JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 24 (2022) 242–249
Received: 13 June 2022
Accepted: 6 July 2022

Comparative analysis of reactive and imperative approach in Java web application development

Analiza porównawcza podejścia reaktywnego i imperatywnego w tworzeniu aplikacji internetowych w języku Java

Sebastian Iwanowski*, Grzegorz Kozieł

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

Abstract
The subject of this study was to compare web applications created using the imperative and reactive approaches in Java. For this purpose, two applications with the same functionalities were developed using both approaches. The study looked at the performance, stability and time-consumption of implementation of each application. Based on the obtained results, it was found that the reactive application processes queries faster, uses less CPU, and is more stable in the case of handling many simultaneous requests, where the processing time is greater than 10 seconds. No significant differences were observed in using the computer's RAM by the applications. In addition, the study showed that reactive application takes more time to create.

Keywords: imperative approach; reactive approach; web applications; Java

Streszczenie
Celem niniejszej pracy było porównanie aplikacji internetowych tworzonych przy pomocy podejścia imperatywnego oraz reaktywnego w języku Java. Do tego celu stworzono dwie aplikacje z takimi samymi funkcjonalnościami używając obu podejść. Badanie dotyczyło wydajności, stabilności oraz czasochłonności implementacji każdej z aplikacji. Na podstawie uzyskanych wyników, stwierdzono, że aplikacja reaktywna szybciej przetwarza zapytania, w mniejszym stopniu obciąża procesor oraz jest stabilniejsza w przypadku obsługi wielu jednoczesnych żądań, gdzie czas przetworzenia jest większy niż 10 sekund. W przypadku wykorzystania ilości pamięci RAM przez aplikacje nie zaobserwowano znaczących różnic. Ponadto badanie pokazało, że stworzenie aplikacji reaktywnej jest bardziej czasochłonne.

Słowa kluczowe: podejście imperatywne; podejście reaktywne; aplikacje internetowe; Java

*Corresponding author
Email address: **sebastian.iwanowski@pollub.edu.pl** (S.Iwanowski)

1. Wstęp

Aplikacje internetowe są obecnie jednymi z podstawowych form dostarczania usług. Aplikacje takie coraz częściej tworzone są z myślą o dużej liczbie odbiorców, a co za tym idzie z ukierunkowaniem na większy zysk. Programiści muszą systematycznie mierzyć się z zadaniem doboru odpowiednich technologii i narzędzi, aby jak najlepiej sprostać oczekiwaniom narzuconym im przez klientów. Niejednokrotnie dobór nieadekwatnych rozwiązań, może wiązać się z dużymi kosztami, będącymi skutkiem konieczności przebudowy części lub, co gorsza, całej aplikacji.

Jedną z popularniejszych obecnie technologii wybieranych do tworzenia oprogramowania internetowego jest język Java, oraz tworzony i rozwijany od wielu lat, wzorzec programistyczny Spring Framework. Szkielet ten, pozwala w stosunkowo prosty i szybki sposób, tworzyć różnego rodzaju aplikacje internetowe. Z racji tego, że wzorzec ten jest bardzo rozbudowany, programista korzystający z niego musi wybrać odpowiednie biblioteki, które decydować będą o tym jak działać będzie w przyszłości dany produkt. Głównymi bibliotekami w przypadku aplikacji internetowych są Spring MVC oraz Spring WebFlux, będącymi odpowiednikami imperatywnego i reaktywnego stosu technologicznego. Oba wybory mają zalety i wady. Aplikacje tworzone w oparciu o drugą z wymienionych bibliotek, jak opisują sami autorzy, powinny być głównie przeznaczone dla oprogramowania, które posiadać będzie znaczące opóźnienia, np. powolne operacje wejścia/wyjścia lub dla programów, które muszą zachować wysoką stabilność. Dla klasycznych aplikacji, nie posiadających wymienionych wcześniej cech, zalecają pozostanie przy Spring MVC, który jest prostszy i szybszy w zrozumieniu niż jej reaktywna alternatywa [1].

Twórcy obu bibliotek nie udostępniają wielu porównań wydajnościowych obu produktów. Jeżeli różnica w szybkości czy też stabilności okazałaby się niewielka, kosztem wydłużenia czasu wytwarzania oprogramowania, spowodowanym wyborem trudniejszej technologii, to należałoby przeprowadzić analizę czy byłoby to opłacalne zarówno dla programistów, którzy musieliby najprawdopodobniej przeznaczyć dodatkowy czas na pracę z oprogramowaniem oraz dla klientów którzy otrzymali by nieznacznie lepszą aplikację w zamian za dłuższy czas oczekiwania.

Celem artykułu jest potwierdzenie tezy: aplikacje reaktywne są wydajniejsze i stabilniejsze niż aplikacje imperatywne w przypadku jednoczesnej obsługi wielu zapytań. Dodatkowo zdefiniowano cztery hipotezy:

1. Zapytania w aplikacjach reaktywnych są przetwarzane szybciej niż w aplikacjach imperatywnych w przypadku obsługi wielu użytkowników jednocześnie,
2. Aplikacje reaktywne wykorzystują mniej zasobów procesora oraz pamięci RAM komputera,
3. Aplikacje reaktywne są bardziej stabilne i posiadają mniejszy procent nieobsłużonych zapytań w przypadku większej liczby zapytań,
4. Stworzenie aplikacji reaktywnej jest bardziej czasochłonne niż stworzenie aplikacji imperatywnej.

2. Analiza literatury

Analiza literatury wykazała niewielką popularność badań dotyczących podejścia reaktywnego na przykładzie języka Java. Dodatkowo, przeprowadzony przegląd nie wykazał, żadnej publikacji, która porównywała by podejście imperatywne oraz reaktywne w języku Java, zarówno pod względem wydajności jak i czasochłonności implementacji.

