
UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

# COMPUTER ENGINEERING MASTER'S DEGREE

## Master's course of Software Platforms

Documentation of the group project

Andrea Bruttomesso and Davide Seghetto

August 2024

# 1 Introduction

This project was developed by two students enrolled in the Computer Engineering Master's program, specifically within the Software Platforms course at the University of Padova. In today's world, characterized by an ever-growing influx of online content, the need for advanced tools to monitor and analyze public discussions on relevant topics has become increasingly important. This project aims to develop an innovative platform for monitoring and analyzing articles from online newspapers, utilizing a microservices architecture.

The platform is tailored to meet the needs of expert users, such as journalists and sociologists, who wish to explore and understand the themes discussed in the media regarding specific topics of interest. The project is divided into two main phases. The first phase involves monitoring and collecting articles related to a particular topic of interest, such as *Artificial Intelligence*, through a targeted search query. These articles are subsequently stored in a database.

The second phase focuses on analyzing the themes discussed within a subset of the collected articles. In this phase, users can specify an additional query, such as *ChatGPT*, to extract and analyze the primary themes covered in articles that match this query within the initial corpus. The platform then provides representations of these emerging themes, helping users to understand prevailing trends and discussions in the media.

Specifically, newspaper articles are retrieved from The Guardian API, stored in MongoDB, made searchable via Elasticsearch, and analyzed using Mallet for topic modeling. To ensure scalability and modularity, the platform is structured into several services, each running within Docker containers. Users interact with the platform through HTTP requests facilitated by a user-friendly UI, making the system accessible and efficient.

The following sections will delve into the implementation choices that guided the development of this platform, leading to the final product.

# 2 Design

The application is based on the microservices architecture. It is an architectural pattern that arranges an application as a collection of loosely coupled services, communicating through lightweight protocols, in this case, HTTP. The primary services involved are the frontend service, the backend (or monitoring) service, two Elasticsearch-based services for indexing and searching, the topic modeling service that uses Mallet, and MongoDB for data storage.
We will now discuss more in depth the application architecture and its control flow.

## 2.1 Overall architecture

The following schema shows the overall architecture of our application.

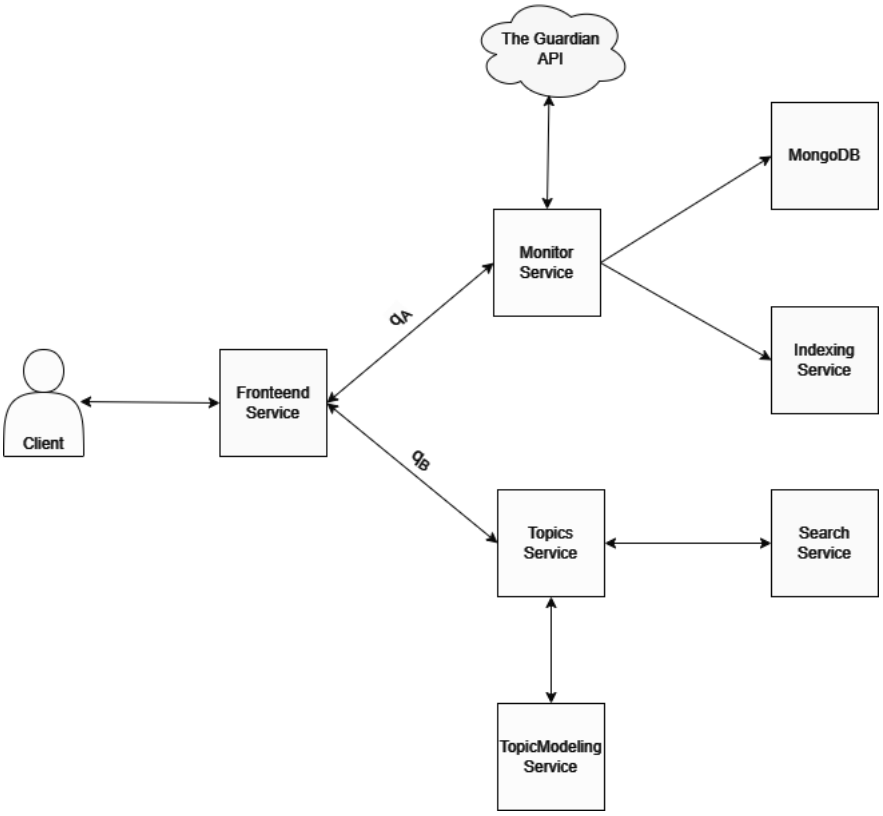


Figure 1: Overall architecture of our microservices based application.

