JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 34 (2025) 68–75
Received: 16 October 2024
Accepted: 1 December 2024

Multi-aspect comparative analysis of JavaScript programming frameworks - React.js and Solid.js

Wieloaspektowa analiza porównawcza szkieletów programistycznych języka JavaScript - React.js i Solid.js

Jakub Kryk*, Małgorzata Plechawska-Wójcik

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

Abstract
Every year, new frameworks and programming libraries appear on the market, trying to gain user interest by presenting their advantages over the competition. The purpose of this article was to compare many aspects of one if such modern solution, i.e. the Solid.js library with a tool recognized on the market, the React.js library, in order to be able to form an opinion on the better choice for a programmer in 2024. The comparative analysis was divided into two parts – performance and theoretical. The research showed Solid.js' performance advantage, but also highlighted its immature ecosystem and limited number of active people in the community.

Keywords: web applications; programming frameworks; programming libraries; application performance

Streszczenie
Co roku na rynku pojawiają się nowe szkielety i biblioteki programistyczne starające się pozyskać zainteresowanie użytkownika prezentując swoje zalety nad konkurencją. Celem tego artykułu było porównanie wielu aspektów jednego z takich nowoczesnych rozwiązań, czyli biblioteki Solid.js oraz uznanego na rynku narzędzia, biblioteki React.js, by móc postawić opinię na temat lepszego wyboru dla programisty w 2024 roku. Analiza porównawcza została podzielona na dwie części – wydajnościową i teoretyczną. Badania wykazały przewagę Solid.js na polu wydajności, lecz podkreśliły również jej niedojrzały ekosystem i ograniczoną liczbę aktywnych osób w społeczności.

Słowa kluczowe: aplikacje internetowe; szkielety programistyczne; biblioteki programistyczne; wydajność aplikacji

*Corresponding author
Email address: **jakub.kryk@pollub.edu.pl** (J. Kryk)

1. Wstęp

W dzisiejszym dynamicznie zmieniającym się świecie, gdzie tworzenie aplikacji internetowych jest integralną częścią cyfrowej rzeczywistości, narzędzia programistyczne, w tym szkielety i biblioteki odgrywają kluczową rolę w ułatwianiu i przyspieszaniu procesu tworzenia innowacyjnych i efektywnych rozwiązań. Z roku na rok rynek tychże technologii nieustannie się rozwija i stawia przed programistami nowe wybory dotyczące narzędzi, przy których pomocy mogą realizować postawione przez siebie cele.

Ważną częścią każdej aplikacji internetowej jest jej strona wizualna. Odpowiada ona za kontakt pomiędzy logiką aplikacji a użytkownikiem gotowym wykorzystać jej potencjał. Jest ona nieodłącznym elementem komercyjnych rozwiązań, oprócz części użytkowej przedstawiana jest również jako wizytówka gotowego rozwiązania. Dlatego też każda osoba odpowiedzialna za rozwój oprogramowania powinna zwrócić uwagę, by ta jego część była dopracowana, wydajna i przyjemna dla oka. Aby wyjść naprzeciw wciąż rosnącym wymaganiom klientów, programiści tworzący tego typu aplikacje używają bibliotek czy szkieletów programistycznych mających na celu ułatwienie oraz przyśpieszenie pracy w budowaniu gotowego rozwiązania. Oferują one narzędzia pozwalające w krótkim czasie stworzyć ogólny zarys architektoniczny aplikacji, zapewniają lepszą modularność i reużywalność kodu, przez co koder może skupić się na rozwoju unikalnych funkcjonalności aplikacji, posiadając istniejące rozwiązania typowych problemów.

Popularność takich technologii powoduje, że na rynek, w dużych ilościach, trafiają rozwiązania bardzo do siebie podobne. Nawet doświadczeni programiści stają często przed trudnym wyborem odpowiedniego narzędzia, nie znając jego wszystkich zalet i wad. Z kolei osoby zaczynające przygodę z pisaniem aplikacji klienckich w największej liczbie przypadków nie potrafią sami ocenić mocnych i słabych stron potencjalnego kandydata. Dlatego popularnym trendem stało się tworzenie artykułów internetowych czy filmów publikowanych w przeróżnych serwisach, mających na celu rozjaśnienie sytuacji na rynku. Niekiedy jednak autorzy tego typu źródeł wiedzy nierzetelnie lub niejasno przedstawiający swoje zdanie mogą wprowadzić swojego odbiorcę w błąd. Jest to szczególnie prawdopodobne, w przypadku omawiania nowych technologii, gdzie brakuje badań na ich temat.

Aby spróbować sprostać problemowi wynikającemu z przedstawionej w poprzednim akapicie sytuacji, artykuł ten porusza temat wieloaspektowej analizy szkieletów programistycznych dla języka JavaScript. W niniejszej opracowaniu przedstawione zostanie porównanie dwóch technologii, jednej uznanej i z głęboko zakorzenioną

renomą oraz drugiej stosunkowo nowej i dopiero wzbudzającej szersze zainteresowanie świata programistów aplikacji klienckich. Analiza obu narzędzi zostanie podzielona na przeprowadzenie badania wydajności podobnych do siebie aplikacji napisanych za pomocą obu z nich oraz na omówieniu kluczowych aspektów obu technologii w kontekście budowy współczesnego rozwiązania zgodnego ze standardami branży.

2. Cel i zakres badań

Celem niniejszego badania było przeprowadzenie zróżnicowanej analizy porównawczej bibliotek programistycznych React i Solid. Analiza ta została podzielona na dwie części, jedna z nich skupiała się na badaniu wydajności bliźniaczych do siebie aplikacji, napisanych za pomocą szkieletów programistycznych dostępnych dla wymienionych technologii. Programy te służyły generowaniu i edycji list oraz sprawdzaniu czasów, w jakich renderują się one w przeglądarce internetowej. Aplikacje posiadały również funkcję wyświetlania wykresów zbierających wyniki przeprowadzonych testów. Dalszy etap prac skupił się na przedstawieniu i porównaniu różnic architekturalnych oraz niuansów technicznych danych narzędzi. Dodatkowo poruszony został temat zaangażowania społeczności i trudności wdrożenia w każdą z technologii. Wnioski wyciągnięte w procesie badania i omawiania każdego z tych aspektów pozwoliły na określenie zalet i wad danych narzędzi i pomogły rozpoznać, które z nich jest bardziej interesującym wyborem dla programisty w 2024 roku.

