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 [1] 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** [11], 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** [6] as the primary **Document Database**, with carefully designed collections, indexes, and aggregation pipelines. Integrate a **Key-Value store** (**Redis** [7]) 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 [9] and Flask-RESTX [3]. 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 [5]. 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 - Data 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.

1.4 Repository

- Backend: Smart News Aggregator API
- Fronted: Smart News Aggregator Frontend React
- Documentation: Smart News Aggregator Documentation

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.

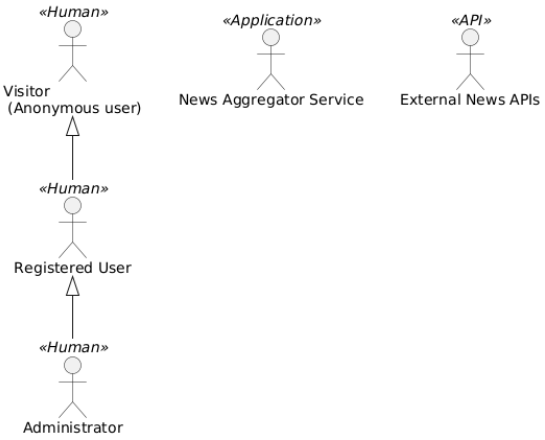

Figure 2.1: Use Case Diagram : Actors

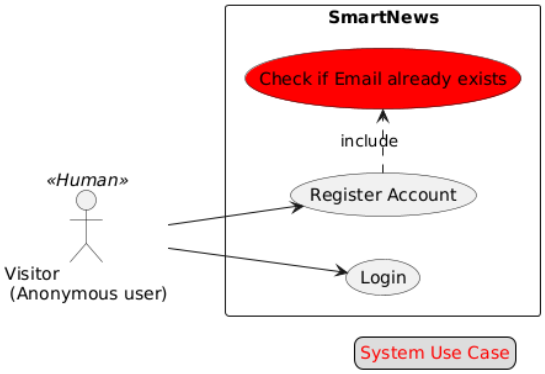

Figure 2.2: Use Case Diagram: Anonymous User

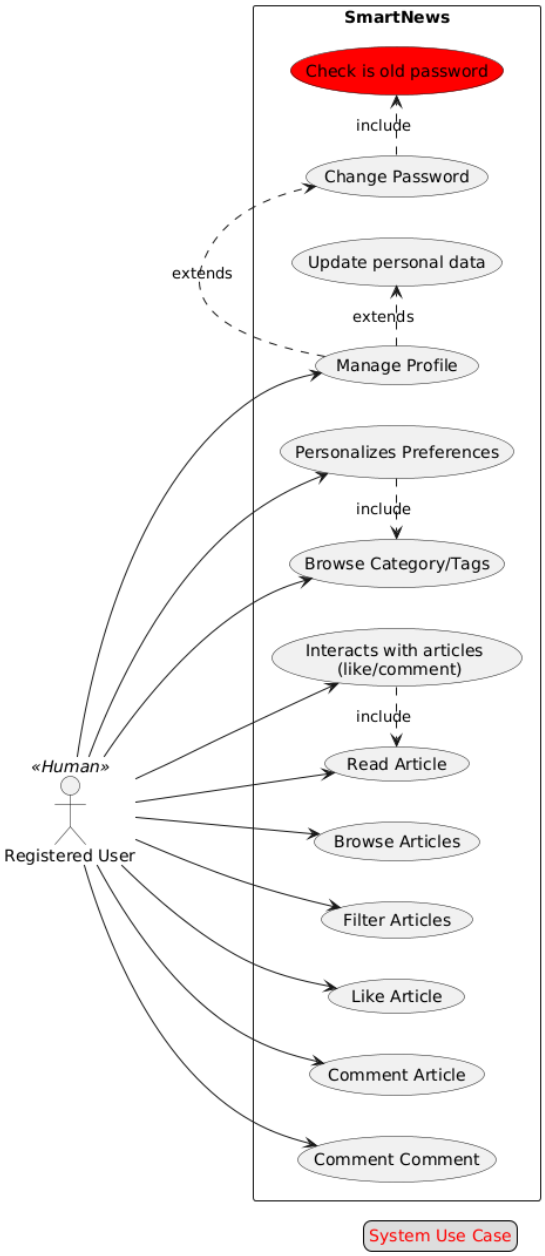

Figure 2.3: Use Case Diagram: Registered User

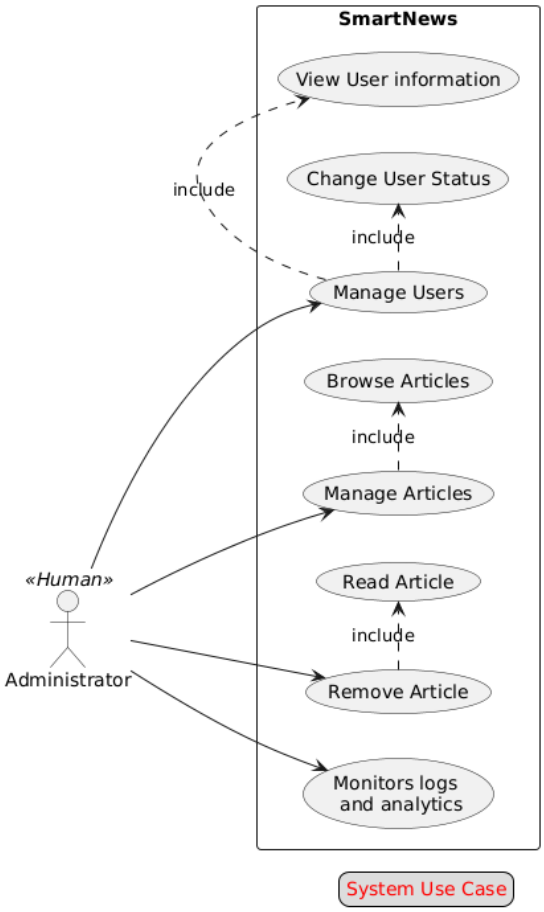

Figure 2.4: Use Case Diagram: Administrator

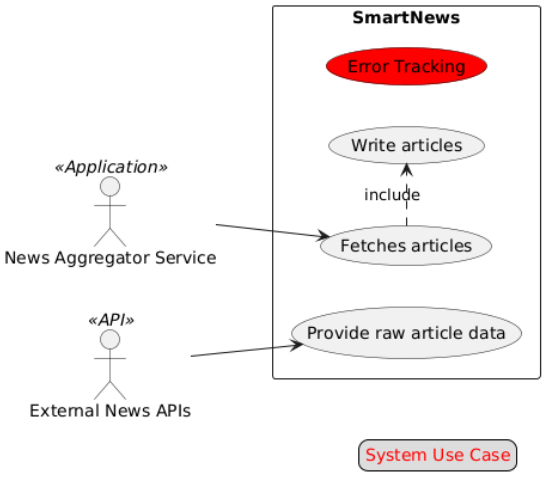

Figure 2.5: Use Case Diagram: Systen

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 [4,12] + 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

