Migrating from Developing Asynchronous Multi-threading Programs to Reactive Programs in Java

Andrei Zbarcea and Cătălin Tudose *

Posted Date: 20 November 2024

doi: 10.20944/preprints202411.1467.v1

Keywords: Java; asynchronous programming; multi-threading; reactive programming; migration from asynchronous to reactive

Article

Migrating from Developing Asynchronous Multi-Threading Programs to Reactive Programs in Java

Andrei Zbarcea [1] and Cătălin Tudose [1,2,*]

[1] Faculty of Automatic Control and Computers, National University of Science and Technology POLITEHNICA Bucharest, 060042 Bucharest, Romania; andrei.zbarcea@gmail.com

[2] Luxoft Romania, 060042 Bucharest, Romania

* Correspondence: catalin.tudose@gmail.com

Abstract: Modern software application development imposes standards regarding high performance, scalability, and minimal system latency. Multi-threading asynchronous programming is one of the standard solutions proposed by the industry for achieving such objectives. However, the recent introduction of the reactive programming interface in Java presents a potential alternative approach for addressing such challenges, promising performance improvements while minimizing resource utilization. The research examines the migration process from the asynchronous paradigm to the reactive paradigm, highlighting the implications, benefits, and challenges resulting from this transition. To this end, the architecture, technologies, and design of a support application will be presented, designed to outline the practical aspects of this experimental process while closely monitoring the phased migration. The results are examined in terms of functional equivalence, testing, and comparative analysis of response times and resource utilization, as well as the cases where the reactive paradigm proves to be a solution worth considering. Additionally, possible directions for further research and development are presented.This paper not only investigates the design and implementation process but also sets a foundation for future research and innovation in dependable systems, collaborative technologies, sustainable solutions, and distributed system architecture.

Keywords: Java; asynchronous programming; multi-threading; reactive programming; migration from asynchronous to reactive

1. Introduction

1.1. Research Overview

Web platforms are commonly used to distribute software to users. As the user base grew, applications evolved to improve the user experience. This progress has led to the development of new requirements and standards for high performance, scalability, and rapid response to user inputs [1, 2]. Over time, developers have used a variety of strategies and technologies to meet these goals, focusing on different areas of the software development process.

Initially, early solutions aimed to improve current hardware and optimize code for better efficiency. Later on, there was a significant transition to a paradigm that included the use of multi-threading asynchronous programming approaches, which allow the application to handle multiple processes simultaneously to improve performance by parallelizing the program flow.

Within the Java ecosystem [3, 4], multiple programming interfaces have been created to support the asynchronous multi-threading paradigm. Some of these include *wait/notify* mechanisms and the *java.util.concurrent* library, and abstractions such as *CompletableFuture* and *Executors* [5]. These advancements have enabled the development of newer, more efficient, and faster applications. However, they have not eliminated the complexity of thread management and concurrency. This signifies the need for further enhancements in the simplification and optimization of modern application development.

 doi:10.20944/preprints202411.1467.v1

Reactive programming [6] aims to address these challenges by reducing the complexity of concurrency management while enhancing application performance and scalability. This approach has its roots in a paradigm that leverages non-blocking I/O [7] and an event-driven architecture [8]. The integration of reactive programming interfaces into Java streamlines the development of applications that meet contemporary performance and scalability requirements while minimizing system resource utilization.

1.2. Problem and Approach

Reactive programming has the potential to offer a more elegant and simpler alternative through the event-based and non-blocking I/O model, contrasting with the increased complexity in managing concurrency and system resources of the asynchronous multi-threading approach which, while it has brought significant improvements over previous models and has been the basis for many of the solutions subsequently developed, also has the potential to become the limiting factor in terms of scalability and extensibility. The main issue addressed by this research is the exploration of a migration methodology and feasibility assessment, as well as the quantification of the benefits that could be achieved. The untapped potential of reactive programming and the opportunities to optimize the performance and efficiency of current applications are the central motivation of the research, outlining a promising option to address current challenges.

However, migrating from asynchronous multi-threading programming to reactive programming is not a universal formula for performance enhancement under any conditions and is not without challenges, featuring its particularities, limitations, and trade-offs. While it has been successfully adopted in some areas, such as real-time data stream processing or distributed application development, migrating existing applications requires a detailed evaluation of the architecture, possible infrastructure adjustments, adherence to guidelines and practices to ensure the transition, and an openness to understanding the different paradigm, demanding a shift in perspective from writing imperative code. Analysis of the benefits and limitations of reactive programming, such as identifying the use cases for which it is suitable and the appropriate strategies for dealing with complexity, remains defining and specific to each product.

The research directs its attention on the migration process from the asynchronous approach towards the reactive paradigm, following closely the steps to achieve the transition in a controlled and efficient way based on the purpose-built application. The step-by-step process aims at investigating some aspects of interest, trying to clarify some technical details, but also capturing some consequences of the adoption of this new paradigm, and assessing its effectiveness in enhancing system performance.

1.3. Objectives

The research's preliminary goal is to develop a core application that uses industry-standard asynchronous multi-threading programming techniques. This will allow the analysis of the strengths and limits of the approach. This initial stage will serve as a foundation for subsequent examination and comparison to the reactive paradigm.

The primary purpose is to migrate the application to the reactive programming approach. This will involve gradually transitioning components to use the reactive interface and concepts, as well as adjusting to complementing technologies like non-blocking web servers and reactive database connections [9]. The process will be systematically detailed, highlighting the adjustments and challenges experienced along the way while also providing solutions, recommendations, and strategies to assess and decide on the feasibility and benefits of a smooth and efficient transition toward this paradigm.

The study will conclude with a comparative performance analysis between the applications using both paradigms, including metrics such as memory usage, CPU consumption, response times, and the number of execution threads employed in different scenarios. With this analysis, the research aims to provide a deeper and more nuanced understanding of the benefits and limitations related to reactive programming in contrast to the conventional asynchronous approach.

 doi:10.20944/preprints202411.1467.v1

Achieving these objectives will make this research a considerable and useful reference for properly implementing the migration process and evaluating its expected impact, thereby constructively contributing to both the theoretical and practical components of software development.

2. Theoretical Background

To examine the background and motivation of the research, as well as the challenges involved in the migration process, fundamental concepts will be reviewed in this chapter.

2.1. Concurrency

The capability of a system to handle multiple tasks at the same time is known as concurrency. Concurrency in Java [5, 10] may be achieved through the usage of threads. The program can perform multiple tasks simultaneously using threads, which are self-contained subunits that can run in parallel. Concurrency is an extensively explored principle for optimizing resource utilization and increasing system efficiency. By using multiple threads to parallelize the execution, response times can be reduced and a better distribution of workload can be achieved.

Race conditions and deadlocks can occur as a result of complex and error-prone concurrency management in multi-threading environments [11].

A race condition happens when the state of a resource depends on the sequence or timing of uncontrollable events, producing inconsistent or unexpected results. The execution of each thread may take a different amount of time than expected, they can finish in a different order than expected, generating unanticipated behavior.

A deadlock occurs when two or more threads are waiting for resources that are blocked by each other, leading to a state of complete standstill, which has the potential to severely compromise application stability and performance. There are different strategies available to prevent, detect, or resolve these problems, such as using advanced resource access scheduling algorithms or sophisticated deadlock avoidance mechanisms [12].

To streamline concurrency management as well as improve performance and scalability, alternative concurrency models have emerged, such as event-driven and reactive programming.

2.2. Asynchronous Programming

In asynchronous programming, by allowing multiple processes to be executed simultaneously, the overall performance and scalability of the application can be improved significantly. In traditional synchronous programming, there is a linear flow of execution, as a task can only be performed after the previous one is completed. There are various interfaces provided by the Java ecosystem that can be used to develop asynchronous programs: *Future*s and *Executor*s [5]. When the result of an asynchronous operation becomes available, it can be accessed in a non-blocking way using a *Future*. A pool of threads is managed by *Executors* to perform tasks asynchronously while also allowing for more efficient usage of system resources, as the threads managed through these implementations can be reused instead of being created and deleted for each task. Where tasks are I/O-bound or have high latency, the employment of asynchronous programming can significantly increase application performance and scalability. However, it can also present some issues, such as increased complexity in controlling the execution flow and handling concurrency errors [12].

2.3. Non-Blocking I/O

The concept of non-blocking I/O refers to the ability of an application thread to perform other tasks instead of waiting for a response to a previously initiated request [13]. This allows for more efficient utilization of system resources, which can lead to improved performance and scalability. The NIO library, which provides classes such as *Selector* and *Channel* for handling non-blocking I/O operations, is mainly used in Java for interfacing with these scenarios [7]. In addition, non-blocking I/O operations can be handled using the asynchronous paradigm, to further increase efficiency, employing methods such as event-based programming or the use of *Future*s and *Executor*s.

 doi:10.20944/preprints202411.1467.v1

2.4. Event-Driven Architecture

An event-driven architecture is a software design pattern in which messages or events produced by external sources control the flow of the program [14]. In this architecture, the system consists of small, self-contained components that respond to events and connect via a centralized event bus. Since components can be added, removed, or updated swiftly without affecting the core of the program, this method allows for a more flexible and scalable solution [15]. Microservices, real-time applications, and distributed systems frequently make use of the event-driven design [16]. When developing functionalities for a performance-critical program, event-driven architecture is an option worth considering, allowing for asynchronous and non-blocking data flow which leads to improved efficiency and scalability [17].

2.5. Reactive Programming

Reactive programming is a paradigm focused on data flows and propagation of changes. It is based on the Observer pattern [18], under which an entity, known as the subject, keeps track of a list of observers, and automatically notifies them about any change in the state of the application. As a result, data flow processing becomes more efficient and flexible, enabling non-blocking and asynchronous data handling.

Reactive streams are a specification that defines a set of interfaces and methods for asynchronous processing with non-blocking backpressure. The mechanisms employed to take advantage of the programming concepts of reactive streams are implemented within various libraries such as Project Reactor or RxJava [5, 6]. These libraries provide a broad set of high-level operations, including filtering, mapping, and reducing, which are employed for dealing with reactive data streams efficiently. These powerful abstractions in Java applications can significantly streamline the development process, improving both performance and scalability.

From straightforward data processing flows to complex event-driven systems, reactive programming has a wide range of applications. This is going to be the main focus of the research, particularly looking at the benefits and difficulties of transitioning from the conventional solution, the asynchronous multi-threading programming, towards a reactive programming approach.

3. Related Work

In recent years, the evolution of asynchronous and reactive programming has significantly impacted the software development landscape across various programming environments. With growing demands for performance, scalability, and responsiveness in high-concurrency scenarios, many developers are transitioning from traditional asynchronous programming models to fully reactive architectures. Existing studies have explored the benefits and challenges associated with this transition, highlighting performance improvements, optimized resource management, and additional complexities. This section reviews key contributions that examine both the development and application of these paradigms, focusing on the frameworks and strategies that support this migration.

3.1. Evaluating Hibernate Reactive for Scalable Database Solutions

Grinovero, in the study conducted by the Hibernate Team, evaluates the use of Hibernate Reactive for asynchronous, non-blocking database access in Java applications [19]. Hibernate Reactive builds on reactive programming principles to enhance scalability, allowing for higher levels of concurrency with reduced latency and optimized resource usage. In performance benchmarks, Hibernate Reactive maintained response times under 10ms at loads of 35000 requests per second, whereas its synchronous counterpart, Hibernate ORM, only achieved similar performance up to around 20000 requests per second. These findings highlight Hibernate Reactive's advantages in handling high-load scenarios, with fewer threads required to achieve these results.

Additionally, the study points out specific cases where Hibernate Reactive proves beneficial, particularly in applications where scalability and resource efficiency are priorities [19]. However, certain limitations are noted, such as potential overhead in complex transactions and multi-join

 doi:10.20944/preprints202411.1467.v1

operations, which could impact performance. As a result, the study suggests that while Hibernate Reactive offers significant value within a reactive architecture, a careful assessment of its applicability is essential, especially in environments with complex data requirements.

3.2. Assessing R2DBC in High-Concurrency Web Applications

Ju et al. [20] explored the impact of asynchronous frameworks and database connection pools on web application performance under high-concurrency conditions. The study utilized a configuration combining Spring WebFlux, R2DBC, and database connection pools to handle simultaneous requests in a non-blocking manner. Tests conducted with varying loads showed that this configuration maintained an average response time of 7 ms at 500 requests, compared to 8 ms in traditional JDBC. For 50000 requests, the asynchronous setup with a connection pool achieved 556 ms, significantly outperforming the 723 ms without a pool. Additionally, the error rate in the connection pool configuration remained at 0% for up to 150000 requests, in contrast to the higher error rates observed in synchronous models and asynchronous models without a connection pool, which exceeded 50% under similar loads.

The study demonstrates the advantages of the asynchronous model in high-load scenarios, offering up to 20% higher throughput and greater stability than synchronous approaches, which became ineffective under massive request loads. The results suggest that utilizing a connection pool is essential for maximizing efficiency in database access for high-concurrency applications, highlighting the importance of non-blocking, resource-efficient setups in handling concurrent requests with minimal delays [20].

Dahlin [21] examined the performance of R2DBC within Spring WebFlux, comparing it with a traditional Spring MVC setup using JDBC, particularly focusing on database communication efficiency in high-concurrency environments. His study revealed that R2DBC, integrated with Spring WebFlux, reduced CPU and memory usage compared to JDBC in most scenarios, particularly in non-blocking environments. For instance, during tests with 200,000 insertions, R2DBC maintained lower CPU usage, peaking at 10%, whereas JDBC experienced notable delays and risked crashes when memory constraints were applied. The R2DBC configuration also achieved faster response times in handling large data sets, indicating a potential advantage in high-concurrency applications.

Dahlin further identified limitations, such as challenges in handling large BLOB data with R2DBC, as it required loading entire BLOBs into memory. This drawback limits R2DBC's applicability for applications relying heavily on large data fields. Nevertheless, the study emphasizes that R2DBC offers a more efficient solution "out-of-the-box" for standard data transactions, requiring minimal configuration adjustments, in contrast to JDBC, which may demand batch or fetch size adjustments for optimal performance [21].

