JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 13 (2019) 332-338
Wysłane: 2019-09-05
Przyjęte: 2019-09-11

WebAssembly jako alternatywa dla JavaScript w tworzeniu nowoczesnych aplikacji internetowych

Dawid Suryś*, Piotr Szłapa, Maria Skublewska-Paszkowska
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule opisano wpływ wykorzystania standardu WebAssembly na wydajność aplikacji internetowych. Wykorzystano oparty o WebAssembly szkielet aplikacji Blazor. Pokazano, iż można wykonać w pełni funkcjonalną aplikację SPA (ang. Single Page Application) pisząc kod aplikacji w języku C#. Wykonano drugą aplikację wykorzystując szkielet Angular. W obu aplikacjach zaimplementowano te same funkcjonalności. Dla wykonanych aplikacji porównano czasy ładowania w przeglądarce oraz rozmiar przesyłanych danych. Zbadano wpływ pamięci podręcznej przeglądarki i kompresji gzip. Zbadano wydajność obsługi żądań HTTP. Porównano wydajność kodu WebAssembly z kodem JavaScript w zadaniu sortowania n-elementowej listy obiektów. Zbadano wydajność aplikacji Blazor w modyfikowaniu drzewa DOM. Porównano wybrane metryki kodu obu aplikacji. JavaScript okazał się wydajniejszy w zadaniach związanych z wykorzystaniem API przeglądarki. WebAssembly był lepszy w zadaniach obliczeniowych.

Słowa kluczowe: JavaScript; WebAssembly; Blazor; Angular

*Autor do korespondencji.

Adres e-mail: dawid.surys@pollub.edu.pl

WebAssembly as an alternative solution for JavaScript in developing modern web applications

Dawid Suryś*, Piotr Szłapa, Maria Skublewska-Paszkowska
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The article describes the impact of using WebAssembly on the performance of web applications. A Blazor framework based on WebAssembly was used. The paper shows that it is possible to create fully functioning Single Page Application using C# programming language. The second application was made using Angular framework. Both applications implement the same functionalities. For prepared applications loading time and size of transferred data was measured. The impact of using browser cache memory and gzip compression was examined. The performance of handling http requests using GET and POST methods were measured. The performance of WebAssembly against JavaScript code in task of sorting a list of N objects was compared. The performance of modifying html DOM was examined. Selected code metrics of written applications were compared. JavaScript presents better performance with tasks related with browser API. WebAssembly was better for computing.

Keywords: JavaScript; WebAssembly; Blazor; Angular

*Corresponding author.

E-mail addresses: dawid.surys@pollub.edu.pl

1. Wstęp

Ciągły rozwój Internetu i jego rozpowszechnienie pociąga za sobą wzrost wymagań użytkowników. Strony www nie zawierają już tylko statycznych informacji, ale coraz częściej wspierają one różne procesy i tworzą pełnoprawne aplikacje nieustępujące możliwościom tradycyjnym aplikacjom desktopowym. Widoczna jest zmiana sposobu tworzenia takich aplikacji. W nowoczesnych aplikacjach webowych część serwerowa jest odseparowana od części klienckiej. Ta druga z roku na rok staje się coraz ważniejsza i coraz bardziej rozbudowana. Według ankiet przeprowadzonych przez portal *stackoverflow.com* w latach 2015/2016 ok. 6% programistów określało siebie jako front-end developer [1][2], w 2017 jest to 12% [3], w roku 2018 ponad 35% [4]. Od nowoczesnych aplikacji wymaga się również dużej wydajności. Według artykułu [5] 40% użytkowników opuszcza stronę, jeśli wczytuje się on dłużej niż 3 sekundy. Natomiast 79% osób robiących zakupy online deklaruje, że jest mniej prawdopodobne, że wrócą do danego sklepu, jeśli nie byli zadowoleni z wydajności strony.

Do tworzenia warstwy prezentacji aplikacji internetowych od wielu lat wykorzystywany jest język JavaScript. Początkowo został on zaprojektowany do prostych manipulacji na stronie www. Jednak bardzo szybki rozwój tego języka spowodował, że aktualnie oferuje on bardzo wiele możliwości i jest wykorzystywany zarówno do tworzenia warstwy usług jak i warstwy prezentacji aplikacji internetowych. Mimo dużej ewolucji jaką przeszedł język JavaScript wciąż posiada on problemy związane z wydajnością. Odpowiedzią na część z tych problemów jest niskopoziomowy język WebAssembly, który pozwala na

wykonanie kodu binarnego w przeglądarce z wydajnością bliską wydajności natywnego kodu [6]. WebAssembly nie jest językiem, w którym kod może być bezpośrednio pisany przez programistę. Stanowi raczej kod pośredni do którego mogą być skompilowane programy napisane w takich językach programowania jak np. C, C++, Rust [7].

W niniejszym artykule postanowiono zbadać możliwości standardu WebAssembly w kontekście tworzenie aplikacji internetowych. Postawiono tezę mówiącą, iż aplikacje wykorzystujące technologię WebAssembly oferują większą wydajność niż te które jej nie wykorzystują. Skupiono się na praktycznych przykładach. W tym celu wykorzystano rozwijany przez Microsoft framework *Blazor,* który swoje działanie opiera właśnie na języku WebAssembly. Przy pomocy wspomnianego szkieletu aplikacji (ang. framework) stworzono demonstracyjną aplikację. Aplikacja posłużyła do przeprowadzenia badań. Jako punkt odniesienia w analizie porównawczej wykonano analogiczną aplikację w technologii Angular która opiera swoje działanie o JavaScript.