W artykule [2] autorzy porównują paradygmat reaktywny z tradycyjnym paradygmatem obiektowym, w odniesieniu do zrozumienia programu. Swoje badania przeprowadzili na 127 osobach podzielonych na 2 grupy, które miały do wykonania 10 zadań i ankietę dotyczącą aplikacji napisanych zarówno przy użyciu podejścia reaktywnego jak i obiektowego. Eksperyment brał pod uwagę takie czynniki jak, poprawność zrozumienia, wymagany czas oraz poziom umiejętności jaki był niezbędny do prawidłowej odpowiedzi. Twórcy wykazali, że aplikacje oparte o paradygmat reaktywny są prostsze w zrozumieniu, a dodatkowo poziom umiejętności programistycznych jaki wymagany jest do prawidłowej analizy był mniejszy niż w przypadku aplikacji opartych o paradygmat obiektowy. Autorzy dodają również, że przyczynami które wpływają na taki rezultat mogą być zmniejszona powtarzalność kodu oraz jego lepsza czytelność.

Artykuł [3] przedstawia porównanie aplikacji REST (Representational State Transfer) stworzonych w technologii Spring Boot oraz MS.NET. Badane programy posiadały identyczne funkcjonalności, odpowiadające czterem podstawowym operacjom: tworzenia, odczytywania, aktualizowania oraz usuwania (CRUD). Badania zostały przeprowadzone z wykorzystaniem narzędzia Apache JMeter i uwzględniały średnie czasy odpowiedzi, liczbę błędów w aplikacjach oraz wykorzystanie zasobów CPU oraz pamięci komputera. Na podstawie uzyskanych wyników, autorzy stwierdzili, że aplikacje oparte o technologię MS.NET zapewniają szybszy czas odpowiedzi, niż aplikacje stworzone w Spring Boot w każdym z testowanych scenariuszy. Twórcy zaobserwowali również, że program ten wykorzystuję mniej zasobów CPU oraz pamięci komputera. Ponadto, na podstawie analizy liczby błędów, twórcy uznali, że obie aplikacje są równie stabilne.

Autorzy artykułu [4] przedstawiają rezultaty badań dotyczących analizy porównawczej dwóch wielowątkowych paradygmatów: programowania reaktywnego oraz stylu kontynuacji-przejścia. Kryteriami jakimi posłużyli się twórcy w celu przeprowadzenia badania były utrzymywalność, wydajność oraz testowalność. Badanie zostało przeprowadzone na zaimplementowanych w oparciu o wymienione wyżej paradygmaty aplikacjach pozwalających na kompresję plików wideo. Twórcy wykorzystali do tego biblioteki RxJava oraz Kotlin Coroutines. Na podstawie uzyskanych wyników, twórcy stwierdzili, że obie technologie osiągają podobne wyniki, jeśli chodzi o wydajność oraz testowalność, lecz istnieją znaczące różnice pod względem utrzymywalności. Z przeprowadzonego przez nich badania, korzystającego z metryk Halsteada, wywnioskowali oni również, że aplikacja oparta na metodzie Coroutines jest łatwiejsza w zaimplementowaniu niż aplikacja oparta na RxJava. Zaznaczają, że jest to najprawdopobodniej spowodowane faktem, że będący przedmiotem badań styl kontynuacji-przejścia przypomina tradycyjny, imperatywny styl programowania.

3. Omówienie pojęć

3.1. Aplikacje imperatywne

Aplikacja imperatywna jest to program, którego kod wykonuje się w sposób sekwencyjny. Oznacza to, że każda linia kodu może się wykonać, jedynie po tym jak poprzednia instrukcja została zakończona. Idea w programach imperatywnych, opiera się więc na wstrzymywaniu wykonywania się kolejnych instrukcji.

W niniejszej pracy, program imperatywny zbudowany został w oparciu o wzorzec programistyczny Spring Boot i Spring MVC. Korzysta on z domyślnego dla tych bibliotek serwera – Tomcat [5]. Serwer ten jest przykładem serwletu, który został napisany z myślą o obsłudze kodu imperatywnego. Jednym z podstawowych schematów jego działania jest tworzenie wątku dla każdego odebranego żądania. Domyślnie jego maksymalna pula wątków wynosi 200, co oznacza, że maksymalnie jest w stanie obsłużyć 200 jednoczesnych zapytań, po przekroczeniu tego limitu, serwer zaczyna odrzucać lub kolejkować inne żądania. Serwer imperatywny jest serwerem blokującym co oznacza, że w momencie przypisania wątkowi sekwencji instrukcji do wykonania, jest on blokowany aż do czasu zakończenia ostatniej z nich. Dodatkowo w przypadku, gdy dane żądanie będzie powiązane z operacją wejścia/wyjścia np. zapis do pliku, i będzie zajmować stosunkowo dużą ilość czasu. Wątek po zleceniu wykonania tej czynności jest przez pewien czas niedostępny, pomimo, ze nie wykonuje praktycznie żadnej operacji [6].

3.2. Aplikacje reaktywne

Aplikacja reaktywna to aplikacja, która używa asynchronicznego podejścia do wykonywania szeregu instrukcji. Głównym założeniem programów reaktywnych jest to, aby sekwencja wykonywania się kolejnych bloków kodu była od siebie jak najbardziej niezależna oraz aby w żadnym momencie nie nastąpiło zablokowanie wątku odpowiedzialnego za jej wykonanie. W odróżnieniu od aplikacji imperatywnych, aplikacje reaktywne nie mają jednoznacznie określonej kolejności wykonywania się instrukcji.

Serwerem użytym w aplikacji reaktywnej, będącej celem badań pracy jest Netty, który został skonfigurowany i dostarczony przez wzorce projektowe Spring Boot oraz Spring WebFlux. Netty zbudowany został na potrzeby aplikacji reaktywnych, potrzebujących zarówno asynchroniczności jak i nieblokowanego przepływu instrukcji [7]. W odróżnieniu od serwera imperatywnego, jego schemat działania nie jest oparty na przypisywaniu żądań do konkretnych wątków, a jak najbardziej optymalnym zarządzaniu stosunkowo małą liczbą wątków. Serwer ten tworzy pętle zdarzeń, która odpowiedzialna jest za odbieranie i przesyłanie żądań. Pętla ta sprawdza, który wątek jest dostępny, a następnie oddelegowuje do niego odebrane zapytanie, przechodząc tym samym do dalszego obsługiwania serwera [8]. W przypadku opisywanego serwera, liczba pętli zależna jest od liczby dostępnych rdzeni procesora, zazwyczaj jest to jedna lub dwie pętle na rdzeń [9].