## 2.2 Sequence diagram

Let's see now more in depth, thanks to the following sequence diagram, the complete control flow of the application.

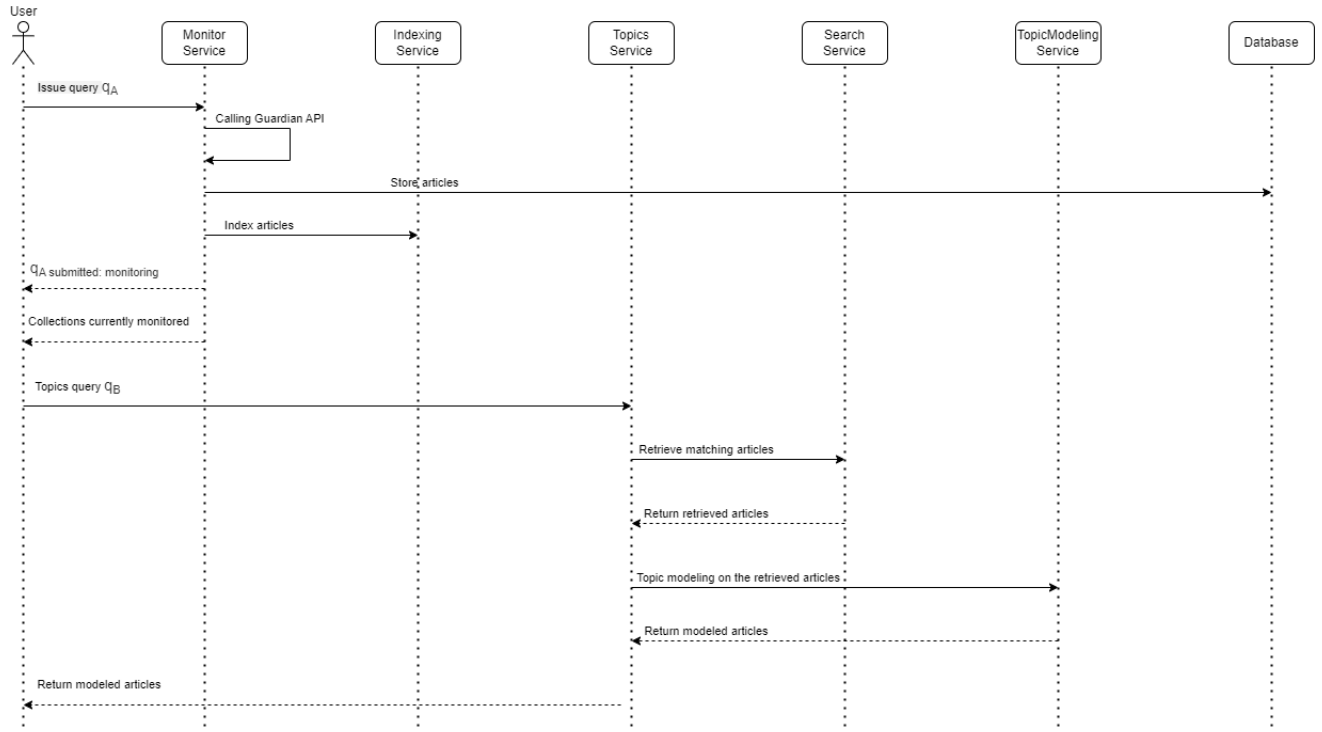


Figure 2: Sequence diagram of our microservices based application.