2. Przegląd literatury

Brakuje publikacji porównującej aplikacje wykorzystujące WebAssembly z aplikacjami zbudowanymi na bazie szkieletów programistycznych opartych na języku JavaScript.

W artykule zatytułowanym "Gap:Analyzing the Performance of WebAssembly vs. Native Code" [6] autorzy dokonali porównania kodu skompilowanego do WebAssembly (WASM) uruchamianego w przeglądarce z natywnym kodem uruchamianym bezpośrednio z poziomu systemu operacyjnego. Autorzy przytoczyli wyniki podobnych badań tego typu. W tych badaniach kod binarny uruchamiany z przeglądarki okazywał się ok. 10% wolniejszy od kodu natywnego. Jednak wszystkie te badania opierały się na kodzie niewielkich rozmiarów (ok 100 linii kodu). W badaniu ze wspomnianego artykułu użyto bardziej rozbudowanej aplikacji napisanej w języku C++. Wyniki wskazały na znacznie gorszą wydajność WebAssemby. W przeglądarce Firefox zanotowano spadek o ok. 50% względem natywnego kodu, natomiast w przeglądarce Google Chrome o 89%.

Artykuł [8] porusza temat wydajności aplikacji SPA Autorzy postanowili poddać analizie metody przyśpieszania procesu ładowania aplikacji tego typu. Na potrzeby badania wykonano prostą aplikację z użyciem frameworka AngularJS. Następnie przeprowadzono serię eksperymentów sprawdzających skuteczność różnych metod poprawy wydajności ładowania aplikacji. Badano rozmiar przesyłanych do przeglądarki plików oraz czas ładowania aplikacji. Pierwsza metoda optymalizacji polegała na połączeniu wszystkich plików CSS oraz JavaScript i przesłaniu ich do klienta w jednym żądaniu. Druga metoda opierała się na usunięciu nieużywanych reguł CSS. Kolejną badaną metodą była minifikacja kodu JavaScript, czyli usunięcie wszystkich białych znaków, komentarzy oraz zmiana nazw zmiennych na krótsze. Ponadto sprawdzono jaki wpływ na wydajność ma użycie protokołu HTTP/2 oraz kompresja z użyciem programu gzip. Najlepsze rezultaty zmniejszenia rozmiaru plików otrzymano dla metody minifikacji kodu JS. Z oryginalnego rozmiaru aplikacji 4,37MB udało się osiągnąć 1,57MB oraz 1,54MB przy dodatkowo zastosowanej kompresji. Rozmiar nagłówków zależy w dużej mierze od liczby żądań. Znaczne ograniczenie liczby żądań osiągnięto przez złącznie plików *.js oraz *.css. Przy takiej samej liczbie żądań lepsze kompresja nagłówków jest zauważalna w protokole HTTP/2. W przypadku analizy czasu ładowania aplikacji nie zaobserwowano widocznej różnicy między zastosowanie protokołu HTTP 1.1 czy HTTP/2. Również mechanizm PUSH PROMISE nie spowodował skrócenia czasu ładowania aplikacji.

Istnieje wiele artykułów, w których dokonano porównania frameworka Angular2 z innymi bibliotekami JavaScript. Jednym z takich artykułów jest ten autorstwa J. Kalinowska, B. Pańczyk [9]. Dokonano w nim analizy porównawczej frameworka Angular2 z ReactJS. Analiza oparta była na przygotowanych specjalnie w tym celu aplikacjach. W porównaniu uwzględniono strukturę i wydajność aplikacji, wybrane metryki kodu, jakość dokumentacji oraz wsparcie społecznościowe. Na podstawie badań przeprowadzono wnioski. Jednym z nich jest stwierdzenie, iż Angular2 jest lepszym narzędziem dla bardziej doświadczonego programisty.

Autorzy artykułu [10] w swojej pracy dokonują analizy porównawczej kilkunastu popularnych bibliotek. Przeprowadzono badanie ankietowe. W badaniu wzięło udział 18 osób z 6 krajów z całego świat. Sklasyfikowano badane osoby według następujących cech: doświadczenie zawodowe, używane biblioteki, dziedzina działalności (programista back-end lub front-end), rozmiar firmy, typ firmy. Największą popularność wśród ankietowanych miała biblioteka jQuery (8 osób) oraz framework AngularJS (6 osób). Dwie badane osoby używają swoich własnych rozwiązań w pracy, jedna osoba odpowiedziała, że nie korzysta z żadnych szkieletów aplikacji. Większość ankietowanych wskazało więcej niż jeden framework. Jako programistę front-end określiło się ośmiu programistów, siedmiu zaznaczyło, że zajmują się zarówno częścią kliencką jak i serwerową, natomiast trzech wyłącznie częścią serwerową. Większość osób pracowało w średniej wielkości firmie. Autorzy badania przyjęli następujące kryteria oceny: wydajność i rozmiar frameworka, stopień skomplikowania i łatwość nauki, wsparcie społeczności, funkcjonalności i cena. Wyciągnięto następujące wnioski. Dokumentacja frameworka powinna jasno wskazywać, jeśli jej poprawne działanie polega głównie na zasobach sprzętowych środowiska, w którym jest uruchamiany. Liczba linii kodu potrzebnych do napisania aplikacji w danym frameworku powinna być tak mała jak to tylko możliwe. Dokumentacja frameworka powinna być precyzyjna, zawierać przykłady implementacji typowych zadań oraz umożliwiać programistom na szybkie znalezienie wskazówek dotyczących implementacji danej funkcjonalności. Biblioteki, które mają rozbudowaną społeczność i są rozwijane od dłuższego czasu, powinny być wybierane. Struktura aplikacji zbudowanej według frameworka powinna być modułowa, a zamiana w jednym module nie powinna wymagać zmiany w inny. Framework

powinien pozwalać na łatwe importowanie do projektu zewnętrznych bibliotek. Darmowe rozwiązania oraz biblioteki rozwijane jako projekty open-source są chętniej wybierane od tych płatnych.

