JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 269–273
Received: 23 June 2020
Accepted: 6 July 2020

Analysis of the Blazor framework in client-hosted mode

Analiza działania szkieletu Blazor w trybie klienta z hostingiem

Karol Kozak*, Jakub Smołka

Abstract

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

Keywords: Blazor; C#; JavaScript; WebAssembly

Streszczenie

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

Słowa kluczowe: Blazor; C#; JavaScript; WebAssembly

*Corresponding author

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

1. Wstęp

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

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

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

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

2. Przegląd literatury

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

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

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

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

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

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

3. Metoda badawcza

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

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

Specyfikacja stanowiska badawczego określona została w Tabeli 1.

Tabela 1: Specyfikacja stanowiska badawczego

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

3.1. Wydajność ładowania aplikacji

Celem eksperymentu było określenie wydajności ładowania aplikacji dla trybu klienta z hostingiem szkieletu Blazor. W tym celu stworzony został komponent wyświetlający 10 kart wraz z obrazkami.
Użytymi parametrami dla pomiarów dotyczących czasu ładowania aplikacji są:

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

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

3.2. Wydajność obliczania liczby PI

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

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

3.3. Wydajność obliczania funkcji Ackermanna

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

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

3.4. Wydajność operacji na kolekcjach LINQ

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

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

3.5. Wydajność aktualizowania elementów DOM

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

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

4. Wyniki

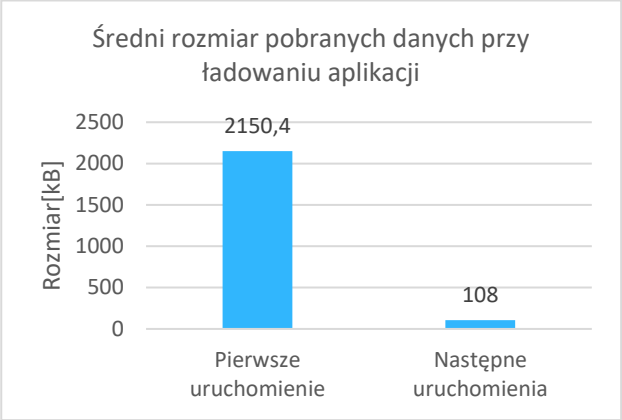
Wyniki pomiarów czasów uruchomienia aplikacji zostały przedstawione w tabelach 2 i 3. Można zauważyć, że czas pierwszego, pełnego załadowania się strony (parametr Finish) jest ok. 10 razy dłuższy niż czas wstępnego załadowania strony (parametr Load). Dzięki zastosowaniu pamięci podręcznej przeglądarki,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

Tabela 2: Wyniki czasów pierwszego uruchomienia aplikacji

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

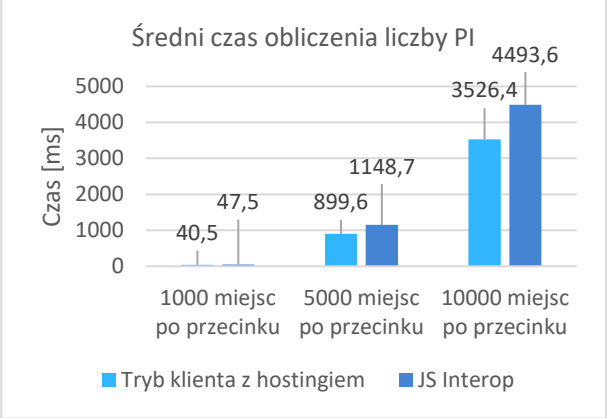
Tabela 3: Wyniki czasów następnych uruchomień aplikacji

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



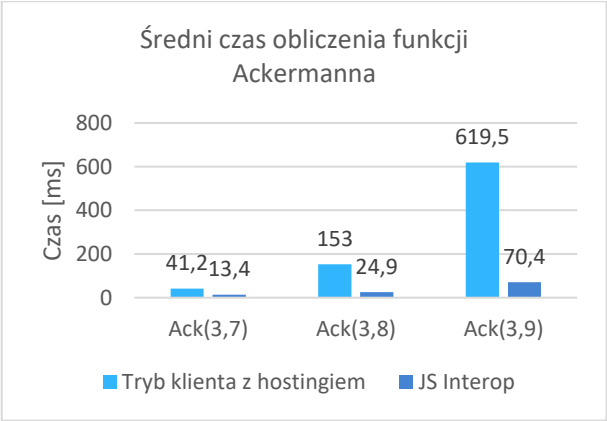
Rysunek 1: Średni rozmiar pobranych danych przy ładowaniu aplikacji

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



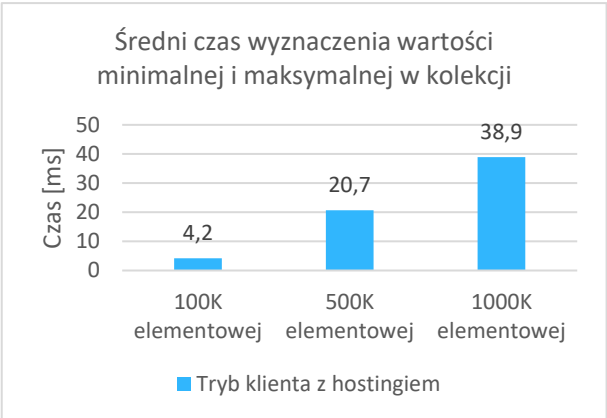
Rysunek 2: Średni czas obliczenia liczby PI

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



Rysunek 3: Średni czas obliczenia funkcji Ackermanna

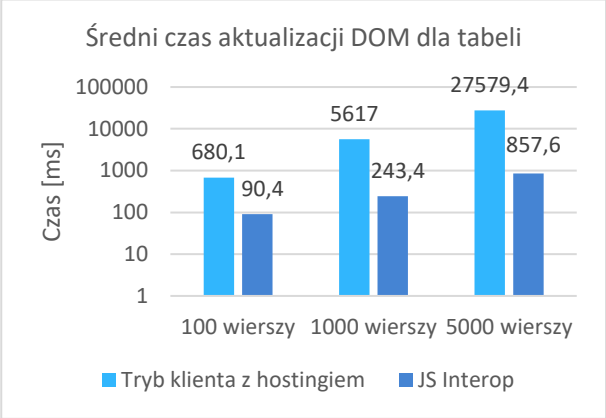
Funkcja Ackermanna jest przykładem algorytmu rekurencyjnego. Pomiary przedstawione na rysunku 3 pokazują, że kod JavaScript jest znacznie szybszy od kodu C# szkieletu Blazor. Wraz ze wzrostem złożoności funkcji, czas obliczeń w JavaScript wzrasta zdecydowanie wolniej niż w przypadku implementacji funkcji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



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

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

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



Rysunek 5: Średni czas aktualizacji DOM w tabeli

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

5. Wnioski

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

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

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

Literatura

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

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

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

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

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

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

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

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