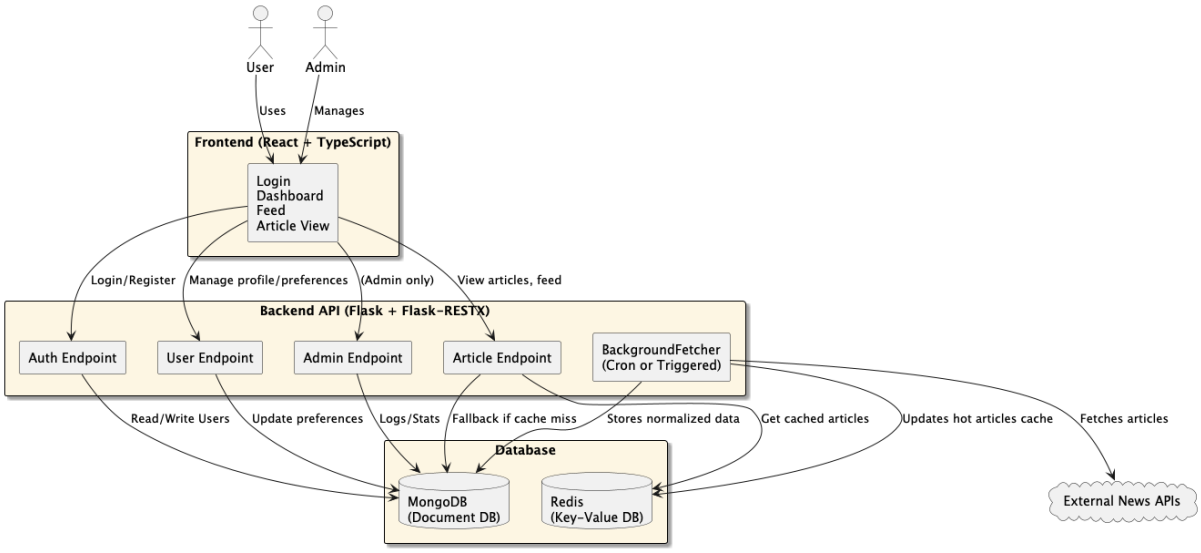

Figure 3.1: System Architecture

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

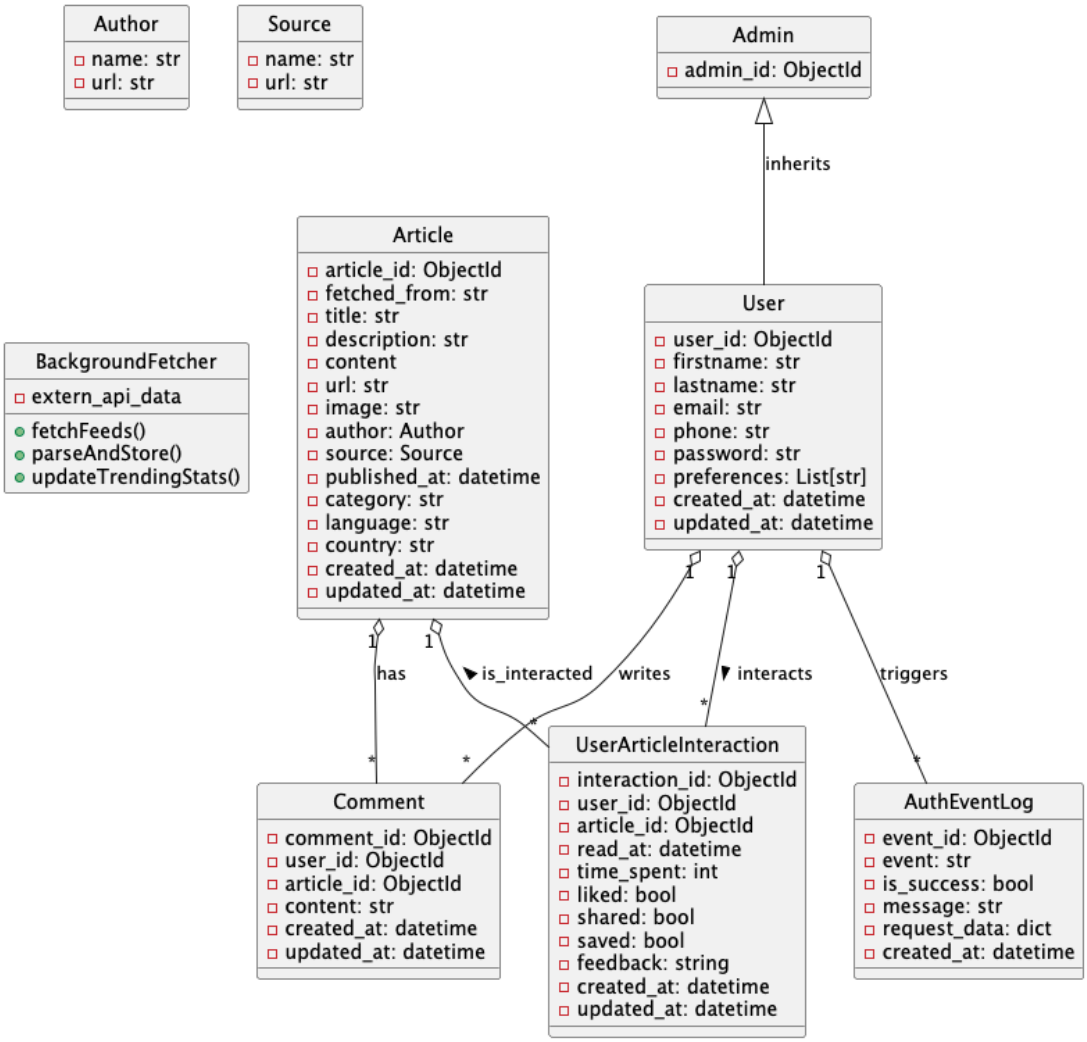

Figure 3.2: Use Case Diagram: Systen

3.4 UI Wireframes and Mockups

3.4.1 Authentication Screens

For registered new user and login:

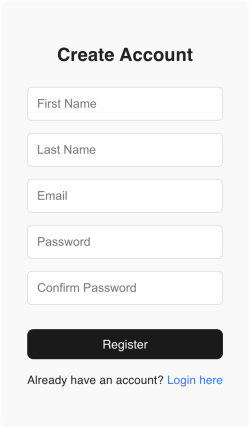

Figure 3.3: Authentication screens: registration page

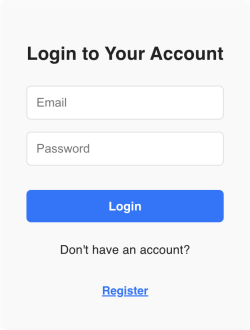

Figure 3.4: Authentication screens: login page

3.4.2 User Screens

For see user all user information. It's not possible for user with role "user" to modify status and role.

