JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 32 (2024) 210–216
Received: 7 June 2024
Accepted: 19 June 2024

Examination of the performance and scalability of a web application in a reactive and imperative approach using the Spring Framework

Badanie wydajności i skalowalności aplikacji webowej w podejściu reaktywnym i imperatywnym z wykorzystaniem Spring Framework

Karol Lis*, Jakub Smołka

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

Abstract

The purpose of this paper was to test the performance and scalability of a web application written in reactive and imperative approaches using the Spring Framework, in order to understand the differences between these approaches and choose the technology that best meets the requirements and provides optimal performance. Two test applications were compared in terms of query processing times and CPU/RAM usage. The effect of Api Gateway microservices on application performance was analyzed. The tests showed that the reactive application processed I/O operations faster, used less RAM but more CPU. The imperative approach was faster for transactional operations performed sequentially. The reactive variant reacted with less latency to the presence of Api Gateway than the imperative approach.

Keywords: spring boot; reactive programming; imperative programming; application performance

Streszczenie

Celem artykułu było zbadanie wydajności i skalowalności aplikacji webowej napisanej w podejściu reaktywnym i imperatywnym z użyciem Spring Framework, aby zrozumieć różnice między tymi podejściami i wybrać technologię, która najlepiej odpowiada wymaganiom i zapewnia optymalną wydajność. Porównano dwie aplikacje testowe pod względem czasów przetwarzania zapytań oraz wykorzystania CPU/RAM. Analizowano wpływ mikroserwisu Api Gateway na wydajność aplikacji. Badania wykazały, że aplikacja reaktywna szybciej przetwarza operacje I/O, zużywa mniej RAM, ale więcej CPU. Podejście imperatywne było szybsze dla operacji transakcyjnych wykonywanych sekwencyjnie. Wariant reaktywny reagował mniejszymi opóźnieniami na obecność Api Gateway niż imperatywne podejście.

Słowa kluczowe: spring boot; programowanie reaktywne; programowanie imperatywne; wydajność aplikacji

*Corresponding author

Email address: **karol.lis@pollub.edu.pl** (K. Lis)

1. Wstęp

W dzisiejszym świecie, gdzie aplikacje internetowe muszą radzić sobie z coraz większym obciążeniem oraz rosnącą liczbą użytkowników, wydajność skalowalność stają się kluczowymi elementami, które decydują o sukcesie aplikacji. Zrozumienie różnic między podejściem reaktywnym a imperatywnym jest niezbędne w celu wyboru odpowiedniej strategii projektowej, która najlepiej odpowiada ich potrzebom i zapewnia optymalną wydajność systemu. Programowanie reaktywne to paradygmat, który koncentruje się na tworzeniu asynchronicznych, nieblokujących aplikacji, które lepiej radzą sobie z dużymi ilościami równoczesnych żądań. W języku Java, szkielet programistyczny Spring oferuje dedykowane narzędzia do tworzenia reaktywnych aplikacji, takie jak Spring WebFlux, które zapewniają obsługę programowania reaktywnego na wysokim poziomie [1]. Programowanie imperatywne polega na pisaniu aplikacji sekwencyjnych. Takie aplikacje są często prostsze i wymagają krótszego czasu implementacji, ale mogą występować w nich problemy z wydajnością przy wysokim obciążeniu, ponieważ blokują wątki podczas operacji wejścia/wyjścia. Spring Framework wspiera ten paradygmat, oferując narzędzia takie jak Spring MVC dla aplikacji webowych, czy Spring Data JPA do obsługi baz danych [2]. Porównanie obu tych podejść jest kluczowe, ponieważ każde z nich ma swoje zalety i wady. Wybór między nimi powinien zależeć od specyficznych wymagań aplikacji, takich jak oczekiwana wydajność, liczba użytkowników oraz dostępne zasoby. Decyzja o wyborze podejścia wpływa na architekturę aplikacji, łatwość jej utrzymania oraz ogólną wydajność w środowisku produkcyjnym.

Celem artykułu jest zbadanie wydajności i skalowalności aplikacji webowej zrealizowanej w podejściu reaktywnym i imperatywnym z wykorzystaniem Spring Framework. Wydajność i skalowalność są bardzo ważne, ponieważ wpływają na zdolność aplikacji do radzenia sobie z rosnącą liczbą użytkowników. Aby osiągnąć oczekiwane działanie aplikacji, kluczowe jest użycie odpowiednich technologii, budowa solidnej architektury i stosowanie efektywnej strategii optymalizacji.

W pracy postawiono następujące hipotezy badawcze:

1. Aplikacja zrealizowana w podejściu reaktywnym charakteryzuje się krótszym czasem odpowiedzi niż aplikacja imperatywna w sytuacjach wysokiego obciążenia.
2. W przypadku rosnącej liczby użytkowników, aplikacja reaktywna efektywniej wykorzystuje zasoby CPU i RAM w porównaniu do aplikacji imperatywnej.