4. Metodyka badań

Przedmiotem badań jest porównanie wydajności, stabilności oraz czasochłonności implementacji aplikacji imperatywnej oraz reaktywnej w takich samych środowiskach oraz przy użyciu tych samych próbek badawczych. Do tego celu stworzone zostały dwie aplikacje napisane w wymienionych wyżej technikach programowania, posiadające te same funkcjonalności. Zbadane zostały czasy przetwarzania zapytań, wykorzystanie zasobów środowiska oraz sprawdzone zostało ile zapytań uda się poprawnie obsłużyć. Ponadto zliczono linie kodu, które były niezbędne do stworzenia każdej z aplikacji, w celu wyznaczenia czasochłonności implementacji takich projektów.

Porównanie zostało przeprowadzone poprzez zmierzenie czasu jaki będzie potrzebny każdej z aplikacji na przetworzenie określonej liczby żądań HTTP. Funkcjonalnościami które były wykorzystywane w celu przeprowadzenia pomiarów są:

- pobranie listy wszystkich pracowników z bazy danych,
- pobranie pojedynczego pracownika z bazy,
- aktualizacja pojedynczego pracownika w bazie danych,
- tworzenie nowego pracownika w bazie,
- pobranie elementu z opóźnieniem serwera równym 10 sekund.

Dla każdej z funkcjonalności badanie zostało powtórzone 50-krotnie w celu uśrednienia otrzymanych wyników. Próby przeprowadzone zostały w takich samych warunkach, dla 1000 elementów w bazie danych oraz odpowiednio 30, 300, 2000 jednoczesnych zapytań do aplikacji, w celu symulacji małej, średniej i dużej liczby zapytań. Dodatkowo przeprowadzono analizę liczby linii kodu obu aplikacji, co było miernikiem czasochłonności tworzenia kodu przy wykorzystaniu danej metody. Przeprowadzone badania zostały powtórzone dla aplikacji o dostępnych do wykorzystania zasobach:

- 2 rdzenie procesora, 4GB RAM,
- 4 rdzenie procesora, 8GB RAM.

Badania zostały przeprowadzone z wykorzystaniem dwóch maszyn o podanych w tabeli 1 parametrach.

Tabela 1: Parametry środowisk testowych

	Maszyna 1	**Maszyna 2**
Rodzaj urządzenia	Laptop	Komputer stacjonarny
Procesor	Intel Core i7 7700HQ, 2.8GHz	Intel Core i7 12700K,3.6GHz
Pamięć RAM	8GB DDR4	32GB DDR4
Dysk	SSD 256GB, interfejs IDE	SSD 2TB, interfejs NVMe
System operacyjny	Windows 10 Home 64-bit	Windows 10 Pro 64-bit

Obie maszyny posiadały zainstalowane JDK w wersji 11 oraz podłączone były do tej samej sieci o szybkości łącza 100MB/s. Maszyna 1 odpowiadała za uruchomienie skryptów badających szybkości odpowiedzi aplikacji oraz zapisanie tych danych w postaci plików HTML. Z kolei maszyna 2 pełniła rolę serwera dla obu aplikacji. Utrzymywała zarówno instancję aplikacji oraz przypisaną do niej bazę danych, jak i narzędzia Prometheus i Grafana, służące do pomiarów wykorzystania zasobów sprzętowych.

5. Wyniki badań

5.1. Czasy odpowiedzi aplikacji

Przeprowadzone badania pozwoliły wyznaczyć średnie czasy odpowiedzi oraz stosunek liczby poprawnie do niepoprawnie obsłużonych zapytań, który oznaczony jest w tabelach jako „OK/KO”. W przypadku każdej z funkcjonalności, średnie czasy odpowiedzi zostały wyliczone uwzględniając jedynie poprawnie obsłużone zapytania.

5.1.1. Ograniczenie 2CPU, 4GB pamięci RAM

W tabelach 2 oraz 3 zawarte zostały wyniki dotyczące aplikacji ograniczonych do 2 rdzeni procesora oraz 4GB pamięci RAM w środowisku Docker Desktop.

Tabela 2: Uśrednione wyniki pomiarów czasu odpowiedzi aplikacji imperatywnej ograniczonej do 2CPU, 4GB RAM

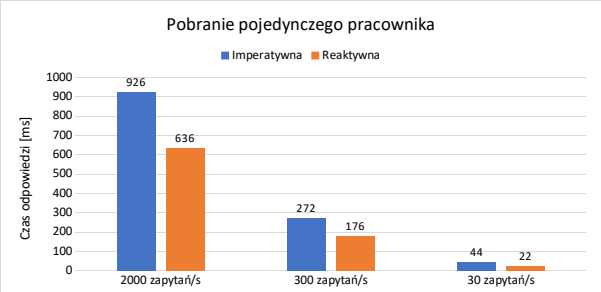
	Aplikacja imperatywna					
	Średni czas odpowiedzi [ms]			**OK/KO [%]**		
Liczba jednoczesnych zapytań	**2000**	**300**	**30**	**2000**	**300**	**30**
Pobranie pojedynczego pracownika	926	272	44	100	100	100
Pobranie wszystkich pracowników	7200	1110	148	96	100	100
Aktualizacja pojedynczego pracownika	1758	384	73	100	100	100
Utworzenie nowego pracownika	966	226	32	100	100	100
Pobranie elementu z opóźnieniem serwera równym 10 sekund	30494	13443	10198	50	100	100

Na podstawie Rysunków od 1 do 5, odczytać można, że w większości scenariuszy, aplikacja reaktywna osiągnęła lepsze wyniki. W przypadku wyników operacji pobierania pojedynczego pracownika przedstawionych na Rysunku 1, widać znaczącą różnicę pomiędzy programami. Aplikacja reaktywna jest w stanie zwrócić odpowiedź o 290ms szybciej niż aplikacja imperatywna w przypadku 2000 zapytań, co stanowi 31% wzrost szybkości.

