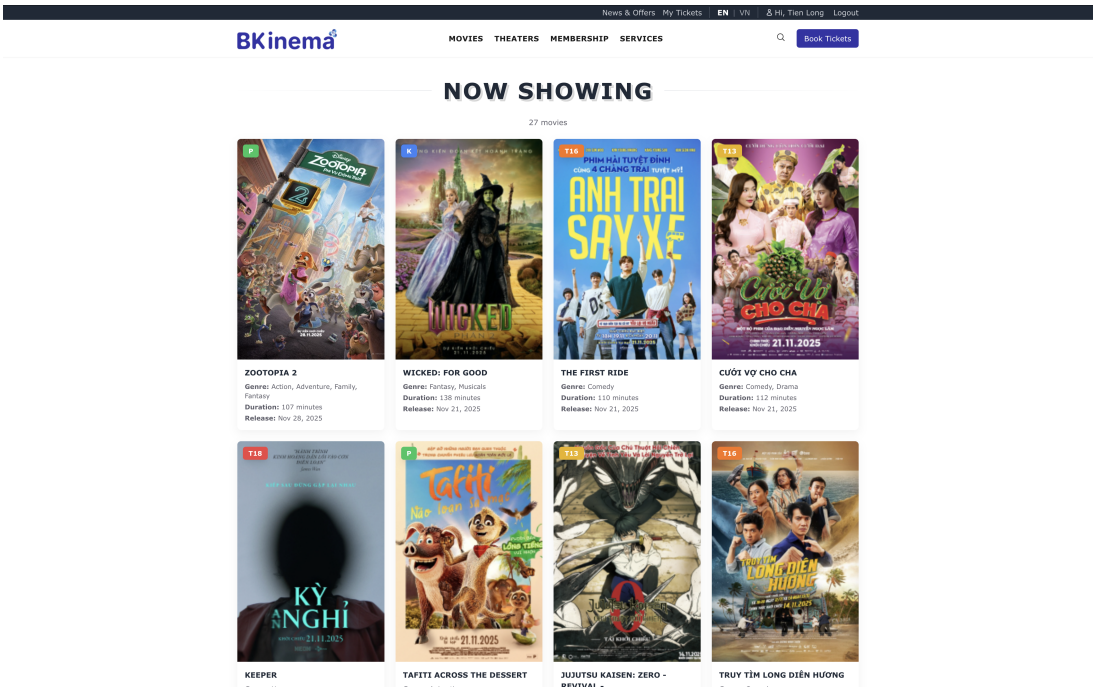

Figure 3: *Movie catalog page showing all now-showing films*

The interface enables users to filter movies by genre and rating, facilitating efficient movie discovery based on personal preferences. Each movie card serves as a clickable link to the detailed movie information page.

6.3.4 Theater Page

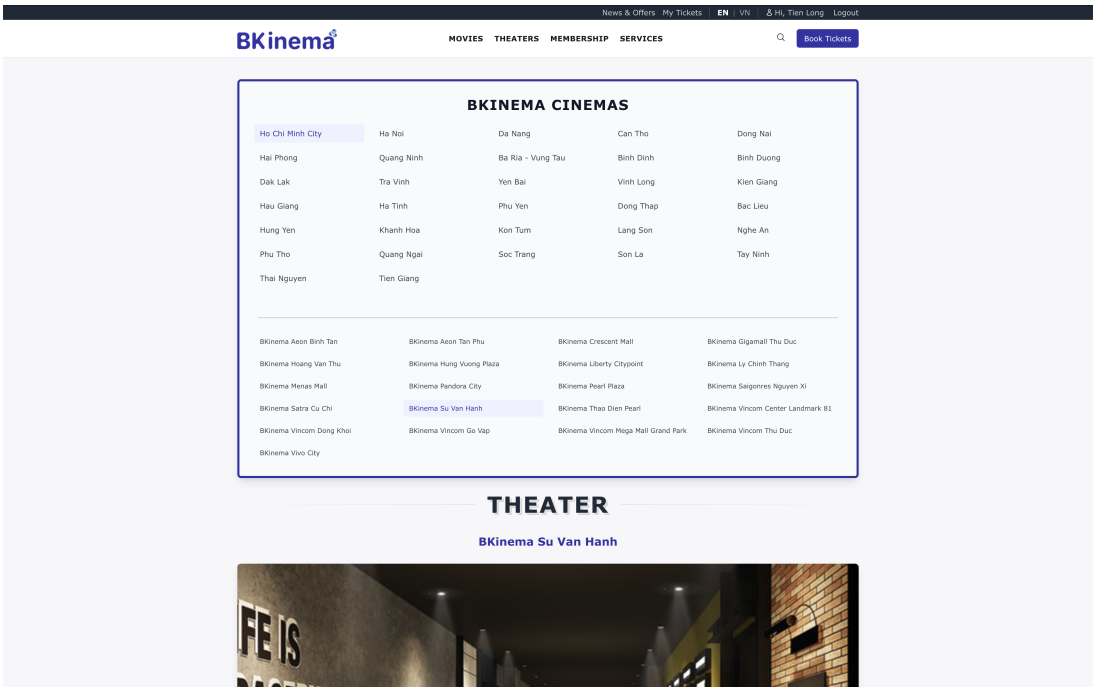

Figure 4: *Theater listing with city selection and showtime display*

Figure 4 presents the theater listing with city-based filtering and showtime availability indicators. Users first select their preferred city from a dropdown menu, which dynamically filters the theater list to display only theaters in that location. Upon selecting a theater, the interface expands to show the theater's address and current showtimes for all movies.

