Uniwersytet WSB Merito w Poznaniu
Wydział Zamiejscowy w Chorzowie

Jakub Kielaszek

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

Praca magisterska

Kierownik naukowy:
dr Tomasz Staś

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

CHORZÓW 2026

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 [31]), 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 [38]. 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 [46]:

- 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.

Podstawę merytoryczną niniejszej pracy stanowią zróżnicowane źródła, obejmujące zarówno literaturę przedmiotu, jak i oficjalną dokumentację techniczną analizowanych technologii. Kluczowe znaczenie mają publikacje naukowe i podręczniki dotyczące algorytmiki, inżynierii oprogramowania oraz wzorców projektowych w aplikacjach webowych.

Ważny materiał badawczy stanowią również oficjalne zasoby udostępniane przez twórców frameworków React i Angular, specyfikacje standardów ECMAScript oraz artykuły branżowe dokumentujące najlepsze praktyki w zakresie optymalizacji wydajności frontendowej. Całość uzupełniają wyniki badań własnych, uzyskane na drodze implementacji i testów porównawczych autorskiego oprogramowania.

W procesie przygotowania niniejszej pracy wykorzystano narzędzia oparte na sztucznej inteligencji (AI), które wspierały autora w weryfikacji poprawności językowej oraz stylistycznym usprawnieniu treści merytorycznej. Ponadto, w ramach prac programistycznych nad autorskimi aplikacjami wizualizacyjnymi, wykorzystano narzędzie GitHub Copilot jako wsparcie w procesie implementacji kodu oraz optymalizacji wybranych fragmentów logiki.

Praca składa się z czterech zasadniczych rozdziałów. W rozdziale pierwszym przedstawiono fundamenty teoretyczne, ewolucję aplikacji webowych oraz charakterystykę analizowanych 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 [11]. Kolejne podsekcje prezentują przegląd dwóch popularnych technologii — React i Angular — wraz z ich kluczowymi założeniami architektonicznymi i podejściami do renderowania interfejsu użytkownika [47]. 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 [11].

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. Pozwoliło to na stworzenie bardziej interaktywnych aplikacji oraz znacząco poprawiło komfort 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 [39].

Dynamiczny rozwój technologii frontendowych doprowadził do powstania Single Page Applications (*aplikacji jednosesyjnych*, SPA), które stanowią obecnie dominujący model budowy interfejsów webowych. SPA charakteryzują się tym, że cała aplikacja jest ładowana jednorazowo, a kolejne interakcje użytkownika prowadzą do aktualizacji tylko tych elementów, które faktycznie ulegają zmianie [31]. 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 [47]. 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 [47, 22].

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 [21]. 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 (*kluczowe wskaźniki internetowe*) pozwala na obiektywną ocenę jakości interakcji użytkownika z witryną [19]. 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 [20]. 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 najważ-

niejszych wyzwań współczesnego frontendu [38]. 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 [28]. 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 event loop (*pętli zdarzeń*), który umożliwia asynchroniczne przetwarzanie operacji bez blokowania interfejsu użytkownika [32].

Istotny wpływ na rozwój JavaScriptu miało wprowadzenie standardu ECMAScript 6 (ES6), który rozbudował język o nowoczesne mechanizmy wspierające programowanie strukturalne i modularne. 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 [38].

Drugim kluczowym elementem ekosystemu frontendowego jest TypeScript — nadzbiór JavaScriptu opracowany przez Microsoft, wprowadzający statyczne typowanie oraz szereg mechanizmów znanych z języków obiektowych. 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 [45]. 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 [39], 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 [47]. W odpowiedzi na te potrzeby powstały liczne bi-

blioteki i frameworki frontendowe, reprezentujące odmienne paradygmaty projektowania i przetwarzania danych w kontekście interfejsu użytkownika [47].

Jednym z kluczowych paradygmatów, który znacząco wpłynął na sposób budowania interfejsów webowych, jest programowanie komponentowe [2]. 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 [3]. 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 [55]. 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 [54]. 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. React szybko zyskał szerokie zastosowanie w przemyśle dzięki swojej prostocie, modularności oraz nowatorskiemu podejściu do renderowania komponentów [36].

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 [36]. 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 [47].

React wprowadza komponentową strukturę aplikacji, w której interfejs podzielony jest na niewielkie, niezależne moduły odpowiadające za określoną funkcjonalność lub fragment widoku [2]. 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 [47].

```jsx
import React from 'react';

const Welcome = ({ name }) => {
  return (
    <div className="container">
      <h1>Witaj, {name}!</h1>
      <p>Komponenty React łączą logikę 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 [34]. 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 [47].

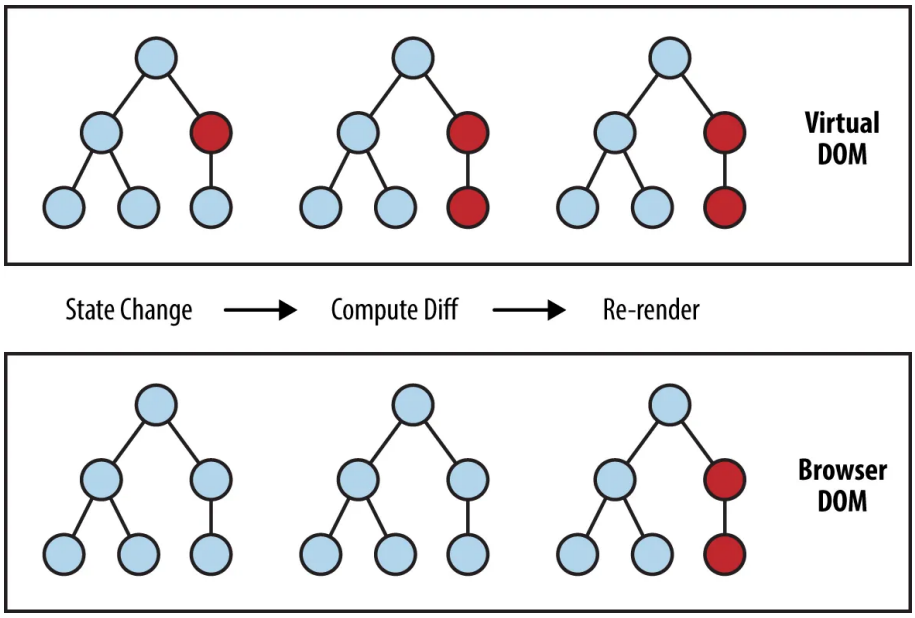
1.4.1 Architektura i kluczowe koncepcje

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 [2].

Kolejną kluczową koncepcją Reacta jest deklaratywny sposób definiowania interfejsu użytkownika [47]. 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 [3].

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 [36].

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 [24]. 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 [54].



Rysunek 1.3: Schemat procesu aktualizacji Virtual DOM i rzeczywistego DOM w React. [51]

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 [35]. 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 [53].

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 [39].

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 unidirectional data flow (*jednokierunkowy przepływ danych*), 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 [47].

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 [34].

```
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 props (*właściwości*). Komponent nadrzędny może udostępnić fragment swo-

jego stanu komponentom podrzędnym, które otrzymują go jako parametry wejściowe. Komunikacja w przeciwnym kierunku — od komponentu podrzędnego do nadrzędnego — realizowana jest poprzez przekazywanie callback functions (*funkcje zwrotne*). W przypadku rozbudowanych struktur, gdzie dane muszą zostać przekazane przez wiele pośrednich poziomów, pojawia się wyzwanie określane jako prop drilling (*przekazywanie właściwości przez wiele poziomów*), co może prowadzić do zmniejszenia czytelności kodu i utrudniać jego konserwację [24].

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 provider (*dostawcy*), 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 [8].

