Awesome Modeling

Alberto Basaglia (2119289)
Milica Popovic (2119069)
Andrea Stocco (2108885)

June 24, 2024

1 Introduction

This project was developed by a group of Computer Engineering students enrolled in the master's course Software Platforms at the University of Padova. The primary objective of the project is to build a platform for downloading articles from online newspapers, storing them in a database, making them searchable through a search server, and extracting representations of themes discussed in a set of articles returned as results for a given query.

Newspaper articles are fetched from the Guardian API, saved in MongoDB, made searchable via Elasticsearch, and analyzed using Mallet for topic modeling. Additionally, the project aims at implementing a microservices architecture.

The application comprises a variety of services, that communicate with each-other.

The documentation is divided into the following chapters: Chapter 2, which describes the main concepts and technologies used.

2 Architecture

2.1 Technologies used

Firstly, the used technologies such as Spring, Mongo and Mallet will be briefly described in order to easier follow project description later.

2.1.1 Spring

The Spring Framework is a comprehensive and widely used Java-based framework for building enterprise-level applications. It provides a robust infrastructure for developing Java applications. It simplifies the development of complex applications by promoting good design practices and offering a suite of tools and libraries for building web applications, microservices, and data-driven solutions. Additionally, Spring's modular architecture and extensive ecosystem allow developers to use only the components they need, making it highly flexible and scalable. [**?**]

2.1.2 MongoDB

MongoDB is a popular open-source, document-oriented NoSQL database designed for scalability, flexibility, and performance. It stores data in flexible, JSON-like documents, allowing for varied and dynamic data structures without requiring a fixed schema. MongoDB is known for its ability to handle large volumes of data and its powerful querying and indexing capabilities. It supports a wide range of applications. [**?**]

2.1.3 Python

Python is a versatile and high-level programming language known for its readability, simplicity, and extensive standard library. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python's dynamic typing and interpreted nature make it an excellent choice for rapid application development. Python is also widely used in developing microservices due to its ease of use and speed of development. [**?**]

2.1.4 Elasticsearch

Elasticsearch is a powerful, open-source search and analytics engine designed for horizontal scalability, reliability, and real-time search capabilities. It is built on Apache Lucene and provides a distributed, text search engine with an HTTP web interface and schema-free JSON documents. It is commonly used for log and event data analysis, full-text search, and real-time analytics due to its high performance, flexibility, and ability to handle large volumes of data across distributed systems. [**?**]

2.1.5 Mallet

MALLET (MAchine Learning for LanguagE Toolkit) is a Java-based open-source toolkit for statistical natural language processing, particularly renowned for its implementations of topic modeling algorithms. It supports various algorithms for discovering latent topics in large collections of text documents. MALLET provides tools for preprocessing textual data, training topic models, and evaluating model performance. [**?**]

2.1.6 RabbitMQ

RabbitMQ is a powerful open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It enables seamless communication between distributed systems by acting as a mediator that facilitates the reliable transfer of messages between applications and services. It supports various messaging patterns such as point-to-point, publish/subscribe, and request/response. It is widely used in microservices architectures, IoT applications, and asynchronous communication scenarios where decoupling and reliability are crucial. [**?**]

2.2 Design - overall architecture

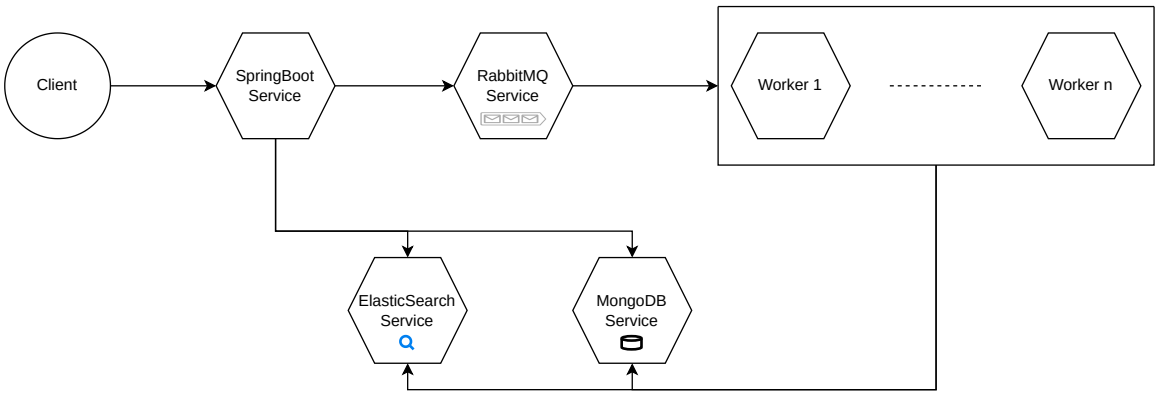

Figure 1: Microservices

Figure 1 shows a description of the subdivision of the system into microservices. First of all we can see that the client can access the application by interacting with the SpringBoot service. This service has access to the ElasticSearch and MongoDB services and can send messages into the queue. In the figure it is also possible to see the workers. These microservices, that will be described in detail later, read messages from the queue and execute the associated actions.