6.3.5 Movie Detail and Showtime Selection

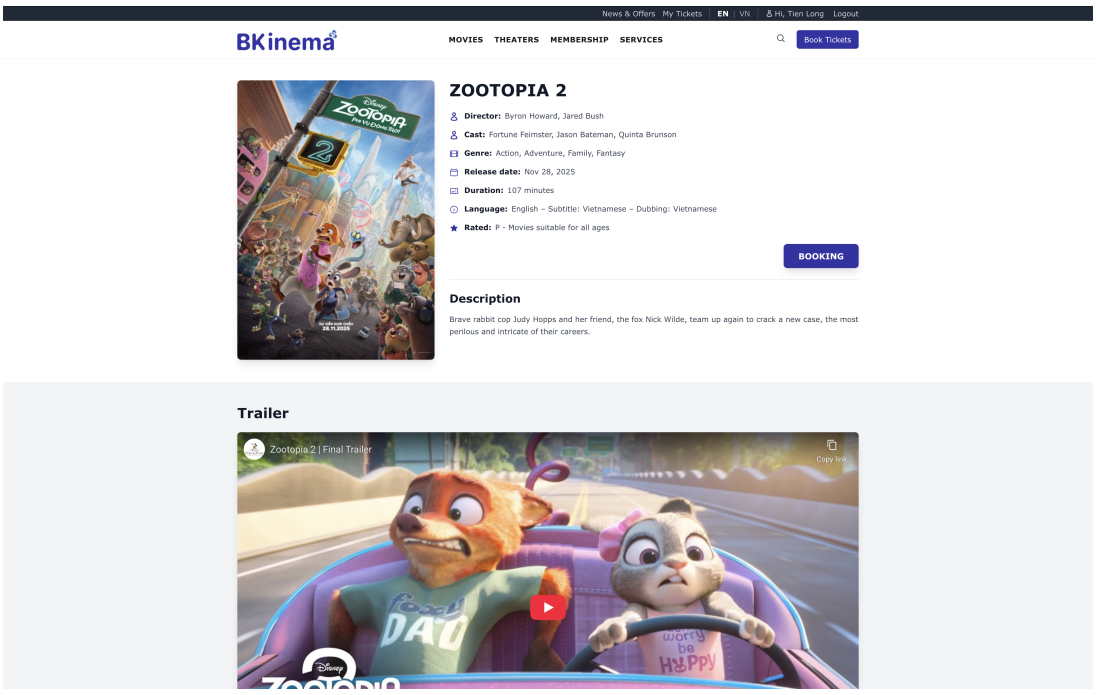

Figure 5: *Movie detail page with comprehensive film information*

The movie detail page aggregates comprehensive information about a selected film, including synopsis, director, cast, duration, language, and age rating. Figure 5 illustrates the detailed movie information layout with integrated booking initiation. The booking button on the movie detail page triggers a showtime selection modal, allowing users to specify their preferred viewing date, city, and theater. Figure 6 demonstrates the showtime selection interface that appears as an overlay, displaying available showtimes organized by date and theater.

