Awesome Modeling

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

June 25, 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 technologies used and the design for the overall system, Chapter 3 which describes the design for each service individually, Chapter 4, which outlines the motivation for the specific design choices, Chapter 5, which explains the deployment of the microservices and Chapter 6, which explains the steps to run the application.

2 Design

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. [6]

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. [5]

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]

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. [1]

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. [4]

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. [7]

2.2 Design choices - overall architecture

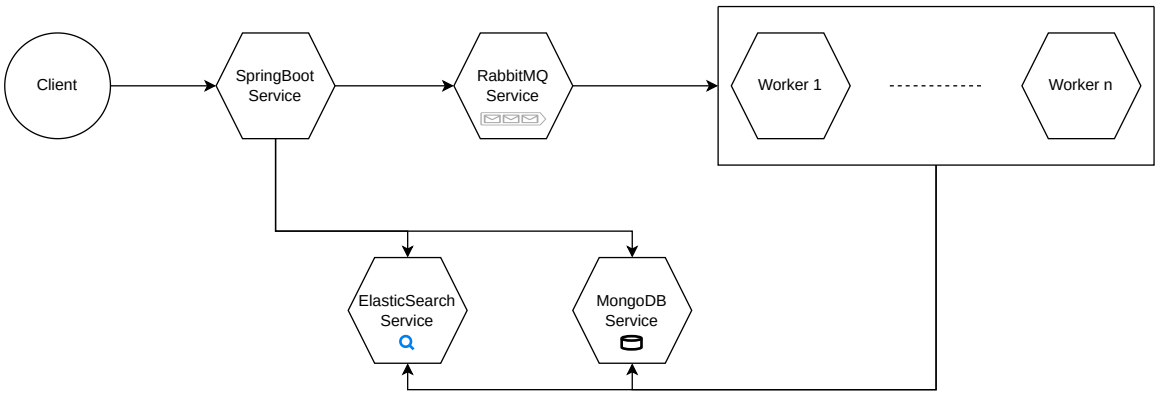

Figure 1: Microservices

Figure 1 shows an illustration 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.

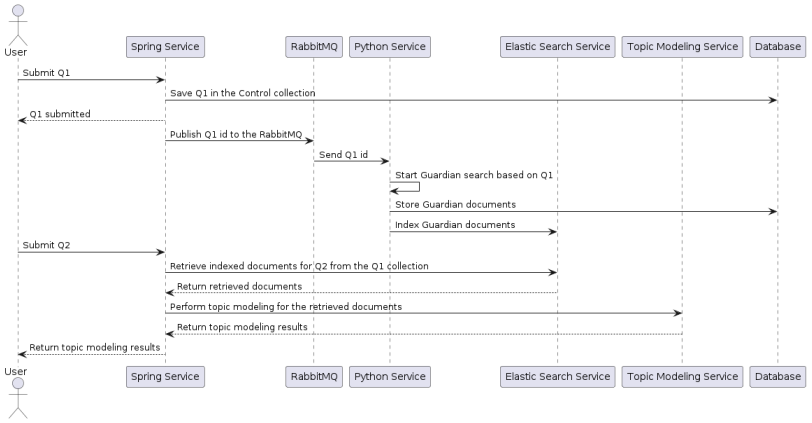

Figure 2: Sequence diagram

Figure 2 shows a sequence diagram for the whole system. The sequence diagram starts with the user submitting Q1 to the Spring Service. The Spring Service saves Q1 in the Control collection and publishes the Q1 id to the RabbitMQ queue. RabbitMQ then sends the Q1 id to the Python Service, which starts a search on Guardian based on Q1. The Python Service stores the retrieved Guardian documents in the database and indexes them using ElasticSearchService. Later, the user submits Q2 to the Spring Service, which retrieves the indexed documents related to Q2 from the the Q1 collection in the database and performs topic modeling on the retrieved documents. It is important to mention that the ElasticSearch Service runs in a separate container, while the Mallet Service is a service within the Spring Service.

