**System Architecture Document**

This document outlines the high-level architecture, data modelling, caching strategy, assumptions & trade-offs, and monitoring & observability for the system.

**1. High-Level Architecture**

The architecture is designed as a stateless microservice to ensure scalability and resilience. The core principle is to optimize heavily for read operations.

* 1. **Core Components**

1. **Clients (Mobile/Web):** End-user applications that consume the Menu Service.
2. **API Gateway:** This is the single-entry point for all clients. Its responsibilities include:

* **Request Routing**: Directs traffic to the appropriate service (e.g., /api/v1/menus/\* to the Menu Service).
* **Authentication & Authorization**: Verifies credentials (e.g., JWT tokens) for write operations (POST, PUT, DELETE), ensuring only restaurant partners can modify their data.
* **Rate Limiting**: Protects the service from abusive traffic and DoS attacks.

1. **Load Balancer:** Distributes incoming traffic evenly across the multiple stateless instances of the Menu Service. This allows for horizontal scaling by simply adding more instances.
2. **Menu Service:** The core application responsible for all menu-related business logic. It's built to be stateless, meaning it doesn't store any session data locally. All state is externalized to the cache and database. This is also consists of a local in-memory L1 cache. The L1 cache will be a crucial component to meet the <100ms p99 response time requirement.
3. **Distributed Cache (Redis):** This will be L2 cache. A crucial component to meet the <100ms p99 response time requirement. We use a distributed cache so that all service instances share the same cache data.
4. **Database (MySQL):** This forms the persistent storage layer. Acts as the source of truth. chosen for its reliability and wide adoption.
   1. **Architectural Flow Diagram**

**A close up of a document

AI-generated content may be incorrect.**

1. **Data Modelling**

MySQL database will be used to store data

**2.1. Tables and Fields**

**1. Restaurant Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Column** | |  | | --- | |  |  |  | | --- | | **Data Type** | | **Constraints** | **Description** |
| RestaurantId | |  | | --- | |  |  |  | | --- | | BINARY(16) | | |  | | --- | |  |  |  | | --- | | Primary Key (PK) | | |  | | --- | |  |  |  | | --- | | UUID stored in binary format | |
| Name | VARCHAR(100) | NOT NULL | Name of the restaurant |
| Address | VARCHAR(255) | NULLABLE (optional) | Physical address |

**2. MenuItems Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Data Type** | **Constraints** | **Description** |
| MenuId | BINARY(16) | Primary Key (PK) | UUID stored in binary format |
| RestaurantId | BINARY(16) | Foreign Key (FK), NOT NULL | References Restaurant (RestaurantId) |
| Name | VARCHAR(100) | NOT NULL | Name of the menu item |
| Price | DECIMAL(7,2) | NOT NULL | Price |
| Availability | BOOLEAN | NOT NULL, DEFAULT TRUE | Whether item is available |
| Veg | BOOLEAN | NOT NULL | Whether the item is vegetarian |

**2.2. Relationships & Constraints**

* **One-to-Many Relationship**  
  Enforced via the foreign key RestaurantId in the MenuItems table.
* **Foreign Key Constraint**  
  Ensures that a menu item must belong to a valid restaurant.

**2.3. Indexing Strategy**

* Primary keys are **auto-indexed**
* Add index on MenuItems.RestaurantId to speed up lookups for a restaurant's menu

**3. Caching Strategy**

**3.1 Cache Architecture**

**L1 Cache (Local In-Memory Cache)**

* Present on each individual service instance.
* Implemented using an in-memory caching library such as **Caffeine**.
* Provides ultra-fast access with no network overhead.
* Reduces load on shared cache and backend database.

**L2 Cache (Distributed Redis Cache)**

* Shared across all service instances.
* Implemented using **Redis**.
* Acts as a centralized backup cache for all instances.
* Ensures data consistency and availability across nodes.

**3.2 Cache Eviction Strategy**

Both **L1 and L2 caches** use a combination of:

* **Time-to-Live (TTL)**:

Automatically expires entries after a configured duration.

Ensures **freshness** of data and avoids serving **stale** content.

**L1 TTL is configured to be shorter than L2**, encouraging periodic refresh from L2 and ensuring closer alignment with the source of truth (DB).

* **Least Recently Used (LRU)**:
  + Evicts the least recently accessed entries when cache reaches memory limits.
  + Ensures efficient memory usage by retaining **hot (frequently accessed)** data in the cache.
  + Prevents memory bloat in both L1 and L2 caches.

**3.3 GET Menu Flow**

1. **Check L1 Cache**:
   * If menu data is present, return it directly (fastest path).
2. **Check L2 Cache**:
   * If not found in L1, query L2 Redis.
   * If found:
     + Update L1 with this data.
     + Return data to client.
3. **Query Database**:
   * If not found in L2, fetch from the database.
   * Update both:
     + L2 Redis cache.
     + L1 cache (on current instance).
   * Return data to client.

