JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 21 (2021) 287–294
Received: 01 September 2021
Accepted: 04 October 2021

The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus

Analiza porównawcza szkieletów programistycznych języka Java: Spring Boot, Micronaut oraz Quarkus

Maciej Jeleń*, Mariusz Dzieńkowski

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

Abstract
The aim of the work is a comparative analysis of three frameworks designed for building web applications for the Java programming language: Spring Boot 2.4.4, Micronaut 2.5.4 and Quarkus 1.13.4.Final. Test applications were prepared, equipped with the same functionality as used in the experiment consisting in measuring the server response times to a request of POST, GET, PUT and DELETE performing operations on the database. For each test application, the scenario aimed at measuring the time of handling requests under various load conditions was repeated five times. During each repetition of the scenario, the load which was the average number of requests sent per second by virtual users was increased. In parallel with performance tests, the reliability of the test applications was measured. Reliability was defined as the percentage of requests sent to the server that ended in a failure. The comparative analysis also took into consideration the volume of the code of the test applications based on the selected frameworks. The performed analyses showed that in terms of most of the criteria considered in this work Micronaut proved to be the best framework.

Keywords: web application; frameworks of the Java programming language; performance analysis; Spring Boot; Micronaut; Quarkus

Streszczenie
Przedmiotem tej pracy jest analiza porównawcza trzech szkieletów programistycznych do budowy aplikacji internetowych dla języka Java: Spring Boot 2.4.4, Micronaut 2.5.4 oraz Quarkus 1.13.4.Final. Przygotowano aplikacje testowe, wyposażone w tą samą funkcjonalność, które wykorzystano w eksperymencie, polegającym na pomiarze czasów odpowiedzi serwera na żądania typu POST, GET, PUT i DELETE – realizujące operacje na bazie danych. Dla każdej aplikacji testowej, powtórzono pięciokrotnie scenariusz, który miał na celu zmierzyć czas obsługi żądań w różnych warunkach obciążeniowych. Podczas każdego powtórzenia zwiększano wielkość obciążenia, które oznaczało średnią liczbę wysyłanych żądań na sekundę przez wirtualnych użytkowników. Równolegle z badaniami wydajności wykonano pomiary niezawodności aplikacji testowych. Niezawodność zdefiniowano jako odsetek żądań wysyłanych do serwera, które zakończyły się niepowodzeniem. W porównaniach wzięto również pod uwagę objętość kodu aplikacji testowych opartych na wybranych szkieletach. Z przeprowadzonych analiz wynikło, że pod względem większości rozpatrywanych w ramach tej pracy kryteriów najlepszym szkieletem programistycznym okazał się Micronaut.

Słowa kluczowe: aplikacja internetowa; szkielety programistyczne języka Java; analiza wydajności; Spring Boot; Micronaut; Quarkus

*Corresponding author
Email address: maciej.jelen@pollub.edu.pl (M. Jeleń)

1. Wstęp

W czasach, gdy internet nie był jeszcze tak rozpowszechniony, jak to jest obecnie, większość aplikacji komputerowych była dostępna w formie desktopowej, co oznaczało, że każdy klient musiał zainstalować aplikację oddzielnie na swoim komputerze. Niosło to ze sobą pewne konsekwencje, takie jak na przykład konieczność uaktualniania danego oprogramowania u każdego klienta, co zmniejszało produktywność oraz zabierało więcej zasobów. W 1995 roku pojawiła się pierwsza wersja języka Java, umożliwiająca tworzenie apletów, które wkrótce mogły być uruchamiane w przeglądarkach internetowych.

W celu przyspieszenia tworzenia aplikacji internetowych w Javie, powstały szkielety programistyczne dla tego języka – zwane często z języka angielskiego frameworkami. Ich stosowanie ułatwia tworzenie aplikacji internetowych oraz redukuje ilość powtarzalnego kodu, jaki programista musiał dodać przed faktycznym utworzeniem głównej części oprogramowania. Wynikało to z tego, że przystępując do implementacji logiki biznesowej, należało zaimplementować takie elementy jak konfiguracje, połączenia z bazami danych czy innymi serwisami internetowymi za pośrednictwem interfejsu API (ang. Application Programming Interface – Interfejs Programowania Aplikacji).

Jednym z najbardziej popularnych obecnie szkieletów programistycznych opartych na języku Java jest Spring Framework, który jest następcą technologii Java EE. Na bazie tych szkieletów powstawały i ciągle powstają nowe rozwiązania, wykorzystujące zalety swoich poprzedników oraz naprawiając ich wady. Przykładem

nowych frameworków są rozpatrywane w ramach tej pracy Micronaut oraz Quarkus. Pierwszy szkielet został utworzony przez grupę programistów, którzy wcześniej przyczynili się do powstania frameworka Grails, natomiast drugi jest efektem prac inżynierów firmy Red Hat.

Obecnie istnieje bardzo duża liczba języków programowania, które mają swoje frameworki - najlepszym przykładem jest język Java. Szeroki wybór oraz różnorodność powodują, że trudno jest się zdecydować na najlepsze, najbardziej optymalne rozwiązanie dostosowane do konkretnego przedsięwzięcia programistycznego. W związku z tym powstaje wiele publikacji, w których podejmowany jest ten problem, i które mają ułatwić i przyśpieszyć proces decyzyjny dotyczący wyboru właściwej technologii. W ramach tej pracy, przeanalizowano trzy szkielety programistyczne dla języka Java: Spring Boot, Micronaut oraz Quarkus, które mają uniwersalne zastosowania aplikacyjne.

