

WYŻSZA SZKOŁA INFORMATYKI I ZARZĄDZANIA „COPERNICUS” WE WROCŁAWIU

WYDZIAŁ INFORMATYKI, ADMINISTRACJI I FIZJOTERAPII

Kierunek studiów: **Informatyka**

Poziom studiów: **Studia drugiego stopnia-inżynierskie**

Specjalność: **Inżynieria systemów informatycznych**

PRACA DYPLOMOWA INŻYNIERSKA

Daniel Słaby

Analiza porównawcza nowoczesnych rozwiązań technologicznych aplikacji SPA za pomocą jednakowej aplikacji zaimplementowanej w każdym z wybranych rozwiązań

**Comparative analysis of modern technological solutions of SPA
applications using the same application implemented in each of the
selected solutions**

Ocena pracy:
(ocena pracy dyplomowej, data, podpis promotora)

.....
(pieczętka uczelni)

Promotor:

Dr inż. Grzegorz Debita

WROCŁAW 2020

Spis treści

1 Wstęp	3
1.1 Wprowadzenie	3
1.2 Cel pracy	3
1.3 Motywacja	3
1.4 Zakres	4
 Część przeglądowa	 5
2 Część przeglądowa	6
2.1 Słowniczek Pojęć	6
2.2 Przegląd prac o podobnej tematyce	6
2.2.1 Comparison of Single-Page Application Frameworks	6
2.2.2 Lighthouse	9
2.2.3 Porównanie narzędzi do tworzenia aplikacji typu SPA na przykładzie Angular2 i React	10
 Część praktyczna	 12
3 Część praktyczna	13
3.1 Problem pomiaru wydajności aplikacji SPA	14
3.2 Specyfikacja wymagań	17
3.3 Architektura rozwiązania	17
3.3.1 Projekt testu wydajności	18
3.3.2 Budowa aplikacji	22
3.3.3 Przygotowanie aplikacji do badania	24
3.3.4 Konteneryzacja aplikacji	24
3.3.5 Moduł badania	27
3.3.6 Moduł analizy	34
3.4 Analiza wyników	38
3.4.1 Czas załadowania aplikacji	38
3.4.2 Dodanie 1000 wierszy na początku listy	41
3.4.3 Dodanie 1000 wierszy na końcu listy	42
3.4.4 Zamiana wszystkich wierszy na liście	43
3.4.5 Aktualizacja 500 wierszy	44

Spis treści

3.4.6	Zamiana miejscami dwóch wierszy	45
3.4.7	Usunięcie pojedynczego wiersza	46
3.4.8	Usunięcie wszystkich wierszy	47
4	Podsumowanie	48
	Bibliografia	49
	Spis rysunków	53
	Oświadczenie o udostępnianiu pracy dyplomowej	55
	Oświadczenie autorskie	56

1. Wstęp

1.1 Wprowadzenie

Technologia front-end ewoluje w zastraszającym tempie. 20 października 2010 rozpoczęła się nowa era technologii webowych wraz z wydaniem biblioteki AngularJS. Był to swoisty początek aplikacji SPA (Single Page Application) i całkowitej zmiany podejścia do tworzenia webowych aplikacji klienckich.

Dzisiaj większość narzędzi jakie używamy w życiu codziennym albo w pełni znajduje się w przeglądarce albo ma już na niej swoje odpowiedniki. Znaczna część aplikacji z których korzystamy na co dzień wykorzystuje te narzędzia jako rdzeń swojego działania. Ze znanych aplikacji z jakich korzysta na co dzień użytkownik możemy wymienić jako przykład odpowiednio: React: Facebook, Instagram, Netflix, WhatsApp Angular 2: GitHub Community Forum, Microsoft Office Home, Google Marketing Platform VueJS: FontAwesome, ChatWoot, Moonitor, Leave Dates

1.2 Cel pracy

W pracy tej, przeanalizowano trzy najpopularniejsze rozwiązania do tworzenia aplikacji SPA. Każde z nich pozwala na stworzenie takiej samej aplikacji z punktu widzenia końcowego użytkownika. Każda z nich jednak w inny sposób rozwiązuje problemy zarządzania zasobami oraz optymalizacji renderowania. W tym celu skonstruuje narzędzie pozwalające na badanie dynamicznych aplikacji webowych i postaram się odpowiedzieć na pytanie, czy różnica wydajności pomiędzy frameworkami jest tak duża, że powinna mieć wpływ na wybór konkretnego rozwiązania w ogólnym przypadku?