Through a UI made available by the frontend service, the user submits to the monitor service an initial issue query $q_A$, whose JSON representation is as follows:

```
{
    issueQuery: "artificial intelligence",
    tag: "ai",
    startDate: 01-01-2022,
    endDate: 12-31-2022
}
```

Listing 1: JSON representation of $q_A$

When the monitor service receives the JSON file, it uses the data within it to call "The Guardian" API in order to retrieve the articles that match the issue query. Once the articles are fetched, they are stored in a MongoDB collection, whose name is the same of the *TAG* in Listing 1, and indexed via the indexing service based on Elasticsearch. At this point the user receives an acknowledgment that the issue query was processed successfully. Furthermore, the currently monitored collections are shown on the screen in such a way that the user can select the one for which to perform the second query $q_B$. An example of its JSON representation is the following:

```
{
    query: "ChatGPT",
    numOfTopics: 3,
    numOfTopWords: 5,
    collectionId: "ai"
}
```

Listing 2: JSON representation of $q_B$

The query $q_B$ is submitted to the search service to retrieve relevant documents from the specified collection based on the given query. The service returns all matching articles and forwards them to the MALLET service, which is responsible for extracting the specified number of topics and identifying the top words for each topic that will be shown to the user.

# 3 Design - per service

Let's see now the specific design for each service.

## 3.1 Monitor service

The monitor service relies on the Spring Boot framework. It is used for building Java-based applications, particularly web applications, that follow the principles of the Spring Framework. The main goal of Spring Boot is to simplify the development of new Spring applications by minimizing the amount of boilerplate code and configuration needed [1].
The service is responsible for receiving query $q_A$ and for interfacing with The Guardian API for retrieving the needed articles. In order to make the API call efficient and not to exceed the maximum number of allowed calls (500), we added to the URL the filter "*page-size*" [2]. This filter is used to set the number of articles retrieved in one single API call (the maximum that can be set is 200), allowing us to perform the minimum number of calls which otherwise would be rather large for certain issue query. This also allows us to reduce the total processing time avoiding the overhead generated by each API call. We also added the filter "*show-fields=body*" which makes The Guardian API return the JSON file with the body field too, that is the actual text of each article.
Once the monitor service receives the articles from The Guardian API, it stores them in the MongoDB collection. For each article only few fields are saved, since they are the ones needed for the topic modeling task: *id*, *webTitle* and *body*.
Each article is then sent to the indexing service which uses Elasticsearch and builds an index for each collection named as the recieved *tag*.

## 3.2 APIs

Our application exposes different endpoints to perform various functionalities. In the following subsections, each API endpoint is documented with details on requests and responses.

### 3.2.1 POST /collect

This endpoint accepts a JSON payload with specific parameters, calls The Guardian's API to fetch all articles related to the 'issueQuery' published between 'startDate' and 'endDate', indexes them, and saves them into a MongoDB collection under the name specified by the 'tag'.

**Request**

- **URL:**

  `/collect`

- **Method:** POST

- **Body:**

```
{
  "issueQuery": "string",
  "tag": "string",
  "startDate": "date",
  "endDate": "date"
}
```

**Response**

- **Status Code:** 200 OK
- **Body:**

```
{
  "status": "monitoring"
}
```

- **Status Code:** 500 Internal Server Error
- **Body:**

```
{
  "status": "error",
  "message": "string"
}
```

### 3.2.2 GET /collections

This endpoint returns a list of all collection names currently stored in MongoDB.

**Request**

- **URL:**

```
/collections
```

- **Method:** GET

**Response**

- **Status Code:** 200 OK
- **Body:**

```
{
  "data": [
    {"collection": "string"},
    ...
  ]
}
```

### 3.2.3 DELETE /collections/collectionId

This endpoint deletes a specific collection from MongoDB, if it exists.

**Request**

- **URL:**

```
/collections/{collectionId}
```

- **Method:** DELETE

- **Path Parameter:**
  - **collectionId:** The name of the collection to delete.

**Response**

- **Status Code:** 204 No Content
- **Status Code:** 404 Not Found

#### 3.2.4 POST /topics

This endpoint accepts a JSON payload with parameters related to topics, processes the request, and returns a list of topics and top word for each topic based on the provided data.

