CS4750 Project - Cooper Bebeau (xqt5sy) & Matthew Caputo (ykk9fx)

Final Deliverable Report

Part 1: Database Design

Assumptions:

Team belongs to exactly one League.

Game belongs to exactly one League.

Each Game has exactly one home_team and one away_team; home_team ≠ away_team; both teams are in the Game's League.

GameRound is identified by (game_id, round_number); rounds may not exist yet for a Game.

WinnerTeam is optional and, if present, is one of that Game's two teams.

RoundScore exists only for teams in that Game; at most one score per (game_id, round_number, team_id).

Every RoundEvent occurs in exactly one GameRound; it has exactly one actor_player and one actor_team; opponent_* are optional.

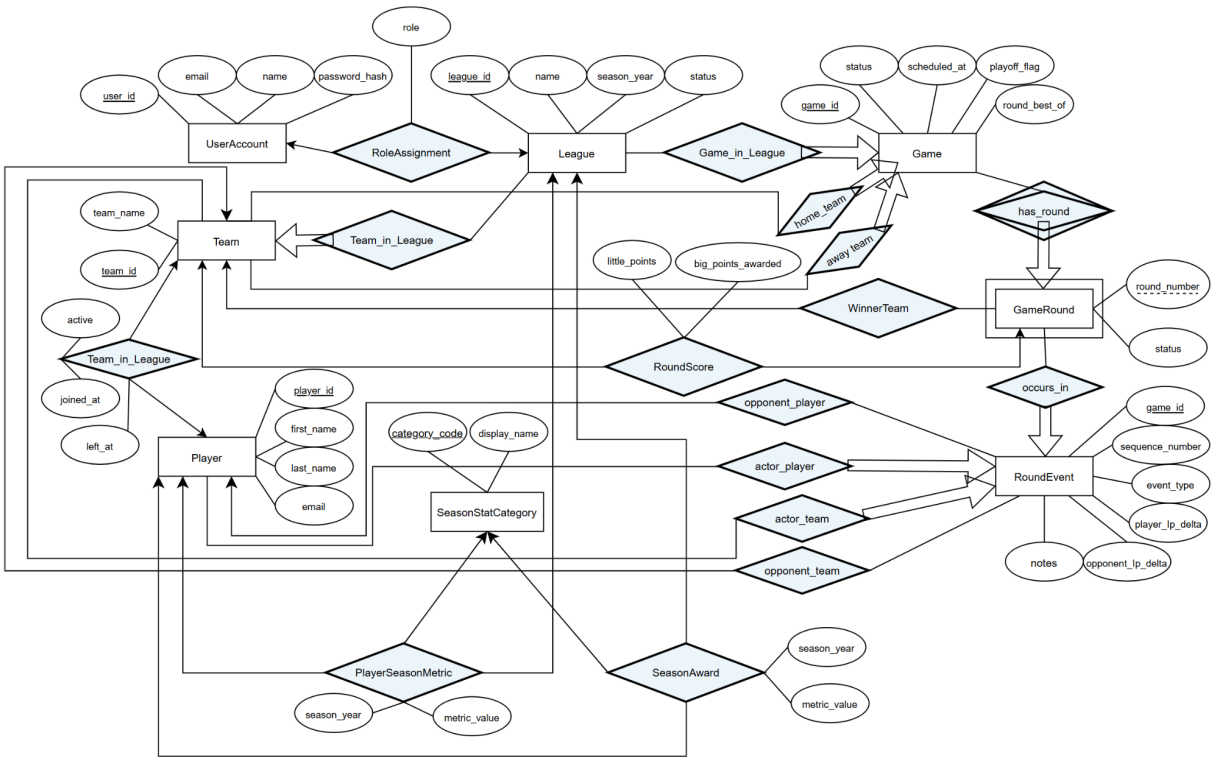
TeamMembership stores one row per (team_id, player_id); joined_at ≤ left_at when set.

PlayerSeasonMetric fact is per (league_id, season_year, player_id, category_code).

