JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 7 (2018) 142-149
Wysłane: 2017-11-28
Przyjęte: 2017-12-02

Zastosowanie .NET Core w budowie aplikacji webowych

Ewelina Piątkowska*, Katarzyna Wąsik*, Małgorzata Plechawska-Wójcik
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule przedstawiono zastosowanie .NET Core w budowie aplikacji webowych. Poddano analizie aplikację zbudowaną w oparciu o framework .NET Core. Do analizy frameworka wybrano następujące kryteria porównawcze: wyszukiwanie wzorca w tekście, liczenie czasu parsowania pliku, realizacja operacji na plikach graficznych, dodawanie plików do archiwum, szybkość operacji CRUD. Wymienione kryteria będą porównywane na dwóch różnych systemach operacyjnych – Windows i Linux. Dodatkowo w artykule ukazano wyniki uzyskane po przeprowadzeniu badań oraz ich analizę. Postawiona hipoteza mówiąca, że testy zostaną wykonane szybciej w systemie Windows niż w systemie Linux, została jedynie częściowo potwierdzona.

Słowa kluczowe: aplikacja webowa; .NET Core; Angular 2; Entity Framework

*Autor do korespondencji.

Adresy e-mail: ewelina.piatkowska@pollub.edu.pl, katarzyna.wasik@pollub.edu.pl

The use of .NET Core in web applications development

Ewelina Piątkowska*, Katarzyna Wąsik*, Małgorzata Plechawska-Wójcik
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The article presents the use of .NET Core in web applications development. The analysis covers tests performed on a test application based on the .NET Core framework. The following benchmarking criteria were selected for the framework analysis: pattern search in text, counting parsing time, rendering operations on graphical files, adding files to archives, CRUD operation speed. These criteria were compared on two different operating systems - Windows, Linux. Additionally, the paper presents results obtained after the tests and their analysis. The hypothesis that tests will be executed faster on Windows than on Linux has only been partially confirmed.

Keywords: web application; .NET Core; Angular 2; Entity Framework

*Corresponding author.

E-mail addresses: ewelina.piatkowska@pollub.edu.pl, katarzyna.wasik@pollub.edu.pl

1. Wstęp

W obecnych latach obserwuje się bardzo szybki rozwój Internetu i stron WWW, stąd aplikacje internetowe stały się powszechnie znane i używane na całym świecie. Coraz więcej firm polega na aplikacjach sieciowych, dlatego wartościowość i spójność aplikacji stała się niezbędna [1]. Zaletą aplikacji webowych jest także możliwość korzystania z nich przez dowolną liczbę użytkowników. Poza tym do uruchomienia wystarczy przeglądarka internetowa taka jak np. Mozilla Firefox czy Google Chrome. W tym celu potrzebny jest tylko dostęp do sieci [2].

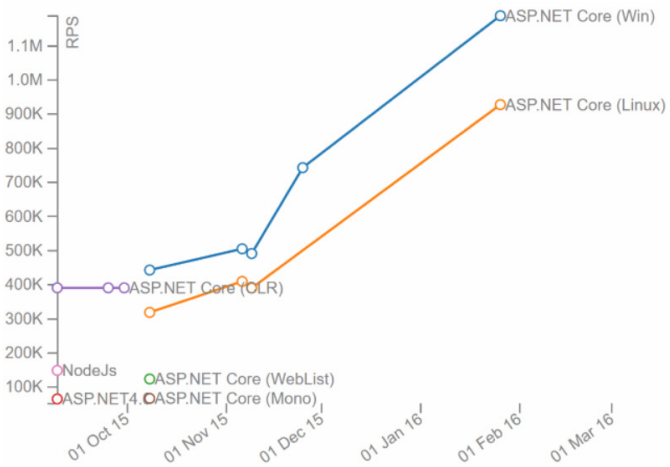
Testowanie jest jednym z pięciu głównych technicznych obszarów działalności inżynierii oprogramowania. Może ono odkryć większość błędów oprogramowania. Tak więc aplikacje internetowe powinny być testowane ostrożnie, aby upewnić się, że wymagane specyfikacje są spełnione przez aplikacje. Testowanie aplikacji webowych jest bardziej złożone niż zwykłych programów, ze względu na charakter dystrybucji, heterogeniczność, niezależność platformy i zgodność [3].

Testowanie aplikacji można realizować za pomocą kodu lub wykorzystując gotowe rozwiązania. Przykładem zewnętrznych oprogramowań są NUnit, XUnit lub Selenium. Niestety nie zawsze jest możliwość wykorzystania tych narzędzi do zbadania wszystkich części aplikacji. Przykładem tego mogą być pomiar wyników czasów realizacji poszczególnych operacji na poziomie serwera [4, 5].

