JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 28 (2023) 242–247
Received: 2 June 2023
Accepted: 3 July 2023

Performance comparison of microservices written using reactive and imperative approaches

Porównanie wydajności mikroserwisów napisanych w oparciu o podejście reaktywne i imperatywne

Kacper Mochniej*, Marcin Badurowicz

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The purpose of this paper was to compare the performance of microservices based on reactive and imperative approaches. To accomplish this task, two microservice applications written in Java using the Spring programming framework were developed. The Spring Web and Spring Webflux modules were used for the conventional and reactive versions, respectively. During the tests, functionalities related to operations of retrieving and inserting records into the database, data processing and file transfer were invoked. The Gatling tool was used to conduct the tests. The tests showed that reactive microservices can be more efficient in particular when there are delays in communication with services or the database. Otherwise, it depends on the complexity of the operations being performed. Microservices based on the reactive paradigm also use less RAM compared to conventional counterparts.

Keywords: microservices; reactive programming; imperative programming

Streszczenie

Celem pracy było porównanie wydajności mikroserwisów opartych o podejście reaktywne i imperatywne. Aby wykonać zadanie, stworzono dwie aplikacje mikroserwisowe napisane w języku Java z użyciem szkieletu programowania Spring. Wykorzystane zostały moduły Spring Web oraz Spring Webflux odpowiednio dla wersji konwencjonalnej i reaktywnej. W trakcie badań wywoływane były funkcjonalności związane z operacjami pobierania i wstawiania rekordów do bazy danych, przetwarzania danych, przesyłania plików. Do przeprowadzenia testów wykorzystano narzędzie Gatling. Badania wykazały, że mikroserwisy reaktywne mogą być wydajniejsze w szczególności w przypadku występowania opóźnień w komunikacji z serwisami lub bazą danych. W innym razie jest to zależne od złożoności wykonywanych operacji. Mikroserwisy oparte o paradygmat reaktywny, wykorzystują również mniej pamięci RAM w porównaniu z konwencjonalnymi odpowiednikami.

Słowa kluczowe: mikroserwisy; programowanie reaktywne; programowanie imperatywne

*Corresponding author

Email address: kacper.mochniej@pollub.edu.pl

©Published under Creative Common License (CC BY-SA v4.0)

1. Introduction

A microservice is a small application which can be deployed, scaled and tested independently and has single responsibility. It can, for example, read data from a queue, execute small pieces of business logic. Such applications are easy to maintain, so the microservice approach has become very popular in enterprise IT. It was introduced in 2014 by J. Lewis and M. Fowler. As a result, applications began to be divided into smaller, cooperating components [1].

Reactive programming is focused on reacting to changes such as data values or events. It allows to program asynchronous and event-driven use cases much easier, without the need for a deep understanding of low-level computer processes and the need to define the complex interactions of state, particularly across thread and network boundaries. Reactive programming is useful in following scenarios:

- processing user events or signal changes,
- handling latency-bound I/O events,
- handling events pushed to the application [2].

Java-based Spring framework is one of the most popular solutions for microservices development. That's because it contains a lot of functionality that helps developers create both small and large projects. Along with Spring 5, the Spring WebFlux [3] module was released for creating reactive applications. It uses Project Reactor [4] library which is an implementation of Reactive Streams - standard for asynchronous stream processing with non-blocking back pressure adopted in Java 9. Spring WebFlux also provides support for non-servlet containers such as Netty or Undertow.

The purpose of the work is to compare the performance of microservices based on reactive and imperative approaches, considering:

- communication with the database,
- operations on data,
- communication between services.

The following hypotheses have been defined:

1. Reactive microservices are more efficient for data-intensive tasks than conventional ones,

2. Reactive microservices are more efficient for latency-bound operations compared to non-reactive ones,
3. Reactive microservices use less RAM than conventional ones.

