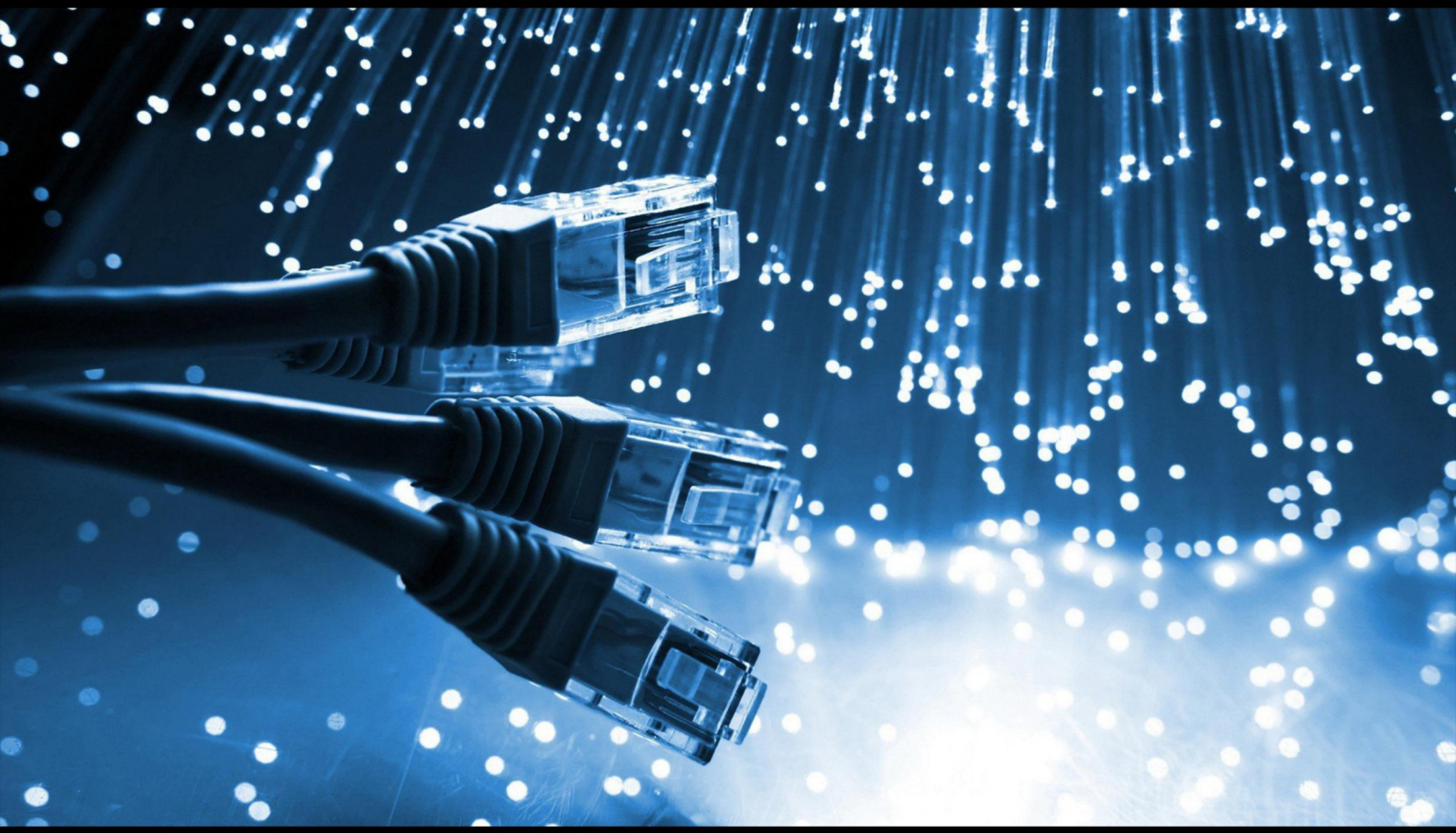
JCSI

Journal of Computer Sciences Institute

Volume 19/2021



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

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

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

Recenzenci numeru:

dr inż. Kamil Żyła
dr Mariusz Dzieńkowski
dr inż. Elżbieta Miłosz
dr inż. Marcin Badurowicz
dr inż. Dariusz Gutek
dr inż. Maria Skublewska-Paszkowska
dr inż. Marek Miłosz, prof. PL
dr inż. Jacek ęsik
dr inż. Grzegorz Kozieł
dr inż. Maciej Pańczyk
dr hab. Paweł Karczmarek, prof. PL
dr inż. Piotr Muryjas
dr inż. Jakub Smołka

Skład komputerowy:
Anna Sałamacha
e-mail: a,salamacha@pollub.pl

Projekt okładki:
Marta Zbańska

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

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

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

Reviewers:

Kamil Żyła
Mariusz Dzieńkowski
Elżbieta Miłosz
Marcin Badurowicz
Dariusz Gutek
Maria Skublewska-Paszkowska
Marek Miłosz
Jacek ęsik
Grzegorz Kozieł
Maciej Pańczyk
Paweł Karczmarek
Piotr Muryjas
Jakub Smołka

Computer typesetting:
Anna Sałamacha
e-mail: a.salamacha@pollub.pl

Cover design:
Marta Zbańska

ISSN 2544-0764

Spis treści

Contents

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 67–74
Received: 13 January 2021
Accepted: 21 January 2021

Comparison of WebSocket and HTTP protocol performance

Porównanie wydajności protokołu WebSocket i HTTP

Wojciech Paweł Łasocha*, Marcin Badurowicz

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

Abstract

The purpose of the author of this article is to compare the performance of the WebSocket and HTTP protocols. For this purpose, LAN equipment and a self-made testing application were used. It was used to measure the time of sending and downloading/receiving 100-character texts in a specified number of copies, considering the speed of laptops and web browsers. The conducted research shows that when transmitting more than 100 copies of data using the WebSocket protocol (compared to HTTP), performance can be increased by several hundred percent. In addition, it has been proven that adding excess overhead to HTTP requests can slow it down considerably. In contrast, TLS encryption has little effect on the speed of both protocols. It was concluded that the WebSocket protocol is good for sending hundreds or thousands of small serving of data per second, because for a smaller number of them, a simple HTTP polling is absolutely enough.

Keywords: websocket protocol; http protocol; protocols performance comparison

Streszczenie

Celem autora tego artykułu jest porównanie wydajności protokołu WebSocket i HTTP. W tym celu wykorzystano sprzęt pracujący w sieci LAN oraz samodzielnie wykonaną aplikację testującą. Za jej pomocą zmierzono czas wysyłania oraz pobierania/odbierania 100-znakowych tekstów w określonej liczbie kopii z uwzględnieniem szybkości laptopów i przeglądarek WWW. Z przeprowadzonych badań wynika, że przy transmisji powyżej 100 kopii danych za pomocą protokołu WebSocket (w porównaniu do HTTP) można uzyskać wzrost wydajności o kilkaset procent. Ponadto udowodniono, że dodawanie nadmiarowych narzutów do żądań HTTP może go bardzo spowalniać. Natomiast szyfrowanie TLS ma znikomy wpływ na szybkość obu protokołów. Wywnioskowano, że protokół WebSocket dobrze sprawdzi się w przesyłaniu setek lub tysięcy małych porcji danych na sekundę, gdyż w przypadku mniejszej ich liczby w zupełności wystarczy zwykłe odpytywanie HTTP.

Słowa kluczowe: protokół websocket; protokół http; porównanie wydajności protokołów

*Corresponding author

Email address: wojciech.lasocha@pollub.edu.pl (W. P. Łasocha)

1. Wstęp

Protokół HTTP to jeden z najważniejszych elementów sieci WWW [1]. Umożliwia wymianę danych pomiędzy klientem a serwerem zgodnie z modelem żądanie-odpowiedź [2]. Według artykułu [3] początkowo internautom to w zupełności wystarczało, gdyż byli skoncentrowani na wyszukiwaniu i pobieraniu informacji.

Wraz z rozwojem technologii webowych pojawiała się potrzeba większej interakcji użytkowników na stronach internetowych. Ważnym momentem w rozwoju sieci Web było wprowadzenie techniki AJAX, która pozwalała na asynchroniczną komunikację przeglądarki internetowej z serwerem WWW [4]. Jednak technologia ta nadal bazowała na modelu żądanie-odpowiedź. Programiści aplikacji internetowych próbowali ją wykorzystać do emulacji komunikacji w czasie rzeczywistym, poprzez zastosowanie długiego odpytywania HTTP (ang. *Long Polling HTTP*). Mimo, że metoda ta dalej jest używana w całym Internecie, nadal nie rozwiązuje tego problemu, że dane mogą zostać przesłane jedynie podczas zainicjowania żądania przez przeglądarkę – w tym wypadku nie ma możliwości bezpośredniego wysłania informacji przez serwer do klienta bez jego wcześniejszego żądania o nie. Ponadto, technika ta może powodować poważne problemy, ze względu na generowane duże obciążenia zarówno przeglądarki klienta, jak i serwera [5].

Jak wyjaśniają artykuły [6-7], przełomowym momentem było wprowadzenie w 2008 r. mechanizmu WebSocket. Odzwierciedla on sieć WWW czasu rzeczywistego (ang. *Real-Time Web*), która według artykułu [8] umożliwia użytkownikom otrzymywanie informacji natychmiast po ich opublikowaniu przez ich autorów, zamiast wymagać od nich okresowego sprawdzania źródła pod kątem aktualizacji. Znakomicie zmniejsza to obciążenie sieci, serwera i samej przeglądarki. Unika się dodatkowych komunikatów przesyłanych i przetwarzanych po obu stronach tylko dla utrzymania stałej komunikacji oraz umożliwia serwerowi przesyłanie do przeglądarki w czasie rzeczywistym nowych danych, kiedy tylko pojawią się na serwerze. Rozwiązania te mają korzystny wpływ na wydajność.

Według autora artykułu [5] wykonanie pojedynczego żądania na połączenie z wykorzystaniem biblioteki WebSocket Socket.IO jest o połowę wolniejsze niż w przypadku protokołu HTTP, ponieważ połączenie musi zostać najpierw ustanowione. Natomiast wykonanie 50 żądań w ramach tego samego połączenia poprzez WebSocket jest o połowę szybsze niż używając do tego

celu HTTP. W tym samym artykule można również przeczytać, że HTTP osiąga szczytową przepustowość przy około ~950 żądaniach na sekundę, podczas gdy Socket.IO obsługuje około ~3900 żądań na sekundę.

Badania przeprowadzone przez autorów artykułu [9] dowodzą, że protokół WebSocket jest nieco szybszy niż HTTP. Wysyłanie na serwer 40000 bajtów danych w sieci WAN za pomocą techniki AJAX trwało około 39 ms, natomiast protokół WebSocket zrobił to samo w około 29 ms. Wysyłanie tych samych danych w sieci LAN wyszło również korzystniej dla tej technologii i było o 4 ms szybsze.

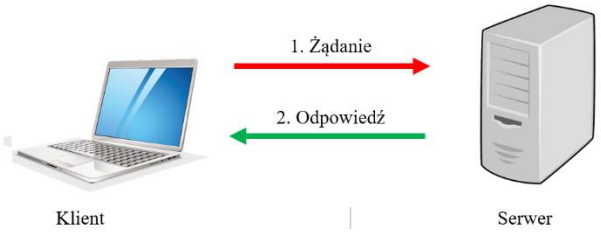
Natomiast autor badań opisanych w artykule [10] udowodnił, że zastosowanie bardzo szybkich serwerów (w szczególności ich procesorów) pozwala na obsługę 5 milionów jednocześnie podłączonych klientów z wykorzystaniem protokołu WebSocket.

Wyniki badań w powyższych artykułach [5, 9-10] motywują do przeprowadzenia własnych badań nad wydajnością obu protokołów. Autor niniejszego artykułu skoncentrował się na zmierzeniu szybkości częstego (wielokrotnego) przesyłania małych porcji danych, transmitowanych w setkach, a nawet tysiącach komunikatów na sekundę. Dlatego przedstawione wyniki opisują stan rzeczy, jak dany protokół sprawdzi się w takich zastosowaniach jak: gry online, pokoje czatowe, czy pobieranie w czasie rzeczywistym notowań giełdowych. Rozważanie przypadku przesyłania plików o dużych rozmiarach jest zbędne, gdyż protokół WebSocket po prostu do tego się nie nadaje ze względu na fragmentację danych na części po kilkadziesiąt kilobajtów każda. Dlatego przesyłanie pliku o rozmiarze 1 MB może trwać tyle samo czasu dla HTTP, jak i dla WebSocket.

2. Przedmiot badań

2.1. Charakterystyka protokołu HTTP

Protokół HTTP tworzy kanał wymiany danych pomiędzy dwoma końcami komunikacji, którymi najczęściej są klient (na przykład przeglądarka internetowa) i serwer (na przykład aplikacja działająca na komputerze hostującym witrynę internetową). Protokół ten bazuje na modelu żądanie-odpowiedź, tzn. klient wysyła do serwera komunikat żądania, po czym serwer zwraca komunikat odpowiedzi zawierający plik HTML lub inny zasób. Odpowiedź zawiera informację o statusie ukończenia wykonanego żądania i może również zawierać żądaną treść w ciele wiadomości (na przykład strukturę pliku HTML). Graficzny model wymiany danych pomiędzy klientem a serwerem z wykorzystaniem protokołu HTTP został przedstawiony na rysunku numer 1.



Rysunek 1: Model wymiany danych pomiędzy klientem a serwerem z wykorzystaniem protokołu http.

Do zadań klienta HTTP zalicza się: wysyłanie żądania – inicjowanie połączenia HTTP (poprzez URL), pobieranie zasobu z serwera, prezentacja danych (budowa strony internetowej), interakcja z użytkownikiem, buforowanie danych z serwera oraz kontrola spójności z serwerem, szyfrowanie-deszyfrowanie (HTTPS).

Natomiast do zadań serwera HTTP zalicza się: obsługa żądań HTTP – uruchamianie skryptów i wysyłanie odpowiedzi, rejestracja i kolejkowanie żądań, uwierzytelnianie i kontrola dostępu, szyfrowanie-deszyfrowanie (HTTPS), wybór wersji językowej wysyłanych dokumentów.

HTTP to protokół aplikacji, znajdujący się w warstwie 7. modelu OSI/ISO. Jest oparty na TCP (protokół warstwy transportowej) i domyślnie wykorzystuje port 80 (dla połączeń nieszyfrowanych) lub 443 (dla połączeń szyfrowanych). Chociaż, że może przesyłać nie tylko dane tekstowe, ale również binarne, na ogół uważany jest za tekstowy, ponieważ używa znakowych poleceń i komunikatów. Ważną jego cechą jest to, że zalicza się do protokołów bezstanowych, ponieważ nie zachowuje żadnych informacji o poprzednich transakcjach z klientem (po zakończeniu transakcji wszystko zostaje „zapominane"). Zalety takiego rozwiązania to lepsze wykorzystanie zasobów (obiekt po stronie serwera obsługujący żądanie po wysłaniu danych może rozpocząć obsługę kolejnego żądania) oraz prostota migracji ruchu na inne serwery. Znacznie to zmniejsza obciążenie serwera, jednak jest utrudnieniem w sytuacji, gdy na przykład trzeba zapamiętać konkretny stan dla użytkownika, który wcześniej już łączył się z serwerem. Programiści aplikacji internetowych zwykle rozwiązują ten problem poprzez wykorzystanie mechanizmu ciasteczek lub sesji po stronie serwera, czy wprowadzając ukryte parametry (w formularzu lub adresie URL).

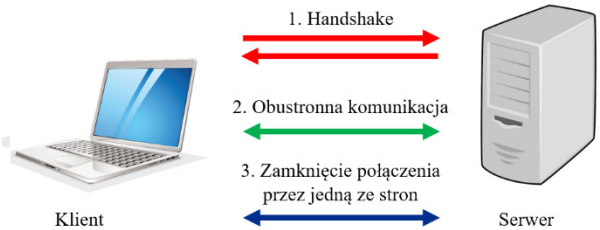
2.2. Charakterystyka protokołu WebSocket

Jak można przeczytać w artykule [6] WebSocket to komputerowy protokół komunikacyjny, zapewniający kanały komunikacji za pośrednictwem jednego połączenia TCP pomiędzy klientem (na przykład przeglądarką internetową) a serwerem. Istotne jest to, że zapewnia wymianę danych w czasie rzeczywistym, tzn. serwer może wysłać do klienta wiadomość bez wcześniejszego żądania klienta o nią. W ten sposób może odbywać się dwukierunkowa trwająca rozmowa między klientem a serwerem, utrzymując połączenie otwarte. W przeciwieństwie do protokołu HTTP, protokół WebSocket umożliwia przesyłanie danych w trybie pełnego dupleksu, czyli może jednocześnie wysyłać i odbierać dane. Protokół ten ma znacznie mniejsze narzuty i nie wysyła zbędnych komunikatów jak w przypadku odpytywania HTTP (ang. *HTTP Polling*), przez co zmniejszają się opóźnienia w komunikacji, obciążenie sieci, serwera oraz samej przeglądarki internetowej, co ma korzystny wpływ na wydajność.

Protokół WebSocket jest ogromnym krokiem naprzód, pozwalającym na tworzenie aplikacji czasu rzeczywistego opartego na zdarzeniach. Można rzec, że WebSocket daje możliwości, które w przypadku proto-

kołu HTTP, były praktycznie nieosiągalne. Dzięki niemu klient nie musi prosić serwera o aktualne dane, lecz otrzyma je automatycznie w chwili ich pojawienia się na serwerze. Możliwości jego zastosowania są ogromne, na przykład obserwowanie cen akcji, informacji prasowych, sprzedaży biletów, natężenia ruchu, odczytów z urządzeń medycznych, na żywo. Należy jednak wspomnieć, że protokół HTTP przeznaczony jest do przesyłania danych tekstowych i binarnych, natomiast protokół WebSocket służy do szybkiej wymiany danych w formacie JSON (ang. *JavaScript Object Notation*). Przeglądarki internetowe potrafią wyświetlić stronę WWW na podstawie kodu HTML. Jednak obecnie nie ma takiego standardu, który mógłby zrobić to samo analizując dane JSON. Z tego względu można stwierdzić, że protokół WebSocket stanowi niejako dodatek do obecnej sieci Web, a nie jest następcą tradycyjnego protokołu HTTP.

Zanim protokół WebSocket rozpocznie wymianę danych pomiędzy klientem a serwerem, klient wysyła do serwera wiadomość w postaci zwykłego żądania HTTP z prośbą o zmianę protokołu i nawiązanie stałego połączenia – to tzw. uścisk dłoni (ang. *Handshake*). Jeśli serwer zaakceptuje ją, wysyła do klienta wiadomość zwrotną z pozytywną odpowiedzią. W tym momencie obie strony komunikacji mogą wysyłać do siebie dane. Oczywiście, w każdym momencie połączenie może być jawnie zamknięte. Odłączenie klienta lub serwera zawsze zostanie wykryte i o fakcie tym może zostać poinformowany użytkownik. Graficzny model wymiany danych pomiędzy klientem a serwerem z wykorzystaniem protokołu WebSocket został przedstawiony na rysunku numer 2.



Rysunek 2: Model wymiany danych pomiędzy klientem a serwerem z wykorzystaniem protokołu WebSocket.

WebSocket, podobnie jak protokół HTTP, to protokół aplikacji znajdujący się w warstwie 7. modelu OSI/ISO, oparty na TCP (protokół warstwy transportowej). WebSocket, choć różni się od HTTP, jest zaprojektowany do pracy przez porty HTTP 80 (dla połączeń nieszyfrowanych) i 443 (dla połączeń szyfrowanych), dlatego jest z nim zgodny. Jest zdolny do pracy przez istniejącą infrastrukturę przystosowaną do transmisji HTTP, czyli przez już skonfigurowane do tego celu serwery pośredniczące (ang. *Proxy*) oraz przechodził przez ustawienia zapory sieciowej, jeżeli tylko dopuszczany jest ruch HTTP.

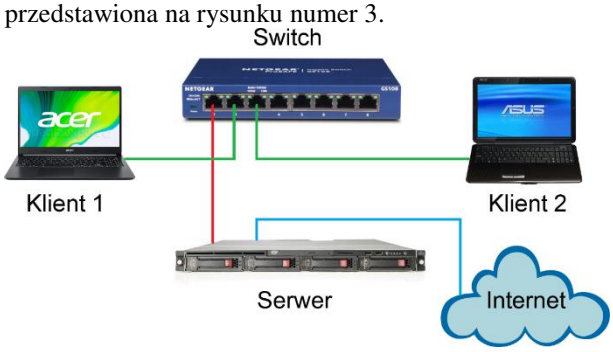
3. Metodyka badań

Do wykonania badań nad wydajnością protokołów HTTP i WebSocket autor niniejszego artykułu zdecydował się zbudować własne środowisko badawcze. System ten składa się z następujących elementów:

- Infrastruktura – fizyczna część systemu składająca się z serwera, przełącznika niezarządzalnego (ang. *Switch*) i dwóch komputerów klienckich, połączonych kablem w sieć lokalną LAN (ang. *Local Area Network*) za pomocą skrętki czteroparowej.
- Bazowe oprogramowanie klienckie – dwa odrębne środowiska zainstalowane na komputerach klienckich, w skład których zawierają się systemy operacyjne Microsoft Windows 10 i Linux Fedora 33 Workstation oraz przeglądarki internetowe Google Chrome, Mozilla Firefox, Microsoft Edge.
- Bazowe oprogramowanie serwerowe – środowisko zainstalowane na serwerze, oparte na systemie operacyjnym Debian 10 (Buster), platformie Node.js oraz bibliotek Express.js (ułatwiającej zbudowanie serwera WWW wykorzystującego protokół HTTP) i Socket.IO (ułatwiającej tworzenie aplikacji komunikujących się za pomocą protokołu WebSocket).
- Aplikacja testująca – samodzielnie wykonana internetowa aplikacja webowa (uruchamiana w przeglądarce internetowej) oparta na platformie Node.js dzieląca się na część serwerową i kliencką, mierząca czas transmisji danych o określonych parametrach za pomocą protokołów HTTP i WebSocket.

3.1. Wykorzystana infrastruktura

Struktura zbudowanej przewodowej sieci lokalnej wraz z połączeniem ze sobą wszystkich urządzeń została przedstawiona na rysunku numer 3.



Rysunek 3: Struktura zbudowanej przewodowej sieci lokalnej wykorzystanej w badaniach.

Centralnym punktem wiążącym sieć jest przełącznik. Do jego pierwszego portu podłączono serwer, natomiast do drugiego i trzeciego portu podłączono komputery klienckie. Drugi interfejs sieciowy serwera łączy się z Internetem. Poszczególne wykorzystane urządzenia charakteryzują się następującymi parametrami:

- Switch Netgear GS108GE
  - 8 portów Gigabit Ethernet
  - Niezarządzalny
  - Brak PoE (ang. *Power over Ethernet*)
- Serwer HP ProLiant DL320 G5
  - System operacyjny Linux Debian 10 (Buster)
  - Procesor Quad-Core Intel Xeon 2,4 GHz
  - Pamięć operacyjna DDR2-800 6 GB
  - 4 dyski twarde SAS 15000 po 72 GB każdy
  - 2 interfejsy sieciowe Gigabit Ethernet

- Laptop 1 Acer Aspire 5
  - System operacyjny Microsoft Windows 10
  - Procesor Intel Core i5-7200U 2,5 GHz
  - Pamięć operacyjna DDR4-2133 4 GB
  - Dysk twardy SATA III 5400 1000 GB
  - Interfejs sieciowy Gigabit Ethernet
- Laptop 2 Asus K50IN
  - System operacyjny Linux Fedora 33 Workstation
  - Procesor Intel Pentium Dual Core T4200 2 GHz
  - Pamięć operacyjna DDR2-800 4 GB
  - Dysk twardy SATA I 5400 60 GB
  - Interfejs sieciowy Gigabit Ethernet
- Okablowanie
  - Skrętka czteroparowa
  - Kategoria 6

Serwer posiada cztery dyski twarde połączone w macierz dyskową RAID 0. Z tego względu można z góry wykluczyć, że wąskim gardłem systemu był zapis i odczyt danych wysyłanych do serwera bądź z niego pobieranych. Transmisję danych mógł jedynie spowalniać komputer kliencki Asus K50IN (ponieważ jest to stary sprzęt), co oczywiście było uwzględnione w testach. Aczkolwiek, zastosowanie Gigabitowych interfejsów sieciowych w każdym z urządzeń, powinno dać możliwie jak najbardziej realistyczne wyniki przeprowadzanych badań.

3.2. Instalacja i konfiguracja oprogramowania

Do wykonania badań, autor niniejszego artykułu wykorzystał odpowiednie oprogramowanie – zarówno dla komputerów klienckich, jak i serwera.

Na nowszym i szybszym laptopie Acer Aspire 5 został wykorzystany system operacyjny Microsoft Windows 10, natomiast na starszym i wolniejszym Asus K50IN – Linux Fedora 33 Workstation. Do tego pierwszego domyśle jest dołączana przeglądarka internetowa Microsoft Edge, a do drugiego Mozilla Firefox. Jednak w testach została użyta również przeglądarka WWW Google Chrome, która nie była dołączona do żadnego OS-u. Dlatego brakujące programy wykorzystane w testach musiały być ręcznie doinstalowane.

Najważniejszym oprogramowaniem wykorzystanym na serwerze HP ProLiant DL320 G5 był system operacyjny Linux Debian 10 (Buster), który z reguły przeznaczony jest do zastosowań serwerowych. Po jego zainstalowaniu, interfejsy sieciowe serwera zostały tak skonfigurowane, aby jedna karta sieciowa miała dostęp do Internetu, druga zaś była rozpoznawalna w sieci lokalnej, do której były podłączone komputery klienckie. Aby zaoszczędzić sobie pracy w konfiguracji sieci na klientach, na serwerze został skonfigurowany serwer DHCP. Następnie została zainstalowana platforma Node.js wraz z menadżerem pakietów npm oraz dodatkowo narzędzie OpenSSL. Te ostatnie służy do wygenerowania certyfikatu CSR i klucza prywatnego, potrzebnych do zestawiania połączeń szyfrowanych SSL/TLS (HTTPS oraz WebSocket Secure).

Po wykonaniu powyższych czynności, wszystkie urządzenia (a w pierwszej kolejności serwer) zostały ponownie uruchomione.

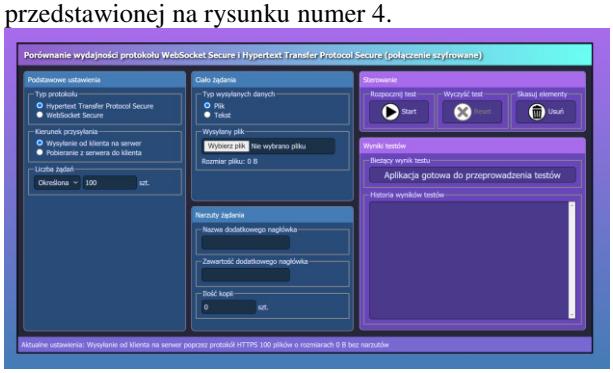
3.3. Aplikacja testująca

W celu zainstalowania aplikacji testującej na serwerze należało skopiować rekurencyjnie jej główny katalog na serwer, następnie przejść do niego i wydać polecenie `npm install`. Potem konieczne było zmodyfikowanie zawartości pliku `settings.txt` znajdującego się w katalogu `public` na adres IP interfejsu sieciowego serwera pod jakim ma być rozpoznawalny w sieci LAN, do której były podłączone komputery klienckie.

W celu uruchomienia aplikacji testującej na serwerze należało przejść do jej głównego katalogu, a następnie w zależności czy do połączeń ma być wykorzystywane szyfrowanie TLS, czy też nie, wykonano jedno z dwóch poniższych poleceń:

- Dla połączeń nieszyfrowanych:
  `node index.js --encryption=no`
- Natomiast dla połączeń szyfrowanych:
  `node index.js --encryption=yes`

Od tego momentu na wszystkich podłączonych klientach była dostępna aplikacja (od strony klienckiej) w przeglądarkach internetowych pod adresem URL równym adresowi IP serwera, poprzedzonym prefiksem używanego protokołu (`http://` lub `https://`), w zależności od tego, czy zostało zastosowane szyfrowanie w przesyłaniu danych. Po otwarciu strony internetowej ukarze się główny interfejs aplikacji testującej, przedstawionej na rysunku numer 4.



Rysunek 4: Główny interfejs użytkownika aplikacji testującej otwartej w przeglądarce internetowej.

Interfejs użytkownika aplikacji testującej przedstawionej na powyższym rysunku składa się z dwóch najważniejszych części. Po jego lewej stronie (ramki w kolorze niebieskim) użytkownik może dowolnie dostosować ustawienia testu, który ma zostać przeprowadzony. Zmiany tychże ustawień są na bieżąco wyświetlane w pasku stanu (znajdującego się na dole aplikacji testującej), który zwięźle i kompleksowo opisuje aktualnie wybrane ustawienia. Po prawej stronie interfejsu aplikacji (ramki w kolorze fioletowym) użytkownik może sterować testem oraz analizować jego wyniki po ukończeniu badań. Wybór ustawień testu sprowadza się do typu protokołu (HTTP/HTTPS lub WebSocket/WebSocket Secure), kierunku przesyłania (wysyłanie na serwer, pobieranie lub odbieranie z serwera, czy transfer pomiędzy klientami), rodzaju przesyłanych danych (plik lub tekst) oraz liczby ich kopii, a także opcjonalnych narzutów (w przypadku protokołu HTTP/HTTPS).

Po skonfigurowaniu aplikacji testującej, można rozpocząć badanie klikając przycisk `Start`. Na ekranie serwera na bieżąco pojawiają się postępy w jego wykonaniu, jak dla przykładu pokazano na rysunku numer 5 – w tym przypadku wysyłanie od klienta na serwer poprzez szyfrowany protokół WSS (ang. *WebSocket Secure*) 10 tekstów o długościach 16 znaków każdy. Końcowy wynik wykonania operacji jest zawsze wyświetlany zarówno w aplikacji klienckiej, jak i serwerowej.

```
10:45:27,852 - Zlecono: Wysyłanie od klienta na serwer poprzez protokół WSS
10 tekstów o długościach 16 znaków
10:45:27,855 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 10,00%)
10:45:27,857 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 20,00%)
10:45:27,858 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 30,00%)
10:45:27,859 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 40,00%)
10:45:27,861 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 50,00%)
10:45:27,862 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 60,00%)
10:45:27,864 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 70,00%)
10:45:27,866 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 80,00%)
10:45:27,866 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 90,00%)
10:45:27,867 - Pomyślnie odebrano i zapisano tekst na dysku twardym serwera
(całkowity postęp: 100,00%)
10:45:27,867 - Pomyślnie ukończono wysyłanie tekstów od klienta na serwer (w
ysłano 10 tekstów w czasie 0,015 sekund)
```

Rysunek 5: Ekran konsoli Linuksa na serwerze po pomyślnie wykonanym teście.

Klikając przyciski `Reset` i `Start` można kontynuować badania i analizować wyniki, a w razie konieczności zmieniać parametry testów. Pamiętać również trzeba, aby przed każdym wysyłaniem danych na serwer usunąć z niego elementy, które były wysyłane do niego poprzednim razem, aby uniknąć kolizji wśród plików – można to zrobić klikając przycisk `Usuń`. Niestety, aby włączyć lub wyłączyć szyfrowanie w przesyłanych danych, należy od nowa uruchomić aplikację – zarówno od strony serwerowej, jak i klienckiej. Aplikację serwerową można zakończyć naciskając kombinację klawiszy `Ctrl + C`.

4. Wyniki przeprowadzonych badań

4.1. Wyjaśnienia wstępne

Badania przeprowadzone przez autora niniejszego artykułu polegają na testach przesyłania danych poprzez protokoły HTTP i WebSocket, a dokładniej zmierzeniu czasów wysyłania i pobierania (lub odbierania w przypadku WebSocket) krótkich tekstów. Każdy wspomniany tekst ma rozmiar około 100 bajtów, gdyż składa się z ciągu 100 znaków, a jego dokładna treść to:

`1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890`

Mimo, że powyższy tekst ma stały i z góry określony rozmiar, wykonane badania koncentrują się na porównaniu czasów przesyłania danych w różnych ilościach kopii, tzn. liczby kolejno następujących po sobie żądań-odpowiedzi (dla protokołu HTTP) lub wiadomości (dla protokołu WebSocket). W pomiarach uwzględniono przypadki dla 1, 3, 10, 30, 100, 300, 1000 i 3000 kopii, a wyniki przedstawiono w formie wykresów z dokładnością co do milisekundy (ms).

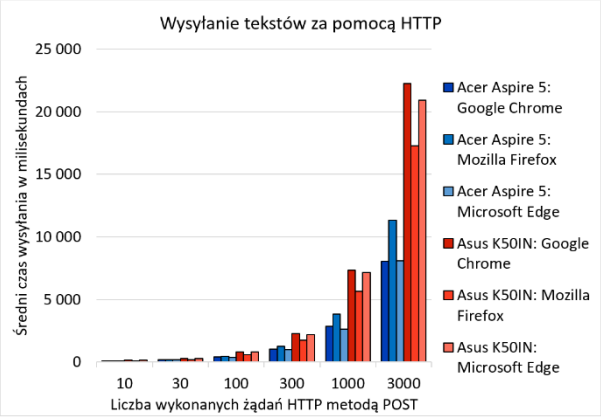
W badaniach każda próba została wykonana 10-krotnie, a na koniec obliczono z nich wartość średnią. W testach uwzględniono szybkość komputerów klienckich oraz wydajność zainstalowanych na nich systemów operacyjnych i przeglądarek internetowych, gdyż każdy test został przeprowadzony odrębnie dla danej platformy klienckiej.

Należy dodać, że badania zostały przeprowadzone w pełni domyślnych ustawieniach protokołów, stąd można uzyskać o wiele gorsze wyniki dla protokołu HTTP, jeśli zwiększy się jego narzuty (co zostanie udowodnione w dalszej części tego artykułu), lecz rzadko się to robi. Jednocześnie, protokół WebSocket potrafi czasami zaskakiwać, gdyż w badaniach do odebrania z serwera 10 wiadomości potrzebował około 90 ms, a zdarzało się, że robił to w 15 ms.

4.2. Wysyłanie i pobieranie poprzez HTTP

Pierwszym etapem niniejszych prac było zbadanie szybkości przesyłania 100-znakowych tekstów za pomocą protokołu HTTP.

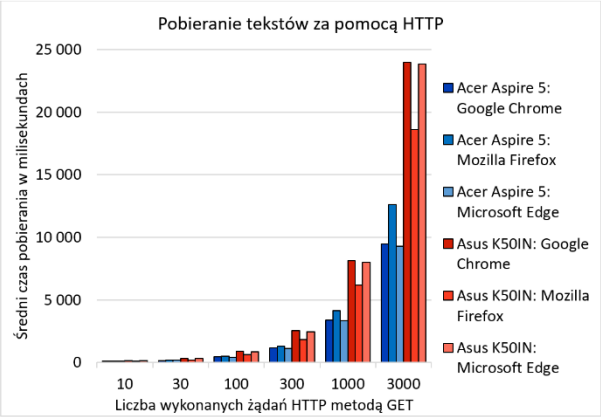
W pierwszym kroku zmierzono czas wysyłania danych z komputerów klienckich na serwer za pomocą metody POST. Następnie wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 6.



Rysunek 6: Średnie czasy wysyłania tekstów z komputerów klienckich na serwer za pomocą protokołu http.

Na powyższym wykresie wyraźnie widać, że wykonywanie większej liczby żądań HTTP (powyżej 30) metodą POST i przetwarzanie ich odpowiedzi, szybciej przebiegło w przypadku komputera klienckiego Acer Aspire 5 niż z Asus K50IN, gdyż jest to o wiele lat nowszy sprzęt (i co oczywiste szybszy). Co ciekawe, można również zauważyć dużą różnicę w wydajności przeglądarki internetowej Mozilla Firefox w zależności od platformy na jakiej pracuje (nie wiadomo jednak, czy zależy to od szybkości komputera, czy od systemu operacyjnego: Windows 10 lub Fedora 33). Można jednak przepuszczać, że najlepiej sprawdzają się przeglądarki WWW w systemach operacyjnych domyślnie do nich dołączane (Edge dla Windows oraz Firefox dla Fedory).

W drugim kroku zmierzono czas pobierania danych z serwera na komputery klienckie za pomocą metody GET. Następnie wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 7.



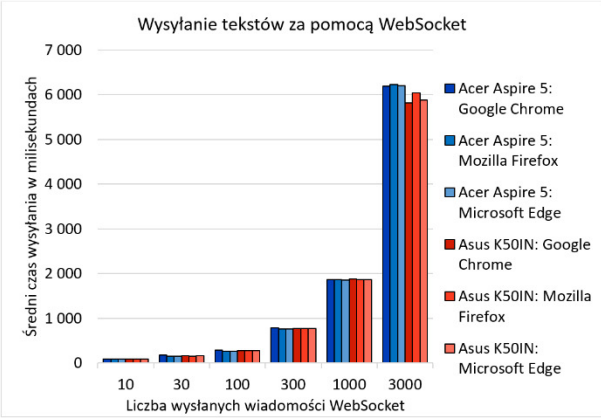
Rysunek 7: Średnie czasy pobierania tekstów z serwera przez komputery klienckie za pomocą protokołu http.

Na powyższym wykresie można zauważyć większą szybkość wykonywania żądań HTTP metodą GET w przypadku szybszego komputera klienckiego. Wystąpił również wzrost wydajności przeglądarki internetowej Mozilla Firefox (w porównaniu do pozostałych przeglądarek WWW) w zależności od laptopa i systemu operacyjnego na którym została użyta. Można stwierdzić, że program Microsoft Edge bez względu na szybkość sprzętu tak samo dobrze radził sobie z pobieraniem danych wykonując żądania HTTP GET jak Google Chrome.

4.3. Wysyłanie i odbieranie poprzez WebSocket

Drugim etapem niniejszych prac było zbadanie szybkości przesyłania 100-znakowych tekstów za pomocą protokołu WebSocket.

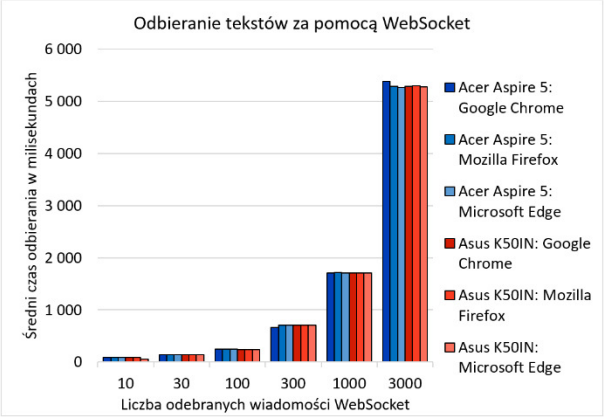
W pierwszym kroku zmierzono czas wysyłania danych z komputerów klienckich na serwer. Następnie wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 8.



Rysunek 8: Średnie czasy wysyłania tekstów z komputerów klienckich na serwer za pomocą protokołu WebSocket.

Najbardziej co wrzuca się w oczy to fakt, że wydajność protokołu WebSocket nie zależy od wykorzystywanego oprogramowania (systemów operacyjnych i przeglądarek internetowych), a nawet od szybkości komputerów klienckich. Z tego względu programiści aplikacji wykorzystujących protokół WebSocket nie muszą martwić się o użytkowników klienckich z powolnym sprzętem, czy rzadko używanym oprogramowaniem.

W drugim kroku zmierzono czas odbierania danych z serwera przez komputery klienckie. Następnie wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 9.

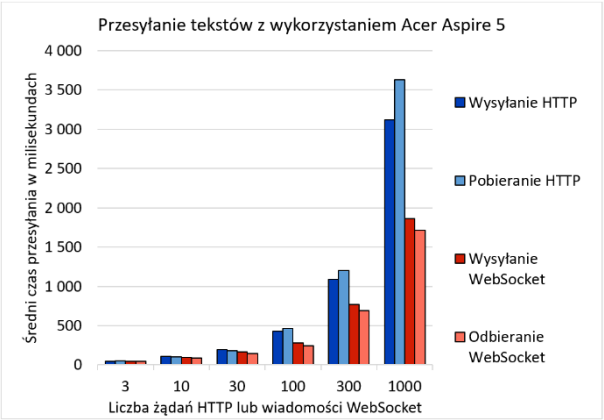


Rysunek 9: Średnie czasy odbierania tekstów z serwera przez komputery klienckie za pomocą protokołu WebSocket.

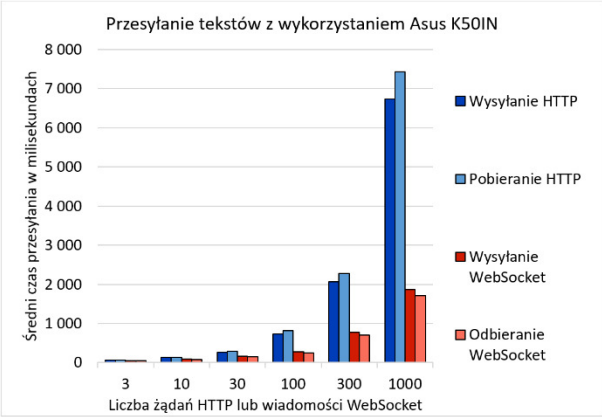
Na powyższym wykresie można dostrzec brak wpływu oprogramowania bazowego (systemów operacyjnych i przeglądarek internetowych) oraz sprzętu klienckiego na wydajność protokołu WebSocket.

4.4. Porównanie wydajności HTTP i WebSocket

W tej części artykułu podjęto próbę porównania wydajności protokołów HTTP i WebSocket poprzez analizę ich szybkości wysyłania i pobierania (odbierania w przypadku WebSocket) tekstów. Porównania dokonano na podstawie wyników badań uzyskanych w dwóch poprzednich podrozdziałach, kategoryzując je według sprzętu (szybszego i wolniejszego laptopa), na których zostały użyte, nie uwzględniając jednak przeglądarek internetowych (tzn. uśredniając uzyskane przez nie wyniki). W tym celu wykonano odpowiednie wykresy słupkowe przedstawione na rysunkach od numeru 10 do numeru 11.



Rysunek 10: Średnie czasy przesyłania tekstów pomiędzy komputerem klienckim Acer Aspire 5 a serwerem za pomocą protokołów HTTP i WebSocket.



Rysunek 11: Średnie czasy przesyłania tekstów pomiędzy komputerem klienckim Asus K50IN a serwerem za pomocą protokołów HTTP i WebSocket.

Najbardziej co wrzuca się w oczy, patrząc na powyższe wykresy to fakt, że protokół WebSocket jest znacznie szybszy od HTTP w przypadku wysyłania lub pobierania (odbierania) powyżej 100 kopii danych (żądań-odpowiedzi dla HTTP lub wiadomości dla WebSocket). Różnica ta w szczególności jest większa dla wolniejszego laptopa, gdyż jak wcześniej zauważono, wydajność protokołu WebSocket jest niezależna od platformy klienckiej. Tym samym można stwierdzić, że nie ma sensu implementować technologii WebSocket dla przesłania kilku lub kilkunastu wiadomości – dlatego w tym przypadku wystarczy zastosować zwykłe odpytywanie HTTP – tym bardziej, że powstają coraz szybsze komputery, które coraz bardziej sprawnie wykonują żądania HTTP i obsługują jego odpowiedzi. Tak więc protokół WebSocket świetnie sprawdzi się w aplikacjach sieciowych, gdzie wymagane jest przesyłanie dużej ilości (setek, a nawet tysięcy) małych porcji danych na sekundę.

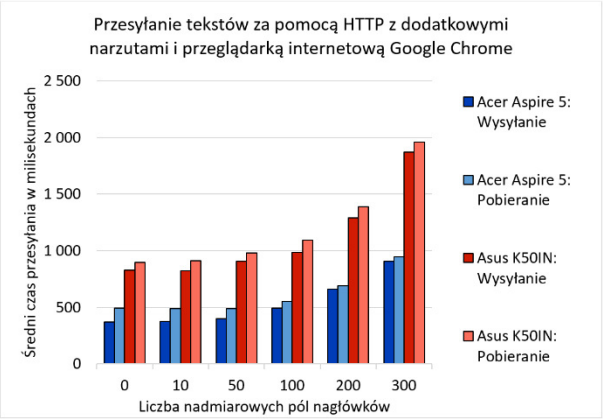
Najogólniej i powszechnie uważa się, że nie ma różnicy w wydajności pomiędzy pobieraniem danych z serwera za pomocą metody GET protokołu HTTP, a ich wysyłaniem za pomocą metody POST. Jednak jak wynika z badań, dla większej ilości zapytań HTTP, tj. 100 i więcej można zauważyć większą szybkość w przypadku tej drugiej metody. Wynika to z faktu, że komputery klienckie (a dokładniej przeglądarki internetowe) nie są w stanie na raz obsłużyć tak dużej ilości odpowiedzi HTTP GET z serwera.

W przypadku protokołu WebSocket odbieranie danych z serwera zawsze przebiegało trochę szybciej niż ich wysyłanie na serwer lub przekazywanie do innych klientów. Prawdopodobnie na werdykt ten wpłyną fakt, iż technologia ta nie wymaga od nich wysokiej wydajności, lecz jest zależna od mocy obliczeniowej serwera, na którym została użyta.

Ostatecznie, stosując protokół WebSocket w przesyłaniu dużej liczby kopii niewielkich danych w zadanym czasie można uzyskać wzrost wydajności nawet o kilkaset procent. Można nawet przewidywać, że za jego pomocą przesłanie kilku lub jednej kopii takich danych (w porównaniu do zwykłego protokołu HTTP) może być nieco szybsze.

4.5. Wpływ narzutów na szybkość przesyłania

Kolejnym etapem niniejszych prac było zbadanie szybkości wysyłania 100 kopii danych z komputerów klienckich na serwer za pomocą metody POST oraz ich pobieraniem z serwera za pomocą metody GET wraz z określonymi narzutami żądania HTTP. W badaniach każde żądanie HTTP (spośród 100 wykonanych) wzbogacono o daną liczbę dodatkowych pól nagłówków w formacie:
`nazwa-1: zawartosc`, `nazwa-2: zawartosc`, itd. Testy obejmowały przypadki dodania 10, 50, 100, 200 i 300 nadmiarowych narzutów. Należy nadmienić, że technika ta jest możliwa jedynie w przypadku protokołu HTTP – protokół WebSocket uniemożliwia wykonywanie takich czynności. Na podstawie wyników wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 12.



Rysunek 12: Średnie czasy przesyłania tekstów w 100 kopiach pomiędzy komputerami klienckimi a serwerem za pomocą HTTP z dodatkowymi narzutami.

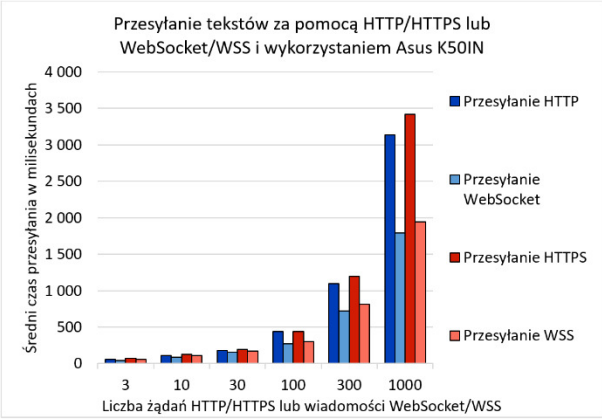
Analizując uzyskane wyniki, można stwierdzić, że dodanie kilku lub kilkunastu dodatkowych pól nagłówków nie ma znaczącego wpływu na wydajność protokołu HTTP. Jeśli jednak ich liczba wyniesie 100, co zazwyczaj rzadko się zdarza, szybkość przesyłania tekstów – według przeprowadzonych badań – może spaść o około 20%. W tym przypadku rozmiar nadmiarowych narzutów w każdym żądaniu HTTP wyniesie około 1900 znaków (bajtów), stąd ten wynik. Należy podkreślić, że w testach uwzględniono dodawanie dodatkowych narzutów dla każdego żądania HTTP, bo jeśli zostaną one dodane również do odpowiedzi HTTP, wynik będzie jeszcze gorszy.

4.6. Wpływ szyfrowania na szybkość przesyłania

Ostatnim etapem niniejszych prac było zbadanie szybkości przesyłania tekstów poprzez protokół HTTPS (czyli poprzez HTTP z szyfrowaniem TLS) oraz protokół WebSocket Secure (czyli poprzez WebSocket z szyfrowaniem TLS – w skrócie WSS) oraz porównano je z zwykłymi odpowiednikami.

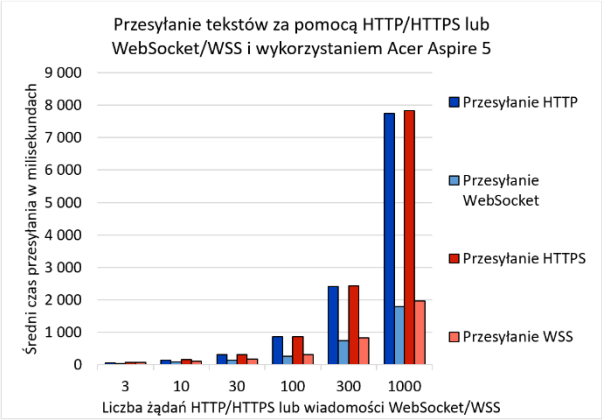
W pierwszym kroku zmierzono czas przesyłania tekstów pomiędzy starszym, wolniejszym komputerem klienckim a serwerem za pomocą protokołu HTTP i HTTPS oraz WebSocket i WebSocket Secure

w określonej ilości żądań. Na podstawie wyników wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 13.



Rysunek 13: Średnie czasy przesyłania tekstów pomiędzy komputerem klienckim Asus K50IN a serwerem za pomocą protokołów HTTP/HTTPS i WebSocket/WSS.

W drugim kroku zmierzono czas przesyłania tekstów pomiędzy nowszym, szybszym komputerem klienckim a serwerem za pomocą protokołu HTTP i HTTPS oraz WebSocket i WebSocket Secure w określonej ilości żądań. Na podstawie wyników wykonano wykres słupkowy ze średnich wartości, który przedstawiono na rysunku numer 14.



Rysunek 14: Średnie czasy przesyłania tekstów pomiędzy komputerem klienckim Acer Aspire 5 a serwerem za pomocą protokołów HTTP/HTTPS i WebSocket/WSS

Najważniejszym faktem do stwierdzenia na powyższych wykresach jest to, że zastosowanie szyfrowania TLS dla protokołu HTTP oraz WebSocket, ma minimalnie negatywny wpływ na czas w wysyłaniu danych z komputerów klienckich na serwer i ich pobieraniu z serwera poprzez klientów, bez względu na szybkość komputera klienckiego. Przyczyną takiego zjawiska są niskie wymagania sprzętowe w szyfrowaniu i deszyfrowaniu danych.

5. Wnioski

Z przeprowadzonych badań wynika, że wykorzystując protokół WebSocket, w porównaniu do HTTP, można osiągnąć wzrost wydajności o kilkaset procent. Można nawet przewidywać, że za jego pomocą przesłanie kilku lub jednej kopii takich danych (w porównaniu do zwykłego protokołu HTTP) może być nieco szybsze. Wzrost ten jest bardziej zauważalny dla transmisji powyżej 100 kopii danych, dlatego zastosowanie tej nowej technologii świetnie sprawdzi się w aplikacjach wymagających przesyłanie setek, a nawet tysięcy porcji danych na sekundę. Autor nadmienił, że badania zostały wykonane w pełni domyślnych ustawieniach protokołów, dlatego jeśli zwiększy się narzuty żądania lub odpowiedzi HTTP, różnica będzie jeszcze większa, co również udowodniono. Zauważono również, że protokół WebSocket tak samo wydajnie pracował na wolniejszym, jak i szybszym laptopie, natomiast wykonywanie żądań HTTP i przetwarzanie jego odpowiedzi znacznie lepiej przebiegało na nowszym komputerze klienckim. Zastosowanie szyfrowania TLS, zarówno dla HTTP, jak i WebSocket ma znikomy wpływ na obniżenie szybkości tych protokołów.

Ostatecznie, można wysunąć wniosek, że nie ma sensu implementować technologii WebSocket dla przesłania kilku lub kilkunastu porcji danych, gdyż w tym wypadku wystarczy wykorzystać zwykłe odpytywanie HTTP – tym bardziej, że powstają coraz szybsze komputery, które coraz sprawniej obsłużą tą technikę, pomijając sam fakt nadchodzącego szybkiego Internetu, który niweluje nadmiarowość i opóźnienia generowane przez to rozwiązanie.

Literatura

[1] World Wide Web, w: Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/World_Wide_Web, 13.01.2021.

[2] Hypertext Transfer Protocol, w: Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, [13.01.2021].

[3] WebSockets – A Conceptual Deep-Dive, w: Ably Realtime, https://www.ably.io/concepts/websockets, [13.01.2021].

[4] Ajax (programming), w: Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Ajax_(programming), [13.01.2021].

[5] WebSocket Simplified, w: Coding Simplified With Shad, https://iamshadmirza.hashnode.dev/websocket-simplified-cjxjzcu0m002i3hs1eewt2p80, [13.01.2021].

[6] WebSocket, w: Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/WebSocket, [13.01.2021].

[7] RFC 6455 – The WebSocket Protocol, w: IETF Tools, https://tools.ietf.org/html/rfc6455, [13.01.2021].

[8] Real-time web, w Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Real-time_web, 13.01.2021.

[9] W. Słodziak, Z. Nowak: Performance Analysis of Web Systems Based on XMLHttpRequest, Server-Sent Events and WebSocket. Springer International Publishing, 2016.

[10] Benchmark 5-milion Websockets, w: Oat++, https://oatpp.io/benchmark/websocket/5-million/, [13.01.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 75–80
Received: 1 February 2021
Accepted: 15 February 2021

Comparative analysis of JavaScript package managers - yarn and NPM

Analiza porównawcza menadżerów pakietów JavaScript – yarn oraz NPM

Michał Chodorowski*

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

Abstract

In this article, two leading solutions for managing packages in projects which are using JavaScript technology (yarn and npm) were subjected to a comparative analysis. As part of the implementation, two configuration files were created, one of which represents an empty application created on the basis of an application template based on the Angular framework in version 8. The second file reflects an extensive web application based on the same framework, but with the addiction of over 100 dependencies. The research was focused on the time efficiency of both solutions.

Keywords: NPM; yarn; package manager; performance testing

Streszczenie

W niniejszym artykule analizie porównawczej poddano dwa wiodące rozwiązania służące do zarządzania pakietami w projektach wykorzystujących technologię JavaScript (yarn oraz npm). W ramach realizacji powstały dwa pliki konfiguracyjne, z których jeden reprezentuje pustą aplikację stworzoną na podstawie szablonu aplikacji opartej na szkielecie programistycznym Angular w wersji 8. Drugi plik odzwierciedla rozbudowaną aplikację internetową opartą o ten sam szkielet programistyczny, lecz z dodatkiem ponad 100 zależności. Badania ukierunkowane zostały na wydajność czasową obu rozwiązań.

Słowa kluczowe: NPM; yarn; menadżer pakietów; testy wydajnościowe

*Corresponding author
Email address: mic.chodowrowski@gmail.com (M. Chodorowski)

1. Wstęp

Wraz z postępującym coraz szybciej zjawiskiem cyfryzacji, w naturalny sposób rośnie zapotrzebowanie na tworzenie szeroko rozumianego oprogramowania. To z kolei powoduje szybszy rozwój języków programowania, szkieletów, architektury oraz narzędzi do wytwarzania oprogramowania. Obecnie na rynku istnieje dostęp do bardzo wielu narzędzi, które mają za zadanie ułatwiać pracę programistom, często przez oszczędność czasu, który jest największym kosztem w całym procesie wytwarzania oprogramowania. Z tego powodu, przed przystąpieniem do pisania kodu, zespoły projektowe zastanawiają się nad doborem technologii oraz narzędzi, przy pomocy których będą tworzyć projekt. Jednym z fundamentalnych i niezbędnych narzędzi, bez których żadna współczesna aplikacja internetowa nie może się obyć, to menadżer pakietów. Wydawać by się mogło, że to mało istotna kwestia, ponieważ zarówno wybrany do analizy yarn oraz NPM, są rozwiązaniami używanymi przez miliony programistów na całym świecie. Jednak, przyglądając się bliżej tworzeniu oprogramowania można zaoszczędzić nawet kilka sekund na często wykonywanej operacji, to na przestrzeni dłuższego okresu czasu, oraz biorąc pod uwagę fakt, że nad daną aplikacją nie pracuje nigdy tylko jeden programista, zyskać można wiele godzin, które można by przeznaczyć na programowanie zamiast na czekanie. Wyniki analizy porównawczej zawarte w tym artykule mogą pomóc i oszczędzić wiele godzin koniecznych do wykonania analiz NPM i yarn.

2. Cel badań i hipoteza

Do badań wykorzystano dwa testowe pliki konfiguracyjne oparte o ten sam szkielet programistyczny, lecz z różną liczbą zależności, które zostały dobrane tak aby pierwszy plik konfiguracyjny odwzorowywał pustą aplikację zawierającą tylko niezbędne zależności, a drugi odwzorowywał rozbudowaną aplikację. Celem badań było przeanalizowanie wydajności czasowej tych dwóch rozwiązań.
Za hipotezę badawczą przyjęto:
Pomimo faktu, iż npm jest narzędziem chętniej wybieranym przez programistów, to yarn jest lepszym wyborem pod względem wydajności czasowej w aplikacjach zawierających dużą ilość pakietów.

3. Systemy zarządzania pakietami JavaScript

3.1. NPM

NPM (ang. „Node Package Menager") jest domyślnym menadżerem pakietów dla środowiska uruchomieniowego Node.js dla języka JavaScript [1]. NPM dzieli się na dwie główne części [2]:

- CLI (ang. Command-Line Interface) –interfejs wiersza poleceń - narzędzie do pobierania oraz do udostępniania pakietów;
- internetowe repozytorium - zawiera pakiety JavaScript [3], które zostały stworzone i opublikowane przez społeczność programistów, którzy udostępnili swoje rozwiązania dla szerszego grona odbiorców.

Instalacja

Instalacja NPM sprowadza się do zainstalowania Node js, korzystając z oficjalnej strony producenta. NPM zostaje automatycznie zainstalowany wraz z Node js.

Skrypty NPM

Plik konfiguracyjny (szerszy opis znajduje się w rozdziale 3.3), wspiera własność ***scripts*** (Listing 1), która może być zdefiniowana w celu uruchomienia CLI, w celu zainstalowania pakietów, wykonania konkretnej operacji, lub zestawu operacji. Często programiści definiują takie skrypty, aby nie trzeba było za każdym razem wykonywać długich sekwencji wywołań poszczególnych komend przez CLI, usprawnić sobie prace i zaoszczędzić czas poprzez wywołanie jednego skryptu, który zadziała automatycznie wraz z wywołaniem np. komendy **`npm install`**.

Listing 1: Skrypt NPM wykorzystujący własność scripts.

```
{ "scripts": {
   "build": "tsc",
   "format": "prettier --write **/*.ts",
   "format-check": "prettier --check
           **/*.ts",
    "lint": "eslint src/**/*.ts",
    "pack": "ncc build",
    "test": "jest",
    "all": "npm run build && npm run for
     mat && npm run lint && npm test"
  }
}
```

Semantyka wersjonowania

W celu zapewnienia niezawodności, funkcjonowania oraz bezpieczeństwa projektu JavaScript, za każdym razem, gdy dokonywane są znaczące zmiany w jakimś pakiecie, zalecane jest publikowanie nowej wersji pakietu ze zmienionym numerem wersji w pliku konfiguracyjnym pozostającym w zgodności z dyrektywami semantyk NPM. Dokładne przestrzeganie semantyki wersjonowania pomaga innym programistom, którzy polegają na danym pakiecie, zrozumieć jakie zmiany zostały dokonane, i czy wskutek tych zmian Ci programiści muszą dostosowywać swoją aplikację tak aby nadal działała prawidłowo, z nową wersją pakietu. Zwiększenie numeru wersji jest zalecane, zwłaszcza jeżeli zmiany zrywają jakąkolwiek zależność lub zmieniają układ zależności. Zmiany tego typu są w stanie uszkodzić cały projekt.

Plik blokady

Plik blokady [5] używany jest w celu wyznaczenia dokładnej wersji zależności, która musi zostać użyta w danym projekcie, aby zapewnić jego poprawne działanie. Jeżeli w pliku blokady (package-lock.json) nie została zdefiniowana dokładna wersja, którejkolwiek z zależności – w takiej sytuacji zostanie pobrana najnowsza stabilna wersja danej zależności. Posiadanie pliku blokady jest szczególnie ważne na serwerach CI (ang. „Continous Integration”), gdzie spoczywają produkcyjne wersje aplikacji. Najważniejsze komendy NPM przedstawia Tabela 1.

Tabela 1: Najczęściej używane komendy w NPM CLI

Komenda	Opis działania
npm install	Instaluje zależności
npm ci	Generuje produkcyjną wersje projektu
npm audit	Skanuje projekt w poszukiwaniu zależności
npm audit fix	Automatyczna naprawa podatności

3.2. Yarn

Yarn jest menadżerem pakietów JavaScript, który został wprowadzony przez Facebooka w 2017 roku. Obecnie wspiera go również Google. Yarn stał się głównym substytutem dla istniejącego już na rynku konkurencyjnego rozwiązania NPM. Yarn jest w pełni kompatybilny wstecz ze strukturą NPM [4].

Powody, dla których zespół Facebook’a zdecydował się na stworzenie własnego menadżera pakietów Yarn:

- możliwość działania w trybie bez dostępu do internetu, ponieważ Yarn posiada mechanizm zapamiętywania pakietów w pamięci podręcznej, dzięki czemu raz załadowane pakiety, zostają w tzw. cache, zamiast być pobierane ponownie, co przekłada się bezpośrednio na wydajność czasową,
- uruchamianie instalacji domyślnie w trybie deterministycznym, dzięki czemu struktura folderu zależności na każdej maszynie pozostaje identyczna. Zabieg ten zapobiega powstawaniu tzw. „piekła zależności”, zwłaszcza gdy nad projektem pracuje w danym momencie wielu programistów jednocześnie, na różnych maszynach. Piekło zależności (ang. dependency hell) – potoczny termin używany przez programistów, który określa błędnie zdefiniowane lub trudne do spełnienia zależności, uniemożliwiające lub utrudniające poprawną instalacje lub działanie aplikacji.

Instalacja

W celu zainstalowania Yarn wystarczy posiadać Node.js na maszynie, na której odbywać się będzie instalacja. Yarn jest pakietem dostępnym w repozytorium pakietów NPM. Aby zainstalować go globalnie (posiadając Node.js) wystarczy wykonać komendę:

```
npm install -g yarn
```

Skrypty yarn

Yarn podobnie do NPM ma możliwość uruchamiania zdefiniowanych skryptów w pliku konfiguracyjnym, jednak yarn umożliwia programiście przekazanie jako wartość cały plik JavaScript zawierający skrypt, który ma się uruchomić po podaniu klucza, pod którym zapi-

sany został w pliku konfiguracyjnym. Służy do tego komenda **`yarn run <nazwa_skryptu>`**.

