JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 24 (2022) 224–227
Received: 22 May 2022
Accepted: 12 June 2022

Comparative analysis of React, Next and Gatsby programming frameworks for creating SPA applications

Analiza porównawcza szkieletów programistycznych React, Next oraz Gatsby do tworzenia aplikacji typu SPA

Adam Świątkowski*, Karol Ścibior

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

Abstract

This article presents a performance analysis of some of the most popular development frameworks based on the React library. The aim of the study was to show which of the technologies used to create the visual parts of web applications is the most efficient. The research was conducted with the use of 3 applications representing the same research content but based on the above-mentioned frontend technologies. In order to evaluate the performance, web browser development tools and React library were used, which proved that vanilla React is the most efficient for rendering pages with a lot of data.
Keywords: SPA; React; Next; Gatsby

Streszczenie

W niniejszym artykule przeprowadzono analizę wydajnościową jednych z najpopularniejszych szkieletów programistycznych opartych na bibliotece React. Celem badania było wykazanie, która z technologii wykorzystywanych do tworzenia części wizualnych aplikacji internetowych jest najwydajniejsza. Badanie przeprowadzono z wykorzystaniem 3 aplikacji reprezentujących tę samą treść badawczą, ale opartych na tytułowych technologiach typu front-end. W celu oceny wydajności wykorzystano narzędzia deweloperskie przeglądarek internetowych oraz biblioteki React dzięki czemu dowiedziono, że czysty React jest najwydajniejszy przy generowaniu stron o dużej liczbie danych.
Słowa kluczowe: SPA; React; Next; Gatsby

*Corresponding author

Email address: adam.swiatkowski@pollub.edu.pl (A. Świątkowski)

1. Wstęp

Zapotrzebowanie na efektowne strony internetowe rośnie z dnia na dzień. Firmy prześcigają się w tym, by ich wizytówki w postaci stron internetowych prezentowały się jak najlepiej. Z perspektywy programistycznej ważnym aspektem jest, aby proces wytwarzania aplikacji internetowych, był również efektywny. Dzięki podejściu w tworzeniu aplikacji o nazwie SPA (ang. Single Page Application) można osiągnąć obydwa omawiane aspekty. Model SPA pozwala na tworzenie dynamicznych interfejsów użytkownika, co eliminuje konieczność przeładowania strony, przy chęci przejścia na podstronę oraz zapewnia zdecydowaną przewagę w szybkości ładowania treści względem klasycznych stron internetowych [1].

Jedną z najpopularniejszych bibliotek do tworzenia części wizualnych stron internetowych, napisanej w języku JavaScript, jest React. Biblioteka ta została stworzona przez programistów odpowiedzialnych za stworzenie sieci społecznościowej Facebook. Na dzień dzisiejszy pochwalić się ona może ponad 15 mln. pobrań tygodniowo [2].

Technologia Next jest szkieletem programistycznym opartym na bibliotece React. Fundamentalną różnicą względem działania tej technologii względem biblioteki React jest fakt, że treść prezentowana użytkownikowi jest wstępnie generowana na serwerze, dzięki czemu drastycznie zmniejsza się czas potrzebny na załadowanie strony [3].

Technologia Gatsby jest również szkieletem programistycznym opartym na bibliotece React. Głównym założeniem twórców tej technologii była jej zgodność ze wzorcem architektonicznym stworzonym przez firmę Google - PRPL (Push, Render, Pre-cache, Lazy-load). Pozwala to na zauważalne przyspieszenie załadowania treści przez klienta przeglądarkowego [4].

W celu oceny wydajności omówionych technologii wykorzystany zostanie panel deweloperski dostępny w przeglądarkach internetowych Chrome oraz Firefox. Narzędzia deweloperskie w obu przeglądarkach pozwalają na sprawdzenie zarówno czasu ładowania całej strony jak i poszczególnych jej elementów (komponentów).

2. Cel i zakres badań

Celem badań była ocena wydajności wybranych szkieletów programistycznych w porównaniu z biblioteką bazową React. Pozwoli to na sprawdzenie jak różne koncepcje implementacji tej samej biblioteki przenoszą się na rzeczywiste wyniki wydajności. Na tej podstawie wskazane zostanie najwydajniejsze rozwiązanie.