Od 17 maja 2016 roku ASP.NET MVC funkcjonuje jako ASP.NET Core MVC. Jest jedną z najczęściej wybieranych technologii do tworzenia aplikacji internetowych [6]. ASP.NET Core MVC jest to wieloplatformowa, darmowa struktura programistyczna wspomagająca tworzenie aplikacji i usług internetowych z wykorzystaniem wzorca MVC. Dzieli on aplikację na trzy główne składniki: Model, Widok i Kontroler [7]. Łączy zalety starszych frameworków takich jak: ASP.NET MVC, ASP.NET i ASP.NET Web API zawierając je w jednej modułowej strukturze. Poza tym oferuje ona znacznie większą wydajność niż jej poprzednicy i może być wdrożona niemal wszędzie, w tym na Windows Server, Microsoft Azure, Linux czy macOS. Posiada również wbudowane narzędzia ułatwiające współpracę aplikacji ASP.NET Core MVC z kontenerami takimi jak Docker i Pivotal Cloud Foundry [8].

Na poniższym wykresie (Rys. 1), dotyczącym postępu wydajności ASP.NET Core, przedstawiona została zależność liczby żądań na sekundę (RPS) w wyznaczonym czasie [8]. ASP.NET 4.6 oraz Node.Js znajdują się na samym dole i są prawie niewidoczne. Największe postępy w wydajności

można zaobserwować w przypadku bardziej zwinnych systemów, jakimi są Windows i Linux [10]. System Windows został zaznaczony na wykresie niebieską linią, zaś Linux został oznaczony kolorem pomarańczowym [9].



Rys. 1. Wykres postępu wydajności ASP.NET Core [8]

1.1. Cel i obszar badań

Celem badań jest zbadanie wydajności stworzonej aplikacji testowej, bazując na określonych wcześniej kryteriach oraz uruchamiając ją na różnych platformach. W ten sposób będzie możliwe zbadanie jak .NET Core radzi sobie na dwóch różnych systemach: Windows i Linux.

W tym celu pod uwagę wzięto następujące kryteria: wyszukiwanie wzorca w tekście, obliczenie czasu parsowania pliku, realizacja operacji na plikach graficznych (zmiana rozmiaru obrazów), dodawanie plików do archiwum oraz szybkość realizacji operacji CRUD.

1.2. Opis aplikacji testowej

Aplikacja testowa została stworzona w technologii .NET Core w programie Microsoft Visual Studio w języku C#. W celu łatwiejszego tworzenia aplikacji opartej o architekturę MVC skorzystano z frameworka JavaScriptowego - Angular 2. Aplikacja umożliwia rejestrację i logowanie użytkowników w systemie, co daje im możliwość dodawania komentarzy i ich podgląd. Dodawane komentarze pojawiają się na żywo, dzięki wykorzystaniu biblioteki SignalR, bez konieczności odświeżania strony. Są one widoczne dla wszystkich zalogowanych użytkowników i automatycznie zapisywane w stworzonej na Microsoft SQL Server bazie danych. Ułatwieniem logowania jest możliwość zalogowania się za pomocą konta Facebook.

Dodatkowo w celu umożliwienia przetestowania frameworka, każdy zalogowany użytkownik może przetestować aplikację pod kątem dokładniej opisanych w rozdziale 2, niniejszego artykułu, kryteriów. Możliwe jest to po kliknięciu przycisku ”Tests”, znajdującego się w lewym górnym rogu aplikacji.

1.3. Hipotezy badawcze

W każdej pracy badawczej należy określić hipotezę badawczą, która jest jej istotną częścią. W artykule postawiono hipotezę dotyczącą porównania wydajności .NET Core. Założono, że testy zostaną wykonane w systemie Windows w krótszym czasie niż w systemie Linux. Zostanie to sprawdzone za pomocą wyżej wymienionych kryteriów.

2. Kryteria analizy

Poniżej skupiono się na dokładniejszym opisie wszystkich określonych wcześniej kryteriów służących sprawdzeniu wydajności frameworka. Wszystkie testy zostały zrealizowane za pomocą kodu, co stanowi duże ułatwienie w testowaniu i dalszej analizie uzyskanych wyników.

2.1. Wyszukiwanie wzorca w tekście

Dany test polega na wyszukaniu w dużej ilości tekstu konkretnego fragmentu tekstu, czyli wzorca. W jednym z folderów znajduje się plik z tekstem nazwany Text.txt , który zajmuje około 300 megabajtów.

Test wyszukiwania wzorca w tekście znajduje się w metodzie FindStringInText(). Na początku do pamięci jest wczytywany cały tekst zawarty w pliku Text.txt, następnie wyszukiwany jest ostatni indeks, w którym występuje szukane słowo „Lorem”, co ilustruje poniższy przykład 1. Zwracany typ double pokazuje czas działania testu.

Przykład 1. Metoda FindStringInText()

```
public double FindStringInText(string allText, string findingText)
{
    var watch = Stopwatch.StartNew();
    string text = File.ReadAllText(@".\Media\Text.txt");
    text.LastIndexOf("Lorem");
    watch.Stop();
    return TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds).TotalSeconds;
}
```

2.2. Liczenie czasu parsowania pliku

W przypadku tego testu zajęto się obliczeniem czasu przetwarzania obiektu do obiektu typu JSON. Początkowo tworzona jest lista obiektów typu String. W pętli lista jest uzupełniana danymi, a następnie cała lista zostaje parsowana do typu JSON. Wykonywane jest to w linijce JsonConvert.SerializeObject(files), co widoczne jest na przykładzie 2. Do parsowania obiektów jest wykorzystywana biblioteka Newtonsoft.Json.

