Zentry Service Architecture

Document Version: 1.0
Date: April 30, 2025
Author: Team Zentry

Table of Contents

Executive Summary

Zentry is a sleep tracking social application that allows users to log their sleep patterns and follow other users to view their sleep entries. The application employs a distributed microservice architecture with event-driven processing to handle data consistency across services while maintaining scalability and reliability.

This document details the service architecture, explaining how the various components work together to deliver a robust, scalable application that can handle both regular users and highly-followed “celebrity” users efficiently.

System Overview

Purpose and Scope

Zentry provides functionality for: - Recording and managing sleep entries - Following/unfollowing other users - Viewing a personalized feed of sleep entries from followed users - Analyzing sleep patterns and statistics

Technical Stack

- **Backend Framework**: Ruby on Rails 8.0 (API-only mode)
- **Primary Database**: PostgreSQL (relational data)
- **Search & Analytics**: Elasticsearch 7.13 (feed and search functionality)
- **Message Broker**: Kafka (event processing)
- **Containerization**: Docker
- **Deployment**: Kamal

Design Principles

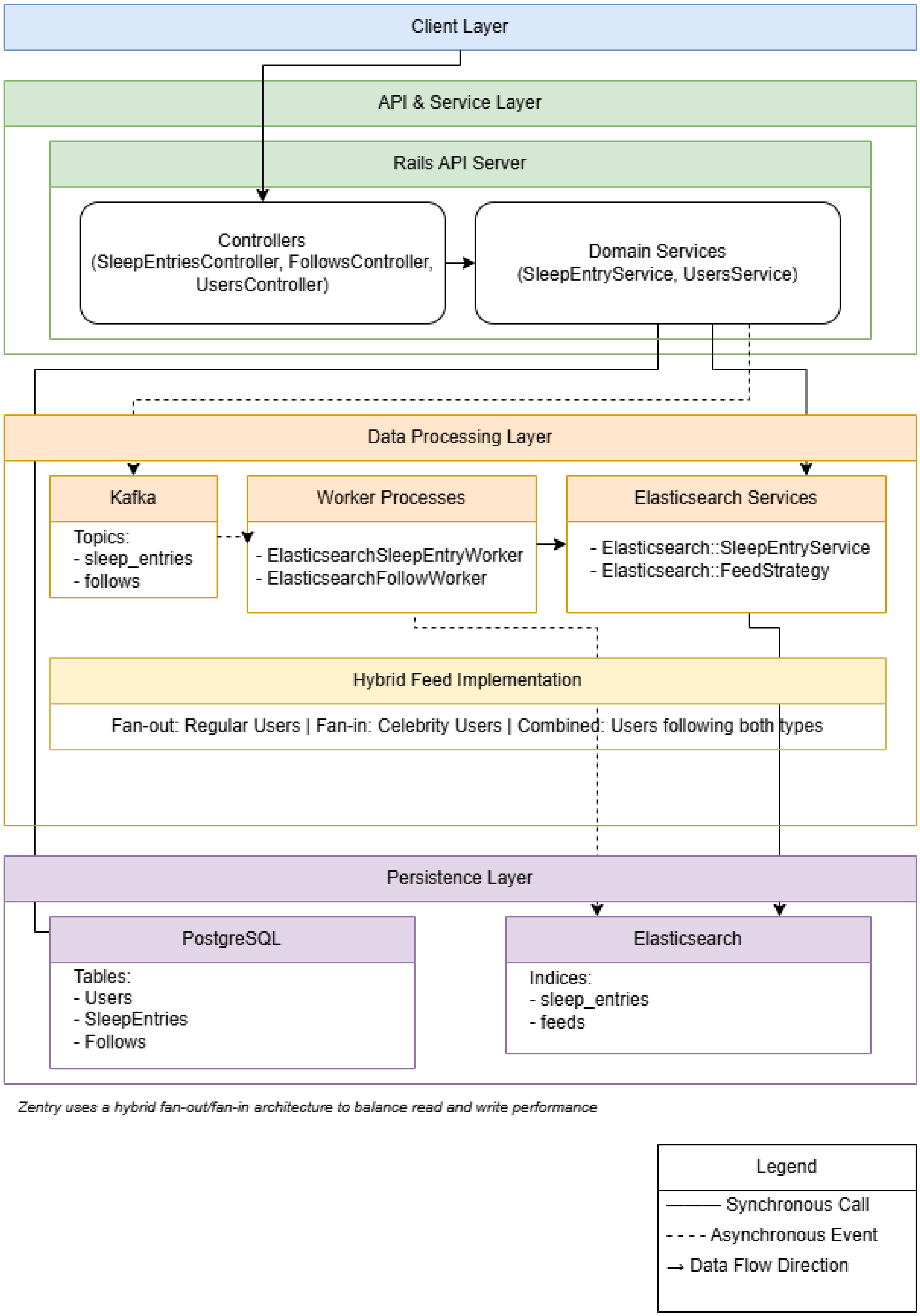
The architecture follows these core principles: - Event-driven processing for asynchronous operations - Hybrid fan-out/fan-in approach for social feed implementation - Separation of read and write paths to optimize performance - Domain-driven design with service boundaries

