JOURNAL
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

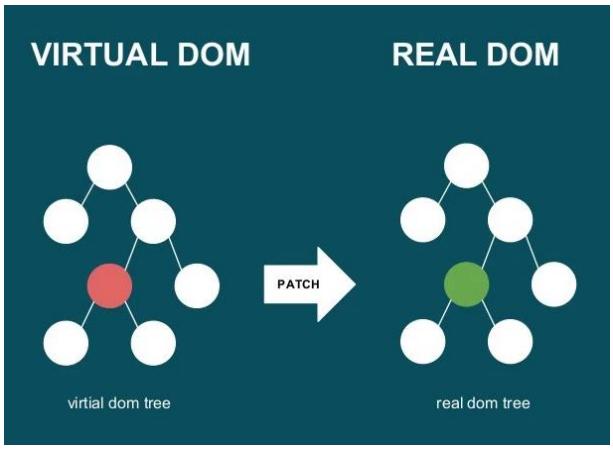
Email address: gabriel.bialecki@pollub.edu.pl

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

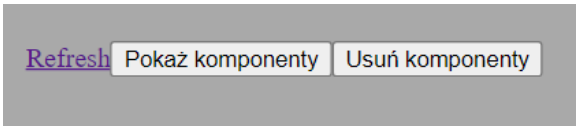
Badania wydajnościowe zostały przeprowadzone w oparciu o dwie proste, takie same co do funkcjonalności aplikacje testowe, zaimplementowane odpowiednio w Svelte, wersja 3.0 i Angular, wersja 10.2. Aplikacje zostały przygotowane w taki sposób, aby można było zmierzyć czasy renderowania i usuwania różnej 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

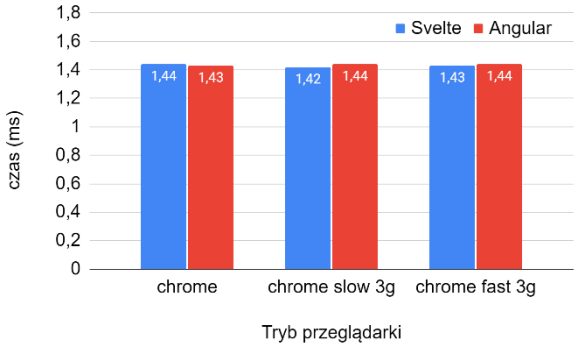
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	1,44	14,18	143,50
slow 3g	1,42	13,40	142,68
fast 3g	1,43	13,87	144,11
incognito	1,46	14,22	142,49
slow 3g incognito	1,42	13,72	143,64
fast 3g incognito	1,55	14,01	144,98

Tabela 3: Średnie czasy (ms) renderowania komponentów w Angular

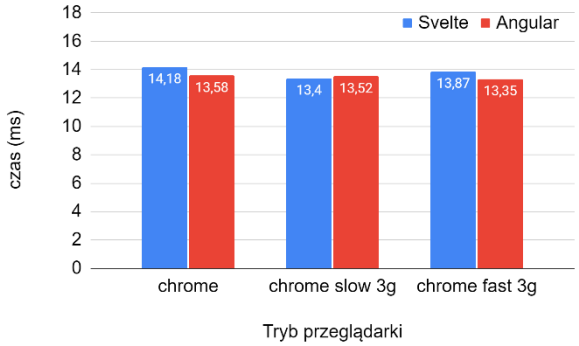
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	17,97	150,96	822,54
slow 3g	18,34	150,59	818,28
fast 3g	18,33	150,42	845,31
incognito	18,73	153,85	820,62
slow 3g incognito	18,89	154,52	850,90
fast 3g incognito	18,89	153,89	827,94

Tabela 4: Średnie czasy (ms) renderowania komponentów -w Svelte

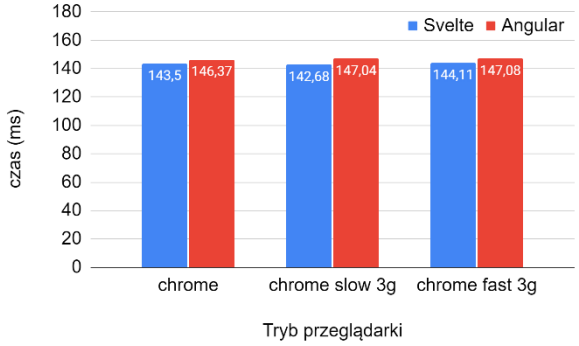
Tryb przeglądarki	Liczba komponentów		
	100	1 000	10 000
normal	2,67	21,12	195,70
slow 3g	2,71	21,90	277,67
fast 3g	2,65	21,10	238,21
incognito	2,51	20,60	190,17
slow 3g incognito	2,59	21,99	267,91
fast 3g incognito	2,53	20,93	230,99



