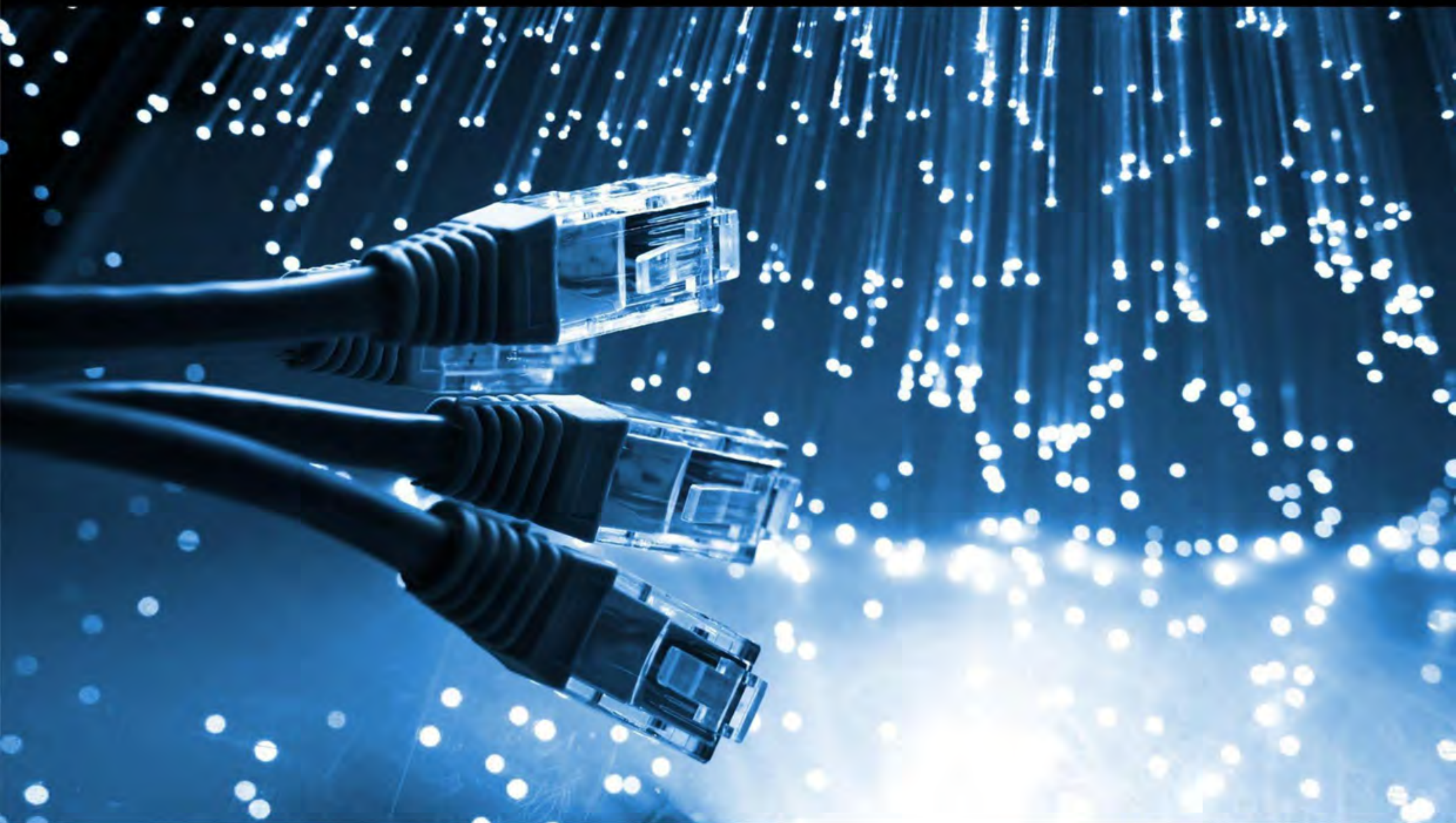
JCSI

Journal of Computer Sciences Institute

Volume 16/2020



Department of Computer Science
Lublin University of Technology

jcsi.pollub.pl

ISSN: 2544-0764

Redakcja JCSI
e-mail: jcsi@pollub.pl
www: jcsi.pollub.pl
Katedra Informatyki
Wydział Elektrotechniki i Informatyki

Politechnika Lubelska
ul. Nadbystrzycka 36 b
20-618 Lublin

JCSI Editorial
e-mail: jcsi@pollub.pl
www: jcsi.pollub.pl
Department of Computer Science
Faculty of Electrical Engineering and Computer Science
Lublin University of Technology
ul. Nadbystrzycka 36 b
20-618 Lublin, Poland

Redaktor naczelny:
Tomasz Zientarski
e-mail: t.zientarski@pollub.pl

Editor in Chief:
Tomasz Zientarski
e-mail: t.zientarski@pollub.pl

Redaktor techniczny:
Beata Pańczyk,
e-mail: b.panczyk@pollub.pl

Assistant editor:
Beata Pańczyk,
e-mail: b.panczyk@pollub.pl

Recenzenci numeru:

dr inż. Grzegorz Kozieł
dr inż. Jakub Smołka
dr inż. Maciej Pańczyk
dr inż. Piotr Kopniak
dr Mariusz Dzieńkowski
dr inż. Elżbieta Miłosz
dr inż. Jacek Kęsik
dr inż. Marek Miłosz, prof. PL
dr inż. Marcin Badurowicz
dr inż. Maria Skublewska-Paszkowska
dr Edyta Łukasik
dr inż. Kamil Żyła

Reviewers:

Grzegorz Kozieł
Jakub Smołka
Maciej Pańczyk
Piotr Kopniak
Mariusz Dzieńkowski
Elżbieta Miłosz
Jacek Kęsik
Marek Miłosz
Marcin Badurowicz
Maria Skublewska-Paszkowska
Edyta Łukasik
Kamil Żyła

Skład komputerowy:
Monika Kaczorowska
e-mail: m.kaczorowska@pollub.pl

Computer typesetting:
Monika Kaczorowska
e-mail: m.kaczorowska@pollub.pl

Projekt okładki:
Marta Zbańska

Cover design:
Marta Zbańska

ISSN 2544-0764

Spis treści

1. **Wydajność kodu Java i Kotlin w wybranych szkieletach aplikacji internetowych**
Grzegorz Bujnowski, Jakub Smołka......219-226
2. **Blender jako narzędzie do generacji danych syntetycznych**
Rafał Sieczka, Maciej Pańczyk......227-232
3. **Analiza wybranych metod tworzenia sztucznej inteligencji na przykładzie popularnej gry w karty**
Łukasz Gałka, Mariusz Dzieńkowski......233-240
4. **Analiza porównawcza współprogramów języka Kotlin z językami Java i Scala w przetwarzaniu równoległym**
Adrian Zieliński......241-246
5. **Badanie wydajności elementów bibliotek STL i Qt w przetwarzaniu wielowątkowym**
Piotr Krasowski, Jakub Smołka......247-251
6. **Analiza bezpieczeństwa mechanizmów uwierzytelniania oraz autoryzacji implementowanych w aplikacjach internetowych zbudowanych w oparciu o architekturę REST**
Tomasz Muszyński, Grzegorz Koziel......252-260
7. **Badanie bezpieczeństwa wybranych platform CMS za pomocą skanerów podatności**
Patryk Zamościński, Grzegorz Kozieł......261-268
8. **Analiza działania szkieletu Blazor w trybie klienta z hostingiem**
Karol Kozak, Jakub Smołka......269-273
9. **Wdrożenie rozwiązań do zarządzania zespołem rozproszonym w firmach sektora IT**
Mykhailo Kuzyk, Elżbieta Miłosz......274-278
10. **Porównanie wydajności baz danych MySQL, MSSQL, PostgreSQL oraz Oracle z uwzględnieniem wirtualizacji**
Wojciech Truskowski, Rafał Klewek, Maria Skublewska-Paszkowska......279-284
11. **Analiza porównawcza wybranych systemów mapowania obiektoworelacyjnego dla platformy .NET**
Krzysztof Drzazga, Marcin Bobel, Maria Skublewska-Paszkowska......285-292
12. **Analiza zastosowania języków Java oraz C# do budowy aplikacji mobilnej na platformę Android**
Michał Jankowski, Maria Skublewska-Paszkowska......293-299
13. **Porównanie wydajności wybranych parserów JSON z parserem używającym innej metody odczytu**
Przemysław Koter......300-304
14. **Porównanie Objective-C oraz Swift na przykładzie gry mobilnej**
Karolina Banach, Maria Skublewska-Paszkowska......305-308
15. **Porównanie wydajności technologii webowych REST i GraphQL**
Mateusz Mikuła, Mariusz Dzieńkowski......309-316
16. **Ocena zanieczyszczenia powietrza na podstawie zdjęcia wiązki laserA**
Anna Pawelec, Rafał Maksim, Maria Skublewska-Paszkowska......317-325

Contents

1. **JAVA AND KOTLIN CODE PERFORMANCE IN SELECTED WEB FRAMEWORKS**
GRZEGORZ BUJNOWSKI, JAKUB SMOŁKA........219-226
2. **BLENDER AS A TOOL FOR GENERATING SYNTHETIC DATA**
RAFAŁ SIECZKA, MACIEJ PAŃCZYK........227-232
3. **ANALYSIS OF SELECTED METHODS OF CREATING ARTIFICIAL INTELLIGENCE ON THE EXAMPLE OF A POPULAR CARD GAME**
ŁUKASZ GAŁKA, MARIUSZ DZIEŃKOWSKI........233-240
4. **COMPARATIVE ANALYSIS OF KOTLIN COROUTINES WITH JAVA AND SCALA IN PARALLEL PROGRAMMING**
ADRIAN ZIELIŃSKI........241-246
5. **PERFORMANCE TESTING OF STL AND QT LIBRARY ELEMENTS IN MULTI-THREADED PROCESSING**
PIOTR KRASOWSKI, JAKUB SMOŁKA........247-251
6. **A SECURITY ANALYSIS OF AUTHENTICATION AND AUTHORIZATION IMPLEMENTED IN WEB APPLICATIONS BASED ON THE REST ARCHITECTURE**
TOMASZ MUSZYŃSKI, GRZEGORZ KOZIEL........252-260
7. **ANALYSIS OF SECURITY CMS PLATFORMS BY VULNERABILITY SCANNERS**
PATRYK ZAMOŚCIŃSKI, GRZEGORZ KOZIEŁ........261-268
8. **ANALYSIS OF THE BLAZOR FRAMEWORK IN CLIENT-HOSTED MODE**
KAROL KOZAK, JAKUB SMOŁKA........269-273
9. **IMPLEMENTATION OF SOLUTIONS FOR DISTRIBUTED TEAM MANAGEMENT IN IT SECTOR COMPANIES**
MYKHAILO KUZYK, ELŻBIETA MIŁOSZ........274-278
10. **COMPARISON OF MYSQL, MSSQL, POSTGRESQL, ORACLE DATABASES PERFORMANCE, INCLUDING VIRTUALIZATION**
WOJCIECH TRUSKOWSKI, RAFAŁ KLEWEK, MARIA SKUBLEWSKA-PASZKOWSKA........279-284
11. **COMPARATIVE ANALYSIS OF SELECTED OBJECT-RELATIONAL MAPPING SYSTEMS FOR THE .NET PLATFORM**
KRZYSZTOF DRZAZGA, MARCIN BOBEL, MARIA SKUBLEWSKA-PASZKOWSKA........285-292
12. **ANALYSIS OF THE USE OF JAVA AND C# LANGUAGES FOR BUILDING A MOBILE APPLICATION FOR THE ANDROID PLATFORM.**
MICHAŁ JANKOWSKI, MARIA SKUBLEWSKA-PASZKOWSKA........293-299
13. **PERFORMANCE COMPARISON OF CHOSEN JSON PARSERS AND A PARSER THAT EMPLOYS A DIFFERENT READING METHOD**
PRZEMYSŁAW KOTER........300-304
14. **COMPARISON OF OBJECTIVE-C AND SWIFT ON THE EXAMPLE OF A MOBILE GAME**
KAROLINA BANACH, MARIA SKUBLEWSKA-PASZKOWSKA........305-308
15. **COMPARISON OF REST AND GRAPHQL WEB TECHNOLOGY PERFORMANCE**
MATEUSZ MIKUŁA, MARIUSZ DZIEŃKOWSKI........309-316
16. **THE ANALYSIS OF AIR POLLUTION BASED ON LASER BEAM PHOTO**
ANNA PAWELEC, RAFAŁ MAKSIM, MARIA SKUBLEWSKA-PASZKOWSKA........317-325

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 219–226
Received: 25 June 2020
Accepted: 15 July 2020

Java and Kotlin code performance in selected web frameworks

Wydajność kodu Java i Kotlin w wybranych szkieletach aplikacji internetowych

Grzegorz Bujnowski*, Jakub Smołka

Department of Computer Science, Lublin University of Technology, ul. Nadbystrzycka 38, 20-618 Lublin, Poland

Abstract

This paper discusses the issue of comparing Java and Kotlin technologies based on the web application framework. The criteria taken into account for testing purposes are: execution time, memory usage, CPU load, database response in set time. Series of tests and their in-depth comparative analysis are carried out. For this case, tests and code analysis were carried out to draw comparative conclusions. The performance in terms of web frameworks, database response speed and tests implementation in different languages - in all these Kotlin proved to be less efficient. There is no significant difference for CPU load. Between individual measurements, the difference does not exceed 2%. Implementation in the Kotlin language has never achieved the best result in any group of measurements.

Keywords: kotlin; jvm; java; benchmark

Streszczenie

W tym artykule omówiono kwestię porówniania technologii Java i Kotlin w oparciu o szkielet aplikacji internetowych. Kryteria brane pod uwagę dla celów testowych to: czas wykonania, wykorzystanie pamięci, obciążenie procesora, liczba odpowiedzi z bazy danych w zadanym czasie. Przeprowadzana jest seria testów i ich dogłębna analiza porównawcza. Przeprowadzono testy i analizę kodu. Wydajność pod względem szkieletów aplikacji internetowych, szybkości odpowiedzi bazy danych i szybkości działania testów - we wszystkich Kotlin okazał się mniej wydajny. Nie ma znaczącej różnicy dla obciążenia procesora. Pomiędzy poszczególnymi pomiarami, różnica nie przekracza 2%. Implementacja w języku Kotlin nigdy nie osiągnęła najlepszego wyniku w żadnej grupie pomiarów.

Słowa kluczowe: kotlin; wirtualna maszyna javy; testy wydajnościowe

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

1. Wstęp

Poniższa praca porównuje technologię Java oraz Kotlin na podstawie szkieletów aplikacji internetowych. Uwzględnione w badaniach kryteria to: wydajność rozwiązywania problemów, zużycie pamięci oraz obciążenie procesora, szybkość działania na bazach danych.

Spośród ogromnej liczby powstałych języków programowania, Java wyróżniła się wyraźnie, stając się przez kilka dziesięcioleci jedną z preferowanych technologii przez programistów. Po ponad 25 latach od powstania Javy, w tle pojawił się nowy konkurent - Kotlin. Nowy oficjalny język programowania promowany przez firmę Google od 2017 roku. Jest kolejnym z wielu języków korzystających z Wirtualnej Maszyny Java. Według indeksu TIOBE [10] w marcu 2020 roku technologia ta zajmowała 30 miejsce, w rankingu popularności.

Od oficjalnego wprowadzenia ćwierć wieku temu obiektowego języka programowania, nadal Java jest najpopularniejszym językiem programowania w 2020 roku. Jest to język programowania początkowo opracowany przez Sun Microsystems, a obecnie będący własnością Oracle Corporation. Jednym z głównych powodów sukcesu Javy jest niezależność platformy – kod jest kompilowany do kodu bajtowego a program wykonywany za pośrednictwem wirtualnej maszyny Java (JVM). Dla wirtualnej maszyny powstało wiele alternatyw dla języka Java, takich jak Groovy, Clojure, Scala i Kotlin. Społeczność programistyczna zwraca szczególną uwagę na Kotlin, ze względu na wsparcie gigantów technologicznych. Kotlin stał się językiem traktowanym priorytetowo przez dominujące na rynku technologicznym.

Zaletą języka Kotlin i Java jest to, że mogą ze sobą współistnieć w zakresie jednego projektu. Kotlin jest kompatybilny z Javą, co oznacza że istnieje możliwość posiadania kodu Kotlin i Java w tym samym projekcie. W związku z tym, że Kotlin jest najszybciej rosnącym w popularności językiem na platformie programistycznej GitHub, jak i to że firma Google oficjalnie zaleca stosowanie tej technologii dla rozwijania aplikacji na telefony Android.

Artykuł próbuje odpowiedzieć na pytania:

1. Czy istnieje różnica w wydajności Java i Kotlin na

*Corresponding author
Email address: gbujnow@gmail.com (G. Bujnowski)

przykładzie szkieletu aplikacji internetowej?

2. W jaki sposób można dokonać porównania wydajności Java i Kotlin na szkielecie aplikacji internetowej?

2. Przegląd publikacji naukowych

Artykuł pod tytułem: "How and Why did developers migrate applications from Java to Kotlin? A study based on code analysis and interviews with developers" (Universite Polytechnique Hauts-de-France) opisuje trzy przypadki:

- gdy programiści zaczynają pisać aplikacje od początku używając języka Kotlin,
- rozwijają kod napisany w Java za pomocą Kotlin,
- migrują całkowicie aplikacje z technologii Java do Kotlin [1].

Za pomocą analizy statycznej kodu, pod obserwację wzięto historię rozwijania projektów. Badanie opiera się na studium 374 aplikacji internetowych i 78 wywiadach z programistami odpowiadającymi za użycie nowego języka programowania. Praca rozpoznaje różnicę pomiędzy dwoma językami programowania. Porównuje języki programowania Java i Kotlin. Praca wskazuje, lepszą jakość kodu w projektach napisanych w języku Kotlin. Podane przez programistów powody do zadowolenia z języka Kotlin to: uniknięcie wyjątków typu null pointer (stos błędu pokazywał wyjątek null w 51,96% projektach opisujących ponad 600 projektów w Javie), Kotlin pozawala wyeliminować ten typ problemów z perspektywy kompilatora.

Ankietowani programiści zdecydowali się na migrację z Java, w związku z bardziej defensywnymi technikami programowania. Badania opinii społecznej pokazały, że zmiany w wyborze technologii dotyczących nowych projektów, wynika głównie z wzrastającego trendu narzuconego przez politykę firmy Google. Trend ten wzmacniany jest również badaniami, które pokazują mniejszą złożoność językową programów napisanych w Kotlin.

Drugi artykuł to „Comparative study Java vs Kotlin", praca strukturalnie i semantycznie porównuje oba języki programowania, określa zalety i wady każdego z nich [2]. Artykuł teoretyczny omawiający podstawy składni i zastosowania. Zwraca uwagę na skróceniu składni
i opisuje nowe funkcjonalności.

3. Cel i przedmiot badań

Celem badania jest porównanie języków Java oraz Kotlin na przykładzie szkieletu aplikacji internetowej.
W artykule, dogłębnej analizie zostanie poddana szybkość rozwiązywania problemów, jak i zużycie pamięci oraz obciążenie procesora, szybkość działania na bazie danych. Do testów zostaną użyte algorytmy porównujące wydajność pomiędzy różnymi językami programowania.

Zostaną stworzone aplikacje internetowe, rozwiązujące typowe problemy w obu językach. Celem będzie porównanie szybkości obsługiwania bazy danych i rozwiązywania problemów. Implementacja algorytmów testowych w Java i Kotlin ma zbadać różnice jakościowe i ilościowe pomiędzy szkieletami aplikacji internetowych (stworzonych w obu językach). Omówiona zostanie wydajność rozwiązań w technologii Java i Kotlin.
Zakres badania to:

- omówienie i porównanie wydajności tworzenia rozwiązań technologicznych na podstawie szkieletów aplikacji internetowych,
- implementacja i opis algorytmów testowych,
- analiza wydajności kodu wykonanych w obu językach programowania,
- testy porównawcze: szybkości wykonywania programów, zużycia pamięci, obciążenia procesora oraz wydajności odpowiedzi z bazy danych, wszystko w zależności od implementacji Kotlin lub Java.

4. Omówienie języków Java i Kotlin

4.1. Java

Projekt Java został zainicjowany w 1991 roku przez Jamesa Goslinga i jego współpracowników. Na początku język nazywał się „Oak" (dąb). Później nazwa projektu została zmieniona na „Java" na odniesienie do kawy Java. Z tego powodu logo tego języka to filiżanka kawy. Firma Sun Microsystems wydała Javę 1.0 w 1996 roku. Następnie nowe wersje wydawane były co 1-3 lata.
Od wersji Java 9 wydanej w 2017 r. Nowe wersje wydawane są co 6 miesięcy.

Java obsługuje wiele paradygmatów programowania, jest językiem imperatywnym opartym na koncepcji obiektowej: prawie każda część programu jest obiektem. Dlatego sam program można uznać za zestaw współdziałających obiektów. Ponadto częściowo obsługuje nowoczesne paradygmaty programowania, takie jak programowanie ogólne, programowanie współbieżne, programowanie funkcyjne i inne.

4.2. Kotlin

Kotlin to statycznie typowany język, w którym możemy pisać kod jak w językach dynamicznych bez utraty wydajności, jak reklamują twórcy - praca zbada ową wydajność [3]. Posiada takie właściciwości jak:

- wsparcie dla inicjalizacji obiektu, klasy mogą mieć dane, które mogą być zmienne (var) lub tylko do odczytu (val),
- względnie lekka biblioteka standardowa w porównaniu do Javy,
- niska złożoność technologii oraz podobieństwo do Javy, oznacza dużą łatwość w zdobyciu biegłości w wyżej wspomnianym języku,

- możliwość wywoływania kodu Java bezpośrednio z bazy kodu Kotlin,
- wsparcie dla typów zmiennych non-nullable, zabezpieczenie przed błędem null pointer exception (null safety support),
- ulepszona semantyka w stosunku do Javy,
- brak średników w kodzie.

Firma JetBrains w lipcu 2011 r. zaprezentowała projekt Kotlin, jako nowy język dla platformy JVM. Nazwa technologii pochodzi od wyspy Kotlin, niedaleko Sankt Petersburga w Rosji. Głównym celem tego projektu było zapewnienie bezpieczniejszej i bardziej zwięzłej alternatywy dla języka Java we wszystkich kontekstach, w których Java jest obecnie używana. W 2016 roku została wydana pierwsza oficjalna stabilna wersja (Kotlin v1.0). Społeczność programistów była już zainteresowana korzystaniem z tego języka, zwłaszcza na Androidzie, z powodu walki patentowej firm Google oraz Oracle (2019 rok).

Na konferencji Google I / O 2017 ogłosił wsparcie dla Kotlin na Androida. W tej chwili Kotlin jest uważany za język ogólnego zastosowania dla wielu platform, nie tylko dla Androida. Język ma kilka wydań rocznie. Ponieważ Kotlin jest stosunkowo nowoczesnym językiem, nie posiada starych interfejsów API i przestarzałych koncepcji. Jest nowym pomysłem, względem starych paradygmatów. Najważniejsze nowe funkcje to:

- funkcje rozszerzające (możliwość przyklejania statycznych metod do istniejącego typu)
- leniwe właściwości (lazy properties, odroczenie inicjalizacji obiektów, optymalizacja)
- wsparcie dla typów zmiennych non-nullable, zabezpieczenie przed błędem null pointer exception (null safety support, nie można wywołać metody na obiekcie null)

Podstawowe koncepcje które odróżniają język Kotlin od Javy to posiadanie:

- wyrażeń lambda z funkcjami inline,
- funkcje rozszerzające (extensions functions), możliwość rozszerzania klas bez dziedziczenia po innych klasach,
- zabezpieczenie przed wywołaniem referencji null,
- rozszerzony mechanizm rzutowania (smart cast – operator is, as, as?),
- szablony dla typu String, ewaluacja wyrażeń za pomocą operatora $,
- klasa, może posiadać własności (properties – val, var),
- konstruktor główny i konstruktory dodatkowe,
- przeładowanie operatorów (jak w C++)
- wbudowany typ DTO (obiekty z typem 'data')

5. Metoda prowadzenia badań

Badanie stara się odpowiedzień na pytanie: Czy istnieją znaczące różnice między wydajnością pomiędzy różnymi implementacjami tego samego testu porównawczego w Kotlin i Javie przy użyciu Wirtualnej Maszyny Java na przykładzie szkieletu aplikacji internetowej?

Jest wiele metod porównywania języków programowania. Jednym z najchętniej cytowanych badań nad różnymi językami programowania jest tzw. „The Computer Langugae Benchmark Game" (CLBG) [4]. Dynamiczne porównanie metryk będzie oparte na idei propagowanej przez Isaac Gouy. W 2000 roku rozpoczął się projekt CLBG, którego celem jest porównanie wszystkich głównych języków programowania, dzięki korzystaniu z obiektywnych testów porównawczych. Autorzy projektu zwracają szczególną uwagę na praktyczność wniosków z badań. Projekt skupia się na stworzeniu metryki, testu, który zależy od kontekstów takich jak:

1. cele badania,
2. konsekwencje możliwych błędów,
3. rodzaje ocenianego systemu komputerowego,
4. środowiska wykonawczego
5. infrastruktury i jakości testów porównawczych [5].

CLBG przedstawia zestaw 10 różnych problemów algorytmicznych. Wszystkie problemy zostały przedstawione i opisane na oficjalnej stronie internetowej [4]. Wprowadzone algorytmy mają ścisłe reguły dotyczące ich implementacji, z czego korzystają ich twórcy.

Biorąc pod uwagę wszystkie algorytmy, istnieją tylko trzy testy porównawcze CLBG, które zostały użyte w końcowym eksperymencie. Reszta testów porównawczych została odrzucona, aby zachować zgodność z założeniem: że kod Java musi być konwertowany na język Kotlin bez potrzeby wprowadzania dużych ilości zmian w kodzie. Dodatkowo, każdy użyty szkielet aplikacji internetowej zostanie przetestowany wydajnościowo na szybkość dostępu do bazy danych.

5.1. Implementacja i opis badania

Istnieją dwie implementacje dla każdego testu porównawczego w tym eksperymencie. Badanie zawiera algorytmy napisane w języku Java i Kotlin. Przypadek testu bazy danych zawiera przekonwertowany kod Java na Kotlin.

Wszystkie algorytmy zostały wykonane i zmierzone przez zewnętrzny skrypt Pythona (według zaleceń CLBG). Żadna z implementacji nie ma nieistotnych części kodu poza szkieletem aplikacji internetowej. Nadmiarowy kod, mogłyby zakłócić ostateczny wynik.

Pamięć podręczna i przestrzeń wymiany systemu plików są czyszczone przed wykonaniem pomiarów (skrypt Phyton CLBG) - każdy program ma podobny kontekst.

Wszystkie kody Java zostały pobrane bezpośrednio z repozytorium „The Computer Language Benchmark Game", bez dokonywania w nich zmian. Kody algorytmów w Kotlin, zostały stworzone na podstawie kodu Java z użyciem nowoczesnych wyrażeń z rozszerzenia języka. Kody użyte w eksperymencie to te, które osiągnęły najlepsze wyniki, zgodnie z tabelą wyników dostępnych

na stronie CLBG. Testy były uruchamiane w ramach aplikacji stworzonej w szkieletach aplikacji internetowych takich jak: Spring, Micronaut, Ktor.

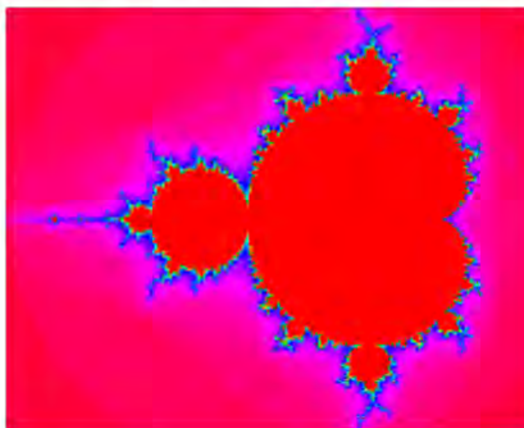
5.2. Opis użytych algorytmów

5.2.1. Mandelbrot

Termin zbioru Mandelbrota stosuje się zarówno w odniesieniu do ogólnej klasy zbiorów fraktalnych, jak i do konkretnego wystąpienia takiego zbioru. Zestaw Mandelbrota oznacza zbiór punktów na płaszczyźnie zespolonej, tak że odpowiadający mu zbiór Julii (dwa komplementarne tzn. będące swoimi dopełnieniami zbiory zdefiniowane przez odwzorowanie będące funkcją wymierną) jest przestrzenią spójną niemożliwą do obliczenia. „Mandelbrot" jest zbiorem uzyskanym z kwadratowego równania rekurencyjnego [6].

$$z_{(n+1)} = z^2{}_i + C$$

$z_0 = C$ gdzie punkty C w płaszczyźnie zespolonej, dla których orbita z_n nie ma tendencji do nieskończoności, w zbiorze. Ustawienie równego każdemu punktowi w zestawie, który nie jest punktem okresowym, daje ten sam wynik. Zestaw Mandelbrota został pierwotnie nazwany przez Mandelbrota cząsteczką mu. Jest to zazwyczaj fraktal przedstawiany za pomocą różnych kolorów, aby można zaobserwować różnicę w równaniach (Rysunek 1). Użyty w badaniu algorytm wykreśla zbiór Mandelbrota dla zbioru [-1,5-i, 0,5 + i] na mapie bitowej N-na-N punktów. Algorytm zapisuje bajt po bajcie w formacie bitmapy. Wyjście programu jest ustawione dla N=6000.



Rysunek 1: Jedna z repreznetacji zbioru Mandelbrot

5.2.2. FASTA

FASTA jest to w skrócie format zapisu sekwencji kwasów nukleinowych oraz białek.

W bioinformatyce i biochemii format FASTA jest formatem tekstowym do reprezentowania sekwencji (DNA, RNA) lub sekwencji aminokwasowych (białkowych), w których nukleotydy lub aminokwasy są reprezentowane przy użyciu kodów jednoliterowych. Format pozwala na nazywanie i komentowanie sekwencji [9]. Format pochodzi z pakietu oprogramowania o tożsamej nazwie FASTA, obecnie stał się uniwersalnym standardem w dziedzinie bioinformatyki.

Linia z opisem rozpoczyna się od znaku "większe niż" ("¿"). Pierwsze słowo po tym znaku służy jako identyfikator sekwencji. Dalej w tej samej linii umieszczany jest opis. W kolejnych liniach znajduje się ciąg znaków składający się na sekwencję.

Przykładowa sekwencje w formacie FASTA wygląda następująco:
¿MCHU - Calmodulin - Human, rabbit ADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAELQDMINEVDADGNGTIDFPE*

Algorytm zastosowany w eksperymencie generuje sekwencje DNA, kopiując z dane z innej sekwencji. Następnie generuje sekwencje DNA na podstawie losowej selekcji z 2 tablic alfabetów.

5.2.3. Drzewo binarne

Algorytm do testowania wydajnościowego to uproszczona adaptacja GCBench - Hansa Boehm'a, która została zaadaptowana na podstawie testu porównawczego przez Johna Ellisa i Pete Kovaca.

Program polega na stworzeniu idealnych drzew binarnych - zanim jakiekolwiek węzły drzew zostaną poddane odśmiecaniu pamięci. Węzły liścia są takie same jak węzły wewnętrzne, pamięć jest przydzielana w ten sam sposób.

Program definiuję klasę i metody węzła drzewa. Przydziela drzewo binarne do pamięci, sprawdza poprawność i jego istnienie. Następnie dealokuje obiekt drzewa z pamięci. Po utworzeniu drzewa binarnego implementacja koncentruje się na obliczeniu długowiecznego drzewa lub poddrzewa z parametrami maxDepth i stretchDepth. Algorytm sprawdza drzewo i jego poddrzewa, aby znaleźć drzewo o najdłuższym okresie życia.
Aby je znaleźć, algorytm przydziela drzewo binarne. Następnie je sprawdzana i cofa się po jego węzłach. Po znalezieniu najdłużej żyjącego drzewa usuwa je z pamięci [4].

5.3. Metryki

Każdy test jest uruchamiany i mierzony przy najmniejszej wartości wejściowej, dane wyjściowe programu są przekierowywane do pliku i porównywane z oczekiwanymi danymi wyjściowymi ze strony CLBG. Tak długo, jak dane wyjściowe odpowiadają oczekiwanym wynikom. Program jest uruchamiany ponownie i testowany przy następnej większej wartości wejściowej, aż do wykonania wszystkich zadeklarowanych pomiarów.

Jeśli program zwróci prawidłowe dane w czasie krótszym niż 120 sekund, wtedy algorytm jest uruchamiany pięć razy ponownie. Jeśli program nie da oczekiwanego wyniku w czasie godziny, eksperyment jest przerywany.

Otrzymane dane z testów zostały skrócone o wyniki z najmniejszym i największym zużyciem pamięci dla

pomiarów. Testy trwające dłużej niż 2 godzin zostały wykonane tylko raz.

5.4. Sposób mierzenia czasu

Podjęta została decyzja o zastosowaniu oficjalnych technik mierzenia i metryk z benchmarku CLBG. Skrypty użyte do eksperymentu, są opisywane w innych pracach naukowych. Każdy algorytm jest uruchamiany jako proces dziecko ze skryptu Python, za pomocą funkcji Popen. Czas jest mierzony przed uruchomieniem algorytmu do do momentu jego zamknięcia.

5.5. Obciążenie procesora

Obciążenie procesora jest mierzone za pomocą funkcji QueryInformation JobObject(hJob,JobObject BasicAccountingInformation) Metoda zwraca całkowity czas wykonywania aktywnych procesów powiązanych z zadaniem od ostatniego wywołania.

5.6. Obciążenie pamięci

Funkcja użyta z API Windows nazywa się: QueryInformation JobObject(hJob,JobObject ExtendedLimitInformation)

5.7. Obciążenie dostępu do bazy danych

Test jest wykonany z pomocą 3 frameworków (Spring, Micronaut, Ktor), dla każdego stosując język Java i Kotlin.
Techniki wydajnościowe, użyte w badaniu to test aktualizacji danych w bazie. Techniki wydajnościowe, użyte w badaniu szybkości działania bazy danych na poszczególnym szkielecie to test aktualizacji danych w bazie. Test wydajnościowy sprawdza: - trwałość obiektów ORM, - wydajność sterownika bazy danych przy uruchamianiu instrukcji UPDATE. Test polega na wykonywaniu dużej liczby operacji na bazie danych w stylu odczytu i zapisu.

5.8. Środowisko eksperymentu

Użyte w badaniu środowisko do programowania Java (java -version):

- java version "11.0.7" 2020-04-14 LTS
- Java(TM) SE Runtime Environment 18.9 (build 11.0.7+8-LTS)
- Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.7+8-LTS, mixed mode)

Użyte w badaniu środowisko do programowania Kotlin (kotlinc -version):

- info: kotlinc-native 1.3.72-eap-463 (JRE 11.0.7+8-LTS)
- Kotlin/Native: 1.3.72

Środowsiko programistyczne, na którym zostały uruchomione testy przedstawia tabela 1 opisująca parametry dysku, tabela 2 pamięć RAM oraz tabela 3 CPU.

Tabela 1: Dysk

Model	Seagate ST1000LM024 HN-M101MBB
Wielkość	1 TB
Szybkość	5400 RPM
Wersja SATA	SATA 3.0 3Gb/s

Tabela 2: Pamięć RAM

Model	Seagate ST1000LM024 HN-M101MBB
Wielkość	8 GB
Taktowanie	800 MHz

Tabela 3: Procesor:

Architektura	x86_64
CPUs	4
Wątki na procesor	2
Nazwa modelu	Intel Core i3-3120M
Technologia procesora	22nm
Taktowanie CPU	2500 MHz

6. Testy

Skrypt w języku Python organizuje wykonanie eksperymentu. Język zarządzający środowiskiem dla tego typu badania nie ma wpływu na wyniki.
Został on wybrany na podstawie łatwości zaimplementowania i zintegrowania ze skryptem Python
z repozytorium „The Computer Language Benchmarks Game". Dane dotyczące wydajności, obciążenia procesora i pamięci zostały dostarczone przez system w postaci plików CSV. Reszta danych razem z wynikiem testów bazy danych, została uporządkowana manualnie. W celu ułatwienia pracy, badanie zostało zautomatyzowane.

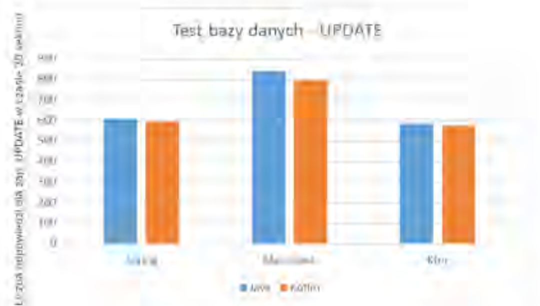
W tym celu skrypt wykonuje następujące kroki:

1. Przygotowywane jest środowisko testowe.
2. Przy pomocy narzędzi automatyzujących budowanie projektów (Maven), przygotowane zostają szkielety aplikacji internetowych. Każda ze zbudowanych aplikacji, posiada zaimplementowane testy wydajnościowe, które można uruchomić metodą http.
3. W zależności od badanego języka programowania, uruchamiany zostaje zadany szkielet aplikacji.
4. Zostają wywołane metody http uruchamiające poszczególne testy wydajnościowe. Testy są wywołane w kolejności: Fasta, Mandelbrot, tworzenie drzew binarnych, test bazy danych.
5. Po ukończeniu zadanych testów, wyniki zapisywane są do plików CSV.

7. Analiza wykonanych testów

7.1. Test odpowiedzi bazy danych

Porównanie wyników testów na bazie danych (liczba odpowiedzi na 20 sekund zapytań) pokazuje (Rysunek 2), że najbardziej optymalnym pod względem odpowiedzi jest szkielet Micronaut. Spośród trzech badanych platform, średnia dla Kotlina jest za każdym razem gorsza



Rysunek 2: Testy na bazie danych dla języków: Kotlin oraz Java (Spring, Micronaut, Ktor)

niż dla Javy. Eksperyment sprawdzający wydajność instrukcji UPDATE, pozawala zauważyć, że różnica w ilości odpowiedzi, może wynikać z różnicy szybkości działania Wirtualnej Maszyny Java.
Kotlin w każdym teście z bazami danych wypada gorzej niż Java.

7.2. Test obciążenia procesora

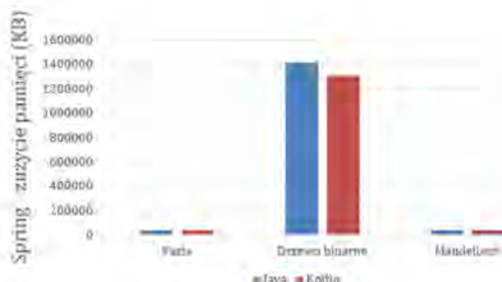
Obciążenie procesora jest podobne do siebie względem algorytmów i badanych języków programowania, niezależnie od szkieletu aplikacji – na której został uruchomiony test. Architektura procesora, wirtualna maszyna java, kod bajtowy – to wszystko ma znaczenie w tym eksperymencie.

Oba języki obciążają procesor w podobny sposób. Nie ma zauważalnego spadku obciążenia w żadnym z tych języków lub implementacji.

7.3. Zużycie pamięci

Znacząca zmienność wyników pomiaru zużycia pamięci jest widoczna w teście porównawczym „Drzewa binarne". Jak wspomniano wcześniej, test ten, głównie manipuluje liczbami całkowitymi. Istnieje różnica w jaki sposób, Java i Kotlin zarządzają liczbami całkowitymi.

7.3.1. Spring

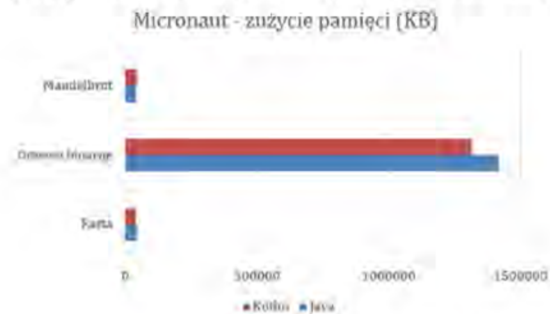


Rysunek 3: Test zużycia pamięci dla szkieletu Spring

W Javie liczby są w większości zapisywane jako typy wbudowane (prymitywne), w przeciwieństwie do Kotlina, gdzie wszystkie liczby są spakowane jako obiekty. Różnica ta jest widoczna na rysunku nr 3.

7.3.2. Micronaut

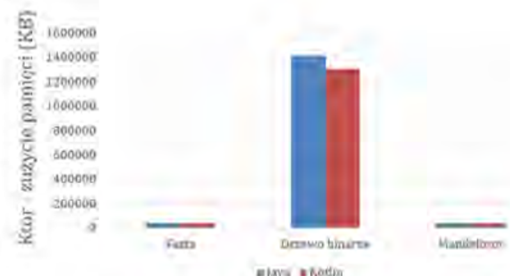
Micronaut to nowoczesny szkielet, oparty na JVM. Mikrousługa z pełnym wsparciem, zaprojektowany do budowania i testowania modułowych projektów.
Został opracowany przez twórców Grails i czerpie inspirację z tworzenia rzeczywistych aplikacji od monolitów po mikrousługi przy użyciu Spring, Spring Boot i Grails. Historycznie szkielet aplikacji, takie jak Spring i Grails, nie były zaprojektowane do działania w scenariuszach takich jak funkcje bez serwera (serverless - model usług w chmurze), aplikacje na Androida lub mikrousługi o niskim zużyciu pamięci. Micronaut został zaprojektowany odpowiednio dla tych wszystkich scenariuszy. Już



Rysunek 4: Test zużycia pamięci dla szkieletu Micronaut

rysunki 3 i 4 pozwalają stwierdzić, że użyty język i szkielet aplikacji nie koreluje, pod względem zużycia pamięci.

7.3.3. Ktor



Rysunek 5: Test zużycia pamięci dla szkieletu Ktor

Zużycia pamięci dla testu tworzenia drzew binarnych jest mniejszcze dla języka Kotlin niż Java również na szkielecie aplikacji Ktor, co widzać na rysunku 5.

7.4. Analiza czasu wykonania algorytmów

Analiza czasu wykonania algorytmów oparta jest na technikach mierzenia i metryk z benchmarku CLBG. Każdy algorytm jest uruchamiany jako proces dziecko ze skryptu Python, za pomocą funkcji Popen. Czas jest mierzony przed uruchomieniem algorytmu do momentu jego zamknięcia.

7.4.1. Spring

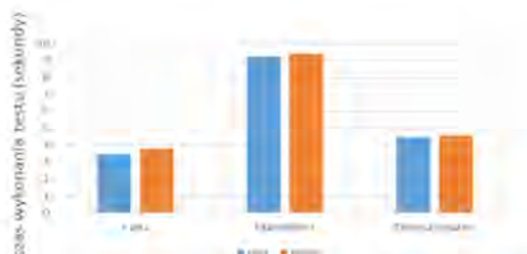
Z wykresu rysunku 6 można dostrzec, że w przypadku testu na tworzenie fraktali, kod Java wykonuje się znacząco szybciej. Z danych, można zauważyć, że w każdym

przypadku – test napisany w kodzie Java wykonuje się szybciej.

Rysunek 6: Test szybkości wykonania kodu dla szkieletu Spring

7.4.2. Micronaut

Podobnie jak we wcześniejszych testach wykres z rysunku 7, nie odbiega wynikami od innych szkieletów aplikacji internetowych. Wydajność jest niekorzystna dla technilogii Kotlin.



Rysunek 7: Test szybkości wykonania kodu dla szkieletu Micronaut

7.4.3. Ktor

Rysunek 8, przedstawia wyniki testu dla szkieletu Ktor. Pomiędzy szkieletami aplikacji internetowymi, można dostrzec niewielką różnicę w czasie wykonywania zadań. Wynika ona, prawdopodobnie z niedeterministycznego środowiska, na którym odbywały się testy.

Rysunek 8: Test szybkości wykonania kodu dla szkieletu Ktor

8. Wnioski

Aby zrozumieć koncepcję porównywania języków na przykładzie szkieletów aplikacji internetowej, w pracy przygotowano, krótki wstęp w tematykę semantyczną Java i Kotlin. Opisano przebieg generowania kodu wykonywalnego, wraz z użyciem Wirtualnej Maszyny Java.

Następnie na podstawie przeglądu artykułów naukowych wybrano i omówiono sposób porównywania. Trzy algorytmy testujące języki programowania z „The Computer Langugae Benchmark Game" (CLBG) zostały przedstawione jako metryka dla badań. Każdy z algorytmów testowych został szczegółowo opisany. Dodatkowo przedstawiono i przetestowano wydajność zapytań do bazy danych na różnych szkieletach aplikacji internetowej z użyciem Java i Kotlin.

Został uruchomiony skrypt zarządzający testami. Zmierzona została ilość odpowiedzi z bazy danych w czasie 20 sekund wysyłania zapytań z poszczególnego szkieletu aplikacji internetowej do bazy danych H2. Następnie skrypt Python z repozytorium kodu „The Computer Langugae Benchmark Game" uruchomił na każdym z szkieletów tj. Spring, Micronaut, Ktor zestaw testów:

- badanie obciążenia procesora,
- badanie czasu wykonywania programu,
- zużycie pamięci.

Kolejnym krokiem była analiza na podstawie wyników. Analizowano współczynniki takie jak: ilość odpowiedzi z serwera w zadanym czasie, procentowe obciążenie CPU, szybkość zwracania wyników w sekundach oraz zużycie pamięci w KB. Wnioski z pracy są następujące: Implementacja w języku Kotlin nigdy nie osiągnęła najlepszego wyniku w żadnej grupie pomiarów. Zróżnicowanie pod względem szkieletów aplikacji internetowych, szybkości odpowiedzi bazy danych i implementacji testów w różnych językach – we wszystkich testach wychodzi na niekorzyść języka Kotlin. We wszystkich przypadkach, wydajność była gorsza o średnio 5,6%. Pomiędzy szkieletami Spring i Ktor, nie dostrzega się dużej różnicy. Na tle obu wyróżnia się Micronaut z najlepszymi wynikami w badaniu. Szkielet Micronaut nie korzysta z refleksji, co może wpływać na lepszą wydajność w stosunku do konkurencyjnych rozwiązań.

Nie ma znaczącej różnicy pomiędzy obciążeniem procesora pomiędzy poszczególnymi pomiarami, różnica nie przekracza 2%. Takie różnice mogą wynikać ze względu na środowisko badawcze tj. sprzęt komputerowy i oprogramowanie systemu, W dziedzinie testów CLBG, porównując parametr szybkości wykonania algorytmu, najszybciej wykonał się test Mandelbrot dla szkieletu Spring. W każdym z testów szybciej wykonują się programy zaimplementowane w języku Java na każdym z szkieletów internetowych.

Nie ma dużej różnicy pomiędzy szkieletami aplikacji. Największe znaczenia ma język implementacji. Kod Java osiąga najlepszy czas wykonania, we wszystkich testach. Wynika, to z optymalizacji Wirtualnej Maszyna Java dla tego języka.

Porównanie zużycia pamięci okazało się najniższe w implementacji Java, na wszystkich aplikacjach internetowych i uruchomionych testach, z jednym wyjątkiem. Kotlin zużywa mniej pamięci w implementacji te-

stu tworzenia drzew binarnych. Jest to 8% różnica na korzyść Kotlin.
Szkielety aplikacji internetowych nie wykazują większego wpływu na zróżnicowanie badania.

Głębsza analiza danych pozwoliłaby na wyciągniecie jeszcze większej ilości wniosków. Wyniki z badań zawartych w tym artykule skłaniają do kolejnych badań i porównań, skupionych na kodzie bajtowym i Wirtualnej Maszynie Java.

Literatura

[1] M. Martinez, B. Gois. How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers, arXiv preprint arXiv:2003.12730 (2020).

[2] S. Bose, A comparative study: java vs kotlin programming in android application development, International Journal of Advanced Research in Computer Science (9) (2018) 41-45.

[3] T. Kalibera, R. Jones, Rigorous benchmarking in reasonable time, in Proceedings of the 2013 international symposium on memory management (2013) 63-74.

[4] I. Gouy, The Computer Language Benchmarks Game. Web.(https://benchmarksgame-team.pages.debian.net/benchmarksgame/).

[5] A. Prokopec, Oracle Labs: On Evaluating the Renaissance Benchmarking Suite: Variety, Performance, and Complexity, arXiv preprint arXiv:1903.10267 (2019).

[6] P. Alfeld, The Mandelbrot Set (https://www.math.utah.edu/~alfeld/math/mandelbrot/mandelbrot.html)

[7] D. Stepanov, M. Akhin, M. Belyaev, How We Stopped Worrying About Bugs in Kotlin Compiler, in 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE (2019) 317-326.

[8] B. G. Mateus, M. Martinez, An empirical study on quality of Android applications written in Kotlin language, Empirical Software Engineering (2019) 3356-3393.

[9] D. J. Lipman, W. R. Pearson, Rapid and sensitive protein similarity searches, Science 227 (4693) (1985) 1435–1441.

[10] TIOBE index (https://www.tiobe.com/tiobe-index/)

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 227–232
Received: 25 July 2020
Accepted: 3 September 2020

Blender as a tool for generating synthetic data

Blender jako narzędzie do generacji danych syntetycznych

Rafał Sieczka*, Maciej Pańczyk

Department of Computer Science, Lublin University of Technology, ul. Nadbystrzycka 38, 20-618 Lublin, Poland

Abstract

Acquiring data for neural network training is an expensive and labour-intensive task, especially when such data is difficult to access. This article proposes the use of 3D Blender graphics software as a tool to automatically generate synthetic image data on the example of price labels. Using the fastai library, price label classifiers were trained on a set of synthetic data, which were compared with classifiers trained on a real data set. The comparison of the results showed that it is possible to use Blender to generate synthetic data. This allows for a significant acceleration of the data acquisition process and consequently, the learning process of neural networks.

Keywords: artificial neural networks; convolutional neural network; synthetic data; blender

Streszczenie

Pozyskiwanie danych do treningu sieci neuronowych, jest kosztownym i pracochłonnym zadaniem, szczególnie kiedy takie dane są trudno dostępne. W niniejszym artykule zostało zaproponowane użycie programu do grafiki 3D Blender, jako narzędzia do automatycznej generacji danych syntetycznych zdjęć, na przykładzie etykiet cenowych. Przy użyciu biblioteki fastai, zostały wytrenowane klasyfikatory etykiet cenowych, na zbiorze danych syntetycznych, które porównano z klasyfikatorami trenowanymi na zbiorze danych rzeczywistych. Porównanie wyników wykazało, że możliwe jest użycie programu Blender do generacji danych syntetycznych. Pozwala to w znaczącym stopniu przyśpieszyć proces pozyskiwania danych, a co za tym idzie proces uczenia sieci neuronowych.

Słowa kluczowe: sztuczne sieci neuronowe; konwolucyjne sieci neuronowe; dane syntetyczne; blender

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

1. Wstęp

W ostatniej dekadzie sztuczne sieci neuronowe znalazły zastosowanie w wielu dziedzinach. Jedną z nich jest widzenie komputerowe (ang. computer vision), gdzie głębokie uczenie (ang. deep learning), wykorzystywane jest przy rozwiązywaniu wielu problemów z tej dziedziny [1] m.in. detekcja obiektów, rozpoznawanie aktywności, twarzy i szacowanie pozycji człowieka [2–5]. Trenowanie jak i testowanie głębokich sieci neuronowych jest czasochłonne oraz kosztowne, ponieważ należy zebrać dużą ilość danych, następnie ręcznie je opisać, aby wykorzystać je w uczeniu nadzorowanym. Jest to problem występujący nie tylko przy głębokim uczeniu ale w szeroko pojętym uczeniu maszynowym (ang. machine learning) [6], mimo że istnieją łatwo dostępne, gotowe zbiory danych np.: VGG-Sound [7], RoadText [8], PandaSet [9], czy też COVID-CT [10]. Może się jednak okazać, że taki zbiór nie pasuje do problemu, który osoba trenująca próbuje rozwiązać, ponieważ zbiór jest zbyt mały lub słabej jakości aby uzyskać zadowalające wynik, bądź względy prawne nie pozwalają go wykorzystać. Obiecującym rozwiązaniem jest użycie automatycznie wygenerowanych danych syntetycznych. W zależności od rodzaju danych jakie są potrzebne - dane tabelaryczne, zdjęcia, wideo, dźwięk - proces generowania takich danych - jeśli możliwy, może być różny.

W przypadku wizji komputerowej, generacja danych syntetycznych, może odbywać się przy pomocy silników gier, bądź programów do grafiki 3D takich jak Unity 3D, Unreal Engine, Cinema4D, Blender [11]. Umożliwia to automatyzację procesu generowania i opisywania danych potrzebnych do treningu sieci. Przy użyciu tego typu programów powstało kilka zbiorów danych syntetycznych [12–22] zdjęć i filmów. W większości przypadków, te zbiory danych są kosztowne w wygenerowaniu. Wymagają od osoby tworzącej sceny, aby ta w szczegółach odpowiadała określonemu środowisku. Te zbiory danych były z powodzeniem używane do rozwiązywania geometrycznych problemów takich jak przepływ optyczny, sceniczny i szacowania pozycji kamery.

Dane syntetyczne zostały wykorzystane z sukcesem w pracy [23], gdzie zostało przedstawione użycie danych syntetycznych podczas trenowania modeli, do rozpoznawania tekstu na zdjęciach przedstawiających sceny ze-

*Corresponding author
Email address: rafal.sieczkaa@gmail.com (R. Sieczka)

wnętrzne. Dane syntetyczne w tym przypadku generowane są jako tekst nakładany na zdjęcia ze zbioru ICDAR 2003 i SVT. Podobne rozwiązanie zostało przedstawione w [24], jednak tym razem na zdjęcia nakładane są obiekty, a nie tekst. W tym celu zostały wykorzystane darmowe modele 3D CAD. Po wyrenderowaniu modeli z teksturami, w różnych pozach, obiekt zostaje naniesiony na różne tła. Na tak przygotowanych zdjęciach zostały wytrenowane detektory obiektów. Kolejne dwie prace [25, 26] także skupiają się na detekcji obiektów, używając przy tym Blendera i Unity 3D. Rozwiązanie wykorzystujące Blendera, podczas renderowania używa silnika Cycles, aby wyrenderowane obiekty były bardziej zbliżone do rzeczywistych obiektów, wpływa to jednak na prędkość generacji. Dane syntetyczne używane były również przy szacowaniu pozycji człowieka. W tym celu został stworzony zbiór realistycznych danych syntetycznych [27]. Istnieją jeszcze inne prace [28–32], które skupiają się na generacji danych syntetycznych lub ich wykorzystaniu do treningu modeli sieci neuronowych.

Przedstawione powyżej rozwiązania w dużej mierze skupiają się na uzyskaniu jak najbardziej realistycznych zdjęć obiektów, pomieszczeń, co odbija się na prędkości generacji takich danych. Przy małej liczbie zdjęć nie stanowi to dużego problemu, jednak kiedy potrzebujemy setek tysięcy zdjęć do treningu, ten czas się kumuluje. Skupiają się też one na jak najdokładniejszym odwzorowaniu otoczenia, w którym znajduje się obiekt, co również jest czasochłonne. Aby zaadresować te problemy, w tym artykule został przedstawiony sposób generowania danych syntetycznych, wykorzystując do tego program Blender, WeasyPrint, python-barcode. Do sprawdzenia użyteczności wygenerowanych danych, przygotowanym programem, zostały wytrenowane klasyfikatory etykiet cenowych na zbiorze treningowym danych syntetycznych i rzeczywistych. Przedstawione rozwiązanie wykorzystuje silnik renderujący Eevee, który zapewnia szybszy czas renderowania, w porównaniu do silnika Cycles. Otoczenie etykiet cenowych jest bardzo proste, aby nie tracić czasu na jego modelowanie.

2. Generacja danych i trening

W tej sekcji zostały opisane kroki w generacji danych syntetycznych etykiet cenowych oraz sposób w jaki zostały pozyskane dane rzeczywiste użyte w zbiorze ewaluacyjnym i testowym. Dodatkowo został opisany sposób trenowania sieci neuronowych z użyciem biblioteki fastai [33].

2.1. Generacja danych z wykorzystaniem Blendera

Przy tworzeniu programu do generacji danych syntetycznych pod uwagę brana była modularność projektu, aby w łatwy sposób móc dodawać generację innych obiektów niż etykiety cenowe. Dlatego obiekty, które są używane podczas generacji nie znajdują się w tym samym pliku blend co scena, w której są generowane. Pozwala to na łatwą zmianę obiektów, bez zaśmiecania głównej sceny oraz na wymianę sceny, jeśli np.: potrzebne są zdjęcia na zewnątrz lub wewnątrz budynku. Dla każdego typu obiektu tworzony jest osobny plik blend, w którym znajdują się obiekty tylko danego typu np.: samochody, książki, biurka itd. Pozwala to na łatwiejsze zarządzanie obiektami.

Aby było wiadomo, gdzie znajduje się jaki typ obiektu i mieć możliwość definiowania specjalnych zmiennych, zamiast ustawiać je na sztywno w kodzie, stworzony został plik json, w którym znajduje się konfiguracja. Przykład zawartości konfiguracji znajduje się na Listingu 1. W pliku json, niezależnie od typu obiektu znajdują się pola: type - typ obiektu, object_name - nazwa obiektu w pliku blend, blend_file - ścieżka do pliku blend, gdzie znajduje się model obiektu. Przy generacji wykorzystywany jest Eevee, który jest silnikiem renderującym czasu rzeczywistego. Skraca to znacząco czas potrzebny na wyrenderowanie zdjęcia nawet prostej sceny.

Listing 1: Przykładowy plik konfiguracyjny

```
1  {
2  "objects": [
3      {
4          "type": "price_label",
5          "object_name": "electronic_label_1
               ",
6          "template_file": "
               electronic_label_1.html",
7          "css_file": "electronic_label_1.
               css",
8          "price_label_type": "electronic",
9          "blend_file": "etykiety.blend",
10         "barcode": "ean8",
11         "barcode_options": {
12             "dpi": 300,
13             "module_height": 3.0,
14             "quiet_zone": 0.5,
15             "write_text": false
16         }
17     }
18 ],
19 "renderer_settings": {
20     "engine": "evee",
21     "width": 1920,
22     "height": 1080,
23     "samples": 64
24 }
25 }
```

Scena, w której renderowane były zdjęcia etykiet cenowych, została przedstawiona na Rysunku 1. W scenie została użyta mapa hdr jako tło. Scena składa się z prostego pomieszczenia, w którym znajdują się dwa regały sklepowe. Na regałach znajdują się różnego koloru sześciany. Zostały użyte 3 źródła światła typu Area.

Tekstury etykiet cenowych są generowane przy użyciu WeasyPrint, który potrafi zamienić plik html i css w zdjęcie. Dzięki temu możliwe jest stworzenie wielu różnych układów etykiet cenowych. Pozwala to również na

Rysunek 1: Scena, która była użyta przy generacji syntetycznych zdjęć etykiet cenowych

umieszczanie losowych wartości w polach etykiety takich jak: nazwa produktu, cena, waga itp. Kody kreskowe generowane są przy pomocy biblioteki python-barcode.

Proces generacji zdjęć etykiet cenowych można podzielić na 3 kroki.

1. Inicjalizacja programu
   W tym kroku, program wczytuje plik konfiguracyjny i tworzone są niezbędne obiekty generatorów, dla każdego typu obiektu. Zadaniem generatora jest wyrenderowanie zdjęć danego obiektu. Do pliku sceny zostają również dołączone wymagane obiekty z innych plików blend.
2. Tworzenie tekstur
   W tym kroku wybierana jest losowo jedna z etykiet cenowych. Dla wybranej etykiety cenowej, wczytywany jest plik html oraz css i na ich podstawie generowana jest tekstura etykiety cenowej. Przykładowa tekstura została przedstawiona na Rysunku 2.

Rysunek 2: Przykładowa wygenerowana tekstura etykiety elektronicznej

3. Rednering zdjęcia
   Kiedy tekstura zostanie wygenerowana, wczytywana jest ona do programu blender i przypisywana do odpowiedniego materiału. Dany obiekt jest ustawiany w wyznaczonym miejscu, kamera zostaje ustawiona w losowym miejscu, w ustalonej odległości od obiektu, po czym wykonywany jest rendering zdjęcia. Po wyrenderowaniu, zdjęcie zostaje przycięte, w taki sposób aby etykieta zajmowała jak najwięcej miejsca na zdjęciu. Po zapisaniu zdjęcia, proces powraca do kroku 2, aż nie zostanie wygenerowana zadana liczba zdjęć.

Przykładowe wygenerowane zdjęcia różnych klas etykiet cenowych, zostały przedstawione na Rysunku 3.

Rysunek 3: Przykład wygenerowanych etykiet cenowych

2.2. Trenowanie modeli z wykorzystaniem biblioteki fastai

Zostały wytrenowane po 3 modele na zbiór danych syntetycznych i rzeczywistych (DenseNet121, ResNet50, VGG19) przy użyciu biblioteki fastai. Zbiór treningowy danych syntetycznych składa się z 5000 zdjęć, które były generowane w rozdzielczości 768x768px, zbiór danych ewaluacyjnych z 2016 zdjęć rzeczywistych, zbiór testowy z 1008 zdjęć rzeczywistych a zbiór danych rzeczywistych z 3070 zdjęć. Każdy ze zbiorów został podzielony na 3 klasy etykiet - paper, electronic i handwritten. Zdjęcia etykiet rzeczywistych zostały wycięte ze zdjęć półek sklepowych.

Modele zostały wytrenowane przy użyciu polityki 1 cyklu. Były one trenowane na zdjęciach ze zwiększającą się rozdzielczością - 32x32px, 64x64px i 128x128px. Na zdjęciach w zbiorze treningowym wykonane zostały augmentacje: crop_pad, flip_lr, symmetric_wrap, zoom, brightness oraz contrast. W czasie treningu, najlepszy punkt kontrolny był zapisywany i każdy kolejny trening na większej rozdzielczości zdjęć, dla danego zbioru i modelu, był wznawiany z tego punktu kontrolnego.

3. Wyniki

Dane syntetyczne zostały wygenerowane 10 razy po 5000 zdjęć, aby sprawdzić średni czas generacji zdjęć. Dla 5000 zdjęć średni czas generacji wyniósł 7507 sekund, czyli 2 godziny i 8 minut, natomiast średni czas renderowania jednego zdjęcia - używając do tego karty graficznej GTX 1660 Ti - wyniósł 1 sekundę, gdzie dla porównania, średni czas renderowania jednego zdjęcia, z użyciem silnika cycles, wyniósł 1 minutę i 2 sekundy. Podczas renderowania zdjęć, przy użyciu silnika cycles, wpływ na szybkość renderingu ma wiele ustawień, najważniejszymi z nich są ilość próbek i wielkość części zdjęcia, które w danej chwili jest renderowane. Mniejsza liczba próbek przyśpieszy renderowanie, jednak zdjęcie będzie gorszej jakości - bardziej ziarniste. Natomiast największy wpływ na czas generacji jednego zdjęcia miało renderowanie oraz tworzenie tekstury etykiety, dlatego całkowity czas generacji zdjęcia etykiety to suma czasu potrzebnego na generację tekstury i renderowanie zdjęcia.

W tabeli 1 zostały przedstawione procentowe wyniki dokładności sklasyfikowania etykiet cenowych, uzyskane na zbiorze testowym dla klasyfikatorów trenowanych na

danych syntetycznych i rzeczywistych.

Tabela 1: Procentowy wynik dokładności sklasyfikowania etykiet cenowych dla modeli trenowanych na danych syntetycznych i rzeczywistych

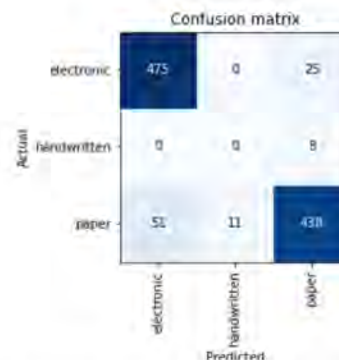
Zbiór danych	Architektura	Wynik precyzji na zbiorze testowym
Dane syntetyczne	ResNet50	91.5%
	DenseNet121	90.6%
	VGG19	70.5%
Dane rzeczywiste	ResNet50	98.9%
	DenseNet121	98.4%
	VGG19	98.1%

Różnica pomiędzy najlepszym klasyfikatorem, który został wytrenowany na danych rzeczywistych, a klasyfikatorem, który został wytrenowany na danych syntetycznych, wynosi 7.4%. Klasyfikatory ResNet50 oraz DenseNet121 trenowane na syntetycznych zdjęciach, uzyskały zbliżone wyniki, które można by polepszyć, dostosowując hiperparametry uczenia. Podczas treningu zmieniany był tylko współczynnik nauki (ang. Learning rate). Najgorszy wynik uzyskał klasyfikator oparty o architekturę VGG19.