**Request**

- **URL:**

  `/topics`

- **Method:** POST
- **Body:**

```
{
  "query": "string",
  "numOfTopics": "int",
  "numOfTopWords": "int",
  "collectionId": "string"
}
```

**Response**

- **Status Code:** 200 OK
- **Body:**

```
{
  "query": "string",
  "topics": [
    {
      "topicId": "int",
      "words": ["string", "string", ...]
    },
    ...
  ]
}
```

#### 3.2.5 Internal API: POST /api/mallet/topics

This internal endpoint processes a list of documents and extracts topics using the Mallet service. It is not exposed to the user.

**Request**

- **URL:**

  `/api/mallet/topics`

- **Method:** POST

- **Body:**

```
[
  "document1",
  "document2",
  ...
]
```

- **Query Parameters:**
  - **numTopics:** The number of topics to extract.
  - **numWords:** The number of words per topic.

**Response**

- **Status Code:** 200 OK

- **Body:**

```
[
  {
    "topicId": "int",
    "words": ["word1", "word2", ...]
  },
  ...
]
```

### 3.3 MongoDB service

MongoDB is a popular NoSQL database that provides a flexible, scalable solution for managing large amounts of unstructured or semi-structured data. It stores data in JSON-like documents, which allows for a flexible schema. This means you can store complex, hierarchical data structures directly within documents, making it easier to work with data that doesn't fit neatly into tables [3].
This service allows us to store the articles in an unstructured way. As described before, when submitting the query $q_A$ the user defines a tag that will be used to decide in which collection to save the retrieved articles. The user can thus decide whether to store all the articles in a single collection or to separate them in different collections.

### 3.4 Indexing and Search services

These services relie on Elasticsearch. It is a powerful, open-source search and analytics engine designed to handle large amounts of data quickly and in near real-time. It is built on top of Apache Lucene and is widely used for searching, analyzing, and visualizing data across a wide range of use cases [4].
The *indexing service* creates and index, if not existing, with the same name as the collection considered and with the following mapping:

```
{
  "mappings": {
    "properties": {
      "webTitle": {
        "type": "text",
        "analyzer": "english"
      },
      "body": {
        "type": "text",
        "analyzer": "english"
      }
    }
  }
}
```

Listing 3: JSON representation of the index mapping

Then each article is insert into the index, the article *id* also identifies it into the index in order to properly manage eventual duplicates.
The *search index* is responsible for accepting a collection, an index name, and a query. It returns all the articles in the collection that meet a minimum score threshold, ensuring that the topic modeling is performed only on relevant documents. The *min_score* is dynamically calculated as $max\_score \times 0.5$, where the *max_score* is determined through an initial search operation. This approach allows us to obtain a sufficient number of relevant results while filtering out non-relevant ones. Also, the searching operation is performed on both *webTitle* and *body* fields giving a weight of 1 to the title and a weight of 3 to the body.

### 3.5 TopicModeling service

This service relies on MALLET (MAchine Learning for LanguagE Toolkit). It is an open-source Java-based toolkit designed for natural language processing (NLP) tasks, particularly focusing on text classification, clustering, topic modeling, and sequence tagging [5].
To prepare the textual data for topic modeling, the service constructs a preprocessing pipeline using a series of transformations on the text. Then the model performs a cycle of *numIterations* iterations in order to find the required topics and top words. This parameter is currently set to 800 in order to have a slower but more precise result but it can be tuned on the user necessities.

# 4 Deployment

The deployment of this application is managed using Docker and Docker Compose [6], which provides an efficient way to define and orchestrate multi-container applications. By using Docker Compose, the entire application can be deployed in a consistent and repeatable manner, making it easy to set up the development and production environments. The application consists of several microservices, each encapsulated within its own Docker container, ensuring modularity, scalability, and ease of deployment.

