Universidade de Coimbra

Department de Engenharia Informática

Systems Integration Assignment 2 Report

Bruno Almeida 2021237081

Marco Lucas 2021219146

Supervisor: Andre Bento

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Master of Computer Engineering in
Software Engineering

November 8, 2024

Abstract

This report presents a practical and analytical study of a web application project based on **Reactor Core/Web Reactive** technology. Developed by two students of MEI at UC, this project demonstrates the effectiveness of reactive programming in managing asynchronous data flows and user interactions with media content. The main goals were to leverage the performance and responsiveness of reactive systems, create efficient data streaming, and provide a good solution for showing and rating media items such as movies and TV shows. This report discusses the architecture, methods, results, and optimizations applied to deal with a certain amount of data queries in real-time.

Introduction:

Modern web applications often require efficient, asynchronous data handling to manage dynamic content and provide a responsive user experience. This project, "Reactor Core/Web Reactive for Media Management," uses **Reactor Core/Web Reactive** technology to implement a reactive web application in **Java Spring Boot**, exploring its effectiveness in handling complex user, media data and them related. The goal was to create an application that can dynamically manage data with low delays, enabling real-time interaction between client and server. This project centers on managing media entities and their interactions with users through reactive programming techniques, showcasing how this approach outperforms traditional synchronous models. Throughout this report, we will demonstrate the processes used, the results obtained, and provide a critical assessment of the performance of reactive programming compared to traditional methods.

I Core Concepts

Reactive programming, based on asynchronous, non-blocking data handling, is vital in applications with extensive data exchanges and event responses. This project uses **Reactor Core**, a powerful library for building reactive applications that react to changes in data or events dynamically. Key components include **Flux** for handling multi-item data streams and **Mono** for single-item streams. These elements allow for efficient and responsive data processing, crucial in this system where media entities and user interactions are continuously updated and also allowing some more refined and controlled manipulation of asynchronous events and data.

Additionally, **lambda expressions** enable concise functional programming in Java, allowing for more readable and maintainable code, especially when managing data transformations and event-driven logic. This combination of reactive streams and lambda expressions supports the development of a scalable, responsive application, essential for data-intensive operations like querying media records and managing user interactions. These approaches are fundamental in modern communication of systems, ensuring that data is processed and transmitted efficiently, regardless of technology techniques or programming languages involved.

II Project Architecture

The project architecture consists of two main parts: the Server and the Client. Each of the parts run in separate Docker containers, with dedicated "docker-compose" scripts.

Server: Built with Java Spring Boot, the server is responsible for managing media and user data, storing it in a relational database, and exposing it through RESTful APIs. These services perform CRUD operations (Create, Read, Update, and Delete) on media and user entities, ensuring that the data is always accessible and up-to-date. The server uses Spring Data with a reactive R2DBC (Reactive Relational Database Connectivity) layer to efficiently handle database interactions in a non-blocking way, improving scalability under load. Both the server and the database are running in a container within their own Docker network.

Client: The client is where the user can interact with the server by requesting data from the server. This request is done through Spring WebClient, is handled efficiently, and the relevant data is retrieved and transmitted back to the client. The different aspect here is that the client not only receive this information, but also to process it reactively and store them in text files. This functionality allows a bigger flexibility and accessibility of the data, making it available for future analysis or for use in different applications. Key client operations include fetching media details, like name and release date of the movies, also retrieving the relationships between users and media, and calculating various statistics (e.g., average ratings). This client-server interaction forms a responsive, efficient ecosystem for real-time media management between 2 applications running in separate and different Docker environments.

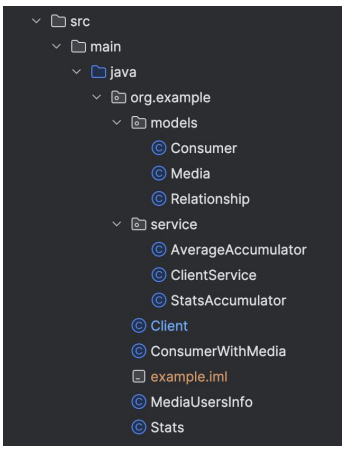

Figure 1: Client Side

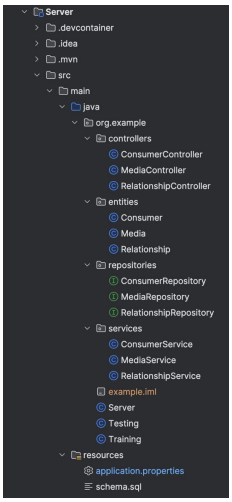

Figure 2: Server Side

Figure 3: req TXT Files

III Server Implementation

The server side of the project includes key classes and configurations:

- Server: The main one, this is what runs the aplication.
- Entities: The Media, Consumer and relationship classes represent core entities, the media has an id, a title, a release date, an average rating and a type, the consumer class has an id, a name, an age and a gender, the relationship class has an id, a consumer ID, a media ID and a rating.
- Repositories: Using RelationshipRepository, the MediaRepository and ConsumerRepository manage interactions with the database related to the Media and the Users.
- Services: MediaService and ConsumerService encapsulate business logic, they use the Repositories to manage the data by creating, updating, searching and deleting some info about the enteties.
- Controllers: MediaController, ConsumerController and RelationshipController handle HTTP requests, calling the appropriate services for CRUD operations and sending responses to the client. They map some URLs to make some operations, like getting all medias, updating a media and deleting a media.
- Testing Class: We also have a Testing class that acts as a testing program for CRUD operations and queries on Media, Consumer and Relationship entities in a media management system. Uses the Spring WebFlux WebClient for asynchronous requests, allowing data retrieval, creation and deletion. It also performs advanced queries and writes results to files, using efficient and reactive operations.

