Uniwersytet WSB Merito w Poznaniu
Wydział Zamiejscowy w Chorzowie

Jakub Kielaszek

Architektura i optymalizacja renderowania w aplikacjach webowych na przykładzie wizualizera algorytmów sortowania

Projekt dyplomowy

Kierownik naukowy:
dr Tomasz Staś

Kierunek: Informatyka
Specjalność: Zaawansowane systemy baz danych
Numer albumu: 180757

CHORZÓW 2026

STRESZCZENIE

Spis treści

WSTĘP

Dynamiczny rozwój technologii internetowych oraz rosnące oczekiwania użytkowników wobec szybkości, interaktywności i responsywności aplikacji webowych sprawiają, że wydajność renderowania staje się jednym z kluczowych wyzwań współczesnego frontendu. Wraz z ewolucją od prostych, statycznych stron HTML do wysoce złożonych aplikacji jednosesyjnych (SPA), znacząco wzrosła zarówno złożoność logiki interfejsu użytkownika, jak i liczba aktualizacji widoku wykonywanych w bardzo krótkich interwałach czasowych. W tym kontekście kluczowego znaczenia nabierają nowoczesne frameworki frontendowe, takie jak React oraz Angular, które, mimo wspólnego celu, oferują fundamentalnie odmienne podejścia architektoniczne i mechanizmy synchronizacji stanu z modelem dokumentu (DOM). Zrozumienie tych różnic jest niezbędne nie tylko z punktu widoku teoretycznego, ale przede wszystkim dla inżynierów oprogramowania dążących do budowy skalowalnych i wydajnych systemów webowych.

Rosnąca złożoność współczesnych aplikacji frontendowych wynika z przenoszenia coraz większej liczby operacji z warstwy serwerowej bezpośrednio do przeglądarki klienta. Mechanizmy takie jak dynamiczne komponenty, rozbudowane drzewa zależności stanu, interaktywne animacje czy obsługa danych w czasie rzeczywistym wymagają wysokowydajnych metod aktualizacji widoku. React i Angular podchodzą do tych wyzwań w różny sposób: React opiera się na deklaratywnym modelu komponentów i mechanizmie wirtualnego drzewa widoku (Virtual DOM), podczas gdy Angular implementuje kompleksową architekturę z silnym systemem wstrzykiwania zależności oraz strukturalnym podejściem do detekcji zmian. Wybór między tymi technologiami często determinuje nie tylko wydajność końcową produktu, ale także ergonomię pracy deweloperów oraz łatwość utrzymania kodu w długim terminie.

Celem niniejszej pracy jest przeprowadzenie dogłębnej analizy i porównania podejść do renderowania interfejsu użytkownika w frameworkach React i Angular, ze szczególnym uwzględnieniem aspektów wydajnościowych, architektonicznych oraz ergonomii programistycznej. Wybór wizualizera algorytmów sortowania jako głównego studium przypadku podyktowany jest specyfiką tego rodzaju aplikacji — wymagają one bardzo wysokiej częstotliwości aktualizacji widoku przy jednoczesnej manipulacji dużą liczbą elementów graficznych. Każda operacja zamiany elementów lub ich porównania generuje zdarzenie,

które musi zostać odzwierciedlone w modelu dokumentu, co stawia ekstremalne wymagania przed mechanizmami detekcji zmian oraz procesem reconciliation, służącym do synchronizacji wirtualnej reprezentacji komponentów z rzeczywistym obiektem DOM. Taka charakterystyka pracy pozwala na precyzyjne przetestowanie granic wydajności silników renderujących oraz mechanizmów detekcji zmian w obu technologiach.

W pracy przygotowano autorski wizualizator algorytmów sortowania, zaimplementowany równolegle w obu technologiach. Ma on służyć jako platforma badawcza do systematycznego i porównywalnego testowania efektywności renderowania, zużycia zasobów procesora oraz płynności interfejsu podczas intensywnych operacji na danych. Wizualizacja procesów algorytmicznych w czasie rzeczywistym jest wyzwaniem nie tylko ze względu na liczbę operacji DOM, ale także konieczność zachowania wysokiej częstotliwości odświeżania (FPS), co bezpośrednio przekłada się na percepcję płynności ruchu przez użytkownika. Dzięki temu możliwe jest sformułowanie merytorycznych wniosków dotyczących mocnych i słabych stron przeanalizowanych frameworków w kontekście nowoczesnych, wysokowydajnych aplikacji frontendowych.

Cele szczegółowe pracy obejmują:

- przygotowanie szczegółowego przeglądu architektury oraz kluczowych mechanizmów działania frameworków React i Angular,
- opracowanie i implementację wizualizatora algorytmów sortowania w obu technologiach przy zachowaniu identycznych założeń funkcjonalnych,
- identyfikację i analizę zaawansowanych technik optymalizacji renderowania specyficznych dla każdej z technologii,
- przeprowadzenie pomiarów wydajnościowych i ocenę ich wpływu na płynność interfejsu oraz stabilność aplikacji,
- porównanie złożoności implementacyjnej, czytelności kodu oraz ogólnej ergonomii pracy programisty w obu ekosystemach,

Zakres pracy koncentruje się na analizie porównawczej mechanizmów renderowania i aktualizacji interfejsu w React oraz Angular. Część praktyczna skupia się na implementacji wizualizatora, który stanowi wspólny mianownik dla testów wydajnościowych. Praca nie porusza zagadnień związanych z architekturą backendową, bezpieczeństwem przesyłu danych ani optymalizacją po stronie serwera (SSR), skupiając się wyłącznie na warstwie prezentacji i logice wykonywanej w przeglądarce.

Praca składa się z czterech zasadniczych rozdziałów. W rozdziale pierwszym przedstawiono fundamenty teoretyczne, ewolucję aplikacji webowych oraz charakterystykę anali-

zowanych frameworków. Rozdział drugi opisuje przyjętą metodykę badań oraz proces implementacji platformy testowej. W trzecim rozdziale zaprezentowano i przeanalizowano wyniki pomiarów wydajnościowych oraz porównanie produktywności. Pracę kończy rozdział czwarty, zawierający syntezę wyników oraz wnioski końcowe.

ROZDZIAŁ PIERWSZY

Podstawy teoretyczne i przegląd technologii frontendowych

Podstawy teoretyczne stanowią istotny fundament dla dalszej części pracy, obejmującej analizę architektury oraz mechanizmów działania nowoczesnych frameworków frontendowych. W rozdziale przedstawiono ewolucję aplikacji webowych oraz omówiono najważniejsze koncepcje związane z ich tworzeniem, ze szczególnym uwzględnieniem współczesnych paradygmatów i modeli programowania w JavaScripcie. Kolejne podsekcje prezentują przegląd dwóch popularnych technologii — React i Angular — wraz z ich kluczowymi założeniami architektonicznymi i podejściami do renderowania interfejsu użytkownika. Rozdział kończy omówienie podstaw algorytmów sortowania oraz metod ich wizualizacji, co stanowi kontekst dla implementacji aplikacji prezentowanych w dalszej części pracy.

1.1 Ewolucja i znaczenie nowoczesnych aplikacji webowych

Początki aplikacji webowych sięgają statycznych stron HTML, które pełniły funkcję prostych dokumentów prezentowanych użytkownikowi bez możliwości interakcji. W modelu tym cała logika przetwarzania danych oraz generowania treści znajdowała się po stronie serwera, natomiast przeglądarka pełniła wyłącznie rolę klienta wyświetlającego przygotowaną wcześniej zawartość. Tego typu podejście było wystarczające w czasach niskich wymagań funkcjonalnych, jednak wraz z rosnącą popularnością internetu i pojawieniem się bardziej złożonych serwisów zaczęło okazywać się niewystarczające.

Kolejnym etapem rozwoju było wprowadzenie mechanizmów dynamicznego generowania stron oraz technologii takich jak JavaScript i AJAX, które umożliwiły odświeżanie wybranych fragmentów interfejsu bez konieczności przeładowywania całej strony [1]. Pozwoliło to na stworzenie bardziej interaktywnych aplikacji oraz znacząco poprawiło kom-

fort użytkownika. Wraz z tymi zmianami zaczęły pojawiać się pierwsze rozwiązania nakierowane na organizację kodu po stronie klienta, a także frameworki wspierające tworzenie modularnych komponentów interfejsu.

Dynamiczny rozwój technologii frontendowych doprowadził do powstania aplikacji jednosesyjnych (Single Page Applications, SPA), które stanowią obecnie dominujący model budowy interfejsów webowych [2]. SPA charakteryzują się tym, że cała aplikacja jest ładowana jednorazowo, a kolejne interakcje użytkownika prowadzą do aktualizacji tylko tych elementów, które faktycznie ulegają zmianie. Przeniesienie znacznej części logiki na stronę klienta wymusiło jednak opracowanie bardziej zaawansowanych mechanizmów zarządzania stanem oraz aktualizacji widoku, ponieważ rosnąca złożoność komponentów oraz liczba zmian w ich stanie stawiały wysokie wymagania dotyczące efektywności renderowania.

W tym kontekście istotnego znaczenia nabrały nowoczesne frameworki frontendowe, takie jak React i Angular, które proponują odmienne podejścia architektoniczne do organizacji kodu, zarządzania stanem oraz aktualizacji interfejsu użytkownika. React, oparty na koncepcji deklaratywnego programowania i wykorzystaniu wirtualnego drzewa DOM, znacząco zmienił sposób myślenia o komponowaniu interfejsu oraz odświeżaniu widoku. Angular natomiast rozwija architekturę opartą na komponentach, modułach i mechanizmach detekcji zmian, zapewniając ustrukturyzowane środowisko do tworzenia rozbudowanych aplikacji o dużej skali.