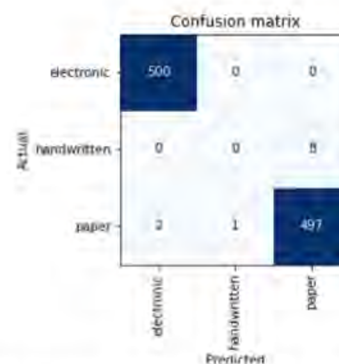
Mimo, że do renderowania etykiet nie był używany silnik Cycles, który renderuje zdjęcia, bardziej zbliżone do rzeczywistości, wyniki uzyskane przez najlepszy klasyfikator, trenowany na danych syntetycznych nie jest mniejszy od klasyfikatora trenowanego na danych rzeczywistych o 10%.

Patrząc na matrycę pomyłek dla najlepszego modelu trenowanego na danych syntetycznych (Rysunek 4) i matrycę pomyłek najlepszego modelu trenowanego na danych rzeczywistych (Rysunek 5), widać, że obydwa klasyfikatory, ani razu nie sklasyfikowały dobrze etykiet klasy handwritten. Przy klasyfikatorze trenowanym na danych rzeczywistych, wynika to z małej liczby zdjęć etykiet klasy handwritten w zbiorze treningowym. Przy klasyfikatorze trenowanym na danych syntetycznych, była wystarczająca liczba zdjęć tej klasy. Ten błąd wynika z tego, że syntetyczne dane etykiet klasy handwritten, za bardzo przypominały etykiety paper. Wpływ na to ma rodzaj użytej czcionki, która nie różni się znacząco od czcionek użytych przy etykietach paper.

Wpływ na wyniki uzyskane dla klasyfikatorów, trenowanych na danych syntetycznych ma również nałożenie tekstur na obiekty etykiet. Tekstury generowały się z losową nazwą produktu, ceną, kodem kreskowym itd. więc zdarzało się, że niektóre tekstury źle nakładały się na obiekty. Kolejnym czynnikiem mającym wpływ na wynik jest odwzorowanie układów występujących w rzeczywistości. Może się zdarzyć, że dany układ etykiety występuje rzadko w danych rzeczywistych, jednak w danych syntetycznych, występuje z częstotliwością zbliżoną do innych układów etykiet. W tym przypadku widać to na etykietach typu handwritten, których w zbiorach składających się z danych rzeczywistych jest bardzo mało, w porównaniu do innych klas etykiet, natomiast w zbiorze syntetycznym, występują porównywalnie często co inne klasy etykiet. Powoduje to, że taki klasyfikator ma większą szansę, źle sklasyfikować etykietę jako handwritten.



Rysunek 4: Matryca pomyłek dla modelu ResNet50 trenowanego na danych syntetycznych



Rysunek 5: Matryca pomyłek dla modelu ResNet50 trenowanego na danych rzeczywistych

4. Podsumowanie i wnioski

Przedstawiona metoda generacji danych syntetycznych przy użyciu Blendera, kiedy używamy do renderowania silnika Eevee, umożliwia szybką generację danych do treningu sieci neuronowych. Dzięki temu proces zbierania i opisywania danych staję się znacząco szybszy, co pozwala na szybsze dostarczanie wytrenowanych modeli.

Sieć wytrenowaną na danych syntetycznych, można użyć w dalszej części procesu, kiedy to duża liczba danych rzeczywistych będzie dostępna. Trzeba jednak zauważyć, że wytrenowanie sieci na danych syntetycznych, może się wiązać z większą ilością kroków, potrzebną do uzyskania wymaganych wyników.

Patrząc na uzyskane wyniki, można stwierdzić, że użycie silnika Eevee - szybszego ale renderującego mniej realistyczne zdjęcia – do renderowania syntetycznych zdjęć, pozwala na uzyskanie zadowalających wyników i znacząco skraca czas potrzebny na renderowanie.

Klasyfikator VGG19 uzyskał najgorszy wynik, kiedy został wytrenowany na danych syntetycznych, ponieważ jest to architektura, która posiada najwięcej parame-

trów, co może sugerować, że architektury posiadające bardzo dużą liczbę parametrów, uczą się gorzej na danych syntetycznych.

Literatura

[1] A. Voulodimos, N. Doulamis, A. Doulamis, E. Protopapadakis, Deep learning for computer vision: A brief review, Computational intelligence and neuroscience (2018).

[2] Z. Cao, G. Martinez, T. Simon, S. Wei, Y. A. Sheikh, Openpose: Realtime multi-person 2d pose estimation using part affinity fields. IEEE Transactions on Pattern Analysis and Machine Intelligence (2019) 7291-7299.

[3] T Simon, H Joo, I Matthews, Y Sheikh, Hand keypoint detection in single images using multiview bootstrapping, CVPR (2017) 1145-1153.

[4] Z Cao, T Simon, S EnWei, Y. Sheikh, Realtime multi-person 2d pose estimation using part affinity fields, CVPR (2017) 7291-7299.

[5] S. En Wei, V. Ramakrishna, T. Kanade, Y. Sheikh, Convolutional pose machines, CVPR (2016) 4724-4732.

[6] Y. Roh, G. Heo, S. E. Whang. A survey on data collection for machine learning: a big data – ai integration perspective, IEEE Transactions on Knowledge and Data Engineering (2019).

[7] C. Xie, L. Vedaldi, P. Zisserman. Vgg-sound: A largescale audio-visual dataset (2020).

[8] S. Reddy, M. Mathew, L. Gomez, M. Rusinol, D. Karatzas., C. V. Jawahar, Roadtext-1k: Text detection and recognition dataset for driving videos, 2020 IEEE International Conference on Robotics and Automation (2020) 11074-11080.

[9] S. Hesai, Pandaset - public large-scale dataset for autonomous driving.

[10] J. Zhao, Y. Zhang, X. He, P. Xie. Covid-ct-dataset: a ct scan dataset about covid-19. arXiv preprint arXiv:2003.13865 (2020).

[11] Blender Online Community. Blender - a 3D modelling and rendering package. Blender Foundation, Stichting Blender Foundation, Amsterdam (2018).

[12] A.Tsirikoglou, J. Kronander, M. Wrenninge, J. Unger, Procedural modeling and physically based rendering for synthetic data generation in automotive applications, arXiv preprint arXiv:1710.06270 (2017).

[13] A. Gaidon, Q. Wang, Y. Cabon, E. Vig, Virtual worlds as proxy for multi-object tracking analysis, proceedings of the IEEE conference on computer vision and pattern recognition (2016) 4340-4349.

[14] M. Muller, V. Casser, J. Lahoud, N. Smith, B. Ghanem, Sim4cv: A photo-realistic simulator for computer vision applications, International Journal of Computer Vision, 126(9) (2018) 902-919.

[15] J. McCormac, A. Handa, S. Leutenegger, A. J. Davison, Scenenet rgb-d: 5m photorealistic images of synthetic indoor trajectories with ground truth, arXiv preprint arXiv:1612.05079 (2016).

[16] Y. Zhang, W. Qiu, Q. Chen, X. Hu, A. Yuille, Unrealstereo: Controlling hazardous factors to analyze stereo vision, in proceedings of International Conference on 3D Vision (3DV) (2018) 228-237.

[17] W. Qiu, A. Yuille, Unrealcv: Connecting computer vision to unreal engine, in proceedings of European Conference on Computer Vision (2016) (909-916).

[18] N. Mayer, E. Ilg, P. Hausser, P. Fischer, D. Cremers, A. Dosovitskiy, T. Brox, A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation, in Proceedings of the IEEE conference on computer vision and pattern recognition (2016) 4040-4048.

[19] P. Fischer, A. Dosovitskiy, E. Ilg, P. Hausser, C. Hazırba¸s, V. Golkov, P. van der Smagt, D. Cremers, T. Brox, Flownet: Learning optical flow with convolutional networks, in Proceedings of the IEEE international conference on computer vision (2015) 2758-2766.

[20] S. R. Richter, V. Vineet, S. Roth, V. Koltun, Playing for data: Ground truth from computer games, in European conference on computer vision (2016) 102–118.

[21] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, A. M. Lopez, The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes, IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016) 3234–3243.

[22] D. J. Butler, J. Wulff, G. B. Stanley, M. J. Black, A naturalistic open source movie for optical flow evaluation, in proceedings of Computer Vision – ECCV (2012) 611–625.

[23] M. Jaderberg, K. Simonyan, A. Vedaldi, A. Zisserman, Synthetic data and artificial neural networks for natural scene text recognition, arXiv preprint arXiv:1406.2227 (2014).

[24] X. Peng, B. Sun, K. Ali, K. Saenko. Learning deep object detectors from 3d models, in Proceedings of the IEEE International Conference on Computer Vision (2015) 1278-1286.

[25] P. S. Rajpura, H. Bojinov, R. S. Hegde, Object detection using deep cnns trained on synthetic images, arXiv preprint arXiv:1706.06782 (2017).

[26] K. Wang, F. Shi, W. Wang, Y. Nan, S. Lian, Synthetic data generation and adaption for object detection in smart vending machines, arXiv preprint arXiv:1904.12294 (2019).

[27] G. Varol, J. Romero, X. Martin, N. Mahmood, M. J. Black, I. Laptev, C. Schmid. Learning from synthetic humans, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017) 109–117.

[28] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, S. Birchfield, Training deep networks with synthetic data: Bridging the reality gap by domain randomization, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (2018) 969–977.

[29] C. Mitash, K. E. Bekris, A. Boularias, A self-supervised learning system for object detection using physics simulation and multi-view pose estimation, in IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2017) 545-551.

[30] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, P. Abbeel, Domain randomization for transferring deep neural networks from simulation to the real world, in proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2017) 23-30.

[31] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, A. M. Lopez, The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes, IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016) 3234–3243.

[32] H. Hattori, V. N. Boddeti, K. M. Kitani, T. Kanade, Learning scene-specific pedestrian detectors without real data, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2015) 3819–3827.

[33] J. Howard, S. Gugger. fastai: A layered api for deep learning, Information 11(2) (2020) 108.

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 233–240
Received: 27 August 2020
Accepted: 9 September 2020

Analysis of selected methods of creating artificial intelligence on the example of a popular card game

Analiza wybranych metod tworzenia sztucznej inteligencji na przykładzie popularnej gry w karty

Łukasz Gałka*, Mariusz Dzieńkowski

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

Abstract
The aim of the article was to analyze selected methods of creating artificial intelligence in a popular card game. Two experiments were conducted: with a human and with a computer. The following algorithms were analyzed: random, min-max, based on a neural network, statistical and statistical with the use of "cheating" technique. The examined parameters were as follows: efficiency, execution time, number of implementation code lines, implementation time and training duration. The indicator with the greatest impact on the selection of the most optimal method was efficiency. The research has shown no difference in efficiency for the neural network-based algorithm and the statistical algorithm. In other cases, the differences in this feature were significant. The use of the "cheating" technique has increased the efficiency.

Keywords: artificial intelligence; machine learning; algorithm efficiency evaluation; computer games

Streszczenie
Celem artykułu była analiza wybranych metod tworzenia sztucznej inteligencji w popularnej grze w karty. Zostały przeprowadzone dwa eksperymenty: z człowiekiem oraz z komputerem. Analizie poddano algorytmy: losowy, min-max, bazujący na sieci neuronowej, statystyczny oraz statystyczny z użyciem techniki „oszukiwania". Zbadano takie parametry jak: skuteczność, czas wykonania, liczbę linii kodu implementacji, czas implementacji oraz czas trwania treningu. Wskaźnikiem mającym największy wpływ na wybór najbardziej optymalnej metody była skuteczność. Badania wykazały brak różnic w skuteczności dla algorytmu bazującego na sieci neuronowej i algorytmu statystycznego. W pozostałych przypadkach różnice tej cechy były istotne. Użycie techniki „oszukiwania" zwiększyło skuteczność.

Słowa kluczowe: sztuczna inteligencja; uczenie maszynowe; ocena skuteczności algorytmów; gry komputerowe

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

1. Wstęp

Postęp techniczny i zwiększenie wydajności obliczeniowej komputerów pozwoliły na stworzenie sztucznej inteligencji [1]. Poprzez dziesięciolecia sztuczna inteligencja była wykorzystywana w rozwiązywaniu najtrudniejszych problemów współczesnego świata. Komputery potrafią podejmować skomplikowane decyzje, bazując na danych i informacjach, których ludzki umysł nie jest w stanie samodzielnie przetworzyć [2]. W praktyce używa się pojęcia uczenia maszynowego ze względu na specyfikację podejmowanych przez komputer decyzji [3]. Wyróżnia się dwa główne nurty: uczenie z nadzorem oraz uczenie bez nadzoru. Uczenie z nadzorem polega na wykorzystaniu przez algorytm opracowanych gotowych wyników. Natomiast uczenie bez nadzoru nie wykorzystuje wyników, ale próbuje je przewidzieć poprzez analizę danych wejściowych [3, 4]. Do czołowych rozwiązań można zaliczyć: techniki eksploracji danych, sieci neuronowe, metody grupowania danych, metody statystyczne, drzewa decyzyjne czy algorytmy ewolucyjne [5–10]. Często techniki sztucznej inteligencji wykorzystuje się do rozwiązywania problemów, których precyzyjne rozwiązanie jest niewykonalne obliczeniowo. Przy implementacji należy brać pod uwagę dostępne gotowe narzędzia pozwalające na realizację konkretnych algorytmów. Popularnym językiem z bogatą kolekcją bibliotek uczenia maszynowego jest język Java. Należy tu wymienić najpopularniejsze biblioteki takie jak: RapidMiner, Weka, Deeplearning4j [11].

Pod koniec XX wieku rozpoczęła się era gier komputerowych, które również zaczęły używać inteligentnych rozwiązań w celu sterowania wirtualnymi światami [12]. Rzeczywiste rozgrywki zaczęły być przenoszone do wirtualnej rzeczywistości. Przykładem jest gra kar-

*Corresponding author
Email address: lukasz.galka2@pollub.edu.pl (Ł. Gałka)

ciana „Tysiąc" pierwotnie rozgrywana przy stołach wyłącznie z graczami będącymi ludźmi. W dobie superszybkich komputerów i mikroprocesorów istnieje możliwość gry z przeciwnikami sterowanymi przez maszyny i mogącymi dorównać inteligencją człowiekowi. Programiści implementują interakcję pomiędzy człowiekiem, a programem komputerowym w sposób bardzo naturalny i podobny do rzeczywistych interakcji graczy [13,14]. Dobór odpowiednich metod sztucznej inteligencji jest kluczowym aspektem. Algorytmy muszą być wydajne obliczeniowo i działać w czasie rzeczywistym. Ponadto muszą reprezentować odpowiedni poziom trudności rozgrywki, aby w pełni sprostać wymaganiom nawet najbardziej wymagających umysłów. Istnieje wiele rodzajów gier, które można podzielić na następujące gatunki: sportowe, zręcznościowe, platformowe, klasyczne, karciane, RPG, FPS czy RTS. Każdy gatunek cechują odmienne metody sterowania światem wirtualnym i jego elementami. Głównym celem artykułu jest analiza wybranych metod tworzenia sztucznej inteligencji i w wyniku tej analizy wybór najlepszej metody.

2. Aplikacja testowa

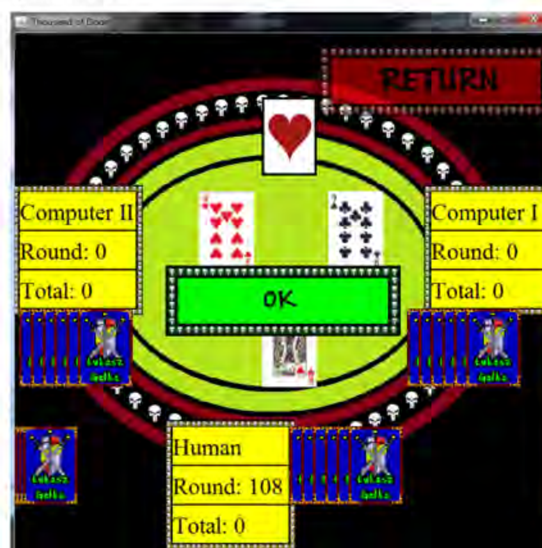
W celu analizy podstawowych parametrów metod tworzenia sztucznej inteligencji stworzono aplikację do rozgrywki w grę karcianą „Tysiąc". Pełna wersja aplikacji została zaprogramowana w wieloplatformowym języku Java [15–17] przy użyciu graficznego interfejsu użytkownika bazującego na bibliotece Swing [18,19] oraz magazynu danych w postaci bazy danych SQLite [20,21].

2.1. Opis aplikacji

Gra obejmuje takie elementy jak: tasowanie kart, rozdawanie kart, licytację, zbieranie „musu", wymianę kart, podbicie stawki, rozgrywkę, meldowanie atutu, zbieranie lew. W rozgrywce uczestniczy trzech graczy. Zachowanie zawodników, polegające na wyborze algorytmu rozgrywki dla graczy komputerowych można określić w ustawieniach gry. Każdy gracz może być człowiekiem lub komputerem wyposażonym w sztuczną inteligencję opartą na jednym z pięciu algorytmów: losowego, minmax, opartego na sieci neuronowej, statystycznego oraz statystycznego z „oszukiwaniem". Rysunek 1 przedstawia interfejs, dzięki któremu możliwa jest zmiana ustawień gry.

Do celów badawczych powstały dwie, okrojone wersje aplikacji: do rozgrywki z człowiekiem oraz do rozgrywki wyłącznie między graczami komputerowymi. Obie wersje gry zawierają te same funkcjonalności oraz te same etapy gry. W celu dokonania pomiarów, a następnie porównań badawczych dokonano pewnych wstępnych założeń: w obu wersjach pozostawiono tylko etap rozgrywki, natomiast pozostałe etapy takie jak: licytacja, zbieranie „musu", wymiana kart oraz podbijanie stawki zostały usunięte. Dla wersji z rozegraniami z graczem ludzkim gracz pierwszy był sterowany przez człowieka, natomiast w drugiej wersji był sterowany kolejno poprzez graczy komputerowych dla wszystkich pięciu algorytmów rozgrywki. Cechy wspólne dwóch wersji to zachowanie graczy drugiego i trzeciego. Gracz drugi był sterowany przez kolejne zaimplementowane algorytmy rozgrywki. Gracz trzeci był graczem sterowanym przez sztuczną inteligencję na bazie algorytmu losowego. Zrzut ekranu z interfejsu rozgrywki przedstawiono na Rysunku 2.

Rysunek 1: Zrzut ekranu z menu opcji



Rysunek 2: Zrzut ekranu z rozgrywki

2.2. Zasady gry

W rozgrywce używa się talii 24 kart w czterech kolorach: pik, trefl, karo, kier przy wartościach: dziewiątka, dziesiątka, walet, dama, król, as. Każda karta posiada przyporządkowaną wartość punktową: dziewiątka - 0 pkt., walet - 2 pkt., dama - 3 pkt., król - 4 pkt., dziesiątka - 10 pkt., as - 11 pkt. Gracz rozpoczynający rundę, wyrzuca dowolną kartę. Kolejni gracze dorzucają karty w kolorze pierwszej karty. W przypadku braku

karty w podanym kolorze gracz może wyrzucić dowolną kartę. Gracz nie ma obowiązku przebijania wyrzuconych kart. Karty zbiera gracz, który posiada najwyższą punktowo kartę w kolorze pierwszej karty. Dodatkowo para dama i król w jednym kolorze może zostać zameldowana poprzez wyrzucenie jednej karty z tej pary, w przypadku, gdy gracz rozpoczyna rundę. Gracz, który melduje, otrzymuje dodatkowe punkty w zależności od koloru meldunku: pik - 40 pkt., trefl - 60 pkt., karo - 80 pkt., kier - 100 pkt. Ponadto kolor meldunku staje się atutem. W przypadku kolejnych meldunków stary atut traci ważność i jest zastępowany nowym kolorem meldunku. Atut koloru polega na możliwości przebicia kart w innych kolorach w przypadku, gdy gracz nie posiada już kart w kolorze pierwszej wyrzuconej karty [22, 23].

2.3. Algorytmy

W badanej grze zaimplementowano 5 algorytmów bazujących na różnych technikach tworzenia sztucznej inteligencji. Wszystkie zaimplementowane rozwiązania zachowują zgodność z przedstawionymi zasadami gry w „Tysiąc" zaprezentowanymi w poprzednim rozdziale.

2.3.1. Algorytm losowy

Jest to prosty algorytm, oparty na losowaniu karty z dostępnej puli kart. Pulę kart do wyrzucenia tworzy się, bazując na zasadach gry, w taki sposób, aby w puli znajdowały się tylko te karty, które mogą być wyrzucone w danym rozegraniu. Fragment kodu źródłowego algorytmu losowego przedstawiono na Listingu 1.

Listing 1: Fragment kodu źródłowego algorytmu losowego

```
@Override
public Card getPlayingCard(PlayableTable ↩
    table) {
    List<Card> playingCards = ↩
        ThousandRulesHelper.↩
        getPlayingCards(table);
    Card playingCard = playingCards.get(↩
        random.nextInt(playingCards.size↩
        ()));
    //...
    return playingCard;
}
```

2.3.2. Algorytm min-max

Algorytm ten poszukuje rozwiązania problemu w taki sposób, aby maksymalizować zyski oraz minimalizować straty. W przypadku rozpatrywanej gry karcianej jako zysk traktowana będzie liczba punktów uzyskana poprzez zbiórkę lew oraz meldowanie atutu. Natomiast w przypadku braku możliwości zebrania lewy algorytm będzie zmniejszał liczbę punktów, które gracz utraci poprzez przegranie rundy. Charakterystyczną cechą tego rozwiązania jest konieczność znajomości taktyki gry. Jest to duże ograniczenie, natomiast pozwala w prosty i stosunkowo szybki sposób osiągnąć obraz inteligentnych zachowań. Przy implementacji w pierwszym kroku sprawdzana jest możliwość meldunku atutu. Jest to spowodowane największą wartością punktową, jaką gracz może otrzymać w danych rozegraniu. Jeżeli meldunek jest możliwy, zostanie wykorzystany, w przeciwnym razie sprawdzeniu zostaje poddany warunek przebicia kart na stole rozgrywki. W przypadku możliwości przebicia zostanie rzucona odpowiednia kart. W przeciwnym przypadku zostanie wyrzucona karta powodująca najmniejszą stratę punktów. Fragment kodu źródłowego algorytmu min-max przedstawiono na Listingu 2.

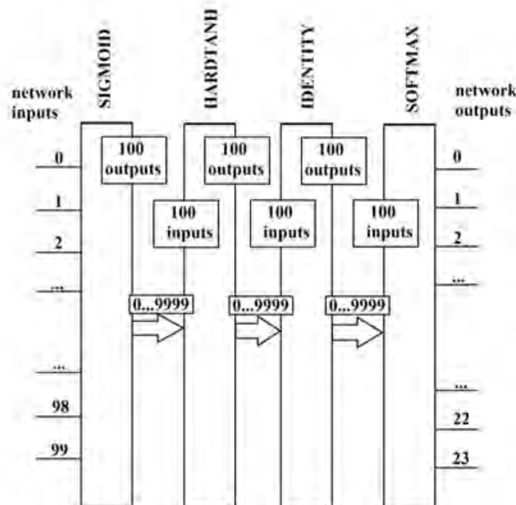
Listing 2: Fragment kodu źródłowego algorytmu min-max

```
@Override
public Card getPlayingCard(PlayableTable ↩
    table) {
    Card playingCard = null;
    List<Card> playingCards = ↩
        ThousandRulesHelper.↩
        getPlayingCards(table);

    if (table.getActualPlayerIndex() == ↩
        table.getActualWinnerIndex()) {
        int maxDispatchIndex = ↩
            getMaxDispatchIndex(↩
            playingCards);
        //...
    } else {
        int maxOvertrumpIndex = ↩
            getOvertrumpCardIndex(table, ↩
            playingCards);
        //...
    }
    //...
    return playingCard;
}
```

2.3.3. Algorytm bazujący na sieci neuronowej

Kolejnym zaimplementowanym rozwiązaniem jest algorytm bazujący na sieci neuronowej. Koncepcja ta oparta jest na wielowarstwowej sieci neuronowej ze wsteczną propagacją błędu. Trening sieci przeprowadzono na danych pozyskanych z wielokrotnych rozegrań algorytmu statystycznego ze zwiększoną liczbą analizowanych gier losowych. Utworzono dwa zbiory danych: uczący z milionem rekordów oraz walidacyjny ze stu tysiącami rekordów. Trening prowadzono do momentu, w którym błąd na zbiorze walidacyjnym zaczął się zwiększać. Struktura sieci została przedstawiona na Rysunku 3. Sieć zawiera 4 warstwy z różnymi funkcjami aktywacji. Pierwsza warstwa wejściowa posiada 100 wejść. Pierwsze 24 wejścia są odpowiedzialne za karty gracza używającego jednego z pięciu algorytmów, kolejne 24 wejścia są odpowiedzialne za kartę gracza poprzedniego, kolejne 24 wejścia są odpowiedzialne za kartę gracza następnego, kolejne 24 wejścia to zebrane karty na stole rozgrywki i ostatnie 4 wejścia są odpowiedzialne za kolor atutu. Sieć zawiera 24 wyjścia stanowiące prawdopodobieństwa wyrzucenia kart, które dadzą najlepsze wyniki dla algorytmu. Fragment kodu źródłowego algorytmu opar-



Rysunek 3: Struktura sieci neuronowej wykorzystanej w aplikacji

tego na sieci neuronowej przedstawiono na Listingu 3.

Listing 3: Fragment kodu źródłowego algorytmu opartego na sieci neuronowej

```
@Override
public Card getPlayingCard(PlayableTable ↩
    table) {
    List<Card> playingCards = ↩
        ThousandRulesHelper.↩
        getPlayingCards(table);

    double[][] featuresArray = new double↩
        [1][100];
    CardsTranslator.setCardsToVector(0, ↩
        featuresArray[0], table.↩
        getActualPlayer().getCards());
    //...
    INDArray features = Nd4j.create(↩
        featuresArray);
    INDArray predicted = network.output(↩
        features, false);
    double[] predictedArray = predicted.↩
        toDoubleVector();
    //...
}
```

2.3.4. Algorytm statystyczny

Algorytm bazuje na obliczeniach statystycznych, rozgrywa wielokrotne losowe rundy z kartami widocznymi w danej rundzie. Jest on kosztowny obliczeniowo ze względu na dużą liczbę rozegrań wykonywaną w celu podjęcia najlepszej decyzji o rzuceniu najbardziej optymalnej karty w danym rozegraniu. Duża zaletą algorytmu jest brak wiedzy programisty na temat taktyki używanej w grze. Przy każdym rozegraniu jest inicjalizowany stół będący kopią aktualnie rozgrywanego, z tą różnicą, że nieznane karty graczy są losowo rozdawane przeciwnikom. Następnie algorytm przeprowadza wielokrotne rozegrania całego rozdania z użyciem algorytmu losowego dla wszystkich graczy. Wyniki poszczególnych kart są sumowane i wybierany jest wynik dający największą liczbę punktów. Fragment kodu źródłowego algorytmu statystycznego przedstawiono na Listingu 4.

Listing 4: Fragment kodu źródłowego algorytmu statystycznego

```
@Override
public Card getPlayingCard(PlayableTable ↩
    table) {
    int sampleNumber = 100;
    //...
        oneRoundRandomTableContinuation.↩
            initializeWithoutCheating(↩
            table);
        oneRoundRandomTableContinuation.↩
            play(canPlayCards.get(j));
        sampleTable[j] += ↩
            oneRoundRandomTableContinuation↩
            .getPlayer(table.↩
            getActualPlayerIndex()).↩
            getScore();
    //...
}
```

2.3.5. Algorytm statystyczny z „oszukiwaniem"

Ostatnim rozpatrywanym algorytmem jest algorytm statystyczny z użyciem techniki „oszukiwania". Jest to technika szeroko stosowana w branży gier komputerowych w wielu jej wariantach. W celu zwiększenia wydajności lub skuteczności algorytmu stosuje się niedopuszczalne w normalnych warunkach metody, których działanie jest niezauważalne dla gracza sterowanego przez człowieka. W niniejszym przypadku zmodyfikowano poprzedni algorytm statystyczny w taki sposób, aby inicjalizacja losowego stołu była idealną kopią kart, które posiadają gracze. Dzięki temu posunięciu algorytm losowy przeprowadza symulację dla dokładnie takich kart, jakie posiadają przeciwnicy. Fragment kodu źródłowego algorytmu statystycznego z użyciem techniki „oszukiwania" przedstawiono na Listingu 5.

Listing 5: Fragment kodu źródłowego algorytmu statystycznego z użyciem techniki „oszukiwania"

```
@Override
public Card getPlayingCard(PlayableTable ↩
    table) {
    int sampleNumber = 100;
    //...
        oneRoundRandomTableContinuation.↩
            initializeWithCheating(table)↩
            ;
        oneRoundRandomTableContinuation.↩
            play(canPlayCards.get(j));
        sampleTable[j] += ↩
            oneRoundRandomTableContinuation↩
            .getPlayer(table.↩
            getActualPlayerIndex()).↩
            getScore();
    //...
}
```

Tabela 1: Średnia liczba punktów dla gracza 2

algorytm gracza 1	algorytm gracza 2				
	losowy	min-max	sieć neuronowa	statystyczny	statystyczny z „oszukiwaniem”
człowiek	44,582	64,626	70,708	76,688	83,282
losowy	52,878	77,843	84,890	85,581	87,407
min-max	43,773	68,916	74,522	75,400	76,423
sieć neuronowa	40,277	66,312	71,807	71,799	74,141
statystyczny	40,087	65,160	70,522	70,694	73,800
statystyczny z „oszukiwaniem”	38,760	64,346	70,052	70,307	72,764

3. Metodyka badań

Zostały przeprowadzone dwa eksperymenty:

- z graczem sterowanym przez człowieka
- z graczem sterowanym przez komputer

W pierwszym badaniu człowiek rozgrywał rundy z kolejnymi zaimplementowanymi algorytmami, gdzie rozegrano po 500 rozdań na każdy algorytm. W drugim badaniu przeprowadzono eksperyment krzyżowy dla wszystkich par algorytmów, gdzie rozegrano po 10000 rozdań na każdą parę algorytmów.

Dokonano pomiarów następujących parametrów:

- skuteczność
- czas wykonania
- liczba linii kodu
- czas implementacji
- czas trwania treningu.

Wszystkie parametry poddane zostały standaryzacji według wzoru:

$$P_{i_std} = \frac{P_i - P_{min}}{P_{max} - P_{min}} \quad (1)$$

gdzie P_{i_std} jest zestandaryzowaną wartością danego parametru i-tego algorytmu, P_i jest wartością danego parametru i-tego algorytmu, P_{min} i P_{max} są odpowiednio minimalną i maksymalną wartością danego parametru dla wszystkich algorytmów. W przypadku wielokrotnych pomiarów brano pod uwagę średnią arytmetyczną wartości parametrów.

3.1. Parametry

Przy pomiarze skuteczności przyjęto jako miarę liczbę punktów zdobytych przez gracza drugiego. Wskaźnik czasu wykonania został zmierzony dla każdego algorytmu po 40000 próbek na każdy algorytm. Czas mierzono przy użyciu funkcji System.nanoTime() języka Java. Do analizy liczby linii kodu źródłowego wzięto pod uwagę wyłącznie kod stworzony w czasie implementacji algorytmu. Podczas zliczania nie brano pod uwagę pustych linii oraz komentarzy. Ponadto uwzględniono wyłącznie kod badanego algorytmu. Każdy algorytm był rozpatrywany niezależnie. Drugim parametrem powiązanym z wytworzeniem rozwiązania był czas implementacji. Każdy algorytm był programowany przez tego samego programistę. Dokonano pomiaru czasu implementacji wyłącznie tej części implementacji, która była odpowiedzialna za dany algorytm. Nie uwzględniano oprogramowania innych części aplikacji. Ostatnim rozpatrywanym parametrem był koszt związany z czasem trwania treningu rozwiązań opartych na poszczególnych algorytmach. Ten parametr dotyczy tylko algorytmu bazującego na sieci neuronowej i był równy 45 godzinom dla sieci przedstawionej we wcześniejszych rozdziałach. Trening był prowadzony na zbiorze uczącym, liczącym milion rekordów. Trening prowadzono do momentu, aż błąd na zbiorze walidacyjnym zaczął rosnąć. Zbiór walidacyjny obejmował 100 tysięcy rekordów. Dobór liczebności zbiorów uczącego i walidacyjnego był uwarunkowany wykorzystanym do obliczeń sprzętem komputerowym. Dane do treningu zostały wygenerowane przez algorytm statystyczny ze zwiększoną liczbą rozegrań [24, 25].

3.2. Środowisko testowe i narzędzia badawcze

Wszystkie czynności zostały przeprowadzone na komputerze Dell Inspiron 3543 z procesorem Intel Core i7-5500U, wyposażonym w 16GB pamięci RAM oraz zainstalowanym systemem operacyjnym Windows 7 Professional 64-bit. Do pomiaru liczby linii kodu źródłowego został wykorzystany program Cloc [26]. Analiza statystyczna, omówiona w rozdziale 3.3.1 została wykonana za pomocą dwóch aplikacji: Matlab R2019b oraz Statistica 13 64-bit.

3.3. Wyniki

3.3.1. Skuteczność

Wyniki pomiarów przedstawiono w Tabeli 1. Można tam zauważyć różne skuteczności zaimplementowanych rozwiązań. Ponadto wyniki można uszeregować według średnich od najsłabszego do najsilniejszego: losowy, min-max, sieć neuronowa, statystyczny, statystyczny z „oszukiwaniem”.W celu analizy statystycznej otrzymanych wyników pod kątem istotności różnic pomiędzy algorytmami sprawdzono spełnienie warunku o normalności rozkładów badanych danych dla testu jednoczynnikowej analizy wariancji. Normalność rozkładów rozpatrywanych danych sprawdzono testem Kołmogorowa-Smirnowa przy poziomie istotności 0,05. Niestety założenia o normalności nie zostały spełnione i w każdym przypadku wartość istotności wynosiła $p<0{,}01$. Z tego względu istotność różnic pomiędzy algorytmami sprawdzono nieparametrycznym testem rang Kruskala-Wallisa na poziomie istotności 0,05. Wyniki testów przedstawiono w Tabeli 3. Na podstawie testu

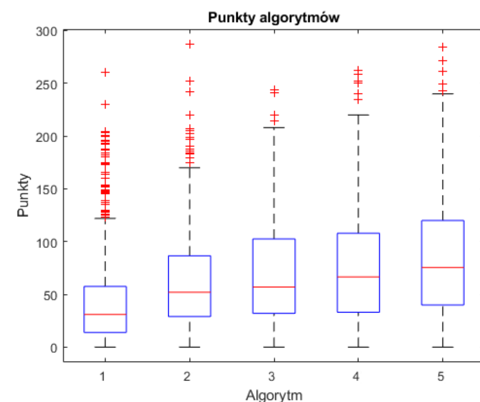
Tabela 2: Wyniki porównań wielokrotnych testem Dunna dla algorytmów wykazujących różnice skuteczności

algorytm I	algorytm II	algorytm gracza 1					
		człowiek	losowy	min-max	sieć neuronowa	statystyczny	statystyczny z „oszukiwaniem”
losowy	min-max	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
losowy	sieć neuronowa	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
losowy	statystyczny	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
losowy	statystyczny z „oszukiwaniem”	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
min-max	sieć neuronowa	p=0,4588	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
min-max	statystyczny	p=0,0053	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
min-max	statystyczny z „oszukiwaniem”	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001	p<0,0001
sieć neuronowa	statystyczny	p=1,0000	p=1,0000	p=1,0000	p=1,0000	p=1,0000	p=1,0000
sieć neuronowa	statystyczny z „oszukiwaniem”	p=0,0077	p=0,0026	p=0,0667	p=0,0099	p<0,0001	p=0,0016
statystyczny	statystyczny z „oszukiwaniem”	p=0,5797	p=0,0749	p=1,0000	p=0,0029	p<0,0001	p=0,0193

Tabela 3: Wyniki testów Kruskala-Wallisa

algorytm gracza 1	p-wartość
człowiek	p < 0,0001
losowy	p < 0,0001
min-max	p < 0,0001
sieć neuronowa	p < 0,0001
statystyczny	p < 0,0001
statystyczny z „oszukiwaniem”	p < 0,0001

można zauważyć istotne statystycznie różnice w każdej z badanych grup ($p < 0{,}0001$). Ponadto wykonano testy Dunna na poziomie istotności 0,05 w celu sprawdzenia, które podgrupy danych wykazują różnice. Wyniki testów Dunna dla rozpatrywanych algorytmów przedstawiono w Tabeli 2. W grupach gdzie p-wartość była na poziomie poniżej 0,05, różnice były istotne statystycznie. Dla pary algorytmów statystycznego oraz bazującego na sieci neuronowej widoczny był brak różnic we wszystkich badanych grupach. Pozostałe pary w większości wypadków wykazują pomiędzy sobą różnice istotne statystycznie. Wykres pudełkowy przedstawiający rozkład liczby punktów gracza drugiego dla grupy z graczem ludzkim przedstawiono na Rysunku 4.



Rysunek 4: Wykres pudełkowy rozkładu liczby punktów gracza drugiego dla grupy z graczem ludzkim oraz algorytmów: losowego(1), min-max(2), sieci neuronowej(3), statystycznego(4) oraz statystycznego z „oszukiwaniem”(5)

3.3.2. Czas wykonania

Drugim badanym parametrem był czas wykonania algorytmu powiązany z jego złożonością obliczeniową. Wyniki pomiarów przedstawiono w Tabeli 4. W celu analizy różnic pomiędzy czasami sprawdzono spełnienie założeń o normalności rozkładów testem Kołmogorowa-Smirnowa przy poziomie istotności 0,05. Niestety założenie o normalności nie zostało spełnione i w każdym przypadku wartość istotności wynosiła $p<0{,}01$. Z tego względu istotność różnic pomiędzy czasami wykonania algorytmów sprawdzono testem rang Kruskala-Wallisa przy poziomie istotności 0,05. P-wartość wyniosła $p<0{,}0001$, przez co różnice w czasach są istotne statystycznie. W celu sprawdzenia, które pary różnią się, wykonano testy Dunna przy poziomie istotności 0,05. Wyniki testów dla porównań wielokrotnych przedstawiono w Tabeli 5. Można zauważyć, że każda para algorytmów posiada istotne statystycznie różnice czasów wykonania. Można je uszeregować od najszybszego do najwolniejszego: losowy, min-max, sieć neuronowa,

Tabela 4: Średni czas wykonania algorytmów

algorytm	średni czas wykonania [ms]
losowy	0,014
min-max	0,064
sieć neuronowa	4,632
statystyczny	22,518
statystyczny z „oszukiwaniem”	18,138

Tabela 5: Wyniki porównań wielokrotnych czasów wykonania testem Dunna

algorytm I	algorytm II	p-wartość
losowy	min-max	$p < 0,0001$
losowy	sieć neuronowa	$p < 0,0001$
losowy	statystyczny	$p < 0,0001$
losowy	statystyczny z „oszukiwaniem”	$p < 0,0001$
min-max	sieć neuronowa	$p < 0,0001$
min-max	statystyczny	$p < 0,0001$
min-max	statystyczny z „oszukiwaniem”	$p < 0,0001$
sieć neuronowa	statystyczny	$p < 0,0001$
sieć neuronowa	statystyczny z „oszukiwaniem”	$p < 0,0001$
statystyczny	statystyczny z „oszukiwaniem”	$p < 0,0001$

statystyczny z „oszukiwaniem”, statystyczny. Wykorzystanie techniki „oszukiwania” spowodowało zmniejszenie się czasu wykonania algorytmu, w porównaniu ze standardowym algorytmem statystycznym.

3.3.3. Liczba linii kodu

Kolejnym badanym parametrem była właściwość związana z wytworzeniem poszczególnego rozwiązania. Wyniki pomiarów przedstawiono w Tabeli 6. Najkrótszą implementację pod względem liczby linii kodu miał algorytm losowy. Przy algorytmie bazującym na sieci neuronowej sama implementacja sieci była niewielka, natomiast dodatkowy kod związany z wytrenowaniem sieci był bardzo obszerny. W tym przypadku może istnieć możliwość użycia pre-trenowanych rozwiązań, które zmniejszyłyby koszt treningu sieci. Przy porównaniu algorytmu statystycznego z użyciem techniki „oszukiwania” oraz bez jej użycia liczba linii kodu była na podobnym poziomie.

Tabela 6: Liczba linii kodu dla algorytmów

algorytm	liczba linii kodu
losowy	54
min-max	226
sieć neuronowa rdzeń	163
sieć neuronowa trening	1167
statystyczny	384
statystyczny z „oszukiwaniem”	367

3.3.4. Czas implementacji

Drugim parametrem związanym z wytworzeniem rozwiązania był czas jego implementacji. Wyniki czasów implementacji są proporcjonalne do liczby linii kodu dla poszczególnych algorytmów (Tabela 7). Najszybszy w implementacji był algorytm losowy. W tym przypadku również należałoby się zastanowić nad koniecznością treningu sieci neuronowej i wykorzystaniem pretrenowanego rozwiązania ze względu na duży narzut na trening. Oba rozwiązania statystyczne mają identyczny czas implementacji.

Tabela 7: Czas implementacji algorytmów

algorytm	czas implementacji [godziny]
losowy	0,25
min-max	2,33
sieć neuronowa rdzeń	4,17
sieć neuronowa trening	25,92
statystyczny	16,50
statystyczny z „oszukiwaniem”	16,50

3.3.5. Ocena końcowa algorytmów

W niniejszym podrozdziale przedstawiono oceny poszczególnych algorytmów. Tabela 8 przedstawia uzyskane wyniki badanych algorytmów dla poszczególnych kryteriów. W przypadku parametru skuteczności można zauważyć dużą różnicę pomiędzy algorytmem losowym, a pozostałymi algorytmami. Dla pozostałych parametrów algorytm losowy i min-max uzyskują podobne wartości. Algorytm bazujący na sieci neuronowej w większości przypadków ma najgorsze parametry. Rozwiązania statystyczne cechują się dużym podobieństwem wskaźników.

4. Wnioski i podsumowanie

Badania różnych metod tworzenia sztucznej inteligencji na przykładzie gry w „Tysiąc” pozwoliły scharakteryzować techniki wykorzystywane do tworzenia inteligencji w grach komputerowych. W pracy zbadano takie parametry jak: skuteczność inteligencji, czas wykonania, liczbę linii kodu, czas implementacji oraz czas trwania treningu rozwiązań.

Badania wskazały istotne różnice we wziętych pod uwagę w badaniach kryteriach oceny algorytmów. Ponadto wykorzystanie technik „oszukiwania” pozwala zwiększyć skuteczność bez pogorszenia pozostałych parametrów algorytmu. Najwyższą skuteczność wykazują algorytmy bazujące na obliczeniach statystycznych, ale charakteryzują się one najgorszymi czasami wykonania. Dobrą alternatywą dla obliczeń statystycznych może być algorytm bazujący na sieci neuronowej, gdzie przy niewielkiej utracie skuteczności zyskuje się wysoki zysk w czasie wykonania. Najgorszą skuteczność wykazuje algorytm losowy, choć był on wykorzystywany w obu algorytmach statystycznych. Na uwagę zasługuje rozwiązanie min-max, w którym przy stosunkowo dużej skuteczności uzyskano bardzo dobre pozostałe parametry. W przypadku lepszego dopracowania tej metody mogłaby ona zyskać jeszcze większą skuteczność. Niestety algorytm min-max wymaga od programisty wiedzy na temat taktyki gry. Może to zniechęcić programistów w przypadku bardziej złożonych problemów.

W przypadku parametru objętości kodu źródłowego i powiązanym z nim czasem implementacji najgorszym

Tabela 8: Zbiorcze oceny parametrów dla badanych algorytmów

parametr	algorytm				
	losowy	min-max	sieć neuronowa	statystyczny	statystyczny z „oszukiwaniem"
skuteczność	0,00	0,70	0,88	0,92	1,00
czas wykonania	0,00	0,00	0,21	1,00	0,81
liczba linii kodu	0,00	0,13	1,00	0,26	0,25
czas implementacji	0,00	0,08	1,00	0,63	0,63
czas treningu	0,00	0,00	1,00	0,00	0,00

rozwiązaniem była sieć neuronowa. Spowodowane było to dużym narzutem na wytworzenie sieci i napisanie oprogramowania do treningu. Jednocześnie sieć posiada skuteczność zbliżoną do algorytmu statystycznego w przypadku średniej liczby punktów. Badania pokazały brak istotnych statystycznie różnic w skuteczności algorytmu bazującego na sieci neuronowej i algorytmu statystycznego. Z tego względu algorytm statystyczny można zastąpić algorytmem opartym na sieci neuronowej i zaoszczędzić zasoby obliczeniowe. Dodatkowo w przypadku bardziej popularnych zastosowań istnieje możliwość wykorzystania pre-trenowanej sieci, co może skutkować rezygnacją z czasochłonnego treningu.

Literatura

[1] D. S. Cohen, T.J. Park, S.A Bustamante, Producing Games: From Business and Budgets to Creativity and Design, Routledge, 2010.

[2] T. Walsh, To żyje! Sztuczna inteligencja, Wydawnictwo Naukowe PWN, 2018.

[3] K. Wołk, Zabawa ze sztuczną inteligencją, Psychoskok, 2018.

[4] S. Nowaczyk, Frontiers in Artificial Intelligence and Applications, Thirteenth Scandinavian Conference on Artificial Intelligence, Tom 278, Holandia, 2015.

[5] T. Burczyński, W. Cholewa, W. Moczulski, Methods of artificial intelligence, Silesian University of Technology, 2009.

[6] S. Castellano, Artificial Intelligence, Genuine Learning, Talent Development, Tom 73, Wydanie 10, 2019.

[7] J. Łęski, Systemy neuronowo-rozmyte, Wydawnictwo WNT, 2008.

[8] L. Rutkowski, Metody i techniki sztucznej inteligencji, Wydawnictwo Naukowe PWN, 2009.

[9] Z. Shi, Advanced Artificial Intelligence, World Scientific, Singapur, 2011.

[10] B. Zohuri, M. Moghaddam, Neural Network Driven Artificial Intelligence : Decision Making Based on Fuzzy Logic. Nova Science Publisher, Nowy Jork, 2017.

[11] N. Joshi, Hands-On Artificial Intelligence with Java for Beginners : Build Intelligent Apps Using Machine Learning and Deep Learning with Deeplearning4j, Packt Publishing, 2018.

[12] Przegląd branży gier wideo, https://www.wepc.com/news/video-game-statistics, [13.07.2020].

[13] G. N. Yannakakis, J. Togelius, Artificial intelligence and games, Cham: Springer, 2018.

[14] G. N. Yannakakis, J. Togelius, A Panorama of Artificial and Computational Intelligence in Games, IEEE Transactions on Computational Intelligence and AI in Games, 2015.

[15] D. Brackeen, Java : tworzenie gier, Wydawnictwo Helion, Gliwice, 2004.

[16] J. S. Harbour, Beginning Java SE 6 Game Programming, MA: Course PTR, Boston, 2012.

[17] B. Kaluza, Machine Learning in Java, Packt Publishing, 2016.

[18] W. McAllister, J. Fritz, Programming Fundamentals Using Java : A Game Application Approach, Dulles, Virginia: Mercury Learning & Information, 2015.

[19] S. Pięta, M. Ścibisz, M. Wiśniewski, Podstawy tworzenia interfejsu graficznego aplikacji desktopowych w języku Java, Oficyna Wydawnicza Politechniki Warszawskiej, 2019.

[20] S. K. Aditya, P. Mohanta, V. K. Karn, Android SQLite Essentials, Packt Publishing, 2014.

[21] R. A. Johnson, Java Database Connectivity Using Sqlite: A Tutorial, Java Database Connectivity Using Sqlite: A Tutorial, Proceedings of the Academy of Information & Management Sciences, 2014.

[22] B. Rigal, S. Kupisz, Gry karciane, Helion, 2007.

[23] B. Whiter, J. Kluziński, Gry karciane, K.E. Liber 2004.

[24] T. B. Alakus, R. Das, I. Turkoglu, An Overview of Quality Metrics Used in Estimating Software Faults, International Artificial Intelligence and Data Processing Symposium, 2019.

[25] L. Buglione, A. Abran, Measurement Process: Improving the ISO 15939 Standard, Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, 2014.

[26] Dokumentacja oprogramowania CLOC, https://github.com/AlDanial/cloc, [13.07.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 241–246
Received: 7 June 2020
Accepted: 24 June 2020

Comparative analysis of Kotlin coroutines with Java and Scala in parallel programming

Analiza porównawcza współprogramów języka Kotlin z językami Java i Scala w przetwarzaniu równoległym

Adrian Andrzej Zieliński*

[a] *Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland*

Abstract

The article presents a comparison of Kotlin coroutines with analogous solutions in Java and Scala in parallel programming using chosen metric and non-metric criteria. For that purpose, a multi-module project with corresponding implementations of selected algorithms in all of the three languages was created and then analyzed. The studies were preceded by a description of the created project.

Keywords: kotlin; java; scala; parallel programming

Streszczenie

Artykuł prezentuje porównanie wykorzystania współprogramów języka Kotlin w przetwarzaniu równoległym do analogicznych rozwiązań w Javie i Scali względem wybranych kryteriów mierzalnych i niemierzalnych. W tym celu stworzono oraz przeanalizowano wielomodułową aplikację z odpowiadającymi sobie implementacjami wyselekcjonowanych algorytmów w trzech wspomnianych językach. Analiza poprzedzona została opisem utworzonego projektu.

Słowa kluczowe: kotlin; java; scala; programowanie równoległe

*Corresponding author

Email address: **adrian.zielinski@pollub.edu.pl** (A. A. Zieliński)

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

1. Wstęp

Pomimo leciwości konceptu programowania równoległego, który powstał jeszcze w XIX w., to jego realne zastosowanie w informatyce datuje się dopiero na drugą połowę wieku XX za sprawą popularyzacji procesorów wielordzeniowych [1]. Obecnie nastąpiło już w tej materii takie przesycenie, że koncept ten może być wcielany w życie na wyższych poziomach abstrakcji dzięki mnogości dostępnych platform programistycznych.

Jedną z najciekawszych cech Kotlina, która jest jednocześnie przedmiotem analizy w niniejszym artykule, jest biblioteka kotlinx.coroutines, która została włączona do stabilnego wydania języka wraz z wersją 1.3. Oferuje ona funkcjonalność współprogramów zaproponowanych jeszcze w latach 60 XX wieku przez Conwaya [2] na poziomie języka, z możliwościami równoległego wykonania, podczas gdy Java i Scala operują na klasycznych konceptach wątków czy nieco nowszych obietnic.

Jako, że Kotlin nie jest pierwszym, ani nawet najpopularniejszym językiem oferującym omawianą funkcjonalność, zasadne staje się pytanie jak jego nowe podejście do współprogramów wypada na tle technologii, które zdążyły się już zadomowić na rynku. Niniejsza praca stara się odpowiedzieć na to pytanie poprzez bezpośrednie porównanie implementacji współprogramów w Kotlinie oraz technik przetwarzania równoległego w Javie i Scali, jako, że wszystkie z wymienionych języków mogą działać pod wirtualną maszyną Javy GraalVM.

2. Przegląd literatury

Przez ostatnie kilka lat, tematyka współprogramów przeżywa swój renesans. Coraz bardziej skomplikowane programy oraz rozwój sprzętowy w kierunku przetwarzania wielordzeniowego i wielowątkowego wymusza niejako poszukiwanie coraz to wydajniejszych, bardziej rozbudowanych czy łatwiejszych w użyciu technik przetwarzania asynchronicznego i równoległego.

W 2009 roku De Moura i Ierusalimschy przekonywali, że przywrócenie współprogramów do łask po wielu latach zapomnienia odbyłoby się z korzyścią dla wszystkich. W swojej pracy przedstawili klasyfikację podejść do implementacji tego paradygmatu, argumenty przemawiające za bądź przeciw danemu podejściu oraz wprowadzili nową koncepcję współprogramów w pełni asymetrycznych, która łączy w sobie cechy poprzedników [3].

Racordon w swojej pracy porusza natomiast problem braku możliwości programowania z wykorzystaniem paradygmatu współprogramów w wielu współczesnych językach programowania i proponuje uniwersalny model implementacyjny tychże na bazie funkcji wyższego rzędu w językach je posiadających, prezentując przykłady w języku JavaScript [4].

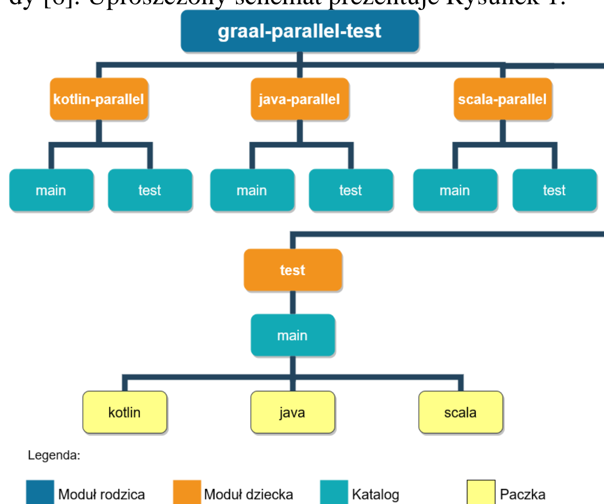
Z kolei Ohlsson i Leffler zwracają uwagę na możliwość implementacji współprogramów w Javie poprzez formę rozszerzenia językowego. Autorzy słusznie zauważają, że istnieją problemy, które naturalniej jest rozwiązać wykorzystując opisywany paradygmat i na przykładzie własnego kodu napisanego w standardowej

Javie pokazują, że przemodelowanie go na wykorzystywanie wcześniej utworzonego rozszerzenia współprogramów pozwala na uzyskanie większej zwięzłości i czytelności [5]. Przytoczone powyżej prace ukazują zwiększone zainteresowanie tematyką współprogramów na przestrzeni poprzednich lat wśród badaczy z dziedziny programowania. Jednocześnie, bezpośrednie porównanie współprogramów z klasycznymi technikami przetwarzania równoległego nie doczekało się obszerniejszej analizy w obrębie maszyny wirtualnej Javy. Niniejszy artykuł rzuca na ten temat nowe spojrzenie, mogące pomóc w wyborze odpowiedniej do danego problemu technologii.

3. Projekt testowy

3.1. Opis

Stworzona z wykorzystaniem systemu budującego i zarządzającego zależnościami Maven aplikacja podzielona została na cztery główne moduły – kotlin-parallel, java-parallel, scala-parallel oraz test. Pierwsze trzy składają się z logiki zdefiniowanej kolejno w Kotlinie, Javie oraz Scali i zawierają możliwie najbardziej zbliżone sobie implementacje trzech wybranych, zrównoleglonych algorytmów – znajdowania liczb pierwszych, wyznaczania otoczki wypukłej punktów w przestrzeni dwuwymiarowej oraz wyliczania transformaty Fouriera. Dodatkowo wytworzono zestaw testów jednostkowych sprawdzających poprawność logiczną. Ostatnia część o nazwie test zawiera odpowiednio podzieloną grupę testów wydajnościowych dla wszystkich badanych implementacji. Do celów jej zmierzenia wykorzystano bibliotekę Java Microbenchmark Harness autorstwa Oracle'a. JMH jest narzędziem przeznaczonym dla platformy Java do wykonywania powtarzalnych i miarodajnych mikro testów wydajnościowych, tj. takich, których zadaniem jest pomiar możliwie najmniejszej części systemu – najlepiej pojedynczej metody [6]. Uproszczony schemat prezentuje Rysunek 1.



Rysunek 1: Uproszczony schemat projektu

3.2. Wykorzystane algorytmy

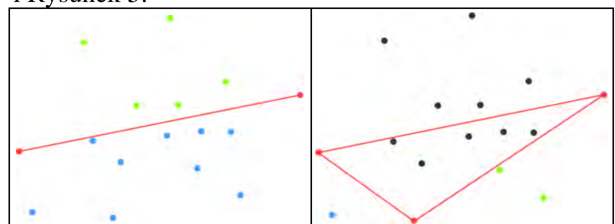
W celu oddania możliwie jak najszerszego zakresu wykorzystania obliczeń równoległych, wybrane zostały trzy algorytmy oddające różne przypadki czy scenariusze testowe. Pierwszym, a zarazem logicznie najprostszym algorytmem jest Sito Eratostenesa służące do wyznaczania liczb pierwszych w przedziale [7]. Blokowa wersja Sita idealnie nadaje się do równoległego wykonywania niemal wszystkich bloków liczb z zakresu i tym samym oddaje pierwszy zakładany przypadek użycia równoległości. Poniższy Rysunek 2. przedstawia zasadę działania dla dwóch pierwszych kroków Sita.

	2	3	4	5	6	7	8	9	10	Prime
11	12	13	14	15	16	17	18	19	20	2
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	

	2	3	4	5	6	7	8	9	10	Prime
11	12	13	14	15	16	17	18	19	20	2 3
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	

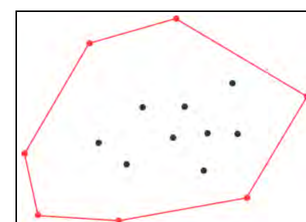
Rysunek 2: Pierwsze dwa kroki podstawowego Sita [8]

Jako drugi algorytm wybrano Quickhull służący do wyznaczania otoczki wypukłej skończonej liczby punktów w przestrzeni. Ze względu na swoją rekursywną naturę algorytm przetwarzany jest w formie drzewa, które rozwiązywane jest od liści do korzenia [9]. Dzięki temu idealnie wpasowuje się w przypadek dużej liczby małych, współbieżnych zadań, które wstrzymywane są na czas wykonywania swoich zależności i następnie wznawiane w celu scalenia wyników. Kroki działania algorytmu pokazują poniższe Rysunek 3., Rysunek 4. i Rysunek 5.



Rysunek 3: Wyznaczenie dwóch zbiorów początkowych [10]

Rysunek 4: Znalezienie punktu maksymalnego w jednym ze zbiorów [11]



Rysunek 5: Wynik rekursji w momencie braku kolejnych zewnętrznych punktów [12]

Ostatnim z wybranych przypadków testowych jest algorytm wyznaczania widma do analizy sygnału audio w nieskompresowanym formacie wav z wykorzystaniem szybkiej transformacji Fouriera (FFT). Rozpatry-

wany problem jest o tyle różny od dwóch pozostałych, że angażuje dodatkowo blokujące zadanie odczytu z pliku, przez co wszystkie zadania, mogące być teoretycznie wykonywane równolegle, są częściowo ograniczone przez możliwości dostarczania kolejnych partii przez strumień danych.

4. Kryteria oceny

4.1. Wydajność

Aspekt wydajnościowy jest po prawdzie jednym z łatwiej mierzalnych w środowisku JVM ze względu na dostępność biblioteki JMH od Oracle'a. Pozwala ona bowiem tworzyć powtarzalne testy w tak trudnym środowisku jak maszyna wirtualna stosująca optymalizacje kompilowanego na bieżąco kodu i to na wielu poziomach, zaczynając od wstępnej optymalizacji podczas pierwszego wykonania.

Wszystkie benchmarki przypadków testowych oparte zostały o wcześniej przedstawioną bibliotekę JMH i zawarte zostały w module test, który przedstawiony jest na Rysunek 1.

Każdy przypadek testowy wykonany zaś został 10 razy z jednakowymi ustawieniami adnotacji na klasach testowych.

4.2. Długość kodu

W przypadku tytułowej metryki postanowiono przyjąć pewne założenia pozwalające na dokonanie miarodajnego porównania. Ocenie podlegała całościowa długość napisanego dla danej implementacji kodu w liniach, przy czym przyjęto, że mniejsza liczba linii oznacza lepszy, tj. czytelniejszy kod. Założenie to wynika głównie z niechęci programistów do technologii wymuszających na nich pisanie tzw. boilerplate code czyli w wolnym tłumaczeniu kodu potrzebnego do rozwiązania problemu algorytmicznego w danym języku, który jednak mógłby być całkowicie zbędny w innych warunkach, np. w innym, bardziej zwięzłym języku. Zaowocowało to nawet, nie licząc mnogości technologii promujących zwięzły kod, poszukiwaniem automatyzowanych rozwiązań tego problemu [13].

Przy formatowaniu źródeł utworzonych programów postawiono na jednolite zasady. Długość linii nie większą niż 120 znaków, z tolerancją ± 10 oraz klamry dla wszystkich instrukcji ich wymagających, np. if, while, for czy definicji metod i funkcji. Pozostałe ustawienia miały domyślne wartości wykorzystanego środowiska programistycznego IntelliJ IDEA 2019.3.4.

4.3. Czas kompilacji

Ważnym czynnikiem przy doborze technologii jest również całkowity czas kompilacji projektu, którego wahania mogą wpływać na wiele kwestii począwszy od produktywności programisty a skończywszy na szybkości integracji rozwiązania np. przy continous integration.

Zmierzenie czasu kompilacji poszczególnych części badanej aplikacji stało się możliwe dzięki wykorzystanemu narzędziu zarządzającemu Maven. Badanie objęło wykonanie 10 prób budowania całego projektu oraz wyciągnięcie średniej.

4.4. Objętość API

Kwestią niewątpliwie istotną z punktu widzenia użytkownika jest objętość czyli w gruncie rzeczy możliwości API (ang. application programming interface), z którym przyjdzie mu pracować. Nie jest tajemnicą, że mniej obszerne możliwości danej technologii wymuszają niekiedy na programiście wykonywanie dodatkowej pracy w celu rozwiązania problemu, który jest tak ogólny, powszechny, a czasem nawet trywialny, że logicznym byłoby umieszczenie odpowiedniego API już w samej bibliotece standardowej co się niestety nie zawsze zdarza.

W celu ocenienia obszerności poszczególnych API zdecydowano się na metodę oceny ilościowej, tj. porównano bezpośrednio liczbę bytów klasopodobnych oraz metod, funkcji i pól w nich zawartych. Byt klasopodobny rozumiany jest tutaj jako każda definicja abstrakcyjnego typu lub kontenera jak klasy, interfejsy, cechy (ang. traits), obiekty (singletonowe definicje w Kotlinie i Scali) czy wreszcie paczki mogące być kontenerami na funkcje niebędące przypisanymi do konkretnych klas (funkcje najwyższego poziomu w Kotlinie bądź obiekty paczkowe w Scali). Dodatkowo, wszelkie elementy definiujące wewnętrzną strukturę bytów klasopodobnych, tak jak wcześniej wymienione metody, czy konstruktory, nazwano dalej zbiorczo własnościami.

5. Badania

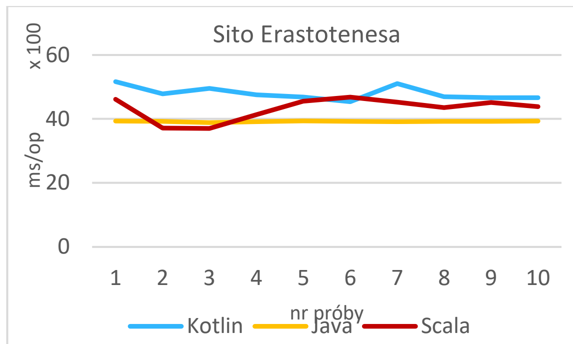
5.1. Wydajność

Wykonany na wszystkich dostępnych implementacjach algorytmu Sita Eratostenesa test nie stwierdził znaczącej różnicy w szybkości wykonania równoległego kodu w implementacjach w poszczególnych językach. Średnia różnica pomiędzy Kotlinem a Javą zamyka się w ok. 18% na niekorzyść tego pierwszego zaś Scala wypadła o ok. 10% gorzej od Javy. Klasę testową dla Kotlina prezentuje Listing 1. zaś szczegóły wszystkich kolejnych prób Rysunek 6.