Punktem wyjścia do dalszych badań były poniższe tezy, które w pracy stały się przedmiotem weryfikacji przeprowadzonych analiz:

T1: Szkielet programistyczny Solid.js prezentuje lepszą ogólną wydajność od szkieletu React.js.

T2: Niezależnie od wydajności szkieletu programistycznego Solid.js, React.js dla programisty nadal jest bardziej konkurencyjną technologią przy wyborze narzędzia w 2024 roku.

3. Przegląd literatury

Technologie służące do tworzenia aplikacji klienckich systemów informatycznych doświadczają ciągłego rozwoju, prowadzącego do powstania nowych rozwiązań coraz lepiej wpasowujących się w wymagania i przyzwyczajenia programistów. W tej ewolucji wyróżniają się narzędzia takie jak React.js i Angular, stworzone przez renomowane firmy: Facebooka i Google, od lat stojące na szczytach każdej listy popularności [19]. Równolegle z tymi dużymi szkieletami czy bibliotekami rywalizuje szereg mniejszych rozwiązań, starających się przyciągnąć uwagę deweloperów poprzez innowacyjne możliwości, zwiększoną wydajność czy optymalizację. Jednym z takich zyskujących na znaczeniu narzędzi, jest Solid.js, który w niniejszym przeglądzie literatury zostanie porównany z wymienioną wcześniej biblioteką React.js. Mimo braku formalnych prac naukowych porównujących Solid.js i React.js, eksploracyjna analiza społeczności programistycznej, forów dyskusyjnych, blogów oraz danych z serwisu GitHub może dostarczyć cennych informacji na temat popularności i funkcjonalności obu narzędzi.

Pomimo podobieństw w koncepcji, różnice w popularności są wyraźne. Prezentują je wykresy ze zbiorczymi danymi zebranymi anonimowo od użytkowników platformy GitHub [19]. Ciekawą obserwacją jest wyższy poziom wskaźnika pożądania tej mniej popularnej technologii, co przekłada się na przeświadczenie, że mimo krótkiego stażu na rynku Solid.js znajduje grono osób doceniających jego mocne strony i chętnych na ich wykorzystanie przy rozwoju swoich nowych projektów. Jest to z dużą dozą pewności związane z wybitną wydajnością i szybkością działania tego narzędzia, mocno przewyższającą obecną konkurencję.

Za ten aspekt odpowiada system opisany przez Nathana Babcocka [25], który po polsku można określić jako „drobnoziarnista reaktywność". Odpowiada on za zdolność reagowania systemu na najdrobniejszą, szczegółową zmianę, jaka wystąpiła w małym fragmencie danych, odciążając w tym wypadku dewelopera, zwalniając jego zasoby pracy od implementacji odpowiedzi systemu na występujące zdarzenia, które obsługiwane są natychmiastowo przez platformę.

Użyteczność narzędzia nie powinna być jednak oceniana na podstawie jednej metryki. Analizując pracę zespołu pod przewodnictwem Marina Kaluży [9] możemy wysnuć wnioski, że szybsze działanie jednego narzędzia nie jest jedynym ważnym aspektem, na podstawie którego warto analizować użyteczność danej technologii. Wieloaspektowe badanie zespołu dostarczyło danych, dzięki którym można zaobserwować w jakich aspektach specjalizuje się dany szkielet programistyczny i gdzie jego wykorzystanie przynosi najlepszy efekt.

Z kolei Marcin Golec [2] pod merytoryczną opieką Małgorzaty Plechawskiej-Wójcik badał wydajność szkieletów programistycznych wykorzystujących TypeScript przy budowie aplikacji serwerowych. Analizę swą opierał na porównywaniu czasów odpowiedzi na dziewięć różnych rodzajów żądań, jakie otrzymywały aplikacje. W rezultacie otrzymano wyniki wyraźnie przedstawiające wyższą wydajność jednego z rozwiązań, wyraźnie przodującego w większości badanych metryk. Analiza ta podnosi istotne kwestie związane z czasami odpowiedzi na różne rodzaje żądań, co może być ważnym kryterium wyboru technologii przy budowie aplikacji zarówno serwerowej, jak i internetowej.

Porównanie aplikacji webowych było tematem pracy [7]. Jej głównym założeniem również było zbadanie wydajności każdego z rozwiązań. Wśród obserwowanych technologii znalazł się React.js [13]. W rezultacie otrzymano dane wskazujące na niską wydajność tego narzędzia, odnosząc się do jego słabych wynikach w testach wydajnościowych. Zwrócono

również uwagę na potrzebę utworzenia znacznie większej ilości kodu podczas jego używania, szczególnie porównując sytuację w odniesieniu do reszty narzędzi wchodzących w skład zestawienia.

Praca autorów pod przewodnictwem Songtao Chena [4] przedstawia odmienny pogląd na technologię React, starając się zrozumieć i przeanalizować jego popularność. W rezultacie autorzy doszli do wniosku, że mocną stroną tego narzędzia jest jego otwartość, prostota użytkowania i wiele innych aspektów takich jak wirtualny DOM czy specjalna składnia, które docenili użytkownicy na całym świecie, budując wokół niego dużą, dobrze zorganizowaną społeczność.