Aby zweryfikować te hipotezy, zastosowano następująca metodę badawczą:

1. Projektowanie i implementacja: Stworzono dwie wersje aplikacji webowej: reaktywną i imperatywną, wykorzystując Spring Framework. Każda wersja aplikacji spełnia te same wymagania funkcjonalne, co pozwala na bezpośrednie porównanie ich wydajności i skalowalności.
2. Testy wydajności i skalowalności: Przeprowadzono testy obu wersji aplikacji, używając narzędzia Apache JMeter do symulacji różnych scenariuszy obciążenia. Testy te obejmowały zarówno warunki, gdzie 100 i 1000 użytkowników jednocześnie wykonuje operacje, aby ocenić, jak każda aplikacja radzi sobie pod dużym obciążeniem.
3. Pomiary parametrów wydajności: Zmierzono i porównano parametry wydajnościowe, takie jak czas odpowiedzi aplikacji, zużycie CPU i RAM przez wirtualną maszynę Javy, oraz szczytową liczbę aktywnych wątków tworzonych przez aplikacje. Parametry te są istotne dla oceny efektywności zużycia zasobów.
4. Monitorowanie i analiza: Zastosowano narzędzie VisualVM do monitorowania wydajności aplikacji Java. VisualVM umożliwiło szczegółową analizę zużycia zasobów przez obie wersji aplikacji w czasie rzeczywistym.

2. Przegląd literatury

W kontekście programowania, wybór odpowiedniego podejścia może istotnie przyspieszyć działanie aplikacji. W wyniku przeglądu literatury można zauważyć, że większość prac koncentruje się na badaniu korzyści płynących z paradygmatu reaktywnego w kontekście wysokich obciążeń serwera. Pomimo tego, brakuje bezpośrednich porównań z imperatywnym podejściem w dostępnym zasobie literatury naukowej. Analizując dostępne publikacje, można zaobserwować, że dyskusje na temat reaktywności stają się bardziej znaczące w miarę wzrostu obciążenia serwera, co sugeruje, że korzyści z tego podejścia stają się bardziej zauważalne w kontekście aplikacji charakteryzujących się bardziej intensywnym ruchem.

W artykule naukowym [3] przeprowadzono eksperyment kontrolowany z udziałem 127 studentów informatyki, którzy zostali podzieleni na dwie grupy: grupę RP (programowanie reaktywne) i grupę OO (programowanie obiektowe). Każda grupa otrzymała 10 zadań polegających na zrozumieniu zachowania aplikacji reaktywnych napisanych w stylu RP lub OO. Zmierzono poprawność i czas odpowiedzi na pytania dotyczące aplikacji, a także poziom umiejętności programistycznych i preferencje uczestników. Wyniki wykazały, że RP poprawia poprawność zrozumienia oprogramowania, nie wpływa negatywnie na czas zrozumienia oprogramowania, nie wymaga wyższego poziomu umiejętności programistycznych i jest preferowane przez programistów. Wysunięto wniosek, że RP jest bardziej odpowiednie do tworzenia aplikacji reaktywnych, które muszą obsługiwać duże ilości żądań i zapewniać wysoką dostępność i niezawodność.

Eksperyment przeprowadzony przez autorów pracy [4], w którym porównywano efektywność programowania głosowego w paradygmatach reaktywnym i imperatywnym, wykorzystał oprogramowanie do rozpoznawania mowy Talon Voice. Dwóch programistów miało za zadanie stworzyć dwie identyczne aplikacje w języku Java, różniące się jedynie paradygmatem programowania. Pomiar efektywności obejmował liczbę użytych słów, sylab i przerw, a analiza statystyczna została przeprowadzona za pomocą testu t-Studenta. Wyniki potwierdziły, że paradygmat reaktywny okazał się bardziej efektywny, wymaga on mniejszej ilości słów, sylab i przerw w porównaniu do paradygmatu imperatywnego. Wnioski z eksperymentu podkreślają, że reaktywny paradygmat jest zgodny ze sposobem myślenia człowieka, lepiej odzwierciedla logikę biznesową i przepływ danych, co ułatwia utrzymanie kodu. Ponadto, autorzy zauważyli potencjał reaktywnego paradygmatu do poprawy jakości kodu i wydajności aplikacji, szczególnie dzięki lepszej obsłudze błędów, asynchroniczności i współbieżności.

Autorzy pracy [5] przygotowali trzy testowe aplikacje oparte na szkielecie Spring Boot, Micronaut i Quarkus, realizujące operacje CRUD na bazie danych. Badania obejmowały pomiary czasu odpowiedzi serwera na różne rodzaje żądań przy różnym obciążeniu, mierzone jako liczba wysyłanych żądań na sekundę. Narzędzie Gatling generowało raporty z wynikami i statystykami. Dodatkowo, autorzy analizowali niezawodność aplikacji oraz porównywali objętość kodu, mierzoną liczbą linii kodu dla każdego szkieletu. Wyniki badań wykazały, że w większości przypadków szkielet Micronaut osiągał najlepsze rezultaty pod względem wydajności, niezawodności i objętości kodu.