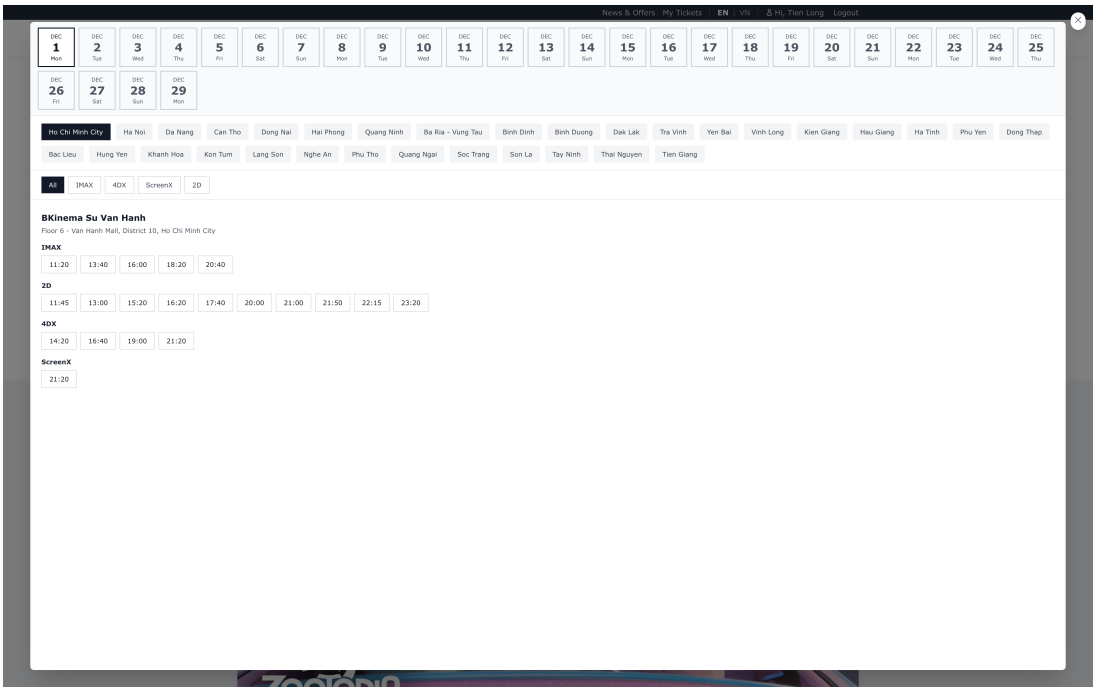

Figure 6: *Showtime selection modal with date, city, and theater filtering*

6.3.6 Seat Selection Interface

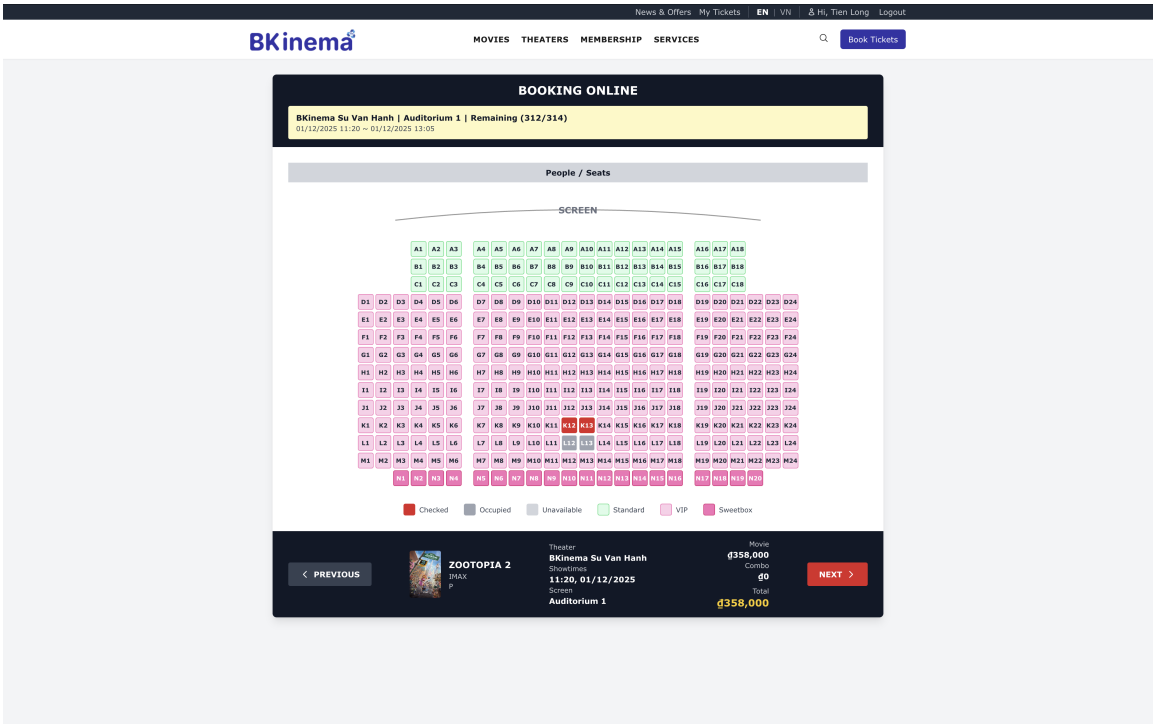

Figure 7: *Interactive seat selection interface with real-time availability*

The seat selection page presents an interactive auditorium layout where users can view seat availability and select their preferred seats. Figure 7 shows the seat map with color-coded indicators for available, occupied, and selected seats.

6.3.7 Food and Beverage Selection

Following seat selection, users proceed to the food and beverage (F&B) selection page where they can add combo items to their order. Figure 8 displays the F&B menu with item descriptions, prices, and quantity selectors.

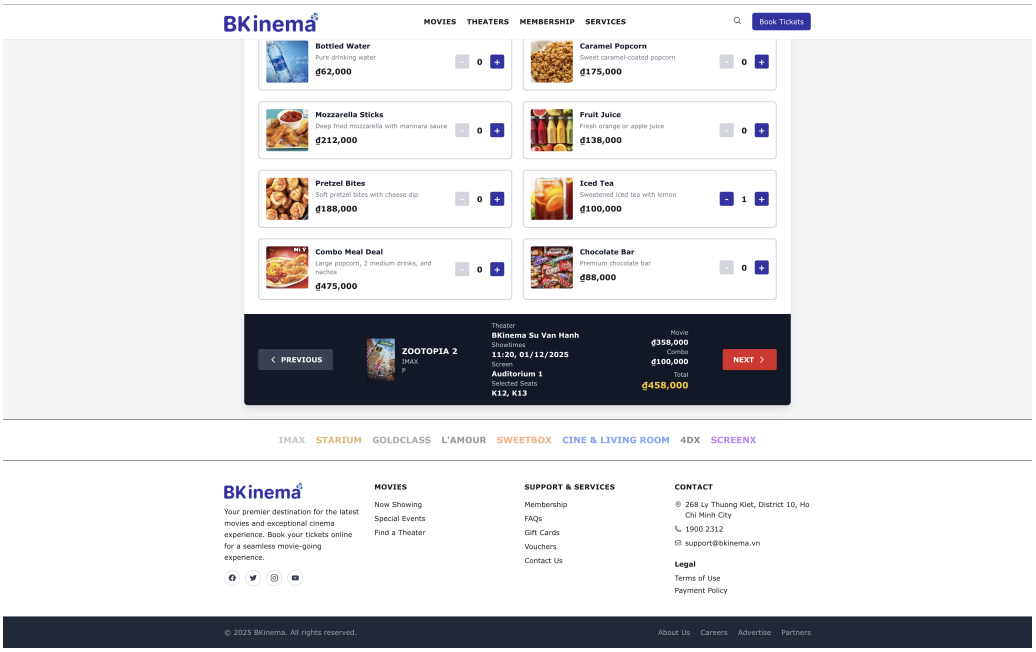

Figure 8: *Food and beverage selection interface*

6.3.8 Payment and Point Redemption

The payment page consolidates the booking summary, calculates the final total including taxes, and provides payment options. Figure 9 illustrates the payment interface with membership point application functionality.