Równocześnie użytkownicy zaczęli oczekiwać od aplikacji webowych płynności i jakości działania porównywalnych z natywnymi aplikacjami mobilnymi czy desktopowymi. Oznacza to konieczność minimalizacji opóźnień, redukcji liczby niepotrzebnych renderów oraz optymalnego zarządzania stanem aplikacji. Wydajność renderowania stała się więc kluczowym aspektem doświadczenia użytkownika, a jednocześnie jednym z najważniejszych kryteriów przy wyborze technologii frontendowych. Zbiór metryk takich jak *Core Web Vitals* pozwala na obiektywną ocenę jakości interakcji użytkownika z witryną [3]. W wielu współczesnych projektach to właśnie sposób aktualizacji interfejsu oraz efektywność wykonywania operacji renderujących wpływają na ogólną jakość aplikacji, jej skalowalność oraz koszty utrzymania.

Znaczenie nowoczesnych aplikacji webowych wykracza poza typowe strony internetowe — obejmuje systemy biznesowe, narzędzia analityczne, aplikacje edukacyjne, panele administracyjne, a także interaktywne wizualizacje danych. W tych zastosowaniach renderowanie elementów interfejsu często odbywa się wielokrotnie w krótkich odstępach czasu, co zwiększa potrzebę stosowania wydajnych mechanizmów aktualizacji widoku. Z tego względu zrozumienie różnic pomiędzy podejściami oferowanymi przez React i Angular jest istotne nie tylko z perspektywy teoretycznej, ale również praktycznej i projektowej.

Podsumowując, rozwój aplikacji webowych od prostych stron HTML do rozbudowanych aplikacji jednosesyjnych znacząco wpłynął na sposób projektowania i implementacji interfejsów użytkownika. Wzrost złożoności logiki po stronie klienta oraz potrzeba częstych aktualizacji widoku sprawiły, że optymalizacja renderowania stała się jednym z najważniejszych wyzwań współczesnego frontendu. Frameworki React i Angular stanowią dwa dominujące podejścia do jego rozwiązania, co czyni ich analizę istotną zarówno z perspektywy inżynierskiej, jak i praktycznej.

1.2 JavaScript i TypeScript we współczesnym ekosystemie frontendu

JavaScript jest podstawowym językiem programowania wykorzystywanym do tworzenia interfejsów użytkownika w aplikacjach webowych. Jego rola stopniowo rosła wraz z ewolucją witryn internetowych od prostych stron statycznych do nowoczesnych aplikacji jednosesyjnych, wymagających dynamicznych aktualizacji widoku oraz częstej komunikacji z serwerem. JavaScript jest językiem interpretowanym, uruchamianym bezpośrednio w przeglądarce, co umożliwia tworzenie interaktywnych elementów oraz reagowanie na zdarzenia użytkownika, takie jak kliknięcia czy zmiany danych wejściowych. Kluczowym mechanizmem pozwalającym na sprawne działanie aplikacji jest model jednowątkowy oparty na pętli zdarzeń (ang. event loop), który umożliwia asynchroniczne przetwarzanie operacji bez blokowania interfejsu użytkownika.

Istotny wpływ na rozwój JavaScriptu miało wprowadzenie standardu ECMAScript 6 (ES6), który rozbudował język o nowoczesne mechanizmy wspierające programowanie strukturalne i modularne [4]. Do najważniejszych rozszerzeń należą moduły import/export, klasy, funkcje strzałkowe oraz usprawnione zarządzanie zmiennymi poprzez słowa kluczowe `let` i `const`. Zmiany te umożliwiły budowanie bardziej przejrzystych projektów, poprawiły czytelność kodu oraz zwiększyły skalowalność aplikacji. Współczesne frameworki frontendowe, takie jak React i Angular, ściśle opierają się na funkcjonalnościach ES6 i nowszych wersji ECMAScript, wykorzystując je do implementacji architektury komponentowej, obsługi stanu oraz mechanizmów aktualizacji widoku.

Drugim kluczowym elementem ekosystemu frontendowego jest TypeScript — nadzbiór JavaScriptu opracowany przez Microsoft, wprowadzający statyczne typowanie oraz szereg mechanizmów znanych z języków obiektowych [5]. TypeScript pozwala na definiowanie typów zmiennych, struktur danych, interfejsów oraz klas, co znacząco zwiększa bezpieczeństwo i przewidywalność kodu, szczególnie w dużych projektach. Kompilacja TypeScriptu do JavaScriptu umożliwia korzystanie z rozszerzeń typów bez rezygnacji z kompatibilności z przeglądarkami.

TypeScript odgrywa szczególnie ważną rolę w Angularze, który został zaprojektowany w pełnej integracji z typowaniem statycznym. Dzięki temu architektura Angulara opiera się na klasach, dekoratorach, wstrzykiwaniu zależności oraz modułach, a statyczne typowanie wspiera analizę błędów jeszcze przed uruchomieniem aplikacji. React z kolei w naturalny sposób wspiera TypeScript, choć nie jest od niego zależny — jednak w praktyce większość nowych projektów React powstaje właśnie z użyciem TypeScriptu ze względu na większą czytelność i bezpieczeństwo kodu.

```
// JavaScript
function getDiscountedPrice(price, ratio) {
  return price * (1 - ratio);
}

// TypeScript
interface Order {
  price: number;
  discountRatio: number;
}

function getDiscountedPrice(order: Order): number {
  return order.price * (1 - order.discountRatio);
}
```

Rysunek 1.1: Porównanie fragmentu kodu w języku JavaScript oraz TypeScript.

Źródło: Opracowanie własne

Zarówno JavaScript, jak i TypeScript stanowią fundament rozwoju nowoczesnych narzędzi frontendowych, umożliwiając tworzenie skalowalnych, modularnych i wydajnych aplikacji webowych. Zrozumienie ich kluczowych mechanizmów, takich jak model zdarzeń, moduły ES6, klasy oraz statyczne typowanie, jest niezbędne dla analizy architektury oraz sposobu działania frameworków React i Angular, omówionych w następnych podsekcjach.

1.3 Przegląd paradygmatów i architektury frameworków TypeScript

Rozwój nowoczesnych frameworków JavaScript wynika bezpośrednio z rosnącej złożoności aplikacji webowych oraz potrzeby stosowania bardziej ustrukturyzowanych metod organizacji kodu po stronie klienta. Wraz z upowszechnieniem się aplikacji jednosesyjnych pojawiło się zapotrzebowanie na narzędzia, które umożliwiałyby efektywne zarządzanie

stanem, modularność, ponowne wykorzystanie komponentów oraz kontrolę nad procesem renderowania interfejsu. W odpowiedzi na te potrzeby powstały liczne biblioteki i frameworki frontendowe, reprezentujące odmienne paradygmaty projektowania i przetwarzania danych w kontekście interfejsu użytkownika.

Jednym z kluczowych paradygmatów, który znacząco wpłynął na sposób budowania interfejsów webowych, jest programowanie komponentowe. Zakłada ono podział aplikacji na małe, niezależne moduły odpowiedzialne za określone fragmenty logiki i widoku. Każdy komponent może definiować swój stan, metody oraz sposób wyświetlania, a następnie być wielokrotnie wykorzystywany w różnych częściach aplikacji. Komponentowe podejście umożliwia przejrzystą strukturę projektu, ułatwia testowanie oraz zwiększa skalowalność, szczególnie w aplikacjach obsługujących dużą liczbę dynamicznych elementów.

Innym istotnym paradygmatem jest programowanie deklaratywne, charakterystyczne przede wszystkim dla Reacta. W podejściu deklaratywnym programista opisuje, jaki stan interfejs ma zostać wyświetlony dla określonych danych, natomiast szczegóły dotyczące aktualizacji, renderowania i synchronizacji widoku z logiką wewnętrzną są ukryte za mechanizmami frameworka. Dzięki temu zmniejsza się liczba błędów wynikających z ręcznego manipulowania DOM, a kod staje się bardziej przewidywalny i łatwiejszy do analizy.

Angular natomiast łączy elementy podejścia deklaratywnego z architekturą opartą na wzorcu Model-View-ViewModel (MVVM) oraz mechanizmem wstrzykiwania zależności, co umożliwia ścisłe rozdzielenie warstw logiki biznesowej, widoku oraz komunikacji z usługami. Duży nacisk położony jest na strukturalność projektu, wykorzystanie modułów oraz silną typizację dzięki integracji z TypeScript. Takie podejście jest szczególnie korzystne w projektach o dużej skali, gdzie kluczowa jest czytelna organizacja kodu i jego łatwa rozbudowa w przyszłości.

Wspólną cechą większości współczesnych frameworków JavaScript jest dążenie do minimalizacji kosztów związanych z aktualizacją widoku. Mechanizmy takie jak Virtual DOM w React, strefy i detekcja zmian w Angularze, reaktywny przepływ danych czy zaawansowane modele kolejkowania operacji renderujących zostały zaprojektowane po to, aby unikać niepotrzebnych odświeżeń interfejsu. Dzięki temu możliwe jest budowanie aplikacji reagujących na częste zmiany stanu i przetwarzających duże ilości danych przy zachowaniu płynnej i stabilnej pracy.

Paradygmaty i architektury wykorzystywane przez współczesne frameworki mają bezpośredni wpływ na wydajność aplikacji oraz wygodę pracy programisty. Różnice w sposobie zarządzania stanem, obsługi cyklu życia komponentów oraz reagowania na zmiany danych sprawiają, że React i Angular sprawdzają się inaczej w zależności od charakteru projektu i wymagań użytkownika. Zrozumienie tych podejść jest kluczowe dla dalszej

analizy technik renderowania oraz porównania obu frameworków w kontekście implementacji wizualizatora algorytmów sortowania.

1.4 Charakterystyka React

React jest jedną z najpopularniejszych bibliotek frontendowych wykorzystywanych do budowy interfejsów użytkownika we współczesnych aplikacjach webowych. Został opracowany przez Facebooka w 2013 roku jako odpowiedź na rosnącą złożoność warstwy prezentacji w dużych systemach internetowych oraz potrzebę efektywnego zarządzania licznymi aktualizacjami widoku [6]. React szybko zyskał szerokie zastosowanie w przemyśle dzięki swojej prostocie, modularności oraz nowatorskiemu podejściu do renderowania komponentów [7].