Postawione zostały następujące tezy badawcze:
T1: Szkielet programistyczny Next jest najbardziej wydajny przy tworzeniu stron o umiarkowanej liczbie danych od 100 do 1000 wierszy.
T2: Klasyczny React jest najbardziej wydajny dla aplikacji internetowych o dużej liczbie danych od 10000 wierszy.

3. Przegląd literatury

Analizując dostępną literaturę ciężko znaleźć zestawienie porównujące jednocześnie wszystkie tytułowe technologie. Zdecydowanie łatwiej było uzyskać źródła z wykorzystaniem ogólnodostępnych wyszukiwarek internetowych. Z uwagi na dosyć młody wiek badanych szkieletów programistycznych znalezienie odpowiednich źródeł akademickich może stanowić duże wyzwanie.

Przykładowym artykułem zbieżnym z zakresem badań jest artykuł pt. „create-react-app vs Gatsby.js vs Next.js" [5], w którym autor zestawia ze sobą wymienione szkielety programistyczne i porównuje je ze sobą pod kątem:

- wydajności przy prezentacji danych,
- wymagań w kontekście utrzymania i konserwacji,
- mechanizmów wspierających pozycjonowanie w wyszukiwarkach internetowych,
- częstotliwości aktualizacji wprowadzanych przez twórców technologii.

Wymienione kryteria są zbieżne z zakresem badawczym niniejszej pracy i pozwalają na wskazanie mocnych i słabych stron każdego z narzędzi.

4. Analizowane technologie

Do przeprowadzenia badań zostały wykorzystane technologie oparte na bibliotece React. Były to: zwykły React (szablon CRA (ang. Create React App)), NextJS oraz GatsbyJS. Wszystkie te technologie są do siebie bardzo podobne, jednak zostały stworzone do różnych celów. Proces twórczy aplikacji opartych na bibliotece React bazuje na modelu deklaratywnym, w którym każdy komponent definiuje fragment aplikacji używając składni JSX.

4.1. React

Szablon *create-react-app* to oficjalna, najprostsza metoda do utworzenia projektu aplikacji typu SPA opartego na bibliotece React. Oferuje on nowoczesną i minimalną konfigurację pozwalającą na natychmiastowe rozpoczęcie pracy bez wprowadzania dodatkowych ustawień i narzędzi.

4.2. Next

Szablon NextJS jest zbudowany na bibliotece React, dokładając do niej własne funkcjonalności i optymalizację. Między innymi pozwala na hybrydowe korzystanie ze statycznego generowania widoków oraz generowania widoków po stronie serwera. Dodatkowo posiada takie funkcjonalności jak automatyczna optymalizacja zdjęć, wbudowane funkcje językowe, wsparcie dla biblioteki TypeScript, czy możliwość pisania punktów końcowych REST API.

4.3. Gatsby

Szablon Gatsby również jest zbudowany w oparciu o bibliotekę React, oferując szybkie i elastyczne rozwiązanie do tworzenia stron internetowych opartych na treściach pobieranych z silników typu CMS. Gatsby najczęściej używany jest jako generator stron statycznych dzięki swojej szybkości i natywnym rozwiązaniom służącym do optymalizacji SEO (ang. Search Engine Optimization) czy zwiększania dostępności stron internetowych.

5. Metoda badań

Do przeprowadzenia badań stworzono trzy aplikacje oparte na narzędziach React, Next oraz Gatsby. Projekty utworzono kolejno za pomocą komend:

- *yarn create react-app react-performance-analysis*,
- *yarn create next-app next-performance-analysis,*
- *npm init gatsby gatsby-performance-analysis*.

W każdym z projektów zaimplementowano identyczne, minimalistyczne interfejsy użytkownika zawierające tabelę wyświetlającą dane wygenerowane przy pomocy biblioteki *hoaxer* [6]. Pomiary przeprowadzono korzystając z przeglądarki internetowej Brave [7] w wersji 1.36.119 z silnikiem Chromium w wersji 99.0.4844.83, ze względu na popularność tego silnika.

