Game Leaderboard Application Technical Documentation

Table of Contents:

1. Project Overview

The **Game Leaderboard Application** is designed to track and rank the performance of players across different game sessions. By displaying top players, their scores, and ranks, it fosters a competitive environment that encourages player engagement and improvement. This application is ideal for online games, tournaments, or gamified platforms where users can compare their performance with others.

Core Features:

- Track player scores for each game session.
- Rank players based on their performance.

- Display top players and their scores.
- Allow users to submit scores and view their own rank.
- Real-time updates and efficient querying (using Redis for caching).

The backend is developed using **Django (Python)**, while the frontend is built with **React (JavaScript)**. **Redis** is used to cache leaderboard data and improve performance.

2. Technologies Used

- **Backend**: Django, Django REST Framework
- **Frontend**: React
- **Database**: SQLite (development), Redis (for caching leaderboard data)
- **Caching**: Redis (in-memory data store for fast operations)
- **Monitoring**: New Reli

3. Setup Instructions

3.1 Backend Setup (Django)

To set up the backend for the Game Leaderboard Application, follow the steps below:

1. **Navigate to the backend directory**:

```
cd leaderboard
```

2. **(Optional) Create and activate a virtual environment**:

```
python3 -m venv venv
source venv/bin/activate
```

3. **Install Python dependencies**:

```
pip install -r requirements.txt
```

4. **Set up environment variables**:

   Ensure your `settings.py` is configured for Redis. Example:

   ```
   LEADERBOARD_ZSET_URL = "redis://localhost:6379/0"
   ```

5. **Apply database migrations**:

   ```
   python manage.py migrate
   ```

6. **(Optional) Seed the database with test data**:

   ```
   python manage.py populatedb
   ```

7. **(Optional) Sync leaderboard data to Redis**:

   ```
   python manage.py sync_leaderboard_to_redis
   ```

8. **Run the backend server**:

   ```
   python manage.py runserver:8000
   ```

3.2 Frontend Setup (React)

To set up the frontend for the Game Leaderboard Application, follow these steps:

1. **Navigate to the frontend directory**:

   ```
   cd ../frontend
   ```

2. **Install Node.js dependencies**:

   ```
   npm install
   ```

3. **Start the frontend development server**:

```
npm start
```

The app will be available at `http://localhost:3000`.

3.3 Redis Setup

1. **Install Redis** (if not already installed):
   - On macOS (using Homebrew):

     ```
     brew install redis
     ```

   - On Ubuntu/Debian:

     ```
     sudo apt-get update
     sudo apt-get install redis-server
     ```

2. **Start Redis server**:

   ```
   redis-server
   ```

   Redis will run by default on `localhost:6379`.

4. API Overview

The backend exposes several RESTful APIs that allow users to interact with the leaderboard system. The core functionalities are:

4.1 Submit Score

- **Endpoint**: `/api/v1/leaderboard/submit/`
- **Method**: POST
- **Description**: Submits a player's score for a game session.
- **Request Body**:

```
{
  "user_id": "integer",
  "score": "integer"
  "game_mode": "string"
}
```

- **Response**:
  - `201 Created` on success.
  - Returns the updated rank and score for the user.

4.2 Get Top Leaderboard

- **Endpoint**: `/api/v1/leaderboard/top/`
- **Method**: GET
- **Description**: Retrieves the list of top players and their scores.
- **Response**:
  - `200 OK` with a list of top players, their scores, and ranks.

4.3 Get User Rank

- **Endpoint**: `/api/v1/leaderboard/rank/{user_id}`
- **Method**: GET
- **Description**: Fetches the current rank and score of a specific user.
- **Query Parameters**:
  - `user_id`: The ID of the user whose rank is to be fetched.
- **Response**:
  - `200 OK` with the user's rank and score.