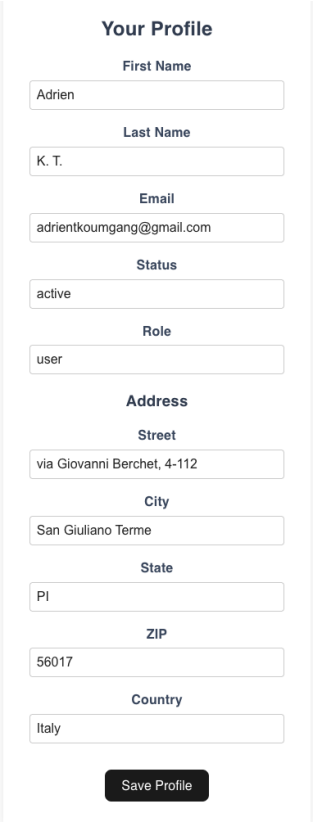

Figure 3.5: User screens: profile page

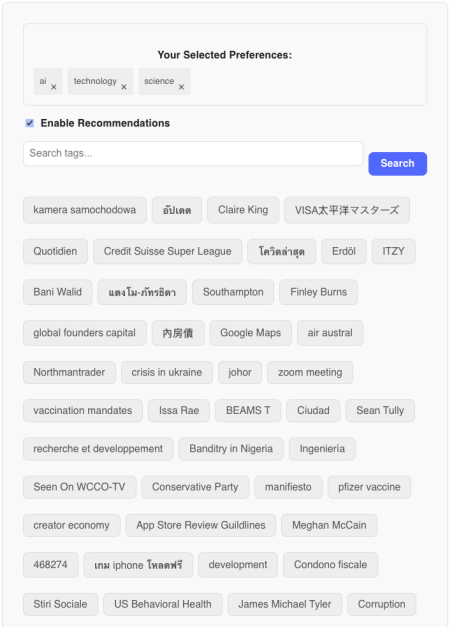

Figure 3.6: User screens: settings page

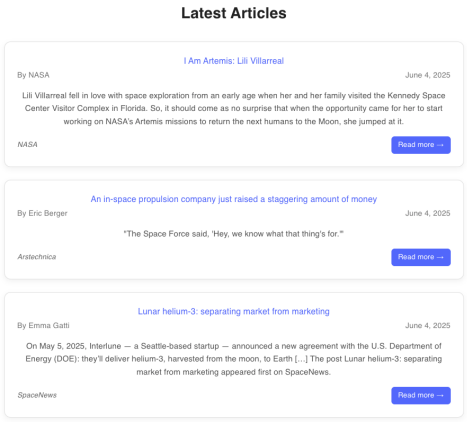

Figure 3.7: User screens: latest page

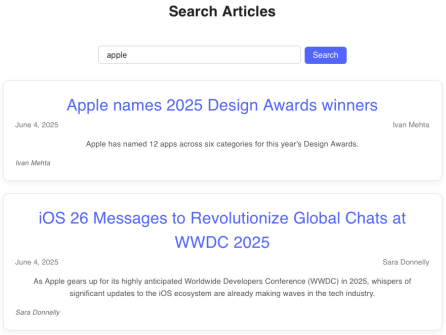

Figure 3.8: User screens: Search page

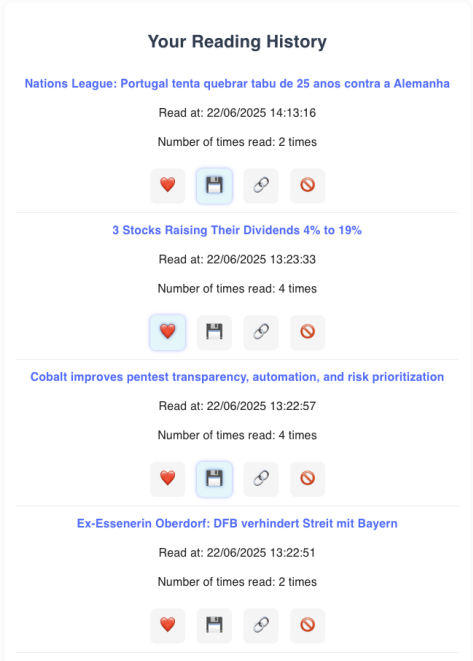

Figure 3.9: User screens: History page

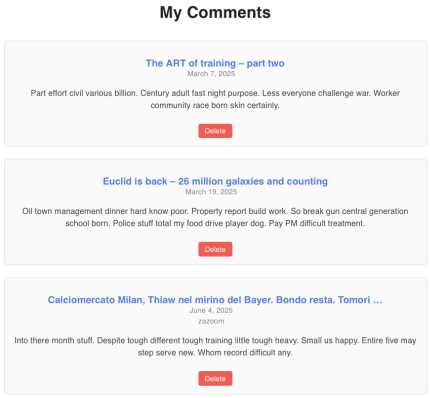

Figure 3.10: User screens: Comments page

3.4.3 Admin Screens

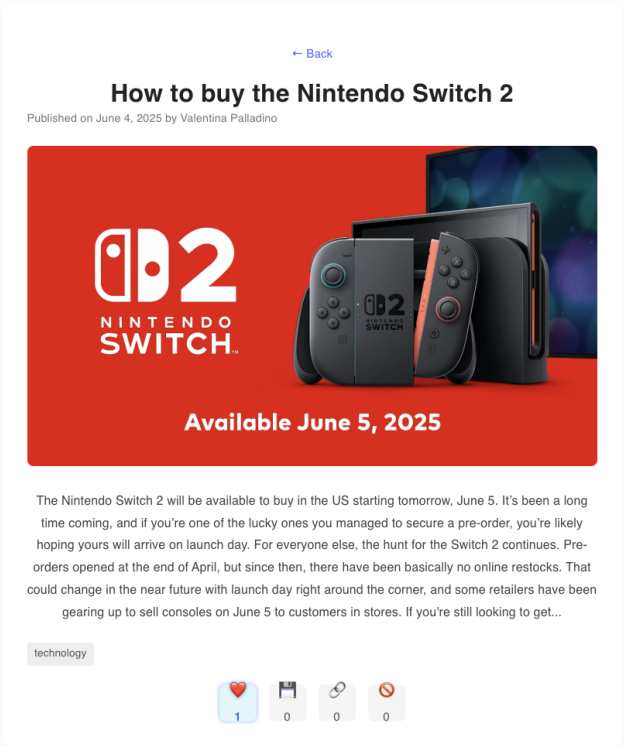

Figure 3.11: User screens: articles details page

Figure 3.12: Admin screens: admin dashboard page

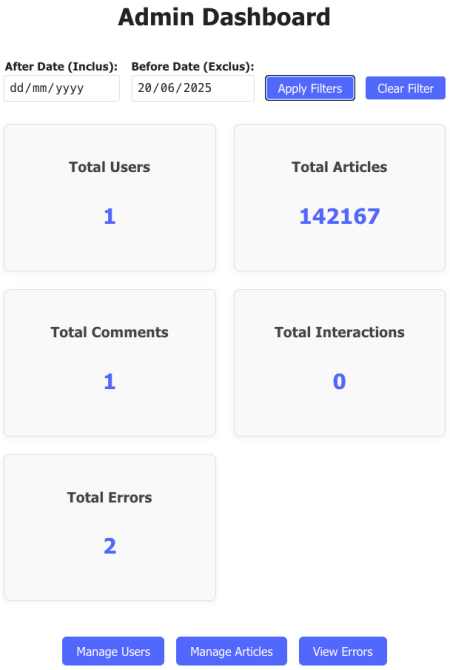

Figure 3.13: Admin screens: admin dashboard page

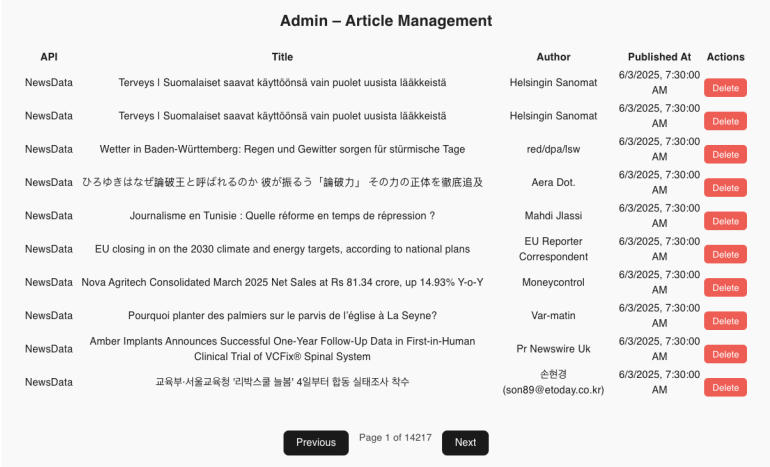

Figure 3.14: Admin screens: admin articles page

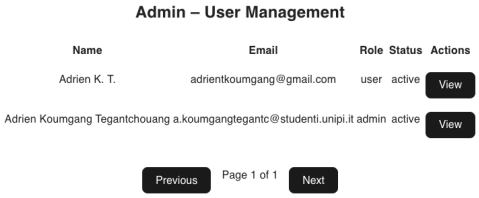

Figure 3.15: Admin screens: admin users page

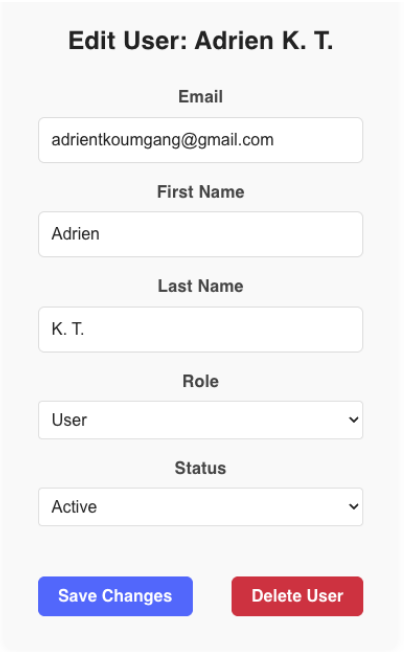

Figure 3.16: Admin screens: admin user details page

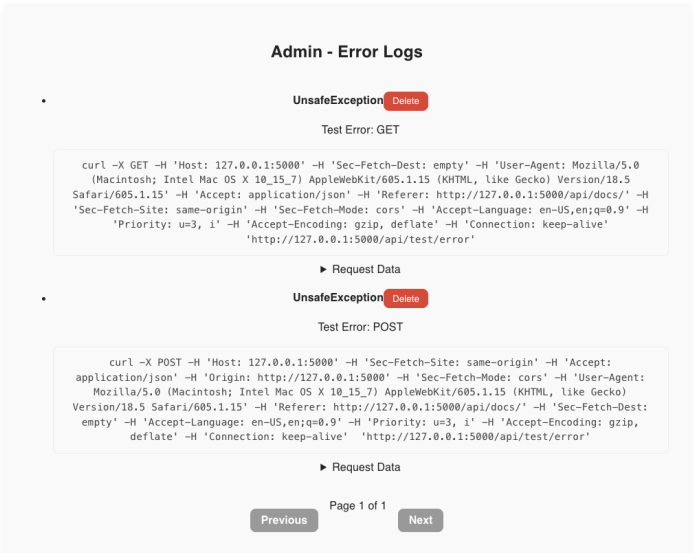

Figure 3.17: Admin screens: admin error logs page

Chapter 4

Data Modeling

In designing the project, I employed a **hybrid data modeling strategy** that combines both **relational** and **NoSQL (document-based and key-value)** approaches to balance data consistency, flexibility, and performance at scale.

4.1 Databases Technologies Used

- **MongoDB (Document Database, version 6.0)**: Used for modeling core entities like user, articles, comments, user interactions and logs.
- **Redis (Key-Value Store, version 7.0)**: Used for caching.

4.2 MongoDB Data Models

MongoDB is used as the primary **document database** to store structured yet flexible data such as users, articles, comment, interactions and errors server. Its schema-less nature and native support for nested documents make it suitable for dynamic content and high-volume ingestion [2].

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

4.2.1 Users collection

```
{
  "id": {
```

```
    "$oid": "683c758466d99ae51f0508d4"
  },
  "createdat": {
    "$date": "2025-06-01T15:45:08.196Z"
  },
  "updatedat": {
    "$date": "2025-06-21T03:25:04.294Z"
  },
  "firstname": "Adrien",
  "lastname": "K. T.",
  "email": "adrientkoumgang@gmail.com",
  "password": "$2b$12$tSDjoy8m4IMBS6T/As5zveHctHxvkUmaslrONegD4kRPpkgPARlJK",
  "account": {
    "status": "active",
    "role": "user"
  },
  "passwordhistory": [
    {
      "password": "$2b$12$guC9jr36y2MEW50le9S4wOdKbZsVnmRrXAvk5S2IMJ4QSkJSTaG5e"
          ,
      "createdat": {
        "$date": "2025-06-01T15:45:08.196Z"
      }
    },
    {
      "password": "$2b$12$tSDjoy8m4IMBS6T/As5zveHctHxvkUmaslrONegD4kRPpkgPARlJK"
          ,
      "createdat": {
        "$date": "2025-06-05T20:51:45.379Z"
      }
    }
  ],
  "preferencesenable": true,
  "preferences": [
    "ai",
    "technology",
    "science"
  ],
  "address": {
```

```
    "street": "via Giovanni Berchet, 4-112",
    "city": "San Giuliano Terme",
    "state": "PI",
    "zip": "56017",
    "country": "Italy"
  }
}
```


Indexes:

1. $\{email : 1\}$ (Unique) : Email indexes enables fast login lookups and registration check.

2. $\{preferences : 1\}$ (Multikey) : Preferences index accelerates personalized feed generation.

4.2.2 Article log requests collection


```
{
  "id": {
    "$oid": "68402a68a26f0389a8c4ed9e"
  },
  "createdat": {
    "$date": "2025-06-04T11:13:44.758Z"
  },
  "updatedat": {
    "$date": "2025-06-04T11:13:44.758Z"
  },
  "source": "CurrentsAPI",
  "url": "https://api.currentsapi.services/v1/latest-news",
  "request": {
    "url": "https://api.currentsapi.services/v1/latest-news",
    "headers": {
      "Authorization": "97ueDaaD6tgJItwJhzKGqd1FYoXm3xZ6H3-lAxKXloZmRYdm"
    },
    "params": {}
  },
  "response": {
    "statuscode": 200,
```

```
    "returned": 30
  },
  "fetchedcount": 30
}
```


Indexes:

1. $\{created\_at : -1, source : 1\}$ (Compound): optimize time-based queries filtered by API source. Monitoring recent API fetch operations by provider.

2. $\{response.status\_code : -1\}$: quickly identify failed requests for error monitoring and alerting.