Listing 2: Przypisanie skryptu do pliku konfiguracyjnego.

```
"scripts": {
"build-project": "node build-project.js"
}
```

W przypadku wywołania komendy

`yarn run build-project`

zostanie uruchomiona komenda wywołująca instrukcję znajdujące się w pliku o tej samej nazwie, z rozszerzeniem js. Nazewnictwo skryptów yarn przekładać się może na moment ich wywołania. Przykładowo nazwa skryptu zawierająca frazę *preinstall* spowoduje, że skrypt zostanie wywołany przed zainstalowaniem pakietu, natomiast w przypadku fraz *postinstall, install, postinstall, prepublish, prepare* skrypt zostanie wywołany po zakończeniu instalacji pakietu.

Plik blokady

W momencie, w którym do zarządzania pakietami używany jest yarn, plik blokady zostaje wygenerowany automatycznie, co jest olbrzymią zaletą tego rozwiązania. Plik blokady tworzony przez yarn nosi nazwę ***yarn.lock***. Kolejną zaletą jest fakt, iż za każdym razem, gdy uruchamiana jest komenda ***yarn install,*** plik blokady jest aktualizowany w sposób automatyczny. Plik blokady w yarn, podobnie jak w przypadku NPM, pozwala programistom uniknąć zjawiska zwanego piekłem zależności. Warto zwrócić uwagę na złożoność oraz mnogość informacji, które dostarcza programistom plik blokady yarn. Plik ten zawiera nazwę pakietu, wersję, link do rejestru repozytoriów, hash integralności oraz zależności, które bazują na tym pakiecie.

Semantyka wersjonowania

Yarn respektuje identyczną politykę wersjonowania co NPM. Dokładniejszy opis tej semantyki podano w rozdziale 3.1.

3.3. Plik konfiguracyjny

Plik konfiguracyjny (*package.json*) – każdy projekt napisany w JavaScript, niezależnie od tego czy jest to pełnoprawna aplikacja serwerowa napisana z Node.js, czy jest to zwykła aplikacja internetowa, może zostać skondensowana do pakietu NPM, zawierającego swoje własne informacje o tym pakiecie oraz swój własny plik konfiguracyjny, który opisuje ten projekt. Plik konfiguracyjny zostaje wygenerowany w momencie wywołania komendy ***npm init***, inicjalizującej projekt JavaScript z domyślnymi metadanymi projektowymi:

- ***name:*** nazwa projektu lub biblioteki,
- ***version:*** wersja projektu, która zazwyczaj jest pomijana, ponieważ w większości przypadków nie ma konieczności jej ustawiania na tym etapie, lub ustawiana jest później bazując np. na dokumentacji projektowej,
- ***description:*** opis projektu,
- ***licenses:*** licencja jaką objęty jest tworzony projekt.

4. Analiza porównawcza

Kryteria, które uwzględniono w prowadzonej analizie porównawczej to [7]:

- instalacja,
- wydajność,
- prędkość,
- niezawodność,
- komendy CLI,
- ilość logów,
- ilość zajmowanego miejsca na dysku przez folder zależności,
- popularność,
- bezpieczeństwo.

Instalacja NPM oraz instalacja yarn została przedstawiona odpowiednio w rozdziałach 3.1. oraz 3.2. Porównując oba te procesy stwierdzono, że NPM jest prostszy w instalacji. Porównania wydajności znajduje się w rozdziale 5, a wyniki przedstawiono w rozdziale 6.

Obydwa rozwiązania pobierają pakiety z tego samego repozytorium, używając w przypadku yarn komendy yarn add a w przypadku NPM jest to npm install. Jednakże yarn jest o wiele szybszy niż NPM, co przedstawione zostało na Rysunku 2, ponieważ yarn instaluje wszystkie pakiety jednocześnie, ponadto yarn zapisuje pobrane pakiety w pamięci podręcznej, co zapobiega pobieraniu tego samego pakietu po raz drugi.
Menadżer pakietów, który stworzyła olbrzymia firma z sektora wytwarzania oprogramowania, ma pewne zalety (yarn), rozwiązanie to jest niezawodne i stabilne, dużo bardziej od NPM, ponieważ yarn używa pliku blokady, oraz deterministycznego algorytmu podczas instalacji każdego z pakietów, yarn również gwarantuje, że jeżeli coś działa w jednym środowisku programistycznym to będzie działało również na każdym innym. Jeżeli w yarn pojawia się jakiś błąd jest bardzo szybko naprawiany przez zespół programistyczny Marka Zuckerberga, czego niestety nie można powiedzieć na temat NPM.

Programiści muszą spędzić wiele godzin studiując komendy wykorzystywane w CLI, każdego narzędzia, w aspekcie menadżerów pakietów nie jest inaczej. W tabeli 2 przedstawiono komendy CLI dla obu pakietów.

Porównanie liczby logów zostało wykonane na podstawie instalacji tego samego pakietu (lodash). Yarn wygenerował 13 linii logów, natomiast NPM około 40 linii. Yarn generuje skondensowane logi, przekazując programiście tylko najważniejsze informacje, więc w liczbie logów wygrywa yarn.

Porównanie ilości zajmowanej przestrzeni dyskowej sprawdzono po wykonaniu komendy instalacyjnej i sprawdzeniu ilość zajmowanej przestrzeni dyskowej.

Dla yarn folder ***node_modules*** zajmował **307 MB**, zaś NPM **317 MB**.

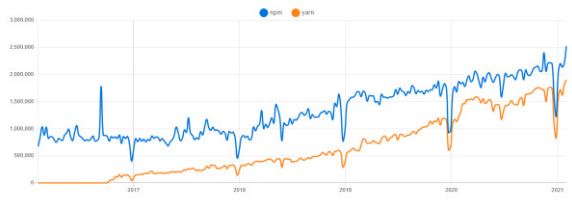
Wybierając menadżer pakietów należy również zwrócić uwagę na popularność rozważanego rozwiązania. Jeżeli wybrana technologia jest bardziej popularna, wówczas można liczyć na szerokie spektrum pomocy od społeczności. Jednak nie warto skreślać nowych rozwiązań, bowiem wszystko co jest nowe i sygnowane przez giganta technologicznego (Facebook w przypadku yarn), będzie zyskiwać szybko na popularności. Rysunek 1 przedstawia wykres liczby pobrań obu rozwiązań na przestrzeni ostatnich pięciu lat. Obserwując ten wykres bez trudu można określić, które rozwiązanie było popularniejsze na przestrzeni tych lat, ale również można dostrzec, że w roku 2020 popularność yarn znacząco wzrosła.

Tabela 2: Komendy CLI różne w obu menadżerach

Komenda	NPM	yarn
Inicjalizacja projektu	npm init	yarn init
Uruchomienie testów dla konkretnego pakietu	npm test	yarn test
Sprawdzenie czy, którykolwiek pakiet uległ przedawnieniu	npm outdated	yarn outdated
Publikacja pakietu do repozytorium NPM	npm publish	yarn publish
Uruchomienie skryptu	npm run	yarn run
Czyszczenie pamięci podręcznej	npm cache clean	yarn cache clean
Proces logowania i wylogowania	npm login / npm logout	yarn login / yarn logout

Tabela 3: Przedstawia komendy CLI identyczne w obu menadżerach.

Komenda	NPM	yarn
Instalacja zależności	npm install	yarn
Instalacja pakietu	npm install [nazwa_pakietu]	yarn add [nazwa_pakietu]
Odinstalowanie pakietu	npm uninstall [nazwa_pakietu]	yarn remove [nazwa_pakietu]
Aktualizacja środowiska	npm update	yarn upgrade
Aktualizacja pakietu	npm update [nazwa_pakietu]	yarn upgrade [nazwa_pakietu]
Instalacja pakietu globalnie	npm install --global [nazwa_pakietu]	yarn global add [nazwa_pakietu]
Odinstalowanie pakietu globalnie	npm uninstall --global [nazwa_pakietu]	yarn global remove [nazwa_pakietu]
Interaktywna aktualizacja	npm run upgrade-interactive	yarn upgrade-interactive



Rysunek 1: Popularność yarn i npm w latach 2016-2021 [6]

Bezpieczeństwo jest jedną z ważniejszych kwestii w porównaniu menadżerów pakietów, zaraz po wydajności. Początkowo yarn był uznawany za najbezpieczniejszy menadżer pakietów, ale zespół programistyczny NPM wykonał dużo pracy, aby zmienić ten fakt. Od wersji 6.0 NPM wprowadził mechanizm wbudowanych zabezpieczeń, dzięki któremu, jeżeli programista będzie chciał zainstalować pakiet ze znaną luką w zabezpieczeniach, NPM wyświetli stosowny komunikat ostrzegawczy.

Yarn natomiast ma inny mechanizm, zapewniający bezpieczeństwo, który polega na weryfikacji sumy kontrolnej każdego pakietu oraz licencji jaką objęty jest dany pakiet, i tego w przypadku NPM wciąż brakuje. Mimo dużego nakładu pracy wykonanego przez zespół NPM, yarn nadal pozostaje lepiej zabezpieczonym menadżerem pakietów niż NPM.

5. Środowisko badawcze

Badania zostały przeprowadzone na komputerze o specyfikacji:

- procesor – Intel Core i5 4460 3.2Ghz, 6 MB,
- dysk – SSD ADATA 256GB 2,5” SATA Ultimate S800,
- karta graficzna – MSI Radeon R9 270X HAWK, 2GB GDDR5,
- płyta główna – Gigabyte GA-H81-D3,
- prędkość łącza internetowego – 250 Mb/s,
- system operacyjny – Windows 10 Pro 64 bit,
- środowisko programistyczne IntelliJ IDEA Ultimate wer. 2020.2.2.

5.1. Aplikacja testowa

Stworzenie aplikacji testowej polegało na wygenerowaniu dwóch nowych projektów za pomocą IntelliJ IDEA wykorzystujących szkielet programistyczny Angular, który stanowił tylko bazę imitującą kompletny i działający projekt. Do pliku konfiguracyjnego projektu pełniącego role dużej aplikacji dodane zostało około 100 dodatkowych losowych zależności. Drugi plik pozostał bez zmian.

6. Badania wydajności czasowej

Badania zostały przeprowadzone w seriach po 100 prób na każdą komendę. Dwie komendy na dwóch plikach konfiguracyjnych, dla obu menadżerów pakietów dało łącznie 800 prób. Z czasów uzyskanych, w przypadku każdej z komend, na poszczególnych plikach konfiguracyjnych, została wyciągnięta średnia arytmetyczna. Wyniki przedstawiono w Tabeli 3.

6.1. Charakterystyka zbioru danych pierwotnych

Dane podchodzą z testów wydajnościowych przeprowadzonych na dwóch plikach konfiguracyjnych za pomocą biblioteki gnomon oraz z wykorzystaniem terminala, dla obu menadżerów pakietów. Wyniki otrzymano z dokładnością do 1 ms. Biblioteka gnomon została wykorzystana do pomiaru czasów wykonania poszczególnych komend, wywoływanych za pomocą terminala.

6.2. Algorytmy służące do obliczania poszukiwanych wartości

Do obliczania poszukiwanych wielkości użyto wzoru na średnią arytmetyczną. Uzyskane wyniki zostały zaokrąglone do dwóch miejsc po przecinku i są zestawione w rozdziale 6.

6.3. Przeprowadzenie badań

Przed przystąpieniem do badania wydajności czasowej menadżerów pakietów, upewniono się, iż żaden zbędny proces, obciążający środowisko badawcze nie pozostał włączony, sprawdzono również obciążenie środowiska badawczego i upewniono się, że żadna usługa sieciowa nie działa w tle i nie pobiera zasobów. Badania były przeprowadzone dla następujących scenariuszy:

- **S1** - plik konfiguracyjny zawiera podstawowe zależności dla projektu Angular.
- **S2** - plik konfiguracyjny zawiera dodatkowe zależności dla projektu Angular.

Scenariusz S1 dla podstawowego projektu Angular

1. Stworzono pusty projekt poprzez użycie komendy *npm init* dla NPM oraz *yarn init* dla yarn.
2. Wykonano komendy *yarn install | gnomon* dla folderu, w którym znajduje się projekt stworzony za pomocą yarn i analogicznie *npm install | gnomon* dla projektu utworzonego przez NPM.
3. Wykonano komendy *npm update | gnomon*, oraz *yarn upgrade | gnomon.*

Scenariusz S2 dla rozbudowanego projektu Angular

Procedura testowa jest zbliżona do S1, z tym, że zanim przystąpiono do wykonania testów, dodane zostało ponad sto dodatkowych zależności, aby odzwierciedlić wykonanie każdej z komend na rozbudowanym projekcie.

7. Wyniki badań

W Tabeli 4 przedstawiono średnie czasy wykonania scenariuszy S1 i S2 dla yarn, natomiast w Tabeli 5 – dla NPM. Rysunki 2 i 3 obrazują rezultaty uzyskane dla obu pakietów.

Tabela 4: Wydajność czasowa yarn

Komenda CLI	Średni czas wykonania komendy dla S1	Średni czas wykonania komendy dla S2
yarn install	45,89s	107,4s
yarn upgrade	19,9s	49,09s

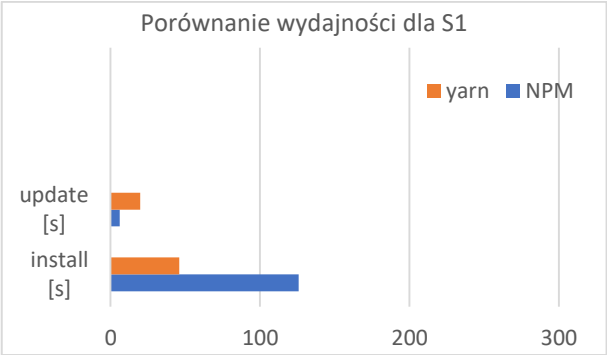
Tabela 5: Wydajność czasowa NPM

Komenda CLI	Średni czas wykonania komendy dla S1	Średni czas wykonania komendy dla S2
npm install	125,8s	258,43s
npm update	6,08s	19,4s

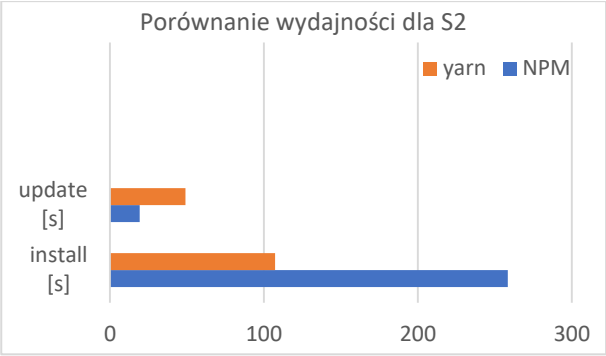
8. Wnioski

Na podstawie przeprowadzonych badań wyciągnięto następujące wnioski:

- Yarn jest wydajniejszy pod względem pobierania wszystkich zależności z użyciem komendy instalacyjnej od NPM, niemal trzykrotnie w przypadku małych aplikacji internetowych oraz nieco ponad dwa i pół razy wydajniejszy w przypadku rozbudowanych aplikacji internetowych.
- NPM jest wydajniejszy pod względem aktualizacji pakietów od yarn, w przypadku zarówno małych aplikacji internetowych, ponad trzykrotnie jak i w przypadku rozbudowanych aplikacji, w przypadku których NPM, jest wydajniejszy ponad dwukrotnie.
- Pomimo faktu, iż NPM jest narzędziem chętniej wybieranym przez programistów, to yarn jest lepszym wyborem pod względem wydajności czasowej w aplikacjach zawierających dużą ilość pakietów.
- Wyniki uzyskane w artykule autorstwa Jacobs Alexander z 2019 roku, które nie dały jednoznacznej odpowiedzi, które rozwiązanie autor poleca do użytku, najbliżej tego wyniku w porównaniu tego autora był NPM. Co tylko potwierdza ogólnie znaną tezę, że w świecie programowania technologie zmieniają się bardzo dynamicznie. Nie inaczej jest w przypadku JavaScript, w 2019 roku NPM wiódł prym, a w 2020 prym wiódł yarn.
- Hipoteza postawiona na początku artykułu została potwierdzona.



Rysunek 2: Porównanie wydajności dla aplikacji z podstawową liczbą zależności



Rysunek 3: Porównanie wydajności aplikacji z rozbudowaną siatką zależności

Yarn osiągnął lepszy wynik od NPM w następujących kategoriach:

- bezpieczeństwo,
- ilość zajmowanej przestrzeni dyskowej,
- ilość produkowanych logów,
- niezawodność,
- prędkość,
- wydajność czasowa.

NPM okazał się lepszym rozwiązaniem pod kątem instalacji oraz popularności wśród społeczności programistów.

Literatura

[1] MSR '16: Proceedings of the 13th International Conference on Mining Software Repositories https://dl.acm.org/doi/abs/10.1145/2901739.2901743, [24.01.2021].

[2] Charakterystyka NPM https://www.freecodecamp.org/news/what-is-npm-a-node-package-manager-tutorial-for-beginners/, [03.01.2021].

[3] Działanie menadżerów pakietów JavaScript https://www.freecodecamp.org/news/javascript-package-managers-101-9afd926add0a/, [03.01.2021].

[4] Charakterystyka yarn https://engineering.fb.com/2016/10/11/web/yarn-a-new-package-manager-for-javascript/, [11.02.2021].

[5] E. Wittern, P. Suter, S, Rajagopalan, A look at the dynamics of the JavaScript package ecosystem, MSR'16: Proceedings of the 13 Conference of Mining Software Repositories, (2016) 351-361, https://dl.acm.org/doi/10.1145/2901739.2901743.

[6] A. Jacobs, Comparsion of Javascript Package Managersm 2019, https://www.theseus.fi/handle/10024/227945, [24.01.2021].

[7] Wykres popularności obu rozwiązań https://www.npmtrends.com/npm-vs-yarn, [24.01.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 81–88
Received: 3 February 2021
Accepted: 10 February 2021

Accessibility assessment of selected university websites

Ocena dostępności wybranych serwisów uczelni wyższych

Wojciech Stasiak*, Mariusz Dzieńkowski

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

Abstract

The accessibility of websites consists in ensuring the possibility of using the information posted there by all users, especially by disabled people. The aim of the study was to examine the levels of accessibility of websites of Lublin universities and to compare them with the websites of two universities in Poland selected for the purpose of the study. With this aim in view, an experiment was developed, which consisted of two parts. In the first part of the experiment, a specially made original checklist was used containing questions about specific functionalities of websites corresponding to important accessibility issues. In the second part, automated tools were used with a view to assessing the accessibility. The research material consisted of eight websites of universities from Lublin and two websites of universities from other cities in Poland that were compared in the study. Those two universities in contrast to the eight ones additionally had special sets of websites for people with disabilities. Such additional sets of websites specifically designed for the needs of people with disabilities were also included in the research. After conducting the experiment, it was possible to identify the universities which have the best and the worst websites in terms of accessibility results. Furthermore, the study revealed specific pages of these websites that were characterized by the highest and the lowest average accessibility ratings. The results of the expert analysis showed that the examined university websites do not have an attached declaration of accessibility, contain documents in the PDF format that are not accessible, often do not allow for changes of colour and do not have a mobile version.

Keywords: website accessibility; accessibility evaluation; web accessibility testing techniques; accessibility automated tools

Streszczenie

Dostępność stron internetowych polega na zapewnieniu możliwości korzystania z informacji tam zamieszczonych przez wszystkich użytkowników, a zwłaszcza przez osoby niepełnosprawne. Celem pracy było zbadanie poziomów dostępności serwisów internetowych uczelni wyższych Lublina i porównanie ich z serwisami dwóch uczelni w Polsce wybranych na potrzeby tego badania. W tym celu opracowano eksperyment, który składał się z dwóch części. W pierwszej części eksperymentu zastosowano autorską listę kontrolną zawierającą pytania dotyczące określonych funkcjonalności serwisów odpowiadających istotnym kwestiom dostępności. W drugiej części do oceny dostępności wykorzystano automatyczne narzędzia. Materiałem badawczym było osiem serwisów uczelni z Lublina oraz dwóch zestawionych z nimi uczelni w innych miastach w Polsce. Uczelnie te dodatkowo posiadały w swoich serwisach specjalnie wydzielone zestawy stron dla osób z niepełnosprawnościami. Takie dodatkowe strony opracowane specjalnie dla potrzeb osób niepełnosprawnych zostały także uwzględnione w badaniach. Po przeprowadzeniu eksperymentu możliwe było wskazanie uczelni, które posiadają najlepsze i najgorsze serwisy pod względem osiąganych wyników dostępności. Dodatkowo badania wyłoniły konkretne strony tych serwisów, które charakteryzowały się najwyższymi i najniższymi średnimi ocenami dostępności. W wyniku analizy eksperckiej okazało się, że niektóre z przebadanych serwisów uczelni nie mają dołączonej deklaracji dostępności, zawierają dokumenty w formacie PDF, które nie są dostępne, często nie umożliwiają zmiany kolorystyki oraz nie posiadają wersji mobilnej.

Słowa kluczowe: dostępność stron internetowych; metody badania dostępności stron www; automatyczne narzędzia do testowania dostępności

*Corresponding author

Email address: wojciech.stasiak@pollub.edu.pl (W. Stasiak)

1. Wstęp

Dostępność jest cechą stron internetowych oznaczającą, że zawarte w nich treści za sprawą wbudowanych mechanizmów są dostępne dla możliwie największej grupy odbiorców [1]. Właściwość ta ma umożliwiać wszystkim osobom, a szczególnie osobom z niepełnosprawnościami odbieranie informacji, zrozumienie informacji, nawigowanie po dostępnych danych oraz wchodzenie w interakcje z siecią [2]. Konieczność zapewnienia dostępności wynika z przesłanek biznesowych, a więc z chęci poszerzenia grona potencjalnych użytkowników oraz z uregulowań prawnych związanych z polityką równości w dostępie do informacji [1]. We wszystkich wiodących dokumentach Unii Europejskiej zawarte są wymagania niedyskryminacji w dostępie do danych, zwłaszcza tych publicznych [1]. W Polsce wiele aktów prawnych gwarantuje równość w dostępie do informacji. Należą do nich Konstytucja RP, ustawy „o dostępie do informacji publicznej" oraz „o informatyzacji działalności podmiotów realizujących zadania publiczne" [3].

W 2016 roku ukazała się dyrektywa 2016/2102 przygotowana przez Parlament Europejski i Radę Unii Europejskiej mówiąca o dostępności stron internetowych i mobilnych aplikacji sektora publicznego zobowiązująca kraje członkowskie do uchwalenia prawa zgodnego z normą EN 301 549, zawierającego wytyczne standardu WCAG na poziomie AA [4]. W związku z tym Rząd Polski przygotował i 4 kwietnia 2019 r. opublikował ustawę o dostępności cyfrowej stron internetowych i aplikacji mobilnych podmiotów publicznych, wprowadzającą takie normy w życie. Przepisy te mówią o obowiązku dostępności cyfrowej, obowiązku umieszczenia deklaracji dostępności, opisują zasady monitorowania dostępności cyfrowej i zasady postępowania w przypadku braku tej dostępności [5].

Publiczne uczelnie wyższe, podobnie jak inne instytucje publiczne, których strony zostały opublikowane przed 23 września 2018 r. zostały zobligowane do dostosowania swoich serwisów do 23 września 2020 roku [6] do standardu WCAG 2.1 na poziomie AA. Uczelnie niepubliczne, nie muszą formalnie spełniać wymogów w zakresie dostępności ich zawartości, ale jest to w ich interesie biznesowym.

Celem niniejszej pracy jest przedstawienie oraz zastosowanie hybrydowej metody oceny dostępności opartej na manualnym sprawdzeniu serwisu przez ekspertów oraz wykorzystującej narzędzia automatyczne do oceny poziomu dostępności serwisów internetowych uczelni publicznych i niepublicznych Lublina w zestawieniu z dwoma uczelniami w Polsce wybranymi dla celów badawczych tej pracy.

2. Przegląd literatury

Dostępność serwisów internetowych wiąże się z dwoma nierozerwalnymi aspektami: informacyjnym i technicznym. Nawet serwis zbudowany jak najlepiej pod względem technicznym, może zawierać informacje, które nie będą dostępne. Także wytwarzanie i publikowanie dostępnych treści bez zapewnienia technicznych rozwiązań w zakresie dostępności może skutkować tym, że strona www nie będzie dostępna [7].

Na dostępność serwisów internetowych, oprócz treści i oprogramowania do prezentacji zawartości www, mają również technologie wspierające (np. czytnik ekranu, specjalne klawiatury, przełączniki, ekranowe lupy), wiedza i doświadczenie odbiorców, świadomość i kompetencje projektantów i twórców serwisów, narzędzia do budowania serwisów oraz narzędzia do ich oceny [8].

Istotnymi dokumentami określającymi dostępność serwisów internetowych są różnego typu standardy. Wśród nich należy wymienić standardy opracowane przez konsorcjum W3C (ang. World Wide Web Consortium), standardy amerykańskie oraz brytyjskie. Utworzona w ramach W3C grupa WAI (ang. Web Accessibility Initiative), zajmująca się w promowaniem standardów służących poprawie dostępności usług internetowych dla osób niepełnosprawnych i wykluczonych cyfrowo, opracowała szereg dokumentów związanych z dostępnością stron www. Zawierają one m.in. zalecenia dla oprogramowania służącego do tworzenia stron internetowych ATAG (ang. Authoring Tool Accessibility Guidelines), zbiory zasad dotyczących dostępności narzędzi wspomagających użytkowników UAAG (ang. User Agent Accessibility Guidelines), wytyczne WCAG (ang. Web Content Accessibility Guidelines) dotyczące dostępności zawartości serwisów www, zalecenia WAI-ARIA (ang. Accessible Rich Internet Application), będące opisem zasad i narzędzi służących do projektowania stron internetowych, zawierających dynamiczną zawartość, opartą na technologii AJAX [9].

Od podmiotów prowadzących własne serwisy internetowe wymaga się, aby były one dostępne na odpowiednim poziomie, co tym samym podlega ocenie. Może być ona przeprowadzana automatycznie przez różnego rodzaju specjalistyczne oprogramowanie lub manualnie, za pomocą technik, metod i narzędzi wykorzystywanych w audycie i ewaluacji serwisów (np. arkusze ocen, listy kontrolne, itp.) [9].

Badania dostępności serwisów uczelni wyższych były przedmiotem badań wielu prac [10-12]. W artykule [10] monitorowano strony uczelni medycznych w Iranie pod względem spełnienia przez nie potrzeb dla osób z niepełnosprawnościami. W badaniach użyto dwóch automatycznych narzędzi do oceny poziomu dostępności: AChecker i Functional Accessibility Evaluator (FAE). Pierwsze narzędzie zwracało liczbę rozpoznanych problemów, prawdopodobnych problemów oraz potencjalnych problemów. Drugie narzędzie generowało ocenę liczbową w zakresie od 0 do 100 dla pojedynczej strony. Za pomocą walidatora AChecker przebadano strony główne 50 uczelni, zaś narzędziem FAE przeanalizowano strony główne oraz po 25 podstron każdego z 50 serwisów uniwersytetów medycznych. Wyniki końcowe przedstawiono w tabeli, podając średnie ilości błędów oraz średnie oceny generowanej przez narzędzie FAE. Po wykonaniu analiz autorzy pracy doszli do wniosku, że dostępność zbadanych stron uczelni medycznych jest na niskim poziomie.

W kolejnej pracy [11] przeprowadzono badania dostępności stron WWW wśród osób całkowicie niewidomych. W tym celu został przygotowany prototypowy serwis internetowy zgodny ze standardem WCAG 2.0. Wykorzystano go w badaniach, w których wzięło udział 16 uczestników. Ich zadaniem było najpierw zapoznanie się z serwisem, a następnie udzielenie odpowiedzi na 14 pytań dotyczących dostępności serwisu. Wyniki z badań ankietowych przeprowadzonych komputerowo pogrupowano według czterech kryteriów, którymi były: postrzegalność, zrozumiałość, operatywność oraz interaktywność strony. Uśrednione wyniki złożyły się na końcową, wysoką ocenę dostępności prototypowego serwisu.

Autorzy artykułu [12] przeprowadzili porównanie dostępności treści edukacyjnych witryn internetowych wybranych uniwersytetów z Jordanii, Anglii oraz z regionu arabskiego. W badaniach skoncentrowano się na pomiarze zgodności stron www ze standardami dostępności dla osób niedowidzących. Najpierw badacze przestudiowali istniejące standardy, a następnie przea-

nalizowali ich zastosowanie w serwisach internetowych instytucji edukacyjnych. Uzyskane wyniki pokazały, że liczba błędów dostępności stron uczelni z Jordanii i regionu arabskiego była odpowiednio 13 i 5 razy większa niż w Wielkiej Brytanii.

3. Metoda badań

W niniejszej pracy opracowano eksperyment, który składał się z dwóch części. Pierwsza część obejmowała analizę ekspercką przeprowadzoną na podstawie specjalnie do tego celu skonstruowanej listy kontrolnej, składającej się z 13 pytań dotyczących różnych mechanizmów dostępności, w które powinien być wyposażony dostępny serwis internetowy (Tabela 1). Odpowiedzi na te pytania miały charakter zero-jedynkowy.

Tabela 1: Lista kontrolna do oceny dostępności serwisów internetowych

Lp.	Treść pytania
1	Czy serwis zawiera deklarację dostępności?
2	Czy serwis umożliwia zmianę kontrastu?
3	Czy serwis zawiera mechanizmy zmiany kolorystyki stron?
4	Czy w serwisie istnieje możliwość powiększenia czcionki?
5	Czy serwis zawiera specjalne strony dla osób niepełnosprawnych?
6	Czy istnieje mobilna wersja serwisu?
7	Czy dokumenty w formacie PDF są dostępne?
8	Czy możliwa jest nawigacja po serwisie za pomocą klawiatury (klawisz TAB)?
9	Czy serwis zawiera mapę i prostą wyszukiwarkę?
10	Czy serwis jest wyposażony w zaawansowaną wyszukiwarkę?
11	Czy formularze zawierają mechanizm sprawdzania poprawności wprowadzanych danych?
12	Czy w przypadku multimediów (pliki audio, filmy) możliwa jest kontrola odtwarzania?
13	Czy multimedia (pliki audio, filmy) zawierają napisy?

W pierwszej części badań wzięło udział dwóch ekspertów znających problematykę dostępności. Ich zadaniem było zapoznanie się z ocenianym serwisem, a następnie podanie zero-jedynkowej odpowiedzi na pytania zawarte w liście kontrolnej. W ten sposób analizowane były wszystkie serwisy poddane ocenie dostępności. Uzyskane wyniki zostały następnie uśrednione.

Druga część eksperymentu, w której wykorzystano automatyczne narzędzia, składała się z trzech etapów:

- etap 1: ocena poziomu dostępności przez automatyczne narzędzie (wyrażona w postaci liczbowej),
- etap 2: ocena dokonywana na podstawie sumarycznej liczby wykrywanych przez dane narzędzie błędów,
- etap 3: ocena na podstawie analizy zdiagnozowanych typów błędów wykrytych podczas analizy każdej z siedmiu stron www danego serwisu.

Dla zobiektywizowania wyników postanowiono te same analizy powtórzyć za pomocą pięciu różnych, dobranych, automatycznych narzędzi, a z otrzymanych wyników obliczyć ocenę średnią. Wybór narzędzi został przeprowadzony na podstawie czterech kryteriów:

- analiza stron/serwisów www według standardu WCAG,
- bezpłatny dostęp (badanie pojedynczych stron lub całych serwisów),
- możliwość uruchomienia narzędzia w oknie przeglądarki (online),
- generowanie ogólnej oceny prezentowanej w postaci liczbowej (punktowej lub procentowej) wyrażającej poziom dostępności strony/serwisu.

Pierwszą fazę drugiej części eksperymentu zrealizowano za pomocą pięciu narzędzi: Lighthouse [13], ACE [14], MAUVE++ [15], FAE [16] oraz Utilitia [17]. Natomiast w drugiej i trzeciej fazie badań wykorzystano trzy narzędzia. Było to spowodowane problemami z dostępem do usługi MAUVE++ oraz pewnymi ograniczeniami narzędzia ACE.

3.1. Materiał badawczy

Analizę poziomu dostępności przeprowadzono na dziewięciu serwisach internetowych uczelni wyższych Lublina oraz dwóch czołowych uczelni w Polsce [18]. Badaniem były objęte serwisy następujących uczelni:

- Politechnika Lubelska (PL) [19]
- Uniwersytet Marii Curie-Skłodowskiej w Lublinie (UMCS) [20]
- Uniwersytet Przyrodniczy w Lublinie (UP) [21]
- Uniwersytet Medyczny w Lublinie (UM) [22]
- Katolicki Uniwersytet Lubelski (KUL) [23]
- Wyższa Szkoła Przedsiębiorczości i Administracji w Lublinie (WSPA) [24]
- Wyższa Szkoła Ekonomii i Innowacji w Lublinie (WSEI) [25]
- Wyższa Szkoła Nauk Społecznych z siedzibą w Lublinie (WSNS) [26]
- Wyższa Szkoła Społeczno-Przyrodnicza im. Wincentego Pola w Lublinie (WSSP) [27]
- Uniwersytet Warszawski (UW) [28]
- Uniwersytet Jagielloński (UJ) [29]

Serwisy dwóch ostatnich uczelni zawierają wydzielone działy, umieszczone w osobnej domenie i przeznaczone dla osób niepełnosprawnych. Również je poddano analizie pod kątem dostępności w ramach niniejszej pracy. Nazwy tych serwisów to:

- Dział ds. Osób Niepełnosprawnych Uniwersytetu Jagiellońskiego (UJ DON) [30]
- Biuro ds. Osób Niepełnosprawnych Uniwersytetu Warszawskiego (UW BON) [31]

Uznano, że badane będą ściśle określone pod względem tematycznym strony spośród wyżej wymienionych serwisów. Wybrano siedem, następujących podstron:

- strona główna uczelni,
- strona z danymi do kontaktu,
- strona główna rekrutacji,
- strona główna biblioteki,
- strona aktualności prezentująca najnowszą wiadomość,
- podstrona z informacjami dla studentów,

- strona Biura Kształcenia Międzynarodowego (BKM) lub strona z informacjami o programie Erasmus.

Podczas badań okazało się, że nie wszystkie serwisy zawierały wyżej określony zestaw stron.

3.2. Narzędzia do badania dostępności stron internetowych

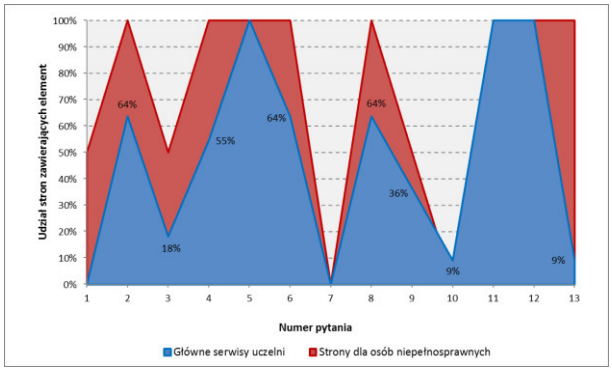
W pracy wykorzystano metodę opartą na analizie eksperckiej wspartej analizą przeprowadzoną za pomocą automatycznych narzędzi. W ocenie eksperckiej wykorzystana została lista kontrolna. Do analizy automatycznej zostały wybrane narzędzia, które spełniały wymagania zdefiniowane przez autorów. Wśród tych narzędzi znalazły się: Lighthouse, ACE (Accessibility Evaluator), MAUVE++, FAE (Functionality Accessibility Evaluator) oraz Utilitia. Pierwsze trzy programy badają pojedynczą stronę serwisu i zwracają wynik w postaci liczbowej w zakresie od 0 do 100. Ostanie narzędzie ma możliwość badania pojedynczej strony lub całego serwisu i również zwraca wynik z zakresu od 0 do 10.

4. Wyniki badań

4.1. Ocena ekspercka

Na pierwszą część eksperymentu składały się badania, które miały odpowiedzieć na pytanie „Jaki poziom dostępności mają badane serwisy uczelni, biorąc pod uwagę zawarte w nich funkcjonalności, które dotyczyły aspektów dostępności?”. W badaniu wzięło udział dwóch niezależnych ekspertów, których zadaniem była najpierw analiza stron internetowych, a następnie wypełnienie list kontrolnych dla każdego badanego serwisu. Odpowiedzi obu specjalistów były w stu procentach zgodne. Rysunek 1 ilustruje uzyskane wyniki. Na osi pionowej prezentowany jest odsetek serwisów uczelni zawierających daną funkcjonalność odnoszącą się do kwestii dostępności.

Tylko jeden z dwóch analizowanych tzw. dostępnych serwisów - UW BON, posiadał wymaganą deklarację dostępności. Natomiast żaden z pozostałych serwisów ujętych w badaniach takiego dokumentu nie posiadał. Większość serwisów uczelni (64%), w tym oba serwisy dostępne, umożliwia zmianę kontrastu.

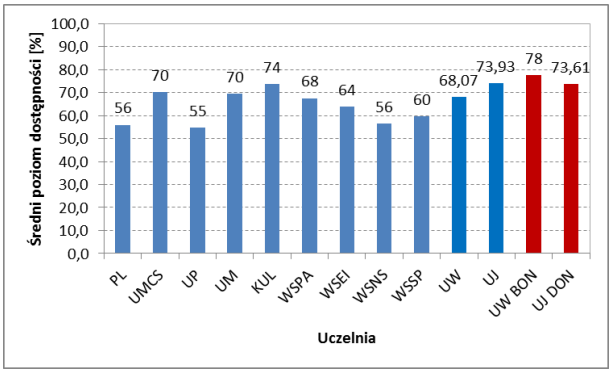


Rysunek 1: Udział serwisów zawierających daną funkcjonalność mającą bezpośredni wpływ na poziom dostępności.

Tylko dwa serwisy uczelni umożliwiają zmianę kolorystyki stron, co dało 18% w grupie ogólnodostępnych, tzn. oficjalnych i 50% w grupie serwisów dostępnych. 55% serwisów ogólnodostępnych i 100% serwisów dostępnych daje możliwość powiększenia wielkości czcionki. Wszystkie spośród z badanych serwisów uczelni wyższych posiadają specjalne strony, zawierające informacje dla osób niepełnosprawnych. Ponad połowa serwisów zawiera wersję mobilną, co stanowi około 64%. Niestety udostępniane na stronach badanych serwisów pliki PDF nie spełniają kryteriów dostępności. W przypadku 64% podstawowych serwisów, możliwa jest ich obsługa za pomocą klawiatury. Również oba serwisy dostępne posiadają taką możliwość. 36% serwisów podstawowych zawiera mapę strony i prostą wyszukiwarkę. Tylko jeden z dwóch serwisów dostępnych posiada proste narzędzie wyszukiwawcze. Wyszukiwarkę zaawansowaną miał wbudowaną tylko serwis UMCS. Formularze znajdujące się na stronach uczelni, są wyposażone w mechanizm walidacji poprawności danych. Również wszystkie strony uczelni miały mechanizm kontroli odtwarzania multimediów. Tylko serwis UW BON zawiera filmy, które wyświetlają napisy podczas ich oglądania.

4.2. Ilościowa ocena dostępności za pomocą narzędzi automatycznych

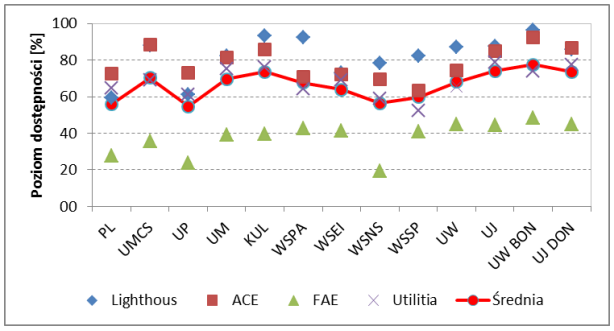
W pierwszej fazie drugiej części eksperymentu dokonano oceny każdej z siedmiu stron internetowych danego serwisu uczelni za pomocą pięciu narzędzi, które zwracały wynik w postaci liczbowej. Na Rysunku 2, oś Y prezentuje średnie poziomy dostępności serwisów poszczególnych uczelni. Z wykresu tego wynika, że serwisy dostępne UW BON i UJ DON mają najwyższe poziomy dostępności na tle serwisów uczelni z Lublina. Podobny, wysoki poziom dostępności, wynoszący około 73%, osiągnęły również serwisy KUL oraz UJ. Natomiast najniższe średnie oceny (poniżej 60%) miały serwisy czterech uczelni: PL, UP, WSNS oraz WSSP.



Rysunek 2: Średni poziom dostępności serwisów wybranych uczelni według automatycznych narzędzi.

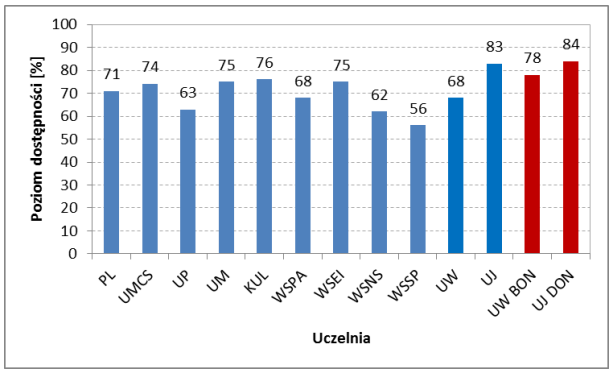
Rysunek 3 przedstawia z kolei oceny dostępności serwisów uczelni wygenerowane przez każde z osobna z czterech narzędzi (bez MAUVE++, które uległo awarii). Z wykresu tego wynika, że zdecydowanie najniższe oceny wystawiało narzędzie FAE (w granicach od 19,20 % do 42,67%). Natomiast oceny podawane przez narzędzia ACE oraz Lighhouse były wyższe od średniej oznaczonej na wykresie czerwoną linią. Narzędziem,

które w większości przypadków generowało wyniki najbardziej zbliżone do średniej było Utilitia.



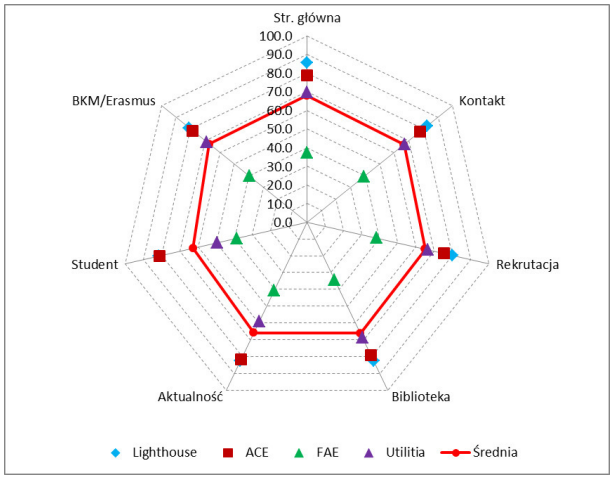
Rysunek 3: Poziom dostępności serwisów uczelni zrealizowany za pomocą automatycznych narzędzi.

Na Rysunku 4 zobrazowano wyniki analizy serwisów uczelni wyższych wykonane za pomocą narzędzia Utilitia. Warto zaznaczyć, że tylko ta aplikacja dawała możliwość jednoczesnego przebadania całych serwisów. W pozostałych przypadkach analizowane były osobno poszczególne strony. W badaniu zrealizowanym tym narzędziem serwisy: UJ i UJ DON osiągnęły najwyższe wyniki - odpowiednio na poziomie 83% i 84%.



Rysunek 4: Poziom dostępności serwisów uczelni wyższych wykonany za pomocą narzędzia Utilitia (analiza całego serwisu).

Na Rysunku 5 zaprezentowano poziomy dostępności dla siedmiu podstron przebadanych za pomocą czterech narzędzi.

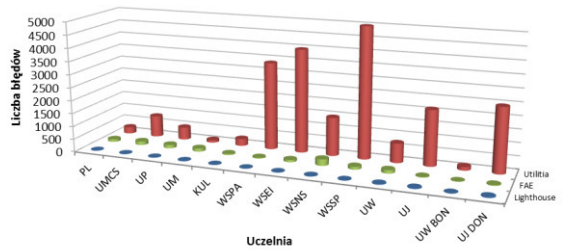


Rysunek 5: Oceny dla poszczególnych siedmiu stron wygenerowane przez automatyczne narzędzia.

Warto zauważyć, że średni poziom dostępności badanych stron, który graficznie przedstawia czerwona linia, jest na bardzo zbliżonym poziomie i oscyluje w granicach 60% - 70%. Najwyższe wyniki generowały narzędzia Lighthouse oraz ACE, a najniższe FAE.

4.3. Analiza całkowitej liczby błędów dostępności zdiagnozowanych przez automatyczne narzędzia

Rysunek 6 pokazuje liczbę błędów wykrytych przez trzy narzędzia w przebadanych stronach serwisów uczelni. Najwięcej problemów wykrywało narzędzie Utilitia i w przeważającej mierze występowały one w arkuszu stylów. Trzeba tu zaznaczyć, że poszczególne narzędzia inaczej klasyfikują błędy. Według narzędzia Utilitia najwięcej błędów zostało wykrytych w trzech serwisach: WSPA, WSEI oraz WSSP. Z kolei według narzędzia FAE najwięcej błędów znajdowało się w serwisie WSNS, a najmniej w serwisie UW BON. Narzędzie Lighthouse wykryło najwięcej błędów w serwisach: UJ DON, UJ i UW, a najmniej w UW BON oraz KUL.



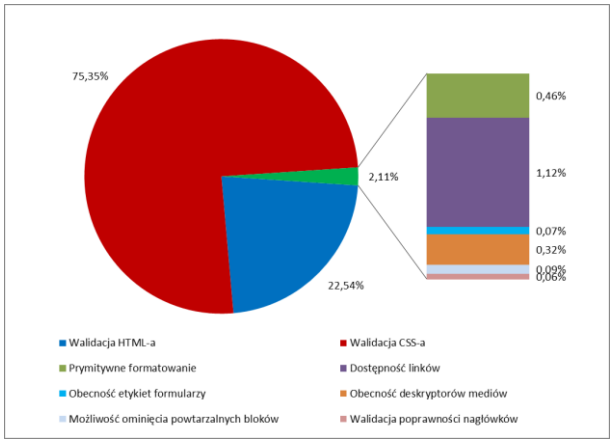
	PL	UMCS	UP	UM	KUL	WSPA	WSEI	WSNS	WSSP	UW	UJ	UW BON	UJ DON
Lighthouse	13.4	13.3	10.8	13.4	7.6	10.5	11.5	16.0	10.3	23.1	26.9	1.3	30.2
FAE	101.3	143.9	132.0	126.0	48.7	38.2	86.8	249.8	111.8	129.9	22.1	7.1	30.2
Utilitia	254.0	833.7	488.5	98.1	271.1	3377.0	3978.5	1480.8	5046.2	746.0	2130.6	137.3	2475.6

Rysunek 6: Średnia liczba błędów dla stron serwisów poszczególnych uczelni wykryta przez automatyczne narzędzia do badania dostępności.

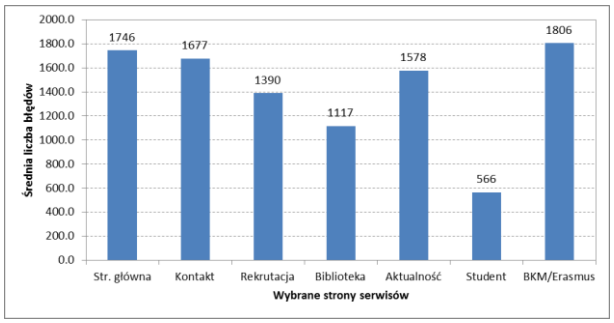
4.4. Klasyfikacja wykrytych błędów przez automatyczne narzędzia

W trzecim etapie badań wyznaczono udział poszczególnych rodzajów błędów występujących w badanych serwisach uczelni. Rysunek 7 pokazuje jakie typy błędów zostały wykryte przez narzędzie Utilitia. Najliczniejszymi dwoma grupami błędów były te, które dotyczyły walidacji plików CSS i HTML i stanowiły odpowiednio 75,35% i 22,54% wszystkich błędów. Trzecią grupą były błędy związane z dostępnością linków. Ich odsetek wynosił 1,12%.

Biorąc pod uwagę wszystkie analizowane serwisy uczelni, średnie liczby błędów dla poszczególnych stron internetowych wybranych do badań zdiagnozowane przez narzędzie Utilitia zostały przedstawione na Rysunku 8. Z wykresu wynika, że najwięcej błędów wystąpiło na stronach BKM/Erasmus, natomiast najmniej błędów odnotowano na stronach z informacjami dla studentów.

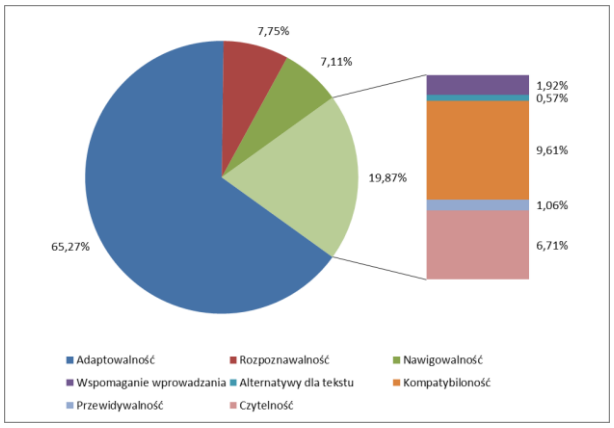


Rysunek 7: Udział poszczególnych typów błędów występujących w serwisach uczelni zdiagnozowanych za pomocą narzędzia Utilitia.



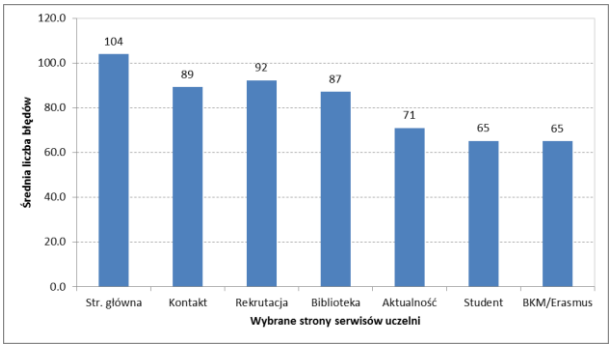
Rysunek 8: Średnia liczba błędów poszczególnych stron serwisów uczelni według narzędzia Utilitia.

Z kolei narzędzie FAE wykryło osiem rodzajów błędów. Największy odsetek, bo aż 65,27%, stanowiły błędy związane z adaptowalnością. Na drugim miejscu (7,75%) sklasyfikowano błędy dotyczące rozpoznawalności, a na trzecim (7,11%) błędy dotyczące nawigowalności (Rysunek 9).



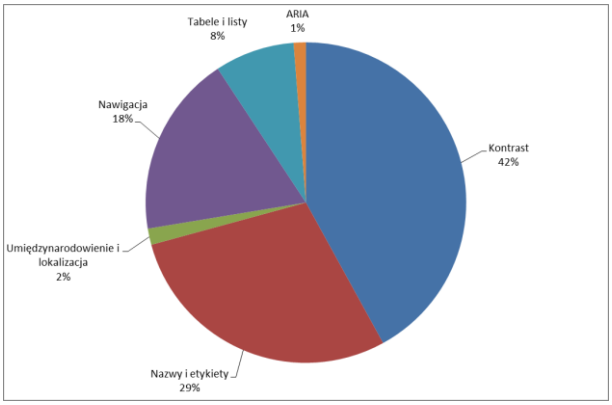
Rysunek 9: Udział poszczególnych typów błędów w badanych serwisach uczelni według narzędzia FAE.

Na Rysunku 10 pokazano, ile średnio na wybranych do badań stronach serwisów wszystkich uczelni wykryto błędów za pomocą narzędzia FAE. Z wykresu tego widać, że średnia ta jest znacznie niższa od średniej liczby błędów wykrytych przez Utilitię. Najwięcej błędów zostało wykrytych na stronie głównej (104), a najmniej na stronach BKM/Erasmus (65) oraz stronach z informacjami dla studentów (65).



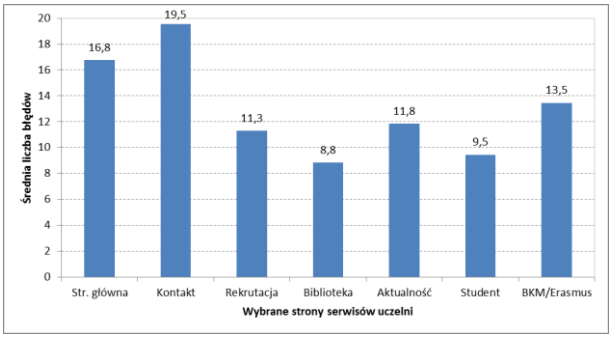
Rysunek 10: Średnia liczba błędów poszczególnych stron serwisów uczelni według narzędzia FAE.

Na Rysunku 11 zaprezentowano udział poszczególnych typów błędów, jakie zostały wykryte za pomocą narzędzia Lighthouse. W tym przypadku wykryto sześć typów błędów, które były związane z takimi elementami jak: kontrast, nazwy i etykiety, umiędzynarodowienie i lokalizacja, nawigacja, tabele i listy oraz ARIA. Najwięcej błędów było związanych z kontrastem (42%) oraz z nazwami i etykietami (29%). Trzecią pod względem wielkości grupą błędów były błędy w nawigacji. Stanowiły one 18% wszystkich błędów.



Rysunek 11: Udział poszczególnych typów błędów w serwisach uczelni według narzędzia Lighthouse.

Rysunek 12 przedstawia średnie liczby błędów dla poszczególnych stron spośród wszystkich serwisów uczelni wykrytych przez Lighthouse. W tym przypadku warto zauważyć, że średnie liczby wykrytych błędów są na niskim poziomie, wynoszącym mniej niż 20 błędów.



Rysunek 12: Średnia liczba błędów poszczególnych stron serwisów uczelni według narzędzia Lighthouse.

5. Wnioski

Dostępność serwisów uczelni wyższych ma obecnie duże znaczenie, gdyż zgodnie z prawem z 2020 roku publiczne instytucje szkolnictwa wyższego są zobligowane do uczynienia swoich serwisów dostępnymi, tzn. spełniającymi kryteria opisane w standardzie WCAG 2.1 na poziomie AA. Na podstawie wyników przedstawionych w poprzednim rozdziale, można stwierdzić, że przed uczelniami jest jeszcze wiele pracy, prowadzącej do zapewnienia odpowiedniego poziomu dostępności serwisów www. Podjęcia działań wymagają głównie następujące kwestie:

- sporządzenie i umieszczenie w serwisie deklaracji dostępności,
- opracowywanie dostępnych plików PDF,
- dodanie mechanizmu zmiany kolorystyki stron,
- wykonanie mobilnej wersji serwisu.

Badania zrealizowane w ramach tej pracy, oparte były na opracowanej metodzie wyznaczającej poziom dostępności serwisów, wykorzystującej automatyczne narzędzia oraz listę kontrolną składającą się z 13 pytań. Uzyskane wyniki pokazały, że poziom dostępności serwisów niektórych uczelni Lublina jest niski, szczególnie w porównaniu z serwisami dwóch najlepszych uczelni w Polsce. Uczelnie UJ i UW w ramach swoich serwisów mają wydzielone podserwisy skierowane do osób ze specjalnymi potrzebami. Można powiedzieć, że te specjalne serwisy wyróżniały się zdecydowanie wyższymi poziomami dostępności na tle serwisów innych uczelni, które były badane.

Opierając się na zdobytym doświadczeniu podczas opracowania i realizacji eksperymentu można wysnuć ogólny wniosek, że badanie i ocena dostępności stron www jest zadaniem trudnym i skomplikowanym. Posługując się automatycznymi narzędziami należy pamiętać, że każde z nich w odmienny sposób podchodzi do różnych aspektów dostępności. Z kolei podczas analizy eksperckiej zastosowano pewne uproszczenia i skoncentrowano się wyłącznie na podstawowych kwestiach dostępności, pomijając te, które według autorów były mniej ważne. Należy również zauważyć, że pewnym ograniczeniem tej pracy jest czas przeprowadzenia badań, które zostały zrealizowane we wrześniu 2020 roku i do momentu publikacji tego artykułu omawiane serwisy mogły ulec modernizacji i w związku z tym ich poziom dostępności mógł także się zmienić.

Literatura

[1] M. Miłosz, Ergonomia systemów informatycznych, Politechnika Lubelska, 2014, http://www.bc.pollub.pl/dlibra/publication/9004/edition/8718, [20.08.2020].

[2] W3C, Supported States and Properties, http://www.w3.org/WAI/intro/accessibility.php, [20.08.2020].

[3] B. Wit, Dostępność zawartości w serwisach WWW przedsiębiorstw przemysłowych, Technologie internetowe – od teorii do praktyki, Polskie Towarzystwo Informatyczne, Warszawa 2008, https://pti.cs.pollub.pl/wp-content/uploads/2018/06/Technologie-internetowe2008.pdf, [20.08.2020].

[4] Dyrektywa Parlamentu Europejskiego i Rady (UE) 2016/2102 z dnia 26 października 2016 r. w sprawie dostępności stron internetowych i mobilnych aplikacji organów sektora publicznego (Dziennik Urzędowy Unii Europejskiej L 327/1,2.12.2016).

[5] Dostępność cyfrowa. Omówienie wymogów dostępności cyfrowej dla podmiotów publicznych, https://www.gov.pl/web/dostepnosc-cyfrowa/omowienie-wymogow-dostepnosci-cyfrowej-dla-podmiotow-publicznych, [20.08.2020].

[6] Ustawa z dnia 4 kwietnia 2019 r. o dostępności cyfrowej stron internetowych i aplikacji mobilnych podmiotów publicznych (Dz. U. 2019 poz. 848).

[7] D. Paszkiewicz, J. Dębski, Dostępność serwisów internetowych. Dobre praktyki w projektowaniu serwisów internetowych dostępnych dla osób z różnymi niepełnosprawnościami, Stowarzyszenie Przyjaciół Integracji, Warszawa, 2013, https://www.power.gov.pl/media/13588/Dostepnosc-serwisow-internetowych-Dominik-Paszkiewicz-Jakub-Debski.pdf, [20.08.2020].

[8] Essential Components of Web Accessibility, http://www.w3.org/WAI/intro/components.php, [20.08.2020].

[9] B. Wit, D. Kuś, M. Malendowski, System informatyczny GeoAzbest. Zintegrowany system zarządzania, Wydawnictwo „Dom Orgganizatora", Toruń, 2013, http://bc.pollub.pl/Content/8644/PDF/4-system.pdf, [20.08.2020].

[10] S. Rahmatizadeh, S. Valizadeh-Haghi, Monitoring for accessibility in medical university websites: Meeting the needs of people with disabilities, Journal of Accessibility and Design for All, 8 (2), (2018) 102-124, , http://www.jacces.org/index.php/jacces/article/view/150/201, [20.08.2020].

[11] M. S. Hassouna, N.Sahari, A. Ismail, University website accessibility for totally blind users, Journal of Information and Communication Technology, 16 (1) (2017) 63-80, http://e–journal.uum.edu.my/index.php/jict/article/view/8218/1236, [22.08.2020].

[12] B. Abu Shawar, Evaluating Web Accessibility of Educational Websites, International Journal of Emerging Technology in Learning, 10 (4), (2015), https://online-journals.org/index.php/i-jet/article/view/4518/3582, [20.08.2020].

[13] Lighthouse, https://developers.google.com/web/tools/lighthouse, [20.09.2020].

[14] Accessibility Evaluator (ACE), https://ace.accessibe.com/, [20.09.2020].

[15] Multiguideline Accessibility and Usability Validation Environment (MAUVE++), https://mauve.isti.cnr.it/, [20.09.2020].

[16] Functional Accessibility Evaluator 2.1 (FAE), https://fae.disability.illinois.edu/anonymous/?Anonymous%20Report=, [20.09.2020].

[17] Utilitia, https://utilitia.pl, [20.09.2020].

[18] Ranking Uczelni Akademickich 2020, http://ranking.perspektywy.pl/2020/ranking/ranking-uczelni-akademickich, [20.09.2020].

[19] Serwis Politechniki Lubelskiej, http://www.pollub.pl, [20.09.2020].

[20] Serwis Uniwersytetu Marii Curie-Skłodowskiej, https://www.umcs.pl, [20.09.2020].

[21] Serwis Uniwersytetu Przyrodniczego w Lublinie, https://www.up.lublin.pl, [20.09.2020].

[22] Serwis Uniwersytetu Medycznego w Lublinie, https://www.umlub.pl, [20.09.2020].

[23] Serwis Katolickiego Uniwersytetu Lubelskiego, https://www.kul.pl, [20.09.2020].

[24] Serwis Wyższej Szkoły Przedsiębiorczości i Administracji w Lublinie, https://wspa.pl, [20.09.2020].

[25] Serwis Wyższej Szkoły Ekonomii i Innowacji w Lublinie, https://www.wsei.lublin.pl, [20.09.2020].

[26] Serwis Wyższej Szkoły Nauk Społecznych w Lublinie, https://www.wsns.lublin.pl, [20.09.2020].

[27] Serwis Wyższej Szkoły Społeczno-Przyrodniczej im. Wincentego Pola w Lublinie, https://wssp.edu.pl, [20.09.2020].

[28] Serwis Uniwersytetu Warszawskiego, https://www.uw.edu.pl, [20.09.2020].

[29] Serwis Uniwersytetu Jagiellońskiego, https://www.uj.edu.pl, [20.09.2020].

[30] Serwis Działu ds. Osób Niepełnosprawnych Uniwersytetu Jagiellońskiego, https://don.uj.edu.pl, [20.09.2020].

[31] Serwis Biura ds. Osób Niepełnosprawnych Uniwersytetu Warszawskiego, https://bon.uw.edu.pl, [20.09.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 89–94
Received: 06 February 2021
Accepted: 13 February 2021

REST and GraphQL comparative analysis

Analiza porównawcza technologii REST i GraphQL

Piotr Margański*, Beata Pańczyk

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

Abstract

The article presents a comparative analysis of the two most commonly used API web design standards - REST and GraphQL. The time and size of HTTP responses returned by applications were tested. Two applications with the same functionalities, performing CRUD operations, on data stored in the non-relational MongoDB database were used for the research. Both applications were based on NodeJS technology. The JMeter tool was used to collect and analyze the data. On the basis of the obtained results, it was found that there were no significant differences in reading the data with a small number of queries and when removing resources. With the increase in the number of queries, a clear advantage of the REST standard was observed. The advantage of GraphQL, both in response time and size, was demonstrated when retrieving specific data.

Keywords: REST; GraphQL; API; application performance

Streszczenie

W artykule przeprowadzono analizę porównawczą dwóch najczęściej stosowanych standardów projektowania internetowego API – REST oraz GraphQL. Badano czas oraz rozmiar odpowiedzi HTTP zwracanych przez aplikacje. Do badań wykorzystano dwie aplikacje o takich samych funkcjonalnościach, realizujących operacje CRUD, na danych przechowywanych w nierelacyjnej bazie MongoDB. Obie aplikacje stworzono w oparciu o technologię NodeJS. Do zebrania i analizy danych zastosowano narzędzie JMeter. Na podstawie otrzymanych wyników stwierdzono brak znacznych różnic w odczycie danych przy małej liczbie zapytań oraz podczas usuwania zasobów. Wraz ze wzrostem liczby zapytań zaobserwowano wyraźną przewagę standardu REST. Przewagę GraphQL, zarówno w czasie jak i rozmiarze odpowiedzi, wykazano w przypadku pobierania specyficznych danych.