2. Przegląd literatury

W przypadku aplikacji internetowych, mających architekturę wielowarstwową, które są uniwersalne i skierowane do szerokiej grupy odbiorców, bardzo ważna jest wydajność i jakość wykonania, a także łatwość ich implementacji. Przedmiotem pracy [1] było porównanie dwóch szkieletów programistycznych: Spring Framework, którego pochodną jest Spring Boot oraz Play Framework. W tym celu przygotowano dwie aplikacje internetowe oparte na badanych szkieletach oraz przeanalizowano wydajność za pomocą narzędzia JMeter. Miarą użytą do porównań był czas wyrażony w milisekundach, w jakiej dany serwer odpowiadał na żądania 100 użytkowników wysyłanych w ciągu jednej sekundy. Wysyłane żądania polegały głównie na przypisaniu oceny studentowi i dodatkowo obejmowały operacje zalogowania się do i wylogowania się z systemu. Na podstawie uzyskanych wyników, wysunięto wnioski, że przy małym obciążeniu szkielet Play Framework był wydajniejszy od Spring Framework. Natomiast w warunkach dużego obciążenia, podczas wykonywania operacji, lepsze wyniki uzyskiwał Spring Framework.

Podobną tematykę podjęli autorzy pracy [2], w której zawarli wyniki analizy trzech szkieletów programistycznych (Enterprise Java Beans 2, Spring Framework, Grails) w środowisku maszyny wirtualnej Java. Jako aplikacji testowej, użyto systemu do automatyzacji procesów wstępnego testowania. Również w tym przypadku porównano czasy odpowiedzi na żądania każdej aplikacji opartej na danym szkielecie. Analizy przeprowadzano przy różnych obciążeniach – podając przez okres 1, 5, 10 i 60 sekund zestaw danych od 20 do 700 użytkowników na sekundę. Do badań wykorzystano bibliotekę JMeter. Podczas testów wysyłano stałą liczbę pakietów dla każdego przypadku. Porównanie tych trzech technologii pod kątem wydajności doprowadziło autorów do wniosku, że szkielet Grails wykorzystujący język Groovy jest platformą wydajniejszą dla realizowanego systemu zarówno dla małych jak i dużych obciążeń.

Quarkus na platformie GraalVM, która jest alternatywą dla wirtualnej maszyny Java, oprócz szkieletów Micronaut oraz Spring Boot na platformie OpenJDK 13, był przedmiotem kolejnej analizy wykonanej w ramach pracy [3]. Autorzy przeprowadzili test, który polegał na pomiarze czasu odpowiedzi serwera przy próbie odczytu danych - w postaci kolekcji obiektów metodą GET pochodzącą z protokołu HTTP. Uzyskane wyniki wydajności wykazały, że najlepiej w testach wypadł szkielet Quarkus wykorzystujący platformę GraalVM, zaś w przypadku platformy OpenJDK 13 lepsze rezultaty uzyskał Micronaut. Szczegółowe wyniki mówią o tym, że Quakus uruchamiany na natywnym obrazie GraalVM jest o prawie 82% szybszy niż ta sama aplikacja napisana w Spring Boot i wykorzystująca OpenJDK 13. Natomiast gdy obie aplikacje działają na platformie OpenJDK 13, wówczas Quarkus jest o 18% szybszy od Spring Boot. Porównując szkielety Quarkus i Micronaut działające na OpenJDK 13, różnica w szybkości między nimi wynosi 17% na korzyść tego drugiego.

W artykule [4] porównywano wydajność dwóch aplikacji opartych na szkieletach Spring Boot oraz Microsoft .NET, wykorzystujących dwa podobne do siebie języki programowania, którymi są Java oraz C#. Wydajność w tych badaniach była określana jako czas odpowiedzi serwera na zadane żądania dla aplikacji wykonanej w Spring Boot oraz Microsoft .NET Core. Dodatkowo w pracy, mierzona była także niezawodność aplikacji, wyrażana w procentach i definiowana jako liczba żądań wysyłanych do serwera, które nie zostały poprawnie obsłużone. Testy były wykonywane pod obciążeniem, które zmieniało się co 30 sekund. Najpierw żądania do serwera wysyłało 1000 użytkowników, następnie 2000, 4000, 8000, 16000, 32000 i w końcu 64000 użytkowników, w każdej sekundzie. W każdym z tych testów, dla żądań reprezentowanych przez metody protokołu HTTP: GET, POST, PUT oraz DELETE, przy małej liczbie użytkowników szkielety miały podobną wydajność. Natomiast przy większych obciążeniach, wyraźnie lepsze wyniki osiągała aplikacja wykorzystująca język C# i platformę programistyczną .NET Core. Biorąc pod uwagę wyniki niezawodności aplikacji opartych na porównywanych szkieletach, można stwierdzić że kształtowały się one na podobnych poziomach. Należy przy tym zaznaczyć, że procent zakończonych niepowodzeniem żądań rósł wraz ze wzrostem obciążenia serwera.

W kolejnym artykule [5] badano wydajność takich samych aplikacji opartych na jednym z czterech szkieletów programistycznych języka Java: Spring Boot, Micronaut, Quarkus oraz Javalin. Dokonano porównań, w których wzięto pod uwagę wydajność, traktowaną jako czas uruchomienia aplikacji w środowisku produkcyjnym i programistycznym, a także zużycie pamięci oraz obciążenie procesora. Wykonano także test wydajności żądań pobrania (metodą HTTP GET) oraz zapisu (metodą HTTP POST) danych. Do serwera wysyłano najpierw 8000 żądań pobierających 1000 rekordów z bazy danych oraz 8000 żądań typu POST, które dodawały 200 rekordów do bazy. Na podstawie uzyska-

nych wyników okazało się, że najwydajniejszym szkieletem jest Javalin, który wyróżniał się najniższym spośród innych zużyciem pamięci, najmniejszym obciążeniem procesora oraz najkrótszym czasem odpowiedzi na żądania.

W każdym z przytoczonych artykułów realizowano badania, które koncentrowały się głównie na analizie wydajności pod obciążeniem aplikacji internetowych opartych na jednym z framworków języka Java. W ramach tej pracy oprócz kwestii wydajnościowych, podjęto problem niezawodności systemu rozumianego jako liczba żądań, na które serwer poprawnie nie odpowiedział. Aspekt ten może mieć kluczowe znaczenie przy podejmowaniu decyzji wyboru technologii dla tworzonego oprogramowania.