Podstawą Reacta jest deklaratywny model programowania, w którym programista opisuje, jak interfejs ma wyglądać dla określonego stanu danych, zamiast zarządzać ręcznie operacjami modyfikującymi DOM. Dzięki temu kod jest bardziej przewidywalny, łatwiejszy do utrzymania oraz mniej podatny na błędy związane z ręczną manipulacją strukturą dokumentu. Kluczowym elementem działania Reacta jest również wykorzystanie koncepcji Virtual DOM — lekkiej reprezentacji drzewa elementów, pozwalającej na efektywne określanie minimalnego zakresu zmian potrzebnych do zaktualizowania widoku.

React wprowadza komponentową strukturę aplikacji, w której interfejs podzielony jest na niewielkie, niezależne moduły odpowiadające za określoną funkcjonalność lub fragment widoku. Każdy komponent może posiadać stan lokalny, reagować na zmiany danych oraz uczestniczyć w przepływie informacji w aplikacji. Taki sposób organizacji sprzyja modularności, ponownemu wykorzystaniu kodu oraz tworzeniu aplikacji o wysokiej skalowalności.

```jsx
import React from 'react';

const Welcome = ({ name }) => {
  return (
    <div className="container">
      <h1>Witaj, {name}!</h1>
      <p>Komponenty React łączą elogik z opisem widoku.</p>
    </div>
  );
};
```

Rysunek 1.2: Przykładowy komponent funkcjonalny w React wykorzystujący składnię JSX.

Źródło: Opracowanie własne

Z perspektywy wydajności istotne znaczenie ma mechanizm ponownego renderowania komponentów, który w React może być optymalizowany poprzez odpowiednie zarządzanie zmianami stanu oraz właściwą organizację struktury komponentowej. Wprowadzenie hooków, takich jak *useState*, *useEffect* czy *useMemo*, umożliwiło bardziej elastyczne zarządzanie cyklem życia komponentów oraz kontrolę nad tym, kiedy i dlaczego następuje ponowne renderowanie. React dostarcza również narzędzia wspierające analizę wydajności, takie jak profiler, co ułatwia identyfikowanie komponentów generujących nadmiarowe odświeżenia.

Zastosowanie Reacta w aplikacjach wymagających częstych aktualizacji interfejsu — takich jak wizualizatory danych czy systemy czasu rzeczywistego — jest szczególnie interesujące ze względu na sposób, w jaki biblioteka zarządza aktualizacjami widoku. Właśnie tego typu zastosowania pozwalają na praktyczną ocenę efektywności mechanizmów renderowania oraz wpływu architektury biblioteki na płynność działania interfejsu użytkownika.

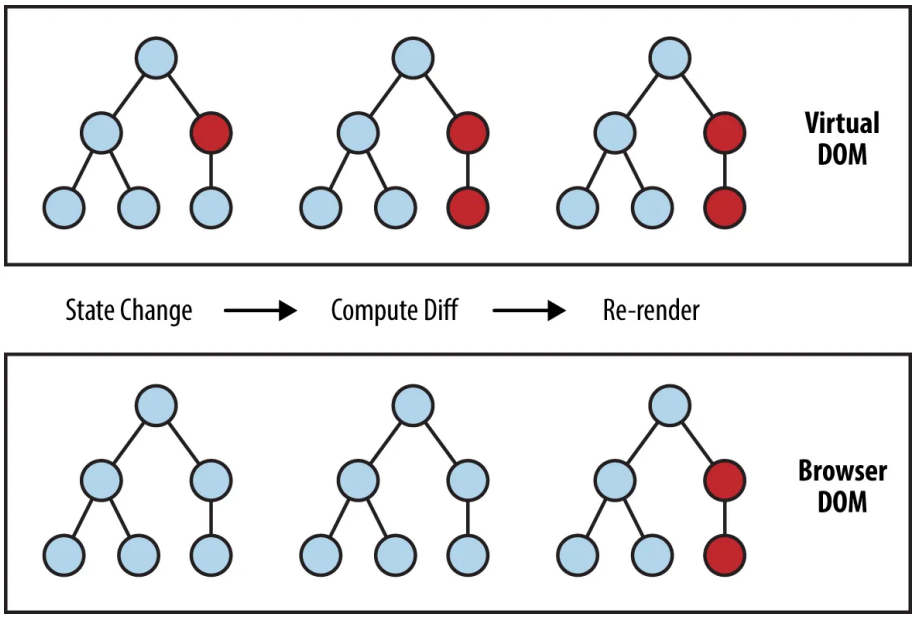
1.4.1 Architektura i kluczowe koncepcje

Architektura React opiera się na kilku fundamentalnych koncepcjach, które definiują sposób budowy interfejsu użytkownika oraz zarządzania jego aktualizacjami. Najważniejszym z nich jest komponentowy model aplikacji, który zakłada podział interfejsu na niezależne, wielokrotnie wykorzystywane elementy. Komponent może reprezentować zarówno niewielki fragment widoku, jak i bardziej złożoną strukturę składającą się z wielu podkomponentów. Dzięki temu kod aplikacji staje się modularny, łatwiejszy do utrzymania oraz podatny na skalowanie.

Kolejną kluczową koncepcją Reacta jest deklaratywny sposób definiowania interfejsu użytkownika. Programista opisuje, jak interfejs ma wyglądać dla określonego stanu danych, natomiast React odpowiada za minimalną liczbę operacji aktualizujących rzeczywisty DOM. Deklaratywność sprawia, że kod jest bardziej przejrzysty i mniej podatny na błędy wynikające z ręcznego manipulowania drzewem DOM, charakterystycznego dla wcześniejszych rozwiązań opartych na podejściu imperatywnym.

React wykorzystuje również własne rozszerzenie składni JavaScript, znane jako JSX lub TSX (odpowiednio JavaScript XML oraz TypeScript XML). Umożliwia to opisywanie struktury komponentów w sposób zbliżony do HTML, a jednocześnie pozwala na osadzanie logiki JavaScript bezpośrednio w definicji widoku. Rozwiązanie to zwiększa czytelność kodu oraz ułatwia łączenie warstwy prezentacji z logiką aplikacji, co jest szczególnie korzystne w przypadku dynamicznych interfejsów wymagających częstych aktualizacji.

Fundamentem wydajności Reacta jest mechanizm Virtual DOM. Jest to lekka reprezentacja drzewa elementów, która przechowuje zapisaną w pamięci strukturę interfejsu. W momencie zmiany stanu komponentu React generuje nowe drzewo Virtual DOM, a następnie porównuje je z poprzednią wersją, aby określić, które elementy interfejsu faktycznie uległy zmianie. Dzięki temu liczba operacji wykonywanych na prawdziwym DOM jest znacząco zredukowana, co wpływa na zwiększenie wydajności renderowania, szczególnie w przypadku aplikacji wymagających częstych odświeżeń widoku.



Rysunek 1.3: Schemat procesu aktualizacji Virtual DOM i rzeczywistego DOM w React. *Źródło:*

Wraz z wprowadzeniem architektury React Fiber przebudowano wewnętrzny mechanizm odpowiedzialny za przetwarzanie aktualizacji komponentów. Fiber stanowi asynchroniczny, priorytetowy model renderowania, który umożliwia dzielenie procesu odświeżania na mniejsze części oraz nadawanie priorytetów poszczególnym aktualizacjom [8]. Dzięki temu React może bardziej efektywnie reagować na interakcje użytkownika oraz zapewniać płynność działania nawet wtedy, gdy aplikacja wykonuje złożone operacje w tle.

Istotnym elementem architektury Reacta jest również cykl życia komponentów, który definiuje, w jakich momentach aplikacja może odwołać się do logiki zewnętrznej lub wykonać działania związane z odświeżaniem widoku. Wraz z wprowadzeniem hooków cykl życia został uproszczony i ujednolicony, co znacząco zwiększyło elastyczność zarządzania stanem oraz zachowaniami komponentów. Hooki, takie jak *useState*, *useEffect* czy *useRef*, pozwalają na precyzyjne kontrolowanie efektów ubocznych, aktualizacji stanu oraz interakcji z elementami DOM.

Podsumowując, architektura Reacta opiera się na czytelnej strukturze komponentów, deklaratywnym opisie interfejsu oraz efektywnym modelu aktualizacji widoku opartym na

Virtual DOM i mechanizmie Fiber. Rozwiązania te umożliwiają tworzenie aplikacji o wysokiej wydajności i dużej skalowalności, a jednocześnie zapewniają elastyczność programistyczną i przejrzystość kodu, co czyni React jednym z najpopularniejszych rozwiązań we współczesnym ekosystemie frontendowym.

1.4.2 Zarządzanie stanem w React

Zarządzanie stanem stanowi jeden z fundamentów programowania w bibliotece React, gdyż to właśnie wartości przechowywane w stanie determinują bieżący wygląd interfejsu oraz momenty jego odświeżania. Architektura Reacta opiera się na koncepcji jednokierunkowego przepływu danych (ang. *unidirectional data flow*), według której informacje są przekazywane hierarchicznie z góry do dołu drzewa komponentów. Zapewnia to wysoką przewidywalność cyklu życia aplikacji oraz ułatwia śledzenie przyczyn zmian w widoku.

Najpowszechniejszym sposobem przechowywania danych w komponentach funkcjonalnych jest wykorzystanie hooka `useState`. Pozwala on na zdefiniowanie lokalnego stanu, którego każda aktualizacja inicjuje proces ponownego renderowania komponentu. React optymalizuje ten proces, aktualizując jedynie te fragmenty rzeczywistego drzewa DOM, które wynikają bezpośrednio ze zmienionych danych.

```
import { useState } from 'react';

function Counter() {
  const [value, setValue] = useState(0);

  const increment = () => {
    setValue(prev => prev + 1);
  };

  return (
    <button onClick={increment}>
      {value}
    </button>
  );
}
```

Rysunek 1.4: Przykład lokalnego zarządzania stanem przy użyciu hooka `useState`.