Listing 1: Kod odpowiedzialny za generowanie danych

```
import hoaxer from 'hoaxer'

export const getMockData = rows =>
 [...Array(rows).keys()].map(index => ({
   id: index + 1,
   name: hoaxer.name.findName(),
   email: hoaxer.internet.email(),
   avatar: hoaxer.internet.avatar(),
   job: hoaxer.name.jobTitle(),
   company: hoaxer.company.companyName(),
   phone: hoaxer.phone.phoneNumber(),
   country: hoaxer.address.country(),
 }))
```

Po zamontowaniu komponentu do struktury przeglądarki DOM (ang. Document Object Model), dane generowane są w sposób synchroniczny oraz ładowane są do stanu komponentu i wyświetlane w tabeli (Listing 2). Dla każdego wiersza wygenerowanych danych, wyświetlany jest wiersz w tabeli, gdzie każda komórka zawiera element HTML typu span z wyświetloną treścią.

Listing 2: Kod odpowiedzialny za załadowanie danych do stanu

```
const [data, setData] = React.useState([])
React.useEffect(() => {
   const mockData = getMockData(1000)
   setData(mockData)
 }, [])
```

Załadowane dane są wyświetlane analogicznie w każdej z napisanych aplikacji za pomocą JSX. Cała tabelka została opakowana w komponent React.Profiler, który pozwala sprawdzić czas w jakim ładowane są jego

komponenty potomne (w tym przypadku jest to tabela). W przypadku aplikacji pomiarowych dla każdego pomiaru wyświetlane są po dwie wartości. Najpierw wyświetlane są pomiary dla pierwszego zamontowania tabelki do strefy DOM (w momencie, gdy tabela jest pusta) oraz drugie wyświetlane w momencie załadowania kolejno 100, 1000 i 10000 wierszy wygenerowanych danych. Najważniejszą wyświetlaną informacją, na której bazowane są pomiary badawcze jest *actualTime*, czyli całkowity czas mierzony od startu do zakończenia ładowania komponentu (Listing 3).

Dzięki temu, że każdy z badanych szablonów programistycznych oparty jest na bibliotece React to sposób deklarowania komponentów jest identyczny. Największe różnice pojawiają się w momencie budowania aplikacji i generowania widoku.

Listing 3: Kod odpowiedzialny za wyświetlanie danych i dokonanie pomiaru

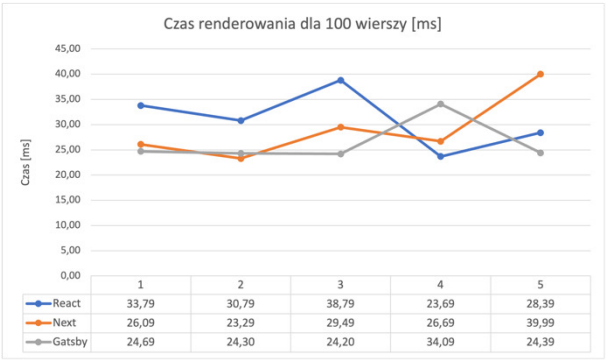
```
<div>
      <header>
        <h1>React Performance Analysis</h1>
      </header>
      <main>
        <React.Profiler id="test"
          onRender={(
            profilerId,
            mode,
            actualTime,
            baseTime,
            startTime,
            commitTime
          ) => {
            console.log({profilerId, mode,
actualTime, baseTime, startTime,commitTime})
          }}
        >
          <table>
            <thead>
              {labels.map(el => (
                <th key={el}>{el}</th>
              ))}
            </thead>
            <tbody>
              {data.map((el, i) => (
                <tr key={i}>
                  {Object.entries(el).map(([key,
label], j) => (
                    <td key={j}>
                      <span>{label}</span>
                    </td>
                  ))}
                </tr>
              ))}
            </tbody>
          </table>
        </React.Profiler>
      </main>
</div>
```

6. Platforma testowa i wyniki badań

Do badań zastosowano środowisko:

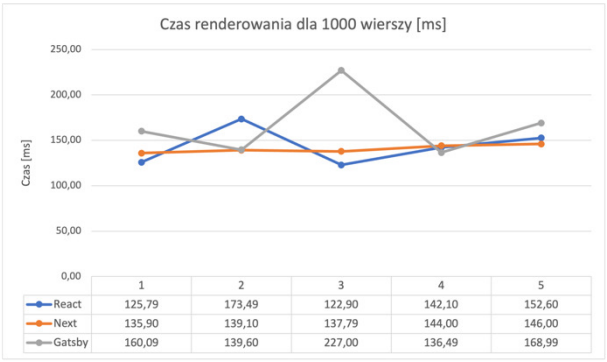
- Przeglądarka internetowa: Brave 1.36.119 Chromium: 99.0.4844.83,
- OS: MacOS Monterey 12.0.1,
- CPU: Intel Core i5 1.4GHz czterordzeniowy,
- RAM: 16GB 2133 MHz LPDDR3,
- GPU: Intel Iris Plus Graphics 645 1536 MB,
- Model: Macbook Pro 13 2020.