3. Eksperyment badawczy

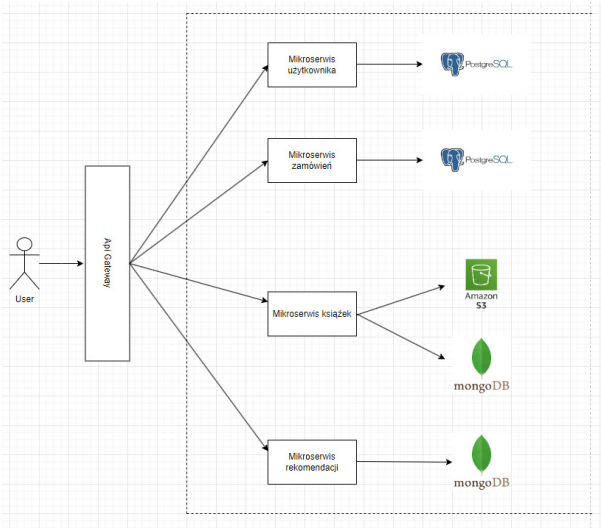
Zbadanie różnic w wydajności podejścia reaktywnego i imperatywnego wymagało stworzenia dwóch bliźniaczych aplikacji. Obydwie oparte są na tej samej architekturze mikroserwisów oraz posiadają identyczne wymagania funkcjonalne. Aplikacja to sklep internetowy z książkami, umożliwiający użytkownikom zakładanie kont, składanie zamówień, oraz generowanie dynamicznych rekomendacji książek na podstawie historii zamówień i preferencji użytkowników.

Rysunek 1 przedstawia architekturę testowej aplikacji. Składa się ona z pięciu mikroserwisów, które zajmują się określonymi operacjami:

1. Mikroserwis Api Gateway zaimplementowany przy użyciu Spring Cloud Gateway [6] pełni funkcję dostępu do mikroserwisów oraz wstępnej autoryzacji użytkowników przy użyciu JSON Web Tokens [7]. Po uwierzytelnieniu, Api Gateway dodaje identyfikator zautoryzowanego użytkownika do nagłówka każdego zapytania kierowanego do mikroserwisów.
2. Mikroserwis użytkownika jest odpowiedzialny za wszystkie operacje związane z zarządzaniem kontem użytkownika, w tym rejestracją, logowaniem, a także dodawaniem i edycją preferencji gatunkowych książek. Mikroserwis ten współpracuje z mikroserwisem

rekomendacji, informując go natychmiast o zmianach w preferencjach użytkownika, co umożliwia dynamiczne dostosowanie sugerowanych książek.

3. Mikroserwis zamówień jest odpowiedzialny za obsługę zamówień użytkownika. Jego kluczową funkcjonalnością jest przetwarzanie nowych zamówień, a także możliwość przeglądania historii zamówień i sprawdzania listy aktualnych bestsellerów. Mikroserwis ten komunikuje się z mikroserwisem książek, odpytując o dostępność magazynową zamawianych pozycji podczas składania zamówienia.
4. Mikroserwis książek jest odpowiedzialny za zarządzaniem stanem magazynowym książek, umożliwiając operacje takie jak pobieranie listy wszystkich książek, wyszukiwanie po gatunkach, oraz dodawanie książek do bazy danych. Komunikuje się również z usługą Amazon S3, gdzie przesyła pliki graficzne okładek książek, a w bazie danych zapisywany jest URL do tych zasobów.
5. Mikroserwis rekomendacji jest odpowiedzialny za generowanie spersonalizowanych rekomendacji dla użytkowników. Wykorzystuje do tego dostępne książki oraz preferencje użytkowników. Na podstawie tych danych tworzy rekomendacje, które są następnie zapisywane w bazie danych.



Rysunek 1: Architektura aplikacji.

Tematem badań jest sprawdzenie, które podejście programistyczne jest bardziej wydajne i skalowalne w takim samym środowisku testowym. Stworzone zostały dwie aplikacje napisane w imperatywnej i reaktywnej technice programowania. Zbadane zostały czasy przetwarzania zapytań, maksymalne wykorzystanie CPU, RAM oraz maksymalna liczba stworzonych aktywnych wątków podczas wykonywania scenariuszy testowych. Analiza objęła także działanie aplikacji z mikroserwisem będącym bramką Api oraz bez niego, aby zbadać wpływ tego elementu na efektywność procesów. Zbadano oba warianty, mając na celu ustalenie, czy obecność mikroserwisu Api Gateway, który jest bardzo powszechnym elementem w architekturze mikroserwisowej, wpływa pozytywnie na wydajność jednego z podejść programowania lub też czy jego brak może przynieść lepsze wyniki. Scenariusze badawcze zostały przeprowadzone według następujących konfiguracji:

- imperatywne podejście z wykorzystaniem mikroserwisu Api Gateway,
- imperatywne podejście bez wykorzystania mikroserwisu Api Gateway,
- reaktywne podejście z wykorzystaniem mikroserwisu Api Gateway,
- reaktywne podejście bez wykorzystania mikroserwisu Api Gateway.