Authenticated customers can apply their accumulated membership points toward the booking cost, with the system displaying available points, conversion rate, and the resulting discount. The point application logic invokes the `sp_apply_points` stored procedure, which validates point availability, calculates the redemption amount, and updates the customer's point balance within a database transaction to ensure atomicity.

Upon payment confirmation, the system processes the transaction through the `sp_process_payment` stored procedure, which creates ticket records, updates seat availability, records the transaction in the payment ledger, and awards membership points based on the customer's tier multiplier.

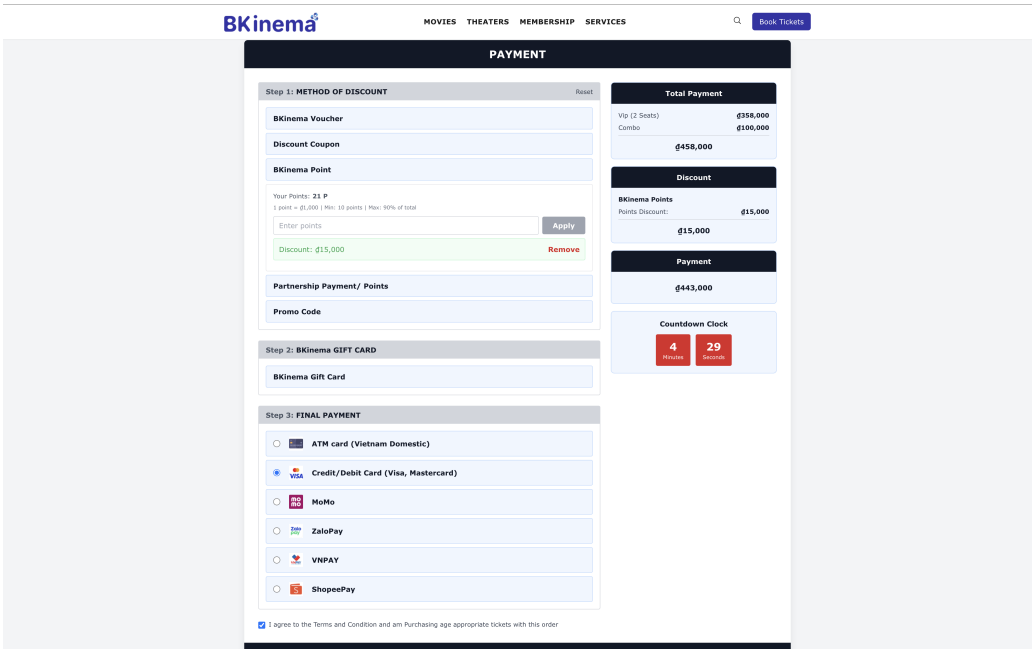

Figure 9: *Payment page with point redemption interface*

6.3.9 Booking Management and History

Logged-in customers can access their booking history through the customer account dashboard. Figure 10 shows the booking history interface with pagination support for efficient navigation through past bookings.