2. Review of the literature

The analysis of the literature showed that the topic covered is still fresh, as there are not many works dedicated to it. In addition, the conclusions of the various works are inconsistent, with some showing that reactive applications are more efficient while others don't. The situation is similar for hardware resources - the results of some works show less RAM or CPU usage for a reactive application, while others for a conventional application.

In the work [5] the author analyzed the features and disadvantages of reactive programming compared to conventional programming using a containerized microservices-based online ticket store programmed both in reactive and non-reactive versions. Tests have shown that there is not much difference from the conventional approach, however reactive programming improves the software development process and the stability of software.

Article [6] compares the reactive and conventional approaches in Java Web application development. For this purpose applications were created both in reactive and non-reactive ways using Spring Boot framework. Query processing times, the use of environment resources, and how many queries can be handled correctly was checked. In addition, the lines of code required to create each application were analyzed to compare the time consumption of their implementation. Results show that the reactive application processes queries faster, uses less CPU and is more stable in the case of handling many simultaneous requests, but it's more time-consuming to create than imperative variant.

Work [7] evaluates the possibility of using reactive programming and R2DBC in Java to communicate with a relational database, it has been done by creating two applications with Spring Boot framework: the reactive and non-reactive one, which include appropriate API to connect with the database (R2DBC and JDBC). The database used in work is MySQL. The study shows that R2DBC is good "out of the box" without need to set specific parameters. However it seems to have slower select queries and BLOB's are not handled optimally.

In the work [8] the author compared reactive and non-reactive applications written in Spring Boot and Quarkus. The project aimed to provide information to decide what framework is preferred to use in which cases. Results show that in Spring Boot reactive applications use more hardware resources than in non-reactive ones unlike Quarkus. Also, the overall use of hardware resources is higher in Spring Boot.

In the article [9] authors share the experiences in building and adapting reactive systems to microservices architecture. They rewrote an existing application using microservices architecture to reactive system and compared performance of both variants. Results show that the performance improvement in reactive system is not dramatic, but there is a large increase in throughput.

The article [10] is a review of the state of the art of reactive microservices. The objective is to explore documents concerning reactive microservices, migrate microservice project to reactive variant, share experiences and evaluate project with the studied metrics. Authors chose the Restaurant Management system to migrate to reactive microservices using the Lagom framework. The implemented solution accomplished goals relative to maintainability, scalability, testability and monitorability. However, it was not possible to obtain reliable results on the performance, also some security vulnerabilities were detected.

The purpose of work [11] is to check the effects of reactive programming. Author compared synchronous and reactive Playtech BGT Sports content server written using Spring Boot framework. The results showed that the CPU usage is similar for both solutions. However, when the content provider transmitted data at 5 ms intervals, the reactive system had lower latency and 100% throughput.

3. Research methodology

The subject of the study is to compare performance of two microservice applications - one written in reactive approach and another in conventional approach. The application consists of 3 microservices each one performing a specific function.

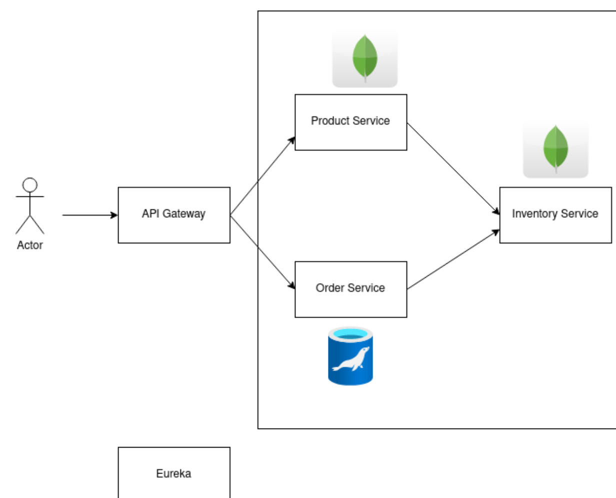

Figure 1: Application architecture.