Słowa kluczowe: REST; GraphQL; API; wydajność aplikacji

*Corresponding author

Email address: **piotr.marganski@pollub.edu.pl** (P. Margański)

1. Wstęp

Wraz z coraz większą świadomością użytkowania aplikacji internetowych, wymaga się od nich dwóch podstawowych cech - niezawodności oraz szybkości. Aplikacje te zazwyczaj nie są pojedynczym programem wykonywalnym. Najczęściej obecnie stosowanym podejściem jest budowa aplikacji oparta na trójwarstwowej architekturze - frontend, backend i baza danych [1]. Elementy te muszą się ze sobą komunikować w odpowiedni i stabilny sposób. W tym celu wprowadzono pojęcie API (ang. Application Programming Interface). Jest to pewnego rodzaju zbiór standardów i protokołów umożliwiający wymianę danych pomiędzy wspomnianymi warstwami [2]. Obecnie można znaleźć i użyć, w zależności od decyzji architektonicznych, wielu implementacji tego interfejsu - między innymi REST, SOAP i GraphQL.

Najczęściej stosowanym standardem jest implementacja REST (ang. REpresentational State Transfer). Zaprezentowany w 2000 roku przez Roya Fieldinga był przedmiotem jego rozprawy doktorskiej [3]. REST nie jest technologią, ale zbiorem zasad narzucanych projektantowi API w taki sposób, aby tworzona implementacja opierała się na koncepcie zasobów identyfikowanych poprzez URI (ang. Uniform Resource Identifier).

W 2015 roku pojawiło się konkurencyjne podejście, rozwijana przez zespół deweloperski Facebooka technologia GraphQL [4]. Jej twórcy przedstawiali ją jako elastyczne rozwiązanie charakteryzujące się bardziej nowoczesnym podejściem do pracy z danymi podczas tworzenia usług sieciowych. Największe zainteresowanie budziła możliwość specyfikacji danych, które programista chce otrzymać, ponieważ jest to funkcjonalność, której REST a wcześniej SOAP nie były w stanie zapewnić.

Wobec tak spektakularnego pojawienia się nowego rozwiązania tworzenia API pojawiają się pytania dotyczące efektywności i łatwości implementacji. Zaprezentowane w niniejszym artykule wyniki mają na celu odpowiedzieć na te pytania i ułatwić wybór projektantom API szukającym odpowiedniego dla siebie rozwiązania.

1.1. Cel i obiekt badań

Celem badań jest analiza porównawcza dwóch implementacji API - REST oraz GraphQL. Analiza dotyczy porównania wydajności weryfikowanej czasem oraz rozmiarem odpowiedzi na żądania protokołu HTTP.

W zakres pracy wchodzi przedstawienie podstawowych pojęć, implementacja aplikacji do

przeprowadzenia testów, wykonanie testów, analiza wyników i sformułowanie płynących z niej wniosków.

W artykule podjęto próbę zweryfikowania następujących hipotez:

- Czas odpowiedzi na zapytanie jest krótszy a rozmiar odpowiedzi jest mniejszy w przypadku zorientowanego na zapytania GraphQL niż w powszechnie używanym REST.
- REST jest efektywniejszy od GraphQL w przypadku pobrania wszystkich danych.
- GraphQL będzie wydajniejszy niż REST tylko wtedy, gdy żądanie będzie dotyczyło pobierania konkretnych danych..

1.2. Przegląd literatury

Analiza dostępnej literatury dotyczącej REST, GraphQL i szeroko pojętego projektowania API jednoznacznie wskazuje, że dyskusje na temat najbardziej optymalnego rozwiązania dotyczącego tworzenia usług sieciowych wystawiających API nie są rozstrzygnięte. Przedmiot tej dyskusji jest wciąż poddawany badaniom i pojawia się w coraz bardziej licznych publikacjach naukowych. Jednym z wielu aspektów, który jest istotny przy wyborze odpowiedniego rozwiązania jest doświadczenie programistów, znajomość danej technologii a przede wszystkim stopień trudności zapoznania się z nią.

Zbadanie ostatniego punktu było przedmiotem badań Brito i Valente z Uniwersytetu Minas Gerais w Brazylii [5]. Na próbie 22 studentów (12 magistrantów oraz 10 studentów studiów licencjackich) dowiedli, że GraphQL wymaga mniej wysiłku i czasu do utworzenia usługi sieciowej niż REST nawet w przypadku osób, które w pracy z REST posiadały już doświadczenie.

Pontus Erlandsson oraz Joakim Remes z Uniwersytetu Skovde w Szwecji w swojej pracy licencjackiej dokonali analizy porównawczej GraphQL, REST oraz SOAP względem wydajności [4]. Osiągnięte przez nich rezultaty wykazały, że najsłabszą wydajnością spośród trzech wymienionych charakteryzuje się GraphQL. Jedynym przypadkiem, w którym czas odpowiedzi na zapytanie był najkrótszy, a jej rozmiar najmniejszy było pobieranie wybranych pól zasobu, niemniej jednak wraz ze wzrostem ich ilości czas się wydłużał a rozmiar się zwiększał, co w sposób negatywny zbliżało GraphQL do wyników osiąganych przez REST.

Zaproponowane przez C. Oggier z Haaga-Helia University of Applied Sciences badanie miało na celu zbadanie czy GraphQL jest szybszy i bardziej zoptymalizowany niż REST [6]. Osiągnięte przez nią rezultaty jednoznacznie wskazują na odpowiedź twierdzącą. W obu z przeprowadzonych testów, z których pierwszy badał żądania przesyłane sekwencyjne, a drugi równolegle, GraphQL osiągał krótsze czasy i mniejsze rozmiary odpowiedzi. W pierwszym przypadku GraphQL był szybszy o 46% i lżejszy o 21% niż REST, natomiast w przypadku drugim było to odpowiednio 35% i 71%.

W pracy dyplomowej API Design in Distributed Systems: A comparison between GraphQL and REST Thomas, Eizinger z University of Applied Sciences Technikum w Wiedniu wskazuje na kolejny znaczący obszar porównania [7]. Wykazuje, że GraphQL w zdecydowanym stopniu przenosi odpowiedzialność za pobierane i wyświetlane dane na stronę klienta, co prowadzi do zwiększenia jego złożoności.

Artykuł REST or GraphQL?: A Performance Comparative Study autorstwa M. Seabra, M.F. Nazario oraz G. Pinto, przedstawia badania dotyczące wydajności trzech aplikacji, mierzonej parametrami czasu oraz średniej szybkości transferu między żądaniami [8]. Każda z aplikacji została zbudowana wykorzystując architekturę REST, a następnie po wykonaniu pomiarów dokonano migracji do GraphQL. Badania wykazały wzrost wydajności po migracji w dwóch z trzech testowanych aplikacji. W przypadku liczby do 100 żądań oba standardy osiągały zbliżone wyniki w zakresie 6,34 - 7,68 żądań na jedną sekundę, jednak wraz ze wzrostem liczby żądań, GraphQL tracił przewagę, a po przekroczeniu granicy 3000 wydajność REST była już na zdecydowanie większym poziomie. Przedstawione badania nie rozstrzygają jednoznacznie, która z omawianych metod tworzenia API jest bardziej wydajna. Należy pamiętać, że wydajność nie jest jedynym kryterium decydującym o wyborze technologii do budowy architektury systemu. Ważnymi czynnikami są również doświadczenie zespołu, wysokość tak zwanego progu wejścia i potrzeby systemu.

2. Metoda badań

2.1. Aplikacje testowe

W celu przeprowadzenia badań zaimplementowano dwie aplikacje wystawiające API. Jedna z nich wykorzystuje technologię GraphQL, druga implementuje REST. Obie zostały stworzone w środowisku Node.js i korzystają ze wspólnej, nierelacyjnej bazy danych MongoDB. Dane przechowywane w bazie odzwierciedlały sytuację w tabeli angielskiej ligi piłkarskiej Premier League po każdej z dotychczas rozegranych kolejek spotkań. Listing 1 przedstawia definicję modelu pojedynczej kolekcji w Mongo (w relacyjnych bazach danych zwanej tabelą).

Listing 1: Kod programu - model tabeli bazy danych

```
const eventSchema = new Schema(
  {
    name: String,
    league_id: Number,
    season_id: Number,
    stage_id: Number,
    standings: [
      {
        position: Number,
        team_id: Number,
        team_name: String,
```

```
        overall: {
          games_played: Number,
          wins: Number,
          draws: Number,
          losses: Number,
          goals_scored: Number,
          goals_conceded: Number
        },
        home: {
          games_played: Number,
          wins: Number,
          draws: Number,
          losses: Number,
          goals_scored: Number,
          goals_conceded: Number
        },
        away: {
          games_played: Number,
          wins: Number,
          draws: Number,
          losses: Number,
          goals_scored: Number,
          goals_conceded: Number
        },
        total: {
          goal_difference: String,
          points: Number
        },
        points: Number,
        recent_form: String,
        status: String
      }
    ]
  }
);
```

Połączenie z bazą danych zostało zrealizowane za pomocą pakietu *mongoose* w wersji 5.10.11. Przykładowe fragmenty kodu mające na celu utworzenie zasobu w bazie danych w technologii GraphQL zostały przedstawione na Listingach 2 i 3.

Listing 2: Dodawanie zasobu w GraphQL

```
createStandings: (args) => {
  const standings = new Standings(
   {...args.standings}
  );
  return standings.save();
}
```

Listing 3: Mutacje GraphQL możliwe do wykonania

```
type Mutation {
 createStandings(standings: StandIn):
  Standings
deleteStandings(id: ID!): Standings
}
```

Listingi 4 i 5 przedstawiają tą samą operację zdefiniowaną w usłudze opartej na architekturze REST.

Listing 4: Dodawanie zasobu w REST

```
exports.create = (req, res) => {
  const standings = new Standings(
   {...req.body}
  );
  standings.save()
   .then(data => res.send(data))
   .catch(err => res.status(500)
    .send({ message: err.message }));
}
```

Listing 5: Endpoint dla dodania zasobu w REST

```
app.post('/standings', controller.create)
```

Aby zbadać wydajność obu usług wykorzystano zaimplementowaną w Javie aplikację Apache JMeter w wersji 5.3, która pozwala na zebranie zdefiniowanie, wykonanie i zebranie wyników puli zapytań, które wykonywane są w sposób wielowątkowy [9]. Żądania definiowane są dokładnie w taki sposób jak w standardowej aplikacji klienckiej, to znaczy podając protokół, domenę, numer portu, rodzaj zapytania oraz nazwę dla endpoint, pod który wysyłane jest zapytanie. Opcjonalnie podaje się parametry i ciało zapytania. Aplikacja ta w przeciwieństwie do przeglądarki nie wykonuje kodu JavaScript ani nie renderuje HTML, wobec tego nie trzeba brać pod uwagę ewentualnych zaburzeń w uzyskiwanych czasach spowodowanych tymi właśnie działaniami.

2.2. Środowisko testowe

W celu maksymalnego ograniczenia możliwych opóźnień generowanych przez sieć przeprowadzono badania na jednej maszynie posiadającej następujące parametry:

- Procesor - Intel(R) Core(™) i5-5200U, 2.20 GHz,
- Pamięć RAM - 8 GB,
- Typ systemu: 64-bit,
- System operacyjny - Windows 10 Education.

2.3. Scenariusze testowe

W ramach eksperymentu badawczego zrealizowano cztery scenariusze przedstawione w Tabeli 1.

Tabela 1: Scenariusze testowe

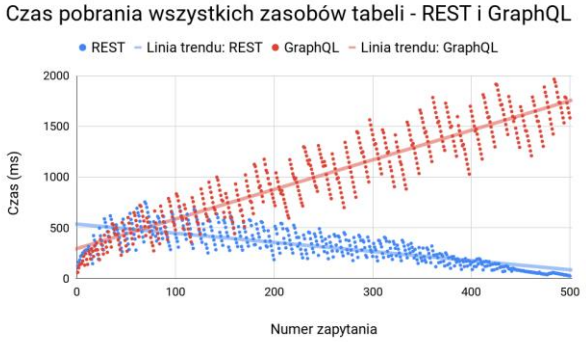
Lp.	Scenariusz
S1.	Zmierzenie czasu oraz rozmiaru odpowiedzi serwera na żądanie wszystkich zasobów danej tabeli.
S2.	Zmierzenie czasu oraz rozmiaru odpowiedzi serwera na żądanie pojedynczego zasobu z wyszczególnieniem konkretnych pól w przypadku GraphQL.
S3.	Zmierzenie jaki czas zajmie serwerowi przetworzenie zapytania POST, które dodaje pojedynczy rekord do bazy danych.
S4.	Zmierzenie jaki czas będzie potrzebował serwer aby usunąć wskazanego za pomocą identyfikatora zasób z bazy.

Jak wyszczególniono na początku artykułu, celem przeprowadzonych badań była analiza porównawcza omawianych technologii, szczególnie względem ich wydajności, z czasem i rozmiarem odpowiedzi jako głównymi jej parametrami. Badanie wydajności aplikacji internetowych jest zadaniem nietrywialnym, ze względu na dużą liczbę czynników, które mogą prowadzić do zaburzenia otrzymywanych wyników, między innymi obciążenia łącza internetowego lub szybkość reakcji. W związku z tym zdecydowano się na symulację działania potencjalnych użytkowników za pomocą narzędzia JMeter w izolowanym środowisku.

3. Wyniki badań

3.1. Wyniki testu pierwszego

W ramach testu pierwszego wykonano 500 zapytań w czasie 5 sekund co przekładało się na wykonanie kolejnego zapytania średnio co 0,01 sekundy. Na Rysunku 1 przedstawiono wyniki w formie wykresu.



Rysunek 1: Czas odpowiedzi REST i GraphQL na żądanie GET wszystkich zasobów.

Wyraźnie widoczna jest różnica czasu przetworzenia żądania oraz tendencja spadkowa wraz ze wzrostem liczby kolejnych zapytań w przypadku REST. Przedstawiono również szczegółowe dane dotyczące minimalnego, maksymalnego oraz średniego czasu i rozmiaru odpowiedzi (Tabela 2).

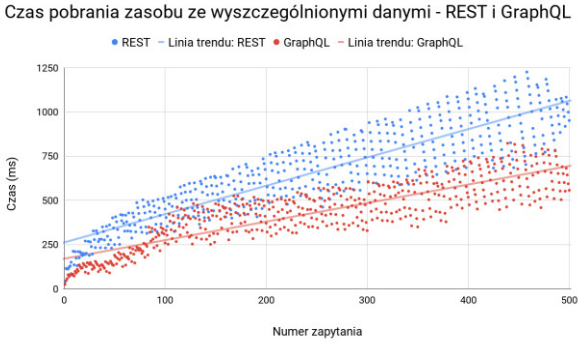
Tabela 2: Wyniki - pobieranie wszystkich zasobów

	REST		GraphQL	
Parametr	czas (ms)	rozmiar (B)	czas (ms)	rozmiar (B)
min	24	9533	59	9554
max	754	9533	1965	9554
średnia	309	9533	1024	9554

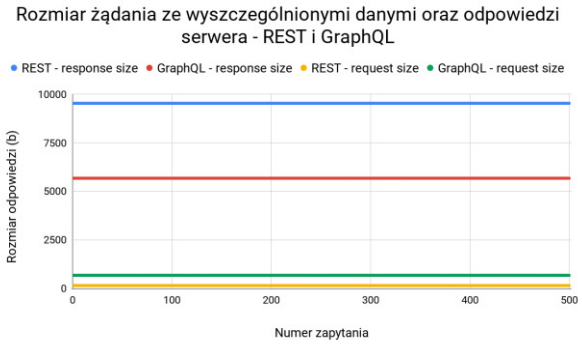
3.2. Wyniki testu drugiego

Drugi test, którego celem było pobranie danych wyszczególnionych w żądaniu, polegał na wysłaniu 500 zapytań w czasie 5 sekund. Rysunek 2 przedstawia wyniki czasu odpowiedzi serwerów, natomiast na Rysunku 3 przedstawiono rozmiar odpowiedzi.

Wyniki tego testu bardzo wyraźnie uwidaczniają powód tak zdecydowanego i dynamicznego wejścia w obszar technologii umożliwiającej projektowanie API. Najbardziej charakterystyczny element GraphQL, a więc pobranie tylko tych danych, które są potrzebne w sposób bezdyskusyjny pozwala na osiąganie krótszych czasów i mniejszych rozmiarów odpowiedzi. Tabela 3 w szczegółowy sposób prezentuje dane dotyczące minimalnego, maksymalnego oraz średniego czasu i rozmiaru odpowiedzi z obydwu usług.



Rysunek 2: Czas przetwarzania żądania konkretnych danych w REST i GraphQL.



Rysunek 3: Rozmiar odpowiedzi serwera REST i serwera GraphQL na żądanie konkretnych danych.

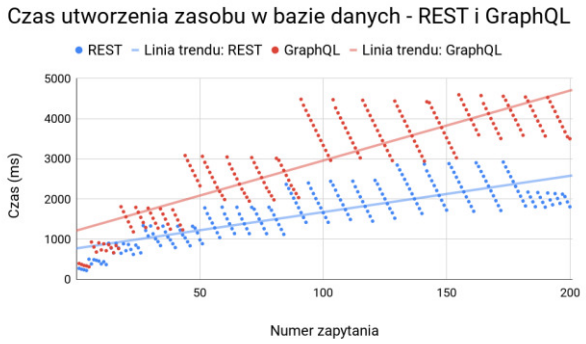
Tabela 3: Wyniki - pobieranie wybranych danych

	REST		GraphQL	
Parametr	czas (ms)	rozmiar (B)	czas (ms)	rozmiar (B)
min	41	9531	25	5674
max	1226	9531	823	5674
średnia	663	9531	432	5674

3.3. Wyniki testu trzeciego

Trzeci test, to wykonanie 200 żądań w czasie 5 sekund, których celem było utworzenie zasobu w bazie danych.

Rysunek 4 przedstawia otrzymane wyniki w formie wykresu, natomiast Tabela 4 prezentuje szczegółowe wyniki tego testu. Zauważono niepodważalną przewagę REST. Czas potrzebny na realizację żądania jest średnio o ponad połowę krótszy niż w przypadku GraphQL (56%).



Rysunek 4: Czas przetwarzania żądania POST w REST i GraphQL.

Tabela 4: Wyniki - dodanie zasobu do bazy danych

	REST		GraphQL	
Parametr	czas (ms)	rozmiar (B)	czas (ms)	rozmiar (B)
min	220	9531	312	275
max	2920	9531	4596	275
średnia	1678	9531	2967	275

3.4. Wyniki testu czwartego

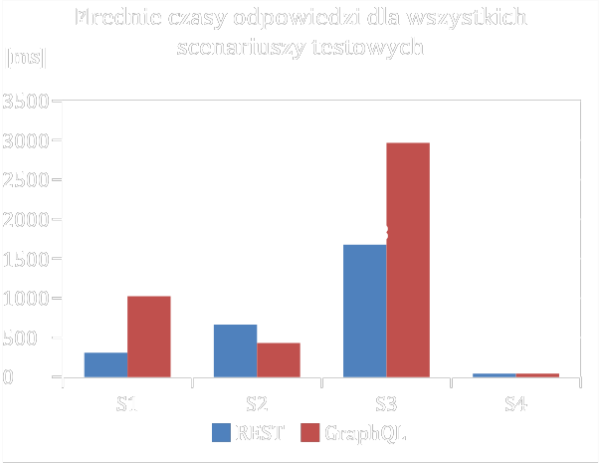
Celem testu czwartego było zmierzenie czasu potrzebnego serwerowi na usunięcie wskazanego zasobu z bazy danych. Aby to osiągnąć wykonano 50 takich żądań, których wyniki zebrano w Tabeli 5. Warto w tym miejscu wspomnieć, że takie żądanie nie jest łatwe do zoptymalizowania ze względu na konieczność podania innego identyfikatora w każdym żądaniu.

Tabela 5: Wyniki - usuwanie zasobu z bazy danych

	REST		GraphQL	
Parametr	czas (ms)	rozmiar (B)	czas (ms)	rozmiar (B)
min	38	244	39	275
max	49	244	46	275
średnia	44	244	43	275

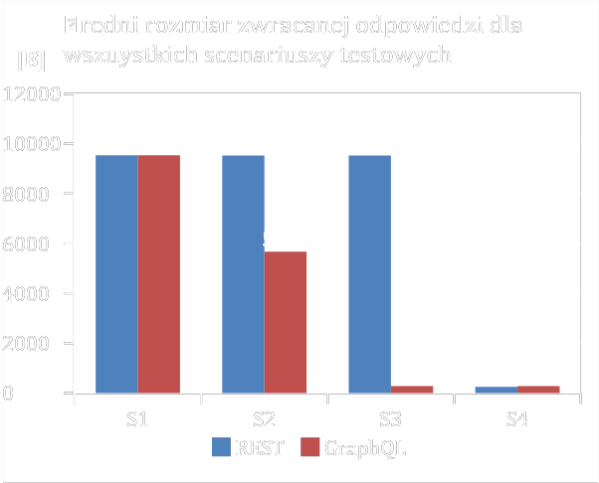
Osiągnięte rezultaty nie wykazują statystycznie istotnych różnic zarówno w czasie, jak i w rozmiarze odpowiedzi serwera. Można więc z całkowitym przekonaniem stwierdzić, że w przypadku usunięcia zasobu z bazy danych REST i GraphQL wykazują różnic w wydajności.

Rysunek 5 przedstawia średnie czasy potrzebne na przetworzenie żądania dla każdego z przeprowadzonych testów. Można zauważyć, że niezależnie od zastosowanego standardu najdłuższym czasem wykonania charakteryzował się przypadek utworzenia zasobu, natomiast najkrótszy czas osiągano w przypadku usunięcia zasobu z bazy danych.



Rysunek 5: Czas przetwarzania żądania POST w REST i GraphQL.

Przedstawiony na Rysunku 6 wykres przedstawia średni rozmiar odpowiedzi zwrotnej z serwera dla każdego z przeprowadzonych scenariuszy testowych.



Rysunek 6: Średni rozmiar odpowiedzi zwrotnej z serwera dla każdego scenariusza testowego.

4. Wnioski

Przeprowadzona analiza miała na celu wskazanie, który z badanych standardów projektowania programistycznego interfejsu aplikacji jest wydajniejszy. Na podstawie uzyskanych wyników można jednoznacznie stwierdzić że w przypadku pobierania wszystkich dostępnych zasobów z danej tabeli oraz tworzenia nowego zasobu w bazie danych REST jest zdecydowanie wydajniejszy od GraphQL. Niemniej jednak należy zaznaczyć że przewaga ta widoczna jest w przypadku przekroczenia liczby 100 żądań. Do wspomnianej granicy obie implementacje wykazują czas oraz rozmiar odpowiedzi na zbliżonym poziomie. Analizując wyniki czwartego testu dotyczącego usunięcia zasobu z bazy danych również zauważalny jest brak przewagi któregoś z serwisów. Różnica w czasie i rozmiarze odpowiedzi zwrotnej jest w tym przypadku statystycznie nieistotna. Ostatnią rzeczą, którą wykazała przeprowadzana analiza jest

zdecydowana przewaga GraphQL nad REST w czasie i rozmiarze odpowiedzi w przypadku ograniczenia ilości pobieranych danych. Wynika to wprost z możliwości i specyfikacji standardów. Stosując rozwiązanie oparte na REST użytkownik musi liczyć się z dodatkowym narzutem danych.

Otrzymane rezultaty pozytywnie weryfikują dwie z trzech postawionych hipotez - REST okazał się wydajniejszy w przypadku pobierania wszystkich zasobów z bazy, natomiast GraphQL był efektywniejszy wyłącznie w przypadku pobierania wyszczególnionych danych. Wyniki te są analogiczne do rezultatów zaprezentowanych w pracy P. Erlandssona i J. Remesa [4]. Hipoteza dotycząca osiągania krótszych czasów i mniejszych rozmiarów odpowiedzi przez GraphQL została zweryfikowana w sposób negatywny. Przeprowadzone badania nie potwierdzają rezultatów otrzymanych przez C. Oggier [6], jednak zbieżne są z wynikami eksperymentu zaprezentowanego w publikacji [8].

Biorąc pod uwagę otrzymane wyniki można stwierdzić, że wybór sposobu i technologii do projektowania API musi spotkać się z wymaganiami stawianymi przez klientów systemu, między innymi rozmiarem aplikacji i liczbą potencjalnych użytkowników. Bardzo istotnym czynnikiem jest również komfort pracy programisty implementującego rozwiązanie - w tym celu, aby ułatwić decyzję dotyczącą wyboru standardu, przedstawiono możliwości i przybliżono specyfikację zarówno REST jak i GraphQL.

Literatura

[1] M. Miłosz, Aplikacje internetowe - od teorii do praktyki, Polskie Towarzystwo Informatyczne, Warszawa, 2008.

[2] What is an API?, Mulesoft, https://www.mulesoft.com/resources/api/what-is-an-api, [22.01.2021].

[3] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, University of California, Irvine, 2020, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, [01.12.2020].

[4] P. Erlandsson, J. Remes, Performance Comparison between GraphQL, REST & SOAP, University of Skovde, Dissertation, http://his.divaportal.org/smash/record.jsf?pid=diva2%3A1449837&dswid=7389, 2020, [19.01.2021].

[5] G. Brito, M. T. Valente, REST vs GraphQL: A Controlled Experiment, IEEE International Conference on Software Architecture (ICSA), Salvador, Brazil, (2020) 81-91, doi: 10.1109/ICSA47634.2020.00016, [21.01.2021].

[6] C. Oggier, How fast GraphQL is compared to REST APIs, Haaga-Helia University of Applied Sciences, 2020, http://urn.fi/URN:NBN:fi:amk-2020052714286, [28.01.2021].

[7] T. Eizinger, API Design in Distributed Systems: A Comparison between GraphQL and REST, Master Thesis, University of Applied Science Technikum Wien, 2017, [20.01.2021].

[8] M. Seabra, M. E. Nazario, G. Pinto, REST or GraphQL?: A Performance Comparative Study, Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse. Association for Computing Machinery, New York, NY, USA, 2019, https://doi.org/10.1145/3357141.3357149, [29.01.2021].

[9] Apache JMeter, https://jmeter.apache.org/, [20.11.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 95–99
Received: 18 February 2021
Accepted: 03 March 2021

Comparative analysis of performance of ASP.NET Core MVC and Symfony 4 programming frameworks

Wydajnościowa analiza porównawcza szkieletów programistycznych ASP.NET Core MVC i Symfony 4

Marcin Górski* , Wojciech Andrzej Piwowarski, Mariusz Dzieńkowski

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

Abstract

The article presents a comparative analysis of popular ASP.NET Core MVC and Symfony 4 frameworks. Two web applications, containing the same functionalities and acting as a simple system for managing articles, were implemented in these technologies. The applications underwent time performance tests during typical operations performed by means of a simple form such as entering, editing, viewing and deleting data. These actions were performed automatically using commands from the Puppeteer library. The listed operations were repeated 10, 100 and 1,000 times in order to obtain precise mean times. On the basis of the obtained results, it was difficult to clearly state which of the compared programming tools is better. The ASP.NET Core MVC framework coped much better with two time-consuming operations, i.e. entering and editing data. Its results in this regard (the average from 1,000 repetitions) were respectively approximately 28% and 25% better compared to the Symphony 4 framework. However, for the two less time-consuming operations, i.e. displaying and deleting articles, the Symphony 4 framework proved to be considerably better. Its results with regard to displaying and deleting articles (the average for 1,000 measurements) were respectively 15% and 36% lower compared to the other of the tested frameworks.

Keywords: performance analysis; automatic testing; framework; ASP.NET Core MVC; Symfony; Puppeteer

Streszczenie

Artykuł przedstawia analizę porównawczą popularnych szkieletów programistycznych ASP.NET Core MVC oraz Symfony 4. W technologiach tych zaimplementowano dwie aplikacje internetowe, zawierające te same funkcjonalności, pełniące funkcję prostego systemu do zarządzania artykułami. Te aplikacje zostały poddane testom wydajności czasowej podczas realizacji typowych operacji wykonywanych za pośrednictwem prostego formularza takich jak wprowadzanie, edycja, wyświetlanie i usuwanie danych. Czynności te były wykonywane automatycznie za pomocą poleceń z biblioteki Puppeteer. Wyszczególnione operacje były powtarzane 10, 100 i 1000 razy w celu uzyskania precyzyjnych średnich czasów. Na podstawie otrzymanych wyników trudno było jednoznacznie stwierdzić, które z porównywanych narzędzi programistycznych jest lepsze. Z dwiema czasochłonnymi operacjami tzn. wprowadzaniem i edycją danych, znacznie lepiej radził sobie framework ASP.NET Core. Jego wyniki pod tym względem (średnia z 1000 powtórzeń) były odpowiednio o około 28% i 25% lepsze w stosunku do szkieletu Symfony 4. Natomiast dla dwóch mniej czasochłonnych operacji, czyli wyświetlania i usuwania artykułów, wyraźnie lepszym okazał się szkielet Symfony 4. Jego wyniki dla wyświetlania i usuwania artykułów (średnia dla 1000 pomiarów) były o 15 i 36 procent odpowiednio niższe w stosunku do drugiego badanego szkieletu.

Słowa kluczowe: analiza wydajnościowa; testowanie automatyczne; ASP.NET Core MVC; Symfony; Puppeteer

*Corresponding author

Email address: marcin.gorski1@pollub.edu.pl (M. Górski)

1. Wstęp

Przed przystąpieniem do realizacji projektu programistycznego ważną kwestią, która może nawet decydować o jego powodzeniu lub porażce, jest dobór odpowiednich technologii. Duża liczba języków i opartych na nich szkieletów programistycznych nie ułatwia tego zadania. Ważnym aspektem, który należy uwzględnić przy podejmowaniu decyzji o wyborze technologii jest jej popularność, którą można sprawdzić za pomocą licznych rankingów i narzędzi porównawczych publikowanych na bieżąco w Internecie, np. TIOBE Index [1] czy Google Trends [2]. Za popularnością często idzie wielkość społeczności wspierającej daną technologię. Większa społeczność daje gwarancje, że technologia będzie rozwijana i będzie łatwiejsza w utrzymaniu.

W procesie wytwarzania oprogramowania bardzo ważnym etapem jest jego testowanie, którego celem jest wykrywanie ewentualnych awarii, będących skutkiem występujących w kodzie błędów. Testowanie prowadzi do zwiększania niezawodności oprogramowania i wpływa na jego jakość. Może ono być wykonywane w sposób manualny - przez wykwalifikowanych testerów lub w sposób automatyczny za pomocą specjalnych narzędzi informatycznych oraz bibliotek. Testy automatyczne, nie wymagają angażowania wielu specjalistów, przez co nie są czasochłonne i kosztochłonne. Raz opracowany test może być uruchamiany wielokrotnie [3].

Jednak istnieją pewne przypadki, które wymagają ręcznej weryfikacji. Takim przykładem może być realizacja złożonych scenariuszy testowych, których automatyzacja jest nieopłacalna. Natomiast w przypadku gdy testy będą wielokrotnie powtarzane, gdy będą wykonywane na wielu zestawach danych, na różnych platformach sprzętowych i programistycznych oraz przy różnych konfiguracjach, wówczas automatyzacja jest jak najbardziej wskazana. Także takie względy jak czasochłonność i niemożność ręcznej weryfikacji przemawiają za zautomatyzowaniem procesu testowania [4].

Twórcy systemów internetowych, szukają najlepszych narzędzi do budowy swoich aplikacji. Z takiego właśnie powodu narodził się pomysł przeprowadzenia w ramach tej pracy analizy dwóch popularnych technologii webowych: ASP.NET Core MVC oraz Symfony 4. W tym celu opracowano eksperyment, w którym zbadano dwie aplikacje webowe, jedna zbudowana na bazie szkieletu programistycznego ASP.NET Core MVC, a druga na podstawie Symfony 4. Badania dotyczyły czasowej oceny wydajności obu rozwiązań.

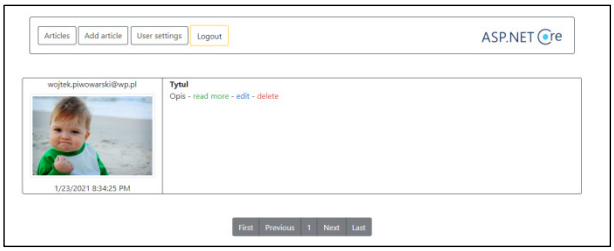
2. Porównywane technologie

Symfony jest darmowym i popularnym w Polsce szkieletem programistycznym. Został on utworzony w języku skryptowym PHP i jest w pełni zorientowany obiektowo. Szkielet wspomaga programowanie dynamicznie generowanych aplikacji webowych, dzięki udostępnianiu gotowej architektury opartej na wzorcu projektowym MVC. Symfony nie jest zależne od konkretnego systemu bazy danych i w związku z tym można skonfigurować i podłączyć dowolną bazę. Szkielet ten został stworzony w oparciu o najlepsze standardy oraz wzorce budowania aplikacji www. Wyposażony jest w podstawowe funkcje gotowe do użycia w aplikacjach, np. walidacja formularzy, zarządzanie sesjami czy autoryzacja. Dzięki narzędziu Composer szybko można rozbudować aplikację oraz dołączyć dodatkowe biblioteki. Symfony ma wbudowany silnik szablonów twig, który jest odpowiedzialny za renderowanie widoków [5].

ASP.NET Core jest darmową, wieloplatformową, wysokowydajną i otwartą platformą programistyczną typu „Open Source", umożliwiającą tworzenie nowoczesnych aplikacji z obsługą chmury. Zaletą ASP.NET Core jest to, że praktycznie po każdej aktualizacji pojawiają się nowe funkcje oraz wzbogacane są funkcje już istniejące, które pozwalają na budowę wysoce skalowalnych i wydajnych aplikacji internetowych. W samym szkielecie dostępnych jest bardzo dużo funkcji, które pozwalają na rozwiązywanie typowych problemów stających przed każdym programistą. Powodują one zwiększenie wydajności samej aplikacji oraz umożliwiają szybkie dodawanie różnych nowych funkcjonalności. Wraz z pojawieniem się technologii .NET Core można tworzyć aplikacje i wdrażać je na różnych systemach operacyjnych takich jak Windows, Linux oraz MacOS. Microsoft i cała społeczność programistyczna wykorzystująca narzędzia do tworzenia swoich aplikacji uczynili system Linux idealną platformą do wdrażania aplikacji ASP.NET Core. Wykorzystanie asynchronicznych wzorców programowania opartych na nowych szablonach MVC, sprawia, że aplikacje opracowane na bazie tego szkieletu są coraz szybsze [6].

3. Aplikacje testowe

Dla celów porównań szkieletów programistycznych opracowano dwie proste aplikacje webowe. Jedna została zbudowana przy pomocy szkieletu Symfony 4, natomiast drugą oparto na ASP.NET Core MVC. Obie aplikacje są identyczne, ponieważ mają zaimplementowane te same funkcjonalności, pozwalające na zarządzanie artykułami za pomocą prostych operacji CRUD (ang. create, read, update, delete) do przeglądania, dodawania, usuwania oraz aktualizowania artykułów (Rysunek 1).



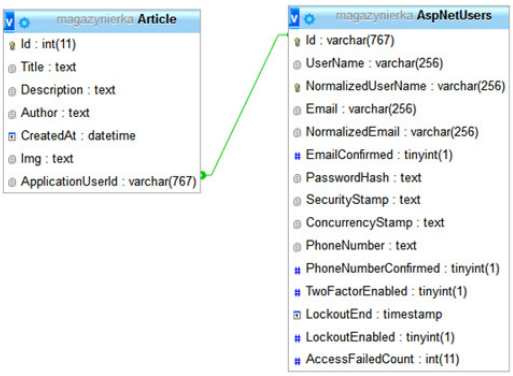
Rysunek 1: Widok strony głównej z artykułami.

Większość działań jest realizowana za pomocą formularza zawierającego trzy pola do uzupełnienia (tytuł artykułu, opis oraz selektor plików do załączania zdjęcia) oraz przycisk potwierdzający daną operację (Rysunek 2). Podczas aktualizacji pola formularza najpierw wypełniane są danymi z bazy, dotyczącymi wybranego artykułu, a następnie są modyfikowane.

Rysunek 2: Tworzenie nowego artykułu.

Jednocześnie na jednej stronie możliwe jest przeglądanie dziesięciu artykułów, co zapewnia pełną kontrolę i swobodę podczas tworzenia nowych treści. Do osiągnięcia tego efektu wykorzystano mechanizm paginacji.

Aplikacje testowe współpracują z prostą, bo składającą się z dwóch tabel, bazą danych (Rysunek 3). Główna tabela służy do przechowywania danych związanych z artykułami. Druga tabela zawiera informacje o użytkownikach, którzy po zalogowaniu się mogą dodawać, edytować i usuwać artykuły. Obie tabele są powiązane ze sobą relacją jeden do wielu.



Rysunek 3: Struktura bazy danych aplikacji.

4. Środowisko testowe

Testy aplikacji zostały przeprowadzone na komputerze Lenovo G580, którego parametry przedstawione są w Tabeli 1. Przed uruchomieniem testów, w celu zwiększenia ich wiarygodności, zostały wyłączone wszystkie zbędne procesy.

Tabela 1: Konfiguracja środowiska uruchomieniowego dla przygotowanych testów

System operacyjny	Windows 10
Procesor	i3-312M 2.50 GHz
Pamięć RAM	8GB
Dysk	SSD

Na potrzeby badań został wykorzystany profesjonalny serwer VPS (ang. Virtual Private Server) oferowany przez firmę OVHCloud [7]. Wirtualny serwer jest wyodrębnionym środowiskiem utworzonym na serwerze fizycznym przy pomocy technologii wirtualizacji. Rozwiązanie to oferuje wszystkie korzyści standardowego serwera, z wydzielonymi zasobami i pełnym dostępem administratora. Użytkownik może wybrać system operacyjny, a także aplikacje, z których chce korzystać w danej chwili, bez żadnych ograniczeń konfiguracyjnych. W Tabeli 2 znajdują się szczegóły dotyczące wykorzystanego w ramach niniejszej pracy środowiska uruchomieniowego.

Tabela 2: Konfiguracja środowiska uruchomieniowego

System Operacyjny	Ubuntu 18.04 Server
Pamięć RAM	2GB
Pamięć dyskowa	SSD 20GB
Serwer	Apache/2.4.29 (Ubuntu)
Serwer MySql	5.7.32
Wersja PHP	7.2.24
Środowisko .Net	ASP.Net Core 3.1
EntityFarmework	3.1
HTTP/SSL	Tak
Symfony	Symfony 4.4.10

5. Metoda badań

5.1. Narzędzie badawcze - biblioteka Puppeteer

Do analizy wydajności aplikacji testowych wykorzystano bibliotekę Puppeteer [8]. Została ona napisana w języku TypeScript i w związku z tym do jej uruchomienia niezbędna jest platforma Node.js, która z jednej strony kompiluje kod TypeScript na JavaScript, a z drugiej umożliwia uruchomianie oprogramowania. Puppeteer pozwala zautomatyzować manualne czynności wykonywane przez człowieka. W ten sposób kod aplikacji, uzupełniony odpowiednimi poleceniami pochodzącymi z tej biblioteki, wykonuje się samodzielnie. Oznacza to, że strona www obsługiwana jest automatycznie za pośrednictwem komend, wywołujących kolejne wymagane do jej obsługi czynności. Wśród nich mogą być na przykład kliknięcia myszą komputerową, pisanie na klawiaturze czy robienie zrzutów ekranowych.

Podczas analizy aplikacji testowej wykonanej za pomocą szkieletu ASP.NET jak i Symfony została wykonana ta sama sekwencja poleceń. Najpierw skrypt uruchamia przeglądarkę internetową i autoryzuje użytkownika. Następnie wykonuje wielokrotnie seriami operacje CRUD, mierzy czasy ich trwania oraz na koniec zamyka przeglądarkę. Listing 1 przedstawia metodę *auth*, przyjmującą dwa parametry i zawierającą instrukcje automatyzujące proces autoryzacji użytkownika.

Listing 1: Automatyczna autoryzacja użytkownika

```
async auth(page, model) {

    await page.goto(model.url + model.auth.url,
                    {waitUntil: 'networkidle0'});
    await page.type(model.auth.login, model.login);
    await page.type(model.auth.password, model.password);

    await Promise.all([
        page.click(model.auth.submit),
        page.waitForNavigation({waitUntil: 'networkidle0'}),
    ]);

    run page;
}
```

Pierwszy parametr (*page*) metody *auth* reprezentuje klasę, która jest odpowiedzialna za obsługę przeglądarki, zaś drugi parametr jest modelem, za pomocą którego dostarczane są niezbędne dane do obsługi testowanej aplikacji. W pierwszej kolejności wywołane jest polecenie *goto*, odpowiedzialne za uruchomienie strony o podanym adresie url, przekazywanym w pierwszym parametrze. W drugim parametrze przesyłany jest obiekt, który nakazuje metodzie odczekać do momentu aż się strona do końca wczyta. Dwa następne polecenia uzupełniają formularz danymi. Są one wywoływane z dwom parametrami. Za pomocą pierwszego wskazywane jest id elementu, dla którego będzie wywoływana jakaś akcja. Drugi parametr służy do przekazania wartości, którą będzie wypełniane pole. Następnie uruchamiana jest metoda *Promise.all*, która inicjuje naciśnięcie przycisku *wyślij*, skutkujące wysłaniem formularza. W tym przypadku także, jako jedyny parametr, został wykorzystany obiekt, którego zadaniem jest odczekanie do momentu aż strona zostanie załadowana.

5.2. Scenariusze badawcze

Eksperyment badawczy został przeprowadzony przez automat, który samodzielnie zrealizował scenariusze badawcze, podczas których mierzono czasy wykonywania się poszczególnych zadań. Średnie czasy trwania

realizacji poszczególnych operacji przyjęto jako wskaźniki wydajności i zostały użyte podczas późniejszych analiz. Scenariusze badawcze dotyczyły realizacji czterech prostych czynności CRUD na tabeli w bazie danych wykonywanych przez aplikacje testowe. Scenariusze te obejmowały następujące operacje:

- wyświetlanie wybranego artykułu,
- wypełnienie pól formularza danymi i dodanie artykułu do bazy,
- wczytanie danych z bazy do formularza, zmodyfikowanie artykułu i ponowny zapis do bazy,
- usuwanie wybranego artykułu.

Scenariusz 1: Wyświetlanie artykułów

Metoda odpowiadająca za wyświetlanie artykułów wykorzystuje tylko jedno polecenie *goto*, realizujące przejście do strony (paginy) zawierającej konkretny artykuł wskazany za pomocą adresu url.

Listing 2: Wyświetlanie artykułów realizowane za pomocą poleceń biblioteki Puppeteer

```
async articleSelect(page, pageNumber, model) {
    await page.goto(model.url + model.select.url + pageNumber,
                    {waitUntil: 'networkidle0'});

    run page;
}
```

Scenariusz 2: Dodawanie artykułów

Skrypt dodawania nowego artykułu (Listing 3) wygląda podobnie jak obsługa formularza przeznaczonego do autoryzacji. W tym przypadku, za pomocą instrukcji *page.type()* uzupełniane są pola z tytułem i opisem artykułu. Inicjowanie kliknięcia na przycisku typu *file* jest realizowane za pomocą polecenia *await page.$(model.new.file),* a następnie za pomocą kolejnej komendy *await input.uploadFile(image)* dodawany jest plik graficzny. Procedurę dodawania artykułu kończy polecenie symulujące naciśnięcie na przycisku *wyślij*.

Listing 3: Dodawanie artykułów realizowane za pomocą poleceń biblioteki Puppeteer

```
async articleNew(page, model) {
    await page.goto(model.url + model.new.url,
                    {waitUntil: 'networkidle0'});
    await page.type(model.new.description, description);
    const input =  await page.$(model.new.file);
    await input.uploadFile(image);
                    {waitUntil: 'networkidle0'});

    await Promise.all([
            page.click(model.new.submit),
            page.waitForNavigation({waitUntil: 'networkidle0'}),
    ]);

    run page;
}
```

Scenariusz 3: Aktualizowanie artykułów

Edycja artykułu została przeprowadzona w sposób podobny jak dodawanie. Zostały przy tym wykorzystane dodatkowe instrukcje, które umożliwiają najpierw uchwycenie (*document.getElementById*), a następnie wyczyszczenie pól formularza.

Listing 3: Aktualizacja artykułów realizowana za pomocą poleceń biblioteki Puppeteer

```
async articleEdit(page, id, model) {
    await page.goto(model.url + model.edit.url + id
            + model.edit.afterUrl, {waitUntil: 'networkidle0'});
    await page.evaluate((model) => { document.getElementById(
            model.edit.title.substring(1)).value = "", model)
    await page.type(model.edit.title, title);
    await page.evaluate((model) => { document.getElementById(
            model.edit.description.substring(1)).value = "", model)
    await page.type(model.edit.description, description);

    const input =  await page.$(model.edit.file);
    await input.uploadFile(image);

    await Promise.all([
            page.click(model.edit.submit),
            page.waitForNavigation({waitUntil: 'networkidle0'}),
    ]);
    run page;
}
```

Scenariusz 4: Usuwanie artykułów

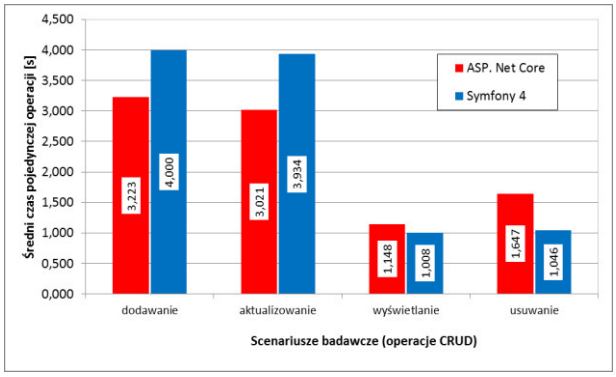
W procesie usuwania danego artykułu (Listing 4), po naciśnięciu przycisku *usuń*, wymagane jest jeszcze potwierdzenie tej operacji w okienku dialogowym. Czynności te są realizowane za pomocą poleceń *page.on('dialog')* oraz *dialog.accept()*.

Listing 4: Usuwanie artykułu realizowane za pomocą poleceń biblioteki Puppeteer

```
async articleDelete(page, model) {
    page.on('dialog', async dialog => {
        await dialog.accept();
    });
    await Promise.all([
            page.click(model.delete.submit),
            page.waitForNavigation({waitUntil: 'networkidle0'}),
    ]);

    run page;
}
```

6. Wyniki badań

Wykonano po trzy serie testów dla obu badanych aplikacji. W każdej serii po kolei były wykonywane scenariusze, które odpowiadały poszczególnym operacjom CRUD. Dla uzyskania jak największej precyzji wyników każda operacja była powtarzana 10 razy w pierwszej serii, 100 razy w drugiej i 1000 razy w trzeciej serii. Wyniki dla poszczególnych scenariuszy zostały uśrednione i przedstawione na Rysunku 4.

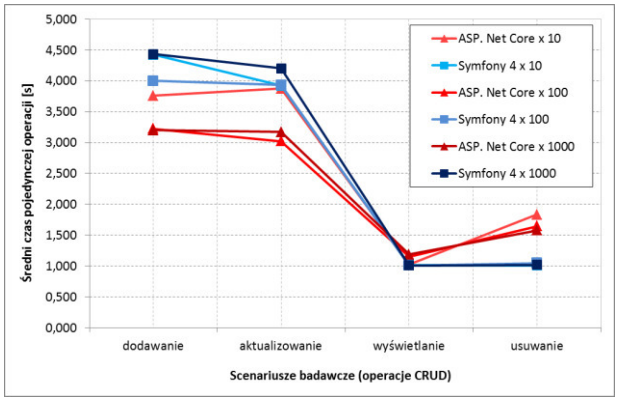


Rysunek 4: Średnie czasy wykonywania operacji przez aplikacje testowe dla 100 powtórzeń.

Powyższy wykres prezentuje średnie czasy ze 100 powtórzeń wykonania poszczególnych scenariuszy, za którymi stoją odpowiednie operacje CRUD. Wynika

z niego, że operacje dodawania i aktualizowania artykułów trwały dłużej dla aplikacji testowej zbudowanej na bazie szkieletu Symfony o odpowiednio 0,777 i 0,913 sekundy w stosunku do aplikacji opartej na platformie programistycznej ASP.NET Core. W przypadku wyświetlania artykułów różnice pomiędzy szkieletami były nieznaczne, gdyż wynosiły 0,14 sekundy na korzyść szkieletu Symfony 4. Także przy operacji usuwania z bazy lepsze czasy uzyskiwała aplikacja oparta na szkielecie Symfony. Dla tego scenariusza różnica była nieco wyższa, wynosiła 0,601 sekundy.

Na Rysunku 5 zestawiono uśrednione wyniki dla obu aplikacji testowych, czterech scenariuszy powtarzanych 10, 100 i 1000 razy.



Rysunek 5: Średnie czasy wykonywania operacji CRUD przez aplikacje testowe dla 10, 100 i 1000 powtórzeń.

W przypadku operacji dodawania dla wszystkich trzech serii (10, 100 i 1000 powtórzeń) średnie wyniki były wyższe dla aplikacji opracowanej na podstawie frameworka Symfony 4. Dla drugiej aplikacji testowej uśrednione wyniki dla serii nr 2 (100 powtórzeń) i serii nr 3 (1000 powtórzeń) były niemal identyczne. Podobna sytuacja była podczas aktualizowania artykułów. Na wykresie wyraźnie widać wyższe średnie poziomy czasów uzyskanych w trzech seriach dla aplikacji zbudowanej za pomocą szkieletu Symfony. W przypadku wyników dla frameworka ASP.NET Core odstające, wyższe wyniki widoczne są dla scenariusza 1, w którym operacja edycji była wykonywana 10 razy. Zwiększenie liczby powtórzeń spowodowało z jednej strony obniżenie średniego czasu oraz bardziej precyzyjne jego przedstawienie. Natomiast dla operacji wyświetlania artykułów, dla obu wersji aplikacji i wszystkich scenariuszy i bez względu na liczbę powtórzeń wyniki były bardzo podobne. W ostatnim scenariuszu, dotyczącym usuwania artykułów, aplikacja bazująca na Symfony 4, niezależnie od liczby powtórzeń operacji, uzyskiwała niemal identyczne średnie wyniki. Znacznie gorsze czasy realizacji zadań na bazie, dla średniej obliczonej z 10 jak i 100 pomiarów, miało oprogramowanie wykonane za pomocą ASP.NET Core.

7. Podsumowanie

W pracy porównywano dwa szkielety programistyczne ASP.NET Core i Symfony 4. Do analiz posłużyły specjalnie do tego celu opracowane dwie webowe aplikacje testowe. Przygotowano eksperyment badawczy, w którym proste czynności wypełniania i zapisywania formularza, edycji danych w formularzu i ich aktualizacji, wyświetlenia artykułów oraz usuwania wybranego artykułu zostały zautomatyzowane przez użycie biblioteki Puppeteer wykorzystywanej do realizacji testów automatycznych. Następnie wykonano wielokrotne pomiary czasów trwania poszczególnych czynności a uzyskane wyniki poddano uśrednianiu. Uzyskane średnie wyniki stanowiły podstawę do porównań wybranych szkieletów programistycznych.

Z dwiema czasochłonnymi operacjami, tzn. wprowadzaniem i edycją danych, znacznie lepiej radzi sobie framework ASP.NET Core. Jego wyniki były odpowiednio o około 28% i 25% (średnia z 1000 powtórzeń) lepsze w stosunku do szkieletu Symfony 4. Natomiast w dwóch mniej czasochłonnych operacjach, czyli wyświetlaniu i usuwaniu artykułów, wyraźnie lepszym okazał się szkielet Symfony w wersji 4. Jego wyniki były o 15 i 36 procent niższe (średnia z 1000 pomiarów) w stosunku do drugiego badanego szkieletu.

Z otrzymanych wyników, wykonanych analiz i przeprowadzonego wnioskowania trudno jest jednoznacznie określić, które z dwóch wziętych pod uwagę w tej pracy narzędzi programistycznych jest lepsze. Dobrym kierunkiem byłoby poszerzenie zakresu badań o jeszcze inne aspekty takie jak na przykład pomiar objętość kodu, ilość przesyłania danych czy zużycie zasobów.

Na koniec należy zwrócić uwagę na pewne ograniczenia zrealizowanych badań. Zostały one przeprowadzone w sztucznych warunkach, na aplikacjach testowych, które były bardzo proste, zaopatrzone w mało rozbudowany formularz, za pośrednictwem, którego wykonywano wielokrotnie powtarzane typowe operacje dodawania, edycji, wyświetlania oraz usuwania danych.

Literatura

[1] Tiobe, https://www.tiobe.com/tiobe-index/, [16.02.2021].

[2] Google Trends, https://trends.google.pl/trends/, [16.02.2021].

[3] A. D. Wac, T. K. Watras, G. Kozieł, Comparative analysis of solutions used in automated testing, Journal of Computer Sciences Institute, 15 (2020) 156-163.

[4] SMARTBEAR, Test Automation Best Practices, https://smartbear.com/learn/automated-testing/best-practices-for-automation/, [16.02.2021].

[5] Symfony, https://symfony.com/, [16.02.2021].

[6] Introduction to ASP.NET Core, https://docs.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0, [16.02.2021].

[7] OVHCloud, https://www.ovh.pl/, [16.02.2021].

[8] Puppeteer, https://developers.google.com/web/tools/puppeteer, [16.02.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 100–106
Received: 8 March 2021
Accepted: 21 March 2021

Comparative analysis of frameworks used in automated testing on example of TestNG and WebdriverIO

Analiza porównawcza frameworków do automatyzacji testowania aplikacji webowych na przykładzie TestNG i WebdriverIO

Alla Shtokal*, Jakub Smołka

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

Abstract

The article presents a comparative analysis of frameworks supporting the development of automated tests for defined test scenarios. The comparative study concerned the TestNG and WebdriverIO frameworks. The overview of the tool has been analyzed both in terms of the test development process as well as the speed and efficiency of their execution. The website github.com was used for the purposes of the work. This application was used to run test scripts written in both frameworks. The results were compared by four defined criteria: the time of running the test scripts with a different maximum number of simultaneously running browser instances, the average time of running all test scripts in headless mode, the average value of memory and CPU usage during the test execution. The summary includes the evaluation of the compared frameworks.

Keywords: Selenium; WebdriverIO; TestNG; framework

Streszczenie

W artykule przedstawiona została analiza porównawcza frameworków wspomagających wytwarzanie testów zautomatyzowanych dla zdefiniowanych scenariuszy testowych. Badanie porównawcze dotyczyło frameworków TestNG oraz WebdriverIO. Omówienie narzędzia zostało przeanalizowane zarówno pod kątem procesu tworzenia testów, jak i szybkości oraz wydajności ich wykonywania. Na potrzeby pracy została wykorzystana strona internetowa github.com. Aplikacja ta posłużyła do przeprowadzania skryptów testowych napisanych w obu frameworkach. Wyniki zostały porównane przez cztery zdefiniowanych kryterium: całkowity czas uruchamiania zbiorów testowych z różną maksymalną liczbą jednocześnie uruchomionych instancji przeglądarki, średni czas uruchamiania wszystkich skryptów testowych w trybie headless, średnia wartość zużycia pamięci oraz CPU podczas wykonania testów. W podsumowaniu zawarta została ocena porównywanych frameworków.

Słowa kluczowe: Selenium; WebdriverIO; TestNG; framework

*Corresponding author

Email address: alla.shtokal@pollub.edu.pl (A. Shtokal)

1. Wstęp

Głównym celem automatyzacji testów jest obniżenie kosztów testowania programu po jego aktualizacji. Okresowo powtarzane kontrole tego samego typu zajmują dużo czasu w cyklu rozwoju. Automatyzacja skraca fazę testów i uwalnia główny zasób firmy - czas pracy specjalistów. Kolejną, nie mniej oczywistą zaletą takich testów jest podniesienie jakości testów, co gwarantuje niezawodność produktu. Przecież straty z tytułu wad ujawnionych dopiero na etapie eksploatacji przemysłowej mogą być bardzo duże. Niezadowolenie ich klientów jest na ogół trudne do oszacowania.

Testowanie ręcznie nie pozwala na kompleksowe testowanie w pełni funkcjonalnych systemów w wyznaczonym przez projekt czasie, co prowadzi do różnego rodzaju negatywnych konsekwencji, dlatego automatyzacja jest niezbędna. Przede wszystkim automatyzacja może poprawić niezawodność oprogramowania i zmniejszyć ryzyko wykrycia usterek na etapie eksploatacji przemysłowej. Lepsza dokładność testowania i możliwość wczesnego znajdowania większej liczby defektów. Możliwe staje się zidentyfikowanie i wyeliminowanie wąskich gardeł wydajności systemu w całym cyklu rozwojowym. Dzięki automatyzacji możesz zobaczyć dokładny obraz wydajności systemu na wszystkich poziomach, w tym podczas uruchamiania. W dużych projektach wybór odpowiedniego frameworku jest kluczowy. Framework musi odpowiadać technologii, na podstawie, której zbudowany jest interfejs graficzny. Na przykład Selenium [1] jest dobre do testowania aplikacji internetowych.

W ramach badania została analiza popularnych na rynku frameworków do automatyzacji testów aplikacji internetowych na przykładzie WebdriverIO oraz TestNG.

2. Przegląd literatury

Analizując problem badawczy można powiedzieć, że istnieje dość dużo badań dotyczących frameworków do automatyzacji testowania. W pracach, które zostały przeanalizowane często poruszanym tematem były zadania i oczekiwania stawiane korzystaniu frameworków do automatyzacji oraz ich użyteczności. W artykule „Test Automation Framework based on WEB" [2] au-

torstwa Fei Wang oraz Wencai Du opisana została zasada działania narzędzia Selenium, który służy do wspierania automatyzacji testów aplikacji internetowych.

W artykule autorzy zaprojektowali framework do automatycznego testowania oprogramowania dla aplikacji internetowej w oparciu o Selenium i JMeter. Do testowania została wybrana aplikacja webowa – translator. Wyniki pokazały, że stworzona struktura oprogramowania poprawia, jakość oprogramowania i zwiększa wydajność.

Kolejna praca, która została uwzględniona w przeglądzie to „Analiza porównawcza rozwiązań wykorzystywanych w testowaniu automatycznym" [3] autorstwa A. Wac oraz T. Watras, G. Kozieł. W tym artykule przedstawiona została analiza porównawcza narzędzi wspomagających wytwarzanie zautomatyzowanych testów. Autorzy zbadali każde narzędzie zarówno pod kątem procesu tworzenia testów, jak i prędkości ich wykonywania. Na podstawie przeprowadzonych badań autorzy podkreślili, że nie można wyznaczyć najlepszego rozwiązania do tworzenia zautomatyzowanych testów.

Podobne podejście prezentowane jest w prace „Comparison analysis of Android GUI testing frameworks by using an experimental study" [4] autorstwa Meiliana, Septian, I., Alianto, R. S. oraz Daniel. W tym badaniu zostały ocenione frameworki testowe systemu Android na podstawie dwóch rodzajów kryteriów. Te pierwsze, podobnie jak w poprzedniej pracy, są bardziej ogólne, kolejne zawierają wyłącznie szczegóły techniczne.

W ramach tego eksperymentalnego badania przez autorów była stworzona prosta aplikacja na Androida, która została używana do oceny wszystkich kryteriów czterech wybranych frameworków. Przez autorów zostały napisane testy do tej aplikacji z użyciem każdego frameworku, wyniki i kryteria zostały zaprezentowane w tabelce. W rezultacie preferowany frameworkiem okazał się Espresso.

Porównania w tych pracach, które zostały przeanalizowane skupiają się na starszych wersjach lub innych frameworkach niż wykorzystane w tej prace, dlatego podejmowanie tego tematu jest aktualne.

3. TestNG

TestNG to platforma testowa zaprojektowana w celu uproszczenia szerokiego zakresu potrzeb testowych, od testowania jednostkowego (testowanie klasy w izolacji od innych) do testów integracyjnych (testowanie całych systemów złożonych z kilku klas, kilku pakietów, a nawet kilku struktur zewnętrznych, takich jak serwery aplikacji). Framework jest przeznaczony i wykorzystany wyłącznie dla języka Java. Eliminując większość ograniczeń starszej platformy, TestNG daje programiście możliwość pisania bardziej elastycznych i wydajnych testów. Ponieważ w dużym stopniu korzysta z adnotacji Java (wprowadzonych z JDK 5.0) do definiowania testów, może również pokazać, jak używać tej nowej funkcji języka Java w rzeczywistym środowisku produkcyjnym.

Dokumentacja dla TestNG jest dość szeroko rozbudowana. Framework posiada bardzo przystępną dla użytkownika dokumentację. Są tam wyszczególnione sekcje. Istnieje 8 głównych sekcji i każda z nich jest podzielona na kilka bloków. Użytkownik może się dowiedzieć z nich jak zaplanować, przeprowadzić oraz przeanalizować wyniki przeprowadzonego testu [5-6].

TestNG zawiera możliwość uruchamiania testów w dowolnie dużych pulach wątków, można też zgrupować wiele przypadków testowych i przekonwertować je na plik XML, a następnie ustawić priorytet przypadku testowego w określonej kolejności. TestNG zapewnia łatwą i elastyczną konfigurację testów, w której nie ma potrzeby pisania kodu, aby modyfikować jakiekolwiek metody testowe.

Raporty TestNG pojawiają się, gdy wykonywane są przypadki testowe przy użyciu TestNG. Po wykonaniu przypadków testowych TestNG wygeneruje domyślny raport HTML. Dodatkowo mogą być wykorzystane inne zewnętrzne frameworki do tworzenia raportów, na przykład Allure Framework. W tym celu wystarczy dodać odpowiednie zależności do pliku pom.xml.

4. WebdriverIO

WebdriverIO to oparty na JavaScript framework do automatyzacji testów zbudowany na Node.js. Jest to projekt typu open source opracowany dla społeczności testujących automatycznie. WebdriverIO jest rozszerzalny, kompatybilny, bogaty w funkcje i łatwy w instalacji. Jest to uważane za platformę automatyzacji testów nowej generacji, która obsługuje zarówno przeglądarki komputerowe, jak i aplikacje mobilne. Co sprawia, że WebdriverIO jest korzystną opcją do testowania automatycznego. Obsługuje koncepcje testowe BDD (Behavior-driven development) i TDD (Test-driven development).

W przeciwieństwie do większości dostępnych obecnie platform automatyzacji przeglądarek, WebdriverIO jest napisane w całości w języku JavaScript. Nawet instalacja Selenium odbywa się za pośrednictwem modułu NPM. Z tego powodu istnieje taka opinia, że programiści front-end powinni pisać testy dla swojego kodu (zarówno jednostkowego, jak i funkcjonalnego). WebdriverIO sprawia, że jest to niezwykle łatwe [7-8].