Na Rysunkach 1-3 zaprezentowano wyniki pomiarów czasu ładowania strony dla 100, 1000 oraz 10000 wierszy w tabeli.



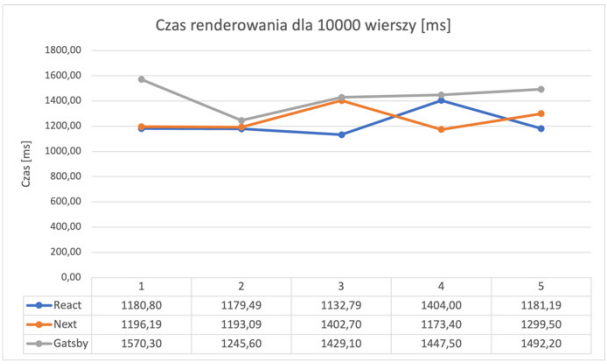
	1	2	3	4	5
React	33,79	30,79	38,79	23,69	28,39
Next	26,09	23,29	29,49	26,69	39,99
Gatsby	24,69	24,30	24,20	34,09	24,39

Rysunek 1: Wyniki pomiarów dla 100 wierszy.



	1	2	3	4	5
React	125,79	173,49	122,90	142,10	152,60
Next	135,90	139,10	137,79	144,00	146,00
Gatsby	160,09	139,60	227,00	136,49	168,99

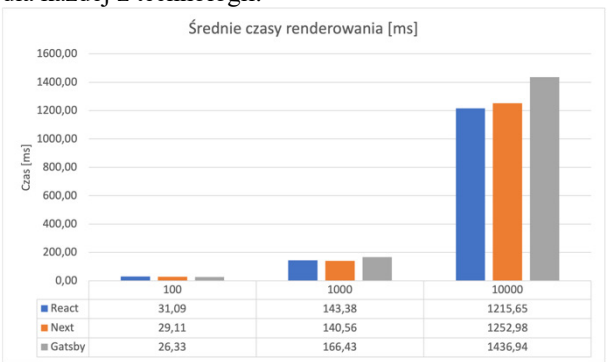
Rysunek 2: Wyniki pomiarów dla 1000 wierszy.



	1	2	3	4	5
React	1180,80	1179,49	1132,79	1404,00	1181,19
Next	1196,19	1193,09	1402,70	1173,40	1299,50
Gatsby	1570,30	1245,60	1429,10	1447,50	1492,20

Rysunek 3: Wyniki pomiarów dla 10000 wierszy.

Rysunek 4 prezentuje średni czas ładowania strony dla każdej z technologii.



	100	1000	10000
React	31,09	143,38	1215,65
Next	29,11	140,56	1252,98
Gatsby	26,33	166,43	1436,94

Rysunek 4: Średnie wyniki pomiarów dla 100,1000 i 10000 wierszy.

7. Analiza wyników

Na podstawie analizy uzyskanych wyników zauważyć można umiarkowane wahania w czasach ładowania strony dla różnej wielkości prezentowanej treści w zależności od badanej technologii.

Dla pierwszej przeprowadzonej serii, czyli badania czasu ładowania strony zawierającej 100 wierszy najlepszą technologią okazał się być szkielet programistyczny Gatsby. Uzyskał on średnio najniższy wynik względem pozostałych badanych technologii – 26,33 ms. Stanowi to różnicę względem technologii React o 4,76 ms. Next uzyskał wynik nieco gorszy od technologii Gatsby i wynosił on 29,11 ms.

Drugą serią badań, były próby czasów ładowania strony zawierającej 1000 wierszy. W tym przypadku patrząc na Rysunek 2, zauważyć można najbardziej stabilną wydajność technologii Next. Szkielet ten okazał się również najbardziej wydajną technologią, dla opisywanej próby i uzyskał średni wynik czasu ładowania strony na poziomie 140,56 ms. React uzyskał wynik na poziomie 143,38 ms, a Gatsby na poziomie 166,43 ms.