Źródło: Opracowanie własne

W architekturze komponentowej kluczowe znaczenie ma mechanizm przekazywania danych poprzez właściwości (ang. *props*). Komponent nadrzędny może udostępnić fragment swojego stanu komponentom podrzędnym, które otrzymują go jako parametry wejściowe.

Komunikacja w przeciwnym kierunku — od komponentu podrzędnego do nadrzędnego — realizowana jest poprzez przekazywanie funkcji zwrotnych (ang. *callback functions*). W przypadku rozbudowanych struktur, gdzie dane muszą zostać przekazane przez wiele pośrednich poziomów, pojawia się wyzwanie określane jako *prop drilling*, co może prowadzić do zmniejszenia czytelności kodu i utrudniać jego konserwację.

Aby uniknąć nadmiernego przekazywania właściwości przez wiele poziomów, React udostępnia mechanizm *Context API*. Pozwala on na stworzenie centralnego źródła danych, do którego dostęp mają wszystkie komponenty znajdujące się wewnątrz danego dostawcy (ang. *provider*), niezależnie od ich pozycji w hierarchii. Jest to rozwiązanie szczególnie przydatne przy zarządzaniu ustawieniami globalnymi, takimi jak motyw graficzny, język aplikacji czy dane sesji użytkownika.

W sytuacjach, gdy logika aktualizacji stanu staje się złożona i zależy od wielu powiązanych ze sobą wartości, zaleca się stosowanie hooka `useReducer`. Implementuje on wzorzec zbliżony do architektury Redux, w którym zmiana stanu odbywa się poprzez wysyłanie akcji (ang. *dispatching actions*) do funkcji redukującej (ang. *reducer*). Podejście to sprzyja separacji logiki biznesowej od warstwy prezentacji i ułatwia testowanie kodu.

```jsx
import { useReducer } from 'react';

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'reset':
      return 0;
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, 0);

  return (
    <div>
      <button onClick={() => dispatch({ type: 'increment' })}>
        {state}
      </button>
      <button onClick={() => dispatch({ type: 'reset' })}>
        Reset
      </button>
    </div>
  );
}
```

Rysunek 1.5: Użycie hooka `useReducer` do obsługi złożonej logiki aktualizacji stanu.

Źródło: Opracowanie własne

W przypadku gdy stan musi być współdzielony pomiędzy wieloma komponentami, React oferuje mechanizm kontekstu (Context API). Pozwala on na przekazywanie danych pomiędzy odległymi elementami drzewa komponentów bez konieczności ręcznego przekazywania ich przez kolejne poziomy hierarchii. Mechanizm ten jest przydatny m.in. do obsługi ustawień aplikacji, motywów graficznych czy globalnych danych, jednak jego nadmierne stosowanie może prowadzić do niepotrzebnych renderów, jeśli struktura kontekstu nie została odpowiednio zaprojektowana.

```
import { createContext, useState } from 'react';

export const AppContext = createContext({
  value: 0,
  setValue: (v) => {}
});

function AppProvider({ children }) {
  const [value, setValue] = useState(0);

  return (
    <AppContext.Provider value={{ value, setValue }}>
      {children}
    </AppContext.Provider>
  );
}
```

Rysunek 1.6: Podstawowy przykład użycia Context API do współdzielenia stanu.

Źródło: Opracowanie własne

W przypadku dużych aplikacji lub projektów o skomplikowanej logice wymiany danych często stosowane są zewnętrzne biblioteki do zarządzania stanem, takie jak Redux, MobX lub Zustand. Redux opiera się na koncepcji pojedynczego źródła prawdy (store) oraz niezmienności danych, co zapewnia wysoki stopień przewidywalności i ułatwia debugowanie, lecz wymaga bardziej rozbudowanej konfiguracji. MobX z kolei stosuje podejście reaktywne, automatycznie monitorując zależności i aktualizując widok tylko tam, gdzie zaszły zmiany. Nowsze rozwiązania, takie jak Zustand, upraszczają zarządzanie stanem, wprowadzając prosty i deklaratywny interfejs oparty na hookach.

Zarządzanie stanem ma bezpośredni wpływ na wydajność aplikacji React. Niewłaściwie zaprojektowana struktura stanów lub nadmierne ich współdzielenie może prowadzić do wielokrotnych i niepotrzebnych renderów komponentów, co negatywnie wpływa na płynność działania interfejsu. Z tego względu niezwykle istotne jest świadome korzystanie z hooków, optymalizacja zakresu kontekstu oraz unikanie przechowywania danych globalnych, które nie muszą być dostępne we wszystkich komponentach. React oferuje również narzędzia umożliwiające optymalizację renderowania, takie jak useMemo, useCallback czy React.memo, które pozwalają ograniczyć liczbę aktualizacji poprzez zapamiętywanie wyników obliczeń lub renderów komponentów.

```
import { memo, useMemo } from 'react';

const Result = memo(function Result({ items }) {
  const sum = useMemo(() => items.reduce((a, b) => a + b, 0), [
   items]);
  return <div>{sum}</div>;
});
```

Rysunek 1.7: Przykład optymalizacji renderowania komponentu przy użyciu `useMemo` oraz `React.memo`.

Źródło: Opracowanie własne

Podsumowując, React dostarcza elastyczne mechanizmy zarządzania stanem, umożliwiające realizację zarówno prostych, jak i bardzo zaawansowanych scenariuszy. Wybór odpowiedniego podejścia zależy od wielkości i charakterystyki aplikacji, jednak świadome wykorzystanie dostępnych narzędzi ma kluczowe znaczenie dla utrzymania przejrzystości kodu oraz zapewnienia wysokiej wydajności renderowania interfejsu.

1.4.3 Optymalizacja wydajności w aplikacjach React

Wydajność renderowania jest jednym z kluczowych aspektów tworzenia aplikacji w React, szczególnie w przypadku interfejsów wymagających częstych aktualizacji stanu lub operujących na dużych zbiorach danych. Chociaż React dzięki mechanizmowi Virtual DOM ogranicza liczbę niezbędnych operacji na rzeczywistym drzewie DOM, to nadmierne i niekontrolowane renderowanie komponentów może nadal prowadzić do spadków płynności działania aplikacji. Z tego powodu React oferuje szereg narzędzi oraz wzorców umożliwiających optymalizację procesu renderowania i minimalizację liczby niepotrzebnych aktualizacji widoku.

Jednym ze sposobów optymalizacji jest wykorzystanie funkcji `React.memo`, która umożliwia zapamiętywanie wyników renderowania komponentów funkcyjnych. Jeśli przekazywane do komponentu właściwości nie uległy zmianie, React renderuje go ponownie wyłącznie wtedy, gdy jest to konieczne. Mechanizm ten jest szczególnie skuteczny w przypadku komponentów o dużej złożoności, które w przeciwnym razie byłyby aktualizowane przy każdej zmianie stanu ich komponentów nadrzędnych.

Drugim istotnym narzędziem są hooki `useMemo` oraz `useCallback`. Pierwszy z nich pozwala na zapamiętywanie wyników kosztownych obliczeń, dzięki czemu funkcja jest ponownie wykonywana tylko wtedy, gdy zmienią się wartości zależności. Z kolei `useCallback` umożliwia zapamiętanie referencji do funkcji, co zapobiega niepotrzebnemu przekazywaniu nowych instancji funkcji jako właściwości komponentów podrzędnych.

Ma to szczególne znaczenie w sytuacjach, gdy komponenty podrzędne są opakowane w `React.memo` i reagują na zmiany referencji funkcji.

Optymalizacja może obejmować również odpowiednie zarządzanie strukturą stanu. Przechowywanie zbyt dużej liczby danych w stanie globalnym lub dzielenie się stanem pomiędzy zbyt wieloma komponentami prowadzi do nadmiarowych renderów. Z tego względu stan aplikacji powinien być możliwie jak najbardziej lokalny, a dane globalne należy wykorzystywać tylko wtedy, gdy faktycznie są współdzielone pomiędzy różnymi częściami interfejsu. Pomocne mogą być także techniki takie jak dzielenie komponentów na mniejsze elementy, aby ograniczyć zakres renderowania tylko do tych fragmentów, które faktycznie uległy zmianie.

W aplikacjach wymagających wysokiej wydajności warto również stosować mechanizmy kolejkowania i opóźniania aktualizacji, dostępne poprzez `useTransition` lub `useDeferredValue`. Funkcje te wprowadzono w wersjach Reacta wspierających renderowanie współbieżne (Concurrent Mode), co pozwala na priorytetyzowanie aktualizacji i zapewnia płynność interfejsu nawet w warunkach dużego obciążenia. Przykładowo, `useTransition` pozwala oznaczyć mniej priorytetowe operacje (np. odświeżenie wizualizacji algorytmu), dzięki czemu interakcje użytkownika (np. suwak prędkości) pozostają responsywne.

```jsx
import { useState, useTransition, useDeferredValue } from 'react
    ';

function SortingVisualizer({ data }) {
  const [isPending, startTransition] = useTransition();
  const deferredData = useDeferredValue(data);

  const handleUpdate = (newData) => {
    startTransition(() => {
      // Lower priority operation
      setSortState(newData);
    });
  };

  return (
    <div style={{ opacity: isPending ? 0.8 : 1 }}>
      {/* Visualization based on deferred data */}
      <Bars items={deferredData} />
    </div>
  );
}
```

Rysunek 1.8: Wykorzystanie hooków `useTransition` i `useDeferredValue` do zarządzania priorytetami renderowania.

