Database Operations in Gaming Club Management System

This document explains how DDL (Data Definition Language), DML (Data Manipulation Language), and DQL (Data Query Language) commands are generated and executed in the Spring Boot Gaming Club application.

Table of Contents

Architecture Overview

The application uses Spring Data JPA with Hibernate as the ORM (Object-Relational Mapping) framework. Here's how SQL commands flow:

```
Java Entity Classes → Hibernate → SQL Commands → MySQL Database
```

DDL (Data Definition Language)

How DDL Commands are Generated

Configuration in `application.properties`:

```
spring.jpa.hibernate.ddl-auto=update
```

DDL Generation Process

1. Automatic Table Creation: Hibernate scans entity classes and generates CREATE TABLE statements
2. Schema Updates: When entities change, Hibernate generates ALTER TABLE statements
3. Constraint Creation: Primary keys, foreign keys, and unique constraints are automatically created

Generated DDL Examples

For `Member` Entity:

```sql
CREATE TABLE members (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    phone VARCHAR(255) NOT NULL UNIQUE,
    balance DOUBLE PRECISION NOT NULL,
    membership_type VARCHAR(255),
    total_games_played INT
);
```

For `Game` Entity:

```
CREATE TABLE games (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DOUBLE PRECISION NOT NULL,
    description VARCHAR(255),
    category VARCHAR(255),
    difficulty_level VARCHAR(255),
    is_available BOOLEAN
);
```

For `PlayHistory` Entity:

```
CREATE TABLE play_history (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    date_time DATETIME NOT NULL,
    amount DOUBLE PRECISION NOT NULL,
    game_id BIGINT,
    member_id BIGINT,
    FOREIGN KEY (game_id) REFERENCES games(id),
    FOREIGN KEY (member_id) REFERENCES members(id)
);
```

DDL Generation Modes

Mode	Description	Usage
`create`	Drops and recreates tables	Development
`update`	Updates schema	Development
`create-drop`	Creates on start, drops on exit	Testing
`validate`	Validates schema	Production
`none`	No DDL generation	Production

DML (Data Manipulation Language)

How DML Commands are Generated

DML commands (INSERT, UPDATE, DELETE) are generated through Spring Data JPA Repository methods.

INSERT Operations

Service Layer:

```java
// MemberService.java
public Member createMember(MemberRequest request) {
    Member member = new Member(request.getName(), request.getPhone(), request.getFee());
    Member savedMember = memberRepository.save(member); // Generates INSERT
}
```

Generated SQL:

```sql
INSERT INTO members (name, phone, balance, membership_type, total_games_played)
VALUES (?, ?, ?, ?, ?)
```

Parameters: - ?1 = "John Doe" - ?2 = "9876543210" - ?3 = 1000.0 - ?4 = "REGULAR" - ?5 = 0

UPDATE Operations

Service Layer:

```java
// PlayService.java
member.setBalance(member.getBalance() - game.getPrice());
member.setTotalGamesPlayed(member.getTotalGamesPlayed() + 1);
memberRepository.save(member); // Generates UPDATE
```

Generated SQL:

```sql
UPDATE members
SET balance = ?, total_games_played = ?
WHERE id = ?
```

DELETE Operations

Repository Usage:

```java
// Direct repository call
memberRepository.deleteById(1L);

// Delete all (used in tests)
memberRepository.deleteAll();
```

Generated SQL:

```sql
DELETE FROM members WHERE id = ?
```

Transaction Management

In Service Layer:

```java
@Transactional
public String playGame(PlayRequest request) {
    // Multiple DML operations in single transaction
    memberRepository.save(member);        // UPDATE
    playHistoryRepository.save(history);  // INSERT
}
```

Generated SQL Sequence:

```sql
START TRANSACTION;

UPDATE members SET balance = 950.0, total_games_played = 1 WHERE id = 1;

INSERT INTO play_history (date_time, amount, game_id, member_id)
VALUES ('2024-01-15 10:30:00', 50.0, 1, 1);

COMMIT;
```

DQL (Data Query Language)

How DQL Commands are Generated

DQL commands (SELECT) are generated through Spring Data JPA query methods and custom `@Query` annotations.