In the next few paragraphs, a more detailed explanation of the whole system will be provided. 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. Listing 2 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;
```

The *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 the Guardian API while *submitted_date* and *finished_date* represent the date of Q1 submission and the 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. The 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 content 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.

3 Design choices - per service

This section presents the design choices for each service.

3.1 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. In the chapter 4 we will discuss design principles we followed and a reason behind that.

3.2 APIs

Our application exposes different endpoints to perform all the functionalities described in Figure 2. In the following subsections, the request and response bodies indicate the type related to each field.

3.2.1 GET /q1

Retrieve a list of all the QOne objects saved in the database

Request

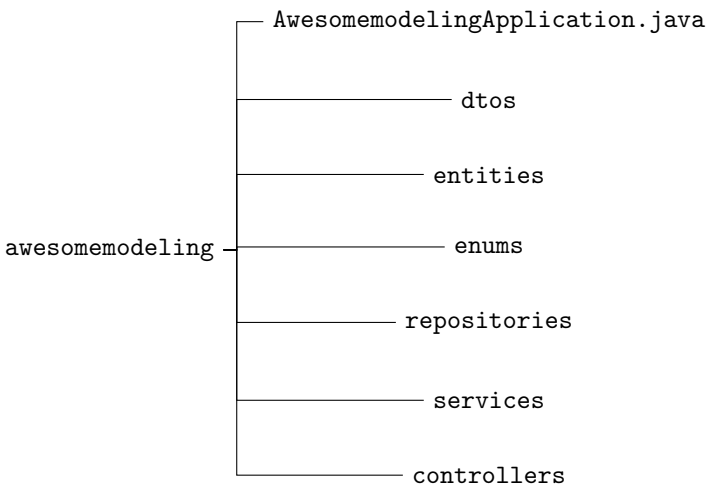

Figure 3: Folder structure of the Spring service

- **URL:**

  `/q1`

- **Method:** GET

Response

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

```
[
  {
    "id": "string",
    "topic": "string",
    "status": "string",
    "submitted_time": "datetime",
    "finished_time": "datetime",
    "local_start_date": "date",
    "local_end_date": "date"
  },
  ...
]
```

3.2.2 POST /q1

Create a new QOne object

Request

- **URL:**

  `/q1`

- **Method:** POST
- **Headers:**
  - **Content-Type:** application/json
- **Body:**

```
{
  "topic": "string",
  "local_start_date": "date",
  "local_end_date": "date"
}
```

Response

- **Status Code:** 201 CREATED
- **Body:**

```
[
  {
    "id": "string",
    "topic": "string",
    "status": "string",
    "submitted_time": "datetime",
    "finished_time": "datetime",
    "local_start_date": "date",
    "local_end_date": "date"
  }
]
```

Errors

- **500 Internal Server Error**: An error occurred while connecting to RabbitMQ

3.2.3 GET /q1/[QOneID]

Retrieve the QOne object specified by `QOneID`

Request

- **URL:**

  `/q1/[QOneID]`

- **Method:** GET

Response

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

```
[
  {
    "id": "[QOneID]",
    "topic": "string",
    "status": "string",
    "submitted_time": "datetime",
    "finished_time": "datetime",
    "local_start_date": "date",
    "local_end_date": "date"
  }
]
```

Errors

- **404 Not Found**: the specified QOne does not exist

3.2.4 GET /q1/[QOneID]/q2

Retrieve the themes related to `QOneID`

Request

- **URL:**

  `/q1/[QOneID]`

- **Parameters:**
  - **query** (mandatory): a query (Q2) related to Q1 on which performing topic modeling
  - **k** (mandatory): the desired number of topics
- **Method:** GET

Response

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