Większość pozostałych pozycji, takich jak przykładowo praca C. L. Mariano [6] bądź książka A. Banksa i E. Porcello [10] rozwija wiedzę na temat technologii badanych w tej pracy naukowej. Na podstawie istniejących dzieł analizujących podobne technologie można jednak wyodrębnić obszary, jakie zasługują na uwagę przy ocenie danego narzędzia. Niezbędne są również dalsze badania poszerzające wiedzę na temat Solid.js i React.js, ponieważ liczba obecnie wykonanych nie jest wystarczająca, by na ich podstawie sformułować znaczące wnioski.

4. Omówienie technologii użytych do wykonania badań

Do przygotowania badań wymienionych w tej pracy potrzebne było wyselekcjonowanie dwóch kandydatów spośród bogatej rodziny szkieletów i bibliotek programistycznych. Wybór ugruntowanego na rynku narzędzia był nieskomplikowany, jedynie kilka z nich konsekwentnie podbija wszelkie ankiety popularności [19], a doświadczenie w pracy z biblioteką React ostatecznie przeważyło w rozstrzygnięciu decyzji. Dobór konkurenta w porównaniach był jednak trudniejszy. Działanie zgodne z wcześniej opisanym założeniem sprawdzenia popularnej technologii z narzędziem względnie nowym na rynku wymagało skrupulatnego przeglądu branży. Ostatecznie wybór padł na Solid.js, głównie ze względu na reklamowaną przez jego twórców wydajność oraz wysokie indeksy zadowolenia twórców z pracy z wykorzystaniem go w swoim projektach [19]. Warto również odnotować, że w literaturze technicznej React.js i Solid.js są kwalifikowane jako biblioteki języka JavaScript, jednak dzięki rozbudowanym ekosystemom oraz strukturalnym konwencjom stosowanym przy budowie aplikacji w praktyce spełniają one rolę szkieletów programistycznych. Dodatkowo aplikacje testowe napisane na potrzeby tej pracy zostały stworzone przy pomocy dedykowanych szkieletów programistycznych dla tych technologii – create-react-app i create-solid. Zatem w niniejszej pracy będziemy używali pojęcia biblioteka i szkielet programistyczny zamiennie.

4.1. React.js

React [13] to popularna biblioteka JavaScript, stworzona przez firmę Facebook w 2013 roku. Programistom tej firmy przyświecał cel ułatwienia tworzenia nowoczesnych i interaktywnych interfejsów użytkownika. Głównym punktem tego narzędzia jest komponentowy model. Komponent to reużywalna część kodu, zarządzająca własnym stanem, co pozwala na jego wielokrotne wykorzystanie w różnych miejscach aplikacji. Od standardowych szkieletów programistycznych różni się podejściem do struktury projektu i do problemów architektonicznych, dając większą swobodę programiście. Przyjęte standardy w branży sprawiają jednak, że pomimo braku sztywnych ram, wykorzystywany jest w ten sam sposób jak inne szkielety programistyczne, walidując zdanie, jako że można go nim nazywać i tak też zostanie potraktowany w tej pracy.

Jednym z cech charakterystycznych Reacta jest zastosowanie nakładki składniowej modyfikującej język JavaScript, szeroko używanej wśród programistów korzystających z tej technologii. JavaScriptXML, znany szerzej jako JSX, łączy działanie JavaScripta z językiem znacznikowym – HTML, usprawniając proces pisania kodu i zmniejszając jego objętość.

Kolejnym z kluczowych elementów tego szkieletu programistycznego jest Virtual DOM, czyli wirtualna kopia prawdziwego drzewa DOM, będącego reprezentacją dokumentu HTML jako węzłów. Takie podejście pozwala na efektywne aktualizowanie zmian w interfejsie, dokonując wymiany tylko tych elementów prawdziwego drzewa, które zostały zmodyfikowane.

React zyskał ogromną popularność i wyrobił sobie renomę na rynku szkieletów programistycznych. Doceniana jest jego prostota i uniwersalność. Jest wspierany przez dużą społeczność i ma bogaty ekosystem narzędzi oraz bibliotek dodatkowo rozszerzających jego możliwości. Stał się on filarem wielu nowoczesnych aplikacji internetowych i rzesza jego fanów nieprzerwanie rośnie.

4.2. Solid.js

Solid [14] to stworzona przez Ryana Carniato wydajna biblioteka zaprojektowana z myślą o optymalizacji wydajności i dbająca o efektywne wykorzystanie zasobów systemowych w procesie budowania interfejsu użytkownika. Główną ideą Solida jest cząsteczkowa reaktywność. Termin ten odnosi się do sposobu zarządzania aktualizacjami drzewa komponentów na bardzo precyzyjnym poziomie. Wszelkie zmiany wykonywane są bezpośrednio na „cząsteczkach” i porzucona zostaję idea wirtualnego drzewa DOM zaprezentowana w bibliotece React. Wynikiem takiego podejścia do problemu jest zwiększona wydajność, żaden dodatkowy transpiler nie musi dodatkowo zużywać zasobów na przeliczanie wykonanych korekt na wirtualnym drzewie węzłów.

Solid.js pomimo wielu zapożyczeń od Reacta porzuca również zasady hooków obecnej u swojego bardziej dojrzałego konkurenta. Takie podejście pozwala na jednorazowe wykonanie komponentu i kontrolowanie jego stanu za pomocą zmian zależności wyrażeń i prymitywów dostępnych dla programisty.

Opisywane narzędzie, które powstało w 2018 roku, mimo stosunkowo niewielkiego stażu na rynku, zyskuje popularność i uznanie deweloperów stawiających na wysoką wydajność tworzonych rozwiązań. Jego ekosystem nadal znajduje się w fazie dynamicznego rozwoju, przez co dostępność dedykowanych narzędzi nie jest tak wysoka jak w przypadku Reacta.