SeasonAward has exactly one winner per (league_id, category_code, season_year).

SeasonStatCategory.category_code is global.

League(name, season_year) is unique; Player.email and UserAccount.email are unique within their tables.



Schema Statements:
UserAccount(user_id, name, email, password_hash)
League(league_id, name, season_year, status)
RoleAssignment (user_id, league_id, role)
Team(team_id, league_id, team_name)
Player(player_id, first_name, last_name, email)
TeamMembership(team_id, player_id, active, joined_at, left_at)
Game(game_id, league_id, status, scheduled_at, home_team, away_team, playoff_flag, round_best_of)
GameRound(game_id, round_number, status, winner_team_id)
RoundScore(game_id, round_number, team_id, little_points, big_point_awarded)
RoundEvent(event_id, game_id, round_number, sequence_number, player_team_id, player_id, opponent_team_id, opponent_player_id, event_type, player_lp_delta, opponent_lp_delta, notes)
SeasonStatCategory(category_code, display_name)
PlayerSeasonMetric(league_id, season_year, player_id, category_code, metric_value)
SeasonAward(league_id, season_year, category_code, winner_player_id, metric_value)

BCNF
UserAccount(user_id, name, email, password_hash)
League(league_id, name, season_year, status)
RoleAssignment (user_id, league_id, role)
Team(team_id, league_id, team_name)
Player(player_id, first_name, last_name, email)
TeamMembership(team_id, player_id, active, joined_at, left_at)
Game(game_id, status, scheduled_at, home_team, away_team, playoff_flag, round_best_of)
GameRound(game_id, round_number, status, winner_team_id)
RoundScore(game_id, round_number, team_id, little_points, big_point_awarded)
RoundEvent(event_id, game_id, round_number, sequence_number, player_team_id, player_id, opponent_team_id, opponent_player_id, event_type, player_lp_delta, opponent_lp_delta, notes)
SeasonStatCategory(category_code, display_name)
PlayerSeasonMetric(league_id, season_year, player_id, category_code, metric_value)
SeasonAward(league_id, season_year, category_code, winner_player_id, metric_value)

Functional Dependencies:
{user_id} -> {name, email, password_hash}
{email} -> {user_id, name, password_hash}
{league_id} -> {name, season_year, status}
{name, season_year} -> {league_id, status}
{team_id} -> {league_id, team_name}
{league_id, team_name} -> {team_id}
{player_id} -> {first_name, last_name, email}
{email} -> {player_id, first_name, last_name}
{team_id, player_id} -> {active, joined_at, left_at}
{game_id} -> {status, scheduled_at, home_team, away_team, playoff_flag, round_best_of}

{game_id, round_number} -> {status, winner_team_id}
{game_id, round_number, team_id} -> {little_points, big_point_awarded}
{event_id} -> {game_id, round_number, sequence_number, player_team_id, player_id, opponent_team_id, opponent_player_id, event_type, player_lp_delta, opponent_lp_delta, notes}
{game_id, round_number, sequence_number} -> {event_id, player_team_id, player_id, opponent_team_id, opponent_player_id, event_type, player_lp_delta, opponent_lp_delta, notes}
{category_code} -> {display_name}
{league_id, season_year, player_id, category_code} -> {metric_value}
{league_id, season_year, category_code} -> {winner_player_id, metric_value}

Part 2: Database Programming

1. Database Hosting: The database is hosted locally on localhost (127.0.0.1) on user ykk9fx's local machine using the MySQL server provided by XAMPP. We chose a local environment for development speed and to avoid cloud latency during testing.

2. Application Hosting: The application is hosted locally using the Apache HTTP Server and PHP environment provided by XAMPP. It runs on the local web server and connects to the MySQL database to process requests.