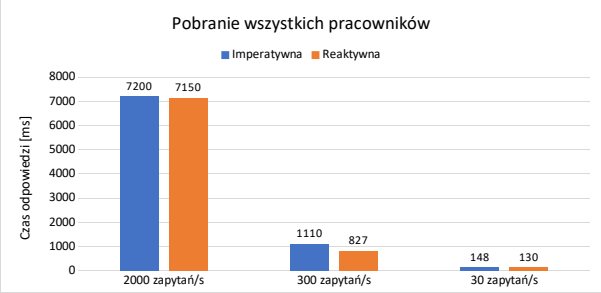
Tabela 3: Uśrednione wyniki pomiarów czasu odpowiedzi aplikacji reaktywnej ograniczonej do 2CPU, 4GB RAM

	Aplikacja reaktywna					
	Średni czas odpowiedzi [ms]			OK/KO [%]		
Liczba jednoczesnych zapytań	2000	300	30	2000	300	30
Pobranie pojedynczego pracownika	636	176	22	100	100	100
Pobranie wszystkich pracowników	7150	827	130	98	100	100
Aktualizacja pojedynczego pracownika	788	171	26	100	100	100
Utworzenie nowego pracownika	719	211	44	100	100	100
Pobranie elementu z opóźnieniem serwera równym 10 sekund	11296	10137	10136	100	100	100

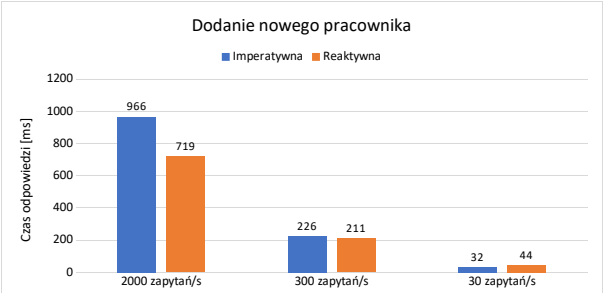
Wykres z Rysunku 2, dotyczący pobierania wszystkich użytkowników, również pokazuje, że stos reaktywny jest szybszy niż imperatywny, lecz w tym przypadku różnica ta jest mniej widoczna. Dla 2000 zapytań, wynosi ona 50 ms, co daje wynik lepszy o niecały procent. W przypadku 300 i 30 zapytań, aplikacja reaktywna jest szybsza o odpowiednio 25% i 12% w stosunku do aplikacji imperatywnej. Wykres z Rysunku 5 przedstawia porównanie czasów w przypadku pobrania elementu, przy którym serwer czeka 10 sekund na jego odesłanie, symulując czasochłonne operacje typu operacje wejścia/wyjścia. W tym przypadku widoczna jest największa różnica, jeśli chodzi o wysokie obciążenie serwera – 2000 zapytań. Aplikacja reaktywna jest w stanie obsłużyć zapytania niemal 3 razy szybciej niż jej imperatywny odpowiednik. Dodatkowo, tak jak widoczne to jest w Tabelach 2 oraz 3, serwer imperatywny odrzucił 50% zapytań, gdzie serwer reaktywny był w stanie obsłużyć każde z nich. W przypadku 300 i 30 zapytań obie aplikacje wypadły podobnie, lecz wyniki nadal świadczą na korzyść stosu reaktywnego.



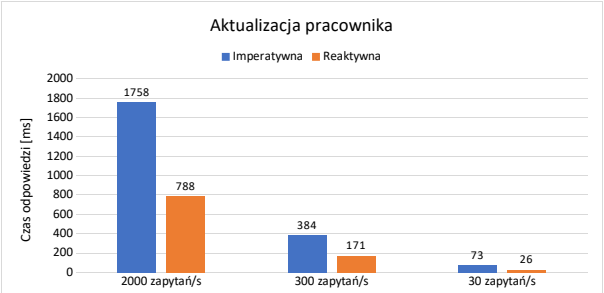
Rysunek 1: Średnie czasy pobrania danych pojedynczego pracownika przy ograniczeniu 2CPU, 4GB RAM.



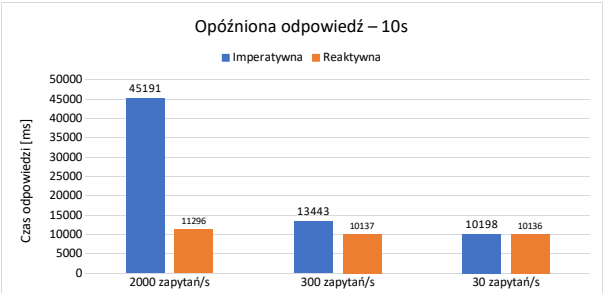
Rysunek 2: Średnie czasy pobrania danych wszystkich pracowników przy ograniczeniu 2CPU, 4GB RAM.



Rysunek 3: Średnie czasy dodania nowego pracownika przy ograniczeniu 2CPU, 4GB RAM.



Rysunek 4: Średnie czasy aktualizacja danych pracownika przy ograniczeniu 2CPU, 4GB RAM.



Rysunek 5: Średnie czasy odpowiedzi przy pobieraniu elementu opóźnionego o 10 sekund przy ograniczeniu 2CPU, 4GB RAM.

5.1.2. Ograniczenie 4CPU, 8GB pamięci RAM

Tabele 3 oraz 4 przedstawiają rezultaty badań dotyczące aplikacji ograniczonych do 4 rdzeni procesora oraz 8GB RAM, tak samo jak w poprzednim wariancie, w środowisku Docker Desktop.

Tak jak widoczne to jest na rysunkach od 6 do 10, w przypadku środowiska z dostępnymi zasobami: 4 rdzenie procesora oraz 8 GB RAM, czasy odpowiedzi również okazały się krótsze dla aplikacji reaktywnych w przypadku większości funkcjonalności.