## 4.1 Configuration of the docker-compose.yaml file

The docker-compose.yml file defines the configuration for all the services involved in the application. Below is a breakdown of each service and its role within the system:

- **MongoDB service 'mongo'**
  - **Purpose**: It is the database used to store the retrieved articles.
  - **Dockerfile**: The MongoDB service uses the official mongo:4.4 image with an additional initialization script copied into the container.
  - **Volumes**: A volume (mongo-data) is created to persist data between container restarts.
  - **Health Check**: A health check is configured to ensure MongoDB is ready to accept connections before dependent services start.
- **Elasticsearch service 'elastic'**
  - **Purpose**: Elasticsearch is used for indexing and making searchable the data retrieved from The Guardian API, providing powerful search and analytics capabilities.
  - **Dockerfile**: The service uses the official elasticsearch:8.13.3 image.
  - **Environment Variables**: The security is disabled (xpack.security.enabled=false), and it's configured to run in a single-node mode (discovery.type=single-node).
  - **Volumes**: A volume (esdata) is created to persist Elasticsearch data.
- **Frontend Service 'frontend'**
  - **Purpose**: This service provides the user interface of the application.
  - **Dockerfile**: The frontend service is built in two stages:
    1. **Build Stage**: The application is compiled and packaged using Maven.
    2. **Run Stage**: The packaged application is then run using OpenJDK 17.
  - **Ports**: The service is exposed on port 3000, mapped to port 8080 inside the container.
- **Monitor Service 'backend'**
  - **Purpose**: The monitor service acts as the core processing unit, interfacing with both MongoDB and Elasticsearch to handle application logic.
  - **Dockerfile**: Similar to the frontend service, the backend is built and then run using OpenJDK 17.

  - **Dependencies**: The service depends on both the MongoDB and Elasticsearch services, ensuring that they are fully initialized before starting.
  - **Environment Variables**: The connection string for MongoDB is passed as an environment variable.

- **MALLET Service 'mallet'**
  - **Purpose**: This service performs the topic modeling task on the interested articles.
  - **Dockerfile**: The MALLET service follows the same multi-stage build process as the other Java-based services.
  - **Dependencies**: The service depends on the Elasticsearch service for storing and retrieving processed data.
  - **Ports**: It is exposed on port 8081, mapped to port 8080 inside the container.

### 4.2 Deployment process

To deploy the application, the following steps are executed:

1. **Build the Services**: Docker Compose builds the images for each service as defined by their respective Dockerfiles. The multi-stage build process ensures that the final image is optimized, containing only the necessary runtime dependencies.

2. **Start the Services**: docker-compose up is used to start all the services in the correct order, ensuring that dependencies are respected. The depends_on directive ensures that services like the backend wait for MongoDB and Elasticsearch to be fully operational before starting.

3. **Health Check**: Health check is in place to monitor the readiness of the mongo service. This ensures that other services do not start or attempt to connect until MongoDB is fully initialized and ready.

4. **Accessing the Application**: The application can be accessed via the exposed ports on the host machine. The frontend service is accessible on port 3000, while the backend services are accessible on ports 8080, 8081 and 9200.

# 5 Testing

Testing was an important part of the development process for this microservices-based application. Following a Test-Driven Development (TDD) approach, we ensured that each feature was rigorously validated through automated tests before implementation. JUnit, a widely-used testing framework in the Java ecosystem, was chosen for writing and executing tests across all services. The use of JUnit greatly enhanced our ability to develop reliable, maintainable, and well-tested code.

## 5.1 Test-Driven Development (TDD) Approach

The TDD approach emphasizes writing tests before writing the actual production code. This methodology offered several benefits throughout the development process:

- **Code Quality**: Writing tests first ensured that the code was written with clear objectives, resulting in a more structured and maintainable codebase.
- **Requirement Validation**: Tests acted as a specification for the desired behavior of the application, ensuring that the implementation matched the functional requirements.
- **Early Bug Detection**: By running tests frequently, we were able to detect and fix bugs early in the development cycle, reducing the time and effort required for debugging later.

