Locofy high level design RFC

1. Overview

This RFC outlines the high-level architecture, communication strategy, caching mechanisms, and logging standards for the **Event Booking Platform**. The platform enables global event searches, ticket booking, and payment processing while ensuring scalability, reliability, and observability.

---

2. High-Level Design

The system comprises multiple microservices orchestrated via Kubernetes and designed to communicate asynchronously using Kafka. Each service is modular, maintains its own database, and operates independently.

Components:

1. **API Gateway**: Central entry point for all external client requests.
2. **User Service**: Manages user authentication and profiles.
3. **Event Service**: Handles event management and ticket inventory.
4. **Search Service**: Provides advanced search functionality via Elasticsearch.
5. **Booking Service**: Manages ticket reservations and interacts with the Payment Service.
6. **Payment Service**: Processes payments using Stripe.
7. **Notification Service**: Sends user notifications via email, SMS, or push notifications.
8. **Monitoring Service**: Tracks system health, logs, and metrics.
9. **Kafka**: Facilitates event-based communication between services.
10. **Redis**: Caching for frequently accessed data.

---

3. How Services Communicate via Kafka

Services exchange events asynchronously via Kafka to ensure scalability and fault tolerance.

Event Workflow Example:

1. **User Registration**:
   - **Producer**: User Service publishes `UserCreated` event.
   - **Consumers**:
     - Booking Service subscribes to validate user details during booking.
     - Notification Service sends a welcome email.
2. **Booking Workflow**:
   - **Producer**: Booking Service publishes `BookingCreated` event.
   - **Consumers**:
     - Payment Service initiates payment processing.
     - Notification Service sends booking confirmation.
3. **Event Cancellation**:
   - **Producer**: Event Service publishes `EventCanceled` event.
   - **Consumers**:
     - Booking Service cancels active bookings.
     - Notification Service notifies users of the cancellation.

Event Types:

Event Name	Producer	Consumer(s)
`UserCreated`	User Service	Booking Service, Notification Service
`EventCreated`	Event Service	Search Service
`BookingCreated`	Booking Service	Payment Service, Notification Service
`PaymentSuccess`	Payment Service	Booking Service, Notification Service
`EventCanceled`	Event Service	Booking Service, Notification Service

Schema Registry:

- Use **Confluent Schema Registry** to enforce consistent event formats.
- Define schemas for each event type to prevent incompatible changes.

---

4. Logging Standards

Purpose:

To provide observability, troubleshooting, and auditing capabilities across services.

Centralized Logging:

- Use **ELK Stack (Elasticsearch, Logstash, Kibana)** or **AWS CloudWatch** for centralized log collection and visualization.

Log Structure:

- All logs must be in JSON format for easier parsing.
- **Standard Fields:**
  - `timestamp`: ISO 8601 timestamp of the log entry.
  - `service_name`: Name of the service generating the log.
  - `log_level`: Severity level (e.g., DEBUG, INFO, WARN, ERROR).
  - `trace_id`: Unique identifier for tracing requests across services.
  - `message`: Description of the log event.
  - `additional_context`: Key-value pairs for extra information.

Example Log Entry:

```
{
  "timestamp": "2024-11-16T12:00:00Z",
  "service_name": "Booking Service",
  "log_level": "INFO",
  "trace_id": "abc123",
```

```
  "message": "Booking created successfully",
  "additional_context": {
    "booking_id": "12345",
    "user_id": "67890",
    "event_id": "54321"
  }
}
```

Error Logging:

- Capture stack traces and errors with `ERROR` level logs.
- Log retry attempts for failed processes, such as Kafka message reprocessing.

---

5. Caching Strategy

Each service uses **Redis** for caching frequently accessed data to improve performance and reduce database load.