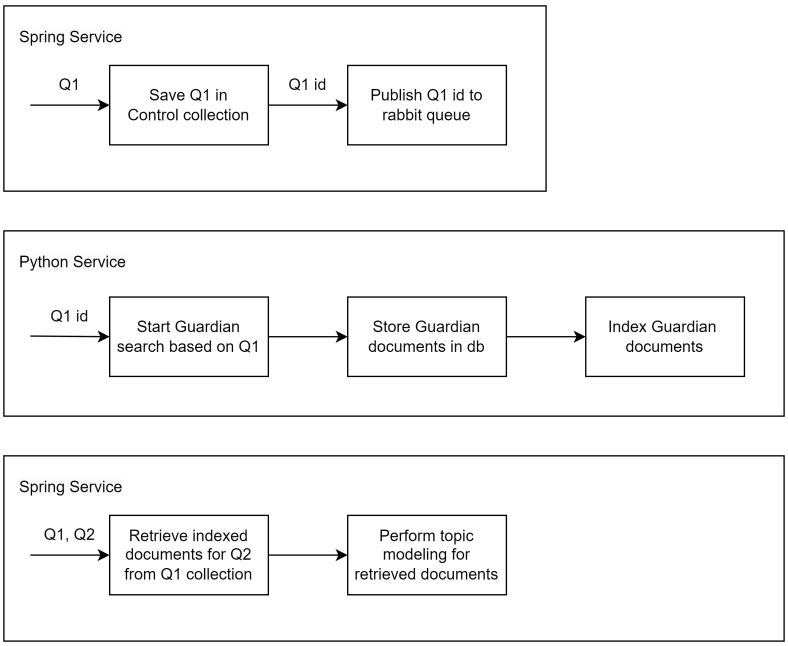

Figure 2: Process flow

Figure 2 shows three possible actions that the system can perform.

Firstly, the Spring service includes a monitoring functionality, initialized by a query referred to as Q1. An example JSON representation of Q1 is shown in Listing 1.

Listing 1: JSON representation of Q1

```
{
  "topic": "science",
  "local_start_date": "2024-05-01",
  "local_end_date": "2024-07-01"
}
```

This query is stored in a MongoDB collection named *Control*, which manages all Q1 queries. Listing2 shows class structure of Q1.

Listing 2: Class definition of QOne

```
String topic;

QOneStatus status;

LocalDateTime submitted_time;
LocalDateTime finished_time;

LocalDate local_start_date;
LocalDate local_end_date;
```

topic field represents name of Q1, *status* indicates the current processing status, which can have the following values: SUBMITTED, PROCESSING and FINISHED. Beside that, *local_start_date* and *local_end_date* are used to filter articles from Guardian API while *submitted_date* and *finished_date* represent date of Q1 submission and date when the documents fetching for Q1 is finished, respectively. Once saved in the database, the entity's ID is published to RabbitMQ.
Concurrently, a Python service subscribes to this queue. Upon receiving the Q1 ID, it triggers fetching from the Guardian API, setting the Q1 document status to PROCESSING. Articles retrieved from the Guardian API are saved in MongoDB in the format shown in Listing 3.

Listing 3: JSON representation of a Guardian Article

```
{
    "title": title,
    "content" : content,
    "guardian_id" : guardian_id,
    "web_date": web_date
}
```

The *title* and *content* fields represent title and contect of an article respectively whereas *guardian_id* and *web_date* represent guardian id and date of publishing an article. Each article is subsequently indexed and stored as a JSON object in the Elasticsearch server.

For the second query, Q2, a user requests k topics. This requires the Q1 ID as a parameter. For instance, if Q1 was about *Science*, and related articles are stored in the database, a subsequent Q2 query like *Nuclear war* expects to retrieve k topics based on that subset of documents.

2.3 Design for Spring service

One of the services in our application is a Spring service, which is responsible for receiving all user requests, saving Q1 in the database, pushing Q1's ID to RabbitMQ, retrieving documents from the ElasticSearch server, and performing the topic modeling part. Figure 3 shows the structure of the Spring service.

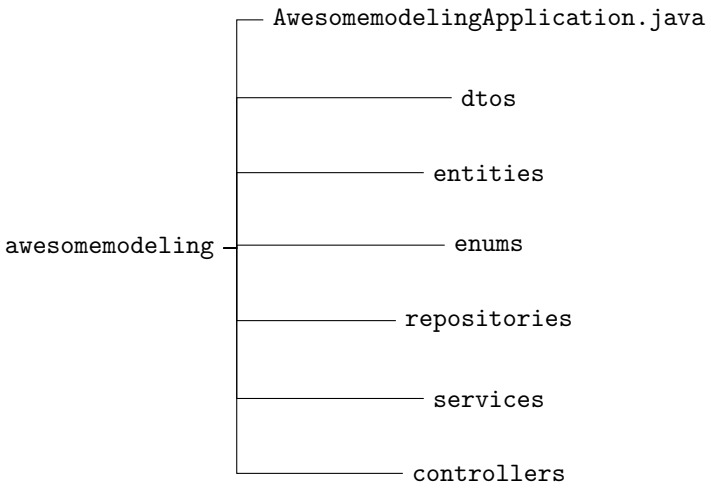

Figure 3: Folder structure of the Spring service

During the development of the Spring service, we followed the *Separation of Concerns* principle, which is a fundamental principle in software engineering and design. It is used to separate an application into units with minimal overlapping between the functions of the individual units [**?**]. This is done by splitting logic into three different layers: *controllers*, *services*, and *repositories*.