Listing 1: Fragment klasy testującej znajdowanie liczb pierwszych w Kotlinie

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 5, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 15, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class SievePrimesBenchmark {

   @Benchmark
   fun findPrimesParallel(blackhole: Blackhole) {
       val primes: List<Int> = runBlock-
ing(Dispatchers.Default) { findPrime-
Numbers(2_100_000_000) }
       blackhole.consume(primes)
   }
}
```



Rysunek 6: Szybkość wykonania metod znajdujących liczby pierwsze w kolejnych próbach

Przed przejściem do następnych testów należy doprecyzować dane, jakie przedstawiają wykresy wydajnościowe. Otóż zarówno powyższy z Rysunek 6, jak i późniejsze, prezentują na skali wartości w formie mikrosekund na jedną operację (ms/op), przy czym operacja definiowana jest jako podstawowa jednostka działania w JMH, oznaczając domyślnie jedno wykonanie metody z adnotacją @Benchmark. Istnieją oczywiście opcje konfiguracyjne pozwalające na zmianę tego zachowania, jednak na potrzeby niniejszego artykułu pozostawiono konfigurację domyślną.

W kolejnym teście bazującym już na algorytmie znajdowania otoczki wypukłej sytuacja jest zgoła odmienna. Implementacja w kotlinowych współprogramach wypadła o ok. 13% lepiej od analogicznej logiki w Javie. Bardzo znaczące jest przy tym, że w wypadku programu javowego zaprezentowany wynik udało się uzyskać dopiero po wielogodzinnych próbach optymalizacyjnych gdyż pierwotny rezultat okazał się być niemal 1,5 razy wolniejszy od implementacji w Kotlinie. Najgorzej poradziła sobie Scala, która ostatecznie oferowała wykonanie o 42% wolniejsze od zwycięzcy próby. Całość wyników prezentuje Rysunek 7., zaś klasę testową Listing 2.

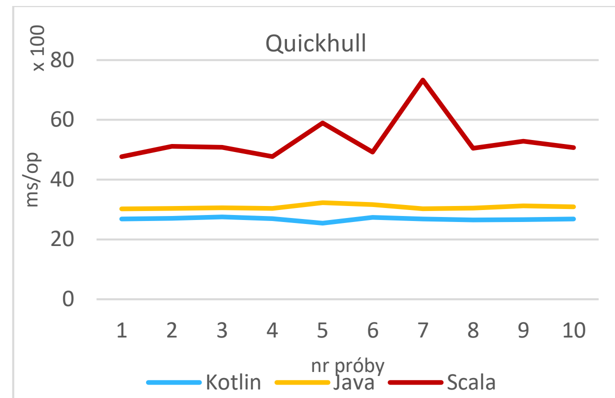
Listing 2: Fragment klasy testująca znajdowanie otoczki wypukłej w Kotlinie.

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 6, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 20, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class QuickHullBenchmark {

   @State(Scope.Benchmark)
   open class Data {
       // (...) inicjalizacja danych testowych
   }

   @Benchmark
   fun findConvexHullParallel(data: Data, black-
hole: Blackhole) {
       val quickHull: QuickHull = Quick-
Hull(data.points)
       val points: List<Point> = runBlock-
ing(Dispatchers.Default) { quickHull.find() }

       blackhole.consume(points)
   }
}
```



Rysunek 7: Szybkość wykonania metod znajdujących otoczkę wypukłą w kolejnych próbach

W ostatnim przypadku testowym szybkiej transformacji Fouriera wyniki prezentują się nadspodziewanie równo. Świadczy o tym niskie odchylenie standardowe, które dla żadnego z badanych języków nie przekroczyło 2500. Spowodowane może to być charakterystyką stworzonego algorytmu, która po części bazuje na blokującym zadaniu odczytywania kolejnych porcji danych z pliku. Przedstawiony na Rysunek 8. wykres pokazuje szczegółowe wyniki w kolejnych próbach. W kwestii wydajności po raz drugi najlepszy okazał się Kotlin z wynikiem nieco ponad 5% lepszym od Javy zaś zdecydowanie najgorzej poradziła sobie implementacja w Scali, która była średnio o 38% wolniejsza od zwycięzcy. Jedną z możliwych przyczyn takiego stanu rzeczy mogą być np. różnice w działaniu podstawowych instrukcji języka, które w pewnych okolicznościach mogą nie zapewniać optymalnej wydajności [14, 15]. Poniższy Listing 3. prezentuje klasę testującą powyższy przypadek w Kotlinie, zaś Rysunek 8. wyniki w kolejnych próbach.

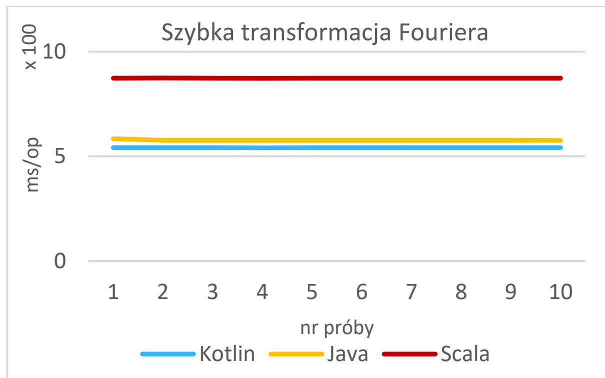
Listing 3: Fragment klasy testującej liczenie FFT.

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 6, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 20, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class WavFileFftBenchmark {

   @State(Scope.Benchmark)
   open class Data {
       // (...) inicjalizacja danych testowych
   }

   @Benchmark
   fun calculateFftParallel(data: Data, blackhole:
Blackhole) {
       val fftInDecibeles: List<DoubleArray> =
runBlocking(Dispatchers.Default) { trans-
form(data.files) }

       blackhole.consume(fftInDecibeles)
   }
}
```



Rysunek 8: Szybkość wykonania metod liczących FFT w kolejnych próbach

5.2. Długość kodu

Kolejną zmierzoną metryką jest długość kodu. Przy pomiarze kolejnych implementacji wzięto pod uwagę wszystkie klasy wymagane do wykonania zadania założonego dla danego algorytmu, łącznie z testami jednostkowymi sprawdzającymi poprawność implementacji. Wyniki prezentuje Tabela 1.

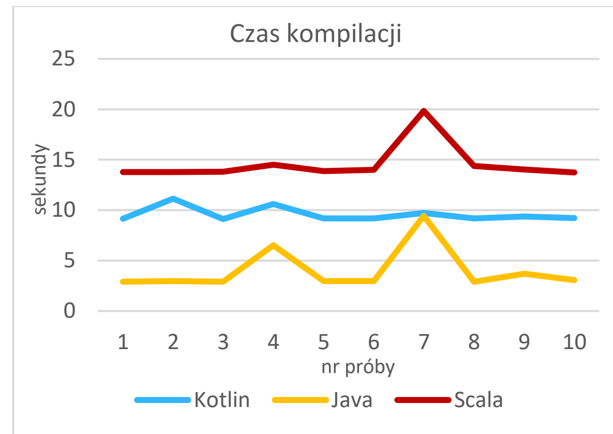
Tabela 1: Pomiary liczby linii kodu poszczególnych implementacji

	Kotlin	Java	Scala
Sito Eratostenesa	108	120	128
Quickhull	149	260	151
FFT	313	334	307
Klasy pomocnicze	21	40	16
Razem	**591**	**754**	**602**

Co łatwo zauważyć po powyższych danych, Java wymaga o ponad 20% więcej kodu do wykonania identycznych zadań co Kotlin lub Scala. Potwierdza to niejako często przytaczany argument o gadatliwości Javy czyli potrzeby pisania nieporównywalnie większej ilości kodu w celu wykonania podobnych zadań co w innych technologiach. Jeśli wejść w szczegóły to Java wypadła stosunkowo najsłabiej przy implementacji Quickhulla oraz klas pomocniczych, które objętościowo zajmowały ponad 50% więcej linii od najlepiej wypadającej w tym punkcie Scali. Całościowo, najlepiej na badanym polu wypadł Kotlin, jednak tuż za nim, z niewiele gorszym wynikiem uplasowała się Scala. Przy wnikliwym porównaniu obu języków uwagę zwraca pomijalnie mała różnica raz na korzyść, a innym razem na niekorzyść Kotlina jednak z jednym wyjątkiem – w przypadku implementacji Sita Eratostenesa Scala wymagała o 15,63% linii kodu więcej niż Kotlin.

5.3. Czas kompilacji

Wykonano 10 prób budowania całego utworzonego na potrzeby artykułu projektu komendą *mvn clean install*. Zadane parametry określają tryby pracy, jakie zdecydowano się włączyć tak, że pierwszy z wymienionych wymusza czyszczenie uprzednio skompilowanych źródeł zaś drugi zleca wykonanie całościowego procesu zbudowania projektu i umieszczenia spakowanego wyniku do lokalnego repozytorium Maven.



Rysunek 9: Czasy kompilacji modułów w kolejnych próbach

Wszystkie pomiary przedstawione zostały na Rysunek 9. Należy tutaj wziąć pod uwagę dosyć niewielki rozmiar projektu, co pokazuje np. badanie długości kodu z podpunktu 4.2., przez co uzyskane wyniki mogą nie być całkowicie reprezentatywne przy rozpatrywaniu możliwości użycia jednej bądź drugiej technologii w dużych aplikacjach jednak same uzyskane wyniki przedstawiają pewien trend, który niewątpliwie mniej lub bardziej utrzyma się nawet w wysokoskalowalnych projektach. Najniższy czas, a zarazem najlepszy wynik uzyskała Java, zaś moduły pozostałych języków budowały się nawet od trzech do czterech razy dłużej.

5.4. Objętość API

Obszerność interfejsów programistycznych do przetwarzania równoległego w wykorzystanych językach była kolejnym kryterium badawczym. Analiza wykonana została dla bytów zawartych w następujących paczkach bibliotek standardowych:

- kotlin.coroutines i kotlinx.coroutines dla Kotlina;
- java.util.concurrent dla Javy;
- scala.concurrent dla Scali.

Dodatkowo, aby nie zaciemniać wyników, pominięto przypadki bytów klasopodobnych, metod, pól oznaczonych jako przestarzałe bądź do usunięcia (poprzez adnotację @Deprecated dla Javy oraz odpowiedników w pozostałych językach) oraz klas niebędących bezpośrednio związanymi z omawianymi mechanizmami, np. w przypadku paczki javowej wyróżnić można wiele implementacji kolekcji dedykowanych do bezpiecznego przetwarzania równoległego, które jednak nie odnoszą się bezpośrednio do badanego obszaru.

Tabela 2: Wyniki obszerności API

	Kotlin	Java	Scala
Byty klasopodobne	52	52	18
Własności	307	415	68

Tabela 2. zawierająca wyniki przeprowadzonej analizy, pokazuje niekwestionowaną przepaść pomiędzy Scalą o dwoma pozostałymi językami. Różnica w przypadku bytów klasopodobnych była ponad dwukrotna, a dla własności wyniosła aż 451% na niekorzyść Scali w porównaniu z drugim wynikiem uzyskanym przez współprogramy Kotlina. Najbardziej obszernym API

może poszczycić się Java, notując zauważalnie wyższą liczbę własności od drugiej z kolei technologii. Jeżeli jednak chodzi o byty klasopodobne to na tym poziomie wypadła równie dobrze co Kotlin.

6. Wnioski

Przeprowadzona analiza porównawcza nie pozwala na jednoznaczne wyłonienie technologii najlepszej do programowania równoległego. Każda z przedstawionych mechanik ma swoje jasne i ciemne strony, tj. w niektórych przypadkach prezentuje się lepiej na tle konkurencji, a w innych gorzej.

Najgorzej w połowie z założonych wcześniej kryteriów porównawczych zachowała się Scala. Słaba na tle konkurentów dokumentacja oraz co najwyżej zbliżona do nich wydajność, wskazują, że język ten jest raczej w odwrocie i sprawdza się dobrze jedynie w bardzo specyficznych niszach. Zaletą współbieżnej Scali jest za to niewątpliwie zwięzłość tworzonych rozwiązań, będąca na zbliżonym poziomie co w Kotlinie i jednocześnie zdecydowanie lepsza niż w Javie.

Z kolei współbieżny interfejs programistyczny Javy zachował się dobrze lub poprawnie w kryteriach takich jak obszerność API czy czas kompilacji. Zdecydowanie gorzej zaprezentował się przy dokumentacji online, długości kodu wynikowego, możliwościach debuggingu oraz podatności na błędy. Wynika to przynajmniej częściowo z długiego stażu tej technologii na rynku i przekłada się na bycie odpowiednim wyborem dla dużych i długookresowych projektów, w których zaangażowane jest znaczne grono osób.

Tytułowe współprogramy Kotlina poradziły sobie w przeprowadzonych badaniach nierówno, tj. w kwestii zwięzłości kodu okazały się prezentować najlepsze wyniki. Idąc dalej, odznaczały się podobną lub nawet lepszą wydajnością od Javy i podobnie rozległym API. Zdecydowanie najsłabiej wyglądało tu kryterium czasu kompilacji, w którym wynik był kilka razy gorszy od Javy i na podobnym poziomie co Scala. Nie jest jednak pewne czy podobna różnica zaistniałaby również w przypadku projektu o dużej skali jako, że stworzone na potrzeby pracy rozwiązanie nie jest bardzo rozległe i z pewnością nie oddaje w 100% przypadków użycia mogących wystąpić w biznesie czy nauce. Wymienione wyżej cechy wskazują na idealne dopasowanie Kotlina do nowych projektów realizowanych, np. przez startupy lub mniejsze firmy oraz, dzięki niskiej podatności na błędy, dla zastosowań akademickich. Krótki okres obecności na rynku, a co za tym idzie mniejsza stabilność zdefiniowanego API czy wsparcia od stron trzecich skłania do wniosku, że w przypadku np. wielkoskalowych projektów o długim czasie wsparcia, ostateczny wybór powinien być poprzedzony dodatkowymi analizami z innych punktów widzenia.

Literatura

[1] Parallel Computing: Background, https://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf, [22.05.2020].

[2] M. E. Conway, Design of a Separable Transition Diagram Compiler, Communications of the ACM 6.7 (1963) 396-408.

[3] A. L. De Moura, R. Ierusalimschy, Revisiting Coroutines, ACM Transactions on Programming Languages and Systems 31.2 (2009) 1-31.

[4] D. Racordon D, Coroutines with Higher Order Functions, arXiv preprint arXiv:1812.08278 (2018).

[5] A. Ohlsson, E. Leffler E, A Coroutine Extension to Java, (2018).

[6] D. E. Damasceno Costa, C. Bezemer, P. Leitner, A. Andrzejak, What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks, IEEE Transactions on Software Engineering (2019).

[7] M. E. O'Neill, The Genuine Sieve of Eratosthenes, Journal of Functional Programming 19.1 (2009) 95-106.

[8] Sieve of Eratostheneses animation, https://en.wikipedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif, [22.05.2020].

[9] J. S. Greenfield, A Proof for a QuickHull Algorithm (1990).

[10] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example3.svg, [22.05.2020].

[11] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example6.svg, [22.05.2020]..

[12] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example7.svg, [22.05.2020]

[13] D. Nam, A. Horvath, A. Macvean, B. Myers, B. Vasilescu, MARBLE: Mining for Boilerplate Code to Identify API Usability Problems, 2019 34th IEEE/ACM International Conference on Automated Software Engineering (2019) 615-627.

[14] Why are Scala for loops slower than logically identical while loops?, https://stackoverflow.com/questions/21373514/why-are-scala-for-loops-slower-than-logically-identical-while-loops, [22.05.2020].

[15] Micro-optimizing your Scala code, https://www.lihaoyi.com/post/MicrooptimizingyourScalacode.html#speed-through-while-loops, [22.05.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 247–251
Received: 9 June 2020
Accepted: 16 June 2020

Performance testing of STL and Qt library elements in multi-threaded processing

Badanie wydajności elementów bibliotek STL i Qt w przetwarzaniu wielowątkowym

Piotr Krasowski*, Jakub Smołka

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

Abstract

In recent years multithreaded processing has become a important programming aspect. Computers with a multi-core processor are now widely available, enabling the creation of more efficient applications. Many libraries support multi-threaded solutions, but performance information is often lacking. The use of appropriate data structures and algorithms significantly speeds up the process of creation and development of applications. Article describes selected elements of the Qt and STL library and compares their performance in concurrent programming. The test was performed with custom applications created with C++. The time needed to perform individual operations was analysed.

Keywords: concurrent computing; multithreading; container performance; data structures

Streszczenie

Przetwarzanie wielowątkowe na przestrzeni ostatnich lat stało się ważnym aspektem programistycznym. Komputery dysponujące procesorem wielordzeniowym są obecnie powszechnie dostępne co umożliwia tworzenie wydajniejszych aplikacji. Wiele bibliotek wspiera rozwiązania wielowątkowe lecz często brakuje informacji o wydajności. W artykule opisano wybrane elementy biblioteki Qt i STL oraz porównano ich wydajność w programowaniu współbieżnym. Testy zostały przeprowadzone za pomocą autorskich aplikacji napisanych w języku C++. Wyniki przedstawiono w postaci analizy czasów potrzebnych na wykonanie poszczególnych operacji.

Słowa kluczowe: przetwarzanie współbieżne; wielowątkowość; wydajność kontenerów; struktury danych

*Corresponding author

Email address: **piotr.krasowski@pollub.edu.pl** (P. Krasowski)

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

1. Wstęp

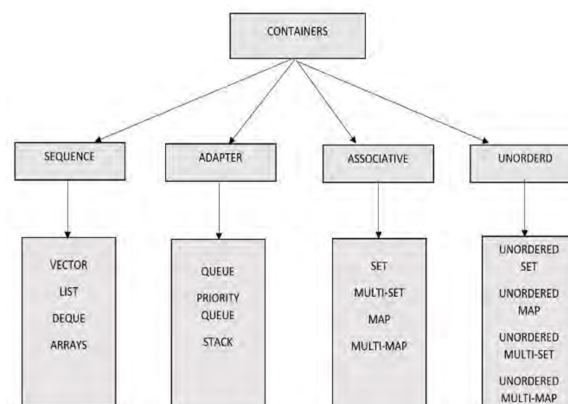
Gromadzenie i przetwarzanie danych to jedno z najważniejszych zagadnień we współczesnym świecie zdominowanym przez komputery. Pomimo imponujących maszyn potrzeba wiele czasu by przetworzyć dane tak by były użyteczne. Dzięki wydajnym komputerom i powszechnemu dostępowi do sieci zbieranie i przetwarzanie informacji jest dużo prostsze niż dawniej lecz nadal szybkość i wydajność tego procesu zależy głównie od programu. Istnieje wiele implementacji algorytmów i struktur przechowujących dane lecz ich wydajność w dużej mierze zależy od doboru odpowiednich rozwiązań do realizowanego zadania.

W dzisiejszych czasach procesory posiadają wiele rdzeni, z których każdy jest w stanie wykonywać niezależne operacje od pozostałych. Uwzględnienie aspektu wielowątkowości podczas tworzenia aplikacji pozwala na wykorzystanie pełni potencjału współczesnych maszyn. Programowanie współbieżne nie jest trywialnym zagadnieniem nawet dla doświadczonych programistów. Kluczowym aspektem na etapie projektowania aplikacji staje się dobór odpowiednich bibliotek.

Ze względu na dużą liczbę wątpliwości dotyczących wydajności poszczególnych bibliotek oraz mnogość dostępnych rozwiązań w ramach artykułu omówiono koncepcję programowania współbieżnego bibliotek Qt i STL. Zbadano i porównano wydajność wybranych kontenerów i funkcji dostępnych w bibliotece Qt w wersji 5.14 i zasobach STL.

2. Struktury danych

Wszystkie wykorzystywane w aplikacji zmienne oraz stałe są wczytywane i przechowywane w pamięci komputera. Uzyskanie dostępu do konkretnej zmiennej wiąże się z odwołaniem się do odpowiedniego adresu w pamięci. Struktury danych [1] czyli tzw. kontenery umożliwiają łatwiejszy dostęp do danych. Typy kontenerów zostały zaprezentowane na rysunku 1.



Rysunek 1: Podział kontenerów ze względu na typ – źródło [7]

Według artykułu [2] kontenery w języku C++ to klasy szablonowe wyższego poziomu. Umożliwiają one programistom łatwiejsze zarządzanie kodem. Pozwalają także na łatwe przetwarzanie danych za pomocą różnego rodzaju algorytmów.

3. Przetwarzanie współbieżne

Wykorzystanie koncepcji programowania współbieżnego pozwala na realizację przez system wielu złożonych operacji jednocześnie przy czym sposób realizacji koncepcji jest ściśle uzależniony od jednostki centralnej danej maszyny. Dawniej procesory były w stanie wykonywać tylko jedną operację jednocześnie. W takim przypadku w celu umożliwienia przetwarzania współbieżnego stosowano tzw. mechanizm przełączania kontekstu [3]. Sprowadza się on do realizacji małych fragmentów poszczególnych zadań naprzemiennie co daje efekt iluzji współbieżności. Upowszechnienie się procesorów wielordzeniowych zmieniło to podejście. Takie jednostki mogą wykonywać wiele operacji jednocześnie bez sztucznego przełączania. Ten rodzaj przetwarzania określany jest mianem współbieżności sprzętowej. Dodatkowo współczesne jednostki centralne mogą posiadać wiele wątków sprzętowych co umożliwia realizację kilku zadań równolegle przez jeden procesor bez wykorzystania mechanizmu przełączania kontekstu.

4. Standardowa biblioteka szablonów

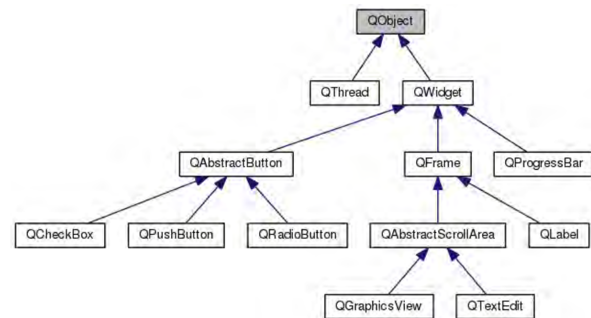
Standardowa biblioteka szablonów (ang. standard template library) [4] została stworzona w latach 90 przez Alexandra Stepanova na długo przed standaryzacją języka C++. Po tym jak jej elementy zostały przeniesione do współczesnej wersji standardu języka C++ zaprzestano używania przedrostka stl na rzecz std.

Biblioteka STL jest zbiorem ustandaryzowanych konstrukcji w formie szablonów służących do przechowywania danych [5]. Jest tzw. biblioteką generyczną, co oznacza, że jej elementy są kompatybilne z natywnymi typami języka C++ jak i typami zdefiniowanymi przez użytkownika. Główne komponenty biblioteki to::

- Algorytmy - funkcje kompatybilne z kontenerami STL, służące do wykonywania operacji takich jak sortowanie czy wyszukiwanie na danym przedziale danych,
- Kontenery - struktury danych umożliwiające łatwiejszy dostęp do danych,
- Funktory - obiekty funkcyjne umożliwiające między innymi na przekazywanie funkcji jako parametr lub przypisanie jej do zmiennej,
- Iteratory - obiekty umożliwiające dostęp do poszczególnych elementów w kontenerach i generalizację algorytmów.

5. Qt

Framework Qt [6] to zestaw bibliotek zawierający setki gotowych rozwiązań pozwalających na konstruowanie aplikacji okienkowych i konsolowych. Dostarcza także pakiet rozwiązań ułatwiający tworzenie złożonych aplikacji wielowątkowych i zarządzanie danymi. Qt jest biblioteką wieloplatformową co sprawia, że raz napisany kod może być kompilowany i uruchamiany na wielu platformach sprzętowych. Biblioteka wspiera takie języki programowania jak C++, Python czy Java Script. Rysunek 2 przedstawia hierarchie klas biblioteki Qt.



Rysunek 2: Schemat hierarchii klas w bibliotece Qt – źródło [7]

Mechanizmem odróżniającym Qt od innych bibliotek jest system sygnałów i slotów [7]. Odpowiada on za komunikację pomiędzy poszczególnymi obiektami.

6. Metoda badań

Celem badań było zbadanie wydajności wybranych kontenerów bibliotek STL i Qt oraz struktur odpowiadających za przetwarzanie wielowątkowe. Podobnie jak w testach opisanych w [8] stworzono aplikacje testujące wydajność każdego z badanych obiektów. Weryfikacja wydajności została przeprowadzona poprzez zmierzenie czasu potrzebnego na wykonanie określonych operacji. Do pomiaru czasu został użyty moduł QTest zawierający się w bibliotece Qt. Tabela 1 przedstawia zestawienie obiektów badanych klas obu bibliotek.

Tabela 1: Zestawienie kontenerów bibliotek Qt i STL

Qt	STL
QVector	std::vector
QList	std::list
QMap	std::map
QHashMap	std::unordered_map

Weryfikacja wydajności kontenerów została przeprowadzona poprzez zmierzenie czasu potrzebnego na wykonanie podstawowych operacji. W przypadku kontenerów sekwencyjnych były to funkcje odczytu i dodawania nowego elementu. W przypadku kontenerów asocjacyjnych i haszowanych zmierzono czas odczytu wartości za pomocą klucza, wstawiania elementu.

Każdy przypadek testowy został powtórzony kilkukrotnie dla różnych rozmiarów kontenera. W pojedynczym teście dana operacja jak np. dodawanie elementu została wykonana n-razy w celu uzyskania dokładniej-

szych wyników. Algorytm pojedynczego testu w przypadku kontenera można przedstawić w następujących krokach:

1. Deklaracja kontenera o rozmiarze r,
2. Wypełnienie kontenera losowymi danymi,
3. Rozpoczęcie pomiaru czasu,
4. Wykonanie danej operacji n razy,
5. Zakończenie pomiaru czasu.

W celu porównania obiektów std::thread i QThread stworzono aplikację obliczającą n-ty wyraz ciągu Fibonacciego, a następnie zmierzono czas potrzebny na wykonanie obliczenia. Każdy przypadek testowy został wykonany kilkukrotnie dla różnej liczby wątków i różnych wartości indeksów kolejnych wyrazów ciągu. Algorytm pojedynczego testu w przypadku wątku został przedstawiony w poniższych krokach:

1. Deklaracja kontenera o rozmiarze r,
2. Wypełnienie kontenera losowymi indeksami wyrazu z zakresu 5-35,
3. Rozpoczęcie pomiaru czasu,
4. Deklaracja liczby użytych wątków,
5. Podział danych w kontenerze i przekazanie ich do poszczególnych wątków,
6. Wyznaczenie wyrazu ciągu n razy na podstawie indeksu,
7. Zakończenie pomiaru czasu.

7. Wyniki badań

Pierwsza część testów dotyczyła porównania czasu odczytu dla kontenerów sekwencyjnych. Wyniki testów dla implementacji poszczególnych bibliotek przedstawia tabela 2 i 3.

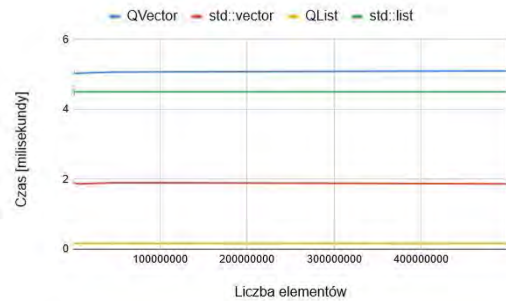
Tabela 2: Czas odczytu i wstawiania dla wektora o rozmiarze r w zależności od pozycji elementu

Typ	QVector			std::vector		
Pozycja	0	r/2	r	0	r/2	r
Czas odczytu (ms)	4,9	5,1	4,8	1,9	1,8	1,9
Czas wstawiania (ms)	23177	22533	0,01	22351	23198	0,01

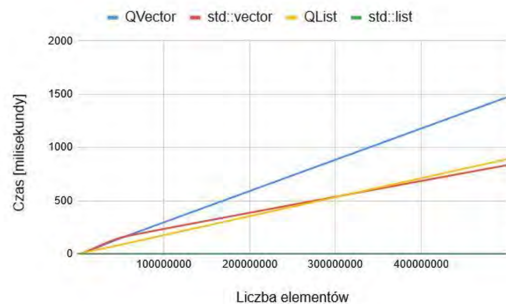
Tabela 3: Czas odczytu i wstawiania dla listy o rozmiarze r w zależności od pozycji elementu

Typ	QList			std::list		
Pozycja	0	r/2	r	0	r/2	r
Czas odczytu (ms)	0,2	0,2	0,2	0,01	0,4	0,01

Wyniki dla poszczególnych rozmiarów kontenera przedstawiono na rysunku 3 i 4.



Rysunek 3: Uśredniony czas odczytu elementu dla kontenerów sekwencyjnych



Rysunek 4: Uśredniony czas dodawania elementu do kontenerów sekwencyjncyh

Druga część testów dotyczyła porównania czasów odczytu dla kontenerów asocjacyjnych i haszujących. Wyniki badań zostały przedstawione w tabelach 4 i 5.

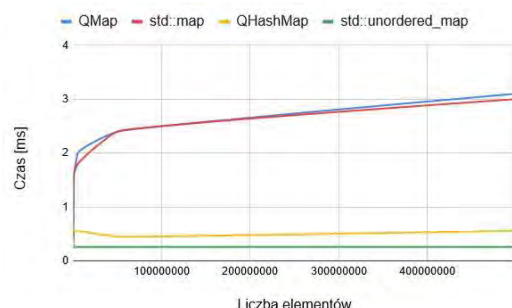
Tabela 4: Czas wykonania poszczególnych operacji dla kontenerów asocjacyjnych

Typ	QMap		std::map	
Operacja	Odczyt	Wstawianie	Odczyt	Wstawianie
Czas (ms)	2,1	2,3	2,1	10,7

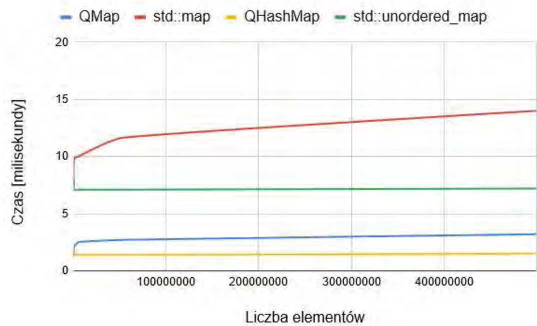
Tabela 5: Czas wykonania poszczególnych operacji dla kontenerów haszujących

Typ	QHashMap		std::unordered_map	
Operacja	Odczyt	Wstawianie	Odczyt	Wstawianie
Czas (ms)	0,58	1,4	0,26	7,2

Wyniki dla danej liczby elementów zawartych w kontenerze przedstawiono na rysunku 5 i 6.



Rysunek 5: Porównanie czasów odczytu



Rysunek 6: Uśredniony czas dodawania elementu dla kontenerów asocjacyjnych i haszowanych

Ostatnia część testów dotyczyła porównania wydajności kontenerów STL i Qt w środowisku wielowątkowym. Wyniki testów dla implementacji z użyciem poszczególnych bibliotek zostały przedstawione w tabelach 6 i 7.

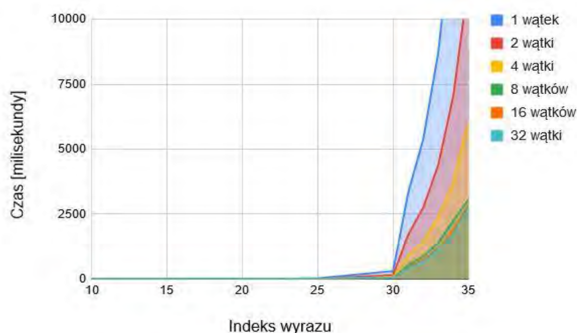
Tabela 6: Uśredniony czas wykonania operacji wyznaczenia n-tego wyrazu ciągu dla liczby użytych wątków z użyciem Qt

Nazwa biblioteki	Qt					
Liczba wątków	1	2	4	8	16	32
Czas (s)	9,4	4,2	1,9	1,2	0,9	1,5

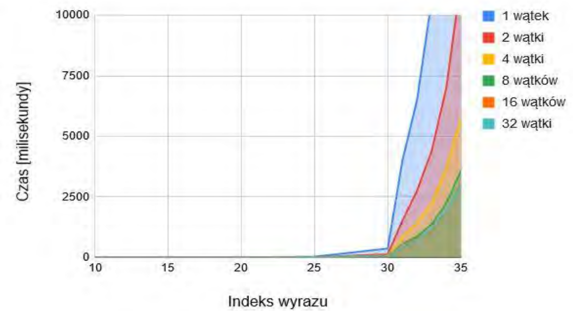
Tabela 7: Uśredniony czas wykonania operacji wyznaczenia n-tego wyrazu ciągu dla liczby użytych wątków z użyciem stl i std

Nazwa biblioteki	STL + elementy standardu C++					
Liczba wątków	1	2	4	8	16	32
Czas (s)	7,8	3,2	1,6	1,1	0,7	0,9

Wyniki dla poszczególnych indeksów ciągu i liczby wątków przedstawiono na Rys 7 i 8 .



Rysunek 7: Czas wyznaczenia wyrazu ciągu Fibonacciego o danym indeksie dla implementacji Qt



Rysunek 8: Czas wyznaczenia wyrazu ciągu Fibonacciego o danym indeksie dla implementacji stl i std

8. Wnioski

Badania pokazały, że wydajność implementacji kontenerów poszczególnych bibliotek jest uzależniona od typu użytego kontenera.

Analizując tabele 2 można stwierdzić, że wektor jest strukturą zoptymalizowaną do umieszczania elementów na końcu kontenera. Dostęp do elementów ma stały czas i nie zależy od rozmiaru struktury. W przypadku operacji wstawiania wydajniejsza jest implementacja QVector, natomiast przy odczycie lepiej sprawdza się std::vector co widać na rysunkach 3 i 4.

Na podstawie wyników przedstawionych w tabeli 3 można stwierdzić, że listy są wydajnymi kontenerami jeżeli chodzi o operacje wstawiania elementów niezależnie od pozycji. W porównaniu do wektora są mniej wydajne w operacjach odczytu.

Wyniki przedstawione w tabeli 4 potwierdzają, że struktury asocjacyjne są zoptymalizowane pod względem wyszukiwania i dostępu do elementu na podstawie klucza. Analizując tabele 5 można stwierdzić, że kontenery haszujące są wydajniejsze przy operacji odczytu i wstawiania. Na podstawie rysunków 5 i 6 można wywnioskować, że w większości przypadków testowych wydajniejszą implementacją kontenera była wersja STL.

Wyniki przedstawione w tabelach 6 i 7 udowadniają, że standard języka C++ dostarcza wydajniejsze struktury wielowątkowe. Analizując rysunki 7 i 8 trzeba stwierdzić, że nadmierny podział zadania głównego na mniejsze zadania wykonywane w środowisku wielowątkowym jest złym podejściem. Czas wykorzystywany inicjalizację obiektów i synchronizację wątków wydłuża wykonywanie obliczeń i komplikuje zarządzanie aplikacją.

Literatura

[1] M. Matsuda, M. Sato, Y. Ishikawa, Parallel array class implementation using C++ STL adaptors, 2006.

[2] H. Bischof H, Generic Parallel Programming Using C++ Templates and Skeletons, 2016, 43–55.

[3] A. Williams, Język C++ i przetwarzanie współbieżne w akcji, 2019.

[4] Opis struktury i zasady działania kontenerów C++. https://www.geeksforgeeks.org/containers-cpp-stl/, 2016, 42-55.

[5] Dokumentacja techniczna C++ dotycząca standardu, https://isocpp.org/std/, [23.05.2020].

[6] R. Penea, Mastering Qt 5, 2016, 45-68.

[7] Dokumentacja biblioteki Qt dotycząc kontenerów, https://doc.qt.io/qt-5/containers.html, [29.05.2020].

[8] B. Kyle, QThreads: An api for programming with millions of lightweight threads, 2010 24-26

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 252–260
Received: 20 May 2020
Accepted: 28 May 2020

A security analysis of authentication and authorization implemented in web applications based on the REST architecture

Analiza bezpieczeństwa mechanizmów uwierzytelniania oraz autoryzacji implementowanych w aplikacjach internetowych zbudowanych w oparciu o architekturę REST

Tomasz Muszyński*, Grzegorz Kozieł

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

Abstract

The purpose of this article is to prepare a security analysis of authentication and authorization mechanisms in web applications based on the REST architecture. The article analyzes the problems encountered during the implementation of the JSON Web Token (JWT) mechanism. The article presents examples of problems related to the implementation of authorization and authentication, and presents good practices that help ensure application security.
Keywords: security; security vulnerability; REST; JWT

Streszczenie

Celem artykułu jest analiza bezpieczeństwa mechanizmów uwierzytelniania oraz autoryzacji w aplikacjach internetowych zbudowanych w oparciu o architekturę REST. W artykule przeanalizowano problemy spotykane podczas implementacji mechanizmu JSON Web Token (JWT). W artykule podano przykłady problemów związanych z wdrożeniem autoryzacji i uwierzytelniania oraz przedstawiono dobre praktyki ułatwiające zapewnienie bezpieczeństwa aplikacji.
Słowa kluczowe: bezpieczeństwo; podatność bezpieczeństwa; REST; JWT

*Corresponding author

Email address: **tomasz.muszynski@pollub.edu.pl** (T. Muszyński)

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

1. Wstęp

Współcześnie trudno spotkać oprogramowanie, które w swojej architekturze nie realizuje procesu uwierzytelnienia oraz autoryzacji.

Uwierzytelnianie to proces, którego celem jest weryfikacja czy zebrane od podmiotu poświadczenia potwierdzają jego tożsamość. Natomiast proces autoryzacji na podstawie zapisanych w systemie uprawnień podejmuje decyzję o odmowie lub przyznaniu dostępu do zasobów. Proces autoryzacji bazuje na zweryfikowanej tożsamości użytkownika.

Niepoprawna implementacja mechanizmów uwierzytelnienia oraz autoryzacji jest częstym powodem problemów bezpieczeństwa w przypadku aplikacji zbudowanych w architekturze REST [1]. Realizacja powyższych procesów najczęściej wykorzystuje mechanizm JSON Web Token, którego złożoność może być przyczyną popełnianych błędów a w konsekwencji wystąpienia podatności bezpieczeństwa. W artykule omówiono kontekst bezpieczeństwa JWT oraz problemy związane z implementacją uwierzytelniania oraz autoryzacji w aplikacjach zbudowanych w architekturze REST.

2. Cel publikacji

Celem publikacji jest przeprowadzenie analizy bezpieczeństwa procesów uwierzytelniania oraz autoryzacji implementowanych w aplikacjach internetowych zbudowanych w architekturze REST.

3. Przegląd literatury

Bezpieczeństwo aplikacji zbudowanych w architekturze REST API w niewielu publikacjach jest traktowane jako oddzielna kategoria poddana analizie. W literaturze polskojęzycznej trudno znaleźć opracowania tej tematyki, dopiero w 2019 roku ukazała się książka pt. „Bezpieczeństwo aplikacji webowych” wydana przez wydawnictwo SECURITUM, w której znajduje się rozdział poświęcony bezpieczeństwu architektury REST. W rozdziale pt. „Bezpieczeństwo API REST”, autor zauważa podobieństwa aplikacji w architekturze REST do pozostałych aplikacji internetowych, natomiast wskazuje również różnice, które powodują, że warto ten temat analizować jako oddzielną kategorię. W rozdziale można znaleźć informację o przyczynach błędów bezpieczeństwa takich jak:

- brak dokładnego określenia jakie metody powinny być użyte do jakich operacji,
- użycie frameworków lub bibliotek zawierających podatności,
- włączony tryb „Debug”,
- różnorodność formatów danych,
- niepoprawna implementacja uwierzytelnienia i autoryzacji, częsty brak wdrożonego uwierzytelnienia.

Autor rozdziału wspomina również o małej liczbie publikacji badających bezpieczeństwo tej architektury. „Osobiście uważam, że zagadnienie jest jeszcze mało zbadane, niewiele mamy również dostępnych informacji

o konkretnych podatnościach w API REST – np. znalezionych w ramach programów bug bounty.”[1].

Kolejnym rozdziałem we wspomnianej książce jest rozdział opisujący niebezpieczeństwo związane z wdrożeniem JSON Web Token (JWT), użycie JWT jest powszechne w procesie wymiany danych uwierzytelniających implementowanych w REST API. W rozdziale można znaleźć analizę aż jedenastu problemów bezpieczeństwa JWT oraz praktyczne rady jak im przeciwdziałać. Temat bezpieczeństwa JWT to podstawowa kwestia związana z problemami spotykanymi w aplikacjach opartych na architekturze REST.

Autorzy badający bezpieczeństwo REST API w dużej mierze skupiają się na problemach, które związane są z implementacją JWT. W artykule pt. „Microservices API Security”[2] autor wspomina o braku szyfrowania JWT. JWT jest podpisane, jednak nie są automatycznie szyfrowane (szyfrowanie JWT jest funkcją opcjonalną). To znaczy że wszelkie dane znajdujące się w tokenie mogą być odczytane przez każdego, kto uzyska do niego dostęp. Autor również jako problem opisuje brak możliwości natychmiastowego zablokowania użytkownika poprzez anulowania JWT, ponieważ token przechowywany jest po stronie klienta.

Autor artykułu „The Protection of Information in Computer Systems”[3] wskazuje na to, że REST API powinno być bezstanowe, w każdym żądaniu powinien być przesyłany komplet informacji pozwalający na ponowne weryfikowanie czy przesłane informacje umożliwiają dostęp do systemu, co również wiąże się z wybraniem JWT jako mechanizmu pozwalającego na dołączanie danych uwierzytelniających w każdym zapytaniu. Autor omawia dobre praktyki projektowania zabezpieczeń, które warto wdrożyć w REST API, opisywane zasady wskazują miedzy innymi, że:

- użytkownik powinien mieć tylko takie uprawnienia, które są niezbędne do wykonywania przez niego zadań, uprawnienia powinny być rozszerzane wyłącznie w ramach potrzeb,
- nie powinno stosować się domyślnych poziomów uprawnień,
- system powinien być możliwie prosty, dzięki temu będzie zawierał mniej błędów,
- każde zapytanie powinno być weryfikowane pod względem uprawnień bazując na rzeczywistym stanie, a nie zbuforowanych danych, tak aby odwołanie uprawnień działało natychmiast.

Kolejnym źródłem wiedzy o bezpieczeństwie aplikacji internetowych są publikacje organizacji OWASP (ang. Open Web Application Security Project). OWASP wydaje specjalne dokumenty miedzy innymi zawierające opis najbardziej krytycznych podatności aplikacji internetowych, organizacja postanowiła wydzielić kategorie podatności API jako oddzielny zbiór. Opisane w publikacji podatności można uznać za ściśle związane z implementacja API w standardzie REST. Wydzielenie tematyki API przez organizację OWASP udowadnia, że ten temat zasługuje na oddzielną analizę.

OWASP opracował dwa dokumenty: „Application Security Verification Standard 4.0”[4] w skrócie określany ASVS oraz „Top 10 Proactive Controls”[5], dokumenty te przeznaczone są jako pomoc w implementacji i w testowaniu bezpieczeństwa aplikacji internetowych. W powyższych dokumentach można znaleźć informacje na temat wymagań, które należy zweryfikować podczas pracy z oprogramowaniem zbudowanym w architekturze REST. Publikacja zawiera wskazania na potrzebę następujących weryfikacji:

- poprawność wprowadzanych danych, wdrożenie odpowiedniej walidacji, sprawdzanie typu danych wejściowych,
- czy komunikacja jest szyfrowana,
- czy JWT zostało prawidłowo zaimplementowane.

W wyniku analizy bieżącego stanu wiedzy należy stwierdzić, że wydzielenie aplikacji zbudowanych w architekturze REST z pośród wszystkich aplikacji internetowych jest zasadne.

Często omawianym słabym punktem związanym z aplikacjami wykorzystującymi REST jest zastosowanie złożonego mechanizmu JWT. Pomimo, że wiele występujących podatności we współczesnych aplikacjach internetowych spowodowanych jest takimi czynnikami jak przeoczenie, niedbałość, brak testowania, to wybór architektury REST może mieć konsekwencje, o których warto wiedzieć podczas projektowania i implementacji.

4. REST

Architektura oprogramowania określa organizację systemu i jego komponentów, definiuje reguły budowy oraz zasady rozwoju. REST (ang. REpresentational State Transfer) jest stylem architektury oprogramowania dedykowanym dla rozproszonych systemów, został zaprezentowany przez Roya Fieldinga w 2000 roku [6]. Z architekturą REST ściśle wiąże się pojęcie API (ang. Application Programming Interface). API jest zbiorem reguł określających zasady komunikacji pomiędzy programami komputerowymi. REST API w ostatnich latach zyskało dużą popularność, wypierając alternatywną metodę komunikacji jaką jest wykorzystanie protokołu SOAP. Rysunek 1 przedstawia zainteresowanie architekturą REST w porównaniu z protokołem SOAP w ujęciu czasowym według wyszukiwarki Google.

Rysunek 1: Porównanie zainteresowania REST API z SOAP [7]

Autor REST zdefiniował sześć następujących ograniczeń:

- bezstanowość (ang. stateless) – każde żądanie jest niezależne od poprzedniego, jest to też własność protokołu HTTP,
- jednolity interfejs (ang. Uniform interface) – jednolity interfejs definiowany jest przez cztery założenia.

Po pierwsze adresy powinny jednoznacznie wskazywać do jakiego zasobu się odwołują. Drugie ograniczenie mówi o tym, że zasoby mogą być modyfikowane przez klienta, posiadającego reprezentację zasobu i dołączone do niego metadane. Trzecie ograniczenie wskazuje, że wiadomości wymieniane powinny zawierać taką ilość informacji, która pozwoli serwerowi na dokładne ich przetworzenie. Ostatnie ograniczenie określa, że serwer może odpowiadać tekstem zawierającym hiperłącza,
- pamięć podręczna (ang. cache) – serwer w odpowiedzi powinien określić czy odpowiedź może zostać buforowana. Dane zwracane przez interfejs mogą być danymi, które często ulegają zmianie wtedy takie dane warto oznaczyć jako niebuforowane. Istnieją też dane, które nie zmieniają się lub zmieniają się rzadko w takich przypadkach zastosowanie buforowania może w znacznym stopniu zmniejszyć liczbę połączeń z serwerem, przez co pozwoli zminimalizować wykorzystywane zasoby,
- klient-serwer (ang. client-server) – odseparowanie zadań wykonywanych przez serwer od interfejsu użytkownika, zwiększa możliwości skalowalności i ułatwia dostarczanie rozwiązań na różne platformy,
- system warstwowy (ang. layered system) – architektura składa się z wydzielonych warstw, każda z warstw ma oddzielne zadania i prowadzi interakcje wyłącznie z przyległymi warstwami,
- kod na żądanie (ang. code on demand) – jest opcjonalnym ograniczeniem, REST pozwala klientowi na rozszerzanie funkcjonalności poprzez odbieranie i wykonywanie kodu w postaci skryptów lub apletów zwiększając funkcjonalność systemu.

Wszystkie informacje związane z REST określane są jako zasób. Zasobem może być tekst, plik multimedialny, użytkownik. Stan zasobu jest określany jako reprezentacja zasobu. Reprezentacja zbudowana jest z danych, hiperłączy, oraz metadanych ich opisujących [6].

5. JSON Web Token

JSON Web Token w skrócie JWT, jest otwartym standardem zdefiniowanym w dokumencie RFC 7519 [8]. JWT został opracowany w celu bezpiecznego przekazywania danych pomiędzy komunikującymi się usługami. Bezpieczeństwo może zostać zrealizowane na dwóch płaszczyznach, pierwsza to integralność danych. Integralność realizowana jest przy wykorzystaniu podpisów. Drugim aspektem bezpieczeństwa jest poufność, dane przekazywane przy użyciu JWT mogą zostać zaszyfrowane [9]. Dane JWT mogą zostać zakodowane w jedną z dwóch struktur:
- JWS (ang. JSON Web Signature),
- JWE (ang. JSON Web Encryption).

W praktyce często JWS nazywany jest jako JWT [1].

JWT ze względu na swoją specyfikę jest najczęściej wykorzystywany w dwóch procesach implementowanych w oprogramowaniu:
- autoryzacja – po udanym uwierzytelnieniu, serwer przesyła do klienta JWT, który może zostać użyty w kolejnych żądaniach jako potwierdzenie tożsamości uprawniające do operacji na autoryzowanych zasobach,
- wymiana informacji – dane są przesyłane między stronami przy wykorzystaniu par kluczy publiczny - prywatny, zapewnia to bezpieczne przesyłanie informacji [9].

JWS składa się z trzech oddzielonych kropkami elementów, każdy z elementów zakodowany jest algorytmem Base64URL:
- nagłówek – zbudowany jest z dwóch elementów określających typ tokena oraz algorytm podpisu,
- payload – zawiera docelowe dane, które będą istotne z punktu widzenia realizowanej funkcjonalności, w ładunku przesyłane są zazwyczaj dane identyfikujące użytkownika systemu, oraz dane pomocnicze takie jak czas ważności,
- podpis – gwarantuje integralność danych, na podpis składa się nagłówek, payload oraz tajny klucz przechowywany po stronie serwera. Podpis jest wynikiem działania algorytmu określonego w nagłówku w parametrze „alg"[9].

Przykładowy JWT został zaprezentowany na Rysunku 2.

Rysunek 2: Przykład JWT [1]

5.1. Kontekst bezpieczeństwa

Struktura JWT jest skomplikowana i nie zawsze jest dobrze zrozumiana przez osoby implementujące jej wykorzystanie. Dane przesyłane w JWS nie są szyfrowane i mogą zostać odczytane , dlatego też w JWS nie powinny znajdować się poufne informacje, które należy chronić, dopiero struktura JWE implementuje mechanizmy szyfrowania danych i jest w stanie zagwarantować ich poufność. JWS umożliwia zapewnienie integralności danych, która realizowana jest przez podpis Jeżeli atakujący zmieni payload podpis nie zostanie poprawnie zweryfikowany przez serwer i JWT zostanie odrzucony. Zmiana podpisu przez atakującego nie jest możliwa ponieważ elementem podpisu jest tajny klucz znajdujący się po stronie serwera. JWT dostarcza dużą elastyczność przy zapewnieniu bezpieczeństwa, natomiast ze względu na złożoność implementacji mogą wystąpić błędy, w wyniku których pojawią się podatności bezpieczeństwa.

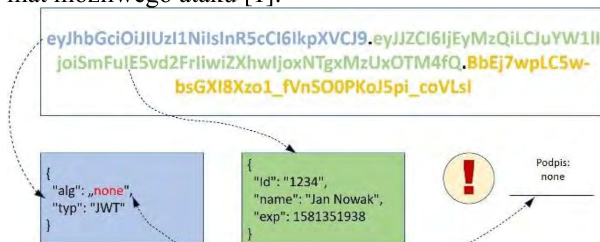
Dokumentacja opisująca specyfikę JWT wskazuje na możliwość użycia JWT bez podpisu, brak podpisu określany jest w nagłówku, który może wyglądać jak na Listingu 1.

Listing 1: Nagłówek JWT określający brak wymagania podpisu

```
{
  "alg": "none",
  "typ": "JWT"
}
```

Atakujący może wykorzystać ten fakt i spróbować zmienić algorytm podpisu na „none", jeżeli zaimplementowana weryfikacja JWT nie uwzględnia takiej sytuacji, napastnikowi może udać się dostać do nieprzeznaczonych dla niego danych.

Podatność występuje w bibliotekach implementujących JWT, biblioteki niepoprawnie realizują funkcjonalność weryfikacji podpisu ponieważ nie są odporne na zmianę typu algorytmu. Atakujący zmienia zawartość payloadu a następnie modyfikuje algorytm podpisu, w kolejnym kroku JWT wysyłany jest w żądaniu a serwer podejmuje próbę weryfikacji. Serwer powinien zwrócić błąd weryfikacji, natomiast ze względu na luki nie zawsze tak się dzieje. Rysunek 3 przedstawia schemat możliwego ataku [1].



Rysunek 3: Schemat ataku [1]

W celu zobrazowania przebiegu ataku wykorzystano środowisko NodeJS w wersji 10.16.3, oraz bibliotekę „jsonwebtoken" w wersji 8.5.1.

Istotę podatności można przedstawić za pomocą prostego kodu programu. W pierwszym kroku tworzony jest nowy JWT, następnie wykonywane jest jego dekodowanie oraz weryfikacja. Kolejnym etapem jest zmiana algorytmu na algorytm „none", następnie ponownie wykonywane jest dekodowanie i weryfikacja. Przykład obrazujący na czym polega luka zaprezentowano na Listingu 2.

Listing 2: Kod JavaScript obrazujący podatność

```
const jwt = require('jsonwebtoken');
const base64url = require('base64url');

const privateKey = '123456789abcdefgh';

//Utworzenie podpisanego tokenu
let token = jwt.sign({ id: '1234', name: 'Jan Nowak', admin: false },
privateKey, { algorithm: 'HS256'});

// Przed ingerencją w token
const decodedToken = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken);

const verifiedToken = jwt.verify(token, privateKey);
console.log('Zweryfikowany token: ');
console.log(verifiedToken);

//Zmiana algorytmu
let tokenHeader = token.split('.')[0];
tokenHeader = JSON.parse(base64url.decode(tokenHeader));
console.log('Nagłówek przed zmianą algorytmu: ');
console.log(tokenHeader);
console.log('Nagłówek po zmianie algorytmu: ');
tokenHeader.alg = 'none';
console.log(tokenHeader);
const newHeader =base64url(JSON.stringify(tokenHeader));
token = newHeader + "." + token.split('.')[1] + "." +
token.split('.')[2];
// Po ingerencji w token
const decodedToken2 = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken2);

console.log('Zweryfikowany token: ');
try {
  const verifiedToken2 = jwt.verify(token, privateKey);
  console.log(verifiedToken2);
} catch(err) {
  console.log('Błąd weryfikacji: ' + err.message);
}
```

Wykorzystana w przykładzie wersja biblioteki okazała się odporna na opisywany atak, natomiast w innych bibliotekach lub własnych implementacjach taki atak może zakończyć się powodzeniem. Wynik działania programu przedstawiono na Rysunku 4.

```
$ node app.js
Zawartość tokenu:
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Zweryfikowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Nagłówek przed zmianą algorytmu:
{ typ: 'JWT', alg: 'HS256' }
Nagłówek po zmianie algorytmu:
{ typ: 'JWT', alg: 'none' }
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587296819 }
Zweryfikowany token:
Błąd weryfikacji: invalid signature
```

Rysunek 4: Zrzut z konsoli prezentującej wynik działania programu

Użyta w przykładzie biblioteka „jsonwebtoken" jest odporna na zmianę rodzaju algorytmu podpisu, nie oznacza to, że inne biblioteki operujące na JWT będą równie dobrze zabezpieczone przed tym rodzajem błędów. Skutkiem ignorowania podpisu może być całkowite przejęcie konta użytkownika. Napastnik może dowolnie zmienić ładunek JWT bez wykrycia tego przez mechanizmy weryfikujące.

Kolejnym problemem może być błędna implementacja procesu weryfikacji, która polega na zastosowaniu nieodpowiedniej metody. JWT jest tylko dekodowany bez weryfikowania czy nie został zmodyfikowany przez użytkownika. W celu zobrazowania podatności wykorzystano środowisko NodeJS w wersji 10.16.3, oraz bibliotekę „jsonwebtoken" w wersji 8.5.1. Poniżej przedstawiono kroki, których wykonanie umożliwia zobrazowanie i zrozumienie przyczyn występowania podatności. Pierwszym krokiem jest utworzenie projektu, w tym celu wykonujemy polecenie npm init -y, wynik polecenia zobrazowano na Rysunku 5.

```
$ npm init -y
wrote to J:\aplikacja\vulnerability_1\package.json:

{
  "name": "vulnerability_1",
  "version": "1.0.0",
  "descriprion": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Rysunek 5: Utworzenie projektu w NodeJS

Następnie należy zainstalować bibliotekę wspomagającą implementację JWT. Proces instalacji biblioteki zobrazowano na Rysunku 6.

```
$ npm install jsonwebtoken
+ jsonwebtoken@8.5.1
added 15 packages from 10 contributors and audited 17 packages in 3.773s
found 0 vulnerabilities
```

Rysunek 6: Instalacja biblioteki „jsonwebtoken

W celu zmiany zawartości payloadu skorzystano z biblioteki „base64url”. Rysunek 7 zawiera szczegóły instalacji.

```
$ npm i base64url
+ base64url@3.0.1
added 1 packages from 1 contributors and audited 18 packages in 4.61s
found 0 vulnerabilities
```

Rysunek 7: Instalacja biblioteki base64url

Kod programu znajdujący się na Listingu 3 tworzy nowy JWT, wykonuje jego dekodowanie i weryfikację, następnie zawartość payloadu jest zmieniana i ponownie wykonywany jest proces dekodowania i weryfikacji. Użycie dekodowania, które nie weryfikuje podpisu a jedynie odczytuje zawartość zamiast weryfikacji umożliwia atakującemu zmianę danych przesyłanych w tokenie.

Listing 3: Kod JavaScript przedstawiający różnicę pomiędzy weryfikacją a dekodowaniem

```
const jwt = require('jsonwebtoken');
const base64url = require('base64url');

const privateKey = '123456789abcdefgh';

//Utworzenie podpisanego tokenu
let token = jwt.sign({ id: '1234', name: 'Jan Nowak', admin: false },
privateKey, { algorithm: 'HS256'});

// Przed ingerencją w token
const decodedToken = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken);

const verifiedToken = jwt.verify(token, privateKey);
console.log('Zweryfikowany token: ');
console.log(verifiedToken);

//Zmiana payloadu
let tokenPayload = token.split('.')[1];
tokenPayload = JSON.parse(base64url.decode(tokenPayload));
tokenPayload.admin = true;

const newPayload =base64url(JSON.stringify(tokenPayload));
token = token.split('.')[0]+ "." + newPayload + "." +
token.split('.')[2];

// Po ingerencji w token
const decodedToken2 = jwt.decode(token, privateKey);
console.log('Zdekodowany token: ');
console.log(decodedToken2);

console.log('Zweryfikowany token: ');
try {
   const verifiedToken2 = jwt.verify(token, privateKey);
   console.log(verifiedToken2);
} catch(err) {
   console.log('Błąd weryfikacji: ' + err.message);
}
```

Dekodowanie nie weryfikuje podpisu, dopiero proces weryfikacji zwraca błąd i uniemożliwia eskalację uprawnień. Wynik działania programu został przedstawiony na Rysunku 8.

```
$ node app.js
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587293399 }
Zweryfikowany token:
{ id: '1234', name: 'Jan Nowak', admin: false, iat: 1587293399 }
Zdekodowany token:
{ id: '1234', name: 'Jan Nowak', admin: true, iat: 1587293399 }
Zweryfikowany token:
Błąd weryfikacji: invalid signature
```

Rysunek 8: Wynik porównania weryfikacji i dekodowania

Zastosowanie dekodowania zamiast weryfikacji może mieć poważne skutki, atakujący może zmienić dane dostarczane w tokenie i zwiększyć uprawnienia lub podszyć się pod innego użytkownika systemu. Wystąpienie podatności może być wynikiem braku znajomości zewnętrznych bibliotek. Dokumentacja wyraźnie opisuje skutki wybrania funkcji dekodowania zamiast weryfikacji, Rysunek 9 przedstawia fragment dokumentacji biblioteki „jsonwebtoken”.

jwt.decode(token [, options])

(Synchronous) Returns the decoded payload without verifying if the signature is valid.

> **Warning:** This will **not** verify whether the signature is valid. You should **not** use this for untrusted messages. You most likely want to use `jwt.verify` instead.

Rysunek 9: Fragment dokumentacji biblioteki „jsonwebtoken” [10]

Jeżeli metoda weryfikacji zostanie poprawnie wybrana oraz zastosuje się odpowiednie zabezpieczenia przed zmianą algorytmu na „none” można napotkać na kolejną trudność polegającą na odpowiednim doborze klucza. Podpis JWT składa się z nagłówka, paylodu oraz tajnego klucza przechowywanego po stronie serwera, wybranie klucza o krótkiej długości może spowodować szybkie jego złamanie. Wszystkie dane potrzebne do próby złamania tajnego klucza znajdują się po stronie użytkownika, czyli osoby atakującej. Wykrycie próby forsowania klucza nie jest możliwe przez serwer ponieważ wszystkie operacje wykonywane są poza serwerem. Przykładowym narzędziem umożliwiającym łamanie tajnego klucza JWT jest narzędzie JWT cracker [11].

W celu zademonstrowania problemu przygotowano JWT ze słabym kluczem tajnym o wartości 1234. W celu przygotowania JWT użyto narzędzia dostępnego pod adresem https://jwt.io/, proces tworzenia JWT zaprezentowano na Rysunku 10.

Rysunek 10: Przygotowanie JWT ze słabym kluczek tajnym

Skrypt umożliwiający złamanie klucza, wykorzystuje narzędzie JWT cracker oraz wyświetla czas rozpoczęcia i czas zakończenia procesu. Treść skryptu została przedstawiona na Listingu 4.

Listing 4 Skrypt łamiący klucz tajny JWT

```
#!/bin/sh
# Skrypt łamiący klucz tajny JWT
date +"%T"
./jwtcrack eyJhbGciOiJIUzI1NiI
sInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3
ODkwIiwibmFtZSI6IkphbiBOb3dhayIsImlhdCI6MTUxNjIzO
TAyMn
0.JuFvEJoJk9BoBZD3vT5NCy5y4SOALp9Pp8mwpvLO3k
date +"%T"
```

Skrypt po czasie nie dłuższym niż dwie minut zwrócił tajny klucz. Wynik działania programu przedstawiono na Rysunku 11.

Rysunek 11: Wynik skryptu łamiącego klucz tajny

Fakt, że dzięki JWT możemy zrealizować uwierzytelnienie bez komunikacji z bazą danych to podstawa kolejnego problemu bezpieczeństwa. Jeżeli tokeny nie będą miały określonych krótkich dat ważności oraz w mechanizmie weryfikującym ważność tokenu nie będzie właściwie sprawdzana, nie będzie możliwa natychmiastowa zmiana uprawnień lub zablokowanie użytkownika. Częstym rozwiązaniem tego problemu jest generowanie dwóch tokenów jeden, który zawiera dane o użytkowniku (token uwierzytelniający) z krótkim czasem ważności i drugi, tak zwany token odświeżającym, który zapisywany jest w bazie danych. Token odświeżający służy do pozyskania nowego tokenu autoryzującego w chwili jego unieważnienia.

6. Autoryzacja oraz uwierzytelnianie

Opisane poniżej podatności związane są z niepoprawnym wdrożeniem mechanizmów uwierzytelniania oraz autoryzacji.

Szczegóły podatności zostały zademonstrowane w oparciu o aplikację REST API. Aplikacja została opracowaną w technologii NodeJS przy wykorzystaniu frameworka Express.

6.1. Autoryzacja na poziomie obiektu

Autoryzacja na poziomie obiektu dotyczy sytuacji, w której użytkownik jest uwierzytelniony i próbuje uzyskać dostęp do konkretnego obiektu. Decyzja o przyznaniu lub odmowie dostępu podejmowana jest na podstawie zapisanych w bazie danych poświadczeń. Atakujący próbuje modyfikować parametry zapytania np. identyfikator obiektu w ten sposób aby w odpowiedzi zostały zwrócone dane, które nie powinny być mu zaprezentowane.

Napastnik zarejestrował się oraz uwierzytelnił, następnie przeprowadza rekonesans, w wyniku którego znalazł punkt końcowy odpowiedzialny za pobranie informacji na temat jego profilu. Żądanie pobierające dane użytkownika przedstawia Listing 5.

Listing 5 Żądanie pobrania informacji o użytkowniku

```
GET /api/users/v1/1 HTTP/1.1
Host: localhost:3010
Accept: application/json, text/plain, */*
Content-Type: application/json
```

W odpowiedzi na powyższe żądanie serwer zwraca informacje na temat użytkownika, odpowiedź serwera przedstawia Listing 6.

Listing 6 Odpowiedź zawierająca dane użytkownika

```
HTTP/1.1 200 OK
[Nagłówki]

{
  "id":1,
  "email": "jan@niepodam.pl",
  "first_name": "Jan",
  "last_name": "Kowalski",
  "id_number": "ABC12345",
  "pesel": "12345612345",
  "phone_number": "111 222 333",
  "password":
"$2a$10$.Ci9NPsLJrppf.81MOoa.uXKghsnsMQcQpc62Zk53szCsnp
Yc1Xf2",
  "active": false,
  "created_at": "2020-05-16T09:31:00.069Z"
}
```

Identyfikator przesyłany w żądaniu ma wartość 1, jest to identyfikator użytkownika, podając inną wartość parametru proces autoryzacji powinien zwrócić odpowiedź z informacją o braku uprawnień. Ponieważ autoryzacja nie została odpowiednio zaimplementowana podanie identyfikatora należącego do innego użytkownika spowoduje przetworzenie zapytania i zwrócenie odpowiedzi z pełnymi danymi. Rysunek 12 przedstawia zmodyfikowane zapytanie wysłane przy pomocy programu Burp Suite.