5. Data Models

5.1 User

- **id**: Unique identifier for the user.
- **username**: Name of the user.
- **join_date:** date the user has created the account

5.2 GameSession

- **id**: Unique identifier for the session.
- **user**: Reference to the User.
- **score**: Score achieved in the session.
- **timestamp**: When the session was played.
- **game_mode**: solo or team game

5.3 Leaderboard

- **id**: Unique identifier for the session.
- **user**: Reference to the User.
- **total_score**: Total score achieved in all games
- **rank**: Current in game rank

6. Use of Redis in the Game Leaderboard Application

How Redis is Used

1. **Leaderboard Storage**:
    - The leaderboard is stored in a Redis **sorted set** called `leaderboard:zset`. Each user's total score is stored as a member in this set, with the score as the value.
2. **Score Submission**:
    - When a user submits a score, the backend:
        1. Updates the user's score in the database.
        2. Updates the score in Redis.

    3. Retrieves the updated rank and score from Redis for immediate feedback.

3. **Fetching Top Players**:
    - The top N players are fetched directly from Redis using commands like `zrevrange`, which is much faster than querying and sorting a database.
4. **Fetching User Rank**:
    - A user's rank and score are instantly available by querying Redis with commands such as `zrevrank`.
5. **Synchronization**:
    - A management command (`sync_leaderboard_to_redis`) is used to sync the database leaderboard with Redis after bulk updates or migrations.

Example Redis Operations:

- `zadd("leaderboard:zset", {user_id: score})`: Adds or updates a user's score.
- `zrevrange("leaderboard:zset", 0, N-1, withscores=True)`: Retrieves the top N players.
- `zscore("leaderboard:zset", user_id)`: Gets a user's score.
- `zrevrank("leaderboard:zset", user_id)`: Gets a user's rank.

7. Low-Level Design (LLD) Implementation

Core Components:

- **Models**: Define Django models for User, GameSession, and Leaderboard.
- **Services**: Service modules like `submit_score_service.py`, `ranking_service.py`, and `top_player_service.py` to encapsulate business logic.
- **Views**: Implement API views using Django REST Framework to expose endpoints for submitting scores, fetching top players, and retrieving user ranks.
- **Redis Integration**: Implement a caching layer using Redis sorted sets for efficient leaderboard operations.

Data Flow:

1. **Score Submission Flow**:
    - User submits a score via the API.
    - The view calls the service layer to process the score.
    - The service updates both the database and Redis.
    - The updated rank and score are returned to the user.
2. **Leaderboard Retrieval Flow**:
    - API view requests top N players from the service.
    - The service fetches data from Redis for performance.
    - The user details are joined from the database as necessary.
    - Results are serialized and returned.

Class and Function Design:

- **Models**:
    - **User**: Contains the user information such as ID, username, etc.
    - **GameSession**: Contains data about game sessions, including scores, timestamps, and the game mode.
    - **Leaderboard**: Tracks scores and ranks over time, possibly holding aggregate or cumulative scores.
- **Services**:
    - **submit_score(user_id, score, game_mode)**: Handles the business logic of score submission. It updates both the database (GameSession and Leaderboard models) and Redis.
    - **get_top_n_players(n)**: Retrieves the top N players. Primarily fetched from Redis (sorted sets for efficiency), and additional player details may be joined from the database.
    - **get_user_rank(user_id)**: Retrieves the rank and score of a specific user. Redis can provide the rank, and the database will hold the actual user

score.

- **Views**:
  - **SubmitScoreAPI**: A Django REST framework API view that handles `POST` requests to submit scores. This triggers the score submission logic in the service layer.
  - **TopLeaderboardAPI**: A `GET` API endpoint that returns the top players. It calls the `get_top_n_players` service to fetch leaderboard data.
  - **UserRankAPI**: A `GET` API endpoint that retrieves the rank of a specific user. It calls the `get_user_rank` service to fetch the user's rank and score.

8. Error Handling and Validation

Implement robust error handling using decorators and exception handling to provide consistent API responses for errors such as invalid input, missing users, or server issues. Error messages should be clear and guide users toward corrective actions.

9. Testing and Quality Assurance

- **Unit Tests**: Write unit tests for each service and view to ensure correctness and reliability.
- **Integration Tests**: Test the integration of Redis with the database to ensure data consistency.
- **Load Testing**: Conduct load testing to ensure the system can handle high user traffic and score submissions.