3. Aplikacja zapisu studentów na dodatkowe zajęcia

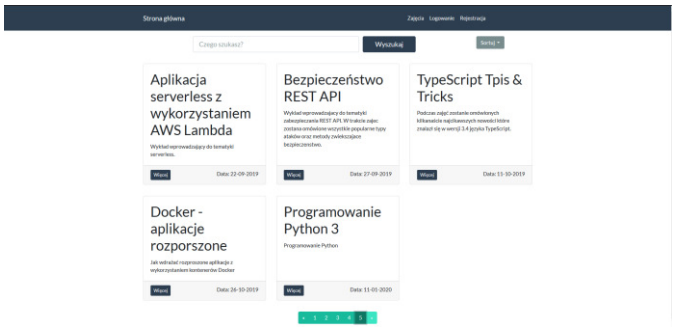
Na potrzeby badań powstały 2 aplikacje o takim samym graficznym interfejsie użytkownika. Każda z nich wykonuje te same zadania. Omawiany system informatyczny wspomaga obsługę zapisywania się studentów na dodatkowe zajęcia. Służy również jako ułatwienie prowadzenia kursów przez wykładowców. Umożliwia tworzenie oraz edycję zajęć dla prowadzących, a także zapisywanie się na zajęcia oraz potwierdzanie obecności w przypadku roli studenta. System wyróżnia następujące poziomy uprawnień: administrator, firma, wykładowca, student. Zawiera mechanizmy uwierzytelniania i autoryzacji użytkowników. Oczywiście w rozdziale zostały wymienione tylko najważniejsze możliwości omawianego systemu informatycznego.

Realizowana aplikacja jest aplikacją webową z rozdzieloną częścią serwerową oraz częścią kliencką. Część kliencką stanowią dwie aplikacje SPA. Do napisania jednej aplikacji wybrano framework Blazor. Aktualnie jest to projekt, który otrzymał oficjalne wsparcie od Microsoft i wchodzi w skład .Net Foundation. Framework ten wybrano, ponieważ działa on w oparciu o WebAssembly, tzn. wykonuje kod binarny w przeglądarce. Standardowe aplikacje wykonują skrypty JavaScript. Kod aplikacji Blazor został napisany w języku C#. Do napisania drugiej aplikacji klienckiej wybrano framework Angular. Od pewnego czasu znajduje się on w czołówce popularności frameworków i bibliotek JavaScript. Wyboru Angulara dokonano, ponieważ pozwala on na zachowanie bardzo przejrzystej struktury projektu. Ponadto oferuje możliwość korzystania z zalet języka TypeScript, takich jak statyczne typowanie. Dzięki temu można częściowo ograniczyć popełnione błędy w trakcie pisania aplikacji.

Część serwerowa została stworzona zgodnie ze wzorcem REST API. Wykonane ją z użyciem języka C# oraz frameworka ASP.NET CORE w wersji 2.2. Serwer stanowi aplikację typu Web API. Aplikacja serwerowa współpracuje z bazą danych SQL Server 2016. Baza danych od firmy Microsoft posiada duże zastosowanie w aplikacjach ASP.NET. Spowodowane jest to tym, iż firma z Redmond udostępnia dość szczegółową oficjalną dokumentację oraz inne źródła wiedzy. Do obsługi bazy danych w aplikacji wykorzystano ORM Entity Framework Core. Do stworzenia struktury bazy danych i relacji wykorzystano popularne środowisko SQL Server Management Studio.

Głównym środowiskiem pracy był rozbudowany edytor tekstu Visual Studio Code. Oferuje on szereg rozszerzeń. Tym samym był wykorzystywany do napisania każdej ze składowych części projektu tj. aplikacji serwerowej i aplikacji klienckich. Jako dodatkowe wsparcie momentami wykorzystywano pełne środowisko Visual Studio.

Rys. 1 przedstawia zrzut ekranu zawierający stronę startową aplikacji.



Rys. 1 Strona startowa aplikacji

4. Przebieg badania

Podstawowym zdaniem, jakie sobie postawiono, było udowodnienie tezy, „Możliwe jest stworzenie funkcjonalnej aplikacji SPA z wykorzystaniem standardu WebAssembly i języka programowania C#". Ponadto aplikacja powinna dać się uruchomić w większości dostępnych przeglądarek internetowych bez konieczności instalowania dodatkowych rozszerzeń lub wtyczek. Wszystkie przewidziane funkcjonalności zostały zaimplementowane. Aplikacja została z powodzeniem przetestowana w przeglądarkach Google Chrom, Firefox, Edge oraz Google Chrome dla systemu Android.

Właściwym etapem badania było zmierzenie wydajności stworzonej aplikacji oraz samego kodu WebAssembly. Jako punkt odniesienia przy komentowaniu wyników wydajności aplikacji Blazor była bliźniacza wersja aplikacji napisana w technologii Angular. Wybrano ten szkielet aplikacji, ponieważ rozwijany jest od dłuższego czasu i powszechnie wykorzystywany przez deweloperów. Porównano czas ładowania aplikacji, ilość przesłanych danych oraz czasy obsługi żądań http.

Obok głównej aplikacji przygotowano osobno implementację dla dwóch scenariuszy badawczych. Pozwoliły one na zmierzenie czasu wykonania kodu sortowania listy obiektów oraz czasu generowania elementów DOM.