Źródło: Opracowanie własne

Oprócz narzędzi programistycznych React udostępnia także profiler, umożliwiający analizę czasu renderowania poszczególnych komponentów. Profilowanie jest kluczowym elementem optymalizacji, ponieważ pozwala precyzyjnie ustalić, które elementy interfejsu generują nadmiarowe aktualizacje lub wykonują kosztowne operacje podczas renderowania.

Podsumowując, React oferuje rozbudowany zestaw narzędzi wspierających optymalizację wydajności aplikacji, obejmujący zarówno kontrolę nad procesem renderowania komponentów, jak i zarządzanie stanem oraz strukturą aplikacji. Świadome stosowanie dostępnych mechanizmów, takich jak `React.memo`, `useMemo`, `useCallback` czy funkcje wspierające renderowanie współbieżne, jest kluczowe dla tworzenia płynnych i responsywnych interfejsów, szczególnie w projektach wymagających dynamicznych i częstych aktualizacji widoku.

1.4.4 Integracja React z TypeScript

Integracja React z TypeScript stanowi obecnie standardowe podejście do tworzenia nowoczesnych aplikacji frontendowych. TypeScript, jako nadzbiór JavaScriptu wprowadzający statyczne typowanie, pozwala na wykrywanie błędów już na etapie kompilacji oraz zapewnia większą kontrolę nad strukturą danych. W połączeniu z deklaratywnym i komponentowym charakterem Reacta umożliwia to zwiększenie czytelności kodu, łatwiejsze utrzymanie projektu oraz wyższą przewidywalność działania aplikacji.

TypeScript znajduje zastosowanie zarówno w definicjach propsów komponentów, jak i podczas opisywania struktur stanu, funkcji pomocniczych, interfejsów oraz modeli danych obsługiwanych przez aplikację. Dzięki temu programista ma możliwość precyzyjnego określenia typów przekazywanych pomiędzy komponentami, co znacząco ogranicza ryzyko wystąpienia błędów wynikających z niezgodności typów. Typowanie jest szczególnie istotne w aplikacjach o dużej liczbie komponentów lub rozbudowanym systemie zarządzania stanem.

Kluczowym elementem integracji Reacta z TypeScriptem jest użycie rozszerzenia TSX (TypeScript XML), które umożliwia łączenie składni TypeScriptu z deklaratywnym opisem komponentów Reacta. Pliki z rozszerzeniem `.tsx` pozwalają na stosowanie składni JSX wraz z pełnym wsparciem typów, co umożliwia definiowanie komponentów w sposób zbliżony do tradycyjnego JSX, ale z dodatkowymi możliwościami walidacji typów podczas kompilacji. TSX zapewnia również podpowiedzi składniowe (intellisense), ułatwia pracę z IDE (ang. Integrated Development Environment) oraz zwiększa czytelność kodu dzięki możliwości opisywania typów dla propsów, obiektów i funkcji bezpośrednio w definicji komponentu.

TypeScript wspiera także typowanie hooków Reacta, takich jak `useState`, `useReducer`, `useRef` czy `useMemo`, co umożliwia tworzenie bardziej przewidywalnych i stabilnych struktur stanu. Przykładowo, zdefiniowanie typu przechowywanej wartości w `useState` pozwala uniknąć nieprawidłowych aktualizacji lub błędów wynikających z niezgodności typów. Z kolei w przypadku kontekstu Reacta (Context API) możliwość typowania wartości przekazywanych pomiędzy komponentami ułatwia tworzenie skalowalnych aplikacji z jednoznacznie zdefiniowanymi strukturami danych.

Integracja z TypeScriptem wpływa również na proces budowy i utrzymania aplikacji. Kompilator TypeScriptu umożliwia wczesne wykrywanie błędów, co skraca czas debugowania oraz zwiększa bezpieczeństwo wdrażania kolejnych zmian. W większych projektach TypeScript poprawia współpracę zespołu przez jednoznaczne definiowanie interfejsów i kontraktów pomiędzy komponentami, co ogranicza ryzyko błędnej interpretacji danych lub nieprzewidzianych zmian w strukturze aplikacji.

Podsumowując, React w połączeniu z TypeScriptem stanowi wydajne i przewidywalne środowisko do tworzenia nowoczesnych aplikacji webowych. Integracja mechanizmów typowania, obsługi plików TSX oraz zaawansowanych narzędzi analizy statycznej pozwala na budowę skalowalnych i dobrze zorganizowanych projektów, w których elementy interfejsu oraz logika biznesowa są jasno zdefiniowane i łatwe do utrzymania.

1.5 Charakterystyka Angular

Angular to kompleksowa platforma i framework programistyczny typu open source, rozwijany przez firmę Google, przeznaczony do budowy rozbudowanych, skalowalnych aplikacji webowych [9]. W przeciwieństwie do Reacta, Angular oferuje pełny, zintegrowany ekosystem narzędzi (*batteries-included approach*), dostarczając ustandaryzowane rozwiązania dla DI (Dependency Injection), obsługi formularzy, routingu oraz komunikacji asynchronicznej [10].

1.5.1 Architektura komponentowa i nowoczesne mechanizmy reaktywności

Architektura Angulara opiera się na hierarchii komponentów i ścisłej separacji logiki od widoku. W przeciwieństwie do Reacta, Angular standardowo oddziela definicję szablonu (HTML), stylów (CSS) oraz logiki (TypeScript) na poziomie plików lub odrębnych sekcji wewnątrz dekoratora komponentu.

```
@Component({
  selector: 'app-user',
  standalone: true,
  template: `
    <div class="user-profile">
      <h2>Profil: {{ username }}</h2>
      <button (click)="updateStatus()">Aktualizuj</button>
    </div>
  `
})
export class UserComponent {
  username = 'Jan Kowalski';
  updateStatus() { /* logika */ }
}
```

Rysunek 1.9: Struktura komponentu w Angularze z wykorzystaniem dekoratora `@Component`.

Źródło: Opracowanie własne

Współczesny Angular (od wersji 17+) przeszedł rewolucję dzięki wprowadzeniu Sygnałów (*Signals*). Sygnały pozwalają frameworkowi na precyzyjne śledzenie zależności (*fine-grained reactivity*), co oznacza, że Angular wie dokładnie, która część szablonu zależy od której wartości stanu.

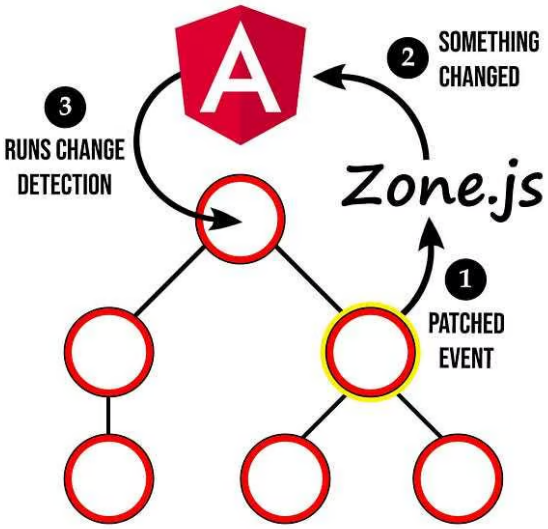
Tradycyjny model detekcji zmian oparty na `Zone.js` monitorował wszystkie asynchroniczne zdarzenia i sprawdzał całe drzewa komponentów. Podejście oparte na Sygnałach pozwala na budowę aplikacji typu *Zoneless*, gdzie proces aktualizacji widoku jest wyzwalany tylko tam, gdzie faktycznie nastąpiła zmiana danych, co drastycznie redukuje narzut obliczeniowy w aplikacjach o wysokiej dynamice, takich jak wizualizery algorytmów.

Kluczową koncepcją organizacyjną do niedawna były moduły (*NgModules*), które grupowały logicznie powiązane komponenty, dyrektywy, potoki i serwisy. Moduły pozwalały na precyzyjne zarządzanie widocznością elementów oraz optymalizację procesu ładowania aplikacji poprzez technologię *lazy loading*. W najnowszych wersjach frameworka wprowadzono tzw. *Standalone Components*, które upraszczają architekturę, eliminując konieczność definiowania modułów dla każdego elementu, co redukuje tzw. *boilerplate code*. Terminem tym określa się powtarzalne fragmenty kodu or charakterze konfiguracyjnym, które są niezbędne do poprawnego działania struktur programu, lecz nie wnoszą bezpośredniej logiki biznesowej i mogą utrudniać analizę właściwej funkcjonalności aplikacji.

Fundamentalnym mechanizmem Angulara, determinującym jego wydajność, jest system detekcji zmian (*Change Detection*). Framework monitoruje stan aplikacji i automatycznie aktualizuje widok, gdy wykryje zmiany w modelach danych. Przez lata mechanizm ten opierał się na bibliotece *Zone.js*, która monitoruje asynchroniczne zdarzenia i uruchamia proces sprawdzania całego drzewa komponentów. Nowoczesne wersje Angulara wprowadzają wspomniany mechanizm *Signals*, który pozwala na precyzyjne śledzenie zależności i informuje framework dokładnie o tym, która część szablonu wymaga odświeżenia. Umożliwia to znaczącą redukcję narzutu obliczeniowego, szczególnie w aplikacjach z dużą liczbą dynamicznych aktualizacji.

1.5.2 Zarządzanie stanem w Angular

Zarządzanie stanem w Angularze jest nierozerwalnie związane z programowaniem reaktywnym i biblioteką RxJS. Framework ten promuje model asynchronicznego przepływu danych, gdzie serwisy pełnią rolę ”magazynów stanu”. Wykorzystują one obiekty typu `Subject` lub `BehaviorSubject` do emitowania aktualnych wartości, które są konsumowane przez komponenty za pomocą subskrypcji.



Rysunek 1.10: Tradycyjny model detekcji zmian w Angularze oparty na Zone.js (sprawdzanie od góry drzewa).