Przykład 2. Metoda ParseJsonObject()

```
public double ParseJsonObject(object objectForParsing)
{
    List<string> files = new List<string>();
    for (int i = 0; i < 500; i++)
    {
        files.Add(i.ToString());
    }
    var watch = Stopwatch.StartNew();
    JsonConvert.SerializeObject(files);
    watch.Stop();
    return TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds).TotalSeconds;
}
```

2.3. Realizacja operacji na plikach graficznych

Do operacji na obrazkach zostanie wykorzystany resize, czyli zmiana wielkości obrazu. Cały proces odbywa się w metodzie ResizeImage() (Przykład 3).

Przykład 3. Metoda ResizeImage()

```
public double ResizeImage()
{
    var watch = Stopwatch.StartNew();
    using (Image<Rgba32> image = Image.Load(@".\Media\Images\earth.jpg"))
    {
        image.Mutate(x => x
             .Resize(image.Width / 2, image.Height / 2)
             .Grayscale());
        image.Save(@".\Media\Images\resizedImage.jpg");
    }
    watch.Stop();
    return TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds).TotalSeconds;
}
```

W danym teście obliczany jest czas zmiany rozmiaru obrazka. W folderze .../Media/Images jest umieszczony obrazek, który ma rozmiar 15000x15000. W trakcie testu jego rozmiar jest zmniejszany o połowę (do 7500x7500) oraz kolory obrazka przetwarzane są na czarno-białe.

Przetworzony obrazek zostaje umieszczony w folderze .../Media/Images/ResizedImages/resizedImage. Do pracy z obrazkami wykorzystywana jest biblioteka SixLabors.ImageSharp, która jest wieloplatformowa. To samo tyczy się obrazka o wymiarach 4000x4000.

Za operacje na obrazkach odpowiada również filtr Gauss. Fragment tej metody jest widoczny na poniższym przykładzie (Przykład 4).

Przykład 4. Metoda ApplyGausBlur()

```
var bigImageTime = Stopwatch.StartNew();
using (Image<Rgba32> image = Image.Load(pathToBigImage))
{
    image.Mutate(x => x.GaussianBlur(10));
    image.Save(pathToBigSavedImage);
}
bigImageTime.Stop();
```

Na początku w tej metodzie podawana jest ścieżka dostępu do obrazków o dwóch różnych rozmiarach podanych wyżej, następnie tworzony jest obiekt obrazu. Za ich przetworzenie odpowiada biblioteka *ImageSharp*, wykorzystująca metodę *GaussianBlur()*.

2.4. Dodawanie plików do archiwum

W danym teście obliczany jest czas dodawania plików do archiwum. Wszystko odbywa się w metodzie ZipFiles() (Przykład 5).

Przykład 5. Metoda ZipFiles()

```
public double ZipFiles()
{
    FilesHelper.CreateRandomFiles();
    FilesHelper.CleanResultDirectory();

    var watch = Stopwatch.StartNew();
    ZipFile.CreateFromDirectory(@".\Media\FilesForZip", @".\Media\ZipedFile\result.zip");
    watch.Stop();
    return TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds).TotalSeconds;
}
```

W folderze .../Media/FilesForZip umieszczone są pliki do archiwizacji. Zostały one wygenerowane w linijce z metodą FilesHelper.CreateRandomFiles(). Pliki generowane są o różnych rozmiarach, łącznie generowane jest 500 plików.

Stworzony plik .zip zawierający wszystkie utworzone pliki zostanie umieszczony w folderze .../Media/ZippedFile/result.zip. Przed rozpoczęciem testu zawsze sprawdzany jest ten folder. W razie gdyby nie był pusty, wszystkie pliki zostaną usunięte przed wykonaniem testu.

2.5. Szybkość realizacji operacji CRUD

Szybkość realizacji operacji CRUD obliczana jest w metodzie CountSQLQueriesGeneratingTime(). Oblicza ona czas zapytań wykonywanych w bazie danych. Metoda zwraca trzy parametry typu double (czas wyrażony jest w sekundach). Pierwszy parametr jest czasem odpowiedzi wykonania operacji typu INSERT, drugi parametr jest czasem wykonania zapytań typu SELECT, trzeci parametr jest parametrem czasowym dla operacji typu DELETE (Przykład 6).

Przykład 6. Metoda CountSQLQueriesGeneratingTime()

```
public double CountSQLQueriesGeneratingTime()
{
    var watch = Stopwatch.StartNew();

    _commentRepository.GetAll().AsQueryable();

    watch.Stop();
    return TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds).TotalSeconds;
}
```

Komunikacja z bazą danych polega na korzystaniu z frameworka Entity Framework, który jest narzędziem służącym do pracy z bazą danych jak ze zwykłymi obiektami. W sekcji przy obliczeniu operacji INSERT w pętli są tworzone obiekty typu Comment i od razu dodawane są one do kolekcji bazodanowej Comments. Po dodaniu danych do

bazy danych wywoływane jest zapytanie pobierania danych z bazy (SELECT). Po wykonaniu instrukcji SELECT, wszystkie dane będą usunięte - w tym miejscu jest obliczany czas dla operacji typu DELETE.