```
{
  "topics": [
    {
      "topic": "string[10]"
    },
```

```
        ...
    ]
}
```

Errors

- **404 Not Found**: the specified QOne does not exist
- **500 Internal Server Error**: topic retrieval failed

3.3 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.

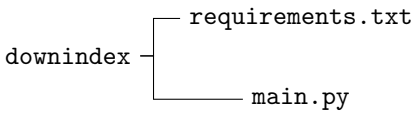

Figure 4: Folder structure of the Python service

3.4 Message Queue

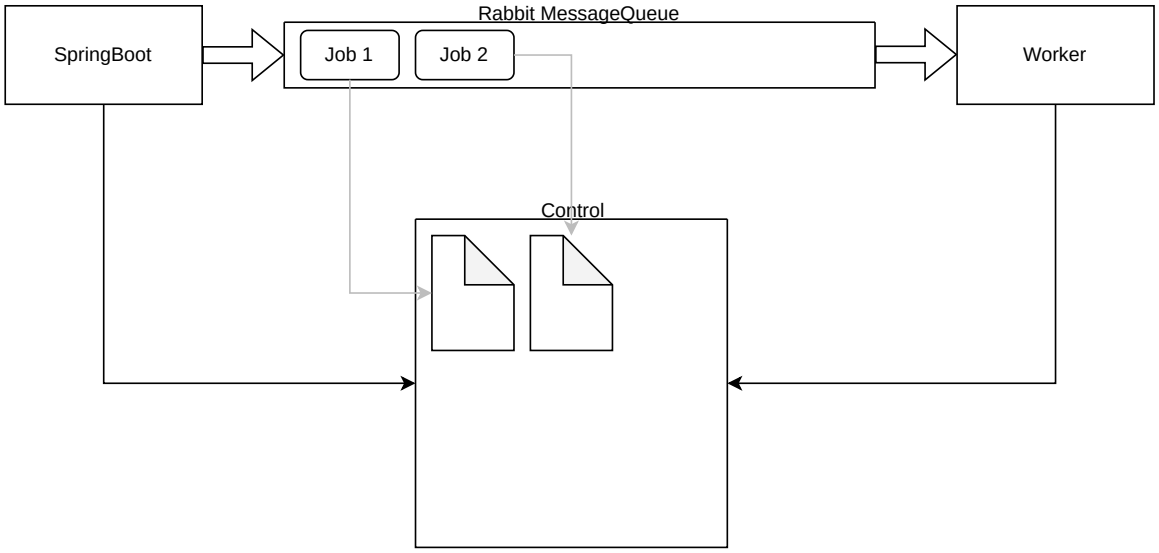

Figure 5: Queue

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.

4 Motivation for the specific design choices

4.1 Overall architecture

First of all, we decided to detach the indexing and downloading of articles from the endpoint. This was done for one main reason: the retrieval and indexing of documents can be a rather long process, and we didn't want to do it during the API call. For this reason we decided to do this asynchronously. The user will send the request to the endpoint which is going to insert the job into a queue. A response is sent to the client immediately, telling it that the request is going to be processed. In a second moment one of the worker nodes will read the job request from the queue and will start to process it. Once it is completed it will write into a control collection in the database that it completed the job so that, an user sending a request to check the status of the process, can see that the request is completed.

This can also help us to solve a second problem, *Scalability*. By having detached the endpoint and the "worker nodes" we can have a large number of them, working on the same queue. The queue will act as a *Load Balancer*, indirectly distributing the jobs to the different workers.

4.2 API Endpoint (Spring)

In this chapter we will explain design choices we made and the reasons for that.

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 [3]. This is done by splitting the 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 the business logic and the 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 the 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 the layers of the application, especially between the client and the server.

4.3 Worker node

Since the worker node needs to interact with the Mongo database and send REST requests to Elastic to index documents, we decided to develop it using Python. This allows us to manage json objects with dictionaries and to send HTTP requests with the *requests* library.

5 Deployment

In this project, we utilized Docker Compose to manage the deployment of a microservices architecture. Docker Compose allows us to define and run multi-container Docker applications, ensuring a consistent and reproducible deployment environment.

Below are the some considerations for deploying our services using Docker Compose:

5.1 Services

Since our application is based on a microservices architecture, we decided to put every service inside a Docker container. We declared the following services in the docker compose configuration:

- **MongoDB** that contains a process running the MongoDB server.
- **ElasticSearch** containing a running ElasticSearch server.
- **RabbitMQ** for hosting the message queue.
- **SpringBoot** that handles the client REST requests and communicates with the other microservices.
- **Worker** that waits for messages to be published in the queue and executes jobs.

Some additional services:

- **Kibana**, a user interface for Elastic, used as a debugging tool.
- **Mongo Express**, a user interface for MongoDB, useful for navigating the collections with an user interface.
- **FrontEnd**, a nginx process used to serve to the user a small and lightweight Angular applications. This is the way a "final" user could use the application.

It is important to note that we used the following configuration:

Listing 4: Replicas of a services