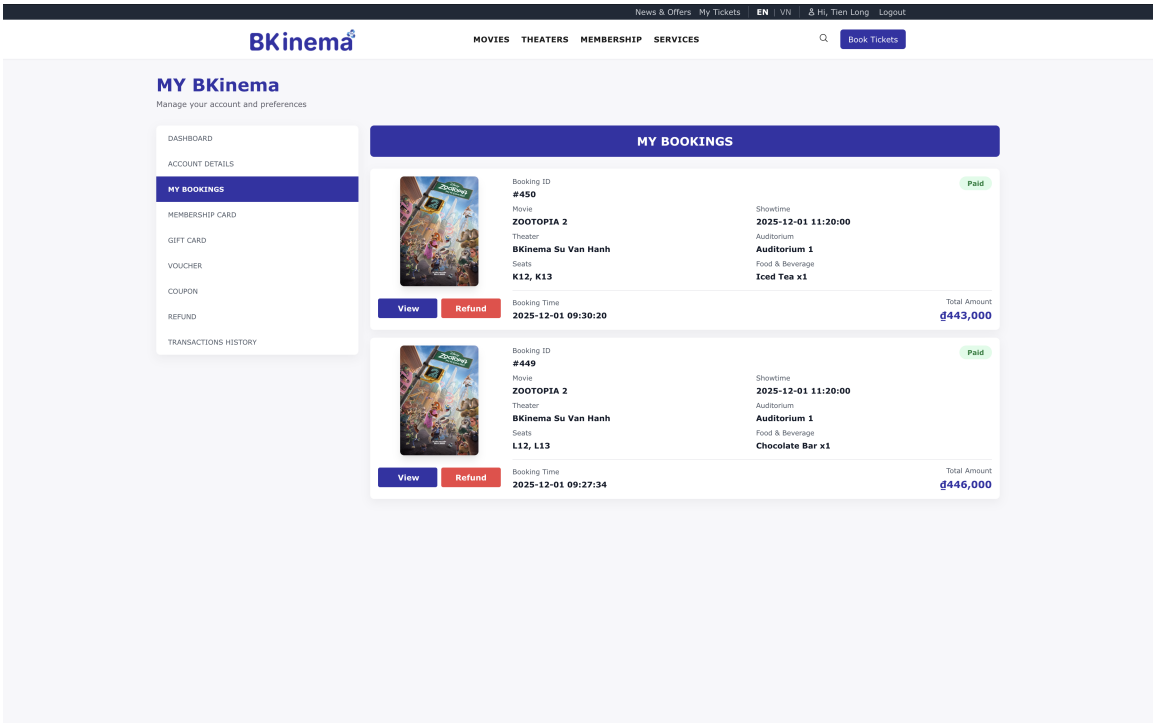

Figure 10: *Customer booking history with pagination (5 bookings per page)*

Each booking entry displays essential information including movie title, showtime, theater, seats, total cost, and booking status (Pending, Confirmed, Cancelled). The interface implements server-side pagination with a fixed limit of 5 bookings per page, querying the database through the `getMyBookings` service method that executes appropriate SELECT statements with LIMIT and OFFSET clauses.

6.3.10 Membership Card

The membership dashboard provides customers with visibility into their current membership tier, accumulated points, and points transaction history. Figure 11 displays the membership interface showing tier benefits and point balance.

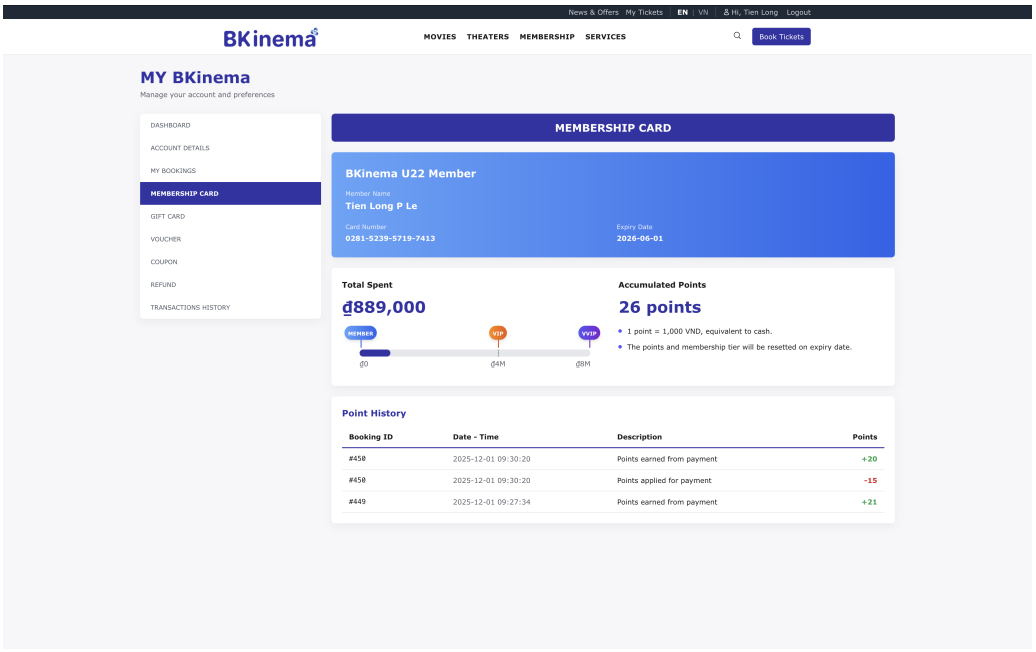

Figure 11: *Membership dashboard showing tier status and points history*

The points history section displays all point transactions including earnings from bookings, redemptions for discounts, and expirations due to the 365-day validity period. The interface invokes the `fn_get_available_points` function to calculate the current valid point balance, excluding expired points from the total.

The tier display shows progression toward the next membership level, calculated based on the total booking amount over the past 365 days. This information is retrieved through queries that aggregate booking totals and compare against the tier thresholds defined in the `Membership` table, as established in Section 2.

7 Conclusion

7.1 Summary

This report presented the complete implementation of the BKinema movie ticket booking system, demonstrating a fully functional solution that integrates an optimized database design with modern full-stack web development. The system successfully evolved from conceptual modeling to a deployable application capable of managing complex workflows including booking, membership, and payment processing.

The database consists of 14 relational tables supporting authentication with role-based access control, theater and auditorium configuration, rich movie metadata, dynamic seat allocation, multi-method payment processing, a tiered membership model, food and beverage management, and coupon/refund operations. Referential integrity is enforced through foreign keys, CHECK constraints, and ENUM definitions.

Advanced SQL programming is implemented through four triggers and five stored procedures to automate business rules such as showtime conflict validation, age-rating enforcement, member profile creation, booking and seat allocation, and payment handling with point accumulation. Four user-defined functions provide reusable logic for point calculation, tier upgrade cost evaluation, seat availability, and age verification. Comprehensive validation, transaction management, and detailed error reporting ensure data consistency.

The backend, built with NestJS, exposes a type-safe API layer that interacts with the database via parameterized stored procedure calls. The React-based frontend delivers responsive interfaces for browsing

movies, selecting seats, completing bookings, and managing payments. Critical operations rely on transactional guarantees, while performance optimization uses aggregated queries, efficient JOIN patterns, and server-side pagination.