3. Wykorzystane technologie i narzędzia

3.1. Java 11

Język Java jest obecnie jednym z najpopularniejszych języków programowania. W rankingu TIOBE [6] z lipca 2021 roku, zajmuje on wysokie trzecie miejsce, zaraz za językiem C oraz językiem Python. Java zaczerpnęła podstawowe koncepcje z takich języków jak Smalltalk i C++. Na bazie Smalltalk powstała maszyna wirtualna Javy i narzędzie do zarządzania pamięcią (ang. garbage collector). Z kolei z języka C++ Java przejęła dużą część składni. Głównymi koncepcjami tego języka jest obiektowość, dziedziczenie oraz niezależność architektury, co pozwala na uruchomienie kodu napisanego w tym języku na wszystkich dostępnych systemach operacyjnych takich jak Microsoft Windows, różnych dystrybucjach Linuxa np. Ubuntu, Mint, Fedora oraz na systemach firmy Apple. Wersja 11 języka Java jest ostatnią dostępną, stabilną i długoterminową wersją oznaczaną skrótem LTS (ang. Long Term Support) [7].

3.2. REST

REST (ang. Representional State Transfer) jest stylem architektury do budowy aplikacji internetowych, w którym nacisk jest kładziony na używanie dostępu do zasobów za pomocą metod protokołu HTTP z użyciem odpowiedniego URL (ang. Uniform Resource Locator). Architektura ta nakłada pewne restrykcje, takie jak nieużywanie podkreśleń („_”), używanie wyłącznie małych liter oraz słów w liczbie mnogiej [8]. W tym przypadku ciało żądania do serwera wysyła się najczęściej w formacie JSON. Wcześniej, do budowy aplikacji internetowych opartych na usługach sieciowych, wykorzystywany był protokół SOAP (ang. Simple Object Access Protocol). Obecnie SOAP jest wypierany przez REST, przede wszystkim ze względu na prostotę tego drugiego rozwiązania. Do głównych cech architektury REST należą [9]:

- jednolity interfejs komunikacyjny,
- podział na aplikacje klient-serwer,
- bezstanowość,
- przechowywanie danych w pamięci podręcznej.

3.3. Narzędzie Gatling

Gatling jest darmowym narzędziem do wykonywania testów wydajnościowych, udostępnionym na licencji Apache License 2.0 w 2011 roku [10]. Narzędzie to oparte jest głównie na języku Scala z użyciem składni DSL (ang. Domain Specific Language), nieblokującym serwerze Netty oraz szkielecie programistycznym Akka. Gatling został zaprojektowany z myślą o łatwości użycia, wysokiej wydajności i łatwości utrzymania [11]. Z jednej maszyny JVM można utworzyć kilka tysięcy współistniejących wirtualnych użytkowników i nie ma potrzeby tworzenia rozproszonej sieci maszyn do przeprowadzania testów [12].

Narzędzie to pozwala określić warunki przeprowadzanych testów pod obciążeniem takich jak: stała lub zmienna liczba użytkowników w każdej sekundzie, czas trwania testu, ustalenie wielkości wzrostu liczby użytkowników w zadanym interwale czasowym, włączenie opcji uśpienia testu w momencie oczekiwania na odpowiedź serwera, gdy potrzebna jest sekwencja żądań do serwera.

Główne właściwości narzędzia Gatling są wymienione poniżej [13]:

- obsługuje HTTP i JMS (Java Message Service),
- skrypty są pisane w języku Scala z wykorzystaniem łatwego w użyciu DSL,
- składnia Scala DSL dobrze współpracuje z IDE (auto-uzupełnianie, refaktoring, kontrola wersji, debugowanie testów) i pozwala na szybkie pisanie i łatwe utrzymywanie istniejących scenariuszy,
- dane testowe do scenariuszy można przekazywać z plików w formacie CSV, TSV, JSON, SSV oraz zbiorów danych Redis,
- udostępnia asercje i sprawdza, czy mogą one być wykonane na otrzymanej odpowiedzi,
- posiada wbudowany rejestrator scenariuszy, który przechwytuje komunikację między przeglądarką, a serwerem,
- generuje bogate i przyjazne, graficzne, łatwo konfigurowalne raporty w formie pliku HTML,
- posiada rozszerzenia do prostej integracji z narzędziami takimi jak Maven, Jenkins i SBT.

Do odwzorowania typowego zachowania użytkowników, testerzy definiują scenariusze w postaci skryptów przekazywanych następnie do Gatlinga. Opisem testu obciążeniowego jest symulacja, która określa, jaki scenariusz będzie realizowany i w jaki sposób dodawani (wstrzykiwani) będą nowi użytkownicy. Wprowadzenie danych do scenariusza może być realizowane z różnych źródeł za pomocą komponentu zwanego feederem, który mapuje zmienne z wartościami i dostarcza je do sesji użytkownika wirtualnego [14].

4. Porównywane szkielety programistyczne

Dla Javy powstało bardzo wiele szkieletów programistycznych, wśród których jedne są bardziej, inne mniej popularne. Ich zadaniem jest m.in. ułatwienie pracy programistom, zwiększenie bezpieczeństwa aplikacji poprzez zastosowanie SSL, a także zaoferowanie pod-

stawowej autoryzacji hasłem i loginem oraz automatyzacja wielu procesów związanych szczególnie z początkową fazą realizacji projektów, takich jak konfiguracja bazy danych.

W ramach pracy została przeprowadzona analiza porównawcza trzech platform programistycznych języka Java: Spring Boot, Micronaut oraz Quarkus. Wybór padł na te szkielety, ze względu na obserwowany w ostatnim czasie wzrost zainteresowania dwoma ostatnimi technologiami, co można potwierdzić coraz większą ilością pojawiających się w sieci artykułów na ich temat, w których są one rekomendowane jako dobra alternatywa dla Spring Boota.