4.3. Pozostałe technologie

Do budowy aplikacji testowych wykorzystano również wiele innych narzędzi. Jednym z nich była biblioteka JavaScript Faker.js, pozwalająca na generowanie losowych danych, które służyły jako obiekt badań nad wydajnością aplikacji. Aplikacje zostały przetestowane na przeglądarkach Mozilla Firefox i Google Chrome w najnowszych stabilnych wersjach. Obie zostały napisane w edytorze kodu źródłowego – Visual Studio Code, który zawiera wbudowane wsparcie dla języka JavaScript i pozwala na dowolne rozszerzanie swoich funkcjonalności przy pomocy licznych dostępnych wtyczek.

5. Metoda badań

5.1. Środowisko testowe

Poniższa tabela (Tabela 1) przedstawia środowisko testowe użyte do przeprowadzenia badań. Zawiera informacje o komputerze użytym do badań oraz o narzędziach niezbędnych do przeprowadzenia testu.

Tabela 1: Specyfikacja sprzętowa urządzenia wykorzystywanego w procesie badania wydajności aplikacji

Sprzęt	
Procesor	Intel I5-9300H
Karta graficzna	Nvidia GeForce GTX1650
Pamięć RAM	16GB 2666MHz CL 16
System operacyjny	Windows 10 v.10.0.1945
Biblioteki programistyczne	
React.JS	18.2.0
Solid.JS	1.8.16
Narzędzia	
Mozilla Firefox	124.0.1
Google Chrome	129.0.6668.90
Faker.JS	8.4.1

5.2. Opis eksperymentu

Eksperyment badawczy został podzielony na dwie części i składał się z badań wydajnościowych przedstawionych narzędzi oraz na teoretycznym omówieniu różnic i cech obu technologii. Taka struktura pracy miała za zadanie umożliwić kompleksowe podejście do tematu, dając pełniejszy obraz użyteczności i zastosowania dla obu badanych narzędzi.

W celu przeprowadzenia badań wydajnościowych utworzono dwie aplikacje z zaimplementowanymi identycznymi funkcjonalnościami, które w jak największym stopniu są zbliżone do siebie budową i wykorzystaną architekturą. Obie aplikacje posiadają podobne interfejsy użytkownika, różniące się jedynie kolorem tła. Ma to na celu łatwiejsze odróżnienie ich od siebie. Utworzone programy mają na celu zmierzenie czasów renderowania tabeli przedstawiającej wygenerowane dane oraz porównanie czasów ponownego przygotowania gotowego interfejsu w przypadku edycji jednej z kolumn tabeli. Do generowania danych wykorzystano bibliotekę Faker,js. Narzędzie to pozwoliło dynamicznie generować liczbę pakietów testowych i tworzyć zróżnicowane scenariusze testowe. Listing 1 przedstawia strukturę przygotowywanych danych.

Listing 1: Fragment kodu zawierający funkcję odpowiedzialną za generowanie danych

```
const generateData = (count) => {
    unicodeEmoji.getEmojis()
    const data = [];
    for (let i = 0; i < count; i++) {
      const instance = {
        name: faker.person.fullName(),
        email: faker.internet.email(),
        address: faker.location.streetAddress(),
        city: faker.location.city(),
        country: faker.location.country(),
        emoji: "😀"
      };
      data.push(instance);
    }
    console.log(data);
    return data;
  };
```

Przygotowane dane przekazywane były do komponentu renderującego poprzez element „props”, ładowane do jego stanu, a następnie wyświetlane przed interfejs w postaci tabeli na stronie dostępnej dla użytkownika poprzez przeglądarkę internetową. Ten sam komponent zawierał również mechanizm obliczania czasu renderowania i malowania zawartości na stronie (Listing 2). Opierał się on na użyciu kombinacji hooka useEffect w React.js lub createEffect dla Solid.js oraz funkcji JavaScript o nazwie requestAnimationFrame, do której podłączone było API window.performance. Gdy dane wejściowe otrzymywane przez komponent ulegały zmianie, uruchamiał się efekt wywoływany przez useEffect. Na początku rejestrował on początkowy czas jeszcze przed rozpoczęciem procesu renderowania, następnie wywoływał requestAnimationFrame, funkcję służącą do efektywnego synchronizowania cyklu działania komponentu z odświeżaniem przeglądarki. Pierwsze wywołanie tej funkcji ustawiało ramkę dokładnie w momencie, kiedy komponent został wstępnie wyrenderowany, ale nie został jeszcze narysowany. Dlatego drugie wywołanie wcześniej wymienionej funkcji było niezbędne. Przed rozpoczęciem działania sprawdzała ona referencję do komponentu i jeżeli nadal istniał proces był kontynuowany. Po wykonaniu się funkcji drugi raz

otrzymywany zostawał dokładny moment narysowania komponentu, który był rejestrowany jako czas końcowy.

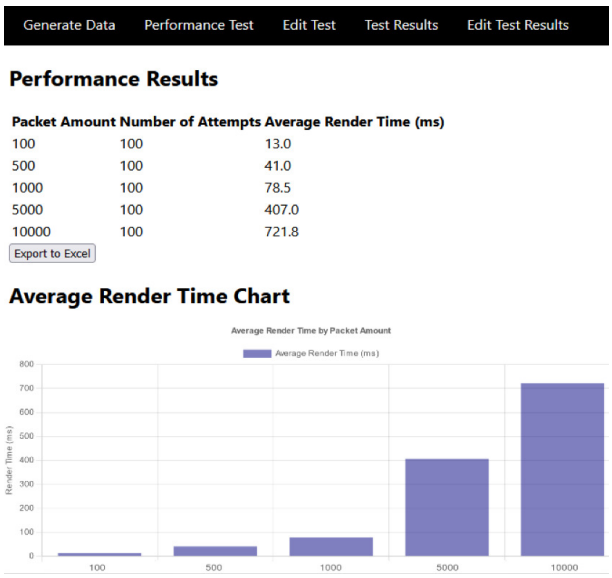
Listing 2: Fragment kodu przedstawiający mechanizm obliczania czasu renderowania komponentu

```
useEffect(() => {
  const start = window.performance.now();

  requestAnimationFrame(() => {
    if (componentRef.current) {
      requestAnimationFrame(() => {
        const end = window.performance.now();
        const timeElapsed = end - start;
        setRenderTime(timeElapsed);

        const existingHistory = JSON.parse(localStorage.getItem('renderTimeHistory')) || [];
        const updatedHistory = [...existingHistory, { repeatAmount: props.amount,
          renderTime: parseFloat(timeElapsed.toFixed(3)) }];
        localStorage.setItem('renderTimeHistory', JSON.stringify(updatedHistory));
      });
    }
  });
}, [props.data, props.amount]);
```