Rysunek 12 Zmodyfikowane żądanie

W odpowiedzi na żądanie została zwrócona odpowiedź zawierająca dane innego użytkownika (Rysunek 13), dzieje się tak, ponieważ serwer polega na pa-

rametrach, które przesyła użytkownik, a nie na jego uprawnieniach.

```
 1 HTTP/1.1 200 OK
 2 X-DNS-Prefetch-Control: off
 3 X-Frame-Options: SAMEORIGIN
 4 Strict-Transport-Security: max-age=15552000; includeSubDomains
 5 X-Download-Options: noopen
 6 X-Content-Type-Options: nosniff
 7 X-XSS-Protection: 1; mode=block
 8 Content-Type: application/json; charset=utf-8
 9 Content-Length: 603
10 ETag: W/"25b-oTcm2C63x+a6FYj3FmIaD2EM1Hw"
11 Date: Thu, 21 May 2020 14:26:00 GMT
12 Connection: keep-alive
13
14 {
     "response":{
       "id":2,
       "email":"michal@niepodam.pl",
       "first_name":"Michał",
       "last_name":"Nowak",
       "id_number":"CBD 23456",
       "pesel":"77777766666",
       "phone_number":"222 333 444",
       "password":"$2a$10$2jE5LtLpvCICjQvWx/04X.9IjHykypDuo6bNZDNUpOKkRBGHGNd8C",
       "active":false,
       "created_at":"2020-05-16T09:34:43.799Z",
       "isAdmin":false
     },
     "sql":"{ method: 'select',\n        options: {},\n        timeout: false,\n
```

Rysunek 13 Odpowiedź na zmodyfikowane żądanie

Mechanizmy kontrolujące dostęp użytkowników do obiektów zazwyczaj są mechanizmami złożonymi, przez co są bardziej podatne na luki bezpieczeństwa. Należy bezwzględnie zweryfikować, czy mechanizm autoryzacji bazuje na uprawnieniach użytkowników.

6.2. Błędy w uwierzytelnianiu użytkowników

Punkty końcowe odpowiedzialne za proces uwierzytelniania są dostępne na zewnątrz, przez co często są pierwszym wektorem ataku. W przypadku przełamania uwierzytelniania atakujący może przejąć konto, które zawiera poufne dane oraz uprawnia do wykonywania operacji modyfikujących lub trwale usuwających dane.

Przykładowym problemem, który można zaliczyć do tej kategorii jest brak odpowiedniej polityki bezpieczeństwa haseł, Podczas rejestracji akceptowalne są krótkie hasła o małej złożoności w wyniku czego możliwe jest przeprowadzenie skutecznego ataku siłowego. Listing 7 zawiera treść zapytania, umożliwiającego uwierzytelnienie użytkownika.

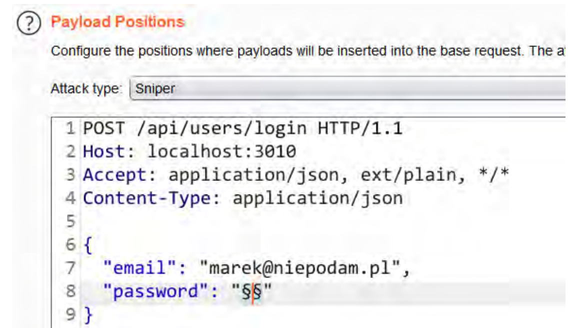
Listing 7 Treść zapytania wykorzystywanego do uwierzytelnienie

```
POST /api/users/login HTTP/1.1
Host: localhost:3010
Accept: application/json, text/plain, */*
Content-Type: application/json

{
  "email": "jan@niepodam.pl",
  "password": "*******"
}
```

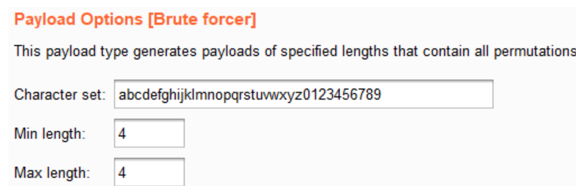
W ciele żądania przekazywany jest adres e-mail oraz hasło użytkownika, po odebraniu żądania przez serwer następuje weryfikacja i podejmowana jest decyzja o uwierzytelnieniu. Uwierzytelnienie polega na zwróceniu JWT, który będzie dołączany do kolejnych żądań w celu potwierdzenia tożsamości.

Do przeprowadzenia ataku siłowego na hasło można wykorzystać narzędzie Intruder z pakietu Burp Suite. Rysunek 14 przedstawia odpowiednio przygotowaną treść zapytania.



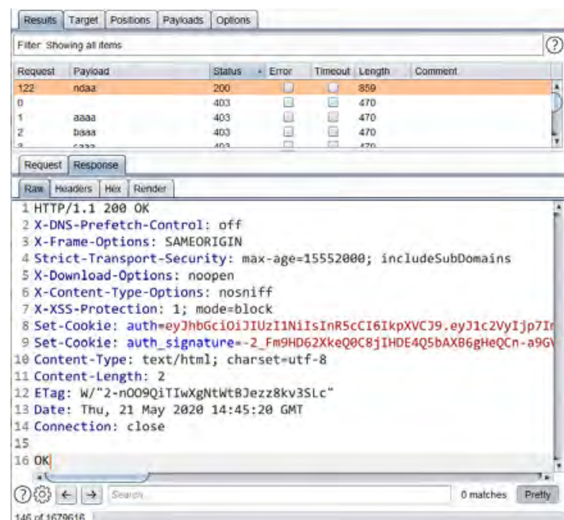
Rysunek 14 Treść sparametryzowanego żądania uwierzytelnienia

W miejscu hasła zostaną wstawione wartości zdefiniowane w opcjach przedstawionych na Rysunku 15.



Rysunek 15 Definicja wartości parametrów wstawianych w żądaniu

W wyniku przeprowadzonego ataku siłowego zostało odgadnięte hasło, które umożliwi przejęcie konta użytkownika i wykonywanie operacji w jego imieniu. Rysunek 16 prezentuje wynik ataku. Hasło użytkownika ma wartość „ndaa”.



Rysunek 16 Wynik przeprowadzonego ataku siłowego na hasło

Pomocnicze metody takie jak, resetowanie hasła powinny być równie dobrze zabezpieczone jak metoda uwierzytelniająca. Aplikacja realizuje odzyskiwanie haseł przez wysłanie tokenów na adres e-mail powiązany z kontem. Możliwy jest scenariusz, w którym atakujący rejestruje się w systemie, następnie resetuje hasło i ocenia, że token resetowania hasła zbudowany jest z 6 cyfr. Słaba złożoność tokenu umożliwia atakującemu przeprowadzenie ataku siłowego na token i odzyskanie hasła do konta innego użytkownika.

Podczas projektowania systemu należy zdefiniować wymagania dotyczące złożoności haseł oraz tokenów. W celu odpowiedniego określenia wymagań można posłużyć się wytycznymi przygotowanymi przez Natio-

nal Institute of Standards and Technology (NIST). Według zaleceń hasło powinno składać się z minimum 8 znaków, nie powinno być zbudowane z podciągu nazwy aplikacji. Hasło przed ustawieniem powinno być weryfikowane ze słownikiem najpopularniejszych haseł. Nie powinno stawiać się wymagań odnośnie występowania znaków specjalnych, natomiast dobrą praktyką jest umieszczenie miernika siły hasła [12]. Dodatkowym zaleceniem poprawiającym bezpieczeństwo w zakresie uwierzytelniania jest wprowadzenie podwójnej weryfikacji.

6.3. Problem z autoryzacją na poziomie funkcji

Atakujący w wyniku rekonesansu znalazł punkt końcowy, który powinien być udostępniany tylko administratorom, okazało się, że punkt końcowy przy wykorzystaniu metody GET nie implementuje kontroli autoryzacji. Każdy kto pozna adres zasobu, który nie wdraża autoryzacji może wysłać żądanie dzięki któremu otrzyma dane.

Znanym przykładem problemów z autoryzacją na poziomie funkcji jest podatność znaleziona w aplikacji wykorzystywanej do przeprowadzania głosowania w Izraelu [13]. Serwer zwracał odpowiedź zawierającą poufne dane bez konieczności autoryzacji użytkownika wysyłającego żądanie. Adres zasobu o postaci: „/get-admin-users", został znaleziony w źródłach strony. Odpowiedź na żądanie pod znalezionym adresem zwracała dane administratorów systemu, część odpowiedzi została zaprezentowana na Rysunku 17.


```
96      {
97          "oid": 86,
98          "name": "הליכוד לכנסת",
99          "password": "",
100         "email": "",
101         "campaignName": "כנסת 23 ליכוד",
102         "maxUsers": 10000,
103         "businessId": "",
104         "businessName": "",
105         "expirationDate": "19-01-2024 ",
106         "mainOfficeAddress": "",
107         "cost": null,
108         "purchasedSms": 710000,
109         "remainingSms": 340788,
110         "phone": "05
111         "creationTime": "2020-01-19",
112         "usersCount": 100,
113         "adminId": "",
114         "allowExcelUpload": false,
115         "active": true,
116         "dataEncrypted": false
117     }
```


Rysunek 17 Poufne dane zwrócone bez wymaganej autoryzacji [13]

Ze względu na uporządkowanie API budowanego w architekturze REST odnalezienie punktów dostępów jest łatwym zadaniem nawet dla osób niedoświadczonych. Przed wdrożeniem aplikacji należy upewnić się, że wszystkie punkty końcowe zwracające poufne dane mają zaimplementowany i włączony mechanizm autoryzacji.

7. Wnioski

Na etapie projektowania powinny zostać określone wszystkie punkty końcowe wymagające uwierzytelnienia. Wdrażając mechanizmy uwierzytelnienia zaleca się korzystanie ze sprawdzonych standardów, podczas prób implementacji własnych rozwiązań istnieje wysokie prawdopodobieństwo popełnienia błędów. Punkty końcowe weryfikujące tożsamość użytkownika powinny implementować mechanizmy zabezpieczające przed atakami siłowymi, przykładem mechanizmu może być np. captcha. W systemie istnieje zazwyczaj kilka miejsc, są to miedzy innymi: resetowanie hasła, zmiana adresu e-mail, tak samo wrażliwych jak uwierzytelnianie, w tych miejscach warto wdrożyć podobne standardy. Jeżeli jest to możliwe warto zaimplementować mechanizm podwójnego uwierzytelnienia [14].

W celu ograniczenia problemów występujących podczas autoryzacji, należy zaprojektować proces z uwzględnieniem złożonej hierarchii użytkowników. Mechanizm autoryzacji powinien być wprowadzony na poziomie funkcji oraz każdego obiektu. Dobrą praktyką ograniczającą możliwości wysłania niepożądanych zapytań jest zastosowanie identyfikatorów GUID zamiast identyfikatorów, które pozwalają na odgadnięcie kolejnych wartości. Domyślnie przyznawane uprawnienia powinny być możliwie najmniejsze, uprawnienia należy zwiększać adekwatnie do potrzeb z wyraźną akceptacją przez uprawnione osoby. Mechanizm autoryzacji powinien być testowany automatycznie, testy powinny być wykonywane po każdej ingerencji w działanie mechanizmu [14].

Bezpieczna implementacja JWT możliwa jest wyłącznie przy pełnym zrozumieniu budowy i sposobów wykorzystania. Na etapie projektowania należy wybrać odpowiednie algorytmy podpisu, a w przypadku JWE także algorytmy szyfrowania. Podczas wdrożenia powinno się używać silnych kluczy kryptograficznych. Bezwzględnie należy zabezpieczyć system przed akceptacją algorytmu podpisu „none" i zwrócić szczególną uwagę na poprawność weryfikacji podpisu. Do wdrożenia JWT powinno używać się gotowych i sprawdzonych bibliotek, ponieważ własna implementacja jest narażona na błędy bezpieczeństwa. Przed wykorzystaniem biblioteki powinno zapoznać się z jej dokumentacją oraz wdrożyć monitorowanie wykrytych podatności i procedury aktualizacji. Poza ustawianiem krótkiego czasu ważności tokenów system powinien umożliwiać ręczne ich unieważnienie [1].

Należy zauważyć, że satysfakcjonujący poziom bezpieczeństwa można osiągnąć tylko przez spełnienie wszystkich wymogów bezpieczeństwa. Zaniedbanie któregokolwiek z nich powoduje powstanie luki, przez którą atakujący może uzyskać dostęp do danych lub funkcji.

8. Literatura

[1] Praca zbiorowa, Bezpieczeństwo aplikacji webowych, Securitum, Kraków 2019.

[2] J. S. Karsun, Solutions LLC Microservices API Security https://www.ijert.org/research/microservices-api-security-IJERTV7IS010137.pdf

[3] J. H. Saltzer, M. D. Schroeder, The Protection of Information in Computer Systems http://web.mit.edu/Saltzer/www/publications/protection/index.html

[4] OWASP Application Security Verification Standard, https://owasp.org/www-project-application-security-verification-standard/ [28.04.2020]

[5] OWASP Proactive Controls, https://owasp.org/www-project-proactive-controls/ [06.05.2020]

[6] Opis architektury REST, https://restfulapi.net/ [9.04.2020] .

[7] Porównanie popularność architektury REST i protokołu SOAP https://trends.google.com/trends/explore?date=all&q=REST%20API,%2Fm%2F077dn [18.04.2020].

[8] Specyfikcja JWT, https://tools.ietf.org/html/rfc7519 [11.04.2020].

[9] Informacje o JWT, https://jwt.io/introduction/ [11.04.2020]

[10] Opis biblioteki jsonwebtoken, https://www.npmjs.com/package/jsonwebtoken [11.04.2020].

[11] Narzędzie JWT cracker, https://github.com/brendan-rius/c-jwt-cracker [28.04.2020].

[12] Zalecenia dotyczące polityki bezpieczeństwa hasła https://pages.nist.gov/800-63-3/sp800-63b.html#sec5 [06.05.2020].

[13] Opis podatności związanej z brakiem uwierzytelniania https://dzone.com/articles/api-security-weekly-issue-70 [06.05.2020].

[14] Opis najbardziej krytycznych podatności API, https://owasp.org/www-project-api-security/ [11.04.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 261–268
Received: 20 June 2020
Accepted: 30 June 2020

Analysis of security CMS platforms by vulnerability scanners

Badanie bezpieczeństwa wybranych platform CMS za pomocą skanerów podatności

Patryk Zamościński*, Grzegorz Kozieł

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

Abstract
Subject of security the most popular CMS platforms has been undertaken in the following thesis. There were introduced fundamental informations about subject CMS platforms and vulnerability scanners utilised to research. For research purposes Wordpress and Joomla websites were created and investigated for security by vulnerability scanners OWASP ZAP, Vega, Detectify and Skipfish. Results were grouped by some criteria: vulnerabilities by category and vulnerabilities by threat level. Obtained results were examined in two ways: analysis of residual results, for each website scanning and analysis of aggregated results from all scanners. After that, conclusions about CMS platforms security have been drawn.

Keywords: CMS; security; vulnerability scanner

Streszczenie
W niniejszej pracy został podjęty temat bezpieczeństwa najbardziej popularnych platform CMS. Przedstawiono podstawowe informacje o badanych platformach CMS oraz o skanerach podatności użytych w badaniach. W celach badawczych zostały utworzone witryny Wordpress i Joomla, które następnie przebadano pod kątem bezpieczeństwa za pomocą skanerów OWASP ZAP, Vega, Detectify oraz Skipfish. Wyniki zostały pogrupowane według kryteriów: podatności w poszczególnych kategoriach oraz podatności według poziomu zagrożenia. Wyniki były rozpatrywane na dwa sposoby: analiza wyników szczątkowych, oraz analiza zbiorcza zsumowanych wyników ze wszystkich skanerów. Na koniec zostały wyciągnięte wnioski w odniesieniu do obu platform.

Słowa kluczowe: CMS; bezpieczeństwo; skaner podatności

*Corresponding author

Email address: patryk.zamoscinski@pollub.edu.pl (P. Zamościński), g.koziel@pollub.pl (G. Kozieł)

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

1. Wstęp

W ostatnich dekadach nastąpił gwałtowny rozwój Internetu. Co się z tym wiąże, duża część życia społecznego została przeniesiona do sieci. W dzisiejszym świecie bez wychodzenia z domu można zrobić wiele rzeczy, poczynając od rozrywki i realizacji swoich pasji (fora internetowe skupiające entuzjastów, blogi), poprzez robienie zakupów a kończąc na załatwianiu spraw urzędowych czy pracy zdalnej.

Zagadnienia związane z bezpieczeństwem w sieci mają kluczowe znaczenie w dzisiejszym świecie. Dane, które podajemy podczas rejestracji w różnego rodzaju stronach internetowych przedstawiają dużą wartość dla cyberprzestępców. Może się wydawać, że jedyne miejsca, gdzie cyberbezpieczeństwo jest szczególnie ważne to aplikacje bankowe i poczta e-mail. Jednak każda strona, która posiada jakieś informacje o swoich użytkownikach jest potencjalnym celem atakującego.

W dzisiejszych czasach systemy CMS są bardzo popularne ze względu na to, że nie wymagają od użytkownika umiejętności programistycznych przy tworzeniu strony WWW. Statystyki pokazują, że aż 46% aplikacji internetowych posiada podatności, które mają krytyczne znaczenie dla bezpieczeństwa [1]. Dlatego autor tej pracy postanowił sobie za cel zbadanie dwóch najpopularniejszych platform CMS (WordPress i Joomla) dostępnych na rynku, opisanie wykrytych luk w zabezpieczeniach oraz wyłonienie bezpieczniejszej platformy.

2. Przegląd literatury

W pracy [2] autorzy porównali pod kątem bezpieczeństwa platformy Drupal i Joomla. Proces badawczy składał się z 4 etapów: instalacja i konfiguracja witryn, przeprowadzenie prostych, zautomatyzowanych skanów, analiza plików źródłowych oraz wykonanie ukierunkowanych ataków. Wyniki przedstawiono w postaci tabeli zawierającej kategorie bezpieczeństwa z wyszczególnionymi kryteriami, oraz oceną tych kryteriów.

Szersze podejście do badania bezpieczeństwa współczesnych stron internetowych można odnaleźć w pracy [1]. Autorzy użyli skanera Acunetix Online do przebadania losowo wybranych 10 000 stron internetowych. Z przeprowadzonych badań wysnuto wniosek, że w 35% stron internetowych wykryto co najmniej jedną podatność o wysokim stopniu zagrożenia.

W trakcie przeszukiwania zbioru pozycji literaturowych odnaleziono prace traktujące o bezpieczeństwie systemów typu CMS [3, 4, 5, 6]. Okazało się, że praca Comparative Analysis Of Web Security In Open Source Content Management System [3] porusza tą samą tematykę co niniejsza praca, z tym że autorzy porównywali jeszcze platformę Drupal.

Po przeprowadzeniu badań literaturowych stwierdzono, że co prawda poruszony temat był badany, jednakże nie przeprowadzono dogłębnych badań, które naświetliłyby luki w zabezpieczeniach badanych systemów CMS i dałyby wyczerpującą odpowiedź na pytania o bezpieczeństwo tych systemów.

3. Metodyka badawcza

Badania zostały przeprowadzone w następujący sposób: na serwerze lokalnym zostały utworzone dwie witryny, jedna stworzona za pomocą platformy WordPress a druga za pomocą Joomla. Na potrzeby badania skanerem Detectify należało stworzyć witryny na zewnętrznym hostingu, identyczne, jak te wykorzystywane w pozostałych badaniach. Obie witryny zawierają podstawowe elementy platform CMS takie jak: podstrony, posty, formularze logowania, pole „szukaj", kategorie oraz tagi. Następnie tak przygotowane witryny zostały poddane skanowaniu pod kątem bezpieczeństwa za pomocą wybranych skanerów bezpieczeństwa.

Wyniki zostały pogrupowane według kryteriów: liczba podatności w poszczególnych kategoriach oraz liczba podatności dla każdego poziomu zagrożenia. Otrzymane wyniki były rozpatrywane na dwa sposoby: analiza wyników szczątkowych, po każdym skanowaniu witryny, oraz w podsumowaniu analiza zbiorcza zsumowanych wyników otrzymanych ze wszystkich skanerów.

4. Badane platformy CMS

4.1. WordPress

Jak pokazują badania, WordPress jest aktualnie najbardziej popularnym systemem CMS na świecie z udziałami w rynku rzędu 63,3% [6]. Aż 36,4% wszystkich stron internetowych w jakiś sposób używa WordPress [6]. Jeśli chodzi o platformy CMS, to WordPress jest niekwestionowanym liderem na rynku. udostępnia szeroki wachlarz możliwości tworzenia stron internetowych, od zwykłych witryn, aplikacji internetowych, po rozbudowane serwisy, a nawet sklepy internetowe.

Swoją popularność WordPress zawdzięcza głównie bardzo dużej i aktywnie działającej społeczności. Z jednej strony społeczność pomaga początkującym użytkownikom służąc poradami i wychwytuje różnego rodzaju błędy platformy. Z drugiej strony społeczność to twórcy, którzy są odpowiedzialni za tworzenie rozszerzeń do WordPress, takich jak wtyczki lub motywy. To właśnie wtyczki są największą zaletą tego CMS. Ogromna (największa spośród wszystkich platform CMS) liczba rozszerzeń pozwala na modyfikację każdego aspektu witryny internetowej [4]. Liczba tylko bezpłatnych wtyczek szacowana jest na ponad 10000 i cały czas tworzone są nowe.

WordPress ma bardzo dużo różnego rodzaju funkcjonalności, dodatkowo rozszerzanych przez wtyczki, jednak warto przedstawić najważniejsze z nich [7,8]:

- możliwość definiowania własnych taksonomii, czyli kategorii i tagów,
- ochrona przed spamem,
- zarządzanie użytkownikami,
- możliwość zabezpieczenia hasłem konkretnego postu lub strony,
- łatwy import zasobów,
- inteligentne formatowanie tekstu,
- zarządzanie przepływem pracy,
- dostęp do panelu administracyjnego z aplikacji mobilnej,
- wyszukiwarka,
- przyjazne dla użytkownika adresy URL,
- przetłumaczony na ponad 180 języków,
- interfejs XML-RPC umożliwiający wymianę danych z innymi systemami.

4.2. Joomla

Drugim co do popularności systemem CMS jest Joomla. Zgodnie z raportem firmy w3techs 2.4% wszystkich współczesnych stron internetowych używa tej platformy, natomiast udziały w rynku tego CMS są na poziomie 4.1% [6].

Na popularność tej platformy składa się kilka czynników, jednym z nich jest mnogość zastosowań. Joomla na początku był używany głównie do tworzenia witryn publicystycznych, takich jak blogi czy portale informacyjne, jednak wraz z rozwojem platformy zostały dodane funkcjonalności umożliwiające tworzenie rozbudowanych portali, stron internetowych, korporacyjnych sieci intranet lub extranet a także serwisów eCommerce. Poza tym istnieje możliwość używani przez deweloperów pełnoprawnego szkieletu projektowego Joomla Framework, co jeszcze poszerza zakres możliwości wykorzystania tej platformy. Joomla cechuje się elastycznością, liczne rozszerzenia i szablony pozwalają dostosować witrynę do określonych wymagań.

O wysokiej pozycji Joomla na tle innych platform CMS decyduje duża liczba udostępnianych funkcjonalności [7,9]:

- zarządzanie użytkownikami i ich uprawnieniami,
- wyszukiwarka,
- elastyczność w tworzeniu różnego rodzaju witryn,
- duża liczba rozszerzeń (ponad 8000),
- zarządzanie menu,
- zarządzanie kategoriami,
- zarządzanie poszczególnymi modułami witryny,
- zarządzanie ustawieniami pamięci podręcznej cache,
- zarządzanie szablonami,
- zarządzanie banerami i reklamami,
- zaawansowany edytor tekstowy,
- rozszerzona dokumentacja dla developerów,
- tłumaczenie na 70 języków.

5. Wykorzystane skanery podatności

5.1. OWASP ZAP

OWASP Zed Attack Proxy (OWASP ZAP) jest najczęściej używanym przez testerów skanerem podatności. Ze względu na prostotę w instalacji i użytkowaniu ZAP może być używany zarówno przez początkujących, jak i zaawansowanych testerów. OWASP ZAP ma postać aplikacji desktopowej dostępnej na platformach

Windows, Linux oraz macOS. Jest to oprogramowanie typu open source rozwijane w przez organizację OWASP.

OWASP ZAP posiada funkcjonalności przydatne dla każdego profesjonalnego testera bezpieczeństwa:

- lokalne proxy - pozwala przechwytywać i modyfikować przesyłane żądania,
- pająk – narzędzie służące do przeszukiwania witryny i znajdowania powiązań,
- pająk AJAX – do przeszukiwania witryny wykorzystuje Crawjax, aby odkryć wszystkie powiązania stron [10],
- automatyczne skanowanie – w wersji pasywnej (analiza wysłanych i odebranych żądań) oraz aktywnej (preparowanie i wysyłanie żądań do witryny),
- fuzzer – umożliwia wysyłanie do celu dużej ilości niepoprawnych i niespodziewanych danych. Użytkownik wybiera jakie dane zostaną wysłane, można też definiować własne zestawy danych [10].

5.2. Vega

Vega jest to projekt open source firmy Subgraph zajmującej się cyberbezpieczeństwem. Vega jest aplikacją desktopową, posiada GUI stworzone w języku Java oraz działa na systemach Windows, Linux oraz macOS [11].

Vega posiada funkcjonalności pozwalające na kompleksowe przebadanie danej witryny:

- rozszerzalność – Vega udostępnia API do tworzenia własnych scenariuszy ataku [11],
- zaawansowany automatyczny skaner,
- proxy dedykowane do obserwacji i ingerencji w przesyłane żądania [11],
- skaner proxy – umożliwia uruchomienie modułów ataku podczas gdy użytkownik przegląda stronę za pomocą proxy. Umożliwia to wykonywanie połowicznie automatycznych testów bezpieczeństwa, aby zapewnić możliwie jak największe pokrycie kodu [11].

5.3. Detectify

Detectify został założony w 2013 roku. Jest to projekt komercyjny, udostępniający 14-dniowy okres próbny. Detectify jako jedyny skaner z zestawienia jest opartą na SaaS (Software as a Service) aplikacją internetową [12]. Według twórców ten skaner testuje strony internetowe pod kątem obecności ponad 1500 podatności, w tym tych z zestawienia OWASP Top 10.

Jako jedyny z używanych skanerów Detectify stosuje weryfikację, czy domena należy do użytkownika. Aby potwierdzić własność witryny, użytkownik musi pobrać plik tekstowy i wysłać go na serwer.

Detectify udostępnia swoim użytkownikom wiele ciekawych funkcjonalności, które pomagają monitorować stan bezpieczeństwa strony internetowej:

- skaner sprawdzający strony na obecność ponad 1500 podatności [12],
- aktualizowane testy bezpieczeństwa tworzone przez etycznych hakerów [12],
- monitorowanie zasobów – pozwala na monitorowanie witryny pod kątem bezpieczeństwa i - jeśli zostanie wykryta podejrzana działalność - użytkownik jest o tym powiadamiany,
- skanowanie cykliczne,
- rozbudowane statystyki.

5.4. Skipfish

Skipfish to skaner podatności open source stworzony przez Michała Zalewskiego a udostępniany przez Google. Skipfish nie ma interfejsu użytkownika, jest przeznaczony do pracy w środowisku Linux, Windows oraz macOS, jako aplikacja konsolowa [13]. Twórca przyznaje, że Skipfish nie spełnia wielu wymagań stawianych przed skanerami podatności, jednak to narzędzie rozwiązuje pewne niedogodności spotykane w innych skanerach i może być pomocne w naprawdę wielu przypadkach [13].

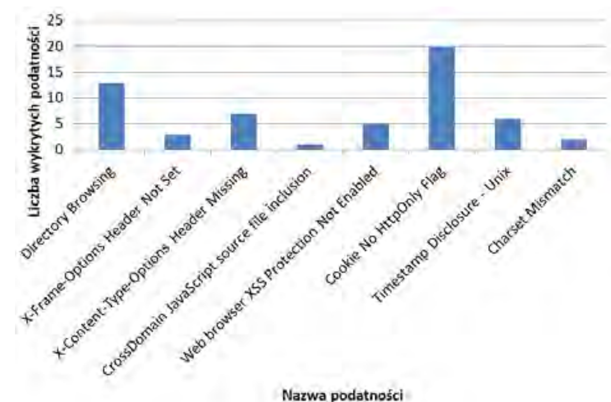
Mimo że Skipfish nie jest w pełni kompleksowym narzędziem to jednak zawiera wiele funkcjonalności, które wyróżniają ten skaner na tle innych:

- wysoka wydajność – wysyłanie ponad 500 żądań na sekundę do celów dostępnych przez Internet, ponad 2000 do celów dostępnych przez LAN/MAN oraz ponad 7000 do celów dostępnych lokalnie sprawiają, że Skipfish jest bardzo wydajnym narzędziem [13],
- multipleksowanie, działanie asynchroniczne oraz model przetwarzania danych sprawiają, że Skipfish zużywa relatywnie mało zasobów [13],
- łatwość w użyciu,
- dobrze zaprojektowane testy bezpieczeństwa [13].

6. Wyniki badań

6.1. Wyniki audytu OWASP ZAP

Na rysunku 6.1 przedstawiono podatności wykryte podczas skanowania witryny WordPress wraz z liczbą adresów URL zawierających daną podatność.

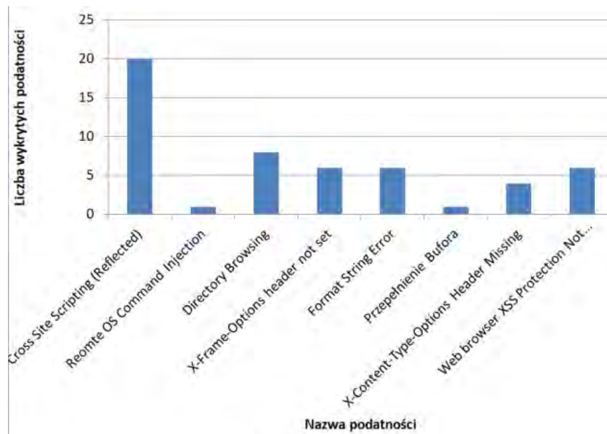


Rysunek 1: Podatności wykryte podczas skanowania witryny WordPress za pomocą skanera OWASP ZAP

Najliczniej występującą podatnością wykrytą podczas skanowania jest Cookie No HttpOnly Flag. Ciasteczka tej strony mogą zostać przejęte i przeniesione na inną stronę. Stwarza to możliwość przejęcia sesji użytkownika.

Najgroźniejszą podatnością wykrytą podczas skanowania jest Directory Browsing. Atakujący może dowolnie przeglądać listę katalogów na serwerze, co umożliwia poznanie struktury aplikacji. Można w taki sposób uzyskać dostęp do ukrytych skryptów, kopii zapasowych, które pomogą zaplanować dalsze etapy ataku.

Na rys. 2 przedstawiono wyniki skanowania witryny Joomla za pomocą skanera OWASP ZAP.

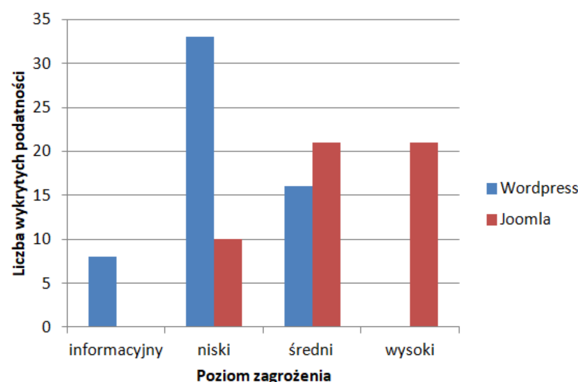


Rysunek 2: Podatności wykryte podczas skanowania witryny Joomla za pomocą skanera OWASP ZAP

W tym przypadku najgroźniejszą oraz najczęściej występującą podatnością jest Cross Site Scripting (Reflected).

Kolejnym słabym punktem witryny jest wrażliwość na atak Remote OS Command Injection. Atak ten umożliwia nieautoryzowane wykonanie poleceń systemowych poprzez brak filtrowania danych wejściowych używanych do tworzenia poleceń systemowych. Wstrzykiwanie poleceń systemowych jest również możliwe w przypadku niepoprawnego wywoływania programów zewnętrznych. Podatność została odnaleziona w podstronie *about*.

Na rys. 3 przedstawiono liczbę wykrytych podatności według poziomu zagrożenia.



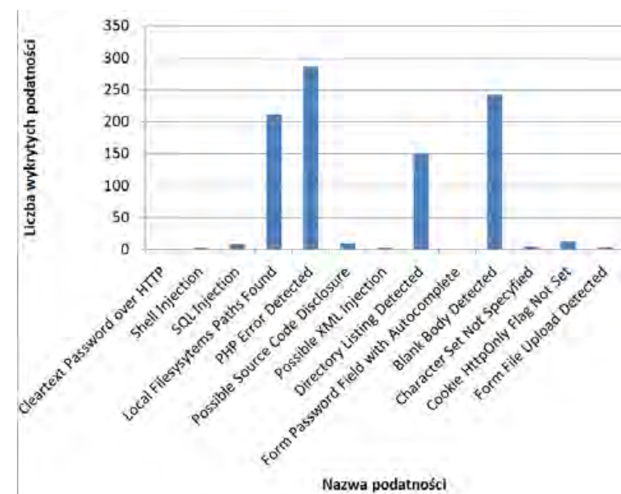
Rysunek 3: Liczba podatności według poziomu zagrożenia wykryta podczas skanowania programem OWASP ZAP

Mimo że witryna WordPress miała wykrytych więcej podatności, to jednak większość z nich stanowią te o niskim stopniu zagrożenia. Podczas skanowania nie została wykryta żadna podatność w wysokim stopniu wpływająca na bezpieczeństwo witryny.

W przypadku witryny Joomla sytuacja jest odwrotna. Znaczna większość wykrytych podatności ma przypisany poziom średni lub wysoki. Podatności o niskim stopniu zagrożenia zostało wykrytych niewiele, a tych o najniższym poziomie nie znaleziono wcale. Oznacza to, że mimo tego, że witryna WordPress miała więcej luk bezpieczeństwa, to jednak witryna Joomla jest relatywnie gorzej zabezpieczona.

6.2. Wyniki audytu Vega

Poniżej na rysunku 4 przedstawiono podatności wykryte w witrynie WordPress.



Rysunek 4: Podatności wykryte podczas skanowania witryny WordPress za pomocą skanera Vega

Najczęściej występującą podatnością wykrytą podczas skanowania jest PHP Error Detected. Gdy podczas otwierania strony zostanie wykryty błąd, zamiast docelowej strony zostaje wyświetlona wygenerowana automatycznie strona, która zawiera szczegółowe informacje dotyczące występującego błędu. Atakujący poprzez odpowiednią preparację żądań i analizę wywołanych błędów PHP może poznać schemat działania aplikacji.

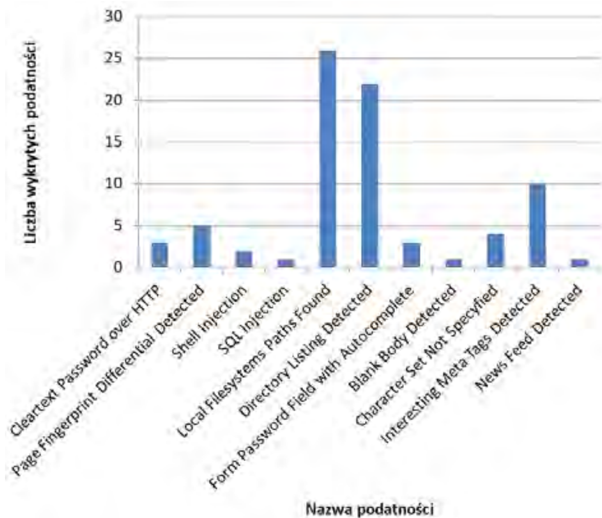
Najgroźniejszymi wykrytymi podatnościami są SQL Injection oraz Shell Injection.

Po przeskanowaniu witryny WordPress przeprowadzono audyt dla witryny Joomla. Otrzymane wyniki zostały przedstawione na rysunku 5.

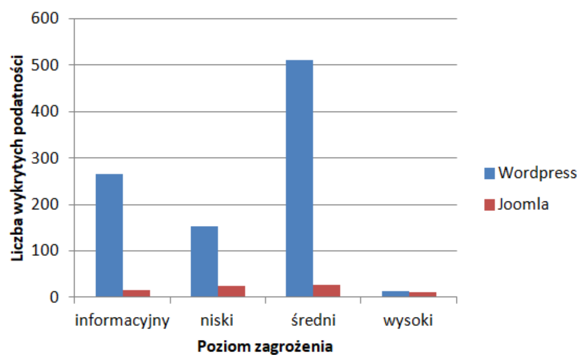
Najliczniej występujące to: Local Filesystems Paths Found oraz Directory Listing Detected.

Pierwsza z wymienionych podatności polega na wyświetlaniu ścieżki bezwzględnej pliku, najczęściej w odpowiedzi na błąd. Jest to dosyć poważna podatność, umożliwiająca atakującemu pozyskanie informacji o środowisku serwerowym. Znajomość układu plików na serwerze może zwiększyć szansę powodzenia ataku typu „blind attack”.

W kolejnym etapie badań liczbę wykrytych podatności podzielono według stopnia niebezpieczeństwa nadanego przez skaner. Wyniki przedstawiono na rysunku 6.



Rysunek 5: Podatności wykryte podczas skanowania witryny Joomla za pomocą skanera Vega



Rysunek 6: Liczba podatności według poziomu zagrożenia wykryta podczas skanowania programem Vega

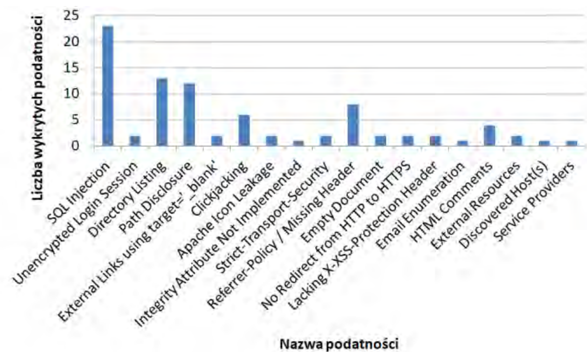
Widoczna jest duża różnica pomiędzy wynikami dla WordPress i Joomla. Podczas skanowania witryny WordPress znaleziono kilka typów podatności, które zawierają znaczącą większość wszystkich wykrytych podatności. Niemal wszystkie z tych podatności odnoszą się do plików zasobów. Spośród tych 4 typów podatności, 3 zostały również odkryte podczas skanowania witryny Joomla, jednak tam liczba wykrytych podatności dla każdego typu wynosiła ok. 20. Prawdą jest, że WordPress zawiera więcej zasobów niż Joomla, jednak główną przyczyną tak dużej rozbieżności w liczbie wykrytych podatności jest gorsze zabezpieczenie zasobów witryny WordPress. W każdej kategorii więcej podatności zostało odnalezionych w witrynie WordPress.

6.3. Wyniki audytu Detectify

Na rysunku 7 przedstawiono podatności wykryte podczas skanowania witryny WordPress wraz z liczbą adresów URL zawierających daną podatność.

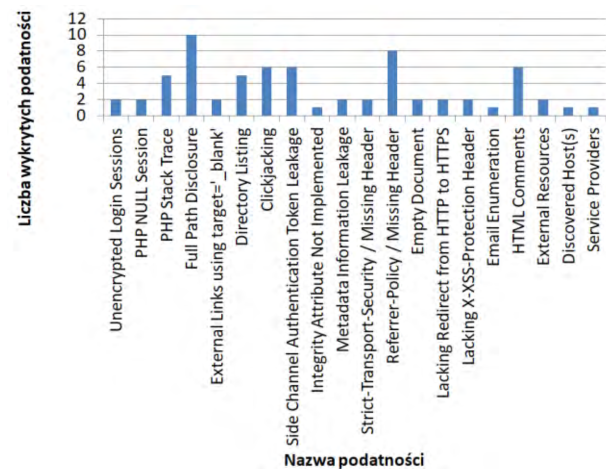
Najczęściej występującą i najgroźniejszą podatnością wykrytą podczas skanowania jest SQL Injection.

Bardzo niepokojącą podatnością znalezioną podczas przeprowadzania badań jest Unencrypted Login Session. Oznacza to, że jeżeli atakującemu uda się przechwycić ruch internetowy, może on odczytać dane logowania, ponieważ są przesyłane w postaci tekstu jawnego. Ta podatność jest tym groźniejsza, że została znaleziona w formularzu logowania do panelu administracyjnego. W witrynie WordPress został zaimplementowany protokół HTTPS jednak jest on nieużywany, jak wskazuje podatność Lacking redirect from HTTP to HTTPS



Rysunek 7: Podatności wykryte podczas skanowania witryny WordPress za pomocą skanera Detectify

Następnie badaniu została poddana witryna Joomla. Wyniki zostały przedstawione na rysunku 8.



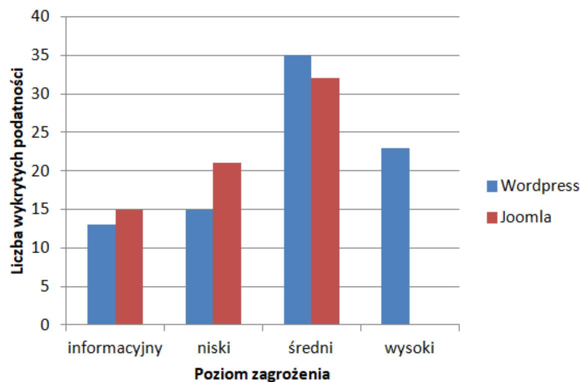
Rysunek 8: Podatności wykryte podczas skanowania witryny Joomla za pomocą skanera Detectify

Najczęściej wykrywaną podatnością podczas skanowania okazało się Full Path Disclosure. Oznacza to, że istnieje możliwość podejrzenia pełnej ścieżki niektórych plików na serwerze. Może to pomóc atakującemu zorientować się w strukturze katalogów i plików, może stanowić również bazę do innych ataków, takich jak Local File Inclusion, zwłaszcza w połączeniu z również wykrytą podatnością PHP Null Session.

Kolejną, często występującą podatnością jest Side Channel Authentication Token Leakage. W niektórych starszych przeglądarkach możliwe jest zbieranie informacji o bieżących działaniach kryptograficznych poprzez wykorzystanie elementów strony internetowej jako bocznych kanałów. Następnie zdobyte informacje można wykorzystać podczas inżynierii wstecznej do odwrócenia tokenów bezpieczeństwa. Tym sposo-

bem możliwe jest wyodrębnienie danych użytkownika takich jak login lub adres e-mail.

Kolejnym krokiem było przedstawienie znalezionych podatności dla obu witryn z poziomem zagrożenia przypisanym przez skaner, tak jak to pokazano na rys. 9.

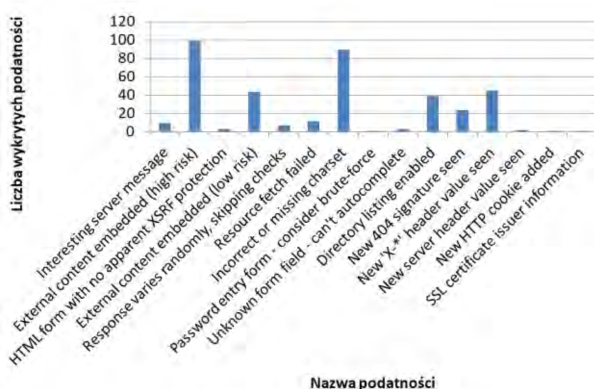


Rysunek 9: Liczba podatności według poziomu zagrożenia wykryta podczas skanowania programem Detectify

Porównując otrzymane wyniki można zauważyć, że liczba wykrytych podatności poziomu informacyjnego, niskiego i średniego dla obu witryn jest zbliżona. Widać przewagę WordPress w zagrożeniach poziomu informacyjnego i niskiego objawiającą się mniejszą liczbą wykrytych podatności, natomiast biorąc pod uwagę zagrożenia o średnim poziomie zagrożenia przewagę ma Joomla. Jednakże w witrynie Joomla nie wykryto żadnych podatności o wysokim stopniu zagrożenia, w przeciwieństwie do WordPress. W WordPress wykryto 23 podatności o wysokim stopniu niebezpieczeństwa dla witryny, są to podatności typu SQL Injection. Z powyższego porównania można wyciągnąć wniosek, że w badaniu skanerem Detectify witryna WordPress okazała się znacznie gorzej zabezpieczona niż witryna Joomla.

6.4. Wyniki audytu Skipfish

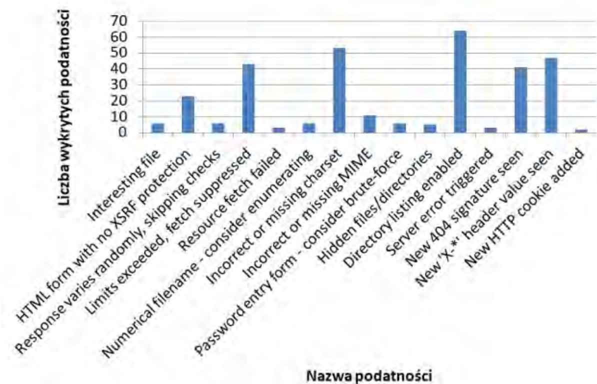
Na rys. 10 przedstawiono podatności wykryte w witrynie WordPress, pogrupowane według kategorii podatności.



Rysunek 10: Podatności wykryte podczas skanowania witryny WordPress za pomocą skanera Skipfish

Spośród wszystkich wykrytych podatności najczęściej występowała podatność External content embedded on a page (high risk), wykryto również inny wariant tej podatności (low risk). Obie podatności oznaczają, że strona pobiera dane z zewnętrznego źródła bez odpowiedniego filtrowania i walidacji. Brak poprawnego sprawdzania danych wejściowych może otworzyć drogę do przeprowadzenia najbardziej niebezpiecznych ataków takich jak Cross-Site Scripting i SQL Injection.

Następnie badaniu została poddana witryna Joomla. Rezultaty przedstawiono na rysunku 11.

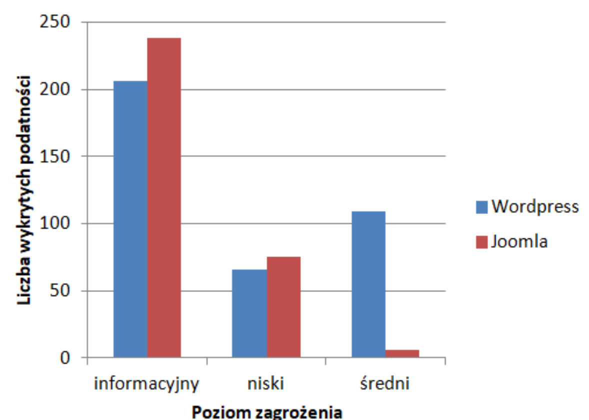


Rysunek 11: Podatności wykryte podczas skanowania witryny Joomla za pomocą skanera Skipfish

W przypadku tej witryny najczęściej wystepujacą podatnością było Directory listing enabled.

Podatność, która występowała niemal równie często podczas skanowania to Incorrect or missing charset. Brak zestawu znaków powoduje, że przeglądarki próbują same odnaleźć pasujący zestaw znaków. W tym momencie atakujący dostaje możliwość stworzenia i wysłania pewnej treści na stronę www. W ostateczności brak zestawu znaków może umożliwić atak typu Cross-Site Scripting. Podczas skanowania znaleziono również formularze, które nie miały zaimplementowanej ochrony przeciwko atakowi typu XSRF (Cross-Site Request Forgery).

Na rysunku 12 przedstawiono liczbę znalezionych podatności podzielonych według poziomu zagrożenia.



Rysunek 12: Liczba podatności według poziomu zagrożenia wykryta podczas skanowania programem Skipfish

Podczas skanowania narzędziem Skipfish w badanych witrynach nie wykryto podatności o wysokim

stopniu zagrożenia. W zagrożeniach poziomu informacyjnego i niskiego widać, że wyniki są podobne, w obu przypadkach więcej podatności wykryto w witrynie Joomla. Natomiast dla zagrożeń poziomu średniego, czyli najgroźniejszych wykrytych podatności, witryna WordPress prezentuje się o wiele gorzej od Joomla. W pierwszej kolejności w WordPress wykryto aż 109 podatności poziomu średniego, podczas gdy w Joomla liczba ta wynosi 6. Poza tym w witrynie Joomla wykryto tylko jedną kategorię zagrożeń poziomu średniego, to jest kategorię Interesting file. Natomiast w WordPress wykryto dwie takie kategorie (Interesting server messager oraz External content embedded). Biorąc to wszystko pod uwagę, w tym porównaniu WordPress wygląda na witrynę gorzej zabezpieczoną.

6.5. Wyniki zbiorcze

Tabela 1: Suma podatności znalezionych przez skanery podczas badań witryn WordPress i Joomla

	WordPress	Joomla
Liczba wykrytych podatności	1444	508

Po przeanalizowaniu danych z tab. 1 widać znaczną przewagę liczby podatności znalezionych w WordPress, w tej witrynie znaleziono ok. 3 razy więcej podatności niż w Joomla. W znacznym stopniu odpowiedzialne za ten wynik są cztery kategorie podatności, gdzie w ramach każdej znaleziono ponad 200 podatności. W sumie te cztery kategorie zawierają połowę wszystkich wykrytych podatności w witrynie WordPress. Ten CMS udostępnia więcej funkcjonalności niż Joomla, co przekłada się na bardziej rozbudowane drzewo katalogów i więcej plików źródłowych, z których każdy musi być zabezpieczony. Drugim powodem wykrycia tak dużej liczby podatności jest niewątpliwie słabe zabezpieczenie tych plików w witrynie WordPress. Po dokładniejszym przeanalizowaniu wyników okazało się, że w wybranych katalogach obu witryn o podobnej liczbie zasobów, w których wykryto daną podatność, w witrynie WordPress większość zasobów danego katalogu zawierało podatność podczas gdy w witrynie Joomla było tylko kilka wykrytych przypadków.

Tabela 2: Podatności odkryte podczas badań pogrupowane według poziomu zagrożenia

	Witryna	
Poziom zagrożenia	WordPress	Joomla
wysoki	38	34
średni	837	160
niski	97	109
informacyjny	472	205

Po zsumowaniu podatności według poziomów zagrożenia (tab. 2) okazało się, że w każdym poziomie, oprócz niskiego, Joomla okazuje się bardziej bezpiecznym systemem CMS. WordPress przegrywa w najważniejszych kategoriach, czyli w poziomie wysokim i średnim. Różnica wyników dla poziomu wysokiego nie jest znaczna, nie rozstrzyga o tym, która witryna jest lepiej zabezpieczona, jednak jeśli wziąć pod uwagę rozkład podatności w poszczególnych kategoriach, okazuje się, że w WordPress wykryto bardzo dużo podatności typu SQL injection. Ta podatność została uznana w raporcie OWASP ZAP Top 10 [12] za najgroźniejszą podatność wykrywaną w aplikacjach internetowych. W podatnościach poziomu średniego widać znaczącą przewagę witryny Joomla. Na tym poziomie zagrożenia przeważają podatności odnoszące się do plików źródłowych, a w tej kategorii, jak już wspomniano wcześniej, WordPress pod względem bezpieczeństwa przegrywa z Joomla.

7. Wnioski

Podczas analizy okazało się, że wyniki szczegółowe nie wskazują jednoznacznie na to, która witryna jest lepiej zabezpieczona.

Po zagłębieniu się w wyniki szczegółowe widoczne jest, że w większości porównań witryna Joomla wypada lepiej, jednak były też takie dane, które wskazywały na WordPress jako witrynę lepiej zabezpieczoną.

Jednak po zsumowaniu wyników i rozpatrywaniu ich jako całość można wyciągnąć tylko jeden wniosek: to Joomla jest lepiej zabezpieczoną witryną. Joomla wygrywa z WordPress w każdej z kategorii: podatności w poszczególnych kategoriach oraz wykryte podatności według poziomu zagrożenia. Jedną z przyczyn takiego stanu rzeczy jest niewątpliwie to, że WordPress jest bardziej rozbudowaną platformą, wprowadza więcej funkcjonalności niż Joomla, co przekłada się na więcej zasobów, które należy zabezpieczyć. Jednak często wykrywano kilka podatności, które odnosiły się do tego samego miejsca, co wskazuje na niższy poziom bezpieczeństwa witryny WordPress.

Podsumowując, przeprowadzone badania wskazują na Joomla jako lepiej zabezpieczoną platformę CMS. Jednak obie witryny mają poważne luki w zabezpieczeniach. Z tego powodu żadna z badanych witryn w podstawowej konfiguracji nie powinna przechowywać wrażliwych danych.

Literatura

[1] Acunetix Web Application Vulnerability Report 2019, https://cdn2.hubspot.net/hubfs/4595665/Acunetix_web_application_vulnerability_report_2019.pdf, [04.05.2020].

[2] M. Meike, J. Sametinger, A. Wiesauer, Security in Open Source Web Content Management Systems, IEEE Security and Privacy Magazine, 2009.

[3] S.K. Patel, V.R Rathod, S. Parikh, Comparative Analysis Of Web Security In Open Source Content Management System, ISSP, 2013.

[4] S.K. Patel, V.R Rathod, S. Parikh, Joomla. Drupal and WordPress - A Statistical Comparison of Open Source CMS, IEEE, 2011.

[5] A. Sagala, E. Manurung, Testing and Comparing Result Scanning Using Web Vulnerability Scanner, American Scientific Publishers, 2015.

[6] Usage statistics of content management systems, https://w3techs.com/technologies/overview/content_management, [07.04.2020].

[7] C. Pepper., M. Tietz, D. Weeks, Open Source Development and Application Security Survey Analysis, Securosis, 2014.

[8] WordPress, https://WordPress.org, [07.04.2020].

[9] Joomla!, https://www.joomla.org/, [07.04.2020].

[10] OWASP ZAP, https://www.zaproxy.org/, [21.05.2020].

[11] Vega, https://subgraph.com/vega/, [21.05.2020].

[12] Detectify, https://detectify.com/, [21.05.2020].

[13] SkipfishDoc, https://code.google.com/archive/p/skipfish/wikis/SkipfishDoc.wiki, [21.05.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE
JCSI 16 (2020) 269–273
Received: 23 June 2020
Accepted: 6 July 2020

Analysis of the Blazor framework in client-hosted mode

Analiza działania szkieletu Blazor w trybie klienta z hostingiem

Karol Kozak*, Jakub Smołka

Abstract

The purpose of the article is to analyze the Blazor framework in client mode with the hosting option, used to create SPA applications. A test application has been created for the purposes of testing. The application loading efficiency and the size of downloaded data were examined for the completed application. The performance in calculation tests, operations on collections and the efficiency of generating DOM elements were determined. JavaScript code performance has been compared. Blazor offers good performance in calculation scenarios and operations on collections. JavaScript is more efficient in generating DOM elements and performing recursive functions. Blazor is a good example of using the potential of the WebAssembly standard in creating Internet applications.

Keywords: Blazor; C#; JavaScript; WebAssembly

Streszczenie

Celem artykułu jest analiza działania szkieletu Blazor w trybie klienta z opcją hostingu, służącego do tworzenia aplikacji SPA. Na potrzeby wykonania badań stworzona została aplikacja testowa. Dla wykonanej aplikacji zbadano wydajność ładowania aplikacji oraz rozmiar pobranych danych. Określono także wydajność w testach obliczeniowych, operacjach na kolekcjach oraz zbadano wydajność generowania elementów DOM. Porównana została wydajność kodu JavaScript. Blazor oferuje dobrą wydajność w scenariuszach obliczeniowych i operacjach na kolekcjach. JavaScript jest wydajniejszy w generowaniu elementów DOM i wykonywaniu funkcji rekurencyjnych. Blazor jest dobrym przykładem wykorzystania potencjału standardu WebAssembly w tworzeniu aplikacji internetowych.

Słowa kluczowe: Blazor; C#; JavaScript; WebAssembly

*Corresponding author

Email address: karol.kozak1@pollub.edu.pl (K. Kozak)

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

1. Wstęp

Sposób wykorzystywania Internetu ewoluował i ciągle ewoluuje w czasie. Rozwój Internetu mimo, że dynamiczny, nie znaczy to, że nastąpił z dnia na dzień. W przeszłości, strony internetowe miały głównie zawartość statyczną. Było to podyktowane ograniczoną przepustowością łącz użytkowników oraz narzędziami i technologiami jakimi dysponowali programiści. Wraz z rozwojem dostępu do Internetu oraz wzrostem prędkości Internetu zaczęto go wykorzystywać w szerszej liczbie aspektów. Obecnie strony internetowe często pełnią rolę aplikacji internetowych. Takie podejście pozwala na stworzenie aplikacji, z której skorzysta każdy użytkownik na każdym urządzeniu, które pozwala na przeglądanie Internetu. Niwelowana jest w ten sposób konieczność pisania aplikacji na wiele różnych platform. Wśród dostępnych konwencji tworzenia aplikacji internetowych, jedną z konwencji stale zyskującą na popularności jest SPA (Single Page Application). Korzystając z takiej aplikacji, użytkownik ma wrażenie korzystania z aplikacji natywnej, ponieważ każda wykonana akcja wiąże się z płynną aktualizacją zawartości strony bez potrzeby jej powtórnego załadowania.

Firma Microsoft już wcześniej próbowała wprowadzić C# do strony klienta aplikacji internetowej, czyli bezpośrednio do przeglądarki. Prekursorem był ASP.NET WebForms, który pozwalał budować dynamiczne aplikacje internetowe z wykorzystaniem dostępnej biblioteki gotowych kontrolek. Podejście to znane jest twórcom aplikacji desktopowych. Technologia choć ciągle używana, nie zdobyła popularności z uwagi na ociężałość stron na niej opartych oraz dużą ilość przesyłanych danych pomiędzy klientem i serwerem. Inną próbą było stworzenie platformy Silverlight. Umożliwiała ona stworzenie strony internetowej w oparciu o WPF (język XAML) oraz C#. Do obsługi aplikacji wykorzystujących tą technologię wymagana była wtyczka zainstalowana w przeglądarce internetowej. Została ona jednak wyparta przez standard HTML5. Wsparcie dla tej technologii zostanie zakończone w październiku 2021r. Silverlight wspierał procesory oparte na architekturze x86 i nie doczekał się wsparcia dla urządzeń mobilny tj. Android czy iOS. W 2009 powstał ASP.NET MVC, który jest wykorzystywany do dziś. Został on doceniony przez społeczność programistyczną za jego stabilność i przemyślanie rozwiązania pozwalające na pisanie dobrej jakości aplikacji internetowych. W 2016 roku powstał ASP.NET Core jako darmowy i otwarto źródłowy następca platformy ASP.NET. Cechuje się on przede wszystkim wsparciem dla wielu platform, wysoką wydajnością i wsparciem dla wielu nowoczesnych rozwiązań tj. wirtualizacja Docker [1][2][3].

Mimo ciągłego rozwoju platformy ASP.NET, programiści w dalszym ciągu zmuszeni byli do tworzenia front-endu na bazie innych szkieletów wykorzystujących język JavaScript tj. Angular. Wraz z ogłoszeniem .NET Core 3.0, przedstawiony został szkielet Blazor, który umożliwia tworzenie stron SPA z wykorzystaniem języka C# i standardu WebAssembly.

W poniższej pracy przedstawiona zostanie analiza działania szkieletu Blazor w trybie klienta z hostingiem. Tryb ten zostanie zbadany pod kątem oferowanej wydajności w zastosowaniach typowych dla aplikacji internetowych.

2. Przegląd literatury

W literaturze można znaleźć wiele informacji na temat tworzenia aplikacji internetowych w szkielecie Blazor, jednakże tryb klienta z hostingiem nie został zbadany pod kątem oferowanej wydajności. Istnieje kilka publikacji dotyczących porównania Blazora i innych konkurencyjnych szkieletów oraz publikacji dotyczących WebAssembly wykorzystywanego w trybie klienta szkieletu Blazor.

W artykule [4] Autorzy starali się sprawdzić czy WebAssembly może stać się alternatywą dla JavaScript przy tworzeniu aplikacji internetowych. W tym celu opracowano dwie bliźniacze aplikacje typu SPA z wykorzystaniem szkieletu Blazor w trybie klienta i szkieletu Angular oraz implementacje niektórych scenariuszy testowych w języku C++ skompilowanym do kodu WebAssembly. Została zbadana wydajność dla następujących kryteriów: ładowanie aplikacji, operacje na kolekcjach obiektów, generowanie elementów DOM, obsługa żądania http oraz porównano metryki kodu. Szkielet Blazor osiągał gorsze wyniki od szkieletu Angular w prawie wszystkich testowanych kryteriach. Wskazano na wysoką wydajność w scenariuszach obliczeniowych implementacji w języku C++ kompilowanym do WebAssembly. Kod ten był wyraźnie szybszy od JavaScript i dwóch testowanych szkieletów. Autorzy ocenili szkielet Blazor, jako dobre narzędzie do tworzenia aplikacji SPA z wykorzystaniem WebAssembly zamiast JavaScript, choć istniała konieczność użycia języka JavaScript do implementacji pewnych funkcjonalności.

W pracy [5] autorzy porównali szkielet Blazor w trybie klienta do innych popularnych szkieletów tj. Angular, React i Vue.js. Porównanie to dotyczyło m.in. krzywej uczenia, wymagań dotyczących pamięci RAM oraz rozmiaru strony. Autorzy ocenili szkielet Blazor jako średnio trudny do nauki z uwagi na konieczność nauki jego architektury. Jednocześnie wg. autorów jest on łatwiejszy do nauki od Angulara i trudniejszy niż Vue.js. W kwestii wykorzystania pamięci RAM, Blazor osiągnął najgorszy rezultat, wykazując około pięciokrotnie większe zapotrzebowanie pamięci względem React i Vue.js. W kategorii rozmiaru strony, Blazor również zajął ostatnią pozycję. W podsumowaniu, zaznaczono, że testy wykonane zostały z wykorzystaniem wersji zapoznawczej szkieletu Blazor w związku z czym w przyszłości może on osiągać lepsze wyniki i stać się popularnym szkieletem do tworzenia aplikacji internetowych.

Praca [6] skupiona jest na przedstawieniu możliwości standardu WebAssembly i szkieletu Blazor. Autor przedstawia kluczowe rozwiązania wdrożone i obsługiwane przez Blazor poprzez implementacje aplikacji internetowej. Wykorzystany został tryb klienta z opcją hostowania w wersji 3.0.0 preview4. Widoczny był duży rozmiar strony przy pierwszym jej uruchomieniu sięgający 12.4 MB. Zwrócono jednak uwagę, na dobre wykorzystanie pamięci cache przeglądarki, ponieważ następne uruchomienia strony powodowały pobranie tylko 32,5 KB danych.

Artykuł [7] jest analizą wydajności kodu WebAssembly. Przeprowadzone zostały testy porównawcze na różnych systemach operacyjnych oraz różnych przeglądarkach. Porównana została wydajność natywnego kodu C z kodem skompilowanym do WebAssembly dla różnych przeglądarek. Wyniki wykazały, że w 5 z 12 testów, kod WebAssembly był szybszy od kodu natywnego. Jednakże podsumowując wszystkie wykonane testy, kod WebAssembly okazał się mniej wydajny od kodu natywnego C. Następne porównanie wydajności dotyczyło WebAssembly i JavaScript. Testy wykonane zostały z wykorzystaniem różnych platform i różnych przeglądarek internetowych. Dla każdej badanej konfiguracji kod WebAssembly był co najmniej 1.5x szybszy od kodu napisanego w języku JavaScript.

Przedstawiony przegląd literatury ukazuje brak bezpośredniego porównania wydajności kodu Blazor oraz kodu JavaScript, wywołanego z poziomu szkieletu Blazor. Istnieje więc potrzeba sprawdzenia, czy w niektórych przypadkach użycie kodu JavaScript pozwoli na osiągniecie lepszej wydajności.

3. Metoda badawcza

Na potrzeby wykonania pomiarów wydajności, stworzona została aplikacja testowa na platformie ASP.NET Core 3.1. Wykorzystany został szablon Blazor WebAssembly 3.2.0 Release Candidate.

Obszar badań dotyczy wydajności ładowania aplikacji, wydajności w scenariuszach obliczeniowych, wydajności wykonywania operacji na kolekcjach LINQ oraz wydajności aktualizowania struktury DOM. Dla części badanych scenariuszy porównana została wydajność z kodem JavaScript. Pomiary zostały wykonane w oparciu o narzędzia deweloperskie Google Chrome oraz z wykorzystaniem klasy Stopwatch platformy .NET.