3.3. Benchmarking Virtual Threads and Reactive WebFlux for Concurrent Web Services

Joo and Haneklint [22] analyzed the performance differences between Virtual Threads and Reactive WebFlux in Spring applications, aiming to determine the advantages of each concurrency solution under high-demand conditions. The study involved the development of three Spring application prototypes: one using normal threads, one with virtual threads, and another based on the Reactive WebFlux model. The prototypes were tested by simulating interactions between an aggregation endpoint and two underlying services, each configured with fixed delays of 100 ms and 500 ms.

The results indicate that the virtual thread prototype demonstrated superior performance in most tests, maintaining low latency and handling a high rate of requests per second even under heavy load. For example, the virtual thread prototype responded seamlessly up to 700 requests per second at a 100 ms delay, while WebFlux reached its limit at approximately 600 requests, and the synchronous model became unresponsive at 100 requests. With a 500 ms delay, virtual threads again outperformed WebFlux, handling 250 requests per second compared to WebFlux's 225 [22].

While the performance of Virtual Threads is promising, the study notes that WebFlux might yield different results on a non-blocking web server like Netty, given that the tests were conducted

 doi:10.20944/preprints202411.1467.v1

on Tomcat, a blocking server. In conclusion, these results suggest that virtual threads present a viable alternative for high-concurrency applications in Spring, although further testing in real production environments is needed to confirm their advantages over reactive solutions [22].

3.4. Reactive and Imperative Approaches in Microservice Performance

Mochniej and Badurowicz [23] examined the performance of microservices developed using reactive and imperative approaches by implementing two Java-based microservices with the Spring framework. In their study, they conducted performance tests for operations such as data retrieval and insertion, processing, and file transfer, comparing reactive and imperative microservices under varying load scenarios (100 and 3000 simultaneous users with multiple service instances). Their findings indicate that reactive applications performed better in cases with delays in communication with databases or other services, reducing response time and RAM usage by up to 36% in certain scenarios. However, for CPU-intensive tasks, reactive applications proved slower by up to 46%, suggesting additional complexity in handling reactive streams compared to imperative code.

Additionally, the study highlights the advantages of reactive applications in managing inter-service communication and optimizing hardware resource usage. Reactive applications required fewer threads due to the event loop model, thereby reducing RAM demands in scenarios such as order processing or barcode generation, with memory usage up to 57% lower. Although reactive applications showed advantages in I/O-intensive scenarios, their limitations for CPU-intensive processing emphasize the importance of assessing the context of use before implementing a reactive system, ensuring that the selected model aligns with the system's performance requirements [23].

3.5. Actor-Oriented Databases for Scalable and Reactive IoT Data Management

Wang's research introduces Actor-Oriented Databases (AODBs) to enhance scalable and reactive data management in IoT, specifically addressing the challenges of high concurrency and dynamic data handling. Dolphin, a prototype M-AODB, leverages actors as modular, stateful entities that communicate asynchronously, suitable for managing IoT entities in mobile contexts. This system was tested with different spatial and reactive settings, achieving 3349 moves per second under Actor-Based Freshness semantics, with a 50% latency of 6.26 ms, and 5211 moves per second under Snap(1s), which maintained a 50% latency of 0.59 ms [24]. By utilizing actor isolation and spatial partitioning, Dolphin adapts to high-demand IoT environments, supporting low-latency responses even under variable client loads.

To handle the demands of reactive IoT applications, Dolphin employs a "moving actor" abstraction and spatial partitioning techniques to ensure efficient data distribution and balanced workload management. Testing results indicate that Dolphin's Actor-Based Snapshot semantics can handle high spatial skew, with a throughput increase of up to 1.52x when batching reactions, maintaining stability across different server configurations. This approach allowed Dolphin to scale effectively in IoT scenarios like vehicle tracking, where reactivity and resource optimization are crucial [24].

3.6. Temporal and Type-Driven Approaches to Asynchronous Reactive Programming

Bansal, Namjoshi, and Sa'ar tackle the challenge of constructing asynchronous programs directly from temporal specifications, proposing an approach to simplify and make asynchronous synthesis practically feasible [25]. Traditional methods for synthesizing asynchronous programs from temporal logic were highly complex and prone to exponential growth in state space, making them challenging to implement. This work introduces a novel, compact automaton construction, which avoids the exponential state blowup common in previous approaches and reduces asynchronous synthesis to synchronous synthesis. The resulting automaton has at most twice the states of the input specification, significantly improving efficiency. Furthermore, for specific temporal properties, the approach replaces automaton construction with Boolean constraint solving, further simplifying the synthesis process [25].

 doi:10.20944/preprints202411.1467.v1

The study also demonstrates the practicality of their approach through experiments with a prototype tool, BAS, which incorporates their construction method along with existing synthesis solvers. BAS efficiently handles complex specifications that previously required impractical computation times. For example, it synthesizes an asynchronous arbiter specification within seconds, while previous methods took over eight hours. These advancements make asynchronous synthesis a feasible alternative for constructing reactive systems, broadening the applicability of automated synthesis in real-world asynchronous and distributed system design [25].

Bahr, Houlborg, and Rørdam present Async Rattus, a functional reactive programming (FRP) language embedded in Haskell, designed to handle asynchronous computations in a type-safe and efficient manner using modal types [26]. Async Rattus extends the Rattus FRP language by introducing dynamic, local clocks for asynchronous subsystems, allowing components to operate independently rather than synchronously under a global clock. This approach reduces inefficiencies, particularly in applications where components react to distinct events, such as graphical user interfaces. Async Rattus employs two modal types, to track delayed and stable computations, preventing space leaks and ensuring causality. Its Haskell compiler plugin automates type-checking for clock dependencies while enabling developers to leverage Haskell's extensive ecosystem.

The effectiveness of Async Rattus is demonstrated through various examples, including an interactive console application that dynamically updates values based on user input without relying on synchronous timing. The prototype shows that Async Rattus maintains low memory overhead and effectively mitigates space leaks common in asynchronous FRP, supporting more complex, concurrent applications within Haskell's functional paradigm [26]. The implementation highlights Async Rattus's potential for broader adoption in real-world scenarios that demand scalable, event-driven processing with minimal manual configuration.

While valuable insights into the performance and viability of reactive solutions have been provided by these studies, much of the existing research focuses on evaluating specific frameworks or comparing asynchronous and reactive models in isolated contexts. In contrast, this work captures the step-by-step migration process, addressing challenges encountered during the transition from an existing asynchronous architecture to a reactive one. By emphasizing practical considerations and migration strategies, this study aims to bridge a gap in current research, offering guidance on adapting an established codebase to a reactive paradigm while preserving core functionalities.

4. Current State of Technology

Reactive programming and applications consist of several main concepts and tangential aspects. The adoption of microservices-based architectures is explored, introducing the popular Spring framework [27] and its reactive version Spring WebFlux [28], analyzing the performance and costs of using reactive libraries, and looking at real-world examples of companies that have successfully adopted reactive programming in their projects. This analysis is primarily intended to provide a high-level understanding of the current state of reactive programming and its applications in practice.

4.1. Microservices

Microservices are a software architecture design approach that structures an application as a suite of loosely coupled services [29]. Each service operates in its own process and communicates with other services through straightforward mechanisms, typically through programmable interfaces (APIs). The main goal of microservices is to provide modular, scalable, and maintainable solutions for complex applications. The increasing adoption of microservices in modern applications is driven by the demand for scalable, flexible, and agile software development.

Generally, companies are leveraging established technologies to implement, interconnect, and deploy services. Communication technologies and standards such as RESTful HTTP [30, 31, 32] and containerization tools like Docker [33] are valued for their commanding interoperability and portability, which facilitate the decoupling and variation within each system. Java emerges as a preferred solution due to the large availability of developers proficient in this programming language. Additionally, systems developed for external clients tend to feature less decentralization

 doi:10.20944/preprints202411.1467.v1

and fewer product-specific particularities compared to solutions designed specifically for internal usage [29].

Regarding the impact of microservices on the overall quality of programs, it is prevalently assessed as positive or neutral. Furthermore, maintainability is also being regarded widely as positive, although the transition from monolithic solutions can be seen as problematic security-wise due to the rising requirement of implementing compliant mechanisms across each module, while also continuously monitoring and updating each component to address vulnerabilities when they are discovered, and last but not least, to provide secure ways of communication between the services [34, 35].

4.2. The Reactive Manifesto

The Reactive Manifesto [36] is a set of principles and recommendations for the development of reactive systems. It was developed in 2014 by a group of software development specialists to provide a clear and concise overview of the core design aspects that are addressed by reactive systems. The manifesto outlines four main characteristics of reactive systems: responsiveness, resilience, elasticity, and message-driven operation, as outlined in Figure 1.

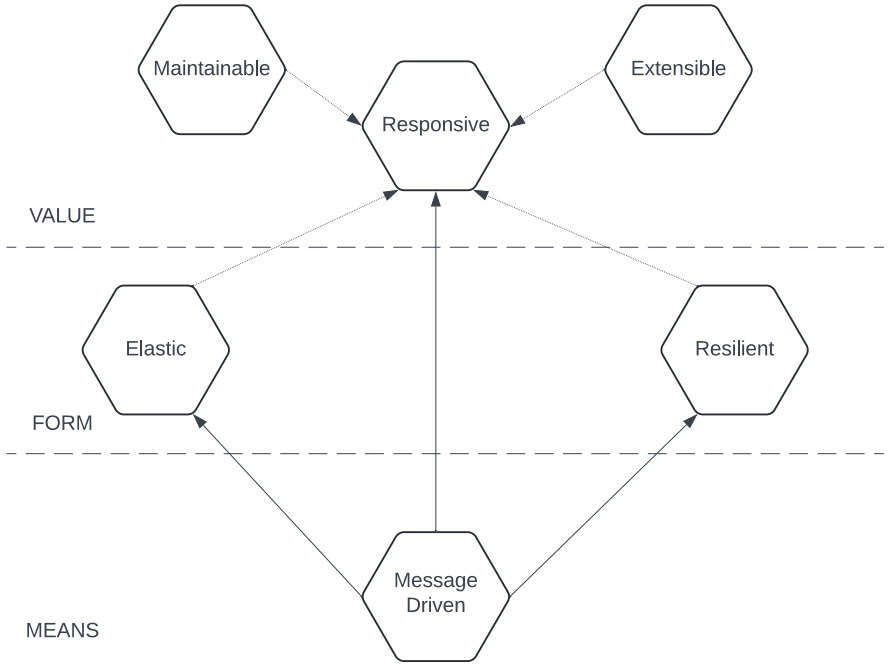


Figure 1. Characteristics of reactivity [36].

Responsiveness refers to the system's ability to handle user requests in a timely and efficient manner, even under heavy load conditions. Resilience refers to the ability of the system to quickly recover from problems and continue to function properly, even in the face of unpredictable situations. Elasticity refers to the ability of the system to dynamically adjust its resources as needed to accommodate changes in demand. Message-driven functionality refers to the ability of the system to use message transmission as the primary method of communication between its components, ensuring loose coupling and flexibility.

The Reactive Manifesto has been widely adopted by the community, and it has become a foundation for developing reactive systems [37]. Since its initial version, it has been constantly refined and enhanced as the field of reactive programming has matured and evolved, with new tools and technologies becoming available. The principles outlined in the manifesto are now being recognized and integrated into the development process as best practices for building scalable and resilient systems [38].

 doi:10.20944/preprints202411.1467.v1

4.3. Spring Boot and Spring WebFlux

Spring Boot [39] is a popular open-source framework for developing Java applications. It offers a variety of features that streamline the development process, including easy and automatic configuration with minimal requirements of programmatic adjustments, making it a popular choice for developers, especially for fast-paced environments, given its focus on simplicity and the availability of extensive functionalities and a wide array of plugins which ultimately enable the rapid creation and deployment of robust solutions.

WebFlux [40] is a reactive web framework that became available with the release of Spring 5, specifically designed for building efficient, scalable, and non-blocking web applications. WebFlux provides the required foundation and tools for developing high-performance applications that demand improved concurrency management and robust data flow processing. Compared to conventional blocking I/O, the reactive programming model enables improved utilization of system resources as well as significantly improving overall performance, particularly expected within applications designed to enable increased user concurrency and demanding workloads.

In addition to reactive programming features, WebFlux also features integration and support for functional programming and is based on an event-driven architecture [41, 42]. This allows for the development of highly responsive applications that can handle large numbers of requests and data flows.

Spring Boot and WebFlux provide a solid combination for developing modern Java web applications, ensuring a simple and efficient way to build, test, and deploy applications with minimal effort [40].

4.4. Performance and Cost Analysis

When developing reactive applications, one important aspect is the appropriate selection of tools for the application's specific requirements. RxJava and Project Reactor are two of the most popular libraries currently available, offering a considerable number of built-in features that allow for improved concurrency, scalability, and performance [5]. Some benchmarks have been undertaken and will be analyzed further to provide relevant information on their performance and costs.

In terms of individual reactive operations, RxJava outperformed Project Reactor, proving to be the superior choice for reactive processing in the conducted benchmarks, which covered scenarios such as transforming and selecting values, chaining individual operations, and applying multiple composed operations [43]. This is important because asynchronous individual operations are frequently used in situations such as database insert requests or message acknowledgment. The results are reported in Table 1.

Table 1. Throughput for individual operations [43].

Operation type	RxJava (ops/ms)	Reactor (ops/ms)	Base (ops/ms)
Map	33000	10000	63000
Chain	23000	13000	63000
Multiple operators	12000	5000	28000

In terms of event stream processing, Project Reactor performed better than RxJava, displaying an overall improvement in processing time and the ability to handle a larger number of events. This is equally important, as event streams are widely used in modern applications, especially in real-time data processing platforms such as streaming services or IoT devices. Results are available in Table 2.

Table 2. Throughput for event streams [43].

Operation type	RxJava (ops/ms)	Reactor (ops/ms)	Base (ops/ms)
Map	240	250	-
ManyToMany	130	140	-
Filters	130	150	-

 doi:10.20944/preprints202411.1467.v1

Multiple operators	100	100	120

For I/O-bound operations, both RxJava and Project Reactor performed similarly in terms of processing time. However, Project Reactor had a slight advantage in terms of resource consumption, utilizing less memory and CPU compared to RxJava. The efficient management of I/O-bound operations is also a relevant topic, as these operations are frequently encountered for accessing data from external resources such as databases, file systems, or network services.

