Web Reactive Application Integration

Žan Pižmoht*, João Samuel Diogo*
*University of Coimbra, DEI
Emails: zp6881@student.uni-lj.si, uc2021219898@student.uc.pt

***Abstract*—This paper presents an exploration of a reactive client-server application designed to manage media content. The main focus lies on understanding how reactive programming models are implemented on both server and client sides, emphasizing complexities such as reactivity, database access, and optimization strategies for performance. The reactive approach helps improve resource efficiency and responsiveness. The server leverages Spring WebFlux for handling non-blocking requests, while the client uses WebClient for data collection. By applying reactive programming, we tackled the complexities of managing relationships across multiple services in a non-blocking manner. This work provides valuable insights into handling legacy services reactively.**

I. Introduction

In this project, we implemented a reactive client-server system to manage media content, specifically movies and TV shows, along with user interactions. The server side exposes media and user data using Spring WebFlux and Reactive CRUD operations, while the client side performs reactive data retrieval and generates a variety of reports. This project serves as an exploration of how modern reactive programming concepts can be effectively utilized in a media management context, especially when interacting with a legacy system where data aggregation is not straightforward. The aim was not only to fulfill functional requirements but also to achieve optimal system performance through careful handling of reactivity and resource management.

II. Server-Side Complexity and Reactivity

The server side of our project uses Spring WebFlux to handle requests in a reactive, non-blocking way. This section explains some of the more challenging parts of the server, especially how we dealt with reactivity, accessing the data, and handling exceptions.

A. Reactivity in the Controller Layer

The main controllers (`MediaController` and `UserController`) rely on the reactive programming concepts in Spring, using `Mono` and `Flux` to manage incoming requests. This means instead of blocking and waiting for data, the server can handle multiple things at once, making it more efficient.

- **Handling Requests Reactively:** Each endpoint returns a stream of data or a single item using `Mono` or `Flux`. For example, the `getAllMedia()` endpoint can stream all media items without tying up the server's resources. This helps a lot when there are many requests at the same time.
- **Logging and Errors:** We used operators like `doOnNext` to log what is happening at different stages, which was super useful for debugging. With `doOnError`, we were able to catch and log errors if something went wrong. This way, we could understand and fix issues without crashing the entire flow.

B. Data Access Layer and Managing Relationships

We used `ReactiveCrudRepository` for database access, which is part of Spring's reactive tools. It lets us interact with the PostgreSQL database without blocking. While the CRUD operations were mostly straightforward, managing the relationships between users and media was more complicated.

- **Handling Relationships:** The `Relationship` entity is used to map which users are connected to which media. Instead of directly embedding user or media lists into these entities, we used a separate table. This made the database more efficient and avoided overly complex queries. It also fit well with the project requirements, which didn't allow us to directly load all relationships.
- **Chaining Queries Reactively:** To get information like which users are subscribed to a particular media item, we had to chain multiple reactive queries together. For instance, `getUsersByMediaId()` starts with finding the relationships and then extracts the user IDs. This chaining added complexity, but it ensured we were only pulling in the data we needed when we needed it.

C. Handling Errors in Reactive Streams

Dealing with errors reactively was another key part of the server setup. Using `doOnError`, we managed to log any issues that happened during the reactive data flow. This helped make sure that one problem wouldn't disrupt other ongoing operations, which is crucial for maintaining smooth performance when multiple users are interacting with the system.

III. Client-Side Implementation

The client-side of our project is significantly more complex than the server side, especially due to the requirements to perform various calculations and fetch data reactively from a legacy server. In this section, we explain how each of the requirements was implemented, the challenges we faced, and the optimizations that helped make the queries more efficient.

A. Reactive Requests and Relationship Handling

One of the main challenges was to manage reactive calls efficiently while fetching related data. Since the server was set up as a legacy third-party service, our client couldn't retrieve entire entities in a single call. For instance, to get information about the users subscribed to a particular media item, we first had to fetch the IDs using one endpoint and then use those IDs to gather further details. This added a lot of complexity, as it meant chaining multiple asynchronous queries, but it also forced us to optimize and reuse parts of our code to avoid redundant calls.

B. Implementation of Requirements

The client, implemented in the `App.java` class, consists of several methods, each corresponding to one of the requirements:

- **REQ 1 (Titles and Release Dates)**: We used a simple reactive pattern to fetch all media titles and their release dates, leveraging the `bodyToFlux()` method to stream the data and map it to a formatted string. This helped keep memory usage low while processing the data in chunks.
- **REQ 2 (Total Media Count)**: To get the total count of media items, we used a `count()` operation on the Flux of media, which gave us a reactive way to count elements without collecting them all into memory.
- **REQ 3 (Count of High-Rated Media)**: For finding the media items with a rating greater than 8, we used a filter operation followed by a count. The filter worked reactively, ensuring that only relevant items were processed.
- **REQ 4 (Count of Subscribed Media Items)**: For this requirement, we needed to count the media items that had users subscribed to them. We first retrieved all media and then used a nested reactive call to check if any users were associated with each media item. This was done using a combination of `flatMap()` and `hasElements()`, allowing us to skip any unnecessary processing if no users were linked to a given media.
- **REQ 5 (Media from the 80s Sorted by Rating)**: To meet this requirement, we filtered media based on release dates and sorted them by average rating. This was challenging due to the need to ensure the data was handled reactively; we couldn't load all items into memory at once, so we used `sort()` and then mapped each item to a formatted output string.
- **REQ 6 (Average and Standard Deviation of Ratings)**: This requirement involved calculating statistics across all media ratings without breaking the reactive flow. We used the `reduce()` operator to aggregate ratings, calculating the average and standard deviation. This helped maintain non-blocking operations and improved performance.
- **REQ 7 (Oldest Media Item)**: The oldest media item was found by reducing the Flux of media items using a comparison of release dates. The `reduce()` operator let us iteratively compare each item to find the one with the earliest date.
- **REQ 8 (Average Number of Users per Media)**: Calculating the average number of users per media item required us to chain multiple reactive calls to count users for each media item. Instead of collecting all counts in a list, we used a `reduce()` operation to calculate the average in a streaming manner. This approach avoided memory accumulation and kept the logic reactive.
- **REQ 9 (Users Sorted by Age for Each Media)**: This requirement involved fetching user details for each media item and then sorting users by age in descending order. We used `flatMap()` to first get user IDs for a media item, and then a second `flatMap()` to retrieve details for each user, sorting them reactively without collecting into lists.
- **REQ 10 (Complete User Data with Subscribed Media Titles)**: For this, we fetched all users and, for each user, retrieved their subscribed media titles. We used a combination of `flatMap()` and `reduce()` to gather media titles reactively and format them into a single string.

C. Network Resilience

To bolster the client-side resilience against network disruptions, our application employs a retry mechanism that makes up to three reconnection attempts upon network failures.

Additionally, each reconnection attempt is paired with a timeout setting of ten seconds. If the server fails to respond within this interval, the attempt is aborted, and a retry may be initiated.

This strategy is only employed on the first requirement, but could easily be applied to the remaining ones.

D. Optimizations and Lessons Learned

One major optimization was avoiding collecting data into memory whenever possible. Requirements like REQ 6, REQ 8, and REQ 10 previously used `collectList()` but were refactored to use streaming aggregation with `reduce()` to maintain a fully reactive approach. This change minimized memory usage and improved the overall performance of the client application.

We also learned that chaining multiple requests in a reactive flow can get complicated, especially when handling legacy APIs that don't support bulk queries. Using `flatMap()` allowed us to do these asynchronous calls without blocking, but it made the logic harder to follow and required more careful error handling. Adding `doOnError()` to catch errors at each stage helped us debug and keep track of where issues were happening.

In conclusion, building the client required us to think deeply about managing reactivity, chaining calls efficiently, and making sure that our application could handle data streams without overwhelming resources. The lessons learned regarding non-blocking flows and optimizing chained queries have definitely been valuable in understanding how to work with legacy APIs in a reactive system.

IV. Client-Side Exception Handling

Client-side error handling in the application is managed using the `onErrorResume()` method from the Reactor

library, which allows for graceful error recovery without interrupting the flow of the program. This method ensures that when an error occurs during data retrieval or processing, the application can log the error and continue executing subsequent operations. Each reactive chain of requests is wrapped with error handling to provide resilience to individual failures, ensuring that the system remains operational.

To log errors, the application utilizes SLF4J, a simple logging facade for Java. The log configurations, typically found in the `application.properties` file, ensure that only relevant error messages are captured and displayed. These configurations help to manage log verbosity and focus on critical issues.

A. Error Handling Approach

When an error is encountered, the `onErrorResume()` operator is used to log the error and continue the execution. This ensures that the pipeline does not terminate prematurely and continues to execute other operations. This approach minimizes the impact of individual failures and prevents the system from crashing entirely.

V. Conclusion

In this project, we implemented a reactive client-server application that efficiently manages media and user interactions using Spring WebFlux. The server-side emphasized reactivity through non-blocking CRUD operations, allowing us to handle multiple requests concurrently. The client-side posed greater challenges due to the need for reactive data retrieval, efficient relationship management, and multiple chained queries to fulfill the requirements.

We successfully tackled the complexities of dealing with a legacy server by implementing solutions that prevented blocking, optimized data flows, and improved overall performance. The project highlighted the importance of balancing reactivity, memory usage, and asynchronous call chaining to create an application that is not only functional but also scalable and efficient. Lessons on performance tuning and handling reactivity were crucial outcomes, offering a deeper understanding of working with modern reactive technologies and integrating them into real-world applications.