Framework posiada bardzo przystępną dla użytkownika dokumentację. Strony zawierają materiały referencyjne dla wszystkich zaimplementowanych wiązań i poleceń. WebdriverIO ma zaimplementowane wszystkie polecenia protokołu JSONWire, a także obsługuje specjalne powiązania dla Appium (narzędzia do testów aplikacji mobilnych). WebdriverIO używa Selenium, dlatego możliwe jest uruchamianie testów we wszystkich rodzajach przeglądarek. Selenium to bardzo dobra platforma i lider w branży do uruchamiania automatyzacji przeglądarek [9].

WebdriverIO nie posiada własnego narzędzia do raportowania. Natomiast ma możliwość podłączenia wielu różnych pakietów do tworzenia raportów od in-

nych narzędzi. Na przykład, jest możliwość konfiguracji wtyczki do tworzenia raportów testowych takich jak Allure, HTML-reporter oraz Spec-reporter.

5. Scenariusze testowe

W celach badania do testowania została wybrana dość znana aplikacja webowa, służąca do wspólnego tworzenia oprogramowania: www.github.com. Następnie, na bazie tej strony webowej zostało stworzonych siedem scenariusze testowych, na podstawie, których, zostały wykonane testy automatyczne w analizowanych frameworkach: TestNG (w języku Java) oraz WebdriverIO (w języku JavaScrypt). Wyniki testów zostały porównane pod kontem szybkości uruchomienia oraz zużyciem pamięci komputera. Scenariusze testowe podane w poniższych tabelach (Tabele 1 - 7).

Tabela 1: Scenariusz do zdarzenia Mouse Hover

Nazwa		Zdarzenie Mouse Hover
Cel		Sprawdzenie poprawności działania menu rozwijanego po najechaniu myszki
Warunki wstępne		brak
Wymagania		brak
Kroki		
Opis wykonanych czynności		Oczekiwany wynik
1	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2	Najechanie myszki na element menu:	Pojawia się nowy blok - menu rozwijane pod wskazanym elementem

Tabela 2: Scenariusz zapisywania danych do pliku

Nazwa		Zapisywanie do pliku tekstowego
Cel		Sprawdzenie odczytu elementów ze strony i zapis do pliku
Warunki wstępne		Na stronie jest wyświetlane, co najmniej jedne dostępne stanowisko
Wymagania		Plik do zapisu już istnieje
Kroki		
Opis wykonanych czynności		Oczekiwany wynik
1	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2	Najechanie myszki na element menu:	Pojawia się nowy blok - menu rozwijane pod wskazanym elementem
3	Rozwijanie każdego bloku pozycji i zapis nazw stanowisk do pliku	Nazwy stanowisk z każdego bloku są odczytane ze strony i zapisane do pliku tekstowego

Tabela 3: Scenariusz zapisywania danych do pliku

Nazwa		Pobranie pliku pdf
Cel		Sprawdzenie możliwości pobrania pliku pdf ze strony
Warunki wstępne		Plik istnieje
Wymagania		Plik dostępny do pobrania niezalogowanym użytkownikom
Kroki		
Opis wykonanych czynności		Oczekiwany wynik
1	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2	Przejście do odnośnika „Download Security Guide”	Przeniesienie na witrynę z przyciskiem do pobrania pliku
	Kliknięcie przycisku do pobrania	Plik został pobrany i otworzony w nowej zakładce

Kolejne cztery scenariusze, które są podane w (Tabelach 4-7) będą przeprowadzone za pomocą zbioru danych do testów. W przypadku scenariusza logowania to będą dane użytkowników (nazwa użytkownika oraz hasło), w przypadku scenariusza wyszukiwania – słowa kluczowe, dla scenariusza sortowania – lista dostępnych pól i odpowiednio dla scenariuszy filtrowania – lista języków do pokazania. Te dane zostały wcześniej przygotowane i będą pobierane z pliku .xls podczas uruchamiania testów (Rysunek 3).

Dla każdego scenariusza potrzebującego takiego zbioru danych istnieje osobna zakładka w pliku.

Tabela 4: Scenariusz logowania

Nazwa		Zalogowanie się do aplikacji
Cel		Sprawdzenie poprawności logowania do aplikacji
Warunki wstępne		Wylogowany użytkownik
Wymagania		Brak
Kroki		
Opis wykonanych czynności		Oczekiwany wynik
1	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2	Przejście do odnośnika „SignIn”	Przeniesienie na witrynę logowania
3	Uzupełnienie danych logowania	Uzupełnione wartości
4	Kliknięcie przycisku do logowania	Przejście do witryny konta użytkownika, sprawdzanie pola email/username

Tabela 5: Scenariusz wyszukiwania

Nazwa		Wyszukiwanie przez słowo kluczowe
Cel		Sprawdzenie poprawności wyszukiwania przez słowo kluczowe
Warunki wstępne		Istnieją repozytoria /użytkownicy za podanym słowom kluczowym
Wymagania		Brak
Kroki		
Opis wykonanych czynności		Oczekiwany wynik
1	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2	Kliknięcie w pole wyszukiwania	Kursor myszy został umieszczony w polu wyszukiwania
3	Podanie słowa kluczowego oraz kliknięcie Enter	Przeniesienie na witrynę z wynikami wyszukiwania, sprawdzanie, czy w wynikach wyszukiwania znajduje się słowo kluczowe w nazwie repozytorium lub nazwie użytkownika

Tabela 6: Scenariusz sortowania

Nazwa		Sortowanie
Cel		Sprawdzenie poprawności sortowania wyników wyszukiwania
Warunki wstępne		brak
Wymagania		brak
Kroki		
Opis wykonanych czynności		Oczekiwany Wynik
1.	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2.	Kliknięcie w pole wyszukiwania	Kursor myszy został umieszczony w polu wyszukiwania
3.	Podanie słowa kluczowego oraz kliknięcie Enter	Przeniesienie na witrynę z wynikami wyszukiwania
4.	Kliknięcie w wybrany rodzaj sortowania	Sprawdzanie wybranego rodzaju sortowania

Tabela 7: Scenariusz filtrowania

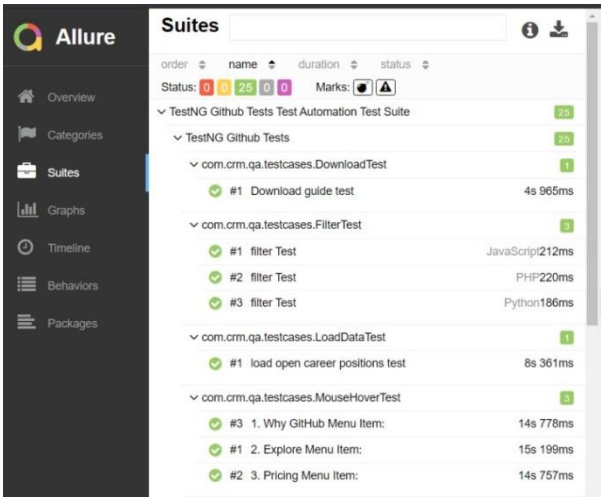
Nazwa		Filtrowanie
Cel		Sprawdzenie poprawności filtrowania wyników wyszukiwania
Warunki wstępne		brak
Wymagania		brak
Kroki		
Opis wykonanych czynności		Oczekiwany Wynik
1.	Wejście na stronę www.github.com	Przeniesienie do witryny portalu github
2.	Kliknięcie w pole wyszukiwania	Kursor myszy został umieszczony w polu wyszukiwania
3.	Podanie słowa kluczowego, kliknięcie Enter	Przeniesienie na witrynę z wynikami wyszukiwania
4.	Kliknięcie w wybrany filtr	Sprawdzanie wyników według wybranego filtru

6. Eksperyment

Skrypty testowe zostały zaimplementowane w obu frameworkach i odpowiadają zaplanowanym wcześniej scenariuszom testowym. One korzystają z tych samych zbiorów danych testowych. Jeden zestaw testów (test suite) składa się z 25 przypadków testowych (test cases). Czas i wyniki uruchamiania testów zostały przedstawione w wygenerowanym w Allure raporcie.

Na poniższym rysunku (Rysunek 1) przedstawione zżuty ekranu raportu wykonania całego zbioru testów (czyli 25 przypadków testowych). Jak widać wszystkie testy przeszły.

W ramach eksperymentu zostały zdefiniowane scenariusze badawcze, które zostały przedstawione w Tabeli 8.



Rysunek 1: Wygląd wygenerowanego Allure Raportu (TestNG).

Tabela 8: Scenariusze badawcze

	Scenariusz badawczy	Liczba wykonań
1.	Pomiar całkowitego czasu uruchamiania wszystkich skryptów testowych:	50
2.	Pomiar całkowitego czasu uruchamiania skryptów testowych w trybie headles (minimalizacja [h,m,s])	50
3.	Pomiar zużycia pamięci podczas wykonania testów (minimalizacja [MB])	50
4.	Pomiar zużycia CPU podczas wykonania testów (%)	50

Dla czystości eksperymentu testy zostały przeprowadzone w tych samych warunkach. Ponieważ do testów została wykorzystana aplikacja działająca w Internecie, zależy tutaj na połączeniu sieciowym. Akurat w ramach badań stosowane było połączenie przez WIFI uczelni – eduroam (EDUcation ROAMing).

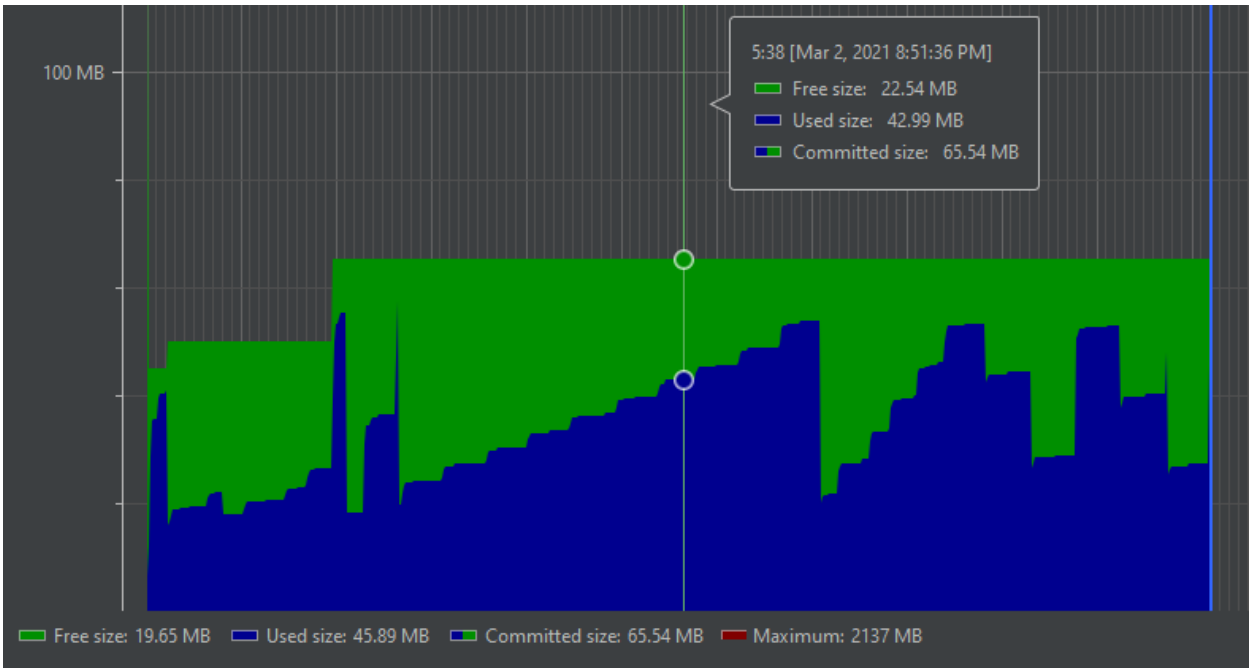
Testy zostały wykonane na komputerze stacjonarnym z następującymi parametrami:

- procesor Intel® Core ™ i7-2760QM (do 3,50 GHz, 4 rdzenie, 8 wątków)
- pamięć RAM 8 GB
- rodzaj dysku SSD
- system operacyjny: Windows 10 Education

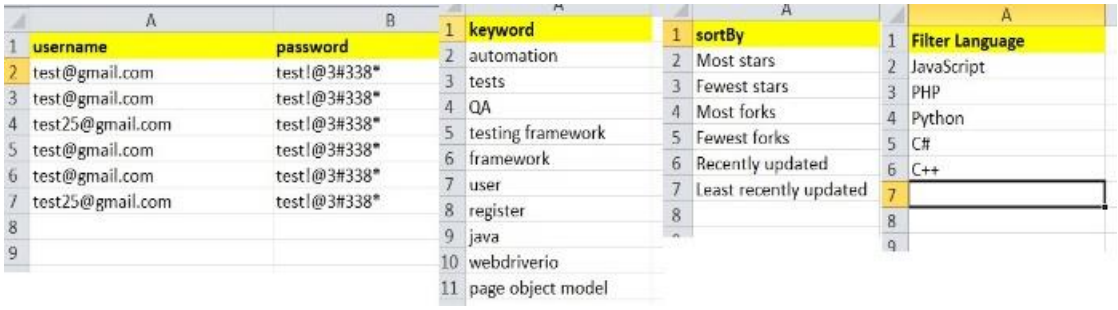
7. Wyniki

Wyniki zostały porównane przez cztery zdefiniowane kryteria: czas uruchamiania zbiorów testowych w różnych trybach działania przeglądarki, średnia wartość zużycia pamięci oraz CPU podczas wykonania testów.

Za pomocą narzędzi pomiarowych zostało zmierzone średnie zużycie pamięci podczas wykonania testów. Dla TestNG ta wartość wynosi 45,89 MB, a dla WebdriverIO 65 MB. Przykładowy wykres dla TestNG podano na Rysunku 2.



Rysunek 2: Pomiar zużycia pamięci podczas wykonania testów (TestNG).



	A	B
1	username	password
2	test@gmail.com	test!@3#338"
3	test@gmail.com	test!@3#338"
4	test25@gmail.com	test!@3#338"
5	test@gmail.com	test!@3#338"
6	test@gmail.com	test!@3#338"
7	test25@gmail.com	test!@3#338"
8		
9		

	A
1	keyword
2	automation
3	tests
4	QA
5	testing framework
6	framework
7	user
8	register
9	java
10	webdriverio
11	page object model

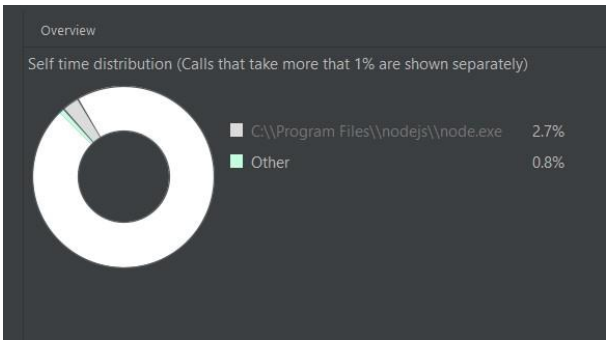
	A
1	sortBy
2	Most stars
3	Fewest stars
4	Most forks
5	Fewest forks
6	Recently updated
7	Least recently updated
8	

	A
1	Filter Language
2	JavaScript
3	PHP
4	Python
5	C#
6	C++
7	
8	

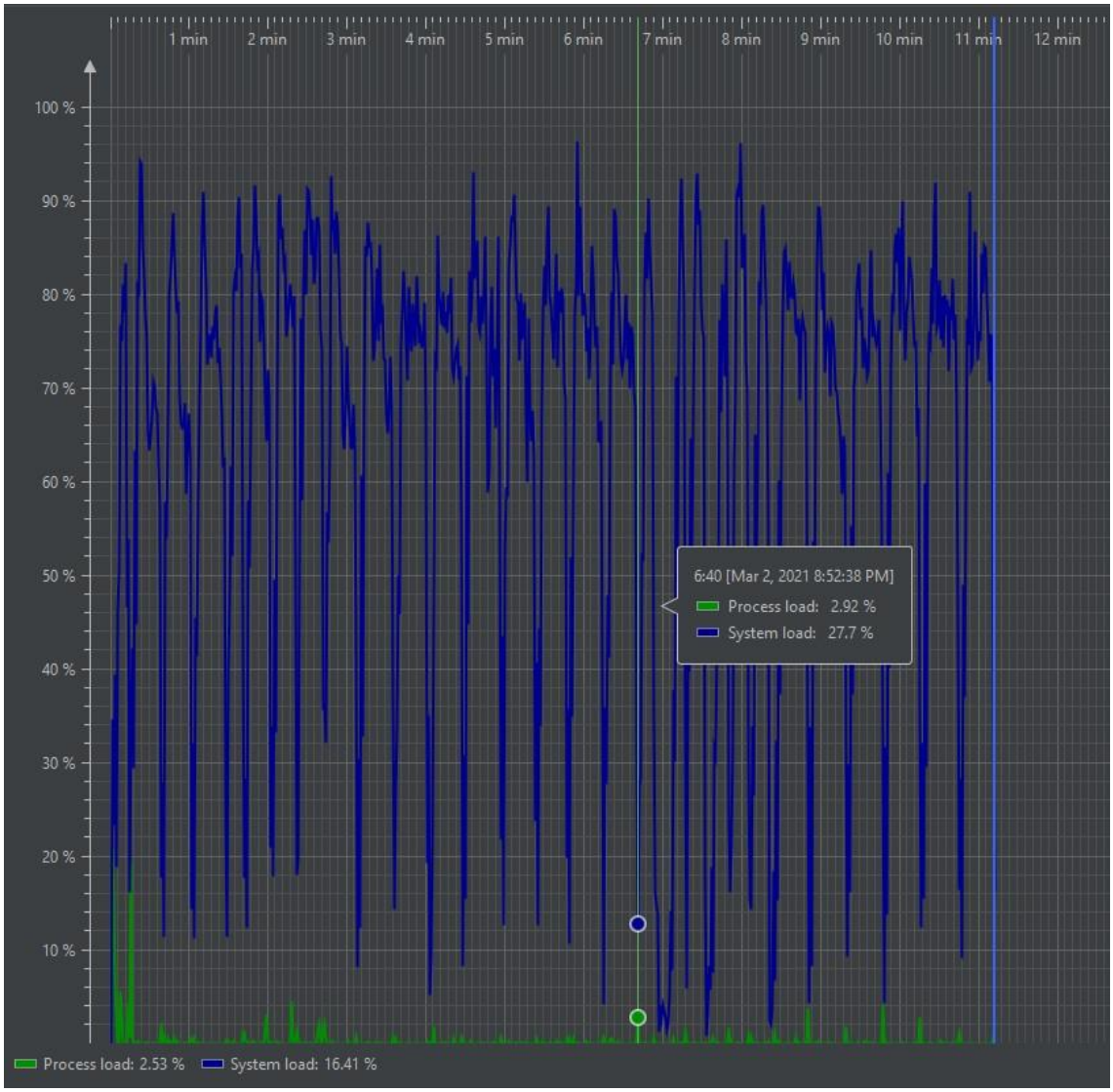
Rysunek 3: Zbiór danych z pliku .xls.

Następnie zmierzono średnie użycie CPU podczas wykonania testów. Dla frameworku TestNG to 2.53%, a dla WebdriverIO ta wartość wynosi 2.7%.

Przykładowy wykres uruchomianego testu podano na rysunkach poniżej, dla WebdriverIO – Rysunek 4, dla TestNG – Rysunek 5.



Rysunek 4: Średnia zużyta CPU dla testów WebdriverIO.



Rysunek 5: Średnia zużyta CPU dla testów TestNG.

Podsumowująca tabela pokazująca wyniki eksperymentu podana poniżej (Tabela 9).

Tabela 9: Wyniki eksperymentu

	Scenariusz badawczy	TestNG	Web-driverIO
1.	Średni czas uruchamiania zbioru skryptów testowych	11m 10s	7m 5s
2.	Średni czas uruchamiania skryptów testowych w trybie headles (minimalizacja [h,m,s])	6m 1s	3m 43s
3.	Pomiar zużycia pamięci podczas wykonania testów (minimalizacja [MB])	45.89 MB	65 MB
4.	Pomiar zużycia CPU podczas wykonania testów (%)	2.53%	2.7 %

Na podstawie przeprowadzonych badań lepszym okazał się framework WebdriverIO, chociaż różnica wyników nie jest dość duża. W pierwszym badaniu uruchamiania zbioru skryptów z jednej jak i z pięciu instancji przeglądarki framework WebdriverIO okazał się 1.61 razy szybszy.

W drugim badaniu, podczas uruchamiania testów w trybie headless WebdriverIO też pokazał lepszy wynik i okazał się 1.58 razy szybszy. Kolejnym badaniem było porównanie użycia pamięci podczas uruchomienia testów, to okazało się, że framework WebdriverIO wymaga więcej pamięci niż TestNG. Nie jest ta różnica zbyt duża i wynosi 19.11 MB. Ostatnie badanie dotyczyło pomiarów zużycia CPU komputera, na którym zostały uruchomione skrypty testowe. Wystąpiła niewielka różnica wynosząca 0.17% na korzyść TestNG.

Pod względem kryteriów ogólnych, warto wyróżnić WebdriverIO z tego powodu, że jest napisany i przeznaczony dla napisania testów w języku JavaScript. Jest to plusem, ponieważ ten język jest często wykorzystywany do napisania stron internetowych i programistom nie ma potrzeby używać innego języka, żeby zacząć pisać testy, innymi słowami WebdriverIO jest łatwiejszy do wdrożenia.

Z przeprowadzonych badań wynika, że jeśli chodzi o wybór lepszego frameworku testowego, to preferowany jest WebdriverIO.

Literatura

[1] T. J. Naidu, N. A. Basri, S. Nagenthram, "Selenium: A comparative analysis," 2014 Internet Conference Contempt Computer Informatics, IC3I 2014, 2014.

[2] S. Jagannatha, et al, Comparative Study on Automation Testing using Selenium Testing Framework and QTP, International Journal of Computer Science and Mobile Computing, 3(10) (2014) 258-267.

[3] Wac, T. Watras, G. Kozieł, Analiza porównawcza rozwiązań wykorzystywanych w testowaniu automatycznym, 2019.

[4] S. Jagannatha, et al, Comparative Study on Automation Testing using Selenium Testing Framework and QTP International Journal of Computer Science and Mobile Computing, 3(10) (2014) 258-267.

[5] M. Meiliana, I. Septian, R.S. Alianto, Daniel.. Comparison Analysis of Android GUI Testing Frameworks by Using an Experimental Study, Procedia Computer Science, 135 (2018) 736–748. doi: 10.1016/j.procs.2018.08.211.

[6] S.M. Srinivasan, R.S. Sangwan, Web App Security: A Comparison and Categorization of Testing Frameworks, IEEE Software, 34(1) (2017) 99–102.

[7] S. Sharma, "Study and analysis of automation testing techniques," J. Global Research Computer Science, 3(12) (2012) 36-43.

[8] V. N. Maurya, E. R. Kumar, Analytical Study on Manual vs. Automated Testing Using with Simplistic Cost Model, 2012.

[9] K. Bahl, Software Testing Tools Techniques for Web Applications, 2015.

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 107–113
Received: 23 March 2021
Accepted: 09 April 2021

A comparison of mobile applications built with the use of Android and Flutter Software Development Kits based on many criteria

Porównanie aplikacji mobilnych zbudowanych przy zastosowaniu zestawów narzędzi programistycznych Android oraz Flutter z użyciem wielu kryteriów

Damian Gałan, Konrad Fisz*, Piotr Kopniak

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

Abstract

This publication presents a multi-criteria comparison of two mobile applications built with the use of Android and Flutter SDK. The former has been implemented with Kotlin and the latter with Dart. The benchmarking process examines factors such as execution time and CPU usage during data and disk operations. During the analysis, attention was paid to the length and volume the source code, community support and the availability of libraries. The comparative analysis shows that a mobile application written using Android SDK is often not only faster and more efficient, but also has greater community support and the number of libraries available. In addition, an analysis of the source code volume showed that Flutter has more concise but more difficult to navigate code than Android.

Keywords: Android SDK; Flutter; benchmark; comparison

Streszczenie

Niniejsza publikacja przedstawia wielokryterialne porównanie dwóch aplikacji mobilnych zbudowanych przy zastosowaniu Android oraz Flutter SDK. Pierwsza z nich zaimplementowana została w języku Kotlin, natomiast druga w Dart. Proces porównawczy bada takie czynniki jak czas wykonania oraz użycie procesora podczas operacji dyskowych oraz na danych. Dodatkowo podczas analizy zwrócono uwagę na długość oraz objętość kodu, wsparcie społeczności oraz dostępność bibliotek. Analiza wykazała, że aplikacja napisana przy użyciu Android jest nie tylko często szybsza oraz wydajniejsza ale również posiada większe wsparcie społeczności oraz liczbę dostępnych bibliotek. Ponadto analiza objętości kodu źródłowego wykazała, że Flutter posiada kod bardziej zwięzły lecz trudniejszy do nawigowania od Android.

Keywords: Android SDK; Flutter; test wydajnościowy; porównanie

*Corresponding authors

Email addresses: damian.galan@pollub.edu.pl (D. Gałan), konrad.fisz@pollub.edu.pl (K. Fisz)

1. Wstęp

Od wielu lat zaobserwować można bardzo dynamiczny rozwój technologii w dziedzinie urządzeń mobilnych. Kiedyś telefon komórkowy służył jedynie do komunikacji z innym człowiekiem, dzisiaj jest wszechstronnym urządzeniem ułatwiającym codzienne funkcjonowanie. Dzięki możliwości instalacji szerokiej gamy aplikacji pełni on funkcję odtwarzacza audio i wideo, kalendarza, notatnika, budzika, nawigacji i wiele innych. Koncept inteligentnego domu również staje się coraz bardziej powszechny i nikogo nie dziwi już sterowanie urządzeniami AGD lub RTV za pomocą smartfona. Bardzo prężnie rozwijającym się rynkiem jest też segment gier mobilnych, które generują rekordowe przychody i nic nie zapowiada tego by ten trend miał się odwrócić [1].

Mając to wszystko na uwadze producenci telefonów kładą olbrzymi nacisk na to by ich produkty były coraz to bardziej wydajne, a programiści dbają o to by dostępne aplikacje były coraz lepiej zoptymalizowane i dostosowane do potrzeb użytkowników.

Jednakże, z racji istnienia wielu narzędzi pozwalających na implementację danej aplikacji, programiści często stają przed dylematem, która technologia pozwoli na uzyskanie najlepszych rezultatów.

W dzisiejszych czasach najpopularniejszymi systemami operacyjnymi na urządzenia mobilne są Android oraz iOS. Z tych dwóch częściej wybierany jest system Android, z blisko 80% udziałem w całym rynku systemów operacyjnych na urządzenia mobilne [2].

Do tworzenia aplikacji dla systemu Android najczęściej wykorzystuje się zestaw narzędzi programistycznych Android, a sam kod źródłowy najczęściej napisany jest w językach programowania Java lub Kotlin.

Z uwagi jednak na wciąż znaczący udział systemu iOS w rynku, narodziła się idea programowania międzyplatformowych aplikacji mobilnych, które posiadając jedną bazę kodu pozwalają uruchomić kod źródłowy na obu platformach. Takie rozwiązanie pozwala znacznie ograniczyć zasoby ludzkie, potrzebne do wytworzenia, a następnie utrzymania takiej aplikacji mobilnej. Na rynku mamy kilka rozwiązań pozwalających na programowanie międzyplatformowej aplikacji mobilnych, jednak ostatnim czasie coraz większą popularność zdobywa zestaw narzędzi programistycznych Flutter stworzony przez Google [3].

Swoją premierę Flutter miał w 2015 roku podczas Dart Developer Summit, pod nazwą Sky. Od tamtego czasu prace nad Flutter'em cały czas trwają, a stosunkowo niedawno pokazano możliwość wykonania tego samego kodu na systemy operacyjne Linux, Mac, Windows, Google Fuchsia, oraz w przeglądarce jako aplikacja internetowa, co spowodowało, że zainteresowanie Flutterem znacznie wzrosło.

Niemniej jednak z racji tego, że zestaw narzędzi programistycznych Flutter jest nową technologią, nie można jednoznacznie stwierdzić, czy wykorzystanie wspomnianego narzędzia jest lepszym rozwiązaniem niż sprawdzony już na rynku zestaw narzędzi programistycznych Android. Na samą decyzję wpływu nie powinna mieć jedynie wydajność samej aplikacji ale również kryteria od niej różne takie jak: długość oraz objętość kodu źródłowego, wsparcie społeczności czy dostępność bibliotek. Przedstawiony artykuł podejmuje tematykę porównania wielokryterialnego dwóch funkcjonalnie identycznych aplikacji mobilnych stworzonych z wykorzystaniem zestawów narzędzi programistycznych Android oraz Flutter w oparciu o wyżej wspomniane kryteria oceny.

2. Przegląd literatury

Przed przystąpieniem do wykonania porównania wielokryterialnego zestawów narzędzi programistycznych Android oraz Flutter przeprowadzono badanie literaturowe na temat omawianych narzędzi. Przegląd literatury wykazał, że temat Fluttera nie jest dostatecznie wyczerpany z uwagi na to, że jest on technologią, która jest obecna na rynku od niedawna. Niemniej jednak bazy naukowe są bogate w literaturę na temat porównania mobilnych technolgii natywnych z wieloplatformowymi innymi niż Flutter, co jest istotne z punktu widzenia przedstawionego tematu pracy.

W publikacji G. Kozieła oraz D. Sulowskiego *Analiza porównawcza języków Kotlin i Java używanych do tworzenia aplikacji na system Android* [4] przeprowadzono analizę porównawczą dwóch języków programowania Java oraz Kotlin. W celu porównania obu tych języków wybrano następujące kryteria: obciążenie procesora oraz pamięci RAM, czas kompilacji i wykonania programu. Dodatkowo, języki poddano analizie pod względem struktury kodu, obsługi bazy danych, dostępności bibliotek, popularności oraz wsparcia społeczności. Na potrzeby artykułu stworzona została specjalna aplikacja mobilna, której zadaniem było sortowanie zbiorów liczb z wykorzystaniem kilku algorytmów oraz wyświetlenie prostej animacji. Autorzy artykułu stwierdzają, iż nie ma wyraźnej różnicy pomiędzy językami i nie można wskazać który język programowania jest lepszym wyborem przy tworzeniu aplikacji mobilnych dla systemu Android. Przy większości kryteriów istotną kwestią było to, jaki telefon był używany podczas testów. Zauważalne różnice zostały stwierdzone przy określaniu takich aspektów jak struktura kodu czy popularność. Autorzy artykułu doszli do wniosków, iż język Java jest lepszym wyborem dla początkujących programistów, z powodu większej ilości materiałów pomocniczych. Natomiast Kotlin może być lepszym wyborem dla osób z doświadczeniem w programowaniu aplikacji mobilnych.

Trzy kolejne publikacje, zawierają porównania wielu różnych narzędzi programistycznych do budowy aplikacji mobilnych. W artykule P. Kotarskiego, K. Śledzia oraz J. Smołki *Analiza wydajności aplikacji mobilnych przy zastosowaniu różnych narzędzi programistycznych do ich budowy* [5] zbadano wydajność aplikacji mobilnych przy zastosowaniu różnych narzędzi programistycznych takich Android SDK z wykorzystaniem języka Java, Android NDK, Xamarin oraz Apache Cordova. Dwa pierwsze narzędzia pozwalają na stworzenie aplikacji dla urządzeń wyposażonych z system Android. Kolejne z nich, Xamarin, pozwala na tworzenie aplikacji dla systemu Windows Phone. Apache Cordova pozwala na wykonanie aplikacji na urządzenia mobilne z wykorzystaniem technologii webowych. W tej pracy do zadań zaimplementowanych aplikacji należało: sortowanie tablicy oraz odczyt i zapis danych do pliku. Podczas przeprowadzonych badań, autorzy przeprowadzonej analizy nie mogli jednoznacznie stwierdzić, które z wcześniej wymienionych narzędzi pozwala na stworzenie najwydajniejszej aplikacji.

W pozostałych dwóch z trzech wspomnianych publikacji dokonano porównania aplikacji wykonanych przy użyciu Xamarina, Androida, iOS oraz postawiono pytanie czy zalecane jest użycie technologii wieloplatformowych w tworzeniu aplikacji mobilnych. W pierwszej z nich *The comparison of native apps performance on iOS (Swift) and Android with cross-platform application - Xamarin: student project* [6] autorzy stwierdzają, że aplikacja napisana w wieloplatformowej technologii Xamarin może być tak samo wydajna jak aplikacja natywna napisana w Android lub iOS i wybór stosownej technologii powinien być uzależniony od funkcjonalności wytwarzanej aplikacji mobilnej. Ponadto autorzy zaznaczają, że Xamarin Forms jest także w trakcie aktywnego rozwijania, więc nie jest on produktem całkowicie gotowym w środowisku produkcyjnym. Analiza wyników jakie zostały otrzymane w publikacji *Performance analysis of native and cross-platform mobile applications* [7] pokazały natomiast, iż w większości przypadków lepsze wyniki pod względem wydajności osiągały aplikacje natywne niż ta zaimplementowana w technologii Xamarin. Ważnym faktem stwierdzonym w obu publikacjach jest to, że Xamarin Forms znacznie upraszcza proces wytwarzania oprogramowania i pozwala skrócić czas potrzebny na wytworzenie aplikacji mobilnej, jednak wiedza na temat natywnych technologii jest konieczna by go używać.

Wszystkie z opisanych wyżej publikacji pozwalają wyciągnąć wniosek, iż przeprowadzanie badań porównujących różne technologie do tworzenia aplikacji mobilnych jest istotnym zagadnieniem. Takie badania pozwalają zgłębić wiedzę dotyczącą zagadnień wydajnościowych jak i poza wydajnościowych jakie mogą mieć wpływ przy tworzeniu aplikacji na urządzenia mobilne oraz mogą pomóc przy wyborze z wykorzysta-

niem jakiej technologii tworzenie takich aplikacji jest najbardziej optymalne. Dodatkowo, nie znaleziono publikacji która traktowałaby o porównaniu narzędzia Flutter z innym narzędziem, co było powodem dla którego autorzy niniejszego artykułu podjęli się tejże analizy.

3. Metoda badawcza

W celu przeprowadzenia badań wydajności utworzono dwie funkcjonalnie identyczne aplikacje mobilne przeznaczone na systemy Android przy zastosowaniu zestawów narzędzi programistycznych Android oraz Flutter.

Na potrzeby porównania wydajności tych dwóch zestawów narzędzi programistycznych przygotowano następujące scenariusze testowe:

- Sortowanie 10000, 100000 oraz 1000000 liczb naturalnych algorytmem Dual-Pivot Quicksort.
- Zapisanie do lokalnej bazy danych 100, 500 oraz 1000 rekordów.
- Odczytanie z lokalnej bazy danych 100, 500 oraz 1000 rekordów.
- Zapisanie 1000, 10000 oraz 100000 znaków alfanumerycznych do pliku tekstowego.
- Odczytanie 1000, 10000 oraz 100000 znaków alfanumerycznych z pliku tekstowego.

3.1. Środowisko badawcze

Do zbadania obciążenia procesora podczas operacji sortowania danych wykorzystano moduł Android Profiler oraz narzędzie DevTools. Natomiast do zbadania czasu wykonania scenariuszy testowych opisanych w punkcie 3 wykorzystane zostały dwie funkcjonalnie identyczne aplikacje testowe monitorujące oraz zapisujące do bazy danych czasy wykonania poszczególnych operacji.

Szczegółowa specyfikacja urządzenia fizycznego na którym uruchomiono aplikacje testowe przedstawiona została w Tabeli 1.

Tabela 1: Parametry techniczne urządzenia testowego

Model	Motorola G8 Power
Procesor	Qualcomm Snapdragon 665
Taktowanie procesora	2000 MHz
Liczba rdzeni	8
Pamięć RAM	4 GB
Pamięć ROM	64 GB
System operacyjny	Android 10

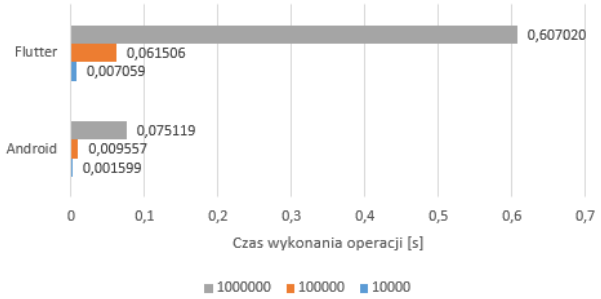
4. Analiza wyników badań wydajnościowych

W punkcie 4 zamieszczono wyniki przeprowadzonych badań wydajnościowych. Wszystkie czasy badań zostały przedstawione w sekundach.

Wykresy prezentują średnią trzydziestokrotnego wykonania poszczególnych eksperymentów na podstawie scenariuszy testowych opisanych w punkcie 3.

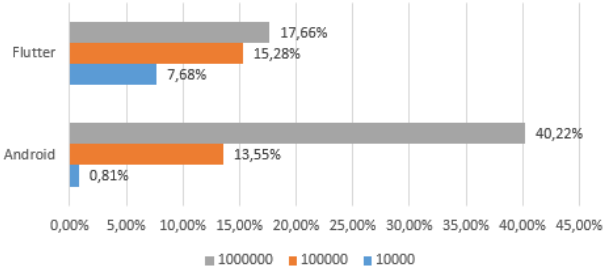
4.1. Sortowanie danych

Rysunek 1 przedstawia średni czas sortowania 10000, 100000 oraz 1000000 liczb naturalnych. Na wykresie można zauważyć, że aplikacja zbudowana w Androidzie wykonała te operacje szybciej od aplikacji wykonanej we Flutterze. Znaczącą różnicę widać przy każdej liczbie sortowanych liczb. Przy tych próbach aplikacja napisana w Androidzie okazuje się około siedmiokrotnie szybsza.



Rysunek 1: Zestawienie wyników pomiaru czasu wykonania operacji sortowania elementów.

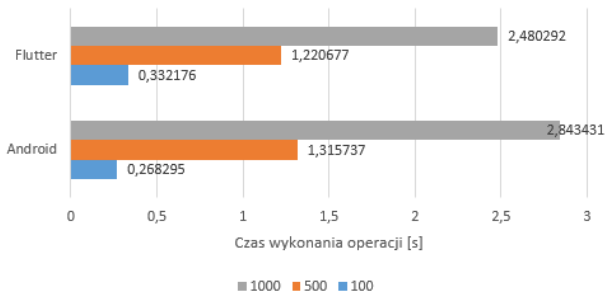
Na Rysunku 2 zamieszczono wyniki porównania obciążenia procesora podczas operacji sortowania elementów w strukturze danych. Na przedstawionym wykresie można zauważyć, że to Android mniej obciążył procesor podczas operacji sortowania. Warto podkreślić jednak dużą przewagę po stronie Fluttera podczas sortowania 1000000 liczb. W tym przypadku Flutter poradził sobie znacznie lepiej.



Rysunek 2: Zestawienie wyników pomiaru użycia procesora podczas wykonania operacji sortowania elementów.

4.2. Zapis do bazy danych

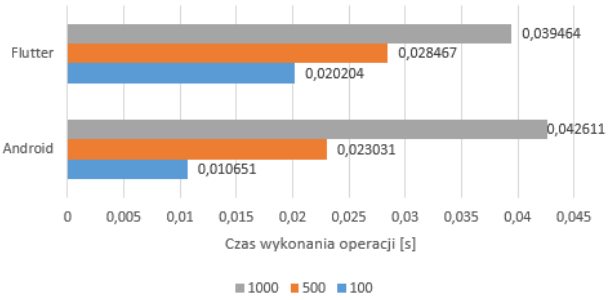
Na Rysunku 3 zaprezentowano średni czas zapisu danych do lokalnej bazy danych wykorzystującej silnik baz danych SQLite. Na podstawie przedstawionego wykresu, można wywnioskować, że aplikacja zaimplementowana przy użyciu Flutter SDK lepiej radziła sobie od aplikacji zaimplementowanej w Androidzie przy zapisie większej ilości danych. Android natomiast charakteryzuje się szybszym zapisem danych do bazy przy niewielkiej ilości.



Rysunek 3: Zestawienie wyników pomiaru czasu wykonania operacji zapisu do bazy danych.

4.3. Odczyt z bazy danych

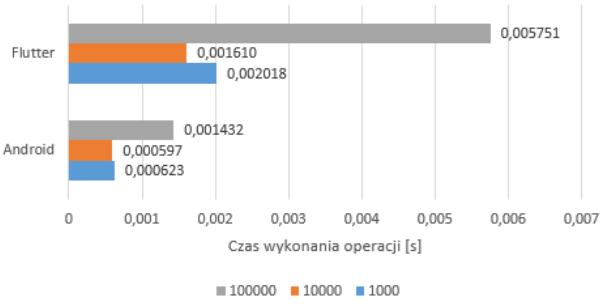
Wykres na Rysunku 4 przedstawia średni czas odczytu danych z lokalnej bazy danych opartej o silnik baz danych SQLite. Podobnie jak przy zapisie do bazy, podczas odczytu danych z bazy danych Flutter również okazał się dużo szybszy od Androida przy większej ilości danych. Przewaga Androida widoczna jest przy zapisie 100 elementów.



Rysunek 4: Zestawienie wyników pomiaru czasu wykonania operacji odczytu z bazy danych.

4.4. Zapis do pliku

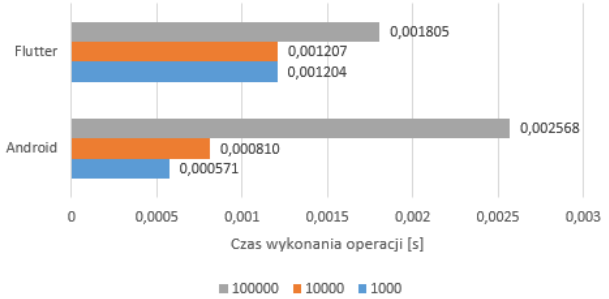
Wykres na Rysunku 5 prezentuje średni czas zapisu 1000, 10000, 100000 znaków alfanumerycznych do pliku tekstowego. Przy tym badaniu Android lepiej poradził sobie wykazując się mniejszymi średnimi czasami zapisu. Dużą przewagę Androida widać szczególnie przy zapisie każdej ilości znaków. Średnio Android wykonuje operację dwa razy szybciej.



Rysunek 5: Zestawienie wyników pomiaru czasu wykonania operacji zapisu do pliku.

4.5. Odczyt z pliku

Rysunek 5 przedstawia średni czas odczytu 1000, 10000 oraz 100000 znaków alfanumerycznych z pliku tekstowego. Z wykresu widać, że Android średnio szybciej przeprowadził badaną operację. Największa różnica widoczna jest przy 1000 i 10000 znakach. Podczas odczytu 100000 znaków to Flutter wykonał te operacje średnio w krótszym czasie lecz jego przewaga nad Androidem nie jest znacząca.



Rysunek 6: Zestawienie wyników pomiaru czasu wykonania operacji odczytu z pliku. odczytu z pliku.

5. Analiza wyników badań poza wydajnościowych

Przy tworzeniu aplikacji mobilnych istotną kwestią jest dobór odpowiedniej technologii, która pozwoli na uzyskanie jak najlepszych wyników pod względem wydajności. Jednakże, przy doborze technologii nie mniej ważne są kwestie różne od wydajności, która mają istotny wpływ na szybkość oraz jakość wytwarzanego oprogramowania. Dlatego podczas porównania Androida oraz Fluttera zbadano również cechy różne inne niż wydajność, tzn.:

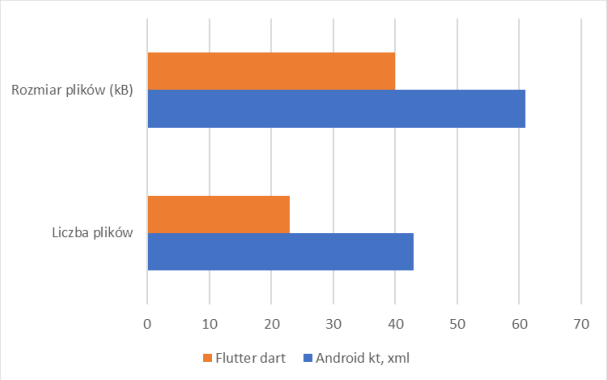
- liczba linii kodu źródłowego,
- dostępność bibliotek,
- wsparcie społeczności.

5.1. Liczba linii kodu źródłowego

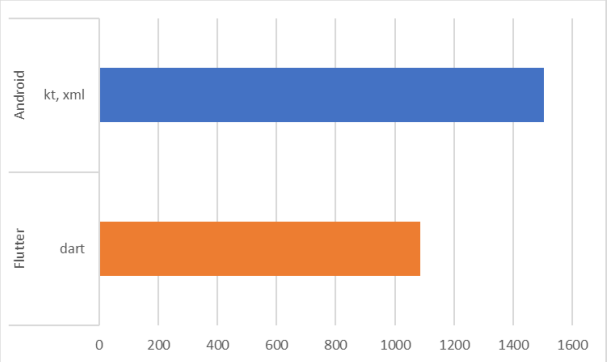
Do zbadania objętości kodu źródłowego posłużono się aplikacjami testowymi zaimplementowanymi w Android oraz Flutter SDK. Były to te same aplikacje, których użyto w porównaniu wydajnościowym. Badanie przeprowadzone zostało z użyciem wtyczki o nazwie *Statistic* dostępnej dla zintegrowanego środowiska programistycznego *Android Studio*.

Na potrzeby wykonania tego porównania analizie poddano pliki o rozszerzeniu:

- .dart - kod źródłowy aplikacji zaimplementowanej we Flutter SDK, który odpowiada zarówno za logikę jak i warstwę prezentacji
- .kt - kod źródłowy aplikacji zaimplementowanej w Android SDK odpowiadający za logikę
- .xml - kod źródłowy aplikacji zaimplementowanej w Android SDK odpowiadający za warstwę prezentacji oraz konfigurację



Rysunek 7: Porównanie rozmiaru oraz liczby plików zawierających kod źródłowy aplikacji testowych.

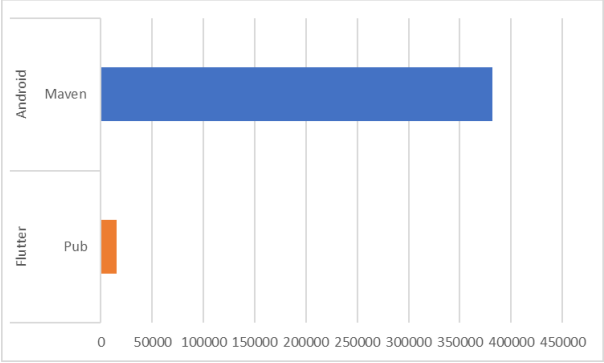


Rysunek 8: Porównanie liczby linii kodu.

Na podstawie analizy, której wyniki zamieszczono na wykresach zamieszczonych na Rysunku 7 oraz Rysunku 8 stwierdzono, że to Flutter posiada bardziej zwięzły kod. Wynikać to może z faktu, że zarówno logikę jak i warstwę prezentacji poszczególnych widoków zakodowano w tych samych plikach. W Androidzie natomiast istnieje jasny podział na kod odpowiadający za warstwę logiczną oraz prezentacji. Z praktycznego punktu widzenia może być to postrzegane jako zaleta ponieważ dzięki temu istnieje bardziej klarowny podział odpowiedzialności pomiędzy plikami źródłowymi. Podział ten ułatwia programiście poruszania się po kodzie źródłowym, szczególnie w zaawansowanych projektach programistycznych.

5.2. Dostępność bibliotek

W celu określenia liczby bibliotek dostępnych dla technologii Flutter wykorzystano repozytorium *Pub*, które jest menedżerem pakietów dla języka programowania Dart. Zawiera on biblioteki wielokrotnego użytku i pakiety dla programów Flutter, AngularDart i ogólnych programów Dart. Natomiast do zbadania dostępność bibliotek dla Android użyto repozytorium *Maven*, w którym przechowywane są wszystkie pliki z rozszerzeniem .jar bibliotek, wtyczki lub inne artefakty specyficzne dla projektów opartych na technologii Java.

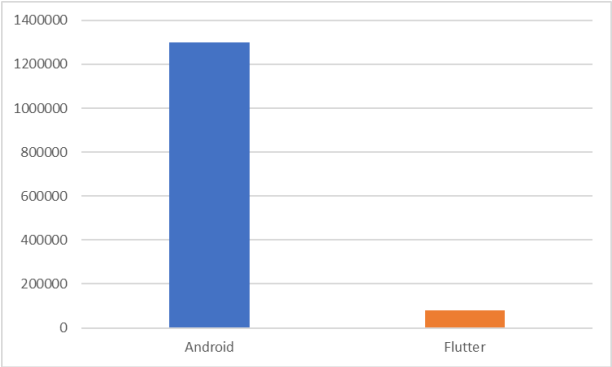


Rysunek 9: Porównanie liczby dostępnych bibliotek.

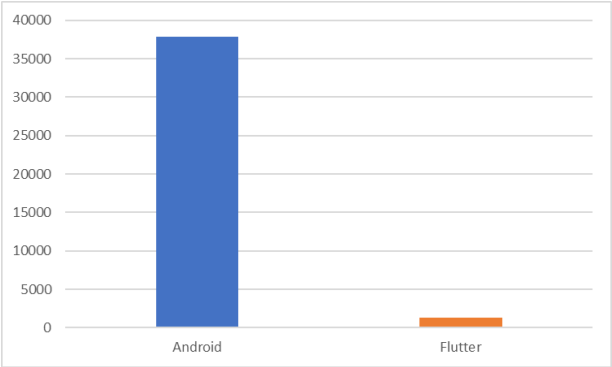
Wyniki przeprowadzonego porównania pod kątem liczby bibliotek przedstawiono na wykresie na Rysunku 9. Na ich podstawie widać, że repozytorium Maven, z którego korzysta Android jest znacznie bardziej liczne. Pod uwagę wziąć należy jednak fakt, że w przypadku Androida trudno jest określić jaki procent całości dostępnych bibliotek znajdzie swoje zastosowanie podczas wytwarzania aplikacji mobilnej przy użyciu Android SDK ponieważ w repozytorium Maven znajdują się biblioteki dla wszystkich technologii opartych o wirtualną maszynę Java. Niemniej jednak, nawet mając ten fakt na uwadze przewaga po stronie Androida wydaję się być na tyle znacząca, że to właśnie technologię Android można uznać za tę z większą liczbą dostępnych bibliotek.

5.3. Wsparcie społeczności

Do analizy wsparcia społeczności wykorzystano dane dostępne na serwisie społecznościowym *Stack Overflow*, na którym programiści zadają pytania dotyczące szeroko pojętego wytwarzania oprogramowania oraz na serwisie internetowym *Reddit*. Serwis ten prezentuje linki do różnorodnych informacji, które ukazały się w Internecie. Dane zebrane na Rysunku 10 oraz 11 przedstawiają te zapytania użytkowników, które zostały oznaczone pojęciem „Android” oraz „Flutter” w wyżej wymienionych serwisach.



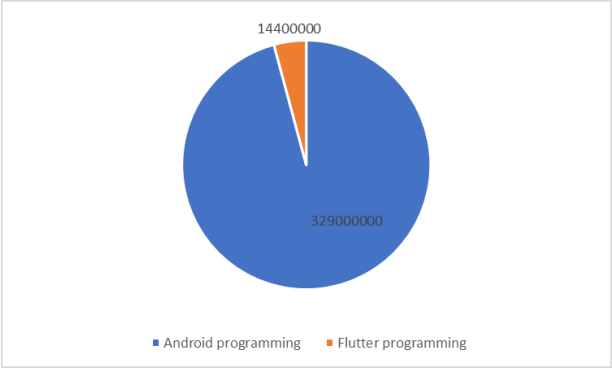
Rysunek 10: Porównanie liczby wyników na Stack Overflow.



Rysunek 11: Porównanie liczby wyników na Reddit.

Na podstawie otrzymanych wyników widać, że na zapytanie „Android" otrzymujemy znacznie więcej wyników niż na zapytanie „Flutter". Różnica na korzyść Androida jest bardzo znacząca ponieważ wynosi około 1,2 mln na Stack Overflow oraz 36,5 tys. na Reddit. Należy mieć jednak na uwadze fakt, iż w przypadku pojęcia „Android" wyszukiwarki pokazują nie tylko informacje o samym programowaniu w Android SDK, ale również wątki związane z systemem operacyjnym Android. Niestety obecnie żaden z wymienionych portali nie posiada na tyle zaawansowanej funkcji filtracji, która pozwalałaby by precyzyjne zawęzić tematykę prezentowanych wyników.

Oprócz sprawdzenia liczby zwracanych informacji na temat badanych zestawów narzędzi programistycznych na Stack Overflow oraz Reddit, autorzy postanowili zbadać ilość wyników zwracanych przez wyszukiwarkę Google odpowiednio dla pojęć „Flutter programming" oraz „Android programming". Wyniki z przeprowadzonego badania przedstawiono na wykresie kołowym na Rysunku 12.



Rysunek 12: Porównanie liczby wyników w wyszukiwarce Google.

Poddając analizie powyższe wyniki, stwierdzono, że to dla frazy „Android programming" wyszukiwarka Google zwraca znacznie więcej wyników.

Biorąc pod uwagę również poprzednie badanie dotyczące liczby wyników na Stack Overflow oraz Reddit stwierdzono, że przewaga na korzyść Android jeżeli chodzi o to kryterium porównawcze jest znacząca i to w przypadku wytwarzania oprogramowania za pomocą Android SDK programista może liczyć na większe wsparcie społeczności.

6. Wnioski

Celem przeprowadzonych badań było porównanie wielokryterialne aplikacji mobilnych zbudowanych przy zastosowaniu zestawów narzędzi programistycznych Android oraz Flutter.

Pierwszym z badanych kryteriów było obciążenie procesora podczas sortowania danych. W przeprowadzonym badaniu stwierdzono, iż aplikacja zbudowana za pomocą Androida w większości przypadków, mniej obciąża procesor podczas tej operacji. Podczas sortowania danych zmierzono również czasy, które Android oraz Flutter potrzebowały na jego przeprowadzenie. Android szybciej posortował dane w strukturze danych, niezależnie od jej wielkości.

Drugim kryterium był zapis oraz odczyt danych z lokalnej bazy danych. Po przeanalizowaniu wyników zaobserwowano, badania nie pozwalają na wysunięcie jednoznacznego wniosku. Przy małej ilości danych to Android charakteryzuje się krótszym czasem, natomiast Flutter lepiej poradził sobie z większą ilością danych.

Aplikacja zaimplementowana w Android SDK okazała się również bardziej wydajna w większości badań zapisu oraz odczytu danych z pliku tekstowego. Biorąc pod uwagę to oraz poprzednie przeprowadzone badanie dotyczące zapisu oraz odczytu z lokalnej bazy danych, nie jest możliwe jednoznaczne stwierdzenie, która z badanych technologii szybciej zapisuje lub odczytuje dane zapisane w pamięci lokalnej urządzenia.

Kolejne badania dotyczyły cech innych niż wydajność. Bazując na analizie kodu źródłowego aplikacji testowych stwierdzono, że Flutter posiada kod bardziej zwięzły lecz trudniejszy do nawigowania od programu natywnego ze względu na brak klarownego podziału pomiędzy warstwą logiki a prezentacji. Łatwość poruszania się po kodzie źródłowym może mieć kluczowe znaczenie szczególnie dla zaawansowanych projektów programistycznych dlatego pomimo mniejszej objętości kodu to Android zdaniem autorów prezentuje bardziej przemyślane podejście do podziału obowiązków pomiędzy plikami od Flutter.

Dwa kolejne kryteria różne od wydajności dotyczyły liczby dostępnych bibliotek oraz wsparcia społeczności. Android SDK okazał się narzędziem z dużo większym wsparciem społeczności programistycznej, co z kolei mogło mieć pewne przełożenie na liczbę dostępnych bibliotek zewnętrznych, które w większości są tworzone a następnie publikowane przez samą społeczność.

Na podstawie przeprowadzonych badań można stwierdzić, iż aplikacja zaimplementowana z wykorzystaniem zestawu narzędzi programistycznych Android jest często nie tylko podobnie wydajna lub nawet wydajniejsza ale również posiada lepiej zorganizowany kod źródłowy, większe wsparcie społeczności oraz liczbę dostępnych bibliotek zewnętrznych od aplikacji zbudowanej przy użyciu Flutter SDK. Mając to wszystko na uwadze stwierdzono, że to Android SDK jest obecnie bardziej zalecanym zestawem narzędzi programistycznych do wytwarzania oprogramowania na systemy operacyjne Android od Flutter SDK.

Literatura

[1] M. A. Khaled, R. S. Tariq, S. Khaled, Mobile Gaming Trends and Revenue Models, Springer International Publishing (2016).

[2] M. Naldi, Concentration in the mobile operating systems market, Universit'y of Rome Tor Vergata (2016).

[3] M. Napoli, Beginning Flutter: A Hands On Guide to App Development, John Wiley & Sons, 2019.

[4] D. Sulowski, G. Kozieł, Analiza porównawcza języków Kotlin i Java używanych do tworzenia aplikacji na system Android, Journal of Computer Sciences Institute 13 (2019) 354-358. https://doi.org/10.35784/jcsi.1332

[5] P. Kotarski, K. Śledź, J. Smołka, Analiza wydajności aplikacji mobilnych przy zastosowaniu różnych narzędzi programistycznych do ich budowy, Journal of Computer Sciences Institute 6 (2018) 68-72. https://doi.org/10.35784/jcsi.642

[6] D. Dobrzański, W. Zabierowski, The comparison of native apps performance on iOS (Swift) and Android with cross-platform application - Xamarin: student project, International Journal of Microelectronics and Computer Science 8 (2018) 112-116.

[7] P. Grzmil, M. Skublewska-Paszkowska, E. Łukasik, J. Smołka, Performance analysis of native and cross-platform mobile applications, Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska 7 (2017) 50-53. https://doi.org/10.5604/01.3001.0010.4838

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 114–120
Received: 25 March 2021
Accepted: 29 March 2021

Evaluation of the availability of websites of communes in the Lubelskie Province

Ocena dostępności stron internetowych urzędów gmin w województwie lubelskim

Michał Bednarczyk*, Mariusz Dzieńkowski

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

Abstract

The article discusses the results of the research on the accessibility of 190 official websites of commune offices in the Lubelskie Province. For this purpose, an experiment was developed which consisted of two stages: the first one in which three automatic tools were used, and the second one for the needs of which a checklist containing eight criteria was developed and used. In this manner, two indicators were obtained specifying the percentage level of website availability. Afterwards the conditions specifying when the website of the commune office will be considered as meeting accessibility requirements were adopted. It was established that fulfilling the requirements would take place after the website achieved the result no lower than 80% in the automatic survey and no lower than 50% in the expert analysis. After conducting the research, on the basis of the collected results, the general level of accessibility of websites of communes from the Lubelskie Province was specified. For the automatic analysis, it was 74.92% and for the expert analysis – 45.99% and in both cases it was lower than the assumed thresholds. Only 33 communes reached or exceeded both of the established accessibility thresholds.

Keywords: website accessibility; accessibility evaluation; web accessibility testing techniques; accessibility automated tools; commune office websites

Streszczenie

W artykule omówiono wyniki badań dostępności 190 serwisów internetowych urzędów gmin województwa lubelskiego. W tym celu przygotowano eksperyment, który składał się z dwóch etapów: pierwszego, w którym wykorzystano trzy narzędzia automatyczne oraz drugiego, na potrzeby którego opracowano, a następnie zastosowano listę kontrolną zawierającą osiem kryteriów. W ten sposób uzyskano dwa wskaźniki określające procentowy poziom dostępności serwisów www. Następnie przyjęto warunki precyzujące, kiedy strona internetowa urzędu gminy będzie uznawana za spełniającą wymagania dostępności. Ustalono, że będzie miało to miejsce, gdy serwis uzyska wynik co najmniej 80% w badaniu automatycznym oraz co najmniej 50% w analizie eksperckiej. Po przeprowadzeniu badań, na podstawie zebranych wyników określono ogólny poziom dostępności serwisów gmin z woj. lubelskiego. Dla analizy automatycznej wyniósł on 74,92%, a dla eksperckiej 45,99% i w obu przypadkach był niższy od założonych progów. Okazało się, że tylko 33 gminy osiągnęły lub przekroczyły oba ustalone progi dostępności.

Słowa kluczowe: dostępność stron internetowych; ocena dostępności; metody badania dostępności stron www; automatyczne narzędzia do testowania dostępności; strony urzędów gminy

*Corresponding author

Email address: michal.bednarczyk1@pollub.edu.pl (M. Bednarczyk)

1. Wstęp

Rozwój techniki wraz z postępującym rozpowszechnieniem szerokopasmowego internetu zaowocował zwiększeniem dostępności do informacji, ułatwił bezpośredni kontakt firm i instytucji z potencjalnymi klientami i użytkownikami. Rozwój portali informacyjnych, społecznościowych i stron firmowych, a także postęp w obszarze urządzeń mobilnych, smartfonów i laptopów znacząco zmniejszył bariery komunikacyjne i informacyjne. Przykładami serwisów internetowych, do których niezbędny jest szybki, niezawodny i bezustanny dostęp należą strony urzędów gmin i miast, serwisy szkół i uczelni, witryny urzędów państwowych i organów kontroli państwowej, a także banków i firm. Obecnie praktycznie każdy z wymienionych podmiotów posiada własną witrynę, która zawiera oprócz podstawowego opisu działalności i danych kontaktowych wiele innych istotnych informacji.

Zdecydowana większość współczesnego społeczeństwa poszukuje i otrzymuje interesujące informacje poprzez sieć i witryny internetowe. Według badań Głównego Urzędu Statystycznego przeprowadzonych w 2020 roku [1], w województwie lubelskim na 652 126 gospodarstw domowych 568 978, czyli 87,2% posiada stały dostęp do Internetu. W skali kraju, przy liczbie 1 918 900 osób niepełnosprawnych korzystanie z Internetu deklaruje aż 1 315 622 czyli 68,6%. W związku z tym rodzi się potrzeba zapewnienia swobodnego dostępu wszystkim zainteresowanym tak, aby każdy bez większego problemu mógł korzystać ze współczesnych rozwiązań. W społeczeństwie należy szczególną uwagę zwrócić na osoby z różnymi rodzajami i stopniami nie-

pełnosprawności. Witryny internetowe powinny być dostępne zarówno pod względem informacyjnym jak i technicznym. To, że serwis będzie dostępny na dowolnej, lub dużej liczbie urządzeń nie oznacza jeszcze, że spełnia on wymagania dostępności. Istnieje wiele zaleceń i aktów prawnych, wydanych zarówno przez państwo polskie [2-3] jak i Parlament Europejski [4-5], które ściśle określają, w jaki sposób witryna internetowa powinna być zaprojektowana oraz w jakie funkcjonalności powinna być wyposażona. Pomimo tego, że o problematyce dostępności witryn internetowych mówi się od bardzo dawna, w szczególności w przypadku jednostek publicznych, wiele z nich nadal używa przestarzałych rozwiązań [6-7]. Doprowadza to do sytuacji, w której nawet osoby pełnosprawne natrafiają na problemy w użytkowaniu takich witryn, a dla osób niepełnosprawnych stają się one zupełnie niedostępne. Jest to obecnie duży problem, który w pewnym stopniu został już zauważony [8].