Basic Find Methods

Repository Interface:

```java
public interface MemberRepository extends JpaRepository<Member, Long> {
    Optional<Member> findByPhone(String phone);
    boolean existsByPhone(String phone);
}
```

Generated SQL for `findByPhone`:

```sql
SELECT m.id, m.name, m.phone, m.balance, m.membership_type, m.total_games_played
FROM members m
WHERE m.phone = ?
```

Generated SQL for `existsByPhone`:

```sql
SELECT COUNT(m.id) > 0
FROM members m
WHERE m.phone = ?
```

Custom Query Methods

Repository with Custom Query:

```
public interface RechargeHistoryRepository extends JpaRepository<RechargeHistory, Long> {

    @Query("SELECT rh FROM RechargeHistory rh WHERE DATE(rh.dateTime) = :date")
    List<RechargeHistory> findByDate(LocalDate date);

    List<RechargeHistory> findByMemberId(Long memberId);
}
```

Generated SQL for **`findByDate`**:

```
SELECT rh.id, rh.amount, rh.date_time, rh.member_id
FROM recharge_history rh
WHERE DATE(rh.date_time) = ?
```

Generated SQL for **`findByMemberId`**:

```
SELECT rh.id, rh.amount, rh.date_time, rh.member_id
FROM recharge_history rh
WHERE rh.member_id = ?
```

Complex Query Generation

Member Search Service:

```
public MemberResponse searchMember(String phone) {
    // Multiple queries executed
    Member member = memberRepository.findByPhone(phone);
    List<RechargeHistory> recharges = rechargeHistoryRepository.findByMemberId(member.getId(
    List<Game> games = gameRepository.findByIsAvailableTrue();
    List<PlayHistory> playHistory = playHistoryRepository.findByMemberId(member.getId());
}
```

Generated SQL Sequence:

```
-- Query 1: Find member by phone
SELECT * FROM members WHERE phone = '9876543210';

-- Query 2: Find recharge history
SELECT * FROM recharge_history WHERE member_id = 1;

-- Query 3: Find available games
SELECT * FROM games WHERE is_available = true;

-- Query 4: Find play history
SELECT ph.*, g.name as game_name, g.category as game_category
FROM play_history ph
JOIN games g ON ph.game_id = g.id
WHERE ph.member_id = 1;
```

SQL Generation Examples

Complete Flow Example: Play Game Operation

Java Code:

```java
@Transactional
public String playGame(PlayRequest request) {
    // 1. SELECT - Find member
    Member member = memberRepository.findById(request.getMemberId());

    // 2. SELECT - Find game
    Game game = gameRepository.findById(request.getGameId());

    // 3. UPDATE - Deduct balance
    member.setBalance(member.getBalance() - game.getPrice());
    memberRepository.save(member);

    // 4. INSERT - Create play history
    PlayHistory history = new PlayHistory(member, game, game.getPrice());
    playHistoryRepository.save(history);
}
```

Generated SQL:

```sql
-- 1. Find Member
SELECT m.id, m.name, m.phone, m.balance, m.membership_type, m.total_games_played
FROM members m
WHERE m.id = 1;

-- 2. Find Game
SELECT g.id, g.name, g.price, g.description, g.category, g.difficulty_level, g.is_available
FROM games g
WHERE g.id = 1;

-- 3. Update Member Balance
UPDATE members
SET balance = 450.0, total_games_played = 1
WHERE id = 1;

-- 4. Insert Play History
INSERT INTO play_history (date_time, amount, game_id, member_id)
VALUES ('2024-01-15 10:30:00', 50.0, 1, 1);
```

Complex Join Query Example

Generated from Member Search:

```sql
-- Getting member with all related data
SELECT m.* FROM members m WHERE m.phone = '9876543210';

SELECT rh.* FROM recharge_history rh WHERE rh.member_id = 1;

SELECT g.* FROM games g WHERE g.is_available = true;

SELECT ph.id, ph.date_time, ph.amount, g.name as game_name, g.category as game_category
FROM play_history ph
JOIN games g ON ph.game_id = g.id
WHERE ph.member_id = 1
ORDER BY ph.date_time DESC;
```

Configuration

JPA Configuration

`application.properties`:

```
# Database Connection
spring.datasource.url=jdbc:mysql://localhost:3306/gaming_club
spring.datasource.username=root
spring.datasource.password=password

# JPA Properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.properties.hibernate.format_sql=true

# Logging SQL (for debugging)
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

SQL Logging Output Example

When `show-sql=true`, you'll see:

```
Hibernate:
    insert
    into
        members
        (balance, membership_type, name, phone, total_games_played)
    values
        (?, ?, ?, ?, ?)
```

Performance Considerations

1. N+1 Query Problem: Eager loading of relationships can cause multiple queries
2. Lazy Loading: Use `@OneToMany(fetch = FetchType.LAZY)` to avoid unnecessary joins
3. Batch Operations: Use `@BatchSize` for optimizing collection loading
4. Query Optimization: Use `@Query` with JOIN FETCH for complex relationships

Best Practices

1. Use Repository Pattern: All database operations through repositories
2. Transactional Boundaries: Use `@Transactional` for write operations
3. Lazy Loading: Prefer lazy loading for relationships
4. Indexed Fields: Ensure frequently searched fields (phone, email) are indexed
5. Query Optimization: Use specific `@Query` annotations for complex queries

This architecture ensures that all database operations are type-safe, maintainable, and follow Spring Data JPA best practices while automatically generating optimized SQL for MySQL database.