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 dispatching actions (*wysyłanie akcji*) do reducer (*funkcji redukującej*). Podejście to sprzyja separacji logiki biznesowej od warstwy prezentacji i ułatwia testowanie kodu [3].

```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 [36].

```jsx
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 [2].

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 [38].

```
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 [47].

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 [38].

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 [39].

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 [3].

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 [24].

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 [3]), 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 [16].

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 [38].

1.4.4 Integracja React z TypeScript

Integracja React z TypeScript stanowi obecnie standardowe podejście do tworzenia nowoczesnych aplikacji frontendowych [23]. 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 [9].

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 Integrated Development Environment (*zintegrowane środowisko programistyczne*) oraz zwiększa czytelność kodu dzięki możliwości opisywania typów dla propsów, obiektów i funkcji bezpośrednio w definicji komponentu [37].

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 [45].

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 [46].

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 [23].

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 [17]. W przeciwieństwie do Reacta, Angular oferuje pełny, zintegrowany ekosystem narzędzi, realizując batteries-included approach (*podejście z kompletem wbudowanych rozwiązań*), dostarczając ustandaryzowane rozwiązania dla DI (Dependency Injection), obsługi formularzy, routingu oraz komunikacji asynchronicznej.

1.5.1 Architektura komponentowa i nowoczesne mechanizmy reaktywności

Architektura Angulara opiera się na hierarchii komponentów i ścisłej separacji logiki od widoku [55]. 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`.

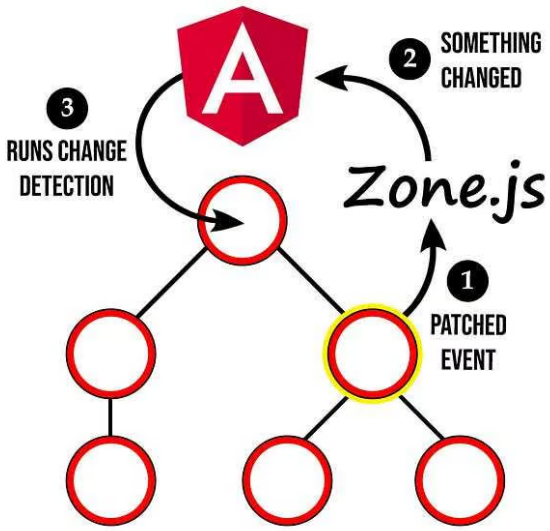
Źródło: Opracowanie własne

Współczesny Angular (od wersji 17+) przeszedł rewolucję dzięki wprowadzeniu signals (*sygnałów*), których szczegółową specyfikację opisano w ramach propozycji zmian architektonicznych [7]. Signals pozwalają frameworkowi na precyzyjne śledzenie zależności — fine-grained reactivity (*reaktywność o drobnej ziarnistości*), co oznacza, że Angular wie dokładnie, która część szablonu zależy od której wartości stanu [17].

Tradycyjny model detekcji zmian oparty na `Zone.js` monitorował wszystkie asynchroniczne zdarzenia i sprawdzał całe drzewa komponentów [55]. 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 [17]. 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*) [17]. 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 [23].



Rysunek 1.10: Tradycyjny model detekcji zmian w Angularze oparty na Zone.js (sprawdzanie od góry drzewa). [6]

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.

Podejście oparte na RxJS pozwala na potężną manipulację strumieniami danych poprzez operatory takie jak `map`, `filter` czy `switchMap` [40]. 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`).

```
@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 [17, 38]:

- **Kompilator Ivy:** Generuje mniejsze pakiety danych dzięki mechanizmowi *tree-shaking* oraz przyspiesza proces uruchamiania aplikacji [25].

- **TrackBy:** Podczas renderowania list, funkcja `trackBy` pozwala Angularowi zidentyfikować, które elementy DOM odpowiadają rekordom w danych [17], 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*) [17].

- **Pure Pipes:** Stosowanie czystych potoków, które są uruchamiane tylko wtedy, gdy ich wejście ulegnie zmianie, co oszczędza cykle procesora na transformacje danych [17].

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 [55, 23]:

- **Type-safe Dependency Injection:** System DI wykorzystuje typy parametrów w konstruktorze do identyfikacji zależności [55], 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* [17].
- **Zaawansowane narzędzia IDE:** Dzięki statycznej analizie kodu, deweloperzy mogą korzystać z błyskawicznej nawigacji i automatycznej refaktoryzacji nawet w bardzo rozbudowanych projektach [23].

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 [50]. Klasyczna literatura przedmiotu definiuje szeroki wachlarz technik optymalizacji tych procesów, które do dziś stanowią bazę dla współczesnych implementacji w silnikach JavaScript [13]. 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 [49]. 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 [38]. Stabilność algorytmu natomiast informuje o tym, czy za-

chowuje 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*) [43]. 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 [12].

- **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 [50].

- **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 [13].

- **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 [50]. 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 [21].

Istnieją dwie główne architektury synchronizacji logiki algorytmu z widokiem [26, 33]:

- **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 [26]. 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 [33]), 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 [1]. Kluczowe jest oznaczanie kolorami [21]:

- **Wybór i porównanie:** Elementy aktualnie porównywane są zazwyczaj wyróżniane kontrastowym kolorem (np. czerwonym) [1].
- **Zamiana:** Elementy zmieniające pozycję mogą pulsować lub zmieniać nasycenie koloru [1].
- **Uporządkowanie:** Fragmenty tablicy, które algorytm uznał za ostatecznie posortowane, są oznaczane kolorem ”bezpiecznym”(np. zielonym) [1].

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 było stworzenie dwóch funkcjonalnie identycznych aplikacji, które umożliwią obiektywne porównanie wydajności renderowania w frameworkach React i Angular. Wybór wizualizacji algorytmów sortowania jako przedmiotu badania wynika z faktu, że proces ten generuje ogromną liczbę operacji na modelu danych w bardzo krótkich interwałach czasowych. Z punktu widzenia inżynierii oprogramowania, operacje te stanowią ”najgorszy możliwy przypadek”(*worst-case scenario*) dla systemów detekcji zmian, ponieważ wymuszają one ciągłą rewalidację całego drzewa komponentów. Wizualizator musi sprostać wyzwaniu płynnego wyświetlania setek operacji na sekundę przy zachowaniu wysokiej interaktywności interfejsu, co stanowi idealny poligon doświadczalny dla mechanizmów renderowania Virtual DOM (React) i reaktywności drobnoziarnistej (Angular). Istotnym aspektem było zapewnienie technologicznej równości — obie aplikacje muszą realizować dokładnie te same zadania matematyczne, korzystając z tych samych konfiguracji systemowych, co eliminuje błąd pomiarowy wynikający z różnić w implementacji logiki biznesowej.

2.1.1 Wymagania funkcjonalne i scenariusze użycia

Projektowana aplikacja musiała spełniać szereg zaawansowanych wymagań funkcjonalnych, które zostały podzielone na wymagania niskopoziomowe (związane z obliczeniami) oraz wysokopoziomowe (związane z interakcją). Do kluczowych funkcji należą:

1. **Wielowątkowość logiczna i porównywanie równoległe**: Możliwość jednoczesnego uruchomienia wielu algorytmów w niezależnych kontenerach wizualnych. Użytkownik może wybrać różne algorytmy (np. Quick Sort vs Merge Sort) i obserwować ich zachowanie na tym samym zbiorze danych wejściowych. Wymaga to od frame-

worków efektywnego zarządzania wieloma niezależnymi strumieniami aktualizacji stanu bez interferencji między nimi.

2. **Determinizm algorytmów**: Każda implementacja algorytmu musi być całkowicie deterministyczna. Oznacza to, że dla danego ziarna (*seed*) generatora liczb losowych oraz tej samej wielkości tablicy, zarówno wersja React, jak i Angular, muszą wygenerować identyczną sekwencję porównań i zamian. Jest to fundament rzetelności naukowej badania, pozwalający przypisać wszelkie różnice czasowe wyłącznie narzutowi frameworka.