Na podstawie rysunku 6 można odczytać, że wzrost szybkości odpowiedzi serwera reaktywnego w przypadku obciążenia 2 tysięcy zapytań dla funkcjonalności pobrania danych pojedynczego pracownika wynosi 33%, czyli 2% szybciej niż w przypadku wyniku dla poprzedniego ograniczenia. Na rysunku 7 widoczne jest, że operacja pobierania danych wszystkich pracowników z bazy danych zajmuje więcej czasu w przypadku zastosowania stosu reaktywnego. W przypadku 2000 oraz 300 zapytań, aplikacja imperatywna okazała się szybsza o odpowiednio 7% i 2%. W przypadku 30 zapytań, wynik świadczy jednak o przewadze programu reaktywnego.

Tabela 3: Uśrednione wyniki pomiarów czasu odpowiedzi aplikacji imperatywnej ograniczonej do 4CPU, 8GB RAM

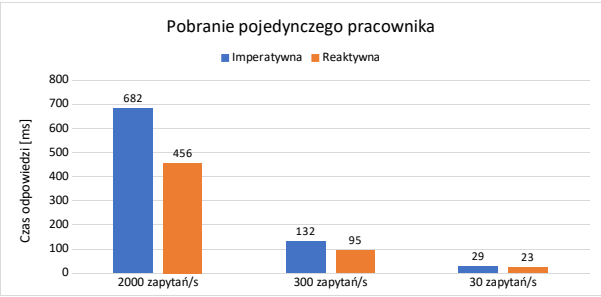
	Aplikacja imperatywna					
	Średni czas odpowiedzi [ms]			OK/KO [%]		
Liczba jednoczesnych zapytań	**2000**	**300**	**30**	**2000**	**300**	**30**
Pobranie pojedynczego pracownika	682	132	29	100	100	100
Pobranie wszystkich pracowników	5720	831	163	96	100	100
Aktualizacja pojedynczego pracownika	874	200	34	100	100	100
Utworzenie nowego pracownika	583	122	29	100	100	100
Pobranie elementu z opóźnieniem serwera równym 10 sekund	30485	13487	10063	51	100	100

Tabela 4: Uśrednione wyniki pomiarów czasu odpowiedzi aplikacji reaktywnej ograniczonej do 4CPU, 8GB RAM

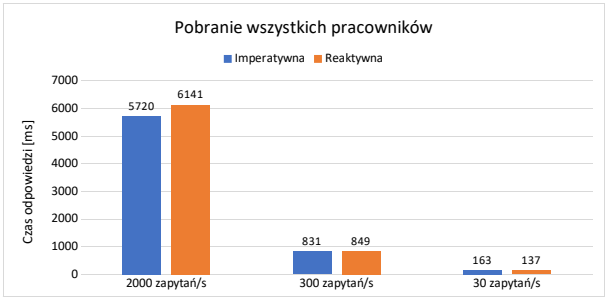
	Aplikacja reaktywna					
	Średni czas odpowiedzi [ms]			OK/KO [%]		
Liczba jednoczesnych zapytań	**2000**	**300**	**30**	**2000**	**300**	**30**
Pobranie pojedynczego pracownika	456	95	23	100	100	100
Pobranie wszystkich pracowników	6141	849	137	97	100	100
Aktualizacja pojedynczego pracownika	531	128	23	100	100	100
Utworzenie nowego pracownika	107	94	25	100	100	100
Pobranie elementu z opóźnieniem serwera równym 10 sekund	11394	10139	10131	100	100	100

Dla funkcjonalności przedstawionych na Rysunkach 8 oraz 9, różnica jest najbardziej widoczna w scenariuszach z 2000 zapytaniami na sekundę, gdzie w przypadku dodania nowego pracownika, wynik ten jest ponad pięciokrotnie lepszy dla aplikacji reaktywnej. Dla 300 i 30 zapytań, rezultaty są lepsze średnio o około 30% dla aktualizacji i o 19% w wypadku dodawania nowego pracownika do bazy, również na rzecz stosu reaktywnego.

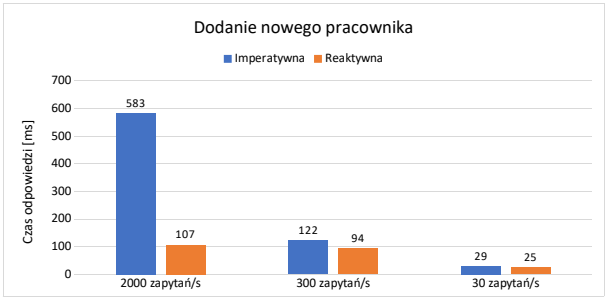
Analizując wykres przedstawiony na Rysunku 10, można dostrzec identyczną sytuację jak w poprzednim wariancie, aplikacja reaktywna uzyskała niemal 3 razy lepszy wynik w przypadku wariantu z 2000 zapytań.



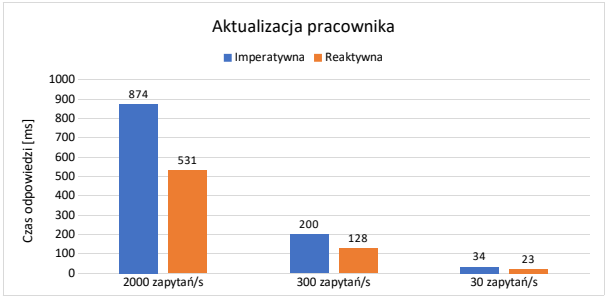
Rysunek 6: Średnie czasy pobrania danych pojedynczego pracownika przy ograniczeniu 4CPU, 8GB RAM.



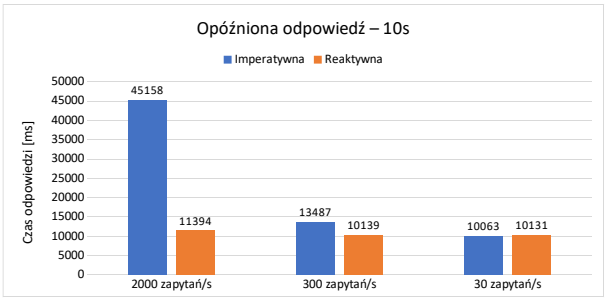
Rysunek 7: Średnie czasy pobrania danych wszystkich pracowników przy ograniczeniu 4CPU, 8GB RAM.