Service Architecture

Core Components

Service Architecture Diagram

Zentry Service Architecture



Zentry uses a hybrid fan-out/fan-in architecture to balance read and write performance

The system consists of the following major components:

1. **API Layer**: Rails controllers handling HTTP requests
2. **Domain Services**: Business logic implementation
3. **Data Access Layer**: Models and repositories
4. **Event Processing System**: Kafka producers and consumers
5. **Search & Feed Infrastructure**: Elasticsearch services and indices
6. **Background Workers**: Service workers for asynchronous processing

Component Details

API Layer

The API layer provides RESTful endpoints for all client interactions:

- **`SleepEntriesController`**: CRUD operations for sleep entries
- **`FollowsController`**: Managing follow relationships
- **`UsersController`**: User profile and authentication operations

Domain Services

Services encapsulate the core business logic:

- **`SleepEntryService`**: Management of sleep entry creation, updates and retrieval
- **`UsersService`**: User relationship management and feed generation
- **`Elasticsearch::SleepEntryService`**: Search and feed assembly from Elasticsearch

Data Access Layer

The persistence layer uses both PostgreSQL and Elasticsearch:

- **PostgreSQL Models**:
  - `User`: User account information
  - `SleepEntry`: Sleep record data
  - `Follow`: Social graph relationships
- **Elasticsearch Indices**:
  - `sleep_entries`: Primary storage for searchable sleep entries
  - `feeds`: Denormalized feed entries for regular users

Event Processing System

The event system handles data consistency across services:

- **Kafka Topics**:
  - `sleep_entries`: Events related to sleep entry operations
  - `follows`: Events related to social graph changes
- **Producers**:

  - `Kafka::Producer`: Generic producer for publishing events
- **Consumers**:
  - `ElasticsearchSleepEntryWorker`: Consumes sleep entry events
  - `ElasticsearchFollowWorker`: Consumes follow relationship events

Data Flow

Create Sleep Entry Flow

1. Client makes a POST request to `/sleep_entries`
2. `SleepEntriesController` validates input and calls `SleepEntryService.create`
3. `SleepEntryService` creates a record in PostgreSQL
4. After successful creation, a `sleep_entry_created` event is published to Kafka
5. `ElasticsearchSleepEntryWorker` consumes the event and:
   - Indexes the entry in the `sleep_entries` Elasticsearch index
   - For non-celebrity users, copies the entry to each follower's feed in the `feeds` index (fan-out)

Follow User Flow

1. Client makes a POST request to `/follows`
2. `FollowsController` validates input and calls `UsersService.create_follow`
3. `UsersService` creates a follow relationship in PostgreSQL
4. After successful creation, a `follow_created` event is published to Kafka
5. `ElasticsearchFollowWorker` consumes the event and:
   - For non-celebrity users, copies all of their sleep entries to the follower's feed (fan-out)
   - For celebrity users, no immediate action (fan-in strategy)

View Feed Flow

1. Client makes a GET request to `/feed`
2. `SleepEntriesController` calls `SleepEntryService.get_followers_feed`
3. `SleepEntryService` delegates to `Elasticsearch::SleepEntryService.feed_for_user`
4. `Elasticsearch::SleepEntryService` identifies the user's mix of followed users and:
   - For regular followed users: queries pre-computed entries in the `feeds` index (fan-out)
   - For celebrity followed users: queries their entries directly from `sleep_entries` index (fan-in)
   - Combines both result sets, sorts, and paginates for final response

Component Interactions

Hybrid Feed Implementation

The system implements a hybrid approach for feed generation:

