UNIVERSITÀ DI PISA

Master's Degree in Artificial Intelligence and Data Engineering

Design and develop of an Application interacting with NoSQL Databases:

Smart News Aggregator

Instructors:
Prof. Pietro Ducange
Prof. Alessio Schiavo

Student:
Adrien Koumgang Tegantchouang

Academic Year 2024/2025

Contents

Chapter 1

Introduction

1.1 Overview

In the digital age, the volume of news articles produced every day is staggering. Readers are increasingly overwhelmed by the amount of available content and often struggle to find reliable and relevant information that matches their personal interests. To address this challenge, we propose the development of a **Smart News Aggregator Reader Personalization Platform**.

This platform collects news articles from multiple external APIs, stores and processes them using NoSQL databases (MongoDB and Redis), and delivers a personalized reading experience to users. It incorporates intelligent recommendation features, secure user authentication via JWT tokens, and real-time analytics to improve user engagement.

1.2 Objectives

The primary objective of this project is to **design and implement a distributed news aggregation application** of intelligently retrieving, storing, analyzing, and serving large-scale multi-structured data using multiple NoSQL database technologies.

In line with the course requirements for **Large Scale and Multi-Structured Databases**, the specific goals of this project include:

- **Data Acquisition & Preprocessing**: Retrieve a real-world, high-volume dataset (50MB) from multiple external news APIs (e.g., MediaStack, newsData, The

Guardian, NYTimes), ensuring **variety** and **velocity**.

- **Distributed NoSQL Architecture**: Use **MongoDB** as the primary **Document Database**, with carefully designed collections, indexes, and aggregation pipelines. Integrate a **Key-Value store** (**Redis**) to optimize caching and access to hot data.
- **System Design and Modeling**: Define functional and non-functional requirements. Design UML class diagrams and user interface mockups. Model the data schema for each NoSQL DB in use.
- **RESTful API Backend**: Build a scalable backend using Flask and Flask-RESTX. Expose secure endpoints with JWT-based authentication. Provide API documentation via Swagger and test interfaces.
- **Advanced Analytics**: Implement at least three real aggregation pipelines in MongoDB for summarizing and analyzing article content and user activity. Provide user-personalized views and filtering using advanced queries.
- **Monitoring, Testing, and Depployment**: Deploy on both and UNIPI virtual clusters. Include performance tests, system logging, and analysis of read/write throughput under different consistency models. Offer a complete presentation and demonstration, including API functionality and analytics insights.

By achieving these goals, the project demonstrates my ability to handle real-world large-scale data applications, integrating theoretical design with practical deployment, and ensuring cross-database consistency and performance.

1.3 Structure of the presentation

The documentation is structured as follows:

- **Chapter 1 - Introduction**: Presents the context, objectives, and structure of the project, outlining the motivations and academic scope.
- **Chapter 2 - Requirements Analysis**: Identifies the functional and non-functional requirements of the application, defines the system actors, and outlines key use cases.

- **Chapter 3 - System Design**: Includes the architectural overview, UML class diagrams, and mockup wireframes of the application's user interface.
- **Chapter 4 - NoSQL Database Modeling**: Describes the design choices for MongoDB (Document DB) and Redis (Key-Value DB), along with schema definitions, key structures, and CAP theorem considerations.
- **Chapter 5 - Data Ingestion and Integration**: Explains how external APIs are connected, data is fetched and transformed, and stored into the databases. Also includes logging and handling of errors and API rate limits.
- **Chapter 6 - Backend Implementation**: Details the development of RESTful API endpoints using Flask and Flask-RESTX, JWT authentication, Swagger documentation, and the modular blueprint structure.
- **Chapter 7 - Advanced Aggregations and Analytics**: Presents non-trivial aggregation pipelines in MongoDB, user-specific personalization features, and analytics insights extracted from the dataset.
- **Chapter 8 - Deployment and Testing**: Discusses deployment on local and virtualized environments, performance testing scenarios, and consistency benchmarking across NoSQL systems.
- **Chapter 9 - Conclusion**: Summarizes achievements, reflects on encountered challenges, and suggests possible extensions to improve scalability, interactivity, and intelligence.