Specyfikacja stanowiska badawczego określona została w Tabeli 1.

Tabela 1: Specyfikacja stanowiska badawczego

Procesor	Pamięć RAM	Dysk twardy	System operacyjny
Intel Core i5-6300HQ	8GB DDR4 2133MHz	Samsung SSD 850 EVO M.2 250GB (SATA III)	Windows 10 Education 1909 64bit

3.1. Wydajność ładowania aplikacji

Celem eksperymentu było określenie wydajności ładowania aplikacji dla trybu klienta z hostingiem szkieletu Blazor. W tym celu stworzony został komponent wyświetlający 10 kart wraz z obrazkami.

Użytymi parametrami dla pomiarów dotyczących czasu ładowania aplikacji są:

- Parametr „Load" – określający czas w którym zakończono ładowanie HTML oraz wszystkich zapytań blokujących tj. pobranie CSS, plików JavaScript itp.
- Parametr „Finish" – określający czas pełnego załadowania strony. W tym parametrze zawarty jest czas ładowania pozostałych elementów strony, których ładowanie kontynuowane było dalej, po wystąpieniu zdarzenia load, na przykład asynchroniczne ładowanie elementów. [8]

Do pomiaru rozmiaru pobranej strony użyto parametru „transferred", oznaczającego ilość danych pobranych z serwera przez przeglądarkę.

3.2. Wydajność obliczania liczby PI

Eksperyment polegał na określeniu wydajności obliczania liczby PI. Implementacja funkcji w języku C# opiera się na podstawowych działaniach matematycznych na liczbach całkowitych bez znaku typu uint wraz z wykorzystaniem tablic. Algorytm w języku JavaScript powstał na podstawie algorytmu napisanego w języku C# z kilkoma niezbędnymi zmianami. Dotyczyły one obsługi typów liczbowych bez znaku, co było ważne z punktu widzenia poprawności obliczeń. W przypadku zmiennych tablicowych, wykorzystano tablice typu Uint32Array, przechowujące 32-bitowe liczby całkowite bez znaku. W przypadku zmiennych, w których należało dokonywać obliczeń w przestrzeni liczb całkowitych bez znaku, zastosowano operator >>>, dokonujący przesunięcie bitowe w prawo bez znaku.
Badanie zostało przeprowadzone dla następujących wielkości liczby PI:

- 1000 miejsc po przecinku,
- 5000 miejsc po przecinku,
- 10000 miejsc po przecinku.

3.3. Wydajność obliczania funkcji Ackermanna

W tym eksperymencie zmierzono wydajność szkieletu Blazor w trybie klienta z hostingiem i języka JavaScript w obliczaniu funkcji Ackermanna. Funkcja ta jest funkcją rekurencyjną, cechującą się szybkim wzrostem wartości. Wykorzystywana jest często do badania jakości optymalizacji kompilatorów pod kątem wydajności rekurencji. Aby porównanie było odpowiednie, implementacja funkcji w JavaScript jest identyczna do implementacji funkcji w języku C#.
Badanie zostało przeprowadzone dla następujących parametrów funkcji:

- A(3, 7), gdzie m = 3, n = 7,
- A(3, 8), gdzie m = 3, n = 8,
- A(3, 9), gdzie m = 3, n = 9.

3.4. Wydajność operacji na kolekcjach LINQ

Kolejnym eksperymentem było określenie wydajności operacji na kolekcjach LINQ. W tym celu, dla kolekcji o określonej liczbie elementów wyszukiwano maksymalną i minimalną liczbę parzystą. Kolekcje tworzone były poprzez zapełnianie ich losowymi liczbami całkowitymi. Sam proces tworzenia kolekcji nie był brany pod uwagę w pomiarach. Pomiar dotyczył tylko czasu potrzebnego do określenia minimalnej i maksymalnej liczby parzystej w kolekcji.
Badanie zostało przeprowadzone dla następujących liczebności kolekcji:

- 100.000 elementów zbioru,
- 500.000 elementów zbioru,
- 1.000.000 elementów zbioru.

3.5. Wydajność aktualizowania elementów DOM

Kolejnym eksperymentem było określenie wydajności aktualizowania drzewa DOM dla szkieletu Blazor w trybie klienta z hostingiem oraz języka JavaScript. W tym celu, dla aplikacji Blazor, opracowany został komponent Razor, który generuje tabelę z trzema kolumnami i liczbą wierszy określoną jako parametr komponentu. Napisany został również kod w języku JavaScript implementujący tą samą funkcjonalność. W przeciwieństwie do Blazora, JavaScript posiada bezpośredni dostęp do struktury DOM, przez co można sądzić, iż będzie on szybszy. Do wykonania pomiaru wykorzystana została funkcja nagrywania w zakładce Performance w narzędziach deweloperskich przeglądarki Google Chrome. Czas mierzono z punktu widzenia użytkownika – to znaczy od momentu kliknięcia przycisku uruchamiającego akcję generowania tabeli, do momentu wyświetlenia tabeli na stronie.
Badanie zostało przeprowadzone dla następującej liczby wierszy tabeli:

- 100 wierszy,
- 1000 wierszy,
- 5000 wierszy.

4. Wyniki

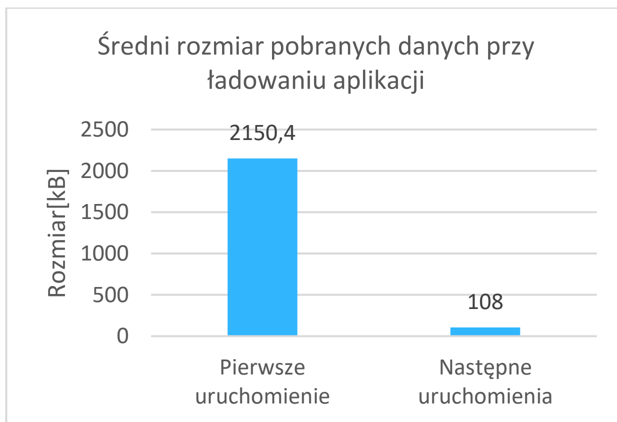
Wyniki pomiarów czasów uruchomienia aplikacji zostały przedstawione w tabelach 2 i 3. Można zauważyć, że czas pierwszego, pełnego załadowania się strony (parametr Finish) jest ok. 10 razy dłuższy niż czas wstępnego załadowania strony (parametr Load). Dzięki zastosowaniu pamięci podręcznej przeglądarki, następnie uruchomienia aplikacji są odpowiednio o 12% (Load) i 32% (Finish) krótsze. Analizując wartości odchyleń standardowych, można stwierdzić, że strona testowa nie ma dużej zmienności czasów ładowania.

Tabela 2: Wyniki czasów pierwszego uruchomienia aplikacji

Pomiar	Czas pierwszego uruchomienia aplikacji	
	Load [ms]	Finish [ms]
1	105	942
2	97	1000
3	85	1010
4	89	972
5	94	1040
6	88	980
7	87	931
8	88	970
9	90	1030
10	98	979
Średnia	**92**	**985**
Odchylenie standardowe	**±6,26**	**±35,15**

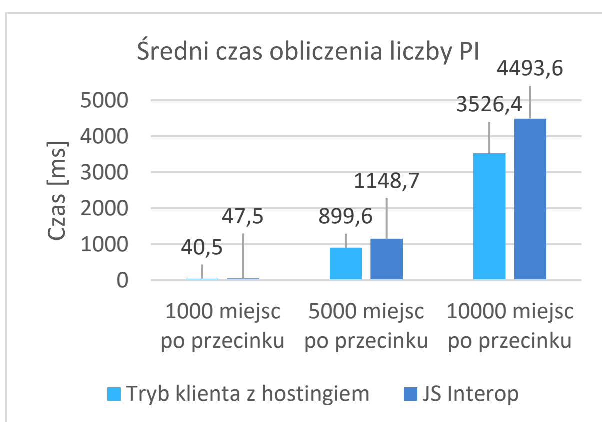
Tabela 3: Wyniki czasów następnych uruchomień aplikacji

Pomiar	Czas następnych uruchomień aplikacji	
	Load [ms]	Finish [ms]
1	94	687
2	79	677
3	81	692
4	80	661
5	74	608
6	76	600
7	92	688
8	80	718
9	79	678
10	79	665
Średnia	**81**	**667**
Odchylenie standardowe	**±6,47**	**±36,96**



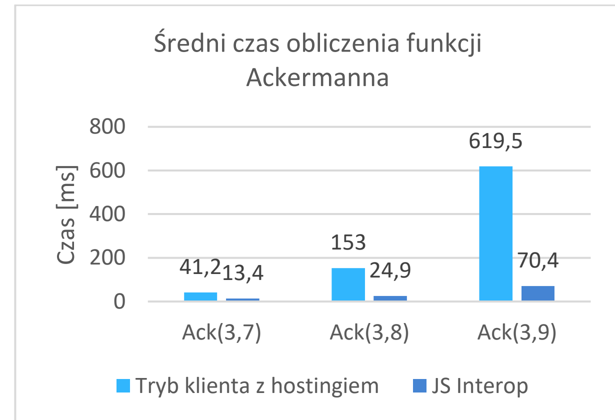
Rysunek 1: Średni rozmiar pobranych danych przy ładowaniu aplikacji

Rozmiar pobranych danych przy pierwszym uruchomieniu aplikacji jest znacznie większy niż w przypadku następnych uruchomień, co zostało przedstawione na rysunku 1. Było to oczekiwane, ponieważ aplikacja przy pierwszym uruchomieniu pobiera całe środowisko uruchomieniowe w formacie WebAssembly wraz z zależnościami w formie plików .dll. Różnica rozmiaru pobranych danych pomiędzy pierwszym a następnymi uruchomieniami sięga 95%. Świadczy to o dobrym wykorzystaniu pamięci podręcznej przeglądarki przez szkielet Blazor.



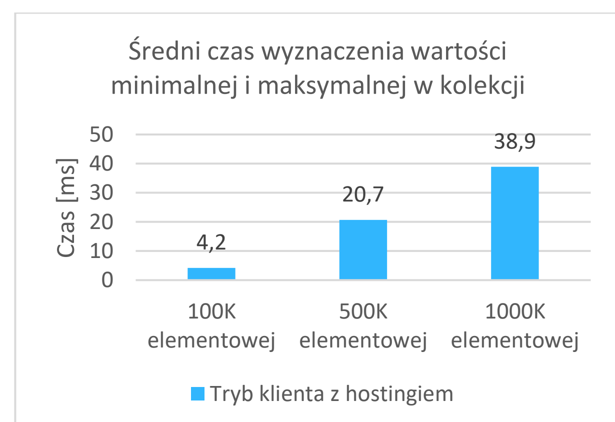
Rysunek 2: Średni czas obliczenia liczby PI

Algorytm obliczania liczby PI jest przykładem algorytmu iteracyjnego. Szkielet Blazor w trybie klienta z hostingiem wykazuje lepszą wydajność w obliczaniu liczby PI od kodu napisanego w języku JavaScript (Rysunek 2). Różnica zwiększa się wraz ze złożonością liczby PI. W przypadku 1000 miejsc po przecinku, kod C# jest 1,17 razy szybszy od kodu JavaScript. Dla 5000 i 10000 miejsc po przecinku różnica sięga 1,27 raza.



Rysunek 3: Średni czas obliczenia funkcji Ackermanna

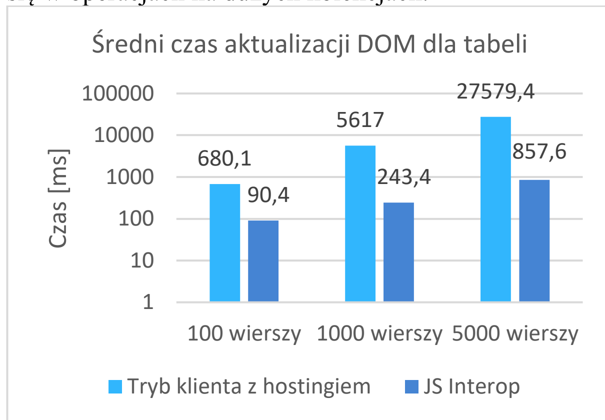
Funkcja Ackermanna jest przykładem algorytmu rekurencyjnego. Pomiary przedstawione na rysunku 3 pokazują, że kod JavaScript jest znacznie szybszy od kodu C# szkieletu Blazor. Wraz ze wzrostem złożoności funkcji, czas obliczeń w JavaScript wzrasta zdecydowanie wolniej niż w przypadku implementacji funkcji w C#. Dla argumentów m = 3, n = 7, Blazor jest 3 razy wolniejszy od JavaScript. Dla argumentów m = 3, n = 8, różnica jest 6 krotnie większa. Dla argumentów m = 3, n = 9, JavaScript jest prawie 9 razy szybszy. Można z tego wysnuć wniosek że środowisko uruchomieniowe JavaScript, jest lepiej zoptymalizowane pod kątem wywołań rekurencyjnych niż środowisko uruchomieniowe .NET.



Rysunek 4: Średni czas wyznaczenia wartości minimalnej i maksymalnej w kolekcji

Na rysunku 4, przedstawiono wyniki średnich czasów wyznaczenia wartości minimalnej i maksymalnej w kolekcji. Warto zauważyć, że wzrost czasów wyznaczenia wartości w kolekcji i wzrost liczby elementów w kolekcji nie są do siebie wprost proporcjonalne.

Można z tego wywnioskować, że LINQ dobrze skaluje się w operacjach na dużych kolekcjach.



Rysunek 5: Średni czas aktualizacji DOM w tabeli

JavaScript wykazuje lepszą wydajność w kwestii modyfikowania struktury DOM, co przedstawione zostało na rysunku 5. Można zaobserwować, że wraz ze wzrostem ilości elementów DOM do wygenerowania – w tym przypadku wierszy tabeli, czas aktualizowania struktury w JavaScript wzrasta zdecydowanie wolniej niż w przypadku implementacji funkcji w C#. Dla 100 wierszy różnica wydajności wynosi 86%. Dla 1000 wierszy różnica wzrasta do 95%. Dla 5000 wierszy różnica wynosi prawie 97%.

5. Wnioski

Przedstawiono możliwości szkieletu Blazor w trybie klienta z hostingiem. Pokazano wydajność aplikacji testowej opartej na tym szkielecie w typowych zastosowaniach aplikacji internetowej. Dla niektórych scenariuszy testowych wykonano porównanie wydajności kodu C# z kodem napisanym w języku JavaScript.

Szkielet Blazor bazujący na standarcie WebAssembly z hostingiem prezentuje dobry poziom wydajności w kwestii operacji na kolekcjach a także w scenariuszach typowo obliczeniowych. Należy zwrócić uwagę na stosunkowo duży rozmiar aplikacji przy pierwszym uruchomieniu. Słabo wypada optymalizacja szkieletu Blazor w kwestii obsługi funkcji rekurencyjnych. W takich przypadkach warto rozważyć użycie kodu JavaScript, który oferuje znacznie lepszą wydajność. Wydajność generowania struktury DOM również nie jest mocną stroną szkieletu Blazor. Spowodowane jest to ograniczeniem standardu WebAssembly w kwestii dostępu do struktury DOM.

Szkielet Blazor jest dobrym przykładem wykorzystania potencjału standardu WebAssembly w tworzeniu aplikacji internetowych. Wraz z jego dalszym rozwojem, spodziewać się można kolejnych usprawnień w kwestii wydajności, co może przełożyć się na wzrost popularności szkieletu Blazor wśród programistów.

Literatura

[1] The History of ASP.NET Part 1, https://www.dotnetcurry.com/aspnet/1492/aspnet-history-part-1 [26.05.2020].

[2] The History of ASP.NET Part 2, https://www.dotnetcurry.com/aspnet/1493/aspnet-history-part-2-mvc [26.05.2020].

[3] The History of ASP.NET Part 3, https://www.dotnetcurry.com/aspnet/1494/aspnet-history-part-3-core [26.05.2020].

[4] D. Suryś, P. Szłapa, M. Skublewska-Paszkowska: WebAssembly jako alternatywa dla JavaScript w tworzeniu nowoczesnych aplikacji internetowych, 2019.

[5] M. Lang, M. Skotnica: WebAssembly Approach to Client-side Web Development using Blazor Framework, 2019.

[6] M. Horáček: Aplikace demonstrující možnosti webového standardu WebAssembly a webového frameworku Blazor, 2019.

[7] D. Herrera, H. Chen, E. Lavoie: WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices, 2018.

[8] Evaluating Web Performance, https://cantina.co/web-performance-part-i/ [15.05.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 274–278
Received: 10 June 2020
Accepted: 3 July 2020

Implementation of solutions for distributed team management in IT sector companies

Wdrożenie rozwiązań do zarządzania zespołem rozproszonym w firmach sektora IT

Mykhailo Kuzyk*, Elżbieta Miłosz

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

Abstract

The article presents the results of a comparative analysis of selected IT tools supporting the management of a distributed IT team in companies from the IT sector. The research was carried out using two methods: scientific experiment (implementation of a test project in the tested tools) and comparative analysis (assessment of tools from the side of management and contractors of a distributed team according to the proposed criteria). The research results were aimed at choosing the best solutions that could be implemented to manage a distributed team in IT sector companies.

Keywords: IT project management; distributed teams; remote work; IT tools

Streszczenie

Artykuł przedstawia wyniki analizy porównawczej wybranych narzędzi informatycznych wspomagających zarządzanie rozproszonym zespołem informatyków w firmach z sektora IT. Badania przeprowadzono z wykorzystaniem dwóch metod: eksperymentu naukowego (wykonanie testowego projektu w badanych narzędziach) i analizy porównawczej (ocena narzędzi od strony kierownictwa i wykonawców rozproszonego zespołu wg zaproponowanych kryteriów). Wyniki badań miały na celu wybór najlepszych rozwiązań, jakie mogłyby zostać wdrożone do zarządzania zespołem rozproszonym w firmach sektora IT.

Słowa kluczowe: zarządzanie projektem informatycznym; zespoły rozproszone; praca zdalna; narzędzia informatyczne

*Corresponding author

Email address: **mykhailo.kuzyk@pollub.edu.pl (M. Kuzyk)**

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

1. Wstęp

Intensywny rozwój branży IT oraz zapotrzebowanie na systemy informatyczne w biznesie stawia nowe wyzwania wielu firmom sektora IT, tworzącym rozwiązania informatyczne. Konkurencja na rynku pracy, brak specjalistów z branży IT, rywalizacja w firmach informatycznych sprawia, że wiele z nich podejmuje decyzje o tworzeniu międzynarodowych zespołów rozproszonych, których członkowie znajdują się fizycznie w różnych lokalizacjach i w formie zdalnej pracują nad jednym projektem.

W ostatnich dziesięcioleciach zdalna praca silne umocniła swoją pozycję na rynku, światowe korporacje informatyczne zatrudniają wykwalifikowanych specjalistów głównie w formie pracy zdalnej. Takie korporacje jak Procter & Gamble, IBM, Accenture i AT & T, częściowo lub całkowicie zrezygnowały z tradycyjnego biura pracy. Dotyczy to głównie obszarów wykorzystania informatycznych technologii, wytwarzania oprogramowania oraz księgowości, co pozwala na znaczne zwiększenie efektywności pracy [1]. Wzrost ilości pracy wykonywanej zdalnie jest związany z globalizacją rynku pracy, zwiększeniem zapotrzebowania na specjalistów wąskich specjalizacji i wymagań klientów.

Zespoły rozproszone i zdalna praca stawiają nowe wyzwania w obszarze komunikacji i zarządzania. Obecnie dla zarządu korporacji i pracowników – członków zespołów rozproszonych istnieje wiele skutecznych narzędzi do zarządzania projektem dla elastycznego prowadzenia pracy zdalnej i organizacji pracy. Wdrożenie odpowiednich rozwiązań oznacza dla firm umocnienie przewagi ekonomicznej wśród innych firm na rynku [2]. W konsekwencji zatrudniani w tej formie specjaliści utrzymują swoje stanowisko przez długi czas i są bardziej zmotywowani i lojalni w stosunku do pracodawców. Zmniejszenie liczby pełnoetatowych pracowników w biurze, pozwala korporacjom na odzyskanie wysokiego współczynnika skuteczności i znacznie zaoszczędzić na wynajmie miejsc przeznaczonych dla zdalnych pracowników. International Business Machines Corp. oszczędza około 100 mln. $ rocznie ze zmianą formy pracy na zdalną. W tej firmie około 42% pracowników pracuje zdalnie [3].

Wdrożenie rozwiązań do zarządzania zespołem rozproszonym w firmach sektora IT powinno być poprzedzone analizą dostępnych narzędzi informatycznych i wyborem najlepszych z nich. Przeprowadzona analiza porównawcza może dać rekomendacje w tym obszarze dla firm wytwarzających oprogramowanie.

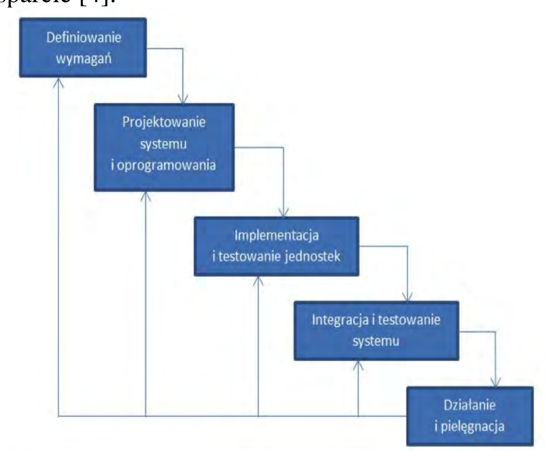
2. Podstawowe modele rozwoju oprogramowania

2.2. Model kaskadowy

Model kaskadowy – to model procesu wytwarzania oprogramowania, w którym proces rozwoju jest przed-

stawiony jako strumień (Rys.1). Dany strumień sekwencyjnie przechodzi przez fazę analizy wymagań, projektowania, wdrażania, testowania, integracji i wsparcia.

Po przeanalizowaniu wymagań generowana jest lista wymagań w stosunku do oprogramowania. Po jej sformułowaniu, opracowywany jest projekt, podczas którego tworzona jest dokumentacja z opisem wymagań dla programistów i planem realizacji. Następnie projekt jest implementowany w wybranym środowisku programistycznym, kolejne jednostki programistyczne poddawane są testowaniu. Na kolejnym etapie odbywa się integracja poszczególnych komponentów utworzonych wcześniej przez programistów. Po zakończeniu implementacji i integracji przeprowadzane są testy i debugowanie produktu, gdzie są eliminowane wszystkie wady poprzednich etapów wytwarzania oprogramowania. Potem odbywa się etap wdrożenia produktu i jego wsparcie [4].



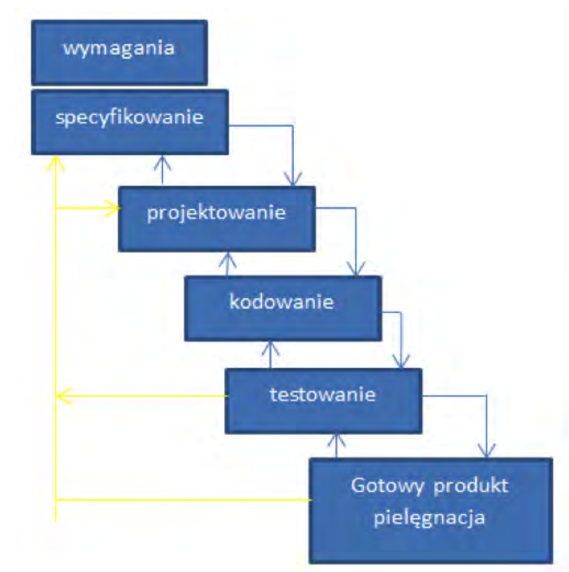
Rysunek 1: Model kaskadowy [8]

2.2. Model iteracyjny

Iteracyjny model rozwoju oprogramowania to proces tworzenia oprogramowania, poprzez małe etapy, z równoległą analizą uzyskanych pośrednich wyników, tworzone są nowe wymagania i korygowane są poprzednie etapy pracy (Rys.2). Cykl życia projektu podczas iteracyjnego oprogramowania jest podzielony na kolejne iteracje (etapy), każda z nich tworzy projekt w miniaturze, w tym wszystkie procesy tworzenia oprogramowania, ale w ramach jednego etapu opracowywana jest, tworzona tylko jedna wersja lub oddzielna część [5].

Celem iteracji jest uzyskanie wersji oprogramowania, wraz z nowymi lub zmienionymi funkcjami zaimplementowanymi w bieżącej iteracji, jak i we wszystkich poprzednich. Końcowa iteracja implementuje wszystkie wymagane funkcje produktu.

Czasowe ograniczenia i budżet projektu nie są ustalane, i formują się w trakcie realizacji, jako że wynik niemożliwe jest dokładne określenie zakresu prac dla realizacji ostatecznej wersji.



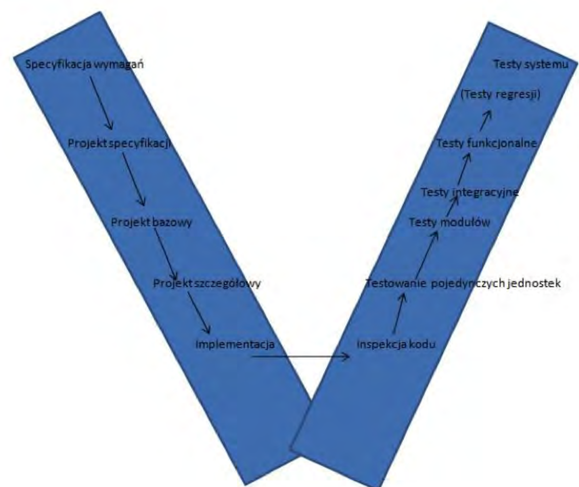
Rysunek 2: Model iteracyjny [8]

2.2. Model V

Przykładowe Model V – to ulepszona wersja modelu kaskadowego (Rys.3). Na każdym realizowanym etapie przeprowadzana jest kontrola bieżącego procesu, aby potwierdzić możliwość przejścia do następnego etapu. Testowanie rozpoczyna się od etapu wymagań i dla każdego etapu ustalany jest zestaw testów, które muszą zostać zaliczone, aby produkt mógł wejść w kolejny etap rozwoju [6].

Rysunek 3 prezentuje przebieg pracy w modelu V. Lewe część schematu przedstawia etapy projektowania i programowania, natomiast prawa - etap testowania. Przejście między jednym a drugim etapem przedstawione jest strzałką łączącą oba ramiona.

Dla każdego poziomu testowego tworzony jest dedykowany plan testowy, podczas testowania bieżącego poziomu równolegle tworzy się następny test. Wyniki testów i określenie kryteriów wejścia i wyjścia każdego etapu są określane podczas tworzenia planów testowych. Testowanie w ten sposób odgrywa szczególną rolę w tym modelu projektowania [6].



Rysunek 3: Model V [8]

2.2. Model spiralny

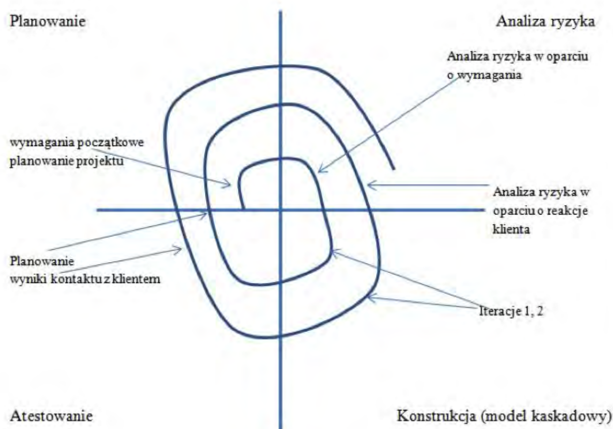
Model spiralny zawiera podstawowe zalety modelu kaskadowego (Rys.4). W tym przypadku w modelu włączono etapy: analizy i zarządzanie ryzykiem, a także wsparcie i zarządzanie. Tworzenie oprogramowania realizowane jest metodą szybkiego wytwarzania aplikacji przy użyciu języków programowania i innych narzędzi programistycznych [7].

Model polega na tym, że każda pętla zawiera zestaw operacji, który ma taką samą liczbę etapów jak modele kaskadowe. Każda część produktu i każdy poziom złożoności, zaczynając od sformułowania wymagań i kończąc kodowaniem programu, ma specjalną uwagę w tym modelu.

Początek procesu wytwarzania rozpoczyna się z określania celów projektu i ich alternatywnych wariantów celów, badania ryzyka, mapowanie planów ich realizacji i przygotowanie do następnej iteracji [7].

Kwadranty nie mają liczby cykli, są wybierane z konieczności, a iteracje są dostosowywane do projektu indywidualnie. Należy zwrócić uwagę, że kodowanie odbywa się później niż w innych modelach w celu minimalizacji ryzyka poprzez wyjaśnienie wymagań dotyczących produktu.

Najważniejszą różnicą od innych jest uwzględnienie ryzyka, na przykład przekroczenie terminów a przekroczenie wydatków. Ten model jest rzadko używany, ale przyczynił się do badania natury iteracyjnego rozwoju i znaczenia uwzględniania ryzyka.



Rysunek 4: Model spiralny [8]

3. Założenia badań

Celem badań jest określenie najlepszego rozwiązania do zarządzania rozproszonym zespołem za pomocą analizy porównawczej wybranych narzędzi informatycznych. Do badań zostało wybranych 6 najpopularniejszych narzędzi wspomagających zarządzanie projektami:

1. GanttPro.
2. Freecamp.
3. SmartSheet.
4. Kanbanize.
5. Clarizen.
6. ActiveCollab

Badania realizowane są za pomocą dwóch metod badawczych: analizy porównawczej i eksperymentu naukowego badającego przydatność tych narzędzi do zarządzania zespołem rozproszonym.

W badaniu postawiono następujące hipotezy badawcze:

H1. Program Kanbanize najlepiej realizuje wszystkie priorytetowe wymagania - z punktu widzenia wykonawców.

H2. Program Smartsheet najlepiej spełnia wszystkie priorytetowe wymagania, które są niezbędne dla skutecznego systemu zarządzania firmą IT - z punktu widzenia kierownictwa.

Na potrzeby badań zdefiniowano kryteria porównawcze, na podstawie których będą oceniane wyżej wymienionych narzędzia - w celu weryfikacji postawionych hipotezy badawcze.

Analizując potrzeby uczestników rozproszonych zespołów informatycznych dla weryfikacji hipotezy 1 zdefiniowano następujące kryteria:

K1: Jakość komunikacji
- Wygoda i możliwość współredagowania
- Efektywne zarządzanie plikami
- Integracja z e-mailem

K2: Elementy wizualne i doświadczenie użytkownika
- Intuicyjność
- Ocena graficznej części interfejsu
- Dostosowanie do wymagań użytkownika
- Ocena procesu tworzenia konta i zapraszanie innych uczestników

K3: Integracja z różnymi platformami.
- Urządzenia mobilne (Android, iOS)
- Prawidłowa współpraca z przeglądarkami

K4: Dodatkowe funkcje
- Wielojęzyczność
- Bezpieczne i szybkie logowanie za pomocą metodą SA,2FA,SSL
- Dostępność i poziom elastyczności konfiguracji
- Wsparcie dla dodatkowych wtyczek

Analizując potrzeby kadry zarządzającej rozproszonym zespołem informatycznym dla weryfikacji hipotezy 2 zdefiniowano następujące kryteria:

K1: Definiowanie funkcjonalności programu
- Możliwość tworzenia wielu projektów
- Wsparcie dla schematu Gantta
- Wsparcie dla metody rozwoju Kanban
- Zarządzanie wieloma użytkownikami
- Szablony projektów
- Integracja z systemami plików
- Import \ Eksport do XML, CVS i inne
- Kontrola wersji
- Możliwość archiwizacji projektu

K2: Zarządzanie zasobami

K3: Raportowanie i analiza

K4: Możliwości produktu
- Okres próbny do zapoznania się
- Funkcjonalność dostępna w wersji podstawowej
- Cena.

Każde badane narzędzie informatyczne zostało ocenione w trakcie tworzenia projektu testowego według

ww. kryteriów, których skale ocen zostały przedstawione w Tabeli 1 i Tabeli 2.

Tabela 1. Ocena kryteriów dla H1

Dla uczestników zespołów:	Ocena
Jakość komunikacji	0 - 6
Wizualizacja interfejsu	0 - 12
Integracja z różnymi platformami.	0 - 4
Dodatkowe funkcje	0 - 12

Tabela 2. Ocena kryteriów dla H2

Dla kierowników:	Ocena
Definicja funkcjonalności programu:	0 - 19
Zarządzanie zasobami	0 - 10
Raportowanie i analiza	0 - 10
Funkcje darmowej wersji	0 - 7

4. Wyniki analizy

W eksperymencie oceny narzędzi wykorzystano przykładowy testowy projekt - stworzenie systemu informatycznego do kompleksowej automatyzacji logistyki transportu. Przy ocenie kryteriów zastosowano system naliczania punktów, na przykład, jeśli funkcja określona jako kryterium oceny jest dostępna dla podstawowej wersji produktu to przyznajemy 1 pkt, w przypadku braku – 0 pkt. Jeśli funkcjonalność ma elastyczne zastosowanie lub dostatecznie rozwinięty potencjał, to wynik może być wyższy. Testowy projekt został zastosowany do wszystkich wybranych programów. Suma punktów dla każdego z kryteriów została pogrubiona, podobnie jak suma wszystkich punktów każdego programu dla każdej grupy potencjalnych uczestników (Tabela 3).

5. Wnioski

Wyniki analizy porównawczej zamieszczone w Tabeli 3 pokazują, że program Smartsheet uzyskał najwyższy wynik spośród wszystkich produktów. Ponadto produkt Kanbanize ma wystarczająco dużo punktów, aby pokazać szeroką funkcjonalność do pracy w projektach ze strony wykonawcy projektu – uczestnika zespołu rozproszonego, co potwierdza pierwszą hipotezę. Z jego mocnych stron warto zwrócić uwagę na prawidłowe wsparcie dla różnych przeglądarek i dostęp do mobilnych wersji tego produktu. Inne narzędzia wykazują niższe wartości sumarycznego kryterium. Podczas analizy wielokrotnie wystąpiła sytuacja, w której w. nie mógł skorzystać z istniejącej, ale niedziałającej funkcjonalności. Przykłady takich sytuacji to: jak import projektu do programu Freecamp, nieprawidłowe działanie Clarizen w przeglądarce Chrome, wyświetlanie Smartsheet w przeglądarce Explorer.

Tabela 3. Wyniki badań

Kryterium \ Program	GranttPro	Freedcamp	Smartsheet	Kanbanize	Clarizen	lab
K1: Definicja funkcjonalności programu	**16**	**11**	**15**	**12**	**10**	**8**
Możliwość tworzenia wielu projektów	1	2	2	2	2	1
Wsparcie dla schematu Gantta	2	1	1	1	1	1
Obsługa metody rozwoju Kanban	1	1	2	2	1	1
Zarządzanie wieloma użytkownikami	2	2	2	2	1	2
Szablony projektów	1	1	2	1	1	1
Integracja z systemami plików	2	2	2	2	0	1
Importuj \ Eksportuj pliki	3	1	3	1	1	1
Kontrola wersji	2	0	1	0	2	0
Możliwość archiwizacji projektu	2	1	0	1	1	0
K2: Zarządzanie zasobami	**6**	**8**	**9**	**4**	**2**	**7**
K3: Raportowanie i analiza	**6**	**7**	**8**	**8**	**2**	**9**
K4: Możliwości produktu	**5**	**5**	**5**	**6**	**4**	**7**
Okres próbny do zapoznania się	1	1	2	2	2	2
Dostępna funkcjonalność wersji podstawowej	1	1	2	2	2	2
Koszt	3	3	1	2	0	3
K1: Jakość komunikacji	**4**	**4**	**6**	**4**	**3**	**4**
Wygoda i możliwość współredagowania	2	2	2	2	1	2
Efektywne zarządzanie plikami	2	1	2	1	0	1
Praca z pocztą e-mail	0	1	2	1	2	1
K2: Wizualizacja interfejsu	**11**	**8**	**10**	**10**	**6**	**11**
Intuicyjne zrozumienie	3	1	3	3	3	3
Ocena części graficznej interfejsu	3	3	3	3	2	3
Dostosowywanie do wymagań użytkownika	2	1	2	1	0	2
Ocena procesu rejestracji i zapraszanie innych uczestników	3	3	2	3	1	3
K3: Integracja z różnymi platformami	**2**	**4**	**3**	**4**	**2**	**2**
Rodzaje urządzeń mobilnych	0	2	2	2	2	0
Prawidłowe działanie różnych przeglądarek	2	2	1	2	0	2
K4: dodatkowe funkcje	**8**	**9**	**10**	**8**	**2**	**8**
Wielojęzyczność	1	2	1	1	0	1
Bezpieczne i szybkie logowanie za pomocą technologii	2	1	2	2	1	1
Dostępność i poziom elastycznej konfiguracji	3	3	3	3	0	2
Obsługa dodatkowych wtyczek	2	3	4	2	1	4
Dla kierowców:	$\frac{33}{46}$	$\frac{31}{46}$	$\frac{37}{46}$	$\frac{30}{46}$	$\frac{18}{46}$	$\frac{31}{46}$
Dla wykonawców:	$\frac{25}{34}$	$\frac{25}{34}$	$\frac{29}{34}$	$\frac{26}{34}$	$\frac{13}{34}$	$\frac{25}{34}$

Powoduje to wiele błędów w przypadku braku szczegółowej dokumentacji i utrata kontekstu podczas tłumaczenia języka interfejsu. GanttPro doskonale nadaje się do realizacji podstawowych funkcji projektu, takich jak alokacja ról, zasobów wirtualnych, ma dobre narzędzia do importowania i eksportowania programów, co zapewnia wygodę w realizacji małych projektów.

Clarizen, który w badaniach uzyskał najniższą notę, ma niską elastyczność, zarówno dla menedżerów, jak i dla wykonawców projektu, wiele potrzebnych funkcji nie jest dostępnych lub mają istotne ograniczenia.

Wszystkie badane produkty miały zaimplementowane główne funkcje, takie jak tworzenie wielu projektów z podstawowymi informacjami w przestrzeni roboczej, podstawowe udogodnienia w zarządzaniu zespołem i zapewnianie przypisywania ról i ograniczeń w pracy. Obsługiwane były również powiadomienia dla uczestników w różnej lokalizacji dla szybkiej reakcji i kontroli. Przy intuicyjnym rozumieniu interfejsu zostało zauważone indywidualne podejście do każdego produktu, niektóre programy posiadały podobny interfejs do powszechnie znanych programów lub umożliwiały dostosowanie układu interfejsu do wymagań użytkownika i różnych form instrukcji.

Wykonana analiza porównawcza potwierdziła słuszność postawionych hipotez badawczych. Wyniki badań pokazały, że efektywna praca personelu w ramach jednego projektu, szybkie współdziałanie ze sobą, utrzymanie odpowiedniego poziomu motywacji i zainteresowania wprost proporcjonalnie wpływają na sukces projektu w przyszłości. Co więcej dobór odpowiedniego narzędzia zgodnie z wymaganiami kierownictwa, pozwala pomóc kontrolować postęp projektu jako całości, szybko reagować na problemy lub zmiany w trakcie pracy, podsumować wykonaną pracę, co przynosi maksymalne korzyści zarówno menedżerom, tak i wykonawcom. Elastyczne ustawienia, w tym dostęp do wersji mobilnej, pozwala zmaksymalizować efektywność pracy nad zadaniami, a szybki przegląd wykonanej pracy, pozwala zrealizować wszystkie wymagania wykonawców, co ułatwia pracę łatwiejszą i pozwala na dostęp z dowolnego miejsca na świecie.

Literatura

[1] J. Mulki, F. Bardhi, F. Lassk, J. Nanavaty-Dahl, Set up Remote Workers to Thrive, 2009, https://www.researchgate.net/publication/264844669_Set_Up_Remote_Workers_to_Thrive , [25.05.2020].

[2] Przyszłość pracy zdalnej, https://www.apa.org/monitor/2019/10/cover-remote-work

[3] Zespół wirtualny pochodzenie, definicja i zakres, https://www.managementstudyguide.com/virtual-team.htm [25.05.2020].

[4] M. Con, Niestandardowe historie: elastyczny rozwój oprogramowania, St. Petersburg, 2019.

[5] Definicja modelu integracyjnego, https://www.qalight.com.ua/baza-znaniy/iterativnaya-model-iterative-model , [25.05.2020].

[6] V. Pirogov, Systemy informacyjne i bazy danych: organizacja i projektowanie, BHV-Petersburg, 2009.

[7] Yu. Izbachkov, V. Petrov, A.,Vasiliev, I. Telina, Systemy informacyjne: Podręcznik dla szkół średnich, wydanie trzecie, St. Petersburg, 2011.

[8] Miejsce testowania w modelach cyklu tworzenia oprogramowania, https://www.testerzy.pl/artykuly/miejsce-testowania-w-modelach-cyklu-tworzenia-oprogramowania , [25.05.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 279–284
Received: 26 June 2020
Accepted: 6 July 2020

Comparison of MySQL, MSSQL, PostgreSQL, Oracle databases performance, including virtualization

Porównanie wydajności baz danych MySQL, MSSQL, PostgreSQL oraz Oracle z uwzględnieniem wirtualizacji

Rafał Kleweka*, Wojciech Truskowski*, Maria Skublewska-Paszkowska

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

Abstract

Oracle, MSSQL, MySQL and PostgreSQL are four of the most popular relational databases. They are often used in internet applications. This paper aims to compare the efficiency of these technologies in terms of speed using containerization with Docker. No publications that include that aspect were found among previous papers. After review of the literature, it was hypothesized that the Oracle engine would be the fastest. During the research, a series of experiments was carried out using the application, in which tests for measuring the time of instruction execution were implemented. Each query was measured 100 times and the first measurement was rejected. The obtained results confirmed the hypothesis about the superiority of the Oracle database. As in previous studies, it proved to be the fastest, also using containerization.

Keywords: virtualization; Docker; database performance

Streszczenie

Oracle, MSSQL, MySQL i PostgreSQL to cztery z najpopularniejszych relacyjnych baz danych. Są one często wykorzystywane w aplikacjach internetowych. Artykuł ma za cel porównanie efektywności tych technologii pod względem szybkości z wykorzystaniem konteneryzacji przy pomocy Docker. Wśród dotychczasowych publikacji nie znaleziono takich, które uwzględniałyby ten aspekt. Po przeglądzie literatury postawiono hipotezę, że silnik Oracle będzie najszybszy. Podczas badań przeprowadzono serię eksperymentów z użyciem aplikacji, w której zaimplementowane zostały testy do pomiaru czasu wykonania instrukcji. Każde zapytanie zostało zmierzone 100-krotnie, a pierwszy pomiar odrzucony. Uzyskane rezultaty potwierdziły hipotezę o przewadze bazy Oracle. Podobnie jak w dotychczasowych badaniach okazała się ona najszybsza, także z użyciem konteneryzacji.

Słowa kluczowe wirtualizacja; Docker; wydajność bazy danych

*Corresponding author

Email address: rafal.klewek@pollub.edu.pl (R. Klewek), wojciech.truskowski@pollub.edu.pl (W. Truskowski)

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

1. Wstęp

Bazy danych umożliwiają przechowywanie informacji wytworzonych przez działające systemy informatyczne w sposób ustandaryzowany i trwały. Stanowią nieodłączny element zdecydowanej większości wykorzystywanych na świecie aplikacji internetowych oraz programów komputerowych. Bez nich nie byłaby możliwa realizacja podstawowych zadań, których oczekuje się od systemów takich jak magazynowanie, przetwarzanie oraz zarządzanie danymi. Obecnie na rynku relacyjnych systemów zarządzania bazami danych jednymi z najpopularniejszych są cztery rozwiązania w następującej kolejności: Oracle, MySQL, Microsoft SQL Server oraz PostgreSQL [1].

Wirtualizacja jest pojęciem, które w dziedzinie technologii informatycznych odnosi się do procesu tworzenia wirtualnych zasobów, takich jak: całe platformy sprzętowe, pamięć operacyjna, nośniki danych, czy zasoby sieciowe. W roku 2013 na rynku pojawiło się oprogramowanie Docker, które umożliwia tworzenie izolowanych środowisk uruchomieniowych dla aplikacji występujących pod nazwą kontenerów, gdzie umieszczone są same aplikacje oraz biblioteki i zależności konieczne do ich poprawnego działania [2]. Technologię tę można wykorzystać do uruchomienia instancji środowisk bazodanowych na podstawie oficjalnych obrazów udostępnionych przez twórców. Po zainstalowaniu oprogramowania Docker instancjonowanie kolejnych baz danych sprowadza się do wskazania obrazu, który użytkownik chce użyć oraz uruchomienia nowego kontenera. Nie jest potrzebna konfiguracja komputera związana z konkretnym silnikiem bazodanowym, instalacja dodatkowego oprogramowania, sterowników oraz wymaganych bibliotek. Oprogramowanie to umożliwia także integrację z bardzo dynamicznie rozwijającymi się w ostatnich latach rozwiązaniami chmurowymi. Wszystkie te zalety sprawiają, że coraz więcej firm decyduje się na zastąpienie lokalnych instalacji środowisk bazodanowych na rzecz technologii Docker i wykorzystania kontenerów.

W nawiązaniu do trendów opisanych powyżej niniejszy artykuł podejmuje tematykę porównania wydajności jednych z najpopularniejszych na rynku relacyjnych rozwiązań baz danych z uwzględnieniem konteneryzacji przy pomocy oprogramowania Docker.

2. Przegląd literatury

Autorzy w artykule Performance Evaluation MySQL InnoDB and Microsoft SQL Server 2012 for Decision Support Environments [3] porównują wydajność Microsoft SQL Server 2012 z MySQL InnoDB. Autorzy porównywali czas zapytań pobierających dane na zestawach danych o wielkości 1GB, 3GB, 6GB, 12GB i 24GB. Bazą danych do analizy była hurtowania składająca się z jednej tabeli faktu i czterech tabel wymiarów. Z artykułu wynika, że Microsoft SQL Server 2012 osiąga większą wydajność niż MySQL InnoDB. Wyniki badań pokazują, że wraz ze wzrostem liczby danych spadała będzie wydajność bazy. Można wywnioskować, że Microsoft SQL Server 2012 nadaje się do operacji na małych oraz średnich zestawach danych, a MySQL InnoDB na małych zestawach danych.

Autorzy w artykule Comparison of query performance in relational a non-relation databases [4] porównują wydajność relacyjnych i nierelacyjnych baz danych. Do oceny autorzy wybrali relacyjne bazy danych Oracle, MySQL oraz MSSQL. Jako nierelacyjne technologie zostały wybrane Redis, Mongo, GraphQl i Cassandra. Autorzy mierzyli czas wykonania operacji select, insert, update oraz delete. Testy były przeprowadzane na zestawach danych liczących 10 000 i 100 000 rekordów. Wyniki badań pokazują, że nierelacyjne bazy danych są bardziej wydajne od technologii relacyjnych. Stosunek wydajności rozwiązań nierelacyjnych do baz relacyjnych wyniósł 1:3 dla operacji select, 1:15 dla operacji insert, 1:9 dla instrukcji update i 1:6 dla operacji delete. Z poddanych analizie relacyjnych silników najwydajniejszy był MSSQL. MySQL miał najniższą wydajność. W porównaniu tylko nierelacyjnych baz danych najwydajniejszy był Mongo, najgorzej wypadł Redis i Cassandra.

Autorzy publikacji An Introduction to Docker and Analysis of its Performance [5] szczegółowo opisali elementy oprogramowania Docker tj. klient, serwer, obrazy, rejestr i kontenery. W artykule autorzy przedstawili porównanie technologii Docker oraz maszyny wirtualnej KVM. Analiza pokazuje, że w przypadku Dockera można łatwiej zarządzać zasobami, szybkość obliczeniowa jest większa, a uruchomienie kontenera jest znacznie szybsze niż uruchomienie maszyny wirtualnej. Maszyna wirtualna natomiast jest lepszym wyborem, gdy głównym kryterium jest poziom izolacji procesu.

Na konferencji ICACEA Ann Joy przedstawiła porównanie pomiędzy kontenerami linuksowymi [6], a maszynami wirtualnymi. Autorka wskazała, że kontenery są bardziej wydajne i lepiej przystosowane do skalowania. Z przeprowadzonych badań wynika, że kontenery skalują się 22 razy szybciej niż maszyny wirtualne. Ze względu na lepsze gospodarowanie zasobami i skalowalność rozwiązania wykorzystujące konteneryzację zmniejszają wykorzystanie zasobów. Autorka wskazuje, iż maszyny wirtualne posiadają lepsze zabezpieczenia, przez co lepiej nadają się do wykorzystania w przypadku serwisów wymagających dużej izolacji.

W artykule Performance analysis of selected database systems: MySQL, MS SQL, PostgerSQL in the context of web applications [7] autor analizował wydajność trzech systemów bazodanowych MySQL, MSSQL i PostgreSQL. Do badań twórca pracy wykorzystał aplikację internetową. Eksperymenty zostały przeprowadzone przy pomocy oprogramowania Apache JMeter, które do połączeń z bazą danych używa interfejsu JDBC. Autor porównywał średni czas zapytań odczytu, dodawania, modyfikacji oraz usuwania rekordów. Wyliczona została także mediana oraz odchylenie standardowe dla każdego scenariusza badawczego. Z badań wynika, że najwydajniejszą bazą danych jest PostgreSQL. Najgorszą wydajność zaobserwowano na silniku MySQL. Z artykułu można wnioskować, że baza MSSQL w mniejszym stopniu korzysta z pamięci podręcznej niż MySQL oraz PostgreSQL.

W artykule Comparative analysis of databases working under the control of Windows system [8] autorzy analizowali wydajność oraz wykorzystywanie zasobów trzech systemów bazodanowych MySQL, PostgreSQL oraz Firebird na systemie operacyjnym Windows 10 Pro 64-bit. Analiza wydajności polegała na wykonaniu 10 powtórzeń każdego scenariusza, a wynikiem była średnia wartość z otrzymanych wyników. Z przeprowadzonych badań wynika, że spośród wybranych systemów bazodanowych na systemie Windows najwydajniejszy jest MySQL oraz najmniej obciąża on zasoby dyskowe. Baza danych Firebird najmniej obciąża procesor.

Przegląd artykułów naukowych pokazuje, że istnieje ciągłe zainteresowanie badaniami baz danych oraz wirtualizacją, również z wykorzystaniem oprogramowania Docker. Po przeglądzie literatury autorzy postawili hipotezę, iż wydajność relacyjnych baz danych na małych zbiorach danych jest bardzo zbliżona, ale wraz ze wzrostem liczby danych wydajność bazy Oracle będzie zauważalnie wyższa niż pozostałych. Jednocześnie nie napotkano badań, w których porównania dokonano przy uwzględnieniu wirtualizacji, w związku z czym w eksperymentach uwzględniony zostanie aspekt, którego nie obejmowała żadna z omawianych prac.

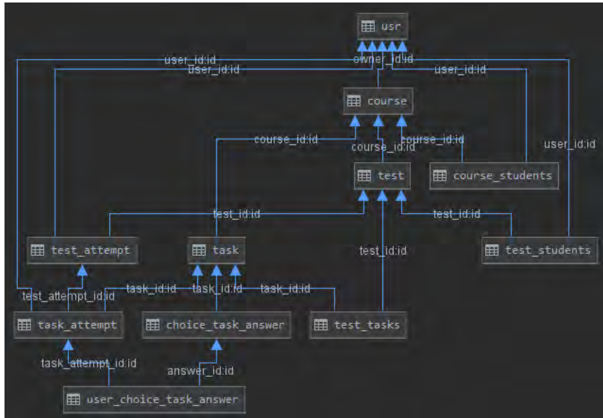
3. Aplikacja testowa

Do przeprowadzania badań stworzona została aplikacja webowa, której podstawowym założeniem jest możliwość tworzenia oraz rozwiązywania testów składających się z pytań wielokrotnego wyboru. Część serwerowa aplikacji została zaimplementowana z użyciem szkieletu aplikacyjnego Symfony [9] dla języka PHP, a do stworzenia interfejsu graficznego użytkownika posłużono się językiem JavaScript, wykorzystując technologię Vue.js [10]. Wśród głównych funkcjonalności należy wymienić:

- możliwość rejestracji oraz logowania do utworzonych kont,
- ograniczenie dostępu do wszystkich zakładek dla niezalogowanych użytkowników,
- trzy poziomy uprawnień w systemie, określone rolami: administrator, egzaminator, kursant,

- tworzenie kursów wewnątrz których można potem tworzyć zadania, gdzie dostępne do określenia są: tytuł, treść, maksymalna liczba punktów do uzyskania oraz lista odpowiedzi z zaznaczeniem, które są poprawne,
- definiowanie testów, w których skład wchodzą utworzone uprzednio zadania,
- wyszukiwanie, rozwiązywanie oraz podgląd wyników dla uzyskanych zestawów testowych.

Do przechowywania danych wygenerowanych podczas korzystania z aplikacji wykorzystywana jest baza danych o schemacie widocznym na rysunku 1.



Rysunek 1: Schemat bazy danych

Oprócz funkcjonalności związanych z interfejsem użytkownika zostały zaimplementowane specjalne klasy operacji służące do wykonywania pojedynczych operacji wymagających połączenia z bazą danych. Klasy te napisane zostały w taki sposób, aby po ich użyciu dostępny był czas wykonania realizowanej instrukcji bazodanowej. Zostały one wykorzystane do badań, gdzie przy użyciu testów jednostkowych zostały wywołane 100-krotnie, a czasy zapytań zapisywane były do plików w formacie csv.

4. Metoda badawcza

Eksperyment polegał na uruchomieniu testów jednostkowych udostępnianych przez dedykowaną aplikację, która realizowała operacje wymagające komunikacji z bazą danych. Do badań użyto baz:

- MySQL ver. 8.0.17,
- Microsoft SQL Server ver. 15.00.4033,
- PostgreSQL ver. 12.3,
- Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production.

Testy były uruchamiane na systemie Windows 10 Education x64 kompilacja 18362 z użyciem Hyper-V i kontenerów linuksowych, a uruchomienie środowisk bazodanowych w kontenerach zostało uzyskane przy pomocy platformy Docker w wersji 19.03.8. Do zachowania stanu bazy katalogi robocze zostały podmontowane z systemu plików systemu macierzystego. Wszystkie obrazy baz danych uruchomione zostały z domyślnymi ustawieniami dostarczonymi przez twórców. Port na którym działała baza w kontenerze odpowiadał portowi na maszynie macierzystej. Podczas każdej próby uruchomiona była tylko jedna instancja kontenera. Schemat bazy danych był tworzony przy zastosowaniu *collation* z alfabetem polskim, niewrażliwy na wielkość znaków oraz niewrażliwy na akcent. W tabeli 1 została przedstawiona specyfikacja techniczna maszyny, która posłużyła do wykonania badań wydajności. Na czas prowadzonych badań środowisko platformy Docker otrzymało zasoby widoczne w tabeli 2.

Tabela 1: Specyfikacja urządzenia testowego

Procesor	Procesor Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2201 MHz, Rdzenie: 6, Procesory logiczne: 12
Dysk	Model SPCC Solid State Disk 500GB
RAM	16GB 2400MHz

Tabela 2: Zasoby platformy Docker

Liczba rdzeni procesora	2
RAM	3,5 GB
Swap	1 GB

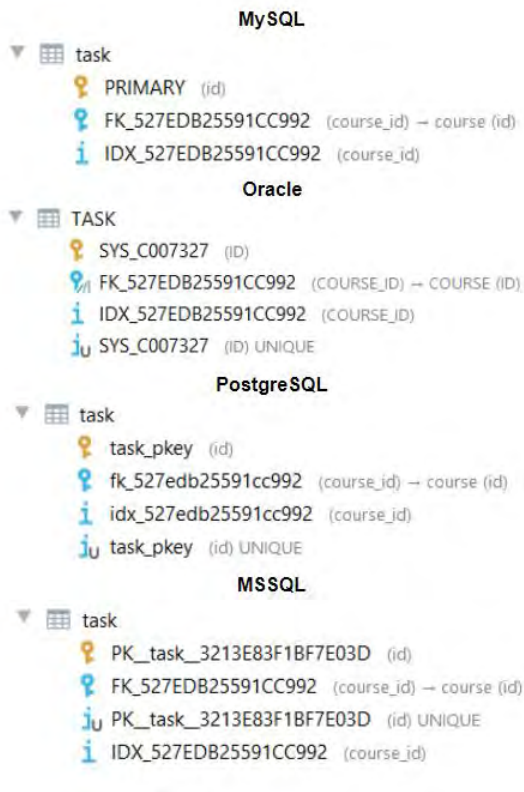
Każda operacja wykonana została 100-krotnie w celu uzyskania jak najbardziej rzetelnych rezultatów. Przy porównywaniu wyników odrzucany był pierwszy rezultat, który ze względu na brak zbudowanego planu zapytania trwał zauważalnie dłużej niż kolejne. Zmierzono czasy wykonywanych instrukcji bazodanowych dla pojedynczych operacji wyszukiwania danych. Podczas pobierania rekordów badany był wpływ różnych warunków zawężających oraz funkcji. Te same pomiary zostały przeprowadzone na każdym z porównywanych rozwiązań relacyjnych baz danych: MySQL, MSSQL, PostgreSQL oraz Oracle. Każdy scenariusz został wykonany dla trzech zestawów danych różniących się liczbą rekordów w poszczególnych tabelach. Dla wszystkich technologii bazodanowych wygenerowane automatycznie informacje, którymi zostały wypełnione, były identyczne. Tabela 3 zawiera zestawienie liczby rekordów we wszystkich zestawach testowych:

Tabela 3: Zestawianie liczby rekordów dla zestawów testowych

Tabela/Nr zestawu	1	2	3
course	5 000	25 000	100 000
task	50 000	250 000	1 000 000
usr	30 000	150 000	600 000
test	10 000	50 000	200 000
course_student	25 000	125 000	500 000

Obrazy baz MySQL, MSSQL i PostgreSQL zostały uruchomione przy użyciu obrazów udostępnionych przez producenta z publicznego repozytorium *DockerHub*. Do budowy obrazu bazy Oracle wykorzystana została instrukcja z artykułu *Oracle magazine* [12]. Schemat bazy danych dla każdego silnika był analogiczny. Utworzona została taka sama liczba kluczy obcych oraz indeksów dla odpowiadających kolumn. Efekt uzyskano dzięki zastosowaniu do tworzenia struktur mechanizmów udostępnianych przez szkielet aplika-

cyjny Symfony. Struktura bazodanowa jest generowana, zgodnie ze wzorcem definiowanym przy pomocy klas zwanych *encjami*. Są to pliki zawierające pojedyncze klasy PHP z dodatkowymi adnotacjami opisującymi atrybuty dla tworzonej tabeli. Na rysunku 2 zostały zaprezentowane indeksy i klucze utworzone dla tabeli *Task*.



Rysunek 2: Indeksy oraz klucze dla tabeli *Task*

5. Scenariusze badawcze

Pierwszy scenariusz polegał na wykonaniu pojedynczego zapytania wybierającego dane bez żadnych zawężeń zbioru wynikowego. Zapytanie badane w czasie uruchamiania tego testu jest widoczne na listingu 1.

Listing 1: Kod instrukcji SQL wykonywanej w pierwszym teście

```
SELECT t0_.id                AS id_0,
       t0_.name              AS name_1,
       t0_.number_of_points  AS number_of_points_2,
       t0_.content           AS content_3,
       t0_.creation_date     AS creation_date_4,
       t0_.enabled           AS enabled_5,
       t0_.type              AS type_6,
       t0_.image_name        AS image_name_7,
       t0_.image_size        AS image_size_8,
       t0_.updated_at        AS updated_at_9
FROM task t0_
ORDER BY t0_.name ASC
LIMIT 20
```

Dla kolejnego scenariusza pod uwagę brana była szybkość zapytania wyszukującego korzystającego z klauzuli zawężającej *WHERE*. Treść polecenia realizowanego przez bazę służy do pobrania rekordów z tabeli *Task* zawężonych o wartość z kolumny *name*. Wygenerowane zapytanie zostało przedstawione na listingu 2.

Listing 2: Zapytanie używane dla scenariusza drugiego

```
SELECT t0_.id                AS id_0,
       t0_.name              AS name_1,
       t0_.number_of_points  AS number_of_points_2,
       t0_.content           AS content_3,
       t0_.creation_date     AS creation_date_4,
       t0_.enabled           AS enabled_5,
       t0_.type              AS type_6,
       t0_.image_name        AS image_name_7,
       t0_.image_size        AS image_size_8,
       t0_.updated_at        AS updated_at_9
FROM task t0_
WHERE t0_.id = 1
ORDER BY t0_.id ASC
```

Scenariusz trzeci użyty został do sprawdzenia wydajności porównywanych rozwiązań relacyjnych baz danych w przypadku wyszukiwania danych z wykorzystaniem podzapytania skorelowanego. Wyszukiwane są kursy oraz nazwy użytkowników tworzących, gdzie dla drugiej kolumny do wybrania wartości użyte zostało podzapytanie skorelowane. Treść wynikowej instrukcji SQL została zamieszczona na listingu 3.

Listing 3: Select z użyciem podzapytania

```
SELECT c0_.name AS name_0,
       (
         SELECT u1_.username
         FROM usr u1_
         WHERE u1_.id = c0_.owner_id
       ) AS sclr_1
FROM course c0_
ORDER BY c0_.name ASC
LIMIT 20
```

Kolejny test pozwolił na mierzenie szybkości realizacji operacji wybierania danych z wielokrotnym łączeniem tabel oraz klauzulą grupującą. Zliczana była liczba zadań dla poszczególnych testów w systemie, a pełna wersja zapytania przedstawiona jest na listingu 4.

Listing 4: Ciało zapytania z wielokrotnym łączeniem tabel oraz klauzulą grupującą

```
SELECT t0_.id AS id_0, t0_.name AS name_1, count(t1_.id) AS sclr_2
FROM test t0_
       INNER JOIN test_tasks t2_ ON (t0_.id = t2_.test_id)
       INNER JOIN task t1_ ON (t1_.id = t2_.task_id)
       INNER JOIN course c3_ ON (c3_.id = t0_.course_id)
GROUP BY t0_.id, t0_.name
LIMIT 20
```

6. Analiza wyników

W pierwszym badanym scenariuszu sprawdzany był czas wykonania instrukcji *SELECT* z pojedynczej tabeli *Task* bez żadnych dodatkowych warunków ograniczających. Rezultat miał być posortowany alfabetycznie według wartości kolumny *name*. Przyglądając się wynikom prób zauważyć można, że zdecydowanie najlepiej

pod względem wydajności dla instrukcji wybierania danych z pojedynczej tabeli wypadło rozwiązanie firmy Oracle. Czas wykonania zapytania był bliski jednej milisekundzie dla wszystkich zbiorów danych, gdzie pozostałe rozwiązania bazodanowe uzyskały wyniki na poziomie kilkudziesięciu lub kilkuset ms. Na podstawie minimalnych czasów zauważyć można, iż najgorzej w tej próbie wypadły technologie MySQL oraz MSSQL, które dla największego zestawu danych uzyskały średnie czasy około 700 ms, gdzie dla identycznego zestawu rekordów system PostgreSQL osiągnął wynik 289 ms, a Oracle zaledwie 1,19 ms. Ogromna różnica przemawiająca na korzyść ostatniego badanego rozwiązania, co zostało zobrazowane na tabeli 4.

Tabela 4: Średni czas wykonania zapytań dla scenariusza drugiego

Baza/Nr zestawu	Średni czas dla 1 zestawu [ms]	Średni czas dla 2 zestawu [ms]	Średni czas dla 3 zestawu [ms]
MySSQL	32,5106	156,774	694,385
MSSQL	35,3669	165,584	712,929
PostgrSQL	26,033	73,3064	289,373
Oracle	1,2326	1,2168	1,1903

W następnym scenariuszu była sprawdzana wydajność baz danych podczas wykonania zapytania wybierającego dane z pojedynczej tabeli *Task* przy ograniczeniu na kolumnie, dla której nie było założonego indeksu. Do tego celu wykorzystana została instrukcją języka SQL reprezentowana przez klauzulę *WHERE*. Rozpatrując wyniki zaprezentowane na tabeli 5 można zauważyć, że najgorsze wyniki uzyskał MySQL. Rozmiar bazy danych nie wpłynął na wydajność technologii firmy Oracle. Średni czas zapytania na każdym zestawie rekordów dla bazy danych Oracle wynosił ok. 1,3 ms. MSSQL i PostgreSQL uzyskały zbliżoną wydajność na poziomie 80-90 milisekund. Wraz ze wzrostem liczby rekordów różnica wydajności pomiędzy MSSQL i PostgreSQL zauważalnie się zwiększała. Przy trzecim zestawie danych PostgreSQL osiągnął lepszy czas o 16,799 ms od bazy firmy Microsoft.