Rysunek 8: Średnie czasy dodania nowego pracownika przy ograniczeniu 4CPU, 8GB RAM.



Rysunek 9: Średnie czasy aktualizacja danych pracownika przy ograniczeniu 4CPU, 8GB RAM.



Rysunek 10: Średnie czasy odpowiedzi przy pobieraniu elementu opóźnionego o 10 sekund przy ograniczeniu 4CPU, 8GB RAM.

5.2. Wykorzystanie zasobów sprzętowych

5.2.1. Ograniczenie 2CPU, 4GB pamięci RAM

Tabela 5 oraz Tabela 6 przedstawia średnie zużycie zasobów sprzętowych przez testowane aplikacje ograniczone do 2 rdzeni procesora oraz 4GB RAM.

Na podstawie analizy obu tabel powiedzieć można, że aplikacja reaktywna wykorzystuje moc procesora w znacznie mniejszym stopniu niż aplikacja imperatywna. Różnice sięgają od 7%, w przypadku dodawania nowego pracownika przy 30 zapytaniach na sekundę, do

nawet 31%, w momencie pobierania danych wszystkich pracowników z bazy.

Na wykresie przedstawionym na Rysunku 11 widoczne jest, że średnie wykorzystanie procesora ze wszystkich scenariuszy wynosi 52.8% dla aplikacji imperatywnej, a 33.7% dla aplikacji reaktywnej, co stanowi ponad 36% różnicę w stosunku do programu imperatywnego. Jedynie w przypadku ostatniej funkcjonalności w tabelach widoczne jest, że aplikacja reaktywna wykorzystuję nieznacznie więcej dostępnych zasobów procesora niż jej imperatywny odpowiednik. W tym przypadku spowodowano jest to głównie tym, że aplikacja imperatywna odrzuca 50% zapytań, podczas gdy reaktywna obsługuje każde z nich. Widać, że dla 300 i 30 zapytań różnica wynosi jedynie 1%. Jedynie przypadek 2000 zapytań wykazuje 4% mniejsze wykorzystanie CPU przez program imperatywny.

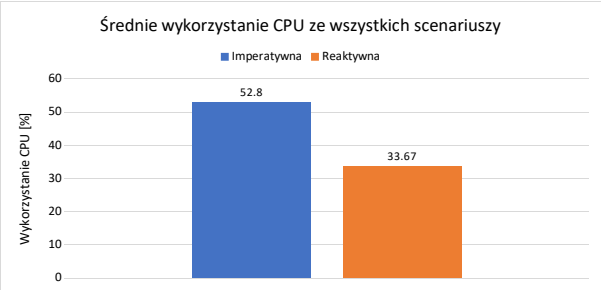
Analizując zużycie pamięci RAM obu aplikacji trudno jednoznacznie określić, która z nich zużywa jej mniej. Oba programy uzyskały podobne wyniki jeśli chodzi o średnią ze wszystkich scenariuszy, rysunek 12 pokazuje, że wynik aplikacji imperatywnej to 11%, a reaktywnej 10%. W przypadku scenariusza dotyczącego pobrania danych pojedynczego pracownika, zaobserwować można, że aplikacja reaktywna zużywa mniej pamięci dla 2000 i 300 zapytań, a aplikacja imperatywna dla 30 zapytań/s. Różnice w pozostałych scenariuszach są niewielkie, często zaledwie 1 procentowe.

Tabela 5: Uśrednione wyniki pomiarów wykorzystania zasobów komputera przez aplikację imperatywną ograniczoną do 2CPU, 4GB RAM

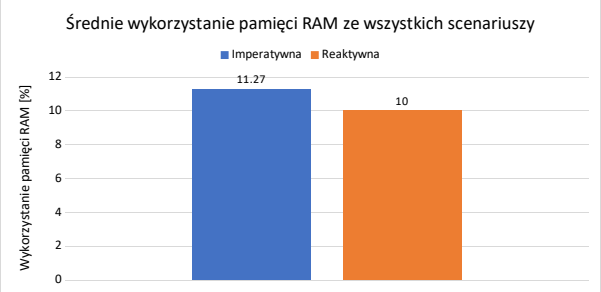
	Aplikacja imperatywna					
	Wykorzystanie CPU [%]			Wykorzystanie pamięci RAM [%]		
Liczba jednoczesnych zapytań	**2000**	**300**	**30**	**2000**	**300**	**30**
Pobranie pojedynczego pracownika	82	86	29	22	11	5
Pobranie wszystkich pracowników	62	63	31	32	13	7
Aktualizacja pojedynczego pracownika	94	92	61	16	7	6
Utworzenie nowego pracownika	83	68	35	11	12	5
Pobranie elementu z opóźnieniem serwera równym 10 sekund	3	2	1	9	7	6

Tabela 6: Uśrednione wyniki pomiarów wykorzystania zasobów komputera przez aplikację reaktywną ograniczoną do 2CPU, 4GB RAM

	Aplikacja reaktywna					
	Wykorzystanie CPU [%]			Wykorzystanie pamięci RAM [%]		
Liczba jednoczesnych zapytań	**2000**	**300**	**30**	**2000**	**300**	**30**
Pobranie pojedynczego pracownika	55	55	12	12	6	9
Pobranie wszystkich pracowników	31	55	18	31	13	5
Aktualizacja pojedynczego pracownika	57	68	10	11	7	6
Utworzenie nowego pracownika	57	47	28	13	8	6
Pobranie elementu z opóźnieniem serwera równym 10 sekund	7	3	2	9	7	7



Rysunek 11: Średnie wykorzystanie CPU ze wszystkich scenariuszy przy ograniczeniu 2CPU, 4GB RAM.



Rysunek 12: Średnie wykorzystanie pamięci RAM ze wszystkich scenariuszy przy ograniczeniu 2CPU, 4GB RAM.