The results of these benchmarks indicate that there is no universally optimal solution for all programming tasks, with both RxJava and Project Reactor demonstrating effectiveness in different scenarios. In scenarios involving individual operations, RxJava proved to be superior, whereas Project Reactor is better suited for managing operations with event streams. So, each framework has its own unique strengths, making them both valuable depending on the application's specific requirements. For I/O-bound operations, both libraries performed similarly, with Project Reactor showing a slight advantage in terms of resource consumption. When deciding on migrating to reactive programming, it is important to consider the specific requirements and constraints of the application and choose the library that best meets them. The cost of migration can be high, but the long-term benefits of improved scalability, resilience, and resource efficiency can compensate for these costs in many situations, such as real-time data processing or handling large amounts of data. Ultimately, the decision of whether to migrate should be based on a detailed analysis of the performance and cost benefits that can be achieved for each business use case.

4.5. Adoption of Reactive Programming

The reactive paradigm's ability to efficiently process large amounts of data while maintaining an optimal usage of resources has made it more popular within the enterprise world as a considerable choice in critical components within the systems, where high performance and scalability are the key aspects. Reactive programming is often leveraged by companies to perform only partial migrations rather than transitioning the whole system, allowing for improved performance of critical parts without changing the rest of the application.

Netflix is a company that provides streaming services featuring movies, series, and other online content, allowing its users to watch their favorite content anytime, anywhere, from multiple devices, while also being one of the largest contributors to the Java ecosystem, developing numerous tools and frameworks for the community. On the reactive programming side, Netflix is contributing towards the development of the open-source RxJava (Reactive Extensions) project, leveraging the reactive programming model for superior server-side concurrency and successfully reducing network interference. Netflix's implementation relies heavily on Observables, and the service layer is asynchronously structured around it. In addition, the service layer implementation leverages a functional programming methodology, which is compatible with the reactive paradigm, as a way of writing maintainable code that is capable of handling concurrency, employing multiple threads, leveraging non-blocking I/O, or relying on caching without changing the way client code communicates with or structures responses. Furthermore, the reactive programming model provides a series of operators for filtering, selecting, converting, combining, and composing observables quickly, which streamlines the management of interconnected asynchronous processes, features that Netflix services strongly benefit from [44].

Oracle, a company specializing in database software and technology, has significantly contributed to the advancement of reactive programming within the enterprise sector. In 2017, Oracle initiated the development of the Asynchronous Database Access API (ADBA), aiming to establish a standardized, non-blocking framework for Java applications interfacing with relational databases. However, by September 2019, Oracle ceased ADBA development, opting to focus on Project Loom, which introduces lightweight, user-mode threads known as fibers. This decision was based on the premise that fibers could simplify concurrent programming by allowing developers to write straightforward, sequential code without sacrificing scalability. In parallel, Oracle has embraced the Reactive Relational Database Connectivity (R2DBC) initiative, releasing the Oracle R2DBC driver in

 doi:10.20944/preprints202411.1467.v1

March 2021, facilitating non-blocking, reactive database operations, enabling seamless integration with reactive frameworks like Project Reactor and RxJava [45]. Oracle underscores its commitment to providing developers with robust tools for building efficient, scalable, and responsive applications by aligning with R2DBC and investing in Project Loom, considerable contributions to the reactive ecosystem.

PayPal is one of the leading technology companies that owns a global payments platform. In the context of PayPal, this enables a seamless user experience and the ability to seamlessly process transactions, manage real-time updates and changes to payment status, account balances, and other important information, while responding to these changes promptly. PayPal has embraced reactive programming by relying on Akka as a framework to develop its services. It addresses issues such as scalability, latency, and resiliency resulting from the presence of a large number of low-throughput application components. Since Akka proposes a functional programming model, the code becomes easier to comprehend and test, allowing for accelerated development and appropriate error handling. PayPal has also chosen to contribute and make use of the concept of "squbs", a stack that simplifies the creation and management of loosely coupled components. Through this approach, manageability is improved, ensuring symmetry and loose coupling between services [46].

Cloud Foundry is a Platform as a Service (PaaS) that experimented with reactive programming to provide an efficient and scalable solution for building and deploying cloud-native applications. Reactive programming is a paradigm that focuses on data flow, as well as efficient manipulation of that data, using programming models that are non-blocking and event-driven. The implementation of the reactive components in Java attempted by Cloud Foundry is based on Project Reactor, the standard implementation assumed by the Spring development team. Their platform leans towards reactive programming for robust management of the request and data flows between different application components, such as the frontend and backend, therefore going a long way towards ensuring that the system remains responsive and scalable even when faced with high volumes of data and traffic. At the same time, the adoption of reactive programming in Cloud Foundry may also help in reducing latency and downtime by enabling real-time processing of data and events [47].

Moreover, other important companies have recognized the value of reactive programming and are accelerating its development. Facebook, Alibaba, and other companies decided to collaborate and organize a community under the Linux Foundation, focusing on the advancement of reactive programming and technologies such as RSocket, designed to ensure efficient, resilient communication between microservices, ensuring robust and scalable application performance across diverse environments and devices, showcasing the growing commitment and highlighting its potential to improve modern software development industry-wide.

5. Proposed Solution

5.1. Architecture

The project is based on a microservices architecture that simulates the minimal backend of a Twitter-like social media platform, designed to demonstrate multiple scenarios and allow for a more detailed comparison between the asynchronous and reactive programming paradigms. The architecture leverages multiple core services, each with distinct functions within the application ecosystem. Figure 2 provides a detailed representation of the employed architecture, highlighting both the individual components and the established interconnections.

 doi:10.20944/preprints202411.1467.v1

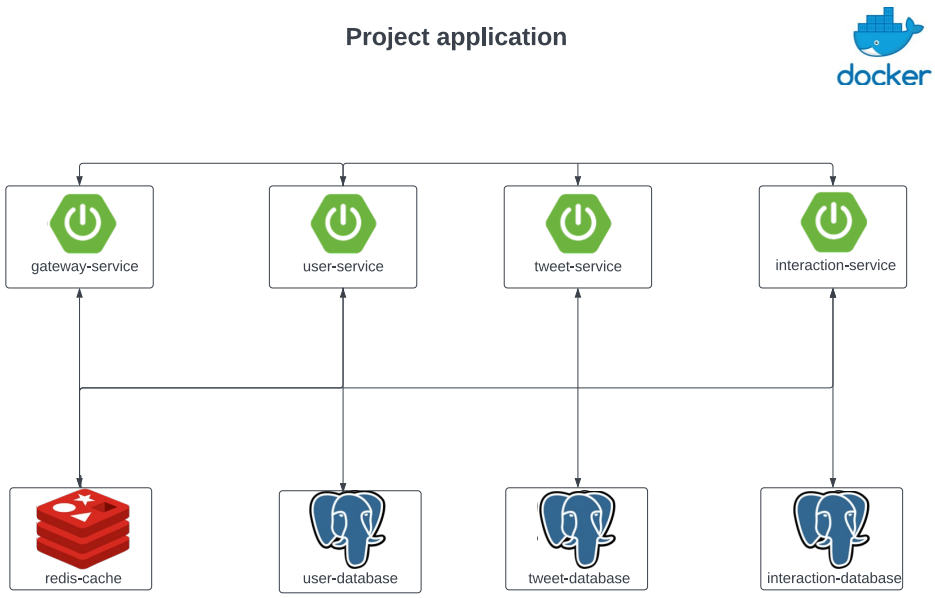


Figure 2. Architecture diagram.

The Gateway Service represents the application's entry point, allowing incoming requests to be routed to the appropriate services. It is also responsible for the validation process within the mechanism used for authentication and authorization, based on the JWT token standard [48, 49]. This ensures a centralized authentication process, removing validation pressure from downstream services by simplifying their involvement to only perform the extraction and post-processing steps of the required token data.

The User Service is responsible for managing all aspects related to the user, including the initial login process, issuing the JWT tokens, user registration, the handling of profile information, and any profile updates. Although not as diverse in terms of dedicated use cases, it is an integrated part of operations delegated to other services, which must retrieve additional user information to perform the computations and fulfill the requests.

The Tweet Service manages tweet-related operations, such as the creation, modification, deletion, and retrieval of user posts. It is also responsible for handling the tokenization and extraction of mentions and hashtags, as well as generating the user feeds, by aggregating all the necessary data and computing it into operable results.

The Interaction Service enables interactions within the social media platform between users, allowing the management of follows, likes, retweets, and replies, therefore adding a degree of dynamism and user engagement.

This architecture adheres to the "database-per-service" microservices pattern [50], ensuring the independence and scalability of individual components. Each service leverages its dedicated instance of a relational database while also incorporating a caching mechanism for guaranteed performance by minimizing database read operations and inter-service exchange of recently computed information.

5.2. Tools and Technologies

The development process for the asynchronous and reactive projects employed a series of diverse technologies that fulfill different functions within the context of each paradigm. The primary goal is to highlight the strategic selection and effective usage of two distinct sets of technologies within each model while also accounting for the adoption of these tools within applications already available on the market:

Java and Spring Boot Versions

- *Java 21* and *Spring Boot 3* were utilized for the development of the asynchronous and reactive applications, ensuring stability and compatibility with mainstream technologies, and providing

 doi:10.20944/preprints202411.1467.v1

robust support for development, and concurrency. These are the latest long-term support versions, which should soon be considered the standard for the current application development, and are already a requirement for most of the tools and frameworks actively developed in today's technology market [13].

Asynchronous Operations

- *CompletableFuture* was introduced in Java 8, and provides a robust interface for asynchronous code development, allowing the chaining of multiple steps of the computation process, abstracting multiple aspects of thread management. Compared to its predecessor, the *Future* interface, it offers a more fluid, non-blocking design and supports other ecosystem advances such as lambda expressions [6].
- *ExecutorService* allows for granular control over concurrency and asynchronous task execution. It allows the allocation, configuration, and utilization of dedicated thread pools, which can improve application performance in critical scenarios.
- *Mono* and *Flux* are dedicated types that support the reactive and functional approach, as well as straightforward handling of data streams. The interfaces are based on the Project Reactor library [5], leveraged to support the development of the WebFlux component in the Spring ecosystem [13].
- *Schedulers* are the primary mechanism enabling concurrency management in the reactive model, allowing for enhanced flexibility and granular control through the selection of execution contexts and thread pools such as *boundedElastic*, parallel, immediate, and single, the appropriate selection is imperative for maintaining low resource consumption and high performance.

HTTP Clients

- Java *HttpClient* was utilized in the asynchronous application for delivering a rich interface, abstracting, and enabling communication with external services based on HTTP requests [51]. It provides direct configuration and integration with *ExecutorService*, which allows asynchronous processing of the response.
- *WebClient* was utilized in the reactive application due to being part of the WebFlux module, providing a non-blocking implementation, advanced features and support for reactive data flows [28]. The interface is more lightweight and streamlined compared to other implementations given its tight integration within the Spring ecosystem.

API Gateway

- **Zuul v1** is a completely integrated extension within the Java ecosystem, developed by Netflix, picked in the asynchronous model to perform dynamic routing [52]. One main issue is its blocking nature, prompting the development of a new, non-blocking variant.
- **Spring Cloud Gateway** is the counterpart developed by the Spring team, offering advanced routing capabilities, improved performance, and an extended set of functionalities within the reactive ecosystem [53].

Database Management

- **PostgreSQL** was utilized as the relational database in both models, extending conventional SQL with object-oriented capabilities and advanced functionality for data storage and manipulation. It supports complex data types including JSON, arrays, and key-value pairs, while also allowing the definition of custom types and functions [54].
- **Hibernate/JPA** is applied in the asynchronous model, providing a conventional blocking approach for data access and manipulation, object-relational mapping, and database schema generation, which allows for emphasis on business logic [55, 56].
- **R2DBC** provides streamlined connectivity and non-blocking interactions with the database, representing a specification that provides a reactive driver for PostgreSQL, integrates, and is even preferred in the development of fully reactive applications [9].

Data Caching

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 November 2024 doi:10.20944/preprints202411.1467.v1

- **Redis** is used across both models for its high-performance caching capabilities, significantly reducing response times and load on databases and application components. Redis supports both blocking and non-blocking operations, making it a versatile choice for both systems [57].

Framework Architecture

In addition to the specific technologies, another important aspect is the significant architectural difference residing right at the framework level between the asynchronous and reactive approaches, specifically the Spring MVC and Spring WebFlux variants, as depicted in Figure 3.

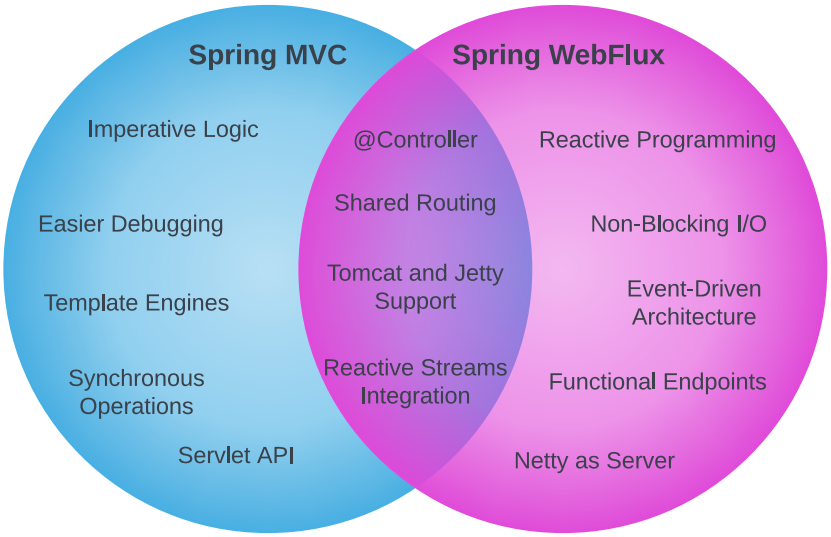


Figure 3. Comparison between Spring MVC and Spring WebFlux [58].

Spring MVC is the conventional variant of the framework, based on an imperative programming model. It leverages blocking I/O and utilizes a thread for each request [27]. Although this model is simple to understand, implement, and debug, it can quickly become inefficient when facing a large number of concurrent requests, due to the blocking resources and the complexity associated with managing the lifecycle of threads.