W niniejszej pracy za cel obrano wykonanie oceny dostępności witryn internetowych urzędów gmin województwa lubelskiego. W związku z tym przeprowadzono wieloaspektową analizę zrealizowaną przy pomocy narzędzi automatycznych oraz przygotowanej listy kontrolnej. Badania objęły między innymi takie aspekty jak responsywność witryn, dostępność funkcji ułatwiających korzystanie ze stron osobom niepełnosprawnym, a także zgodność z wymaganiami określonymi przez organizację W3C i standard WCAG 2.1 [9].

2. Przegląd literatury

Od instytucji prowadzących własne serwisy internetowe wymaga się, aby były one dostępne na określonym poziomie. W przypadku podmiotów publicznych kwestie te są uregulowane prawnie. W deklaracji dostępności, instytucje publiczne są zobowiązane określić aktualny stopień dostępności oraz plan dojścia do wymaganego poziomu. Dotyczy to nie tylko stron internetowych, ale także innych aspektów, np. dostępności architektonicznej siedziby podmiotu dla osób niepełnosprawnych [2]. W związku z tym dostępność powinna być okresowo kontrolowana i podlegać ewaluacji. Taka ocena może być przeprowadzana automatycznie przez różnego rodzaju specjalistyczne oprogramowanie lub manualnie za pomocą technik, metod i narzędzi wykorzystywanych w audycie i ewaluacji serwisów [10]. Wymienione metody są oczywiście pomocne, ale ostateczna decyzja o dostępności powinna być podejmowana po badaniach przeprowadzonych na docelowej grupie odbiorców. Według autorów pracy [11] testowanie z udziałem ludzi jest najlepszą metodą oceny dostępności stron internetowych, chociaż w publikacjach z tej tematyki [12] częściej stosuje się automatyczne narzędzia do oceny dostępności. Użytkowników w testach eksperckich i badaniach kwestionariuszowych wykorzystał Jaeger [13]. Z kolei Ichsani [14] prowadził badania użyteczności i dostępności witryn administracji przy użyciu metody „think aloud". Autorów wielu publikacji wskazuje, że najskuteczniejszym sposobem oceny zgodności danej strony internetowej z dostępnością jest połączenie zarówno narzędzi automatycznych, jak i manualnych procedur oceny. Narzędzia automatyczne skutecznie identyfikują błędy dostępności, ale nie na takim poziomie, który może osiągnąć doświadczony użytkownik. Z tego względu zautomatyzowane programy nie powinny zastępować ręcznej oceny, ale stanowić uzupełnienie pełnej procedury oceny [15-16].

Autorzy pracy [17] oceniali poziom dostępności serwisów internetowych uczelni wyższych Lublina i porównali je z serwisami dwóch wiodących uczelni w Polsce. W swoich badaniach wykorzystali dziesięciopunkową listę kontrolną oraz zestaw czterech narzędzi automatycznych. Materiał badawczy obejmował po osiem stron z każdego sprawdzanego serwisu.

W artykule [10] do badań wykorzystano również narzędzia automatyzujące czynności polegające na weryfikacji dostępności. Przebadano cztery strony www użyteczności publicznej. Podczas ich analizy uwzględniono rozmiar analizowanych serwisów wyrażony w postaci liczby zawartych w nich elementów.

W ramach kolejnej pracy [12], przebadano rządowe serwisy trzydziestu czterech prowincji Indonezji, używając dwóch narzędzi automatycznych bazujących na wytycznych dotyczących dostępności treści internetowych WCAG 2.0. W tym przypadku w procesie oceny dostępności skoncentrowano się wyłącznie na stronach głównych, odpowiadających za pierwsze wrażenie i stanowiących bramę do całego serwisu.

Kompleksową procedurę oceny dostępności stron www [18] zaproponowali badacze z Portugali. Opiera się ona na podstawie iberyjskiej zgodności w zakresie dostępności e-zdrowia i jest odpowiedzią na istniejące problemy związane z dostępnością. Procedura ta składa się z trzech powiązanych ze sobą perspektyw: automatycznej oceny dostępności stron www, ręcznej oceny dostępności oraz oceny heurystycznej użyteczności witryn www.

3. Metoda badań

Badaniu dostępności w niniejszej pracy została poddana duża większość, bo aż 190 z 213 stron gminnych w województwie lubelskim [19]. Ze względu na złożoność badanego zagadnienia badanie zostało podzielone na dwa etapy: analizę automatyczną i ekspercką. Analiza automatyczna została przeprowadzona przy pomocy trzech narzędzi: Utilitia [20], Tingun Page Checker [21] oraz Google PageSpeed Insights [22]. Procedura badawcza polegała na wprowadzeniu adresu URL kolejno każdej z badanych gmin do odpowiednich pól formularzy wybranych walidatorów i zapisaniu wyniku uzyskanego przez serwis danej gminy. Przy wyborze narzędzi kierowano się kilkoma kryteriami. Najważniejszym z nich było zwracanie jednolitego wyniku w postaci liczbowej w skali od 0 do 100. Pozwoliło to na łatwe porównanie danych z wszystkich trzech walidatorów i obliczenie średniej oceny dostępności serwisu www każdej gminy. Narzędzia dobrano w taki sposób, aby się wzajemnie uzupełniały i przez to dokładnie i wieloaspektowo analizowały strony www danej gminy. Usługa Utilitia umożliwia testowanie całych serwisów według

wszystkich trzech poziomów dostępności WCAG. Narzędzie Tingtun Page Checker, podobnie jak serwis Utilitia, pozwala określić poprawność kodu HTML i CSS oraz dodatkowo sprawdzić poprawność znaczników i obecność tekstów alternatywnych dla zdjęć. Z kolei narzędzie Google PageSpeed Insights koncentruje się głównie na kwestiach wydajnościowych, biorąc pod uwagę czas dostępu do informacji. Ponadto potrafi zdiagnozować błędy dotyczące optymalizacji użytych obrazów graficznych na stronie oraz sprawdzić ich rozmiar i czas wczytywania zarówno na komputerach jak i urządzeniach mobilnych. Podaje również wskazówki mające na celu zoptymalizowanie witryny, od której zależy wysoka pozycja serwisu w wyszukiwarkach, co ma wpływ na promocję gminy w internecie. Narzędzia Utilitia i Google PageSpeed Insights są jednymi z najbardziej popularnych narzędzi do przeprowadzania testów, a wskazania z Tingtun Page Checker są uznawane jako wiarygodne przez Unię Europejską. Ważnym aspektem jest to, iż wybrane narzędzia są dostępne bezpłatnie.

W ramach niniejszej pracy wyniki z badań dostępności witryn dla każdego powiatu zostały umieszczone w oddzielnych tabelach przechowujących następujące dane: nazwę gminy, trzy oceny z programów walidacyjnych oraz obliczoną z tych ocen średnią arytmetyczną. Identyczne tabele zostały przygotowane dla wszystkich dwudziestu przebadanych powiatów. Dla badania automatycznego, jako punkt graniczny przyjęto poziom wskaźnika dostępności na poziomie 80%. Serwisy osiągające taki lub wyższy wynik kwalifikowane były jako dostępne.

Badania eksperckie zostały wykonane przez jedną osobę dobrze znającą tematykę dostępności stron www oraz mającą kwalifikacje i doświadczenie w tym zakresie. Dokonana analiza polegała na ocenie ośmiu najważniejszych według autorów pracy kryteriów dostępności ujętych w postaci listy kontrolnej. Każde z kryteriów podlegało ocenie w skali 0 - 2 punkty. Maksymalny wynik, który był możliwy do uzyskania przez serwis gminy to 16 punktów. Do oceny wybrano następujące kryteria dostępności:

Responsywność

- 0 pkt – brak responsywności (brak możliwości korzystania ze strony na urządzeniach mobilnych, strona nie reaguje na zmianę rozdzielczości i rozmiaru okna przeglądarki)
- 1 pkt – częściowa responsywność (możliwe ograniczone korzystanie ze strony, strona w pewnym stopniu reaguje na zmianę rozdzielczości i rozmiaru okna przeglądarki)
- 2 pkt – pełna responsywność (brak błędów lub drobne błędy nie mające wpływu na korzystanie z witryny)

Kontrast

- 0 pkt – brak możliwości zmiany kontrastu
- 1 pkt – możliwość zmiany kontrastu (1 opcja)
- 2 pkt – możliwość zmiany kontrastu (więcej niż 1 opcja)

Zmiana rozmiaru czcionki

- 0 pkt – brak możliwości zmiany rozmiaru czcionki
- 1 pkt – możliwość zmiany rozmiaru czcionki (1 opcja)
- 2 pkt – możliwość zmiany rozmiaru czcionki (opcje: pomniejsz, powiększ oraz resetuj rozmiar)

Opcje drukowania i wyświetlania w formacie PDF

- 0 pkt – brak opcji druku i zapisu do PDF
- 1 pkt – opcja wydruku lub zapisu do PDF
- 2 pkt – dostępne obie opcje

Funkcja lektora

- 0 pkt – brak opcji lektora
- 1 pkt – lektor tylko dla treści wpisów
- 2 pkt – lektor dla całości witryny

Wyszukiwarka

- 0 pkt – brak wyszukiwarki
- 1 pkt – wyszukiwarka podstawowa
- 2 pkt – wyszukiwarka zaawansowana

Dokumenty wymagane przez prawo

- 0 pkt – brak dokumentów Deklaracji dostępności i Polityki prywatności
- 1 pkt – obecność jednego dokumentu: Deklaracji dostępności lub Polityki prywatności
- 2 pkt – obecność obu dokumentów: Deklaracji dostępności i Polityki prywatności

Wybór języka

- 0 pkt – brak możliwości zmiany języka
- 1 pkt – zmiana języka (1 opcja)
- 2 pkt – zmiana języka (więcej niż 1 opcja)

W celu czytelnego przedstawienia danych pozyskanych z badań, wyniki zostały zebrane do oddzielnych tabel pogrupowanych według przynależności do danego powiatu. Zsumowane punkty z wszystkich kryteriów i przeliczone na wartość procentową dla danej gminy stanowią wskaźnik dostępności serwisu pochodzący z analizy eksperckiej przeprowadzonej manualnie przy pomocy listy kontrolnej. Dla tej analizy przyjęto, że serwisy, których wskaźnik dostępności osiągnie lub przekroczy próg 50% będą kwalifikowane jako dostępne. Ostatecznie strona danej gminy zostanie uznana jako dostępna, jeśli oba wskaźniki spełnią kryterium dostępności tzn. dla analizy automatycznej równy lub wyższy 80% oraz dla analizy eksperckiej równy lub wyższy 50%.

Badania przeprowadzono na komputerze, który zawierał odpowiednią ilość pamięci operacyjnej dla przeglądarki internetowej i był połączony z monitorem o rozdzielczości FullHD. Taka konfiguracja stwarzała optymalne warunki umożliwiające bezproblemowe analizowanie serwisów internetowych i nie wpływała na uzyskiwane wyniki. Wszystkie witryny były otwierane i następnie poddawane analizie w przeglądarce Google Chrome. Natomiast przy sprawdzaniu responsywności, realizowanej podczas analizy eksperckiej, został użyty tryb programisty tej samej przeglądarki. Ważną kwestią podczas badań było zapewnienie stabilnego połączenia

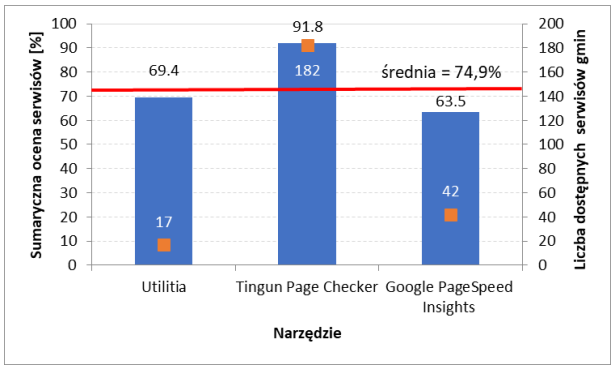
internetowego, które miało przepustowość 20 Mb/s. Wykorzystane oprogramowanie oraz sprzęt były zaktualizowane do najnowszej, udostępnionej przez producentów wersji. Dokładna konfiguracja stanowiska badawczego została przedstawiona w Tabeli 1.

Tabela 1: Stanowisko badawcze wykorzystane w badaniach dostępności

Laptop	**Asus Vivobook s15 S510UN-16** • Procesor Intel Core i7-8550U (4 rdzenie, 8 wątków, 1.80-4.00 Ghz, 8MB cache) • Pamięć RAM 16 GB SO-DIMM DDR4, 2133Mhz • Karta graficzna NVIDIA GeForce MX150 2048MB GDDR5 • Ekran matowy, LED 15,6" 1920x1080 (FullHD) • Wi-Fi 5 (802.11 a/b/g/n/ac) • Windows 10 Home 64-bit
Przeglądarka internetowa	**Google Chrome** • wersja 87.0.4280.66 (64-bitowa)
Router	**Netgar R6220 (802.11a/b/g/n/ac 1200 Mb/s)** • Obsługiwany standard Wi-Fi 5 (802.11 a/b/g/n/ac) • Częstotliwość pracy 2.4 / 5 Ghz (Dual-band)
Programy walidujące	**Utilitia** [20] **Tingun Page Checker** [21] **Google PageSpeed Insights** [22]

4. Wyniki badań

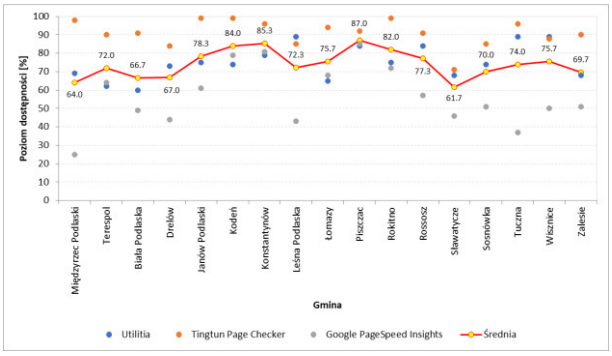
Podczas analizy automatycznej wykorzystano narzędzia, które jak pokazuje Rysunek 1, bardzo różnie oceniały poddane badaniom serwisy. Wykres przedstawia średnie oceny wystawione przez każdy walidator dla 190 serwisów urzędów gmin z woj. lubelskiego.



Rysunek 1: Ocena dostępności wszystkich przebadanych gmin za pomocą trzech automatycznych narzędzi wraz z liczbą serwisów gmin, które osiągnęły poziom dostępności powyżej 80%.

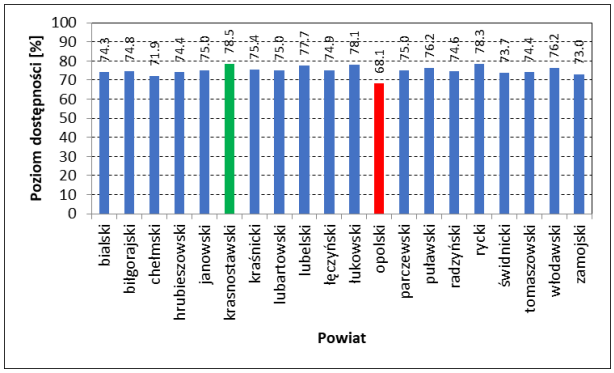
Najwyższe oceny (91,8%) generował Tingun Page Checker, bliskie średniej Utilitia (69,4%), natomiast najniższe Google PageSpeed Insights (63,5%). Na rysunku pokazano również liczbę serwisów, które były oceniane według danego walidatora jako dostępne (wynik ≥ 80%). Po sprawdzeniu serwisów narzędziem Utilitia tylko 17 z nich osiągnęło lub przekroczyło próg 80%, natomiast zdecydowanie najwięcej, bo aż 182 serwisy zostały uznane za dostępne po badaniach zrealizowanych za pomocą usługi Tingun Page Checker.

Badaniom zostały poddane prawie wszystkie serwisy gmin z dwudziestu powiatów należących do województwa lubelskiego. Podczas analizy automatycznej każdy serwis urzędu gminy został przetestowany trzema dobranymi narzędziami. Średnia trzech wyników zwróconych przez walidatory stanowiła końcową ocenę dostępności serwisu. Uzyskane wyniki dla poszczególnych gmin pogrupowano według powiatów, umieszczono w specjalnie do tego celu przygotowanych tabelach i zilustrowano w postaci wykresów. Na Rysunku 2 pokazano przykładowy wykres obrazujący wyniki dostępności dla serwisów wszystkich gmin z powiatu bialskiego.



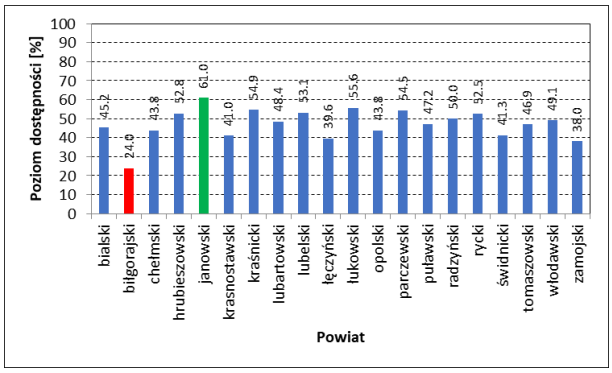
Rysunek 2: Ocena dostępności serwisów wszystkich gmin powiatu bialskiego wykonana przy pomocy narzędzi automatycznych.

Na Rysunku 3 przedstawiono zsumowane i uśrednione wyniki dostępności wszystkich gmin dla poszczególnych powiatów, które zostały wykonane narzędziami automatycznymi. Wśród dwudziestu powiatów województwa lubelskiego najwyższy wskaźnik dostępności, na poziomie 78,5% osiągnął powiat krasnostawski, składający się z dziesięciu gmin. Z kolei najniższą średnią ocenę uzyskał powiat opolski (68,1%), w skład którego wchodzi siedem gmin.



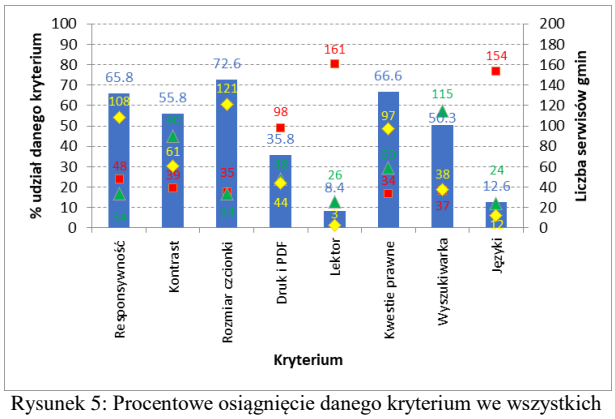
Rysunek 3: Poziomy dostępności serwisów urzędów gmin pogrupowane według powiatów (analiza wykorzystująca automatyczne narzędzia).

Analiza ekspercka (Rysunek 4) wykazała, że najwyższy poziom dostępności osiągnęły serwisy gmin wchodzących w skład powiatu janowskiego (61%), a wyraźnie najniższy poziom uzyskały serwisy gmin powiatu biłgorajskiego (24%).



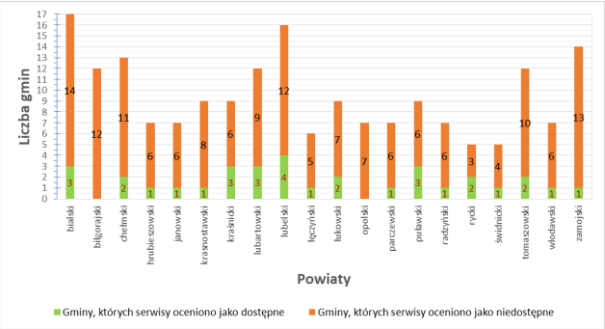
Rysunek 4: Poziomy dostępności serwisów urzędów gmin pogrupowane według powiatów (analiza ekspercka).

Na Rysunku 5, w formie graficznej przedstawiono wyniki ilościowe ukazujące liczbę gmin i jednocześnie poziom osiągnięcia przez nie danego kryterium po analizie wykorzystującej listę kontrolną. Do tego celu wykorzystano trzy znaczniki o różnych kształtach i kolorach, nad lub pod którymi została umieszczona wartość liczbowa. Na wykresie w postaci niebieskich słupków pokazano także poziom osiągnięcia danego kryterium przez wszystkie przebadane serwisy. Wykres pokazuje, że spośród funkcjonalności powiązanych bezpośrednio z dostępnością, najwięcej serwisów, bo aż 121 (72,6%) umożliwiało zmianę rozmiaru czcionki, trochę mniej - 108 (65,8%) pozwalało uzyskać pełną responsywność, natomiast 97 (66,6%) zawierało niezbędne dokumenty wymagane prawnie. Tylko 12 (12,6%) witryn umożliwiało zmianę języka i tylko 3 (8,4%) serwisy udostępniały funkcję lektora. Na poniższym wykresie kolorem żółtym oznaczono liczbę serwisów gmin, które w pełni osiągają dane kryterium, kolorem zielonym tylko te serwisy, które częściowo spełniają określone kryterium oraz kolorem czerwonym, witryny, które nie spełniają danego kryterium.



Rysunek 5: Procentowe osiągnięcie danego kryterium we wszystkich przebadanych serwisach urzędów gmin woj. lubelskiego oraz liczba gmin, które uzyskały wynik 0 pkt (znacznik w kolorze czerwonym), 1 pkt (znacznik w kolorze zielonym) i 2 pkt (znacznik w kolorze żółtym).

Zgodnie z przyjętymi założeniami, serwis gminy był uznawany za spełniający kryteria dostępności, jeśli zaliczył zarówno test automatyczny (wynik równy lub wyższy niż 80%) jak i test ekspercki (wynik równy lub wyższy niż 50%). Jak pokazuje Rysunek 5, najwięcej gmin posiadających serwisy dostępne znalazło się w powiecie lubelskim (4 gminy) oraz bialskim, kraśnickim, lubartowskim i puławskim (po 3 gminy). Natomiast w powiatach biłgorajskim i opolskim żadna strona gminy nie została oceniona jako dostępna. Niewiele lepiej było w powiatach hrubieszowskim, janowskim, krasnostawskim, łęczyńskim, parczewskim, radzyńskim, świdnickim, włodawskim i zamojskim, w których tylko jedna gmina dysponowała tzw. dostępnym serwisem urzędu.



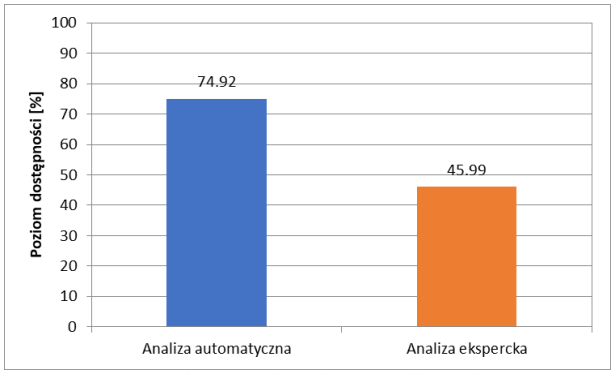
Rysunek 6: Liczba gmin zawierających serwisy dostępne/niedostępne według powiatów (analiza ekspercka).

W Tabeli 2 zestawiono wszystkie 33 gminy z województwa lubelskiego, których serwisy spełniają wymagania dostępności (2 wskaźniki osiągnęły lub przekroczyły ustalone progi), co stanowi 17,4% ze 190 przeanalizowanych witryn urzędów gmin.

Tabela 2. Lista gmin spełniających kryteria dostępności

Gmina		Wynik badania automatycznego	Wynik badania eksperckiego
1	Kodeń	84	56
2	Konstantynów	85	56
3	Piszczac	87	69
4	Siedliszcze	81	50
5	Wojsławice	82	75
6	Werbkowice	83	69
7	Batorz	78	75
8	Izbica	83	75
9	Annopol	82	69
10	Dzierzkowice	80	50
11	Urzędów	84	63
12	Abramów	82	63
13	Ostrówek	87	56
14	Uścimów	84	56
15	Konopnica	84	63
16	Niedrzwica Duża	82	50
17	Niemce	81	69
18	Zakrzew	81	56
19	Spiczyn	84	50
20	Stoczek Łukowski	84	50
21	Adamów	81	75
22	Dębowa Kłoda	83	50
23	Kurów	83	63
24	Markuszów	80	56
25	Wąwolnica	80	56
26	Wohyń	81	75
27	Dęblin	81	75
28	Kłoczew	88	75
29	Piaski	86	50
30	Jarczów	89	56
31	Łaszczów	85	50
32	Wyryki	81	75
33	Adamów	80	75

Rysunek 7 stanowi podsumowanie zrealizowanych badań. Pokazuje on poziomy wskaźników dostępności, które osiągnęły przebadane serwisy urzędów gmin z województwa lubelskiego. Pierwszy wskaźnik, wyznaczony na podstawie analizy automatycznej, osiągnął wynik 74,92%, czyli prawie o 5% mniej niż przyjęta granica 80% wyznaczająca próg dostępności. Natomiast wynik drugiego wskaźnika, którego wartość została określona na bazie analizy eksperckiej, wyniósł 45,99%, czyli około 4% mniej niż założona granica 50%.



Rysunek 7: Ocena dostępności serwisów urzędów gmin przeprowadzona dwoma metodami.

5. Podsumowanie

W ramach pracy przebadano pod względem dostępności 190 na 213 gmin z woj. lubelskiego, co dało bardzo duży zbiór danych. Każdą gminę przetestowano trzema narzędziami automatycznymi oraz manualnie przy pomocy ośmiokryterialnej listy kontrolnej. Po uśrednieniu wyników otrzymano dwie oceny. Pierwsza stanowiła wskaźnik dostępności wyznaczony jako średnia rezultatów pochodzących z trzech walidatorów natomiast druga stanowiła średnią punktów przyznanych dla poszczególnych kryteriów podczas analizy eksperckiej.

Na podstawie uzyskanych wyników ogólny poziom dostępności wybranych serwisów gmin z woj. lubelskiego określony za pomocą dwóch wskaźników: z analizy automatycznej i eksperckiej wyniósł odpowiednio 74,92% oraz 45,99% i był w obu przypadkach niższy od założonych progów. Okazało się również, że tylko 33 gminy osiągnęły lub przekroczyły oba ustalone progi dostępności. Wśród 190 gmin najwyższe wyniki, biorąc pod uwagę oba wskaźniki, osiągnęły gminy Kłoczew (88% i 75%), Izbica (83% i 75%) oraz Wojsławice (82% i 75%). W skali powiatu najwyższe wyniki uzyskały powiaty janowski (75% i 61%) i łukowski (78,1%, 55,6%).

W podsumowaniu należy wspomnieć, że niniejsza praca zawiera pewne ograniczenia. Z powodów technicznych, niektórych serwisów gmin nie udało się przetestować. Kwestią dyskusyjną może być dobór narzędzi automatycznych oraz dobór kryteriów przy ocenie eksperckiej. Kolejny aspekt to czas realizacji omawianych badań, które zostały przeprowadzone kilka miesięcy temu. Gminy są zobligowane prawnie do zapewnienia dostępności do swoich serwisów i w wielu przypadkach prace nad poprawą dostępności na pewno cały czas trwają, a więc obecnie poziom dostępności serwisów może odbiegać od wyników przedstawionych w niniejszej pracy.

Przeprowadzone badania pokazały pewien obraz stanu, w jakim znajdują się serwisy www jednostek publicznych woj. lubelskiego. Wynika z niego, że w sferze dostępności stron internetowych jest jeszcze dużo do zrobienia. Uzyskane wyniki i sformułowane wnioski pokazują także, jakie elementy wymagają interwencji lub poprawy. Nakładane przez prawo obowiązki względem podmiotów publicznych odnośnie zapewnienia dostępności nie tylko osób niepełnosprawnych, ale także dla różnych grup społeczeństwa, wzmacniają potrzebę przeprowadzania badań w opisywanym zakresie, szczególnie, gdy celem badawczym jest nie tylko diagnoza obserwowanej sytuacji, ale także pośredni wpływ na stopniową poprawę stanu e-administracji.

Literatura

[1] Główny Urząd Statystyczny, Wykorzystanie technologii informacyjno-komunikacyjnych w jednostkach administracji publicznej, przedsiębiorstwach i gospodarstwach domowych w 2020 roku, https://stat.gov.pl/obszary-tematyczne/nauka-i-technika-spoleczenstwo-informacyjne/spoleczenstwo-informacyjne/wykorzystanie-technologii-informacyjno-komunikacyjnych-w-jednostkach-administracji-publicznej-przedsiebiorstwach-i-gospodarstwach-domowych-w-2020-roku,3,19.html#, [14.03.2021].

[2] Ustawa z dnia 4 kwietnia 2019 r. o dostępności cyfrowej stron internetowych i aplikacji mobilnych podmiotów publicznych, https://isap.sejm.gov.pl/isap.nsf/download.xsp/WDU20190000848/T/D20190848L.pdf, [14.03.2021].

[3] Ustawa z dnia 10 maja 2018 roku o ochronie danych osobowych, https://isap.sejm.gov.pl/isap.nsf/download.xsp/WDU20180001000/T/D20181000L.pdf, [14.03.2021].

[4] Dyrektywa Parlamentu Europejskiego i Rady UE 2016/2102 z dnia 26 października 2016 r. w sprawie dostępności stron internetowych i mobilnych aplikacji organów sektora publicznego, https://eur-lex.europa.eu/legal-content/PL/TXT/PDF/?uri=CELEX:32016L2102&from=PL, [14.03.2021].

[5] Rozporządzenie Parlamentu Europejskiego i Rady Unii Europejskiej 2016/679 z dnia 27 kwietnia 2016 roku w sprawie ochrony osób fizycznych w związku z przetwarzaniem danych osobowych i w sprawie swobodnego przepływu takich danych, https://www.uodo.gov.pl/pl/131/224, [14.03.2021].

[6] K. Kowalik, Samorządowe media internetowe–uwarunkowania społecznoprawne wdrażania wymagań WCAG 2.0. Próba diagnozy dostępności (web accessibility) i użyteczności (web usability), Studia Medioznawcze, 61(2) (2015) 55-64, http://cejsh.icm.edu.pl/cejsh/element/bwmeta1.element.desklight-a9bef399-8863-4895-8244-dc7553bad518, [14.03.2021].

[7] D. Paszkiewicz, J. Dębski, Dostępność serwisów internetowych. Dobre praktyki w projektowaniu serwisów internetowych dostępnych dla osób z różnymi

rodzajami niepełnosprawności, Stowarzyszenie Przyjaciół Integracji, Warszawa, 2013, https://www.power.gov.pl/media/13588/Dostepnosc-serwisow-internetowych-Dominik-Paszkiewicz-Jakub-Debski.pdf, [14.03.2021].

[8] A. Dejnaka, Internet bez barier - accessibility oraz usability a potrzeby osób niepełnosprawnych. Niepełnosprawność - zagadnienia, problemy, rozwiązania, nr II/2012(3) (2012) 37-51, https://www.pfron.org.pl/fileadmin/files/0/171_03_Agnieszka_Dejnaka.pdf, [14.03.2021].

[9] D. Zdonek, S. Spałek, Metody oceny dostępności stron internetowych i problemy związane z ich wiarygodnością, Zeszyty Naukowe. Organizacja i Zarządzanie, Politechnika Śląska, 2013, http://www.woiz.polsl.pl/znwoiz/z64/p/Zdonek_D_SpalekS_Metody_dost%eapno%9cci_korekta_do_druku.pdf, [14.03.2021].

[10] M. Proskura, S. Podkościelna, G. Kozieł, An Examination of Selected Websites Availability For People With Various Types of Disabilities, Journal of Computer Sciences Institute, 17 (2020) 345-350, https://doi.org/10.35784/jcsi.2167, [14.03.2021].

[11] N. E. Youngblood, J. MacKiewicz, A usability analysis of municipal government website home pages in Alabama, *Government Information Quarterly*, 29(4) (2012) 582-588, https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6087239&tag=1, [14.03.2021].

[12] I. G. B. N. E. Darmaputra, S. S. Wijaya, M. A. Ayu, Evaluating the accessibility of provinces' e-government websites in Indonesia, 2017 5th International Conference on Cyber and IT Service Management (CITSM), Denpasar, Indonesia, (2017) 1-6, https://ieeexplore.ieee.org/document/8089322, [20.03.2021].

[13] P. T. Jaeger, The endless wire: E-government as global phenomenon, Government Information Quarterly, 20(4), (2003) 323-331, http://dx.doi.org/10.1016/j.giq.2003.08.003, [20.03.2021].

[14] Y. Ichsani, Y. Nurhadryani, F. Ardiansyah, Framework Development to Measure Usability and Accessibility on the e-Government Websites of Indonesia Provinces, 2012, http://repository.ipb.ac.id/handle/123456789/59445, [20.03.2021].

[15] B. Gohin, V. Vinod, A study on web accessibility in perspective of evaluation tools, International Review on Computers and Software, 8(11) (2013) 2648-2654.

[16] K.H. LEE, A study on non-blind algorithm of subarray signal processing for desired signal estimation, International Journal of Control and Automation, 6 (2013) 373-380, http://article.nadiapub.com/IJCA/vol6_no2/36.pdf, [20.03.2021].

[17] W. Stasiak, M. Dzieńkowski, Accessibility assessment of selected university websites, Journal of Computer Sciences Institute, 18, (2021) [w druku].

[18] J. Martins, R. Gonçalves, F. Branco, A full scope Web accessibility evaluation procedure proposal based on Iberian eHealth Accessibility Compliance, Computers in Human Behavior, 73 (2016) 676-684, https://www.sciencedirect.com/science/article/pii/S0747563216308317, [20.03.2021].

[19] Urząd Statystyczny w Lublinie, Województwo Lubelskie. Podregiony. Powiaty. Gminy 2019, https://lublin.stat.gov.pl/publikacje-i-foldery/roczniki-statystyczne/wojewodztwo-lubelskie-podregiony-powiaty-gminy-2019,1,16.html, [11.08.2020].

[20] Utilitia, https://validator.utilitia.pl/analyses/new, [11.08.2020].

[21] Tingun Page Checker, http://checkers.eiii.eu/, [11.08.2020].

[22] Google PageSpeed Insights, https://developers.google.com/speed/pagespeed/insights/?hl=pl, [11.08.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 121-125
Received: 29 March 2021
Accepted: 04 April 2021

REST API performance comparison of web applications based on JavaScript programming frameworks

Porównanie wydajności aplikacji internetowych REST API opartych na szkieletach programistycznych JavaScript

Marcin Grudniak* , Mariusz Dzieńkowski

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

Abstract

The aim of the work was to compare two technologies for creating server applications based on the JavaScript programming language. For the purposes of the research, two test applications were created. The first one was built on the basis of the Express programming framework and the second one on the basis of the Hapi framework. The client part of both applications was prepared using the React library. The client and server parts communicated with each other by means of REST API – the universal HTTP interface. The client application sent requests to the server application which then performed basic operations on the MongoDB basis and returned the result. As part of the work, an experiment consisting of four scenarios was developed. In each scenario, a different type of data was taken into consideration: a string of characters, an array, an object and an array of objects. The research focused on the efficiency aspect – measuring the response time of requests during GET, POST, PUT and DELETE operations. The tests were performed on two computers and the measurements were made in two ways: using a single code embedded in test applications and using the Postman tool. The obtained results, after averaging and analyzing them allowed for the conclusion that the Express framework proved to be more efficient than Hapi due to the shorter response time of requests. Only in the scenario where operations with large datasets were performed was the response time of requests at a similar level.

Keywords: Express; Hapi; performance analysis; response time of requests

Streszczenie

Celem pracy było porównanie dwóch technologii do tworzenia aplikacji serwerowych opartych na języku programowania JavaScript. Na potrzeby badań utworzono dwie aplikacje testowe: pierwszą zbudowano na podstawie szkieletu programistycznego Express, a druga została wykonana na bazie szkieletu Hapi. Część kliencką obu aplikacji przygotowano za pomocą biblioteki React. Część kliencka i serwerowa komunikowały się ze sobą za pośrednictwem REST API - uniwersalnego interfejsu HTTP. Aplikacja kliencka wysyłała żądania do aplikacji serwerowej, która następnie wykonywała podstawowe operacje na bazie MongoDB i zwracała rezultat. W ramach pracy opracowano eksperyment składający się z czterech scenariuszy. W każdym scenariuszu operowano na innym typie danych: łańcuchu znaków, tablicy, obiekcie oraz tablicy obiektów. W badaniach skoncentrowano się na aspekcie wydajnościowym - pomiarze czasów obsługi żądań podczas operacji GET, POST, PUT i DELETE. Badania przeprowadzono na dwóch komputerach, a pomiary wykonano dwoma sposobami: za pomocą prostego kodu wbudowanego w aplikacje testowe oraz za pomocą narzędzia Postman. Uzyskane wyniki, po ich uśrednieniu i przeanalizowaniu pozwoliły na sformułowanie wniosku, że szkielet Express okazał się wydajniejszy niż Hapi, ze względu na krótsze czasy obsługi żądań. Tylko w scenariuszu, w którym wykonywano operacje na dużych zbiorach danych, czasy obsługi żądań były na podobnym poziomie.

Słowa kluczowe: Express; Hapi; analiza wydajności; czas obsługi żądań

*Corresponding author

Email address: marcin.grudniak@pollub.edu.pl (M. Grudniak)

1. Wstęp

Współcześnie do budowy aplikacji internetowych najczęściej wykorzystuje się szkielety programistyczne, które definiują strukturę aplikacji i dostarczają narzędzi do realizacji określonych zadań. Upraszczają one także proces programowania, wykorzystując dobre praktyki wypracowane przez społeczność programistów zaangażowaną w ich rozwój. Wybór optymalnej platformy programistycznej, przed którym stoją deweloperzy oprogramowania, dostosowanej do wymagań konkretnego przypadku nie jest łatwym zadaniem. Sytuację komplikuje nie tylko duży wybór szkieletów programistycznych dla danego języka oraz ich szybka ewolucja, ale także ciągłe powstawanie nowych rozwiązań lepiej realizujących podobne zadania. Podjęcie właściwych decyzji, co do wyboru technologii, może zaowocować wytworzeniem bardziej interaktywnych, przyjaznych oraz łatwych w obsłudze aplikacji. Zastosowanie odpowiedniego szkieletu może mieć wpływ na skrócenie czasu powstawania finalnego produktu, zmniejszenie nakładu pracy pracowników oraz obniżenie kosztów przedsięwzięcia informatycznego. Niemniej ważne kwestie związane z wyborem frameworka to także zapewnienie wysokiej jakość kodu oraz odpowiedniego poziomu bezpieczeństwa tworzonego oprogramowania [1]. W przypadku wytwarzania aplikacji internetowych,

z których będzie korzystało jednocześnie wielu użytkowników, również niezwykle ważnym zagadnieniem, które należy uwzględnić dobierając platformę programistyczną, jest odpowiednio wysoka wydajność i wynikająca z niej szybkość transmisji danych.

2. Prezentacja porównywanych technologii

Node jest środowiskiem uruchomieniowym, działającym poza przeglądarką i współpracującym z systemem operacyjnym. Ze względu na swoją szybkość i wykorzystanie języka JavaScript, jest obecnie jedną z wiodących platform do tworzenia aplikacji internetowych po stronie serwera. Aplikacje pracujące na tej platformie mają dostęp przez API do systemu operacyjnego, do jego systemu plików, bibliotek systemowych, uruchomionych procesów, w tym serwerów HTTP [2]. Jądro Node jest mocno ograniczone i tym samym wywołuje potrzebę zastosowania frameworka, niezbędnego do zadań porządkowych oraz zwiększenia produktywności i jakości aplikacji [3].

2.1. Szkielet programistyczny Express

Najpopularniejszym szkieletem dla aplikacji serwerowych operujących na platformie Node jest Express. Potwierdzają to różne serwisy publikujące aktualne rankingi [4-6]. Według raportu opublikowanego w serwisie stackoverflow w lutym 2020 roku szkielet ten był na pierwszym miejscu wśród frameworków serwerowych na platformę Node. Natomiast biorąc pod uwagę inne środowiska wytwarzania aplikacji webowych zajął on piątą pozycję, plasując się za jQuery, React, Angular oraz ASP.NET. Na Express wskazało 21,2% wszystkich respondentów, a wśród nich 29,9% profesjonalnych deweloperów oprogramowania [4]. W serwisie GitHub, biorąc pod uwagę szkielety środowiska Node, Express także lokuje się na pierwszej pozycji, mając przyznanych 52,5 tys. gwiazdek (stan na dzień 22.03.2021) [5].

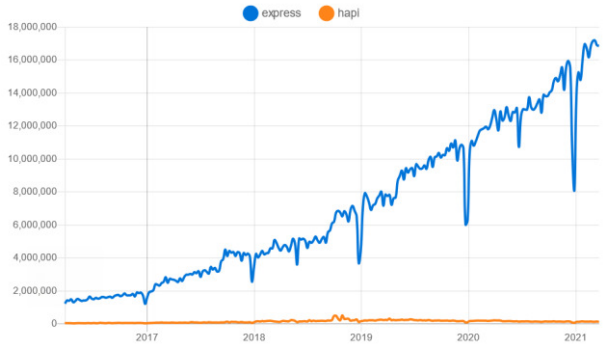
Wysokie notowania w rankingach wynikają z tego, że Express ma wiele zalet. Jest on wydajny, solidny, elastyczny, minimalny, dobrze przetestowany i posiada dużą społeczność. Poza tym Express jest dojrzałym, ciągle aktualizowanym, dobrze udokumentowanym i łatwym w użyciu frameworkiem. Został on zaprojektowany do tworzenia jednostronicowych, wielostronicowych i hybrydowych aplikacji, zapewniając solidny zestaw funkcji dla aplikacji internetowych i mobilnych. Bardzo dobrze nadaje się zarówno do dużych jak i małych projektów [7]. Na bazie tego szkieletu powstało wiele innych szkieletów m.in. KeystoneJS, Blueprint, NestJS i Locomotive [8]. Choć Express nie ma gotowego do użycia mechanizmu mapowania obiektowo-relacyjnego, to ma swobodę w łączeniu się z innymi technologiami, co ułatwia jego błyskawiczna konfiguracja. Express skutecznie rozwija backendowe części stosów MEAN oraz MERN z frontendowym szkieletem Angular lub biblioteką React oraz bazą danych MongoDB.

2.2. Szkielet programistyczny Hapi

Najważniejszą kwestią, która była brana pod uwagę podczas tworzenia frameworka Hapi, było stworzenie narzędzia zapewniającego duże wsparcie programistyczne dla dużych zespołów zarządzających wieloma zadaniami. Niemniej ważnym aspektem było dostarczenie bogatych i niezawodnych funkcji bezpieczeństwa. Z tego względu szkielet Hapi jest wykorzystywany przez programistów do tworzenia serwerów proxy, interfejsów REST API i innych aplikacji, dla których ważne są niezawodność i bezpieczeństwo. Duża liczba wbudowanych w ten szkielet wtyczek eliminuje potrzebę stosowania niepewnego oprogramowania pośredniego (middleware). Hapi powinien być pierwszym wyborem dla deweloperów wytwarzających bezpieczne, skalowalne aplikacje do obsługi mediów społecznościowych w czasie rzeczywistym [9]. Przykładem dużej skali serwisów opartych na tej technologii są PayPal, Disney i Macy's.

Hapi, będąc potężną i wysokowydajną platformą, pracującą w środowisku rozproszonym, charakteryzuje się lekkością i bogactwem funkcji. Zawiera m.in. wbudowaną funkcję buforowania, która ułatwia uruchamianie aplikacji czasu rzeczywistego wymagających dużej ilości danych pochodzących z wielu urządzeń.

Każdy szkielet programistyczny ma swoje mocne i słabe strony. Hapi jest dla Node bardziej abstrakcyjnym frameworkiem niż Express, którego kod wygląda bardziej jak natywny kod Node.js. Zaletami obu szkieletów są rozszerzalność i elastyczność. Pod względem popularności, Express jest zdecydowanie częściej używany niż Hapi i, jak pokazuje rysunek 1, jego popularność ciągle wzrasta. Poniższy wykres przedstawia liczbę pobrań pakietów w ciągu jednego tygodnia (oś Y) za pomocą menadżera npm za okres ostatnich pięciu lat (oś X).



Rysunek 1: Popularność szkieletów programistycznych Express i Hapi wyrażona liczbą pobrań pakietów na tydzień w okresie ostatnich pięciu lat [10].

3. Cel i zakres pracy

W pracy skoncentrowano się na kwestiach wydajnościowych dwóch szkieletów aplikacyjnych opartych na platformie Node. Wybór padł na Express - najpopularniejszy szkielet tej platformy oraz na Hapi – szkielet dedykowany dla dużych, jeśli chodzi o skalę, przedsięwzięć informatycznych.

W związku z tym, celem artykułu jest analiza porównawcza dwóch szkieletów programistycznych Express i Hapi, opartych na architekturze REST, umożliwiającej wygodną wymianę danych pomiędzy częścią serwerową i kliencką aplikacji. Kryterium porównawczym, które zostało wzięte pod uwagę podczas analiz, była wydajność czasowa. Motywem do podjęcia tego tematu była z jednej strony mała liczba artykułów opublikowanych w internecie dotyczących porównania tych dwóch szkieletów programistycznych, a z drugiej strony chęć porównania uzyskanych w ramach tej pracy wyników z rezultatami prezentowanymi przez innych autorów [11-13] .

4. Testowe aplikacje serwerowe

Główną różnicą między obydwoma szkieletami jest ich sposób obsługi żądań. W przypadku Hapi (Listing 1) wszystkie informacje przekazuje się w postaci obiektu, na który składają się metoda, ścieżka i obsługa [14]. W przypadku szkieletu Express (Listing 2), hierarchia jest następująca: najpierw, jako parametr funkcji, podaje się ścieżkę, a następnie metodę HTTP z dwoma parametrami, spośród których drugi zwraca odpowiedź [15].

Listing 1: Fragment kodu aplikacji odpowiedzialny za obsługę żądania dodania nowego dokumentu zrealizowanego w Hapi

```
server. route({
    method: "POST",
    path: "/add/{length}/{size}",
    handler: (req, res) => {
        return obj(req)
            .save()
            .then((err, res) => {
                if(err){
                    return err;
                }
                return res;
            });
    },
});
```

Listing 2: Obsługa żądania dodawania nowego dokumentu w Express

```
router. route("/add/:length/:size").post((req, res) => {
    obj(req)
        .save()
        .then((post) => {
            res.json(post);
        })
        .catch((err) => {
            console.error(err);
            res.status(500).json({ error: err.code });
        });
});
```

5. Metoda badań

5.1. Środowisko badawcze

W celu większej wiarygodności wyników, badania zostały przeprowadzone na dwóch stacjach badawczych, różniących się konfiguracją (Tabela 3).

Tabela 1: Konfiguracja środowisk badawczych

	Komputer 1	Komputer 2
System operacyjny	Windows 10	Ubuntu 18.04
Procesor	Intel Xeon X5690 3.47GHz	Intel i3-4005U
Pamięć RAM	12 GB	4 GB
Dysk	SSD	SSD
Łącze	LAN	WiFi 2.4GHz

Na obu komputerach zostały zainstalowane aplikacje testowe, realizujące obsługujące żądania wykonujące cztery podstawowe operacje na bazie danych, za pośrednictwem metod REST API [16] odpowiedzialnych za tworzenie, odczytywanie, aktualizowanie i usuwanie danych. Żądania te inicjowały działania na dokumentach przechowywanych w bazie MongoDB mających postać rekordów, które zawierają swego rodzaju kontener o strukturze:

- tekstowego typu danych,
- tablicy,
- obiektu,
- tablicy obiektów.

Każdy kontener wypełniany był odpowiednią serią danych: losowym ciągiem znaków o zadanej długości, ustaloną ilością komórek w tablicy, bądź określoną liczbą atrybutów w obiekcie.

W badaniach, oprócz aplikacji testowych, użyto popularnego narzędzia Postman [17], które może być wykorzystywane do prostego wysyłania żądań do API, zapisywania żądań w celu ich późniejszego użycia oraz testowania API.

Do zbierania wyników – czasów obsługi żądań, wykorzystano dwie metody. Takie podejście miało sprawdzić czy wykonywane pomiary dawały takie same lub porównywalne wyniki. Jedną z metod było użycie modułu wewnątrzserwerowego, który mierzy czasy od wysłania żądania do uzyskania odpowiedzi (cykl życia żądania). Druga metoda wykorzystywała aplikację zewnętrzną, która posiadała wbudowany skrypt do rejestracji czasów wykonywanych żądań. W obu przypadkach żądania były wysyłane w sposób iteracyjny, jedno po drugim.

5.2. Scenariusze badawcze

Opracowano eksperyment, w ramach którego testowano podstawowe operacje umożliwiające zarządzanie danymi w bazie danych. W tym celu opracowano cztery scenariusze badawcze. Każdy z nich dotyczył operacji pobierania, wysyłania, modyfikowania i usuwania innego zestawu danych, różniących się strukturą i rozmiarem. W scenariuszu pierwszym operowano na rekordzie, którego elementem był tekstowy typ danych, w postaci ciągu o długości 1000 znaków. W drugim scenariuszu realizowano działania na tablicy zawierającej 100 elementów, którymi były losowe ciągi 100 znaków. Trzeci scenariusz dotyczył obiektu składającego się ze 100 atrybutów, z których każdy zawierał wartość w postaci losowego ciągu o długości 100 znaków. W ostatnim, czwartym scenariuszu operowano na 50 elementowej tablicy składającej się z obiektów o 50

atrybutach, wypełnionych losowymi ciągami zawierającymi po 50 znaków.

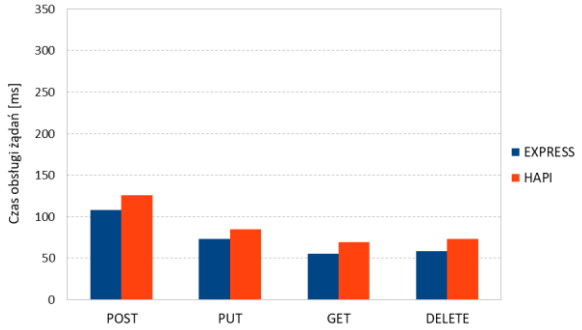
Operację REST API modyfikowania rekordu w bazie danych realizowano z takimi samymi parametrami jak przy tworzeniu kolejnego, nowego dokumentu. Natomiast pobieranie i usuwanie odbywało się po sparametryzowaniu żądania przez podawanie numeru identyfikującego rekord w bazie danych.

Każde żądanie wysyłano zarówno z aplikacji klienckiej jak i z narzędzia Postman po 100 razy. Wyniki eksperymentu były zliczane dwoma sposobami za pomocą modułu natywnego wkomponowanego w kod obsługi cyklu żądania oraz przy użyciu aplikacji Postman. W pojedynczym scenariuszu wykonano cztery typy żądań (pobieranie, aktualizowanie dodawanie, usuwanie), co dawało w sumie 800 wyników. Uwzględniając wszystkie cztery scenariusze – dla różnych typów danych, otrzymano 3200 wyników. Należy jeszcze pamiętać, że testowano dwa szkielety programistyczne Express i Hapi, a badania realizowano równolegle na dwóch komputerach. W ten sposób po przeprowadzeniu eksperymentu uzyskano w sumie 12800 wyników.

6. Wyniki badań

6.1. Scenariusz 1 – pomiar czasu obsługi żądania podczas wykonywania operacji na tekstowym typie danych

Operacje na małych porcjach danych w postaci łańcucha znaków były wykonywane w stosunkowo krótkim czasie. Najdłużej trwała obsługa żądania POST, które miało na celu przesyłanie danych do serwera. Na Rysunku 2 przedstawiono wyniki, które są obliczonymi wartościami średnimi.

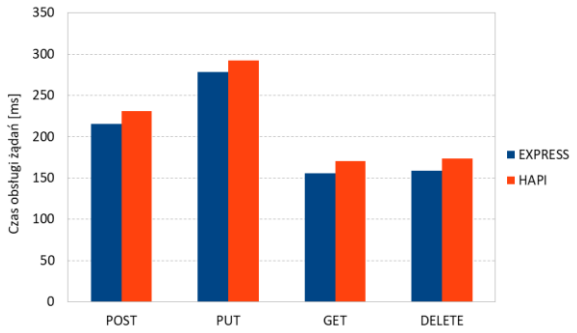


Rysunek 2: Średnie czasy wykonywania cyklu żądań realizujących poszczególne operacje na danych w postaci łańcucha znaków.

Na podstawie uzyskanych wyników można stwierdzić, że dla małych porcji danych, czas obsługi żądań był krótszy, w przypadku aplikacji testowej opracowanej na bazie szkieletu Express, bez względu na typ realizowanej operacji na danych. Czasy realizacji żądań dla metod POST, PUT, GET i DELETE były krótsze o odpowiednio 14%, 14%, 20% oraz 19%.

6.2. Scenariusz 2 – pomiar czasu obsługi żądania podczas wykonywania operacji na tablicy

Operacje na tablicach przechowujących w swych komórkach łańcuchy zawierające po 100 znaków były wykonywane około dwa razy dłużej niż działania na danych w pierwszym scenariuszu. Najbardziej czasochłonna była obsługa żądania PUT, polegająca na aktualizacji danych przechowywanych w określonych rekordach bazy danych. Rysunek 3 prezentuje średnie czasy otrzymane po przeprowadzeniu eksperymentu.

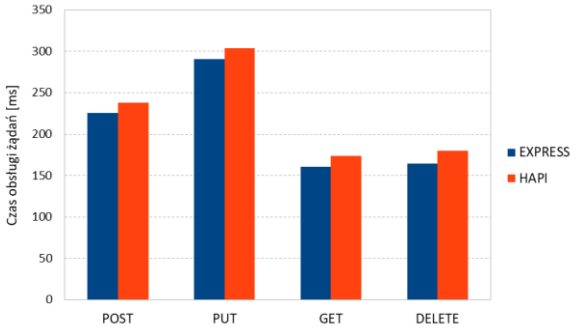


Rysunek 3: Średnie czasy wykonywania cyklu żądań realizujących poszczególne operacje na danych w postaci tablicy składającej się z łańcuchów znaków.

Również w tym przypadku aplikacja testowa zbudowana przy pomocy szkieletu Express pracowała wydajniej, choć jej przewaga, tzn. różnice czasowe były mniejsze niż w przypadku scenariusza 1 i wahały się w zakresie 5% - 9%. W związku z tym, można wysnuć wniosek, że operacje na większych porcjach danych zmniejszały różnice między porównywanymi frameworkami języka JavaScript.

6.3. Scenariusz 3 – pomiar czasu obsługi żądania podczas wykonywania operacji na obiekcie

Działania na średniej wielkości kontenerach na dane, którymi w tym przypadku były duże obiekty, składające się ze 100 pól tekstowych przechowujących stuznakowy tekst, wymagały podobnych jak w scenariuszu 3 czasów obsługi żądania. Również w tym scenariuszu najdłużej trwała obsługa żądania modyfikacji rekordów. Na Rysunku 4 zostały zaprezentowane wyniki - średnie czasy trwania cyklu żądania podczas których realizowano odpowiednie operacje na danych w bazie.

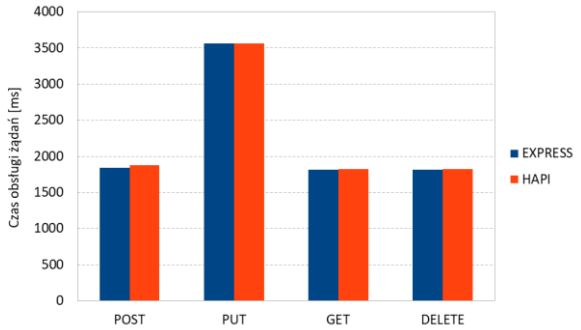


Rysunek 4: Średnie czasy wykonywania cyklu żądań realizujących poszczególne operacje na danych w formie rozbudowanego obiektu.

Także w tej części eksperymentu okazało się, że szkielet Express jest wydajniejszy niż Hapi, choć uzyskiwał on tylko nieznacznie lepsze czasy obsługi żądań (w granicach 1% - 5%).

6.4. Scenariusz 4 – pomiar czasu obsługi żądania podczas wykonywania operacji na tablicy obiektów

W ostatniej fazie badań dokonano pomiarów czasu trwania cyklu żądania podczas realizacji operacji na rekordach bazy danych zawierających duże partie danych, którymi były 50 elementowe tablice obiektów składające się z 50 atrybutów przechowujących łańcuchy znakowe o długości 50 znaków.



Rysunek 5: Średnie czasy wykonywania cyklu żądań realizujących poszczególne operacje na danych w postaci tablicy obiektów.

Wyniki zobrazowane na Rysunku 5 pokazują, że podczas wszystkich operacji na dużych zbiorach danych czasy obsługi żądań aplikacji wytworzonej za pomocą szkieletu Express oraz aplikacji zbudowanej na podstawie szkieletu Hapi zrównały się. Niewątpliwie czasy te były bardzo długie i w przypadku aktualizacji rekordów osiągały wartości powyżej 3500 ms. Dla pozostałych trzech operacji czasy te mieściły się w zakresie 1700 – 1900 ms.

7. Wnioski

W pracy skoncentrowano się na pomiarach i analizie czasów obsługi żądań, które miały za zadanie realizację czterech podstawowych operacji REST na nierelacyjnej bazie danych MongoDB.

Biorąc pod uwagę całość badań, na które złożyły się cztery scenariusze, aplikacja testowa zbudowana na podstawie szkieletu Express miała nieznacznie lepszą wydajność niż aplikacja oparta na frameworku Hapi. Tylko w jednym przypadku, w scenariuszu nr 4, wydajności obu aplikacji testowych były niemal identyczne. Dotyczyło to sytuacji, w której żądania operowały na rekordach zawierających bardzo rozbudowane i duże objętościowo zbiory danych, którymi były w tym przypadku tablice dużych obiektów. Spośród czterech operacji CRUD, aktualizacja danych trwała najdłużej, a pobieranie najkrócej. Wynikało to z tego, że podczas modyfikacji należało najpierw zlokalizować dokument w bazie, odczytać jego zawartość oraz na koniec nadpisać. W przypadku pobierania dokumentu schemat postępowania był prostszy i polegał jedynie na identyfikacji i zwróceniu w odpowiedzi danych.

Zrealizowane w pracy badania mają swoje ograniczenia, ponieważ skupiono się w nich wyłącznie na aspekcie wydajnościowym, a pozostałe kwestie pominięto. Jednak mimo tego, wykonana praca i zaprezentowane wyniki mogą być wykorzystane praktycznie przez programistów tworzących aplikacje webowe.

Literatura

[1] M. Pilar Salas-Zarate, G. Alor-Hernandez, R. Valencia-Garcia, L. Rodriguez-Mazahua, A. Rodriguez-Gonzalez, J. L. Lopez Cuadro, Analyzing best practices on Web development frameworks: The lift approach, Science of Computer Programming, Science of Computer Programming, 102, (2015) 1-19, https://doi.org/10.1016/j.scico.2014.12.004, [16.03.2021].

[2] Mozilla Developer Network, Wprowadzenie do Express/Node, https://developer.mozilla.org/pl/docs/Learn/Server-side/Express_Nodejs/Introduction, [16.03.2021].

[3] Offshore Web Developer, Hapi vs Express: Which NodeJS Frameworks is the Better?, http://www.offshorewebdeveloper.com/blog/hapi-vs-express/, [16.03.2021].

[4] Stackoverflow.com, https://insights.stackoverflow.com/survey/2020#technology, [27.03.2021].

[5] Github.com, https://github.com/showcases/web-application-frameworks, [27.03.2021].

[6] Stateofjs, https://2020.stateofjs.com/en-US/technologies/back-end-frameworks/, [27.03.2021].

[7] RapidAPI Blog, The Best NodeJS Frameworks for 2021, https://rapidapi.com/blog/best-nodejs-frameworks/, [25.03.2021].

[8] Frameworks built on Express, https://expressjs.com/, https://expressjs.com/en/resources/frameworks.html, [25.03.2021].

[9] S. Martin, Top 10 Node.js Frameworks For Web App Development in 2021, https://javascript.plainenglish.io/top-10-node-js-frameworks-for-web-app-development-in-2020-21-38e3ea2a57e5, [25.03.2021].

[10] Npm trends, https://www.npmtrends.com/express-vs-hapi, [27.03.2021].

[11] D. Swerski, Hapi vs. Express in 2019: Node.js framework comparison, https://raygun.com/blog/hapi-vs-express/, [26.10.2020].

[12] Raygun, Node.js performance vs Hapi, Express, Restify, Koa & More, https://raygun.com/blog/nodejs-vs-hapi-express-restify-koa/, [26.10.2020].

[13] B. Miłosierny, M. Dzieńkowski, The comparative analysis of web application frameworks in the Node.js ecosystem, Journal of Computer Science Institute, 18, (2021) 42-48, https://doi.org/10.35784/jcsi.2423, [30.03.2021].

[14] J. Brett, Getting Started with hapi.js, Packt Publishing, 2016.

[15] E. M. Hahn, Express in action. Writing, building, and testing Node.js applications, Manning Publications Co., 2016.

[16] M. Massé, REST API Design Rulebook, O'Reilly, 2012.

[17] Postman, https://www.postman.com/, [27.03.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 126–131
Received: 19 April 2021
Accepted: 26 April 2021

The comparative analysis of modern ETL tools

Analiza porównawcza współczesnych narzędzi ETL

Ivan Falchuk, Vitalii Mayuk, Piotr Muryjas*

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

Abstract

Each data warehouse requires loading properly processed transactional data. The process that performs this task is known as extract-transform-load (ETL). The efficiency of its implementation affects how quickly the user will have the access to the current analytical data. The paper presents the results of research efficiency of ETL performance of its stage with the use of Azure Synapse (AS) and Azure Data Factory (ADF). The research included selection, sorting and aggregating data, joining tables, and loading data into target tables. To evaluate the efficiency of these operations, the criterion of their execution time has been used. The obtained results indicate that the ADF tool provides a much higher time efficiency of loading transactional data into the data warehouse comparing to AS.

Keywords: Azure Synapse; Azure Data Factory; ETL tools

Streszczenie

Każda hurtownia danych wymaga ładowania odpowiednio przetworzonych danych transakcyjnych. Procesy realizujące to zadanie określane są jako ekstrakcja-transformacja-ładowanie (ETL). Od efektywności ich wykonania zależy jak szybko użytkownik będzie miał dostęp do bieżących danych analitycznych. W artykule przedstawiono istotę procesu ETL oraz wyniki badań efektywności realizacji jego etapów z użyciem Azure Synapse (AS) oraz Azure Data Factory (ADF). Badania obejmowały selekcję, sortowanie i agregację danych, złączenie tabel oraz zapis danych do tabel docelowych. Do oceny efektywności tych operacji zastosowano kryterium czasu ich wykonania. Uzyskane wyniki wskazują, iż narzędzie ADF zapewnia znacznie wyższą efektywność czasową ładowania danych transakcyjnych do hurtowni danych w porównaniu do AS.