Różnica pomiędzy czasem końcowym, a początkowym dla danego renderowania zapisywana była jako element tablicy w localStorage, skąd można było pobrać historię analizowanych czasów za pomocą funkcji ukrytej pod przyciskiem w aplikacji, na stronie prezentującej wyniki pomiarów (Rysunek 1). Bliźniaczy proces wykonywany był w przypadku mierzenia czasu edycji kolumny. Otrzymane dane zostały przetworzone i przeanalizowane, a wyniki eksperymentu zaprezentowane w tej pracy.

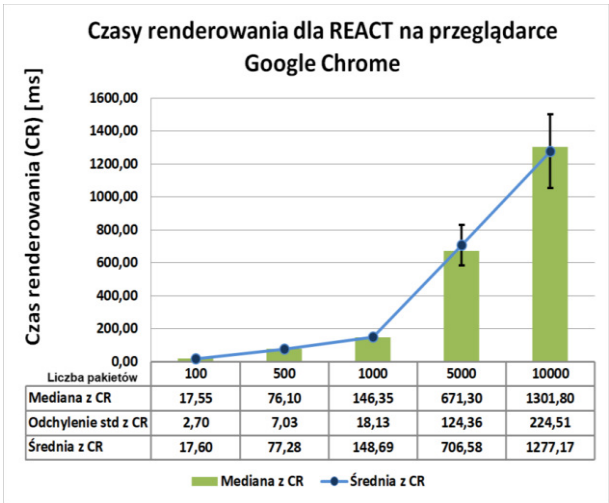


Rysunek 1: Zrzut ekranu przedstawiający wygląd interfejsu użytkownika na jednej z podstron aplikacji.

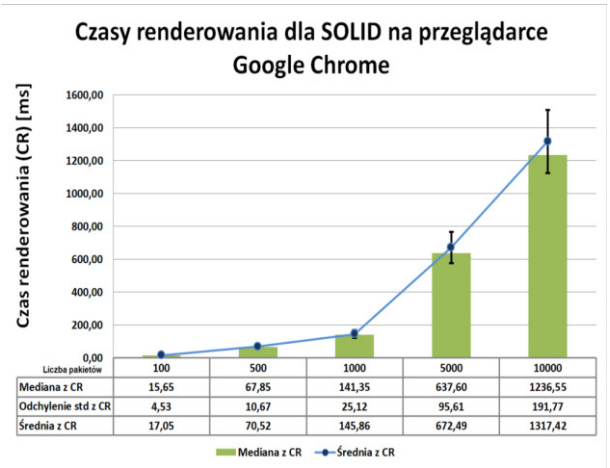
Drugą częścią analizy było omówienie teoretycznych różnic pomiędzy dwoma technologiami. Każde z zagadnień odnosiło się do kluczowych dla programisty aspektów i służyło do odnalezienia lepszego rozwiązania w tych dziedzinach. Ostatecznie postanowiono skupić się na badaniu zależności aplikacji w pięciu obszarach przedstawionych w dalszej części pracy.

6. Wyniki analizy wydajnościowej

Analiza obejmowała porównanie czasów renderowania i ponownego renderowania po edycji dla utworzonych pakietów. Tabele z pakietami ładowano odpowiednio grupami 100, 500, 1000, 5000 i 10000 jednostek. Dla pogłębienia wiedzy na temat fluktuacji czasów renderowania wykonano również analizę mediany i odchylenia standardowego pomiarów, przykład takiej analizy widoczny jest na Rysunku 2 i Rysunku 3, gdzie przedstawiono wykresy zawierające średnie czasy renderowania dla danej technologii w jednej z przeglądarek oraz ich medianę oraz odchylenie standardowe, pozwalające lepiej zrozumieć finalne dane, przekazując informację o ich typowej wartości i zmienności.

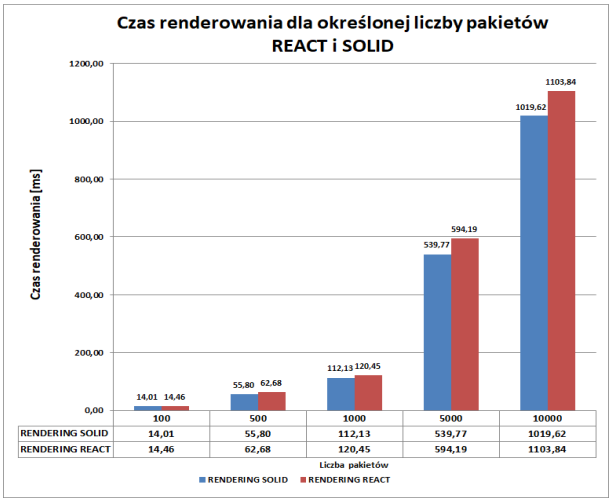


Rysunek 2: Wyniki analizy pomiarów dla testów renderowania technologii React na przeglądarce Google Chrome.