Tabela 1 zawiera wyniki popularności, dla trzech wybranych frameworków. Analiza została przeprowadzona przy pomocy trzech narzędzi: wyszukiwarki Google, serwisu społecznościowego dla programistów Stack Overflow oraz hostingowego serwisu internetowego przeznaczonego dla projektów programistycznych GitHub. W serwisach tych przeprowadzono wyszukiwanie wg słów kluczowych będących nazwami badanych szkieletów. Przedstawione w tabeli wyniki reprezentują odpowiednio liczbę artykułów znalezionych przy użyciu wyszukiwarki Google, liczbę istniejących wątków w serwisie StackOverflow oraz liczbę repozytoriów w systemie GitHub.

Tabela 1: Popularność szkieletów programistycznych języka Java

Szkielet	Rok wydania	Google	Stack Overflow	GitHub
Spring Boot	2004	17 700 000	107 000	249 383
Micronaut	2018	365 000	1 100	4 061
Quarkus	2019	958 000	1 700	8 049

Wyniki te pokazują, że najbardziej popularnym szkieletem jest Spring Boot. Framework ten deklasuje dwie pozostałe technologie. Z kolei szkielet Quarkus ma wyniki popularności około dwa razy wyższe niż Micronaut. Z dużą popularnością danego szkieletu wiąże się wielkość społeczności skupionej wokół technologii, co z kolei przekłada się na łatwiejsze rozwiązywanie potencjalnych problemów, szczególnie w przypadku kiedy oficjalna dokumentacja nie pomaga w ich rozwiązaniu.

4.1. Spring Boot

Spring Boot jest frameworkiem opracowanym z myślą o tworzeniu mikroserwisów [15]. Szkielet ten automatyzuje konfigurację, która w Spring Framework była kopiowana z aplikacji do aplikacji. Szybki start aplikacji sprawia, że framework ten jest wygodny do tworzenia mikroserwisów. W tym celu utworzono stronę start.spring.io [16], miejsce w którym można wygenerować szkielet projektu gotowy do implementacji oraz uruchomienia, z wszystkimi niezbędnymi zależnościami takimi jak komunikacja z bazą danych (Spring Data JPA), czy komunikacja poprzez HTTP przy pomocy architektury REST. Serwis ten pozwala także na wybór narzędzia budującego projekt (Maven lub Gradle) oraz na wybór wersji języka Java, którego będzie można używać w projekcie.

4.2. Micronaut

Micronaut, jest także szkieletem opartym na maszynie wirtualnej Javy, co pozwala na jego integrację z wszystkimi językami opartymi na tej platformie, takimi jak Java, Scala czy Kotlin. Podobnie jak Spring Boot, kładzie on nacisk na szybkie implementowanie aplikacji w architekturze mikroserwisowej. Natomiast w odróżnieniu od Spring Boot, twórcy Micronauta zwrócili szczególną uwagę na niskie zużycie pamięci [17]. Kolejną różnicą jest natywne wsparcie Micronauta dla nowopowstałej platformy GraalVM [18], co nie zostało jeszcze zaimplementowane w Spring Boot, a co ma wpływ na uzyskanie wysokiej wydajności i dostępności tworzonych aplikacji. Podobnie jak Spring Boot, Micronaut posiada mechanizm kontenera aplikacji, ale zarządzanie nim jest zaimplementowane w inny sposób. Spring Boot, korzysta z zalet refleksji języka Java, natomiast Micronaut wykorzystuje procesor adnotacji, co pozwala na szybsze uruchomienie aplikacji oraz wykrywanie błędów już na etapie kompilacji, zamiast w momencie uruchomienia aplikacji, jak to ma miejsce w przypadku szkieletu Spring Boot.

4.3. Quarkus

Quarkus to framework, który jest preferowany do natywnych aplikacji wykorzystujących narzędzie do konteneryzacji o nazwie Kubernetes [19]. Szkielet ten został utworzony ze względu na zwiększające się zapotrzebowanie na aplikacje oparte na mikroserwisach i na aplikacje chmurowe. Quarkus, inaczej niż Spring Boot oraz Micronaut, oferuje w pełni skonfigurowany projekt dostępny na głównej stronie serwisu [20] służący do szybkiego tworzenia szablonu aplikacji. W wygenerowanym projekcie szkielet posiada pliki konfiguracyjne, wśród których jeden o nazwie Dockerfile.jvm zawiera konfigurację aplikacji niezbędną do jej uruchomienia. Quarkus korzysta z narzędzi niedostępnych w Spring Boot i Micronaut, z czego mogą wyniknąć pewne trudności podczas implementacji. Zaletą tego szkieletu jest możliwość budowania lekkich aplikacji pod względem rozmiaru i wykorzystania pamięci [3], uproszczony kod dla typowych zastosowań i możliwość szybkiego przeładowania konfiguracji, dzięki czemu zmiany aplikowane są natychmiast po odświeżeniu uruchomionej aplikacji [21].

5. Metoda badań

5.1. Opis eksperymentu

Analizę porównawczą szkieletów programistycznych Spring Boot, Micronaut i Quarkus przeprowadzono na podstawie trzech scenariuszy badawczych, w ramach których wykonano:

1. pomiar czasów odpowiedzi na żądania POST, GET, PUT i DELETE pod różnym obciążeniem uzależnionymi od liczby wysyłanych żądań przez wirtualnych użytkowników,
2. pomiar niezawodności działania badanych aplikacji testowych,

3. pomiar objętości kodu aplikacji zaimplementowanych na bazie danego szkieletu, polegający na zliczeniu liczby linii kodu.

W pierwszej części eksperymentu zrealizowano scenariusze pierwszy i drugi. Badanie wydajności polegało na wysyłaniu żądań do serwera metodami POST, GET, PUT i DELETE z protokołu HTTP. Testy wydajnościowe i niezawodności przeprowadzono za pomocą biblioteki Gatling. Dokonano pomiarów czasów obsługi żądań, które następnie zostały uśrednione. Po przeprowadzeniu testów, Gatling wygenerował obszerny raport zawierający wyniki i statystyki w postaci atrakcyjnych wykresów i tabel.

Druga część eksperymentu dotyczyła trzeciego scenariusza, w którym dokonano pomiaru objętości kodu aplikacji. Czynność ta została w bardzo prosty sposób zrealizowana za pomocą wtyczki Statistics dostępnej w zintegrowanym środowisku programistycznym IntelliJ IDEA Ultimate 2020.3. Otrzymane wyniki po analizie kodu trzech aplikacji zaprezentowano na wykresie na rysunku 7.

