Low-Level Design (LLD) Document for Weather Application

1. Overview

This document provides a detailed low-level design (LLD) for the weather application, describing its components, data models, and interactions. The system follows a microservices-based architecture with REST APIs, load balancing, caching, and real-time data processing capabilities.

2. Component Breakdown

2.1 User

- Attributes:
  - `id: int`
  - `name: string`
  - `email: string`
- Methods:
  - `registerUser(): void`
  - `loginUser(): void`
  - `updateProfile(): void`

2.2 Content Delivery Network (CDN)

- Attributes:
  - `imageId: int`
  - `fileId: int`
- Methods:
  - `getImageById(): void`
  - `getFile(): void`

2.3 Gateway Service

- Attributes:
  - `gatewayId: int`
  - `requestId: int`
  - `status: string (SUCCESS/FAILED)`
- Methods:
  - `forwardRequest(request: APIRequest): APIResponse`
  - `authorizeRequest(token: string): boolean`
  - `routeToService(serviceName: string, request: APIRequest): void`
  - `logRequest(request: APIRequest): void`

2.4 Load Balancer

- Attributes:
  - `balancerId: int`

    - `strategy: string (ROUND_ROBIN/LEAST_CONNECTION)`
    - `healthCheckInterval: int`
- Methods:
    - `distributeRequest(request: APIRequest): void`
    - `checkInstanceHealth(): void`
    - `scaleUp(): void`
    - `scaleDown(): void`

2.5 Service Manager

- Attributes:
    - `serviceId: int`
- Methods:
    - `checkServiceStatus(): void`
    - `manageServices(): void`

2.6 Rate Limiter

- Attributes:
    - `ruleId: int`
    - `apiEndpoint: string`
    - `maxRequestsPerMinute: int`
    - `status: string (ACTIVE/BLOCKED)`
- Methods:
    - `checkLimit(apiEndpoint: string, userId: int): boolean`
    - `enforceLimit(apiEndpoint: string): void`
    - `resetLimit(): void`

2.7 Location Service

- Attributes:
    - `place: string`
- Methods:
    - `currentLocation(): string`

2.8 Temperature Service

- Attributes:
    - `temperature: int`
- Methods:
    - `currentTemperature(location: string): int`

2.9 Humidity Service

- Attributes:
    - `humidity: float`
- Methods:
    - `currentHumidity(): float`

2.10 Wind Service

- Attributes:
    - `windSpeed: int`

    - windDirection: string
- Methods:
    - windSpeedByLocation(location: string): int
    - windDirectionByLocation(location: string): string

2.11 Profile Service

- Attributes:
    - userId: int
    - name: string
    - email: string
- Methods:
    - getProfile(userId: int): void
    - updateProfile(userId: int, data: userProfile): void

2.12 Notification Service

- Attributes:
    - notificationsProvider: string
- Methods:
    - sendNotification(): void

2.13 Message Queue

- Attributes:
    - queueId: int
    - queueName: string
    - maxSize: int
    - messageRetentionTime: int
    - consumerCount: int
- Methods:
    - enqueueMessage(message: Message): void
    - dequeueMessage(): void
    - deleteMessage(messageId: int): void
    - retryFailedMessage(): void

2.14 Monitoring Service

- Attributes:
    - metricId: int
    - serviceName: string
    - cpuUsage: float
    - memoryUsage: float
    - requestCount: int
    - errorRate: float
- Methods:
    - collectMetrics(serviceName: string): Metrics
    - analyzePerformance(): string
    - triggerAlert(serviceName: string): void

2.15 Logging Service

- Attributes:
    - logId: int
    - logLevel: string (INFO/ERROR/DEBUG)

    - `message: string`
    - `timestamp: datetime`
- Methods:
    - `storeLog(log): void`
    - `retrieveLogs(serviceName: string, level: string): List<Log>`
    - `analyzeError(serviceName: string): string`

3. Data Flow

User Request Flow

1. User sends a request to the **Gateway Service**.
2. The **Rate Limiter** validates the request limit.
3. The request is forwarded to the appropriate microservice.
4. The **Load Balancer** distributes the request among available instances.
5. The **Service Manager** manages the lifecycle and ensures availability.
6. The microservices fetch relevant data from databases.
7. The response is sent back to the user via the **Gateway Service**.

Data Processing Flow

1. **Location Service** determines the user's location.
2. **Temperature, Humidity, and Wind Services** fetch respective weather data.
3. The **Service Manager** coordinates service interactions.
4. The **Notification Service** sends alerts if necessary.
5. The **Monitoring Service** logs system metrics.
6. **Logging Service** records logs for debugging and analysis.

4. Conclusion

This LLD document provides an in-depth breakdown of components, data flow, and system interactions for the weather application. The design ensures high availability, scalability, and fault tolerance using a microservices architecture with a Kafka-based event-driven approach, caching via Redis, and monitoring with Prometheus & Grafana.