Testy będą wywoływane wielokrotnie dla każdego typu operacji. Dla każdego rodzaju operacji z osobna zostały wykonane testy z różną ilością powtórzeń wykonania danego zapytania. Na przykład, w przypadku operacji wstawiania wykonywane będzie wstawianie wielu rekordów kolejno. W pierwszym teście operacja ta zostanie powtórzona 1 000 razy, w następnym kroku 5 000 razy, w kolejnym 50 000 razy itd.

3. Metody i przebieg badań

Do badań wybrano dwa systemy operacyjne: Windows i Linux, z założeniem sprawdzenia wydajności platformy .NET Core. Celem optymalnej realizacji badań należało wybrać właściwe metody badawcze. W związku z tym, zdecydowano się na wybór metody porównawczej. Metoda ta została wykorzystana do porównania wyżej wymienionych systemów operacyjnych i wyników przeprowadzonych na nich testów na podstawie stworzonej aplikacji. Uwidoczniła ona różnice w działaniu frameworka na każdym z badanych systemów.

Na początku zainstalowano maszynę wirtualną, a na niej oba systemy operacyjne o takich samych parametrach. Po konfiguracji środowiska i pobraniu niezbędnych narzędzi nastąpiło uruchomienie stworzonej wcześniej aplikacji testowej. Następnie uruchomiono zaprojektowane uprzednio odpowiednie testy.

W celu sprawdzenia jak .NET Core radzi sobie na każdym z badanych systemów należało uruchomić wcześniej stworzoną aplikację zawierającą opracowane testy i przeanalizować uzyskane wyniki, a w końcowym etapie sporządzić ich interpretacje graficzne. Dla uzyskania powyższego rezultatu należało w uruchomionej aplikacji przejść do strony z testami, uruchomić je i poczekać na ich wykonanie.

W przypadku systemu Windows należało, po zainstalowaniu go na maszynie wirtualnej o nazwie VirtualBox i doinstalowaniu potrzebnych narzędzi, uruchomić stworzoną na potrzeby pracy aplikację z testami i przejść do strony, gdzie będą wykonywane testy, następnie uruchomić je i poczekać na wyniki. Jeśli chodzi o system Linux czynności do wykonania pozostają identyczne.

Testy, na każdym z systemów, zostały powtórzone dla operacji CRUD po trzy razy dla 13 wybranych wartości, zaś dla pozostałych kryteriów pięciokrotnie. Z uzyskanych wyników wyliczono wartości średnie i przedstawiono je na wykresach. Całość opisana została w rozdziale 4.

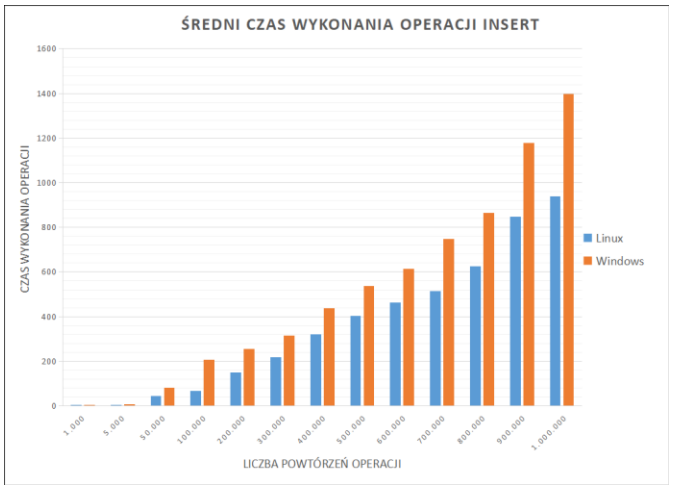
4. Wyniki badań

Po przeprowadzeniu testów poszczególne wyniki dla wybranych uprzednio kryteriów zostały umieszczone w kolejnych tabelkach, zamieszczonych poniżej. W tabeli 1 przedstawiono średnie czasy wykonania powtórzeń instrukcji INSERT dla 13 różnych wartości: 1 000, 5 000, 50 000, 100 000, 200 000, 300 000, 400 000, 500 000, 600 000, 700 000, 800 000, 900 000, 1 000 000 na dwóch badanych systemach operacyjnych: Windows i Linux wraz z odchyleniem standardowym.