### 5.1.1 JUnit

JUnit [7], a powerful testing framework for Java, was central to our testing strategy. Its robust feature set made it ideal for various types of tests. Some of the key benefits and features of JUnit that were leveraged include:

1. **Annotations and Assertions**:
   - JUnit provides a rich set of annotations and assertions that simplify the process of writing and organizing tests.
   - Assertions such as *assertEquals*, *assertTrue*, and *assertNotNull* allowed for concise and readable test cases that clearly expressed the expected outcomes.
   - Example:

```
@Test
    public void testIndexArticle() throws Exception {
        Article article = new Article();
        article.setId("test");
        article.setWebTitle("Test␣Article");
        article.setBody("This␣is␣a␣test␣article␣body");

        indexingService.indexArticle(article, "index");

        List<String> results = searchService.searchArticles("index", "test");

        assertNotNull(results);
        assertEquals(1, results.size());
        assertEquals(article.getBody(), results.get(0));
    }
```

2. **Integration with Build Tools**:

   - JUnit seamlessly integrated with build tools like Maven [8], enabling automated test execution as part of the build process. This integration ensured that tests were run consistently, and any issues were identified immediately, maintaining the integrity of the codebase.
   - The use of Maven allowed us to include JUnit dependencies easily and manage them alongside other project dependencies. The build process was configured to run tests automatically with every build, enforcing a test-first culture within the development team.

3. **Mocking with Mockito**:

   - JUnit's compatibility with Mockito [9], a popular mocking framework, was instrumental in writing unit tests that isolated components from their dependencies.
   - Mockito was used to create mock objects and define their behavior, allowing us to focus on testing the logic of the class under test without involving actual dependencies like databases or external APIs. Furthermore, MockMvc allowed us to simulate HTTP requests and validate responses without the need to start a full web server, making tests faster and more focused.
   - Example

```
@Test
public void testInvalidRequestOnTopicsRetrieval() throws Exception {
    TopicsRequestDTO invalidRequest = new TopicsRequestDTO("", -1, -5, "");

    mockMvc.perform(post("/api/topics")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(invalidRequest)))
            .andExpect(status().isBadRequest());
}
```

## 5.2 End-to-End Testing

End-to-End (E2E) testing is a software testing methodology that evaluates the functionality and overall performance of an application from the perspective of the end user. In this project, E2E testing was employed to ensure that all components of the application, after being developed with Test Driven Development (TDD), interact correctly and perform as expected in a production-like environment. While TDD was used during the coding phase to verify the correctness of individual components, E2E testing was crucial in assessing the overall user experience, confirming that the services built meet the required performance standards and function seamlessly together in real-world usage scenarios.

# 6 Reproducibility

To facilitate reproducibility, a Docker Compose configuration has been created to orchestrate the deployment of the various microservices that make up the application. By containerizing each service, we can ensure that all dependencies, configurations, and environments are identical across different machines, making it easy for anyone to replicate the setup. The docker-compose.yml file defines how each service is built, configured, and interconnected.
To reproduce the environment and run the application, follow these steps:

1. **Clone the Repository**: Ensure you have a local copy of the project repository.
2. **Install Docker and Docker Compose**: Make sure Docker and Docker Compose are installed on your system. Docker Compose is essential for orchestrating the multi-container application
3. **Run Docker Compose**: Navigate to the root directory of the project where the docker-compose.yaml file is located and execute the following command: **docker compose up –build**.
   This command will build and start all the services defined in the docker-compose.yml file.
4. **Verify the Deployment**: Once the services are running, you can verify the deployment by accessing the frontend at http://localhost:3000 and the API endpoints as documented.
5. **Data Persistence**: The data generated or modified during the execution of the services will be stored in the Docker volumes (mongo-data and esdata), ensuring that data is not lost when the containers are stopped.

By following the steps outlined above, anyone should be able to replicate the exact environment in which this application was developed and tested.

## References


[1] *Spring by VMware Tanzu:* https://spring.io/.

[2] *The Guardian Open Platform:* https://open-platform.theguardian.com/.

[3] *MongoDB:* https://www.mongodb.com/.

[4] *Elasticsearch, The heart of the free and open Elastic Stack:* https://www.elastic.co/elasticsearch/.

[5] *Mallet:* https://mimno.github.io/Mallet/.

[6] *Docker.docs:* https://docs.docker.com/.

[7] *JUnit 5:* https://junit.org/junit5/.

[8] *Maven:* https://maven.apache.org/.

[9] *Mockito:* https://site.mockito.org/.