Spring WebFlux, on the other hand, embraces the reactive model, using non-blocking I/O, the execution is based on an event loop [28]. The pool of threads is reduced, and unlike the conventional model, there is no direct mapping between active threads and requests, which fundamentally enables this model to efficiently handle a large number of concurrent requests using a limited number of threads. By leveraging reactive data types and functional paradigm principles, the reactive variant makes it possible to manage data flows and backpressure optimally, while providing much greater scalability under conditions of intensive load.

5.3. Project Structure

The structure of the proposed solution will be analyzed from two perspectives: the database structure, focusing on the role of the tables and their relationships, and the organization of the backend projects, describing the defined packages and the general role of the classes within each.

Database Structure

The database structure supporting the application's services and functionalities is defined according to the diagram in Figure 4.

 doi:10.20944/preprints202411.1467.v1

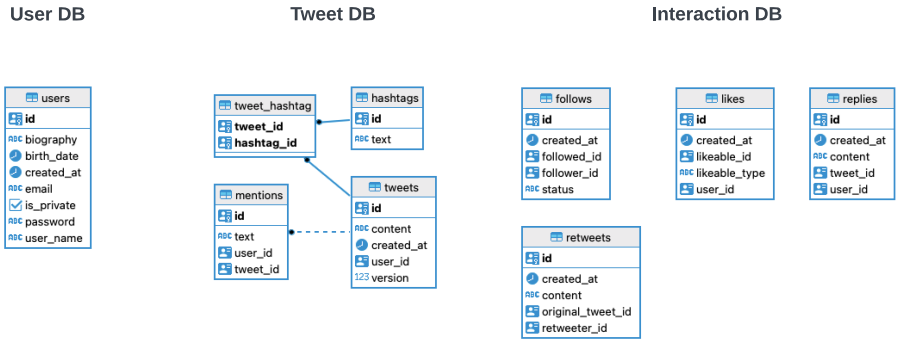


Figure 4. Database diagram.

- The “users” table defines the list of user accounts within the application.
- The “tweets” defines the list of tweets within the application. This table has a many-to-one relationship with the “mentions” table, using the foreign key “tweet_id” to reference the associated entity.
- The “hashtags” table defines the list of hashtags extracted from tweet contents.
- The “mentions” table defines the user mentions within tweets. This table has a many-to-one relationship with the “tweets” table using the foreign key “tweet_id” to reference the associated entity.
- The “tweet_hashtag” table defines the join enabling the many-to-many relationship between tweets and hashtags. This table has a many-to-one relationship with the “tweets” using the foreign key “tweet_id” and with the “hashtags” table using the foreign key “hashtag_id” to reference the associated entities.
- The “follows” table defines the *follow* relationships between users.
- The “likes” table stores the list of likes given by users to existing tweets or replies.
- The “replies” table stores the list of submitted replies by users to existing tweets.
- The “retweets” table stores the list of redistribution of existing tweets by the users.

Project Organization

The asynchronous and reactive projects share a similar structure for organizing the packages, with the primary aim of capturing and separating the responsibilities of each module, thereby facilitating maintenance and code flexibility. Figure 5 presents the project’s detailed package organization structure.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 November 2024 doi:10.20944/preprints202411.1467.v1

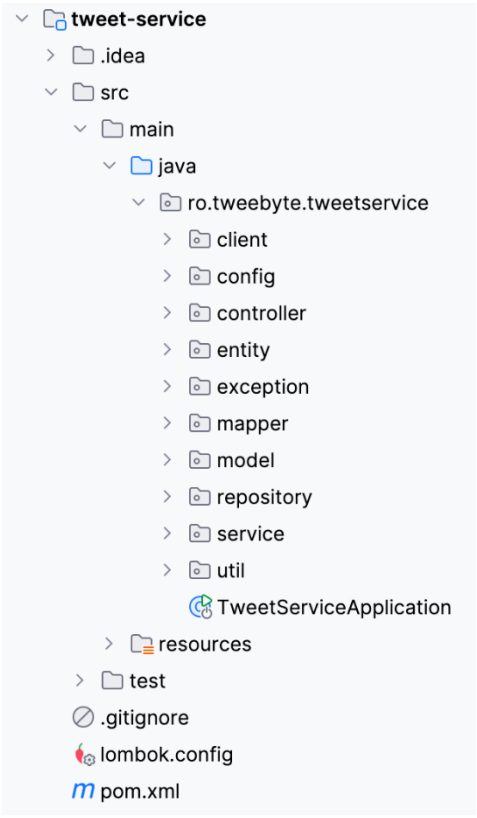


Figure 5. Package structure for backend services.

The main packages within the projects are:

- The "client" package contains classes for communicating with the other services in the application.
- The "config" package centralizes config classes for various components of the application, such as caching, security, etc.
- The "controller" package includes classes that expose the application's entry points, receiving and processing user requests.
- The "entity" package contains the entities or data models equivalent to the objects persisted in the database.
- The "exception" package defines specific exceptions used in the application for error handling.
- The "mapper" package groups the classes that provide the conversion between data transfer objects (DTO) and entities.
- The "model" package contains models used for transferring information, without directly exposing persistent entities.
- The "repository" package includes interfaces and implementations for accessing persistent data, and managing database operations.
- The "service" package contains the core business logic of the application, which is implemented in classes that organize and sequence calls to repositories and other necessary components.
- The "util" package centralizes utility classes, used in various parts of the application, such as validation methods, conversions, etc.

5.4. Migration Aspects and Stages

The main aspects and stages of the migration process are described in Table 3, with an emphasis on clearly outlining the available counterpart in each paradigm for specific elements.

Table 3. Migration aspects and stages.

Aspect	Asynchronous	Reactive

 doi:10.20944/preprints202411.1467.v1

Programming model	Imperative	Reactive
Java version	Java 21	Java 21
Framework	Spring Boot 3	Spring Boot 3
Single result	*CompletableFuture<T>*	*Mono<T>*
Collection / Data stream	*CompletableFuture<Collection<T>>*	*Flux<T>*
Thread pools	*ExecutorService*	*Schedulers*
API	Standard Java API	Project Reactor, Spring WebFlux
Error handling	*try-catch, CompletableFuture, exceptionally()*	*Mono / Flux, onErrorReturn(), onErrorResume(), onErrorMap()*
Operation retries	Programmatic	Native, *Mono / Flux retry()*
Blocking operations	*CompletableFuture get()*	*Mono / Flux block()*
Data access	JDBC, Hibernate, JPA	R2DBC
Web server	Tomcat	Netty
HTTP communication	Java *HttpClient*	Spring *WebClient*

Migrating from an asynchronous to a reactive model requires a series of changes at the level of code and infrastructure. Although the transition can be complex depending on the size of the project and the technologies employed, the long-term benefits are considerable, as evaluated in this research.

6. Implementation Details

6.1. Updating Java and Spring Versions

Recent versions of Java and Spring provide access to the latest functionality, optimizations, and enhancements while ensuring long-term relevance, compatibility, and essential security fixes against numerous vulnerabilities. Upgrading to the latest versions is therefore crucial for a successful migration process and lays a solid foundation for future adjustments. However, it's worth noting that most industry applications are still running on lower versions of Java. While the asynchronous and reactive projects are based on Java 21, many organizations may find it more practical to adopt an intermediate step.

Java 17 serves as a bridge between the two major versions of Spring Boot, namely Spring Boot 2 and Spring Boot 3. It is compatible with both, whereas Java 11 is not supported by Spring Boot 3. Therefore, for existing applications, particularly those on older Spring Boot versions, the first step should be updating to Java 17. This transition allows for a smoother migration path, as Java 17 facilitates compatibility while mitigating potential issues arising from outdated or deprecated specifications. This approach ensures that applications remain in line with industry standards while taking advantage of the new features and improvements available in later versions. This step should also be reasonably straightforward, as most Java versions are designed to maintain very good backward compatibility, facilitating a smooth transition.

One of the most significant changes is the management of Jakarta EE dependencies. Newer versions of Spring Boot no longer use "javax" dependencies, necessitating a migration to their "jakarta" equivalents. As a result, developers need to update the import statements in the source code wherever these old specifications are used. While this transition might seem straightforward in the context of the own codebase, complications often arise with external dependencies because there is no interoperability between the two specifications.

 doi:10.20944/preprints202411.1467.v1

The most reliable approach is to update the maintained dependencies to their latest versions, which should be compatible with Jakarta EE, and then thoroughly test the application to ensure that its functionality remains intact. However, if an immediate upgrade isn't feasible, either because a compatible version of the dependency isn't available or because the upgrade process is too complex, a temporary solution might involve using tools such as Eclipse Transformer [59]. This tool works by directly modifying the bytecode of the existing JAR files, converting "javax" references to "jakarta" equivalents without requiring access to the original source code.

This process begins by identifying all dependencies that still rely on the "javax" namespace. Once identified, Eclipse Transformer or a similar utility can be used to rewrite the bytecode within these JAR files, adapting them to the new Jakarta specifications. After the transformation, the modified JAR files are then integrated back into the project, replacing the old versions. Although this approach allows you to continue using older dependencies temporarily, it's crucial to conduct thorough testing of the application to ensure that these transformations have not introduced any issues or altered the application's behavior. The transformation process must be repeated for each affected dependency, which can make the procedure quite time-consuming and complex. Moreover, because this process does not guarantee complete success, extensive testing is essential to confirm that the application's functionality is preserved.

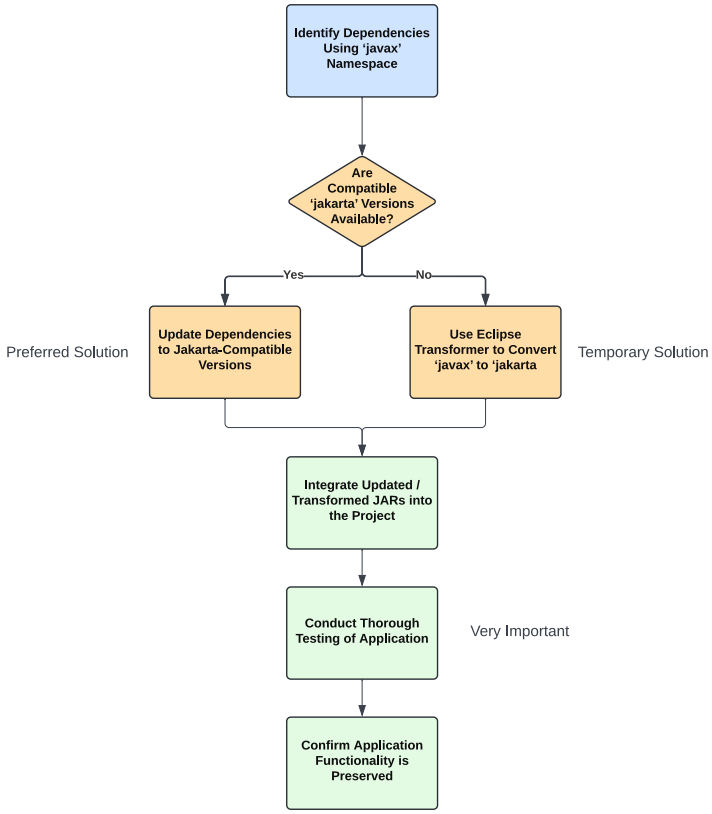


Figure 6. Migration Process from "javax" to "jakarta" Dependencies.

Another change that may potentially impact the operability of the application is the changed default behavior for a trailing slash in HTTP request paths. In the previous version of Spring, a controller method annotated with *@GetMapping("/feed")* would have accepted requests coming in on both the "/feed" path, as in the example, and also on the "/feed/". This default behavior is no longer present in the new version, resulting in an HTTP 404 error that can cause instability and operational disruptions for clients establishing communication using the trailing slash version. Nevertheless, the problem can be solved by reintroducing the mechanism that treats both variants equivalently, with the procedure being similar for both paradigms. It is required to define a configuration class by implementing the *WebMvcConfigurer* interface in the case of the asynchronous application and

 doi:10.20944/preprints202411.1467.v1

WebFluxConfigurer in the case of the reactive one, where the preference to treat both paths as equivalent is redefined programmatically.

The Java and Spring release upgrading process also includes other steps, depending on the specifications and technologies employed within each project, but these steps are an integrated part of the process, independent of the selected development paradigm.

6.2. Implementation and Management of Components

The Spring Framework promotes the organization of applications using a component structure, based on the design pattern of Inversion of Control/Dependency Injection [60]. The main benefit of this approach is the simplified management of dependencies and application configurations. The framework defines a set of specific stereotypes, such as *Controller*, *Service*, and *Repository* by using annotations, to properly differentiate between each of their purposes and functionalities [27].

User requests will be received and handled within the controller, which is the main entry point within the application. In terms of code structure and syntax, the differences at this level are minimal, with the only notable difference being the type of data used to return the computed results, as can be seen in Figure 7.

The responsibility of aggregating, orchestrating, and performing the application logic belongs to the service component, which is the intermediary of all interactions between the user and the data necessary to fulfill each request. Due to the role carried out by this component, most of the defined code logic is at this level, which makes service migration the main topic under analysis to understand the differences and the necessary adaptation steps in the reactive context.

Asynchronous programming structures the flow of the program as chains of asynchronous operations that use the results from the preceding level. *CompletableFuture* is intended to store both single results from a given level as well as collections of objects. The exposed methods allow for streamlined manipulation and further processing of the results, simplifying the chaining of asynchronous operations [6]. Methods such as *supplyAsync* and *runAsync* are used to schedule tasks for execution in a separate thread that will not block the main one, and the result processing steps will be defined using methods such as *thenApply*, to apply a direct transformation to the result when it becomes available, and *thenCompose*, to connect the current operation with another asynchronous operation whose execution depends on the result at the current level. Context switching can be achieved both at the beginning of the operation sequence and between steps using variations such as *thenApplyAsync*, which will not restrict the execution of subsequent steps to be performed on the same thread. In addition, such methods allow the delegation of a specific thread pool to handle the processing by accepting an *ExecutorService* as a method parameter.