3. **Dynamiczne sterowanie czasem wykonania**: System zapewnia płynną regulację prędkości animacji w czasie rzeczywistym. Programista musi mieć możliwość modyfikacji opóźnienia (*delay*) między krokami w zakresie od 1 ms (maksymalne obciążenie) do 1000 ms (tryb edukacyjny). Zmiana tego parametru nie może przerywać trwającej animacji ani powodować błędów synchronizacji w asynchronicznych funkcjach sterujących.

4. **Zarządzanie stanem początkowym danych**: Aplikacja musi umożliwiać generowanie danych o różnej charakterystyce: całkowicie losowe, już posortowane, posortowane odwrotnie oraz z dużą liczbą duplikatów. Każdy z tych przypadków generuje inną liczbę kroków wizualizacji, co pozwala na testowanie stabilności frameworków przy skrajnie różnych obciążeniach.

2.1.2 Standardy percepcji wizualnej i kodowanie kolorystyczne

Z punktu widzenia efektywności przekazu informacji, interfejs wizualizatora wykorzystuje zaawansowany system kodowania kolorystycznego, który stawia dodatkowe wyzwania przed silnikiem renderującym. Każda zmiana koloru słupka w wykresie jest traktowana jako osobna operacja aktualizacji atrybutów elementu graficznego. Przyjęto następujący standard wizualny:

- **Stan spoczynku**: Słupki reprezentowane są w kolorze neutralnym, zapewniającym dobrą czytelność na ciemnym tle interfejsu (*Dark Mode*).

- **Porównanie (Active Reading)**: Elementy aktualnie odczytywane przez wskaźniki algorytmu są podświetlane kontrastowym kolorem bursztynowym. Wymaga to niezwykle szybkiej zmiany klas CSS lub stylów *inline* przy każdym kroku iteracji.

- **Zamiana i modyfikacja (Write/Swap)**: Operacje fizycznej zamiany miejsc w pamięci są sygnalizowane kolorem czerwonym. Jest to moment o największym obcią-

żeniu, ponieważ wiąże się nie tylko ze zmianą koloru, ale również z rekompozycją pozycji w drzewie DOM lub zmianą atrybutów transformacji CSS.

- **Elementy specjalne (Pivoting)**: W algorytmach typu *divide and conquer*, elementy pełniące funkcję punktów odniesienia są wyróżniane kolorem niebieskim, co pozwala na śledzenie struktury rekurencyjnej algorytmu.

2.1.3 Wymagania techniczne i jakość kodu

Całość projektu technicznego oparto na języku TypeScript w wersji 5.6+, co zapewnia najwyższy poziom bezpieczeństwa typologicznego. Wykorzystanie zaawansowanych konceptów języka, takich jak *Generics* i *Discriminated Unions*, pozwoliło na stworzenie jednolitego modelu stanu wizualizacji (*Domain Model*), który jest współdzielony między frameworkami. Architektura została zaprojektowana zgodnie z zasadą *Clean Architecture* — logika matematyczna algorytmów jest całkowicie odizolowana od warstwy widoku (*Presentation Layer*) i wstrzykiwana jako niezależne moduły.

Pod względem technologicznym, obie aplikacje muszą korzystać z identycznych bibliotek pomocniczych. Do stylizacji wybrano Tailwind CSS v4 ze względu na jego zero-runtime approach, co gwarantuje, że style są generowane na etapie budowania i nie obciążają przeglądarki podczas pomiarów wydajnościowych. System budowania oparto na Vite jako najnowocześniejszym narzędziu zapewniającym natychmiastowe odświeżanie modułów (*HMR*) i wydajną minifikację produkcyjną (*tree shaking*). Dzięki takiemu podejściu, wszelkie zaobserwowane różnice w wydajności można przypisać bezpośrednio mechanizmom wewnętrznym frameworków (Virtual DOM vs Signals), co stanowi istotę naukową niniejszej pracy.

Ważnym aspektem implementacyjnym było również zapewnienie izolacji procesów obliczeniowych. Logika algorytmów została zaimplementowana w sposób całkowicie bezstanowy (*stateless*), co pozwala na ich uruchamianie w różnych środowiskach bez konieczności modyfikacji kodu źródłowego. Wykorzystanie wzorca *Strategy* umożliwiło dynamiczne przełączanie algorytmów w trakcie działania aplikacji, co jest kluczowe dla scenariuszy porównawczych, gdzie użytkownik chce natychmiast zestawić wyniki sortowania bąbelkowego z szybkim sortowaniem na tym samym zbiorze danych. Ponadto, wdrożono zaawansowany system logowania zdarzeń (*Telemetry*), który w tle zbiera informacje o liczbie operacji porównań i zamian, co służy do późniejszej walidacji poprawności implementacji w obu frameworkach. Tego typu rygor architektoniczny jest niezbędny, aby uniknąć sytuacji, w której jeden z frameworków jest faworyzowany przez bardziej zoptymalizowaną logikę biznesową.

2.2 Opis implementacji aplikacji wizualizującej w React

Aplikacja w wersji React została zaprojektowana i zrealizowana z wykorzystaniem najnowszych standardów biblioteki w wersji 19 oraz nowoczesnych narzędzi ekosystemu JavaScript, co pozwoliło na stworzenie wydajnego i elastycznego środowiska badawczego. Głównym celem implementacji było wykorzystanie modelu programowania deklaratywnego do opisania skomplikowanych i szybko zmieniających się stanów wizualnych.

2.2.1 Architektura projektu i stos technologiczny

Struktura projektu opiera się na nowoczesnym podejściu do budowania aplikacji typu Single Page Application (SPA) [31], gdzie kluczową rolę odgrywa wyraźna separacja logiki biznesowej od warstwy prezentacji. Wykorzystano bibliotekę React Router v7 do obsługi nawigacji, co umożliwiło płynne przejścia między stroną powitalną (*Landing Page*), wprowadzającą w tematykę badań, a głównym panelem badawczym (*Dashboard*), w którym odbywa się właściwa wizualizacja. Warstwa wizualna została zbudowana w oparciu o modularną bibliotekę komponentów, co zapewnia spójność interfejsu i ułatwia jego modyfikację. Centralnym punktem aplikacji jest komponent `SortingProgressChart`, który pełni funkcję dynamicznego płótna dla algorytmów.

Wybór narzędzi deweloperskich był podyktowany chęcią uzyskania jak najwyższej wydajności nie tylko w fazie uruchomieniowej, ale i deweloperskiej. Jako system budowania wykorzystano Vite [52], który dzięki zastosowaniu natywnych modułów ES [14] w trybie deweloperskim oraz silnika Rollup w trybie budowania produkcyjnego, zapewnia niemal natychystowe odświeżanie zmian. Stylizacja opiera się na najnowszej wersji Tailwind CSS v4, która wprowadza nowy silnik generowania stylów i pozwala na implementację zaawansowanych efektów wizualnych, takich jak *glassmorphism* (efekt oszronionego szkła) [48]. Aby przyspieszyć proces tworzenia interfejsu bez utraty kontroli nad jakością kodu, włączono do projektu zestaw komponentów Shadcn/ui [44]. Biblioteka ta, w przeciwieństwie do tradycyjnych pakietów NPM, dostarcza kod źródłowy komponentów bezpośrednio do projektu, co pozwala na ich pełną personalizację pod kątem specyficznych wymagań wydajnościowych wizualizatora.

Dodatkowym atutem wybranego stosu jest pełna integracja ze środowiswem TypeScript [37]. Każdy komponent posiada ściśle zdefiniowany interfejs (*props*), co eliminuje błędy związane z przekazywaniem nieprawidłowych danych w trakcie intensywnej pracy algorytmu [23]. Wykorzystanie *Utility Types* pozwoliło na tworzenie elastycznych wariantów ułożenia słupków wykresu w zależności od rozdzielczości ekranu.