Źródło:

Podejście oparte na RxJS pozwala na potężną manipulację strumieniami danych poprzez operatory takie jak `map`, `filter` czy `switchMap` [11]. Jest to kluczowe w wizualizatorach algorytmów, gdzie kroki sortowania mogą być emitowane jako strumień, który następnie jest ”spowolniany”, aby zapewnić czytelną animację.

```
@Injectable({ providedIn: 'root' })
export class SortingStateService {
  private barsSubject = new BehaviorSubject<number[]>([]);
  bars$ = this.barsSubject.asObservable();

  updateState(newBars: number[]) {
    this.barsSubject.next([...newBars]);
  }
}
```

Rysunek 1.11: Przykład zaawansowanego serwisu w Angularze wykorzystującego RxJS do zarządzania stanem wizualizacji.

Źródło: Opracowanie własne

Nowoczesny Angular wprowadza Sygnały jako alternatywę dla stanu synchronicznego. Sygnały eliminują potrzebę ręcznego zarządzania subskrypcjami i oferują naturalną składnię dla operacji takich jak obliczanie wartości pochodnych (`computed`) czy wywoływanie efektów (`effect`).

```typescript
@Component({
  selector: 'app-visualizer',
  standalone: true,
  template: `<div>Value: {{ count() }}</div>`
})
export class VisualizerComponent {
  // Signal definition
  count = signal(0);

  // Computed value automatically updated
  doubleCount = computed(() => this.count() * 2);

  increment() {
    this.count.update(v => v + 1);
  }
}
```

Rysunek 1.12: Zarządzanie stanem w Angularze przy użyciu mechanizmu Sygnałów.

Źródło: Opracowanie własne

W bardzo dużych aplikacjach stosuje się NgRx, który implementuje wzorzec Redux, zapewniając pełną przewidywalność zmian stanu i ułatwiając debugowanie.

1.5.3 Optymalizacja wydajności w aplikacjach Angular

Wydajność w Angularze zależy przede wszystkim od efektywności procesu detekcji zmian. Kluczową techniką jest strategia `OnPush` (*ChangeDetectionStrategy.OnPush*), która instruuje framework, aby sprawdzał komponent tylko wtedy, gdy zmienią się referencje jego wejściowych właściwości (`@Input()`) lub zostanie wyemitowane zdarzenie. Zastosowanie `OnPush` w połączeniu z niezmiennością danych (*immutability*) pozwala na drastyczne zredukowanie liczby operacji detekcji zmian, co jest krytyczne dla zachowania płynności animacji w wizualizatorze.

Inne techniki optymalizacji obejmują:

- **Kompilator Ivy:** Generuje mniejsze pakiety danych dzięki mechanizmowi *tree-shaking* oraz przyspiesza proces uruchamiania aplikacji.

- **TrackBy:** Podczas renderowania list, funkcja `trackBy` pozwala Angularowi zidentyfikować, które elementy DOM odpowiadają rekordom w danych, unikając niszczenia i ponownego tworzenia całego DOM-u przy każdej zmianie w tablicy.

- **Deferrable Views:** Nowa funkcja umożliwiająca deklaratywne opóźnianie ładowania i renderowania fragmentów szablonu do czasu, gdy np. stają się widoczne na ekranie (*viewport visibility*).
- **Pure Pipes:** Stosowanie czystych potoków, które są uruchamiane tylko wtedy, gdy ich wejście ulegnie zmianie, co oszczędza cykle procesora na transformacje danych.

1.5.4 Angular i TypeScript – natywna integracja

Angular to framework zbudowany ”przez TypeScript i dla TypeScripta”. To partnerstwo zaowocowało modelem programowania, w którym cechy języka są fundamentem działania frameworka. Wszystkie kluczowe mechanizmy — komponenty, serwisy, moduły — są definiowanymi jako klasy TypeScript z dekoratorami (np. `@Component`, `@Injectable`). Dekoratory dołączają metadane do klas, informując kompilator o ich przeznaczeniu, co pozwala na wyraźne oddzielenie konfiguracji od logiki biznesowej.

Statyczne typowanie w Angularze przynosi korzyści w postaci:

- **Type-safe Dependency Injection:** System DI wykorzystuje typy parametrów w konstruktorze do identyfikacji zależności, co zapobiega błędom w czasie uruchomienia.
- **Strict Template Checking:** Kompilator Angulara potrafi sprawdzić poprawność typów wewnątrz szablonów HTML, co drastycznie redukuje liczbę błędów typu *undefined*.
- **Zaawansowane narzędzia IDE:** Dzięki statycznej analizie kodu, deweloperzy mogą korzystać z błyskawicznej nawigacji i automatycznej refaktoryzacji nawet w bardzo rozbudowanych projektach.

Podsumowując, natywna integracja z TypeScriptem sprawia, że Angular jest frameworkiem wyjątkowo stabilnym i skalowalnym, idealnym do budowy złożonych systemów wymagających wysokiej precyzji i wydajności.

1.6 Podstawy algorytmów sortujących i ich wizualizacji

Algorytmy sortowania stanowią jeden z fundamentalnych obszarów informatyki, służąc do porządkowania elementów w zbiorze według określonej relacji [12]. W kontekście niniejszej pracy pełnią one rolę generatora intensywnego obciążenia dla silników renderujących, ponieważ proces ich wykonywania wiąże się z sekwencją licznych porównań i zamian miejsc elementów zbioru.

1.6.1 Klasyfikacja i charakterystyka wybranych algorytmów sortujących

Algorytmy sortujące można klasyfikować według wielu kryteriów, z których najważniejszymi są złożoność czasowa oraz stabilność. Złożoność czasowa określa, jak czas wykonania algorytmu rośnie wraz ze wzrostem liczby elementów (n). Wyróżniamy złożoność optymistyczną, średnią oraz pesymistyczną, przy czym ta ostatnia jest kluczowa dla określenia granic wydajności systemu.

Kolejnym ważnym parametrem jest złożoność pamięciowa (*space complexity*), określająca ilość dodatkowej pamięci potrzebnej do wykonania operacji. Algorytmy działające w miejscu (*in-place*) są preferowane w środowiskach o ograniczonych zasobach, takich jak przeglądarki mobilne. Stabilność algorytmu natomiast informuje o tym, czy zachowuje on relatywną kolejność elementów o tych samych kluczach, co może być istotne przy sortowaniu obiektów o wielu atrybutach.

W pracy wybrano reprezentatywne algorytmy o różnej charakterystyce wydajnościowej i architektonicznej:

- **Sortowanie bąbelkowe (Bubble Sort):** Jeden z najprostszych algorytmów iteracyjnych o złożoności $O(n^2)$. Choć nieefektywny dla dużych zbiorów, jest idealny do demonstracji mechanizmów detekcji zmian ze względu na bardzo regularną strukturę operacji zamiany (*swap*). W wizualizacji pozwala on na obserwację ”wypływania”największych elementów na koniec tablicy, co jest czytelne dla użytkownika, ale obciążające dla renderera przez dużą liczbę drobnych aktualizacji.

- **Sortowanie przez wstawianie (Insertion Sort):** Algorytm o złożoności $O(n^2)$, który buduje posortowaną część tablicy, wstawiając do niej kolejne elementy. W przeciwieństwie do Bubble Sort, znacznie rzadziej wykonuje operacje zamiany miejsc, co pozwala na porównanie, jak frameworki radzą sobie z przesuwaniem większych bloków danych w strukturze DOM.

- **Sortowanie szybkie (Quick Sort):** Algorytm typu ”dziel i zwyciężaj”o złożoności średniej $O(n \log n)$. Wykorzystuje element osiowy (pivot) do partycjonowania tablicy. Z punktu widzenia wizualizacji jest on wyzwaniem, ponieważ operacje następują w różnych, często odległych od siebie miejscach tablicy, co wymusza na silnikach renderujących częste zmiany w rozproszonych gałęziach drzewa widoku. Jest on kluczowy do testowania wydajności przy operacjach o wysokiej dynamice.

- **Sortowanie przez scalanie (Merge Sort):** Algorytm o stabilnej złożoności $O(n \log n)$, który rekurencyjnie dzieli zbiór na mniejsze części i scala je w uporządkowany sposób. Merge Sort wymaga dodatkowej pamięci ($O(n)$), co w wizualizatorze często

wiąże się z koniecznością wyświetlenia ”tablic pomocniczych”lub animowania procesu kopiowania danych. Pozwala to na testowanie wydajności renderowania przy dynamicznym tworzeniu i niszczeniu elementów interfejsu.

- **Sortowanie stogowe (Heap Sort):** Algorytm wykorzystujący strukturę kopca binarnego o złożoności $O(n \log n)$. Jego wizualizacja jest specyficzna, ponieważ wymaga odwzorowania liniowej tablicy jako struktury drzewiastej. Pozwala to na analizę, jak frameworki radzą sobie z prezentacją tych samych danych w dwóch różnych formach graficznych jednocześnie.

```
async function bubbleSort(array: number[], update: (arr: number
    []) => void) {
  const n = array.length;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (array[j] > array[j + 1]) {
        // Swap elements
        [array[j], array[j + 1]] = [array[j + 1], array[j]];

        // State update and forcing a break for the renderer
        update([...array]);
        await new Promise(resolve => setTimeout(resolve, 10));
      }
    }
  }
}
```

Rysunek 1.13: Szkielet asynchronicznego algorytmu sortowania (Bubble Sort) przygotowany pod wizualizację.

Źródło: Opracowanie własne

1.6.2 Metody wizualizacji algorytmów

Wizualizacja algorytmów sortowania w aplikacjach webowych sprowadza się do graficznej reprezentacji elementów zbioru (najczęściej jako słupków o różnej wysokości, gdzie wysokość odpowiada wartości elementu) oraz dynamicznej aktualizacji ich wyglądu w czasie rzeczywistym.