The reactive model is fundamentally based on managing tasks as a continuous stream of data, using two fundamental types that encapsulate and allow non-blocking manipulation of the elements. *Mono* is used to issue at most one element, while *Flux* can also issue multiple values, and both primitives may also issue no elements at all. The data is processed in a functional style by applying a series of transforming, filtering, or combining operators to the output streams. For scenarios requiring the transformation of each element within the stream into a different shape or format, the conversion or transformer function is applied to each element available in the stream using the *map* operator, while *flatMap* allows the application of a function that issues, in turn, another reactive data stream, the computation of which is dependent on the previous result, thereby also "flattening" the resulting stream in the process. In certain situations where data streams may not output any elements, a default value or an error can be specified using operators such as *switchIfEmpty*, while combining elements of the same stream can be achieved by applying a function using the *reduce* operator. Excluding certain elements that do not fulfill specific processing conditions can be accomplished by applying a function that tests compliance with these conditions using the *filter* operator.

In addition to the previously outlined scenarios, applications must also accommodate the combination of data from multiple sources, which involves launching and aggregating results from multiple operations asynchronously. Conversely, in a reactive context, this involves combining multiple streams. The *allOf* method accepts multiple instances of *CompletableFuture* as parameters,

 doi:10.20944/preprints202411.1467.v1

enabling the orchestration of parallel scenarios that depend on the completion of previously initiated actions. Adaptation to the reactive paradigm is achieved using the *zip* operator, available for both fundamental types, which allows combining flows from different sources into a single final entity.

6.3. Communication Using HTTP Clients

Modern architectures based on microservices require effective communication between different system components [13]. Within the developed system, communication is achieved using HTTP requests, employing dedicated client implementations. The selected implementation primarily takes into account interoperability with the types and mechanisms in each paradigm while also providing efficient interactions.

For the asynchronous application, it was decided to use the HTTP client provided by Java. This interface integrates with the primitives specific to the paradigm and offers robust support for asynchronous operations. Requests can be launched asynchronously using methods such as *sendAsync*, with the results made available in a *Future* object. The interface also allows the configuration of a dedicated *ExecutorService* to achieve greater control and flexibility in concurrency management. The main downside of this solution is the lack of integration with Spring, with response transformations and data conversions being handled programmatically, introducing additional logic and complexity.

In the reactive approach, it was opted for the usage of *WebClient* from Spring WebFlux. Similar to *HttpClient*, it provides native integration with *Mono* and *Flux* reactive types but also supports automatic serialization and deserialization of data, increasing the clarity and efficiency of the implementation. As an integrated part of the reactive ecosystem, *WebClient* is developed with paradigm principles in mind so that requests and responses can be handled in a non-blocking way, making it a natural choice for scenarios implying heavy traffic conditions and high-performance requirements.

6.4. Handling System Errors

Software systems are prone to unpredictable behavior and errors, but maintaining resilience and reliability is not possible without effective management. There are several strategies for handling exceptional cases, and implementation methods are also different depending on the paradigm.

The standard approach implies the provision of a mechanism for computing an alternate response or an error handler that can vary in complexity, used at the moment when the application encounters the exceptional case. While this addresses the punctual error and provides a degree of control over the execution flow, it does not deal with the root cause and does not facilitate the recovery of the system from its unstable state.

CompletableFuture provides the *exceptionally* method for handling errors in asynchronous operations, while *Mono* and *Flux* provide equivalent methods for handling errors in reactive flows, such as *onErrorResume*. If the primary process fails, a default value can be returned or an attempt can be made to retrieve the data from alternate sources. For example, in the scenario where the *getFollowedIds* method fails, the data available in the cache is inspected by calling the *getFollowedUsersFromCache* method. This approach can be further extended by executing the operations or streams that provide access to the same required data in parallel, e.g. a simultaneous database and cache lookup, instead of waiting for the regular version to fail. *CompletableFuture*'s *anyOf* method allows asynchronous operations to be launched simultaneously, with the result being determined by the first completed operation, while the *firstWithValue* method provided in the reactive context waits for the first successful response, ignoring the failed ones.

Occasionally, the errors encountered in applications may be temporary or non-deterministic, caused by fluctuating conditions. In such situations, retrying the operations is an elegant and practical solution for dealing with exceptional cases without requiring the development of specific and complex solutions.

The asynchronous specification does not provide any implementation for retrying failed operations, requiring a programmatic solution such as recursion or the utilization of existing

 doi:10.20944/preprints202411.1467.v1

framework implementations such as Spring *Retry*. Instead, the *retry* operator is available in the reactive context, which allows for detailed configuration by specifying the number of retries, timing strategy, and re-execution conditions for the operation.

However, a more robust approach for dealing with errors in unpredictable situations, such as a sudden spike in traffic, is to leverage the "Circuit Breaker" pattern [61], which is designed to detect problems in real-time and prevent errors from propagating throughout the system. A fallback method is configured, and when the main component becomes non-functional resulting in the number of errors increasing, automatic redirection of requests to the fallback method takes place, ensuring a minimum level of operation and avoiding a total system outage. Periodically an attempt will be made to gradually restore traffic through the main flow, and once the errors have completely disappeared, the fallback method is no longer invoked and the system returns to normal operating conditions. The Resilience4j library provides versions compatible with both variants [62]. Figure 7 illustrates an example of use in the reactive application.

```
@CircuitBreaker(
    name = "followedIdsCircuitBreaker",
    fallbackMethod = "getUserFeedWithCachedFollowed"
)
public Flux<TweetDto> getUserFeed(UUID userId) {
    return interactionClient.getFollowerIds(userId)
        .collectList()
        .flatMapMany(
            tweetRepository::findByUserIdInOrderByCreatedAtDesc
        )
        .flatMap(this::enrichTweetDto);
}

public Flux<TweetDto> getUserFeedWithCachedFollowed(
    UUID userId, Throwable t
) {
    String key = FOLLOWED_CACHE + "::" + userId;
    return redisTemplate.opsForValue().get(key)
        .map(v -> {
            try {
                return objectMapper.readValue(
                   v, new TypeReference<List<UUID>>() {}


                );
            } catch (JsonProcessingException e) {
                throw new RuntimeException(e);
            }
        }).flatMapMany(Flux::fromIterable)
        .collectList()
        .flatMapMany(
            tweetRepository::findByUserIdInOrderByCreatedAtDesc
        )
        .flatMap(this::enrichTweetDto);
}
```

Figure 7. Example of using Circuit Breaker in reactive applications.

In the code above, the *getFollowerIdsFromCache* method will be called automatically if a high number of errors are detected in the communication with the interaction microservice. Once appropriate functionality is restored and the service can accept requests again, the traffic will be completely re-routed back through the default method.

6.5. Task Scheduling Mechanisms

 doi:10.20944/preprints202411.1467.v1

Scheduling mechanisms enable applications to automatically perform periodic tasks, such as cleaning up redundant resources or updating cached data, which are common scenarios in distributed microservice-based systems. Effective scheduling of periodic tasks has an important role to play in maintaining functional integrity, quality, and system performance over time. Although framework solutions are available, the required mechanisms are already embedded within both paradigms. One scenario arising in the resulting project was cleaning up interactions after deleting a tweet since the entries are stored in different databases, which can lead to potential inconsistencies. However, the temporary presence of such inconsistencies does not have any visible impact on the user. By scheduling periodic tasks to be performed automatically, one can avoid programmatic cleanup for each deletion action, therefore reducing the additional load on the system.

The asynchronous model employs the dedicated *ScheduledExecutorService* implementation to regularly plan and execute such tasks, granting detailed control over job scheduling but also providing the ability to define the initial execution time and the intervals between recurrent executions.

Project Reactor provides an elegant and efficient mechanism for planning tasks that naturally integrates within the reactive context using *Scheduler*s, with combinations of operators such as interval and *flatMap*, allowing in turn a fine-grained specification of execution conditions and frequency.

Although the migration process reveals some differences at the code level, certain similarities, and a common structure are preserved: the usage of an *Executor* or *Scheduler* instance to configure the execution, the definition of jobs using the available interfaces and proper lifecycle management, correctly shutting down jobs and deallocating resources by leveraging methods such as *shutdown* or *dispose*.

6.6. Database Interactions

The asynchronous solution relies on a blocking model for database access but benefits from a full mapping of Java classes to the database structures, thanks to a comprehensive set of functionalities provided by Hibernate and JPA [55]. The database schema initialization happens automatically at runtime once the application is launched, based exclusively on metadata extracted from annotations.

The integration with Hibernate allows the automatic generation of unique identifiers for each entity, e.g. annotating the *id* field of the *TweetEntity* with *@GeneratedValue*, and the definition and management of data relationships, including those made using link tables by specifying association columns. In this example, the relationships between tweets, mentions, and hashtags tables are defined using the *@OneToMany*, *@ManyToMany*, *@JoinTable*, and *@JoinColumn* annotations. In addition, data integrity is ensured thanks to support for cascading operations, but also the automatic removal of entities whose parent is deleted, achieved in this scenario by specifying *CascadeType.ALL* and enabling *orphanRemoval* for mentions and hashtags.

Implementing database access and performing CRUD (Create, Read, Update, Delete) operations for managed entities is achieved by defining repositories that extend the *JpaRepository* interface, considerably simplifying the integration and manipulation process.

By providing an extensive suite of predefined methods and the ability to define additional queries by simply respecting a method naming convention, e.g. the repository method *findByTweetIdOrderByCreatedAtDesc* which allows for retrieval of tweet replies in descending order by their creation date, the functionality available in the asynchronous model through the integration with JPA enables quick and efficient access to stored data, including the ability to execute queries of considerable complexity.

The definition of native queries allows direct use of SQL syntax, providing complete control over execution beyond the abstraction provided by the framework, which may allow certain scenarios to adjust and optimize the behavior as well as to use particular functionalities specific to the database system. In the previous example, a native query was defined as the *searchUsers* method to look up users by their names in a case-insensitive way. Last but not least, JPQL queries, such as the one

 doi:10.20944/preprints202411.1467.v1

defined to retrieve the top reply of a tweet by calling the method *findTopReplyByLikesForTweetId*, allow for the extraction of information from entities directly in the desired format through direct access to public model class constructors, and functionalities such as paging and *EntityGraph*, which were employed to eagerly fetch the hashtags and mentions only in this scenario since they were always required, provide granular control over loading strategies and efficient data access management.

Figure 8 includes a subset of methods and queries that were used in the asynchronous implementation, in addition to the predefined ones.

```
List<ReplyEntity> findByTweetIdOrderByCreatedAtDesc(UUID tweetId);

@EntityGraph(attributePaths = {"mentions", "hashtags"})
List<TweetEntity> findByUserIdInOrderByCreatedAtDesc(
    List<UUID> userIds
);

@Query(
  value = "SELECT * FROM users WHERE user_name ILIKE '%:query%'",
  nativeQuery = true
)
List<UserEntity> searchUsers(String query);

@Query(
  "SELECT NEW ro.tweebyte.interactionservice.model.ReplyDto" +
  "(r.id, r.userId, r.content, r.createdAt, CAST(COALESCE(COUNT(l), 0)
AS long)) " +
  "FROM ReplyEntity r " +
  "LEFT JOIN LikeEntity l ON r.id = l.likeableId AND l.likeableType =
'REPLY' " +
  "WHERE r.tweetId = :tweetId " +
  "GROUP BY r.id, r.userId, r.content, r.createdAt " +
  "ORDER BY COUNT(l) DESC, r.createdAt DESC"
)
Page<ReplyDto>   findTopReplyByLikesForTweetId(@Param("tweetId")   UUID
tweetId, Pageable pageable);
```

Figure 8. Examples of methods for database access in asynchronous applications.

Reactive applications can only reach their performance potential in a fully integrated and reactive ecosystem, where database access is one of the most common scenarios across current systems. Due to these constraints, adopting a reactive database access solution is a mandatory requirement. Otherwise, there is the risk of completely inheriting the complexity and overhead associated with the migration process, without being able to benefit from any of the advantages and performance improvements at a high potential level.

R2DBC is the specification selected for defining a fully responsive system and managing database interactions in a non-blocking way. One first limitation is the lack of functionality for initializing the database schema using annotations. The alternative is to explicitly define native SQL code and store it in a file within the application resources, hence setting up automatic execution at the application startup.

The entities defined with R2DBC are significantly stripped down, providing only minimal functionality for specifying the properties and metadata required for mapping these objects to the corresponding database tables. This might reduce some overhead and complexity, but it would also require a programmatic approach to achieve some functionalities comparable to ORMs. Consequently, any potential benefits could be mitigated, possibly making the system even more complex and inefficient.

Recent implementations of R2DBC provide support for version control and transactional mechanisms, but currently, there is no native support for entities using composite primary keys. This limitation required the definition of an additional unique identifier and the creation of a composite

 doi:10.20944/preprints202411.1467.v1

index for the fields in question, intending to preserve the equivalence following the migration process.

The implementation of data access and execution of CRUD operations is performed using the *ReactiveCrudRepository*, which is the element that allows non-blocking interaction based on reactive types, ensuring seamless flow and access to data in an efficient and optimized way.

R2DBC offers a subset of predefined queries and supports custom query definitions via method naming conventions. However, it lacks several advanced features that JPA provides, such as support for JPQL (Java Persistence Query Language), which facilitates complex, object-oriented queries. Additionally, R2DBC does not offer built-in support for caching, lazy loading, or managing relationships between entities, requiring considerable adjustments to the source code to simulate and achieve functional equivalence through programmatic management, which can be time-consuming and error-prone for bigger projects, as illustrated in Figure 9.

```
private Mono<TweetDto> enrichTweetDto(TweetEntity tweetEntity) {
    Mono<Long> likesMono = interactionClient.getLikesCount(tweetEn-
tity.getId());
    Mono<Long> repliesMono = interactionClient
         .getRepliesCount(tweetEntity.getId());
    Mono<Long> retweetsMono = interactionClient
         .getRetweetsCount(tweetEntity.getId());
    Mono<ReplyDto> replyMono = interactionClient
         .getTopReply(tweetEntity.getId());

    Mono<List<HashtagEntity>> hashtagsMono = hashtagRepository
        .findHashtagsByTweetId(tweetEntity.getId()).collectList();
    Mono<List<MentionEntity>> mentionsMono = mentionRepository
        .findMentionsByTweetId(tweetEntity.getId()).collectList();

    return Mono.zip(likesMono, repliesMono, retweetsMono, replyMono,
hashtagsMono, mentionsMono)
        .map(data -> tweetMapper.mapEntityToDto(
            tweetEntity, data.getT1(), data.getT2(), data.getT3(),
data.getT4(), data.getT5(), data.getT6()
        ));
}
```

Figure 9. Example of loading entity relationships in reactive applications.