4.1. Stanowisko badawcze

Wszystkie opisane w procedurze badawczej eksperymenty zostały wykonane na jednym stanowisku badawczym. Stanowisko badawcze składa się z komputera stacjonarnego PC. Szczegółowa specyfikacja została zaprezentowana w tabeli 1.

Tabela 1. Specyfikacja stanowiska badawczego

System operacyjny	Procesor	Pamięć	Dysk twardy
Windows 10 Pro 1809 64bit	Intel Core i3-4100M 2,5GHz	12GB DDR3 (800MHz)	Samsung SSD 840 EVO 120GB

Podczas przeprowadzania badania zostały wykorzystane narzędzia:

- Nginx 1.16
- Kestrel Web Server
- Sql Server 13.0.4001.0
- Google Chrome 75
- Emscripten 1.38.21

4.2. Wydajność ładowania aplikacji

Celem eksperymentu było określenie wydajności ładowania aplikacji w przeglądarce. Do określenia wydajności przyjęto takie parametry jak całkowity czas ładowania w przeglądarce, moment zdarzenia „DOMContentLoaded", moment zdarzenia „load", liczba wykonanych żądań http i rozmiar pobranych danych. Eksperyment został wykonany w przeglądarce Google Chrome. Testowane aplikacje były ładowane z uruchomionego lokalnie serwera Nginx. Serwer nasłuchiwał na portach 81 i 82 kolejno dla aplikacji Blazor oraz aplikacji Angular. Aplikacje komunikowały się przez protokół http z aplikacją serwerową ASP.NET Core uruchomioną lokalnie w tym samym systemie przy użyciu Kestrel Web Server. Aplikacja serwerowa ze swojej strony łączyła się z bazą danych SQL Server również hostowaną lokalnie na tym samym systemie.

Eksperyment wykonano w 3 wariantach:

- wariant A - została wyłączona pamięć podręczna w przeglądarce oraz kompresja danych gzip na serwerze;
- wariant B – została włączona pamięć podręczna przeglądarki z pozostawioną nieaktywną kompresją;
- wariant C - wyłączono pamięć podręczną, natomiast włączono kompresję gzip na serwerze Nginx;

W przykładzie 1 przedstawiono użytą konfigurację modułu gzip.

Przykład 1. Konfiguracja Gzip

```
gzip  on;
gzip_min_length 1000;
gzip_proxied    expired no-cache no-store private auth;
gzip_types text/plain text/css application/json application/javascript
application/x-javascript text/javascript text/xml application/xml
application/rss+xml application/atom+xml application/rdf+xml
application/octet-stream application/wasm image/jpeg;
```

4.3. Wydajność operowania na kolekcji obiektów

Celem eksperymentu było zmierzenie czasu jaki jest potrzebny aplikacji na wykonanie sortowania listy obiektów n elementowych. Wykorzystano algorytm sortowania szybkiego (quick sort). Przygotowano cztery implementacje opisanego eksperymentu, są to:

- TypeScript z frameworkiem Angular
- JavaScript bez dodatkowych bibliotek
- C# z frameworkiem Blazor (WebAssembly)
- C++ skompilowany do kodu WASM

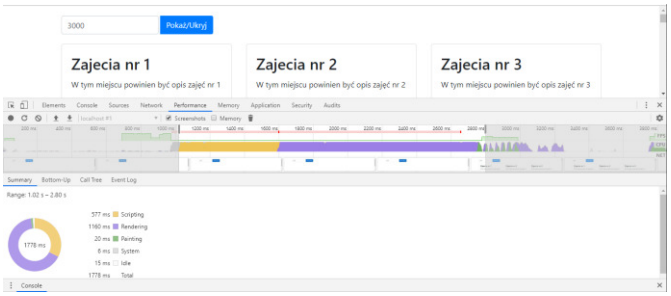
Czas wykonania sortowania zmierzono bezpośrednio w kodzie programu. Dla implementacji wykorzystujących JavaScript skorzystano z funkcji performance.now(), dla języka C# użyto klasy StopWatch z przestrzeni nazw System.Diagnostics, natomiast w przypadku C++ wykorzystano funkcję high_resolution_clock. Fragment kodu odpowiedzialny za pomiar czasu wykonania operacji został przedstawiony w przykładzie 2. Eksperyment przeprowadzono dla kolekcji 1 tys. elementów, 3 tys. elementów oraz 10 tys. elementów. Dla każdego rozmiaru wykonano po 10 prób dla każdej z implementacji.

Przykład 2. Pomiar czasu sortowania w C++

```
std::chrono::time_point<std::chrono::high_resolution_clock> t1 =
std::chrono::high_resolution_clock::now();
quicksort(lectures, 0, n);
std::chrono::time_point<std::chrono::high_resolution_clock> t2 =
std::chrono::high_resolution_clock::now();
std::chrono::duration<double> time_span =
std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1);
```

4.4. Wydajność generowania elementów DOM

W eksperymencie zamierzano sprawdzić jak framework Blazor poradzi sobie z dynamicznym modyfikowaniem drzewa DOM. Ze względu na fakt, że WebAssembly obecnie nie posiada bezpośredniego dostępu do DOM`u strony spodziewano się opóźnienia w działaniu względem aplikacji Angular. Do zmierzenia czasu renderowania w aplikacji wykorzystano dostępne w przeglądarce Google Chrome narzędzie do pomiaru wydajności. Wykonano testy generowania 1000, 3000 oraz 10000 elementów na stronie. Dla każdej serii wykonano 10 prób dla aplikacji Blazor oraz Angular. Rys.2 przestawia przykładowy pomiar wykonany dla 3 tys. elementów.