1.3 Motywacja

Za motywacją do stworzenia tejże pracy stały dwa główne powody. Pierwszym z nich jest chęć porównania trzech najczęściej wybieranych narzędzi w środowisku front-end. Każde z nich rozwiązuje ten sam problem, jednak nie jest do jasne, jak wygląda ich wydajność w przypadku użytkowania aplikacji. Jest to bowiem problem który trudno zbadać, co prowadzi nas do drugiego powodu. Badanie aplikacji typu SPA w momencie ich działania jest niezwykle trudnym zadaniem, dlatego też, celem praktycznym pracy jest stworzenie narzędzia do rozwiązania tego problemu.

1.4 Zakres

Zadanie będzie polegać na zaprojektowaniu odpowiedniej aplikacji która zawiera kilka znanych problemów wydajnościowych występujących w świecie aplikacji SPA. Następnie implementacja takiej aplikacji w każdym z wybranych rozwiązań. Kolejno na specjalnie stworzonej maszynie wirtualnej z ograniczonymi zasobami, wykonane zostaną testy wydajnościowe za pomocą wbudowanego profilera.

Mając już konkretne dane, wyniki zostaną przeanalizowane porównując które rozwiązanie jest najwydajniejsze w konkretnym przypadku co pozwoli na odpowiedź na pytanie: czy różnica wydajności pomiędzy narzędziami jest tak duża, że powinna mieć wpływ na wybór konkretnego rozwiązania w ogólnym przypadku?

Część przeglądowa

2. Część przeglądowa

- JSON [1] (JavaScript Object Notation) - jest to tekstowy format wymiany danych, bazującym na podzbiorze języka JavaScript. JSON przechowuje dane określone przez standard ECMA-262 3-cia edycja - Grudzień 1999.
- Transpilacja [2] - Transpilatory lub kompilatory typu źródło-źródło to narzędzia, które odczytują kod źródłowy napisany w jednym języku programowania i wytwarzają równoważny kod w innym języku. Języki, które piszesz, które są transpilowane na JavaScript, są często nazywane językami komplikacji do JS i mówi się, że są ukierunkowane na JavaScript.
- SPA [3] (także Single Page Application) - jednostronna aplikacja internetowa, czyli taka, która posiada tylko jeden plik HTML. Taka aplikacja nie przeładowuje strony w trakcie użytkowania. Może w tym celu korzystać z technologii AJAX lub innych dostępnych w przeglądarkach internetowych. Logika aplikacji SPA napisana jest w JavaScript lub w języku transpilowanym do języka JavaScript Framework.

2.1 Słowniczek Pojęć

2.2 Przegląd prac o podobnej tematyce

W części tej zajmiemy się przeglądem dostępnych rozwiązań oraz prac dotyczących pomiaru wydajności aplikacji internetowych z naciskiem na aplikacje typu SPA.

2.2.1 Comparison of Single-Page Application Frameworks

Pierwszą pozycją jest praca autorstwa Pan Eric'a Molina z instytutu KTH Royal Institute of Technology w Sztokholmie [4]. Autor stara się porównać frameworki na pełnej płazczyźnie ich złożoności. W jego pracy znaleźć możemy pełen spis kryteriów oceny frameworków wraz z wagami przypisanymi do cech określonymi za pomocą wywiadu środowiskowego jaki autor przeprowadził na grupie 10 profesjonalnych deweloperów mających doświadczenie w tych zagadnieniach przedstawionych na rysunku 2.1.

W pracy tej, porównywany frameworkami są AngularJS, Angular2 oraz React. I tutaj wyróżnić możemy już pierwsze różnice pomiędzy pracami. Praca Pana Molina została wykonana w 2016 roku. JavaScript jest jednym z najszybciej rozwijających się oraz najczęściej spotykanych technologii webowych na świecie [5]. Przez okres 4 lat, firmy masowo porzuciły technologie AngularJS na rzecz Angular 2 oraz Reacta. Dodatkowo, w międzyczasie do wyścigu dołączył Vue.js [6]. Przez ten okres, także implementacja tychże rozwiązań znacznie dojrzała. Angular z wersji 2 ewoluował już do wersji 9 [7]. Także React ewoluował z wersji

Criteria	Frequency
Cache performance	2
Compatibility	4
Developer guidelines	3
Documentation	6
Does “in-house” experience exists?	2
Simple to use	6
Maintainability	5
Maturity	4
Modularity	3
Performance	5
Popularity	8
Portability	4
Scalability	3
Security	2
Framework’s size	1
Testability	3
Who is the developer?	4

Rysunek 2.1. Grafika przedstawiająca ocenę wpływu wybranego elementu na atrakcyjność danego narzędzia [4]

0.14.8 (użyta w pracy Pana Molina) do wersji 16.13.1 [8]. Najważniejszą zmianą którą należy tutaj wspomnieć jest zmiana silnika reacta nosząca nazwę React Fiber która nastąpiła w wersji 16.0.0. Był to kamień milowy, i z punktu widzenia badania wydajności, jest to całkowicie nowe rozwiązanie.