2.2.2 Podejście do zarządzania stanem i logika animacji

Zarządzanie stanem wizualizacji zrealizowano za pomocą autorskiego hooka `useSorting`, co wpisuje się w nowoczesne wzorce projektowe biblioteki React, promujące enkapsulację logiki i współdzielenie zachowań między komponentami [24]. Implementacja opiera się na architekturze opartej na migawkach (*Snapshot-based approach*). Zamiast modyfikować tablicę danych w czasie rzeczywistym podczas trwania animacji, algorytmy najpierw generują kompletną listę kroków (`SortStep`). Każdy taki krok definiuje unikalny stan aplikacji w danym momencie.

```
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` pełni funkcję kontrolera, który zarządza indeksem aktywnego kroku oraz precyzyjnym czasem wyświetlania kolejnych klatek animacji. Wykorzystanie `window.setInterval` w połączeniu z referencjami (`useRef`) do przechowywania aktualnego stanu animacji pozwala na odizolowanie tempa animacji od cyklu renderowania samego frameworka. React otrzymuje polecenie aktualizacji stanu co określony interwał, co wymusza przejście przez proces re-renderowania, jednak sam licznik i kolejka kroków pozostają poza mechanizmem *useState*, co drastycznie oszczędza zasoby procesora przy bardzo małych opóźnieniach.

Jednym z najciekawszych wyzwań inżynieryjnych było zaprojektowanie systemu ”pauzowania”logiki JavaScript w sposób, który nie blokuje głównej nitki renderowania przeglądarki. W React 19 wykorzystano do tego celu połączenie generatorów TypeScript oraz asynchronicznych funkcji sterujących. Pozwala to na uniknięcie przepełnienia stosu wywołań (*stack overflow*) przy głębokiej rekurencji algorytmów takich jak Quick Sort, co jest częstym problemem w naiwnych implementacjach vizualizatorów. Implementacja ta wymagała stworzenia zaawansowanego wzorca *Iterator-Observer*, gdzie algorytm dostarcza kolejne stany danych, a framework reaguje na nie poprzez synchronizację widoku.

W procesie implementacji szczególną uwagę poświęcono obsłudze asynchroniczności. Ponieważ JavaScript jest językiem jednowątkowym [30], każda długa operacja blokuje inter-

fejs (*Main Thread Blocking*), co wynika bezpośrednio z natury mechanizmu Event Loop [32]. Aby temu zapobiec, zdecydowano się na wykorzystanie `AbortController`, co pozwala na natychmiastowe przerwanie pracy algorytmu w dowolnym momencie (np. gdy użytkownik kliknie przycisk ”Stop”). W React 19 zintegrowano to z hookiem `useEffect`, co gwarantuje poprawne ”sprzątanie”wątków i zapobiega wyciekom pamięci wynikającym z osieroconych procesów animacji. Taki poziom kontroli nad cyklem życia procesu jest kluczowy dla rzetelności pomiarów *Total Blocking Time*.

2.2.3 Specyficzne techniki optymalizacyjne w React 19

Aby zapewnić płynność wizualizacji przy setkach aktualizacji na sekundę, wdrożono zaawansowane techniki optymalizacji renderowania, które są specyficzne dla modelu pracy Reacta i jego architektury Fiber [4]. Jednym z kluczowych mechanizmów jest wykorzystanie wyższego rzędu komponentu `React.memo` [36]. Zastosowano go do opakowania komponentu `SortingProgressChart`, co wymusza na Reactcie płytkie porównywanie właściwości (*props*) przed podjęciem decyzji o ponownym renderowaniu.

Kolejną istotną optymalizacją jest wykorzystanie hooka `useMemo` do przygotowania zbiorów danych przed samym procesem rysowania słupków. Indeksy elementów podlegających porównaniu lub zamianie są w każdym kroku konwertowane na obiekty typu `Set` [36]. Wykorzystanie struktur danych typu *Set* zamiast tradycyjnych tablic pozwala na uzyskanie złożoności obliczeniowej $O(1)$ przy sprawdzaniu stanu każdego z setek słupków podczas renderowania pętli wykresu.

Dodatkowo, wykorzystano nową funkcjonalność React 19 — *Transitions API*. Pozwala ono na oznaczenie pewnych aktualizacji stanu jako niskopriorytetowych. W wizualizatorze, podczas gdy zmiana danych słupków jest priorytetowa, aktualizacja statystyk (np. licznik porównań w rogu ekranu) może odbywać się w tle, co zapobiega blokowaniu głównej nitki renderowania przy najbardziej intensywnych momentach sortowania.

Warto również zwrócić uwagę na sposób zarządzania listą kluczy (`key`) w pętlach renderujących. W React 19 poprawiono algorytm *diffingu* w przypadku list. W wizualizatorze, każdy słupek posiada stały unikalny identyfikator, co jest kluczowe dla mechanizmu *Persistence of DOM elements*. Dzięki temu, podczas operacji zamiany dwóch elementów (`swap`), React nie usuwa i nie tworzy nowych węzłów DOM, lecz jedynie modyfikuje ich parametry wizualne lub używa *CSS Transforms* do przemieszczenia istniejących węzłów. Takie podejście drastycznie redukuje presję na Garbage Collector, co jest widoczne w stabilniejszym przebiegu wykresu użycia pamięci sterty (Heap Memory) w narzędziach Chrome DevTools.

Całkowita separacja fazy obliczeń od fazy prezentacji minimalizuje ryzyko zjawiska *jank*, co jest szczególnie ważne w środowisku takim jak React, gdzie proces odświeżania Virtual DOM generuje stały narzut na CPU. W badaniu monitorowano również parametr *Recalculate Style*, który w wersji React 19 wykazuje mniejszą zmienność dzięki optymalizacjom w silniku Fiber, co bezpośrednio przekłada się na płynność animacji przy wysokich częstoltiwościach odświeżania.

Warto również podkreślić specyficzną dla Reacta charakterystykę zarządzania pamięcią operacyjną. Regularne re-renderowanie komponentów wiąże się z tworzeniem dużej liczby krótkotrwałych obiektów w pamięci (*Garbage Generation*) [38]. Mimo że silnik V8 w przeglądarce Chrome jest niezwykle efektywny w sprzątaniu tzw. ”młodej generacji”obiektów, ciągła alokacja nowych struktur Virtual DOM podczas wizualizacji (np. przy sortowaniu 1000 elementów z opóźnieniem 1ms) generuje zauważalny narzut. W badaniu zaobserwowano, że wykres użycia pamięci w React przypomina klasyczne ”zęby piły”— pamięć stale rośnie do pewnego progu, po czym następuje gwałtowny spadek wywołany akcją Garbage Collector. Choć React 19 wprowadza optymalizacje w recyklingu węzłów Fiber, narzut ten jest nieunikniony przy architekturze opartej na niemutowalnym stanie. Z perspektywy sprzętowej, rygorystyczne przestrzeganie zasady *Immutability* wymusza częste kopiowanie dużych tablic danych, co przy bardzo szybkich operacjach może prowadzić do zjawiska *Cache Miss* w procesorze Apple M4 Pro, zmuszając system do częstszego odwoływania się do głównej pamięci RAM.

2.3 Opis implementacji aplikacji wizualizującej w Angular

Wersja aplikacji przygotowana w frameworku Angular została zaimplementowana z wykorzystaniem najnowszego stabilnego wydania (wersja 19). Pozwoliło to na pełne wykorzystanie rewolucyjnych zmian, jakie zaszły w ekosystemie Angulara w ostatnich miesiącach, szczególnie w obszarze reaktywności oraz drastycznego uproszczenia architektury komponentów, co stawia ten framework w nowym świetle w kontekście wydajności i czytelności kodu [17].

2.3.1 Nowoczesna architektura i stos technologiczny

Architektura aplikacji Angular została zaprojektowana zgodnie z paradygmatem *Standalone first*, co stanowi odejście od historycznie złożonych struktur opartych na modułach. Takie podejście eliminuje konieczność definiowania i zarządzania wieloma plikami konfiguracji `NgModule`, co nie tylko drastycznie redukuje ilość kodu nadmiarowego (*boilerplate*), ale także pozwala kompilatorowi Angulara Ivy [25] na aplikowanie znacznie agre-

sywniejszych technik optymalizacji drzewa zależności (*tree shaking*) [27]. Dzięki temu końcowy pakiet aplikacji zawiera wyłącznie kod faktycznie wykorzystywany, co jest kluczowe dla szybkości inicjalizacji środowiska badawczego i minimalizacji narzutu związanego z parsowaniem JavaScriptu przez przeglądarkę.

Logika aplikacji została zorganizowana w modularne warstwy o jasno zdefiniowanych odpowiedzialnościach. Centralnym elementem są serwisy rdzeniowe (*Core Services*), które dzięki systemowi wstrzykiwania zależności (*Dependency Injection*) pełnią rolę niezależnych silników obliczeniowych [55]. To właśnie w nich zamknięto logikę algorytmów sortowania, izolując ją od cyklu życia komponentów wizualnych. Warstwa prezentacji skupia się wokół komponentu *Visualizer Dashboard*, który integruje dynamiczne parametry wejściowe (takie jak rozmiar tablicy czy wybór algorytmu) z silnikiem renderującym.

Stos technologiczny Angulara został celowo dobrany tak, aby stanowił bezpośredni punkt odniesienia dla wersji React. Wykorzystanie oficjalnego narzędzia Angular CLI zapewnia stabilny i powtarzalny proces budowania, wykorzystujący pod maską silnik Esbuild [15]. Stylizacja interfejsu opiera się na Tailwind CSS v4, co gwarantuje, że różnice w wydajności nie będą wynikać z odmiennych arkuszy stylów. Jako bibliotekę wizualizacji wybrano Ngx-Charts [10], która została zintegrowana w sposób zapewniający płynną współpracę z nowym mechanizmem detekcji zmian. Aby zapewnić profesjonalny wygląd i wysoką dostępność interfejsu użytkownika, wdrożono zestaw komponentów PrimeNG [41].

2.3.2 Integracja z PrimeNG i Optymalizacja PrimeFlex

Wybór PrimeNG jako biblioteki komponentów UI był podyktowany jej natywnym wsparciem dla Sygnałów w wersji 19. Dzięki temu komponenty takie jak suwaki prędkości (*Sliders*), przyciski sterujące czy menu wyboru algorytmu, reagują na zmiany stanu bez żadnego narzutu obliczeniowego. Wykorzystano również system PrimeFlex do budowy responsywnego układu Dashboardu.

Ważnym aspektem wydajnościowym było uniknięcie ”ciężkich”animacji CSS dostarczanych domyślnie przez biblioteki UI. W tym celu większość komponentów PrimeNG została skonfigurowana w trybie *headless* (tam gdzie było to możliwe) lub z nadpisanymi stylami, aby zminimalizować liczbę operacji *Paint* i *Composite* wykonywanych przez przeglądarkę podczas trwania wizualizacji. Pozwoliło to na zachowanie estetyki nowoczesnej aplikacji typu ”Enterprise”bez poświęcania cennych milisekund z budżetu klatki animacji.

2.3.3 Reaktywność drobnoziarnista i mechanizm Sygnałów

Najważniejszym elementem implementacji, stanowiącym o technologicznym skoku wersji 19, jest pełne przejście na mechanizm Angular Signals jako fundament zarządzania stanem [9]. Sygnały (*WritableSignal*) wprowadzają do Angulara model reaktywności drobnoziarnistej (*Fine-grained Reactivity*) [7], który diametralnie zmienia sposób, w jaki framework reaguje na zmiany danych. Zamiast polegać na asynchronicznych strumieniach RxJS [40], które wymagają ręcznego zarządzania subskrypcjami, sygnały oferują deklaratywny i bezpieczny sposób definiowania przepływu informacji.

W przeciwieństwie do Reacta, gdzie zmiana stanu wymusza ponowne wykonanie funkcji komponentu w celu wygenerowania nowego drzewa Virtual DOM, Sygnały w Angularze pozwalają frameworkowi na bezpośrednią identyfikację konkretnego miejsca w szablonie, które wymaga aktualizacji. Jest to proces znacznie bardziej precyzyjny — framework "wie", który konkretny słupek na wykresie zmienił kolor, i modyfikuje tylko ten jeden element DOM, bez dotykania pozostałych.

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

// Inicjalizacja stanu w komponencie Standalone
export class SortVisualizer {
  readonly sortingState = signal<SortingState>({
    values: [],
    comparingIndices: [],
    swappingIndices: [],
    pivotIndex: null,
    sortedIndices: []
  });

  // Inteligentne obliczanie wartosci pochodnych
  readonly currentValues = computed(() => this.sortingState().
   values);
}
```