Rys. 2 Test wydajności renderowania widoku

4.5. Wydajność obsługi żądań HTTP

W tym eksperymencie postanowiono zmierzyć opóźnienie powstałe podczas wykonywania żądań HTTP. Na podstawie napisanych aplikacji Angular oraz Blazor zmierzono czas potrzebny na wysłanie żądania oraz otrzymania odpowiedzi. Podobnie jak w poprzednich eksperymentach aplikacje klienckie oraz aplikacja serwerowa zostały uruchomione lokalnie na tej samej maszynie. Wykonano żądania GET oraz POST w obydwu aplikacjach. Każdą operację powtórzono 10 razy. Żądanie GET zwracało do aplikacji listę zajęć dla pojedynczej wyświetlanej strony (6 elementów). Żądanie POST zapisywało w bazie nowe zajęcia na podstawie danych wysłanych z formularza.

4.6. Metryki kodu

W sytuacji, gdy wydajność aplikacji nie jest tak istotna lub dostępne rozwiązania nie różnią się znacząco w tym zakresie, większego znaczenia nabiera sam proces wytwarzania oprogramowania i związany z tym komfort pracy programisty. Mimo tego, że język JavaScript różni się znacząco od języka C# to dzięki zmianom wprowadzonym w standardzie ECMAScript 6 (m.in. klasy i dziedziczenie) oraz rozszerzeniu

możliwości języka oferowanej przez TypeScript tworzony kod ma wiele cech charakterystycznych dla paradygmatu OOP ang. Object Oriented Programming. W związku z powyższym kod aplikacji Blazor napisany w języku C# nie różni się znacząco od kodu aplikacji Angular napisanego w języku TypeScript. W obydwu aplikacjach zaimplementowano te same funkcjonalności. Ponadto obydwa frameworki posiadają podobną strukturę opartą o komponenty. Jednak każde rozwiązanie posiada swoje własne rozwiązania bardziej szczegółowych zagadnień oraz inną składnię tworzenia warstwy wizualnej. Blazor korzysta z znanego z aplikacji ASP silnika Razor, natomiast Angular posiada swój autorski silnik. Opisane różnie przekładają się na ilość kodu wymaganą do napisania przez programistę a pośrednio na jego wydajność. Zmierzono napisany kod przy wykonanych aplikacjach w postaci linii kodu programu. Porównano ze sobą osobno kod dla każdego komponentu aplikacji. Rozdzielono również część logiki komponentu od warstwy prezentacji. Usunięto wszystkie puste linie oraz komentarze.

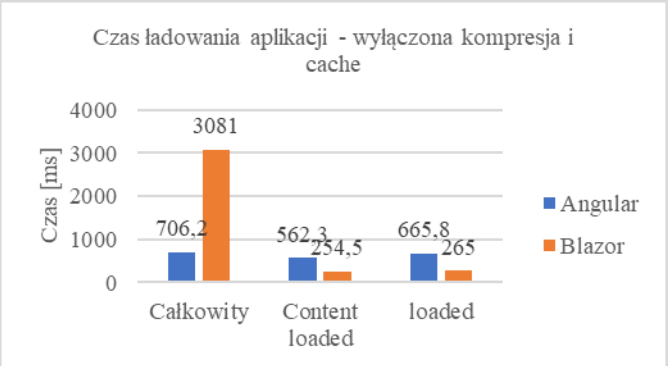
5. Wyniki

Rozmiar zbudowanej aplikacji Blazor jest znacznie większy od aplikacji Angular. Jest to spowodowane koniecznością pobrania środowiska uruchomieniowego Mono w formie kodu WebAssembly oraz wszystkich wykorzystywanych standardowych bibliotek środowiska .Net jako pliki .dll. Nie ma potrzeby pobierania wszystkich tych danych przy każdorazowym ładowaniu aplikacji. Przeglądarki internetowe przechowują je w pamięci cache, co znacząco zwiększa wydajność. Zastosowanie pamięci cache w przeglądarce Google Chrome pozwoliło na zmniejszenie początkowego rozmiaru 5,9MB do 4,1KB. Aplikacja Angular przy włączonej pamięci podręcznej posiadała rozmiar 7,7KB. Zastosowanie kompresji gzip na serwerze nginx dało podobny efekt dla obydwu typów aplikacji. Rozmiar aplikacji Angular został zmniejszony o 57% natomiast aplikacji Blazor o 56%. W tabeli 2 przedstawiono rozmiary aplikacji w różnych wariantach.

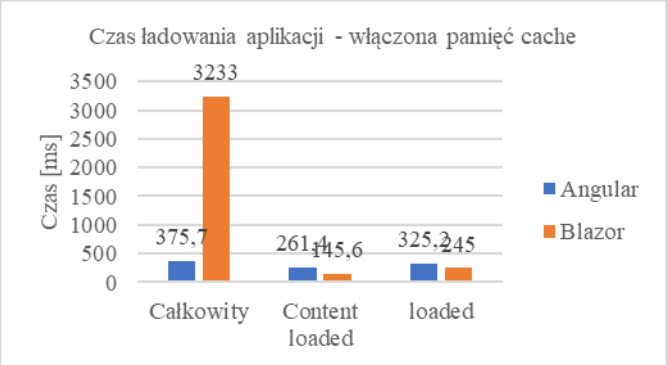
Tabela 2 Rozmiar aplikacji Blazor i Angular

Aplikacja / Rozmiar[KB]	Angular	Blazor
Całkowity	1500	5900
Włączony cache	7,7	4,1
Włączona kompresja gzip	644	2600