4.2.3 Articles collection


```
{
  "id": {
    "$oid": "6840d72e34b1dd6ef4c87b05"
  },
  "createdat": {
    "$date": "2025-06-04T23:30:53.992Z"
  },
  "updatedat": {
    "$date": "2025-06-04T23:30:53.992Z"
  },
  "externid": "b13b8ed2543f2ca7e1874fc2e63282e0",
  "externapi": "NewsData",
  "title": "EU closing in on the 2030 climate and energy targets, according to
      national plans",
  "description": "EU member states have significantly closed the gap to
      achieving the 2030 energy and climate targets, according to the European
      Commission's assessment of the National Energy and Climate Plans (NECPs).
      EU countries have substantially improved their plans following Commission
      recommendations in December 2023. As a result, the EU is closing in
      collectively on a 55% reduction in greenhouse gas (GHG) emissions, as
      committed [...]",
  "content": "ONLY AVAILABLE IN PAID PLANS",
  "url": "https://www.eureporter.co/environment/2025/06/03/eu-closing-in-on-the
      -2030-climate-and-energy-targets-according-to-national-plans/",
```

```
  "author": {
    "name": "EU Reporter Correspondent"
  },
  "source": {
    "name": "Eureporter Co",
    "url": "https://www.eureporter.co"
  },
  "publishedat": "2025-06-03 07:30:00",
  "language": "english",
  "country": "united kingdom",
  "tags": [
    "environment",
    "full-image",
    "climate-neutral economy",
    "climate change",
    "featured"
  ]
}
```


Indexes:

1. $\{external\_api : 1, external\_id : 1, title : 1\}$ (Compound, Unique) : External ID index prevents duplicates articles.
2. $\{published\_at : -1\}$: Date indexes support chronological queries
3. $\{tags : 1, published\_at : -1\}$ (Compound): this compound index optimizes feed generation
4. $\{tags : 1\}$: for search all possible tags in all articles
5. $\{title : "text", description : "text"\}$ (Text): Text index enables full-text search. (Creation failure due to invalid characters such as Japanese and other non-Latin characters.)

4.2.4 Comments collection


```
{
  "id": {
```

```
    "$oid": "684d39437a0814577744e01d"
  },
  "createdat": {
    "$date": "2025-06-14T08:26:06.817Z"
  },
  "updatedat": {
    "$date": "2025-06-14T08:26:06.818Z"
  },
  "userid": "683c758466d99ae51f0508d4",
  "articleid": "6840dbaa2c0129e299fbd9db",
  "content": "My first comment"
}
```


Indexes:

1. $\{article\_id : 1, created\_at : -1\}$ (Compound): this index optimizes comment threading.
2. $\{user\_id : 1\}$: User index supports profile activity views
3. $\{user\_id : 1, created\_at : -1\}$ (Compound): User index supports profile activity views
4. $\{comment\_fk : 1, created\_at : -1\}$ (Compound): this index enables efficient nested comment retrieval

4.2.5 User Interactions collection


```
{
  "id": {
    "$oid": "6855a9d4ecbd85c7cbce08e2"
  },
  "articleid": "6840db802c0129e299fbbe29",
  "userid": "683c758466d99ae51f0508d4",
  "articletitle": "Apple names 2025 Design Awards winners",
  "levelinteraction": "article",
  "liked": true,
  "readat": {
    "$date": "2025-06-20T18:35:00.451Z"
```

```
  },
  "saved": true,
  "shared": false,
  "timespent": 2,
  "updatedat": {
    "$date": "2025-06-20T18:35:00.451Z"
  },
  "report": false
}
```


Indexes:

1. $\{user\_id : 1, read\_at : -1\}$ (Compound): this index powers reading history
2. $\{article\_id : 1\}$: this index supports engagement analytics
3. $\{article\_id : 1, updated\_at : -1\}$ (Compound): this index powers reading article stats

4.2.6 Server Error Logs collection


```
{
  "_id": {
    "$oid": "684241a852621fd670c36a1c"
  },
  "createdat": {
    "$date": "2025-06-06T01:17:05.512Z"
  },
  "updatedat": {
    "$date": "2025-06-06T01:17:05.512Z"
  },
  "requestdata": {
    "url": "http://127.0.0.1:5000/api/test/error",
    "method": "POST",
    "body": {},
    "args": {},
    "headers": {
      "Host": "127.0.0.1:5000",
      "Sec-Fetch-Site": "same-origin",
      "Accept": "application/json",
```

```
      "Origin": "http://127.0.0.1:5000",
      "Sec-Fetch-Mode": "cors",
      "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10 1 5 7 ) AppleWebKit
          /605.1.15 (KHTML, like Gecko) Version/18.5 Safari/605.1.15",
      "Referer": "http://127.0.0.1:5000/api/docs/",
      "Sec-Fetch-Dest": "empty",
      "Content-Length": "0",
      "Accept-Language": "en-US,en;q=0.9",
      "Priority": "u=3, i",
      "Accept-Encoding": "gzip, deflate",
      "Connection": "keep-alive"
    },
    "form": {}
  },
  "curl": "curl -X POST -H 'Host: 127.0.0.1:5000' -H 'Sec-Fetch-Site: same-
      origin' -H 'Accept: application/json' -H 'Origin: http://127.0.0.1:5000' -
      H 'Sec-Fetch-Mode: cors' -H 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
       OS X 10 1 5 7 ) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/18.5
      Safari/605.1.15' -H 'Referer: http://127.0.0.1:5000/api/docs/' -H 'Sec-
      Fetch-Dest: empty' -H 'Accept-Language: en-US,en;q=0.9' -H 'Priority: u=3,
       i' -H 'Accept-Encoding: gzip, deflate' -H 'Connection: keep-alive'  'http
      ://127.0.0.1:5000/api/test/error'",
  "exceptionname": "UnsafeException",
  "exceptionmessage": "Test Error: POST"
}
```


Indexes:

1. $\{created\_at : -1\}$: this index support log rotation

4.3 Redis Data Structures

4.3.1 Article Caching

When an article appears in a user's News Feed, there's a probability that the user will want to read it, so the article is fully cached for quick access. This with a ttl of 10 minutes.

4.3.2 User caching

The user comes fully cached for fast access especially for his reading preferences. This with an infinite ttl.

Chapter 5

Data Ingestion and Integration

The **Smart News Aggregator** relies on real-time and scheduled data ingestion from multiple external news APIs. This chapter details the data pipeline architecture, API integration strategies, preprocessing steps, and error handling mechanisms to ensure a consistent and reliable flow of news articles into the system.

5.1 External API Integration

The system integrates with the following news providers:

API	Description	Rate Limits	Data Forma
MediaStack	News articles from 7,500+ sources	500 requests/month	JSON
CurrentsApi	-	x requests/month	JSON
Gnews	-	x requests/month	JSON
MarketAux	-	x requests/month	JSON
NYTimes	Premium news content	4000 requests/month	JSON
News Api	-	x requests/month	JSON
News Data	Global news coverage	100 requests/month	JSON
Space Flight News Api	-	x requests/month	JSON
The Guardian	High-quality journalism	5,000 requests/month	JSON

Table 5.1: External Api Integration

Each API uses API keys stored securely in environment variables. Scheduled cron job triggers API calls every monday at 00:00. Requests are made via requests. Re-

sponse Handling: On success, Normalize and store in MongoDB. On failure, Log error, retry with backoff (max 3 attempts).

5.2 Data Processing

5.2.1 Data Validation

- **Duplicate Detection**: checks *externalid* and mongodb index $\{external\_api : 1, external\_id : 1, title : 1\}$ (Compound, Unique)
- **Schema Validation**: Ensures required field ($title$, *url published_at*)

5.2.2 Data Normalization

Standardized Schema

5.3 Error Handling and Logging

For error type is **rate limit exceeded**, retry the next week. For **network failure**, exponential backoff (max 3 retries). And for **malformed data**, skips invalid entries.

Chapter 6

Backend Implementation

This backend of the **Smart News Aggregator** is built using **Flask** (Python) with **Flask-RESTX** for API development.

6.1 System Architecture

6.1.1 Key Components

Component	Technology	Purpose
API Server	Flask [3, 9] + Gunicor	HTTP request handling
Auth	JWT(RS256) [1]	User authentication
Caching	Redis	Session/store hot data
Asybc Tasks	Flask Cron	Background jobs (API fetching)
Docs	Swagger UI [10]	Interactive API documentation

6.2 API Endpoints

6.2.1 Authentication

Endpoint	Method	Description
$/auth/register$	POST	User registration
$/auth/login$	POST	JWT token generation
$/auth/login-alt$	POST	JWT token generation without password validation
$/auth/change\_password$	POST	Update user password
$/auth/me$	GET	Get current user information like email, name, status, role

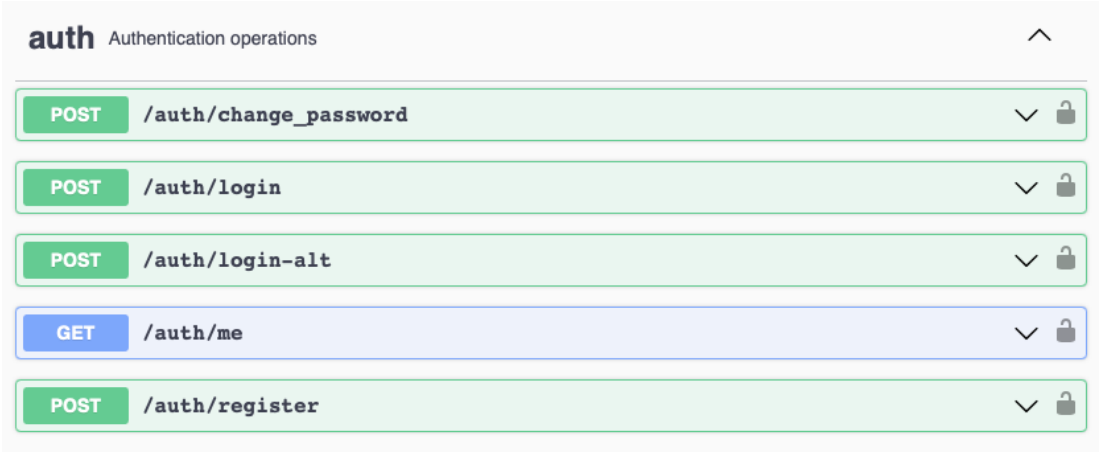

Figure 6.1: API Endpoint: Authentication endpoint

6.2.2 User Management

Endpoint	Method	Description
$/user/article/preference$	POST	Add article preference tags for current user
$/user/article/preference$	GET	Get article preference tags for current user
$/user/me$	POST	Update current user information
$/user/me$	GET	Get current user information