Chapter 2

Requirements Analysis

2.1 Problem Statement

With the exponential growth of online content, users are overwhelmed by the volume of news available across platforms. This leads to difficulty in identifying relevant and trustworthy information. The proposed system aims to aggregate articles from various news APIs and personalize the reading experience based on user behavior and preferences, while maintaining scalability and high performance.

2.2 Stackeholders and Actors

The successful deployment and functioning of the Smart News Aggregator & Reader Personalization Platform depends on multiple stakeholders and actors who interact directly or indirectly with the system. This section outlines their roles and responsibilities.

2.2.1 Stackeholders

- **End Users**: Consume and interact with news content; expect personalized and relevant information.
- **Platform Administrator**: Oversees user activity, ensures data integrity, and manages access or moderation tasks.

- **Project Developers**: Build and maintain the system backend, frontend, and data processing pipelines.
- **External API Providers**: Supply news content (e.g., NYTimes, Guardian, NewsData, CurrentsAPI, MediaStack).
- **Academic Supervisors**: Oversee the project's architecture, correctness, and evaluate its educational objectives.

2.2.2 Actors

- **Visitor (Anonymous user)**: Accesses the welcome page and Can register to become a user.
- **Registered User**: Logs into the platform, Personalizes preferences, Views news feed and Interacts with articles (like/comment).
- **Administrator**: Manages users and platform configurations and Monitors logs and analytics.
- **News Aggregator Service**: Fetches articles from external APIs and Normalizes and stores them into MongoDB.
- **External News APIs**: Provide raw article data in JSON format for ingestion by the system.

Each actor has specific actions and permissions that are further detailed in the Use Case and UML Diagrams.

2.3 Functional Requirements

The Smart News Aggregator & Reader Personalization Platform offers a suite of functionalities that serve both user-facing and administrative purposes. The following functional requirements have been identified:

2.3.1 User Account Management

- **Registration**: Users must be able to register with valid email and password.
- **Login/Logout**: Authenticated access via JWT-based login; logout clears session on frontend.
- **Profile Management**: Users can update personal data (e.g., name, preferences).
- **Password Handling**: Secure password storage and update with history tracking.

2.3.2 News Aggregation & Display

- **Fetch Articles from External APIs**: The system retrieves and normalizes article data from third-party providers such as NYTimes, NewsData, CurrentsAPI, etc.
- **Categorized News Display**: Articles are displayed by category (e.g., politics, tech, sports).
- **Search & Filter**: Users can search articles using keywords and filter by category, source, or date.
- **Pagination**: Article lists are paginated to handle large datasets.

2.3.3 Personalization & Recommendation

- **Save Preferences**: Users can set preferred categories, sources, or languages.
- **Personalized Feed**: Articles are filtered and ranked based on user preferences.
- **Like & Save Articles**: Users can like or save articles for future reading.
- **Commenting System**: Users can add comments and replies to articles.

2.3.4 Admin & Analytics

- **Dashboard Access**: Admin can view usage statistics and system logs.
- **User Management**: Admin can deactivate or delete user accounts.

- **API Health Monitoring**: Admin is notified of errors when external APIs fail.
- **Article Quality Control**: Admin can remove duplicated or malformed articles.

2.3.5 System & Logging

- **Authentication Event Logs**: Login/Logout attempts are store for audit.
- **Error Tracking**: Failed requests or errors are stored in MongoDB.

2.4 Non-Functional Requirements

This section outlines the quality attributes and constraints the Smart News Aggregator & Reader Personalization Platform must meet to ensure reliability, security, and scalability.

2.4.1 Performance

The system must handle a high number of concurrent users with minimal latency. Article feed pages must load within 1 second under normal network conditions. External API data must be cached using Redis to reduce response time and load.

2.4.2 Scalability

The backend architecture must support horizontal scaling to accommodate growing numbers of users and API integrations. MongoDB and Redis must be configured to handle large-scale document and key-value datasets efficiently.