7.2 Future Works

Future development can improve scalability, automation, and user experience. Booking timeout handling may be enhanced through MySQL event scheduling to automatically release expired seat locks. Search functionality could be upgraded using MySQL FULLTEXT or external engines such as Elasticsearch for relevance ranking and fuzzy matching.
Analytical features could be expanded through stored procedures and views to support reporting on revenue, utilization, membership behavior, and sales insights. Integration with email/SMS services would enable automated confirmations, reminders, and promotions. Caching with Redis would improve performance for frequently requested data, while cloud storage (e.g., AWS S3, GCS, Cloudinary) would enhance media management and delivery.
Architecture evolution toward microservices and WebSocket-based real-time seat updates would improve scalability and responsiveness. Native mobile applications could extend accessibility and integrate biometric authentication and push notifications. An administrative dashboard would provide comprehensive operational control.
Advanced extensions include machine-learning-based recommendation systems and dynamic pricing models, as well as third-party integrations such as TMDB/IMDb for metadata and VNPay, MoMo, or ZaloPay for localized payment support.

A Additional Application Screenshots

This appendix contains additional screenshots of the BKinema web application user interface that complement the main application flow described in Section 6.

A.1 FAQ Page

The FAQ (Frequently Asked Questions) page provides users with answers to common questions about the booking process, payment methods, refund policies, and membership benefits.

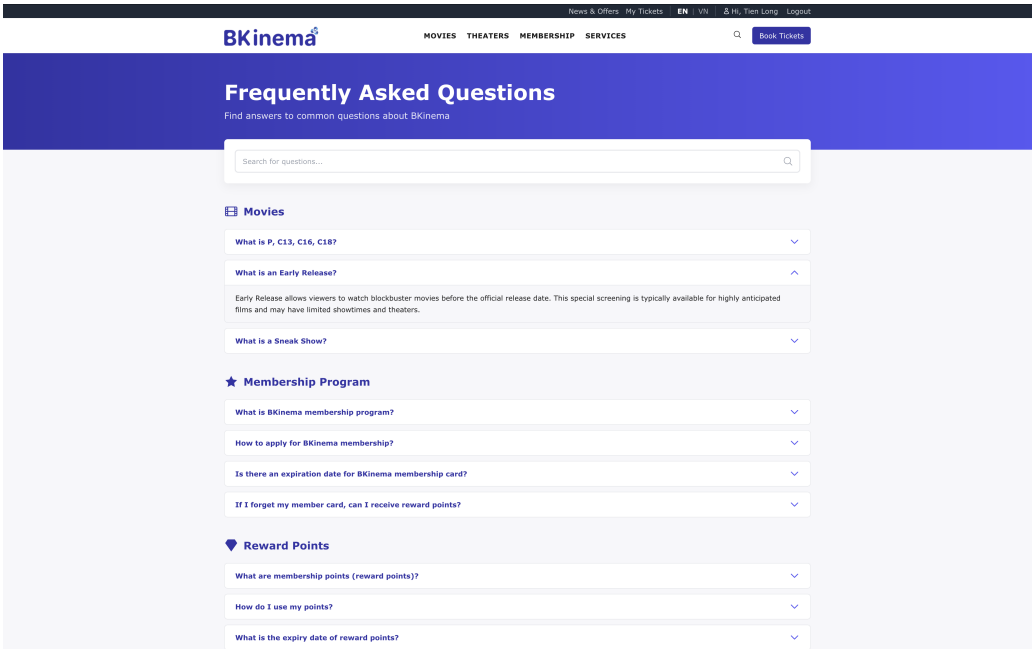

Figure 12: *FAQ page interface*

A.2 Membership Information Page

The membership page displays detailed information about BKinema's loyalty program tiers (Member, U22, VIP, VVIP), including benefits, point accumulation rates, and tier requirements.

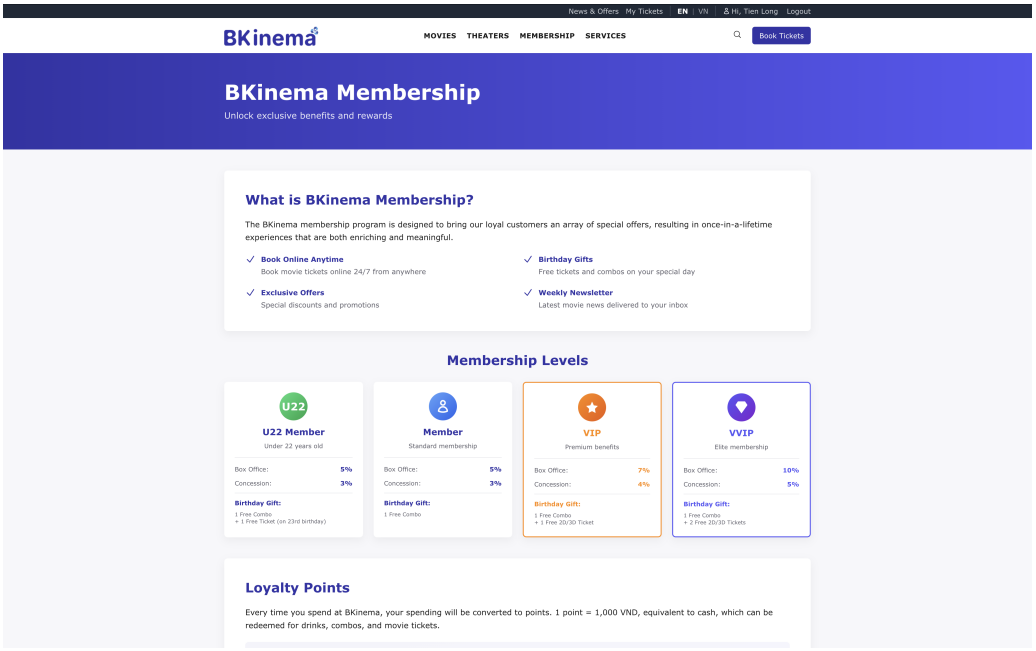

Figure 13: *Membership information page*

A.3 Payment Policy Page

This page outlines the payment policies, accepted payment methods, security measures, and terms and conditions for online transactions.