1. **Fan-out pattern**:
   - Used for regular users with moderate follower counts
   - When a user posts a new sleep entry, it's copied to each follower's feed
   - Improves read performance at the cost of write amplification
   - Implemented in `ElasticsearchSleepEntryWorker.index_sleep_entry`
2. **Fan-in pattern**:
   - Used for "celebrity" users with large follower counts
   - Sleep entries are stored only once in the `sleep_entries` index
   - When a follower requests their feed, celebrity entries are fetched on-demand
   - Optimizes write performance while slightly increasing read complexity
   - Implemented in `Elasticsearch::SleepEntryService.fan_in_feed_query`
3. **Combined approach**:
   - For users following both regular and celebrity accounts
   - Results from both approaches are merged and sorted at query time
   - Ensures consistent behavior regardless of the followed users' profile
   - Implemented in `Elasticsearch::SleepEntryService.combined_feed_query`

Asynchronous Event Processing

The system uses Kafka for reliable event handling:

1. **Transaction with Kafka**:
   - Database operations and Kafka publishing are wrapped in transactions
   - Ensures consistency between database state and events
   - Implemented in service patterns like `UsersService.create_follow`
2. **Idempotent Workers**:
   - Workers are designed to handle duplicate messages safely
   - Each operation is idempotent, allowing for at-least-once delivery semantics
   - Retry mechanisms ensure eventual consistency

Scaling Considerations

Horizontal Scaling

The architecture supports horizontal scaling through:

1. **Stateless API Servers**:
   - Can be scaled independently based on request volume

2. **Separate Worker Processes**:
   - Kafka consumers can be deployed separately from API servers
   - Consumer groups ensure work distribution across worker instances
3. **Elasticsearch Cluster**:
   - Supports sharding and replication for performance and reliability
   - Can be scaled independently of other components

Performance Optimizations

1. **Celebrity Detection**:
   - `FeedStrategy` service determines optimal feed strategy based on follower count
   - Prevents write amplification for highly followed accounts
2. **Query Optimizations**:
   - Sorting and pagination handled efficiently
   - Date range filtering at the query level
3. **Bulk Operations**:
   - Bulk indexing for efficiency when processing large data sets

Deployment Strategy

Infrastructure Components

The production deployment consists of:

1. **Application Servers**:
   - Multiple instances of the Rails application
   - Deployed and managed using Kamal
2. **Data Stores**:
   - PostgreSQL database cluster with replication
   - Elasticsearch cluster with proper sharding
3. **Kafka Cluster**:
   - Multiple brokers for reliability
   - Zookeeper ensemble for coordination

Deployment Process

WIP

Monitoring and Observability

Logging Strategy

The application employs structured logging:

1. **Application Logs**:
   - Rails application logs

- Worker-specific logs
2. **Service Logs**:
   - Elasticsearch operational logs
   - Kafka broker and consumer group logs

Performance Monitoring

Monitoring focuses on:

1. **API Performance**:
   - Request latency
   - Error rates
2. **Worker Performance**:
   - Message processing rates
   - Consumer lag
3. **Search Performance**:
   - Query performance
   - Index health metrics

Appendix

Application Initialization Flow

1. Rails server starts
2. `elasticsearch_worker.rb` initializer starts worker threads:
   - `ElasticsearchSleepEntryWorker` for sleep entry events
   - `ElasticsearchFollowWorker` for follow relationship events
3. Workers connect to Kafka and begin consuming messages

Database Schema

Key tables and relationships:

- `users`: User profiles and authentication
- `sleep_entries`: Sleep record data linked to users
- `follows`: Social graph with follower/following relationships

Elasticsearch Index Mappings

- **sleep_entries index**:

```json
{
  "properties": {
    "id": { "type": "long" },
    "user_id": { "type": "long" },
    "sleep_entry_id": { "type": "long" },
    "sleep_duration": { "type": "long" },
    "sleep_start_at": { "type": "date" },
    "created_at": { "type": "date" },
```

```
    "updated_at": { "type": "date" }
  }
}
```

- **feeds index**:

```
{
  "properties": {
    "user_id": { "type": "long" },
    "author_id": { "type": "long" },
    "sleep_entry_id": { "type": "long" },
    "sleep_duration": { "type": "long" },
    "sleep_start_at": { "type": "date" },
    "created_at": { "type": "date" },
    "updated_at": { "type": "date" }
  }
}
```