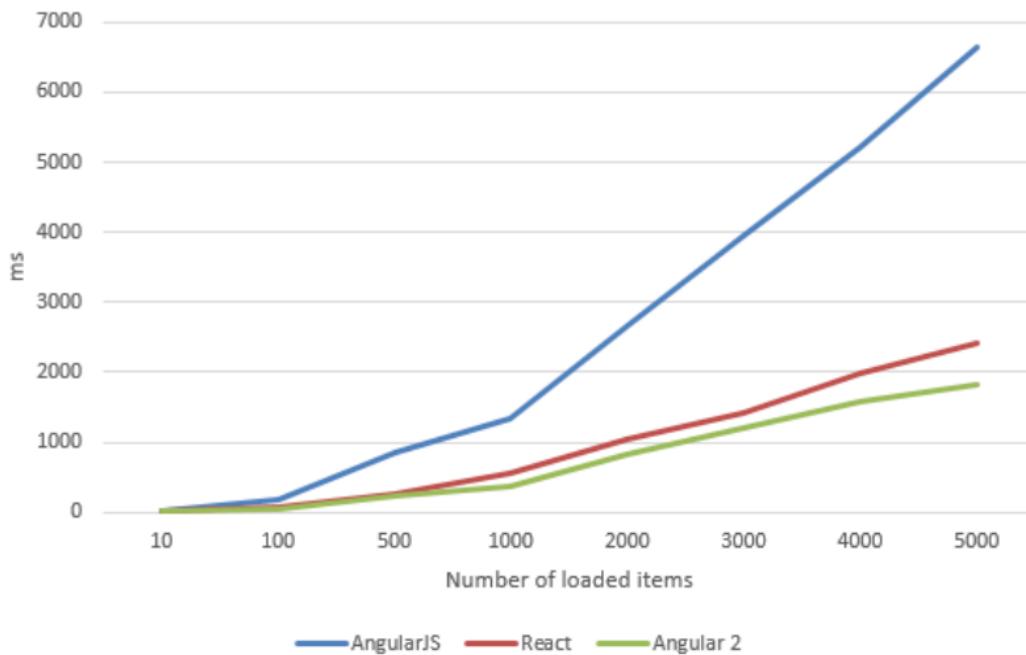
Następną sekcją która jest dla nas interesująca jest sekcja 5.2 w której autor prezentuje wyniki swoich testów. Wartym zauważenia jest wersja przeglądarki chrome, która wynosi 50.0.26661.102m. Najnowsza wersja chrome, w czasie pisania tego tekstu, to wersja 80.0.3987.16 czyli aż o 30 wersji nowszej.

W pierwszym teście, autor porównuje wydajność pomiędzy rozwiązaniami na przykładzie ładowania elementów od 10 aż do 5000 przedstawionym na rysunku 2.2.

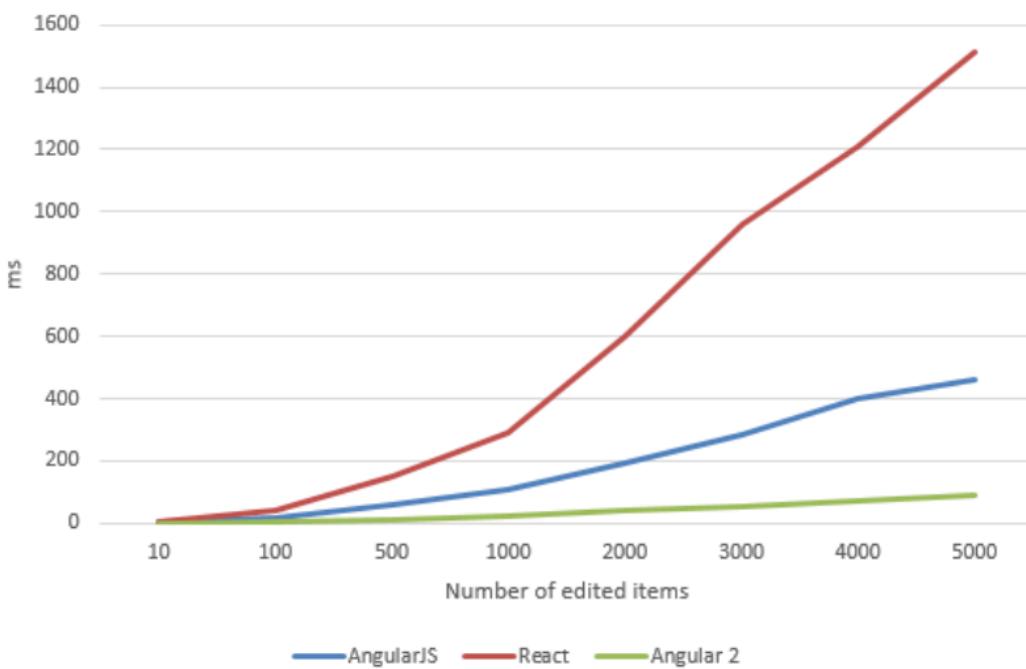
Z badania tego, możemy zauważyć jak bardzo AngularJS różni się od pozostałych, nowszych rozwiązań. Do pierwszych 100 elementów, wyniki są zbliżone, natomiast powyżej tej wartości, mamy nagłą zmianę stopnia nachylenia funkcji. Przyczyną takiego zachowania jest fakt, iż AngularJS nigdy nie był zoptymalizowany dla takich przypadków użycia podczas, gdy React oraz Angular2 już tak. W drugim teście, autor bada czas potrzebny do załadowania określonej ilości elementów. Wyniki przedstawiono na rysunku 2.3.