Tabela 1. Zestawienie średnich wyników przeprowadzonych testów dla operacji INSERT wraz z odchyleniem standardowym

Liczba powtórzeń/ Badany system	**Linux Średni czas [s]**	**Windows Średni czas [s]**	**Odchylenie std dla wyników Linux**	**Odchylenie std dla wyników Windows**
1 000	1,3393	1,8723	0,1386	0,4824
5 000	3,8673	6,6817	0,6474	0,6674
50 000	43,6067	80,682	4,5801	60,6126
100 000	65,6977	206,5337	7,0350	45,5416
200 000	150,1527	254,1607	17,3437	41,6918
300 000	217,637	315,5603	10,628	143,0129
400 000	319,639	438,065	54,3253	248,591
500 000	402,568	538,4593	5,3889	204,3592
600 000	463,533	615,0413	63,4849	215,1631
700 000	513,208	748,264	27,4094	174,56
800 000	625,3034	865,6873	158,9147	118,7052
900 000	846,366	1177,797	66,7951	256,0625
1 000 000	938,0707	1393,6947	123,0148	272,2751

Na rysunku 2 zobrazowano wyniki przedstawione w tabeli 1 dotyczące średniego czasu wykonania operacji INSERT. Na osi X znajduje się liczba powtórzeń wykonania danej operacji, a na osi Y – badana wartość, czyli czas wykonywania operacji. Przykładowo dla 400 000 powtórzeń, dla systemu Linux osiągnięto wynik 319,639s, a dla systemu Windows 438,065s.



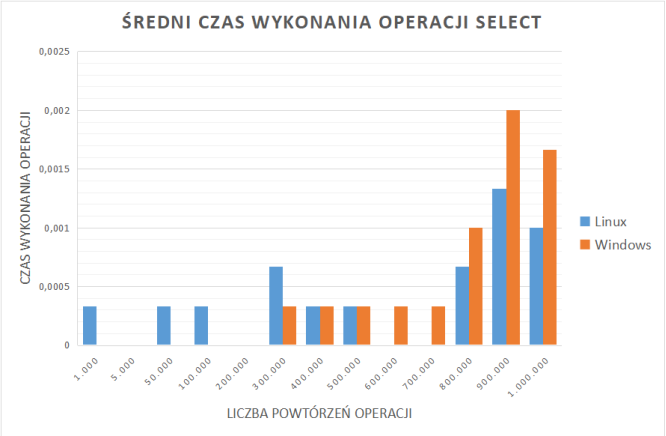
Rys. 2. Średni czas wykonania operacji INSERT

W tabeli 2 przedstawiono średnie czasy wykonania powtórzeń instrukcji SELECT dla 13 różnych wartości na dwóch badanych systemach operacyjnych wraz z odchyleniem standardowym.

Tabela 2. Zestawienie średnich wyników przeprowadzonych testów dla operacji SELECT wraz z odchyleniem standardowym

Liczba powtórzeń / Badany system	Linux Średni czas [s]	Windows Średni czas [s]	Odchyle-nie std dla wyników Linux	Odchyle-nie std dla wyników Windows
1 000	0,00033	0	0,000577	0
5 000	0	0	0	0
50 000	0,0003333	0	0,000577	0
100 000	0,00033	0	0,000577	0
200 000	0	0	0	0
300 000	0,00067	0,000333	0,001154	0,000577
400 000	0,00033	0,000333	0,000577	0,000577
500 000	0,00033	0,000333	0,000577	0,000577
600 000	0	0,000333	0	0,000577
700 000	0	0,000333	0	0,000577
800 000	0,0006667	0,001	0,001154	0,001173
900 000	0,00133	0,002	0,001154	0,001173
1 000 000	0,001	0,001667	0,001732	0,002081

Na rysunku 3 przedstawiono graficzną reprezentację wyników umieszczonych w tabeli 2. Przykładowo dla 1 000 000 powtórzeń w systemie Linux otrzymano wartość średnią równą 0,001s, a dla Windows 0,001667s.



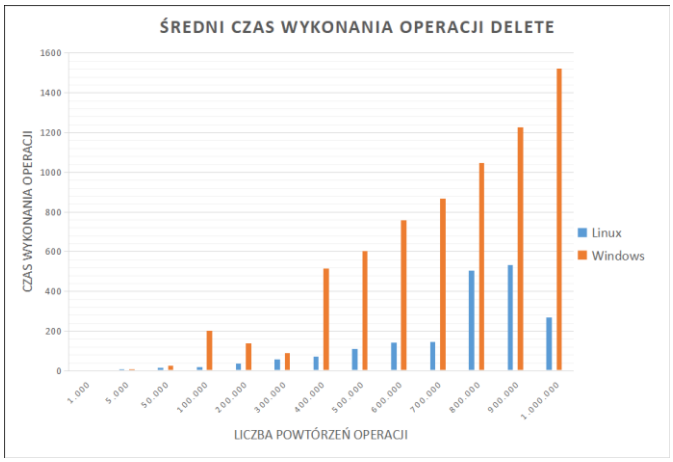
Rys. 3. Średni czas wykonania operacji SELECT

W tabeli 3 przedstawiono średnie czasy wykonania powtórzeń instrukcji DELETE dla 13 różnych wartości na uprzednio wymienionych systemach wraz z odchyleniem standardowym.