```
deploy:
  mode: replicated
  replicas: 2
```

In this way docker compose will create more than one container for the same image. We set this value to 2 because we are in a "development" environment. If this application was being run on a more powerful machine (or even better on a cluster of machines) we could use much higher values. This parameter it is useful since it will allow our application to process more than one job at the same time, using the queue as a load balancer.

Health checks are crucial for the correct execution of services. They make sure that each service is operational before other dependent services start. MongoDB and RabbitMQ have specific health check commands configured. In this way, for example, we will avoid that a worker (that depends on the message queue) is started before the queue is operational.

The *depends_on* attribute in Docker Compose ensures that services start in the correct order. For example, the Python application waits for MongoDB and RabbitMQ to be healthy before it starts, ensuring that all required dependencies are available.

The full configuration for the docker compose can be seen in the *docker-compose.yml* file.

5.2 Volumes

In order to avoid losing all the date when the container is destroyed, we created 2 volumes. One will be mounted in the data folder of the MongoDB container and the other one in the data folder of the ElasticSearch container. In this way, both the database and Elastic will store their data in a volume.

6 Reproducibility

To run the application an user would only need to

- install Docker and Docker Compose on its machine;
- run `docker compose up --build`

Docker Compose will take care of downloading all the base images, building all the images specified in the *Dockerfile* file and spinning up all the containers.

The API will be available at *http://localhost:8080* and the front-end at *http://localhost:8082*.

7 Testing

In order to test if the application works as expected, we used 2 different approaches.

7.1 Unit Testing

The first one, *Unit Testing*, was used for the Spring Microservice. As we discussed in the previous sections, the Spring application contains a Service that we use to interact with the Mallet library. To test this service we wrote a specific class, called `MalletServiceTest`, that checks whether the calls to the Mallet library work as we expect.

7.2 End-to-end testing

End-to-end (E2E) testing is a methodology used to validate the complete and integrated functionality of an application, ensuring that the entire workflow behaves as expected from start to finish. Differently from unit testing, which focuses on individual components or services, E2E testing simulates real user scenarios and interactions with the system, covering all interconnected parts.

For our application, we designed a E2E test that covers a typical workflow the final user could go through. In order to accomplish this, we wrote a python script that, with the use of the *requests* library, interacts with the REST endpoint and checks if everything works.

The script goes through the following steps:

- **Setup of the application**: for this step we use the command that deploys the application using docker compose.
- **Creation of Q1**: we create a Q1.
- **Await for the completion**: the script waits (by sending a request every few seconds) for the workers to download and index the collection.
- **Generating topics (Q2)**: the script sends a request for the generation of topics based on a second query (Q2). The script then checks whether the api returned k valid topics.
- **Shutting down the application**: we use the docker compose command that stops and deletes all the container associated with the application.

References

[1] Elastic. Elasticsearch. https://www.elastic.co/, 2024. Accessed: 2024-06-22.

[2] Python Software Foundation. Python programming language. https://www.python.org/, 2024. Accessed: 2024-06-22.

[3] GeeksforGeeks. Separation of concerns (soc), 2021. URL https://www.geeksforgeeks.org/separation-of-concerns-soc/. Accessed: 2024-06-24.

[4] David Mimno. Mallet documentation: Topics. https://mimno.github.io/Mallet/topics.html, 2024. Accessed: 2024-06-22.

[5] Inc. MongoDB. Mongodb. https://www.mongodb.com/, 2024. Accessed: 2024-06-22.

[6] Inc. Pivotal Software. Why spring? https://spring.io/why-spring, 2024. Accessed: 2024-06-22.

[7] RabbitMQ. Rabbitmq documentation. https://www.rabbitmq.com/, 2024. Accessed: 2024-06-22.