Real-Time Leaderboard Microservice Design & Implementation

Stack: NestJS (TypeScript), Event Streaming (Kafka), Cache Layer, Persistent Storage

Context

This microservice is a critical component of a promotional rewards system. The leaderboard rankings directly determine reward distribution to players, making data accuracy and consistency paramount. Rewards are calculated and distributed based on leaderboard partitions (e.g., top 10%, top 100 players, tier-based rewards), so the system must guarantee reliable and auditable ranking calculations.

Objective

Design and implement a production-ready microservice that maintains real-time leaderboards with high-performance reads and consistent data integrity. The service should demonstrate understanding of distributed systems patterns, event-driven architecture, and scalability considerations.

Core Requirements

1. System Design

Design a microservice that:

- Processes high-volume score events asynchronously
- Maintains multiple concurrent leaderboard windows
- Serves sub-100ms read latency at scale
- Guarantees data consistency across cache and persistent layers
- Supports horizontal scaling

2. Leaderboard Windows

Implement flexible time-based aggregations:

- **Daily** - UTC-based rotation
- **Weekly** - Document your week boundary definition
- **All-Time** - Cumulative scores
- **Custom Period** - Dynamic date ranges

3. Event Processing

Design the event consumption layer with:

- Idempotent event processing (define your deduplication strategy)
- At-least-once delivery guarantee handling
- Event ordering considerations for tie-breaking
- Error handling and dead letter queue patterns
- Backpressure management

4. Data Architecture

Design and implement:

- **Cache Strategy**: Define invalidation policies, TTL, and warming strategies
- **Persistence Layer**: Choose and justify your database selection
- **Data Consistency**: Explain your approach to eventual consistency
- **Recovery Mechanism**: Design cache rebuild from source of truth
- **Partitioning Strategy**: Document how you'd partition data for scale

5. API Design

Create RESTful endpoints following microservice best practices:

- Leaderboard queries with pagination and filtering
- Player ranking lookup with surrounding context
- Health checks and observability endpoints

Required endpoints:

- `GET /leaderboards/:id/rankings` - Paginated rankings with flexible windowing
- `GET /leaderboards/:id/players/:userId` - Player position and context
- `GET /leaderboards/:id/export` - Streaming data export
- `GET /health` - Service health with dependency status

6. Infrastructure & Deployment

Provide containerized setup with:

- Multi-stage Dockerfile optimized for production
- Docker Compose for local development environment
- Environment-based configuration management
- Graceful shutdown handling
- Resource limits and health probes

Deliverables

1. **Source Code**
   - Clean, production-quality NestJS application
   - Comprehensive error handling

    - Unit tests for critical business logic
    - Integration test examples
2. **Architecture Documentation**
    - System architecture diagram showing all components
    - Data flow diagram from event ingestion to API response
    - Sequence diagrams for critical flows (event processing, cache miss scenario)
3. **Technical Design Document** (2-3 pages)
    - **Design Decisions**: Database choice, caching strategy, event processing patterns
    - **Scalability Analysis**: How would you scale to 1M events/minute?
    - **Consistency Model**: CAP theorem trade-offs in your design
    - **Failure Scenarios**: How does the system handle cache failure, DB outage, Kafka downtime?
    - **Performance Optimizations**: Query optimization, indexing strategy, caching patterns
    - **Monitoring Strategy**: Key metrics and alerts you would implement
4. **README**
    - Quick start guide
    - API documentation
    - Configuration options
    - Development workflow

Evaluation Criteria

- **System Design** (30%): Architecture decisions, scalability considerations, trade-offs
- **Code Quality** (25%): Clean code, SOLID principles, error handling, testability
- **Performance** (20%): Efficient queries, proper caching, resource optimization
- **Reliability** (15%): Idempotency, consistency, failure handling
- **Documentation** (10%): Clear explanations, diagrams, setup instructions