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

Źródło: Opracowanie własne

Aktualizacja widoku w tym modelu odbywa się poprzez metodę `update()`, która modyfikuje wartość sygnału. Wyzwala to asynchroniczny, lecz niezwykle precyzyjny proces detekcji zmian, który omija zbędne porównywanie całych gałęzi drzewa komponentów. Dodatkowo, wykorzystanie `computed()` pozwala na tworzenie reaktywnych wartości pochodnych, które są przeliczane leniwie (*lazy evaluation*) – tylko wtedy, gdy ich wynik jest faktycznie potrzebny w widoku, co oszczędza cenne cykle procesora podczas szybkich animacji.

2.3.4 Zoneless Angular i wydajność animacji

W wersji 19 wprowadzono również możliwość pracy w trybie *Zoneless*, co zostało w pełni wykorzystane w projekcie. Tradycyjnie Angular polegał na bibliotece Zone.js, która przechwytywała wszystkie zdarzenia asynchroniczne w celu wyzwolenia detekcji zmian. W wizualizatorze, gdzie generujemy setki zdarzeń na sekundę, narzut Zone.js byłby zauważalny. Rezygnacja z tej biblioteki na rzecz czystych Sygnałów pozwoliła na wyeliminowanie zbędnych cykli sprawdzania stanu, co skutkuje niższym użyciem pamięci i bardziej stabilnym czasem klatek (*frame timing*). Dzięki temu wizualizacja w Angularze może osiągać stabilne 60 FPS nawet przy ekstremalnie dużych zbiorach danych i minimalnych opóźnieniach.

2.3.5 Efektywność pętli renderujących i dyrektywa @for

Innym przełomowym elementem wpływającym na wydajność w wersji 19 jest nowa składnia szablonów i wprowadzenie natywnej kontroli przepływu w formie instrukcji block-based (`@for`, `@if`). W przeciwieństwie do starej dyrektywy `*ngFor`, nowa składnia `@for` jest znacznie lepiej zoptymalizowana pod kątem wydajności *Runtime*.

W wizualizatorze algorytmów, gdzie mamy do czynienia z iteracją po setkach elementów wykresu, `@for` zapewnia niemal natychmiastowe aktualizacje pozycji. Nowy mechanizm śledzenia zmian (*tracking mechanism*) wymaga podania unikalnego klucza w parametrze `track`, co pozwala Angularowi na precyzyjne operacje na liście bez konieczności re-renderowania całego zbioru danych. Zjawisko to, w połączeniu z Sygnałami, tworzy synergię, która niemal całkowicie eliminuje operacje na DOM, które nie są bezpośrednio związane z aktualizacją konkretnego, zmienionego elementu. W testach porównawczych pozwala to Angularowi na zachowanie płynności interfejsu nawet przy opóźnie-

niach rzędu 1ms, gdzie React zaczyna wykazywać oznaki ”zadyszki”ze względu na narzut Virtual DOM.