5.2.2. Ograniczenie 4CPU, 8GB pamięci RAM

W Tabelach 7 i 8 przedstawione zostały wyniki badań dotyczących aplikacji ograniczonych do 4 rdzeni procesora oraz 8GB pamięci RAM.

Tabele te przedstawiają analogiczną sytuację jak w przypadku poprzedniego ograniczenia. Aplikacja reaktywna w mniejszym stopniu obciąża CPU we wszystkich, poza ostatnim scenariuszem dotyczącym pobierania elementu z opóźnieniem. Najmniejsza różnica w wykorzystaniu procesora wynosi jednak tutaj 2% dla przypadku dodawania nowego użytkownika podczas obciążenia 30 zapytań/s, a największa równa jest 36% w przypadku aktualizacji danych użytkownika dla 2000 zapytań. W przypadku funkcjonalności pobierania elementu z opóźnieniem, sytuacja jest jednakowa jak poprzednio, aplikacja reaktywna wykorzystuje nieznacznie więcej zasobów w zamian oferując lepszą stabilność aplikacji.

Z Rysunku 13 odczytać można, że średnie wykorzystanie procesora wynosi 33.6% dla aplikacji imperatywnej, a 24.3% dla aplikacji reaktywnej, co stanowi 27% spadek zużycia w stosunku do programu imperatywnego.

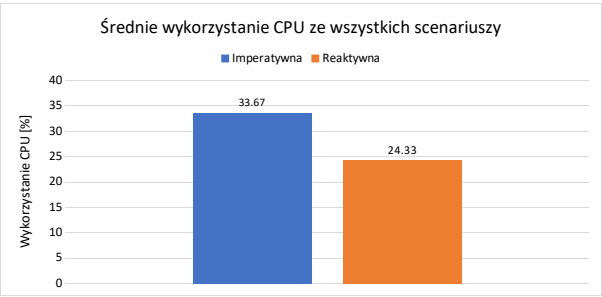
Tak jak w poprzednim przypadku zużycie pamięci RAM jest tutaj bardzo podobne dla obu aplikacji, nie widać jednak różnicy w przypadku funkcjonalności pobierania danych pojedynczego pracownika. Średnie wartości ze wszystkich scenariuszy przedstawione zostały na Rysunku 14 i wynoszą odpowiednio 6.6% oraz 6.4% dla aplikacji imperatywnej i reaktywnej.

Tabela 7: Uśrednione wyniki pomiarów wykorzystania zasobów komputera przez aplikację imperatywną ograniczoną do 4CPU, 8GB RAM

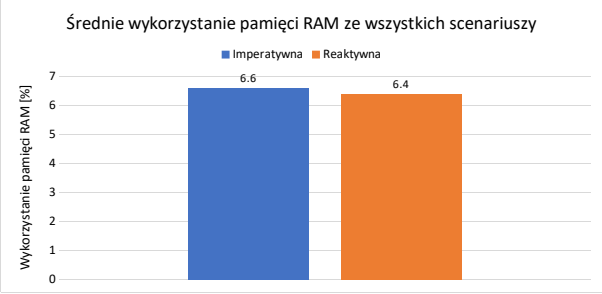
	Aplikacja imperatywna					
	Wykorzystanie CPU [%]			Wykorzystanie pamięci RAM [%]		
Liczba jednoczesnych zapytań	2000	300	30	2000	300	30
Pobranie pojedynczego pracownika	54	43	18	9	5	6
Pobranie wszystkich pracowników	41	40	37	15	6	6
Aktualizacja pojedynczego pracownika	87	69	17	8	5	5
Utworzenie nowego pracownika	53	28	12	6	8	4
Pobranie elementu z opóźnieniem serwera równym 10 sekund	2	2	2	6	5	5

Tabela 8: Uśrednione wyniki pomiarów wykorzystania zasobów komputera przez aplikację reaktywną ograniczoną do 4CPU, 8GB RAM

	Aplikacja reaktywna					
	Wykorzystanie CPU [%]			Wykorzystanie pamięci RAM [%]		
Liczba jednoczesnych zapytań	2000	300	30	2000	300	30
Pobranie pojedynczego pracownika	47	27	13	10	6	4
Pobranie wszystkich pracowników	24	26	27	15	5	5
Aktualizacja pojedynczego pracownika	51	57	5	7	7	5
Utworzenie nowego pracownika	38	31	10	5	6	4
Pobranie elementu z opóźnieniem serwera równym 10 sekund	5	2	2	6	6	5



Rysunek 13: Średnie wykorzystanie CPU ze wszystkich scenariuszy przy ograniczeniu 4CPU, 8GB RAM.



Rysunek 14: Średnie wykorzystanie pamięci RAM ze wszystkich scenariuszy przy ograniczeniu 4CPU, 8GB RAM.

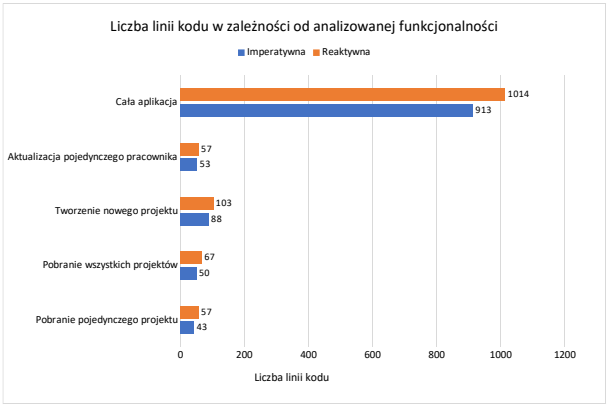
5.3. Czasochłonność implementacji

Czasochłonność implementacji została wyznaczona poprzez porównanie liczby linii kodu potrzebnej do stworzenia wybranych funkcjonalności oraz całej aplikacji dla każdego z programów. Zliczanie linii zostało przeprowadzone wyłączając puste linie oraz uwzględniając domyślne formatowanie dostarczone przez środowisko programistyczne IntelliJ IDEA. Tabela 9 pokazuje rezultaty tego badania.