ProductService is responsible for adding products to the catalog, as well as updating their inventory by connecting to InventoryService. It also allows for exporting, as well as multiple saving using a csv file. It uses a MongoDB database. The previously mentioned InventoryService has access to the inventory of products. It checks at the time of ordering whether the product is in stock, if so, it subtracts the quantity ordered from the current product stock, if not, it returns the product code. It also uses a MongoDB database. OrderService is used to place orders, connects to

InventoryService, uses MariaDB database. Average request processing times for a given scenario, number of instances and number of users were compared. The Gatling [12] tool was used to create and send requests to the applications. The operations that were used to perform the tests are:

- inserting records into the database,
- retrieving records from the database,
- object mapping,
- uploading files to server,
- downloading files from server,
- sending data between services.

This will be done by performing operations such as placing orders, adding, retrieving products or importing/exporting them via csv file. The tests were conducted for each functionality 3 times for both variants of the application with different numbers of microservice instances (1 and 3) and simultaneous requests to the application (100 and 3000). Running tests for different numbers of microservices instances was intended to verify whether the number of instances running and the chosen paradigm are related in terms of performance. The following table presents a detailed description of the scenarios.

Table 1: Test scenarios

Scenario	Description	Number of users
Placing incorrect orders	Sending 55 objects representing invalid orders to OrderService, then transfer to InventoryService , remap and validate, return the order codes to OrderService, and then return to the user.	100, 3000
Placing incorrect orders (single response delayed)	„Placing incorrect orders" scenario with changed logic of order validation - each order is sent individually, a delay of 100ms was added before returning the response from the service.	100
Updating stock	Updating the stock of a product using its id: check if a product with a given exists in the database, if so, change the quantity in stock	100, 3000
Product exports	Retrieving large number of records (271600) from database, map objects to rows, export to csv file	10
Product imports	Uploading the csv file with 18084 rows to the service, remap the row to an object and save it to the database	10
Adding a product	Inserting a single record into the database	100, 3000
Retrieving a product	Retrieving a record from the database	100, 3000
Delay 100 ms	Simulating a delay before returning the response from the ProductService	100, 3000
Barcode generation	Generating a barcode for the product	100, 3000

A platform with the following specifications was used:

- Processor: Intel Core i5 8250U 1.6-3.4GHz,
- RAM: 8GB DDR4,
- Drive: 256GB NVME SSD.

Both microservices and tests were running on the same machine. The environment configuration for running multiple instances was the same as for a single instance.

4. Research results

4.1. Application response times

The results of the tests are statistics of response times to requests including average times.

Table 2: Average response times for conventional application requests

Conventional variant (avg response time [ms])				
Scenario	1 instance		3 instances	
	100 users	3000 users	100 users	3000 users
Placing incorrect orders	537	3711	537	4228
Placing incorrect orders (concurrent internal requests with delay)	6667	-	6649	-
Updating stock	488	3263	522	3263
Adding product	350	2163	351	2319
Fetching product	338	2483	339	2395
Delay 100ms	327	2100	310	2006
Generate barcode for product	342	2907	369	3460
	10 users (working on large datasets)			
Import products	2337		2686	
Export products	15375		16573	

Based on Tables 2 and 3, it can be seen that in most cases the reactive application responds to requests in comparable or worse time than the conventional counterpart. In the test of placing incorrect orders, the reactive application performed much worse. In the case of 100 simultaneous users and 1 instance, the difference is about 46%, for 3 instances the difference is 38%. For 3000 users the conventional application is more than 3 times faster for both 1 and 3 instances. In the second scenario the same functionality is used but it's adjusted to take advantage of the strengths of the non-blocking http client. This time the reactive application was 2 times faster for 100 concurrent users for 1 and 3 instances. "Updating stock" scenario focuses more on exploring performance for inter-service communication alone, without costly stream operations. For 1 instance