Implementacja w wersji 19 wykazuje również zgoła odmienną charakterystykę pamięciową w porównaniu do modelu opartego na Virtual DOM. Ponieważ Sygnały są stabilnymi referencjami do wartości, a detekcja zmian w trybie *Zoneless* nie wymusza ponownego wykonania całego mechanizmu renderującego, liczba alokowanych obiektów na sekundę jest drastycznie niższa. Wykres zajętości sterty w Angularze jest znacznie bardziej płaski, co sugeruje lepszą przydatność tego frameworka do aplikacji typu „Real-time Dashboard”, gdzie stabilność i przewidywalność są priorytetem. Brak biblioteki Zone.js dodatkowo odciąża pamięć, eliminując konieczność utrzymywania skomplikowanej mapy stref asynchronicznych. Dodatkowo, podejście Angulara oparte na kompilacji do bezpośrednich instrukcji aktualizacji DOM (Ivy) pozwala na lepsze wykorzystanie predykcji skoków w procesorze M4 Pro, co przekłada się na mniejsze zużycie energii i niższą temperaturę pracy urządzenia podczas długotrwałych benchmarków na dużych zbiorach danych.

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. Proces ten jest szczególnie wymagający, ponieważ wymaga od silnika przeglądarki stałego balansowania między wykonywaniem skomplikowanej logiki biznesowej (algorytmy sortowania) a odświeżaniem komponentów graficznych w tempie zbliżonym do 60 klatek na sekundę.

2.4.1 Środowisko testowe i konfiguracja sprzętowa

Wszystkie pomiary wydajności zostały przeprowadzone w ściśle kontrolowanym środowisku sprzętowo-programowym, co pozwoliło na wyeliminowanie wpływu zmiennych zewnętrznych na ostateczne wyniki. Jako platformę sprzętową wykorzystano laptop wyposażony w procesor Apple M4 Pro (architektura ARMv9.4-A z 12 rdzeniami: 8 wysokowydajnych P-cores oraz 4 energooszczędne E-cores) oraz 24 GB zunifikowanej pamięci RAM LPDDR5x. Wybór tej platformy miał kluczowe znaczenie ze względu na wysoką przepustowość pamięci (do 273 GB/s), co minimalizuje opóźnienia w wymianie danych między procesorem a układem graficznym podczas renderowania struktur SVG. Warstwę systemową stanowił macOS Tahoe 26.2, skonfigurowany w trybie wysokiej wydajności,

z wyłączonymi funkcjami oszczędzania energii oraz zawieszonymi procesami indeksowania plików (Spotlight) i automatycznych aktualizacji systemowych.

Pomiary realizowano w najnowszej wersji przeglądarki Google Chrome (wersja 143), która została uruchomiona w dedykowanym profilu programistycznym w trybie incognito. Takie podejście gwarantuje pełną izolację środowiska testowego — żadne rozszerzenia przeglądarkowe (takie jak AdBlock czy React DevTools), procesy synchronizacji konta Google czy pliki cache nie zakłócają pracy głównego wątku JavaScript (*Main Thread*). Ponadto, proces przeglądarki został przypisany do rdzeni typu P-cores, aby uniknąć zjawiska *context switching* pomiędzy różnymi typami rdzeni, co mogłoby wprowadzić błędy pomiarowe rzędu kilku procent.

Kluczowym elementem metodologii było testowanie wyłącznie wersji produkcyjnych aplikacji. Obie aplikacje (React i Angular) zostały skompilowane z optymalizacjami typu *Ahead-of-Time* (AOT) oraz zaawansowaną minifikacją kodu (Terser). W celu wyeliminowania wpływu latencji sieciowej oraz fluktuacji protokołu HTTP, aplikacje były serwowane lokalnie za pomocą wydajnego serwera statycznego (`serve`), opartego na protokole HTTP/2, co zapewniło stabilność transferu danych do silnika przeglądarki.

2.4.2 Narzędzia diagnostyczne i metryki wydajnościowe

Do zbierania danych diagnostycznych wykorzystano zaawansowaną synergię narzędzia **Chrome DevTools Performance** oraz programistycznego interfejsu **PerformanceObserver** wbudowanego w silnik V8 [18]. Taka kombinacja umożliwiła precyzyjne śledzenie cyklu życia każdej klatki obrazu, oferując wgląd zarówno w wysokopoziomowe metryki użytkowe [19], jak i niskopoziomowe zdarzenia systemowe silnika JavaScript [16]. Wykorzystanie `PerformanceObserver` pozwoliło na programistyczne przechwytywanie zdarzeń typu *long-task*, co umożliwiło automatyczną dokumentację momentów, w których frameworki przestawały być interaktywne.

W procesie analizy skoncentrowano się na pięciu kluczowych metrykach, które najlepiej oddają charakterystykę pracy współczesnych frameworków frontendowych pod dużym obciążeniem dynamicznym [19, 16]:

1. **Scripting Time**: Sumaryczny czas procesora poświęcony na wykonanie skryptów JavaScript. Obejmuje on logikę algorytmów, mechanizmy wewnętrzne frameworków (reconciler w React, signal graph w Angularze) oraz procesy pomocnicze, takie jak *Garbage Collection* [18]. Jest to kluczowy wskaźnik efektywności samego modelu programistycznego frameworka.

2. **Rendering & Painting Time**: Czas potrzebny silnikowi przeglądarki na obliczenie geometrii elementów (*Layout*), wyliczenie stylów CSS (*Recalculate Styles*) oraz fizyczne narysowanie pikseli na ekranie (*Paint*) [26]. Ta metryka pozwala ocenić, jak bardzo dany framework obciąża drzewo DOM i jak efektywnie komunikuje się z warstwą graficzną przeglądarki.

3. **System FPS (Frames Per Second)**: Stałość i poziom klatkażu animacji. W badaniu przyjęto, że każda klatka trwająca powyżej 16.67 ms (odpowiednik 60 Hz) stanowi błąd płynności (*frame drop*) [38]. Analiza rozkładu FPS pozwala na wykrycie zjawiska *micro-stuttering*, które jest niewidoczne przy prostym uśrednianiu wyników.

4. **Total Blocking Time (TBT)**: Suma okresów, w których główny wątek był zablokowany przez zadania trwające powyżej 50 ms [19]. Metryka ta bezpośrednio przekłada się na doznania użytkownika (*User Experience*) i responsywność interfejsu na próby interakcji w trakcie trwania wizualizacji.

5. **Heap Memory Usage**: Dynamika zajętości pamięci operacyjnej przez stertę JavaScript [18]. Monitorowanie tego parametru pozwala na ocenę stabilności aplikacji w długim terminie oraz efektywności systemu zarządzania pamięcią silnika V8 w kontekście milionów krótkożyjących obiektów tworzonych podczas sortowania.

2.4.3 Procedura badawcza i standaryzacja testów

Badanie zostało podzielone na konkretne scenariusze testowe, które różniły się między sobą poziomem złożoności obliczeniowej oraz gęstością aktualizacji interfejsu. Każdy algorytm (Bubble Sort, Quick Sort, Merge Sort, Insertion Sort) był analizowany przy trzech różnych wielkościach zbiorów danych: 100, 500 oraz 1000 elementów. Taki dobór parametrów pozwolił na zaobserwowanie, jak narzut frameworka skaluje się wraz ze wzrostem liczby operacji na sekundę. Aby wyniki były w pełni porównywalne, szybkość animacji (interwał kroków) została zestandaryzowana na poziomie 1 ms dla wszystkich testów, co miało na celu wymuszenie na przeglądarce pracy na granicy wydajności.