The analyzed project used complex and ORM-specific features, but it was manageable to integrate R2DBC after some adjustments and trade-offs. However, many limitations and little or no support for certain functionalities were observed, which may be a determining factor when considering the migration process, especially within systems that are heavily reliant on complex or specific database access abstraction functionalities currently available only in mature technologies.

7. Evaluation of Results

The evaluation of results gathered from the migration process focuses on the functional equivalence of the two applications and performance analysis in different use cases. Multiple test scenarios have been analyzed and various metrics have been compared to provide an overview of the advantages and limitations of each paradigm.

7.1. Functional Equivalence

An inherent part of the migration process involved ensuring that both implementations produced identical results when performing the same operations. This consistency goes beyond preserving existing functionality but also ensures accurate performance analysis, as comparing performance indicators would be unreliable due to functional differences. Achieving functional equivalence required a robust testing methodology, featuring tests at different levels of abstraction and ensuring correct execution. Some tests validate individual components, ensuring smaller code

 doi:10.20944/preprints202411.1467.v1

units perform as expected, while others evaluate overall system behavior, emphasizing interactions and integrations by simulating real user actions. This approach guarantees that both individual components and the system as a whole are properly evaluated, preserving functionality throughout the transition to the reactive model.

Unit tests, created using JUnit 5 [63], were essential to check the accuracy and consistency of individual units of code. Aiming to be as similar as possible in both implementations, these tests were tailored only to account for differences such as data types and methods specific to each paradigm. Each test is structured to prepare the conditions and prerequisites, execute the tested functionality, and verify the expected results, thereby ensuring reliable testing of individual components throughout the migration process. Figure 10 illustrates an example of a unit test for the user login functionality in the asynchronous approach.

```
@Test
void testLoginSuccess() throws ExecutionException,
                               InterruptedException {
    // arrange
    UserLoginRequest request = new UserLoginRequest(
         "user@example.com", "correctpassword"
    );
    UserEntity userEntity = new UserEntity();
    userEntity.setEmail("user@example.com");
    userEntity.setPassword("$2a$10$SomeHashedPasswordHere");
    userEntity.setId(UUID.randomUUID());

    when(userRepository.findByEmail(any()))
        .thenReturn(Optional.of(userEntity));

    when(
        encoder
            .matches(request.getPassword(), userEntity.getPassword())
    ).thenReturn(true);

    // act
    AuthenticationResponse result = authenticationService
         .login(request).get();

    // assert
    assertNotNull(result);
    assertFalse(result.getToken().isEmpty());
    verify(userRepository).findByEmail(request.getEmail());
    verify(encoder)
        .matches(request.getPassword(), userEntity.getPassword());
```

Figure 10. Example of unit test covering the user login in the asynchronous approach.

The introduced test example validates the user login functionality by setting up a login request with valid credentials and a mocked user entity. It configures the repository and the encoder mocks to return the expected values, simulating real-world conditions. Following this setup, the test calls the actual login method on the authentication service, concluding with response checks to ensure a successful login. Assertions ensure that the response is not null, that the token is present, and that the correct methods were invoked with the expected parameters, confirming the login process's accuracy and reliability.

Cucumber [64] was also employed to define Behavior-Driven Development (BDD) automated tests. Written in Gherkin, these tests describe software behavior in clear and understandable terms, remaining agnostic of implementation details so they can be used across both versions of the application. These testing scenarios focus on assessing functional consistency by outlining the expected system behavior from a user's perspective, described in a human-readable language. The abstraction from the code streamlines the validation of overall system functionality and integration

 doi:10.20944/preprints202411.1467.v1

points, ensuring a consistent user experience while also serving as a helpful instrument in identifying and addressing any discrepancies during the development and migration process. Figure 11 illustrates a partial Cucumber feature file for tweet management.

```
Feature: User Tweet Management

    Scenario: Register a user, create a tweet, and verify tweet existence
        Given a new user is registered with valid details
        When the user posts a valid tweet
        And the user retrieves their tweets
        Then the user's tweet should be included in the retrieved tweets

    Scenario: Attempt to create a tweet with no content
        Given a new user is registered with valid details
        When the user attempts to post a tweet with no content
        Then the response should indicate a content validation error

    Scenario: Attempt to create a tweet with insufficient content length
        Given a new user is registered with valid details
        When the user attempts to post a tweet with insufficient content
        Then the response should indicate a minimum content length error

    # other scenarios
```

Figure 11. Example of a piece of Cucumber feature for tweet management testing scenarios.

This Cucumber feature tests the tweet management functionalities, including one scenario that verifies that a new user can be registered, create a valid tweet, and retrieve it. It also includes scenarios such as trying to create a tweet with no content or insufficient content length, which should trigger specific content validation errors. By incorporating such negative checks and edge cases, these tests ensure the system can process user requests as expected, but also properly validate inputs and handle unexpected situations effectively.

To execute the Cucumber feature files, the steps must be followed to define the actual translation from Gherkin scenarios to executable Java code. These step implementations run independently against the different backends, ensuring appropriate validation regardless of the underlying implementation. Figure 12 presents two examples of step implementations, showcasing the actual mapping between the high-level descriptions and the required interactions to perform the respective actions.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 November 2024 doi:10.20944/preprints202411.1467.v1


```
public void registerNewUser() throws JsonProcessingException {
     String url = USER_SERVICE_BASE_URL + "/auth/register";
     HttpHeaders headers = new HttpHeaders();
     headers.setContentType(MediaType.MULTIPART_FORM_DATA);

     LinkedMultiValueMap<String, String> body =
             new LinkedMultiValueMap<>();
     body.add("userName", generateRandomUsername());
     body.add("email", generateRandomEmail());
     // addition of the other fields

     HttpEntity<LinkedMultiValueMap<String, String>> requestEntity =
         new HttpEntity<>(body, headers);
     ResponseEntity<String> response = restTemplate
         .postForEntity(url, requestEntity, String.class);
     assertEquals(HttpStatus.OK, response.getStatusCode());

     AuthenticationResponse authResponse = objectMapper
         .readValue(response.getBody(), AuthenticationResponse.class);
     userId = UUID.fromString(
         getClaimFromToken(authResponse.getToken(), "user_id")
     );
}

@Then("the user's tweet should be included in the retrieved tweets")
public void verifyTweetInclusion() throws Exception {
     TweetDto[] tweetsArray = objectMapper
         .readValue(response.getBody(), TweetDto[].class);
     List<TweetDto> tweets = Arrays.asList(tweetsArray);

     assertTrue(
          tweets
               .stream()
               .anyMatch(
                   tweet ->
                        "This is a valid tweet."
                             .equals(tweet.getContent()) &&
                        tweetId.equals(tweet.getId())
               )
     );
}
```

Figure 12. Example of step implementations for Cucumber feature scenarios.

Both method implementations are annotated and parameterized to align with the steps defined in the feature file. As per the first scenario listed, the *registerNewUser* method is connected to the *Given* step, while the *verifyTweetInclusion* is tied to the *Then* step. These annotations and the specified parameters establish the direct link between the Gherkin steps and the executable code, ensuring that each scenario step triggers the appropriate method.

By implementing an average of approximately 150 tests per service, including both unit tests and also various features and scenarios, an estimated code coverage baseline of over 90% was registered, demonstrating that the migration from asynchronous to reactive specifications did not compromise the application's functionality. The test suites consistently produced the same expected results across both implementations, confirming functional equivalence and reinforcing the feasibility of an effective and complete transition to the reactive approach.

7.2. Performance Analysis

The performance analysis is focused on comparing resource utilization and efficiency for each approach. For this study, both static parameters as well as the dynamics of the applications under concurrent conditions were considered.

Static and Initialization Metrics

 doi:10.20944/preprints202411.1467.v1

These properties outline a preliminary overview of the first differences resulting after the migration process, summarized in Table 4. Although the impact concerning behavior during operation is limited, some of these characteristics might still be relevant considering some convenience aspects of CI/CD (Continuous Integration and Continuous Delivery) processes [65].

Table 4. Static and initialization metrics.

Microservice (paradigm)	Heap memory consumption (MB)	Startup time (ms)	JAR size (MB)
User Service (async)	61	2363	52
User Service (reactive)	38	1522	37
Tweet Service (async)	62	2921	61
Tweet Service (reactive)	45	1874	45
Interaction Service (async)	70	3526	61
Interaction Service (reactive)	51	2107	41

The memory consumed by reactive applications is about 35% less compared to the asynchronous ones, which would allow for faster scheduling inside a cluster, reducing deployment duration while also optimizing resource allocation and utilization inside orchestration platforms such as Kubernetes.

The startup time for reactive microservices is approximately 35% lower than that of asynchronous variants, contributing to an optimal transition in the case of a rolling deployment strategy, minimizing downtime, and even preventing readiness errors when deploying several replicas, therefore ensuring continuity of service operations, which is important in distributed environments focused on maintaining high availability and reliability.

The size of the resulting artifacts is also notably smaller for reactive applications, with a difference of about 30%, mainly due to the smaller number of dependencies required. The deployment of these smaller artifacts is inherently accelerated, simplifying the storage processes on various platforms and reducing the size and download times of container images, such as those used in Docker.

Evaluation Under Concurrent Load Conditions

Analyzing the performance of the two approaches under concurrent conditions involved both the selection of appropriate tools for simulating and monitoring the traffic with a variable number of users and the definition of a test plan that would provide relevant data and a fair evaluation, minimizing errors and avoiding sudden overload.

The simulation of concurrency with varying numbers of users was performed using Apache JMeter [66]. The testing started with a 60-second ramp-up period per scenario, during which the number of requests was gradually increased. The actual tests were run over 3-minute durations, with the ramp-up period data deliberately excluded from the reports as the focus was on monitoring and analyzing the performance of both systems after reaching a stable state at the desired level of concurrency.

The metrics were collected using the reports generated by JMeter, which captured a detailed overview of the response times and throughput rates, but also using VisualVM, which allowed for the monitoring and recording of resources such as memory consumption and CPU usage, together offering insights into the system's performance under high concurrency and the ability to handle an increasing number of requests over time, ensuring a solid evaluation of both efficiency and scalability.

 doi:10.20944/preprints202411.1467.v1

The first analyzed scenario implied the fetching of a user profile summary. A list of user IDs extracted from the database was loaded into the test plan defined with JMeter, and virtual users launched concurrent requests using distinct IDs. The plots illustrating the heap memory and throughput variation are displayed in Figure 13.

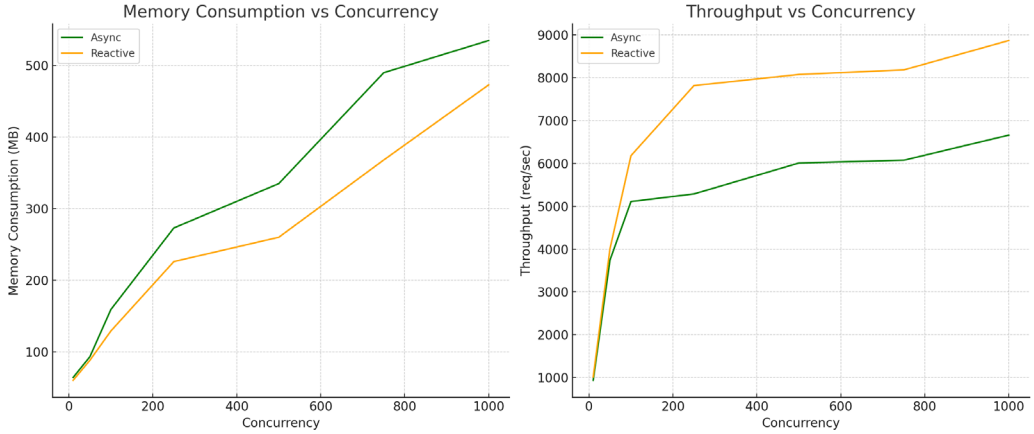


Figure 13. Performance results for fetching user profile summaries.

The reactive model achieved better performance, with reduced heap memory consumption by up to 12% during high concurrency scenarios (473 MB vs. 535 MB). CPU usage remains similar between the two models, with the asynchronous variant recording marginally lower values gradually as the concurrency increased. The average response time is lower for the reactive application and the throughput is superior, with 8867 requests/s compared to only 6659 requests/s for the asynchronous approach at 1000 concurrent users, up to 33% higher.

The second investigated scenario involved simulating user interactions within the application, specifically the action of following other users. In this setup, both a list of user IDs that will follow and a list of user IDs to be followed were added to the JMeter test plan, with multiple unique application user identities used across different virtual users. This approach evaluated the performance impact of multiple virtual users executing follow operations simultaneously. The plots displaying the memory and throughput variation are presented in Figure 14.

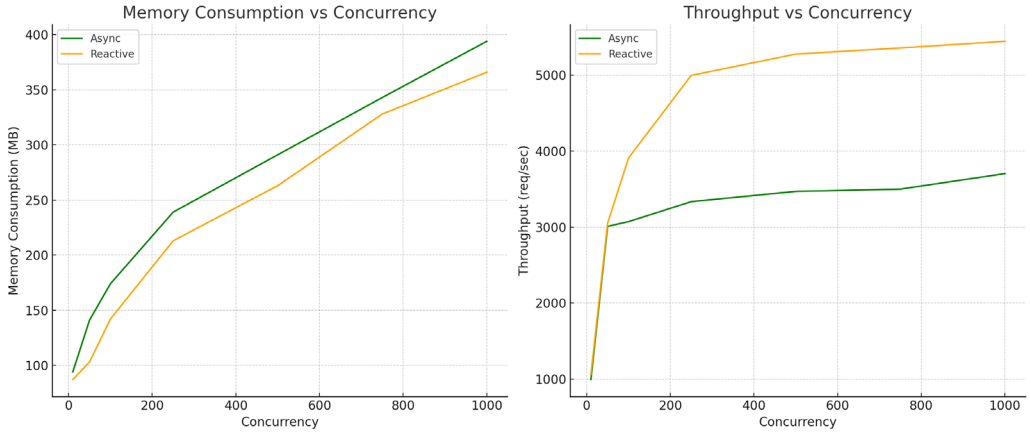


Figure 14. Performance results for users following.