Tabela 3. Zestawienie średnich wyników przeprowadzonych testów dla operacji DELETE wraz z odchyleniem standardowym

Liczba powtórzeń / Badany system	Linux Średni czas [s]	Windows Średni czas [s]	Odchyle-nie std dla wyników Linux	Odchylenie std dla wyników Windows
1 000	1,6317	2,2823	0,2194	0,7599
5 000	3,475	8,415	3,6525	1,5167
50 000	13,9693	27,3037	3,9594	22,5443
100 000	19,9017	203,3063	1,8361	54,1722
200 000	36,078	139,4483	4,1181	178,5141
300 000	56,4167	89,1813	2,8853	44,0074
400 000	71,9973	514,9533	7,4012	174,185
500 000	110,907	602,113	13,1123	181,5452
600 000	142,103	756,9773	29,3997	178,5347
700 000	145,295	868,9963	19,112	174,9403
800 000	506,684	1048,5933	528,8153	282,8456
900 000	533,106	1227,003	489,6729	413,0618
1 000 000	268,009	1522,1483	40,5415	383,4439

Rysunek 4 prezentuje średnie wyniki otrzymane po wykonaniu operacji DELETE. Dokładne dane przedstawione są w tabeli 3.



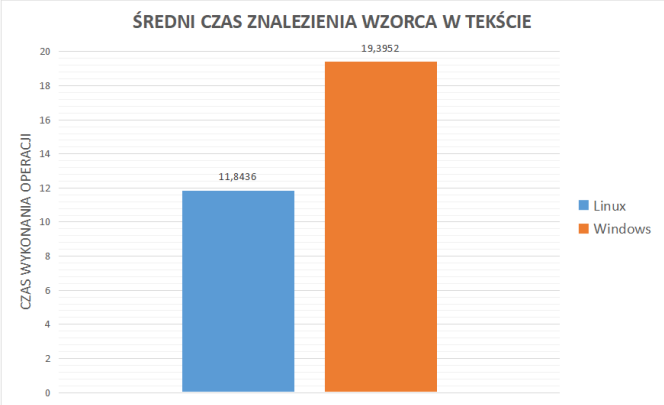
Rys. 4. Średni czas wykonania operacji DELETE

Tabela 4 zawiera czasy wyszukiwania wzorca w tekście, dla pięciu powtórzeń wraz z wartością średnią i odchyleniem standardowym. W systemie Linux średni czas wykonania tej operacji wynosi 11,8436s, zaś w systemie Windows 19,3952s.

Tabela 4. Zestawienie wyników przeprowadzonych testów dla czasu wyszukiwania wzorca w tekście wraz z odchyleniem standardowym

Badany system	Czas [s]
Linux	8,337
	5,604
	5,842
	5,631
	33,804
Średnia dla wyników Linux	**11,8436**
Odchylenie standardowe dla wyników Linux	**12,3299**
Windows	18,558
	18,878
	14,41
	23,999
	21,131
Średnia dla wyników Windows	**19,3952**
Odchylenie standardowe dla wyników Windows	**3,5365**

Na rysunku 5 zestawiono porównanie wartości średnich wyników z tabeli 4.



Rys. 5. Średni czas znalezienia wzorca w tekście

Tabela 5 demonstruje wartości uzyskane podczas testów mierzenia czasu parsowania pliku dla pięciu powtórzeń wraz z wartością średnią i odchyleniem standardowym. W systemie Linux średni czas wykonania tej operacji wynosi 1,689s, zaś w systemie Windows 1,5648s.

Tabela 5. Zestawienie wyników przeprowadzonych testów dla czasu parsowania pliku wraz z odchyleniem standardowym

Badany system	**Czas [s]**
Linux	1,472
	1,607
	1,647
	1,489
	2,23
Średnia wyników dla Linux	**1,689**
Odchylenie standardowe dla wyników Linux	**0,3116**
Windows	1,326
	1,202
	1,512
	1,374
	2,41
Średnia wyników dla Windows	**1,5648**
Odchylenie standardowe dla wyników Windows	**0,4853**

Poniższy rysunek (Rys. 6) pozwala dokładniej zaobserwować niewielką różnicę miedzy otrzymanymi średnimi wynikami.

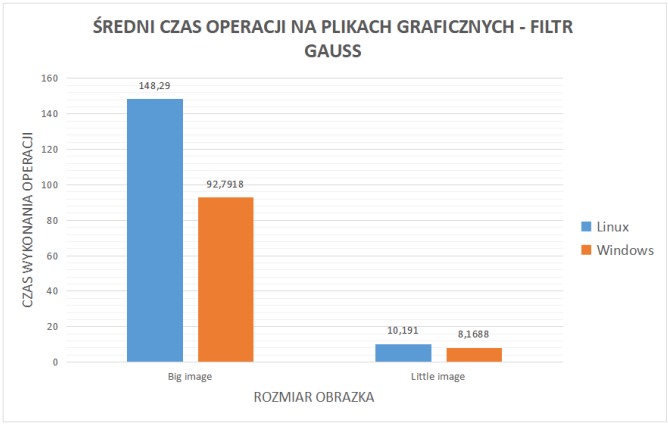
Podczas wykonywania testów na plikach graficznych rozpoczęto od naniesienia filtru Gaussa na obrazy. Uzyskane wyniki zawiera tabela 6 dla pięciu powtórzeń wraz z wartościami średnimi i odchyleniem standardowym. W systemie Linux średni czas wykonania tej operacji dla dużego obrazka wynosi 148,29s, a dla małego 10,191s, zaś w systemie Windows średni czas wykonania operacji na dużym obrazku wynosi 92,7918s, a na małym obrazku 8,1688s.