W Tabeli 1 widoczna jest specyfikacja maszyny testowej, na której zostały przeprowadzone badania.

Tabela 1: Specyfikacja maszyny testowej

Komponent	Specyfikacja
Procesor	AMD Ryzen 5 5600H 12 CPUs 3.3 GHz
Pamięć RAM	32GB DDR4
Dysk	512 GB, interfejs M.2
System operacyjny	Windows 11 home 64-bit

Scenariusze testowe, widoczne w Tabeli 2 zostały powtórzone dziesięciokrotnie. Próby badawcze zostały przeprowadzone na tej samej maszynie testowej oraz w takich samych warunkach. W celu symulacji obciążenia scenariusze zostały wykonane dla 100 i 1000 jednoczesnych zapytań. W przypadku podejścia z wykorzystaniem mikroserwisu Api Gateway konieczne było pobranie tokenów JWT z bazy danych, które następnie zapisano w pliku CSV. Zgromadzone tokeny zostały wykorzystane w narzędziu Apache JMeter jako nagłówek autoryzacyjny. Dla architektury bez elementu pełniącego rolę bramki dostępu, pobrano z bazy danych identyfikatory użytkowników, które następnie stosowano jako nagłówki HTTP.

Tabela 2: Scenariusze badawcze

Scenariusz	Opis
Złożenie zamówienia	Przesłanie obiektu zamówienia. Sprawdzenie dostępności pozycji. Aktualizacja stanu magazynowego, zapisanie zamówienia
Dodanie książki	Przesłanie obiektu reprezentującego książkę. Zapis zdjęcia w usłudze chmurowej S3. Zapisanie obiektu książki w bazie danych.
Pobranie książek	Pobranie wszystkich rekordów książek z bazy danych.
Dodanie preferencji użytkownika	Dodanie preferencji gatunkowych książek. wygenerowanie rekomendacji książkowych.
Rejestracja użytkownika	Zarejestrowanie nowego użytkownika w systemie, wygenerowanie tokenu dostępu.
Pobranie rekomendacji	Pobranie z bazy danych wygenerowanych dla użytkownika rekomendacji książkowych.

4. Wyniki badań

Tabela 3 i Tabela 4 zawierają wyniki pomiarów średnich czasów odpowiedzi uzyskanych podczas realizacji scenariuszy badawczych w testowanych aplikacjach.

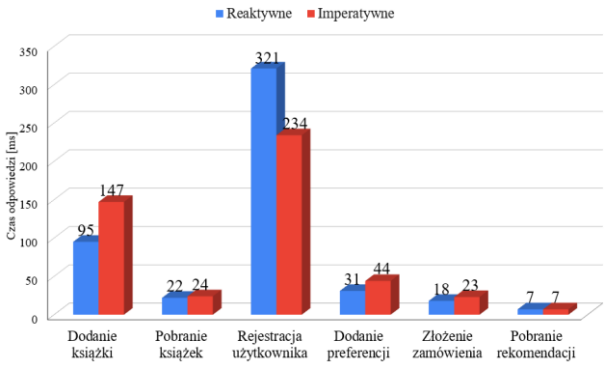
Tabela 3: Zgromadzone wyniki średnich czasów odpowiedzi aplikacji napisanej w podejściu imperatywnym

Wariant imperatywny				
Scenariusz	Średni czas odpowiedzi [ms]			
	Z bramką Api		Bez bramki Api	
	100 użytkowników	1000 użytkowników	100 użytkowników	1000 użytkowników
Dodanie książki	147	3386	99	3020
Pobranie książek	24	2168	19	1770
Rejestracja użytkownika	234	6138	155	6066
Dodanie preferencji	44	1085	33	745
Złożenie zamówienia	23	280	14	257
Pobranie rekomendacji	7	31	4	15