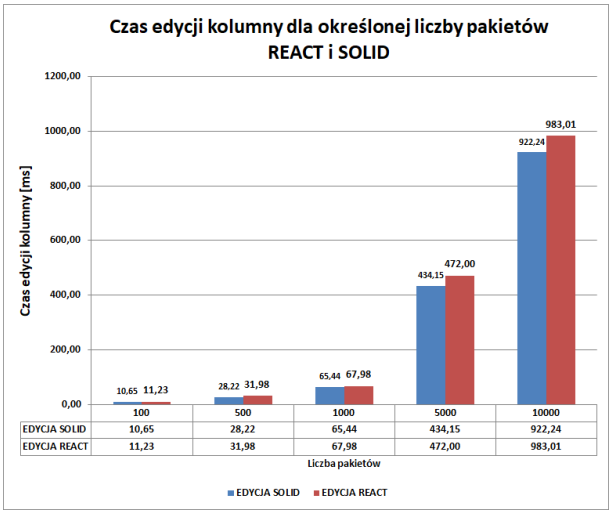


Rysunek 3: Wyniki analizy pomiarów dla testów renderowania technologii Solid na przeglądarce Google Chrome.

Kolejne ilustracje przedstawiają główne wykresy zawierające informacje o wydajności poszczególnych szkieletów programistycznych (Rysunek 4 i Rysunek 5). Dają one dostęp do informacji o uśrednionych czasach renderowania z wszystkich przypadków testowych, informują o ogólnej wydajności tych technologii, zarówno podczas testów pierwszego renderowania, jak i testów ponownego renderowania po edycji kolumny.



Rysunek 4: Uśrednione czasy renderowania dla poszczególnej liczby pakietów przy użyciu bibliotek React i Solid.



Rysunek 5: Uśrednione czasy ponownego renderowania po edycji kolumny z danymi dla poszczególnej liczby pakietów przy użyciu bibliotek React i Solid.

Na podstawie przedstawionych wyników badań można zaobserwować zauważalną różnice w zmierzonych czasach pierwszego renderowania oraz renderowania po edycji kolumny z przewagą dla szkieletu programistycznego Solid.js. Pierwsze renderowanie, w zależności od liczby pakietów jest dla tej technologii wydajniejsze od 4 do 11 procent, wtórne renderowanie po zmianie zawartości kolumny o od 4 do 12 procent. Wyniki szczegółowe jednak, przedstawione dla jednego z przypadków testowych wskazują, że mogą wydarzyć się sytuacje, w których to drugi ze szkieletów daje lepszy wynik. Jest to jednak przypadek nieczęsty, gdyż zbiorcze zestawienie zmierzonych czasów nie pozostawia wątpliwości o przewadze wydajnościowej narzędzia Solid.js.

7. Teoretyczna analiza porównawcza

Teoretyczne omówienie kluczowych aspektów badanych szkieletów programistycznych było równie ważnym aspektem, co uprzednio opisany eksperyment wydajnościowy. Pozwala ono na lepsze zrozumienie słabszych i mocnych stron rozwiązań technicznych poszczególnych narzędzi, zrozumienie ich perspektyw rozwoju oraz poziomu zaangażowania społeczności w jej zmiany. Takie podejście dostarcza większej ilości informacji, ułatwiając potencjalnemu użytkownikowi podjęcie odpowiedniej decyzji podczas wyboru odpowiedniej dla siebie platformy.

7.1. Architektura i filozofia projektowa

Obie biblioteki, pomimo wielu podobieństw implementacyjnych, reprezentują odrębne podejścia do architektury i ogólnej filozofii projektowej. Główną koncepcją architekturalną narzędzia React jest wirtualne drzewo DOM [13] jako warstwa pośrednia pomiędzy aktualnym stanem aplikacji a drzewem DOM przeglądarki (rzeczywistym elementem DOM). Po zmianie stanu komponentu React aktualizuje najpierw swoje wirtualne drzewo, następnie porównuje każdy element starej i nowej wersji drzewa. Proces ten nazywany jest rekoncyliacją. Następnie identyfikuje jakie zmiany pomiędzy nimi należy wprowadzić. Takie podejście ma swoje zalety oraz wady, operowanie na dodatkowej warstwie prowadzi do początkowej utraty wydajności w porównaniu z rozwiązaniami nieposiadającymi wirtualnego drzewa, lecz bardzo duże i kosztowne operacje mocno obciążające rzeczywiste drzewo przeglądarki są dzięki takiej koncepcji minimalizowane. Solid z kolei jest reprezentantem przeciwnej idei wprowadzania zmian do drzewa DOM przeglądarki. Działa bezpośrednio na nim i unika korzystania z jej wirtualnej reprezentacji. Każda zmiana jest automatycznie przeliczana i odwzorowywana w odpowiednim fragmencie realnego drzewa, proces ten nazywany jest cząsteczkową reaktywnością. Eliminacja dodatkowej warstwy i potrzeby przeliczania zmian pomiędzy nimi powoduje, że w przypadku małych, nieobciążających zmian Solid posiada bardzo wysoką wydajność, mocno zbliżoną do czystego języka JavaScript. Jednak w przypadku pracy na dużych zbiorach danych oraz generowania znacznej liczby przeliczeń drzewa DOM przeglądarki różnica wydajności pomiędzy oboma szkieletami powinna się wyrównywać.

Pomimo tej znaczącej różnicy obie technologie prezentują zbliżone rozwiązania w aspekcie budowy aplikacji przez programistę. W pewnym stopniu jest to wypadkowa bezpośredniej inspiracji narzędziem React przez twórców młodszego rozwiązania. Jednym z takich podobieństw jest używanie komponentów funkcyjnych oraz nakładki JSX na język Javascript, pozwalającej na używanie znaczników HTML w procesie pisania kodu [16]. Obie technologie modyfikują stan komponentów za pomocą specjalnie przygotowanych do tego celu funkcji – hooków. Elementy te, oprócz zarządzania stanem komponentów, pozwalają również na wykonywanie efektów ubocznych w komponencie, odpowiadają więc za jego dynamiczny charakter. W obu rozwiązaniach posiadają one różne nazwy, lecz zasada ich działania jest bardzo do siebie zbliżona.