2.4.3 Availability

The application must ensure high availability (99.9%) during user access hours. In the case of third-party API failure, the system should provide fallback responses or cached data when available.

2.4.4 Security

User authentication must use JWT with asymmetric encryption (RS256). Passwords must be securely hashed (using bcrypt) and stored with history to prevent reuse. All endpoints must enforce token validation for sensitive operations. Cross-Origin Resource Sharing (CORS) must be properly configured to restrict access to allowed domains.

2.4.5 Maintainability

The system must be modular, separating concerns by feature (auth, articles, users). Logging and exception handling must be centralized to simplify debugging and maintenance. Code must be written following best practices and TypeScript (frontend) and Python (backend) style guides.

2.4.6 Usability

The frontend must offer an intuitive user interface with minimal learning curve. Responsive design must ensure accessibility on desktops, tablets, and mobile devices.

2.4.7 Portability

The application must run on Unix-based systems and be container-ready (Docker-compatible). APIs must be documented using Swagger/OpenAPI to allow easy integration by external systems.

2.5 Use Case Diagrams

These diagrams were created using PlantUML.

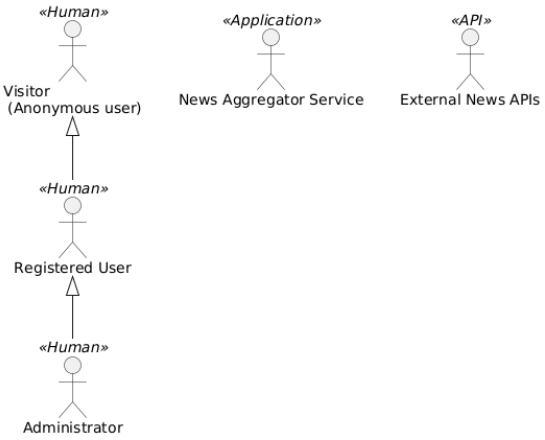

Figure 2.1: Use Case Diagram : Actors

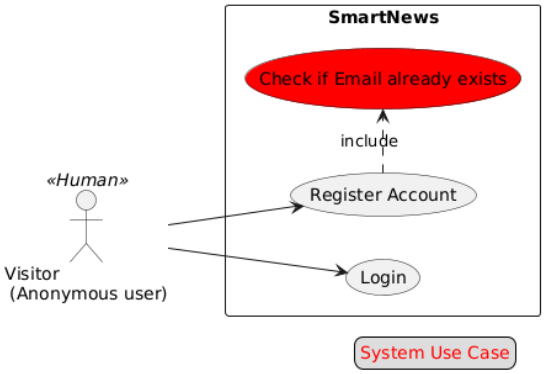

Figure 2.2: Use Case Diagram: Anonymous User

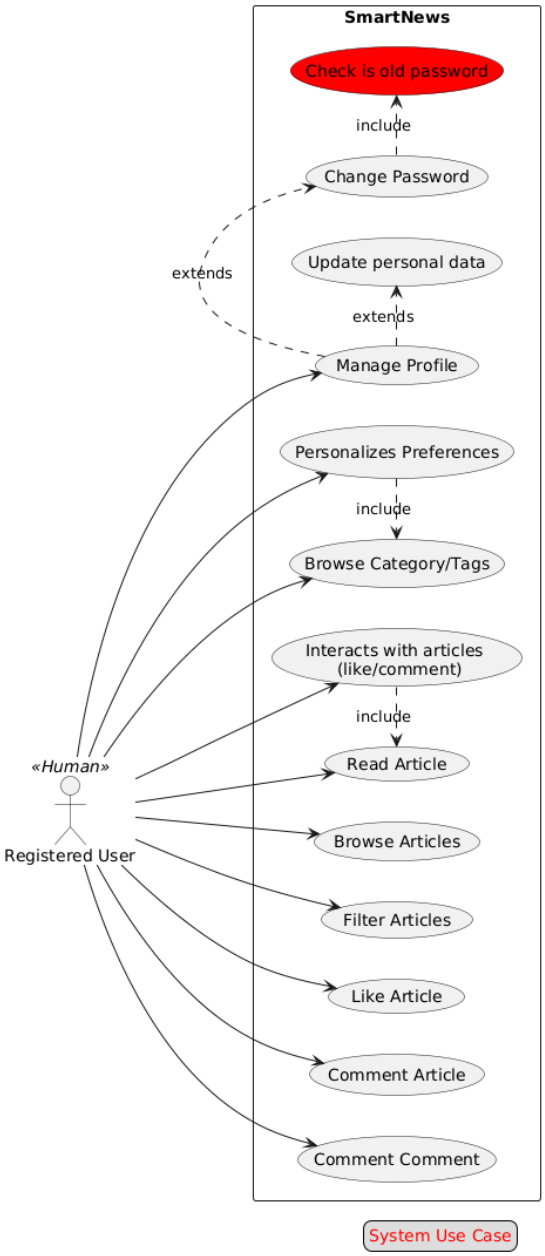

Figure 2.3: Use Case Diagram: Registered User

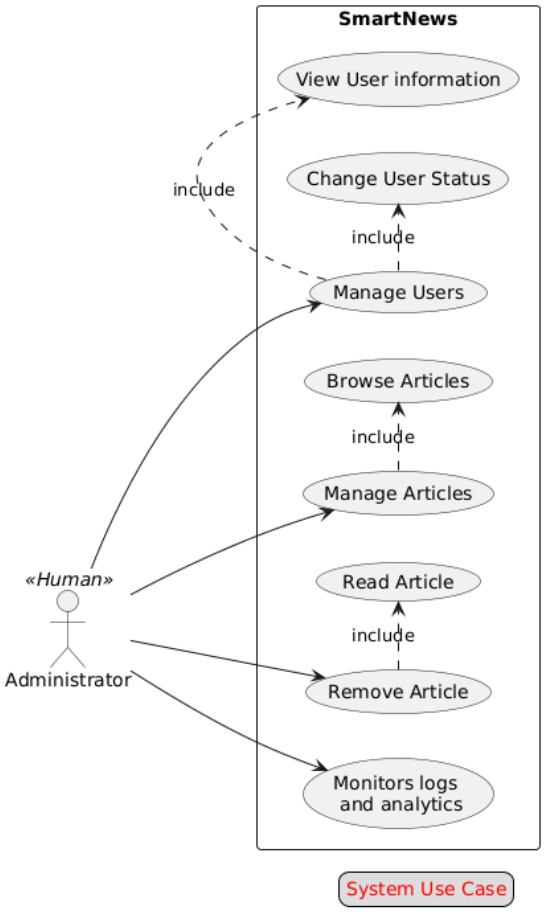

Figure 2.4: Use Case Diagram: Administrator

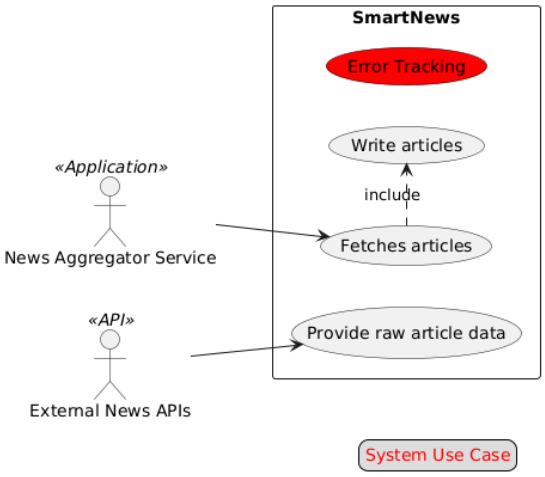

Figure 2.5: Use Case Diagram: Systen

Chapter 3

System Design

This chapter presents the architectural and structural design of the Smart News Aggregator & Reader Personalization Platform. It includes a high-level system architecture, database modeling strategy, UML class diagrams, and mockups of key user interfaces.