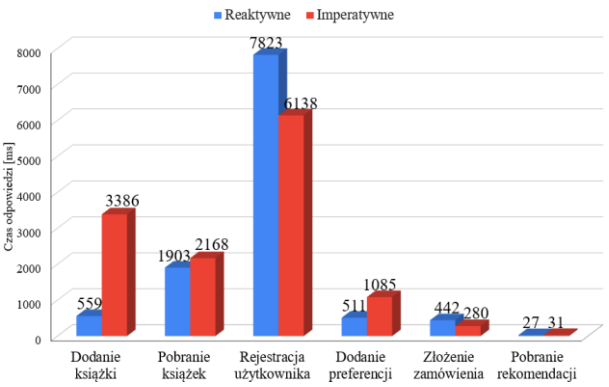
Tabela 4: Zgromadzone wyniki średnich czasów odpowiedzi aplikacji napisanej w podejściu reaktywnym

Wariant reaktywny				
Scenariusz	Średni czas odpowiedzi [ms]			
	Z bramką Api		Bez bramki Api	
	100 użytkowników	1000 użytkowników	100 użytkowników	1000 użytkowników
Dodanie książki	95	559	91	549
Pobranie książek	22	1903	18	1491
Rejestracja użytkownika	321	7823	305	7400
Dodanie preferencji	31	511	15	351
Złożenie zamówienia	18	442	15	388
Pobranie rekomendacji	7	37	4	30

Wykres widoczny na Rysunku 2 przedstawia porównanie średnich czasów odpowiedzi aplikacji dla 100 użytkowników z podziałem na dwa podejścia: reaktywne i imperatywne. Podejście reaktywne wykazuje krótsze czasy odpowiedzi w większości scenariuszy. Dla scenariusza „dodanie książki" czas odpowiedzi wynosi 95 ms dla podejścia reaktywnego i 147 ms dla imperatywnego, co oznacza, że podejście reaktywne jest szybsze o około 35%. W scenariuszu „rejestracja użytkownika" podejście imperatywne okazuje się szybsze z czasem 234 ms w porównaniu do 321 ms dla reaktywnego, co oznacza różnicę około 27%. W przypadku scenariusza „dodania preferencji" podejście reaktywne wynosi 31 ms, a imperatywne 44 ms, co daje różnicę 30%. Dla „złożenia zamówienia" czas odpowiedzi wynosi 18 ms dla reaktywnego i 23 ms dla imperatywnego, co oznacza różnicę około 22%. W scenariuszu „pobranie rekomendacji" oba podejścia mają taki sam czas odpowiedzi wynoszący 7 ms. Na podstawie wykresu można zauważyć, że podejście reaktywne zazwyczaj prowadzi do krótszych czasów odpowiedzi z wyjątkiem scenariusza „rejestracja użytkownika", gdzie podejście imperatywne okazuje się szybsze.



Rysunek 2: Średnie czasy odpowiedzi aplikacji dla 100 użytkowników z podziałem na mikroserwisy, porównujące podejście reaktywne i imperatywne.



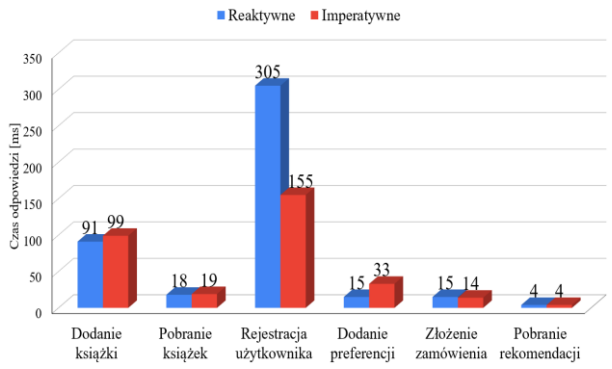
Rysunek 3: Średnie czasy odpowiedzi aplikacji dla 1000 użytkowników z podziałem na mikroserwisy, porównujące podejście reaktywne i imperatywne.

Analizując Rysunek 3 można zauważyć, że dla scenariusza „dodanie książki" czas odpowiedzi wynosi 559 ms dla podejścia reaktywnego i 3386 ms dla imperatywnego, co oznacza, że podejście reaktywne jest szybsze o prawie 84%. W przypadku scenariusza „pobranie książki" czas odpowiedzi dla reaktywnego wynosi 1903 ms, a dla imperatywnego 2168 ms, co oznacza różnicę 12%. W scenariuszu „rejestracja użytkownika" podejście reaktywne okazuje się szybsze z czasem 7823 ms w porównaniu do 6138 ms dla imperatywnego, co oznacza różnicę 22%. W przypadku „dodania preferencji" podejście reaktywne wynosi 511 ms, a imperatywne 1085 ms, co przekłada się różnicę wynoszącą aż 53%. Dla 100 użytkowników podejście reaktywne wykazuje lepszą wydajność niż imperatywne w większości scenariuszy, z