5.2. Środowisko testowe

W tabeli 2 przedstawiono środowisko testowe, za pomocą którego przeprowadzono badania. Oprócz parametrów komputera, znajdują się tutaj wersje analizowanych szkieletów programistycznych oraz wersje narzędzi użytych do realizacji testów.

Tabela 2: Środowisko testowe

Sprzęt	
Procesor	Intel® Core™ i5-9300H
Pamięć RAM	16GB
Karta sieciowa	Killer E2500 Gigabit Ethernet Controller
System operacyjny	Windows 10 Pro
Oprogramowanie	
Spring Boot	2.4.4
Micronaut	2.5.4
Quarkus	1.13.4.Final
Gatling	3.5.1
Statistic	4.1.7

5.3. Pomiar czasów obsługi żądań POST

W pierwszym scenariuszu badawczym, czas odpowiedzi serwera na żądanie był mierzony od momentu jego wysłania przez aplikację kliencką, do prawidłowej i zakończonej powodzeniem odpowiedzi serwera, sygnalizowanej zwróceniem statusu HTTP 200. Dla każdej aplikacji utworzonej za pomocą poszczególnych szkieletów pięć razy powtórzono scenariusz dla różnych obciążeń tzn. dla 500, 1000, 2000, 5000 oraz 10000 użytkowników, którzy w ciągu minuty trwania testu wysyłali po 60 żądań, czyli średnio jedno żądanie na sekundę.

Obiektami badań były trzy aplikacje testowe, z których każda została zbudowana za pomocą innego szkieletu programistycznego języka Java. Aplikacje te realizują cztery podstawowe operacje na bazie danych.

Kontrolery trzech aplikacji testowych korzystają ze wspólnej klasy projektu `UserFolderRepository`, której metodę o nazwie `save` zaprezentowano na listingu 1.

Listing 1: Kod wspólnego repozytorium dla każdej aplikacji testowej

```
@RequiredArgsConstructor
public abstract class UserFolderRepository {
   private final DSLContext dslContext;

   protected void save(CreateUserFolderRequest request) {
      final var address = request.getAddress();
      dslContext.insertInto(USER_FOLDER)
            .columns(USER_FOLDER.LOGIN,
                  USER_FOLDER.NAME,
                  USER_FOLDER.SECOND_NAME,
                  USER_FOLDER.SURNAME,
                  USER_FOLDER.EMAIL,
                  USER_FOLDER.AGE,
                  USER_FOLDER.ADDRESS_LINE1,
                  USER_FOLDER.ADDRESS_LINE2,
                  USER_FOLDER.CITY,
                  USER_FOLDER.COUNTRY_SUBDIVISION,
                  USER_FOLDER.POSTAL_CODE,
                  USER_FOLDER.COUNTRY
                  )
            .values(request.getLogin(),
                  request.getName(),
                  request.getSecondName(),
                  request.getSurname(),
                  request.getEmailAddress(),
                  request.getAge(),
                  address.getLine1(),
                  address.getLine2(),
                  address.getCity(),
                  address.getCountrySubdivision(),
                  address.getPostalCode(),
                  address.getCountry())
      .execute();
   }
}
```

W pomiarach czasów, kluczową rolę odegrała biblioteka Gatling, za pomocą której utworzono oddzielną aplikację kliencką, w ramach której utworzono i wykonano po 100 symulacji dla każdej aplikacji testowej (po pięć powtórzeń dla czterech typów operacji CRUD i dla pięciu rodzajów obciążenia).

Listing 2 pokazuje fragment kodu zawierający zmienną httpProtocol, za pomocą której komponowane jest żądanie POST, zawierające dane klienta przesyłane do bazy.

Listing 2: Zapis scenariusza badawczego Gatlingu

```
val httpProtocol: HttpProtocolBuilder = http
   .baseUrl("http://localhost:8082")
   .header("Accept", "application/json")
   .header("Content-Type", "application/json")

setUp(
    scenario("Test Spring Boot app performance for 1000 users")
    .exec(
      http("Create User")
        .post("/spring-boot/users")
        .body(StringBody("{\n  \"login\": \"qwerty\",\n  \"name\": \"Jan\",
          \n  \"age\": \"18\",\n  \"secondName\": \"Tadeusz\",
          \n..\"surname\": \"Kowalski\",
          \n  \"emailAddress\": \"jan.kowalski@email.com\",
          \n  \"address\": {\n  \"line1\": \"ul. Debowa 32\",
          \n  \"line2\": \"Lublin, 21211\",\n  \"city\": \"Lublin\",
          \n  \"countrySubdivision\": \"Lubelskie\",
          \n  \"postalCode\": \"21000\",\n  \"country\": \"Polska\"\n  }\n}"))
    )
    .inject(constantUsersPerSec(1000) during 1.minutes)
    .protocols(httpProtocol),
)
```

Metoda `setUp` służy do konfiguracji symulacji. Definicja scenariusza rozpoczyna się od słowa `scenario`, której parametrem jest jego nazwa, natomiast parametrem części `http` jest nazwa danego żądania. Słowo `post` określa typ metody, która zostanie wykonana, z parametrem który zostanie połączony z właściwością `baseUri` zmiennej `httpProtocol`. Część `body` definiuje ciało żądania, a metoda `inject` umożliwia ustawienie liczby wirtualnych użytkowników - która może być stała lub zmienna oraz pozwala na określenie czasu trwania testu.

5.4. Pomiar niezawodności aplikacji

Wyzwaniem dla każdej aplikacji serwerowej jest osiągnięcie jak najwyższej niezawodności określanej jako najmniejsza liczba żądań zakończonych niepowodzeniem czyli takich, w przypadku których serwer zwracał status inny niż HTTP 200. Niezawodność jest wyrażana procentowo. Pomiar tego wskaźnika został przeprowadzony jednocześnie podczas badań wydajności w scenariuszu 1.