Tabela 5: Średni czas wykonania zapytań dla scenariusza drugiego

Baza/Nr zestawu	Średni czas dla 1 zestawu [ms]	Średni czas dla 2 zestawu [ms]	Średni czas dla 3 zestawu [ms]
MySSQL	8,931	163,688	734,137
MSSQL	10,303	27,668	92,925
PostgrSQL	8,9309	24,867	76,126
Oracle	1,282	1,279	1,316

W trzecim scenariuszu badaniu poddana została instrukcja języka SQL wykorzystująca podzapytanie skorelowane. Wykorzystane one zostało do uzyskania listy kursów w systemie wraz z nazwą użytkownika tworzącego. Z przeprowadzonych testów wynika, iż dla pierwszego zestawu danych o najmniejszej liczbie rekordów w tabelach średnie czasy były zbliżone dla wszystkich baz, z wyjątkiem Oracle, które już w tej próbie uzyskało wynik ponad dwukrotnie gorszy niż pozostałe. Dla kolejnych zbiorów testowych tendencja utrzymała się i najdłużej zapytanie wykonywane było dla bazy danych Oracle. W trzeciej próbie średni czas wyniósł 199,53 ms. Różnica pomiędzy pozostałymi rozwiązaniami relacyjnych baz danych najlepiej widoczna była podczas testów dla najliczniejszego zestawu danych testowych. Z analizy wyników dla bazy wypełnionej największą liczbą rekordów wynika, że najlepiej poradził sobie MSSQL ze średnim czasem realizacji zapytania równym 29,2807 ms, następnie PostgreSQL z wynikiem 50,2726 ms. Trzeci był MySQL z rezultatem na poziomie 72,3806 ms. Średnie czasy uzyskane podczas wykonywania instrukcji z podzapytaniem skorelowanym dla poszczególnych silników bazodanowych zostały zaprezentowane na tabeli 6.

Tabela 6: Średni czas wykonania zapytań dla scenariusza trzeciego

Baza/Nr zestawu	Średni czas dla 1 zestawu [ms]	Średni czas dla 2 zestawu [ms]	Średni czas dla 3 zestawu [ms]
MySSQL	5,1094	19,4097	72,3806
MSSQL	5,522	14,4066	29,2807
PostgrSQL	5,3445	14,5357	50,2726
Oracle	12,8493	50,1	199,527

Scenariusz czwarty dotyczył porównania baz w przypadku wykonywania zapytania korzystającego z wielu złączeń pomiędzy tabelami, grupowania oraz zliczania. Klauzula *GROUP BY* pozwala na grupowanie zbioru wynikowego na podstawie identycznej wartości we wskazanej kolumnie. Testowane zapytania pozwala uzyskać liczbę zadań w poszczególnych testach. Z analizy danych wynika, że zdecydowanie najgorzej poradził sobie MySQL Jest to widoczne na tabeli 7, gdzie zastosowano skalę logarytmiczną. Dla największego zestawu danych średni czas wykonania zapytania wyniósł blisko 6 sekund, gdzie dla identycznego zbioru testowego Oracle osiągnął czas równy 245,7870 ms. Widać tutaj, że silnik MySQL jest złym wyborem, jeśli chodzi o operacje grupowania danych przy wielokrotnych złączeniach pomiędzy tabelami. Dla baz PostgreSQL oraz MSSQL wyniki były zbliżone, niezależnie od liczby rekordów w systemie. Najlepszym rozwiązaniem bazodanowym podczas tego badania okazał się MSSQL.

Tabela 7: Średni czas wykonania zapytań dla scenariusza czwartego

Baza/Nr zestawu	Średni czas dla 1 zestawu [ms]	Średni czas dla 2 zestawu [ms]	Średni czas dla 3 zestawu [ms]
MySSQL	147,634	1191,55	5702,37
MSSQL	1,5111	1,495	1,5836
PostgrSQL	3,2147	3,2906	3,2823
Oracle	13,8178	60,9568	245,787

7. Wnioski

W artykule przeprowadzone zostały badania wydajności czterech najpopularniejszych na rynku relacyjnych baz danych, czyli Oracle, MSSQL, MySQL oraz PostgreSQL. Wszystkie technologie bazodanowe uruchomione zostały w środowisku wirtualnym z zastosowaniem oprogramowanie Docker. Po przeglądzie literatury podejmującej temat porównania wydajności baz danych uruchamianych w wyniku standardowej instalacji postawiona została teza, iż podczas zastosowania konteneryzacji przy pomocy oprogramowania Docker najlepiej poradzą sobie silniki Oracle oraz MSSQL.

Udało się wykonać wszystkie zaplanowane scenariusze testowe, a uzyskane rezultaty pozwoliły na wyciągnięcie następujących wniosków:

- podczas operacji wyszukiwania danych bez użycia żadnych warunków zawężających najszybsza okazała się baza Oracle. Niezależnie od rozmiaru tabeli, dla której wykonywane było zapytanie średni czas wykonania wyniósł około 1,2 ms, co zostało przedstawione na tabeli 4. Analiza wyników pozwoliła zaobserwować, że ta sama operacja na innych silnikach trwała w przypadku trzeciego zestawu testowego kilkaset razy dłużej. Najgorzej w tej próbie wypadł system bazodanowy MSSQL
- przy teście zapytania SQL korzystającego z klauzuli WHERE ponownie bezkonkurencyjna okazała się baza Oracle, niezależnie od liczby rekordów. Dla pozostałych technologii zwiększony rozmiar danych powodował wydłużenie czasu wykonania. Najgorzej poradziła sobie baza MySQL, co widać na tabeli 5,
- dla testowanych zapytań bazodanowych, system Oracle najgorzej poradził sobie w przypadku instrukcji SQL, w której użyto podzapytania skorelowanego, co pokazuje tabela 6. Średni czas wykonania był najgorszy dla wszystkich badanych zbiorów danych. W teście tym największą efektywność osiągnął MSSQL, co jest widoczne zwłaszcza dla ostatniego zestawu testowego, gdzie średni wynik wyniósł 29,28 ms,
- analiza rezultatów badań dla złożonego zapytania wykorzystującego wielokrotne złączenia, grupowanie oraz zliczanie rekordów pozwoliła wykazać, że w przypadku tego typu operacji najlepszym okazał się silnik MSSQL. Co ciekawe, średni czas wykonania był zbliżony niezależnie od liczby rekordów w tabelach. Badanie to pozwoliło także zaobserwować, że baza MySQL źle radzi sobie w przypadku zapytań, gdzie zastosowano wielokrotne złączenia. Minimalny czas dla najliczniejszego zbioru danych wyniósł ponad 5 sekund, gdzie najszybsza baza osiągnęła rezultat na poziomie 1,5836 ms, co widać na tabeli 7.

Uzyskane wyniki pokazują, że podczas korzystania z relacyjnych baz danych w środowisku wirtualnym uruchomionym przy pomocy oprogramowania Docker, w większości przypadków najbardziej wydajnym okazała się baza Oracle. Warto również zauważyć, że do testów wykorzystana została darmowa wersja Express, gdzie dla edycji komercyjnej rezultaty mogłyby być jeszcze lepsze. Uzyskane rezultaty pokazały, że hipoteza postawiona po przeglądzie literatury była zasadna. Potwierdziło się, że czas wyszukiwania danych rośnie wraz z wielkością przeszukiwanego zbioru rekordów w bazie, a Oracle radzi sobie najlepiej w większości sprawdzanych scenariuszy.

Literatura

[1] G. Eason, B. Noble, I. N. Sneddon, On certain integrals of Lipschitz-Hankel type involving products of Bessel functions, Phil. Trans. Roy. Soc. London A247 (1955) 529–551.

[2] Ranking najpopularniejszych baz danych, https://pypl.github.io/DB.html, [24.06.2020]

[3] Czym jest Docker, https://docs.docker.com/get-started/overview/, [24.06.2020]

[4] R. Almeida, P. Furtado, J. Bernardino, Performance Evaluation MySQL InnoDB and Microsoft SQL Server 2012 for Decision Support Environments, Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering - C3S2E '15. (2008).

[5] R. Čerešňák, M. Kvet, Comparison of query performance in relational a non-relation databases, Transportation Research Procedia. 40 (2019) 170–177.

[6] M. Yasir, A Review on Introduction to Docker and its Features, International Journal of Advanced Research in Computer Science and Software Engineering. 8 (2018) 12.

[7] A.M. Joy, Performance comparison between Linux containers and virtual machines, 2015 International Conference on Advances in Computer Engineering and Applications. (2015).

[8] K. Lachewicz, Performance analysis of selected database systems: MySQL, MS SQL, PostgerSQL in the context of web applications. Journal of Computer Sciences Institute. 14, (Mar. 2020), 94-100.

[9] S. Stets, G. Kozieł, Comparative analysis of databases working under the control of Windows system, Journal of Computer Sciences Institute. 13 (2019) 298–301.

[10] Salehi, S. 2016. Mastering symfony. Packt Publishing Limited.

[11] Czym jest Vue.js, https://vuejs.org/v2/guide/, [24.06.2020].

[12] Tworzenie kontenera dla bazy danych Oracle, https://blogs.oracle.com/oraclemagazine/deliver-oracle-database-18c-express-edition-in-containers [24.06.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 285–292
Received: 23 June 2020
Accepted: 7 July 2020

Comparative analysis of selected object-relational mapping systems for the .NET platform

Analiza porównawcza wybranych systemów mapowania obiektowo-relacyjnego dla platformy .NET

Marcin Bobel*, Krzysztof Drzazga*, Maria Skublewska-Paszkowska

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

Abstract

This article is devoted to the comparison of two object-relational mapping systems supported by .NET platform - Entity Framework Core and NHibernate. The research hypothesis "framework NHibernate is more efficient than Entity Framework Core in the context of DML operations" was put forward. In order to make an efficiency analysis of ORM frameworks, a desktop application was designed and implemented to enable testing and visualization of results. The NHibernate framework turned out to be much more efficient than Entity Framework Core in single tests and slightly faster in bulk tests. The stability of both frameworks was similar.

Keywords: .NET; ORM; Entity Framework Core; NHibernate

Streszczenie

Niniejszy artykuł został poświęcony porównaniu dwóch systemów mapowania obiektowo-relacyjnego wspieranych przez platformę .NET – Entity Framework Core oraz NHibernate. Postawiono hipotezę badawczą, „szkielet NHibernate jest wydajniejszy niż Entity Framework Core w kontekście operacji DML". W celu przeprowadzenia analizy wydajności szkieletów ORM, zaprojektowano i zaimplementowano aplikację desktopową umożliwiającą wykonanie testów oraz wizualizację wyników. Szkielet NHibernate okazał się zdecydowanie wydajniejszy niż Entity Framework Core w testach pojedynczych i nieznacznie szybszy w testach masowych. Stabilność obu szkieletów kształtowała się na podobnym poziomie.

Słowa kluczowe: .NET; ORM; Entity Framework Core; NHibernate

*Corresponding author

Email addresses: **marcin.bobel@pollub.edu.pl** (M. Bobel), **krzysztof.drzazga@pollub.edu.pl** (K. Drzazga)

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

1. Wstęp

Na przestrzeni ostatnich lat, wraz ze wzrostem zapotrzebowania na usługi informatyczne, nastąpił rozwój narzędzi umożliwiających ich realizację. Można zaobserwować powstawanie nowych języków programowania, a także dynamiczny rozwój istniejących języków oraz szkieletów programistycznych. Wszelkie zmiany mają na celu poprawę takich aspektów rozwiązań informatycznych jak bezpieczeństwo i wydajność, jednocześnie ułatwiając programistom ich implementację.

Nowoczesne aplikacje zwykle nie powstają w oparciu o jedną technologię, lecz są zbiorem kilku współgrających ze sobą rozwiązań. Programista ma do dyspozycji coraz bardziej zaawansowane szkielety programistyczne, które pozwalają na szybszą implementację rozwiązania. Ponadto, wspomniane szkielety są odpowiedzialne za niektóre czynności, których implementacja niegdyś była zadaniem programisty. W przypadku takich rozwiązań, ważna jest równowaga pomiędzy czynnościami wykonywanymi niejawnie przez szkielet programistyczny, a czynnościami zaimplementowanymi przez twórcę aplikacji. Istotna jest również świadomość i kontrola nad szkieletem programistycznym.

Systemy mapowania obiektowo-relacyjnego znane powszechnie jako systemy ORM (ang. Object-Relational Mapping), są odpowiedzialne za mapowanie rekordów bazodanowych na obiekty w konkretnym języku programowania [1]. Takie podejście znacząco przyspiesza implementację operacji bazodanowych takich jak: wstawianie danych, aktualizacja danych, usuwanie danych jak również odczytywanie danych.

W niniejszej pracy poddano analizie systemy mapowania obiektowo-relacyjnego wspierane przez platformę .NET. Wspomniana platforma została stworzona przez firmę Microsoft i obecnie stanowi jedną z najpopularniejszych platform programistycznych. Postawiono hipotezę badawczą, „szkielet NHibernate jest wydajniejszy niż Entity Framework Core w kontekście operacji DML". Trafność hipotezy zweryfikowano na podstawie przeprowadzonych badań, których wyniki przedstawiono w dalszej części artykułu.

2. Przegląd literatury

W niniejszym rozdziale zostaną zaprezentowane źródła literaturowe, których tematyka jest zbliżona do tematu niniejszej pracy. Pierwsza ze znalezionych pozycji literaturowych to artykuł zatytułowany "Wydajność pracy z bazami danych w aplikacjach ASP.NET MVC", autorstwa Pawła Borysa i Beaty Pańczyk [2]. W tym artykule porównywane są szkielety Entity Framework 6.1.3 i NHibernate w wersji 4.6.1, dla systemów bazodano-

wych MySQL oraz MSSQL. Analiza przeprowadzona została pod kątem wydajności czasowej jak również pamięciowej dla operacji wybierania oraz wstawiania danych. W większości scenariuszy testowych wydajniejszy okazał się szkielet NHibernate.

Kolejną pozycją jest prelekcja pod tytułem "Fetch performance comparison of object relational mapper in .NET platform" wygłoszona przez Witoona Wiphusitphunpola oraz Thitiporna Lertrudachakula podczas czternastej międzynarodowej konferencji ECTI-CON [3]. Jednym z wykorzystanych szkieletów był Entity Framework Core w wersji 1.1.0. Zostały zbadane operacje wybierania danych w jednym wątku oraz w wielu wątkach.

Podczas trzeciej międzynarodowej konferencji dotyczącej obiektów i baz danych, która odbyła się w 2010 roku we Frankfurcie, przedstawione zostały wyniki badań Stevicy Cvetkovicia i Dragana Jankovicia na temat "Analiza porównawcza funkcji i wydajności narzędzi ORM w środowisku .NET" [4]. Autorzy w pracy użyli szkieletów NHibernate 2.0.1 i Entity Framework 4, a środowiskiem testowym był MS SQL Server 2005. Testowane były operacje wstawiania, wybierania i usuwania rekordów. Dla operacji wstawiania szybszym szkieletem okazał się Entity Framework natomiast dla operacji usuwania NHibernate.

Ostatnim źródłem konferencyjnym jest prelekcja pochodząca z Międzynarodowej konferencji Beyond Databases Architectures and Structures pod tytułem "Analiza wydajnościowa szkieletów mapowania obiektowo relacyjnego dla platformy .NET" [5]. Wykorzystane narzędzia to Entity Framework 5.0, NHibernate 3.3.1, natomiast platformy testowe to Microsoft SQL Server 2012 i Postgre SQL 9.2. Testy wydajnościowe zostały przeprowadzone dla operacji wstawiania, wybierania, modyfikacji i usuwania rekordów z tabel. W testach wstawiania, jak również usuwania, wydajniejszy okazał się szkielet NHibernate, natomiast dla operacji modyfikacji szkielet Entity Framework.

W artykule "Porównanie wydajności metod CRUD (przyp. ang. Create, Read, Update, Delete) przy użyciu systemów mapowania obiektowo relacyjnego dla .NET", porównane zostały szkielety Entity Framework Core 2.2, NHibernate 5.2.3 i Dapper 1.50.5 [6]. Platformą testową był MS SQL, a testowane operacje to wstawianie, wybieranie, modyfikacja i usuwanie rekordów dla relacji jeden do jednego oraz jeden do wielu. Testowano czas wykonania operacji i wykorzystaną pamięć. W testach wstawiania oraz w niektórych testach usuwania rekordów, wydajniejszy okazał się szkielet NHibernate, natomiast w testach modyfikacji oraz w części testów usuwania rekordów lepsze wyniki uzyskał szkielet Entity Framework Core.

3. Obiekty badań

3.1. Entity Framework Core w wersji 2.2.4

Entity Framework Core jest szkieletem programistycznym klasy ORM autorstwa firmy Microsoft [7]. Jego wykorzystanie pozwala niemalże na eliminację konieczności implementacji operacji dostępu do danych z baz danych. Obecnie istnieją dwie wersje omawianego systemu mapowania obiektowo-relacyjnego. Podstawowa wersja Entity Framework 6 jest rozwijana i stabilizowana od kilku lat, jednakże wspiera jedynie urządzenia z systemem Windows. Drugą, znacznie młodszą wersją, jest Entity Framework Core, którego platformą docelową jest .NET Core zapewniający wieloplatformowość. Cechuje go lekkość i rozszerzalność, a sam projekt jest otwartoźródłowy, co pozwala na lepsze wsparcie społeczności programistów.

3.2. NHibernate w wersji 5.1.1

NHibernate to otwartoźródłowy szkielet programistyczny dla platformy .NET [8]. Szkielet ten jest implementowany w ramach projektu Hibernate przez społeczność programistyczną. Zapewnia wsparcie dla wielu popularnych relacyjnych systemów baz danych. Pozwala na mapowanie obiektów ze strony aplikacji na rekordy i relacje po stronie systemu bazodanowego. Umożliwia tworzenie więzów integralności dla tabel oraz struktur bazodanowych. Konfiguracja odbywa się z poziomu pliku konfiguracyjnego XML lub poprzez kod C#.

4. Metoda badawcza

W celu weryfikacji poprawności postawionej hipotezy przeprowadzono badania wydajności szkieletów Entity Framework Core oraz NHibernate za pomocą zaimplementowanej w tym celu aplikacji desktopowej. Główne zadanie aplikacji stanowiło przeprowadzenie testów według wskazanych parametrów. Dodatkowo do rozwiązania zostały dodane funkcjonalności pozwalające na automatyzację wizualizacji wyników, obliczanie parametrów statystycznych oraz utrwalanie wyników.

Testy wydajności wykonane we wspomnianej wyżej aplikacji zostały przeprowadzone według przyjętych scenariuszy testowych przedstawionych w tabeli 1. Ze względu na duży narzut czasowy wykonywania pojedynczego scenariusza oraz deterministyczne zachowanie szkieletów podczas testów, zdecydowano o wykonaniu jednego powtórzenia testu dla każdego scenariusza.

Tabela 1: Scenariusze testowe

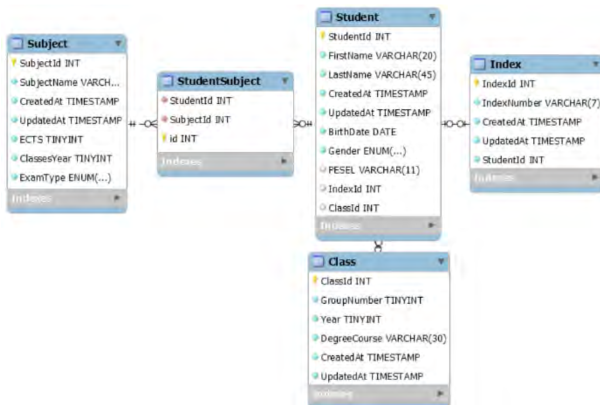
Numer scenariusza	Liczba rekordów/powtórzeń	Operacja
1.	500	Create
2.		Update
3.		Delete
4.	5000	Create
5.		Update
6.		Delete

W celu eliminacji wpływu jednego testu na drugi, każdą z trzech rodzajów operacji (create, update, delete) testowano oddzielnie. Aplikacja umożliwia wykonywanie testów masowych oraz pojedynczych. Liczby wskazane w drugiej kolumnie tabeli 1, oznaczają liczbę powtórzeń testu pojedynczego oraz liczbę rekordów dla jakiej wykonano test masowy.

Przyjęto następujące progi liczby rekordów/liczby powtórzeń:

- 500 – mała liczba rekordów/powtórzeń;
- 5000 – duża liczba rekordów/powtórzeń.

Na rysunku 1 przedstawiono strukturę bazy danych wykorzystaną w testach. Podczas projektowania bazy danych głównym założeniem było wykorzystanie wszystkich binarnych relacji bazodanowych. W celu umożliwienia testowania poszczególnych relacji oddzielnie, przyjęto pewne uproszczenia w modelu bazy danych. Jednym z nich jest opcjonalność relacji pomiędzy tabelą *Student* a *Index*. Inny przykład stanowi brak unikalności dla kolumn reprezentujących numer indeksu i numer PESEL.



Rysunek 1: Schemat testowej bazy danych

W tabeli 2 przedstawiono parametry techniczne sprzętu wykorzystanego do przeprowadzenia testów wydajności.

Tabela 2: Parametry sprzętu testowego

Nazwa modelu	Dell Inspiron 5570
System operacyjny	Windows 10 Home
Procesor	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80GHz
Pamięć RAM	16 GB
Typ systemu	x64
Dysk	SSD – 256GB, HDD – 1 TB

5. Aplikacja wspomagająca analizę wydajności

Aplikacja wspomagająca analizę wydajności powstała jako aplikacja desktopowa WPF dla platformy .NET Framework. Zostały w niej wykorzystane trzy utworzone wcześniej biblioteki: biblioteka zawierająca główny moduł testujący oraz dwie biblioteki implementujące operacje bazodanowe według określonego interfejsu dla Entity Framework Core i NHibernate. Baza danych została utworzona za pomocą silnika bazodanowego SQL Server Express LocalDB w wersji 13.0.4001, wbudowanego w narzędzie Visual Studio.

Aplikacja posiada takie funkcjonalności jak:

- parametryzacja testu;
- obliczenie parametrów statystycznych;
- zapis/odczyt parametrów statystycznych do/z pliku;
- eksport pomiarów do arkusza kalkulacyjnego;
- generowanie wykresów;
- zapis wykresów do pliku.

Na rysunku 2 przedstawiono okno parametryzacji testów. Składa się ono z trzech kolumn, z których każda zawiera parametry testów określonej operacji. Możliwe jest wykonanie testu dla następujących rodzajów relacji: jeden do jednego, jeden do wielu, wiele do wielu oraz dla pojedynczej tabeli. Za pomocą przycisków typu *checkbox* wskazywana jest również informacja czy ma zostać wykonany test masowy czy też pojedynczy. W polach tekstowych określana jest liczba powtórzeń/rekordów wykorzystanych w danym teście.

Rysunek 2: Okno parametryzacji pomiaru

Na listingu 1 przedstawiono implementację pomiaru czasu wykonania operacji tworzenia rekordu w relacji jeden do jednego dla Entity Framework Core. Na początku generowane są dane testowe, następnie uruchamiany jest pomiar czasu, po czym obiekty dodawane są do kontekstu. Wywołanie metody *SaveChanges()* powoduje utworzenie transakcji wstawiającej rekordy do bazy danych. Na końcu następuje zatrzymanie pomiaru czasu i zwracany jest wynik. Pozostałe operacje zostały zaimplementowane w sposób analogiczny.

Na listingu 2 przedstawiono implementację identycznej operacji jak w poprzednim akapicie z tym, że dla szkieletu NHibernate. Na początku, podobnie jak w przypadku Entity Framework Core, tworzone są dane testowe i uruchamiany jest pomiar czasu. Następnie w sposób jawny tworzona jest transakcja i w niej obiekty dodawane są do sesji. Transakcja zatwierdzana jest za pomocą metody *Commit()*. Na końcu zatrzymywany jest pomiar czasu oraz następuje zwrócenie wyniku.

Listing 1: Implementacja pomiaru czasu w Entity Framework Core

```
public TimeSpan SingleCreateOneToOne()
{
    Index index = TestDataFactory.GetIndex();
    Student student = TestDataFactory.GetStudent();

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    db.Indexes.Add(index);
    db.Students.Add(student);
    student.Index = index;
    db.SaveChanges();

    stopwatch.Stop();

    return stopwatch.Elapsed;
}
```

Listing 2: Implementacja pomiaru czasu operacji w NHibernate

```
public TimeSpan SingleCreateOneToOne()
{
    var student = NHibernateDataGenerator.GetStudent();
    var index = NHibernateDataGenerator.GetIndex(1);

    var watch = new Stopwatch();
    watch.Start();

    using (var transaction = session.BeginTransaction())
    {

        student.IndexId = index;
        session.Save(index);
        session.Save(student);

        transaction.Commit();
    }

    watch.Stop();
    return watch.Elapsed;
}
```

6. Wyniki badań

Wyniki badań z podziałem na relacje przedstawiono w kolejnych podrozdziałach zgodnie z podziałem na operacje. Parametry, które zaprezentowano w tabelach to średni czas wykonania operacji dla pojedynczego rekordu oraz odchylenie standardowe dla testów pojedynczych. Wykresy ilustrują współczynnik zmienności.

6.1. Operacja wstawiania rekordów

W tabeli 3 został zaprezentowany średni czas wykonania pojedynczej operacji dla scenariusza wstawiania pięciuset rekordów. W przedstawionym scenariuszu w większości testów wydajniejszym szkieletem okazał się NHibernate. Jedynym wyjątkiem jest test masowy dla relacji wiele do wielu. Warto zauważyć, że w testach masowych różnice pomiędzy szkieletami były nieduże.

W tabeli 4 zaprezentowano średni czas wykonania pojedynczej operacji w scenariuszu wstawiania pięciu tysięcy rekordów. W przedstawionych wynikach widoczny jest fakt osiągnięcia przewagi wydajności przez szkielet NHibernate w testach pojedynczych. W testach masowych wartości te są porównywalne. Można zauważyć również drastyczną różnicę pomiędzy testami pojedynczymi a masowymi, które w skrajnym przypadku są kilkaset razy mniejsze w testach masowych.

W przedstawionych scenariuszach testowych widoczny jest wzrost średnich wartości pomiędzy scenariuszami w testach pojedynczych, natomiast w przypadku testów masowych wartości średnie zmalały. Dodatkowo wartości odchylenia standardowego są większe w przypadku szkieletu Entity Framework Core niż szkieletu NHibernate.

Tabela 3: Czas wykonania pojedynczej operacji dla scenariusza wstawiania pięciuset rekordów

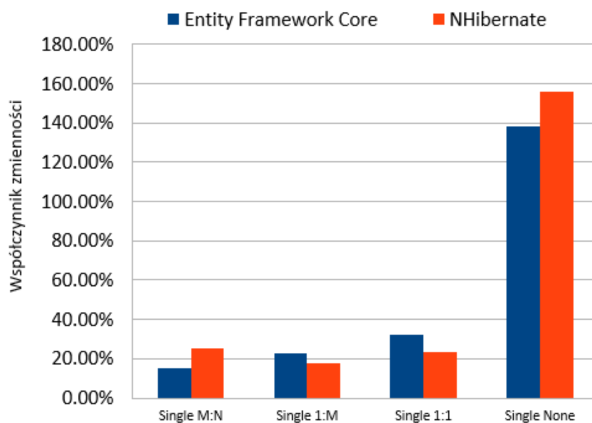
Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji -NHibernate (ms)
Pojedynczy M:N	79,371 ± 12,14	31,243 ± 7,91
Pojedynczy 1:N	53,417 ± 12,07	24,03 ± 4,27
Pojedynczy 1:1	32,33 ± 10,4	13,175 ± 3,07
Pojedynczy brak relacji	10,835 ± 14,99	5,599 ± 8,72
Masowy M:N	1,184	1,231
Masowy 1:N	1,622	1,282
Masowy 1:1	3,078	1,972
Masowy brak relacji	1,371	0,986

Tabela 4: Czas wykonania pojedynczej operacji dla scenariusza wstawiania pięciu tysięcy rekordów

Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji –NHibernate (ms)
Pojedynczy M:N	678,461 ± 90,42	243,566 ± 45,27
Pojedynczy 1:N	405,371 ± 67,73	117,22 ± 28,59
Pojedynczy 1:1	185,509 ± 59,13	50, 371 ± 14,18
Pojedynczy brak relacji	41,521 ± 23,56	13,359 ± 7,26
Masowy M:N	0,7	0,795
Masowy 1:N	0,992	0,723
Masowy 1:1	1,288	1,122
Masowy brak relacji	0,632	0,577

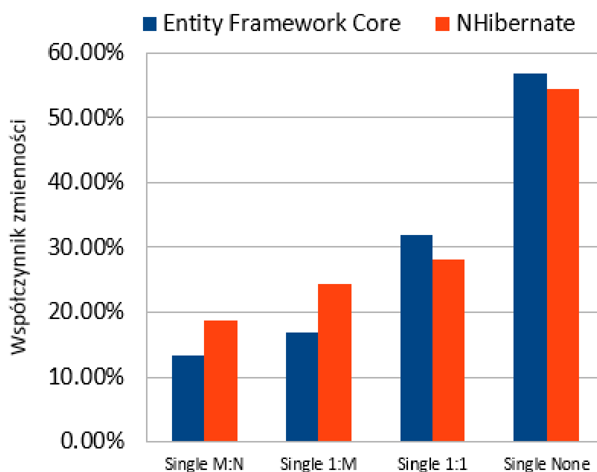
Na rysunku 3 przedstawiony został wykres wartości współczynnika zmienności w scenariuszu wstawiania pięciuset rekordów. Współczynnik ten jest to miara względna odchylenia standardowego oraz średniego czasu wstawienia pojedynczego rekordu i może być interpretowana jako stabilność. Uzyskane wyniki dla tego scenariusza są zbliżone dla obu szkieletów. Najmniejszą stabilnością charakteryzuje się operacja wsta-

wiania przy braku relacji. W pozostałych testach wartości są dużo mniejsze



Rysunek 3: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza wstawiania pięciuset rekordów

Na rysunku 4 został przedstawiony współczynnik zmienności dla przeprowadzonych testów pojedynczych w scenariuszu wstawiania pięciu tysięcy rekordów. Wartości dla testów relacji wiele do wielu i jeden do wielu są mniejsze dla szkieletu Entity Framework Core, w pozostałych przypadkach mniejsze wartości tego współczynnika osiągnął szkielet NHibernate. W ogólności oba szkielety uzyskały podobny poziom stabilności.



Rysunek 4: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza wstawiania pięciu tysięcy rekordów

6.2. Operacja modyfikacji rekordów

W tabeli 5 został przedstawiony średni czas wykonania pojedynczej operacji dla scenariusza modyfikacji pięciuset rekordów. W tym scenariuszu średni czas wykonania operacji dla szkieletu NHibernate okazał się dużo mniejszy niż dla szkieletu Entity Framework Core. Różnice te są najbardziej widoczne w testach masowych.

Tabela 5: Czas wykonania pojedynczej operacji dla scenariusza modyfikacji pięciuset rekordów

Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji –NHibernate (ms)
Pojedynczy M:N	80,503 ± 12	33,355 ± 5,47
Pojedynczy 1:N	53,191 ± 11,9	20,691 ± 4,02
Pojedynczy 1:1	32,093 ± 10	11,519 ± 2,78
Pojedynczy brak relacji	9,856 ± 3,98	4,721 ± 1,54
Masowy M:N	1,305	0,152
Masowy 1:N	3,995	0,354
Masowy 1:1	3,292	0,24
Masowy brak relacji	3,468	0,142

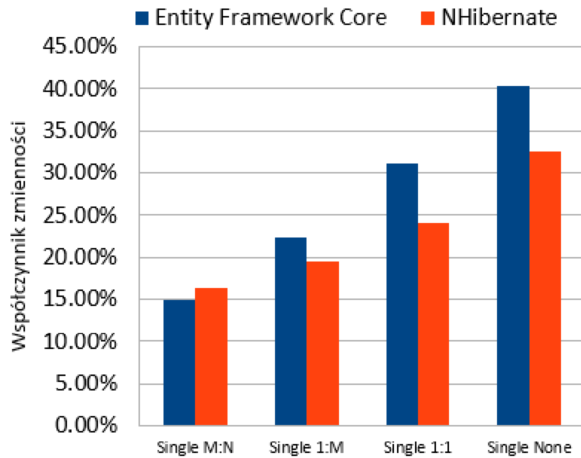
W tabeli 6 przedstawiono średni czas wykonania pojedynczej operacji w scenariuszu modyfikacji pięciu tysięcy rekordów. Wydajniejszym szkieletem w niniejszym teście okazał się szkielet NHibernate, jednak w testach masowych różnice są nieduże.

W obu scenariuszach średnie czasy wykonania operacji w testach pojedynczych okazały się zdecydowanie większe od testów masowych

Tabela 6: Czas wykonania pojedynczej operacji dla scenariusza modyfikacji pięciu tysięcy rekordów

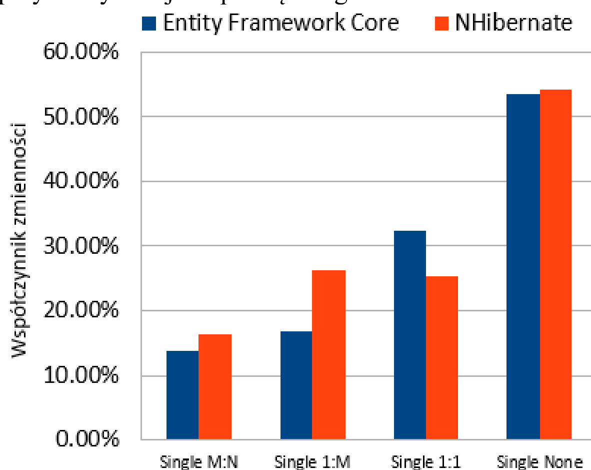
Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji –NHibernate (ms)
Pojedynczy M:N	694,553 ± 96,43	269,033 ± 43,72
Pojedynczy 1:N	406,224 ± 68,58	140,966 ± 36,98
Pojedynczy 1:1	184,931 ± 59,99	64,783 ± 16,37
Pojedynczy brak relacji	42,112 ± 22,51	25,168 ± 13,66
Masowy M:N	0,388	0,321
Masowy 1:N	1,325	0,426
Masowy 1:1	0,904	0,152
Masowy brak relacji	0,84	0,143

Na rysunku 5 przedstawiony został wykres wartości współczynnika zmienności dla scenariusza modyfikacji pięciuset rekordów. Poza testem dla relacji wiele do wielu stabilniejszym szkieletem okazał się NHibernate. Różnice pomiędzy szkieletami nie są jednak duże, a wartość współczynnika zmienności nie przekracza 40%.



Rysunek 5: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza modyfikacji pięciuset rekordów

Na rysunku 6 zostały przedstawione wartości współczynnika zmienności dla testów pojedynczych w scenariuszu modyfikacji pięciu tysięcy rekordów. Dla trzech z czterech relacji stabilniejszym szkieletem okazał się Entity Framework Core – wyniki te zostały osiągnięte dla relacji wiele do wielu, jeden do wielu oraz przy modyfikacji niepowiązanego rekordu.



Rysunek 6: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza modyfikacji pięciu tysięcy rekordów

6.3. Operacja usuwania rekordów

W tabeli 7 został przedstawiony średni czas wykonania pojedynczej operacji w scenariuszu usuwania pięciuset rekordów. W większości testów niniejszego scenariusza wydajniejszym szkieletem okazał się NHibernate. Entity Framework Core był szybszy w pojedynczym teście usuwania przy braku relacji oraz w teście masowym dla relacji jeden do wielu.

W tabeli 8 przedstawiono średni czas wykonania pojedynczej operacji w scenariuszu usuwania pięciu tysięcy rekordów. Zdecydowanie wydajniejszym szkieletem ponownie okazał się NHibernate, jednakże różnice w testach masowych nie okazały się duże. W niniejszym scenariuszu można zauważyć duże różnice wartości odchylenia standardowego między szkieletami w pojedynczych testach relacji jeden do wielu oraz wiele do wielu. W pozostałych dwóch testach wartości są zbliżone.

Ponownie średni czas usuwania pojedynczego rekordu w obu scenariuszach w testach masowych jest mniejszy niż w testach pojedynczych. W testach masowych wraz ze wzrostem liczby rekordów wydajność operacji również rośnie.

Tabela 7: Czas wykonania pojedynczej operacji dla scenariusza usuwania pięciuset rekordów

Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji – NHibernate (ms)
Pojedynczy M:N	35,361 ± 10,67	14,202 ± 3,94
Pojedynczy 1:N	10,988 ± 4,75	6,061 ± 1,89
Pojedynczy 1:1	3,626 ± 1,06	3,432 ± 0,54
Pojedynczy brak relacji	2,374 ± 0,5	2,473 ± 0,23
Masowy M:N	1,665	0,156
Masowy 1:N	0,207	0,533
Masowy 1:1	6,876	1,915
Masowy brak relacji	4,112	0,169

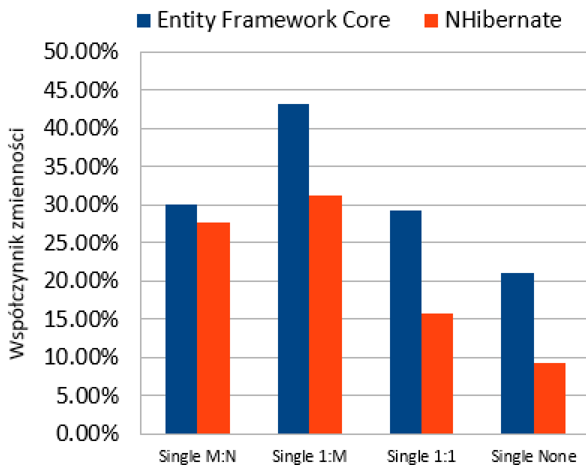
Tabela 8: Czas wykonania pojedynczej operacji dla scenariusza usuwania pięciu tysięcy rekordów

Nazwa testu	Czas wykonania pojedynczej operacji - Entity Framework Core (ms)	Czas wykonania pojedynczej operacji – NHibernate (ms)
Pojedynczy M:N	221,002 ± 60,37	62,924 ± 21,4
Pojedynczy 1:N	53,169 ± 31,98	14,622 ± 7,51
Pojedynczy 1:1	2,913 ± 0,99	2,179 ± 0,4
Pojedynczy brak relacji	2,124 ± 0,55	1,606 ± 0,56
Masowy M:N	0,539	0,088
Masowy 1:N	0,224	0,173
Masowy 1:1	2,425	2,838
Masowy brak relacji	0,945	0,086

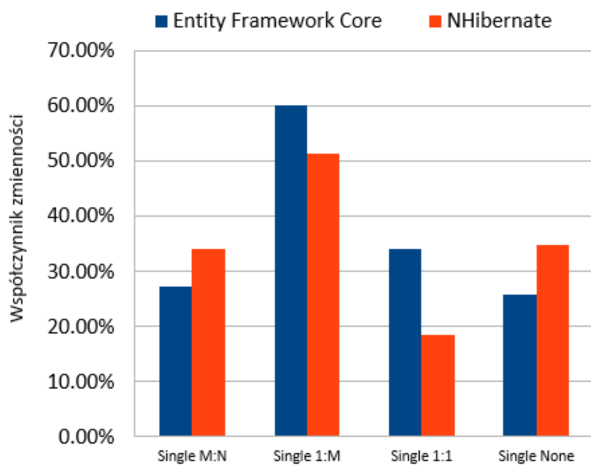
Na rysunku 7 przedstawiono wartość współczynnika zmienności w scenariuszu usuwania pięciuset rekordów. We wszystkich testach stabilniejszy okazał się szkielet NHibernate, jednakże w tym scenariuszu wartości te znacząco różnią się pomiędzy szkieletami dla wszystkich relacji poza relacją wiele do wielu.

Na rysunku 8 przedstawiono wartości współczynnika zmienności dla scenariusza usuwania pięciu tysięcy rekordów. W testach relacji wiele do wielu oraz dla rekordów niepowiązanych, stabilniejszym szkieletem okazał się szkielet Entity Framework Core, natomiast

dla relacji jeden do wielu i jeden do jednego mniejszą wartość współczynnika osiągnął szkielet NHibernate.



Rysunek 7: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza usuwania pięciuset rekordów



Rysunek 8: Wartość współczynnika zmienności w testach pojedynczych dla scenariusza usuwania pięciu tysięcy rekordów

7. Analiza porównawcza

Konfiguracja szkieletów programistycznych wygląda podobnie w obu porównywanych szkieletach programistycznych [7, 8]. Zarówno w Entity Framework Core jak i NHibernate, zależności do projektu można dodać za pomocą managera pakietów NuGet. Oba szkielety wspierają obsługę najpopularniejszych systemów bazodanowych takich jak Sql Server, Sql Lite, PostgreSQL, MySql, Oracle DB, jednakże dostęp do niektórych z nich w Entity Framework Core jest płatny.

W Entity Framework Core modele definiowane są jako klasy składające się z właściwości (ang. *properties*) [7, 8]. Modele opisywane są w dwojaki sposób – za pomocą atrybutów lub tak zwanego *fluent API*. W NHibernate definiowanie modeli przebiega w podobny sposób z taką różnicą, że właściwości muszą zostać oznaczone słowem kluczowym *virtual*. Modele opisywane są za pomocą plików XML lub z wykorzystaniem biblioteki FluentNHibernate.

Implementacja operacji bazodanowych jest mniej złożona w szkielecie Entity Framework Core [7, 8]. W NHibernate operacje te implementowane są w bardziej niskopoziomowy sposób – zarządzanie transakcjami spoczywa na programiście podczas, gdy w Entity Framework Core operacje te wykonywane są niejawnie po stronie szkieletu. Ponadto, wartą zauważenia cechą jest brak wsparcia dla operacji masowych przez NHibernate.

Zarządzenie schematem bazy danych w Entity Framework Core odbywa się za pomocą mechanizmu migracji, który pozwala na generowanie schematu na podstawie modeli i klasy kontekstu (klasy dziedziczącej po *DbContext*) [7, 8]. Powyższa funkcjonalność wykorzystywana jest poprzez wiersz poleceń, a wygenerowane pliki mogą aktualizować strukturę bazy danych zarówno podczas implementacji jak i w czasie wykonywania programu. W NHibernate schemat bazy danych tworzony jest na podstawie plików mapujących, których konfiguracje zawarto w kodzie aplikacji.

Oba szkielety cechuje nowoczesność oraz wspieranie typowych dla szkieletów ORM funkcjonalności takich jak [7, 8, 9, 10]:

- *Forward Engineering*;
- *Reverse Engineering*;
- Surowe zapytania SQL;
- Zapytania asynchroniczne;
- Ładowanie leniwe (ang. *Lazy Loading*);
- Ładowanie chciwe (ang. *Eager Loading*);
- Śledzenie zmian (ang. *Change Tracking*);
- Zapytania LINQ.

NHibernate nie wspiera popularnego mechanizmu wstrzykiwania zależności (ang. *Dependency Injection*).

8. Podsumowanie

Podsumowując badania wydajności, w oparciu o wykresy i tabele zaprezentowane w rozdziale szóstym, zdecydowanie lepszy wynik uzyskał szkielet NHibernate, co udowadnia postawioną hipotezę badawczą - "szkielet NHibernate jest wydajniejszy niż Entity Framework Core w kontekście operacji DML". W testach pojedynczych różnice pomiędzy szkieletami okazały się znaczące, natomiast w testach masowych znikome. Warta odnotowania jest również duża różnica w wynikach pomiędzy testami masowymi oraz pojedynczymi. Współczynnik zmienności prezentuje się na zbliżonym poziomie we wszystkich testach pojedynczych, co wskazuje na podobny poziom stabilności obu porównywanych szkieletów ORM.

Literatura

[1] Object-relational mapping, https://en.wikipedia.org/wiki/Object-relational_mapping, [16.06.2020]

[2] P. Borys, B. Pańczyk: Wydajność pracy z bazami danych w aplikacjach ASP.NET MVC. Journal of Computer Science Institute 6, 2018.

[3] D. Zmaranda, L-L. Pop-Fele, C. Győrödi, R. Győrödi, G. Pecherle: Performance Comparison of CRUD Methodsusing NET Object Relational Mappers: A Case

Study (IJACSA) International Journal of Advanced Computer Science and Applications,Vol. 11, No.1, 2020.

[4] W. Wiphusitphunpol, T. Letrusdachakul: Fetch performance comparison of object relational mapper in .NET platform. [W]: 14th International Conference on Electrical Engineering/Electronics, IEEE, Computer, Telecommunications and Information Technology (ECTI-CON), Phuket, Tajlandia 7 listopada 2017.

[5] S. Cvetković, D. Janković: A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment. [W]: Objects and Databases ICOODB 2010. Lecture Notes in Computer Science, vol 6348. Springer, Berlin, Heidelberg. Frankfurt, Niemcy. 28-30 września 2010.

[6] A. Gruca, P. Podsiadło: Performance Analysis of .NET Based Object–Relational Mapping Frameworks. [W]: Beyond Databases, Architectures, and Structures. BDAS 2014. Communications in Computer and Information Science, vol 424. Springer, Cham. Ustroń Polska, 27-30 maja 2014.

[7] Dokumentacja szkieletu programistycznego Entity Framework Core, https://docs.microsoft.com/en-us/ef/core/, [22.04.2020].

[8] Dokumentacja szkieletu programistycznego NHibernate, https://nhibernate.info/, [29.03.2020].

[9] Dokumentacja biblioteki FluentNHibernate https://github.com/FluentNHibernate/fluent-nhibernate/wiki, [10.05.2020].

[10] Entity Framework Core Tutorial, https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx, [25.04.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 293–299
Received: 23 June 2020
Accepted: 7 July 2020

Analysis of the use of Java and C# languages for building a mobile application for the Android platform

Analiza zastosowania języków Java oraz C# do budowy aplikacji mobilnej na platformę Android

Michał Jankowski*, Maria Skublewska-Paszkowska

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

Abstract

Mobile applications for the Android platform can be implemented using Java or C#. The article presents a comparison of the time performance of these languages when sending various text, image and video files in a mobile application. The tests were carried out using two mobile applications with identical functionalities. Based on the collected data, the server application calculated statistics, such as, for example, the time required to send 1 MB of data depending on the file type and size. Based on the results obtained, it was proved that in the case of data transfer via a wireless network, an application written in Java is characterized by greater time efficiency than an analogous application written in C#.

Keywords: mobile applications; REST API; Java; C#

Streszczenie

Aplikacje mobilne na platformę Android można implementować z użyciem języków Java lub C#. Artykuł przedstawia porównanie wydajności czasowej tych języków podczas przesyłania różnych plików typu tekstowego, obrazu i wideo w aplikacji mobilnej. Badania zostały przeprowadzone z użyciem dwóch aplikacji mobilnych o identycznych funkcjonalnościach. Na podstawie zebranych danych aplikacja serwerowa obliczyła statystyki, takie jak na przykład czas wymagany na przesłanie 1 MB danych w zależności od typu oraz rozmiaru pliku. Na podstawie otrzymanych wyników udowodniono, że w przypadku transferu danych poprzez sieć bezprzewodową aplikacja napisana w języku Java cechuje się większą wydajnością czasową niż analogiczna aplikacja napisana w języku C#.

Słowa kluczowe: aplikacje mobilne; REST API; Java; C#

*Corresponding author

Email address: **michal.jankowski1@pollub.edu.pl** (M. Jankowski)

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

1. Wstęp

Aplikacje mobilne w dzisiejszych czasach towarzyszą ludziom na każdym kroku: czy to bankowość elektroniczna, komunikacja, czy też przeznaczanie czasu na korzystanie z serwisów internetowych lub społecznościowych bądź gier. Narastający trend nowych aplikacji w gwałtownym tempie zaczął rozwijać się po rozpowszechnieniu telefonów komórkowych zdolnych owe aplikacje obsłużyć. Mowa tu konkretnie o telefonach z ekranem dotykowym wyposażonym na przykład w system Android. Nowy trend smartfonów pozwolił utworzyć kolejną gałąź Informatyki w programowaniu, czyli systemy mobilne. Mnogość rozwiązań oraz bezcenna przydatność telefonów komórkowych oraz różnych aplikacji w codziennym życiu zapoczątkowała powstawanie firm specjalizujących się w tworzeniu i rozwijaniu aplikacji mobilnych. Bardzo często wykorzystywane są w nich właśnie języki Java oraz C#.

Celem artykułu jest udowodnienie tezy: Aplikacja mobilna na systemy Android napisana w języku Java oferuje większą wydajność czasową niż analogiczna aplikacja napisana w języku C#. Do przeprowadzenia badania wymagane są elementy bądź czynności, które będzie można zmierzyć odpowiednimi jednostkami miary, takimi jak np. milisekundy. Do tego celu stworzona została aplikacja z funkcjami przesyłania oraz pobierania plików, a także wyświetlaniem statystyk oraz ewentualnym ich pobieraniem. Pozwala to tworzyć raporty oparte o pliki wygenerowane poprzez aplikację i wyświetlać dane na przykład w postaci czytelnych wykresów. Ze względu na konieczność analizy oraz przechowywania zgromadzonych danych stworzona została dodatkowo aplikacja serwerowa. Połączenie pomiędzy aplikacjami mobilnymi, a aplikacją serwerową odbywa się za pomocą sieci bezprzewodowej. Jest to konieczne ze względu na warunki użytkowania telefonów komórkowych – są one połączone rozległymi sieciami, których prędkości transferu danych rosną w coraz szybszym tempie [1].

2. Przegląd literatury

W celu zrozumienia konieczności przeprowadzenia badań w obszarze różnic wydajnościowych czasów należy zasięgnąć literatury. Znalezione materiały porównujące języki Java oraz C# zestawiają najczęściej różnice składniowe obu języków [2-5]. Autorzy prac analizują sposób zarządzania pamięcią oraz różnice w składniach bądź metodach języków. Artykuły opisujące różnice wydajnościowe aplikacji to albo czyste dane zebrane za pomocą konkretnej aplikacji [6, 7], albo szerzej opisane przeprowadzone badania z wykorzysta-

niem metod numerycznych (wielokrotne wykonywanie dzieleń, obliczanie wielomianów) [8]. Literatura skupiająca się na systemie Android w niewielkim stopniu opisuje różnice pomiędzy omawianymi językami. Zazwyczaj są to tak, jak wcześniej wspomniane, różnice składniowe [9, 10], czy też badania na plikach lokalnych [11, 12]. Aktualny stan wiedzy autora pracy pozwala stwierdzić, że nie znaleziono badań skupiających się na zagadnieniu omówionym w niniejszej pracy.

3. Języki oraz style architektury programowania

Języki programowania, których wydajność czasowa badana jest w niniejszej pracy to Java oraz C#. Java stosowana jest zarówno w aplikacjach mobilnych jak i desktopowych, do gier, czy też aplikacji komercyjnych [13]. Najbardziej popularną aplikacją do tworzenia oprogramowania kojarzącą się z systemem Android jest Android Studio, które wykorzystuje właśnie ten język [14, 15].

W dalszym etapie do stworzenia aplikacji serwerowej użyty został szkielet aplikacji Spring [16], czyli inaczej platforma pomagająca w budowaniu aplikacji. Spring Framework oparty został na języku Java. Przenośność, wygoda programowania, szybkość tworzenia aplikacji oraz prosta logika zależności - to cechy opisujące Javę, które przemawiają za częstym wybieraniem jej przez programistów.

Drugi język wykorzystany w pracy to C#. Podobnie jak, Java jest językiem obiektowym z hierarchią klas. Zastosowanie C# również sprowadza się do gier i programów komercyjnych, jednak w tym przypadku są to rozbudowane silniki symulacyjne czy też skomplikowane systemy [17]. Warto tu nadmienić, że firma Microsoft mocno skupia się na wykorzystaniu języka C# w swoich narzędziach (Visual Studio, Office). Firma ta również udostępnia tzw. silnik Unity, czyli środowisko do prostego tworzenia gier na przykład na systemy mobilne. Wcześniej wymienione narzędzie Visual Studio, oczywiście przy użyciu różnych bibliotek i programów kompilujących kod źródłowy, czyli translacji języka na taki zrozumiały dla systemu, pozwoliło stworzyć aplikację napisaną w języku C# dla systemu Android [10].

Badanie, które zostało przedstawione w pracy, umożliwiło zestawić dwa bardzo podobne sobie języki programowania cechujące się obiektowością, czyli paradygmatem bazującym na pojęciach klasy i obiektu, jak również sposobem zarządzania pamięcią [18]. Ten ostatni fakt oznacza to, że programista nie musi martwić się o zapewnienie aplikacji wymaganej pamięci sprzętowej.

Dodatkowo, celem komunikacji i wymiany danych pomiędzy użytkownikami a serwerami stosuje się bardzo często tak zwane REST API, czyli w uproszczeniu pewien styl ze zdefiniowanymi regułami [19]. Pozwala on w sposób prosty zaimplementować metody, które będą potem wykorzystywane przez aplikacje systemów mobilnych do wymiany danych. Wcześniej wymieniona platforma Spring w tym przypadku bardzo dobrze pozwala wykorzystać swój potencjał [20].

4. Metoda badań

Badania zostały przeprowadzone z użyciem dwóch środowisk testowych. W tabeli 1 przedstawione zostały specyfikacje środowisk testowych podczas przeprowadzania badań.

Tabela 1: Specyfikacja środowisk testowych

	Stanowisko nr 1	**Stanowisko nr 2**
Nazwa urządzenia	Laptop	Komputer stacjonarny
Typ pamięci masowej	SSD	SSD RAID 0
Procesor	2 rdzenie; 4 wątki; 2,2 GHZ	4 rdzenie; 4 wątki; 4,4 GHZ
Pamięć RAM	8 GB	12 GB
Połączenie sieciowe	Wi-Fi	LAN

Ustawienie połączeń sieciowych pozwala zasymulować typowe warunki użytkowania telefonu komórkowego połączonego poprzez sieć. Na stanowisku nr 1 uruchomiona została aplikacja serwerowa, zaś na stanowisku nr 2 aplikacja mobilna. Mierzonymi parametrami w aplikacjach są:

- *czas pobierania pliku;*
- czas wysyłania pliku;
- czas opóźnienia komunikacji pomiędzy aplikacją mobilną a serwerową.

Aplikacja serwerowa otrzymuje powyższe parametry od aplikacji mobilnych i zapisuje je w bazie danych pod odpowiednimi indeksami statystyk. Na podstawie tych danych wyliczane są dodatkowe parametry. Poniżej rozpisane formuły przedstawione są dla wysyłania plików. W przypadku pobierania, owe formuły zamiast wyrażeń oznaczających wysyłanie mają zaimplementowane wyrażenia oznaczające pobieranie.

1. Średnia wartość opóźnienia (ms) – wartość wyrażona w milisekundach, opisująca średni czas potrzebny na otrzymanie odpowiedzi na żądanie przez aplikację mobilną od aplikacji serwerowej Formuła wyliczająca ową wartość jest:
   suma czasów zebranych opóźnień operacji / liczba plików wysłanych

2. Czas przesyłania w przeliczeniu na 1 MB dla danego pliku (ms) - oznacza średni czas zagregowany ze wszystkich operacji pobierania oraz wysyłania podzielony na języki aplikacji źródłowych. Formułą licząca nie jest zwykła średnia lecz dokładniejsza,

mniej podatna na niepewność pomiarową dla plików o niskich rozmiarach:
suma czasów wysyłania wszystkich plików / suma rozmiarów wszystkich plików wysłanych

Dla celów badawczych stworzone zostały pliki z różnymi rozszerzeniami oraz rozmiarami mierzonymi w jednostkach informatycznych określające wymiar informacji danych. Typami i rozszerzeniami plików użytych podczas badania są:
- obraz (jpg);
- wideo (mp4);
- tekst (txt).

W przypadku rozmiarów plików stosowany jest rozmiar binarny, czyli np. 1 kilobajt (KB) równy jest 1024 bajtom (B) [21]. Rozmiary plików dla każdego z rozszerzeń:
- 10 KB;
- 100 KB;
- 10 MB;
- 100 MB.

Ze względów technicznych i konieczności przeprowadzenia badania na plikach naturalnych, to znaczy niewygenerowanych w sztuczny sposób (np. za pomocą poleceń wypełniających plik białymi znakami), rozmiary poszczególnych plików mają nieznaczne odchylenia.

4.1. Scenariusze badawcze

Każdy scenariusz obejmował przeprowadzenie transferu plików w liczbie zależnej od jego rozmiaru:
- 10 KB - 1000 razy;
- 100 KB - 1000 razy;
- 10 MB - 100 razy;
- 100 MB – 10 razy.

Wysyłanie plików

Badane parametry:
- średni czas wysyłania w przeliczeniu na 1 MB dla danego pliku (ms);
- średni czas wysyłania w przeliczeniu na 1 MB dla wszystkich plików (ms);
- średni czas opóźnienia podczas wysyłania danego pliku (ms);
- średni czas opóźnienia podczas wysyłania wszystkich plików (ms).

Scenariusz zrealizowany został dla:
- dwóch aplikacji mobilnych (aplikacja Java oraz C#);
- trzech typów plików (obraz, wideo oraz tekst);
- czterech rozmiarów plików (10 KB, 100 KB, 10 MB oraz 100 MB).

Pobieranie plików

Badane parametry:
- średni czas pobierania w przeliczeniu na 1 MB dla danego pliku (ms);
- średni czas pobierania w przeliczeniu na 1 MB dla wszystkich plików (ms);
- średni czas opóźnienia podczas pobierania danego pliku (ms);
- średni czas opóźnienia podczas pobierania wszystkich plików (ms).

Scenariusz zrealizowany został dla:
- dwóch aplikacji mobilnych (aplikacja Java oraz C#);
- trzech typów plików (obraz, wideo oraz tekst);
- czterech rozmiarów plików (10 KB, 100 KB, 10 MB oraz 100 MB).

5. Wyniki badań

5.1. Wysyłanie plików

Przeważająca liczba niższych wartości znajduje się w kolumnie oznaczonej językiem Java (tabela 2). Oznacza to, że w przypadku wysyłania plików aplikacja mobilna w języku Java odznacza się lepszą wydajnością. Również różnice procentowe wartości bardzo często przekraczają 10%. Można więc stwierdzić, że różnica w wydajności porównywanych aplikacji jest znacząca.

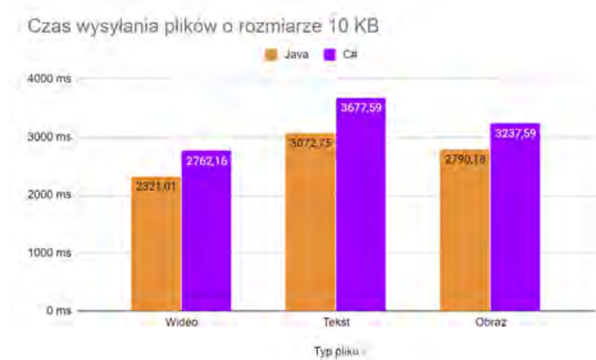
Tabela 2: Dane dotyczące wysyłania plików

	Wysyłanie				
	Rozmiar pliku	Typ pliku	Java	C#	Różnica procentowa
Czas przesyłania w przeliczeniu na 1 MB dla danego pliku [ms]	10 KB	Wideo	2321,01	2762,16	15,97%
		Tekst	3072,75	3677,59	16,45%
		Obraz	2790,18	3237,59	13,82%
	100 KB	Wideo	402,65	493,57	18,42%
		Tekst	524,18	549,69	4,64%
		Obraz	495,35	557,18	11,10%
	10 MB	Wideo	202,42	198,97	1,70%
		Tekst	196,86	204,97	3,96%
		Obraz	201,17	196,03	2,56%
	100 MB	Wideo	190,56	215,06	11,39%
		Tekst	187,7	205,78	8,79%
		Obraz	189,9	209,39	9,31%

Wykresy przedstawione na rysunkach od 1 do 4 pozwolą przeanalizować różnice pomiędzy typami plików dla danych rozmiarów. Tło komórek oznaczone kolorem zielonym oznacza wartość niższą, przy porównywaniu obu języków. Dodatkowo, w celu ułatwienia porównania wartości dodana została dodatkowa kolumna oznaczająca różnice procentową pomiędzy wartością maksymalną a minimalną.

Na rysunkach od 1 do 4 widać, że przypadku przesyłania pliku o najmniejszych rozmiarach, najwydajniejsze jest przesyłanie plików wideo. Elementy o rozmiarach 10 oraz 100 KB wykazują najniższe czasy przesyłania z wyraźnymi różnicami dla aplikacji napisanej w języku Java. Rozmiar plików rzędu 10 MB wykazuje znacznie niższe czasy przesyłania przemawiające za aplikacją C#. W tym też zakresie dominuje obraz jako plik z najbardziej korzystną wartością czasu transferu. Pliki o rozmiarze 10 MB są najbardziej wydajne w przeliczeniu na czas wymagany do przesłania 1 MB danych.

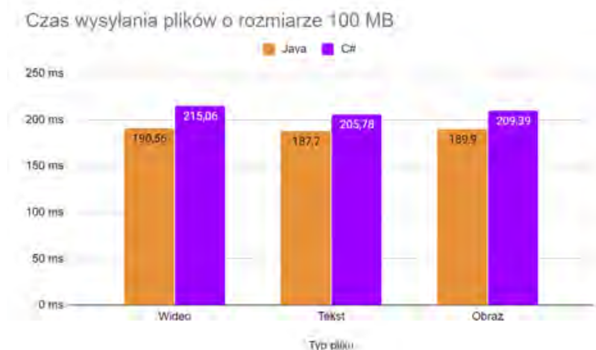
Jednym z wyznaczników rzetelności badania jest współczynnik korelacji Pearsona [22, 23]. Współczynniki korelacji bliższe liczbie 1 bądź -1 oznaczają korelację coraz silniejszą, zaś bliższe 0 oznaczają korelację coraz słabszą. Danymi, których korelację badano były wartości czasów przesyłania w przeliczeniu na 1 MB dla danego pliku w zestawieniu Java oraz C#. W przypadku wysyłania plików, współczynnik ten wynosi 0,9997224976, czyli jest bardzo bliski wartości jeden. Fakt ten oznacza, że wyniki badania nie zostały zaburzone w żaden sposób, nie występują również znaczące odchylenia pojedynczych wartości, co miałoby wpływ na analizę wyników.



Rysunek 1: Czas wysyłania plików o rozmiarze 10 KB

Rysunek 2: Czas wysyłania plików o rozmiarze 100 KB

Rysunek 3: Czas wysyłania plików o rozmiarze 10 MB



Rysunek 4: Czas wysyłania plików o rozmiarze 100 MB

5.2. Pobieranie plików

Zagregowane dane wskazują na zróżnicowanie wyników i wyraźny kontrast dla poszczególnych badanych rozmiarów (tabela 3). Zakres danych dla 10 oraz 100 KB wykazuje większą wydajność w przypadku aplikacji napisanej w języku C#. Pozostałe badane pliki, czyli te o rozmiarach 10 i 100 MB przemawiają natomiast za aplikacją C#, jednakże, z niewielkimi różnicami, rzędu około 6%.