Słowa kluczowe: Azure Synapse; Azure Data Factory; ETL tools

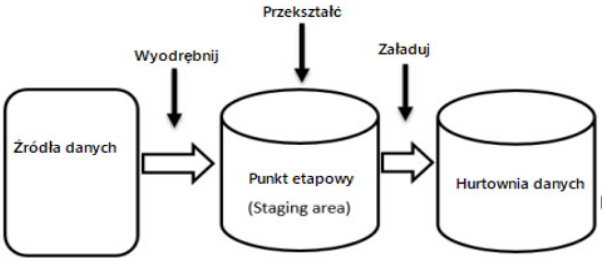
*Corresponding author

Email address: p.muryjas@pollub.pl (P. Muryjas)

1. Wstęp

Każda współczesna organizacja staje dziś przed wyzwaniem przechowywania dużej ilości danych o zróżnicowanej jakości oraz ich efektywnej analizy. Transakcyjna postać tych danych, pochodzących z systemów ERP, nie jest jednak właściwa dla osób, które podejmują decyzje i muszą posługiwać się danymi o pewnym stopniu agregacji. Z tego względu przedsiębiorstwa coraz częściej korzystają z systemów klasy business intelligence (BI), które znacznie zwiększają efektywność analizy danych oraz zaspokajają potrzeby analityczne osób decyzyjnych. Głównym źródłem danych dla tego rodzaju systemów jest hurtownia danych. Jest to miejsce składowania danych analitycznych, które są używane do dokonywania bardziej trafnych wyborów w procesach decyzyjnych [1].

Podejmowanie decyzji wymaga posiadania informacji o aktualnej sytuacji w organizacji. Dlatego też konieczne jest, by dane w hurtowniach były okresowo uzupełniane i aktualizowane. Zadanie to wykonywane jest w ramach działań określanych jako ekstrakcja, transformacja i ładowanie danych (ETL – Extract, Transform, Load), które opisują sposób pobierania danych ze źródeł, logikę ich transformacji zgodną z wymaganiami biznesowymi oraz miejsce ich docelowego składowania w hurtowni danych. Ideę procesu ETL przedstawiono na Rysunku 1.



Rysunek 1: Idea procesu ETL.

Etap ekstrakcji danych polega na ich wyodrębnieniu ze źródeł i zapisaniu w miejscu określanym jako obszar pośredni (staging area). W tym etapie dane nie zmieniają jeszcze swojej pierwotnej, transakcyjnej postaci. W procesie transformacji, który jest realizowany w obszarze pośrednim, dane są oczyszczane, integrowane, grupowane, a także konwertowane do wymaganej postaci, zgodnej ze strukturą przechowywania danych w hurtowni. Na etapie ładowania dane są pobierane z obszaru pośredniego i bezpośrednio umieszczane w tabelach hurtowni danych. Należy zauważyć, że w procesie ETL nie wszystkie dane transakcyjne są brane pod uwagę, ale tylko te, które są nowe lub zostały zmienione [2].

W niniejszej publikacji dokonano analizy wydajności dwóch współczesnych narzędzi ETL firmy Micro-

soft, tj. Azure Data Factory i Azure Synapse. Wykorzystując różne scenariusze procesów ETL i ten sam źródłowy system transakcyjny, przeprowadzono badania, których rezultaty umożliwiły wskazanie narzędzia zapewniającego wyższą efektywność realizacji tych procesów.

2. Charakterystyka narzędzi ETL firmy Microsoft

Azure Data Factory (ADF) to usługa ETL w chmurze platformy Microsoft Azure umożliwiająca skalowalną w poziomie integrację danych i ich transformację bez użycia odrębnego, dedykowanego serwera. Usługa ta dostępna jest poprzez interfejs użytkownika zapewniający intuicyjne tworzenie, monitorowanie i zarządzanie procesem ETL w jednym panelu [3]. Narzędzie ADF zapewnia migrację danych między wieloma lokalnymi i chmurowymi źródłami danych. Lista obsługiwanych platform jest rozbudowana i obejmuje zarówno rozwiązania firmy Microsoft jak i innych dostawców. Jest to zaawansowane narzędzie zapewniające pełną elastyczność przenoszenia ustrukturyzowanych i nieustrukturyzowanych zestawów danych, w tym RDBMS, XML, JSON i różnych magazynów danych typu NoSQL. Jego podstawową zaletą jest duża elastyczność wynikająca z możliwości wykorzystania języka U-SQL lub HiveQL [4-5].

Azure Synapse (AS) jest kolejną usługą wykorzystywaną zarówno do realizacji procesów ETL jak i do analizy danych. Z jej pomocą możliwa jest integracja i transformacja danych transakcyjnych jak również danych typu big data. Narzędzie AS zapewnia wykonywanie zapytań do różnych baz danych bez konieczności użycia zasobów serwerowych. W usłudze tej rozproszone zasoby danych źródłowych są łączone w ujednolicone środowisko umożliwiające pozyskiwanie, eksplorowanie, przygotowywanie i udostępnianie danych oraz zarządzanie nimi na potrzeby analizy biznesowej oraz uczenia maszynowego.

Azure Synapse zapewnia wysoką elastyczność łączenia danych nierelacyjnych oraz relacyjnych. Narzędzie umożliwia łatwy podgląd danych za pomocą wysyłania zapytań do bazy danych Azure SQL. Tworząc przepływy danych przy pomocy predefiniowanych bloków zadaniowych, można zaimplementować zaawansowane scenariusze transformacji i analizy danych bez konieczności pisania kodu źródłowego [6].

Azure Synapse posiada wbudowaną bezserwerową pulę SQL, czyli logiczną przestrzeń, w której udostępniane są w elastyczny sposób różne zasoby umożliwiające efektywną realizację procesów ETL. Każdy obszar roboczy posiada rozbudowaną pulę SQL bez możliwości usunięcia takiej puli. Bezserwerowe pule SQL zapewniają możliwość użycia języka SQL bez konieczności rezerwowania określonej wielkości zasobów pamięci. W przeciwieństwie do dedykowanych pul SQL, rozliczanie dla bezserwerowej puli SQL jest oparte na ilości danych przeskanowanych w celu uruchomienia zapytania, a nie pojemności przydzielonej do puli [7].

3. Metodyka badań

Porównanie wydajności badanych narzędzi ETL wymagało przeprowadzeniu operacji ekstrakcji, transformacji i ładowania danych. Dokonano pomiaru czasu wykonania podstawowych operacji ETL oraz dodatkowo czasu napełniania tabeli faktu w przykładowej hurtowni danych.

Na potrzeby badań, w każdym z narzędzi utworzony został zasób bezserwerowej bazy danych Azure SQL Database o pojemności 5 GB oraz wydajności 100 DTU (Data Transaction Unit). DTU jest to pakiet miary zasobów obliczeniowych (przydzielonych do bazy danych), operacji we/wy i magazynu [8]. W każdej tabeli, stanowiącej źródło danych dla procesów ETL, zostały utworzone klastrowane indeksy kolumnowe w celu zwiększenia wydajności zapytań SQL.

Jako szczegółowe kryteria oceny wydajności narzędzi w każdym scenariuszu badawczym przyjęto:

- czas uruchamiania klastra – czas potrzebny na uruchomienie izolowanego klastra Apache Spark, który jest używany podczas uruchamiania przepływów danych [9];
- czas ekstrakcji danych ze źródła – czas potrzebny na odczyt danych z tabeli źródłowej do pierwszego bloku zadaniowego w przepływie danych;
- czas transformacji danych – czas potrzebny do wykonania operacji przekształcenia oraz czyszczenia danych w przepływie;
- czas zapisywania danych w zasobie docelowym - czas potrzebny na załadowanie danych do tabeli docelowej (np. tabela wymiaru lub faktu w hurtowni danych).

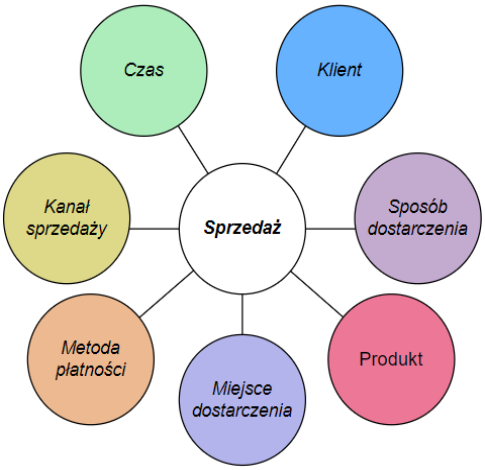
W każdym z narzędzi zostało utworzonych 9 przepływów danych opisujących wykonanie podstawowych operacji przekształcenia danych w procesie ETL. Wykaz tych operacji oraz ich szczegółowy opis biznesowy zostały przedstawione w Tabeli 1.

Tabela 1: Operacje ETL w scenariuszach badawczych

Lp.	Operacja	Opis
1	Selekcja wszystkich kolumn	Selekcja wszystkich kolumn z tabeli zawierającej dane o klientach.
2	Selekcja wybranych kolumn	Selekcja wybranych kolumn z tabeli zawierającej dane o klientach.
3	Sortowanie	Sortowanie danych w tabeli zawierającej dane o produktach.
4	Zliczanie	Zliczanie faktur wszystkich klientów.
5	Zliczanie unikalnych wartości	Zliczanie liczby sprzedanych produktów.
6	Sumowanie	Sumowanie wartości sprzedanych produktów.
7	Złączenie	Złączenie danych o nagłówkach i pozycjach faktur sprzedaży.
8	Złączenie + Sumowanie	Złączenie danych o produktach i pozycjach faktur oraz sumowanie wartości sprzedaży według grupy zapotrzebowania.

9	Filtrowanie	Wyszukiwanie faktur korygujących z podanym kodem przyczyny korekty.

Dodatkowo, w ramach przeprowadzonych badań, zbudowano 6 przepływów danych realizujących zadanie napełnienia tabel w hurtowni danych. Pięć przepływów danych służyło do napełnienia tabel wymiarów oraz jeden – do napełniania tabeli faktu opisującego sprzedaż. Schemat konceptualny wykorzystanej hurtowni danych został przedstawiony na Rysunku 2.



Rysunek 2: Schemat konceptualny hurtowni danych wykorzystanej w badaniach.

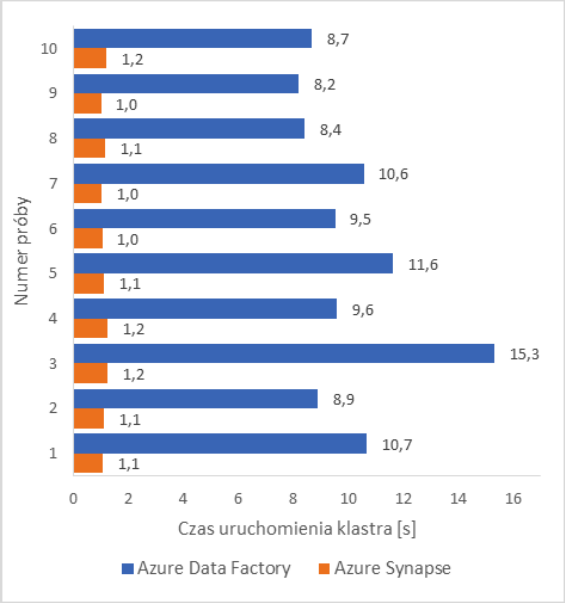
Każdy z przepływów danych, tj. 9 do wykonania podstawowych operacji ETL i 1 do napełniania tabeli faktu, został uruchomiony 10 razy. Wyniki przeprowadzonych badań zostały zapisane w plikach *csv*. Czas wykonania badanych operacji był mierzony przy pomocy wbudowanego w każde z narzędzi środowiska monitorowania, które podaje szczegółowe informacje o wykonanych przepływach danych [10].

4. Wyniki badań

Wyniki przeprowadzonych badan wydajnościowych zostaną przedstawione dla każdego kryterium czasowego wymienionego w rozdziale 3. Dodatkowo, na wstępie rozdziału zaprezentowano rezultaty badań dotyczących szybkości uruchomienia klastra Apache Spark, w którym zlokalizowane są dane źródłowe. Czas wykonania tej operacji ma istotny wpływ na całkowity czas realizacji procesu ETL. Dlatego też zasadne jest, aby określić jego wielkość dla poszczególnych narzędzi.

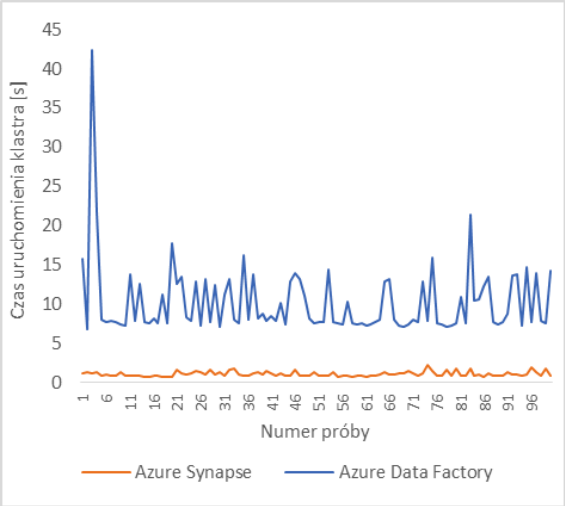
4.1. Uruchomienie klastra

Na Rysunku 3 przedstawiono wyniki badań dotyczących szybkości uruchomienia klastra Apache Spark przez ten sam proces ETL wykonywany 10-cio krotnie w każdym z badanych narzędzi. Można zauważyć, że czas uruchomienia klastra w narzędziu Azure Synapse jest średnio 9 razy krótszy niż w środowisku Azure Data Factory.



Rysunek 3: Czas uruchomienia klastra Apache Spark.

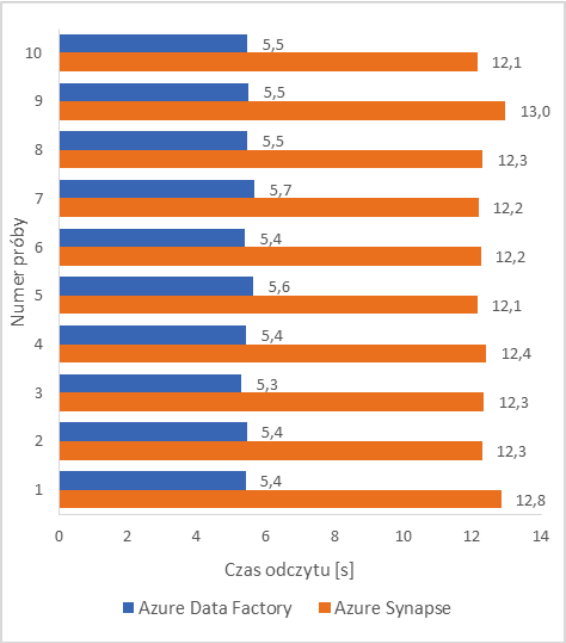
Ponadto, na podstawie 10-krotnego powtórzenia tego badania dla wszystkich 10-ciu scenariuszy można stwierdzić, że klaster używany w narzędziu Azure Synapse jest bardziej stabilny niż w narzędziu Azure Data Factory (Rysunek 4).



Rysunek 4: Czasy wielokrotnego uruchomienia klastra dla wszystkich scenariuszy badawczych.

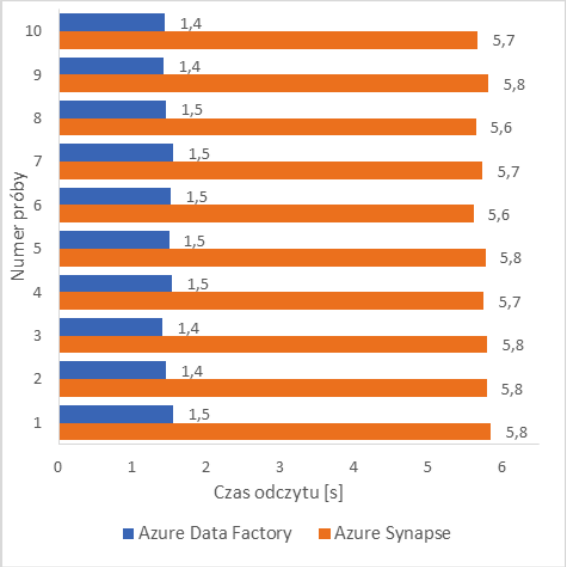
4.2. Ekstrakcja danych ze źródła

Kolejne badanie miało na celu identyfikację wpływu liczności kolumn w zapytaniu do tabeli źródłowej na długość czasu jego wykonania. Operacje odczytu wykonano z użyciem tabeli zawierającej 49 383 rekordy opisujące klientów. W pierwszej kolejności zmierzono czas odczytu danych ze wszystkich szesnastu kolumn tej tabeli (Rysunek 5), a następnie z pięciu wybranych (Rysunek 6).



Rysunek 5: Czasy odczytów danych ze wszystkich kolumn tabeli *Klienci*.

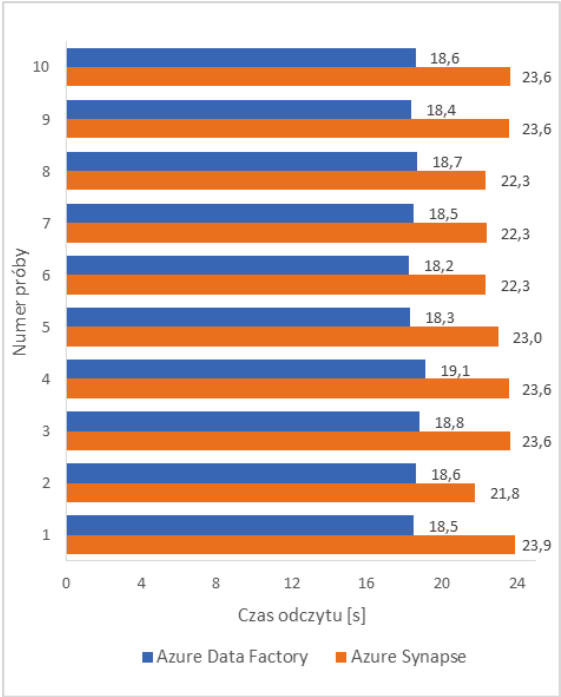
Na podstawie otrzymanych wyników badań można stwierdzić, że czas odczytu danych ze wszystkich kolumn tabeli źródłowej jest ponad 2 razy dłuższy w środowisku Azure Synapse niż w Azure Data Factory.



Rysunek 6: Czasy odczytów z wybranych 5 kolumn tabeli *Klienci*.

Natomiast zapytanie zawierające 5 wybranych kolumn tabeli źródłowej, przy tej samej liczbie rekordów, wykonuje się niemal 4 razy szybciej w narzędziu Azure Data Factory niż w Azure Synapse.

Na podstawie przeprowadzonych eksperymentów stwierdzono, że zwiększenie liczby odczytywanych rekordów i kolumn z tabeli źródłowej prowadzi do zmniejszenia różnicy w czasie wykonania zapytań w badanych środowiskach. W przypadku ekstrakcji danych z tabeli opisującej produkty, zawierającej 921 855 rekordy i 41 kolumn, zaobserwowano, iż zapytanie w środowisku Azure Data Factory wykonywało się już tylko 1,3 razy szybciej niż w narzędziu Azure Synapse.



Rysunek 7: Czasy odczytu wszystkich kolumn z tabeli *Produkty*.

4.3. Transformacja danych

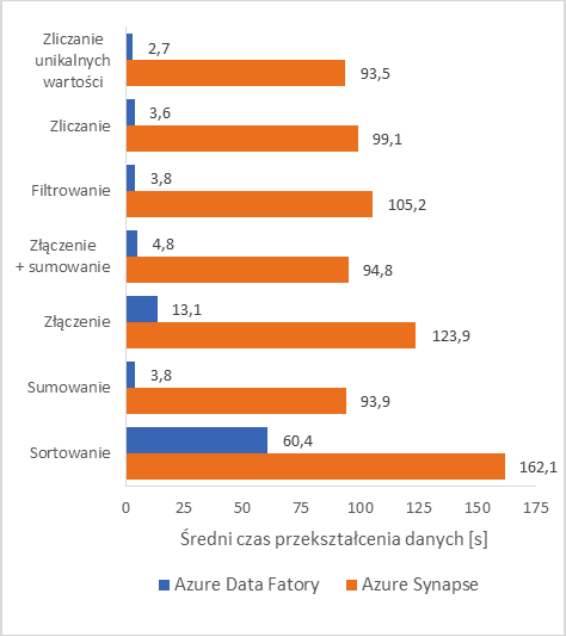
W następnym etapie badań dokonano porównania średniego czasu przekształcenia danych w poszczególnych środowiskach. Pod uwagę wzięto następujące operacje:

- sortowanie,
- zliczanie,
- zliczanie unikalnych wartości,
- sumowanie,
- złączenie,
- złączenie + sumowanie,
- filtrowanie.

Sortowanie danych zostało wykonane z użyciem danych z tabeli *Produkty*, zawierającej 921 855 rekordów. Źródłem danych dla operacji agregacji, tj. zliczania rekordów, zliczania unikalnych wartości i sumowania, oraz złączenia danych była tabela opisująca nagłówek faktury sprzedażowej oraz tabela opisująca poszczególne pozycje tych faktur. Zawierały one odpowiednio 44 570 i 66 995 rekordów. W operacji filtrowania danych została ponownie użyta tabela nagłówków faktur sprzedaży oraz tabela kodów przyczyny zwrotu produktu, zawierająca 5 018 rekordów. Operację złączenia i agregacji danych przeprowadzono z wykorzystaniem tabeli *Produkty* i tabeli z pozycjami faktur sprzedaży.

Uzyskane czasy 10-ciokrotnego wykonania poszczególnych operacji zostały uśrednione i przedstawione na Rysunku 8. Na podstawie przeprowadzonych badań można stwierdzić, że wszystkie działania przekształcenia danych wykonują się znacznie szybciej w środowisku Azure Data Factory niż w Azure Synapse, przy czym najmniejsza różnica średnich czasów

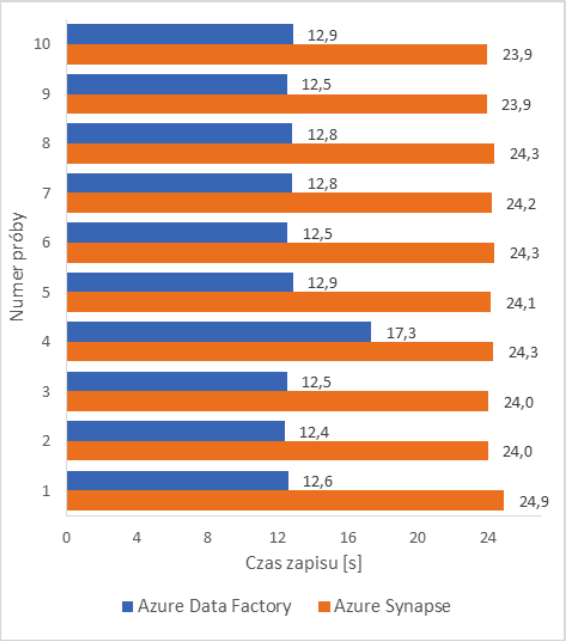
wykonania dotyczy operacji sortowania rekordów (ADF jest niemal 3-krotnie szybszy niż AS), natomiast największa różnica występuje w przypadku operacji zliczania unikalnych wartości (ADF jest 35 razy szybszy od AS).



Rysunek 8: Średni czas transformacji danych dla różnych operacji.

4.4. Zapis danych w zasobie docelowym

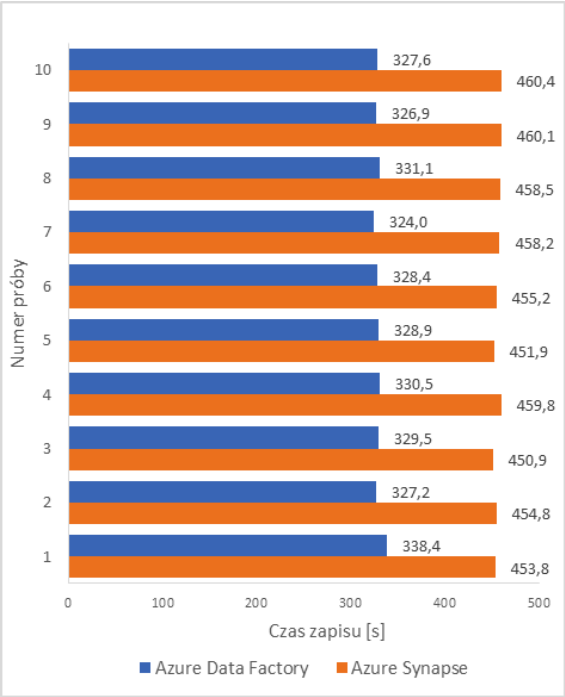
Kolejnym etapem badań była identyfikacja długości czasu ładowania danych do docelowych tabel wymiarów i faktu przy użyciu analizowanych narzędzi. W wyniku transformacji transakcyjnych danych o klientach, do tabeli wymiaru *Klient* załadowano 49 383 rekordy. Czasy wykonania serii 10-ciu prób tej operacji zostały przedstawione na Rysunku 9.



Rysunek 9: Czasy zapisu danych do tabeli wymiaru *Klient*.

Na podstawie otrzymanych rezultatów badań można stwierdzić, że ładowanie danych z użyciem narzędzia Azure Data Factory odbywa się dwukrotnie szybciej niż przy wykorzystaniu środowiska Azure Synapse.

W celu zweryfikowania wpływu liczby rekordów na czas wykonania ładowania danych, przeprowadzono eksperyment polegający na wstawieniu 921 855 rekordów do tabeli wymiaru *Produkt*. Wyniki serii 10-ciu prób tej operacji zobrazowano na Rysunku 10.



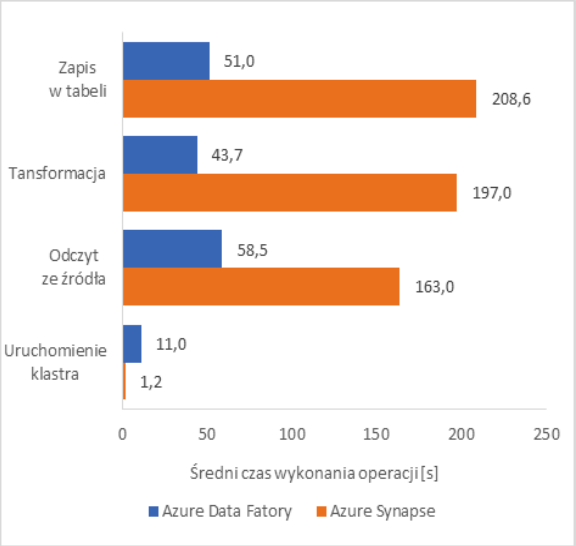
Rysunek 10: Czasy zapisu danych do tabeli wymiaru *Produkt*.

Otrzymane wyniki badania potwierdzają przewagę Azure Data Factory na Azure Synapse również w przypadku ładowania bardzo dużych zbiorów danych. Jednak warto zauważyć znaczne zmniejszenie się różnicy pomiędzy czasami wykonania tej operacji z użyciem wspomnianych narzędzi w miarę wzrostu liczby ładowanych rekordów. Obecnie różnica ta wynosi 39%, podczas gdy dla mniejszej ilości danych była ona równa 91%.

W ostatnim etapie prac badawczych dokonano analizy czasów trwania pełnego cyklu ładowania danych do tabeli faktu *Sprzedaz* z użyciem prezentowanych narzędzi ETL. Uwzględniono tutaj wszystkie operacje, począwszy od pobrania danych z tabel transakcyjnych, opisujących faktury i ich pozycje, poprzez ich transformację do postaci analitycznej, a skończywszy na załadowaniu zagregowanych danych do tabeli faktu. Dodatkowo, w badaniu tym wykorzystano uprzednio utworzone tabele wymiarów *Klient* i *Produkt*. Cykl ładowania został powtórzony 10-cio krotnie w każdym środowisku. W kolejnych powtórzeniach tego procesu zmierzono czasy trwania każdej operacji, a następnie dokonano ich uśrednienia. Otrzymane wyniki zostały zaprezentowane na Rysunku 11.

Na podstawie otrzymanych rezultatów badań można stwierdzić, że proces ETL, odpowiedzialny za ładowa-

nie danych do tabeli faktu, jest realizowany znacznie efektywniej z wykorzystaniem narzędzia Azure Data Factory niż z użyciem Azure Synapse.



Rysunek 11: Średni czas realizacji etapów procesu ETL podczas ładowania danych do tabeli faktu *Sprzedaz*.

Przewaga Azure Synapse jest dostrzegalna tylko w fazie uruchamiania klastra z danymi źródłowymi. Jednak wszystkie dalsze etapy procesu ETL są wykonywane 3-4 krotnie szybciej w środowisku Azure Data Factory.

5. Wnioski

Celem niniejszego artykułu była analiza porównawcza wydajności narzędzi ETL Azure Synapse oraz Azure Data Factory oferowanych przez firmę Microsoft. Na potrzeby badań utworzono analityczną bazę danych, w której alokowano dane przetworzone zgodnie z przyjętymi scenariuszami badawczymi, uwzględniającymi wszystkie typowe operacje realizowane w ramach działań ETL.

Uzyskane rezultaty badań pozwalają jednoznacznie stwierdzić, iż w porównaniu do Azure Synapse, narzędzie Azure Data Factory zapewnia znacznie wyższą efektywność czasową ładowania danych z systemów transakcyjnych do hurtowni danych. Przewaga ADF nad AS jest widoczna zarówno podczas wykonywania zasilania tabel wymiarów jak i tabel faktów. Szczegółowa analiza czasów wykonania poszczególnych etapów pełnego cyklu ETL pozwala dostrzec znaczne różnice w ich wielkości na korzyść ADF. Wprawdzie Azure Synapse zapewnia krótszy czas uruchomienia klastra z danymi źródłowymi, to jednak proces ten jest realizowany tylko raz, na początku procesu ETL, w celu zapewnienia dostępu do danych transakcyjnych. Z punktu widzenia całości przetwarzania danych w ramach ETL, najistotniejsza jest efektywność czasowa wykonania transformacji danych i ich ładowania do tabel docelowych, a w tym obszarze Azure Data Factory dominuje nad Azure Synapse.

Warto jednak zauważyć, że przewaga ADF nad AS maleje wraz ze wzrostem liczby przetwarzanych rekordów. Oznacza to, że wybór właściwego narzędzia do realizacji zadań ETL, które zapewni wysoką efektywność tego działania, powinien być poprzedzony głęboką analizą specyfiki źródłowych tabel relacyjnych.

Literatura

[1] Ł. Bielak, P. Muryjas, Integracja Big Data i Business Intelligence jako innowacyjne rozwiązanie wspomagające funkcjonowanie nowoczesnych organizacji, Journal of Computer Sciences Institute 1 (2016) 6–13.

[2] А. С. Черняев, ETL: обзор инструментов, Молодой ученый, 1 (2019) 23–26, https://moluch.ru/archive/239/55368/, [16.04.2021].

[3] Azure Data Factory documentation, https://docs.microsoft.com/en-us/azure/data-factory/, [16.04.2021].

[4] R. Sudhir, A. Narain, Understanding Azure Data Factory: Operationalizing Big Data and Advanced Analytics Solutions, Apress, Berkeley, 2019.

[5] A. Leonard, K. Bradshaw, SQL Server Data Automation Through Frameworks. Building Metadata-Driven Frameworks with T-SQL, SSIS, and Azure Data Factory, Apress, Berkeley, 2020.

[6] Dokumentacja narzędzia Azure Synapse Analytics, https://azure.microsoft.com/pl-pl/services/synapse-analytics/, [16.04.2021].

[7] Architektura dedykowanej puli SQL (dawniej SQL DW) w usłudze Azure Synapse Analytics, https://docs.microsoft.com/pl-pl/azure/synapse-analytics/sql-data-warehouse/massively-parallel-processing-mpp-architecture, [16.04.2021].

[8] Wybór między modelami zakupów rdzeń wirtualny i DTU — Azure SQL Database i wystąpienie zarządzane SQL, https://docs.microsoft.com/pl-pl/azure/azure-sql/database/purchasing-models#dtu-based-purchasing-model, [16.04.2021].

[9] Przewodnik dotyczący wydajności i dostrajania przepływu danych, https://docs.microsoft.com/pl-pl/azure/data-factory/concepts-data-flow-performance, [16.04.2021].

[10] Monitorowanie przepływów danych, https://docs.microsoft.com/pl-pl/azure/data-factory/concepts-data-flow-monitoring, [16.04.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 132–138
Received: 19 April 2021
Accepted: 04 May 2021

Compilation of iOS frameworks from Linux operating system using open-source tools

Kompilacja bibliotek iOS w systemie Linux z wykorzystaniem narzędzi open-source

Łukasz Rutkowski*, Piotr Kopniak

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

Abstract

This paper analyzes possibility of using open-source tools to compile iOS frameworks in Linux operating system. The purpose of this analysis was to determine how compilation in Linux could be performed and identify possible limitations when using LLVM compiler. The analysis has been performed on own frameworks written using Objective-C and Swift languages containing graphic and text files in different formats and sizes. Results of the analysis show that compilation of iOS frameworks under Linux operating system is possible unless the compiler frameworks use interface components written in xib format for which there are no compilation tools available on Linux operating system.

Keywords: iOS frameworks; LLVM compiler; cross-compilation; open-source tools

Streszczenie

W artykule opisano analizę możliwości wykorzystania narzędzi open-source do kompilacji bibliotek iOS w systemie operacyjnym Linux. Celem analizy jest sprawdzenie możliwości przeprowadzenia kompilacji w systemie Linux oraz wykrycie potencjalnych ograniczeń przy wykorzystaniu kompilatora LLVM. Badania przeprowadzono na autorskich bibliotekach napisanych w językach Objective-C oraz Swift, które zawierały pliki graficzne o różnych formatach i rozmiarach, jak również pliki tekstowe. Uzyskano wyniki które wskazują, że kompilacja bibliotek iOS w systemie Linux jest możliwa pod warunkiem, że kompilowane biblioteki nie wykorzystują komponentów opisanych w formacie xib, do kompilacji których na systemie Linux nie istnieje odpowiednik narzędzia kompilacyjnego z systemu macOS.

Słowa kluczowe: biblioteki iOS; kompilator LLVM; kompilacja skośna; narzędzia open-source

*Corresponding author

Email address: lukasz.rutkowski1@pollub.edu.pl (Ł. Rutkowski)

1. Wstęp

Zależności wykorzystywane w aplikacjach iOS, czyli tak zwane biblioteki to tworzony przez programistów zbiór klas, funkcji lub innych konstrukcji programistycznych [1]. Wykorzystywane są one przez twórców aplikacji iOS do współdzielenia rozwiązań pomiędzy różnego rodzaju aplikacjami bądź wydzielana pewnych części kodu do niezależnych zbiorów. Oficjalnie Apple, czyli firma tworząca system iOS udostępnia narzędzia pozwalające na kompilację takich bibliotek wyłącznie na komputerach z systemem operacyjnym macOS [2]. Jednym z takich narzędzi jest środowisko programistyczne Xcode oraz wchodzące w jego skład narzędzie wiersza poleceń xcodebuild [3-4].

Niniejszy artykuł ma na celu zbadanie, czy istnieje możliwość przeprowadzenia procesu kompilacji bibliotek iOS na komputerach z systemem operacyjnym Linux. W pierwszej kolejności przygotowano biblioteki, które zostaną poddane kompilacji. Opisano sposób ich budowania w systemie macOS, a następnie przedstawiono w jaki sposób proces ten może zostać przeprowadzony również na komputerach z systemem operacyjnym Linux. Na koniec porównano skompilowane biblioteki pod kątem czasu ich kompilacji, rozmiaru plików wynikowych oraz wydajności podczas działania w aplikacji końcowej.

2. Przegląd stanu badań

Temat kompilacji bibliotek iOS na systemie Linux nie jest czymś co zostało obszernie omówione przez innych autorów. Najbliżej zbliżonym tematycznie rozwiązaniem jakie udało się znaleźć jest mechanizm kompilacji wykorzystywany przez silnik tworzenia gier Godot [5-6]. Pozwala on na kompilację gier iOS z poziomu systemu operacyjnego Linux. Opisano tutaj instrukcję krok po kroku jakie operacje należy wykonać oraz jakie narzędzia wykorzystać, aby zbudować grę na systemie Linux, którą będzie można następnie uruchamiać na urządzeniach z systemem iOS. Proces ten przeprowadzony został za pośrednictwem zestawu narzędzi cctools-port dostępnego na systemie Linux stanowiących odpowiednik narzędzi udostępnianych przez firmę Apple na systemie macOS [7].

Kolejnym przykładem rozwiązania innych autorów jest proces dołączania do aplikacji iOS dynamicznych bibliotek analizowany w projekcie [8]. Projekt ten dotyczył zagadnienia dołączania do skompilowanych aplikacji bibliotek monitorujących ich zachowanie w trakcie ich działania. Opracowano tutaj sposób na odtworzenie procesu dołączania bibliotek do aplikacji iOS tak, aby działał także na systemie Linux. Dzięki temu pozbyto się wymogu posiadania urządzeń z systemem macOS pozwalając pracownikom naukowym zajmującym się

tematami bezpieczeństwa do wykonywania swojej pracy z wykorzystaniem systemów Linux.

3. Biblioteki poddane analizie

W celu analizy procesu kompilacji bibliotek przeznaczonych dla systemu iOS wykonywanego na systemie macOS, a następnie dostosowania go do działania na systemie Linux przygotowano zestaw bibliotek iOS. Biblioteki te dobrane zostały w taki sposób, aby móc zaobserwować jaki wpływ na działanie całego procesu kompilacji mają poszczególne funkcjonalności bądź zależności, z których mogą korzystać twórcy bibliotek.

Większość z analizowanych bibliotek to biblioteki autorskie przygotowane w taki sposób, aby analizowały pojedynczy aspekt wpływający na proces kompilacji. Napisane zostały one w języku programowania Objective-C oraz Swift [9-10]. Pozostałe z badanych bibliotek, czyli biblioteki FBLPromises oraz Promises zostały stworzone przez społeczność open-source i udostępnione publicznie przez repozytorium na stronie GitHub [11].

3.1. Biblioteki napisane w języku Objective-C

Pierwszą grupę z przygotowanych bibliotek stanowią biblioteki napisane w języku programowania Objective-C. Jest to język programowania który w latach 2007-2014 był jedynym językiem wykorzystywanym do tworzenia aplikacji i bibliotek dla systemu iOS. Mimo iż od 2014 roku zastąpił go język Swift, autorzy bibliotek wciąż mają możliwość wykorzystywania go w tworzonych bibliotekach [12].

Pierwsza z przygotowanych bibliotek jest biblioteką składającą się wyłącznie z podstawowego kodu źródłowego. Jest to zestaw pliku nagłówkowego opisującego publiczny interfejs tworzonej klasy oraz pliku źródłowego implementującego funkcje wchodzące w skład niniejszej klasy. Biblioteka ta pozwoli na stwierdzenie, czy przygotowane narzędzia będą w stanie skompilować najprostszą możliwą bibliotekę.

Kolejna biblioteka stanowi rozbudowanie poprzedniej o dodatkowe klasy opisane w nowych plikach nagłówkowych i źródłowych. Celem przygotowania niniejszej biblioteki jest zweryfikowanie jaki wpływ na proces kompilacji ma dołączanie kolejnych plików do kodu biblioteki oraz weryfikacja czy wykorzystywanie większej liczby plików uniemożliwi przeprowadzenie procesu kompilacji na systemie Linux.

Trzecią z przygotowanych bibliotek jest biblioteka, która poza kodem źródłowym wykorzystuje także zasoby graficzne. Istotne jest zweryfikowanie czy wykorzystanie takich zasobów w bibliotekach w jakikolwiek sposób wpłynie na proces ich kompilacji na systemie Linux. Na podobnej zasadzie przygotowane zostały dwie kolejne biblioteki, które odpowiednio wykorzystują katalogi zasobów w formacie xcassets oraz pliki widoków w formacie xib. Katalogi xcassets mogą w sobie zawierać zasoby różnych typów jak pliki graficzne i zestawy kolorów dla elementów graficznych aplikacji, które mogą być przygotowane w kilku wersjach dostosowanych pod różne właściwości urządzeń jak rozmiar wyświetlacza bądź aktywny motyw interfejsu [13]. Pliki xib są plikami XML i wykorzystywane są do opisu graficznego interfejsu użytkownika aplikacji [14].

Ostatnia biblioteka napisana w języku Objective-C została przygotowana w taki sposób, aby w udostępnianej przez siebie funkcjonalności wykorzystywała jedną z poprzednio przygotowanych bibliotek jako zależność dynamiczną. Umożliwi to zweryfikowanie w jaki sposób biblioteki mogą wykorzystywać kod innych bibliotek i jak wpłynie to na sposób kompilacji.

3.2. Biblioteki napisane w języku Swift

W przypadku języka programowania Swift przygotowane zostały jedynie dwie autorskie biblioteki. Pierwsza z nich składa się z minimalnego zestawu elementów, czyli jednego pliku źródłowego niewykorzystującego żadnych dodatkowych zależności. Druga zaś została rozbudowana o wykorzystanie systemowych modułów Foundation (zestaw bazowych funkcjonalności systemowych), Dispatch (zestaw wspierający tworzenie wielowątkowego kodu) oraz UIKit (zestaw do budowy i obsługi interfejsu graficznego użytkownika) [15-17]. Zbadanie czy wykorzystanie tych modułów nie zablokuje możliwości kompilacji bibliotek na systemie Linux jest istotne ze względu na fakt, iż większość z obecnie tworzonych bibliotek wykorzystuje przynajmniej jeden z nich.

Biblioteki wykorzystujące zasoby plikowe, katalogi zasobów bądź pliki xib nie zostały przygotowane w tym języku programowania, ponieważ ich kompilacja dla bibliotek napisanych w języku Swift przebiega w taki sam sposób jak dla bibliotek napisanych w języku Objective-C.

3.3. Dodatkowe biblioteki

Innym zestawem analizowanych bibliotek są biblioteki przygotowane przez społeczność open-source. Są to biblioteki Promises oraz FBLPromises z repozytorium promises stworzonego przez firmę Google [18]. Zawierają one kod ułatwiający wykonywanie asynchronicznych operacji na kilku wątkach. Pierwsza z bibliotek została napisana w całości w języku programowania Objective-C. Druga zaś została napisana w języku Swift oraz dodatkowo jako zależność wykorzystuje kod z biblioteki napisanej w języku Objective-C. Kompilacja niniejszych bibliotek pozwoli stwierdzić, czy cały proces działa poprawnie nawet w przypadku bibliotek o bardziej rozbudowanym kodzie.

Ostatnią analizowaną biblioteką jest biblioteka, której kod źródłowy napisany został zarówno w języku Objective-C oraz języku Swift. Działa ona na zasadzie udostępnienia klas, które w swojej implementacji zawierają funkcje odwołujące się do klas napisanych w drugim języku. Biblioteka ta pozwoli ona na weryfikację czy biblioteki, które wykorzystują oba języki programowania w ramach jednego kodu także będzie można poprawnie skompilować na systemie Linux.

4. Metoda badań

Analizie podlegało badanie możliwości kompilacji bibliotek na systemie Linux, czasu ich kompilacji, rozmiaru plików końcowych oraz wydajności przygotowanych bibliotek iOS na systemie operacyjnym Linux i macOS. Wszystkie z bibliotek skompilowane zostały w wersjach dostosowanych do działania na urządzeniach fizycznych opartych o architekturę arm64. Na systemie macOS każda z bibliotek skompilowana została z wykorzystaniem narzędzia xcodebuild w trybie „Debug" (konfiguracja domyślna) oraz w trybie „Release" (konfiguracja z dodatkowymi optymalizacjami). Dodatkowo wszystkie te biblioteki zostały skompilowane przy użyciu specjalnie przygotowanego skryptu bash bezpośrednio wykorzystującego niskopoziomowe narzędzia kompilujące.

Na systemie operacyjnym Linux wszystkie z przygotowanych bibliotek skompilowane zostały wyłącznie przy użyciu skryptu bash ze względu na brak narzędzia xcodebuild na tym systemie. Wszystkie z wymienionych narzędzi zostały wywołane w powłoce konsoli w ramach systemowego polecenia time pozwalającego mierzyć czas ich trwania.

Do przeprowadzenia kompilacji na systemie macOS wykorzystany został komputer o następującej specyfikacji:

- System operacyjny macOS 11.0.1
- Procesor 3,2 GHz 6-Core Intel Core i7-8700B
- Pamięć 16GB
- Karta graficzna Intel UHD Graphics 630 1536 MB

Wykorzystany komputer z systemem Linux posiadał następującą specyfikację:

- System operacyjny Ubuntu 20.04.1 LTS
- Procesor 3,2 GHz 4-Core Intel Core i5-4460
- Pamięc 8GB
- Karta graficzna Nvidia GeForce GTX 950

4.1. Sposób analizy możliwości kompilacji

Proces kompilacji przeprowadzony na systemie macOS został wykonany w kilku podejściach. Przy każdym podejściu wykorzystane zostały oddzielne narzędzia i konfiguracje. W ten sposób stwierdzono poprawność działania przygotowanych skryptów oraz umożliwiono dokładne porównanie procesu względem tego, który został wykonany na systemie Linux.

Podczas pierwszego podejścia każdą z przygotowanych bibliotek skompilowano przy użyciu narzędzia xcodebuild w konfiguracji Release. W drugim podejściu wykorzystano polecenie xcodebuild w konfiguracji Debug. Trzecie podejście do procesu kompilacji zostało przygotowane z użyciem autorskich skryptów kompilacyjnych omijających narzędzie xcodebuild. Skrypty te przeprowadzają kompilację poprzez wywoływanie bardziej niskopoziomowych od polecenia xcodebuild poleceń kompilatorów clang oraz swift. Przykładem takiego skryptu jest kod widoczny na Listingu 1.

Listing 1: Skrypt kompilujący bibliotekę Objective-C na systemie macOS

```
1  #!/bin/bash
2  set -e
3  FRAMEWORK_NAME="Minimal_Objective_C"
4  WORKING_DIR="$(pwd)"
5  BUILD_DIR="${WORKING_DIR}/build/Device"
6  FRAMEWORK_DIR="${BUILD_DIR}/${FRAMEWORK_NAME}.framework"
7  IOS_SDK="/Applications/Xcode
       .app/Contents/Developer/Platforms/iPhoneOS
       .platform/Developer/SDKs/iPhoneOS14.4.sdk"
8  rm -rf "${BUILD_DIR}"
9
10 # Krok 1. Przygotowanie struktury biblioteki
11 mkdir -p "${FRAMEWORK_DIR}/Headers"
12 mkdir -p "${FRAMEWORK_DIR}/Modules"
13 cp "Sources"/*.h "${FRAMEWORK_DIR}/Headers/"
14 cp "module.modulemap" "${FRAMEWORK_DIR}/Modules/"
15 cp "Info.plist" "${FRAMEWORK_DIR}/"
16
17 # Krok 2. Kompilacja pliku źródłowego
18 clang -x objective-c \
19     -fmodules \
20     -fobjc-arc \
21     -fmodule-name=${FRAMEWORK_NAME} \
22     -target arm64-apple-ios12.3 \
23     -isysroot ${IOS_SDK} \
24     -c "Sources/MOCLibraryInfo.m" \
25     -o "${BUILD_DIR}/MOCLibraryInfo.o"
26 echo "${BUILD_DIR}/MOCLibraryInfo.o" >
       "${BUILD_DIR}/LinkFileList"
27
28 # Krok 3. Linkowanie biblioteki
29 xcrun -sdk iphoneos libtool -dynamic \
30     @"${BUILD_DIR}/LinkFileList" \
31     -install_name
           @rpath/${FRAMEWORK_NAME}
           .framework/${FRAMEWORK_NAME} \
32     -arch_only arm64 \
33     -o "${FRAMEWORK_DIR}/${FRAMEWORK_NAME}" \
34     -syslibroot "${IOS_SDK}" \
35     -lSystem \
36     -compatibility_version 1 \
37     -current_version 1
```

Proces kompilacji bibliotek na systemie Linux przeprowadzony został wyłącznie w jednym podejściu dla każdej biblioteki. Do kompilacji wykorzystane zostały autorskie skrypty przygotowane pod system macOS, do których wprowadzone zostały konieczne modyfikacje polegające na zastąpieniu poleceń kompilatorów clang oraz swift ich odpowiednikami w wersjach działających na systemie Linux. Takie wersje narzędzi przygotowane zostały poprzez pobranie iOS SDK, a następnie kompilację narzędzi z zestawu cctools-port i kompilatora Swift w taki sposób, aby wykorzystywały pobrany iOS SDK.

Listing 2: Skrypt analizujący czas kompilacji

```
1  #!/bin/bash
2
3  LIBRARY_DIR="$1"
4
5  pushd "$LIBRARY_DIR"
6
7  for i in {1..10}
8  do
9      [ -d build ] && rm -r build
10     (time ./build_device.sh > /dev/null) 2>&1 | grep
           real | awk '{print $2}'
11 done
12
13 popd
```

Podczas wykonywania wszystkich operacji kompilacji mierzony był czas ich działania za pośrednictwem specjalnie przygotowanego skryptu zaprezentowanego

na Listingu 2. Działa on na zasadzie uruchomienia polecenia kompilacyjnego 10 razy na podanej bibliotece i wyświetlenia czasu trwania poszczególnych operacji kompilacji w sekundach.

4.2. Sposób analizy rozmiaru bibliotek

Analiza rozmiaru bibliotek przeprowadzona została poprzez uruchomienie przygotowanego skryptu wyświetlającego rozmiary w bajtach bibliotek znajdujących się we wskazanym katalogu.

Listing 3: Skrypt analizujący rozmiar bibliotek

```
1  #!/bin/bash
2
3  DIRECTORY="$1"
4
5  for framework in $(ls $DIRECTORY)
6  do
7      echo -n "$framework - "
8      find "$DIRECTORY/$framework" | xargs stat -f%z | awk
           '{ s+=$1 } END { print s }'
9  done
```

Skrypt ten (Listing 3) działa na zasadzie odwołania się do systemowych poleceń w celu odczytania informacji o rozmiarze całej biblioteki, czyli licząc rozmiar pliku wykonywalnego biblioteki jak i pozostałych plików znajdujących się w jej strukturze. Jako parametr wejściowy przekazany został katalog, w którym umieszczone zostały wszystkie z przygotowanych bibliotek. Dzięki temu otrzymano informacje o rozmiarach każdej biblioteki wykonując tylko jedno wywołanie niniejszego skryptu.

4.3. Sposób analizy wydajności bibliotek

Analiza wydajności przygotowanych bibliotek przeprowadzona została poprzez zbadanie wpływu użycia danej biblioteki na czas uruchamiania aplikacji testowej oraz zbadanie czasu wykonywania funkcji udostępnianych przez wszystkie z bibliotek.

Do pomiaru wpływu bibliotek na czas startu aplikacji wykorzystana została zmienna środowiskowa „DYLD_PRINT_STATISTICS", która aktywuje wbudowaną funkcjonalność pomiarową polecenia dyld odpowiedzialnego za operację wczytywania bibliotek dynamicznych [19]. Zmienna ta została aktywowana w momencie uruchamiania przygotowanej aplikacji testowej. Czas zwracany przez tę funkcjonalność oznacza jak długo trwało ładowanie aplikacji do wywołania jej głównej funkcji „main()". W wyniku zwracany jest zestaw czasów (Rysunek 1), z których interesującym nas wpisem jest wpis „dylib loading time", który bezpośrednio dotyczy czasu wczytywania bibliotek dynamicznych.

```
Total pre-main time: 222.72 milliseconds (100.0%)
         dylib loading time: 142.73 milliseconds (64.0%)
        rebase/binding time:  25.28 milliseconds (11.3%)
            ObjC setup time:   5.91 milliseconds (2.6%)
           initializer time:  48.78 milliseconds (21.9%)
           slowest intializers :
             libSystem.B.dylib :   5.75 milliseconds (2.5%)
   libBacktraceRecording.dylib :  10.48 milliseconds (4.7%)
    libMainThreadChecker.dylib :  28.91 milliseconds (12.9%)
```

Rysunek 1: Przykładowy wynik czasów trwania poszczególnych operacji wykonywanych przy starcie aplikacji

Drugim analizowanym aspektem wydajności bibliotek był pomiar czasu trwania funkcji udostępnianych przez wszystkie biblioteki. W tym celu przygotowany został test jednostkowy. Listing 4 prezentuje jak niniejszy test odwołuje się do funkcji każdej biblioteki w ramach bloku measure. Napisany został z wykorzystaniem biblioteki XCTest będącej systemową biblioteką pozwalającą na pisanie testów jednostkowych z możliwością pomiaru wydajności [20].

Listing 4: Test jednostkowy wykonujący pomiar wydajności bibliotek

```
class TestAppTests: XCTestCase {
    func testPerformance() throws {
        measure {
            _ = MOCLibraryInfo().name
            _ = MOCLibraryInfo().language
            EOCFile1.description()
            EOCFile2.description()
            EOCFile3.description()
            EOCFile4.description()
            EOCFile5.description()
            AOCProvider.provideTextFileContent()
            AOCProvider.provideImage()
            DOCUseDependency.textFromDependency()
            OACReadAssets.readColorFromAssets()
            OACReadAssets.readImageFromAssets()
            OVDemoView.fromNib()
            _ = SwiftLibraryInfo.name
            _ = SwiftLibraryInfo.language
            _ = SwiftFoundationWorker.makeFoundationObject()
            _ = SwiftFoundationWorker.makeUILabel()
            SwiftFoundationWorker.doDelayedWork()
            Promise { "Text" }
                .delay(2)
                .then { print("Swift promises \($0()) work!") }
            MLObjcFile.getSwiftText()
            _ = SwiftFile().getObjcText()
        }
    }
}
```

5. Wyniki badań

5.1. Możliwość kompilacji

Podstawowym celem niniejszej pracy było określenie z jakich zależności bądź funkcjonalności mogą korzystać biblioteki, aby możliwa była ich kompilacja za pośrednictwem systemu Linux. Tabela 1 przedstawia zestawienie poszczególnych elementów z jakich mogą składać się biblioteki w zależności od tego w jakim języku napisana została analizowana biblioteka.

Tabela 1: Wyniki kompilacji bibliotek na systemie Linux

	Swift	Objective-C
Kompilacja możliwa	tak	tak
Systemowe moduły (np. Foundation, UIKit)	tak	tak
Zasoby plikowe (.txt, .png, .jpg)	tak	tak
Katalogi zasobów (.xcassets)	nie	nie
Zasoby widoków (.xib)	nie	nie
Zależności bibliotek	tak	tak

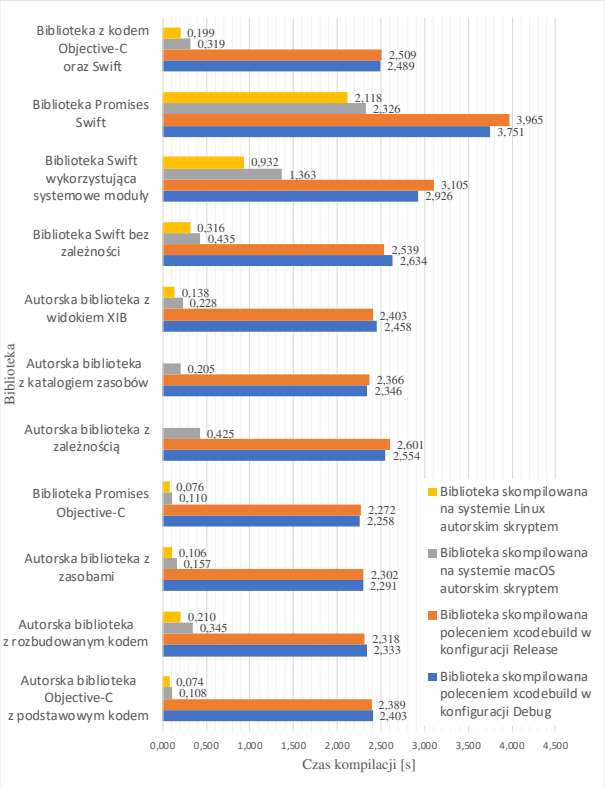
Tabela ta pokazuje, które z bibliotek udało się poprawnie skompilować na danych systemach, a które nie. Stwierdzenie „tak" oznacza, że biblioteka została skompilowana oraz poprawnie uruchomiona w testowej aplikacji. Stwierdzenie „nie" oznacza, że nie udało się znaleźć istniejących narzędzi open-source, które pozwoli-

łyby na przeprowadzenie kompilacji na systemie Linux, w związku z czym stwierdzono, że obecnie taka kompilacja nie jest możliwa. W przypadku systemu macOS będącego oficjalnym systemem do tworzenia bibliotek iOS takie zestawienie nie było konieczne, ponieważ kompilacja udała się z powodzeniem w każdej możliwej konfiguracji.

Główną właściwość jaką można zauważyć w tabeli 1 jest fakt, iż większość funkcjonalności nie blokuje możliwości ich kompilacji na systemie Linux. Jedynie dla bibliotek zawierających katalogi zasobów w postaci xcassets bądź zasoby widoków w postaci plików xib nie udało się pomyślnie przeprowadzić pełnej kompilacji. Kompilacja ta nie udała się z powodu braku odpowiednich narzędzi, które byłyby w stanie przetworzyć podane pliki w postać końcową rozumianą przez aplikacje iOS.

5.2. Czasy kompilacji

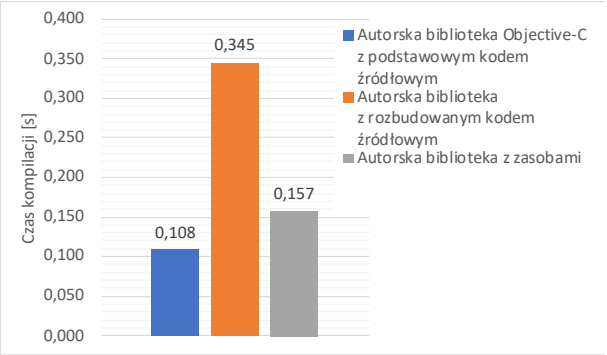
Podczas analizy czasu kompilacji bibliotek pierwszą interesującą cechą jaką zaobserwowano są znacznie dłuższe czasy kompilacji przeprowadzonych z wykorzystaniem polecenia xcodebuild w konfiguracjach Debug lub Release w porównaniu do czasów kompilacji wykonanych z wykorzystaniem autorskiego skryptu. Dla każdej biblioteki czasy kompilacji tymi narzędziami były dłuższe niż 2 sekundy, co widoczne jest na rysunku 2. W odróżnieniu do tego proces kompilacji przeprowadzony z wykorzystaniem autorskich skryptów był krótszy i osiągnął maksymalny czas 2 sekund tylko w przypadku biblioteki Promises napisanej w języku Swift.



Rysunek 2: Średnie czasy kompilacji bibliotek.

Porównując czasy kompilacji różnych bibliotek, które zostały skompilowane autorskim skryptem na systemie macOS, co zostało zaprezentowane na rysunku 3, widać, że dodając kolejne pliki bądź dodatkowe zasoby wydłużamy czas kompilacji.

Jest to spowodowane tym, że dla każdego takiego pliku wywoływane są dodatkowe operacje, które wydłużają cały proces. Widzimy także, że czas kompilacji biblioteki o rozbudowanym kodzie źródłowym jest dłuższy niż czas kompilacji biblioteki z jednym plikiem i z dołączonymi zasobami. Taka różnica wynika z tego, że pliki graficznie nie są poddawane zamianie na pliki wykonywalne, a jedynie są kopiowane do odpowiedniego miejsca w wynikowym folderze biblioteki. Pliki źródłowe zaś muszą zostać poddane wymienionemu procesowi kompilacji, który jest bardziej czasochłonny od zwykłej operacji kopiowania.



Rysunek 3: Porównanie czasów kompilacji różnych bibliotek autorskim skryptem na systemie macOS.

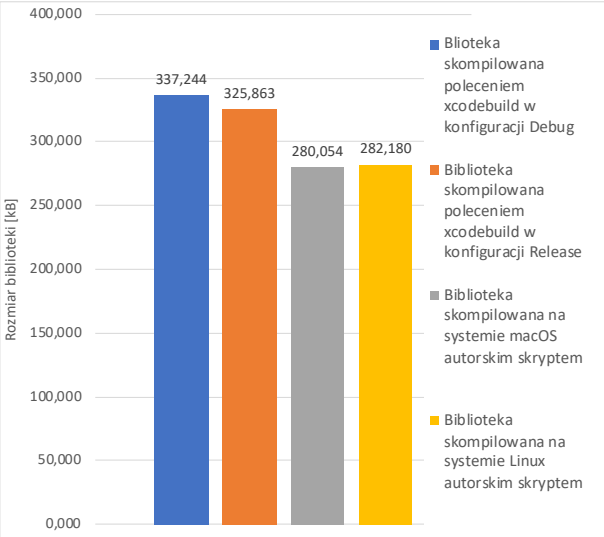
5.3. Rozmiar bibliotek

Kolejnym analizowanym aspektem jest rozmiar skompilowanych bibliotek. Zestawienie wyników dla wszystkich z przygotowanych bibliotek skompilowanych na systemie macOS i Linux przy użyciu różnych sposobów kompilacji zaprezentowano w tabeli 2. Widoczną cechą jest znacznie większy rozmiar bibliotek wykorzystujących zasoby bądź katalog zasobów w porównaniu do pozostałych bibliotek. Wynika to z faktu, iż takie zasoby dołączone są bezpośrednio do struktury bibliotek tym samym zwiększając ich rozmiary. Analizowane biblioteki, które wykorzystywały takie zasoby zawierają w sobie plik graficzny o rozmiarze około 1,5 MB, dlatego też ich rozmiar jest większy od pozostałych bibliotek o taką wartość (pomijając rozmiar samego pliku wykonywalnego).

Zestawienie uśrednionych rozmiarów skompilowanych bibliotek pod względem sposobu ich kompilacji przedstawiono na rysunku 4. Widoczną różnicą jest mniejszy rozmiar bibliotek skompilowanych przy użyciu autorskiego skryptu w porównaniu do bibliotek zbudowanych przy użyciu poleceń xcodebuild. Różnice te wynikają z zestawu operacji wykonywanych w ramach polecenia xcodebuild, które nie są wykonywane podczas kompilacji z użyciem autorskiego skryptu, a które wpływają na rozmiar utworzonego pliku wykonywalnego.

Tabela 2: Rozmiary skompilowanych bibliotek

Biblioteka	Rozmiar biblioteki skompilowanej poleceniem xcodebuild w konfiguracji Debug [kB]	Rozmiar biblioteki skompilowanej poleceniem xcodebuild w konfiguracji Release [kB]	Rozmiar biblioteki skompilowanej na systemie macOS autorskim skryptem [kB]	Rozmiar biblioteki skompilowanej na systemie Linux autorskim skryptem [kB]
Biblioteka z podstawowym kodem źródłowym	92,24	92,016	52,4	53,584
Biblioteka z rozbudowanym kodem źródłowym	80,614	80,502	54,434	55,626
Biblioteka z zależnością	75,187	75,203	58,226	69,758
Biblioteka z katalogiem zasobów	1604,568	1604,584	1581,143	-
Biblioteka z zasobami	1591,421	1591,405	1573,997	1569,001
Biblioteka z widokiem XIB	80,17	80,106	56,423	-
Biblioteka „FBLPromises”	301,498	268,842	222,36	234,22
Biblioteka „Promises”	501,555	436,483	306,572	302,811
Biblioteka Swift bez zależności	126,64	125,48	80,996	82,099
Biblioteka Swift wykorzystująca systemowe moduły	134,971	132,283	87,176	87,035
Biblioteka z kodem Objective-C oraz Swift	131,072	130,552	84,328	85,487



Rysunek 4: Uśrednione rozmiary bibliotek skompilowanych różnymi sposobami.