**3.4 UPDATE Menu Flow**

1. Update the menu in the **database** first.
2. Trigger an **asynchronous update** to the **L2 cache**.
3. L1 caches will be refreshed automatically on next read (due to shorter TTL or cache miss).

**3.5 DELETE Restaurant Flow**

1. Delete the restaurant and its menu items from the **database**.
2. Trigger an **asynchronous eviction** of the corresponding data from **L2 cache**.
3. L1 cache entries will expire naturally due to TTL or get invalidated on future reads.

**4. Assumptions & Trade-offs**

**4.1 Assumptions**

1. **Read-Heavy Access Pattern**
   * The system is optimized for frequent menu reads, which justifies the use of multi-level caching for low-latency access.
2. **Menu Data is Relatively Static**
   * It is assumed that menus change infrequently compared to read operations. This enables effective use of TTL-based caching and eventual consistency mechanisms.
3. **Eventual Cache Consistency is Acceptable**
   * A brief period of stale data is acceptable during asynchronous cache updates or TTL expiry delays.
4. **Service-Level Authentication & Authorization via API Gateway**
   * Only authenticated and authorized restaurant partners are allowed to modify menus.
   * Menu read operations are assumed to be public.
   * Authentication and authorization logic is offloaded to an API Gateway, allowing the Menu Service to remain focused on core functionality.
5. **Multiple Service Instances Behind Load Balancer**
   * The service is deployed as multiple instances behind a load balancer for horizontal scalability and high availability.
6. **Independent L1 Cache Per Instance**
   * Each service instance maintains its own local L1 cache. No cache coordination occurs between instances, simplifying the architecture.
7. **Highly Available Redis for L2 Cache**
   * Redis is assumed to be deployed in a highly available, distributed configuration, ensuring L2 cache resilience and scalability.
8. **Horizontally Scalable Database with Sharding**
   * The database layer supports horizontal scaling with sharding to handle high write/read throughput efficiently and maintain performance under heavy load.

**4.2. Trade-offs**

1. **Consistency vs. Availability in a Read-Heavy System**

* The system is optimized for high availability and low latency reads, which is critical given its read-heavy nature.
* As a result, we consciously trade away immediate consistency in favor of eventual consistency, especially during asynchronous cache updates.
* This ensures responsiveness under high load, even if stale data may occasionally be served for a short window.

1. **Shorter TTL in L1 Leads to Higher L2 Load**

* A shorter TTL in L1 ensures quicker cache refresh and better data freshness.
* However, it increases cache misses at L1 and results in more frequent reads from L2, adding pressure on Redis.

1. **TTL-Based Expiry Over Explicit Invalidation**

* Relying on TTL simplifies the system and avoids the complexity of tracking and invalidating each change.
* The trade-off is the risk of stale data during the TTL duration, particularly if async cache updates fail.

1. **No Active L1 Invalidation Across Instances**

* L1 caches on each service instance are not actively invalidated when L2 is updated.
* This reduces system complexity and avoids inter-instance communication, but it sacrifices immediate global consistency.

1. **Increased Memory Usage Due to Multi-Layer Caching**

* Storing the same menu data in both L1 and L2 consumes more memory.
* This is accepted to achieve low-latency performance and scalability under load.

**5. Monitoring & Observability Ideas**

**5.1. Application Metrics**

**a. Cache Metrics**

* **L1 Cache**
  + Hit rate, miss rate, eviction count (Caffeine metrics)
  + TTL expiration count
* **L2 Cache (Redis)**
  + Hit/miss rate
  + Key eviction rate
  + Redis command latency (e.g., GET, SET, DEL)
  + Connection pool usage

**b. Database Metrics**

* Query latency (especially for GET menu)
* Slow query logs
* DB connection pool usage
* Shard utilization (if sharding is custom)

**c. API Metrics**

* Request count per endpoint
* Request latency (avg, p95, p99)
* Error rate per endpoint (4xx, 5xx)
* Throughput (requests/sec)
* Payload size distribution

**5.2. Alerts**

We can set up alerts for the following key indicators to proactively detect and respond to issues:

* **L1 and L2 Cache Hit Rate**
  + Alert when cache hit rate drops below an acceptable threshold (e.g., 80%)
* **L2 Redis Latency**
  + Alert if Redis response latency exceeds expected limits (e.g., > 100ms)
* **HTTP 5xx Error Rate**
  + Alert when the rate of server errors increases beyond a safe threshold
* **Request Latency (p99)**
  + Alert if **p99 latency for read APIs exceeds 100ms**, violating non-functional performance goals