Leaderboard Service

Problem statement for the machine coding interview round

Overview

Our users are playing casual games on our app for engagement. These casual games have short 2-3 minute sessions and end with a score for the user based on how well they played. A score is always an integer between 0 and 1 billion. We want to launch a new feature to allow users to compare their scores on the app and reward the highest scoring user every day.

Design and develop a leaderboard management service that manages daily leaderboards for several games at once. Leaderboard management involves - leaderboard campaign configuration, score submissions, rank list (full and partial), result computation and so on. Your design should factor in concurrent score submissions.

Note: There are no classes / code expected for managing gameplay. The code that is expected should be around Leaderboard Service and how it processes the score submissions.

Mandatory Functionality

1. **Support for multiple games**. Service should be able to host leaderboards for several games at the same time. We don't want to deploy separate instances for each game.
2. **Each game may have several active leaderboards configured**. Eg. Daily leaderboard, Weekly leaderboard etc. Each leaderboard configuration has a specific start-time and a specific end-time.
3. **Users submit scores for some game, not for a specific leaderboard configured within the game**. This means that the score should automatically be considered for all the active leaderboard configurations that are active for the game at the time of score submission.
4. **A user cannot exist more than one time in a leaderboard**. New score submissions should only be considered if they are higher than the current score the user had submitted in the leaderboard. In simpler terms - only consider the top score the user has scored during the active time period of the leaderboard.
5. **There should be a feature to fetch the top X scorers for a leaderboard**. But to ensure less skilled players can compare themselves against other players as well, provide an API to fetch **X scorers around a user U** that returns X users above the user U and X users below the user U.
6. One should be able to **access older leaderboards** once they become inactive. This is crucial to ensure rewards can be disbursed post completion of the campaign.

Functions to help you think

1. Consider the right data structure for this problem. Leaderboards are an idiomatic use case for self-sorted data structures. **You don't need to implement your solution in the most efficient possible way. But do think about it**. You can choose to do it any way you like, but consider the time complexity of various operations. Aim for accuracy and completeness rather than efficiency for the purpose of the interview.
2. **getSupportedGames()**: This function can be used by the client to get a list of supported GAME IDs.
3. **createLeaderboard(String gameID, int startEpochSeconds, int endEpochSeconds)**: Method used to create a leaderboard campaign for a given game that will be active from the given start time to the given end time. This would return a LeaderboardID.
4. **getLeaderboard(String leaderbordID)**: Returns the leaderboard with a list of **< user IDs, score >**, with the right ranking among the players - high score to low score.
5. **submitScore(String gameID, String userID, int score)**: Submit a score to the game. This should iterate through all the leaderboards configured for the game and check if the leaderboard is active or not (based on start / end time). The score should be entered for the user on all active leaderboards. Note that a lower score should never replace a higher score.
6. **listPlayersNext(String gameID, String leaderboardID, String userID, int nPlayers), listPlayersPrev(String gameID, String leaderboardID, String userID, int nPlayers)**: Method to fetch the next N players and the previous N players to the given user.

Evaluation Criteria

- The code should be working and cover all the mandatory functionalities.
- Code readability, testability, and maintainability.
- Proper separation of concerns using Object-Oriented concepts.
- Proper Algorithm and Data Structures choices.
- SOLID principles should be followed.
- Proper error handling and validations.
- The system should be scalable and extensible for future changes and updates.
- You can use in-memory storage, but keep your data access separate from BLL.
- Feel free to make relevant assumptions wherever required. Please list these assumptions.
- Efficient use of resources, such as memory and processing power.