Caching by Service:

1. **User Service**:
   - Cache user session data and authentication tokens.
   - Key format: `user:session:{user_id}`.
2. **Event Service**:
   - Cache event details and ticket availability.
   - Key format: `event:details:{event_id}`.
3. **Search Service**:
   - Cache popular search queries and results.
   - Key format: `search:query:{hash_of_query}`.
4. **Booking Service**:
   - Cache temporary booking locks (e.g., reserved tickets).

- Key format: `booking:lock:{booking_id}`.
- TTL: 5 minutes.

Cache Eviction Policies:

- Use **Least Recently Used (LRU)** eviction to optimize cache storage.
- Set appropriate TTLs for cached data:
  - Event details: 1 hour.
  - Search results: 10 minutes.
  - Booking locks: 5 minutes.

---

6. Database Design

Per-Service Database Approach:

Each service has its own database to maintain clear boundaries and enable independent scaling.

Service	Database	Key Tables
User Service	User DB	Users, Sessions
Event Service	Event DB	Events, Tickets
Booking Service	Booking DB	Bookings, TicketReservations
Payment Service	Payment DB	Transactions, Refunds
Notification Service	Notification DB	Notifications, Templates

Replication and Backups:

- Enable **read replicas** for scaling read-heavy operations (e.g., Event DB for Search Service).
- Use AWS RDS automated backups for disaster recovery.

---

7. Observability and Metrics

Metrics Collection:

- Use **Prometheus** for collecting system and application metrics.
- Set up **Grafana Dashboards** for real-time visualization.

Key Metrics:

1. **API Gateway**:
    - Total requests, request latency, and error rates.
2. **Kafka**:
    - Message lag, consumer group health.
3. **Per Service**:
    - Success/failure rates for core operations (e.g., bookings, payments).
    - Cache hit/miss ratios.

---

8. Error Handling and Resiliency

Retry Policies:

- Use exponential backoff with retries for failed Kafka message processing.

Dead Letter Queues (DLQ):

- Route unprocessable messages to a DLQ for later inspection and recovery.

Circuit Breakers:

- Use circuit breakers design pattern to prevent cascading failures during service outages.

---

1. Overview

This RFC outlines the high-level architecture, communication strategy, caching mechanisms, and logging standards for the **Event Booking Platform**. The platform enables global event searches, ticket booking, and payment processing while ensuring scalability, reliability, and observability.

---

2. High-Level Design

The system comprises multiple microservices orchestrated via Kubernetes and designed to communicate asynchronously using Kafka. Each service is modular, maintains its own database, and operates independently. The API Gateway is powered by **Traefik**, which handles routing, authentication, and rate-limiting.

Components:

1. **Traefik Ingress (API Gateway)**:
    - Routes client requests to the appropriate backend services.
    - Enforces authentication using the **User Service** as middleware.
    - Handles rate-limiting, request logging, and TLS termination.
2. **User Service**:
    - Provides middleware for authenticating requests passing through Traefik.
    - Manages user authentication, profiles, and session tokens.
3. **Event Service**:
    - Handles event management and ticket inventory.
4. **Search Service**:
    - Provides advanced search functionality via Elasticsearch.
5. **Booking Service**:
    - Manages ticket reservations and interacts with the Payment Service.
6. **Payment Service**:
    - Processes payments using Stripe.
7. **Notification Service**:
    - Sends user notifications via email, SMS, or push notifications.
8. **Monitoring Service**:
    - Tracks system health, logs, and metrics.
9. **Kafka**:
    - Facilitates event-based communication between services.

10. **Redis**:

- Provides caching for frequently accessed data.

---

3. How Services Communicate via Kafka

Services exchange events asynchronously via Kafka to ensure scalability and fault tolerance.