Rys. 6. Średni czas parsowania pliku

Tabela 6. Zestawienie wyników przeprowadzonych testów dla czasu operacji na plikach graficznych – filtr Gauss wraz z odchyleniem standardowym

Badany system	**Czas [s] – duży obrazek**	**Czas [s] – mały obrazek**
Linux	160,664	10,361
	144,184	10,738
	148.158	9,937
	149,212	9,165
	139,232	10,754
Średnia wyników dla Linux	**148,29**	**10,191**
Odchylenie standardowe dla wyników Linux	**7,9513**	**0,6638**
Windows	93,623	6,763
	90,992	6,831
	91,737	6,924
	90,597	6,67
	97,01	13,656
Średnia wyników dla Windows	**92,7918**	**8,1688**
Odchylenie standardowe dla wyników Windows	**2,6293**	**3,0688**

Średnie wyniki przeprowadzonych testów wyrażono na rysunku 7.



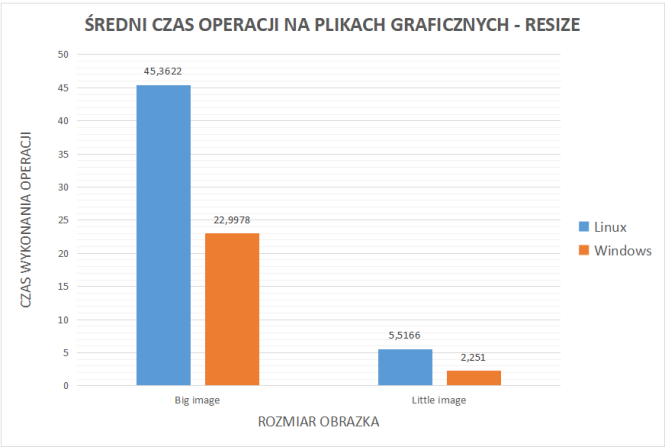
Rys. 7. Średni czas wykonania operacji na plikach graficznych – Filtr Gauss

Wykonując operacje resize otrzymano wyniki, które ukazuje tabela 7 dla pięciu powtórzeń wraz z wartościami średnimi i odchyleniem standardowym. W systemie Linux średni czas wykonania tej operacji dla dużego obrazka wynosi 45,3622s, a dla małego 5,5166s, zaś w systemie Windows średni czas wykonania operacji na dużym obrazku wynosi 22,9978s, a na małym obrazku 2,251s.

Tabela 7. Zestawienie wyników przeprowadzonych testów dla czasu operacji na plikach graficznych – Resize wraz z odchyleniem standardowym

Badany system	**Czas [s] – duży obrazek**	**Czas [s] – mały obrazek**
Linux	48,728	15,94
	44,238	2,642
	35,164	2,422
	52,557	2,86
	46,124	3,719
Średnia wyników dla Linux	**45,3622**	**5,5166**
Odchylenie standardowe dla wyników Linux	**6,4989**	**5,8476**
Windows	21,402	1,945
	19,475	1,475
	31,798	3,743
	18,781	1,419
	23,533	2,673
Średnia wyników dla Windows	**22,9978**	**2,251**
Odchylenie standardowe dla wyników Windows	**5,2552**	**0,9737**

W celu dokładniejszego porównania średnich wyników przedstawiono je również na rysunku (Rys. 8).



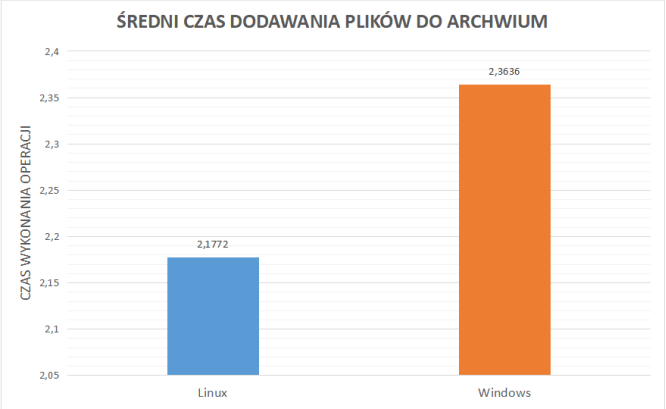
Rys. 8. Średni czas wykonania operacji na plikach graficznych - RESIZE

Tabela 8 pozwala zapoznać się z rezultatem testów mierzenia czasu dodawania plików do archiwum na dwóch systemach operacyjnych dla pięciu powtórzeń wraz z wartością średnią i odchyleniem standardowym. W systemie Linux średni czas wykonania tej operacji wynosi 2,1772s, zaś w systemie Windows 2,3636s.