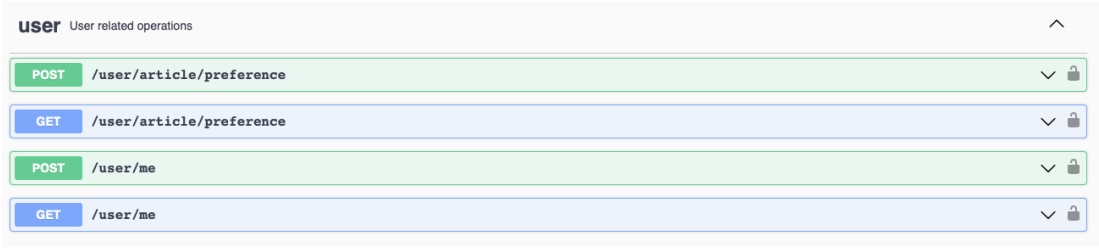

Figure 6.2: API Endpoint: User endpoint

6.2.3 Article Management

Endpoint	Method	Description
/article/comment/me	GET	List of comment make by current user
/article/history	GET	List of read articles
/article/latest	GET	List of all articles order by recent published
/article/search	GET	List of article where title or description content query
/article/tags	POST	-
/article/tags	GET	-
/article/{article_id}	GET	Get all article data
/article/{article_id}/comment	POST	Add comment on article
/article/{article_id}/comment	GET	List of comment for this specific article
/article/{article_id}/comment/{comment_id}	GET	Get all comment data
/article/{article_id}/comment/{comment_id}	DELETE	Delete specific comment
/article/{article_id}/comment/{comment_id}/interaction	POST	add interaction in this comment
/article/{article_id}/comment/{comment_id}/interaction	GET	Get interaction information in this comment
/article/{article_id}/interaction	POST	Add interaction in this article
/article/{article_id}/interaction	GET	Get article interaction information
/article/{article_id}/summary	GET	Get summable details of this article

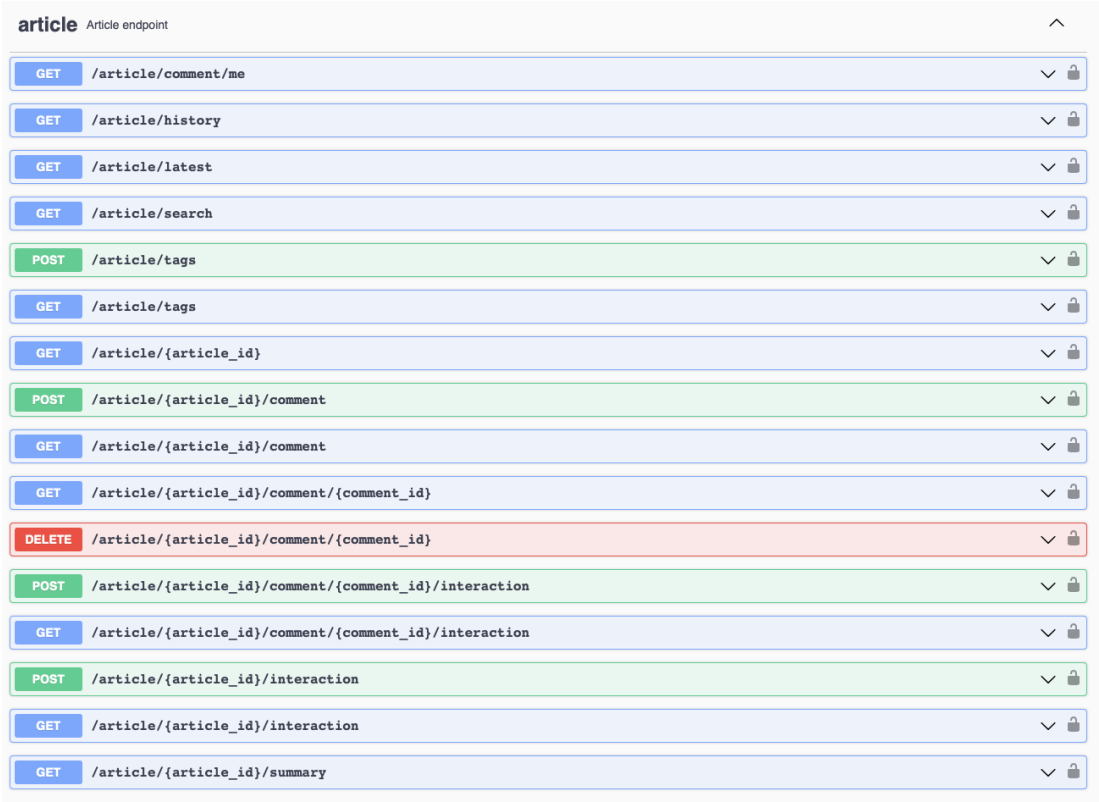

Figure 6.3: API Endpoint: Articles endpoint

6.2.4 Administration Management

Endpoint	Method	Description
/admin/article/{article_id}	DELETE	Delete Article
/admin/articles	GET	List of all articles
/admin/dashboard/errors	GET	List of log of errors occur in server during execution
/admin/dashboard/errors/{server_error_log_id}	DELETE	delete server error log
/admin/users	GET	List of all users
/admin/user/{user_id}	GET	Get all user data
/admin/user/{user_id}	PUT	Add user
/admin/user/{user_id}	POST	Update user
/admin/user/{user_id}	DELETE	Delete User
/admin/	GET	-

Figure 6.4: API Endpoint: Admin endpoint

6.3 Security Implementation

6.3.1 JWT Authentication

Algorithm RS256 (asymmetric) for token and **validation middleware**:

```
def tokenrequired(f):
    @wraps(f)
    def decoratedargs, **kwargs):token = None Extract token from
        Authorization headerauth_header =
        request.headers.get('Authorization','')if auth_header.startswith("Bearer") : token =
        auth_header.split("")[1]if not token : return'error' :' Token is missing!',401try : user =
        TokenManager.decode_token(token = token)if'admin'in request.url and user.role! ='
        admin' :
        raiseUnauthorizedException('You are not authorized to perform this operation')g.user =
        userexceptTokenExceptionase :
        return'error' : str(e),401return f(*args,**kwargs)returndecorated
```

Listing 6.1: Decoration 'Token Required'

6.3.2 Input Validation

All input data is validated via flask-rest's marshal service, which ensures the integrity of input data.

```
@nsarticle.route('\<string:articleid>\interaction')
@nsarticle.param('articleid', 'The article ID')
class ArticleInteractionResource(Resource):
    @tokenrequired
    @nsarticle.expect(ArticleInteractionType.tomodel(namespace=nsarticle))
    @nsarticle.marshalwith(Model.getmessageresponsemodel(namespace=nsarticle))
    def post(self, articleid):
        usertoken: UserToken = g.user


        data = request.getjson()
        interaction = ArticleInteractionType*data)result =
            UserArticleInteractionModel.update_interaction(user_token, interaction =
            interaction, article_id = article_id)if result :
            return"success" : True,"message" : "Interactionupdated"return"success" : False,"message" : "Inter
```

Listing 6.2: Input validation with marshal

6.4 Performance Optimizations

6.4.1 Caching Strategies

Cache Key	Data	TTL
user : *user_id*	Full user json	-
article : *article_id*	-	-
article : *last* : *count*	all last articles count	-
article : *last* : *user_id* : *count*	all last articles count with user preferences	-
article : *last* : *page* : *limit*	full list last articles json by page and limit	-
article : *all* : *count*	all articles count	-
article : *all* : *page* : *limit*	full list articles json by page and limit	-
comment : *all* : *filter* : *page* : *limit*	full list comment by article, page and limit	-

6.4.2 Async Task Processing

When a user accesses a list of items via services such as latest, history, each item in the list is cached.

```
class ArticleModel(ArticleSummaryModel):
    @classmethod
    def cachearticles(cls, usertoken: UserToken, articles: list):
        for article in articles:
              = cls.get(usertoken, str(article.articleid))

    @classmethod
    def cachearticles(cls, usertoken: UserToken, articles: list):
        thread = Thread(target=cls.cachearticles, args=(usertoken,
            articles,))
        thread.daemon = True
        thread.start()
```