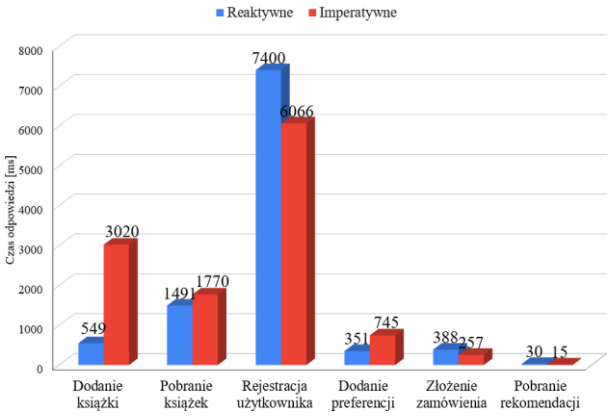
wyjątkiem „rejestracji użytkownika”, gdzie podejście imperatywne jest szybsze. Różnice te nie są drastyczne, ale podejście reaktywne wykazuje krótsze czasy odpowiedzi w większości przypadków. W przypadku 1000 użytkowników podejście reaktywne nadal przeważa w większości scenariuszy, jednak różnice stają się bardziej wyraźne. Szczególnie zauważalne jest to w scenariuszu „dodanie książki”, gdzie czas odpowiedzi w podejściu reaktywnym wynosi 559 ms, a w imperatywnym 3386 ms, co oznacza różnicę o 84%. W scenariuszu „rejestracja użytkownika” czas odpowiedzi dla podejścia reaktywnego wynosi 7823 ms w porównaniu do 6138 ms dla podejścia imperatywnego, co oznacza poprawę czasu odpowiedzi o 22%.

Przy obciążeniu 1000 użytkowników, przewaga podejścia reaktywnego staje się bardziej wyraźna w większości scenariuszy, podczas gdy przy 100 użytkownikach różnice są mniej znaczące. Wyjątkiem są scenariusze „złożenie zamówienia” i „rejestracja użytkownika”, gdzie podejście imperatywne okazuje się szybsze zarówno dla 100, jak i 1000 użytkowników. Różnica jest bardziej znacząca dopiero przy większym obciążeniu. Krótszy czas odpowiedzi w przypadku scenariuszy złożenia zamówienia i rejestracji użytkownika wskazuje, że dla procesów, silnie zależnych od sekwencyjnych operacji bazodanowych i nie wymagających intensywnych operacji I/O, wariant imperatywny wydaje się bardziej wydajny. W kontekście operacji I/O, takich jak przesyłanie plików, które miały miejsce w scenariuszu dodanie książki, model reaktywny pozwala na lepszą wydajność dzięki asynchronicznemu przetwarzaniu, które oznacza, że wątki nie są blokowane, a system może obsługiwać wiele żądań jednocześnie bez potrzeby tworzenia dodatkowych wątków. W efekcie, system może efektywniej zarządzać zasobami.

Porównując wykresy widoczne na Rysunkach 4 i 5 z wykresami na Rysunkach 2 i 3 zauważyć można, że oba podejścia odczuły negatywny wpływ obecności mikroserwisu Api Gateway, jednak w różnym stopniu.



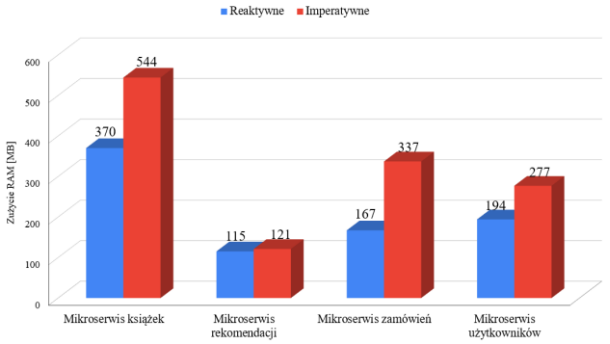
Rysunek 4: Średnie czasy odpowiedzi aplikacji dla 100 użytkowników z podziałem na mikroserwisy, porównujące podejście reaktywne i imperatywne w architekturze bez mikroserwisu Api Gateway.



Rysunek 5: Średnie czasy odpowiedzi aplikacji dla 1000 użytkowników z podziałem na mikroserwisy, porównujące podejście reaktywne i imperatywne w architekturze bez mikroserwisu Api Gateway.

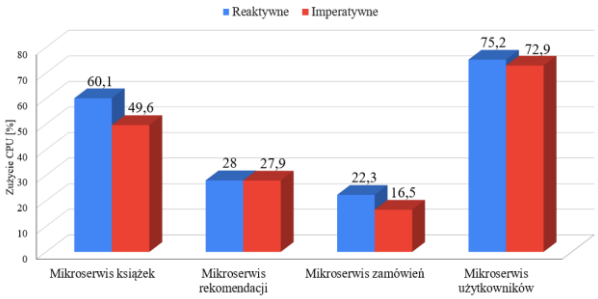
Wariant reaktywny, choć również zareagował negatywnie na Api Gateway, lepiej radził sobie z tym elementem architektury, zwłaszcza w scenariuszach „dodanie książki” oraz „rejestracja użytkownika”. Na podstawie przedstawionych tabel i wykresów można stwierdzić, że wariant reaktywny jest lepszym wyborem w aplikacjach, które wymagają skalowalnego i efektywnego przetwarzania operacji I/O, szczególnie, gdy kluczowa jest minimalizacja czasów odpowiedzi. Wariant imperatywny wydaje się być bardziej odpowiedni w środowiskach, gdzie operacje są sekwencyjne i mniej złożone.