Efektywna wizualizacja musi uwzględniać nie tylko techniczne aspekty renderowania, ale również percepcję użytkownika. Oznacza to konieczność wprowadzenia opóźnień (*delays*) między krokami algorytmu, aby proces był widoczny dla ludzkiego oka, przy jednoczesnym zachowaniu płynności animacji przejść.

Istnieją dwie główne architektury synchronizacji logiki algorytmu z widokiem:

- **Podejście oparte na migawkach (Snapshot-based):** Algorytm jest wykonywany w izolacji od warstwy widoku, a każdy jego ”krok”(porównanie, zamiana) jest zapisywany jako stan tablicy w dedykowanej kolejce (historii ruchów). Po zakończeniu obliczeń, framework odtwarza historię, renderując stany jeden po drugim z określonym interwałem. Metoda ta gwarantuje stabilność interfejsu, ale opóźnia rozpoczęcie wizualizacji do zakończenia obliczeń, co przy rekordowo dużych zbiorach może być odczuwalne.
- **Podejście asynchroniczne w czasie rzeczywistym (Real-time Async):** Algorytm jest wykonywany bezpośrednio w głównej nitce (lub w Web Workerze), a po każdej operacji następuje asynchroniczne przerwanie (`await sleep(ms)`). Podczas tej pauzy, stan jest aktualizowany w frameworku, co wyzwala natychmiastowe renderowanie. Jest to podejście bardziej zbliżone do rzeczywistych aplikacji interaktywnych i pozwala na testowanie reaktywności frameworka pod ciągłym obciążeniem.

Wizualizacja wymaga również precyzyjnego zarządzania atrybutami graficznymi. Kluczowe jest oznaczanie kolorami:

- **Wybór i porównanie:** Elementy aktualnie porównywane są zazwyczaj wyróżniane kontrastowym kolorem (np. czerwonym).
- **Zamiana:** Elementy zmieniające pozycję mogą pulsować lub zmieniać nasycenie koloru.
- **Uporządkowanie:** Fragmenty tablicy, które algorytm uznał za ostatecznie posortowane, są oznaczane kolorem ”bezpiecznym”(np. zielonym).

Płynność tych zmian jest determinowana przez to, jak szybko framework potrafi zidentyfikować zmienione atrybuty CSS (komponentów lub elementów DOM) i przesłać je do silnika kompozycji przeglądarki. Optymalizacja tych przejść (np. poprzez użycie *CSS Transitions* lub *Transforms* zamiast zmiany szerokości/pozycji *top/left*) jest niezbędna dla uniknięcia zjawiska *layout thrashing*.

W dalszej części pracy, oba te podejścia zostaną zaimplementowane i poddane testom obciążeniowym, co pozwoli na ocenę, jak architektura Reacta (Virtual DOM) oraz Angulara (Signals/OnPush) radzi sobie z zarządzaniem setkami dynamicznie zmieniających się obiektów graficznych.

ROZDZIAŁ DRUGI

Metodyka badań i implementacja aplikacji

2.1 Założenia projektowe i wymagania funkcjonalne dla wizualizatora algorytmów

Głównym założeniem projektowym platformy badawczej jest stworzenie dwóch funkcjonalnie identycznych aplikacji, które umożliwią obiektywne porównanie wydajności renderowania w frameworkach React i Angular. Wizualizator musi sprostać wyzwaniu płynnego wyświetlania setek operacji na danych przy zachowaniu wysokiej interaktywności interfejsu.

Do kluczowych wymagań funkcjonalnych zaliczono:

- **Wielowątkowość logiczna:** Możliwość jednoczesnej wizualizacji wielu algorytmów (Bubble Sort, Quick Sort, Merge Sort, Heap Sort) w celu porównania ich dynamiki pracy.
- **Sterowanie czasem rzeczywistym:** Użytkownik musi mieć możliwość płynnej regulacji prędkości animacji (opóźnienia między krokami) w zakresie od 50 ms do 600 ms.
- **Interaktywność:** Możliwość wstrzymywania (Pause), wznawiania (Play), restartowania oraz losowania nowych danych (Shuffle) bez konieczności przeładowywania aplikacji.
- **Precyzja wizualna:** Wyraźne rozróżnienie stanów elementów poprzez kolory: porównywanie (amber), zamiana (red), oś podziału/pivot (blue) oraz elementy posortowane (emerald).

Projekt techniczny zakłada wykorzystanie TypeScriptu jako nadrzędnego języka programowania, co zapewnia spójność typów pomiędzy logiką algorytmów a komponentami interfejsu użytkownika.

2.2 Opis implementacji aplikacji wizualizującej w React

Aplikacja w wersji React została zbudowana z wykorzystaniem najnowszych standardów biblioteki (wersja 19) oraz nowoczesnych narzędzi ekosystemu JavaScript.

2.2.1 Struktura projektu i kluczowe komponenty

Architektura aplikacji opiera się na separacji logiki biznesowej od warstwy prezentacji. Główne elementy struktury to:

- **Routy aplikacji:** Wykorzystanie React Router v7 do obsługi nawigacji między stroną powitalną (*Landing Page*) a głównym panelem badawczym (*Dashboard*).
- **Komponenty UI:** Zastosowanie modułowej biblioteki komponentów (opartej na Tailwind CSS), co zapewnia spójność wizualną i szybki czas budowy interfejsu.
- **SortingProgressChart:** Centralny komponent wizualizacji, odpowiedzialny za renderowanie słupków danych z wykorzystaniem biblioteki Recharts.

2.2.2 Wykorzystane biblioteki i narzędzia

W projekcie wykorzystano następujący stos technologiczny:

- **Vite:** Jako szybki system budowania i serwer deweloperski [13].
- **Tailwind CSS v4:** Do stylizacji z wykorzystaniem nowoczesnych funkcji takich jak *glassmorphism* i zaawansowane gradienty [14].
- **Shadcn/ui** : Biblioteka komponentów UI oparta na Tailwind CSS, oferująca gotowe elementy interfejsu użytkownika.
- **Recharts:** Do efektywnego renderowania wykresów słupkowych przy częstych aktualizacjach stanu.

2.2.3 Podejście do zarządzania stanem

Zarządzanie stanem wizualizacji zrealizowano za pomocą autorskiego hooka `useSorting`. Implementacja opiera się na podejściu opartym na migawkach (*Snapshot-based*). Algorytmy generują listę kroków (`SortStep`), gdzie każdy krok zawiera aktualny stan tablicy oraz informacje o porównywanych indeksach.

```typescript
export type SortingProgress = {
  values: number[];
  comparing?: number[];
  swapping?: number[];
  pivot?: number | null;
  sorted?: number[];
}
```

Rysunek 2.1: Struktura interfejsu opisująca krok wizualizacji w React.

Źródło: Opracowanie własne

Hook `useSorting` zarządza indeksem aktywnego kroku oraz interwałem czasowym za pomocą `window.setInterval`, co pozwala na odseparowanie tempa animacji od szybkości renderowania samej biblioteki React.

2.2.4 Specyficzne techniki optymalizacyjne

Aby zapewnić płynność wizualizacji przy setkach aktualizacji na sekundę, zastosowano następujące techniki:

- **React.memo:** Komponent `SortingProgressChart` został opakowany w `memo`, dzięki czemu nie jest on renderowany ponownie, jeśli dane wizualizacji nie uległy zmianie.
- **useMemo dla zbiorów danych:** Indeksy porównywane i zamieniane są konwertowane na obiekty typu `Set` za pomocą `useMemo`. Pozwala to na uzyskanie złożoności $O(1)$ przy sprawdzaniu koloru każdego słupka podczas renderowania.
- **Separacja logiki:** Obliczanie wszystkich kroków algorytmu odbywa się przed rozpoczęciem animacji, co odciąża główny wątek podczas procesu renderowania.

2.3 Opis implementacji aplikacji wizualizującej w Angular

Wersja aplikacji przygotowana w frameworku Angular została zaimplementowana z wykorzystaniem najnowszego stabilnego wydania (wersja 19), co pozwala na pełne wykorzystanie nowoczesnych mechanizmów reaktywności oraz optymalizacji renderowania.

2.3.1 Struktura projektu i nowoczesna architektura

Architektura aplikacji Angular została uproszczona dzięki zastosowaniu komponentów typu *Standalone*, co eliminuje konieczność definiowania tradycyjnych modułów `NgModule`. Projekt zorganizowano w sposób modułowy, dzieląc kod na:

- **Core Services:** Serwisy odpowiedzialne za logikę algorytmów oraz zarządzanie stanem wizualizacji, wstrzykiwane jako singletony.
- **Shared Components:** Wielokrotnie wykorzystywane elementy interfejsu użytkownika, takie jak przyciski sterujące, suwaki oraz kontenery układu.
- **Visualizer Dashboard:** Główny ekran aplikacji integrujący parametry wejściowe z dynamicznym wykresem słupkowym.

2.3.2 Wykorzystane biblioteki i narzędzia

Stos technologiczny wersji Angular został dobrany tak, aby był bezpośrednim odpowiednikiem narzędzi użytych w wersji React:

- **Angular CLI:** Oficjalne narzędzie do zarządzania cyklem życia aplikacji i procesem budowania.
- **Tailwind CSS v4:** Identyczna konfiguracja stylizacji jak w wersji React, zapewniająca spójność wizualną.
- **Ng-Zorro-Antd lub PrimeNG (wybrane komponenty):** Biblioteki wspomagające budowę zaawansowanych elementów sterujących.
- **Ngx-Charts / Highcharts:** Biblioteka do renderowania wykresów (odpowiednik Recharts), zoptymalizowana pod kątem wydajności w ekosystemie Angular.