results are similar for both 100 and 3,000 users the difference is just over 10% in favor of the conventional version, for 3 instances the difference is greater 13% for 100 users and 23% for 3000 users. In the product addition test, the reactive application achieved better results by being 26% faster for 100 users for 1 instance and 27% for 3 instances and for 3,000 users 10% and 15% respectively. In the product fetching test results were similar, the difference in favor of the reactive version is small and within the limit of measurement error. The latency simulation test showed for 100 users a 21%, and for 3000 users a 12% performance advantage for 1 instance. However for 3 instances results were close. For generating a barcode test for 100 users, the reactive variant was 21% faster for 1 instance, however, for the rest of the cases the results are similar. The last 2 tests were performed for only 10 users, as large datasets were used and it was not possible to perform these tests for more users for performance reasons. Results for products import were very similar, however in the export test the conventional variant was 12% faster for 1 instance and 16% for 3 instances.

Table 3: Average response times for reactive application requests

Reactive variant (avg response time [ms])				
Scenario	1 instance		3 instances	
	100 users	3000 users	100 users	3000 users
Placing incorrect orders	992	11070	867	11077
Placing incorrect orders (concurrent internal requests with delay)	3102	-	3363	-
Updating stock	549	3727	601	4227
Adding product	260	1936	255	1963
Fetching product	322	2321	317	2356
Delay 100ms	259	1845	314	1817
Generate barcode for product	271	2890	338	3266
	10 users (working on large datasets)			
Import products	2311		2582	
Export products	17505		19640	

4.2. RAM usage

The charts in this chapter present the RAM consumption of individual microservices before and after testing for the reactive and conventional versions.

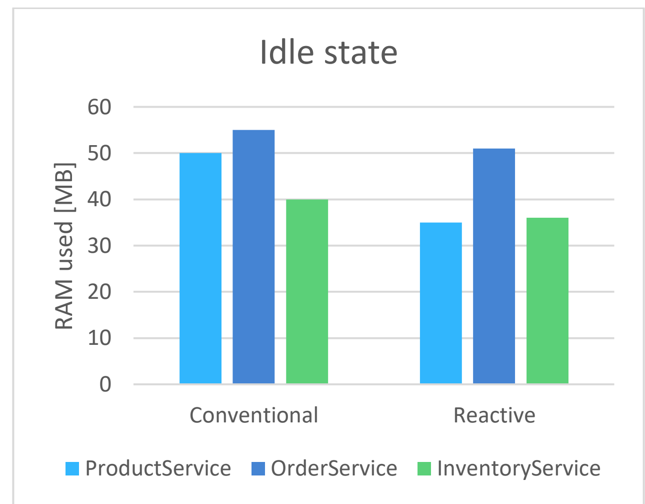

Figure 2: RAM used by microservices in the idle state.

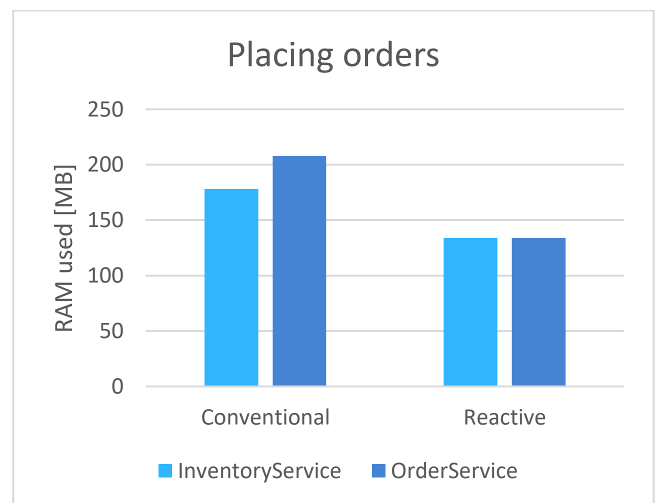

Figure 3: RAM used by microservices in placing incorrect orders scenario.