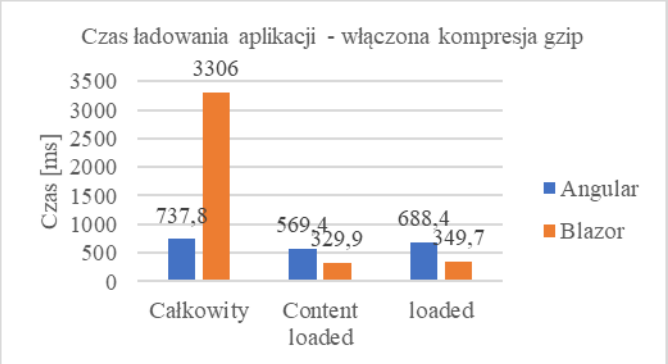
Na czas ładowania badanych aplikacji składa się ich rozmiar oraz liczba wykonanych zapytań. W obydwu kategoriach aplikacja Blazor posiada gorsze wyniki, co przekłada się na znacznie dłuższy czas ładowania. Aplikacja Blazor potrzebuje wykonać 44 zapytania, natomiast aplikacja Angular potrzebuje tych zapytań jedynie 22. Średni całkowity czas ładowania aplikacji Angular to 706.2ms, dla aplikacji Blazor jest to 3081ms. Użycie pamięci cache w przeglądarce spowodowało spadek całkowitego czasu ładowania dla aplikacji Angular a w aplikacji Blazor nie przyniosło żadnej korzyści w tym zakresie. Kompresja gzip choć zmniejszyła rozmiar przesyłanych danych w ostatecznym rezultacie spowodowała wydłużenie czasu ładowania aplikacji. Spowodowane jest to faktem, iż testy przeprowadzono na jednym stanowisku wykluczając tym samym opóźnienia spowodowane przez transfer danych przez sieć. Wykonane testy pokazały, że moment wywołania zdarzeń DOMContentLoaded oraz Load następuje wcześniej w aplikacji Blazor, w każdym z badanych scenariuszy. Rys.3, rys. 4, rys.5 zawierają wykresy kolumnowe przedstawiające dane o czasie ładowania aplikacji.



Rys. 3 Średni czas ładowania dla wariantu A

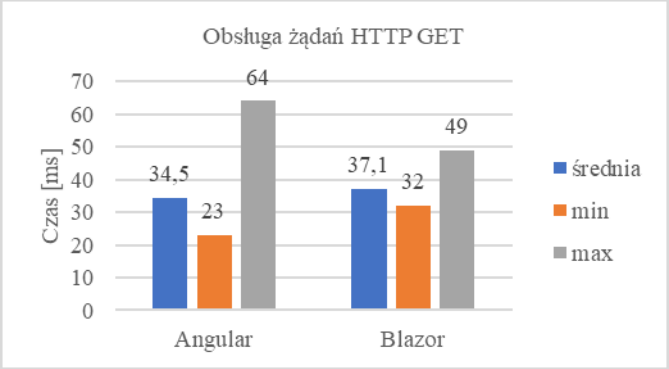


Rys. 4 Średni czas ładowania dla wariantu B

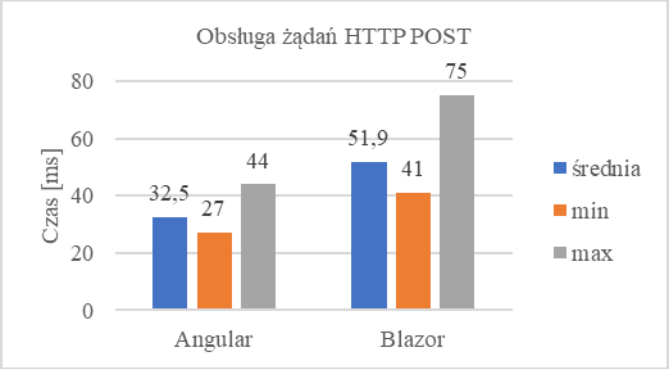


Rys. 5 Średni czas ładowania dla wariantu C

Opóźnienie obsługi żądań HTTP wykonywanych w aplikacji jest większe w przypadku szkieletu Blazor. Jest to zgodne z oczekiwaniami, ponieważ WebAssembly obecnie nie posiada możliwości bezpośredniego wykonywania operacji HTTP. Rozwiązaniem jest interakcja modułu WASM z Fetch API przeglądarki [11]. Żądania GET były obsługiwane średnio 2.6ms dłużej w aplikacji Blazor, żądania POST średnio 19.4ms dłużej. Na rys.6 i rys.7 przedstawiono wykresy średniego, minimalnego i maksymalnego czasu obsługi żądania GET i POST dal obydwu aplikacji.