3. Deployment Instructions

1.) Open XAMPP Control Panel and start Apache and MySQL (make sure it's set to port 3306)
2.) Next, open the shell in XAMPP and cd into the app directory example:
    cd /d "C:\Users\cjbeb\Downloads\die_league_app"
3.) If the database has not been created, run:
    python create_db.py
4.) Once it's been created, or if it already was, run the following to start the app
    python app.py
5.) Then open http://127.0.0.1:5000/

4. Advanced SQL Implementation
We utilized two major advanced SQL features to manage data integrity and business logic:

- Stored Procedure (sp_TallyAndFinalizeGame):

Description: This procedure encapsulates the complex logic of finalizing a game. It calculates season statistics for every player based on the raw round_event logs (aggregating hits, plinks, drops, etc.) and updates the player_season_metric table.
App Integration: This is triggered when the Commissioner clicks "Finalize Game" in the UI. The Python app calls cursor.callproc('sp_TallyAndFinalizeGame', (game_id,)). This offloads heavy calculation from the Python layer to the database layer, ensuring atomic updates.

- Trigger (trg_Game_SameLeague):

Description: A BEFORE INSERT trigger on the Game table checks that the home_team and away_team both belong to the same league as the game being scheduled.
App Integration: This runs automatically whenever a new game is created. If the app tries to

schedule a game between mismatched teams, the database raises an error (Signal SQLState '45000'), preventing invalid data entry.

Part 3: Database Security (Database Level)
Access Control Setup:
I established database security by implementing a dedicated service account for the application interface rather than utilizing the root administrator credentials. This security is configured for the application and developer environment, not for individual end users of the website.

This configuration enforces a strict separation of duties where the application is granted only the permissions necessary for its daily operations. The security model focuses on preventing catastrophic data loss by restricting the application user to data manipulation operations. Specifically, the account allows for selecting, inserting, and updating records but explicitly denies data definition privileges. This ensures that the application cannot structurally modify the database or delete tables, effectively neutralizing the risk of accidental or malicious schema destruction during runtime.

The following SQL commands are executed automatically by the application startup script to provision this secure user and enforce the privilege limits:

```
-- 1. Create a dedicated user for the application context
CREATE USER IF NOT EXISTS 'league_app'@'localhost' IDENTIFIED BY 'secure_pass_123';

-- 2. Grant standard data manipulation permissions (select, insert, update, delete)
GRANT SELECT, INSERT, UPDATE, DELETE ON die_league_db.* TO
'league_app'@'localhost';

-- 3. Note: DROP and ALTER permissions are omitted to protect the schema structure

-- 4. Grant execution rights for the required scoring logic
GRANT EXECUTE ON PROCEDURE die_league_db.sp_TallyAndFinalizeGame TO
'league_app'@'localhost';

-- 5. Apply the privilege changes immediately
FLUSH PRIVILEGES;
```

Part 4: Database Security (Application Level)

At the application level, we implemented three key security measures:

1. Password Hashing: We do not store plain-text passwords. We use Flask-Bcrypt to hash passwords during registration and verify hashes during login. If the database is compromised, user passwords remain secure.

Code:

```
# 1. Hashing the Password
password_hash = bcrypt.generate_password_hash(password).decode('utf-8')
```

2. Session-Based Authentication: We use a Python decorator (@login_required) to protect sensitive routes. This ensures that anonymous users cannot access internal API endpoints (like api/games or api/league).

Code:

```
def login_required(f):
    """Decorator to check if a user is logged into the session."""
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'logged_in' not in session or not session.get('logged_in'):
            return jsonify({"error": "Authorization required"}), 401
        return f(*args, **kwargs)
    return decorated_function
```

3. Parameterized Queries: We use mysql.connector with prepared statements (%s placeholders) for all database queries. This prevents SQL Injection attacks by separating SQL code from user input.

Code:

```
    try:
        # FIX: RoleAssignment -> role_assignment
        query = "SELECT COUNT(*) FROM role_assignment WHERE user_id = %s AND league_id = %s AND role = %s"
        cursor.execute(query, (user_id, league_id, required_role))
        if cursor.fetchone()[0] > 0:
            is_authorized = True
```