Event Workflow Example:

1. **User Authentication**:
    - The User Service authenticates requests routed by Traefik.
    - For authenticated requests, Traefik forwards them to the appropriate service.
2. **User Registration**:
    - **Producer**: User Service publishes `UserCreated` event.
    - **Consumers**:
        - Booking Service validates user details during booking.
        - Notification Service sends a welcome email.
3. **Booking Workflow**:
    - **Producer**: Booking Service publishes `BookingCreated` event.
    - **Consumers**:
        - Payment Service initiates payment processing.
        - Notification Service sends booking confirmation.
4. **Event Cancellation**:
    - **Producer**: Event Service publishes `EventCanceled` event.
    - **Consumers**:
        - Booking Service cancels active bookings.
        - Notification Service notifies users of the cancellation.

Event Types:

Event Name	Producer	Consumer(s)
`UserCreated`	User Service	Booking Service, Notification Service
`EventCreated`	Event Service	Search Service
`BookingCreated`	Booking Service	Payment Service, Notification Service
`PaymentSuccess`	Payment Service	Booking Service, Notification Service
`EventCanceled`	Event Service	Booking Service, Notification Service

Schema Registry:

- Use **Confluent Schema Registry** to enforce consistent event formats.
- Define schemas for each event type to prevent incompatible changes.

---

4. Traefik API Gateway

Responsibilities:

1. **Routing**:
    - Direct requests to appropriate backend services based on predefined routes.
    - Example: `/api/events` → Event Service.
2. **Authentication Middleware**:
    - Delegates authentication to the **User Service**.
    - Checks JWT tokens in incoming requests and validates them against the User Service.
3. **Rate-Limiting**:
    - Prevents abuse by throttling excessive requests from clients.
4. **TLS Termination**:
    - Handles HTTPS encryption for all client-server communication.

Key Features:

- Uses middleware for request processing (e.g., User Service for authentication).
- Logs request metadata (e.g., IP address, headers, and routing decisions).

---

5. Logging Standards

Purpose:

To provide observability, troubleshooting, and auditing capabilities across services.

Traefik Logging:

- Capture HTTP request/response metadata.
- Include `trace_id` for correlating logs across services.

Centralized Logging:

- Use **ELK Stack (Elasticsearch, Logstash, Kibana)** or **AWS CloudWatch** for centralized log collection and visualization.

Log Structure:

- JSON format for easier parsing.
- **Standard Fields**:
  - `timestamp`, `service_name`, `log_level`, `trace_id`, `message`.

Error Logging:

- Capture stack traces and retry attempts for failed processes.

---

6. Caching Strategy

Each service uses **Redis** for caching frequently accessed data to improve performance and reduce database load.

Caching by Service:

1. **User Service**:

- Cache user session data and authentication tokens.
- Key format: `user:session:{user_id}`.

2. **Event Service**:
   - Cache event details and ticket availability.
   - Key format: `event:details:{event_id}`.
3. **Search Service**:
   - Cache popular search queries and results.
   - Key format: `search:query:{hash_of_query}`.
4. **Booking Service**:
   - Cache temporary booking locks (e.g., reserved tickets).
   - Key format: `booking:lock:{booking_id}`.
   - TTL: 5 minutes.

---

7. Database Design

Each service has its own database to maintain clear boundaries and enable independent scaling.

Service	Database	Key Tables
User Service	User DB	Users, Sessions
Event Service	Event DB	Events, Tickets
Booking Service	Booking DB	Bookings, TicketReservations
Payment Service	Payment DB	Transactions, Refunds
Notification Service	Notification DB	Notifications, Templates

Replication and Backups:

- Enable **read replicas** for scaling read-heavy operations.
- Use AWS RDS automated backups for disaster recovery.

---

8. Observability and Metrics

Traefik Metrics:

- Total requests, latency, and error rates.
- Use **Prometheus** with Grafana for real-time visualization.

Per-Service Metrics:

- Kafka: Message lag, consumer health.
- Redis: Cache hit/miss ratios.

---

9. Error Handling and Resiliency

Retry Policies:

- Use exponential backoff for failed Kafka message processing.

Dead Letter Queues (DLQ):

- Route unprocessable messages to DLQs for later inspection and recovery.

Circuit Breakers:

- Prevent cascading failures during outages.