Procedura pomiarowa dla każdego ze scenariuszy była rygorystycznie przestrzegana i składała się z następujących etapów [16]:

1. **Faza przygotowawcza (Warm-up)**: Przed startem właściwego pomiaru, dany algorytm był uruchamiany trzykrotnie na małym zbiorze danych. Miało to na celu wymuszenie na silniku V8 kompilacji JIT (*Just-In-Time*) najczęściej używanych ścieżek kodu [18], tak aby właściwy pomiar dotyczył zoptymalizowanej wersji binarnej, a nie interpretowanego kodu JavaScript.

2. **Inicjalizacja środowiska**: Wymuszenie operacji *Garbage Collection* poprzez interfejs programistyczny DevTools w celu oczyszczenia sterty [16]. Następnie następowało 5-sekundowe oczekiwanie na ustabilizowanie się obciążenia procesora do poziomu spoczynkowego.

3. **Rejestracja właściwa**: Uruchomienie profilera nagrywającego ścieżkę wydajności jednocześnie z wyzwoleniem algorytmu sortowania [16]. Rejestracja trwała dokładnie do momentu osiągnięcia stanu końcowego (posortowana tablica).

4. **Eksport i czyszczenie**: Zapisanie surowych danych profilowania do formatu JSON oraz wykonanie zrzutu pamięci sterty [16].

Wszystkie wyniki prezentowane w dalszej części pracy stanowią średnią arytmetyczną z 10 niezależnych pomiarów dla każdego scenariusza. W celu zapewnienia wysokiej wiarygodności statystycznej, dla każdej średniej obliczono również odchylenie standardowe oraz błąd standardowy. Wyniki, w których odchylenie standardowe przekraczało 15

2.4.4 Wyzwania w standaryzacji i neutralizacja zmiennych

Projektowanie rzetelnego badania dla dwóch tak odmiennych modeli renderowania wiąże się z wyzwaniem tzw. niedopasowania impedancyjnego (*Impedance Mismatch*). Aby zapewnić sprawiedliwość benchmarku, konieczne było zagwarantowanie pełnej parzystości wizualnej — każdy słupek w obu wersjach aplikacji jest renderowany za pomocą identycznej technologii SVG. Standaryzacja polegała na wymuszeniu, aby oba frameworki przetwarzały dokładnie ten sam strumień danych o krokach algorytmu, co pozwoliło na ocenę wewnętrznej sprawności mechanizmów aktualizacji, a nie wydajności samej biblioteki graficznej.

Dodatkowo, podjęto kroki w celu neutralizacji mechanizmów specyficznych dla technologii, które mogłyby zaburzać wyniki. W React 19 kontrolowano mechanizmy asynchronicznego grupowania aktualizacji (*batching*), aby nie poprawiały one sztucznie płynności kosztem precyzji pomiaru czasu rzeczywistego. W Angularze kluczowe było poprawne skonfigurowanie trybu *Zoneless*, aby upewnić się, że zysk wydajności wynika z architektury sygnałowej, a nie z ograniczenia funkcjonalności. Wreszcie, zastosowano asynchroniczny eksport wyników (*background export*) po zakończeniu fazy aktywnej wizualizacji, co pozwoliło odizolować czas czystej pracy frameworka od czasu zapisu danych do pamięci trwałej.

2.5 Metody oceny produktywności programistycznej

Poza analizą czysto wydajnościową, która skupia się na parametrach technicznych działania aplikacji, niniejsza praca podejmuje próbę kompleksowej oceny obu technologii pod kątem ergonomii pracy i produktywności programisty. Jest to aspekt o znaczeniu strategicznym, często decydujący o długofalowym sukcesie projektu IT oraz kosztach jego utrzymania (*Total Cost of Ownership*). Ocena produktywności jest procesem wielowymiarowym, łączącym twarde dane metryczne z subiektywnym odczuciem tzw. *Developer Experience* (DX).

2.5.1 Kryteria oceny ilościowej i analiza artefaktów

W ramach oceny obiektywnej (ilościowej) przeanalizowano szereg mierzalnych parametrów, które bezpośrednio przekładają się na szybkość dostarczania funkcji oraz łatwość refaktoryzacji kodu [42, 29]. Kluczowym wskaźnikiem jest złożoność kodu źródłowego, mierzona nie tylko przez *Source Lines of Code* (SLOC), ale przede wszystkim poprzez analizę złożoności cyklomatycznej (*Cyclomatic Complexity*) oraz złożoności kognitywnej (*Cognitive Complexity*) [46]. Wykorzystanie narzędzi takich jak SonarQube pozwoliło na identyfikację fragmentów kodu, które mimo małej liczby linii, mogą być trudne w zrozumieniu i testowaniu ze względu na gęstą logikę warunkową.

Kolejnym badanym parametrem jest charakterystyka pakietu wynikowego (*Bundle Analysis*). Analiza ta została pogłębiona o podział na kod aplikacji (*first-party code*) oraz kod bibliotek zewnętrznych (*third-party code*). Pozwala to ocenić, jaki procent końcowej wagi aplikacji wynika z narzutu samego frameworka, a jaki z implementacji logiki biznesowej. Weryfikacji poddano również czas budowania aplikacji (*Build Time*) w dwóch trybach: ”zimnym”(bez cache) oraz przyrostowym, co jest kluczowym parametrem wpływającym na płynność pracy dewelopera w codziennych zadaniach. Dodatkowo, przeanalizowano liczbę oraz wagę zależności zewnętrznych, co pozwala ocenić stopień izolacji projektu i potencjalne ryzyka związane z łańcuchem dostaw oprogramowania (*Software Supply Chain Security*).

2.5.2 Kryteria oceny jakościowej i doświadczenie dewelopera

Ocena jakościowa została oparta na ustrukturyzowanych doświadczeniach zebranych podczas pełnego cyklu życia projektu — od inicjalizacji, przez implementację algorytmów,

aż po debugowanie i optymalizację finalną [5]. Skupiono się na trzech fundamentalnych obszarach [42, 5]:

1. **Ekosystem i standardy**: Analiza różnicy między ”wymonitowanym”podejściem Angulara (gdzie framework dostarcza oficjalne rozwiązania dla rutingu, formularzy i walidacji [17]) a zdecentralizowanym ekosystemem Reacta, który oferuje większą swobodę wyboru bibliotek, ale nakłada na dewelopera ciężar podejmowania decyzji architektonicznych i dbania o ich kompatybilność [3].

2. **Bariera wejścia i krzywa uczenia**: Porównanie wysiłku niezbędnego do opanowania zaawansowanych konceptów, takich jak *Dependency Injection* i *Observables* w Angularze [55], w zestawieniu z modelem *Hooks* i rygorem niezmienności (*Immutability*) w React [47]. Ważnym aspektem była ocena, jak szybko nowy programista jest w stanie poprawnie zaimplementować wydajny komponent wizualizujący w każdej z technologii.

3. **Głębokość integracji z TypeScript**: Przeanalizowano, jak silnie frameworki wymuszają poprawne typowanie [23]. Angular, będący od podstaw projektowany z myślą o TypeScript, oferuje silniejsze typowanie szablonów HTML, podczas gdy React bazuje na elastyczności JSX [39], co daje inne doświadczenie w zakresie statycznej analizy kodu i wykrywania błędów na etapie pisania tekstu.

Ostatnim badanym kryterium była jakość narzędzi diagnostycznych dedykowanych dla deweloperów. Porównano funkcjonalność rozszerzeń takich jak *Angular DevTools* i *React Developer Tools* w kontekście inspekcji stanu animacji w czasie rzeczywistym oraz profilowania zmian widoku. Synteza tych danych pozwoli na sformułowanie wniosków dotyczących tego, który framework lepiej wspiera produktywność v projektach o wysokiej dynamice zmian.