Tabela 8. Zestawienie wyników przeprowadzonych testów dla czasu dodania plików do archiwum wraz z odchyleniem standardowym

Badany system	**Czas [s]**
Linux	3,309
	2,937
	3,006
	0,135
	1,499
Średnia wyników dla Linux	**2,1772**
Odchylenie standardowe dla wyników Linux	**1,3394**
Windows	1,523
	1,98
	1,651
	1,337
	5,327
Średnia wyników dla Windows	**2,3636**
Odchylenie standardowe dla wyników Windows	**1,6731**

Średnie wyniki zostały również przeniesione na rysunek (Rys. 9).



Rys. 9. Średni czas dodawania plików do archiwum

5. Wnioski

Celem pracy było sprawdzenie jak .NET Core radzi sobie na dwóch różnych systemach operacyjnych. W tym celu posłużono się metodą porównawczą i założono, że operacje realizowane w systemie Windows będą wykonywane w krótszym czasie. Udało się udowodnić część założeń przyjętych na początku niniejszego artykułu.

Odnośnie wyników pierwszego kryterium jakim jest szybkość wykonywania operacji INSERT, lepszy okazał się system Linux. Udowadniają to wartości zamieszczone w tabeli 1. Z kolei, jeśli chodzi o operacje SELECT, okazało się, że oba systemy mają podobne średnie wyniki. Średni czas wykonania operacji DELETE jednoznacznie wskazuje, że ten test szybciej został wykonany w systemie Linux.

W drugim kryterium uwidacznia się, że Linux jest prawie dwukrotnie szybszy niż Windows. Wyszukiwanie wzorca w tekście odbywa się zdecydowanie sprawniej na tym systemie.

Czas parsowania plików jest porównywalny w obu systemach, choć nieznacznie szybszy jest system Windows, co widoczne jest na rysunku 6.

Przechodząc do omawiania operacji na plikach graficznych należy skupić się w pierwszej kolejności na filtrze Gaussa. Dane przedstawione w tabeli 6 i zilustrowane na rysunku 7 doskonale obrazują szybkość z jaką poradził sobie Windows. Czas wykonywania operacji na plikach graficznych w systemie Windows jest nieporównywalnie krótszy, niż w systemie Linux. Dotyczy to zarówno mniejszych jak i większych obrazów.

Kolejnymi działaniami jakie zostały podjęte na obrazach jest zmiana ich wielkości. Tutaj także widoczne jest, iż Windows uzyskał lepsze rezultaty. Zostało to zilustrowane na rysunku 8.

Średnia dodawania plików do archiwum w obu platformach wynosi nieco powyżej 2 sekund. Jednakże zauważyć można, że Linux poradził sobie z tym nieco lepiej. Windows osiągnął średni wynik 2,3636s, a Linux 2,1772s.

Testy przeprowadzone na podstawie założonych kryteriów pozwalają zauważyć, że postawiona hipoteza została jedynie częściowo potwierdzona. Windows znacznie sprawniej radzi sobie z operacjami na plikach graficznych, takimi jak zmiana rozmiaru obrazu czy nakładanie filtrów. Natomiast jeśli chodzi o operacje związane z plikami tekstowymi to przoduje Linux, na przykład jak ma to miejsce przy wyszukiwaniu wzorca w tekście. Jeżeli chodzi o operacje na bazie danych to testy zostały szybciej wykonane w systemie Linux.

Literatura

[1] P. Nikfard, Functional Testing on Web Applications, University Technology Malaysia, January 2012.

[2] G. A. Di Lucca, A. R. Fasolino, Web Application Testing, Springer, 2006.

[3] P. Nikfard, Testing on Web Applications,Gorgan, Iran, May 2014.

[4] M. Monier, Evaluation of automated web testing tools, International Journal of Computer Applications Technology and Research, Volume 4– Issue 5, 405 - 408, 2015, ISSN:- 2319–8656.

[5] K. S. Dar, S. Tariq, H. J. Akram, U. Ghani, S. Ali, Web Based Programming Languages that Support Selenium Testing, Integrnational Journal of Foresight and Innovation Policy 2015(1), December 2015, s.21-25.

[6] J. Ciliberti, Getting the Most from the New Features in ASP.NET Core MVC, in ASP.NET Core Recipes, 2017, s.139-170.

[7] A. Freeman, Understanding Angular and ASP.NET Core MVC, in Essential Angular for ASP.NET Core MVC, July 2017, s.1-4.

[8] J. Ciliberti, ASP.NET Core Recipes A Problem-Solution Approach, Apress, 2017, s.1-42.

[9] .NET Core, .NET Framework, Xamarin – The “WHAT and WHEN to use it” https://blogs.msdn.microsoft.com/cesardelatorre/2016/06/27/net-core-1-0-net-framework-xamarin-the-whatand-when-to-use-it/ [15.11.2017].

[10] S. Sanderson, What’s the Big Idea?, in Pro ASP.NET MVC Framework, January 2009.