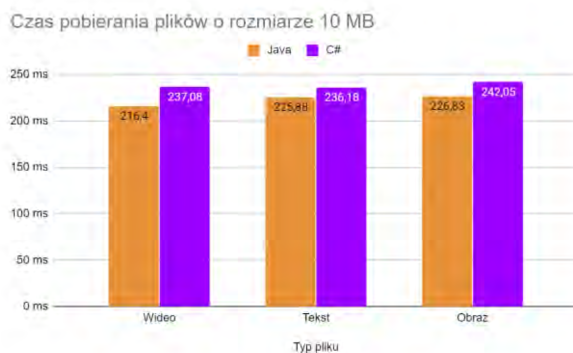
Tabela 3: Dane dotyczące pobierania plików

	Pobieranie				
	Rozmiar pliku	Typ pliku	Java	C#	Różnica procentowa
Czas przesyłania w przeliczeniu na 1 MB dla danego pliku [ms]	10 KB	Wideo	1264,76	1500,11	15,69%
		Tekst	1792,79	1504,17	16,10%
		Obraz	1566,71	1339,21	14,52%
	100 KB	Wideo	331,42	296,58	10,51%
		Tekst	323,19	314,8	2,60%
		Obraz	336,98	332,26	1,40%
	10 MB	Wideo	216,4	237,08	8,72%
		Tekst	225,88	236,18	4,36%
		Obraz	226,83	242,05	6,29%
	100 MB	Wideo	236,45	271,74	12,99%
		Tekst	234	218,65	6,56%
		Obraz	236	249,38	5,37%

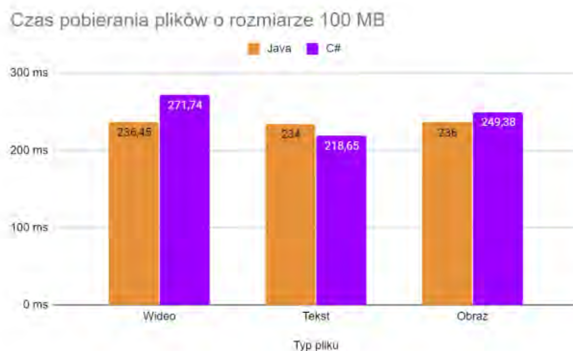
Wykresy na rysunkach od 5 do 8 ukazujące czas pobierania 1 MB danych w zależności od typu dla podanego rozmiaru pliku wykazują unormowane dane z odchyleniami w przypadku pliku tekstowego o rozmiarze 10 KB. Największą wydajność dla pobierania, w odróżnieniu od badania dotyczącego wysyłania plików, wykazują zarówno pliki o rozmiarze 10 MB jak i 100 MB. Niemniej jednak, różnice pomiędzy wartościami są niewielkie.

Rysunek 5: Czas pobierania plików o rozmiarze 10 KB

Rysunek 6: Czas pobierania plików o rozmiarze 100 KB



Rysunek 7: Czas pobierania plików o rozmiarze 10 MB



Rysunek 8: Czas pobierania plików o rozmiarze 100 MB

Wartość współczynnika korelacji Pearsona obliczona została na podstawie wyników z tabeli 3 dla języków Java oraz C#. Współczynnik ten dla pobierania plików, tak jak w przypadku wysyłania, jest bardzo bliski liczbie 1 i wynosi około 0,9774876905. Oznacza to silne powiązanie danych dla mierzonych wartości dzięki czemu badanie można uznać za rzetelne, bez występujących znaczących odchyleń dla pojedynczych wartości.

Tabela 4: Dane ogólne transferu plików dla badania nr 1

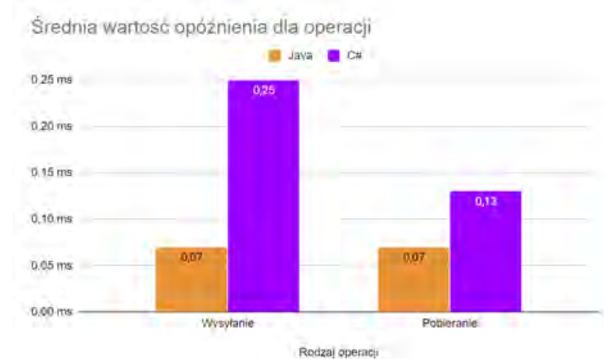
	Wartości średnie danych zebranych w badaniu			
	Rodzaj operacji	Java	C#	Różnica procentowa
Czas przesyłania w przeliczeniu na 1 MB dla danego pliku [ms]	Wysyłanie	220,29	241,56	8,81%
	Pobieranie	240,54	257,96	6,75%
Wartość opóźnienia [ms]	Wysyłanie	0,07	0,25	72,00%
	Pobieranie	0,07	0,13	46,15%

Dane przedstawione w tabeli 4 reprezentują dane uzyskane podczas przeprowadzania całego badania. Od razu zauważyć można dominację koloru zielonego w kolumnie oznaczonej językiem Java. Fakt ten oznacza wartości niższe, a więc bardziej korzystne dla każdego zagregowanego parametru. W przypadku czasu przesyłania plików w przeliczeniu na 1 MB, różnica wartości wynosi odpowiednio dla wysyłania oraz pobierania 8,81% i 6,75%.

Rysunek 9: Zebrany czas operacji transferu plików

Wbrew informacjom uzyskanym z wykresów i tabel w poprzednich punktach, ogólna wydajność, w przypadku wysyłania danych poprzez sieć przemawia za korzystaniem z aplikacji napisanej w języku Java (rysunek 9). Różnice wydajności pomiędzy operacjami wysyłania oraz pobierania są bardzo niewielkie (około 2%).

Z rysunku 10 wynika, że uśrednione wartości opóźnień dla operacji wykonanych na plikach jasno wskazują na komunikację pomiędzy aplikacją Java a aplikacją serwerową jako dużo bardziej wydajną. Współczynnik korelacji obliczony został na podstawie wartości średnich zebranych czasów przesyłania dla wszystkich badanych plików z wykorzystaniem aplikacji Java oraz C# i wynosi on 0,9998774792. Oznacza to silny związek powiązania wartością a co za tym idzie ukazuje to wiarygodność danych i nieznaczne odchylenia w przypadku pojedynczych wartości.



Rysunek 10: Czas opóźnień dla transferu plików

6. Wnioski

Teza zakłada większą wydajność czasową przesyłania danych aplikacji Java niż aplikacji C#. Przeprowadzone badania wykazały słuszność tezy. Przypomnieć należy, że badania wykonane zostały w warunkach typowego użytkowania aplikacji mobilnych, czyli połączenia poprzez sieć bezprzewodową. Wyniki przeprowadzonych badań wykazały około 7-8% przewagę w wydajności aplikacji Java nad aplikacją C#. W badaniach dodatkowo zostały zestawione różne typy oraz rozmiary plików. Badanie pierwsze wykazało największą wydajność przesyłania plików o rozmiarach 10 MB dla wysyłania oraz 10 MB i 100 MB dla pobierania. Jeżeli chodzi o typy plików, to trudno wyłonić taki, który wyróżniałby się większą wydajnością w porównaniu do pozostałych.

Na koniec należy wspomnieć, że pomimo starannych przygotowań środowisk testowych w sposób jednakowy i uniezależnienia ich od ewentualnych błędów pomiarowych to jednak pomiary obarczone są pewnymi drobnymi odchyleniami. Wprawdzie zbadane współczynniki korelacji nie wykazują różnic zależności, to jednak ograniczona zasobami sprzętowymi liczba próbek badawczych oraz wpływ niezależnych czynników na wykonywanie testów takich jak np. zmienne taktowania procesorów, wpływ temperatury na wydajność komponentów, specyfika sieci bezprzewodowej, czyli zdolność do tracenia pakietów i obniżenia prędkości połączeń, czy też zmienne obciążenie procesora funkcjami systemowymi. Pomimo powyższych trudności, których nie sposób zapobiec w warunkach domowych, zdaniem autora badanie zostało przeprowadzone w sposób należyty, rzetelny i z jak największą dokładnością.

7. Podsumowanie

Celem niniejszej pracy było porównanie w zakresie transferu plików dwóch analogicznych aplikacji w językach Java oraz C#. Wymagane było stworzenie aplikacji

serwerowej wraz z bazą danych oraz aplikacji mobilnych w dwóch językach programowania. Połączenie podobnych ze względu na obiektowość języków programowania umożliwiło poznanie oraz nauczenie się nowych rozwiązań programistycznych. Pozwoliło to także na podejście do tematu z dwóch różnych stron i ujawniło, że pomimo różnych środowisk programistycznych, wiele metod ma bardzo podobną składnię oraz zasadę działania. Jednakże, mimo tych podobieństw, otrzymane wyniki nieznacznie się różniły względem początkowych założeń.

Literatura

[1] Sieć 5G, https://www.plus.pl/news/art-8353-nadchodzi-kolejna-generacja-sieci-komorkowej-5g [05.06.2020].

[2] Java vs C#, https://www.educba.com/java-vs-c-sharp/ [05.06.2020].

[3] I. Alkadi, G. Alkadi, H. Etheridge, A Comparative Analysis Of C# And Java As An Introductory Programming Language For Information Systems Students, Review of Business Information Systems (RBIS) 10 (2011) 37-40.

[4] Porównanie składniowe Java oraz C#, https://www.softwaretestinghelp.com/csharp-vs-cpp-vs-java/ [05.06.2020].

[5] Porównanie składniowe Java oraz C#, https://www.guru99.com/java-vs-c-sharp-key-difference.html [05.06.2020].

[6] Porównanie wydajnościowe Java oraz C#, http://www.bentuser.com/article.aspx?ID=323&AspxAutoDetectCookieSupport=1 [05.06.2020].

[7] Porównanie wydajnościowe Java oraz C# https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharp.html [05.06.2020].

[8] Porównanie wydajnościowe Java oraz C# (artykuł), http://www.itu.dk/~sestoft/papers/numericperformance.pdf [05.06.2020].

[9] A. Stasiewicz, Android. Podstawy tworzenia aplikacji, Helion, 2013.

[10] A. Kempa, T. Staś, Wstęp do programowania w C#: Łatwy podręcznik dla początkujących, Tomasz Staś, 2014.

[11] M Reynolds., Xamarin Mobile Application Development for Android, Packt Publishing Ltd, 2014.

[12] I. F. Darwin, Android. Receptury, Helion, 2013.

[13] Zastosowanie języka Java, https://jaki-jezyk-programowania.pl/technologie/java/ [05.06.2020].

[14] Ranking IDE, https://pypl.github.io/IDE.html [05.06.2020].

[15] Android Studio, https://developer.android.com/studio [05.06.2020].

[16] Definicja Spring, https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html [05.06.2020].

[17] Zastosowanie C#, https://testuj.pl/blog/10-pytan-programisty-temat-jezyka-csharp/ [05.06.2020].

[18] F. Friesen, Learn Java for Android Development, Apress, 2013.

[19] B. Burke, RESTful Java with JAX-RS, "O'Reilly Media, Inc.", 2010.

[20] F. Gutierrez, Wprowadzenie do Spring Framework dla programistów Java, Helion, 2015.

[21] Jednostka informacji, https://pl.wikipedia.org/wiki/Jednostka_informacji [05.06.2020].

[22] Współczynnik korelacji Pearsona, https://pogotowiestatystyczne.pl/slowniczek/korelacja-pearsona/ [05.06.2020].

[23] Użyteczność korelacji Pearsona w badaniach, http://zsi.tech.us.edu.pl/~nowak/smad/SMAD_korelacje.pdf [05.06.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 300–304
Received: 18 June 2020
Accepted: 6 July 2020

Performance comparison of chosen JSON parsers and a parser that employs a different reading method

Porównanie wydajności wybranych parserów JSON z parserem używającym innej metody odczytu

Przemysław Grzegorz Koter*

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

Abstract

The analysis of parsing method of currently used JSON parsers, that are comprised of tokenizer and parser module, led to conclusion, that their performance could be improved. As a means to prove it, new JSON parser has been developed, whose modules are dedicated to process structure of either an object or an array. In both modules, tokenization step is combined with building step of document model, representing its structure. Performance test involved processing of sample document 1000 times, per 20 repeats, and was carried out by three compared parsers. Proposed parser was nearly 89% and 42% faster than other two, and its memory usage was mediocre. Results are promising enough to consider real-world usage, thus improving the efficiency of JSON processing.

Keywords: JSON parser; performance

Streszczenie

Analiza metody przetwarzania obecnie używanych parserów JSON, które są złożone z modułu tokenizera i parsera, doprowadziła do wniosku, że ich wydajność może być poprawiona. W celu dowiedzenia tego, opracowano parser, którego moduły są podzielonego względem przetwarzanej struktury, czyli obiektu i tablicy. W obu modułach krok wyodrębniania tokenów dokumentu jest połączony z krokiem budowy reprezentacji struktury dokumentu. Test wydajności obejmował przetworzenie przykładowego dokument 1000 razy w każdym z 20 powtórzeń, przez trzy porównywane parsery. Proponowany parser był szybszy o około 89% i 42% od pozostałych dwóch, jednocześnie zużycie pamięci było przeciętne. Wyniki są na tyle obiecujące, by rozważyć faktyczne użycie, poprawiając efektywność przetwarzania dokumentów JSON.

Słowa kluczowe: JSON; parser; wydajność

*Corresponding author

Email address: **przemyslaw.koter@pollub.edu.pl** (P. G. Koter)

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

1. Wstęp

Niniejszy artykuł traktuje o nowym podejściu w przetwarzaniu danych w formacie JSON [1], opartym na podziale parsera na moduły, rozpoznające i specjalizujące się w odczycie tylko obiektu lub tablicy. Według założeń taka budowa parsera i wybrany sposób odzwierciedlenia pozyskanych danych w modelu hierarchicznym ma uprościć wykonanie procesu odczytu, a co za tym idzie, poprawić jego wydajność. Realizacja założeń algorytmu jest zapewne możliwa w wielu językach programowania obiektowego, jednakże na potrzeby zbadania wydajności, w porównaniu do parserów opartych na innych założeniach, wybrano język C# [2].

2. Przegląd sposobu działania istniejących parserów

Temat budowy wydajności przetwarzania plików JSON został poruszony parokrotnie, jednakże nie w kontekście parserów ogólnego zastosowania. Analizowane publikacje skupiają się raczej na wydajności konkretnych czynności, jak poszukiwanie słów kluczowych [3] lub wyszukiwanie analityczne [4]. W tej sytuacji można mówić o proponowaniu nowej metody przetwarzania, różniącej się od istniejących implementacji.

Parsery realizujące konkurencyjną metodę przetwarzania zostały zaimplementowane w bibliotekach Newtonsoft.Json i System.Text.Json. Pierwsza z bibliotek została wybrana ze względu na jej największą popularność [5], druga zaś jako oficjalna implementacja metod przetwarzania dokumentów JSON z biblioteki standardowej platformy .NET Core [6]. Na poziomie konceptualnym rozwiązania implementowane przez oba parsery są bardzo podobne. Parser tego rodzaju wprowadza przetwarzanie w dwóch modułach. Pierwszy z nich zajmuje się wyodrębnieniem symboli strukturalnych formatu i pojedynczych wartości, takich jak ciąg tekstowy, liczba lub literał, przy każdym wywołaniu. Zapisuje też stan przetwarzania, czyli, między innymi, aktualną pozycję odczytu w dokumencie i stos zawierający informacje o typach struktur formujących obecny poziom zagnieżdżenia. Przy napotkaniu końca struktury (obiektu lub tablicy) na poziomie n hierarchii, moduł musi przywrócić informacje o typie struktury rodzica z poziomu $n-1$. Drugi z modułów tworzy model hierarchiczny, reprezentujący budowę odczytywanego dokumentu. Informacje o typie odczytywanej wartości i struktury pozyskuje od pierwszego modułu. Przełączenie pomiędzy różnymi typami odczytywanej struktury

realizuje w postaci umówionych kodów stanu. Funkcja odczytująca pierwszego modułu jest wywoływana w pętli modułu drugiego, dopóki nie zakończono odczytu dokumentu. Moduł drugi jest z reguły nieświadomy, jakiego rodzaju wartość zostanie zwrócona przy następnym wywołaniu.

Różnica między parserami przejawia się w typie struktur służących przechowaniu odczytanych danych. Pierwszy z nich zapisuje wartości z obiektu i tablicy JSON odpowiednio jako obiekty słownika i listy, które rozszerzają się w miarę dodawania wartości. Drugi przechowuje wszystkie wartości w tablicy bajtów, utrzymując specyficzną strukturę i zasady zapisu. Tablica jest alokowana adekwatnie do rozmiaru całego dokumentu.

3. Omówienie proponowanej metody przetwarzania

Proponowana zmiana opiera się na wprowadzeniu dwóch modułów, przeznaczonych do budowania reprezentacji obiektu i tablicy. Dzięki tej zmianie liczba potencjalnych kombinacji symboli jest ograniczona i przewidywalna, a co za tym idzie, moduł taki może bezpośrednio odczytywać znaki z dokumentu. Funkcja przetwarzania może być wielokrotnie wywoływana przez samą siebie w przypadku odczytu zagnieżdżonych struktur, zaś kończy wykonanie przy napotkaniu końca struktury, zwracając aktualną pozycję w dokumencie. W efekcie stan odczytu istnieje tylko na stosie, powielony n razy, w miejscach wywołań procedury, gdzie n to poziom zagnieżdżenia aktualnej struktury. Stan jest efektywnie reprezentowany przez pozycję w dokumencie, gdyż typ struktury jest zaszyty w kodzie wykonywanej procedury.

W kwestii reprezentacji danych dokumentu zdecydowano się na wybór listy jednokierunkowej, jako struktury stosowanej do reprezentacji zarówno obiektu i tablicy JSON. Zaletą takiego podejścia jest zredukowana liczba realokacji. W celu jednoczesnego przyśpieszenia późniejszych wyszukiwań wartości z obiektu JSON zastosowano grupowanie po typie. Wszystkie wartości nadal są dodawane do jednej listy, jednakże wstawienie wartości może zajść w jednej z trzech referencji, odpowiednio dla obiektów, tablic i wartości prymitywnych (literałów null, true i false, wartości liczbowych i ciągów tekstowych). Obiekt reprezentujący tablicę tworzy dwie listy – listę zawierającą obiekty i tablice, i drugą, zawierającą wartości prymitywne.

W samych metodach budowy modelu pola będące końcami listy jednokierunkowej nie są weryfikowane pod kątem przechowywania wartości null (brak obiektu). To uproszczenie wymaga jednak, by w konstruktorze klasy reprezentującej obiektu lub tablicę JSON umieścić kod tworzący obiekty wartości i zapisujący je pod te pola. Obiekty te nie otrzymają nigdy faktycznej wartości pochodzącej z dokumentu, dlatego na koniec przetwarzania muszą zostać usunięte z list.

Wartości były początkowo zawsze pozyskiwane jako łańcuch tekstowy, jednak takie podejście pokazało poważne braki wydajnościowe i zwiększone użycie pamięci. W ramach poprawy zmieniono sposób odczytu wartości liczbowych, by były od razu konwertowane na odpowiadające im typy long (liczba całkowita) lub double (liczba zmiennoprzecinkowa).

Podczas implementacji metod przetwarzania przyjęto strategię optymalizacji odczytu dla dokumentów zminimalizowanych, czyli pozbawionych wszystkich znaków białych. Jeśli odczytany znak c jest znakiem niedrukowalnym (białym), wówczas trzeba wywołać metodę która odszuka pierwszy drukowalny znak. Jeśli jednak znak c jest znakiem drukowalnym, to oszczędność polega na pominięciu wywołania metody i wykonania minimum 1 sprawdzenia wartości znaku. Sprawdzenie, czy znak jest drukowalny, może być przeprowadzone poprzez porównanie wartości liczbowej znaku c do spacji, co wynika z ułożenia tablicy znaków ASCII [7] (Listing 1).

Listing 1: Kod metody przetwarzającej obiekt JSON – pierwsza i ostatnia instrukcja if odpowiada za uproszczenie znajdywania początku klucza i wartości w obiekcie, jeśli nie ma białych znaków. Metoda FindValue znajduje pozycję w dokumencie za znakiem dwukropka (separatora między kluczem i wartością).

```
internal override int ParseAll(string json, int start)
{
    int c;
    JsonToken token;
    next_token:
    if ((c = json[++start]) == '"')
        goto parse_name;

    start = FindProperty(json, start, ref c);
    if (c == '}')
        goto ret;

    parse_name:
    int index = start + 1;
    start     = FindValue(json, index, out int nameLength);
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);
```

4. Metoda przeprowadzenia testów wydajności

Poprawnie przyprowadzone testy wydajności, pozwalające uzyskać rzetelne wyniki, są niezbędne do oceny przydatności zastosowanego rozwiązania. Proces odczytu przykładowego dokumentu może trwać bardzo krótko (poniżej 1 milisekundy), co czyni trudnym dokładne zbadanie różnic w czasach wykonania. Rozwiązaniem tego problemu jest powtórzenie kroku przetwarzania wielokrotnie. W przeprowadzonym teście liczba powtórzeń n wynosi 1000. Zmierzony czas jest łącznym czasem potrzebnym do wykonania odczytu n razy.

Wyniki nie zostały przeliczone na czas jednostkowy, czyli czas wykonania pojedynczego odczytu, ponieważ bardziej interesująca jest relacja wyników pomiędzy różnymi parserami. Drugim zagadnieniem jest liczba przeprowadzonych prób. Nie może być zbyt mała, gdyż nie ma wtedy możliwości stwierdzenia, czy otrzymane wyniki nie są skrajne.

Listing 2: Kod metody przetwarzającej obiekt JSON – instrukcje w blokach case służą utworzeniu obiektów modelu i ustawieniu odpowiednich referencji listy. Metoda LoadBools wczytuje literały true, false i null. Metoda LoadNumber tworzy obiekt klasy dziedziczący po JsonToken, który posiada dodatkowe pole z wartością typu long lub double.

```
switch (c)
{
    case '{':
        token = new JsonObject((JsonToken)this);
        token._next = _lastObject._next;
        _lastObject._next = token;
        _lastObject = token;
        parse_composite:
        start = token.ParseAll(json, start);
        break;
    case '[':
        token  = new JsonArray((JsonToken)this);
        _lastArray._next = token;
        _lastArray = token;
        goto parse_composite;
    default:
        if (c == '"')
        {
            token = new JsonToken();
            start = token.LoadString(json, start);
        }
        else if (c > '9')
        {
            token = new JsonToken();
            start = token.LoadBools(json, start, c);
        }
        else
        {
            start = LoadNumber(json, start, c);
            token = _last;
            break;
        }

        AddValue(token);
        break;
}
```

Listing 3: Kod metody przetwarzającej obiekt JSON – wszystkim wartościom ustawia się klucz w polu name. Pierwsza instrukcja if odpowiada za pominięcie wywołania metody poszukiwawczej, jeśli po wartości nie ma znaku białego. Kolejny if przenosi kontrolę na początek metody, w celu przetworzenia kolejnej wartości. Na końcu następuje usunięcie pustych obiektów wartości.

```
    token._name = json.Substring(index, nameLength);
    _count++;
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    if (c == ',')
        goto next_token;

    if (c != '}')
        throw new InvalidJsonException();

    ret:
    TrimEmpty();
    return start;
}
```

W teście przyjęto liczbę prób p równą 20. Podczas pojedynczej próby parsery wykonują po kolei n odczytów. Przetwarzany dokument jest zachowany w pamięci operacyjnej jako łańcuch znaków, co eliminuje opóźnienia związane z odczytem pliku z dysku. Cały test został przeprowadzony trzy razy, co pozwoliło upewnić się, że wyniki są powtarzalne. W przypadku pomiarów użycia pamięci operacyjnej można mówić o wartościach przybliżonych. Wynika to z faktu, że wykorzystane API zwraca tylko „najlepsze możliwe przybliżenie" zaalokowanych bajtów w zarządzanej pamięci [8]. Dzięki zastosowaniu odpowiednio dużego dokumentu całkowite użycie pamięci jest wielokrotnie większe niż błąd pomiaru. Nadto wyniki te są traktowane jako wartości referencyjne, gdyż główne kryterium użyteczności to czas wykonania.

Listing 4: Kod metody przetwarzającej tablicę JSON

```
internal override int ParseAll(string json, int start)
{
    JsonComposite token;
    int          c;
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    if (c == ']')
        goto ret;

    goto value_parse;

    next_value:
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    value_parse:
    switch (c)
    {
        case '{':
            token = new JsonObject((JsonToken)this);
            _flags |= (int)JSymbol.OBJECT;
            parse_composite:
            _lastObject._next = token;
            _lastObject = token;
            start = token.ParseAll(json, start);
            token._index = _count;
            break;
        case '[':
            token = new JsonArray((JsonToken)this);
            _flags |= (int)JSymbol.ARRAY;
            goto parse_composite;
        default:
            _valueCount++;
            JsonValue value;
            if (c == '"')
            {
                value = new JsonValue();
                start = value.LoadString(json, start);
            }
            else if (c > '9')
            {
                value = new JsonValue();
                start = value.LoadBools(json, start, c);
            }
            else
            {
                start = LoadNumber(json, start, c);
                break;
            }

            AddValue(value);
            break;
    }

    _count++;
    if ((c = json[++start]) <= ' ')
        start = FindFirstAnything(json, start, out c);

    if (c == ',')
        goto next_value;

    if (c != ']')
        throw new InvalidJsonException();

    ret:
    TrimEmpty();
    return start;
}
```

4.1. Charakterystyka testu

Na potrzeby zbadania i porównania wydajności parserów opracowano test, w którym każda z bibliotek wykonuje odczyt i buduje reprezentację dokumentu, zawartego w obiekcie łańcucha znaków (klasy string). Dokument składa się z tablicy zawierającej 100 identycznych obiektów. Struktura pojedynczego obiektu została przedstawiona na listingu 5. Dane zawarte w dokumencie nie mają znaczenia dla przeprowadzanego testu.

Listing 5: Struktura obiektu JSON, składającego się na dokument testowy.

```
{"nullvalue":null,"falseValue":false,"truevalue":true,
"stringvalue1":"30","stringvalue2":"40",
"stringvalue3":"50","stringvalue4":"60",
"stringvalue5":"70","stringvalue6":"80",
"stringvalue7":"90","stringvalue8":"100",
"stringvalue9":"110","stringvalue10":"120",
"stringvalue11":"130","stringvalue12":"140",
"stringvalue13":"150","stringvalue14":"160",
"stringvalue15":"170","stringvalue16":"180",
"stringvalue17":"190","stringvalue18":"200",
"stringvalue19":"210","stringvalue20":"220",
"stringvalue21":"230","stringvalue22":"240",
"stringvalue23":"250","stringvalue24":"260",
"stringvalue25":"270","stringvalue26":"280",
"stringvalue27":"290","stringvalue28":"300",
"stringvalue29":"310","stringvalue30":"320","iuwwbhh1.kex":
[489,879,1269,1659,2049,2439,2829,3219,3609,3999,4389,4779,
5169,5559,5949,6339,6729,7119,7509,7899,8289,8679,9069,
9459,9849,10239,10629,11019,11409,11799,12189,12579,12969,
13359,13749,14139,14529,14919,15309,15699,16089,16479,
16869,17259,17649,18039,18429,18819,19209,19599]}
```

Tabela 1: Wartości pomiarów czasu realizacji zadania testowego.

Numer próby	Newtonsoft.Json (ms)	System.Text.Json (ms)	Opisany parser (ms)
1.	3723	723	411
2.	3574	659	374
3.	3583	657	387
4.	3543	665	385
5.	3567	658	379
6.	3593	660	383
7.	3558	664	393
8.	3559	659	382
9.	3570	657	384
10.	3564	665	382
11.	3558	657	378
12.	3578	657	378
13.	3537	657	379
14.	3544	656	383
15.	3541	656	380
16.	3542	657	380
17.	3540	662	376
18.	3577	659	385
19.	3548	657	380
20.	3552	659	380

Tabela 2: Średnia pomiarów czasu dla 20 prób.

	Newtonsoft.Json (ms)	System.Text.Json (ms)	Opisany parser (ms)
Średnia	3567,55	662,2	382,95

Tabela 3: Wartości pomiarów użycia pamięci podczas realizacji zadania testowego. We wszystkich próbach uzyskano jednakowe wartości.

Próby	Newtonsoft.Json (B)	System.Text.Json (B)	Opisany parser (B)
1.-20.	2105344	401456	737280

5. Wnioski

Opisany parser w przeprowadzonym zadaniu testowym osiągnął średni wynik około 42% lepszy niż System.Text.Json, a także 89% szybszy niż Newtonsoft.Json. Niska wydajność biblioteki Newtonsoft.Json była przyczyną prac nad nową biblioteką standardową do obsługi dokumentów JSON, a także nad omawianym parserem. Użycie pamięci przez proponowany parser uplasowało się pomiędzy wynikami obu bibliotek, a uzyskana wartość jest bliższa niższej. Na podstawie pozyskanych rezultatów można wysunąć wniosek, że parser o zaproponowanej budowie lepiej radzi sobie z zadaniami, w których wynikiem jest reprezentacja danych dokumentu. Niekoniecznie musi mieć to przełożenie przy np. wyszukiwaniu informacji w dokumencie. Do przeprowadzenia mapowania danych z dokumentu na podany typ (w procesie zwanym deserializacją) parser musiałby zostać zmodyfikowany, poprzez zamianę kodu odpowiedzialnego za budowę obiektów hierarchii, na kod tworzący obiekty danych typów i konwertujący wartości tekstowe na wartości typów prymitywnych. By w pełni ocenić charakterystykę wydajności takiego sposobu przetwarzania, trzeba wykonać wiele testów, z różnymi danymi testowymi. Istnieje bowiem szansa, że wykonywanie wielu małych alokacji będzie sprawdzało się gorzej, gdy rozmiar dokumentu znacznie wzrośnie (np. stukrotnie).

Na ten moment pozostaje jedynie możliwość spekulacji wpływu niektórych decyzji. Prawdopodobnie usunięcie dodatkowych referencji do fragmentów listy wartości, w klasie reprezentującej obiekt JSON, pozwoliłoby poprawić ten wynik dzięki uproszczeniu procesu budowania listy. Mając tylko jedną referencję można też zrezygnować z wstawiania obiektów pustych wartości, co przełoży się na zmniejszenie użycia pamięci.

Literatura

[1] T. Bray i Ed., „The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 8259, 2017.

[2] Standard ECMA-334, C# Language Specification, https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf [06.06.2020].

[3] E. Wahyudi, S. Sfenrianto, M. J. Hakim, R. O. Subandi, R. Sulaeman i R. Setiyawan, „Information Retrieval System for Searching JSON Files with Vector Space Model Method," w 2019 International Conference of Artificial Intelligence and Information Technology (ICAIIT), Yogyakarta, Indonesia, Indonesia, 2019.

[4] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein i D. Kossmann, „Mison: a fast JSON parser for data analytics," Proceedings of the VLDB Endowment, 2017.

[5] Strona repozytorium bibliotek Nuget.org, lista popularnych bibliotek, Microsoft, https://www.nuget.org/packages [06.06.2020].

[6] Strona blogu z nowościami technicznymi, „Try new System.Text.Json APIs", Microsoft, https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-apis/ [06.06.2020].

[7] ANSI, ISO-IR-006: ASCII Graphic character set, https://www.itscj.ipsj.or.jp/iso-ir/006.pdf (1975-12-01), [06.06.2020].

[8] Strona dokumentacji metody użytej przy pomiarach użycia pamięci, https://docs.microsoft.com/en-us/dotnet/api/system.gc.gettotalmemory [06.06.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 305–308
Received: 30 June 2020
Accepted: 27 August 2020

Comparison of Objective-C and Swift on the example of a mobile game

Porównanie Objective-C oraz Swift na przykładzie gry mobilnej

Karolina Sylwia Banach*, Maria Skublewska-Paszkowska

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

Abstract

Mobile applications for the iOS platform can be developped using the Swift and Objective-C languages. The article presents a comparison between these languages based on a created mobile game. The structure and performance of these technologies were examined. Based on three devices, languages have been tested. Aspects such as RAM load, time between views, time to save data to the database and time to save data to file were tested as a part of the analysis. Two research hypotheses have been put forward: "Swift has a better performance than Objective-C" and "Swift has a simpler structure than Objective-C". The results obtained confirm that Swift is more efficient than Objective-C. Research into the structure of codes has proven that the newer language has a simpler structure than its predecessor.

Keywords: Swift; Objective-C; performance; structure

Streszczenie

Aplikacje mobilne na platformę iOS można wytwarzać z użyciem języków Swift oraz Objective-C. Tematyką artykułu jest porównanie tych języków na przykładzie utworzonej gry mobilnej. Zbadana została struktura i wydajność omawianych technologii. Na przykładzie trzech iPhone'ów, języki zostały poddane testom. W ramach przeprowadzonej analizy wydajnościowej zostały przebadane takie aspekty jak: obciążenie pamięci RAM, czas przejścia pomiędzy widokami, czas zapisu danych do bazy oraz czas zapisu danych do pliku. Zostały postawione dwie hipotezy badawcze: "Język Swift jest wydajniejszy niż język Objective-C" oraz "Język Swift posiada prostszą strukturę niż język Objective-C". Otrzymane wyniki potwierdzają, że Swift jest wydajniejszy niż Objective-C. Dzięki badaniom struktury kodów udowodniono, że nowszy język posiada prostszą strukturę niż jego poprzednik.

Słowa kluczowe: Swift; Objective-C; wydajność; struktura

*Corresponding author

Email address: Karolina.sylwia.banach@gmail.com

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

1. Wstęp

W dzisiejszych czasach większość społeczeństwa nie wyobraża sobie funkcjonowania bez smartfonów. Telefony używane są nie tylko w celach komunikacji, ale również do rozrywki, poszerzania wiedzy, a nawet w celach finansowych. Dzięki tym urządzeniom można sprawdzić rozkład jazdy komunikacji, stan swojego konta bankowego, dokonać płatności w sklepach internetowych oraz stacjonarnych lub skontaktować się ze znajomymi poprzez media społecznościowe. Technologie równie szybko muszą się rozwijać i dostarczać coraz więcej rozwiązań, aby nadążyć za postępem, dlatego z każdym rokiem odnawiane są stare języki programowania lub powstają nowe.

W czerwcu 2007 roku została wydana pierwsza wersja systemu iOS, przyjmujący wtedy nazwę iPhone OS. Do czerwca 2014 roku, język Objective-C był jedynym językiem wykorzystywanym do pisania aplikacji mobilnych przeznaczonych dla systemu iOS [1]. Głównym powodem twórców było stworzenie nowego języka programowania, który miał być zastępcą języków C, a zwłaszcza Objective-C. Nowa technologia miała być równie dobrze wydajna co jej poprzednik, ale miała być bardziej bezpieczna. Sam kod miał być bardziej współczesny. Celem jego było przyciągnięcie nowych programistów, którzy byli przyzwyczajeni do nowych technologii takich jak Java, C#, czy JavaScript [2].

Niniejszy artykuł ma na celu porównanie dwóch języków programowania: Swift i Objective-C na przykładzie gry mobilnej. W pierwszej kolejności porównano wydajność stworzonych aplikacji.Dokonano analizy następujących parametrów: obciążenie pamięci RAM, czas przejścia pomiędzy widokami, czas zapisu danych do bazy oraz czas zapisu danych do pliku. W dalszej części zbadano również struktury kodów napisanych w obu językach.

Na podstawie omawianego zagadnienia postawiono dwie hipotezy badawcze: "Język Swift jest wydajniejszy niż język Objective-C" oraz "Język Swift posiada prostszą składnię niż język Objective-C". Zostały one udowodnione na podstawie przeprowadzonych badań.

Istnieje niewiele publikacji naukowych dotyczących porównań języka Objective-C z językiem Swift. Większość z nich skupia się jedynie na porównaniu ich pod względem struktury, jak np. pozycja [3], w której autorzy udowadniają, że składnia Swift jest krótsza, prostsza oraz bardziej przypominająca języki współczesne. Kolejnym przykładem, jest pozycja [4], w której autorzy wyraźnie podkreślają różnice i podobieństwa w strukturze obydwu technologii. Istnieją dokumentacje badające same bezpieczeństwo języków [5]. W artykule [6], porównana jest wydajność Objective-C oraz Swift,

pod względem szybkości wykonywania algorytmów sortowania oraz struktur danych. Po analizie prac literackich, nie spotkano się z analizą, która poruszana jest w niniejszym artykule.

2. Metodyka badań

W celu przeprowadzenia badań zarówno wydajnościowych, jak i strukturalnych, napisano specjalne aplikacje mobilne, które działają identycznie w językach Swift oraz Objective-C. Gra ping-pong polega na odbijaniu piłeczki paletką poprzez ruch dotykowy na ekranie iPhone. Rozgrywka toczy się między graczem a iPhone.

Do utworzenia gry mobilnej ping-pong, potrzebnej do przeprowadzenia badań wykorzystano następujące narzędzia i technologię:

- środowisko programistyczne X-code 11.5 [7];
- system iOS 13.3;
- język programowania Swift 5.0;
- język programowania Objective-C 2.1;
- framework SpriteKit [8];
- system zarządzania bazą danych MySQL;
- format JSON;
- język programowania PHP 7.1.

Obciążenie pamięci RAM zostało zbadane podczas przeprowadzania rozgrywki gracza. Czas przejścia pomiędzy widokami, był pobrany podczas przejścia z widoku rejestracji do widoku logowania. Czas zapisu danych do zewnętrznej bazy został pobrany podczas zapisu wyniku rozgrywki. Ostatnia analiza została przebadana w momencie zapisu danych do pliku.

Badania wydajności przeprowadzono na trzech urządzeniach z systemem iOS:

- iPhone SE 2nd generation;
- iPhone X;
- iPhone 8.

3. Badania wydajności

Na podstawie wyników testów wyznaczono wartość minimalną, wartość maksymalną, średnią i medianę. Każda operacja została wykonana po 20 razy, na każdym z urządzeń oraz technologii.

3.1. Obciążenie pamięci RAM

W tabeli 1 zaprezentowano, jak przedstawia się obciążenie pamięci RAM na badanych smartfonach, uwzględniając omawiane technologie.

Tabela 1: Uśrednione obciążenie pamięci RAM

Technologia	Urządzenie	Wartość minimalna [MB]	Wartość maksymalna [MB]	Średnia [MB]	Mediana [MB]
Swift	iPhone SE 2nd generation	108,05	142,77	125,91	127,36
	iPhone X	115,07	132,04	118,59	115,74
	iPhone 8	121,43	124,74	124,32	124,63
Objective-C	iPhone SE 2nd generation	98,61	106,64	105,53	106,37
	iPhone X	112,00	153,47	116,80	115,05
	iPhone 8	134,78	135,53	135,39	135,42

Na podstawie średniej i mediany można zauważyć, że obciążenie RAM nie zależy jedynie od danego języka. Widać, że im nowsze urządzenie tym Objective-C lepiej sobie radzi niż język Swift. W przypadku iPhone'a X obie technologie zużywają podobną ilość pamięci RAM.

3.2. Czas przejścia pomiędzy widokami

W tabeli 2 przedstawiono uśrednione wyniki czasu przejścia pomiędzy widokami uwzględniając badane smartfony oraz języki programowania.

Tabela 2: Uśredniony czas przejścia pomiędzy widokami

Technologia	Urządzenie	Wartość minimalna [ms]	Wartość maksymalna [ms]	Średnia [ms]	Mediana [ms]
Swift	iPhone SE 2nd generation	4,8150	6,4710	5,9566	6,0075
	iPhone X	2,5610	6,8600	5,4074	5,4790
	iPhone 8	3,0310	6,8340	5,3835	5,4851
Objective-C	iPhone SE 2nd generation	4,2089	6,8970	5,9865	6,2345
	iPhone X	4,6780	8,6850	5,9301	5,9845
	iPhone 8	2,7441	7,6700	5,9412	6,0365

Na podstawie danych z 2 tabeli można stwierdzić, że aplikacja napisana w języku Swift znacznie szybciej przechodzi między widokami aplikacji. Wartości w urządzeniu iPhone SE, są do siebie o wiele bardziej zbliżone, niż w pozostałych smartfonach.

3.3. Czas zapisu danych do zewnętrznej bazy danych

W tabeli 3 przedstawiono wyniki statystyczne czasu zapisu danych do zewnętrznej bazy na badanych urządzeniach w językach Objective-C oraz Swift. Do łączności z bazą MySQL aplikacja wykorzystuje format JSON oraz pliki php.

Wyniki badań przeprowadzonych na iPhone'ach dowodzą, że Objective-C w szybszy sposób zapisuje dane do zewnętrznej bazy danych. Urządzenie nie ma wpływu na wyniki testu, ponieważ średnia wartość danej technologii, dla badanych modeli iPhoneó'w jest do siebie bardzo podobna.

Tabela 3: Uśredniony czas zapisu do zewnętrznej bazy danych

Technologia	Urządzenie	Wartość minimalna [ms]	Wartość maksymalna [ms]	Średnia [ms]	Mediana [ms]
Swift	iPhone SE 2nd generation	0,5180	0,6050	0,5441	0,5420
	iPhone X	0,4660	0,7080	0,5789	0,5710
	iPhone 8	0,4209	0,6930	0,5507	0,5400
Objective-C	iPhone SE 2nd generation	0,2650	0,5200	0,3143	0,3061
	iPhone X	0,2450	0,3549	0,3046	0,3026
	iPhone 8	0,1940	0,4870	0,3530	0,3515

3.4. Czas zapisu danych do pliku

W tabeli 4 przedstawiono uśrednione wyniki czasu zapisu danych do pliku uwzględniając badane iPhone oraz języki programowania.

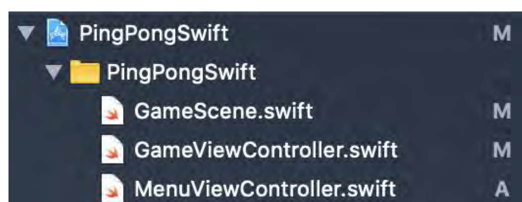
Tabela 4: Uśredniony czas zapisu danych do pliku

Technologia	Urządzenie	Wartość minimalna [ms]	Wartość maksymalna [ms]	Średnia [ms]	Mediana [ms]
Swift	iPhone SE 2nd generation	0,7019	1,5550	0,9293	0,8635
	iPhone X	0,6330	1,5190	0,8640	0,7970
	iPhone 8	0,4190	1,0331	0,8183	0,8135
Objective-C	iPhone SE 2nd generation	0,7960	1,7580	1,3254	1,3396
	iPhone X	0,6371	1,6630	1,2320	1,3075
	iPhone 8	0,8920	1,8040	1,2518	1,2925

Na podstawie średniej i mediany można zauważyć, że czas zapisu danych do pliku w języku Swift jest znacznie krótszy, niż w języku Objective-C. Wartości minimalne i maksymalne w obu technologiach są bardziej zbliżone do siebie, niż wartości średnie oraz mediany.

4. Analiza struktury

Nowa składnia języka Swift pozwala pisać aplikacje w zdecydowanie mniejszej liczbie plików, co można zauważyć na przykładach 1 oraz 2. Klasa języka Swift składa się z jednego pliku z rozszerzeniem ".swift" (rys 1) [9]. W przypadku Objective-C struktura jest 2 razy dłuższa, ponieważ każda klasa jest zbudowana z dwóch plików (przykład 2). Jeden z nich posiada rozszerzenie ".h"i jest on interfejsem klasy, a drugi z rozszerzeniem ".m" jest częścią implementacyjną [10].



Rysunek 1. Fragment struktury gry ping-pong w języku Swift.

Rysunek 2. Fragment struktury gry ping-pong w języku Objective-C.

Składnia omawianych języków programowania jest bardzo podobna do siebie, natomiast różnice występują w strukturze kodu. Język Swift ma zdecydowanie prostszą i bezpieczniejszą strukturę, co dowodzi brak konieczności stawiania średników na końcu linijki. Dodatkowo same odwołanie się do metod odbywa się podobnie, jak we współczesnych językach programowania. Przykładem tego jest obiekt touches, prezentowany w metodzie touchesBegan (listing 1). W przypadku technologii Swift wywołanie metody odbywa się poprzez dodanie do obiektu kropki a następnie nazwy metody, co widać na listingu 1. W Objective-C ta sama czynność odbywa się poprzez dodanie nawiasów kwadratowych przed obiektem, następnie oddzielenie spacją obiektu od metody wywołującej zamknięcie nawiasu oraz postawienie średnika po zakończonej czynności (Listing 2).

Listing 1. Metoda touchesBegan w języku Swift

```
override func touchesBegan(_ touches: Set<UITouch>,
    with event: UIEvent?) {

    for touch in touches{
        let location = touch.location(in: self)
        rocketPlayer.run(SKAction.moveTo(x:
            location.x, duration: 0.1))
    }
}
```

Listing 2. Metoda touchesBegan w języku Objective-C

```
- (void)touchesBegan:(NSSet *)touches
    withEvent:(UIEvent *)event {
    for (UITouch *touch in touches){
        CGPoint location = [touch
            locationInNode:(self)];
        [rocketPlayer runAction:[SKAction
            moveToX:location.x duration:0.1]];
    }
}
```

5. Wnioski

W niniejszym artykule przedstawiono analizę porównawczą języków programowania Swift oraz Objective-C na przykładzie stworzonej gry mobilnej ping-pong. Aplikacja przeszła testy pod względem wydajnościowym oraz strukturalnym.

Na podstawie otrzymanych wyników, można stwierdzić, że wybór technologii jest uzależniony od funkcjonalności przyszłej aplikacji. Język Objective-C w szybszy sposób przekazuje dane do zewnętrznej bazy MySQL. Technologia Swift natomiast szybciej przechodzi między widokami oraz znacznie szybciej zapisuje dane do plików. Język Swift jest w dalszym ciągu ulepszany i dopracowywany, być może w przyszłości poprawią czas pracy z formatem JSON, wtedy wybór będzie bezproblemowy. Język Swift jest wydajniejszy niż język Objective-C, ponieważ więcej testów przeszedł znacznie szybciej.

Struktura kodu Objective-C jest dużo bardziej złożona i precyzyjna, dużo łatwiej popełnić błąd podczas pisania kodu. W strukturze kodu widać dużą przewagę języka Swift nad jego poprzednikiem, jest bardziej intuicyjny oraz współczesny. Sam fakt, że do stworzenia klasy wystarczy tylko jeden plik daje mu ogromną przewagę nad językiem Objective-C. Dodatkowym atutem języka Swift jest fakt, że trudniej popełnić błąd podczas pisania kodu, ponieważ składnia została uproszczona do minimum, np. nie ma potrzeby stawiania średnika po zakończonym poleceniu.

Znacznie lepszym wyborem dla początkujących programistów przy tworzeniu gier dedykowanych na system iOS jest język Swift, ze względu na swoją strukturę, ponieważ jest ona prostsza niż w języku Objective-C.

Literatura

[1] The History of iOS, from Version 1.0 to 13.0, https://www.lifewire.com/ios-versions-4147730 [16.06.2020].

[2] About Swift, https://swift.org/about [16.06.2020].

[3] C. G. Garcia, J. P. Espada, B. C. Pelayo G-Bustelo, J. M. Cueva Lovelle, Swift vs. Objective-C: A New Programming Language,Regular Issue, https://www.researchgate.net/publication/277142254_Swift_vs_Objective-C_A_New_Programming_Language [30.06.2020].

[4] K.E. Sienkiewicz, E. Łukasik, Porównanie aplikacji mobilnej w językach Swift i Objective-C, Journal of Computer Sciences Institutes, http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-875445e3-044-6b5-b16d-880f12cc7ab3 [30.06.2020].

[5] V. M. Santana, P. Centoze, Seciurity mechanisms and analysis for insecure data storage and unintended data leakage for mobile applications, International Journal of Computer and Technology https://www.researchgate.net/publication/324985466_SECURITY_MECHANSMS_AND_ANALYSIS_FOR_INSECURE_DATA_STORAGE_AND_UNINTENDED_DATA_LEAKAGE_FOR_MOBILE_APPLICATIONS [30.06.2020].

[6] K. Gut, M. Skublewska-Paszkowska, E. Łukasik, J. Smołka, Comparison of programming languages on the iOS platform in terms of performance, IAPGOŚ http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech- 0bbfbb45-b7bf-4703-91aa-169a26d70236 [30.06.2020].

[7] Xcode, https://developer.apple.com/xcode/ [16.06.2020].

[8] SpriteKit, https://developer.apple.com/ documentation/spritekit/ [16.06.2020].

[9] M. Lassoff, T. Stachowitz, Swift Fundamentals: The Language of iOS Development. 2014.

[10] D. Chisnall, Objective-C Phrasebook, Second Edition. 2012.

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 309–316
Received: 13 July 2020
Accepted:29 August 2020

Comparison of REST and GraphQL web technology performance

Porównanie wydajności technologii webowych REST i GraphQL

Mateusz Mikuła*, Mariusz Dzieńkowski

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

Abstract

The aim of the study was to compare the performance of two data exchange styles commonly used in web applications, i.e. REST and GraphQL. For the purposes of the study two test applications were developed containing the same functionalities, one of which was REST and the other one was GraphQL. They were used for performance tests done with the help of the JMeter tool, during which measurements of the total processing time of requests and the volume of data downloaded and sent were performed. An experiment was developed that tested the basic operations found in most network services: display, add, update, and delete data. The most attention was devoted to the information display operation in the case of which load tests were done. On the basis of performed studies and obtained results, no differences in performance during the operation of adding, editing and deleting data by applications based on REST API and GraphQL were found. During the display operation under heavy load conditions and while downloading small portions of data, the service using GraphQL had a better performance. When downloading large portions of data, the REST-based service exhibited a higher performance.

Keywords: REST; GraphQL; web service; performance testing

Streszczenie

Zrealizowano badania, których celem było porównanie wydajności dwóch, szeroko stosowanych w aplikacjach webowych stylów wymiany danych REST i GraphQL. Na potrzeby badań opracowano dwie usługi testowe, zawierające te same funkcjonalności, z których jedna była serwisem REST, a druga GraphQL. Posłużyły one do testów wydajnościowych, przeprowadzonych za pomocą narzędzia JMeter, podczas których wykonywano pomiary całkowitego czasu przetworzenia żądań oraz wielkości pobieranych i wysyłanych danych. Opracowano eksperyment, w ramach którego testowano podstawowe operacje występujące w większości usług sieciowych: wyświetlanie, dodawanie, aktualizowanie oraz usuwanie danych. Najwięcej uwagi poświęcono operacji wyświetlania informacji, w przypadku której wykonano testy obciążeniowe. Na podstawie zrealizowanych badań i uzyskanych wyników nie stwierdzono różnic w wydajności podczas realizacji operacji dodawania, edycji i usuwania danych przez aplikacje oparte na REST API i GraphQL. Podczas operacji wyświetlania w warunkach dużego obciążenia i w przypadku pobierania małych porcji danych lepszą wydajność miała usługa wykorzystująca GraphQL. Natomiast w przypadku pobierania dużych porcji danych wyższą wydajność uzyskiwała usługa oparta na REST.

Słowa kluczowe: REST; GraphQL; usługa internetowa; testowanie wydajności

*Corresponding author

Email address: mateusz.mikula@pollub.edu.pl (M. Mikuła)

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

1. Wstęp

1.1. Style wymiany danych REST i GraphQL

Aplikacje webowe są obecnie szeroko stosowane ze względu na popularność Internetu. Działają one w przeglądarce internetowej przez co wielu użytkowników może jednocześnie z nich korzystać. Aplikacje webowe komunikują się z usługami sieciowymi poprzez różne interfejsy programistyczne (API) za pomocą żądań i odpowiedzi używając do tego celu różnych protokołów (HTTP, FTP, POP3, itd.). Do tej pory opracowano wiele standardów, które określają sposób komunikacji klienta z serwerem. Na początku były to protokoły takie jak RPC, COBRA, SOAP, które były złożone, a przez to trudne w użyciu. Problemy te zostały wyeliminowane przez powstały w 2000 roku styl architektoniczny REST (Representational State Transfer) [1], który jest skalowalnym, prostym, rozszerzalnym i zoptymalizowanym wzorcem architektury określającym format przesyłanych informacji [2]. Pomimo wielu zalet i to rozwiązanie ma swoje ograniczenia, które ujawniły się w przypadku popularnych w ostatnim czasie serwisów społecznościowych, w których korzystanie z REST okazało się uciążliwe ze względu na nieakceptowalny przez użytkowników, długi czas wczytywania się wyświetlanych postów na urządzeniach mobilnych [3]. Rozwiązaniem tej kwestii zajęli się programiści firmy Facebook i w 2018 roku opublikowali pierwszą wersję języka zapytań dla interfejsów API o nazwie GraphQL, do komunikowania się z serwerem [4].

Wybór stylu wymiany danych dla tworzonej aplikacji, ma istotne znaczenie w kontekście czasu dostarczania danych. Im więcej informacji aplikacja jest w stanie przetworzyć w określonym przedziale czasu, tym jest ona bardziej wydajna. Należy pamiętać, że wydajność to jeden z najważniejszych aspektów powodzenia funkcjonowania rozwiązania w świecie rzeczywistym [5]. Dotyczy to szczególnie aplikacji internetowych, dających możliwość jednoczesnego dostępu bardzo wielu

użytkownikom, dla których priorytetem jest natychmiastowa i nieprzerwana dostępność usług w sieci [6].

1.2. Testowanie wydajności

Do sprawdzenia czy system, przy obciążeniu zgodnym z planowanym wykorzystaniem, będzie nadal działał prawidłowo, a odpowiedzi będą generowane w akceptowanym czasie, służą testy wydajnościowe [5]. Przeznaczone są one do oceny niezawodności, a co się z tym wiąże, jakości oprogramowania internetowego. Odpowiadają na pytanie, jak duże obciążenie badany system jest w stanie wytrzymać. Głównym celem testów wydajnościowych jest zbadanie systemu pod symulowanym działaniem wirtualnych użytkowników [7]. Najważniejszym parametrem do oceny wydajności jest czas, w którym dany system lub aplikacja musi wykonać żądaną akcję. Aplikacje internetowe są trudne do przetestowania, szczególnie pod względem wydajnościowym ze względu na to, że występują kwestie nieprzewidywalne takie jak wielkość obciążenia czy czas reakcji [8].

Testowanie wydajności najczęściej odbywa się w sposób automatyczny i sprowadza się w większości przypadków do sprawdzenia poprawnego działania badanego systemu pod określonym obciążeniem [7].

Zalety testowania automatycznego to przyspieszenie niektórych czynności, a tym samym skrócenie czasu całego procesu, eliminacja błędów ludzkich, możliwość generowania różnego rodzaju raportów, a także redukcja kosztów na etapie testowania. Narzędziami stosowanymi w automatyzacji testów wydajnościowych są te służące do nagrywania, symulowania i rejestrowania procesu działania aplikacji.

1.3. Przegląd literatury

Na forach i blogach wśród deweloperów oprogramowania trwa dyskusja na temat, który z istniejących modeli architektur (SOAP, REST, GraphQL) będzie najbardziej optymalny dla danego typu aplikacji. Kwestia ta jest ciągle przedmiotem badań, których wyniki pojawiają się w publikacjach naukowych i studenckich pracach dyplomowych.

W pracy [9] porównywano wydajność 3 aplikacji webowych udostępniających usługi internetowe jednocześnie za pomocą dwóch technologii REST i GraphQL. Zaobserwowano, że w dwóch aplikacjach migracja do GraphQL spowodowała wzrost wydajności, biorąc pod uwagę średnią liczbę żądań na sekundę i szybkość przesyłania danych. Jednak dla obciążeń powyżej 3000 żądań usługi oparte na GraphQL działały mniej wydajnie (wyniki w zakresie od 98 do 2159 kB/sek.) od usług typu REST. Natomiast w przypadku typowych obciążeń (100 żądań), usługi oparte na architekturach REST i GraphQL osiągały podobną wydajność z zakresu 6,34 – 7,68 żądań/sek.

Korzyści przejścia z API opracowanego na bazie standardu REST do API zrealizowanego w standardzie GraphQL, zebrane od specjalistów mających doświadczenie w tym zakresie, poddano praktycznej ocenie w artykule [10]. W tym celu przeprowadzono migrację siedmiu systemów z REST do GraphQL i w związku z tym dzięki zastosowaniu GraphQL nastąpiło zmniejszenie rozmiaru dokumentów JSON, zwracanych przez interfejsy REST API o 94% w liczbie pól i 99% w liczbie bajtów (oba wyniki są medianą).

Brito i Valente [3] wykonali doświadczenie w celu porównania wysiłku, jaki trzeba włożyć w generowanie zapytań do GraphQL i implementacje usług webowych REST. Badania, w których wzięło udział 22 studentów, wykazały, że budowanie zapytań GraphQL jest szybsze niż implementacja usług REST.

Analizą porównawczą wydajności rozwiązań wykorzystujących GraphQL i REST zajmował się Cederlund w swojej dysertacji [11]. Jako przypadków testowych używał rzeczywistych aplikacji do oceny opóźnienia, objętości danych i liczby zapytań. Przypadki testowe zostały tak zaprojektowane, aby możliwe było testowanie pojedynczych żądań oraz równoległych i sekwencyjnych przepływów danych. W badaniach skoncentrowano się na pomiarach czasów odpowiedzi zmieniających się w zależności od wielkości przesyłanych danych. Wyniki analiz pokazały, że zastosowanie GraphQL zmniejszyło czas odpowiedzi dla równoległych i sekwencyjnych przepływów danych, jednak użycie pojedynczego, zwykłego endpointa REST gwarantowało najlepszą wydajność.

Taskula w swojej pracy dyplomowej [12] porównywał dwa rozwiązania REST i GraphQL, które wykorzystywał do pobierania danych. W ramach studium przypadku wykonane zostały pomiary wydajności, która zamieniała się wraz ze zmianami rozmiaru transferowanych danych. Okazało się, że w wielu przypadkach testowych GrahpQL lepiej wykorzystywał dane, natomiast REST mógł się równać tylko w przypadku intensywnego filtrowania pól. Dodatkowo w pracy porównywano złożoność i możliwość dalszego rozwijania aplikacji z API wykonanego na bazie REST oraz GraphQL. Analiza jakościowa wykazała, że realizacja API w technologii GraphQL skutkowała mniejszą jego złożonością oraz dawała w przyszłości większe możliwości rozwoju.

W wyniku przeprowadzenia pilotażowego testu [13], porównującego REST z GraphQL okazało się, że pierwsze, starsze rozwiązanie jest szybsze dla prostych ustrukturyzowanych zapytań, takich jak pobieranie informacji tylko z jednego źródła lub tabeli. Ponadto okazało się, że różnica w wydajności w zakresie czasu odpowiedzi rośnie wykładniczo wraz z rozmiarem bazy danych. Pobierając z kolei więcej informacji, GraphQL generuje wolniejsze odpowiedzi w zakresie od 64% do 115%. W przypadku większych baz danych i konstruowaniu bardziej złożonych zapytań GraphQL lepiej się sprawdzał. Różnica pomiędzy GraphQL i REST wynosiła 25%.

W ramach pracy [14], zaproponowano metodologię, która umożliwiła odpowiedź na pytanie, która technologia GraphQL czy REST jest szybsza i lepiej zoptymalizowana. Badaniom poddano start-up - społecznościową aplikację internetową, w której testowano korzyści z migracji z REST do GraphQL. Badania obejmowały

dwie sytuacje: w pierwszej żądania były przesyłane w sposób sekwencyjny, a w drugiej równoległy. Główny wniosek płynący z pierwszej badanej sytuacji jest taki, że GraphQL oferuje krótszy czas odpowiedzi dla każdej sekwencji zapytań. Średnia różnica między obiema technologiami wyniosła 159 ms, co czyni GraphQL o 46% szybszym od REST. Z drugiej badanej sytuacji wynika, że także i w tym przypadku GraphQL był o 35% szybszy niż REST. Ten 11% spadek pomiędzy obiema sytuacjami można wytłumaczyć rodzajem wykonywanych żądań. W przypadku wykonania sekwencyjnego drugiego testu, realizacja za pomocą REST zajęłaby około 3 razy więcej czasu. W związku z tym schemat równoległy sprawia, że REST staje się bardziej konkurencyjny w stosunku do GraphQL. Biorąc pod uwagę rozmiar odpowiedzi, w pierwszej sytuacji GraphQL był o 21% lżejszy, natomiast w drugiej o 71%.

Przywołane prace i ich wyniki nie wskazują jednoznacznie, która technologia jest zdecydowanie lepsza. Odpowiedź na to pytanie jest uwarunkowana kilkoma czynnikami jak na przykład typ i przeznaczenie danego oprogramowania oraz rodzaj i wielkość grupy potencjalnych użytkowników.

1.4. Cel i zakres pracy

W Internecie znajduje się dużo publikacji pokazujących zalety i wady stylów wymiany danych REST i GraphQL. Jednak wciąż niewiele artykułów odpowiada na pytanie, które rozwiązanie jest lepsze, potwierdzając równocześnie to w sposób ilościowy na podstawie wyników z przeprowadzonych badań. Celem tej pracy jest dalsze wypełnienie tej luki poprzez przeprowadzenie analizy porównawczej wydajności dwóch technologii internetowych REST i GraphQL.

Zakres pracy obejmował następujące elementy: opracowanie aplikacji testowych, przygotowanie eksperymentu, dobór narzędzia diagnostycznego, zrealizowanie zautomatyzowanych testów wydajnościowych oraz opracowanie wyników i ich interpretację.

W pracy postawiono tezę badawczą, którą sformułowano w następujący sposób:
Aplikacje wykorzystujące rozwiązanie GraphQL w sytuacjach, w których klient potrzebuje zestawu danych w określonym formacie i możliwie najmniejszym rozmiarze, mają lepszą wydajność w porównaniu do aplikacji stosujących REST.

2. Metoda badań

2.1. Aplikacje testowe

Na potrzeby badań utworzono dwie usługi - aplikacje webowe zawierające te same funkcjonalności, ale wykorzystujące odmienne style wymiany danych: REST i GraphQL. W związku z tym w obu aplikacjach zaimplementowano odpowiednie API i poddano je testom sprawdzającym ich wydajność. Zadaniem przygotowanego oprogramowania było zarządzanie danymi dotyczącymi przeprowadzonych testów: informacjami na temat uczniów, nauczycieli, kursów oraz budynków, w których odbywały się egzaminy. Warstwy serwerowe aplikacji pobierają określone informacje z bazy danych, której diagram ERD został przedstawiony na rysunku 1.

Rysunek 1: Diagram związków encji

2.2. Środowisko testowe

Do przeprowadzenia badań wykorzystano dwa komputery przenośne. Jeden z nich pełnił rolę serwera, na którym uruchomione zostały dwie usługi webowe: pierwsza zawierająca REST API, druga GraphQL API. Drugi komputer pełnił rolę klienta, na którym zostało zainstalowane narzędzie umożliwiające przeprowadzenie testów - JMeter w wersji 5.2.1. Parametry poszczególnych urządzeń wykorzystanych do badań oraz opis serwera bazy danych, serwera HTTP i parametrów sieci znajdują się w tabelach 1, 2 i 3.

Tabela 1: Opis komputera pełniącego rolę serwera

Procesor	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 GHz, 2601MHZ, Rdzenie: 4, Procesory logiczne: 8
Pamięć RAM	16 GB
Karta sieciowa	Intel(R) Dual Band Wireless-AC 3165
System operacyjny	Microsoft Windows 10 Pro
Serwer bazy danych	MariaDB 10.1.25
Serwer HTTP	Apache Tomcat 9.0.3

Tabela 2: Opis komputera pełniącego rolę klienta

Procesor	Intel(R) Core(TM) i3-2350M CPU @ 2.30 GHz, 2300 MHz, Rdzenie: 2, Procesory logiczne: 4
Pamięć RAM	6 GB
Karta sieciowa	Intel(R) Centrino(R) Wireless-N 130
System operacyjny	Windows 7 Home Premium

Tabela 3: Opis sieci Wi-Fi

Protokół	Wi-Fi 4 (802.11n)
Pasmo sieci	2.4 GHz
Średnia prędkość pobierania danych	16383 kbps
Średnia prędkość wysyłania danych	1052 kbps

2.3. Narzędzie badawcze

Do testów wydajnościowych użyto programu Apache JMeter, który jest narzędziem typu open source napisanym w języku Java z wykorzystaniem biblioteki Swing [15]. Oprogramowanie to umożliwia automatyczne analizowanie wydajności różnych usług (w tym aplikacji internetowych) pod dużym obciążeniem, przy równoczesnej pracy wielu użytkowników. JMeter został wybrany do przeprowadzenia badań z następujących względów [9]:

- program jest dostępny bezpłatnie,
- posiada czytelny i intuicyjny interfejs użytkownika,
- daje możliwość tworzenia symulacji korzystania z aplikacji przez wielu użytkowników jako oddzielnych wątków,
- obsługuje wiele protokołów w tym HTTP, SOAP, POP3, itd.,
- udostępnia możliwość tworzenia zapytań HTTP korzystając z jego metod (GET, POST) oraz pozwala na przetwarzanie wyników odpowiedzi,
- umożliwia opracowanie zaawansowanych scenariuszy testowania,
- wyniki testów można prezentować za pomocą drzew, tabeli i wykresów.

JMeter jest zaawansowanym i kompleksowym rozwiązaniem do wykonywania nie tylko testów wydajnościowo-obciążeniowych, ale również testów jednostkowych, funkcjonalnych i regresyjnych. Osobom używającym tego narzędzia nie stawia wysokich wymagań co do ich umiejętności informatycznych – wymagana jest jedynie znajomość technologii HTML oraz konstruowania wyrażeń regularnych.