Podsumowując rozważania na ten temat, można z dużą dozą pewności stwierdzić, że pomimo znacznej różnicy w koncepcji renderowania w przypadku obu szkieletów programistycznych istnieje wiele idei, które te technologie ze sobą dzielą. Nie jest więc trudno wcześniej korzystając z jednej z nich, zrozumieć działanie drugiej.

7.2. Obsługa renderowania po stronie serwera

Podobnie jak w kontekście architektury, renderowanie po stronie serwera (SSR, ang. *Server-Side Rendering)*, koncepcyjnie nie różni się znacznie pomiędzy obiema technologiami. Jest to technika polegająca na generowaniu zawartości strony na serwerze i przesyłaniu statycznego kodu HTML do przeglądarki. Poprawia to znacznie czas ładowania po stronie klienta. Największa różnica pomiędzy obiema technologiami kolejny raz sprowadza się do odmiennej filozofii zarządzania drzewem DOM przeglądarki. Obie biblioteki w swoich rdzeniach nie posiadają narzędzi do obsługi SSR, lecz są one dostępne w ich ekosystemie jako oficjalne narzędzia dystrybuowane w pakietach server (React) i web (Solid). Zarówno składnia jak i dostępne API są bardzo do siebie zbliżone, oba rozwiązania zapewniają strumieniowe renderowanie, czyli przesyłanie kodu HTML w mniejszych pakietach, a ich główne funkcje posiadają takie same nazwy i zasady działania. Małą różnicą w funkcji głównej renderToString jest większa zwięzłość składni po stronie biblioteki Solid. Kolejną różnicą jest również wydajność przeprowadzanej hydracji, czyli przekształcanie przez klienta statycznego kodu otrzymanego od serwera w interaktywny interfejs użytkownika. Sam proces kodowania nie rożni się znacznie pomiędzy technologiami, wpływ na wydajność ma wymieniona wcześniej technika stosowania wirtualnego drzewa DOM, dodając dodatkową pracę bibliotece React. Drzewo to przyczynia się również do potrzeby upewniania się przez programistę w przypadku pracy z kontekstem, że jest on odpowiednio ustawiony i przekazywany do strony klienckiej z serwera. Solid nie sprawia takiego problemu, ponieważ zmiany stanu są automatycznie synchronizowane z drzewem DOM przeglądarki.

7.3. Zarządzanie cyklem życia aplikacji

Zarządzanie cyklem życia aplikacji różni się znacznie pomiędzy oboma rozwiązaniami. React oddaje większą kontrolę programiście, który za pomocą komponentów klasowych lub hooków może w wybranym momencie życia cyklu wykonywać odpowiednie operacje. Przykładem takiego działania może być wywołanie metody componentDidMount, używanej często jako punkt, w którym komponent kontaktuje się z interfejsem zewnętrznej aplikacji, ponieważ mechanizm ten aktywowany jest po pierwszym renderowaniu aplikacji [22]. Z drugiej strony szkielet Solid zdecydowanie upraszcza zarządzanie cyklem życia aplikacji, automatycznie śledzi zmiany zachodzące w stanie komponentu i na bieżąco aktualizuje drzewo DOM przeglądarki. Hook useEffect może być użyty w dowolnym momencie, jest on automatycznie używany według potrzeb kodu [14]. Każda z tych idei działania ma swoje zastosowania, zatem w tym przypadku wybór lepszego rozwiązania zależy od programisty i jego personalnych wymagań.

7.4. Ekosystem i wsparcie społeczności

Kolejnym punktem wartym analizy jest wielkość społeczności i ekosystemu zgromadzonego wokół danego narzędzia. W tym przypadku biblioteka React jest zdecydowanie bardziej rozwinięta. Według informacji podanych na stronie internetowej menadżera pakietów dla platformy Node.js (npm) React.js posiada ponad 238 tysięcy pakietów zależnych od jej najważniejszego, podstawowego pakietu. W przypadku Solid.js ta liczba to tylko 663 pakiety. Widoczna jest tutaj różnica w wielkości ekosystemów. Jest ona spowodowana ilością czasu, od jakiego dane rozwiązanie jest dostępne na rynku, ale także od liczby użytkowników danej biblioteki. React.js jest tygodniowo pobierany ponad 25 milionów razy, gdy Solid.js ponad 266 tysięcy razy [24]. Taka sama tendencja ukazana jest w przypadku porównania liczby utworzonych wątków w serwisie GitHub dla obu narzędzi. React.js znowu częściej wzbudza zainteresowanie użytkowników, posiada na moment pisania pracy 665 otwartych i ponad 12 tysięcy zamkniętych wątków [21], Solid.js odpowiednio 78 otwartych i 934 zamkniętych [20].

7.5. Krzywa nauki danej technologii

W aspekcie dostępności materiałów i łatwości nauki szkielet programistyczny React ponownie pozytywnie wyróżnia się na tle konkurenta. Największą przewagę gwarantuje mu liczba dostępnych materiałów i kursów, dających możliwość samodzielnej nauki aspirującym programistom. Analizując liczbę kursów na edukacyjnej platformie Udemy [28], można dostrzec zdecydowaną przewagę starszej z bibliotek. Posiada ona tysiące kursów, gdy jej młodszy odpowiednik jedynie kilkadziesiąt. W tym aspekcie, czyli dostępności materiałów do nauki, leży największy problem dla nowego użytkownika tej technologii, gdyż ogólna tendencja społeczności, zaobserwowana na platformie społecznościowej Reddit [18] wskazuje, że obie technologie są podobnie trudne do przyswojenia. Jest to spowodowane zastosowaniem podobnej semantyki oraz rozwiązań technicznych w tych narzędziach.

8. Wnioski

Podsumowując otrzymane wyniki wydajnościowe badanych aplikacji oraz wyciągając wnioski z teoretycznej analizy ich cech i możliwości, obie technologie mają swoje mocne strony wyróżniające je ponad konkurencję. Młodsza z nich – biblioteka Solid.js była z dużą dozą prawdopodobieństwa tworzona z myślą o wydajności, zauważalnie szybciej wykonuje ona postawione przez sobą zadania. Wniosek ten potwierdza tezę badawczą T1.