3.1 Architecture Overview

The platform is built using a **modular and layered architecture** to separate concerns and ensure maintainability, performance, and scalability. It integrates multiple technologies:

3.1.1 Key Components

- **Frontend (React + TypeScript)**: Handles UI, user interaction, API communication and Token-based authentication.
- **Backend (Flask + Flask-RESTX)**: RESTful API with Blueprint organization, JWT authentication with RS256 and Logging and error handling.
- **Document Database (MongoDB)**: Stores user profiles, articles, comments, and logs. Supports complex queries and aggregation pipelines.
- **Key-Value Store (Redis)**: Caches trending articles and recent queries. Session tracking.

- **External APIs**: Article data sources such as CurrentsAPI, NewsData, NYTimes, Guardian.

3.2 System Architecture Diagram

The architecture is divided into four layers:

- **Client Layer**: Browser-based React frontend
- **Application Layer**: Flask server handling HTTP requests, routing, and validation
- **Data Layer**: MongoDB for structured content, Redis for quick key access
- **External Sources**: Third-party APIs for news ingestion

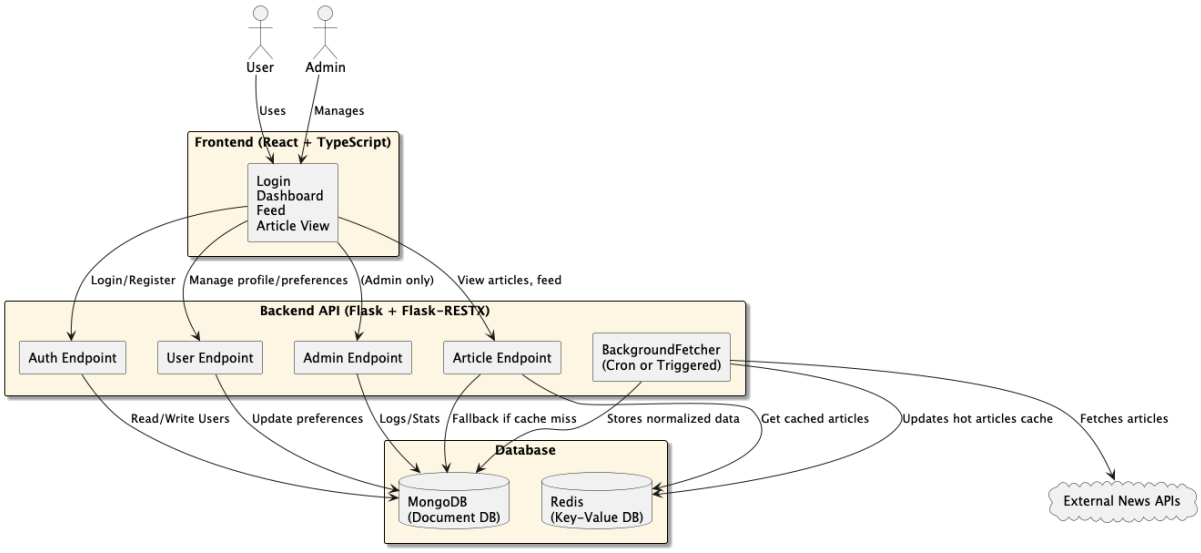

Figure 3.1: Use Case Diagram: Systen

3.2.1 Client Layer (Frontend)

At the forefront is the **Client Layer**, which consists of a web-based frontend developed using React and TypeScript. This layer handles the user interface, manages routing, and communicates with the backend API through Axios. JWT tokens are stored in the browser's localStorage for session management. Users interact with interfaces such as login, registration, the personalized news feed, article detail views, and, for administrators, a dedicated dashboard.

3.2.2 Application Layer (Flask Backend)

The **Application Layer** is built on Flask with Flask-RESTX to expose RESTful endpoints in a modular architecture. This layer is responsible for authenticating users using asymmetric JWT (RS256), routing client requests to appropriate service handlers, and managing background tasks like scheduled fetching of articles. It includes modules like auth_endpoint, user_endpoint, article_endpoint, and admin_endpoint, as well as libraries for handling authentication keys and logging system events into MongoDB.