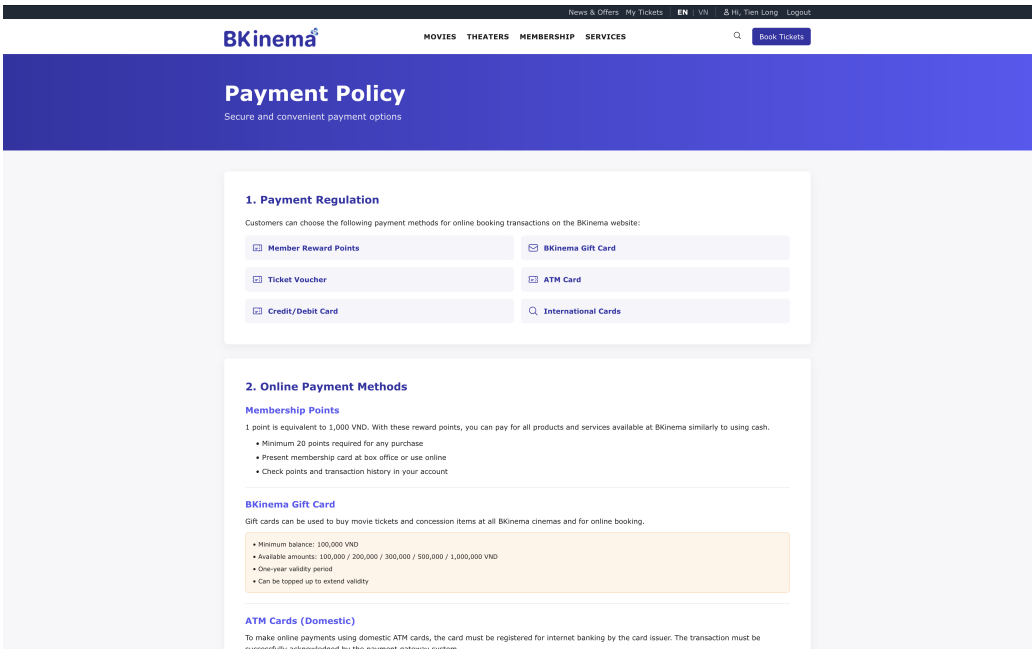

Figure 14: *Payment policy page*

A.4 Terms of Use Page

The Terms of Use page contains the legal terms and conditions that govern the use of the BKinema platform, including user responsibilities, prohibited activities, and liability disclaimers.

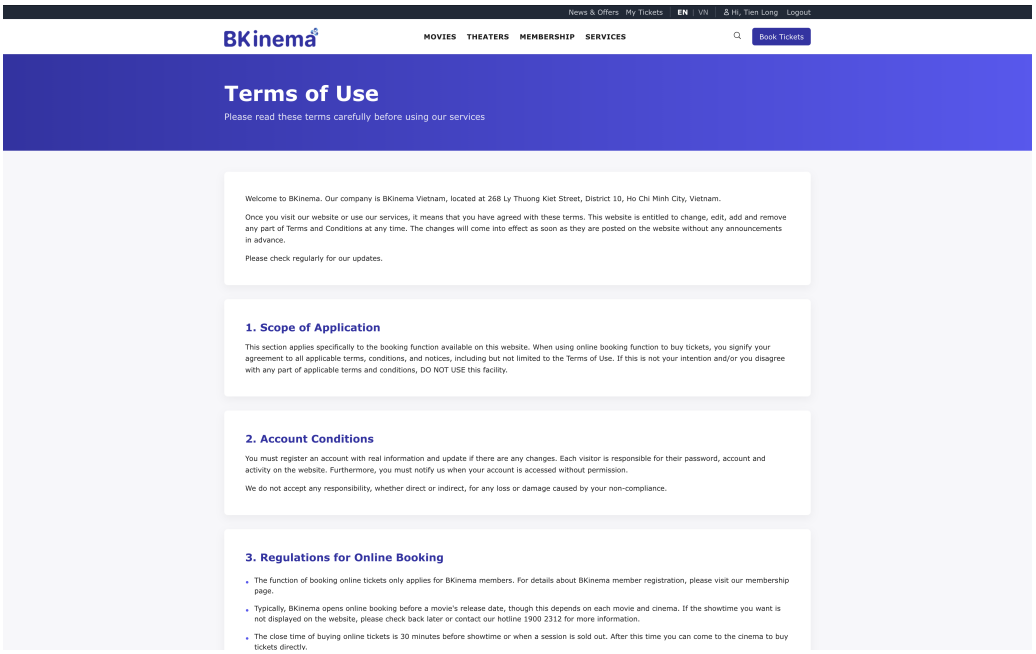

Figure 15: *Terms of use page*

A.5 Refund Request Interface

The refund request interface allows customers to submit refund requests for their bookings. The system validates refund eligibility based on showtime timing and processes the request through stored procedures.