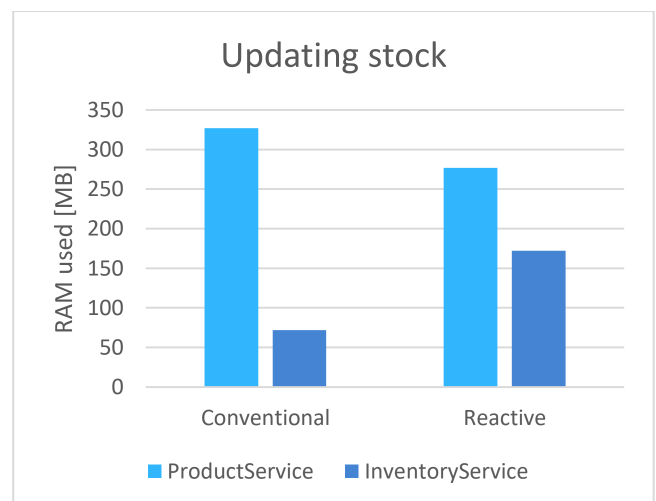

Figure 4: RAM used by microservices in updating stock scenario.

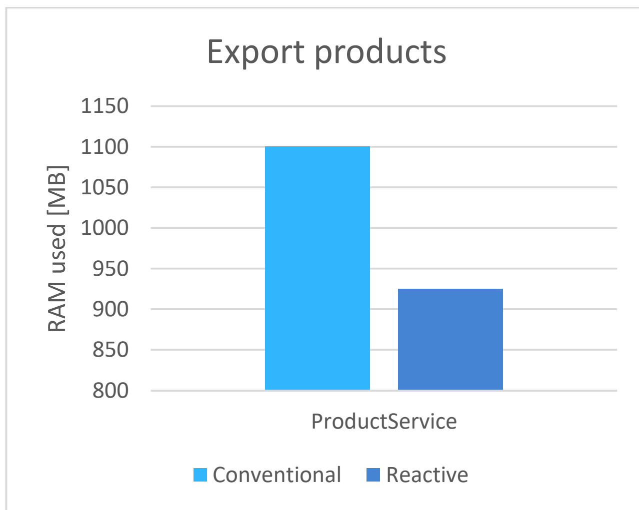

Figure 5: RAM used by microservices in export products scenario.

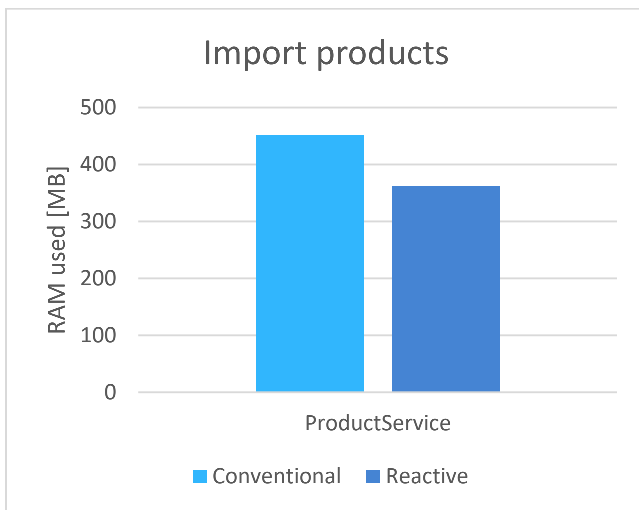

Figure 6: RAM used by microservices in import products scenario.