Biblioteka React z kolei to bardziej dojrzała platforma z dużym wsparciem społeczności i ogromnym

zasobem dodatkowych pakietów rozwijających jej funkcjonalność. Łatwiej jest znaleźć do niej materiały do nauki, częściej występuje jako strona badana w pracach naukowych. Dojrzały ekosystem i zaangażowanie społeczności świadczą też o jej niezaprzeczalnej przewadze, jeśli chodzi o łatwość rozpoczęcia z nią pracy, gdyż inne aspekty, w porównaniu do biblioteki Solid, nie mają znaczącego wpływu na komfort przy jej użyciu. Jest to spowodowane podobieństwem w składni każdego z narzędzi oraz wieloma zbliżonymi pomysłami implementacyjnymi wszelakich funkcjonalności. Argument ten weryfikuje więc tezę badawczą T2 i pozwala stwierdzić, iż w 2024 roku, Solid.js nie jest jeszcze wystarczająco popularny w środowisku programistycznym, by móc go polecić nowym użytkownikom. Świadomi i doświadczeni odbiorcy docenią jednak jego wydajność i podobieństwo do biblioteki React, które może być czynnikiem ułatwiającym przeniesienie się na tą technologię.

Do ostatecznym wniosków analizy należy też dodać potrzebę cyklicznego badania rozwoju nowo powstałych technologii porównując je do narzędzi z wyrobioną renomą na rynku. Szkielet programistyczny Solid.js nieustannie i dynamicznie się rozwija, przez co wnioski zawarte w tej pracy mogą w perspektywie niedługiego odstępu czasu okazać się nieaktualne. Szkielet ten nie posiada również wielu oficjalnych prac badawczych przeprowadzonych na jego temat, co stawia potrzebę prowadzenia dalszych badań nad jego możliwościami.

Literatura

[1] A. Rauschmayer, Speaking JavaScript, O'Reilly Media, Sebastopol, 2014.

[2] M. Golec, Comparative analysis of frameworks using TypeScript to build server applications, Master thesis, Lublin University of Technology, Lublin, 2022.

[3] F. Gomes, P. Soares, A. A. Araújo, Examining the Performance Implications of Design Patterns in Front-end Web Development: A Preliminar Comparative Study with React and Vue.js, Proceedings of the 29th Brazilian Symposium on Multimedia and the Web (2023) 255-259, https://dl.acm.org/doi/10.1145/3617023.3617057.

[4] S. Chen, U. R. Thaduri, V. K. R. Ballamudi, Front-End Development in React: An Overview, Engineering International 7 (2019) 117-126.

[5] E. Wohlgethan, Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js, Bachelor thesis, Hamburg University of Applied Sciences, Hamburg, 2018.

[6] C. L. Mariano, Benchmarking JavaScript Frameworks, Master thesis, Technological University of Dublin, Dublin, 2017.

[7] A. Gizas, S. Christodoulou, T. Papatheodorou, Comparative evaluation of Javascript frameworks, WWW '12 Companion, Proceedings of the 21st International Conference on World Wide Web, ACM (2012) 513-514, https://dl.acm.org/doi/10.1145/2187980.2188103.

[8] S. Josefin, Evaluating JavaScript Frameworks, Bachelor thesis, Linnaeus University, Växjö, 2020.

[9] M. Kaluža, K. Troskot, B. Vukelić, Comparison of front-end frameworks for web applications development, Zbornik Veleučilišta u Rijeci 6 (2018) 261-282.

[10] A. Banks, E. Porcello, Learning React, Modern Patterns for Developing React Apps, O'Reilly Media, Sebastopol, 2020.

[11] G. van der Put, React.js Complete Guide To Server-Side Rendering, Independently published, 2020.

[12] S. bin Uzayr, N. Cloud, T. Ambler, JavaScript Frameworks for Modern Web Development, Apress Berkeley, CA, Berkeley, 2019.

[13] React.js – biblioteka programistyczna dla języka JavaScript, https://react.dev/, [08.07.2024].

[14] Solid.js – szkielet programistyczny dla języka JavaScript, https://www.solidjs.com/, [13.07.2024].

[15] JavaScript – dokumentacja do języka JavaScript, https://developer.mozilla.org/en-US/docs/Web/JavaScript, [12.12.2023].

[16] Reactjs.org – Wprowadzenie do nakładki JSX, https://pl.reactjs.org/docs/introducing-jsx.html, [10.03.2024].

[17] Siddhant Varma – Introduction to SolidJS, https://www.loginradius.com/blog/engineering/guest-post/introduction-to-solidjs/, [10.03.2024].

[18] Platforma społecznościowa, serwis Reddit, https://www.reddit.com/, [06.09.2024].

[19] Stackoverflow.co – 2024 Technologies Survey, https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies, [13.09.2024].

[20] Oficjalne repozytorium biblioteki Solid.Js, https://github.com/solidjs/solid, [19.09.2024].

[21] Oficjalne repozytorium biblioteki React.Js, https://github.com/facebook/react, [20.09.2024].

[22] DevDocs – dokumentacja Reacta, https://devdocs.io/react/, [08.03.2024].

[23] Brainhub Ltd. – Strategy and Tips for Migrating to React, https://brainhub.eu/library/migrating-to-react/, [10.12.2023].

[24] Strona menedżera pakietów dla Node.js, https://www.npmjs.com/, [19.09.2024].

[25] SolidJS vs. React: The Go-to Guide, https://www.toptal.com/react/solidjs-vs-react, [17.12.2023].

[26] Platforma z kursami do nauki programowania, https://www.udemy.com/pl/, [20.09.2024].

[27] Dokumentacja Microsoft Visual Studio Code, https://code.visualstudio.com/docs, [08.07.2024].

[28] Oficjalna strona biblioteki Faker.js, https://fakerjs.dev, [16.07.2024].