3.2.3 Data Layer

The **Data Layer** integrates two complementary NoSQL technologies. MongoDB, as a document-oriented database, is used to store structured collections such as users, articles, comments, preferences, and system logs. It supports advanced aggregation pipelines and enables efficient historical analysis. Redis, on the other hand, serves as a key-value store optimized for performance. It caches the latest articles, manages trending topics, and supports rate limiting. Redis ensures low-latency data access and reduces the load on MongoDB for repetitive queries.

3.2.4 External APIs Layer

At the periphery, the **External APIs Layer** includes third-party news services like MediaStack, CurrentsAPI, NewsData, the New York Times API, and the Guardian API. A background fetcher service, either scheduled or triggered, connects to these APIs, retrieves articles, normalizes the data, and saves the cleaned results into MongoDB. This ensures that the content database is continuously updated with fresh news.

The data flow follows a secure and performance-driven path. When a user logs in, they are authenticated via JWT, and the token is returned in the Authorization header. The frontend then retrieves the latest articles by calling the /article/latest endpoint. If the content is cached in Redis, it is served directly; otherwise, Flask queries MongoDB. Meanwhile, the background fetcher periodically updates the article repository. Personalized recommendations are generated by combining user preferences with

MongoDB aggregations and Redis-cached data. Administrators can monitor platform activity and user engagement via metrics obtained through aggregation queries stored in MongoDB.