The reactive model continues to demonstrate considerable advantages across key performance metrics. In this scenario, the reactive application consumed less memory, with up to 7% less at maximum concurrency (366 MB vs. 394 MB). CPU usage continues to remain similar between the two models, ranging from 8.6% up to 37.9% for the reactive application, compared with 10.2% up to 35.6% for the asynchronous approach. The reactive application maintains a lower average response time overall, with throughput remaining notably higher throughout the test, reaching 5446 requests/s at

 doi:10.20944/preprints202411.1467.v1

peak concurrency, compared to 3704 requests/s for the asynchronous variant, a 42% difference under the maximum load conditions.

The third scenario under consideration involves updating an existing tweet. This test was conducted using distinct instances for both users and tweets to evaluate performance during update operations. The memory and throughput variations are plotted in Figure 15.

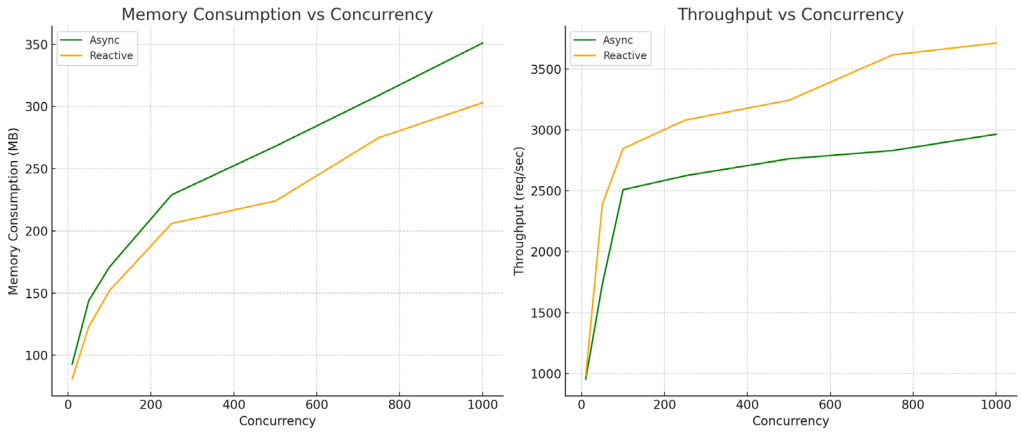
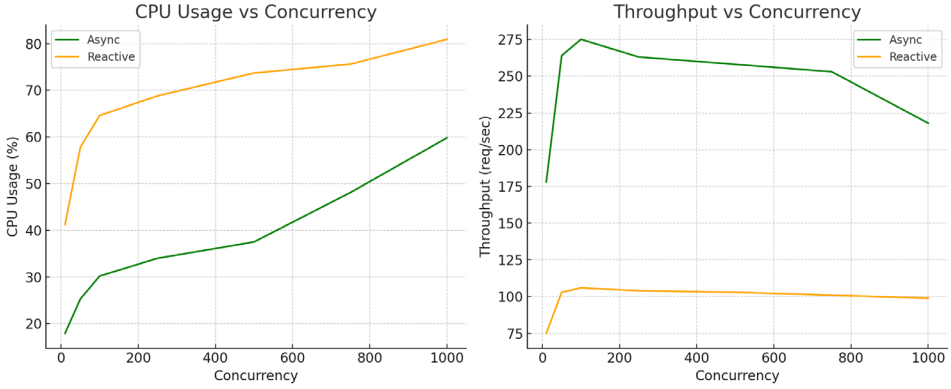


Figure 15. Performance results for updating existing tweets.

In this scenario, the reactive application consistently outperformed in terms of memory usage and throughput, consuming only 303 MB of memory at the maximum concurrency level investigated, whereas the asynchronous application consumed 351 MB, a difference of about 14%. However, CPU usage was slightly lower in this scenario, ranging from 7.9% to up to 26.9%, compared with 9.5% to 28.7% for the asynchronous application. The average response time is lower for the reactive model and the throughput also remained consistently superior, achieving 3714 requests/s at peak concurrency for the reactive application, which is approximately 25% higher compared to the 2965 requests/s of the asynchronous model. This advantage is further reflected in the 90th percentile response times, where the reactive application displayed a more consistent and lower response time of up to 294 ms, compared to 688 ms for the asynchronous application. Therefore, the reactive application handles requests more efficiently under high load while maintaining superior throughput.

The fourth scenario being evaluated was to retrieve a user's tweets. This case was intended to be particularly demanding and has been proven to be perhaps the most challenging of those analyzed, with multiple database entries in the order of millions being loaded this time, with every user in the application having approximately 1000 tweets associated with it, resulting in lower performance metrics and the impossibility of achieving performance levels comparable to the previous tests. The focus of this test was to assess the system's ability to handle larger volumes of data and more complex interactions, as each tweet required data computed in the interaction microservice. The variation in CPU usage and throughput is illustrated in Figure 16.



 doi:10.20944/preprints202411.1467.v1

Figure 16. Performance results for fetching user tweets**.**

This time, the reactive application performed worse than the asynchronous application. Under the maximum load, the reactive application used 691 MB of heap memory and 80.9% of the CPU, compared to 1979 MB and 59.8% for the asynchronous application. Although its memory consumption remained lower throughout the testing, with approximately 65% less than the asynchronous variant, the average response time is higher for the reactive application, rising to 8275 ms compared to 3755 ms for the asynchronous application, with a throughput also considerably lower, standing at only 99 requests/s compared to 218 requests/s, indicating the potential of certain limitations of the reactive paradigm in handling complex tasks and large volumes of data in certain contexts.

To isolate the root cause of the performance limitation, the sequential steps in the code flow were isolated or swapped. Subsequently, JMeter tests were rerun after each modification to determine which of these altered steps resulted in metrics that were comparable to those of the asynchronous application. Initially, the mapping functions responsible for converting entities to DTOs were removed. Next, the Redis cache layer and the calls to the interaction service were excluded to determine if these contributed to the observed latency. Finally, it was concluded that the performance limitations might be related to the database interactions and could be attributed to either the data access abstraction provided by Spring Data, the driver for PostgreSQL, or the limitations of the containerized database. However, it is not excluded that these limitations might occur due to the bare infrastructure, inefficient implementation of the proposed solution, or potential issues in the testing methodology setup.

The final results indicated that although the reactive principles are not cutting-edge and have already gone through many iterations and developments, the reactive paradigm cannot be considered a universal replacement. The last scenario outlined in the performance study demonstrates that there are situations where classical approaches remain preferable. This underlines the importance of carefully assessing the context and particularities of each system before deciding upon a migration. In some cases, traditional technologies are still prevalent, while the transition to new approaches, such as reactive programming, and the adaptation of stable and established principles require considerable time and effort.

While the reactive approach does not always guarantee efficiency improvements, pointing to the counterexamples that disproved the assumption of the absolute superiority of this model during the performance testing, the great results achieved across the other scenarios highlight the potential for higher performance and lower resource consumption due to the non-blocking architecture and the reduction in the number of required threads. At the same time, it is also clear that software development has not reached its limits and that there are still considerable opportunities for innovation and improvement in this area.

Limitations in Local Environment Testing

This testing was performed in a local environment, with containerized databases, thereby influencing the accuracy of higher concurrency simulations. Despite the utilization of a “server” system to host the applications and an additional “client” system to execute the JMeter tests, the local conditions do not fully reflect the behavior in a production environment with real users. In addition, testing in a local environment may not scale to what would be achieved inside cloud or distributed production environments, and the generated traffic is not as diverse as in real-use scenarios.

The hardware specifications for the devices used to perform the testing are as follows:

Server: Intel Core i7 10700 CPU, 32 GB DDR4 RAM, and 1 TB SSD.

Client: Apple M1 chip, 16 GB unified memory, and 256 GB SSD.

These specifications influence the results of performance testing, as the limited resources available for such consumer devices might have introduced variations that would not have been present in a robust production environment. Also, other applications or processes running within the systems used for testing could have interfered with the results, introducing additional variation in the resulting measurements.

 doi:10.20944/preprints202411.1467.v1

The testing was conducted over LAN, which ensures minimal network latency, as displayed in Figure 17.

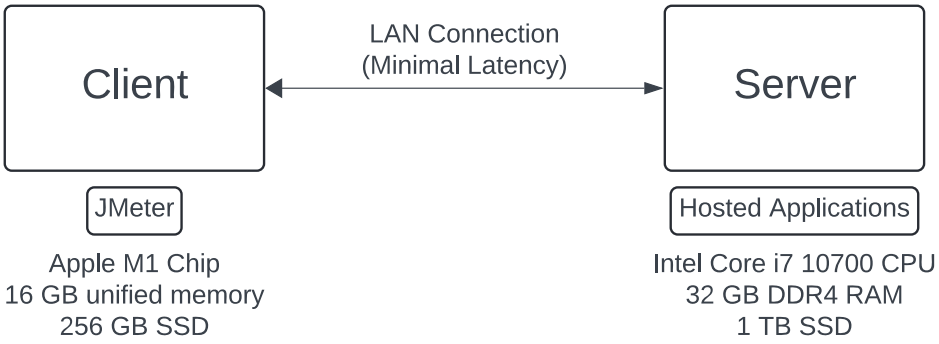


Figure 17. Testing Environment Diagram.

While this was beneficial for achieving consistent and timely results, it does not provide a complete overview of the performance under varied network conditions, which are encountered in distributed environments or on the Internet.

During the performance testing phase, the authentication using JWT tokens was disabled to avoid the constant overhead it introduced. This allowed for a focused evaluation of the core performance of the services. However, a complete evaluation based on real-world scenarios should also consider the overhead introduced by the authentication.

8. Conclusions and Further Development

8.1. Conclusions

Based on the analyses and test results, an iterative refinement process is essential to enhance performance and address identified limitations. This approach, akin to the iterative refinement described in software development methodologies [67], may involve further adjustments to the algorithms and the reactive architecture, ensuring that the system effectively meets scalability and stability requirements.

As part of the project, two complete backends were developed based on the asynchronous and reactive models. Both applications were extensively tested by defining and executing equivalent unit tests using JUnit 5 and identical automated integration tests using Cucumber, accounting for approximately 150 tests per service and achieving an estimated code coverage of 90%. The tests indicated functional equivalence between the two models, with all the considered scenarios demonstrating similar behavior and reinforcing the viability of transitioning towards the reactive paradigm without compromising the existing functionality.

From a performance point of view, the reactive variant achieved significantly better response times but also proved to be more efficient by using a reduced number of threads and resources in most of the evaluated scenarios due to the non-blocking, event-driven architecture. Nonetheless, in some complex scenarios, it still presented limitations and was outperformed by the asynchronous model, highlighting that its effectiveness might fluctuate depending on the nature of the tasks.

In terms of heap memory consumption, the reactive approach consistently outperformed its asynchronous counterpart, showcasing superior efficiency and performance throughout. Even at lower concurrency levels, during the profile fetching scenarios, the reactive application consumed only 60 MB compared to 64 MB for the asynchronous approach, resulting in a 6% reduction. As concurrency levels increased and different cases were evaluated, this pattern of robust memory management persisted. At moderate concurrency levels of 100 and 250 users, the differences in memory usage consistently exceeded 10%, while in more demanding scenarios, such as updating existing tweets, the reactive application consumed only 303 MB at peak concurrency, which is lower

 doi:10.20944/preprints202411.1467.v1

than the 351 MB required by the asynchronous setup, translating into 14% less memory required. In the scenario of the following users, the reactive model required just 366 MB at the maximum concurrency level analyzed, significantly less than the 394 MB consumed by the asynchronous variant, marking a 7% decrease in memory usage. Unlike the asynchronous model, which experienced more pronounced increases in thread count in response to rising user numbers, the reactive model's thread pool did not expand as aggressively, maintaining a more constant count even under increased loads and registering less pronounced memory spikes. This trend was observed and maintained across various testing conditions and diverse operational contexts, highlighting the reactive paradigm's ability to handle an increased load without increasing memory consumption significantly, a solid advantage, especially in environments where maintaining a low memory footprint is mandatory to preserve performance and scalability.

When assessing CPU efficiency, the reactive model demonstrated effective saturation, often leading to better request processing compared to that of the asynchronous model. For instance, in user profile fetching scenarios, the reactive system's CPU consumption increased from 5.2% at lower concurrency levels to 42.1% at the highest, aligning with its design to optimize CPU use without overextending resources. In contrast, the asynchronous system escalated to 40.7% CPU usage, representing a rather modest decrease in demand. However, it's worth mentioning that the asynchronous model was displaying higher memory utilization, while also handling a lower throughput. A tighter gap was observed at moderate concurrency levels, where the reactive system registered a CPU usage of 28.4%, slightly higher than the 26.7% of the asynchronous system, accounting for approximately 6% more. This indicates a potential shift in efficiency under certain conditions. This trend became more pronounced in more complex tasks like fetching user tweets, where the reactive approach peaked at 80.9% CPU consumption, above the 59.8% recorded by the asynchronous system, reflecting a 35% increase. This suggests that the reactive system may require more CPU resources to maintain its throughput, particularly in scenarios that demand intensive data processing. This might suggest a potential trade-off where higher CPU consumption compensates for greater performance stability, underscoring the importance of balanced CPU and memory management strategies. The ability of the reactive model to handle more complex scheduling and backpressure and to manage many small tasks efficiently might reflect the overhead associated with these principles like the workload. Despite the higher CPU demands in certain contexts, the reactive model was still able to deliver sustained performance across varying operational demands, showcasing sustained performance and adaptability in managing resources under varying operational demands.

In evaluating performance indicators such as average response times, throughput, and 90th percentile response times, the reactive model typically showcases greater efficiency across the examined scenarios. For example, when fetching the user profiles at the highest concurrency, an average response time of 93 ms was registered for the reactive approach, which was 25% faster than the 125 ms recorded by the asynchronous counterpart. Additionally, the reactive model managed to process 8867 requests per second, significantly outpacing the 6659 requests per second handled by the asynchronous model, marking a 33% increase in throughput. However, not all data consistently showed the reactive model as superior, even during scenarios where it exhibited enhanced performance, and not all tests favored the reactive model. When fetching user tweets with a higher number of users, the reactive system averaged a considerably slower response time of 8275 ms, compared to the 3755 ms displayed by the asynchronous model, resulting in a response time that was more than double on average. Despite this setback, there were also scenarios presenting seemingly marginal differences when comparing the average response times, such as updating existing tweets at moderate concurrency. In this scenario, both models performed comparably, with the reactive model scoring around 71 ms versus the asynchronous model's 73 ms, accounting for only about a 3% difference. However, the throughput for the reactive model remained superior, reflecting its overall efficiency. The examination of 90th percentile response times offers deeper insights into the consistency of response times under load. For the same scenario of updating user tweets, the 90th percentile provided a clearer differentiation of over 50%, with the reactive model recording 105 ms,

 doi:10.20944/preprints202411.1467.v1

significantly lower than the asynchronous model's 162 ms. This illustrates why, despite similar average response times, the throughput of the reactive approach remained superior, showcasing more stability and a quicker resolution of the bulk of responses. This stability suggests that any occasional spikes that brought the averages closer were rare, underscoring the reactive model's ability to deliver reliable performance, particularly useful in environments where maintaining high service levels is mandatory. The ability to manage requests swiftly and effectively, especially under high demand, confirms the reactive model is a suitable choice for scenarios requiring robust and efficient handling of large volumes of requests, despite occasional setbacks in specific demanding tasks.