2.3.3 Podejście do zarządzania stanem z użyciem Sygnałów

Kluczowym elementem implementacji jest wykorzystanie mechanizmu Angular Signals do zarządzania stanem wizualizacji. Zamiast tradycyjnego podejścia opartego na strumieniach RxJS, zastosowano sygnały (*WritableSignal*), co pozwala na precyzyjne odświeżanie wyłącznie tych fragmentów widoku, które są bezpośrednio powiązane ze zmianą kroku algorytmu.

```typescript
export interface SortingState {
  values: number[];
  comparingIndices: number[];
  swappingIndices: number[];
  pivotIndex: number | null;
  sortedIndices: number[];
}

// Inicjalizacja stanu w komponencie
this.sortingState = signal<SortingState>({
  values: [],
  comparingIndices: [],
  swappingIndices: [],
  pivotIndex: null,
  sortedIndices: []
});
```

Rysunek 2.2: Definicja typu stanu wizualizacji oraz inicjalizacja sygnału w Angularze.

Źródło: Opracowanie własne

Aktualizacja widoku odbywa się poprzez metodę `update()`, która wyzwala asynchroniczny proces detekcji zmian specyficzny dla nowego modelu reaktywności Angulara.

2.3.4 Specyficzne techniki optymalizacyjne

W wersji Angular zastosowano zaawansowane techniki mające na celu minimalizację narzutu procesora:

- **ChangeDetectionStrategy.OnPush:** Każdy komponent wizualizacji korzysta z tej strategii, co wyłącza automatyczną detekcję zmian i pozwala na odświeżanie widoku tylko w momencie aktualizacji sygnałów.
- **Zoneless Angular:** Dzięki Sygnałom aplikacja może pracować bez biblioteki `Zone.js`, co eliminuje sprawdzanie całego drzewa komponentów przy każdym zdarzeniu asynchronicznym.
- **TrackBy w pętlach szablonu:** Zastosowanie funkcji identyfikującej słupki po ich unikalnych wartościach lub pozycjach, co zapobiega zbędnemu niszczeniu i tworzeniu elementów DOM.
- **Web Workers (opcjonalnie):** Przeniesienie obliczeń złożonych algorytmów (np. Quick Sort) do oddzielnego wątku, aby uniknąć blokowania głównego wątku renderującego interfejs.

2.4 Metody i procedury pomiarowe dla analizy wydajności

Analiza porównawcza wydajności frameworków React i Angular wymaga zastosowania rygorystycznej i powtarzalnej metodologii pomiarowej. Celem badań jest określenie, jak każda z technologii radzi sobie z intensywnym obciążeniem wynikającym z częstych aktualizacji interfejsu użytkownika w procesie wizualizacji algorytmów.

2.4.1 Środowisko testowe

Wszystkie pomiary wydajności zostały przeprowadzone w kontrolowanym środowisku sprzętowo-programowym, aby wyeliminować wpływ zmiennych zewnętrznych:

- **Sprzęt:** Laptop z procesorem Apple M4 Pro, 24 GB pamięci RAM.
- **System operacyjny:** macOS Tahoe 26.2.
- **Przeglądarka:** Google Chrome (wersja 143), uruchomiona w trybie incognito bez aktywnych rozszerzeń.
- **Tryb pracy aplikacji:** Wersje produkcyjne (po wykonaniu kompilacji `npm run build`), serwowane lokalnie za pomocą serwera statycznego.

2.4.2 Narzędzia i metryki wydajnościowe

Do zbierania danych wykorzystano narzędzie **Chrome DevTools Performance** [15] oraz interfejs **PerformanceObserver** wbudowany w przeglądarkę. Skoncentrowano się na następujących metrykach:

1. **Scripting Time:** Czas procesora poświęcony na wykonanie skryptów JavaScript, w tym logiki algorytmu oraz mechanizmów wewnętrznych frameworka (przetwarzanie Virtual DOM w React, detekcja zmian w Angular).
2. **Rendering & Painting Time:** Czas potrzebny na obliczenie geometrii elementów (Layout) oraz ich odświeżenie graficzne (Paint).
3. **System FPS (Frames Per Second):** Średni klatkaż podczas trwania animacji. Stałe 60 FPS jest uznawane za wzorcową płynność.
4. **Total Blocking Time (TBT):** Sumaryczny czas, w którym główny wątek był zablokowany przez zadania trwające powyżej 50 ms.

5. **Heap Memory Usage:** Wielkość zajętej pamięci przed, w trakcie i po zakończeniu sortowania, mierzona w celu wykrycia niestabilności w zarządzaniu pamięcią.

2.4.3 Procedura badawcza

Badanie zostało podzielone na scenariusze testowe różniące się liczbą elementów (100, 500, 1000) oraz rodzajem algorytmu. Każdy test trwał identyczny czas, a szybkość animacji została zestandaryzowana. Procedura pomiarowa dla każdego scenariusza obejmowała:

- Wyczyszczenie pamięci podręcznej i śmieci (Garbage Collection).
- Uruchomienie profilera nagrywającego ścieżkę wydajności.
- Wykonanie pełnego cyklu sortowania.
- Eksport wyników do formatu JSON w celu późniejszej analizy statystycznej.

Wyniki końcowe prezentowane w pracy są średnią arytmetyczną z 10 pomiarów dla każdego scenariusza, co pozwoliło na zminimalizowanie błędu pomiarowego wynikającego z chwilowych fluktuacji systemowych.

2.5 Metody oceny produktywności programistycznej

Poza analizą wydajnościową, praca podejmuje próbę oceny obu technologii pod kątem ergonomii pracy i produktywności programisty. Jest to aspekt kluczowy przy wyborze stosu technologicznego dla projektów komercyjnych.

2.5.1 Kryteria oceny ilościowej

W ramach oceny obiektywnej przeanalizowano:

- **Rozmiar kodu źródłowego (SLOC):** Liczba linii kodu źródłowego (bez komentarzy i plików konfiguracyjnych) niezbędna do realizacji identycznych funkcjonalności.
- **Rozmiar pakietu wynikowego (Bundle Size):** Wielkość plików przesyłanych do klienta po procesie budowania wersji produkcyjnej, co bezpośrednio wpływa na czas pierwszego załadowania aplikacji.

- **Liczba zależności zewnętrznych:** Analiza konieczności doinstalowywania dodatkowych bibliotek (routing, stan, formularze) w celu uzyskania pełnej funkcjonalności aplikacji.

2.5.2 Kryteria oceny jakościowej

Ocena subiektywna została oparta na doświadczeniach z procesu implementacji i skupiła się na:

- **Czytelności i strukturze:** Analiza przejrzystości separacji logiki od widoku (JSX vs Szablony Angular).
- **Barierze wejścia:** Trudność początkowej konfiguracji i nauki kluczowych konceptów (hooki vs dekoratory i DI).
- **Jakości narzędzi deweloperskich:** Ocena wsparcia ze strony dedykowanych rozszerzeń przeglądarkowych oraz narzędzi CLI.
- **Kompetencji TypeScript:** Stopień wykorzystania i restrykcyjności typowania wymuszany przez framework.

Synteza tych danych pozwoli na sformułowanie wniosków dotyczących tego, który framework oferuje lepszy stosunek wydajności do nakładów pracy programistycznej w kontekście tworzenia interaktywnych wizualizacji danych.

ROZDZIAŁ TRZECI

Prezentacja i analiza wyników badań

3.1 Wyniki pomiarów wydajności renderowania dla obu implementacji

3.2 Analiza zużycia zasobów i płynności animacji

3.3 Porównanie rozmiarów i złożoności pakietów

3.4 Wyniki oceny łatwości rozwoju i produktywności programistycznej

3.5 Porównawcza analiza zastosowania TypeScript w obu frameworkach

ROZDZIAŁ CZWARTY

Dyskusja wyników, wnioski i rekomendacje

4.1 Odpowiedzi na główne i szczegółowe pytania badawcze

4.2 Porównanie mocnych i słabych stron React i Angular w kontekście dynamicznych wizualizacji

4.3 Implikacje praktyczne dla deweloperów i architektów

4.4 Implikacje teoretyczne dla inżynierii oprogramowania

4.5 Ograniczenia przeprowadzonego badania

BIBLIOGRAFIA

[1] MDN Web Docs. JavaScript — Dynamic client-side scripting. https://developer.mozilla.org/en-US/docs/Learn/JavaScript. Accessed: 12.01.2026.

[2] MDN Web Docs. SPA (Single-page application). https://developer.mozilla.org/en-US/docs/Glossary/SPA. Accessed: 12.01.2026.

[3] Google Developers. Web Vitals: Essential metrics for a healthy site. https://web.dev/vitals/. Accessed: 15.01.2026.

[4] ECMA International. ECMAScript 2015 Language Specification. https://262.ecma-international.org/6.0/. Accessed: 12.01.2026.

[5] Microsoft. TypeScript: JavaScript with syntax for types. https://www.typescriptlang.org/. Accessed: 12.01.2026.

[6] Stoyan Stefanov. *React: Up & Running: Building Web Applications*. O'Reilly Media, 2nd edition, 2021.

[7] Meta Open Source. React: The library for web and native user interfaces. https://react.dev/. Accessed: 12.01.2026.

[8] Meta Open Source. React Reference: Concurrent Rendering. https://react.dev/reference/react/useTransition. Accessed: 10.01.2026.

[9] Google. Angular Documentation. https://angular.io/docs. Accessed: 10.01.2026.

[10] Yakov Fain, Anton Moiseev. *Angular Development with TypeScript*. Manning Publications, 2nd edition, 2018.

[11] Paul P. Daniels, Luis Atencio. *RxJS in Action*. Manning Publications, 2017.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

[13] Vite Team. Vite: Next Generation Frontend Tooling. https://vitejs.dev/. Accessed: 12.01.2026.

[14] Tailwind Labs. Tailwind CSS: Rapidly build modern websites without ever leaving your HTML. https://tailwindcss.com/. Accessed: 12.01.2026.

[15] Google. Analyze runtime performance - Chrome DevTools. https://developer.chrome.com/docs/devtools/performance. Accessed: 12.01.2026.

Spis rysunków