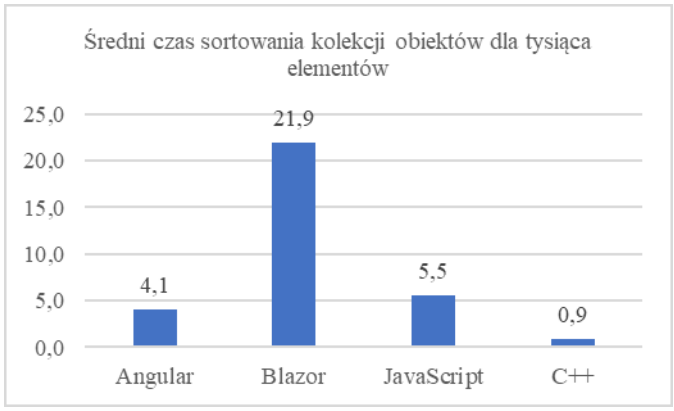
Rys. 6 Czas obsługi żądań http GET



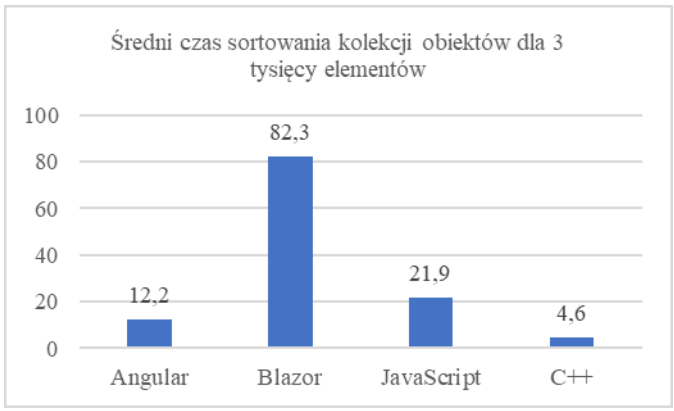
Rys. 7 Czas obsługi żądań http POST

Eksperyment sortowania n-elementowej listy obiektów został przeprowadzony w innej aplikacji niż eksperyment wydajności ładowania. Dla każdego z czterech wariantów (Angular, Blazor, JavaScript, C++) przygotowano osobną aplikację do tego celu. Uzyskane wyniki wskazują znaczną różnicę w wydajności pomiędzy kodem JavaScript a WebAssembly. Kod napisany w C++ i skompilowany do modułu WASM wykonywał operację znacznie szybciej niż analogiczny kod JavaScript. Różnica jest widoczna dla tysiąca, trzech tysięcy i dziesięciu tysięcy elementów. Można również zauważyć, że kod wykorzystany w aplikacji Angular nie traci na wydajności względem kodu napisanego jako pojedynczy skrypt. Uzyskane czasy w obydwu przypadkach są porównywalne. Aplikacja Blazor nie wykorzystuje w tym teście możliwości WebAssembly. Uzyskane w tej aplikacji czasy sortowania są dla wszystkich badanych rozmiarów kolekcji kilkukrotnie dłuższe od tych uzyskanych w aplikacji Angular. Uśrednione wartości czasu sortowania dla poszczególnych rozmiarów kolekcji przedstawiono na wykresach (rys8, rys.9, rys.10).

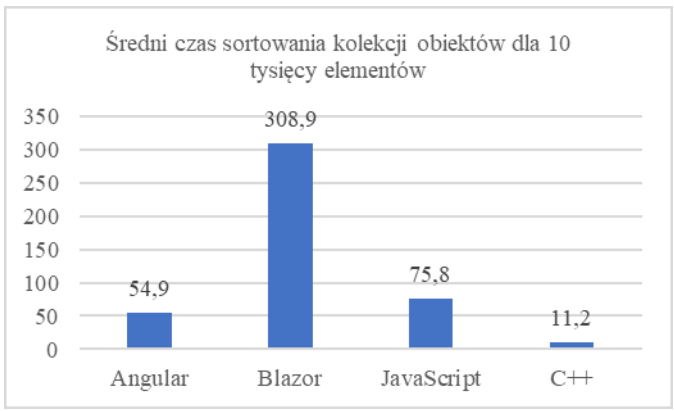
W badaniu wydajności modyfikowania elementów DOM wykorzystano aplikacje Angular i Blazor użyte w eksperymencie sortowani. Część odpowiedzialną za generowanie elementów umieszczono w osobnych komponentach. Aplikacja Blazor prezentuje gorsze wyniki od tych uzyskanych w aplikacji Angular. Czas dodania do drzewa dokumentu tysiąca elementów jest niemal sześciokrotnie dłuższy w aplikacji Blazor. Dla trzech tysięcy elementów jest już ponad sześciokrotnie dłuższy, natomiast dla dziesięciu tysięcy ponad siedmiokrotnie dłuższy. Należy zaznaczyć, że kod WebAssembly aktualnie nie ma możliwości bezpośredniego dostępu do drzewa DOM a wszystkie modyfikacje jego struktury zapisane w aplikacji Blazor są wykonywane przez wygenerowany kod JavaScript. Podobnie jak w przypadku wykonywania operacji HTTP, spodziewano się opóźnienia względem aplikacji Angular. Uśrednione wyniki z podziałem na liczbę generowanych elementów przedstawiono na rys. 11.



Rys. 8 Średni czas sortowania kolekcji tysiąca elementów



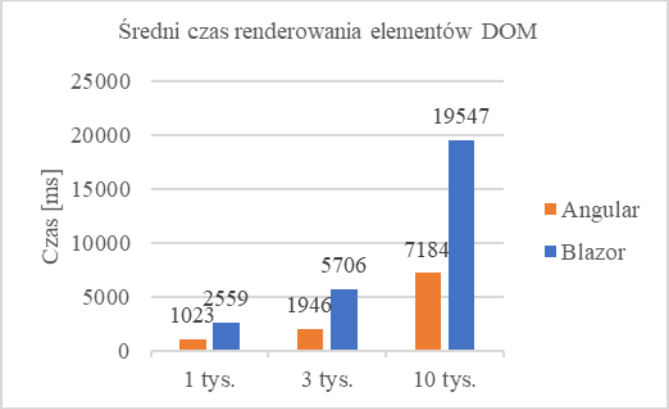
Rys. 9 Średni czas sortowania kolekcji trzech tysięcy elementów