Przykładem takiej operacji jest kompilacja oraz dołączanie dodatkowego pliku uzupełniającego kompilowaną bibliotekę o dodatkowe symbole określające informacje o jej wersji. Generowanie i kompilacja niniejszego pliku została pominięta przy tworzeniu autorskiego skryptu, ponieważ nie jest on konieczny do prawidłowego działania biblioteki, a przygotowanie odpowiednich poleceń komplikowałoby dodatkowo cały proces kompilacji na obu systemach.

5.4. Wydajność bibliotek

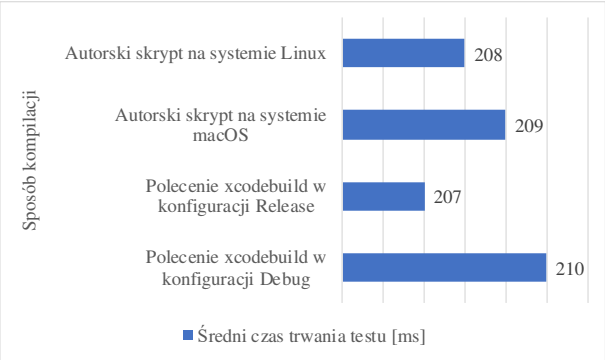
Przy analizie wydajności zbudowanych bibliotek głównym badanym elementem był ich wpływ na czas uruchamiania aplikacji testowej. Tabela 3 pokazuje otrzymane wyniki w rozbiciu na sposób kompilacji bibliotek. Dla bibliotek skompilowanych autorskim skryptem wpływ na czas ich wczytywania przy starcie aplikacji jest podobny bez względu na to, na którym systemie zostały skompilowane. Na tej podstawie możemy stwierdzić, że narzędzia, które zostały przygotowane do kompilacji bibliotek na systemie Linux nie wpłynęły negatywnie na sposób ich działania podczas startu aplikacji.

Większa różnica występuje, jeżeli porównamy biblioteki skompilowane z wykorzystaniem autorskiego skryptu do bibliotek skompilowanych z wykorzystaniem narzędzia xcodebuild, których wczytywanie trwało dłużej średnio o 7-12 ms. Różnica ta wynika głównie z większego rozmiaru bibliotek skompilowanych za pośrednictwem narzędzia xcodebuild.

Drugim mierzonym aspektem wydajności bibliotek był pomiar czasu odwoływania się do funkcji udostępnianych przez publiczny interfejs skompilowanych bibliotek. Uśrednione wyniki takich pomiarów zaprezentowane zostały na rysunku 5. Wszystkie z otrzymanych wyników mieszczą się w przedziale od 207 ms do 210 ms.

Tabela 3: Wyniki czasów wczytywania bibliotek przy starcie aplikacji

Sposób kompilacji	Wartość minimalna [ms]	Wartość maksymalna [ms]	Średnia [ms]	Mediana [ms]
Polecenie xcodebuild w konfiguracji Debug	143,56	149,22	146,97	147,19
Polecenie xcodebuild w konfiguracji Release	144,89	152,23	148,73	152,23
Autorski skrypt na systemie macOS	137,42	143,09	140,66	140,94
Autorski skrypt na systemie Linux	136,97	142,73	139,64	139,18



Rysunek 5: Porównanie wydajności skompilowanych bibliotek.

Takie wyniki oznaczają, że sposób kompilacji nie wpłynął negatywnie na wydajność kodu bibliotek. Zarówno biblioteki, które zostały skompilowane na systemie macOS, jak i te, które zostały skompilowane na systemie Linux działają w ten sam sposób. Jest to spowodowane prostym kodem przygotowanych bibliotek, na którym narzędzia kompilacyjne nie miały możliwości przeprowadzenia operacji optymalizacyjnych.

6. Wnioski i podsumowanie

Niniejszy artykuł opisuje analizę procesu kompilacji bibliotek iOS na systemie Linux. W badaniu w pierwszej kolejności zweryfikowano, czy kompilacja bibliotek iOS jest możliwa na tym systemie dla podstawowych typów bibliotek. Następnie sprawdzono, czy istnieją funkcjonalności wykorzystywane w bibliotekach, które uniemożliwią wykonanie kompilacji na systemie Linux. Ostatecznie zbadano: czas kompilacji, rozmiary oraz wydajność skompilowanych bibliotek.

Wyniki uzyskane w niniejszej pracy pokazały, że kompilacja bibliotek iOS możliwa jest do przeprowadzenia na systemie Linux pod warunkiem, że nie będą one korzystać z niektórych funkcjonalności. Zidentyfikowano, że wykorzystanie katalogów zasobów o rozszerzeniu xcassets bądź plików opisujących widoki interfejsu użytkownika tworzone w postaci plików w formacie xib nie pozwalają na przeprowadzenie kompilacji bibliotek na systemie Linux.

Analiza czasów kompilacji pokazała, że wykorzystując system Linux możliwe jest uzyskanie zbliżonych, a nawet krótszych czasów kompilacji bibliotek w porównaniu do systemu macOS. Niestety dzieje się to kosztem rozmiaru bibliotek, które na podstawie przeprowadzonej analizy są większe w przypadku bibliotek skompilowanych wykorzystując narzędzia na systemie Linux. Ostatnia z analiz, czyli analiza wydajności bibliotek pokazała, że zarówno biblioteki skompilowane na systemie macOS jak i biblioteki skompilowane na systemie Linux mają zbliżoną do siebie wydajność.

Literatura

[1] What are Frameworks?, https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html#//apple_ref/doc/uid/20002303-BBCEIJFI, [03.2021].

[2] S. Grimshaw, Mastering MacOS Programming, Packt Publishin, 2017.

[3] B. Alexander, J. B. Dillon, K. Y. Kim, R. Górczyński, Tworzenie aplikacji na platformę iOS 5: z wykorzystaniem Xcode, Interface Builder, Instruments, GDB oraz innych kluczowych narzędzi, Wydawnictwo Helion, 2012.

[4] R. Pouclet, Pro iOS Continuous Integration, Apress 2014.

[5] Godot Engine – Free and open source 2D and 3D game engine, https://godotengine.org, [03.2021].

[6] Cross-compiling for iOS on Linux – Godot Engine latest documentation, https://docs.huihoo.com/godotengine/godot-docs/godot/reference/cross-compiling_for_ios_on_linux.html, [03.2021].

[7] Apple cctools port for Linux and *BSD, https://github.com/tpoechtrager/cctools-port, [03.2021].

[8] Automated embedding of dynamic libraries into iOS applications from GNU/Linux, https://docplayer.net/60535186-Automated-embedding-of-dynamic-libraries-into-ios-applications-from-gnu-linux.html, [03.2021].

[9] S. G. Kochan, Ł. Piwkom, Objective-C: praktyczny podręcznik tworzenia aplikacji na systemy iOS i Mac OS X!, Helion, 2012.

[10] P. Pasternak, Swift od podstaw: praktyczny przewodnik, Helion, 2017.

[11] A. Pipinellis, GitHub Essentials, Packt Publishing, 2015.

[12] C. G. Garcia, J. P. Espada, B. C. Pelayo G-Bustelo, J. M. Cueva Lovelle, Swift vs. Objective-C: A New Programming Language, International Journal of Interactive Multimedia and Artificial Intelligence 3(3) (2015) 74-81, http://dx.doi.org/10.9781/ijimai.2015.3310.

[13] Asset Catalog Format Reference, https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_ref-Asset_Catalog_Format/index.html, [03.2021].

[14] Xcode Overview: Using Interface Builder, https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html, [03.2021].

[15] Foundation – Apple Developer Documentation, https://developer.apple.com/documentation/foundation, [03.2021].

[16] Dispatch – Apple Developer Documentation, https://developer.apple.com/documentation/dispatch, [03.2021].

[17] UIKit – Apple Developer Documentation, https://developer.apple.com/documentation/uikit, [03.2021].

[18] Promises is a modern framework that provides a synchronization construct for Swift and Objective-C, https://github.com/google/promises, [03.2021].

[19] Man page dla polecenia dyld, https://www.manpagez.com/man/1/dyld/, [03.2021].

[20] N. Godfrey, Agile Swift: Swift Programming Using Agile Tools and Techniques, Apress, 2016.

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 139–143
Received: 19 April 2021
Accepted: 04 May 2021

Performance analysis of Svelte and Angular applications

Analiza wydajnościowa aplikacji Svelte i Angular

Gabriel Białecki*, Beata Pańczyk

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

Abstract

The aim of this article is to check if the Svelte-based client part of a web application is more effective than the standard Angular approach. The article presents a comparison of page components rendering times on the basis of two test applications prepared in both frameworks. For the performance tests, scenarios were prepared in which the times of adding and removing a different number of page components were examined. Application tests were performed using the Selenium Webdriver package. The research results clearly showed that the new approach used for DOM manipulation (Svelte v.3.0) is more efficient than the standard solution used in Angular applications (v.10.2).

Keywords: Svelte; Angular; performance; frontend

Streszczenie

Celem artykułu jest sprawdzenie, czy nowsze rozwiązanie tworzenia części klienckiej aplikacji internetowej oparte na Svelte jest bardziej efektywne w porównaniu do standardowego podejścia stosowanego w Angular. W artykule przedstawiono porównanie czasów renderowania się komponentów strony na przykładzie dwóch aplikacji testowych przygotowanych w obu szkieletach programistycznych. Do przeprowadzenia testów wydajnościowych przygotowano scenariusze, w których zbadano czasy dodawania i usuwania różnej liczby komponentów strony. Testy aplikacji były wykonane przy pomocy pakietu Selenium Webdriver. Wyniki badań wskazały jednoznacznie na fakt, że nowe podejście do manipulacji DOM (Svelte v. 3.0), jest bardziej wydajne niż korzystanie ze standardowego rozwiązania stosowanego w aplikacjach Angular (v.10.2).

Słowa kluczowe: Svelte; Angular; wydajność, frontend

*Corresponding author

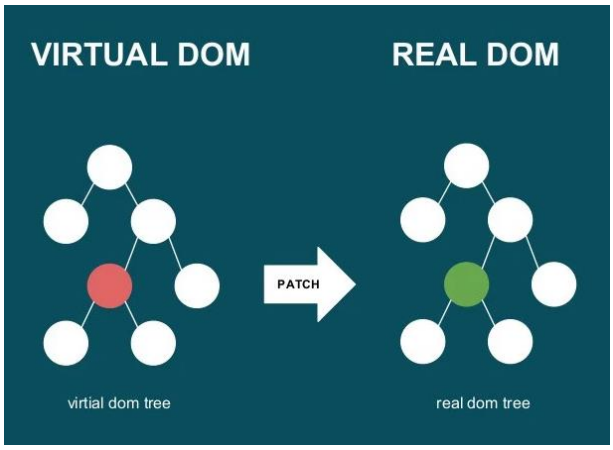
Email address: gabriel.bialecki@pollub.edu.pl

1. Wstęp

Najważniejszą kwestią, na której skupia się programista części klienckiej (ang. Frontend) jest minimalizacja operacji na DOM (ang. Document Object Model), co jest ściśle powiązane ze zwiększeniem wydajności aplikacji internetowej.

Początkowo do tworzenia stron internetowych korzystano z czystego HTML, CSS i JavaScript. Z biegiem lat strony internetowe stały się na tyle obszerne i rozbudowane, że pojawiały się kolejne biblioteki (np. JQuery – rok 2006) i szkielety aplikacji (np. Backbone – 2010, AngularJS – 2010, Ember – 2011) pracujące z rzeczywistym DOM (ang. real DOM). Z czasem i te szkielety programistyczne stały się coraz mniej wydajne. Każda akcja użytkownika wymagała bowiem wyrenderowania całego drzewa DOM od początku, co trwało coraz dłużej, z uwagi na większe wymagania użytkowników i rozbudowane interfejsy graficzne.

Rozwiązaniem tego problemu było pojawienie się biblioteki React (2013) i zastosowanie wirtualnego drzewa DOM (ang. virtual DOM). DOM wirtualny polega na stworzeniu kopii rzeczywistego drzewa DOM w pamięci aplikacji i operowaniu na tej wewnętrznej kopii. Jeśli użytkownik, poprzez swoje działanie zmienia coś na witrynie - aplikacja rejestruje to zdarzenie, ale modyfikacje wprowadza w wirtualnym, wewnętrznym drzewie DOM. Porównując wirtualny DOM z rzeczywistym, renderuje następnie tylko te treści strony, które się zmieniły. Takie podejście do problemu zwiększyło wówczas wydajność i użyteczność stron (Rysunek 1).



Rysunek 1: Sposób działania virtual DOM [1].

Obecnie wszystkie szkielety programistyczne tworzenia aplikacji klienckich korzystają z wzorców architektonicznych, takich jak MVC (ang. Model-View-Controller), MVP (ang. Model-View-Presenter) lub MVVM (ang. Model-View-View-Model), które ułatwiają organizację struktury aplikacji z graficznym interfejsem użytkownika. Aplikacje klienckie tworzone za ich pomocą są osobnymi bytami – aplikacjami typu

SPA (ang. Single Page Application). Wysyłają zapytania do serwera o dane, modyfikują i przechowują te informacje w swoich modelach oraz wyświetlają je w swoich widokach. Aplikacje SPA umożliwiają renderowanie części strony w zależności od akcji użytkownika w ramach jednego dokumentu HTML, w różny sposób optymalizując pracę z DOM.

Jednym z najpopularniejszych obecnie szkielety programistyczne do tworzenia aplikacji typu SPA, jest następca AngularJS - Angular [2], rozwijany przez Google od 2014 roku. Aplikacje Angular są tworzone w języku TypeScript i renderowane po stronie serwera, są modułowe i zorientowane na komponenty. Niestety obecnie wydajność tych aplikacji nie jest zadowalająca.

Tradycyjne frameworki umożliwiają pisanie deklaratywnego kodu sterowanego stanem, ale przeglądarka musi wykonać dodatkową pracę, aby przekonwertować te deklaratywne struktury na operacje DOM, wykorzystując DOM wirtualny.

Alternatywne podejście do problemu manipulacji DOM przedstawia stosunkowo nowy framework Svelte (2018). Działa on podczas kompilacji, przekształcając komponenty w wydajniejszy kod imperatywny, który aktualizuje DOM. W rezultacie można pisać aplikacje o lepszych parametrach wydajnościowych [3]. Svelte wprowadza rozwiązanie hybrydowe. Operuje na real DOM, renderując ponownie jedynie te elementy, które się zmieniły [4-5]. Takie podejście ma skrócić czas renderowania elementów na stronie. Svelte nie jest zwyczajną biblioteką, lecz specjalnym kompilatorem, który tworzy z kodu Svelte niemal czysty kod JavaScript, HTML oraz CSS. Utrzymuje kod danego komponentu w jednym pliku oraz kompiluje go do niewielkich rozmiarów, co pozwala na przyspieszenie pracy na stronie WWW. Svelte rozwija się bardzo dynamicznie i reaguje na zmieniające się środowiska programistyczne po stronie klienta. Framework ten wnosi dużo użytecznych rozwiązań, które wyróżniają go na tle dobrze już znanych frameworków (React, Angular, Vue) [6].

Zarówno Svelte jak i Angular wykorzystują język TypeScript, co stało się powodem wybrania frameworka Angular do porównania wydajności z aplikacją Svelte [3].

2. Cel i teza badawcza

Celem artykułu jest zbadanie, czy aplikacja kliencka tworzona w Svelte jest wydajniejsza w porównaniu do standardowego podejścia stosowanego w aplikacjach Angular.

Postawiono następującą tezę badawczą: ***„Aplikacja Svelte v.3.0 szybciej wykonuje operacje na komponentach graficznego interfejsu użytkownika w porównaniu do analogicznej aplikacji Angular v.10.2”.***

3. Przegląd literatury

Svelte wprowadza nowe rozwiązania [7] w stosunku do dobrze już znanych frameworków takich jak Angular, React czy Vue:

- cały kod jest kompilowany,
- nie ma wirtualnego DOM,
- CSS jest wbudowany.

W artykule [8] autor pisze: „Svelte jest alternatywą do wiodących framworków typu frontend. Podobnie jak React czy Vue może być on używany to tworzenia całych aplikacji, jak również jedynie niestandardowych elementów będących częścią już istniejących rozwiązań. W Svelte postawiono jednak na jeden ważny aspekt, a mianowicie prostotę pisanego kodu”. W pracy tej zwrócono uwagę na ciekawe porównania dotyczące popularności oraz wydajności Svelte w odniesieniu do innych stosowanych obecnie frameworków. Na podstawie podanych tam danych sformułowano wniosek, że Svelte jest szybszy niż aplikacje Vue i React.

Na stronie [9] przedstawiono wyniki porównania prostych aplikacji tworzonych z wykorzystaniem różnych frameworków typu frontend i różnych ich wersji (w sumie 23 aplikacje stworzone m.in. w AngularJS, Ember, Angular, React, Vue i Svelte). W porównaniu uwzględniono trzy kryteria: wydajność, rozmiar (KB) i liczbę linii kodu w podobnych aplikacjach. W przedstawionych tam zestawieniach wynikowych, dla każdego kryterium Svelte plasuje się w pierwszej trójce.

4. Metoda badań

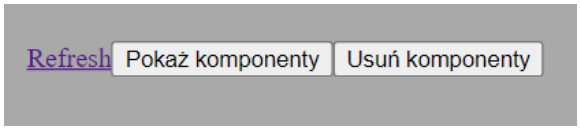
Badania wydajnościowe zostały przeprowadzone w oparciu o dwie proste, takie same co do funkcjonalności aplikacje testowe, zaimplementowane odpowiednio w Svelte, wersja 3.0 i Angular, wersja 10.2. Aplikacje zostały przygotowane w taki sposób, aby można było zmierzyć czasy renderowania i usuwania różnej liczby komponentów graficznego interfejsu.

Testy aplikacji przeprowadzono na komputerze:

- laptop: Lenovo ThinkPad T470p,
- procesor: Intel i7-770HQ,
- system operacyjny: Windows 10 Pro, wersja 1903 (build 18362.756),
- pamięć RAM: 32GB 2400 MHz,
- grafika: Intel HD Graphics 630
- przeglądarka Google Chrome 89.

Obie aplikacje na początkowej stronie zawierają trzy elementy (Rysunek 2):

- link do odświeżenia strony,
- przycisk do pokazania odpowiedniej liczby komponentów,
- przycisk do usunięcia uprzednio pokazanych elementów ze strony.



Rysunek 2: Początkowy wygląd interfejsu aplikacji.

Po wykonaniu akcji pokazania oraz usunięcia elementów wyświetla się czas, który upłynął od przyciśnięcia przycisku i wykonania odpowiedniej akcji.

W zależności od wyboru – będzie pokazywana lub usuwana różna liczba komponentów (100, 1000 lub

10000). Komponent jest prostym polem tekstowym z atrybutami, opisany etykietą.

Testy obu aplikacji były wykonywane przy pomocy pakietu Selenium Webdriver. Jest to narzędzie do automatycznego testowania, które znacząco ułatwiło ten proces, gdyż każda aplikacja była zbadana w 6 różnych trybach (normal, slow3G, fast3G [10] oraz incognito na każdej z tych trzech przepustowości sieci). W każdym teście liczba powtórzeń (renderowanie/usuwanie z witryny odpowiedniej liczby komponentów) wynosiła 100. Skrypt testowy przedstawiony jest na Listingu 1.

Listing 1: Skrypt testujący obie aplikacje

```
require('chromedriver');
const {Builder, By, until, Capabilities} =
require('selenium-webdriver');
(async function example() {
 const chromeCapabilities = Capabilities.chrome();
 const chromeOptions = {'args':
['--test-type', '--incognito']};
 chromeCapabilities.set('goog:chromeOptions',
     chromeOptions);
 let driver = await new Builder()
.forBrowser('chrome').withCapabilities(
chromeCapabilities).build();
 const appUrl = 'http://localhost:4200/';
 const howManyComponents = 100;
 const numberOfRepetitions = 100;
 const networkOptions = [{
   name: "slow3G",
   download_throughput: 500 * 1024 / 8 * .8,
   upload_throughput: 500 * 1024 / 8 * .8,
   latency: 400 * 5 },

   {
     name: "fast3G",
     download_throughput: 1.6 * 1024 * 1024 / 8 * .9,
     upload_throughput: 750 * 1024 / 8 * .9,
     latency: 150 * 3.75,
   }, {
     name: "normal",
     download_throughput: -1,
     upload_throughput: -1,
     latency: 0,
   }
 ]

 const data = {
   ["normalDataFor" + howManyComponents +
"ComponentsAdded"]: '',
   ["normalDataFor" + howManyComponents +
“ComponentsRemoved"]: '',
   ["slow3GDataFor" + howManyComponents +
"ComponentsAdded"]: '',
   ["slow3GDataFor" + howManyComponents +
"ComponentsRemoved"]: '',
   ["fast3GDataFor" + howManyComponents +
"ComponentsAdded"]: '',
   ["fast3GDataFor" + howManyComponents +
"ComponentsRemoved"]: '',
 }

try {
   await driver.get(appUrl);
   let element;
   for (const currentOptions of networkOptions) {
     await driver.setNetworkConditions(currentOptions);
     const dataIndexAdded = currentOptions.name +
"DataFor" + howManyComponents + "ComponentsAdded";
     const dataIndexRemoved = currentOptions.name +
"DataFor" + howManyComponents + "ComponentsRemoved";

     for (let i = 0; i < numberOfRepetitions; i++) {
       element = await
driver.wait(until.elementLocated(By.id(
'show-components')));
       await element.click();
       data[dataIndexAdded] += await driv-
er.findElement(By.id('time-elapsed-render')).getText() +
'\n';
     element = await
driver.wait(until.elementLocated(By.id(
'delete-components')));
       await element.click();
       data[dataIndexRemoved] += await driv-
er.findElement(By.id('time-elapsed-delete')).getText() +
'\n';
       element = await driv-
er.wait(until.elementLocated(By.id('refresh')));
       await element.click();
     }
   }
 } finally {
   driver.executeScript(`
       for (const [key, value] of Ob-
ject.entries(arguments[0])) {
           var file = new Blob([value], {type: 'txt'});
           var a = document.createElement("a"),
               url = URL.createObjectURL(file);
           a.href = url;
           a.download = key + '.txt';
           document.body.appendChild(a);
           a.click();
           setTimeout(function() {
               document.body.removeChild(a);
               window.URL.revokeObjectURL(url);
           }, 0)
       };`, data);
 }
})();
```

Przed uruchomieniem skryptu należy zbudować aplikację, która będzie poddawana testowi i uzupełnić zmienną appUrl odpowiednim adresem URL. Na listingu pokazana jest konfiguracja w trybie incognito dla 100 komponentów.

Po uruchomieniu testu pojawia się okno przeglądarkowe (Google Chrome), w którym skrypt wywołuje akcje pokazania komponentów, zapamiętuje czas wykonania akcji, wykonuje akcję służącą do usunięcia komponentów, zapamiętuje nowy czas i odświeża stronę w celu powtórzenia tych działań po 100 razy. Po zakończeniu wszystkich iteracji skrypt pobiera pliki tekstowe z danymi.

5. Wyniki badań

W sumie zebrano 7 200 pojedynczych pomiarów czasów dodawania i usuwania komponentów HTML ze strony w obu aplikacjach. Wyniki uporządkowano w kategorie odpowiadające liczbom komponentów (100, 1000 i 10000) i trybom przeglądarki (zwykły i incognito). Następnie w każdej kategorii stworzono tabele reprezentujące czasy w obu aplikacjach oraz trybach sieciowych (fast3G, slow3G, normal). Każdy tryb, w każdej kategorii zawierał po 100 wierszy. Z każdego takiego zestawu wierszy wyliczono średnią.

Wyniki porównania średnich czasów usuwania komponentów pokazano na rysunkach 3-5. Rysunki 6-8 przedstawiają średnie czasy renderowania tych komponentów.

Na Rysunkach 9 oraz 10 przedstawiono średnie czasy renderowania się i usuwania 10 000 komponentów w trybie incognito dla każdej z aplikacji. Jak widać na rysunkach, wyniki te nie wykazują znacznych zmian pomiędzy trybem zwykłym a incognito, więc nie umieszczono pełnego zestawu wykresów.

Wszystkie uśrednione dane przedstawiono w Tabelach 1-4.

Tabela 1: Średnie czasy (ms) usuwania komponentów w Angular

Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	1,43	13,58	146,37
slow 3g	1,44	13,52	147,04
fast 3g	1,44	13,35	147,08
incognito	1,45	13,66	145,49
slow 3g incognito	1,38	13,73	145,70
fast 3g incognito	1,50	13,43	145,59

Tabela 2: Średnie czasy (ms) usuwania komponentów w Svelte

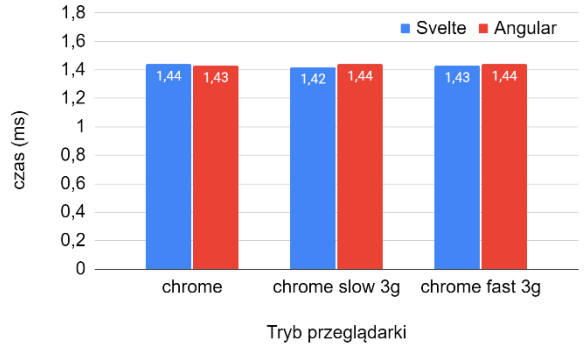
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	1,44	14,18	143,50
slow 3g	1,42	13,40	142,68
fast 3g	1,43	13,87	144,11
incognito	1,46	14,22	142,49
slow 3g incognito	1,42	13,72	143,64
fast 3g incognito	1,55	14,01	144,98

Tabela 3: Średnie czasy (ms) renderowania komponentów w Angular

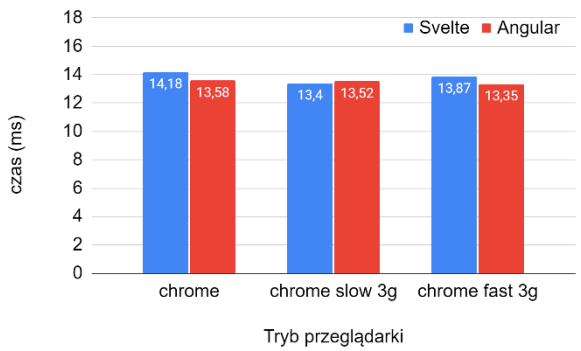
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	17,97	150,96	822,54
slow 3g	18,34	150,59	818,28
fast 3g	18,33	150,42	845,31
incognito	18,73	153,85	820,62
slow 3g incognito	18,89	154,52	850,90
fast 3g incognito	18,89	153,89	827,94

Tabela 4: Średnie czasy (ms) renderowania komponentów -w Svelte

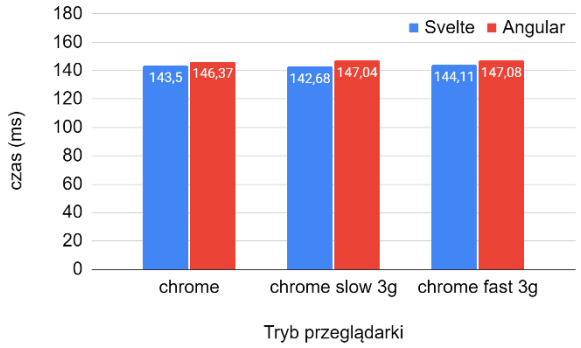
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	2,67	21,12	195,70
slow 3g	2,71	21,90	277,67
fast 3g	2,65	21,10	238,21
incognito	2,51	20,60	190,17
slow 3g incognito	2,59	21,99	267,91
fast 3g incognito	2,53	20,93	230,99



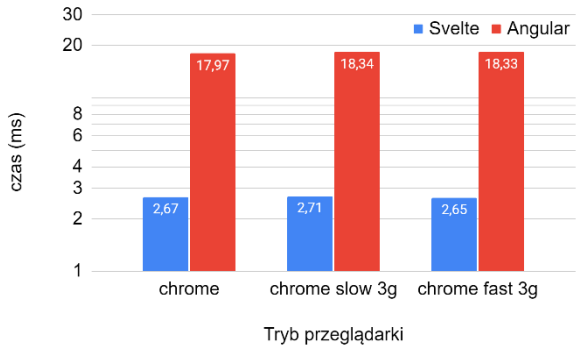
Rysunek 3: Średnie czasy usuwania 100 komponentów w zwykłym trybie.



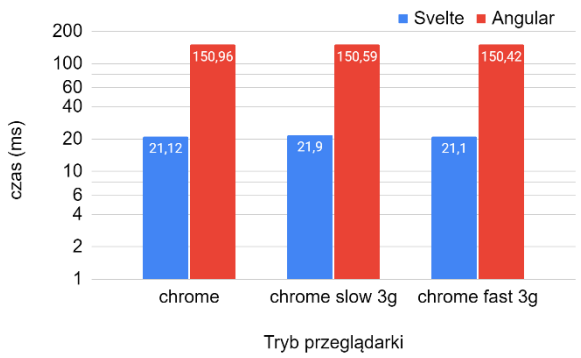
Rysunek 4: Średnie czasy usuwania 1 000 komponentów w zwykłym trybie.



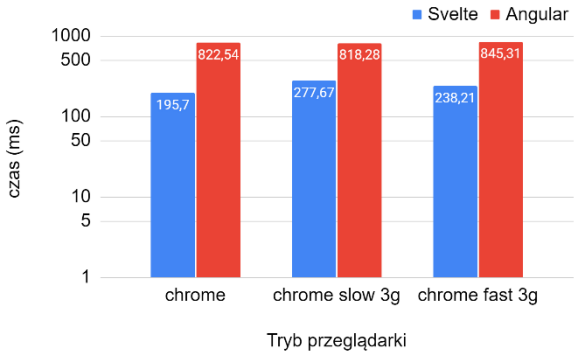
Rysunek 5: Średnie czasy usuwania 10 000 komponentów w zwykłym trybie.



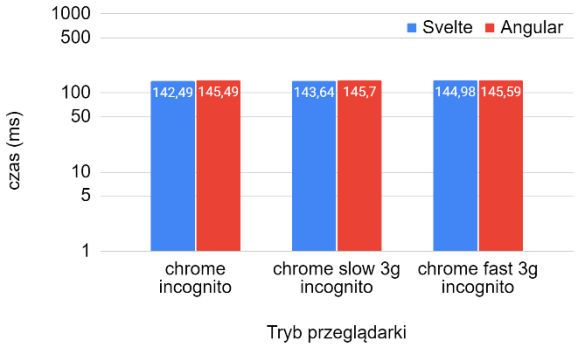
Rysunek 6: Średnie czasy renderowania 100 komponentów w zwykłym trybie.



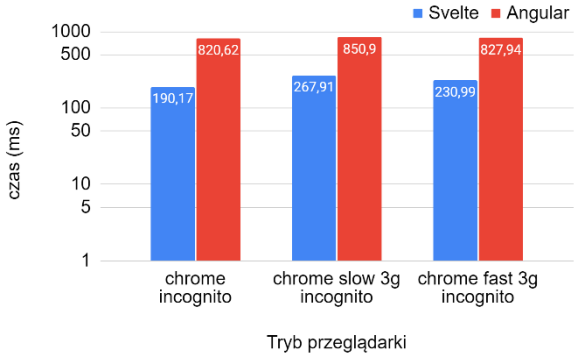
Rysunek 7: Średnie czasy renderowania 1 000 komponentów w zwykłym trybie.



Rysunek 8: Średnie czasy renderowania 10 000 komponentów w zwykłym trybie.



Rysunek 9: Średnie czasy usuwania 10 000 komponentów w trybie incognito.



Rysunek 10: Średnie czasy renderowania 10 000 komponentów w trybie incognito.

6. Wnioski

Na podstawie uzyskanych wyników można sformułować następujące wnioski:

- aplikacja Svelte w przypadku renderowania komponentów jest 4-krotnie (w przypadku renderowania się 10 000 komponentów) szybsza od aplikacji Angular (Rysunki 6-8, 10),
- dla obu aplikacji nie ma dużego znaczenia jaki tryb przeglądarki jest używany (Tabele 1-4),
- w przypadku aplikacji Angular - im wolniejsza sieć tym więcej (~42% w trybie zwykłym *slow3G*, ~22% *fast3G*, ~40% w trybie incognito *slow3G*, ~21% w incognito *fast3G*) czasu potrzeba na wyrenderowanie komponentów (szczególnie widoczne jest to przy 10 000 komponentów, Rysunki 8 i 10),
- w obu przypadkach nie ma różnicy pomiędzy trybem zwykłym a trybem incognito (Rysunki 8 i 10).
- czasy usuwania komponentów w obu przypadkach są podobne (Rysunki 3-5).

Usuwanie elementów HTML nie wymaga dużo pracy ani ze strony JavaScript, ani przeglądarki, dlatego te czasy w obu aplikacjach są porównywalne. Deweloperzy często to wykorzystują - gdy np. użytkownik korzysta z portalu dostarczającego wiele treści wymagającej przewijania w dół strony - po pewnym czasie zwykle treść z góry witryny znika. W ten sposób programiści mogą łatwo i szybko ograniczyć liczbę elementów na stronie, co nie wpływa w znaczący sposób na wydajność aplikacji, niezależnie od technologii w jakiej jest ona stworzona.

Inaczej już wygląda mechanizm renderowania się komponentów w obu aplikacjach. Angular stosuje mechanizm virtual DOM, natomiast Svelte operuje na real DOM.

Uzyskane rezultaty potwierdzają postawioną na początku tezę badawczą w kwestii dodawania komponentów. Wyniki są też zgodne z zestawieniami prezentowanymi w artykułach [8, 9].

Literatura

[1] N. Joshi, obraz real i virtual DOM, https://medium.com/@nami996joshi/real-dom-448076454705, [13.04.2021].

[2] Dokumentacja Angular, https://angular.io/docs, [13.04.2021].

[3] O. Therox, Svelte i TypeScript, https://svelte.dev/blog/svelte-and-typescript, [10.03.2021].

[4] Dokumentacja Svelte, https://svelte.dev/docs, [13.04.2021].

[5] S. Kołodziejczak, Svelte – wszystko, co powinieneś wiedzieć o nowej wersji tego narzędzia, https://geek.justjoin.it/svelte-frontend, [13.04.2021].

[6] A. Haseeb, Real and Virtual DOM, https://medium.com/@ahaseeb12251998/virtual-dom-vs-real-dom-angular-vs-react-framework-vs-libraries-spas-vs-mpa-s-946fceb70955, [13.04.2021].

[7] T. Tolliday, Getting Acquainted With Svelte, the New Framework on the Block, 2020, https://css-tricks.com/getting-acquainted-with-svelte-the-new-framework-on-the-block/, [13.04.2021].

[8] D. Glazer, Svelte – „nowy" framework frontendowy!, https://www.ideo.pl/firma/o-nas/nasze-publikacje/svelte-3-nowosci,150.html, [13.04.2021]

[9] J. Schae, A RealWorld Comparison of Front-End Frameworks with Benchmarks, 2020, https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1/, [13.04.2021].

[10] Narzędzie Chrome Dev Tools na GitHub https://github.com/ChromeDevTools/devtools-frontend/blob/80c102878fd97a7a696572054007d40560dcdd21/front_end/sdk/NetworkManager.js#L252-L274, [04.03.2021].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 144–150
Received: 20 April 2021
Accepted: 04 May 2021

Model of the text classification system using fuzzy sets

Model systemu klasyfikacji tekstu z wykorzystaniem zbiorów rozmytych

Dmytro Salahor*, Jakub Smołka

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

Abstract

Classification of work's subject area by keywords is an actual and important task. This article describes algorithms for classifying keywords by subject area. A model was developed using both algorithms and tested on test data. The results were compared with the results of other existing algorithms suitable for these tasks. The obtained results of the model were analysed. This algorithm can be used in real-life tasks.

Keywords: text classification; "fuzzy" sets; classification; fuzzy rules; fuzzy logic

Streszczenie

Klasyfikacja tematyki pracy według słów kluczowych jest aktualnym i ważnym zadaniem. W artykule opisano algorytmy klasyfikowania słów kluczowych według obszaru tematycznego. Model został opracowany przy użyciu dwóch algorytmów i przetestowany na danych testowych. Uzyskane wyniki porównano z wynikami innych istniejących algorytmów odpowiednich do tego zadania. Uzyskane wyniki modelu analizowano. Algorytm ten może być stosowany w zadaniach rzeczywistych.

Słowa kluczowe: klasyfikacja tekstu; zbiory rozmyte; klasyfikacja; reguły rozmyte: logika rozmyta

*Corresponding author

Email address: s97218@pollub.edu.pl (D. Salahor)

1. Introduction

With the increase of text information amount, it is very important to understand which area the text belongs to. When working with databases containing scientific papers and articles, getting the subject area of the work without looking through its entire text is need. In this case, work attributes such as keywords can be used. However, manually assigning keywords or subject areas will take too much time and resources. Therefore, automating the definition of text attributes will save time and resources. Also, when selecting attributes for text information, in reality, you can get belonging not to one class, but to several. Modern classification systems must support this capability.

The purpose of the article is to develop, describe and test a text classification model. The model must have high accuracy and non-binary classification capability.

This article proposes an algorithm for recognizing the subject area of work by its keywords. Two versions of the classifier are described in detail, models are implemented and tested on real data. A comparison was made with other algorithms such as Decision tree [1-5], Support-vector machine [6-10], Neural network [11-14].

2. Review of text analysis methods

In the [15] a comparative analysis of the best way to classify complaint texts for a state online complaint service in Indonesia was made. The following algorithms participated in the comparison: Naive Bayes [16-17], Maximum Entropy [18-19], K-Nearest Neighbours [20], Random Forest [21-22], and Support Vector Machines, and two ensemble strategies - hard voting and soft voting. The results also indicate that generally all the ensemble methods performed better than the individual classifiers.

In [23] a keyword recognition system for texts using neural networks and hierarchical taxonomies was described. For each category in taxonomy own pre-trained neural network is used. The neural network uses function "logistic" and solver "adam". Using the combined hierarchical system of neural networks, with a sample of 2843 documents, an accuracy of 77.87% was achieved.

In [24] a new approach to categorizing text for category recognition for online newspapers was described. The approach has strict rules that determine a possibility of using a particular database for classification. To classify the system, the methods Support Vector Machine (SVM), Hidden Markov was chosen. Using this approach gives high accuracy. When using Hidden Markov Model (HMM) [25-26], the best accuracy was obtained.

In [27] an automatic text classification system based on the genetic algorithm classifier has been developed. Before classification with genetic algorithm, the text data was pre-processed, translated into correct representation, and the selection of features was made. When testing the system, 20291 documents of 6 categories were used. Using 1000 selected words, a performance of 0.748 was achieved, which is more than using kNN and decision tree classifiers.

In [28] a method for investigating the temporal patterns using keywords in the comments of online newspapers was proposed. The system should return a conclusion based on the content of the comment. Text data has been cleaned and categorized. As a result, the following conclusions were drawn: the drop in activity is associated with the end of the event and the time before

this event; lexeme frequency maps were obtained; using temporal analysis, you can get a clear picture of the sentiment of changes.

In [29] a system for extracting keywords and sentiment from Twitter posts was developed. The Archivist API was used to collect information. The sample consisted of 40 000 tweets. The using the developed system in [30] with knowledge enhancer and synonym binder allowed to improve the results from 0.1% to 55% in comparison with the usual keyword search.

In [30] semi-automatic context analysis and text correction using specialized linguistic graphs was used and a system for text correction and context analysis has developed as a part of logic of web application. Developing the model, graphs composed of special neurons of different types was used. Comparison with similar programs for text correction has performed. The results of [30] are: an effective graph model for writing words and punctuation marks appearing in the examined text; developed methods to obtain texts from several sources for graph construction; text analysis and contextual adjustment methods developed and implemented; web application (website) was made that allows to use of implemented algorithms.

3. Classification model using fuzzy-sets

There are many ways to classify texts. This article will consider a method for classifying texts that is suitable for classifying scientific papers in different subject areas. The model is based on «fuzzy sets» [31,32].

Fuzzy set is a set, each element of which is matched with a real number in the range [0; 1], which indicates how much the element belongs to set [31].

The idea of such a classifier is to assign a real number within the range [0..1] for each possible variant. This classification method is more flexible than the usual classification method, where each variant is assigned an integer from the set {0, 1}. When writing the article, two versions of such a classifier were used and compared: using key phrases and using keywords.

The classifier model can be divided into two parts: training and direct classification. The model uses a normalized fuzzy set during training and classification. Values are normalized upon classification.

3.1. Training model description

Let there be a set containing *N* subject areas, among which the classification should be performed. As input data for training, the subject area and keywords that relate to the subject area are given in a form of a set:

$$T = \{P, \{Tp1, Tp2..Tpa..Tpk\}\} \quad (1)$$

where *P* – subject area for key phrases, *Tp1* .. *Tpk* – key phrases.

Depending on *Tpa*, subject areas *P* are added to the corresponding set *Ma* (2).

A set *Ma* has structure as below:

$$Ma = \{Ipa, \{\{P1, i1\}, \{P2, i2\}..\{Pa, ia\}..\{PN, iN\}\} \quad (2)$$

where *Ipa* – key phrase, *P1..PN* – subject areas, *i1..iN* – integer number, indicating the frequency of use of the key phrase in the subject area. It should be noted that the set does not allow duplicates.

The set of *Ma* sets, each of which describes its own keyword, makes up the set Mp, which is the result of training:

$$Mp = \{M1, M2..Ma..Mm\} \quad (3)$$

where *M1..Mm* – a set containing information about in which subject areas and with what frequency the key phrase occurs.

As a result, the more training material is available for training, the more power the set *Mp* has, and the more accurate results the algorithm can provide.

3.2. Model of classifier description

The input data for the finished classifier is the set *S*, containing the key phrases of the work, the classification of which must be carried out:

$$S = \{s1, s2..sa..sN\} \quad (4)$$

where *s1* .. *sN* - strings containing key phrases.

The key phrases have the form of strings, each of which comprises words separated by whitespace.

The output of the classifier is the set *Out*:

$$Out = \{\{P1, f1\}, \{P2, f2\}..\{Pa, fa\}..\{PN, fN\}\} \quad (5)$$

where *f1* .. *fN* – real numbers in the range [0..1], which show the value of belonging to a particular subject area, *P1* .. *PN* – subject areas.

Also done normalizing each *fa* multiplying it by the normalization factor *j*:

$$j = 1/\max(a1..aN) \quad (6)$$

where *max(a1..aN)* – maximal value in set *{a1..aN}*. The classification algorithm is as follows:

Algorithm 1: Classification

```
Input : N = count of subject areas;
Set Mp;
Output: Out with N elements;
for Each keyword do
    Create set Ma from Mp by keyword;
    Get sum of ia in all Pa sets;
    Write sum of ia in fa in set Out;
    if sum of ia not equal 0 then
        Normalize fa: divide fa by sum of all ia in fa;
    end
end
for Each fa in Out do
    Normalize Out: divide fa by max of fa;
end
return Out;
```

3.3. Classifier modification

To increase recognition by the classifier, a modification of the classifier is proposed. The modification consists in using not key phrases, but their components – keywords for training and classification. Keywords can be obtained by splitting key phrases by words.

In the modified model of the classifier, during training in (1), keywords are given as *Tp1..Tpk*; when classified in (4), keywords are given as *S1..SN*.

4. Model and algorithm realization

The model using the above algorithms was implemented in the Java language. HashMap and ArrayList were used as sets. The input data was submitted as file with CSV format (comma-separated values). Every row in the file has values: authors, title, link of work, author keywords, index keywords, subject area. Output data printed in console.

4.1. Input data preparing

The Scopus database was chosen as the input data source [33]. The database is free and allows you to export the required attributes of the works to a file of the required format. All papers from Lublin University of Technology were taken as exported data. A total of 6543 works were selected, of which 655 were used for testing. The resulting data sample was divided into two parts, 90% and 10%. The first part will be used for training, the second for testing. The classification was made between 27 subject areas.

Next subject areas was chosen:

- engineering,
- materials science,
- physics and astronomy,
- mathematics,
- computer science,
- environmental science,
- chemistry,
- chemical engineering,
- energy,
- agricultural and biological sciences,
- social sciences,
- earth and planetary sciences,
- biochemistry, genetics and molecular biology,
- medicine,
- economics, econometrics and finance,
- business, management and accounting,
- decision sciences,
- multidisciplinary,
- pharmacology, toxicology and pharmaceutics,
- arts and humanities,
- health professions,
- neuroscience,
- immunology and microbiology,
- psychology,
- dentistry,
- nursing,
- veterinary.

4.2 Evaluating classification results method

Since the classification is done by fuzzy sets, you need to decide what values at the output of the classifier are considered sufficient to assign a keyword search. In evaluating using next variables:

The model leaves in set all values greater than 0.5. Choosing the sorting of 0.5 due to the fact that if the *fa* is smaller than 50% in the fuzzy set, such variant under repeated less probable than others.

Then *Out* is compared with the set *R*:

$$R = \{Pa1..Pam\} \tag{7}$$

where *Pa1..Pam* are real subject areas for the *S* keyword list.

The goal of the classifier is to get an *Out* that is as close as possible to *R*. It is also necessary to evaluate the classification accuracy. The model uses the following algorithm:

Algorithm 2: Accuracy evaluation

```
Input : Out, R;
Output: accuracy;
if | Out | ≥ | R | then
  | accuracy = | ( R ∩ Out ) | / | Out |;
else
  | accuracy = | ( R ∩ Out ) | / | R |;
end
return accuracy;
```

In algorithm next statements used:

$$|(R \cap Out)|/|Out| \tag{8}$$

where $|(R \cap Out)|$ - number of right keywords/ key phrases in *Out*, $|Out|$ - number of right keywords/ key phrases in *Out*.

$$|(R \cap Out)|/|R| \tag{9}$$

where $|(R \cap Out)|$ - number of right keywords/ key phrases in *Out*, $|R|$ - number of right keywords/ key phrases in *R*.

5. Text classification using popular classificators

For comparison with the work of the developed algorithm, classifiers of the following types were tested: Decision tree, Support-vector machine, Backpropagation neural network. The following results were compared:

- using first keyword,
- using first three keywords,
- using last three keywords,
- using random three keywords,
- using all keywords.

Since the ability of the model described in the article to perform multiple classification is already superior to the models used in comparison, which perform only binary classification, therefore, to be able to compare

them, the model developed in the article will also perform binary classification.

Model building was performed in KNIME - open-source software for data analysis. KNIME allows you to work with data, perform data processing and visualize the results. A visual editor is used to work in KNIME. It is also possible to use code inserts in programming languages such as Java, Python, Javascript.

When creating the required models, the following elements were used:

- for neural network - Rprop MLP Learner and MultiLayerPerceptron Predictor [34, 11-13],
- for support-vector machine - SVM Learner and SVM Predictor [35,36],
- for decision tree - Decision Tree Learner and Decision Tree Predictor. [37,38].

When selecting keywords, inserts in the Java language were used.

6. Analysis of the obtained results

The analysis is presented separately for the results of comparing binary and non-binary models. In comparison of binary models, the analysis of the results obtained during the operation of the two algorithms described in the article, as well as the results of the operation of the Decision tree, Support-vector machine, Backpropagation neural network models, is given. In comparison of non-binary models, the analysis of the results obtained by the operation of the two algorithms described in the article are presented.

6.1. Analysis of the binary models

In Table 1, the simulation results for binary classifiers are presented.

Table 1: Accuracy values of various algorithms

	Standard	Modified	Decision tree	SVM	NN
Only 1 keyword	0.538	0.586	0.665	0.585	0.585
Only 3 first keywords	0.564	0.611	0.626	0.586	0.586
Only 3 last keywords	0.505	0.611	0.63	0.586	0.585
Random 3 keywords	0.541	0.614	0.611	0.586	0.586
All keywords	0.750	0.771	0.586	0.585	0.585

When using a small count of keywords, the best results are obtained with the Decision tree. The results in the Support-vector machine and Neural network are almost unchanged when the number of keywords is changed. When using the maximum count of keywords, the best results are obtained with the modified algorithm described in the article. A visual representation of the accuracy of these models is shown in Figure 1.

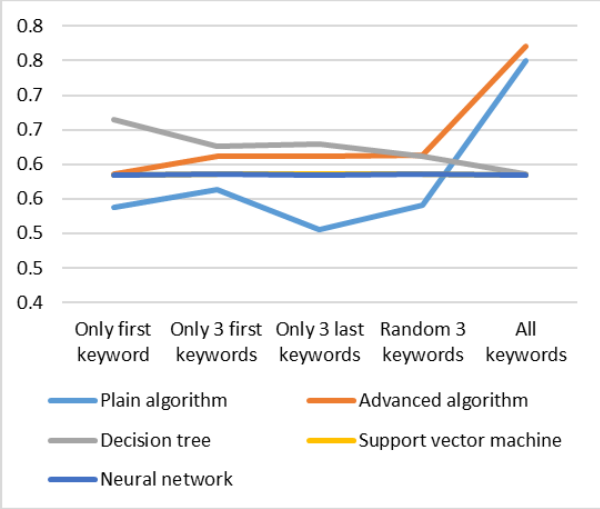


Figure 1: Accuracy values of various algorithm.

6.2. Analysis of the non-binary models

Two different models were implemented: for the standard version of the algorithm, and for the modified one.

For each model, 6554 work data sets were used for training, and 655 work data sets for testing.

When considering the successful operation of the system subject to at least one true subject area model is successful in 79% for the standard algorithm. The arithmetic mean for the standard model is 47 (Table 2).

Table 2: Accuracy values of standard algorithm

	count of sets	%
All classified	118	18.02
At least one classified	520	79.39
No one classified	135	20.61
Summary	655	100
Average value	0.47	
Median value	0.5	

For the modified algorithm, more averaged values were obtained, the total number of positive classifications was 78%. The arithmetic mean for the modified model is 40% (Table 3).

Table 3: Accuracy values of modified algorithm

	count of sets	%
All classified	56	8.55
At least one classified	513	78.32
No one classified	142	21.68
Summary	655	100
Average value	0.40	
Median value	0.33	

The difference of arithmetic mean for the models is due to a more blurred result of the modified algorithm. Considering that is not necessary that for each set of data has been full compliance, it is a satisfactory result.

Figures 2 and 4 show the classification accuracy for standard (Figure 2) and modified (Figure 4) algorithms.

Probability values are grouped for clarity. It shows that in contrast to the standard model in which clearly shows the standard probability distribution, in the modified model blur occurs to low probability values.

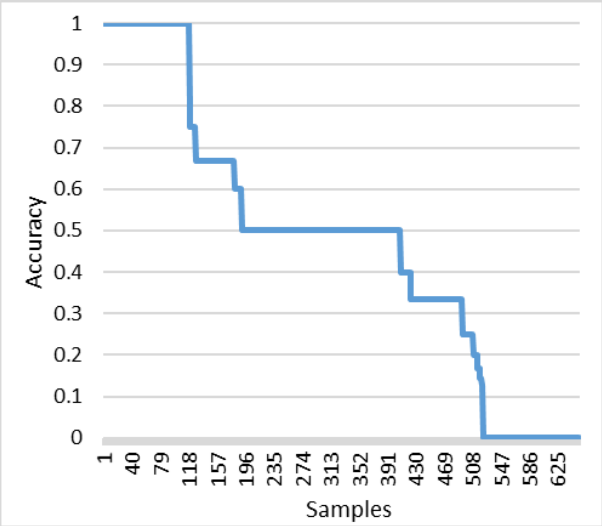


Figure 2: Sorted accuracy values of standard algorithm.

Figures 3 and 5 show the number of correctly defined subject areas for standard (Figure 3) and modified (Figure 5) algorithms. Count of classified values also are grouped for clarity.

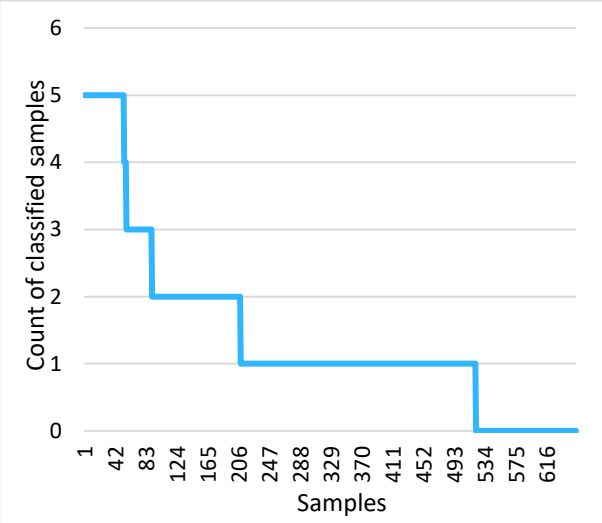


Figure 3: Sorted classified count of subject areas of standard algorithm.

For the standard model, there is a shift in the mean probability of 50%. The majority of correctly classified samples have only one correctly classified subject area.

Similarly, for a number of suitable domains, unlike the modified model in softer distribution - there are less complete correspondences (Figure 4,5). For the modified model, there is a shift in the average probability below 50%. The majority of correctly classified samples (as in the standard model) have only one correctly classified subject area.

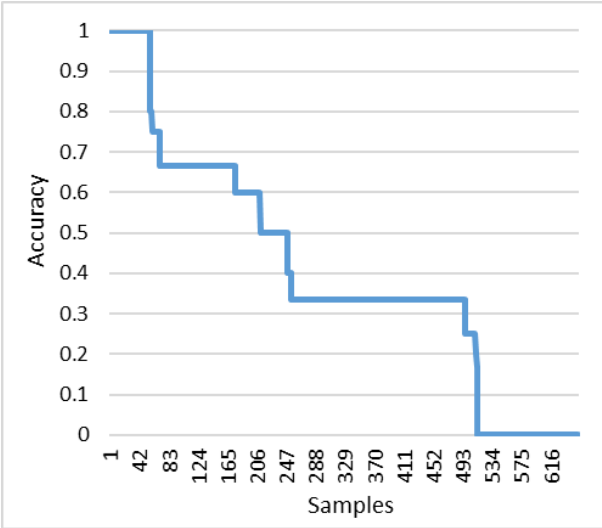


Figure 4: Sorted accuracy values of modified algorithm.

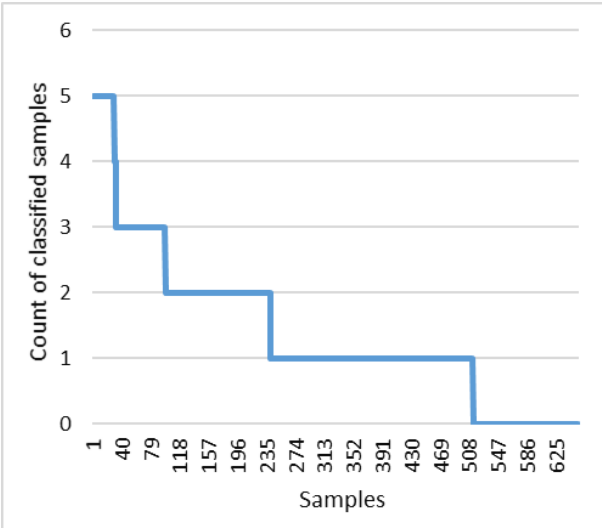


Figure 5: Sorted classified count of subject areas of modified algorithm.

7. Conclusions

The described classification algorithm is suitable for determining the subject area by work's keywords and key phrases. A model has been implemented that uses both versions of the algorithm and tested on real data. The obtained accuracy estimate of 79% allows one to determine with great accuracy not only the main, but also additional areas to which the work is only partially related. The standard version of the algorithm is more successful with large amounts of data, the modified version is more flexible and can work with new and incomplete data. The developed models can be used as binary classifiers with good accuracy. The developed algorithm can be used in databases of scientific papers and other types of text data.

References

[1] L. Breiman, J. Friedman, R. Olshen, C. Stone, Classification and regression trees, Wadsworth & Brooks, Pacific Grove, 1984.

[2] G. V. Kass, An exploratory technique for investigating large quantities of categorical data, Applied Statistics 29 (1980) 119–127.

[3] E. B. Hunt, J. Marin, P. J. Stone, Experiments in induction, Academic, New York, 1966.

[4] R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Machine learning. An artificial intelligence approach (1983) 463–482.

[5] J. R. Quinlan, Induction of decision trees, Machine Learning 1 (1986) 81–106.

[6] B. Boser, I. Guyon, V. Vapnik, A training algorithm for optimal margin classifiers, Proceedings of annual conference computational learning theory, ACM Press, Pittsburgh (1992) 144–152.

[7] C. Cortes, V. Vapnik, Support vector networks, Machine Learning 20 (1995) 273–297.

[8] J. Shawe-Taylor, P. L. Bartlett, R. C. Williamson, M. Anthony, Structural risk minimization over data-dependent hierarchies, IEEE Transactions on Information Theory 44 (1998) 1926–1940.

[9] J. Shawe-Taylor, N. Cristianini, Margin distribution and soft margin, Advances in large margin classifiers, MIT Press, Cambridge (2000) 349–358.

[10] T. Joachims, Text categorization with support vector machines: Learning with many relevant features, Proceedings of the European conference on machine learning, Springer, Berlin (1998) 137–142.

[11] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review 65 (1958) 386–408.

[12] J. Schmidhuber, Deep Learning in Neural Networks: An Overview, Neural Networks 61 (2015) 85–117.

[13] S. E. Dreyfus, Artificial neural networks, back propagation, and the Kelley-Bryson gradient procedure, Journal of Guidance, Control, and Dynamics 13 (1990) 926–928.

[14] E. Mizutani, S. E. Dreyfus, K. Nishio, On derivation of MLP backpropagation from the Kelley-Bryson optimal-control gradient formula and its application, IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium 2 (2000) 167–172.

[15] M. A. Fauzi, Automatic Complaint Classification System Using Classifier Ensembles, Telfor Journal 10 (2018) 123-128.

[16] D. Lewis, Naive Bayes at forty: the independence assumption in information retrieval, Proceedings of the 10th European Conference on Machine Learning, Springer, Berlin (1998) 4–15.

[17] A. McCallum, K. Nigam, A comparison of event models for Naive Bayes text classification, AAAI-98 Workshop on Learning for Text Categorization, AAAI Press, California (1998) 41–48.

[18] R. Lau, R. Rosenfeld, S. Roukos, Adaptive language modelling using the maximum entropy principle. Proceedings of the ARPA Human Language Technology Workshop, San Francisco (1993) 108–113.

[19] A. L. Berger, S. A. Della Pietra, V. J. Della Pietra, A maximum entropy approach to natural language processing, Computational Linguistics 22 (1996) 39–71.

[20] N. S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, The American Statistician 46 (1992) 175–185.

[21] T. K. Ho, Random Decision Forests, Proceedings of the 3rd International Conference on Document Analysis and Recognition 14–16, Montreal (1995) 278–282.

[22] T. K. Ho, The Random Subspace Method for Constructing Decision Forests, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (1998) 832–844, http://dx.doi.org/10.1109/34.709601.

[23] A. Ciapetti, R. Di Florio, L. Lomasto, G. Miscione, G. Ruggiero, D. Toti, NETHIC: A System for Automatic Text Classification using Neural Networks and Hierarchical Taxonomies, Proceedings of the 21st International Conference on Enterprise Information Systems 1 (2019) 296-306.

[24] G. Krishnalal, S. Rengarajan, K. Srinivasagan, A New Text Mining Approach Based on HMM-SVM for Web News Classification, International Journal of Computer Applications 1 (2010) 98-104. DOI. 10.5120/395-589

[25] L. E. Baum, T. Petrie, Statistical Inference for Probabilistic Functions of Finite State Markov Chains, The Annals of Mathematical Statistics 37 (2019) 1554–1563.

[26] L. E. Baum, G. R. Sell, Growth transformations for functions on manifolds, Pacific Journal of Mathematics 27 (1968) 211–227.

[27] M. I. Khaleel, I. I. Hmeidi, H. M. Najadat, An Automatic Text Classification System Based on Genetic Algorithm, Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on Social Informatics 31 (2016) 1–7.

[28] N. Medagoda, S. Shanmuganathan, Keywords based temporal sentiment analysis, 12th International Conference on Fuzzy Systems and Knowledge Discovery (2015) 1418-1425.

[29] R. Batool, A. M. Khattak, J. Maqbool, S. Lee, Precise tweet classification and sentiment analysis, 12th International Conference on Computer and Information Science (2013) 461-466.

[30] M. A. Gadamer, A. Horzyk, Semi-automatic contextual analysis and correction of texts by specialized linguistic graphs, AGH University of Science and Technology, 2019.

[31] L. A. Zadeh, Fuzzy sets, Information and Control 8 (1965) 338-353.

[32] P. Karczmarek, Selected problems of face recognition and decision-making theory, Wydawnictwo Politechniki Lubelskiej, 2018.

[33] The website for Elsevier B.V., Open database, https://www.scopus.com, [01.04.2021].

[34] M. Riedmiller, H. Braun, A direct adaptive method for faster backpropagation learning: the RPROP algorithm, Proceedings of the IEEE International Conference on Neural Networks 16 Piscataway (1993) 586-591.

[35] J Platt. Fast Training of Support Vector Machines Using Sequential Minimal Optimization, Advances in Kernel Methods: Support Vector Learning (1999) 185-208.

[36] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, K. R. K. Murthy, Improvements to Platt's SMO Algorithm for SVM Classifier Design, Neural Computation 13 (2001) 637-649, http://dx.doi.org/10.1162/089976601300014493.

[37] S. L. Salzberg. C4.5: Programs for Machine Learning by J. Ross Quinlan, Machine Learning 16, Morgan Kaufmann Publishers (1994) 235–240, http://dx.doi.org/10.1007/BF00993309.

[38] J. Shafer, R. Agrawal, M. Mehta, SPRINT: A scalable parallel classifier for data mining, VLDB, 2000.

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 151–158
Received: 21 April 2021
Accepted: 13 May 2021

Analysis of the possibilities of optimizing SQL queries

Analiza możliwości optymalizacji zapytań SQL

Piotr Rymarski*, Grzegorz Kozieł

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

Abstract

Most of today's web applications run on relational database systems. Communication with them is possible through statements written in Structured Query Language (SQL). This paper presents the most popular relational database management systems and describes common ways to optimize SQL queries. Using the research environment based on fragment of the imdb.com database, implementing OracleDb, MySQL, Microsoft SQL Server and PostgreSQL engines, a number of test scenarios were performed. The aim was to check the performance changes of SQL queries resulting from syntax modification while maintaining the result, the impact of database organization, indexing and advanced mechanisms aimed at increasing the efficiency of operations performed, delivered in the systems used. The tests were carried out using a proprietary application written in Java using the Hibernate framework.