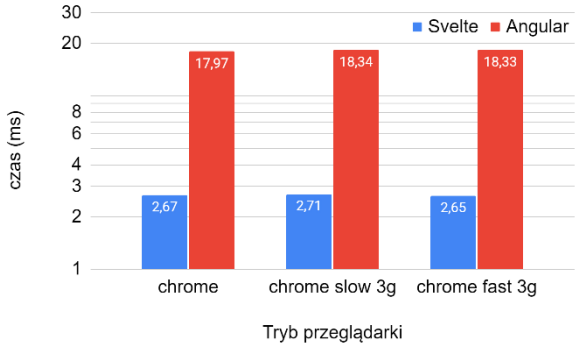
Rysunek 3: Średnie czasy usuwania 100 komponentów w zwykłym trybie.



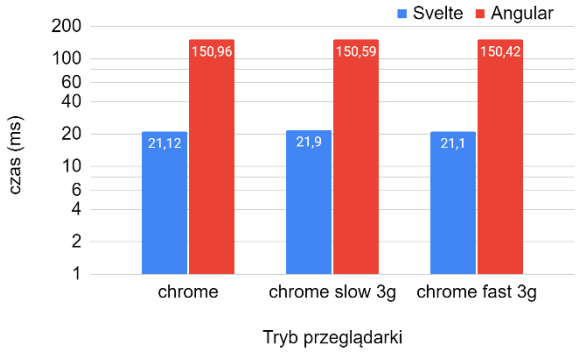
Rysunek 4: Średnie czasy usuwania 1 000 komponentów w zwykłym trybie.



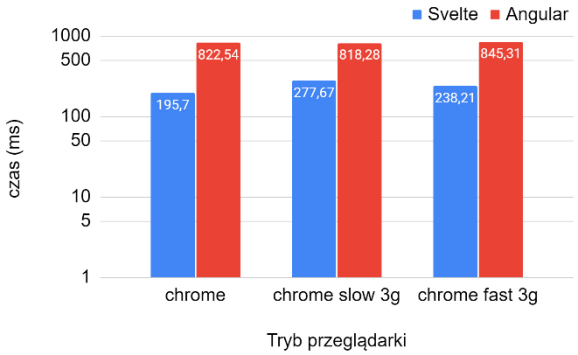
Rysunek 5: Średnie czasy usuwania 10 000 komponentów w zwykłym trybie.



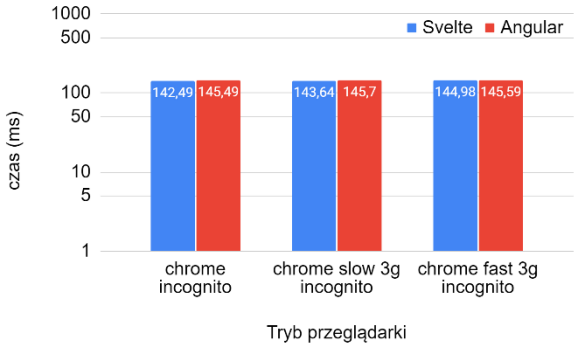
Rysunek 6: Średnie czasy renderowania 100 komponentów w zwykłym trybie.



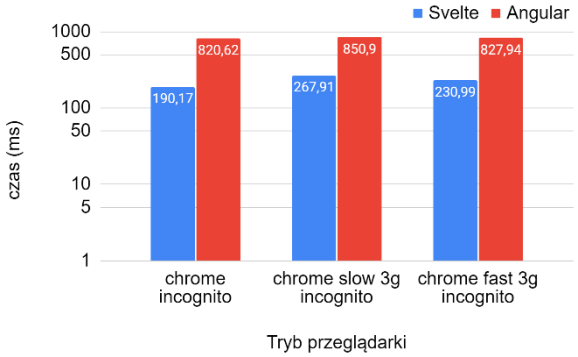
Rysunek 7: Średnie czasy renderowania 1 000 komponentów w zwykłym trybie.



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].