Zużycie zasobów systemowych badano w kontekście wykorzystania zasobów przez JVM, co gwarantuje dokładniejszą i bardziej kontrolowaną metodę monitorowania aplikacji Java. Narzędzie VisualVM umożliwiło rejestrowanie zużycie zasobów systemowych w trakcie testów. Dane zostały zgromadzone poprzez dziesięciokrotne wykonanie każdego scenariusza badawczego. Przeanalizowano procentowe zużycie CPU/RAM oraz liczbę stworzonych aktywnych wątków podczas działania aplikacji. Ze zgromadzonych danych zostały wybrane szczytowe wartości osiągnięte przez każdy z mikroserwisów podczas wykonywania scenariuszy badawczych. Wyniki zostały zaprezentowane w postaci trzech wykresów, co pozwala na precyzyjne porównanie różnic między dwoma badanymi podejściami programistycznymi.



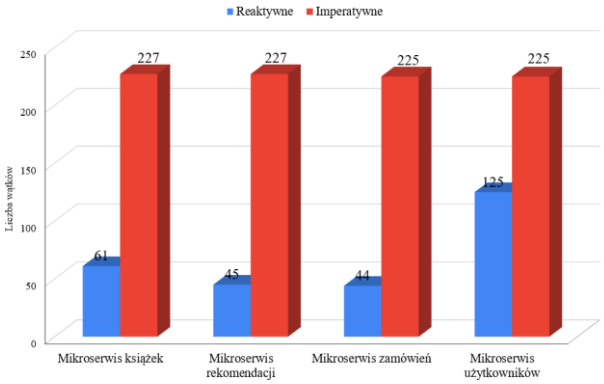
Rysunek 6: Maksymalne zużycie pamięci RAM z podziałem na poszczególne mikroserwisy, podczas wykonywania scenariuszy testowych.

Wykres widoczny na Rysunku 6 przedstawia maksymalne zużycie pamięci RAM przez JVM. Każdy mikroserwis w przypadku podejścia reaktywnego wykorzystywał mniej pamięci RAM. Największa różnica widoczna jest w przypadku mikroserwisu książek, gdzie podczas wykonywania zadań testowych maksymalnie zużycie RAM było niższe aż o 174 MB, co oznacza, że aplikacja imperatywna zużyła o 47% więcej pamięci.



Rysunek 7: Maksymalne zużycie CPU podczas wykonywania scenariuszy testowych.

Dane z Rysunku 7 wskazują, że w przypadku aplikacji, gdzie wszystkie mikroserwisy zostały napisane asynchronicznie, maksymalne zużycie CPU dla każdego mikroserwisu było większe w porównaniu z podejściem imperatywnym. Największą różnicę zaobserwowano w przypadku mikroserwisu obsługującego książki, gdzie mikroserwis napisany w podejściu imperatywnym zużył o 10,6% mniej zasobów procesora w porównaniu do wariantu reaktywnego. Może to wynikać z faktu, że wariant imperatywny efektywniej wykorzystuje dostępne zasoby procesora dzięki zoptymalizowanemu zarządzaniu wątkami lub innym mechanizmom synchronizacji. W przypadku mikroserwisu rekomendacji, którego głównym zadaniem była operacja zapisu, odczytu i wygenerowania rekomendacji książkowych zużycie procentowe CPU było niemalże takie same, z przewagą 0,1% na korzyść podejścia imperatywnego.



Rysunek 8: Maksymalna liczba stworzonych aktywnych wątków podczas wykonywania scenariuszy testowych.

Wykres widoczny na Rysunku 8 przedstawia maksymalną liczbę stworzonych aktywnych wątków przez aplikację. Aplikacja imperatywna stworzyła w każdym mikroserwisie znacznie większą liczbę aktywnych wątków. W aplikacjach imperatywnych często korzysta się z mechanizmów synchronizacji i blokad, które mogą prowadzić do tworzenia większej liczby wątków, co skutkuje zwiększonym zużyciem zasobów. W podejściu reaktywnym wiele operacji jest wykonywanych asynchronicznie, co oznacza, że aplikacja może obsługiwać wiele żądań na jednym wątku. Największą różnicę możemy zaobserwować w przypadku mikroserwisu zamówień i mikroserwisu rekomendacji, gdzie aplikacja napisana w podejściu imperatywnym używała ponad pięciokrotnie więcej aktywnych wątkach od aplikacji reaktywnej. Reaktywne aplikacje często korzystają z mechanizmów buforowania i przetwarzania strumieniowego danych. To może prowadzić do częstego przesyłania danych przez aplikację, co zwiększa obciążenie procesora, ale jednocześnie ogranicza zużycie pamięci RAM poprzez przetwarzanie danych na bieżąco. Aplikacja imperatywna stworzyła znacznie więcej aktywnych wątków, które zużywają zasoby systemowe, a zwłaszcza pamięć RAM [8].