Warto również zauważyć, że ocena produktywności uwzględniała paradygmat „wolności vs standardu”. W procesie tworzenia wizualizatora w React, deweloper staje przed koniecznością samodzielnego doboru bibliotek do zarządzania stanem asynchronicznym czy obsługi routingu, co przy braku odpowiedniego doświadczenia może prowadzić do powstania długu technicznego. Angular, poprzez swoje podejście „battery-included”, narzuca odgórnie sprawdzone wzorce projektowe, co drastycznie skraca fazę planowania architektury, ale może być postrzegane jako ograniczające przy specyficznych, niskopoziomowych optymalizacjach. W badaniu szczególną uwagę poświęcono temu, jak te dwa podejścia wpływają na czas potrzebny na implementację nowej funkcji – np. dodanie nowego typu algorytmu wizualnego – oraz jak frameworki wspierają refaktoryzację kodu w miarę wzrostu jego złożoności.

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] Adam Wathan, Steve Schoger: *Refactoring UI*, Tailwind Labs, 2018.

[2] Addy Osmani: *Learning JavaScript Design Patterns*, 2nd, O'Reilly Media, 2023.

[3] Alex Banks, Eve Porcello: *Learning React: Modern Patterns in Confident Front-End Development*, 2nd, O'Reilly Media, 2020.

[4] Andrew Clark: *React Fiber Architecture*, Dostęp: 15.01.2026, URL: `https://github.com/acdlite/react-fiber-architecture`.

[5] Andrew Hunt, David Thomas: *The Pragmatic Programmer: Your Journey To Mastery*, 20th Anniversary Edition, Addison-Wesley Professional, 2019.

[6] Angular Team: *Angular Change Detection*, Dostęp: 25.01.2026, URL: `https://media.beehiiv.com/cdn-cgi/image/fit=scale-down,format=auto,onerror=redirect,quality=80/uploads/asset/file/051c845e-8e6c-4a15-b822-4a6ca3c41ffb/1_6e8V1ciJnfuYLBXywqer6Q.jpg`.

[7] Angular Team: *RFC: Angular Signals*, Dostęp: 14.01.2026, URL: `https://github.com/angular/angular/discussions/49685`.

[8] Anthony Accomazzo, Ari Lerner, David Guttman, Nate Murray: *Fullstack React: The Complete Guide to ReactJS and Friends*, Fullstack.io, 2020.

[9] Dan Vanderkam: *Effective TypeScript: 83 Specific Ways to Improve Your TypeScript*, 2nd, O'Reilly Media, 2024.

[10] Dariusz Bober: *Angular: Poznaj framework od Google*, Helion, 2023.

[11] David Flanagan: *JavaScript: The Definitive Guide: Master the World's Most Used Programming Language*, 7th, O'Reilly Media, 2020.

[12] Dmitri Pavlutin: *Mastering JavaScript Arrays*, Leanpub, 2020.

[13] Donald E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd, Addison-Wesley Professional, 1998.

[14] ECMA International: *ECMAScript 2015 Language Specification*, `https://262.ecma-international.org/6.0/`, Dostęp: 12.01.2026.

[15] Evan Wallace: *esbuild: An extremely fast JavaScript bundler*, `https://esbuild.github.io/`, Dostęp: 12.01.2026.

[16] Google: *Analyze runtime performance*, Dostęp: 22.01.2026, URL: https://developer.chrome.com/docs/devtools/performance.

[17] Google: *Angular Documentation*, https://angular.io/docs, Dostęp: 10.01.2026.

[18] Google: *V8 JavaScript Engine*, Dostęp: 10.01.2026, URL: https://v8.dev/.

[19] Google Developers: *Web Vitals: Essential metrics for a healthy site*, https://web.dev/vitals/, Dostęp: 15.01.2026.

[20] Ilya Grigorik: *Critical Rendering Path*, Dostęp: 20.01.2026, URL: https://web.dev/critical-rendering-path/.

[21] Jakob Nielsen: *Response Times: The 3 Important Limits*, 1993, URL: https://www.nngroup.com/articles/response-times-3-important-limits/.

[22] Jeremy Wilken: *Angular in Action*, Manning Publications, 2018.

[23] Josh Goldberg: *Learning TypeScript: Enhance Your Web Development Skills Using Type-Safe JavaScript*, O'Reilly Media, 2022.

[24] Kandit Techaichetas: *Learning React Design Patterns*, Packt Publishing, 2023.

[25] Kara Erickson: *Angular Ivy: A New Engine for a New Era*, https://blog.angular.io/angular-ivy-2cac0139cd95, Dostęp: 16.01.2026.

[26] Lin Clark: "Inside a super fast CSS engine: Quantum CSS", w: „Mozilla Hacks" (2017).

[27] Mamta Dalal: *Angular Design Patterns and Best Practices*, Packt Publishing, 2024.

[28] Marijn Haverbeke: *Eloquent JavaScript: A Modern Introduction to Programming*, 3rd, No Starch Press, 2018.

[29] Martin Fowler: *Refactoring: Improving the Design of Existing Code*, 2nd, Addison-Wesley Professional, 2018.

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

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

[32] MDN Web Docs: *The event loop*, Dostęp: 12.01.2026, URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop.

[33] MDN Web Docs: *Web Workers API*, Dostęp: 18.01.2026, URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[34] Meta Open Source: *Introducing Hooks*, Dostęp: 12.01.2026, URL: https://legacy.reactjs.org/docs/hooks-intro.html.

[35] Meta Open Source: *React Reference: Concurrent Rendering*, https://react.dev/reference/react/useTransition, Dostęp: 10.01.2026.

[36] Meta Open Source: *React: The library for web and native user interfaces*, https://react.dev/, Dostęp: 12.01.2026.

[37] Microsoft: *TypeScript: JavaScript with syntax for types*, https://www.typescriptlang.org/, Dostęp: 12.01.2026.

[38] Nicholas C. Zakas: *High Performance JavaScript*, O'Reilly Media, 2010.

[39] Nicolas Bevacqua: *Practical Modern JavaScript*, O'Reilly Media, 2017.

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

[41] PrimeTek: *PrimeNG: The Most Complete UI Component Library for Angular*, https://primeng.org/, Dostęp: 12.01.2026.

[42] Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

[43] Robert Sedgewick, Kevin Wayne: *Algorithms*, 4th, Addison-Wesley Professional, 2011.

[44] shadcn: *shadcn/ui: Beautifully designed components that you can copy and paste into your apps*, https://ui.shadcn.com/, Dostęp: 12.01.2026.

[45] Steve Fenton: *Pro TypeScript: Application-Scale JavaScript Development*, 2nd, Apress, 2017.

[46] Steve McConnell: *Code Complete: A Practical Handbook of Software Construction*, 2nd, Microsoft Press, 2004.

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

[48] Tailwind Labs: *Tailwind CSS: Rapidly build modern websites without ever leaving your HTML*, https://tailwindcss.com/, Dostęp: 12.01.2026.

[49] Thomas H. Cormen: *Algorithms Unlocked*, MIT Press, 2013.

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

[51] Unknown: *React Virtual DOM Update Process*, Dostęp: 25.01.2026, URL: https://miro.medium.com/v2/resize:fit:1400/format:webp/0*_C52yYMRTDuMtdBA.

[52] Vite Team: *Vite: Next Generation Frontend Tooling*, https://vitejs.dev/, Dostęp: 12.01.2026.

[53] Wojciech Maj: *React Lifecycle Methods Diagram*, Dostęp: 12.01.2026, URL: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/.

[54] Y. Chen, X. Su: “Research on Virtual DOM Efficiency in Modern Frameworks”, w: „IEEE Access” 8 (2020), s. 154210–154222.

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

SPIS RYSUNKÓW