5.5. Pomiar objętości kodu

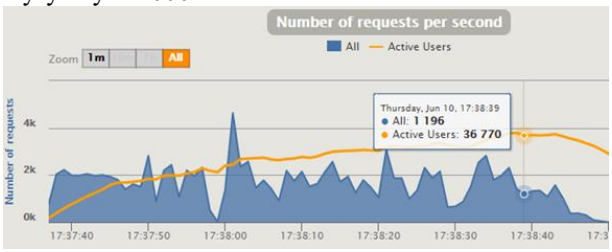
W scenariuszu 3 przeprowadzono oddzielnie pomiar liczby linii kodu trzech porównywanych aplikacji testowych, które realizowały te same zadania. Aplikacje były punktem końcowym serwera służącym do zapisu danych o użytkownikach w bazie wraz z ich adresem. Części wspólne kodu takie jak klasa repozytorium i klasa odwzorowująca ciało żądania zostały pominięte. Wiersze zawierające białe znaki, takie jak znak nowej linii czy nawiasy klamrowe, jeśli występowały samodzielnie w danej linii, także nie były zliczane.

6. Wyniki badań

6.1. Pomiar czasów odpowiedzi serwera na żądania POST

Na Rysunkach 1 i 2 pokazano fragment raportu w postaci wykresu i tabeli generowanego przez narzędzie Gatling, po wykonaniu pojedynczego badania wydajności przeprowadzonego według scenariusza 1 dla aplikacji zbudowanej za pomocą Spring Boota pod obciążeniem 2000 żądań na sekundę. Takie badania zostały powtórzone 15-krotnie (dla trzech aplikacji i pięciu rodzajów obciążeń).

Wykres z Rysunku 1 pokazuje, przypadek kiedy średnio było wysyłanych 2000 żądań przez aktywnych użytkowników, czyli po jednym żądaniu na sekundę. Jak widać, zdarzały się takie sytuacje, kiedy przez kilka sekund użytkownicy nie wysłali żadnego żądania, a w kolejnej chwili wysłali ich jednocześnie ponad 2000. Oznacza to, że w ciągu 60 sekund, w czasie trwania testu, 2000 użytkowników wysyłało po 2000*60 żądań. Liczbę wysyłanych żądań w ciągu całego testu obrazuje na rysunku 1 niebieska krzywa. Przyglądając się jej przebiegowi, widać, że czasem liczba wysyłanych żądań spadała prawie do zera, a momentami przekraczała 4000. Z kolei pomarańczowa linia oznacza liczbę aktywnych użytkowników, wśród których w danej chwili tylko część wysyła żądania. Liczba wysyłanych żądań zmieniała się w trakcie trwania testu (60 sekund), ale średnio w ciągu każdej sekundy było ich wysyłanych 2000.



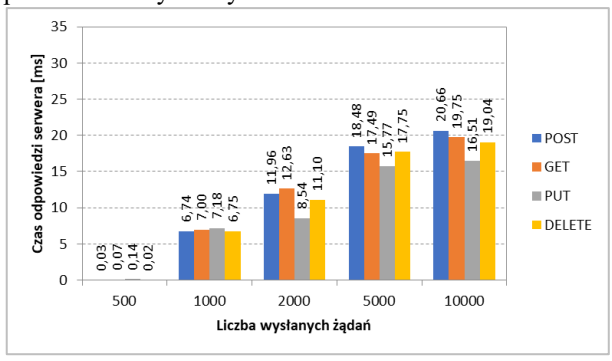
Rysunek 1: Fragment raportu otrzymany z narzędzia Gatling przy obciążaniu 2000 żądań/s dla aplikacji wykonanej za pomocą Spring Boot.

Na Rysunku 2 widoczny jest inny fragment raportu w postaci tabeli z wynikami testów. Znajdują się tutaj informacje o całkowitej liczbie wysłanych żądań, liczbie żądań poprawnie obsłużonych oraz tych, których obsługa zakończyła się niepowodzeniem. Ponadto tabela zawiera szereg danych statystycznych dotyczących czasów obsługi żądań takich jak średnia, mediana, czas minimalny i maksymalny oraz percentyle. Ostatnia metryka wskazuje wartości, poniżej których znajduje się pewien procent danych w zbiorze danych. Na przykład 95-ty percentyl to wartość, dla której 95% przypadków było lepszych tzn. odpowiedź z systemu trwała krócej, a 5% było gorszych czyli trwały dłużej.

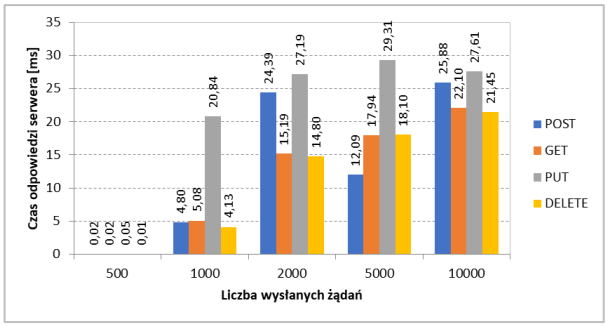
Requests	Executions					Response Time (ms)					
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max
Global Information	60000	36776	23224	39%	600	48	10335	16713	31004	33500	38317
Create User	60000	36776	23224	39%	600	48	10343	16717	31006	33500	38317

Rysunek 2: Fragment tabeli z raportu w postaci tabeli wygenerowany przez narzędzie Gatling.

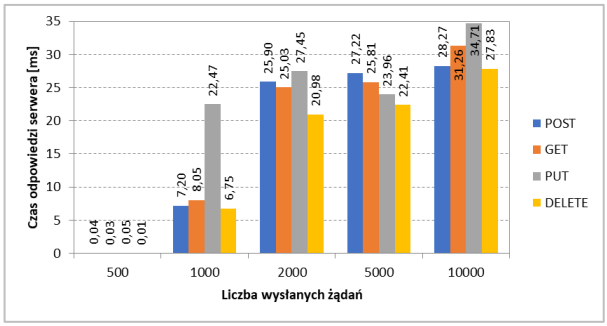
Na Rysunkach 3, 4 i 5 przedstawiono zagregowane wyniki dla scenariusza 1, w ramach którego przeprowadzono testy punktu końcowego podczas którego wykorzystano metody POST, GET, PUT i DELETE przy różnych obciążenia, począwszy od 500, następnie 1000, 2000, 5000 oraz 10000 żądań na sekundę wysyłanych przez wirtualnych użytkowników.