3.3 UML Class Diagram

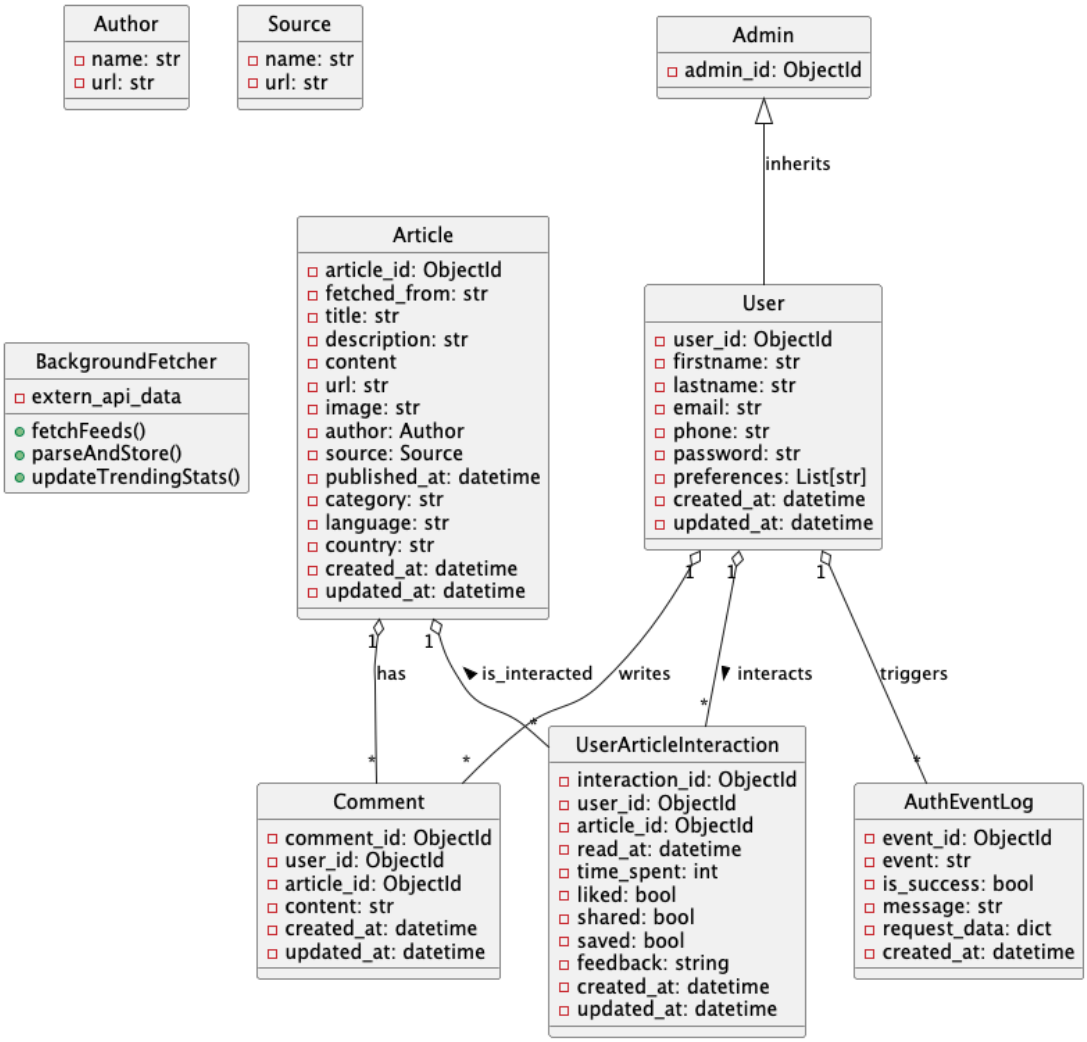

Figure 3.2: Use Case Diagram: Systen

3.4 Database Modeling

3.4.1 Document-Oriented Schema (MongoDB)

MongoDB stores structured documents for core domain entities. Each entity is represented as a collection with JSON-like documents. The key collections are:

- Users Collection

```
{
  "_id": ObjectIt,
  "email": "user@example.com",
  "password": "hashedpassword",
  "firstname": "Alice",
  "lastname": "Smith",
  "createdat": ISODate,
  "preferences": {
    "categories": ["technology", "sports"],
    "languages": ["en"],
    "sources": ["NYTimes", "Guardian"]
  }
}
```

- Articles Collection

```
{
  "_id": ObjectIt,
  "externid": "nyt-123456",
  "title": "Breaking News",
  "content": "Full article content...",
  "author": "Journalist Name",
  "source": "NYTimes",
  "url": "https://...",
  "imageurl": "https://...",
  "publishedat": ISODate,
  "category": "technology",
  "language": "en",
  "country": "us"
}
```

- User Articles Interactions Collection

```
{
  "id": ObjectIt,
  "userid": ObjectIt,
  "articleid": ObjectIt,
  "readat": ISODate,
  "timespent": 120,
  "liked": true,
  "shared": false,
  "saved": false,
  "feedback": "Interesting article on AI trends."
}
```

- Comments Collection

```
{
  "id": ObjectIt,
  "userid": ObjectIt,
  "articleid": ObjectIt,
  "content": "This article was really informative.",
  "createdat": ISODate,
  "replies": [
    {
      "userid": ObjectIt,
      "content": "I agree!",
      "createdat": ISODate
    }
  ]
}
```

- Logs Collection

```
{
  "id": ObjectIt,
  "event": "login",
  "userid": ObjectIt,
  "issuccess": true,
```

```
  "requestdata": {
    "method": "POST",
    "url": "/api/auth/login",
    "headers": { ... },
    "body": { ... }
  },
  "message": "Login successful!",
  "createdat": ISODate
}
```

3.4.2 Key-Value Cache (Redis)

3.5 UI Wireframes and Mockups

3.5.1 Authentication Screens

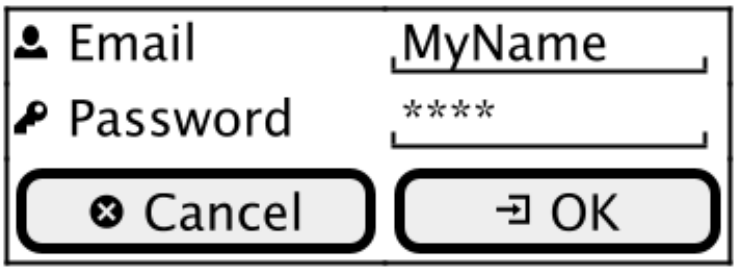

Figure 3.3: Use Case Diagram: Systen

Figure 3.4: Use Case Diagram: Systen

3.5.2 Article Feed View

Bibliography

Acknowledgments

I thank God who gave me the strength to do this project.