Listing 6.3: Async Cache Article

6.5 API Documentation

My api's documentation is auto-generated using flask-restx namesapces and swagger accessible via swagger ui via endpoint /api/docs.

```
class ArticleSummaryModel(MongoDBBaseModel):
    @staticmethod
    def tomodel(namespace: Namespace):
        return namespace.model('ArticleSummaryModel', {
            'articleid': fields.String(required=False),
            'externid': fields.String(required=False),
            'externapi': fields.String(required=True),
            'title': fields.String(required=True),
            'description': fields.String(required=True),
            'author':
                fields.Nested(ArticleSourceModel.tomodel(namespace)),
            'source':
                fields.Nested(ArticleSourceModel.tomodel(namespace)),
            'imageurl': fields.String(required=True),
            'publishedat': fields.String(required=True),
```

```
            'tags': fields.List(fields.String, description="List of tags"),
            'currentuserinteraction':
                fields.Nested(ArticleInteractionStatus.tomodel(namespace)),
            'totaluserinteraction':
                fields.Nested(ArticleInteractionStats.tomodel(namespace)),
        })

    @staticmethod
    def tomodellist(namespace: Namespace):
        return namespace.model('ArticleSummaryModelList', {
            'articles':
                fields.List(fields.Nested(ArticleSummaryModel.tomodel(namespace)),),
            'total': fields.Integer,
            'page': fields.Integer,
            'limit': fields.Integer,
            'pageCount': fields.Integer,
        })
```

Listing 6.4: Api Docuentation Article Summary

Chapter 7

Advanced Aggregations and Analytics

7.1 MongoDB Aggregation Framework

7.1.1 Key Aggregation Concepts

- **Pipeline Stages**: Filter ($*match*), Group ($*group*), Sort ($*sort*), Project ($*project*)
- **Operators**: $*sum*, $*avg*, $*max*, $*arrayElementAt*, $*cond*
- **Performance**: Uses indexes, optimized for large datasets

7.1.2 Pipeline: All tags

```
class ArticleModel(ArticleSummaryModel):
    @classmethod
    def getalltags(cls, usertoken, search: str = None):
        tags = cls.getalltags()
        if tags:
            return tags

        apilogger = ApiLogger(f"[MONGODB] [ARTICLE TAGS] [GET ALL] :
            search = {search}")
        if search:
            pipeline = [
                {"$unwind": "$tags"},
                {"$match": {"tags": {"$regex": f" {search}", "$options":
                    "i"}}},
```

```
                {"$group": {"_id": None, "matchedTags": {"$addToSet":
                    "$tags"}}},
                {"$project": {"_id": 0, "matchedTags": 1}}
            ]
        else:
            pipeline = [
                {"$unwind": "$tags"},
                {"$group": {"_id": None, "matchedTags": {"$addToSet":
                    "$tags"}}},
                {"$project": {"_id": 0, "matchedTags": 1}}
            ]

        with MONGOQUERYTIME.time():
            data = cls.collection().aggregate(pipeline)

        result = list(data)
        tags = result[0]['matchedTags'] if result else []

        apilogger.printlog()

        cls.cachealltags(tags)

        return tags
```

Listing 7.1: Pipeline All Tags

7.1.3 Pipeline: Search Articles

```
class ArticleSearchModel(DataBaseModel):
    @classmethod
    def searcharticles(cls, usertoken: UserToken, query: str, page: int =
        1, limit: int = 10):
        if not query:
            return ArticleModel.lastarticles(usertoken, page=page,
                limit=limit)

        apilogger = ApiLogger(f"[MONGODB] [ARTICLE LATEST] [GET] :
            query={query}, page={page} and limit={limit}")

```

```
        pipeline = [
            {
                "$match": {
                    "$text": {
                        "$search": query,
                        # "$language": "english"
                    }
                }
            },
            {
                "$project": {
                    'articleid': '$id',
                    'externapi': 1,
                    'externid': 1,
                    "title": 1,
                    "description": 1,
                    'source': 1,
                    'author': 1,
                    "score": {"$meta": "textScore"},
                    "publishedat": 1
                }
            },
            {
                "$sort": {"score": -1, "publishedat": -1}  # Relevance +
                    recency
            },
            {
                "$skip": (page - 1) * limit
            },
            {
                "$limit": limit
            }
        ]

        with MONGOQUERYTIME.time():
            results = ArticleModel.collection().aggregate(pipeline)

        if results is None:
            apilogger.printerror("Error occurred during article search")
```

```
47            return []
48
49        apilogger.printlog()
50        return [cls*result) for result in list(results)]
```

Listing 7.2: Pipeline search articles

7.1.4 Pipeline: Article Stats Comment

```
class ArticleCommentStats(DataBaseModel):
    @classmethod
    def getstats(cls, articleid: str, commentid: str = None):
        statslist = cls.getstats(articleid, commentid)
        if statslist is not None:
            return statslist

        apilogger = ApiLogger(f"[MONGODB] [ARTICLE] [MOST COMMENT] :
            article={articleid} and comment={commentid}")

        pipeline = [
            {
                '$addFields': {
                    'articleObjectId': { '$toObjectId': '$articleid' }
                }
            }, {
                '$group': {
                    '_id': '$articleObjectId',
                    'commentcount': { '$sum': 1 }
                }
            }, {
                '$sort': {
                    'commentcount': -1
                }
            }, {
                '$limit': 10
            }, {
                '$lookup': {
                    'from': 'articles',
                    'localField': '_id',
```

```
                    'foreignField': '_id',
                    'as': 'article'
                }
            }, {
                '$unwind': '$article'
            }, {
                '$project': {
                    'articleid': '$_id',
                    'externapi': '$article.externapi',
                    'title': '$article.title',
                    'source': '$article.source',
                    'author': '$article.author',
                    'publishedat': '$article.publishedat',
                    'commentcount': 1,
                    '_id': 0
                }
            }
        ]

        with MONGOQUERYTIME.time():
            stats = CommentModel.collection().aggregate(pipeline)
        if stats is None:
            apilogger.printerror("Error during retrieving statistics")
            return []
        apilogger.printlog()
        statslist = [cls*stat) for stat in
            list(stats)]cls._cache(statslist,articleid,commentid)returnstatslist
```

Listing 7.3: Pipeline Article Stats Comment

7.1.5 Pipeline User Comments with article

```
class CommentDetailsModel(CommentModel):
    @classmethod
    def getusercommentswitharticle(cls, usertoken: UserToken, userid:
        str = None, page: int = 1, limit: int = 10):
        extrafilter = {}

        if userid is None:
```

```
            userid = str(usertoken.userid)

        pipeline = [
            {
                "$match": {
                    "userid": userid
                }
            },
            {
                "$sort": {
                    "createdat": -1
                }
            },
            {
                "$skip": (page - 1) * limit
            },
            {
                "$limit": limit
            },
            {
                "$addFields": {
                    "articleidobj": {
                        "$toObjectId": "$articleid"  # Convert string to
                            ObjectId
                    }
                }
            },
            {
                "$lookup": {
                    "from": "articles",
                    "localField": "articleidobj",
                    "foreignField": "_id",
                    "as": "article"
                }
            },
            {
                "$unwind": "$article"
            },
            {
```

```
                "$project": {
                    "_id": 1,
                    "userid": 1,
                    "author": 1,
                    "articleid": 1,
                    "commentfk": 1,
                    "content": 1,
                    "createdat": 1,
                    "updatedat": 1,
                    "articleinfo": {
                        "externapi": "$article.externapi",
                        "title": "$article.title",
                        "description": "$article.description",
                        "author": "$article.author",
                        "source": "$article.source",
                        "publishedat": "$article.publishedat"
                    }
                }
            }
        ]

        apilogger = ApiLogger(f"[MONGODB] [COMMENT] [GET BY USER] :
            userid={userid}, page={page} and limit={limit}")

        with MONGOQUERYTIME.time():
            results = cls.collection().aggregate(pipeline)

        apilogger.printlog()

        if results:
            return [cls*result) for result in results]return []
```

Listing 7.4: Pipeline User Comments with article

7.1.6 Pipeline User Article Interaction by article

```
class ArticleInteractionStats(DataBaseModel):
    @classmethod
    def getstats(cls, articleid: str, commentid: str = None):
```

```
        apilogger = ApiLogger(f"[MONGODB] [USER ARTICLE INTERACTION] [GET
            STAT] : article={articleid} and comment={commentid}")

        match = {"articleid": articleid}   ({"commentid": commentid} if
            commentid else {})
        pipeline = [
            {"$match": match},
            {
                "$group": {
                    "_id": "$articleid",
                    "liked": {"$sum": {"$cond": ["$liked", 1, 0]}},
                    "saved": {"$sum": {"$cond": ["$saved", 1, 0]}},
                    "shared": {"$sum": {"$cond": ["$shared", 1, 0]}},
                    "report": {"$sum": {"$cond": ["$report", 1, 0]}}
                }
            }
        ]

        with MONGOQUERYTIME.time():
            stats = cls.collection().aggregate(pipeline)
        if stats is None:
            apilogger.printerror("Error during retrieving statistics")
            return ArticleInteractionStats()
        apilogger.printlog()
        statslist = list(stats)
        if statslist:
            return ArticleInteractionStats(
                liked=statslist[0]["liked"],
                saved=statslist[0]["saved"],
                shared=statslist[0]["shared"],
                report=statslist[0]["report"],
            )
        return ArticleInteractionStats()
```