From the diagrams, you can see that the reactive application used less RAM for most cases. In the idle state reactive ProductService used 30% less RAM, OrderService 7% and InventoryService 10%. After placing incorrect orders reactive OrderService used 36% less RAM and InventoryService 23%. A bit different result is seen with the update stock test: reactive ProductService used 15% less RAM, while InventoryService used 58% more. In the adding product test the reactive variant obtained a better result by 13%. The biggest difference was in the fetching product test, reactive service used 61% less memory. In the generating barcode test, the reactive application used 57% less RAM. For the scenario of import products, the reactive variant was better by 20%. In last test – products export, there was an 16% advantage for the reactive variant.

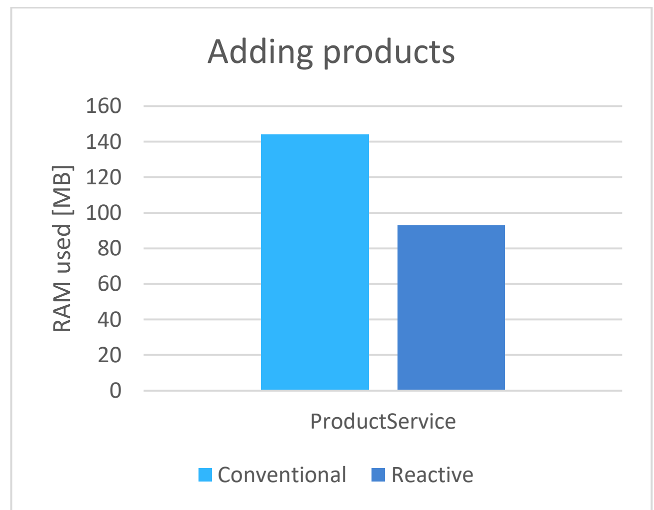

Figure 7: RAM used by microservices in adding products scenario.

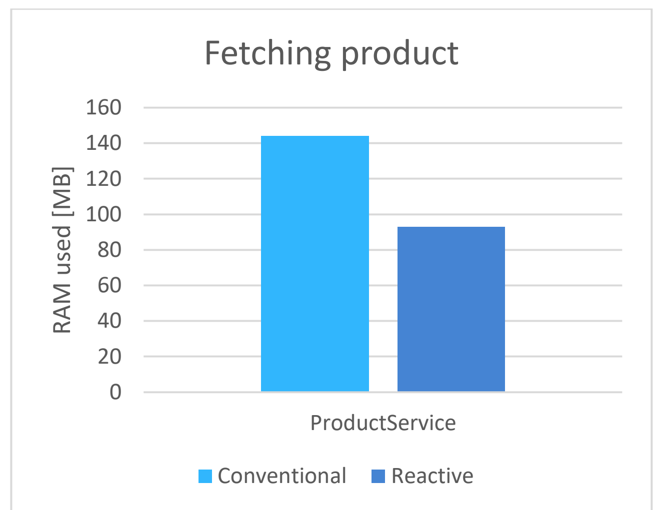

Figure 8: RAM used by microservices in fetching products scenario.

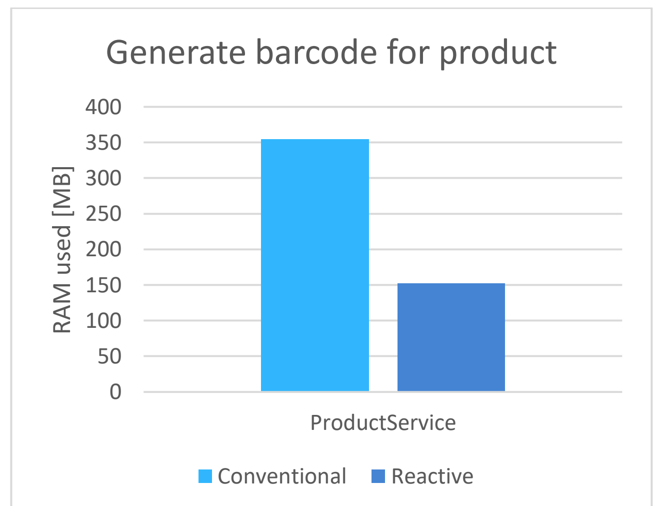

Figure 9: RAM used by microservices in generate barcode for product scenario.