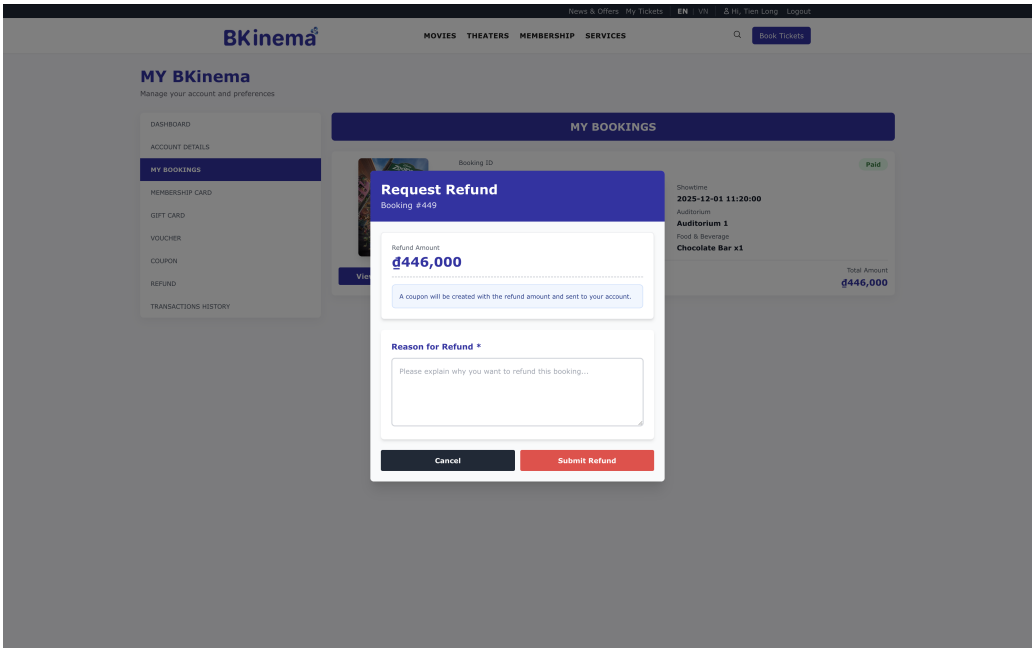

Figure 16: *Refund request interface*

A.6 Ticket View Interface

After successful booking, customers can view their ticket details including QR code, seat information, showtime details, and booking reference number.

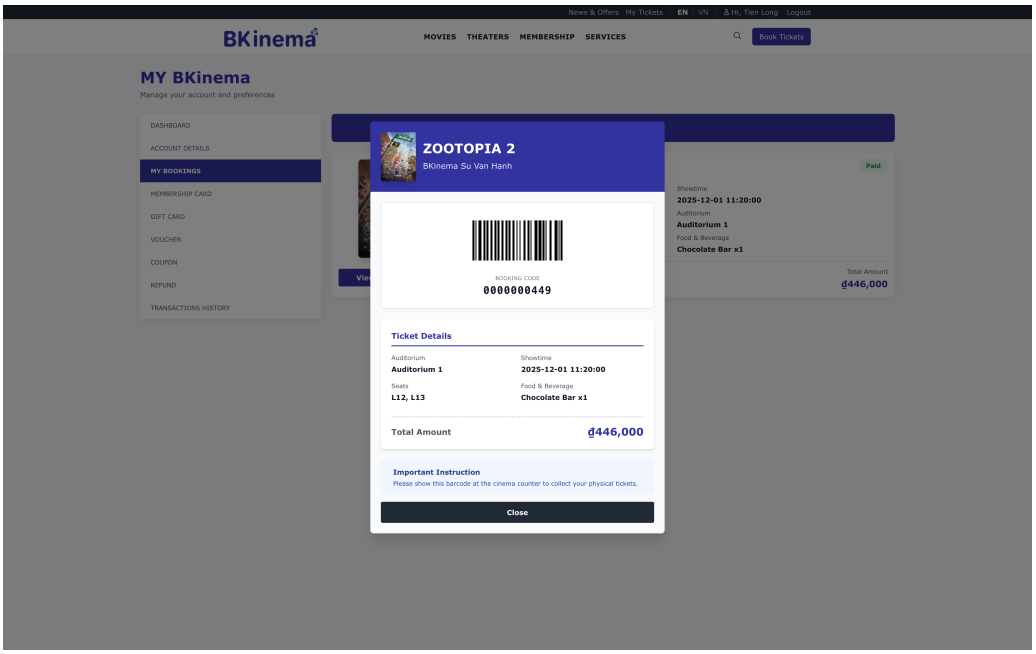

Figure 17: *Ticket view interface*

B Database Connection Configuration

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  inject: [ConfigService],
    useFactory: (config: ConfigService) => {
     const db = config.get('database') as {
      host?: string;
      port?: number;
      username?: string;
      password?: string;
      database?: string;
      ca?: string;
    } | undefined;

      const ssl = db?.ca ? { ca: db.ca } : undefined;

      return {
        type: 'mysql' as const,
        host: db?.host,
        port: db?.port,
        username: db?.username,
        password: db?.password,
        database: db?.database,
        extra: ssl ? { ssl } : {},
      };
    },
}),

export class AppModule {}
```

Listing 52: *Application module with database configuration*

The backend connects to MySQL using TypeORM with connection pooling for optimal performance. The application module configures the database connection asynchronously using the ConfigService to load environment variables.

Code Availability

The complete implementation of the BKinema system, including all source code, stored procedures, triggers, and application code, is available on GitHub. Please access the repository via `https://github.com/longlephamtien/DS251-Assignment`.

Acknowledgement

We would like to express our sincere gratitude to Mr. Le Duc Hoang Nam for valuable guidance and support throughout this study.