Listing 7.5: Pipeline User Article Interaction by article

7.1.7 Pipeline Most Interacted Articles

```
class ArticleInteractionDashboard(DataBaseModel):
```

```
    @classmethod
    def getmostinteractedarticles(cls, datecheck = None):
        apilogger = ApiLogger(f"[MONGODB] [USER ARTICLE INTERACTION]
            [DASHBOARD] [MOST INTERACTED ARTICLES] ")

        if datecheck:
            pipeline = [
                {
                    '$match': {
                        'updatedat': {'$gte': datecheck}
                    }
                }, {
                    '$addFields': {
                        'articleObjectId': {'$toObjectId': '$articleid'}
                    }
                }, {
                    '$group': {
                        '_id': '$articleObjectId',
                        'readcount': {'$sum': {'$cond': [{'$ifNull':
                            ['$readat', False]}, 1, 0]}},
                        'likecount': {'$sum': {'$cond': ['$liked', 1, 0]}},
                        'savecount': {'$sum': {'$cond': ['$saved', 1, 0]}},
                        'sharecount': {'$sum': {'$cond': ['$shared', 1,
                            0]}}
                    }
                }, {
                    '$addFields': {
                        'totalinteractions': {
                            '$add': ['$readcount', '$likecount',
                                '$savecount', '$sharecount']
                        }
                    }
                }, {
                    '$sort': {
                        'totalinteractions': -1
                    }
                }, {
                    '$limit': 5
                }, {
```

```
                '$lookup': {
                    'from': 'articles',
                    'localField': '_id',
                    'foreignField': '_id',
                    'as': 'article'
                }
            }, {
                '$unwind': '$article'
            }, {
                '$project': {
                    'articleid': '$_id',
                    'externapi': '$article.externapi',
                    'title': '$article.title',
                    'publishedat': '$article.publishedat',
                    'author': '$article.author',
                    'source': '$article.source',
                    'totalinteractions': 1,
                    'readcount': 1,
                    'likecount': 1,
                    'savecount': 1,
                    'sharecount': 1,
                    '_id': 0
                }
            }
        ]
    else:
        pipeline = [
            {
                '$addFields': {
                    'articleObjectId': {'$toObjectId': '$articleid'}
                }
            }, {
                '$group': {
                    '_id': '$articleObjectId',
                    'readcount': {'$sum': {'$cond': [{'$ifNull':
                        ['$readat', False]}, 1, 0]}},
                    'likecount': {'$sum': {'$cond': ['$liked', 1, 0]}},
                    'savecount': {'$sum': {'$cond': ['$saved', 1, 0]}},
```

```
                'sharecount': {'$sum': {'$cond': ['$shared', 1,
                    0]}}
            }
        }, {
            '$addFields': {
                'totalinteractions': {'$add': ['$readcount',
                    '$likecount', '$savecount', '$sharecount']}
            }
        }, {
            '$sort': {
                'totalinteractions': -1
            }
        }, {
            '$limit': 5
        }, {
            '$lookup': {
                'from': 'articles',
                'localField': '_id',
                'foreignField': '_id',
                'as': 'article'
            }
        }, {
            '$unwind': '$article'
        }, {
            '$project': {
                'articleid': '$_id',
                'externapi': '$article.externapi',
                'title': '$article.title',
                'publishedat': '$article.publishedat',
                'author': '$article.author',
                'source': '$article.source',
                'totalinteractions': 1,
                'readcount': 1,
                'likecount': 1,
                'savecount': 1,
                'sharecount': 1,
                '_id': 0
            }
        }
```

```
            ]

        with MONGOQUERYTIME.time():
            stats =
                UserArticleInteractionModel.collection().aggregate(pipeline)
        if stats is None:
            apilogger.printerror("Error during retrieving statistics")

        statlist = list(stats)
        apilogger.printlog()
        # print(statlist)

        for stat in statlist:
            stat['articleid'] = str(stat['articleid'])

        return [cls*data) for data in stat_list]
```

Listing 7.6: Pipeline Most Interacted Articles

7.1.8 Pipeline User Preferences Most Used Tags

```
class UserPreferencesDashboard(DataBaseModel):
    @classmethod
    def getmosttags(cls, limit: int = 5):
        apilogger = ApiLogger(f"[MONGODB] [USER TAGS] [DASHBOARD] [MOST
            TAGS IN PREFERENCES] : limit={limit}")

        pipeline = [
            {
                '$match': {
                    'preferences': {'$exists': True, '$ne': []}
                }
            }, {
                '$unwind': '$preferences'
            }, {
                '$group': {
                    '_id': '$preferences',
                    'count': {'$sum': 1}
                }
```

```
            }, {
                '$sort': {
                    'count': -1
                }
            }, {
                '$limit': limit
            }, {
                '$project': {
                    'tag': '$id',
                    'count': 1,
                    'id': 0
                }
            }
        ]

        with MONGOQUERYTIME.time():
            stats = cls.collection().aggregate(pipeline)
        if stats is None:
            apilogger.printerror("Error during retrieving statistics")

        statlist = list(stats)
        apilogger.printlog()

        return [cls*data) for data in stat_list]
```

Listing 7.7: Pipeline User Preferences Most Used Tags

7.2 Performance Optimization

To optimize these aggregation operations, different indexes have been drawn and the aggregation results cached for an average of 1 hour.

- **All Tags**7.1: $\{tags : 1\}$ (1 hour)
- **Search Articles**7.2: $\{title : "text"\}$, $\{description : "text"\}$ and $\{title : "text", description : "text"\}$
- **Article Stats Comment**7.3: $\{article\_id : 1\}$ (10 minutes)

- **User Comments with article**7.4: $\{user\_id : 1\}$ and $\{user\_id : 1, created\_at : -1\}$
- **User Article Interaction by article**7.5: $\{article\_id : 1\}$
- **Most Interacted Articles**7.6: $\{article\_id : 1\}$ and $\{article\_id : 1, updated\_at : -1\}$
- **User Preferences Most Used Tags**7.7: $\{preferences : 1\}$

Chapter 8

Deployment and Scaling

This chapter covers containerized deployment for the Smart News Aggregator.

8.1 Containerization with Docker

8.1.1 Core Services Setup

```
global:
  scrapeinterval: 15s

scrapeconfigs:
  - jobname: 'smart-news-aggregator-api'
    staticconfigs:
      - targets: ['smart-news-aggregator-api:5050']
      # - targets: ['localhost:5050']

  - jobname: 'redis'
    staticconfigs:
      - targets: ['redis-exporter:9121']

  - jobname: 'mongodb'
    staticconfigs:
      - targets: ['mongodb-exporter:9216']
```

Listing 8.1: Prometheus Configuration (prometheus.yml)

```
services:
  redis:
```

```
    image: redis:alpine
    ports:
      - "6379:6379"
    volumes:
      - redisdata:/data
      - redisbackup:/backup
    command: redis-server --save 60 1 --loglevel warning
    restart: unless-stopped

  redis-exporter:
    image: oliver006/redisexporter
    ports:
      - "9121:9121"
    environment:
      REDISADDR: "redis://host.docker.internal:6379"

  mongodb:
    image: mongo:latest
    containername: mongodb
    ports:
      - "27017:27017"
    volumes:
      - mongodbdata:/data/db
      - mongodbbackup:/backup
    platform: linux/arm64
    healthcheck:
      test: echo 'db.runCommand("ping").ok' | mongosh localhost:27017/test
          --quiet
      interval: 10s
      timeout: 10s
      retries: 5
    restart: unless-stopped

  mongodb-exporter:
    image: bitnami/mongodb-exporter:0.40.0
    ports:
      - "9216:9216"
    environment:
      MONGODBURI: "mongodb://host.docker.internal:27017"
```

```
41
42    prometheus:
43      image: prom/prometheus
44      ports:
45        - "9090:9090"
46      volumes:
47        - ./prometheus.yml:/etc/prometheus/prometheus.yml
48
49    grafana:
50      image: grafana/grafana
51      ports:
52        - "3000:3000"
53      volumes:
54        - grafanadata:/var/lib/grafana
55      dependson:
56        - prometheus
57
58  volumes:
59    redisdata:
60    redisbackup:
61    mongodbdata:
62    mongodbbackup:
63    grafanadata:
```

Listing 8.2: Docker compose for core services (docker-compose.yml)

8.1.2 Backend Setup

```
1  # Dockerfile.prod
2  FROM python:3.11-slim
3
4  WORKDIR /app
5
6  COPY requirements.txt .
7  RUN pip install --no-cache-dir -r requirements.txt
8
9  COPY . .
10
11 ENV FLASKAPP=src/app.py
```

```
ENV FLASKENV=production

EXPOSE 5000

# Use Gunicorn for production
CMD ["gunicorn", "-b", "0.0.0.0:5000", "src.app:application"]
```

Listing 8.3: Dockerfile Backend (Production configuration)

```
services:
  smart-news-aggregator-api:
    build:
      context: .
      dockerfile: Dockerfile.prod
    ports:
      - "5050:5000"
    environment:
      - FLASKENV=production
      - FLASKENVFILE=.env.prod
    envfile:
      - .env.prod
```

Listing 8.4: Docker Compose Backend (docker-compose.yml)

8.1.3 Frontend Setup

```
# Stage 1: Build the Vite app
FROM node:20 as builder

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

# Stage 2: Serve with Nginx
FROM nginx:stable-alpine
```

```
14
15 # Copy the build output to Nginx web root
16 COPY --from=builder /app/dist /usr/share/nginx/html
17
18 # Optional: custom nginx config for single-page app (SPA)
19 COPY nginx.conf /etc/nginx/conf.d/default.conf
20
21 EXPOSE 80
22
23 CMD ["nginx", "-g", "daemon off;"]
```

Listing 8.5: Dockerfile Frontend (Production configuration)

```
1 services:
2   smart-news-aggregator-fe:
3     build:
4       context: .
5       dockerfile: Dockerfile.prod
6     ports:
7       - "3000:80" # host:container
```

Listing 8.6: Docker Compose Frontend (docker-compose.yml)

```
```

Listing 8.7: (.yml)

8.2 Testing and Monitoring

To ensure the reliability and correctness of the platform, the services were thoroughly tested and monitored after deployment. All backend and frontend services were containerized using Docker, enabling isolated and consistent environments across development and production.