W trzeciej serii badań, czyli próbie czasów ładowania strony zawierającej 10000 wierszy, różnice czasów między technologiami stały się bardziej zauważalne. Najbardziej wydajna technologia w tej próbie, czyli React, okazała się być lepsza od najmniej wydajnej średnio, aż o 221,29 ms. Tym samym, najwydajniejszą technologią dla najtrudniejszej z prób okazała się być biblioteka React uzyskując średni wynik na poziomie 1215,65 ms.

8. Wnioski

Wyniki badań pokazują umiarkowane dysproporcje w wydajności badanych technologii względem liczby generowanych danych w widoku aplikacji testowej, do której je zastosowano.

Każdy z badanych szkieletów programistycznych bazuje na bibliotece React. Można więc sądzić, że każdy z autorów technologii Next oraz Gatsby, chciał ulepszyć bibliotekę React tworząc dla niej nakładkę, tak by uzyskać jak najlepsze i najbardziej wydajne działanie w różnych scenariuszach. Aplikacja testowa stworzona na potrzeby badań imituje właśnie scenariusze dla prostej aplikacji, gdzie zmienia się jedynie liczba wyświetlanych danych.

Każda z technologii okazała się świadczyć o tym czym kierowali się jej twórcy, co potwierdzają wyniki dwóch pierwszych serii, gdzie wygrały szkielety Next oraz Gatsby. Szkielety te nadają się idealnie do prostych aplikacji internetowych z niewielką ilością danych wyświetlanych na jednej stronie. Wyniki badań częściowo potwierdzają tezę T1 postawioną w niniejszym artykule. Technologia Gatsby okazała się najwydajniejsza dla stron o najmniejszej ilości danych, natomiast dla stron o umiarkowanej ilości danych najlepszy okazał się szkielet programistyczny Next.

Dla aplikacji przedstawiającej dużą liczbę danych najwydajniejsza okazuje się czysta biblioteka React, co potwierdza tezę T2. Warto zauważyć jednak, że wyświetlanie aż tak dużej ilości danych w aplikacjach internetowych, bez użycia metod optymalizacyjnych takich jak ładowanie na żądanie (ang. on-demand loading), znane również jako leniwe ładowanie (ang. lazy loading), lub wirtualizacji. Optymalizacja przez programistę tworzonego kodu, na pewno zmniejszyłaby różnice w wydajności poszczególnych technologii i bibliotek.

Wyniki badań są również zbieżne z wynikami badań przeprowadzonych w artykule pt.: „create-react-app vs Gatsby.js vs Next.js” [5], w którym to autor wskazał technologię Next jako najwydajniejszą, w zestawieniu z pozostałymi technologiami, ze względu na użycie w niej metody generowania widoku po stronie serwera, co przekłada się na wzrost szybkości jej ładowania. Autor zwrócił jednak uwagę, że metoda generowania widoków po stronie serwera jest dość obciążająca, co może przełożyć się na wydajność ładowania widoków, stron o dużej liczbie danych, powyżej 10000 wierszy.

Literatura

[1] E. Scott, SPA Design and Architecture: Understanding Single Page Web Applications, Manning Publications, 2015.

[2] Dokumentacja React, https://pl.reactjs.org/docs/getting-started.html, [22.04.2022].

[3] Dokumentacja Next, https://nextjs.org/docs, [22.04.2022].

[4] Dokumentacja Gatsby, https://www.gatsbyjs.com/docs/, [22.04.2022].

[5] CRA vs Gatsby vs Next, https://www.sensilabs.pl/2020/12/15/create-react-app-vs-gatsby-js-vs-next-js/, [22.04.2022].

[6] Biblioteka Hoaxer, https://www.npmjs.com/package/hoaxer, [22.04.2022].

[7] Przeglądarka internetowa Brave, https://brave.com/, [22.04.2022].

[8] J. Wagner, Web Performance in Action: Building Faster Web Pages, Manning Publications, 2017.

[9] C. R. Adams, Mastering JavaScript High Performance, Packt Publishing, 2015.

[10] R. Nowacki, M. Plechawska-Wójcik, Analiza porównawcza narzędzi do budowania aplikacji Single Page Application - AngularJS, ReactJS, Ember.js, Journal of Computer Sciences Institute 2 (2016) 98-103.

[11] A. M. Vipul, P. Sonpatki, ReactJS by Example - Building Modern Web Applications with React, 2016.