Promofy Leaderboard Service Technical Design

Overview

This document describes the architecture and components of the Promofy Leaderboard Service.

Components

The service consumes Kafka events, processes them, and updates leaderboards. Users can query leaderboards and player positions via an API. Components include:

- **Fake Data Generator** for mock player activity.
- **Kafka Cluster** with partitions and replicas for high availability.
- **Leaderboard Service** to consume events and update leaderboards.
- **PostgreSQL** for persistent leaderboard and player data.
- **Redis Cluster** for caching frequently accessed leaderboard data.

PostgreSQL Schema Options

Option A Single table for all leaderboards. Simple to query and implement, but multiple score updates require overwriting existing scores.

LEADERBOARD_ID	PLAYER_ID	SCORE
1203d9838	101	1500
1	102	1450
2	202	1950

Option B Requires separate tables (`leaderboard`, `leaderboard_delta`) with an n-to-m relationship. Supports tracking score changes over time but requires joins and aggregations for current scores.

ID	TYPE	START_DATE	END_DATE
1	DAILY	2025-09-28 02:00:00	2025-09-29 01:59:59
2	WEEKLY	2025-03-01 02:00:00	2025-03-08 01:59:59

ID	PLAYER_ID	SCORE_DELTA	CREATED_AT
1	101	+20	2025-09-01 04:00:00
3	201	+30	2025-09-03 04:00:00

I ended up choosing Option B for its flexibility in tracking score changes over time, despite the added complexity in querying current scores. Most importantly this allows us to filter scores by time ranges, which is a core requirement.

Possible Performance Optimizations: We could also implement both options and query option A for non-time-sensitive leaderboards and option B for time-sensitive ones. This would increase the storage requirements but improve performance for non-time-sensitive leaderboards.

Caching Strategy

For caching, Redis sorted sets (ZSETs) will be used rather than NestJS in-memory cache to reliably handle frequent leaderboard updates. Upon requesting a leaderboard, the service will first check Redis. If the data is not present, it will query PostgreSQL, update Redis, and return the data.

From then the leaderboard which is cached in Redis will be updated with each new score event. For this a leaderboardSync service keeps track of active and cached leaderboards and ensures they are updated in Redis.

There is no warming strategy as leaderboards are only cached when requested. But it is possible to pre-cache currently active leaderboard. But that would improve performance only for the first request.

Event Processing

The service will consume events from Kafka using a consumer group to allow horizontal scaling. Each event will be processed to update the corresponding player's score in PostgreSQL and Redis.

In case of failures / exceptions, the Leaderboard Service will not auto-commit the Kafka offset, allowing for reprocessing of the event. To avoid duplicate processing, an unique constraint on `(player_id, created_at)` in the `leaderboard_delta` table will be used. `created_at` has a precision of milliseconds and was sent from the producer on event creation.

The Events are expected to specify the playerID as the key, allowing Kafka to route all events for a player to the same partition. With this, we can ensure that the order of events for a player is preserved, which is crucial for accurate score updates.

Failure Handling and Recovery

If the Redis cache is lost, the leaderboardSync service will clear the cached leaderboardIDs list and wait for the connection to be re-established. This is not a critical failure as the service can still function by querying PostgreSQL directly.

If the Database is lost, the service will throw exceptions and not commit Kafka offsets. Retry attempts are configured in the TypeORM connection settings.

A lot of the weight of failure handling is offloaded to Kafka.

CAP theorem trade-offs

The Promofy Leaderboard Service prioritizes availability and partition tolerance over strict consistency. In the event of a network partition, the service will continue to accept and process events, ensuring high availability. However, this may lead to temporary inconsistencies in leaderboard data until all events are processed and the cache is updated.

This trade-off is acceptable for the leaderboard use case, where eventual consistency is sufficient for user experience. We could, for example, wait for a bit and then query yesterdays leaderboard for a more consistent view.

To mitigate potential issues with consistency, the service employs strategies such as idempotent event processing and unique constraints in the database schema. These measures help ensure that even in the face of network partitions or failures, the integrity of player scores is maintained over time.

Scalability Analysis & Performance Optimization

The service is designed to scale horizontally. Multiple instances of the Leaderboard Service can run in parallel, each consuming from the same Kafka topic using a consumer group. This allows for load balancing and fault tolerance.

PostgreSQL can be scaled by using a managed service or by deploying a configurable PostgreSQL helm chart. Same goes for Redis, and of course Kafka.

In the essence of time-saving I wasted some time setting up a production environment with Docker Compose. But in a real-world scenario, I would use Helm but I thought I'd try something different. But the docker-compose.prod.yaml ended up being too tedious and i gave up. In a real-world scenario I would use Helm charts for all services.

Scaling to 1M events/minute Scaling to 1M events/minute will require careful consideration of the architecture and potential bottlenecks. Key areas to focus on include:

- **Kafka Throughput:** Ensure the Kafka cluster is configured to handle high throughput with sufficient partitions and replicas.
- **Database Performance:** Optimize PostgreSQL with indexing, connection pooling, and possibly read replicas.
- **Cache Efficiency:** Utilize Redis effectively by implementing proper eviction policies and monitoring cache hit rates.
- **Stream Processing:** Consider using stream processing frameworks like Apache Flink to decouple event processing from the main service.

Performance Optimization: To optimize performance, consider:

- **Batch Processing:** Process events in batches to reduce the number of database writes.
- **Load Testing:** Regularly perform load testing to identify and address performance bottlenecks.

Monitoring and Logging

Monitoring and logging are crucial for maintaining the health and performance of the Promofy Leaderboard Service. Key aspects include:

- **Metrics Collection:** Use tools like Prometheus and Grafana to collect and visualize metrics related to service performance, such as request latency, error rates, event distribution, and cache hit/miss ratios.
- **Centralized Logging:** Store logs locally and collect them using a centralized logging solution.