Once deployed, the services were interacted with directly through the exposed endpoints to verify the correct execution of key functionalities such as authentication, article retrieval, comment posting, and admin operations (e.g., deleting users [8] or moderating content). API responses were validated, edge cases were tested, and rate limits were checked.

Monitoring was achieved through an integrated dashboard, where metrics such as service uptime, API response times, error logs, and user activity were visualized. This allowed real-time observation of system behavior and rapid identification of anomalies or performance bottlenecks. Container logs were also inspected using docker logs and integration with monitoring tools like **Grafana** and **Prometheus**.

8.3 Deployment Architecture Across Three Machines

The system has been deployed across three virtual machines within the same private network. Each machine hosts a combination of services to ensure high availability, efficient resource distribution, and resilience.

For this deployment, I have this configuration:

Machine	IP Address	Services
Machine 1	10.1.1.17	MongoDB (Primary) + Redis Replica (Slave)
Machine 2	10.1.1.18	Redis Master + MongoDB (Slave)
Machine 3	10.1.1.19	Flask Backend API + React Frontend (Vite + Nginx) + Redis Sentinel (Monitoring)

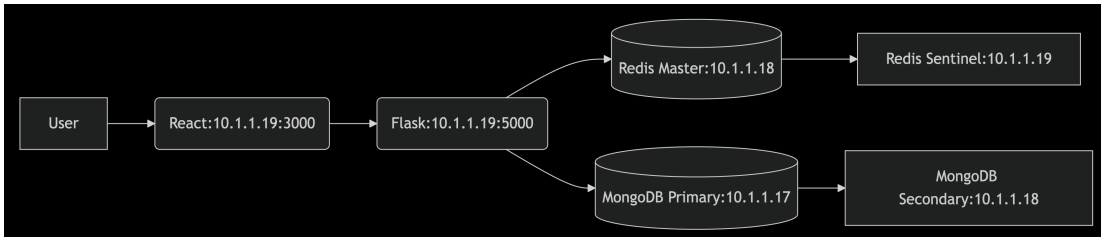

Figure 8.1: Deployment Architecture Accross Three Machines

8.3.1 MongoDB Replicat Set

MongoDB is deployed as a replica set across Machines 1 and 2.

- Primary (10.1.1.17) handles all write operations.
- Secondary (10.1.1.18) replicates the primary and allows read scalability.
- The backend connects using the replica set URI: $MONGODB\_URI = mongodb://10.1.1.17:27017,10.1.1.18:27017/?replicaSet=rs0$.

8.3.2 Redis Replication + Sentinel

Redis is set up with a master-slave configuration:

- Master on 10.1.1.18.
- Replica (slave) on 10.1.1.17.
- Sentinel on 10.1.1.19 monitors the Redis master and handles automatic failover.
- Now, the backend is directly connected to the master.

8.3.3 Flask Backend

The Flask API backend is hosted on Machine 3 (10.1.1.19). It connects to MongoDB and Redis using internal IPs.

It's launched in production using **Gunicorn**, with the following environment configuration:

```
FLASKAPP=src/app.py
FLASKENV=production
MONGODBURI=mongodb://10.1.1.17:27017,10.1.1.18:27017/?replicaSet=rs0
REDISURL=redis://10.1.1.18:6379
```

Listing 8.8: Flask API Backend Environment

Deployment steps from ./run-flask-prod.sh:

```
#!/bin/bash

# Exit if any command fails
set -e

# Define your project root (adjust this if needed)
PROJECTDIR=$(dirname "$0")

cd "$PROJECTDIR"

# Set environment variables
export FLASKAPP=src/app.py
export FLASKENV=production
export FLASKENVFILE=.env.prod
```

```
15
16 # Clean existing virtual environment
17 rm -rf .venv
18
19 python3.13 -m venv --without-pip .venv
20
21 # Activate virtual environment
22 source .venv/bin/activate
23 curl -sS https://bootstrap.pypa.io/get-pip.py | python
24
25 # Install dependencies
26 pip install --upgrade pip
27 pip install --no-cache-dir -r requirements.txt
28
29 # Run using Gunicorn (you can change workers if needed)
30 gunicorn -b 0.0.0.0:5000 src.app:application
```

Listing 8.9: Flask Backend Deployment Step

8.3.4 React Frontend

The React app is built using **Vite** and served via **Nginx** on port 3000 on Machine 3.

It reads the base API URL at build time via .env:

```
1 VITEBASEURLAPI=http://10.1.1.19:5000
```

Listing 8.10: React Frontend Environment

Deployment steps:

```
1   npm install
2   npm run build
3   npm start
```

Listing 8.11: React Frontend Deployment Step

The frontend app is available locally via ssh on

```
1 ssh -N -L 3000:localhost:3000 root@10.1.1.19
```

Listing 8.12: React Frontend Access via SSH Command

Chapter 9

Conclusion

9.1 Project Achievements

This project successfully designed and implemented a **Smart News Aggregator & User Personalization Platform**, leveraging modern full-stack technologies including **Flask**, **MongoDB**, **Redis**, **React**, and **Docker**. The platform allows users to browse, search, and interact with a large volume of news articles in real time while supporting advanced features such as:

- User account and preference management
- Article search and filtering based on user interests
- Commenting, liking, sharing, and saving interactions
- A full-featured admin dashboard for user, article, and error management
- Efficient use of **MongoDB** for document storage and **Redis** for caching frequently accessed data

Several **core achievements** of the project include:

- **Scalable API architecture** supporting multiple concurrent users with JWT-based authentication
- **Real-time aggregation** of user interactions (comments, likes, etc.) using MongoDB aggregation pipelines

- **Dynamic and responsive React UI** with pagination, filtering, and error handling
- **Deployment automation with Docker**, enabling isolated and reproducible environments
- **Monitoring and logging mechanisms** to observe platform behavior and debug failures efficiently

9.2 Challenges Overcome

During development, several technical and architectural challenges were encountered and addressed:

- **CORS preflight request issues** when interacting across frontend and backend containers were resolved through proper flask-cors configuration
- Handling **inconsistent data types** (e.g., ObjectId vs string) in MongoDB required careful pipeline design and casting
- Designing **efficient indexes** in MongoDB to optimize aggregation performance, especially for high-traffic collections like comments and interactions
- Implementing **data parsing and transformation** for various date/time formats using pydantic, datetime, and external APIs
- Managing **rate limits** and error logging during external API interactions with fallback and retry mechanisms

These challenges provided valuable insights into working with real-world data, full-stack integration, and production-level deployment.

9.3 Future Work

While the platform already demonstrates a solid foundation, several opportunities for enhancement exist:

- **GraphQL API Layer**: Flexible frontend data fetching
- **Advanced recommendation engine** using collaborative or content-based filtering to suggest articles based on user behavior and preferences
- Integration with **machine learning models** for topic classification, sentiment analysis, or fake news detection
- **Mobile app version** of the platform using React Native or Flutter to increase accessibility
- **Newsletter Automation**: Implementation of **email notification systems** to alert users of trending articles or replies to their comments
- **Role-based access control** with more granular permissions for moderators and administrators
- **PDF Article Export**: Enhanced accessibility
- Full integration with **monitoring tools** like Prometheus, Grafana, and ELK Stack for real-time performance and alerting
- **CI/CD pipeline setup** for automated deployment, testing, and delivery across environments
- Extension of data sources by connecting to more external news APIs and supporting multilingual content

Bibliography

[1] Auth0: JWT.IO - JSON Web Tokens (2024), https://jwt.io/introduction

[2] Colvin, S.: Pydantic documentation (2024), https://docs.pydantic.dev

[3] Community, P.R.: Flask-restx documentation. https://flask-restx.readthedocs.io/ (2024)

[4] (formerly Facebook), M.: React – A JavaScript Library for Building User Interfaces (2024), https://react.dev

[5] Inc., D.: Docker Documentation (2024), https://docs.docker.com

[6] Inc., M.: MongoDB Manual (2024), https://www.mongodb.com/docs/manual/

[7] Labs, R.: Redis Documentation (2024), https://redis.io/docs

[8] RandomUser.me: Random User Generator API (2024), https://randomuser.me/documentation

[9] Ronacher, A.: Flask Documentation (2024), https://flask.palletsprojects.com

[10] Software, S.: Swagger: Openapi specification. https://swagger.io/specification/ (2024)

[11] Sullivan, D.: NoSQL for Mere Mortals. Addison-Wesley (2015)

[12] You, E.: Vite – Next Generation Frontend Tooling (2024), https://vitejs.dev

Acknowledgments

First and foremost, I would like to express my deepest gratitude to **God**, whose grace, strength, and guidance sustained me throughout the duration of this project.

I sincerely thank all the **professors and academic staff** who provided the foundation, mentorship, and encouragement that made this work possible. Your knowledge and dedication have been invaluable.

I would also like to acknowledge the **authors of NoSQL for Mere Mortals** for their clear explanations and practical approach, which greatly enhanced my understanding of NoSQL data modeling.

My heartfelt thanks to the developers and contributors of the various **tools and frameworks** used in this project — including **Flask**, **React**, **MongoDB**, **Redis**, and **Docker** — as well as the extensive and well-maintained **documentation** that enabled effective development.

Special thanks to **DeepL**, which helped with accurate and nuanced **translations**, and to **ChatGPT** and **DeepSeek**, whose assistance in technical guidance, code generation, and documentation structuring significantly accelerated the progress and quality of this work.

I extend my heartfelt thanks to my friends for their moral support, encouragement, and constant motivation. Their presence has been a great source of strength.

Finally, I thank my girlfriend for her patience, love, and unwavering support throughout this journey.

Finally, I am grateful to all those who supported me, directly or indirectly, in the successful completion of this project.