2.4. Scenariusze badawcze

Opracowano eksperyment, w ramach którego testowano podstawowe operacje umożliwiające zarządzanie danymi w bazie danych występujące w większości usług sieciowych: wyświetlanie, dodawanie, aktualizowanie oraz usuwanie danych. Najwięcej uwagi poświęcono operacji wyświetlania informacji, w zakresie której rozpatrywano trzy przypadki testowe:

1. nadmiarowego pobierania wszystkich danych przez jeden endpoint REST API oraz pobierania ściśle określonych danych poprzez API oparte na GraphQL;
2. pobierania wszystkich danych przez REST API (1 endpoint) oraz pobierania wszystkich danych poprzez GraphQL API;
3. pobierania kompletu danych z kilku endpointów REST API i jednego endpointa GraphQL API.

Badanie operacji wyświetlania (pierwszy i drugi przypadek) zostało przeprowadzone dla różnej liczby użytkowników generujących jednocześnie żądania (1, 5, 50, 100, 500 i 1000) oraz dla różnej liczby rekordów pobieranych z bazy (5, 50, 100). Przy testowaniu aplikacji podczas wykonywania operacji dodawania, usuwania czy edytowania danych nie zmieniano warunków obciążeniowych: tzn. nie brano pod uwagę przypadków testowania dla różnej liczby testów w bazie danych oraz różnej liczby użytkowników, korzystających równocześnie z aplikacji.

Do porównania wydajności aplikacji, wykorzystujących różne style wymiany danych posłużono się trzema metrykami:

- całkowitym czasem przetworzenia żądań wyrażonym w milisekundach,
- rozmiarem danych pobieranych w bajtach,
- rozmiarem danych wysyłanych w bajtach.

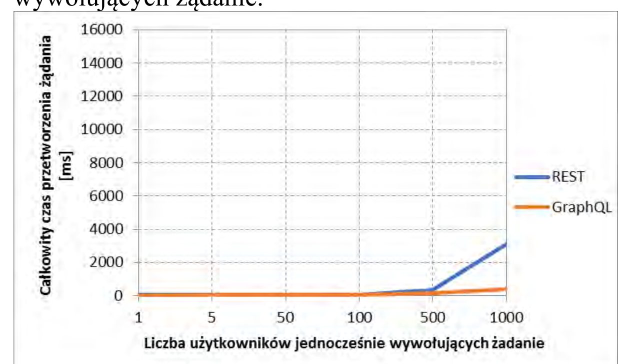
3. Wyniki badań

3.1. Wyświetlanie danych

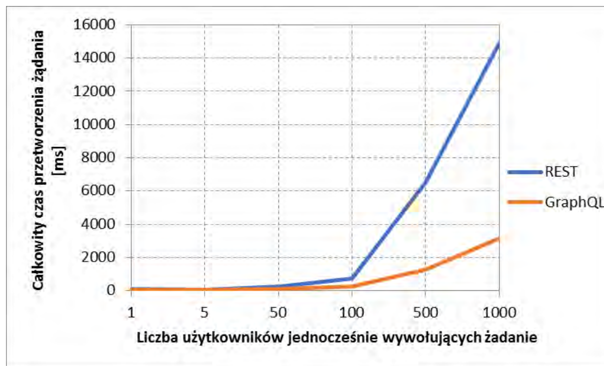
Przypadek 1: *Nadmiarowe pobieranie wszystkich danych poprzez jeden endpoint REST API versus pobieranie ściśle określonych danych poprzez API oparte na GraphQL*

W przygotowanej usłudze internetowej bazującej na architekturze REST został utworzony jeden endpoint /app/testy, poprzez który możliwe było pobranie wszystkich danych dotyczących przeprowadzonych testów wraz z informacjami o uczniu, nauczycielu, kursie oraz budynku, w którym odbywał się egzamin. W wielu sytuacjach klient może nie potrzebować tylu informacji. Ilość otrzymywanych informacji powinna uwzględniać typ (komputer, tablet, smartfon) i specyfikę urządzenia (np. rozdzielczość ekranu, moc obliczeniowa), na którym liczba jednocześnie wyświetlanych informacji może być różna. W części tej zbadano wydajność obu aplikacji w sytuacji, gdy serwis oparty na REST przekazuje często nadmiarowo wszystkie dane, a serwis oparty na GraphQL tylko dane ściśle określone i wymagane przez użytkownika. Ilość pobieranych danych może być również uzależniona i automatycznie dobierana do urządzenia, na którym będą one wyświetlane lub przetwarzane, biorąc pod uwagę rozdzielczość ekranu czy moc obliczeniową.

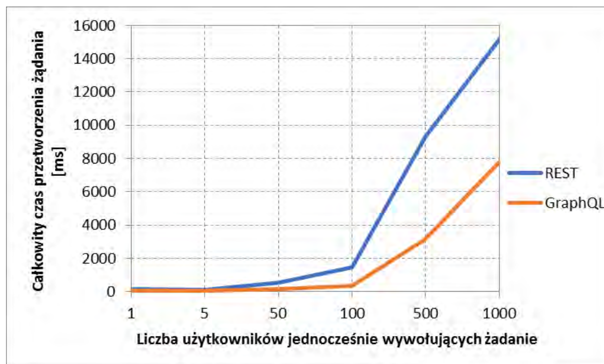
Na rysunkach 2, 3 i 4, w postaci wykresów liniowych, przedstawiono wyniki ukazujące wydajność testowanych aplikacji opartych na dwóch stylach wymiany danych REST i GraphQL. Oś Y reprezentuje całkowity czas przetwarzania żądań wyrażony w milisekundach, natomiast oś X liczbę użytkowników jednocześnie wywołujących żądanie.



Rysunek 2: Wyświetlanie danych (rozmiar: 5 rekordów).

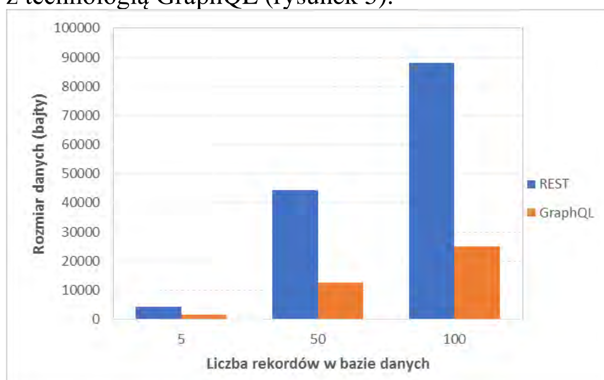


Rysunek 3: Wyświetlanie danych (rozmiar: 50 rekordów).



Rysunek 4: Wyświetlanie danych (rozmiar: 100 rekordów).

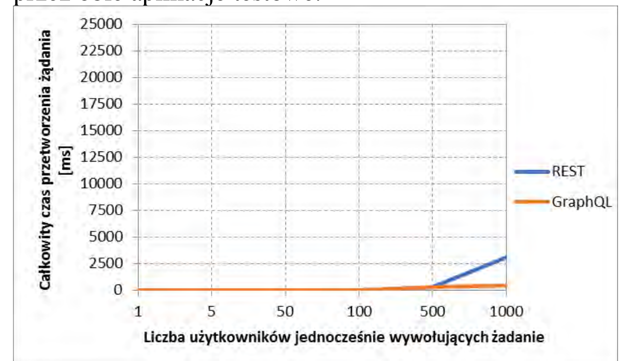
Na wykresach 2, 3 i 4 wyraźnie widać duży spadek wydajności aplikacji wykorzystującej styl REST, przy dużej liczbie użytkowników (>100) wysyłających jednocześnie żądania, zarówno w przypadku gdy otrzymywano 5, 50 czy 100 rekordów. Zaobserwowany duży spadek wydajności był spowodowany tym, że wielkość otrzymywanych danych w przypadku zastosowania architektury REST jest trzy razy większa w porównaniu z technologią GraphQL (rysunek 5).



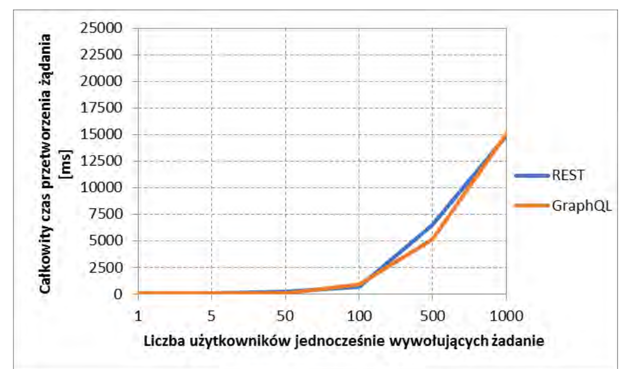
Rysunek 5: Rozmiar otrzymanych danych - REST vs GraphQL.

Przypadek 2: *Jednakowa liczba otrzymywanych (pobieranych) danych - pobieranie wszystkich danych przez REST API (przez 1 endpoint) oraz pobieranie wszystkich danych poprzez GraphQL API*

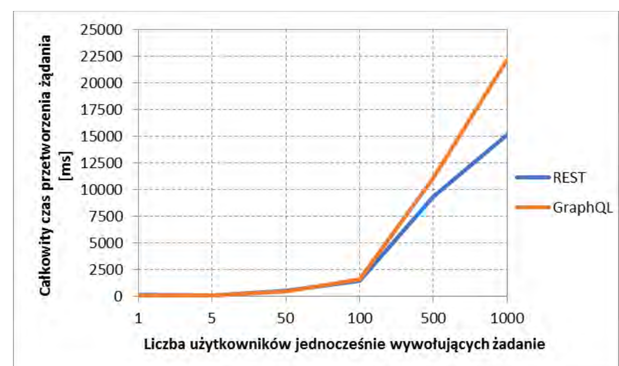
Drugi przypadek polegał na pomiarze tych samych parametrów, tym razem podczas otrzymywania tej samej ilości informacji. W tym celu w aplikacji opartej na GraphQL klient pobierał wszystkie dane dotyczące testu, podobnie jak to miało miejsce w przypadku 1 w aplikacji korzystającej z architektury REST. Rysunki 6, 7 i 8 odzwierciedlają wyniki wydajności, traktowanej jako całkowity czas przetwarzania żądań, uzyskane przez obie aplikacje testowe.



Rysunek 6: Wyświetlanie danych (rozmiar: 5 rekordów).



Rysunek 7: Wyświetlanie danych (rozmiar: 50 rekordów).

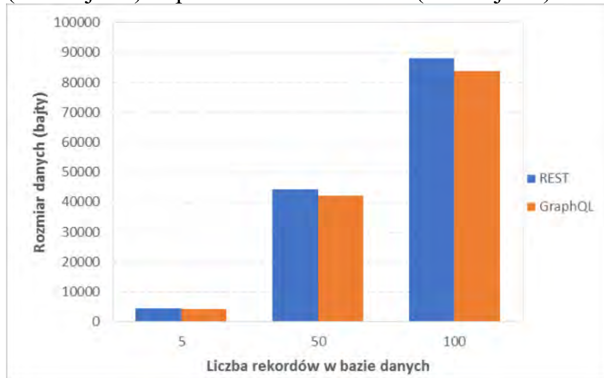


Rysunek 8: Wyświetlanie danych (rozmiar: 100 rekordów).

W przypadku 2, kiedy pobierana jest jednakowa ilość danych, przy dużym obciążeniu (>500 użytkowników), sytuacja staje się bardziej skomplikowana. Zastosowanie GraphQL okazało się bardziej wydajne, gdy pobierana była mała ilość danych: 5 rekordów (rysunek 6) i gdy liczba użytkowników generujących żądania była większa niż 500. W przypadku pobierania 50 rekordów, niezależnie od liczby użytkowników wydajność obu aplikacji była podobna. Usługa oparta na REST okazała się bardziej wydajna podczas jednoczesnego wyświetlania dużych ilości danych (100 rekordów) począwszy od liczby stu użytkowników.

Jeśli chodzi o rozmiar otrzymywanych danych (rysunek 9), to był on w przybliżeniu jednakowy, nato-

miast rozmiar wysłanych danych był zdecydowanie większy po stronie aplikacji wykorzystującej GraphQL (727 bajtów) w porównaniu do REST (133 bajtów).



Rysunek 9: Rozmiar otrzymanych danych - REST vs GraphQL.

Przypadek 3: *Pobieranie kompletu danych z kilku endpointów REST API i jednego endpointa GraphQL API*

Technologie webowe oparte na GraphQL są mniej złożone, gdyż wykorzystują jeden stały endpoint. Z kolei aplikacje bazujące na architekturze REST korzystają z wielu endpointów. W ostatnim przypadku przeanalizowano i porównano wydajność technologii REST i GraphQL podczas pobierania tych samych danych w sytuacji gdy aplikacja z REST korzysta z kilku endpointów, a aplikacja oparta na GraphQL tylko z jednego endpointa.

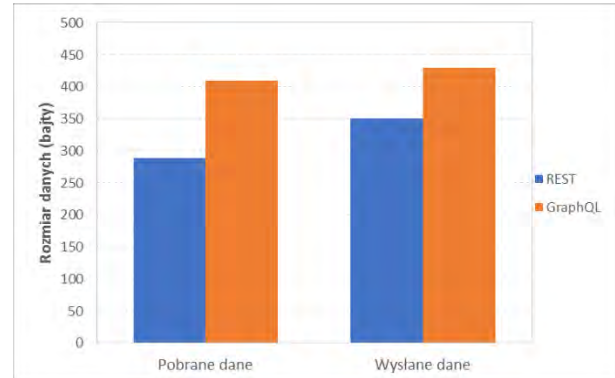
W technologii webowej opartej o REST chcąc uzyskać informacje o określonym kursie, należało użyć odpowiedniego adresu, innego niż na przykład przy otrzymywaniu informacji o danym budynku. Do celów badań została zasymulowana sytuacja, w której klient chce uzyskać pełne informacje o uczniu, nauczycielu i \budynku z określonymi identyfikatorami. W tym przypadku uzyskane czasy przetworzenia żądania były krótsze dla aplikacji wykorzystującej GraphQL. Także rozmiar otrzymanych danych był mniejszy w aplikacji korzystającej z GraphQL, a wysłanych w aplikacji z REST API (tabela 4).

Tabela 4: Porównanie otrzymanych wartości badanych parametrów w aplikacjach z REST i GraphQL

Interfejs (API)	Czas przetworzenia żądań (ms)	Rozmiar wszystkich pobranych danych (bajty)	Rozmiar wszystkich wysłanych danych (bajty)
REST	21	1091	404
GraphQL	15	894	667

3.2. Dodawanie danych

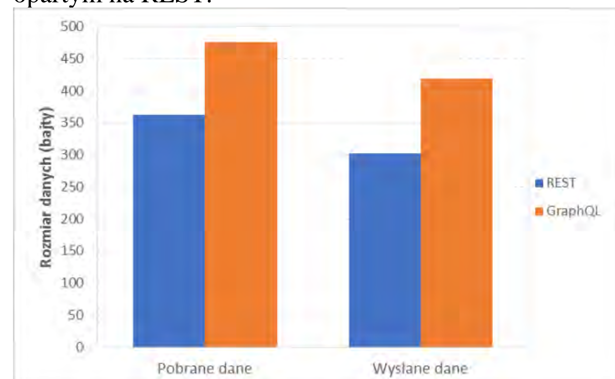
Podczas dodawania rekordu do bazy lepsze rezultaty odnoszące się do rozmiaru otrzymanych jak i wysłanych informacji, dało się zauważyć w usłudze opartej na stylu REST (rysunek 10). Jednak czas przetworzenia żądania był lepszy w przypadku aplikacji z GraphQL. Klient, aby dodać dane o nowym budynku w aplikacji z REST czekał na przetworzenie żądania średnio 144 ms. W GraphQL ta sama operacja była wykonywana w czasie średnio 123 ms.



Rysunek 10: Porównanie rozmiarów danych w REST i GraphQL.

3.3. Aktualizowanie danych

Podczas operacji aktualizowania danych nie zaobserwowano różnicy w czasach przetworzenia żądania w przypadku obu aplikacji testowych. Uzyskane średnie wartości czasu wykonania operacji edycji danych były praktycznie jednakowe (12 milisekund w aplikacji opartej na REST i 13 milisekund dla aplikacji z GraphQL). Natomiast różnice można zauważyć w rozmiarach otrzymanych i wysyłanych danych, podobnie jak to miało miejsce podczas dodawania informacji do bazy danych. Rysunek 11 pokazuje, że w tej operacji lepsze wyniki osiągnął klient, komunikując się z serwisem opartym na REST.



Rysunek 11: Porównanie rozmiarów danych w REST i GraphQL.

3.4. Usuwanie danych

Ostatnią testowaną operacją dla opracowanych usług internetowych było usuwanie danych z bazy. W obu aplikacjach czasy przetworzenia żądań miały podobne wielkości (tabela 5). W usłudze bazującej na interfejsie REST, klient, aby usunąć określony budynek z bazy danych potrzebował przesłać żądanie o rozmiarze 187 bajtów. Natomiast w aplikacji wykorzystującej GraphQL API, klient wysyłał żądanie zawierające 286 bajtów. Z tego wynika, że w tej sytuacji lepiej wypada aplikacja wykorzystująca interfejs REST. Jednak należy pamiętać, że każdy rekord w bazie danych w usłudze z REST API posiada swój odrębny adres, który pozwala na jego usunięcie. W rozwiązaniu stosującym GraphQL klient posługuje się jednym, stały endpointem, co zde-

cydowanie ułatwia komunikację. Kolejną rzeczą, na którą trzeba zwrócić uwagę jest to, że przy próbie usunięcia encji (również przy edycji danych), która nie istnieje w bazie, w podejściu REST klient otrzyma błąd wykonania żądania. W technologii GraphQL klient zaś otrzyma status 200 HTTP, oznaczający poprawne przetworzenie żądania, co może powodować wyświetlanie niewłaściwego komunikatu po stronie klienta (tabela 6).

Tabela 5: Rezultat wykonania żądania usunięcia rekordu z bazy danych w aplikacjach stosujących styl REST i GraphQL

Technologia	Czas przetworzenia żądania (ms)	Status	Rozmiar pobranych danych (bajty)	Rozmiar wysłanych danych (bajty)
REST	535	OK	187	215
GraphQL	495	OK	286	238

Tabela 6: Rezultat wykonania żądania usunięcia rekordu, który nie istnieje w bazie danych w aplikacjach z REST oraz GraphQL

Technologia	Czas przetworzenia żądania (ms)	Status	Rozmiar pobranych danych (bajty)	Rozmiar wysłanych danych (bajty)
REST	56	Błąd	307	215
GraphQL	22	OK	389	238

4. Wnioski

Wyniki badań empirycznych są bardzo pomocne przed podjęciem decyzji o wyborze najbardziej optymalnego rozwiązania w procesie wytwarzania oprogramowania. W związku z tym, w niniejszej pracy przeprowadzono analizę porównawczą pod względem wydajności dwóch, szeroko stosowanych obecnie podejść do budowy interfejsu API: REST i GraphQL. Badania zostały zrealizowane na utworzonych do tego celu dwóch usługach, z których pierwsza do wymiany danych używała architektury REST, natomiast druga wykorzystywała język i silnik GraphQL. Do pomiarów czasów przetwarzania żądań oraz rozmiarów wysyłanych i pobieranych danych posłużyło narzędzie diagnostyczne JMeter.

Zrealizowane badania pokazały, kiedy i jakie różnice występują między obiema technologiami w kontekście ich wydajności. W przypadku operacji wyświetlania, klient korzystający z usługi internetowej opartej na REST, nie może ograniczyć liczby otrzymywanych danych, ponieważ zawsze w odpowiedzi otrzyma pełny zestaw informacji. W tej sytuacji część danych może być dla klienta niepotrzebna. Natomiast w GraphQL to klient określa, posługując się językiem zapytań, jakie informacje chce uzyskać w odpowiedzi od serwera. W tym przypadku, bez względu na liczbę pobieranych rekordów z bazy, usługa internetowa oparta na GraphQL szybciej przetwarzała żądania niż miało to miejsce w aplikacji z interfejsem REST. Wynikało to głównie z ilości otrzymywanych danych, która była ponad trzykrotnie mniejsza w GraphQL niż w REST.

Kluczowym rozpatrywanym przypadkiem, była sytuacja, gdy klient zarówno poprzez jedną jak i drugą usługę otrzymywał tę samą ilość danych (wszystkie dane) i to ta część badań potwierdziła postawioną na początku pracy tezę. W obu technologiach, przy pobieraniu małych porcji danych (5 rekordów) przez 1 endpoint, przy dużym obciążeniu, lepiej wypadała usługa stosująca GraphQL. Jednak w podobnych warunkach, w przypadku pobierania dużych porcji danych (100 rekordów) bardziej wydajna była usługa wykorzystująca interfejs REST. Rozmiar wysyłanych danych był większy w GraphQL, ponieważ klient w przesyłanej wiadomości musiał przy pomocy języka zapytań zawrzeć wszystkie informacje, które chciał uzyskać w odpowiedzi. Należy zatem podkreślić, że zapytania w GraphQL mogą być zawiłe i skomplikowane.

Podczas pobierania kompletu danych z kilku endpointów REST API i jednego endpointa GraphQL API uzyskane czasy przetworzenia żądania były krótsze dla aplikacji wykorzystującej GraphQL. Rozmiar otrzymanych danych był mniejszy w aplikacji korzystającej z GraphQL, a wysłanych w aplikacji z REST API.

W przypadku operacji dodawania, edycji i usuwania danych z bazy, porównywane były czasy przetwarzania pojedynczych żądań. Analiza czasów wykazała, że różnice w wydajności przetwarzania żądań były minimalne. Jeśli chodzi o wielkość wysyłanych i pobieranych informacji, we wszystkich wyżej wymienionych operacjach bardziej wydajny okazywał się interfejs opracowany na bazie technologii REST.

Literatura

[1] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D, University of California, Irvine, 2000.

[2] R. T. Fielding, R. N. Taylor, Principled design of the modern Web architecture, Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium, Limerick, Ireland, 2000: 407-416, https://doi.org/10.1145/337180.337228, [10.07.2020].

[3] G. Brito, M. T. Valente, REST vs GraphQL: A Controlled Experiment, 2020 IEEE International Conference on Software Architecture (ICSA), 81-91, [10.07.2020].

[4] GraphQL, http://spec.graphql.org/, [10.07.2020].

[5] M. Prywata, Testowanie aplikacji i stron internetowych, Polska Agencja Rozwoju przedsiębiorczości, Warszawa, 2009, https://www.parp.gov.pl/publications/publication/testowanie-aplikacji-i-stron-internetowych, [09.07.2020].

[6] S. Sharmila, E. Ramadevi, Analysis of Performance Testing on Web Application, International Journal of Advanced Research in Computer and Communication Engineering, Vol. 3, Issue 3 (2014), https://ijarcce.com/wp-content/uploads/2012/03/IJARCCE4H-s-sharmila-Analysis-of-Performance-Testing-on-Web-Applications.pdf, [10.07.2020].

[7] P. Marek, Weryfikacja i automatyzacja procesu testowania oprogramowania, CORE Magazine, 2010.

[8] S. Dhiman, P. Sharma, Performance Testing: A Comparative Study and Analysis of Web Service Testing Tools, International Journal of Computer Science

and Mobile Computing, Vol. 5, Issue 6, (2016) 507-512, https://www.ijcsmc.com/docs/papers/June2016/V5I6201697.pdf, [09.07.2020].

[9] M. Seabra, F. Nazario, G. Pinto, REST or GraphQL?: A Performance Comparative Study, In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19). Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/3357141.3357149, [27.08.2020].

[10] G. Brito, T. Mombach, M. T. Valente, Migrating to GraphQL: A Practical Assessment, In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 140-150, https://doi.org/10.1109/SANER.2019.8667986, [27.08.2020].

[11] M. Cederlund, Performance of frameworks for declarative data fetching: An evaluation of Falcor and Realy+GraphQL, Dissertation, KTH, 2016, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-196058, [27.08.2020].

[12] T. Taskula, Advanced data fetching with graphql: Case bakery service, Dissertation, Aalto University, 2019, https://aaltodoc.aalto.fi/handle/123456789/37147, [27.08.2020].

[13] A. F. Helgason, Performance analysis of Web Services: Comparison between RESTful & GraphQL web services, Dissertation, University of Skövde, 2017, http://his.diva-portal.org/smash/record.jsf?pid=diva2%3A1107850&dswid=-9534, [29.08.2020].

[14] C. Oggier, How fast GraphQL is compared to REST APIs, Dissertation, Haaga-Helia University of Applied Sciences, 2020, http://urn.fi/URN:NBN:fi:amk-2020052714286, [29.08.2020].

[15] Apache JMeter, http://jmeter.apache.org/, [09.07.2020].

JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 317–325
Received: 28 April 2020
Accepted: 17 May 2020

The analysis of air pollution based on laser beam photo

Ocena zanieczyszczenia powietrza na podstawie zdjęcia wiązki lasera

Anna Pawelec, Rafał Maksim* , Maria Skublewska-Paszkowska

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

Abstract

The article presents a comparison of original methods of air quality measurement with a professional device Air Smartbox v. 1.2. The methodology consisted of laser beam analysis from the device. To enable detailed photo analysis for the research, an Android mobile application was developed. The OpenCV library was used to process the images. In the article, the hypothesis was put forward that the method using a binary threshold with a threshold value of 50 allows to obtain results closest to those of the station. This hypothesis was confirmed by the results of the experiments.
Keywords: air pollution; Android; OpenCV; image processing; Fourier Transform

Streszczenie

Artykuł przedstawia porównanie autorskich metod pomiaru jakości powietrza z profesjonalnym urządzeniem Air Smartbox v. 1.2. Metody polegają na analizie zdjęć wiązki lasera w zanieczyszczonym powietrzu. W celu przeprowadzenia badań została zaimplementowana aplikacja mobilna, dedykowana na system operacyjny Android, która umożliwia wykonanie zdjęć oraz ich późniejszą obróbkę i analizę. Do przetwarzania obrazów zastosowano bibliotekę OpenCV. W artykule postawiono hipotezę, że metoda wykorzystująca progowanie binarne z wartością progowania wynoszącą 50 pozwala uzyskać wyniki najbardziej zbliżone do wyników ze stacji. Hipoteza ta została potwierdzona uzyskanymi wynikami badań.
Słowa kluczowe: zanieczyszczenie powietrza; Android; OpenCV; przetwarzanie obrazów; Transformata Fouriera

*Corresponding author

Email address: rafal.maksim@outlook.com (R. Maksim)

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

1. Wstęp

Wraz ze wzrostem świadomości o zanieczyszczeniu powietrza powstają kolejne metody do jego pomiaru. W związku z tym wzrosła również popularność urządzeń mierzących stężenia tych zanieczyszczeń. Niestety większość profesjonalnych urządzeń jest droga, więc na zakup takiego urządzenia może sobie pozwolić małe grono użytkowników. Autorzy tej pracy postawili sobie za cel stworzenie taniego, ale profesjonalnego sposobu pomiaru jakości powietrza dostępnego w miejscu przebywania użytkownika. W tym celu została zaimplementowana aplikacja mobilna dedykowana na system Android, umożliwiająca analizę zanieczyszczenia powietrza na podstawie analizy zdjęcia wiązki lasera.

Autorzy postawili hipotezę, że metoda wykorzystująca progowanie binarne z wartością progowania wynoszącą 50 (opisywana również jako Binary threshold 50) pozwala uzyskać wyniki najbardziej zbliżone do wyników ze stacji pomiarowej. W niniejszym artykule zostaną przedstawione wyniki udowadniające to założenie.

2. Przetwarzanie i filtrowanie obrazu

2.1. Filtrowanie obrazu

Obraz, czyli macierz, którą komputer odczytuje jako pewną siatkę liczb [1] może być poddany przetwarzaniu i filtracji. Pozwala to na szeroki zakres rozpoznawania i przetwarzania informacji zawartych na obrazie. Filtracją nazywa się pewną funkcję matematyczną, która pozwoli na przekształcenie obrazu, zwykle celem polepszenia jego jakości czy ostrości [2]. Filtry można podzielić na liniowe oraz nieliniowe. Prostsze w wykonaniu są filtry liniowe, ale są one uboższe w możliwości niż filtry nieliniowe. Przykładem filtru liniowego może być średnia arytmetyczna wygładzająca obraz poprzez uśrednianie punktów sąsiedztwa, a nieliniowego – metoda Perony-Malika, używająca współczynnika wpływającego na stopień rozmycia w zależności od estymatora krawędzi [2]. Filtr uśredniający posiada jednak wady w postaci zbyt dużego rozmycia obrazu.

Innym filtrem nieliniowym jest filtr Laplace'a, który ma za zadanie wyostrzenie obrazu. Pozwala on na wydobycie z obrazu małych punktów, jednakże jest ona bardzo czuła na szumy. Opisane wyżej filtry są szeroko stosowane w analizie obrazów.

2.2. Wykrywanie krawędzi

Do wykrywania krawędzi warto rozważać użycie jednego z algorytmów detekcji krawędzi. Są to między innymi filtry Sobela, Robertsa czy Prewitta. Bardzo dokładnym filtrem jest metoda opisana przez Johna F. Canny'ego, która spełnia kryteria [2]:

- dobrej detekcji - znikome prawdopodobieństwo zaznaczeń punktów niebędących krawędzią obrazu;
- dobrej lokalizacji - punkty powinny być możliwie najbliżej krawędzi obrazu;
- pojedynczej odpowiedzi - nie powinny zostać zaznaczone fałszywe krawędzie, zaznaczenie jest tylko jedno.

Canny przedstawił sposób rozwiązania do tego podejścia, polegające na sekwencji poniższych operacji [2]:

1. zastosowanie filtru gaussowskiego celem redukcji szumów;
2. detekcja krawędzi z użyciem operatorów gradientu i określenie orientacji krawędzi;
3. usunięcie pikseli o wartości niższej niż maksymalna;
4. budowa histogramu dla obrazu konturu (konstrukcja krawędzi i progowanie histogramu).

2.3. Dyskretna transformata Fouriera

Francuski matematyk Jean Baptiste Joseph Fourier przedstawił wiele pomysłów wykorzystywanych w dziedzinach przetwarzania sygnałów oraz obrazu [3]. Obraz cyfrowy jest zbiorem liczb - uporządkowanym, zatem jest to po prostu ciąg dwuwymiarowy o wartościach rzeczywistych [4]:

$$L = \{L(m,n) \epsilon \Re:\ m = 0,1,\dots,M = -1; n = 0,1,\dots,N-1\} \quad (1)$$

- założono, że długość ciągu wynosi N, zatem indeksy k będą równe od 0 do N-1;
- $j = \sqrt{-1}$.

W związku z faktem, że obraz jest pewnym uporządkowanym zbiorem liczb można użyć dyskretnej transformacji Fouriera prostym przekształceniem przedstawionym poniżej [4]:

$$F(i,k) = \beta_L \cdot \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} L(m,n) \cdot \exp\left(\frac{-j \cdot 2 \cdot \pi \cdot m \cdot i}{M}\right) \cdot \exp\left(\frac{-j \cdot 2 \cdot \pi \cdot n \cdot k}{N}\right) \quad (2)$$

gdzie:

- β_L współczynnik dla transformacji w przód;
- β_F dla transformacji odwrotnej.

Ponad to, współczynniki te określane są jako [4]:

$$\beta_L = 1;\ \beta_F = \frac{1}{M \cdot N} \quad (3)$$

Jeśli poniższe zależności zostaną spełnione [4]:

$$\beta_L \cdot \beta_F = \frac{1}{M \cdot N} \quad (4)$$

to możliwe jest wykonanie odwrotnej transformacji Fouriera, określonej wzorem [4]:

$$L(m,n) = \beta_F \cdot \sum_{i=0}^{M-1}\sum_{k=0}^{N-1} F(i,k) \cdot \exp\left(\frac{j \cdot 2 \cdot \pi \cdot m \cdot i}{M}\right) \cdot \exp\left(\frac{j \cdot 2 \cdot \pi \cdot n \cdot k}{N}\right) \quad (5)$$

W pracy „Wykorzystanie transformacji Fouriera do filtracji szumu informacyjnego z obrazów fotolotniczych" [5] przedstawiono użycie transformacji Fouriera. Autor wykorzystał niedoskonałe panchromatyczne zdjęcia lotnicze zawierające różne szumy i zniekształcenia. Celem ich usunięcia, użył dyskretnej transformacji Fouriera i usunął z widma pewne elementy. Następnie zastosował odwrotną transformatę dla nowej funkcji widma. Jako efekt przeprowadzonego badania uzyskał on satysfakcjonujący wynik dla dwóch z trzech obrazów, ponieważ po zastosowaniu wyżej wymienionych metod ich jakość poprawiła się, a szumy zostały znacząco osłabione. Wynik ten utwierdził autorów niniejszego artykułu w przekonaniu, że dyskretna transformacja Fouriera będzie odpowiednia do zaimplementowania w wymyślonych przez nich metodach.

2.4. Splot

Innym możliwym do zastosowania podejściem jest splot. Filtr odczytuje piksele z macierzy kolejno zaczynając od lewej strony do prawej oraz od góry do dołu. W przypadku symetrycznego jądra środek macierzy powinien zachodzić na sąsiednie piksele. Każdy element macierzy zostaje pomnożony przez wartość piksela, z którą się pokrywa, natomiast wartości wynikowe zostają zsumowane. Wynik jest nową wartością dla elementu pokrywającego się ze środkiem macierzy. Dla przypadku niesymetrycznego jądra, środek macierzy zostaje obrócony przed wykonaniem funkcji splotu.

2.5. Progowanie

Następną możliwością obróbki graficznej jest progowanie, które ma na celu podział obrazu [6]. Ustalana jest wartość progowa, a każdy punkt zostaje z nią porównany. Obraz po binaryzacji jest podzielony na części, z których jedna składa się z barwy białej, a druga czarnej [2].

3. Aplikacja mobilna

Aplikacja mobilna stworzona przez autorów zaprojektowana została na system Android w wersji Oreo (8.0) lub wyższej. Wykorzystuje ona bibliotekę OpenCV [7]. Użytkownik po wykonaniu zdjęcia lub serii zdjęć może wyliczyć poziom zanieczyszczenia powietrza, sprawdzić archiwalne wyniki oraz dostosować wartość progowania, wybrać metodę analizy i wpisać miejsce pomiaru.

Na rysunku 1 przedstawiono wygląd strony głównej aplikacji. W celu analizy zdjęcia należy wcisnąć przycisk *Analyze*. Następnie należy sprawdzić, czy wiązka lasera jest dobrze widoczna w kadrze i zrobić zdjęcie wiązki lasera (rysunek 5) oraz potwierdzić wykonane zdjęcie do analizy (rysunek 3). Rysunek 4 przedstawia zdjęcie po obróbce oraz wynik otrzymany z analizy.

Zdjęcie, które ma być poddane analizie musi spełniać poniższe kryteria:

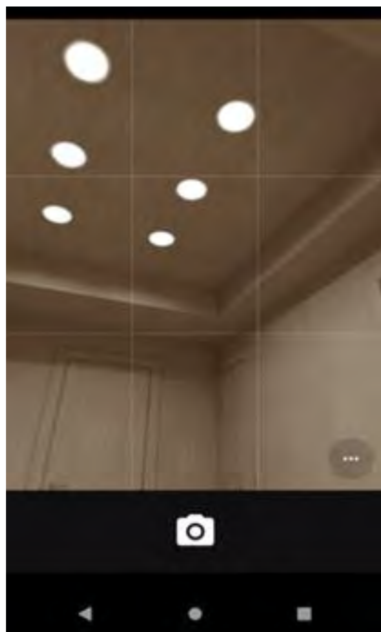
- ostrość musi być ustawiona dokładnie na wiązkę lasera;
- wiązka lasera musi przebiegać możliwie równolegle do poziomych krawędzi kadru;
- nie powinno posiadać innych źródeł światła;
- powinno zostać wykonane przy użyciu statywu i zdalnego wyzwalacza.

Po spełnieniu powyższych kryteriów może zostać ono poddane obróbce graficznej. Na rysunku 5 przedstawiono przykład poprawnie wykonanego zdjęcia.

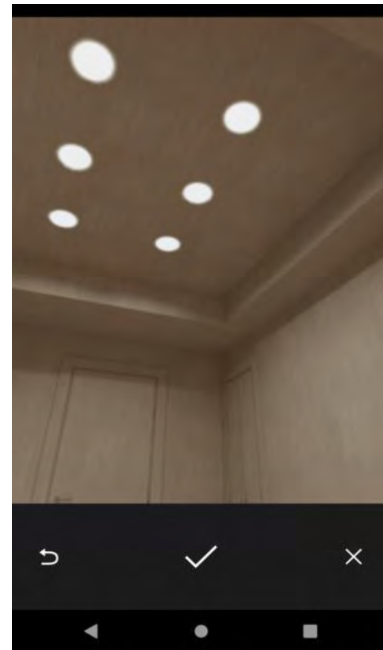
Zdjęcie może być przeanalizowane jedną z trzech metod:

- progowanie;
- dyskretna transformacja Fouriera oraz progowanie;
- zastosowanie splotu oraz progowania.

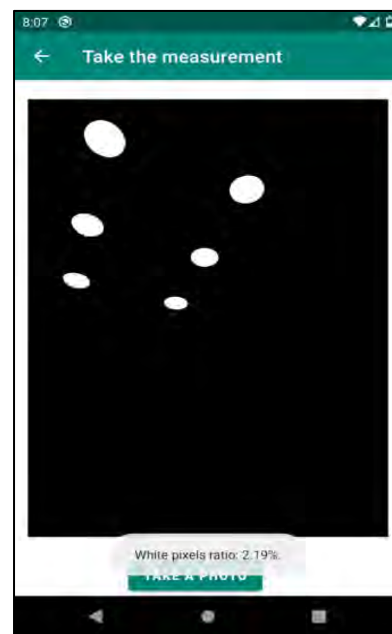
Rysunek 1: Strona główna aplikacji



Rysunek 2: Podgląd aparatu przed wykonaniem zdjęcia



Rysunek 3: Ekran zatwierdzania zdjęcia



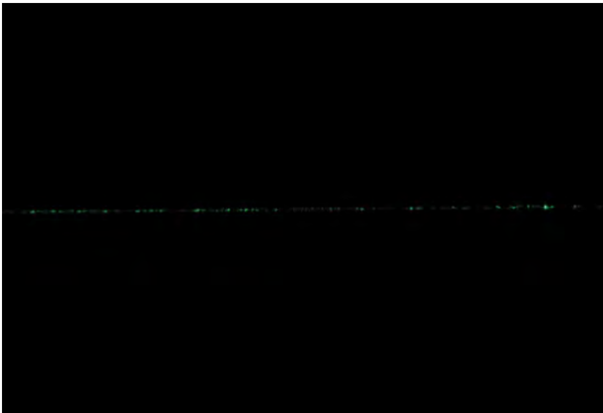
Rysunek 4: Wyświetlenie obrobionego zdjęcia i komunikatu

4. Obróbka graficzna wykonanego zdjęcia

Pierwsza dostępną metodą jest progowanie binarne. Autorzy zdecydowali się na wykorzystanie dwóch wartości progowania: 50 i 100. Zdjęcie jest poddawane działaniu tej metody.

Następną metodą, której można użyć do analizy zdjęcia jest dyskretna transformacja Fouriera i progowanie. Po użyciu dyskretnej transformacji Fouriera na zdjęcie należy nałożyć maskę celem usunięcia niskich częstotliwości. Następnie wykonywana jest odwrotna transformata Fouriera, co przywróci oryginalny format zdjęcia. Na końcu tak przetworzone zdjęcie poddaje się operacji progowania.

Ostatnią możliwą do wykonania metodą analizy jest metoda używająca filtra górnoprzepustowego i progowania. Wykorzystano w tym celu maskę [8] o wymiarach 3x3 przedstawioną na rysunku 6.



Rysunek 5: Poprawnie wykonane zdjęcie

$$\begin{bmatrix} 0,17 & 0,67 & 0,17 \\ 0,67 & -3,33 & 0,67 \\ 0,17 & 0,67 & 0,17 \end{bmatrix}$$

Rysunek 6: Maska o wymiarach 3x3

Zdjęcie jest przetwarzane z wykorzystaniem kernela (rysunek 6) i zostaje poddane progowaniu, czyli pierwszej opisanej metodzie.

Po wykonaniu obróbki zdjęcia można zliczyć wszystkie białe piksele i podzielić ich liczbę przez liczbę wszystkich pikseli na zdjęciu, a następnie pomnożyć przez 100, co pozwoli na uzyskanie procentowej zawartości cząsteczek w powietrzu. Opisane tutaj białe piksele to rozbłyski świetlne, czyli cząsteczki zanieczyszczeń zawieszonych w powietrzu, które odbiły światło lasera.

5. Metoda badawcza

5.1. Opis stanowiska

Rysunek 7 przedstawia stanowisko badawcze. Na stoliku została umieszczona stacja pomiarowa oraz laser ustawiony na statywie w taki sposób, aby wiązka lasera przebiegała równolegle do podłoża. W odległości 70 cm od urządzenia pomiarowego i statywu z laserem został ustawiony telefon, również na statywie.

Telefon użyty do doświadczenia to LG V30, który posiada aparat fotograficzny o następujących parametrach:

- kąt widzenia: 71°;
- maksymalna rozdzielczość zdjęcia: 16 Mpx;
- maksymalna przysłona f/1.6;
- optyczna stabilizacja obrazu;
- stała ogniskowa 30 mm.

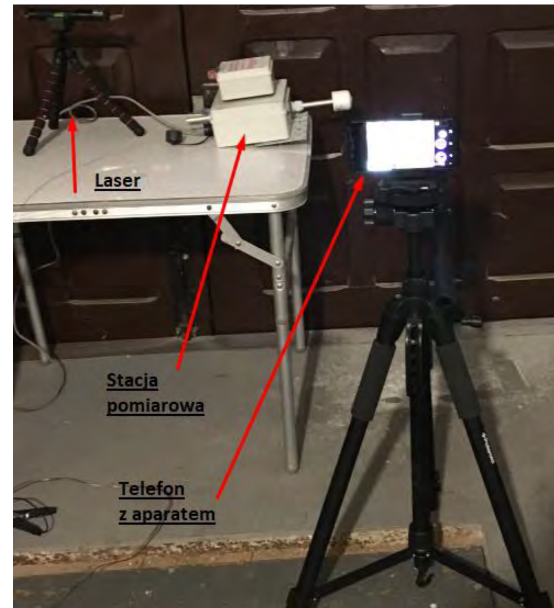
Podczas wykonywania zdjęć aparat miał ręcznie ustawioną ostrość oraz pracował w trybie manualnym z następującymi wartościami:

- wartość przysłony: f/1.6;
- rozdzielczość zdjęcia: 16 Mpx;
- czułość matrycy ISO: 100;
- czas naświetlania: 1 s.

Do eksperymentu wykorzystano laser o następujących parametrach:

- maksymalna moc wyjściowa mniejsza niż 5000 mW;
- długość fali wynosi 532 nm ± 10 nm.

Stacja pomiaru, jako urządzenie referencyjne, użyta do eksperymentu to Air Smartbox v. 1.2. Jest to urządzenie między innymi do pomiaru stężenia pyłów zawieszonych PM2,5, PM10 w powietrzu, mierzące stężenie zanieczyszczenia co pół minuty.



Rysunek 7: Stanowisko eksperymentu

5.2. Przeprowadzenie badania

Badanie zakładało przeprowadzenie dwóch eksperymentów, które odbyły się w późnych godzinach wieczornych. W pomieszczeniu, w którym były wykonywane, zostało zgaszone światło, aby wiązka lasera była dobrze widoczna. W tym czasie odbywały się w nim prace remontowe, co pozwoliło na uzyskanie zmiennego zanieczyszczenia powietrza.

Każdy z eksperymentów polegał na wykonywaniu zdjęcia wiązki lasera co pół minuty przez około 30 minut zdalnym wyzwalaczem migawki. W rezultacie otrzymano 63 zdjęcia dla pierwszego eksperymentu i 71 dla drugiego, które następnie zostały przetworzone przy użyciu aplikacji mobilnej, wykorzystując funkcję przetwarzania sekwencyjnego plików oraz wcześniej opisane metody.

5.3. Analiza porównawcza

Porównanie wyników badań polega na:

- wzrokowym porównaniu wykresów wyników badań z wynikami ze stacji pomiarowej;
- wykonaniu statystyki opisowej;
- analizie pochodnych różnic zbiorów;
- analizie korelacji Kendalla.

5.4. Wyniki i ich analiza

W tabeli 1 przedstawiono statystykę opisową otrzymanych wyników dla eksperymentu pierwszego, zaś w tabeli 2 dla eksperymentu drugiego.
Przyjęto następujące nazewnictwo:

- Binary threshold 100 oznacza metodę progowania binarnego z wartością progowania równą 100;
- Binary threshold 50 oznacza metodę progowania binarnego z wartością progowania równą 50;
- Fourier 100 oznacza metodę z transformatą Fouriera, progowaniem i wartością progowania równą 100;
- Fourier 50 oznacza metodę z transformatą Fouriera, progowaniem i wartością progowania równą 50;
- Kernel 100 oznacza metodę z kernelem, progowaniem i wartością progowania równą 100;
- Kernel 50 oznacza metodę z kernelem, progowaniem i wartością progowania równą 50.

Końcówki Ex1 i Ex2 oznaczają odpowiednio eksperyment 1. i eksperyment 2.

Każdy wykres zbioru danych może zostać w pewien sposób zmodyfikowany. Jedną z najprostszych operacji jest przesunięcie takiego wykresu o zadaną wartość względem osi OY. Przykładowo, prosty wykres funkcji liniowej y = x można przekształcić tak, aby przecinał oś OY w punkcie (0,1) zamiast w (0,0). W tym celu należy przesunąć wykres o 1 jednostkę w górę, co realizowane jest przez dodanie jedności do wzoru funkcji. Tym samym nowy wzór funkcji będzie wyglądał następująco: y = x + 1.
Zakładając, że funkcje f(x) = x oraz g(x) = x + 1 zostaną nałożone na ten sam układ współrzędnych, można łatwo zaobserwować, że oba wykresy zmieniają się w identyczny sposób. Skutkuje to tym, że dla każdego punktu na osi OX różnica g(x) - f(x) będzie taka sama. Na tej podstawie można stworzyć nową funkcję h(x) = g(x) - f(x), która będzie funkcją stałą. Wnioskując, jeżeli wartości dwóch równolicznych zbiorów danych zmieniają się w taki sam sposób to zbiór powstały z odjęcia ich od siebie powinien być funkcją stałą

Rysunki 8,9 oraz 10 przedstawiają wykresy pochodnych różnic wartości pomiędzy pomiarami.

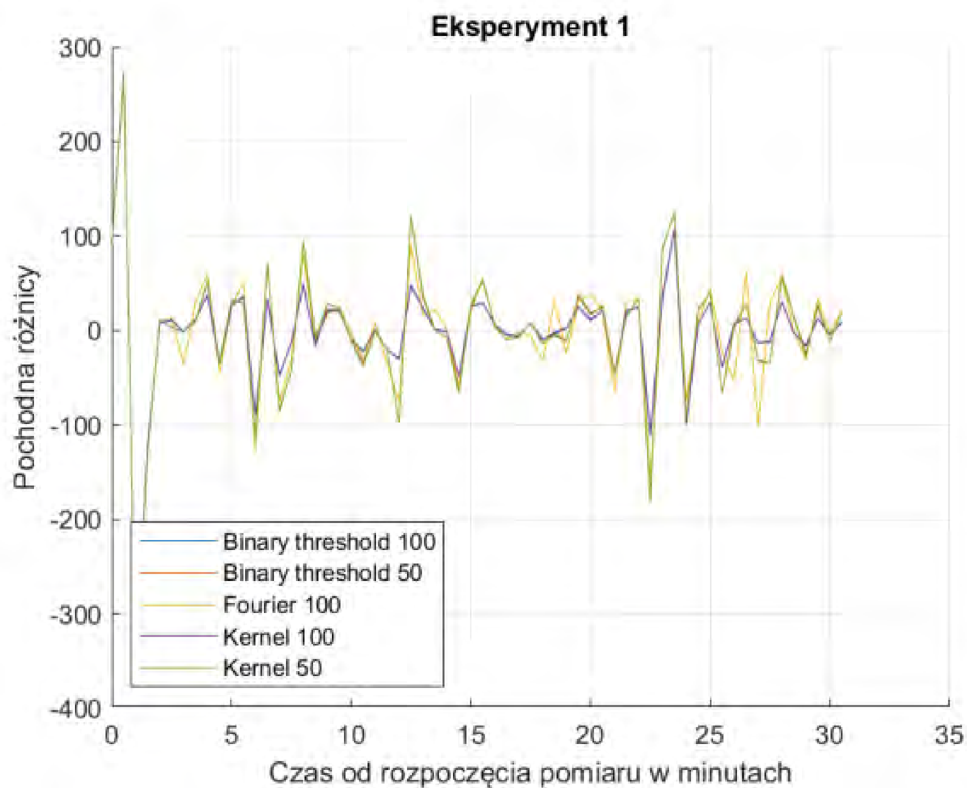
Przy użyciu korelacji Kendalla sprawdzono, w jakim stopniu dane pomiarowe uzyskane przy użyciu metod z aplikacji mobilnej są skorelowane z danymi pomiarowymi ze stacji. Wyniki liczbowe przedstawiono w tabeli 3, zaś wykres na rysunku 11.

Tabela 1: Uzyskane wyniki dla eksperymentu 1

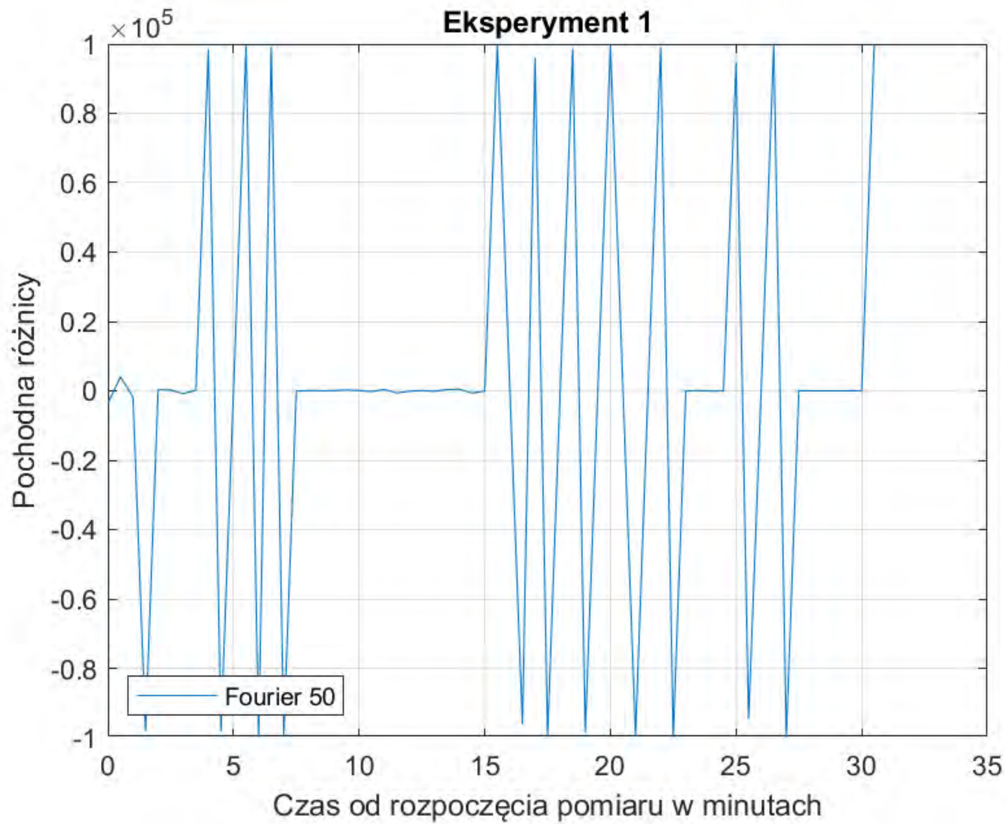
	Fourier Transform 50	Fourier Transform 100	Binary Threshold 50	Binary Threshold 100	Kernel 50	Kernel 100	Stacja pomiarowa
Wartość minimalna	0.070162	0.00436343	0.0185931	0.00214039	0.0172338	0.00280464	1.031000000000000e+02
Wartość maksymalna	99.9951	0.169855	0.235479	0.100555	0.232644	0.104159	5.614000000000000e+02
Mediana	99.7561	0.0419097	0.0794772	0.0215023	0.0732652	0.0216745	1.207000000e+02
Średnia	73.050146084127	0.049287998730159	0.087064885714286	0.026374854603175	0.083030523809524	0.02728346984127	1.372825396825397e+02
Moda	0.070162	0.00436343	0.0678773	0.00214039	0.0172338	0.00280464	1.048000000000000e+02
Odchylenie standardowe	44.210368106011394	0.033786608432889	0.03896402119592	0.019374828849719	0.039534502975415	0.019499110720446	63.075821091359620
Wariancja	1.95450e+03	0.0011449397	0.0015947756	3.75383950e-04	0.0016299513	3.80212145e-04	3.978549208e+03

Tabela 2: Uzyskane wyniki dla eksperymentu 2

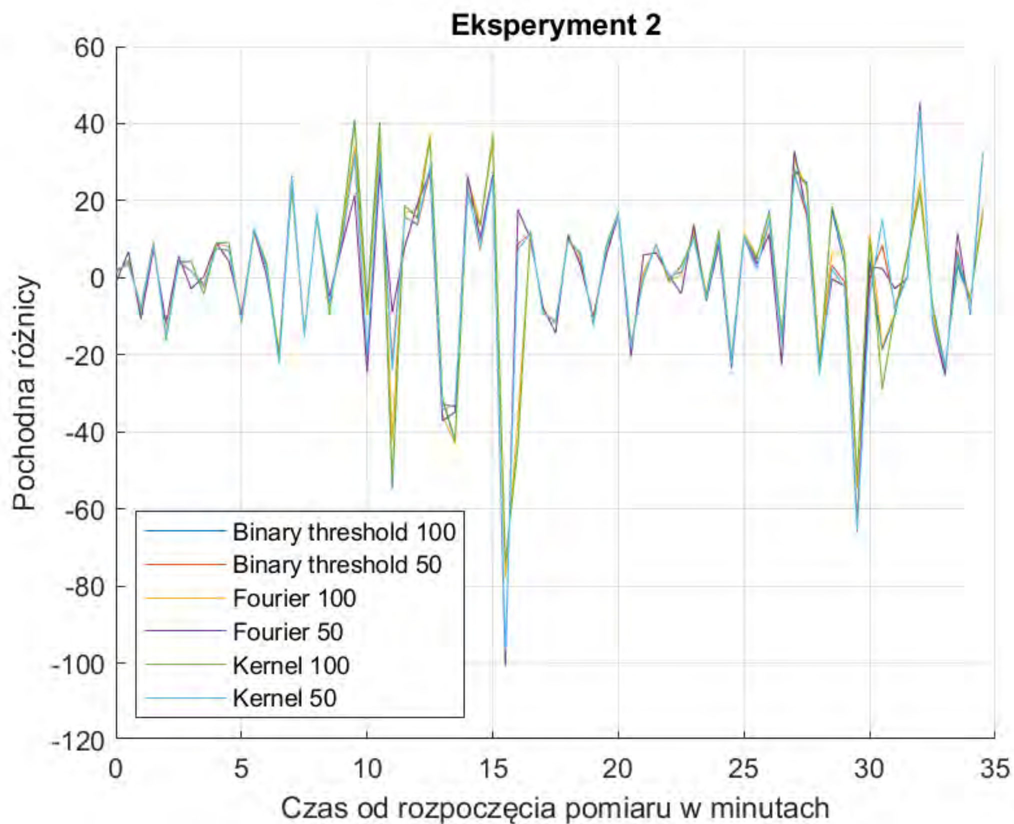
	Fourier Transform 50	Fourier Transform 100	Binary Threshold 50	Binary Threshold 100	Kernel 50	Kernel 100	Stacja pomiarowa
Wartość minimalna	0.00375579	4.398150000000000e-04	0.00103944	3.075270000000000e-05	0.00108864	1.599140000000000e-04	40.7
Wartość maksymalna	0.15886	0.0927315	0.151131000000000	0.0861874	0.156611	0.0761252	179
Mediana	0.017037	0.0060706	0.012215	0.00320443	0.0125901	0.00332129	70.2
Średnia	0.029749526197183	0.010588238802817	0.022115329577465	0.006959891676056	0.022063781830986	0.006845631098592	83.705633802816900
Moda	0.00869213	4.398150000000000e-04	0.00656262	0.00388099	0.00108864	2.398710000000000e-04	52.3
Odchylenie standardowe	0.028174348763095	0.014347697451317	0.026007489844578	0.012390746811494	0.026355425823182	0.011145386684290	35.702772550697524
Wariancja	7.937939282245211e-04	2.058564221545275e-04	6.763895280158524e-04	1.535306065465514e-04	6.946084703212352e-04	1.242196443423580e-04	1.274687967806841e+03



Rysunek 8: Wykres przedstawiający pochodną różnicy wartości pomiędzy pomiarami z aplikacji i stacji pomiarowej wykonanymi w ramach eksperymentu nr 1



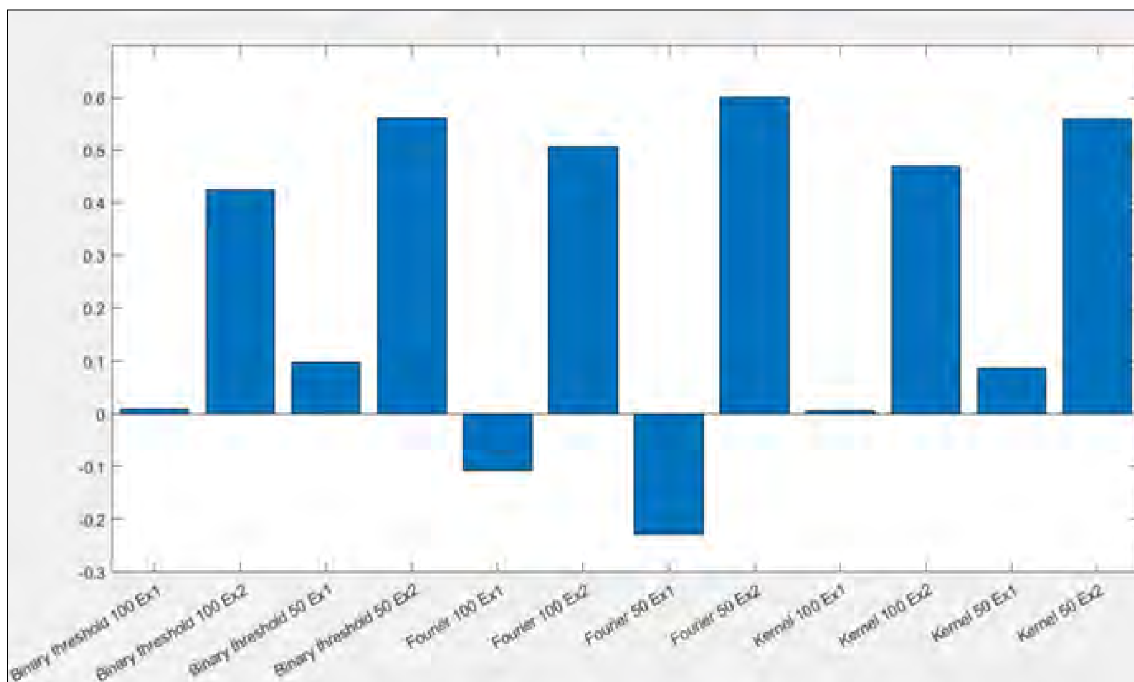
Rysunek 9: Wykres przedstawiający pochodną różnicy wartości pomiędzy pomiarem metody Fourier 50 i stacją pomiarową w ramach eksperymentu 1



Rysunek 10: Wykres przedstawiający pochodną różnicy wartości pomiędzy pomiarami z aplikacji i stacji pomiarowej

Tabela 3: Wyniki korelacji Kendalla

Nazwa metody	Wartość korelacji Kendalla
Binary threshold 100 Ex1	0,0097
Binary threshold 100 Ex2	0,4254
Binary threshold 50 Ex1	0,099
Binary threshold 50 Ex2	0,5615
Fourier 100 Ex1	-0,1061
Fourier 100 Ex2	0,5062
Fourier 50 Ex1	-0,2281
Fourier 50 Ex2	0,601
Kernel 100 Ex1	0,0067
Kernel 100 Ex2	0,4713
Kernel 50 Ex1	0,0866



Rysunek 11: Wartości korelacji Kendalla wyników pomiarów poszczególnych metod z wynikami ze stacji pomiarowej

6. Wnioski

W niniejszym artykule poruszono zagadnienie popularnego tematu jakim jest zanieczyszczenie powietrza. Przed badaniami autorzy spodziewali się, że metoda Binary threshold z wartością progowania wynoszącą 50 pozwala uzyskać wyniki najbardziej zbliżone z wynikami ze stacji pomiarowej.

Korzystając z powyższych informacji, o wykresach przedstawionych na rysunkach 8,9 oraz 10 można powiedzieć, że:

- wyniki pomiarów każdej z metod posiadają duży rozrzut wartości tuż po rozpoczęciu badań, w przedziale ok. 0 - 2 min.;
- wykresy w przeważającej mierze oscylują wokół osi OX w zakresie wartości od -100 do 100;
- wykresy pochodnych metod Kernel 100 i Binary threshold 100 przebiegają niemal identycznie;
- wykresy pochodnych metod Kernel 50 i Binary threshold 50 są do siebie bardzo zbliżone;
- wykresy Kernel 50 i Binary threshold 50 osiągają najmniejsze wartości skrajne.

Powyższe stwierdzenia potwierdzają wnioski, że wyniki z metody Binary threshold 50 najbardziej zbliżone są do wyników ze stacji pomiarowej.

Analizując dane liczbowe oraz wykres słupkowy, przedstawiony na rysunku 11, można stwierdzić, że:

- wyniki uzyskane z wykorzystaniem metod Binary threshold 100 i Kernel 100 w eksperymencie 1 są najmniej skorelowane z wynikami ze stacji;
- wyniki metody Fourier 50 w eksperymencie 2 są najbardziej zbliżone do wyników ze stacji;
- metody Fourier 100 Ex1 i Fourier 50 w eksperymencie 1 są najbardziej przeciwstawne do wyników ze stacji;

- metody Binary threshold 50 i Kernel 50 w eksperymencie 2 wykazały się niewiele mniejszą korelacją od metody Fourier 50;
- metoda Binary threshold 50 okazała się najdokładniejsza, ze względu na największą sumaryczną wartość korelacji z obu eksperymentów;
- średnia arytmetyczna korelacji dla eksperymentu 2 wynosi 0,52, zaś dla eksperymentu 1 wynosi -0,02;
- metody z wartością progowania 50 lepiej sprawdziły się w eksperymencie 2;
- metody Fourier 50 i Fourier 100 posiadają największą rozbieżność pomiędzy eksperymentem 1 i 2.
- Korelacja Kendalla pozwala jednoznacznie stwierdzić, czy prezentowane metody pozwoliły uzyskać wyniki zbliżone do tych ze stacji pomiarowej. Metoda Fourier 50 nie może być uznana za najlepszą, mimo uzyskania największej wartości korelacji w eksperymencie 2, ze względu na przekłamania w eksperymencie 1.

Analiza powyższych wniosków wykazała, że najdokładniejszą metodą jest Binary threshold 50, ze względu na najwyższą sumaryczną wartość korelacji z obu eksperymentów. Tym samym hipoteza postawiona przez autorów została potwierdzona.

Literatura

[1] A. Kaehler, G. Bradski: OpenCV 3. Kompute-rowe rozpoznawanie obrazu w C++ przy użyciu biblioteki OpenCV.

[2] J. Chwastowski, K. Korcyl (red.): Wybrane zagadnienia przetwarzania obrazu, Kraków Wydawnictwo PK 2014.

[3] W. K. Pratt: Digital Image Processing, PIKS Inside, Willey, 2001.

[4] R. Tadeusiewicz, P. Korohoda: Komputerowa analiza i przetwarzanie obrazów, Wydawnictwo Fundacji Postępu Telekomunikacji, Kraków 1997.

[5] J. Miałdun: Wykorzystanie transformacji fouriera do filtracji szumu informacyjnego z obrazów fotolotniczych, Archiwum Fotogrametrii, Karto-grafii i Teledetekcji 2012.

[6] W. Malina, S. Ablameyko, W. Pawlak: Podstawy cyfrowego przetwarzania obrazów, Akademicka Oficyna Wydawnicza EXIT, Warszawa, 2002.

[7] Dokumentacja OpenCV, https://docs.opencv.org/master/d9/df8/tutorial_root.html [11.2019].

[8] Wykład „Image filtering" http://mstrzel.eletel.p.lodz.pl/mstrzel/pattern_rec/filtering .pdf [04.2020].