5. Wnioski

Wyniki badań wykazały, że aplikacja reaktywna, podczas wykonywania scenariuszy badawczych, była zazwyczaj szybsza od wariantu imperatywnego, szczególnie w sytuacjach wysokiego obciążenia. Jednak istniały również scenariusze, w których aplikacja imperatywna radziła sobie lepiej. Co więcej, wyniki potwierdziły, że aplikacja reaktywna lepiej radziła sobie z wysoką skalowalnością i efektywniej wykorzystywała pamięć RAM, szczególnie w przypadku operacji I/O, takich jak przesyłanie plików. Natomiast aplikacja imperatywna tworzyła znacznie więcej aktywnych wątków, co negatywnie wpływało na zużycie pamięci RAM, ale wykorzystała w mniejszym stopniu CPU. Podsumowując, wyniki badań sugerują, że oba podejścia mają swoje zalety i zastosowanie w zależności od wymagań aplikacji. Aplikacja reaktywna jest preferowana w przypadku operacji wymagających skutecznego i skalowalnego przetwarzania operacji I/O, podczas gdy aplikacja imperatywna wydaje się być bardziej odpowiednia w przypadku prostych, sekwencyjnych operacji. Dzięki asynchronicznemu przetwarzaniu, aplikacje reaktywne mogą obsługiwać dużą liczbę równoczesnych połączeń bez nadmiernego zużywania zasobów systemowych. To podejście często używane jest dla systemów czasu rzeczywistego, aplikacji internetowych o wysokim obciążeniu oraz mikroserwisów, które muszą niezawodnie i szybko przetwarzać żądania. Obecność mikroserwisu Api Gateway wprowadza dodatkowe opóźnienia, które wpływają w różnym stopniu na wydajność obu wariantów. Na jego obecność wariant reaktywny reagował mniejszymi opóźnieniami niż imperatywne podejście. Dodatkowo analizując kod aplikacji testowych można stwierdzić, że podejście imperatywne jest bardziej intuicyjne, ponieważ operacje są wykonywane sekwencyjnie. Ułatwia to testowanie i debugowanie aplikacji. Ponadto, istnieje wiele dojrzałych narzędzi, szkieletów programistycznych i bibliotek, które od lat wspierają programowanie imperatywne, co może przekładać się na łatwiejszy rozwój i utrzymanie mikroserwisów.

Decyzja o wyborze konkretnego podejścia programistycznego powinna zostać rozpatrzona analizując specyfikację aplikacji, jej wymagań oraz dostępnych zasobów. Tworząc aplikację, które mają na celu efektywne przetwarzanie operacji I/O należy zastanowić się nad wyborem podejścia reaktywnego, podczas gdy prostsze aplikacje mogą zostać napisane w podejściu imperatywnym. Warto również rozważyć hybrydowy wariant, w którym kluczowe i krytyczne moduły aplikacji korzystają z podejścia reaktywnego, a prostsze, mniej wymagające z podejścia imperatywnego, aby w optymalny sposób wykorzystać zasoby i spełnić wymagania dotyczące skalowalności i wydajności.

Literatura

[1] Poradnik do budowy reaktywnych systemów w języku Java, https://www.baeldung.com/java-reactive-systems, [10.11.2023].

[2] Poradnik do pisania aplikacji z użyciem Spring Framework w Javie, https://www.baeldung.com/spring-tutorial, [10.11.2023].

[3] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, M. Mezini, On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study, In IEEE Transactions on Software Engineering, (2017), 1125–1143, https://doi.org/10.1109/TSE.2017.2655524.

[4] M. Lagergren, M. Soneryd, Programming by voice: Efficiency in the Reactive and Imperative Paradigm, Bachelor's thesis, Mid Sweden University, Östersund, 2021.

[5] M. Jeleń, M. Dzieńkowski, The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus, Journal of Computer Sciences Institute 21 (2021) 287–294, https://doi.org/10.35784/jcsi.2724.

[6] Dokumentacja Spring Cloud Gateway, https://spring.io/projects/spring-cloud-gateway, [10.11.2023].

[7] Dokumentacja projektu JWT.io, https://jwt.io/, [10.11.2023].

[8] B. Goetz, Java Concurrency in practice, Reading, Massachusetts, Addison–Wesley, 2006.