Rysunek 3: Średnie czasy odpowiedzi serwera na cztery typy żądań przy ustalonym obciążeniu dla aplikacji testowej opartej na szkielecie Spring Boot.



Rysunek 4: Średnie czasy odpowiedzi serwera na cztery typy żądań przy ustalonym obciążeniu dla aplikacji testowej opartej na szkielecie Micronaut.



Rysunek 5: Średnie czasy odpowiedzi serwera na cztery typy żądań przy ustalonym obciążeniu dla aplikacji testowej opartej na szkielecie Quarkus.

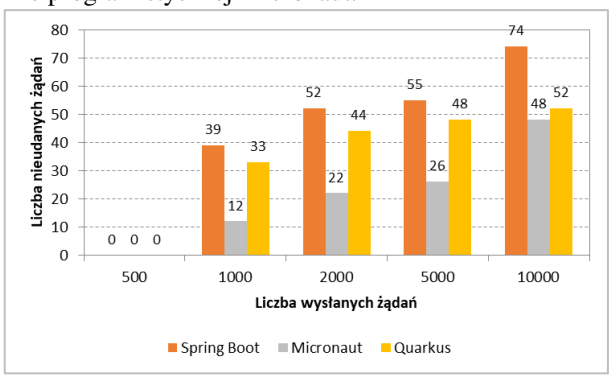
Powyższe rysunki pokazują średnie czasy obsługi poszczególnych rodzajów żądań przy różnych wielkościach obciążenia.

Z wykresów tych widać, że przy niskim obciążeniu (500 żądań/s), choć różnice w wydajności aplikacji opartych na poszczególnych szkieletach są bardzo małe, najlepszy okazał się szkielet Micronaut. Dla obciążenia 1000 żądań/s w przypadku operacji POST, GET i DELETE także najlepsze wyniki osiągnął Micronaut. Wyjątkiem jest tu operacja aktualizująca dane (żądanie PUT), która wymagała znacznie dłuższego czasu obsługi w aplikacjach opartych na szkieletach Micronaut i Quarkus. Przyczyną tego stanu rzeczy może być to, że w sytuacji gdy odsetek nieudanych żądań jest niewielki, żądania te ustawiane są ponownie w kolejce i w ten sposób dodatkowo obciążają serwer.

Dla wyższych obciążeń tzn. 2000, 5000 i 10000 żądań/s najlepsze czasy obsługi czterech rodzajów żądań osiągnął szkielet Spring Boot. W przypadku obciążenia 5000 żądań/s dobre wyniki osiągnął także Micronaut, choć były one o 11% gorsze od wyników szkieletu Spring Boot.

6.2. Pomiar niezawodności aplikacji

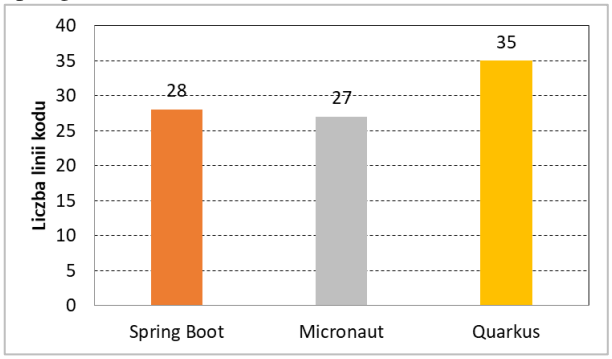
Na Rysunku 6 zaprezentowano niezawodność wyrażoną jako liczba nieobsłużonych żądań. Wykres pokazuje, że dla obciążenia 500 żądań/s trzy aplikacje testowe obsługiwały wszystkie żądania. Dla obciążeń 1000, 2000, 5000 i 1000 żądań/s liczba tych nieobsłużonych była największa w przypadku aplikacji utworzonej na bazie Spring Boot. Trochę lepsze wyniki uzyskała aplikacja wykonana za pomocą szkieletu Quarkus. Natomiast najlepszy wynik, oznaczający najmniejszą liczbę nieobsłużonych żądań, osiągnęła aplikacja oparta na platformie programistycznej Micronaut.



Rysunek 6: Wyniki niezawodności testowanych aplikacji.

6.3. Pomiar objętości kodu

Liczba linii kodu wymaganego do utworzenia aplikacji była brana pod uwagę przy porównaniach badanych szkieletów. Z Rysunku 7 wynika, że najmniejszą liczbę linii kodu niezbędną do osiągnięcia tego samego efektu miała aplikacja utworzona za pomocą szkieletu Micronaut, a najwięcej aplikacja oparta na szkielecie Quarkus. Wynika to m.in. z tego, że w Micronaut pousuwano zbędne według jego twórców adnotacje, które nadal znajdują się w szkielecie Spring Boot. Jako przykład można tu podać adnotację @RequestMapping, którą włączono w Micronaut bezpośrednio do adnotacji @Controller. Poza tym w Micronaut nie ma konieczności użycia adnotacji do wykonania auto-konfiguracji, ponieważ szkielet rozwiązuje to samoistnie - domyślnie ustawiając odpowiednią flagę. Natomiast szkielet Quarkus, jest najbardziej konfigurowalny, stąd potrzebne było dodanie adnotacji określających typ danych przyjmowanych oraz zwracanych - w tym przypadku JSON. Robi się to za pomocą pary adnotacji @Produces oraz @Consumes. Dodatkowo w aplikacji zbudowanej na frameworku Quarkus konieczne było utworzenie klasy implementującej abstrakcyjne repozytorium, co czyniło jej implementację trudniejszą od aplikacji opartej na Spring Boot i Micronaut.