Keywords: SQL; query; optimization; analysis

Streszczenie

Większość obecnie istniejących aplikacji internetowych działa w oparciu o relacyjne systemy baz danych. Komunikacja z nimi jest możliwa poprzez instrukcje zapisywane w Structured Query Language (SQL). Niniejsza publikacja prezentuje najbardziej popularne systemy do zarządzania relacyjnymi bazami danych oraz opisuje powszechne sposoby optymalizacji zapytań SQL. Wykorzystując środowisko badawcze, w którym zaimportowano część bazy danych serwisu imdb.com oraz silniki OracleDb, MySQL, Microsoft SQL Server i PostgreSQL wykonano szereg scenariuszy testowych. Celem było sprawdzenie zmiany wydajności zapytań SQL wynikających z modyfikacji składni przy zachowaniu rezultatu, wpływu organizacji bazy danych, indeksowania oraz zaawansowanych mechanizmów, mających na celu wzrost efektywności wykonywanych operacji, dostarczanych w wykorzystanych systemach. Testy zostały przeprowadzone przy pomocy autorskiej aplikacji napisanej w języku Java z wykorzystaniem szkieletu programistycznego Hibernate.

Słowa kluczowe: SQL; zapytanie; optymalizacja; analiza

1. Wstęp

Większość współczesnych aplikacji internetowych działa w oparciu o architekturę model-widok-kontroler. Widok jest odpowiedzialny za wygląd i zachowanie aplikacji po stronie użytkownika, oraz wysyłanie żądań i danych do aplikacji serwerowej (ang. backend). Choć niedostrzegalny przez odbiorcę końcowego, to właśnie backend dostarcza wymaganych danych. Serwer aplikacji odpowiada za zapis, przetwarzanie i odczyt danych z 'bazy danych. Jedną z najważniejszych cech działania aplikacji internetowej jest jej wydajność. Choć istnieje możliwość, aby część logiki aplikacji została przeniesiona do bazy, zazwyczaj nie jest to pożądane działanie, a baza danych służy tylko i wyłącznie do magazynowania danych. Aby umieścić, odczytać, zmodyfikować lub usunąć dane z bazy, wykorzystywane są specjalne instrukcje. W przypadku relacyjnych baz danych do komunikacji z bazą używa się instrukcji języka SQL (ang. Structured Query Language). Do manipulacji danymi w SQL używane są zapytania typu DML (ang. Data Manipulation Language) [5].

Podczas działania bazy danych, w oparciu o którą działają sesje inicjowane przez użytkowników, wykonywane są symultanicznie tysiące operacji SCAN (służących do pozyskiwania danych przy pomocy instrukcji SELECT) oraz DML. Każde niewłaściwie sformułowane zapytanie generuje opóźnienie spowodowane złożonością obliczeniową operacji. Im większa liczba operacji, tym większe obciążenie serwera bazy danych. Wynikiem czego może być długi czas oczekiwania, lub nawet awaria serwera [8]. Jak istotny jest wpływ składni zapytań SQL na ich wydajność? Przy jakich technologiach i wielkościach baz danych jest to najbardziej zauważal-

*Corresponding author
Email address: piotr.rymarski@pollub.edu.pl (P. Rymarski)

ne, a przy jakich marginalne? Niniejszy artykuł pozwala w pewnym stopniu odpowiedzieć na powyższe pytania, prezentując wyniki przeprowadzonych badań.

2. Wybrane technologie

Już w 1970 roku idea relacyjnych baz danych została opisana przez F. Codda [1] i jej założenia nie zmieniły się do dziś. Relacyjna baza danych używa konceptu połączonych ze sobą dwuwymiarowych tabel składających się z rzędów i kolumn. Kolumny reprezentują poszczególne atrybuty encji, w rzędach zaś zapisywane są kolejne rekordy, stanowiące oddzielne instancje opisywanego obiektu. Obecnie w czołówce silników bazodanowych [6], nie tylko w ujęciu relacyjnych baz danych, znajdują się kolejno:

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL

2.1. Oracle Database

Oracle Database (Oracle DBMS lub tylko Oracle) to wielomodelowy system zarządzania relacyjnymi bazami danych - RDBMS (ang. Relational Database Management System) wyprodukowany i dystrybuowany przez Oracle Corporation. Początki systemu datuje się już na 1977 rok, kiedy to L. Ellison wraz z kolegami utworzył SDL (ang. Software Development Laboratories).

Obecnie najnowszą, stabilną wersją Oracle DBMS jest 19c. Silnik jest dostępny w wielu wariantach, z których wszystkie, za wyjątkiem wersji Express, która nie może być wykorzystywana w warunkach produkcyjnych, wymagają zakupu licencji do zastosowań komercyjnych. Główną cechą OracleDB jest multiplatformowość. System obsługuje ponad 100 platform hardware'owych oraz ponad 20 protokołów sieciowych [7], co zapewnia bezpieczeństwo przy planowaniu aktualizacji i modyfikacji środowiska, w którym pracuje baza. Kolejną zaletą jest możliwość przeniesienia części logiki aplikacji bezpośrednio do serwera bazy danych. Wszystkie wersje silnika zawierają języki i interfejsy, które pozwalają programistom na dostęp i manipulację danymi bezpośrednio w bazie.

Silnik Oracle jest ceniony także za dostarczanie rozbudowanych systemów przywracania Oracle Recovery Manager (RMAN) [2]. RMAN pozwala na jednorazową konfigurację przywracania, zautomatyzowane zarządzanie kopiami zapasowymi i zarchiwizowanymi logami, wznawianie procesów tworzenia kopii zapasowych, ich przywracania oraz na testowanie tych procesów.

2.2. MySQL

MySQL to system zarządzania reelacyjnymi bazami danych stworzony w 1995 roku przez szwedzką firmę MySQLAB. W 2008 roku MySQLAB została nabyta przez Sun Microsystems, która od 2010 jest częścią Oracle Corporation. MySQL Server i jego biblioteki są dystrybuowane w dwóch zakresach licencyjnych. Można ich używać w oparciu o GPL 2 (General Public Use License version 2) lub inne licencje, nie zezwalające na bezpłatne komercyjne wykorzystywanie. System zyskał wielu zwolenników, na co wskazuje fakt użycia MySQL w serwisach takich jak: Amazon, Uber, Twitter, czy Netflix.

Pierwotnie założeniem MySQL było dodanie indeksowania danych, aby przyspieszyć zapytania dla serwera mSQL, poprzez użycie metod sekwencyjnego dostępu ISAM. Osiągnięto to dzięki zastosowaniu specjalnego algorytmu zarządzania danymi nazwanego MyISAM storage engine, co okazało się wielkim sukcesem. W pierwszych fazach rozwoju MySQL stawiano na zapewnienie jak największej szybkości przy ograniczeniu dostępnych funkcjonalności. Brakowało na przykład mechanizmu transakcji, w rezultacie odstraszało to potencjalnych użytkowników. Jednak z czasem i wraz ze zmianami właścicieli systemu sytuacja uległa diametralnej poprawie, jeśli chodzi o zapewniane funkcjonalności. Dziś MySQL jest czymś więcej niż systemem zarządzania bazami danych, zapewniającym szybkie wykonywanie zapytań. Jego rozwój ciągle trwa i w kolejnych wersjach dodawane są coraz to nowe udogodnienia i rozwiązania.

2.3. Microsoft SQL Server

Microsoft SQL Server to system zarządzania relacyjnymi bazami danych wspierany i rozwijany przez Microsoft. Jest platformą typu klient/server. Jego podstawową funkcją jest przechowywanie i zapewnianie danych wymaganych przez aplikacje, które mogą działać w obrębie tego samego urządzenia lub zdalnie przez sieć. Historia tego RDBMS zaczęła się wydaniem SQL Server v1.0, będącego projektem bazującym na Synbase SQL Server. Pierwsza wersja była owocem kooperacji Microsoft z Synbase Inc. oraz Ashton-Tate Corp. W ciągu kolejnych lat firmy zaniechały dalszej współpracy, a Synbase zmieniło nazwę swojego produktu, pozostawiając SQL Server na wyłączność Microsoft.

Obecną najnowszą wersją systemu jest Microsoft SQL Server 2019, choć produkt jest oficjalnie wspierany do pięciu wersji wstecz (Microsoft SQL Server 2012). System dostępny jest w wielu różnych wersjach z podziałem na licencje, np. Enterprise i Express oraz specyfikację użycia, np. Developer i Datawarehouse Appliance Edition [3]. Microsoft SQL Server zawiera szeroki zestaw narzędzi administracyjnych (SQLCMD, SQL Server Management Studio, Azure Data Studio), analitycznych (SQL Server Analysis Services) oraz Business Inteligence.

2.4. PostgreSQL

PostgreSQL (Postgres) to darmowy RDBMS dostępny na zasadach PostgreSQL License (open-source). System zapewnia podobny zakres rozwiązań jak w przypadku konkurencji. Został zaprojektowany, aby radzić sobie z różnymi środowiskami, od pojedynczych maszyn

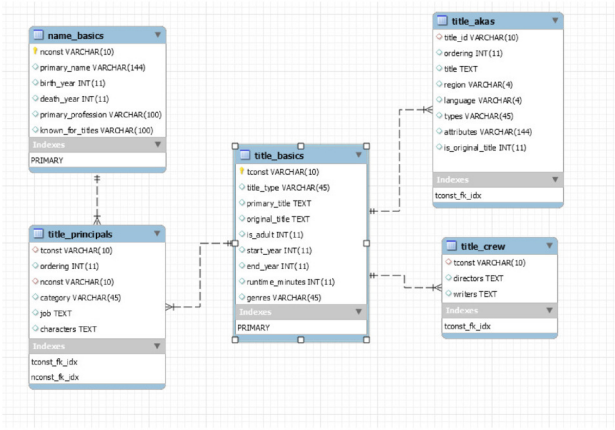
do hurtowni danych oraz serwisów internetowych z wieloma użytkownikami. W odróżnieniu od konkurencji, jego popularność gwałtownie rośnie i z kolejnymi latami przybywa entuzjastów tego rozwiązania.

Jedną z największych zalet silnika jest bardzo duże przystosowanie składni do oryginalnego SQL. W obecnej wersji Postgres spełnia 170 z 179 wymaganych funkcji standardu SQL:2016. Przy czym warto nadmienić, iż żaden z dostępnych RDBMS nie spełnia tego standardu w 100% [4].

Przy integracji z innymi systemami pomocny może być mechanizm FDW (ang. Foreign Data Wrappers), pozwalający na pozyskiwanie danych z zewnątrz (serwisy internetowe, system plików oraz inne bazy danych). Oznacza to, że zwykłe zapytania mogą używać tych źródeł pojedynczo lub dowolnie je łączyć i wykorzystywać i traktować je jak tabele zdefiniowane w bazie. Silnik zapewnia także programowanie proceduralne poprzez przechowywane funkcje i dyrektywę DO. Istotną funkcją PostgreSQL jest implementacja obiektowości poprzez dziedziczenie tabel. Pozwala to na uniknięcie pustych kolumn przy opisywaniu pokrewnych obiektów.

3. Metodyka badawcza

Do badania czasu wykonania zapytań wymagana była baza danych posiadająca tabele magazynujące duże liczby rekordów. Zdecydowano się wykorzystać dane udostępnione przez portal imdb.com [9]. Do niekomercyjnego użytku osobistego można pobrać 7 zestawów danych, spośród których do badania wybrano 5. Na Rysunku 1 przedstawiono diagram ERD bazy danych powstałej w wyniku importu wybranych zestawów.



Rysunek 1: Diagram ERD testowej bazy danych.

Istotna dla przebiegu badań była możliwość podglądu planu wykonania przygotowanych zapytań dla każdego z użytych systemów. Plan wykonania dostarcza wielu istotnych informacji o kolejnych etapach, koszcie i czasie wykonania operacji. Pozwala również na podgląd kolejnych etapów wykonania zapytania, a przede wszystkim faktu wykorzystania, bądź pominięcia istniejących indeksów.

W celach badawczych przygotowano aplikację pozwalającą na pomiar czasu wykonywania zapytań w kontrolowanych warunkach. Kod aplikacji napisano w języku Java z wykorzystaniem narzędzia Maven dla ułatwienia organizacji wymaganych bibliotek – zależności. Istotnym dla działania programu było zastosowanie wzorca programistycznego Hibernate z uwzględnieniem interfejsu Statistics. Pozwoliło to na prostą konfigurację i pomiar wymaganych wartości.

Wszystkie badania zostały przeprowadzone w takim samym środowisku testowym, tj. na tym samym komputerze (Tabela 1), za pośrednictwem lokalnych baz danych oraz przy wykorzystaniu tej samej aplikacji testowej z analogiczną konfiguracją. Aby zapewnić jak najdokładniejsze pomiary w momencie prowadzenia badań na komputerze uruchomione były wyłącznie:

- system operacyjny
- aplikacja testowa
- badany silnik bazy danych

Komputer był odłączony od sieci, a pozostałe aplikacje, przede wszystkim oprogramowanie antywirusowe oraz zapora sieciowa, były wyłączone.

Tabela 1: Specyfikacja komputera, na którym przeprowadzono badania

System operacyjny	Windows 10 Home 64-bit
Procesor	AMD Ryzen 5 2600
Pamięć RAM	16 GB 3000MHz CL15
Dysk	Samsung SSD 970 EVO 1TB

Przed wykonaniem badania przygotowano szereg scenariuszy, z których każdy wykonano dziesięciokrotnie. Po odrzuceniu wyników skrajnych obliczono średni czas wykonania każdego z zapytań dla wszystkich badanych systemów i przedstawiono wyniki badań na wykresach.

4. Wyniki badań

Poniżej przedstawiono wybrane scenariusze badawcze wraz z diagramami prezentującymi ich wyniki.

4.1. Wpływ ograniczania ilości zwracanych danych na wydajność zapytania

Pierwszy scenariusz badawczy polegał na sprawdzeniu wpływu ograniczenia ilości pozyskiwanych jednorazowo danych na czas wykonania zapytania. Badana baza danych zawierała tabele magazynujące miliony rekordów (Tabela 2), więc spodziewano się znacznego spadku czasu wykonania zapytań. Omawiany scenariusz wykonano na dwa sposoby: ograniczając liczbę zwracanych kolumn (scenariusz 1.a) oraz ograniczając liczbę zwracanych rekordów (scenariusz 1.b).

Tabela 2: Liczebność rekordów każdej z zaimportowanych tabel

Tabela	Liczba rekordów
name_basics	10 628 431
title_basics	7 498 564
title_akas	24 757 132
title_principals	42 496 909
title_crew	7 495 387

Ograniczenie ilości zwracanych danych można rozpatrywać w 2 aspektach:

a) Zmniejszenia liczby zwracanych kolumn;

b) Zmniejszenia liczby zwracanych rekordów.

Tabela 3: Czas wykonania zapytań w scenariuszu 1.a

Silnik bazodanowy	Czas wykonania zapytania 1. [s]	Czas wykonania zapytania 2. [s]
Oracle	0,14	0,08
MySQL	7,59	6,63
SQL Server	0,27	0,04
PostgreSQL	4,86	1,05

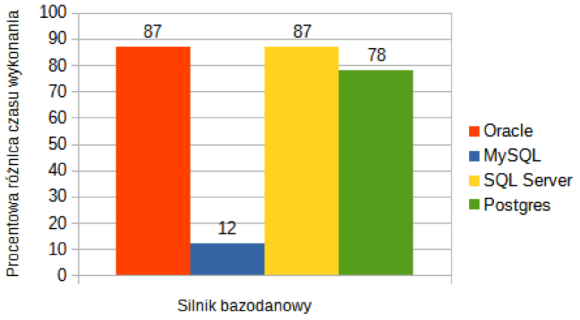
Tabela 3 przedstawia zestawienie czasu wykonania zapytań (Listing 1) w scenariuszu 1.a. Aby jednoznacznie zobrazować efektywność każdego badanego zabiegu optymalizacyjnego, wszystkie wykresy przedstawione dalej w artykule pokazują procentową różnicę pomiędzy średnim czasem wykonania zapytań z uwzględnieniem wszystkich badanych silników bazodanowych. Owa różnica może być interpretowana jako procentowy wzrost wydajności zapytania (chyba, że na wykresie zaznaczono inaczej).

Zdecydowano się na takie rozwiązanie, ponieważ różnice czasu wykonania zapytań pomiędzy pomiarami w przypadku badanych silników były na tyle znaczące, że wynikowe wykresy zawierające dokładne otrzymane wyniki w mikrosekundach były nieczytelne. W przypadku MySQL czas wykonania zapytania potrafił ponad tysiąckrotnie przekraczać wyniki otrzymane z konkurencyjnych silników.

Listing 1: Zapytania użyte w scenariuszu 1.a

```
--wszystkie kolumy
SELECT * FROM title_basics
WHERE start_year = 2010;

--tylko 3 wybrane kolumy
SELECT tconst, primary_title, start_year
FROM title_basics
WHERE start_year = 2010;
```

W scenariuszu 1.a zbadano wpływ zmiany liczby zwracanych kolumn ze wszystkich (9) do tylko 3 wybranych: tconst, primary_title i start_year. Wynik badania zobrazowano na Rysunku 2.

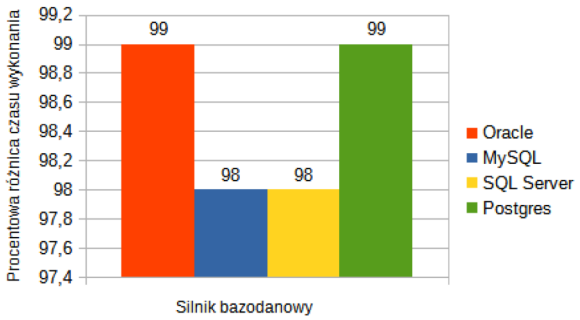


Rysunek 2: Diagram przedstawiający procentowy spadek czasu wykonania przy ograniczeniu liczby zwracanych kolumn.

W tym przypadku scenariusza 1.b zbadano wpływ zmniejszenia liczby zwracanych rekordów (z 76344 do 100 pierwszych) na czas wykonania zapytania (Rysunek 3). Warto zaznaczyć, że w celu osiągnięcia tego samego efektu wymagane było użycie odmiennej składni dla wybranych silników (Listing 2).

Listing 2: Zapytania użyte w scenariuszu 1.b

```
--wszystkie rekordy
SELECT tconst FROM title_basics
WHERE start_year = 2010;

--tylko 100 rekordów
--ORACLE
SELECT  tconst FROM title_basics
WHERE start_year = 2010 AND ROWNUM <= 100;

--MySQL i PostgreSQL
SELECT tconst FROM title_basics
WHERE start_year = 2010 LIMIT 100;

--SQL Server
SELECT TOP 100 tconst FROM title_basics
WHERE start_year = 2010;
```



Rysunek 3: Diagram przedstawiający procentowy spadek czasu wykonania przy ograniczeniu liczby zwracanych rekordów.

Jak łatwo zauważyć, ograniczenie zestawu zwracanych danych w obu przypadkach znacząco wpływa na czas wykonania zapytania.

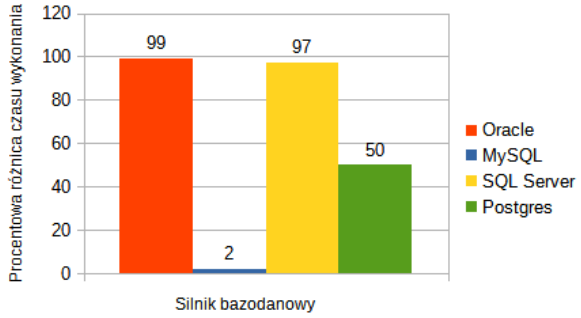
4.2. Optymalizacja klauzuli UNION

W scenariuszu 2 zbadano wpływ użycia klauzuli UNION ALL w miejsce klauzuli UNION na czas wykonania zapytania (Listing 3). Należy zauważyć jednak, iż w celu osiągnięcia identycznego zestawu danych w obu przypadkach nie może on zawierać duplikatów.

Listing 3: Zapytania użyte w scenariuszu 2

```
--UNION
SELECT tconst FROM title_basics
WHERE start_year = 1999
UNION
SELECT tconst FROM title_basics
WHERE start_year = 2000

--UNION ALL
SELECT tconst FROM title_basics
WHERE start_year = 1999
UNION ALL
SELECT tconst FROM title_basics
WHERE start_year = 2000;
```

Rysunek 4 przedstawia wynik przeprowadzonego badania. Wykonana operacja pozwoliła na zaoszczędzenie znacznej ilości czasu. Dzieje się tak, ponieważ operacja UNION ALL nie wykonuje niepotrzebnego filtrowania w poszukiwaniu duplikatów.



Rysunek 4: Diagram przedstawiający procentowy spadek czasu wykonania przy zastosowaniu klauzuli UNION ALL.

4.3. Porównanie wydajności klauzuli filtrującej z konstrukcją JOIN

Rysunek 5 przedstawia wyniki badań przeprowadzonych według scenariusza testowego 3, w którym sprawdzono wpływ zastąpienia klauzuli łączącej JOIN przy pomocy instrukcji WHERE (Listing 4).

Listing 4: Zapytania użyte w scenariuszu 3

```
--INNER JOIN
SELECT nb.primary_name
FROM name_basics nb
JOIN title_principals tp
ON nb.nconst = tp.nconst
WHERE tp.category = 'director'

--WHERE
SELECT nb.primary_name
FROM name_basics nb, title_principals tp
WHERE nb.nconst = tp.nconst
AND tp.category = 'director';
```

We wszystkich badanych systemach, z wyjątkiem Microsoft SQL Server, dla którego wystąpił spadek wydajności, operacja została potraktowana jako łączenie wewnętrzne.

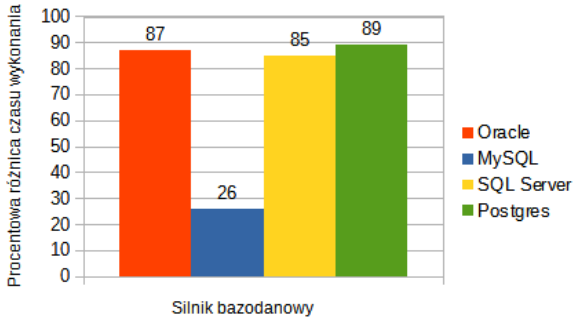
Rysunek 5: Diagram przedstawiający procentowy wzrost czasu wykonania przy zastąpieniu konstrukcji JOIN klauzulą WHERE.

4.4. Optymalizacja klauzuli grupującej

W scenariuszu 4 zbadano wpływ lokowania instrukcji filtrujących w klauzuli grupującej z dyrektywą HAVING (Listing 5). Rysunek 6 przedstawia procentowy wzrost wydajności po wydzieleniu filtrowania do klauzuli WHERE.

Listing 5: Zapytania użyte w scenariuszu 4

```
--Filtrowanie wyłącznie w klauzuli HAVING
SELECT birth_year, COUNT(nconst) AS people_count
FROM name_basics GROUP BY birth_year
HAVING birth_year > 2000 AND COUNT(nconst) > 50;

--Filtrowanie przeniesione do klauzuli WHERE
SELECT birth_year, COUNT(nconst) AS people_count
FROM name_basics WHERE birth_year > 2000
GROUP BY birth_year HAVING COUNT(nconst) > 50;
```



Rysunek 6: Diagram przedstawiający procentowy spadek czasu wykonania zapytania po przeniesieniu filtrowania do klauzuli WHERE.

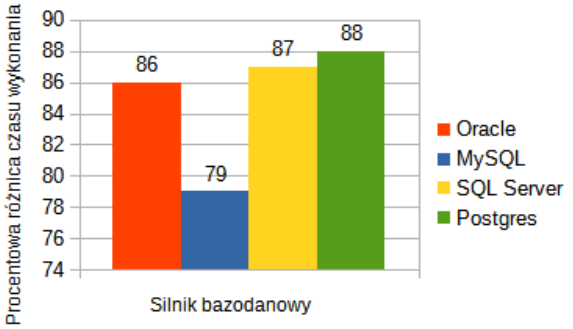
4.5. Zbadanie wydajności operacji full index scan w porównaniu do full table scan

Operacja full index scan ma miejsce wtedy, gdy wszystkie wymagane w zapytaniu kolumny są zdefiniowane wewnątrz indeksu. W takim przypadku nie ma potrzeby odwoływania się do oryginalnej tabeli, a operacja dodatkowo przeprowadzona jest na posortowanym zestawie danych. Zgodnie z założeniami wykonanie zapytania (Listing 6) po tym jak dodano indeks na kolumnie title_basics.start_year prze-

biegło w znacznie krótszym czasie (Rysunek 7).

Listing 6: Zapytanie użyte w scenariuszu 5

```
SELECT start_year, tconst
FROM title_basics
WHERE start_year > 2010;
```



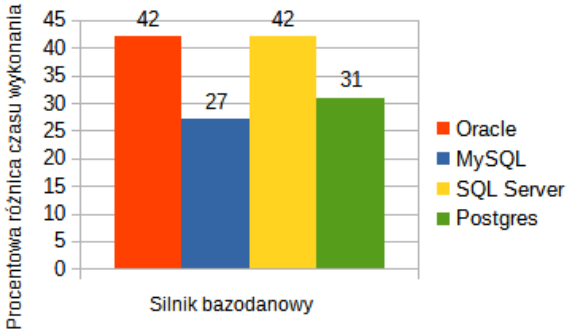
Rysunek 7: Diagram przedstawiający procentowy spadek czasu wykonania zapytania po dodaniu indeksu na filtrowaną kolumnę.

4.6. Sprawdzenie wzrostu wydajności przy pozbyciu się redundantnego sortowania

Domyślnie dane w indeksie posortowane są rosnąco, więc pominięcie dodatkowego sortowania w zapytaniu nie powinno wpłynąć na jego rezultat przy jednoczesnej poprawie wydajności. Aby to sprawdzić porównano czas wykonania zapytania z poprzedniego scenariusza z zapytaniem przedstawionym na Listingu 7. Rezultat badania obrazuje Rysunek 8.

Listing 7: Zapytanie użyte w scenariuszu 6

```
SELECT start_year, tconst
FROM title_basics
WHERE start_year > 2010
ORDER BY start_year;
```



Rysunek 8: Diagram przedstawiający procentowy wzrost czasu wykonania po dodaniu redundantnego sortowania po indeksowanej kolumnie.

4.7. Zbadanie wpływu indeksu na zapytania z filtrowaniem przy użyciu operatora SARGable i Non-SARGable

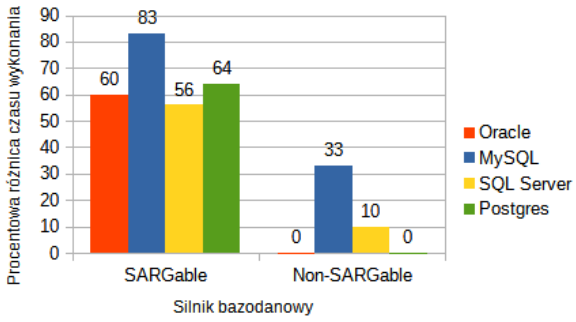
W SQL można wyróżnić 2 typy operatorów porównania:

- SARGable (operatory, dla których serwer może użyć indeksu, mianowicie: =, ¿, ¡, ¿=, ¡=, IN, BETWEEN oraz LIKE, ale tylko w kwestii pasujących przedrostków);
- Non-SARGable (operatory: NOT, NOT IN, ¡¿, LIKE (argument zaczyna się od „%”), a także wszystkie funkcje wykonujące kalkulacje, zmiany oraz konwersje typów na kolumnach tabeli).

W bieżącym scenariuszu zbadano wpływ indeksowania na zapytania zawierającego oba rodzaje operatorów. Aby sprawdzić jak zmieni się czas wykonania dla zapytań zawierających każdy z wymienionych typów operatorów porównania zastosowano parę analogicznych zapytań z Listingu 8 przed oraz po dodaniu indeksu na kolumnie title_basics.is_adult.

Listing 8: Zapytania użyte w scenariuszu 7

```
--SARGable
SELECT primary_title
FROM title_basics
WHERE is_adult = 1
AND start_year > 2010;

--Non-SARGable
SELECT primary_title
FROM title_basics
WHERE is_adult <> 0
AND start_year > 2010;
```



Rysunek 9: Diagram przedstawiający procentowy spadek czasu wykonania po dodaniu indeksu dla funkcji filtrujących typu SARGable i Non-SARGable.

Jednoznaczny wzrost wydajności można zaobserwować dla pierwszego z zapytań, wykorzystującego operator typu SARGable (Rysunek 9). Warto także wspomnieć, że dodanie indeksu miało pozytywny wpływ na czas wykonania drugiego zapytania w środowiskach MySQL oraz SQL Server.

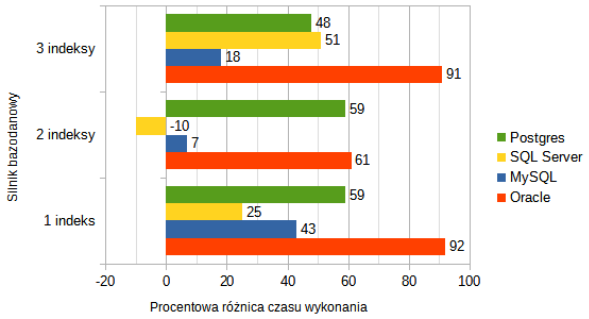
4.8. Indeksowanie a DML

Indeksy odgrywają znaczącą rolę nie tylko w przypadku skanów tabeli, ale także wpływają na wydajność instrukcji DML. Aby to sprawdzić wykonano sekwencję instrukcji DML (Listing 9) kontrolnie przed oraz kolejno po dodaniu indeksów na kolumnach primary_name, birth_year, primary_profession tabeli name_basics.

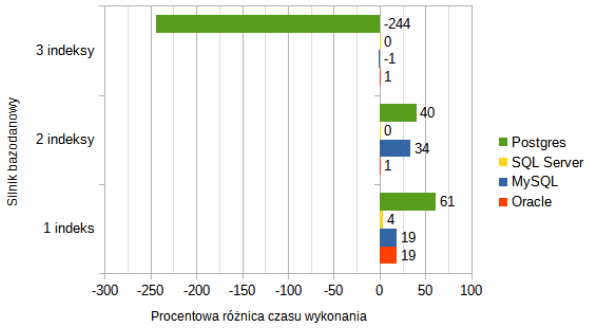
Listing 9: Zapytania użyte w scenariuszu 8

```
--INSERT
INSERT INTO name_basics(
  nconst,
  primary_name,
  birth_year,
  primary_profession
)
VALUES (
  'nm9999999',
  'Unikalne Nazwisko',
  1990,
  'actor'
);

--UPDATE
UPDATE name_basics
SET death_year = 2021
WHERE nconst= 'nm9999999';

--DELETE
DELETE FROM name_basics
WHERE nconst= 'nm9999999';
```

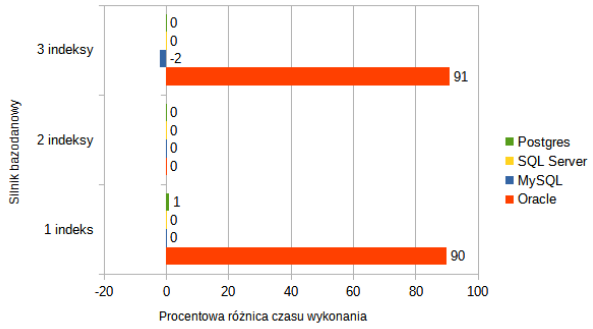
Rysunki 10, 11 i 12 przedstawiają kolejno wyniki badań wpływu indeksowania na operacje INSERT, UPDATE i DELETE.



Rysunek 10: Diagram przedstawiający procentowy spadek czasu wykonania po dodaniu kolejnych indeksów dla operacji INSERT.



Rysunek 11: Diagram przedstawiający procentowy spadek czasu wykonania po dodaniu kolejnych indeksów dla operacji UPDATE.



Rysunek 12: Diagram przedstawiający procentowy spadek czasu wykonania po dodaniu kolejnych indeksów dla operacji DELETE.

Wbrew oczekiwaniom, dodanie indeksów spowodowało głównie wzrost wydajności wykonanych operacji manipulacji danymi.

5. Wnioski

Już podczas przeglądu literatury dało się zauważyć, iż pomimo wspólnego przeznaczenia badanych systemów, istnieją pomiędzy nimi różnice w sposobie konfiguracji, używanym dialekcie SQL oraz czasie wykonania zapytań, które w odczuwalnym stopniu skomplikowały proces badawczy.

Przeprowadzone badania miały na celu wyróżnienie oraz sprawdzenie możliwości optymalizacji zapytań SQL. Nie wszystkie sposoby znalezione w literaturze oraz Internecie znalazły jednak zastosowanie, gdyż w wielu przypadkach optymalizatory badanych systemów wykonywały poprawnie zapytania umyślnie zapisane w nieoptymalny sposób. Przykładem takiej sytuacji było wykonanie optymalnego łączenia wewnętrznego tabel nawet, kiedy do ich łączenia użyto klauzuli WHERE.

Otrzymane wyniki jednoznacznie wskazały ograniczanie zestawu zwracanych danych (kolumn i rekordów) jako najlepszy sposób na zwiększenie wydajności zapytań. Praktykę ograniczania liczebności zbioru danych stosuje się m.in. przy paginacji stosowanej podczas wyświetlania danych. Aplikacje zazwyczaj nie pobierają całego zestawu danych, gdyż jest to bardzo niewydajne. Zamiast tego pobiera się tylko pewien zakres z posortowanych wcześniej danych. Kolejnym ważnym sposobem optymalizacji SQL jest używanie odpowiednich operatorów wewnątrz klauzul filtrujących oraz wykluczenie filtrowanych kolumn z wykonywanych działań. Dzięki temu silnik bazy danych będzie mógł użyć wcześniej utworzonych indeksów. Scenariusze badawcze oparte o wykorzystanie indeksów w znacznym stopniu pokryły się z oczekiwaniami, przynosząc znaczny wzrost wydajności.

Według informacji zebranych w literaturze, liczba indeksów powinna mieć jednoznacznie negatywny wpływ na czas wykonywania instrukcji DML, co nie sprawdziło się podczas procesu badawczego. W badaniu użyto indeksów zbudowanych na pojedynczych kolumnach, więc dostosowywanie indeksu do zmian w oryginalnej tabe-

li miało mniej znaczący wpływ niż korzyści wynikające z użycia indeksów w klauzulach filtrujących. Może to być związane z liczbą kolumn w klauzulach filtrujących w przypadku UPDATE i DELETE oraz złożonością indeksów, które muszą zostać zaktualizowane.

Ze względu na otrzymane rezultaty świadczące o podniesieniu wydajności w wielu scenariuszach badawczych, można jednoznacznie stwierdzić, iż optymalizacja SQL jest możliwa oraz pożądana. Warto także podkreślić fakt, że przy konstrukcji zapytań niezastąpiona jest wiedza na temat bazy danych, przede wszystkim na temat istniejących indeksów, oraz dokładne sprecyzowanie zakresu danych wynikowych.

Literatura

[1] R. Greenwald, R. Stackowiak, J. Stern, Oracle Essentials, Fifth Edition, O'Reilly Media, Sebastopol, 2013.

[2] P. Muryjas, M. Skublewska-Paszkowska, D. Gutek, Współczesne Technologie Informatyczne. Eksploatacja baz danych, Politechnika Lubelska, 2011.

[3] Editions and supported features of SQL Server, https://docs.microsoft.com/en-us/sql/sql-server/editions-and-components-of-sql-server-version-15?view=sql-server-ver15, [29.12.2020].

[4] What is PostgreSQL, https://www.postgresql.org/about/, [29.12.2020].

[5] What is SQL, https://www.infoworld.com/article/3219795/what-is-sql-the-lingua-franca-of-data-analysis.html, [02.01.2021].

[6] DB-Engines Ranking, https://db-engines.com/en/ranking, [02.12.2020].

[7] Oracle Database Features, https://docs.oracle.com/cd/B19306˙01/server.102/b14220/intro.html, [29.12.2020].

[8] Why Do Databases Crash, https://www.zmanda.com/blog/why-do-databases-crash-and-what-to-do-about-it/, [21.01.2021].

[9] IMDb Datasets, https://www.imdb.com/interfaces/, [02.12.2020].

JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 159–164
Received: 27 April 2021
Accepted: 16 May 2021

Comparison of lightweight frameworks for Java by analyzing proprietary web applications

Porównanie lekkich szkieletów dla języka Java poprzez analizę autorskich aplikacji internetowych

Marek Pucek, Michał Błaszczyk*, Piotr Kopniak

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

Abstract

There are many frameworks available in the IT environment that differ in performance, security, complexity, and speed. The user who wants to start working with the selected framework should know whether it will meet the architectural requirements and business assumptions. The aim of this work is to compare the performance and complexity of web applications built using different lightweight frameworks for the Java language. Spring Boot, Micronaut, Quarkus and Javalin frameworks will be compared. At the beginning of the work, the main problems related to the creation of modern applications were discussed. In the following sections, basic analysis of the complexity of the syntax and conventions of the selected skeletons was performed. Then, experiments were conducted to compare performance - response and build times and memory consumption during application development and use. A wide cross-section of efficiency has been obtained in selected lightweight framework usages. The prepared comparison can be used to select the appropriate framework for the project.

Keywords: Spring Boot; Micronaut; Quarkus; Javalin

Streszczenie

W środowisku IT dostępnych jest wiele szkieletów, które różnią się między sobą wydajnością, bezpieczeństwem, złożonością czy szybkością działania. Użytkownik, chcący zacząć pracę z wybranym szkieletem powinien wiedzieć, czy sprosta on wymaganiom architektonicznym oraz założeniom biznesowym. Celem niniejszej pracy jest porównanie wydajności oraz złożoności aplikacji internetowych zbudowanych z wykorzystaniem różnych lekkich szkieletów dla języka Java. Porównane zostaną szkielety Spring Boot, Micronaut, Quarkus oraz Javalin. Na początku pracy omówione zostały główne problemy związane z tworzeniem współczesnych aplikacji. W kolejnych częściach dokonano podstawowej analizy złożoności składni i konwencji wybranych szkieletów. Następnie wykonano eksperymenty mające na celu porównanie wydajności - czasy oraz zużycie pamięci podczas tworzenia i użytkowania aplikacji. Uzyskano szeroki przekrój efektywności w wybranych zastosowaniach lekkich szkieletów. Sporządzone porównanie może być wykorzystane do dobrania odpowiedniego szkieletu do projektu.

Słowa kluczowe: Spring Boot; Micronaut; Quarkus; Javalin

*Corresponding author

Email address: michal.blaszczyk@pollub.edu.pl (M. Błaszczyk)

1. Wstęp

Aplikacje internetowe w dzisiejszych czasach stanowią bardzo skuteczne rozwiązanie dla szerokiej gamy zadań biznesowych. Dzięki łatwości implementacji oraz możliwości dodawania kolejnych funkcjonalności stają się coraz częstszym wyborem wśród większości firm świadczących usługi swoim klientom. Zalety te powodują stopniowe wypieranie rozwiązań tradycyjnych takich jak aplikacje okienkowe.

W dzisiejszych czasach wybór szkieletu programistycznego dla aplikacji internetowej opartej o język Java wydaje się dosyć prosty - największą popularność od lat ma Spring Boot. Jednak istnieje jeszcze wiele innych lekkich szkieletów, których przeznaczeniem są zastosowania serwera aplikacji internetowej [1]. Celem niniejszej pracy jest porównanie oraz wskazanie wad i zalet kolejnych trzech - Micronauta, Javalina oraz Quarkusa.

Zestawienie zostało przedstawione z dwóch różnych punktów widzenia: programisty oraz użytkownika aplikacji. W pierwszym przypadku wzięte zostały pod uwagę aspekty takie jak analiza złożoności i wydajności kodu, konwencji przyjętych w wybranych szkieletach oraz sposobu komunikacji pomiędzy komponentami aplikacji. Dodatkowo porównano czasy kompilacji i uruchamiania aplikacji. Drugi punkt widzenia - użytkownika aplikacji - obejmuje głównie porównanie czasów przesyłania, pobierania oraz przetwarzania danych na serwerze.

2. Szkielety dla języka Java

2.1. Spring Boot

Spring Boot to szkielet aplikacyjny o otwartym kodzie źródłowym służący do tworzenia całej gamy aplikacji Java i umożliwiający uproszczenie procesu wytwarzania oprogramowania. Ma on na celu skrócenie długości kodu i zapewnienie najłatwiejszego sposobu tworzenia

aplikacji internetowych. Dzięki wbudowanym adnotacjom oraz autokonfiguracji Spring Boot pomaga stworzyć w krótkim czasie samodzielną aplikację z mniejszą lub prawie zerową konfiguracją. Oprócz tych podstawowych zalet można również wymienić inne cechy Spring Boota [2-3]:

- Umożliwia bardzo łatwe tworzenie aplikacji z użyciem języków Java lub Groovy.
- Skraca czas projektowania i zwiększa produktywność.
- Pozwala uniknąć pisania wielu linii standardowego kodu, adnotacji i konfiguracji XML.
- Bardzo łatwo jest zintegrować aplikację Spring Boot z ekosystemem Spring, w którego skład wchodzą Spring JDBC, Spring ORM, Spring Data, Spring Security, itp.
- Zapewnia wbudowane serwery HTTP, takie jak Tomcat, Jetty, itp., aby bardzo łatwo tworzyć i testować aplikacje internetowe [4].

2.2. Micronaut

Zwiększenie nacisku na tworzenie aplikacji opartych o mikroserwisy oraz rozwiązania chmurowe były głównym powodem dla stworzenia szkieletu Micronaut. W jego składni widać wiele podobieństw do szkieletu Spring Boot, takich jak podobne konwencje nazw czy praktycznie identyczne mechanizmy adnotacji. Jednak główną różnicą między Micronautem a Spring Bootem jest mechanizm wstrzykiwania zależności już w czasie kompilacji, twórcy Micronauta twierdzą, że dzięki temu udało im się uzyskać niższe czasy uruchamiania aplikacji. Omawiany szkielet wspiera trzy języki programowania Java, Kotlin oraz Groovy. Ze względu na silną optymalizację, twórcy szkieletu Micronaut postawili duży nacisk na integrację, z równie młodą technologią jaką jest GraalVM – uniwersalna lekka maszyna wirtualna [5-6].

2.3. Javalin

Javalin jest lekkim szkieletem dla języka Java oraz Kotlin. Podczas jego tworzenia położono duży nacisk na jak największe uproszczenie kodu. W przeciwieństwie do pozostałych omawianych szkieletów Javalin nie posiada adnotacji oraz mechanizmu refleksji, ma to zapewnić dużą szybkość działania aplikacji przy jednoczesnym oszczędzaniu pamięci i zasobów procesora. W celu uruchomienia aplikacji należy utworzyć obiekt o typie Javalin i wywołać metodę uruchamiającą ją na wybranym porcie. Konfigurację oraz pozostałe aspekty aplikacji wprowadza się edytując stworzony obiekt [7].

2.4. Quarkus

Podobnie jak w przypadku Micronauta szkielet Quarkus jest odpowiedzią na rosnące zapotrzebowanie na aplikacje oparte o mikroserwisy oraz aplikacje chmurowe. Został opracowany przez RedHat i jest przeznaczony do tworzenia aplikacji, które mają być wdrażane w chmurze, natywnie obsługując Kubernetes i umożliwiając łatwe budowanie lekkich aplikacji pod względem dostarczanego rozmiaru i wykorzystania pamięci [8-9]. Pozostałymi, ale nie mniej ważnymi zaletami Quarkusa są również:

- ujednolicona konfiguracja przechowywana w jednym pliku,
- szybkie przeładowanie konfiguracji – wszystkie zmiany w konfiguracji aplikują się wraz z odświeżeniem uruchomionej aplikacji,
- uproszczony kod dla 80% typowych zastosowań, elastyczny dla 20% [10],
- łatwe generowanie natywnych plików wykonywalnych.

3. Cel i obiekt badań

Celem badań jest analiza kodu i wydajności czterech aplikacji stworzonych za pomocą omawianych szkieletów aplikacyjnych, a następnie wskazanie różnic, wad oraz zalet poszczególnych frameworków.

4. Przegląd narzędzi do budowy aplikacji

Do napisania kolejnych aplikacji użyto następujących narzędzi:

- JDK 11 – zestaw narzędzi programistycznych Javy,
- PostgreSQL – baza danych użyta w testowej aplikacji internetowej,
- Maven – narzędzie automatyzujące budowę aplikacji – użytą w przypadku Spring Boota oraz Quarkusa,
- Gradle – narzędzie automatyzujące budowę aplikacji – użytą w przypadku Micronauta i Javalina,
- Tomcat – serwer WWW i kontener aplikacyjny – wykorzystany do tworzenia większości aplikacji oraz jako środowisko produkcyjne,
- Jetty – serwer aplikacyjny – używany przy tworzeniu aplikacji Javalina,
- JMeter – narzędzie testowe wykorzystane przy testach punktów dostępowych,
- Jenkins – narzędzie do automatycznego budowania i uruchamiania projektów.

5. Zaimplementowane aplikacje

Wszystkie z zaimplementowanych aplikacji posiadają podobną architekturę. Przykładowymi klasami na podstawie, których można dokonać analizy są:

- klasa główna aplikacji,
- domeny,
- kontroler,
- repozytorium.

Klasa główna w przypadku Spring Boota, Micronauta oraz Quarkusa wygląda bardzo podobnie, składa się ona z adnotacji specyficznej dla danego szkieletu oraz jednej statycznej metody uruchamiającej aplikacje. W przypadku Javalina główna klasa jest o wiele bardziej rozbudowana. W metodzie main znajduje się inicjalizacja obiektu o typie Javalin oraz z jego konfiguracja.

W przypadku klas encyjnych (domenowych), ponownie w szkieletach Spring Boot, Micronaut oraz Quarkus kod klas wygląda bardzo podobnie. Ponad nagłówkami klas znajdują się adnotacje wskazujące nazwy tabel w bazie danych odnoszących się do danej domeny. Nad poszczególnymi polami znajdują się ad-

notacje zawierające nazwy kolumn oraz właściwości takie jak np. id lub relacje między tabelami. W przypadku Javalina domeny są jedynie klasami ze wszystkimi polami domeny. Częścią wspólną wszystkich klas są dwa konstruktory, jeden pusty – wykorzystywany do serializacji, drugi posiadający pola klasy.

Kontrolery w aplikacjach opartych o szkielety Spring Boot, Micronaut oraz Quarkus także są bardzo do siebie podobne. Różnice polegają jedynie na nazwach adnotacji. Kontroler aplikacji opartej o szkielet Javalin musi posiadać obiekt o typie Javalin utworzony w klasie głównej, następnie przy pomocy metod wykonywanych na tym obiekcie dodawane są kolejne punkty dostępowe.

Ze względu na to, że aplikacje nie posiadają złożonych funkcjonalności operujących na bazie danych, w przypadku aplikacji opartych o szkielety Spring Boot, Micronaut oraz Quarkus możliwe było zastosowanie stylu architektonicznego CRUD. Umożliwia to używanie domyślnych metod operujących na bazie danych. Możliwe jest także tworzenie własnych bardziej skomplikowanych metod używając sugerowanej przez szkielety składni - należy zauważyć, że składnie różnią się między szkieletami. Dzięki temu możliwe było zaimplementowanie metody wyszukującej obiekty posiadające swoje id w podanym przedziale. Szkielet Javalin nie posiada wymienionych wyżej mechanizmów, z tego powodu należało wykorzystać zapytania SQL uzupełniane o wymagane parametry.

6. Metodyki testów

Badanie stara się odpowiedzieć na pytanie: który z frameworków jest najlepszym wyborem przy tworzeniu aplikacji internetowej. Istnieje wiele metod porównywania szkieletów aplikacyjnych. Jednym z nich są testy wydajnościowe przeprowadzone na aplikacji stworzonej z użyciem badanych frameworków. Aby testy były jeszcze bardziej wiarygodne zdecydowano się na przeprowadzenie ich na dwóch maszynach z różnymi systemami operacyjnymi. Ich specyfikacje przedstawiono w Tabeli 1 i 2.

Tabela 1: Parametry maszyny 1

Maszyna nr 1	
Procesor	Intel Core i7 3770K, 4 rdzenie, 8 wątków, taktowanie 3,50 GHz (CPU Mark – 6422)
Pamięć RAM	24,0 GB
Rodzaj dysku	SSD
System operacyjny	Manjaro Linux 20.1.2

Tabela 2: Parametry maszyny 2

Maszyna nr 2	
Procesor	AMD Ryzen 5 2600, 6 rdzeni, 12 wątków, taktowanie 3,40 GHz (CPU Mark – 13219)
Pamięć RAM	32,0 GB
Rodzaj dysku	SSD
System operacyjny	Windows 10 Education 10.0.18363

Dla każdej z omówionych wcześniej aplikacji przeprowadzono cztery testy mające na celu zbadać wydajność każdego ze szkieletów.

6.1. Test uruchamiania aplikacji w środowisku produkcyjnym

Ze względu na rosnącą popularność praktyki rozwoju oprogramowania opierającego się o ciągłą integrację (continuous integration) pierwszy z testów polegał na zmierzeniu czasu uruchamiania aplikacji w środowisku produkcyjnym. Każda z aplikacji została zbudowana oraz umieszczona na serwerze aplikacyjnym. Zdecydowano się na powtórzenie testu 50 razy dla każdej z nich. Test zawierał następujące kroki:

- Wyczyszczenie folderu roboczego (workspace) Jenkinsa,
- Pobranie projektu aplikacji do folderu Jenkinsa,
- Wywołanie zadań narzędzi automatyzujących: clean - czyszczący projekt oraz package/war kompilujący oraz pakujący projekt do paczki dystrybucyjnej,
- Umieszczenie paczki na serwerze Tomcat.

6.2. Test uruchamiania aplikacji w środowisku programistycznym

Podczas tworzenia dużych projektów jednym z bardziej czasochłonnych procesów jest budowanie aplikacji na lokalnym środowisku programistycznym. Problem ten został zaadresowany w drugim z testów. Wszystkie projekty aplikacji zostały uruchomione 20 razy na każdej z testowych maszyn. Proces uruchomienia obejmował zbudowanie aplikacji oraz umieszczenie jej na stosownym serwerze aplikacyjnym. W przypadku Spring Boota oraz Micronauta projekty zostały zbudowane oraz umieszczone na zintegrowanym z IDE serwerze Tomcat, w przypadku Quarkusa projekt zostaje przebudowany i uruchomiony na serwerze wbudowanym Netty. Ze względu na domyślną konfigurację oraz zalecenia dokumentacji szkielet Javalin został zbudowany i umieszczony na serwerze Jetty.

6.3. Test zużycia pamięci oraz procesora podczas działania aplikacji

Podczas wydawania aplikacji do ogólnego użytku ważne jest, aby jak największa liczba urządzeń była w stanie jej używać. Dużą znaczenie ma przy tym obciążenie procesora oraz ilość pamięci zużywanej przez wirtualną maszynę Javy, która przy złym zarządzaniu może przekroczyć możliwości urządzenia i uniemożliwić działanie programu. W kolejnym teście przetestowano zużycie pamięci oraz procesora przez każdą z aplikacji pod wybranym obciążeniem. Wszystkie aplikacje zostały zbudowane oraz umieszczone na serwerze aplikacyjnym. Pierwsza część testu polegała na zbadaniu zużycia w stanie bezczynności. Dla każdego szkieletu wykonano pomiar trwający 10 minut. Przez kolejne 5 minut aplikacje były poddane lekkiemu obciążeniu w postaci zapytań wysyłanych przez jednego użytkownika, pobierających 50 rekordów z bazy danych. W ostatniej fazie testu przez 5 minut wykonano silne obciążenie w posta-

ci zapytań wysyłanych przez 20 użytkowników pobierających 3 miliony rekordów z bazy danych.

6.4. Test szybkości punktów dostępowych

Ostatnim, ale nie mniej ważnym kryterium wziętym pod uwagę w testach jest szybkość komunikacji klienta z serwerem. W dzisiejszych czasach popularne aplikacje muszą być gotowe na odpowiedź na setki a nawet tysiące zapytań w ciągu sekundy. W ostatnim z testów sprawdzono jak szybko omawiane aplikacje odpowiedzą na dużą liczbę zapytań. Wszystkie aplikacje zostały zbudowane oraz umieszczone na serwerze aplikacyjnym. Pierwsza część testu polegała na zmierzeniu czasów odpowiedzi na zapytanie GET. Dla każdego szkieletu wykonano obciążenie w postaci 8000 zapytań, każde pobierające 1000 rekordów z bazy danych. Następnie wysłano 8000 zapytań POST, z których każde dodawało 200 rekordów do bazy danych. Na końcu obliczono jaki był łączny czas wysłania wszystkich zapytań GET oraz POST.

7. Analiza otrzymanych wyników

7.1. Test uruchamiania aplikacji w środowisku produkcyjnym

W pierwszym z testów zestawiono średnie czasy kolejnych kroków budowania aplikacji.

Tabela 3: Średnie czasy kolejnych kroków budowania aplikacji na poszczególnych maszynach

	Kompilacja		Pakowanie		Umieszczanie	
	Masz. 1	Masz. 2	Masz. 1	Masz. 2	Masz. 1	Masz. 2
Spring Boot	1,201 s	1,367 s	1,526 s	2,206 s	5,857 s	12,195 s
Micronaut	1,776 s	2,14 s	0,008 s	0,005 s	4,983 s	10,412 s
Javalin	0,24 s	0,288 s	0,004 s	0,004 s	1,892 s	2,05 s
Quarkus	1,158 s	1,517 s	0,135 s	0,169 s	6,28 s	7,721 s

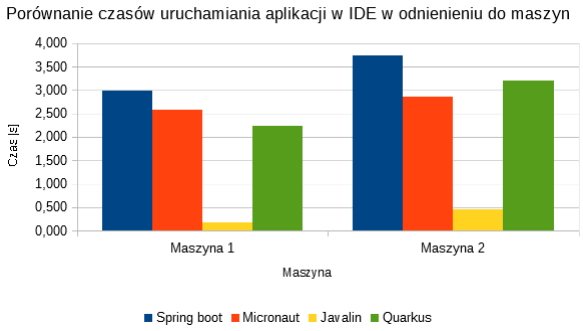
Przedstawione w Tabeli 3 czasy poszczególnych kroków budowania projektu znacznie różnią się między sobą. Aplikacja wykorzystująca Javalin z racji braku skomplikowanych mechanizmów obecnych w innych szkieletach kompiluje się znacznie szybciej od reszty. Quarkus oraz Spring Boot posiadają bardzo zbliżone wyniki. Micronaut ze względu na jego proces wstrzykiwania zależności w czasie kompilacji wykonywał ten proces najdłużej. W przypadku pakowania aplikacji do paczki dystrybucyjnej w przypadku Javalin oraz Micronauta, etap ten wykonywał się on praktycznie natychmiastowo, a nieco wolniejszy okazał się Quarkus. Jedynym szkieletem aplikacyjnym w którym czas był odczuwalny jest Spring Boot. W procesie umieszczania paczki dystrybucyjnej na serwerze najszybszym frameworkiem ponownie jest Javalin, następnie Quarkus - ze względu na swoją budowę uruchamiany na własnym serwerze. Dzięki procesom wykonywanym podczas kompilacji ostatni krok Micronaut wykonał szybciej od Spring Boota.

7.2. Test uruchamiania aplikacji w środowisku programistycznym

W kolejnym teście porównano średnie czasy uruchamiania projektów w zależności od maszyny.

Tabela 4: Porównanie czasów uruchamiania aplikacji między maszynami

	Spring Boot	Micronaut	Javalin	Quarkus
Maszyna 1	2,976 s	2,582 s	0,176 s	2,244 s
Maszyna 2	3,745 s	2,852 s	0,454 s	3,199 s



Rysunek 1: Porównanie czasów uruchamiania aplikacji między maszynami.

Analizując przedstawione w Tabeli 4 średnie czasy uruchamiania aplikacji, zauważyć można dosyć dużą różnicę w czasach pomiędzy Javalinem, a resztą szkieletów. Wynika to z faktu zmiany serwera aplikacyjnego na rekomendowany Jetty – dużo lżejszy w porównaniu do Tomcat, drugim powodem jest budowa samego Javalina mniej skomplikowanego od pozostałych frameworków. Kolejne dwa szkielety z porównywalnymi czasami to Micronaut oraz Quarkus. Najwolniejszy ponownie okazał się Spring Boot uruchamiający się powyżej trzech sekund. Dla lepszego zobrazowania różnic w czasach użyto wykresu kolumnowego przedstawionego na Rysunku 1.

7.3. Test zużycia pamięci oraz procesora podczas działania aplikacji

Następnie przy użyciu narzędzi Prometheus, Grafana oraz VisualVM zbadano zużycie procesora oraz pamięci dla każdej z aplikacji w stanie spoczynku, przy małym oraz dużym obciążeniu.

Analizując wyniki przedstawione w Tabeli 5 można stwierdzić, że nie ma większych różnic w zużyciu zasobów przez omawiane aplikacje w stanie spoczynku. Dla żadnej z nich średnie zużycie procesora nie przekroczyło 4%. Mimo wszystko najlepiej pod tym kątem prezentowała się aplikacja napisana z wykorzystaniem szkieletu Javalin, dla której zużycie procesora utrzymywało się najbliżej 0%. W przypadku tego frameworka średnie

zużycie pamięci również było najwyższe i wynosiło jedynie 30MB na maszynie 2.

Tabela 5: Średnie zużycie zasobów w stanie spoczynku

	Maszyna 1		Maszyna 2	
	CPU	Pamięć (MB)	CPU	Pamięć (MB)
Spring Boot	2%	282	2%	138
Micronaut	4%	301	3%	231
Quarkus	1,5%	184	4%	163
Javalin	0%	50	0%	30

Tabela 6: Średnie zużycie zasobów przy małym obciążeniu

	Maszyna 1		Maszyna 2	
	CPU	Pamięć (MB)	CPU	Pamięć (MB)
Spring Boot	68%	350	58%	350
Micronaut	75%	360	28%	250
Quarkus	68%	350	58%	350
Javalin	1%	325	3%	250

Dla badań przeprowadzonych przy lekkim obciążeniu wyniki okazały się o wiele ciekawsze. Zostały one zaprezentowane w Tabeli 6. Pojawiła się bowiem zauważalna rozbieżność w użyciu procesora między używanymi maszynami. W większości przypadków było ono niższe na maszynie 2, w przypadku Micronauta aż o 47 p.p. Wyjątkiem był ponownie Javalin, dla którego maszyna 1 mogła pochwalić się lepszym rezultatem. W przypadku tego szkieletu zużycie pamięci na obu maszynach było również najmniejsze, ale różnica w stosunku do pozostałych frameworków nie była aż tak znacząca jak w przypadku pierwszego testu.

Tabela 7: Średnie zużycie zasobów przy dużym obciążeniu

	Maszyna 1		Maszyna 2	
	CPU	Pamięć (MB)	CPU	Pamięć (MB)
Spring Boot	100%	200	98%	380
Micronaut	21%	350	27%	210
Quarkus	99%	700	99%	300
Javalin	8%	130	9%	130

Tabela 7 przedstawia wyniki uzyskane przy największym obciążeniu. Dużym zaskoczeniem okazały się aplikacje napisane z użyciem Micronaut oraz Javalin, dla których wykorzystanie procesora było bardzo małe (około 27% w przypadku Micronauta i 9% w przypadku Javalin) w porównaniu do pozostałych, gdzie zużycie wyniosło prawie 100%. Jeżeli chodzi o wykorzystanie pamięci to tak jak w przypadku pierwszego testu było ono wyraźnie niższe dla frameworka Javalin – dla pozostałych wyniki były zbliżone do tych przy małym obciążeniu. Wyjątkiem okazał się szkielet Quarkus, dla którego zużycie pamięciu było kilkukrotnie większe od jego konkurentów.

7.4. Test szybkości punktów dostępowych

Tabela 8: Porównanie czasów odpowiedzi

	GET		POST	
	Maszyna 1	Maszyna 2	Maszyna 1	Maszyna 2
Spring Boot	1398,3 s	2126,1 s	5085,5 s	4024,2 s
Micronaut	1849,8 s	3955,4 s	6855,5 s	5406 s
Quarkus	1509,5 s	2401,7 s	5550,2 s	4660,9 s
Javalin	467,4 s	1430,6 s	886,5 s	1586,4 s

Analizując wyniki przedstawione w Tabeli 8 można stwierdzić, że zdecydowanie najkrótsze czasy odpowiedzi serwera na obu maszynach uzyskała aplikacja napisana przy użyciu szkieletu Javalin. Był on kilkukrotnie krótszy, szczególnie w przypadku zapytania POST niż w przypadku Spring Boota, który zajął drugą lokatę w tym porównaniu. Co ciekawe najgorzej w tym zestawieniu wypadł Micronaut, który z założenia jest przeznaczony do tworzenia architektur mikroserwisowych, w których czas komunikacji z serwerem jest szczególnie istotny.

8. Wnioski

Celem niniejszej pracy było porównanie lekkich szkieletów programistycznych biorąc pod uwagę ich kluczowe elementy takie jak wydajność, szybkość oraz składnię.

Podczas implementacji aplikacji w poszczególnych szkieletach wskazano różnice oraz podobieństwa w konwencjach i rozwiązaniach w nich zaimplementowanych. Dokonano także porównania narzędzi dedykowanych do utworzenia podstawowego projektu. Tworzenie projektu pozwoliło na empiryczne sprawdzenie możliwości porównywanych frameworków. Na tej podstawie można zauważyć bardzo duże podobieństwo trzech szkieletów Spring Boota, Micronauta oraz Quarkusa – przedstawiają one podobne podejścia do konwencji i stylu tworzenia aplikacji. Spośród tych trzech frameworków z punktu widzenia programisty wyróżnia się szkielet Quarkus. Posiada on wiele przydatnych cech takich jak „hot deployment" czy specjalny tryb deweloperski. Całkowicie inne podejście do tematu implementacji aplikacji internetowej pokazuje Javalin. W swojej konstrukcji opiera się na prostocie i posiada jedynie podstawowe funkcjonalności. Niestety wadą tego rozwiązania jest to, że programista musi tworzyć własne

rozwiązania od podstaw lub adaptować zewnętrzne biblioteki.

Przeprowadzone badania wyraźnie wskazują aplikację opartą o szkielet Javalin jako najwydajniejszą. Pozostałe aplikacje wykonywały się znacznie wolniej oraz potrzebowały dużo więcej zasobów do swojej pracy. Najgorsze wyniki podczas testów osiągnął szkielet Spring Boot jednak jest on szeroko stosowany ze względu na dużą funkcjonalności i rozbudowane środowisko dodatków, wspieranych standardów oraz łatwo dostępne wsparcie społeczności.

Literatura

[1] D. Curie, J. Jaison, J. Yadav, J. Fiona, Analysis on Web Frameworks. Journal of Physics: Conference Series, 1362 (2019) 012114 doi:10.1088/1742-6596/1362/1/012114

[2] R. Rakshith Rao, S.R. Swamy, Review on Spring Boot and Spring Webflux for Reactive Web Development, International Research Journal of Engineering and Technology, 7(04) (2020) 3843-3837.

[3] Opis odwrócenia sterowanie w szkielecie Spring Boot, https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring , [28.02.2021].

[4] Oficjalna dokumentacja szkieletu Spring Boot, https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch01s02.html , [28.02.2021].

[5] Oficjalna dokumentacja szkieletu Micronaut, https://micronaut.io/docs, [28.02.2021].

[6] Wprowadzenie do szkieletu Micronaut, https://www.baeldung.com/micronaut , [28.02.2021].

[7] Oficjalna dokumentacja szkieletu Javalin, https://javalin.io/documentation , [28.02.2021].

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

[9] Przegląd funkcjonalności szkieletu Quarkus, https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus , [28.02.2021].

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