Tabela 9: Wyniki pomiaru liczby linii potrzebnych do stworzenia wybranych funkcjonalności oraz całej aplikacji

	Aplikacja imperatywna	Aplikacja reaktywna
Pobranie pojedynczego projektu	43	57
Pobranie wszystkich projektów	50	67
Tworzenie nowego projektu	88	103
Aktualizacja pojedynczego projektu	53	57
Cała aplikacja	913	1014

Rysunek 15 przedstawia wizualne porównanie liczby linii kodu w zależności od analizowanej funkcjonalności. Ostatni diagram dotyczy całej aplikacji, z którego wynika, że różnica w ogólnej ilości kodu potrzebnej do stworzenia aplikacji reaktywnej wynosi o 10% więcej niż aplikacji imperatywnej. Na podstawie wykresu, można również stwierdzić, że aplikacja reaktywna wymaga większej ilości kodu do stworzenia takiej samej funkcjonalności co aplikacja imperatywna. Zależnie od funkcjonalności różnica ta wynosi od 4 do 17 linii kodu. Na podstawie otrzymanych w tym badaniu wyników, można stwierdzić, że stworzenie aplikacji reaktywnej wymaga więcej czasu niż stworzenie aplikacji imperatywnej. Wynika to głównie z braku odpowiedników dla uproszczeń i bibliotek używanych w stosie imperatywnym.



Rysunek 15: Liczba linii kodu w zależności od analizowanej funkcjonalności.

6. Podsumowanie i wnioski

W niniejszym artykule przedstawiono i omówiono wyniki badań dotyczących aplikacji internetowych tworzonych przy pomocy podejścia reaktywnego i imperatywnego w języku Java.

Na podstawie otrzymanych rezultatów można zauważyć kilka różnic w zachowaniu obu aplikacji. Pierwszą jest to, że aplikacje reaktywne okazały się szybsze nie tylko w przypadku obsługi wielu użytkowników jednocześnie, tak jak zakładano w pierwszej hipotezie, lecz w niemal każdym badanym w tej pracy scenariuszu. Znacząca różnica widoczna jest na przykładzie badania dotyczącego funkcjonalności pobierania elementu z 10 sekundowym opóźnieniem, gdzie stos reaktywny osiągnął prawie 3 razy szybszy czas odpowiedzi, względem aplikacji napisanej za pomocą stosu imperatywnego.

Seria badań, dotycząca porównania wydajności aplikacji, pod względem zużywanych zasobów komputera, wykazała, że obie aplikacje, niezależnie od dostępnych zasobów, potrzebują tyle samo pamięci RAM komputera do prawidłowego działania. W przypadku wykorzystania mocy obliczeniowej procesora, aplikacja imperatywna używała średnio 43.2%, gdzie aplikacja reaktywna wykorzystywała średnio jedynie 28.9%. Wynika z tego, że program reaktywny jest lepiej zoptymalizowany pod kątem redukcji zapotrzebowania na moc obliczeniową procesora niż program imperatywny, co z kolei częściowo potwierdza hipotezę numer 2.

W przypadku hipotezy 3 stwierdzono, że obie aplikacje są równie stabilne przy większości badanych operacji, wyłączając wspominaną wcześniej funkcjonalność pobierania elementu z opóźnieniem, w której stos reaktywny obsłużył o 50% więcej zapytań w przypadku wielu jednoczesnych żądań niż aplikacja imperatywna. Wynika to głównie z omawianego modelu działania serwera reaktywnego, który na żadnym etapie działania programu nie blokuje wątku wykonywalnego, a zamiast tego polega na pętli zdarzeń do informowania o odbywających się w systemie operacjach oraz o ich zakończeniu.

Ostatnie z badań dotyczące czasochłonności implementacji, potwierdziły hipotezę mówiącą o tym, że aplikacje reaktywne są bardziej czasochłonne do stworzenia niż aplikacje imperatywne. Otrzymane rezultaty pokazały, że każda z funkcjonalności w programie reaktywnym wymaga większej ilości kodu niż w programie imperatywnym, a dodatkowo kod całej aplikacji reaktywnej jest o średnio 10% dłuższy niż w przypadku aplikacji imperatywnej.

Podsumowując, postawiona w pracy teza „aplikacje reaktywne są wydajniejsze i stabilniejsze niż aplikacje imperatywne w przypadku jednoczesnej obsługi wielu zapytań" została częściowo potwierdzona, ponieważ aplikacje reaktywne są wydajniejsze, wyłączając zużycie pamięci RAM oraz są stabilniejsze, lecz tylko w jednym ze scenariuszy, na podstawie reszty z nich nie udało się stwierdzić tej różnicy. Dodatkowo badanie pokazało, że liczba zapytań nie ma tutaj znaczenia, ponieważ wyniki w każdym przypadku świadczą na korzyść aplikacji reaktywnej.

Literatura

[1] Dokumentacja Spring Framework odnośnie reaktywnych bibliotek, https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html, [28.05.2022].

[2] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, M. Mezini, On the positive effect of reactive programming on software comprehension: An empirical study. IEEE Transactions on Software Engineering, 43(12) (2017) 1125-1143.

[3] H. K. Dhalla, Benchmarking the performance of RESTful applications implemented in spring boot Java and MS. Net core, Journal of Computing Sciences in Colleges, 36(3) (2020) 178-178.

[4] S. Komolov, N. Askarbekuly, M. Mazzara, An empirical study of multi-threading paradigms Reactive programming vs continuation-passing style. 2020 the 3rd International Conference on Computing and Big Data, Taichung, 2020.

[5] G. Amuthan, Spring MVC. Przewodnik dla początkujących, Helion, 2015.

[6] J. Brittain, I. F. Darwin, Tomcat: The Definitive Guide, 2nd Edition, O'Reilly Media, 2003.

[7] O. Dokura, I. Lozynskyi, Hands-On Reactive Programming in Spring 5, Packt Publishing, 2018.

[8] Dokumentacja Spring Framework odnośnie reaktywnych bibliotek, https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html, [28.05.2022].

[9] Opis Spring WebFlux, https://piotrminkowski.com/2020/03/30/a-deep-dive-into-spring-webflux-threading-model/, [28.05.2022].