Nonetheless, it must be emphasized that limitations and difficulties in migrating to the reactive paradigm remain significant. The migration process to a reactive model can be a viable option, whether applying a partial migration to only critical components or transitioning towards fully reactive systems is considered. Analyzing the specifics of each product and making an informed decision remain the determining factors. This project offers a starting point for such investigations and decisions, to promote a better understanding of both the advantages and challenges associated with reactive programming in the Java ecosystem.

8.2. Further Development

Developing a reactive-based graphical user interface. The development and automation of graphical user interface testing with frameworks such as Selenium [68] would strengthen the ideas of equivalence and interchangeability of the two backends (asynchronous and reactive), confirming that the migration does not compromise the user experience at the functional level and ensuring the consistency and reliability of the application. Also, the adoption of protocols such as Server-Sent Events (SSE) [69], WebSocket [70], or RSocket [71] for real-time updates could further refine the user experience by enhancing the application's responsiveness and maintaining seamless activity, as these protocols facilitate efficient, low-latency updates, which are key parts in ensuring that the migration to a reactive-based architecture does not compromise the functional level of the user interface.

Hosting the application on a dedicated platform. Cloud platforms would allow for the simulation of a higher number of concurrent requests, while also allowing for more accurate measurement of the resources and replicas necessary to handle such loads. Therefore, the testing results can be improved by a more accurate evaluation under intensive usage scenarios, more accurately outlining the differences between the asynchronous and reactive variants under more demanding conditions, and providing more detailed performance and resource utilization insights.

Developing and integrating a reactive ORM (Object-Relational Mapping) based on R2DBC. Given the availability of frameworks such as Hibernate Reactive, based on the Vert.x reactive specification [72], but not for R2DBC, the adjustments necessary for the application in the context of migrating to a reactive ORM can also be studied, taking into account the impact and consequences of missing certain functionalities, as observed in the development of the reactive prototype. Performance testing and comparisons using an ORM in both asynchronous and reactive versions could potentially reveal other advantages and limitations of each paradigm.

Author Contributions: Conceptualization, A.Z. and C.T.; formal analysis, A.Z. and C.T.; investigation, A.Z. and C.T.; methodology, A.Z. and C.T.; software, A.Z.; supervision, C.T.; validation, A.Z. and C.T.; writing—original draft, A.Z.; writing—review and editing, C.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Not Applicable.

Data Availability Statement: Data available in publicly accessible repository. The data presented in this study are openly available in GitHub at https://github.com/andreizb/tweebyte.

Conflicts of Interest: The authors declare no conflicts of interest.

References

 doi:10.20944/preprints202411.1467.v1

1. Erder, M.; Pureur, P., Woods, E. *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*; Addison-Wesley Professional, **2021**.
2. Ciceri, C.; Farley, D.; Ford, N.; Harmel-Law, A.; Keeling, M.; Lilienthal, C. *Software Architecture Metrics: Case Studies to Improve the Quality of Your Architecture*; O'Reilly Media, **2022**.
3. Arnold, K.; Gosling, J.; Holmes, D. *The Java Programming Language*, 4th ed.; Addison-Wesley Professional: Glenview, IL, USA, **2005**.
4. Sierra, K.; Bates, B.; Gee, T. *Head First Java: A Brain-Friendly Guide*, 3rd ed.; O'Reilly Media: Sebastopol, CA, USA, **2022**.
5. Davis, A. L., *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*, Berkeley, CA: Apress, **2018**.
6. Urma, R.G.; Fusco M., Mycroft A., *Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming*; 2nd Edition, Manning, **2018**.
7. Hitchens, R., *Java NIO: Regular Expressions and High-Performance I/O*; O'Reilly Media: Sebastopol, CA, USA, **2002**.
8. Nurkiewicz, T.; Christensen, B., *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*; O'Reilly Media: Sebastopol, CA, USA, **2016**.
9. Hedgpeth, R., *R2DBC Revealed: Reactive Relational Database Connectivity for Java and JVM Programmers*; Berkeley, CA: Apress, **2021**.
10. Goetz, B. *Java Concurrency In Practice*; Pearson India, **2016**.
11. Srivastava, R.P.; Nandi, G.C., *Controlling Multi Thread Execution Using Single Thread Event Loop*, 2017 International Conference on Innovations in Control, Communication and Information Systems, **2017**, 88-94.
12. Giebas, D.; Wojszczyk, R., *Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis*, *IEEE Access*, **2021**, *9*, 61298-61323.
13. Malhotra, R., *Rapid Java Persistence and Microservices: Persistence Made Easy Using Java EE8, JPA and Spring*, Berkeley, CA: Apress, **2019**.
14. Söderquist, I., *Event Driven Data Processing Architecture*, *2007 Design, Automation & Test In Europe Conference & Exhibition*, **2007**, *1-3*, 972-976.
15. Laliwala, Z.; Chaudhary, S., *Event-Driven Service-Oriented Architecture*, *2008 5th International Conference on Service Systems and Service Management*, **2008**, *1-2*, 410-415.
16. Bellemare, A., *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*, O'Reilly Media: Sebastopol, CA, USA, **2020**.
17. Woodside, M., *Performance Models of Event-Driven Architectures* Companion of the ACM/Spec International Conference on Performance Engineering, **2021,** 145-149.
18. Gamma E.; Helm R.; Johnson R.; Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, **1994**.
19. Grinovero, S. *Hibernate Reactive: Is it Worth It?*, **2021**, Available online: https://in.relation.to/2021/10/27/hibernate-reactive-performance/.
20. Ju, L.; Yadav, A.; Yadav, D.; Khan, A.; Sah, A.P. *Using Asynchronous Frameworks and Database Connection Pools to Enhance Web Application Performance in High-Concurrency Environments*. Proceedings of the 2024 International Conference on IoT in Social, Mobile, Analytics, and Cloud (I-SMAC 2024); IEEE, **2024**.
21. Dahlin, K. *An Evaluation of Spring WebFlux - With Focus on Built-in SQL Features*. Master's Thesis, Institution of Information Systems and Technology, Mid Sweden University, **2020**.
22. Joo, Y.H.; Haneklint, C. *Comparing Virtual Threads and Reactive WebFlux in Spring: A Comparative Performance Analysis of Concurrency Solutions in Spring*. Bachelor's Thesis, Degree Programme in Computer Engineering, KTH Royal Institute of Technology, Stockholm, Sweden, **2023**.
23. Mochniej, K.; Badurowicz, M. *Performance Comparison of Microservices Written Using Reactive and Imperative Approaches*. Journal of Computer Sciences Institute, **2023**, 28, 242-247.
24. Wang, Y. *Scalable and Reactive Data Management for Mobile Internet-of-Things Applications with Actor-Oriented Databases*. PhD Thesis, University of Copenhagen, Copenhagen, Denmark, **2021**.
25. Bansal, S.; Namjoshi, K.S.; Sa'ar, Y. *Synthesis of Asynchronous Reactive Programs from Temporal Specifications*. Proceedings of the International Conference on Computer Aided Verification, **2018**.
26. Bahr, P.; Houlborg, E.; Rørdam, G.T.S. *Asynchronous Reactive Programming with Modal Types in Haskell*. In Proceedings of Practical Aspects of Declarative Languages; Gebser, M., Sergey, I., Eds.; Springer Nature Switzerland, **2024**; pp. 18–36.
27. Spilcă, L., *Spring Start Here: Learn What You Need and Learn It Well;* Manning: New York, NY, USA, **2021**.
28. Walls, C., *Spring in Action*; Manning: New York, NY, USA, **2022**.
29. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann A., *Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality*, *2019 IEEE International Conference on Software Architecture Companion*, **2019**.

30. Fielding, R.T., *Architectural Styles and the Design of Network-based Software Architectures*, PhD Thesis, University of California, Irvine, CA, USA, **2000**.
31. Saternos, C., *Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful*; O'Reilly Media: Sebastopol, CA, USA, **2014**.
32. Afonso, J.; Caffy, C.; Patrascoiu, M.; Leduc, J.; Davis, M.; Murray, S.; Cortes, P. *An HTTP REST API for Tape-backed Storage*. EPJ Web Conf. **2024**, 295, 01008.
33. Nickoloff, J.; Kuenzli, S., *Docker in Action*, Second Edition; Manning: New York, NY, USA, **2019**.
34. Newman, S., *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*; O'Reilly Media: Sebastopol, CA, USA, **2019**.
35. Vernon, V.; Tomasz, J., *Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture*, Addison-Wesley Publishing, **2022**.
36. Bonér, J.; Farley, D.; Kuhn, R.; Thompson, M., *The Reactive Manifesto*, **2014**, Available online: https://www.reactivemanifesto.org.
37. Pal, N.; Yadav, D.K. *Modeling and verification of software evolution using bigraphical reactive system*. Clust. Comput. **2024**, 27, 12983–13003.
38. Padmanaban, K.; Kalpana, Y.B.; Geetha, M.; Balan, K.; Mani, V.; Sivaraju, S.S. *Simulation and modeling in cloud computing-based smart grid power big data analysis technology*. Int. J. Model. Simul. Sci. Comput. **2024**, 27, 2541005.
39. Ullenboom, C., *Spring Boot 3 and Spring Framework 6*, Rheinwerk Computing, **2023**.
40. Rao, R. R.; Swamy, S. R., *Review on Spring Boot and Spring Webflux for Reactive Web Development*, *International Research Journal of Engineering and Technology*, **2020**, *7*, *04*, 3834-3837.
41. Schoop, S.; Hebisch, E.; Franz, T. *Improving Comprehensibility of Event-Driven Microservice Architectures by Graph-Based Visualizations*. Softw. Archit. ECSA **2024**, 14889, 14.
42. Cabane, H.; Farias, K. *On the impact of event-driven architecture on performance: An exploratory study*. Future Gener. Comput. Syst. **2024**, 153, 52–69.
43. Ponge, J.; Navarro, A.; Escoffier, C.; Le Mouël, F., *Analysing the Performance and Costs of Reactive Programming Libraries in Java, Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, **2021**.
44. Christensen, B.; Husain, J., *Reactive Programming in the Netflix API with RxJava*, **2013**, Available online: https://netflixtechblog.com/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a.
45. Oracle *Java/JDBC Scalability and Asynchrony: Reactive Extension and Fibers*, **2019**, Available online: https://www.oracle.com/a/tech/docs/dev6323-reactivestreams-fiber.pdf.
46. *squbs: A New, Reactive Way for PayPal to Build Applications*, **2016**, Available online: https://medium.com/paypal-tech/squbs-a-new-reactive-way-for-paypal-to-build-applications-127126bf684b.
47. Harris, P.; Hale, B., *Designing, Implementing, and Using Reactive APIs*, **2018**, Available online: https://www.infoq.com/articles/Designing-Implementing-Using-Reactive-APIs/.
48. JSON Web Tokens, Available online: https://jwt.io.
49. Spilcă, L., *Spring Security in Action*; Second Edition; Manning: New York, NY, USA, **2024**.
50. Richardson, C. *Microservice Architecture Pattern*, **2024**, Available online: https://microservices.io/patterns/data/database-per-service.html.
51. Oracle Java Documentation, Available online: https://docs.oracle.com/en/java/.
52. Zuul Documentation, Available online: https://zuul-ci.org/.
53. Spring Cloud Gateway Documentation, Available online: https://spring.io/projects/spring-cloud-gateway.
54. Ferrari, L.; Pirozzi, E., *Learn PostgreSQL: Use, manage and build secure and scalable databases with PostgreSQL 16*, 2nd Edition; Packt Publishing; **2023**.
55. Tudose, C. *Java Persistence with Spring Data and Hibernate*; Manning: New York, NY, USA, **2023**.
56. Bonteanu, A.M.; Tudose, C. *Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA*, *Applied Sciences – Basel*, **2024**, *14*, *7*, 2743.
57. Redis Official Website, Available online: https://redis.io/.
58. Hansson, O., *Spring MVC or Spring WebFlux?*, **2022**, Available online: https://www.squeed.com/julkalender-2022/spring-mvc-or-spring-webflux/.
59. Eclipse Transformer Website, Available online: https://projects.eclipse.org/projects/technology.transformer.
60. Reflectoring Website, Available online: https://reflectoring.io/dependency-injection-and-inversion-of-control/.
61. Montesi, F.; Weber, J., *From the Decorator Pattern to Circuit Breakers in Microservices 33rd Annual ACM Symposium on Applied Computing*, **2018**, 1733-1735.
62. Resilience4j Website, Available online: https://resilience4j.readme.io/docs/getting-started.

 doi:10.20944/preprints202411.1467.v1

63. Tudose, C. *JUnit in Action*; Manning: New York, NY, USA, **2020**.
64. Cucumber Website, Available online: https://cucumber.io/.
65. Van Merode, H., *Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines*, Berkeley, CA: Apress, **2023**.
66. JMeter Website, Available online: https://jmeter.apache.org/.
67. Anghel, I.I.; Calin, R.S.; Nedelea, M.L.; Stanica, I.C.; Tudose, C.; Boiangiu, C.A. *Software development methodologies: A comparative analysis*. UPB Sci. Bull. **2022**, 83, 45–58.
68. Selenium Website, Available online: https://www.selenium.dev/.
69. Server-Sent Events, Available online: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events.
70. WebSockets API, Available online: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
71. RSocket Website, Available online: https://rsocket.io/.
72. Eclipse Vert.x Website, Available online: https://vertx.io/.