Rys. 10 Średni czas sortowania kolekcji dziesięciu tysięcy elementów

Zbadane metryki kodu nie pozwalają na jednoznaczne wskazanie szkieletu aplikacji, który wymaga mniejszego nakładu pracy do zaimplementowania tej samej funkcjonalności. Sześć na dziesięć komponentów aplikacji Angular posiada mniejszą liczbę linii kodu niż analogiczne komponenty aplikacji Blazor. Rozpatrując oddzielnie logikę komponentu i część prezentacji można zauważyć pewne tendencje. W warstwie logiki pięć komponentów aplikacji Blazor posiada mniejszą liczbę linii kodu, w pozostałych

pięciu mniej kodu znajduje się w aplikacji Angular. W warstwie prezentacji w siedmiu komponentach kod aplikacji Angular zawierał mniej linii kodu, w jednym komponencie uzyskano dokładnie tę samą liczbę linii kodu. Można wysnuć wniosek, że składnia używana w aplikacjach Angular jest bardziej precyzyjna, a sam kod bardziej skondensowany. Uzyskane dane zebrano w tabeli 3.



Rys. 11 Średni czas renderowania elementów DOM

Tabela 3 Metryki kodu

	Szablon		Logika	
Aplikacja / Komponent	Angular	Blazor	Angular	Blazor
Home	60	87	45	105
Administrator	88	93	137	144
Lecture	30	44	115	97
Lecturer	145	175	218	198
Lectures	60	77	77	136
New Lecturer	52	52	97	142
Student	260	244	201	190
SignIn	26	29	87	94
SignUp	52	49	119	105
Opinions	24	34	81	47

6. Wnioski

Zbadano możliwości standardu WebAssembly jako alternatywy dla języka JavaScript w tworzeniu nowoczesnych aplikacji internetowych. Punktem wyjścia był dynamicznie rozwijany projekt Blazor. Przygotowana w oparciu o ten szkielet aplikacja wraz z analogiczną aplikację opartą o szkielet Angular i język JavaScript posłużyły do przeprowadzenia badań.

Uzyskane wyniki pozwalają wyciągnąć następujące wnioski. Szkielet aplikacji Blazor stanowi doskonały przykład wykorzystania potencjału WebAssembly. Pozwolił na stworzyć funkcjonalnej aplikacji SPA, wykorzystując inne podejście niż typowe z kodem JavaScript. Uzasadnione więc jest stwierdzenie, że standard WebAssembly może stanowić alternatywę dla języka JavaScript w tworzeniu nowoczesnych aplikacji internetowych. Jednak nie pozwala on całkowitą rezygnację z JavaScript`u. Część funkcjonalności przeglądarek internetowych może być osiągnięta wyłącznie przez JavaScript. W tych przypadkach użycie WebAssembly nie spowoduje zwiększenia wydajności, co zaobserwowano w badaniu czasu obsługi żądań HTTP oraz generowania elementów interfejsu użytkownika. Znaczną korzyść można natomiast uzyskać wykorzystując WebAssembly do zadań obliczeniowych. W eksperymencie sortowania kolekcji obiektów program napisany w C++ i skompilowany bezpośrednio do modułu WASM był kilkukrotnie szybszy od programu napisanego w JavaScript (rys.8-rys.10). Analiza wydajności ładowania wykazała dość duży rozmiar aplikacji Blazor. Jednak problem dotyczy tylko pierwszego załadowania aplikacji. Użyta pamięć cache znacząco zmniejsza rozmiar przesłanych danych.

Postawiona w artykule teza została częściowo potwierdzona. Wykorzystanie WebAssembly może w pewnych sytuacjach wpływać pozytywnie na wydajność aplikacji internetowych.

Zaprezentowany szkielet Blazor stanowi zaledwie jeden przykład implementacji standardu WebAssembly. Można oczekiwać, że pojawią się podobne rozwiązania dla innych języków programowania. W przyszłych wersjach WebAssembly ma się pojawić m.in. implementacji mechanizmu *Garbage collection* [12]. Ułatwi to stworzenie środowiska uruchomieniowego dla języków programowania wyższego poziomu takich jak *Python* czy *Go*.

Literatura

[1] 2015 Developer Survey https://insights.stackoverflow.com /survey/2015 [11.06.2019]

[2] Developer Survey Results 2016 https://insights.stackoverflow. com/survey/2016 [11.06.2019]

[3] Developer Survey Results2017 https://insights.stackoverflow. com/survey/2017 [11.06.2019]

[4] Developer Survey Results2018 https://insights.stackoverflow. com/survey/2018/ [11.06.2019]

[5] D. An, P. Meenan, Why marketers should care about mobile page speed, https://www.thinkwithgoogle.com/marketing-resources/experience-design/mobile-page-speed-load-time/ [11.06.2019]

[6] A. Jangda, A. Guha, B. Powers, E. Berger, Mind the Gap:Analyzing the Performance of WebAssembly vs. Native Code, 2019

[7] C. G. Gallant, WebAssembly in Action, Mananig Publications, 2019

[8] W. Stępniak, Z. Nowak Performance Analysis of SPA Web Systems, 2017

[9] J. Kalinowska, B. Pańczyk, Porównanie narzędzi do tworzenia aplikacji typu SPA na przykładzie Angular2 i React, 2019

[10] A. Pano, D. Graziotin Factors and actors leading to the adoption of a JavaScript framework, 2018

[11] Call a web API from ASP.NET Core Blazor, https://docs. microsoft.com/pl-pl/aspnet/core/blazor/call-web-api?view= aspnetcore-3.0 [25.08.2019]

[12] WASM Features to add after the MVPhttps://webassembly.org /docs/future-features [10.08.2019]