IV Client Implementation

The client application is implemented using Spring WebClient to asynchronously fetch data from the server. The core logic resides in the client class, which manages interactions with the server and processes the data to produce reports based on project requirements but for that class to work we have the heart of the Client Side that is the ClientService class. This class uses Spring WebFlux features, such as WebClient, to make asynchronous and reactive calls to the server. These calls are designed to get, send and manipulate data about Users and Media.
We also have the model classes, such as Media, Consumer and Relationship, pretty much the same as the ones on the Server Side. These classes are direct mappings of data received from the server, facilitating the manipulation and analysis of this data.
We have three great classes that are, **ConsumerWithMedia** that represents a consumer along with a list of the media items that he consumes. This class can be useful for associating basic consumer information with consumed media titles by storing them in a mediaTitles list. It has methods for adding a new media title, allowing you to easily associate new media with the consumer. **MediaUsersInfo** class that represents a media along with the number of consumers using it (userCount). Additionally, it stores a list of Consumers organized in descending order of age. This makes it easier to consult the list for the query 9 but we are going there. The addConsumer method allows you to add a new consumer and keeps the list always ordered. The **Stats** class is used to represent basic statistics, such as the average and standard deviation of a data set. This class is useful for the query 6. Through the getAverage and getStandardDeviation methods, it is possible to access these statistical values, which allows any part of the system to use this information efficiently and without altering the original data.
Now talking about the queries itself to fetch and process information about media items and their users, **query 1**: To fetch the titles and release dates of all media items, we use a GET request to the /media endpoint. The response is converted to a Flux<Media> and mapped to extract only the title and release date for each media item. The resulting Flux allows us to process each media entry individually. **Query 2**: To obtain the total count of media items, we retrieve the media data from the server as a Flux<Media>. The .count() method is applied to count the total number of media items, which returns a Mono<Long> containing the result. **Query 3**: This query involves fetching media items and then filtering for those with an average rating greater than 8. The .filter() operation ensures only the items meeting the rating criteria are counted, with the final result being a Mono<Long> containing the total count. **Query 4**: For this query, we use two GET requests—first to retrieve all consumers and then to access the media subscriptions for each consumer. The distinct() function ensures that only unique media items are counted, with the final result stored as a Mono<Long>. **Query 5**: This query filters media items released between 01-01-1980 and 31-12-1989 with media.getReleasedate().isBefore and media.getReleasedate().isAfter. Once filtered, the media list is sorted by descending average rating using the .sort() method. The sorted Flux<Media> makes it easy to process media data from this era. **Query 6**: To calculate statistics on media ratings, we retrieve all media data and map each entry to its avg rating. A StatsAccumulator object collects and calculates the average and standard deviation, which is stored in a Mono<Stats>. **Query 7**: This query identifies the oldest media item by release date using an if condition after that uses a .sort() operation on the release date, we take the first item in the sorted list. The result is a Flux<Media> containing the oldest item, mapped to extract its name.**Query 8**: To calculate the average number of users per media item, we first count the users associated with each media item. We then use a reducer that accumulates the sum of user counts and the number of media items processed. This information enables the calculation of the average, stored in a Mono<Double>.**Query 9**: This query retrieves each media item and the users associated with it. The users are sorted in descending order by age using .sort() on the Consumer age attribute. A MediaUsersInfo object stores this sorted user list and the user count, making it easy to reference media items by their audience demographics.**Query 10**: Finally, we use a nested query to retrieve full user details and their subscribed media items. Each ConsumerWithMedia object associates user information with their list of subscribed media titles. This structured approach allows easy access to user and subscription data in the system.

V Reflections

In the setup of the Docker containers, there was an issue on loading the .jar files upon building the docker-compose files: the target directory, where all .class files and maven SNAPSHOT jar files reside, wasn't being recognized during building, which lead the students to take an alternative method that worked, which was putting the ".jar" files inside ".devcontainer" folder alongside the docker-compose file.

To facilitate connection between client and server, the students used "host.docker.internal", which works as a way for one container to access another container who is on its own Docker network.

Conclusion:

This project provided a deep dive into reactive programming, lambda expressions, and Spring WebFlux for creating responsive web applications. Despite some initial challenges with configuration—particularly Docker setup—and further complexities during the coding phase to ensure the application's reactivity and responsiveness, we believe we effectively implemented the key objectives. We successfully created a reactive server and client architecture that interacts efficiently and responsively with a database, meeting the project's functional and performance requirements.

References:

- Reactor 3 Reference Guide. URL: https://projectreactor.io/docs/core/release/reference/
- Reactor Core Reference. URL: https://projectreactor.io/docs/core/release/api/index.html
- Spring WebFlux. URL: https://docs.spring.io/spring-framework/reference/web/webflux.html