5. Summary and conclusions

Despite the fact that reactive streams process data asynchronously, it wasn't possible to observe a performance advantage for operations on large data sets. Especially for CPU-intensive tasks, the reactive application performed significantly worse than the conventional one. This could mean that operations on reactive streams have higher complexity and take longer than the corresponding imperative code. This would be indicated by the results of the barcode generation test, where the results perform gently better. It is also a CPU-intensive task, but the operations are performed in a blocking style. This is also noticeable when comparing the tests of placing incorrect orders and updating stock, in both tests data is exchanged between services, but in the latter there are far fewer data operations and the results of the reactive version perform much better here. Two tests (adding a single product and I/O operations) showed the advantage of the reactive variant due to the non-blocking nature of I/O operations for this solution. The surprise, however, is that this gain is not apparent for retrieving a record from the database. As a result, it is difficult to say with certainty which approach provides better performance. Based on measurements for individual tests, one can conclude that for typical purposes (communication with other services, returning data) reactive services are less efficient. In reality, however, this depends on many different factors such as the specification of the server, application design, the database used, so for similar scenarios the results under different conditions can be quite different. This is evident by comparing the results obtained from various works. What's more, all the services and databases were running on the same machine, so there were no delays in the connections between them, which reactive application handles better because it doesn't block the thread, but sends another request. This was confirmed in the ordering test, where order data was sent one at a time, and in a test simulating the delay in processing a request. Therefore, it is important that before deciding on a reactive system, careful consideration should be given to whether the choice is appropriate under the circumstances. Performance also depends on the implementation of the reactive paradigm, in this case, the Spring WebFlux framework and the Java language were used, the results would be quite different when using other development tools. Based on the above results, it is difficult to clearly determine whether the number of instances and the chosen approach are related in terms of performance, sometimes the gain was greater for the reactive application, other times vice versa.

In the tests conducted, reactive services used less RAM in most cases. This is made possible by using an event loop model that takes care of calling the corresponding request and response handling functions. It runs in the background and does not block the main thread; instead, it moves on to the next request, and when the first request is ready, it resumes processing. This also greatly reduces the number of threads created by a reactive application.

References

[1] J. Thönes, Microservices, IEEE Software 32(1) (2015) 113-116.

[2] T. Nurkiewicz, B. Christensen, Reactive Programming with RxJava: Creating asynchronous, event based applications, 1st Edition, O'Reilly Media , 2016.

[3] Spring WebFlux Documentation, https://docs.spring.io/spring-framework/reference/web/webflux.html#webflux, [27.05.2023].

[4] Project Reactor webpage, https://projectreactor.io/, [27.05.2023].

[5] P. Dakowitz, Comparing reactive and conventional programming of Java based microservices in containerized environments, Master thesis, Haw Hamburg, 2018.

[6] S. Iwanowski, G. Kozieł, Comparative analysis of reactive and imperative approach in Java Web application development, Journal of Computer Sciences Institute 24 (2022) 242-249.

[7] K. Dahlin, An evaluation of Spring Webflux with focus on built in SQL features, Master thesis, Mid Sweden University, 2020.

[8] A. Nordlund, N.Nordstrom, Reactive vs Non-reactive Java Framework, Bachelor thesis, Mid Sweden University, 2022.

[9] A. Sim, O. Barus, F. Jaya, Lessons Learned In Applying Reactive System In Microservices, Journal of Physics: Conf. Series 1175 (2019) 1-6.

[10] G. Hochbergs, Reactive Programming and its effect on performance and the development process, Master thesis, Lund University, 2017.

[11] J. Ferreira, Reactive Microservices An Experiment, Master thesis, Polytechnic of Porto, 2022.

[12] Gatling webpage, https://gatling.io/ , [27.05.2023]