Rysunek 7: Liczba linii kodu dla każdego szkieletu.

7. Wnioski

Decyzja dotycząca wyboru optymalnej technologii dostosowanej do rozpoczynanego przedsięwzięcia programistycznego jest bardzo istotna, ponieważ niesie ze

sobą trudne do przewidzenia konsekwencje w przyszłości. Niniejsza praca, skupiająca się na wydajności i niezawodności trzech wybranych platform języka Java, miała na celu ułatwienie podjęcia tej decyzji. W analizie porównawczej szkieletów Spring Boot, Micronaut i Quarkus wzięto pod uwagę cztery kryteria: popularność, wydajność, niezawodność działania oraz objętość kodu.

Biorąc pod uwagę kryterium popularności, framework Spring Boot okazał się liderem według trzech użytych do analiz narzędzi. Popularność tego szkieletu wiąże się przede wszystkim z jego długim istnieniem na rynku trwającym nieprzerwanie od 2004 roku. Pozostałe dwa szkielety: Quarkus i Micronaut powstały stosunkowo niedawno – odpowiednio 2 i 3 lata temu.

Porównując otrzymane wyniki wydajności, można stwierdzić, że szkielet Micronaut osiągnął najlepsze rezultaty w niskich warunkach obciążeniowych. Przy większych obciążeniach najlepiej wypadł Spring Boot, a najgorsze wyniki osiągnął Quarkus.

Pod względem niezawodności jednoznacznie najlepszą platformą programistyczną okazał się Micronaut. W każdych warunkach obciążeniowych aplikacja zbudowana na bazie tego szkieletu była bardziej niezawodna niż aplikacje zbudowane na szkieletach Spring Boot i Quarkus. Biorąc pod uwagę wydajność, Micronaut był lepszy dla mniejszych obciążeń, natomiast dla większych przegrywał z szkieletem Spring Boot. W ostatniej kategorii dotyczącej objętości kodu również Micronaut okazał się najlepszym wyborem.

Z przeprowadzonej analizy można wysnuć generalny wniosek, że najlepszym z trzech badanych szkieletów programistycznych okazał się Micronaut. Dokonana analiza ma jednak swoje ograniczenia, ponieważ dotyczyła bardzo prostej aplikacji testowej, która wykonywała cztery proste czynności – obsługę żądań zapisu, odczytu, aktualizacji i usunięcia danych z bazy. W celu dokonania dokładniejszej oceny technologii należy poszerzyć spektrum badań i uwzględnić jeszcze inne aspekty.

Literatura

[1] E. K. Smyk, Overview of technologies and methods designed to build Java Enterprise web applications. Comparison of Spring and Play Frameworks based on proprietary application (praca magisterska), Politechnika Warszawska, 2014.

[2] P. Dutta, V. Gupta, S. Rana, Performance Comparison on Java Technologies – A Practical Approach, Centre for development of Advanced Computing, Third International Conference on Computer Science, Engineering & Applications (2013) 349-357, https://doi.org/10.5121/CSIT.2013.3536.

[3] M. Šipek, D. Muharemagić, B. Mihaljević, A. Radovan, Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus, 43rd International Convention on Information, Communication and Electronic Technology (MIPRO) (2020) 1746-1751, DOI: 10.23919/MIPRO48935.2020.9245290.

[4] H. K. Dhalla, Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core, Journal of Physis: Conference Series 1933 (2021), https://doi.org/10.1088/1742-6596/1933/1/012041.

[5] M. Pucek, M. Błaszczyk, P. Kopniak, Porównanie lekkich szkieletów dla języka Java poprzez analizę autorskich aplikacji internetowych, Journal of Computer Sciences Institute 19 (2021) 159-164.

[6] TIOBE Index, https://www.tiobe.com/tiobe-index/, [02.07.2021].

[7] Oracle Java SE Support Roadmap, https://www.oracle.com/java/technologies/java-se-support-roadmap.html, [06.07.2021].

[8] M. Masse, REST API Design Rulebook, O'Reilly Media, 2012.

[9] B. Miłosierny, M. Dzieńkowski, Analiza porównawcza szkieletów do budowy aplikacji internetowych w ekosystemie Node.js, Journal of Computer Sciences Institute 18 (2021) 42-48.

[10] M. Herber, Gatling. Testy wydajnościowe w innej formie, https://testerzy.pl/baza-wiedzy/gatling-testy-wydajnosci-w-innej-formie-czesc-1, [02.07.2021].

[11] Xie A., Performance Testing Tutorial: Automation, Gatling, and Jenkins, https://www.educative.io/blog/performance-testing-tutorial-gatling-jenkins, [21.07.2021].

[12] Lee G., Gatling Load Testing: How-To, Distributed Tests & Examples, https://www.loadview-testing.com/blog/gatling-load-testing-how-to-distributed-tests-examples/, [21.07.2021].

[13] A. Ludwikowski, Gatling vs JMeter – czego użyć do testowania wydajności, https://softwaremill.com/gatling-vs-jmeter-testy-wydajnosci/, [02.07.2021].

[14] Gatling, https://gatling.io/docs/gatling/reference/current/general/concepts/, [21.07.2021].

[15] B. Nius, Jak Spring Boot ułatwia tworzenie aplikacji w Javie? https://global4net.com/ecommerce/jak-spring-boot-ulatwia-tworzenie-aplikacji-w-javie/, [05.12.2019].

[16] Spring Initializr, https://start.spring.io/, [02.07.2021].

[17] P. Bykowski, Micronaut – framework dedykowany dla mikroserwisów, https://bykowski.pl/micronaut-framework-dedykowany-dla-mikroserwisow/, [17.10.2019].

[18] Micronaut, https://micronaut.io/, [03.07.2021].

[19] Quarkus – Supersonic Subatomic Java, https://quarkus.io/, [05.07.2021].

[20] Quarkus – start coding with code.quarkus.io, https://code.quarkus.io/, [06.07.2021].

[21] Oficjalna dokumentacja szkieletu Quarkus, https://quarkus.io/, [28.02.2021].