The *controllers* folder contains a class *QOneController.java* responsible for handling HTTP requests and mapping them to the appropriate service methods. This keeps the request handling logic separate from business logic and data access logic.

The *services* folder holds the business logic of the service—in our case, *MalletService.java*. The *repositories* folder contains the data access logic, specifically an interface that extends *MongoRepository—ControlRepository.java*. This separation allows us to change the data access layer without affecting the business logic.

Another good design practice was creating *entities* and *dtos* folders. The *entities* classes represent the data models or domain objects. By keeping them in a separate folder, we ensure that the domain logic is isolated from the rest of the application. The *dtos* folder keeps *Data Transfer Objects* classes separately, which are used to transfer data between layers of the application, especially between the client and server.

2.3.1 API requests

2.4 Design for Python service

Another service that is part of our application is the Python service called *downindex*, which is responsible for listening to incoming messages in *RabbitMQ*, fetching articles from the *Guardian API*, and indexing them using *ElasticSearch*. The structure of this service is very simple and is shown in Figure 4. *main.py* holds all the business logic of the service, while *requirements.txt* lists all the required dependencies.

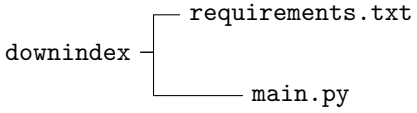

Figure 4: Folder structure of the Python service

2.5 Message Queue

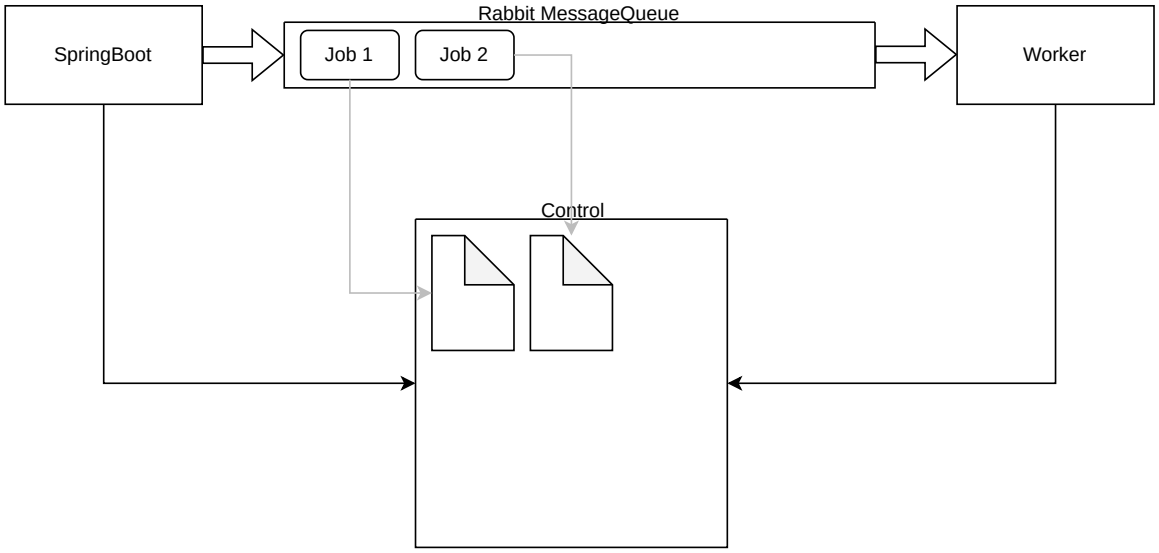

Figure 5: Queue

Figure 5 shows the architecture of the communication between the SpringBoot service (which is the service the user can interact with) and the worker nodes.

In this example only one worker it is present but the same would apply if more than one worker was active. In that case the only difference would be that the messages in the queue could be received by different workers.

The process goes as follows: the SpringBoot application creates a document in the *control* collection containing all the information that the worker needs to know to perform the action. In our system the only action that is submitted in the queue is the download and index of a topic. The Spring microservice then posts a message in the queue containing the id of the document.

This message will be then received by one of the workers. The first action that the worker does is retrieving the associated control document from the collection. Then it writes on the document that the action is being processed (so that an user can see that the system is working on its request) and starts performing it.

When the job is completed, the worker writes on the document that it is done.