Widzimy znaczną dysproporcję pomiędzy każdym z rozwiązań. Z jednej strony AngularJS przypomina funkcję wykładniczą gdzie po przeciwej stronie Angular2 zachowuje się jak funkcja liniowa. Pokazuje to jak bardzo istotne zmiany zaszły pomiędzy zmianą generacji oraz

2. Część przeglądowa



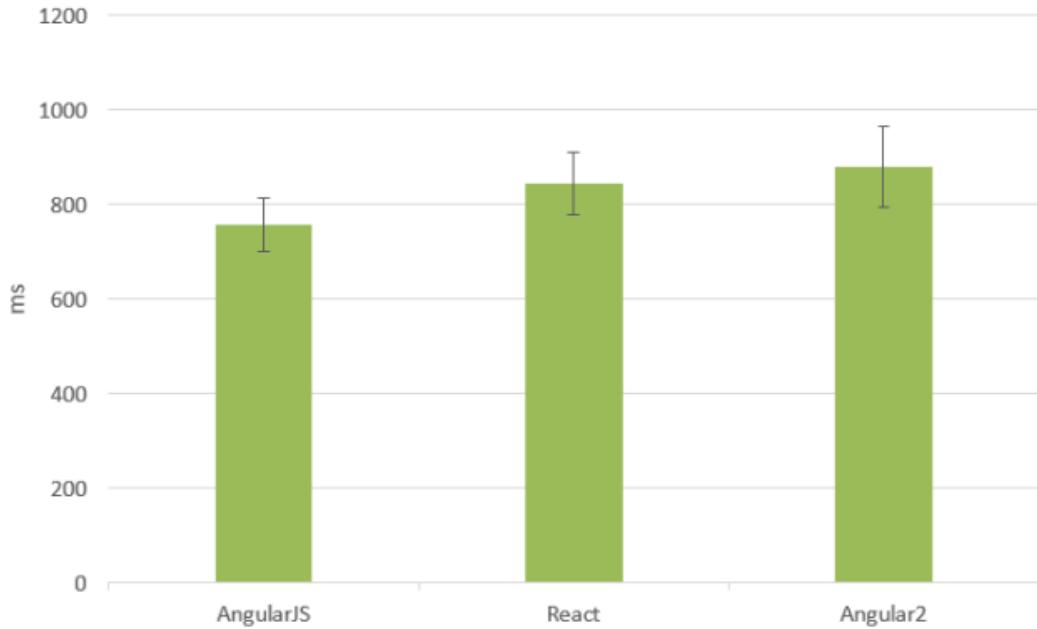
Rysunek 2.2. Grafika przedstawiająca wzrost czasu ładowania elementów od 10 do 5000 przy użyciu AngularJS, Angular2 oraz React [4]



Rysunek 2.3. Grafika przedstawiająca wykres zależności czasu do ilości edytowanych elementów na stronie [4]

wpływ doświadczenia jakie zebraliśmy w okresie panowania framework AngularJS. Wszak AngularJS był pierwszym podejściem do generycznego narzędzia dla dynamicznych aplikacji internetowych. W ostatnim teście wydajnościowym, autor porównuje czas ładowania się aplikacjami.

kacji. Wyniki przedstawiono na rysunku 2.4.



Rysunek 2.4. Grafika przedstawiająca wykres czasu ładowania aplikacji [4]

O ile autor faktycznie porównuje wydajność rozwiązań, nie określa on w sposób klarowny jak zaimplementowano mechanizm dodawania oraz aktualizacji elementów w pierwszym oraz drugim teście. Są to informacje bardzo istotne, gdyż każdy z frameworków działa w inny sposób, i ma to bezpośredni wpływ na otrzymane wyniki. Dodatkowo, autor nie podaje błędu pomiaru [9] który wyznaczył podczas przeprowadzania badania. Jest to bardzo istotna informacja, gdyż frameworki takie opierają się na całej piramidzie warstw oprogramowania które współdzielą zasoby komputera z innymi procesami. Ponadto auto nie zastosował żadnej warstwy abstrakcji nad systemem operacyjnym co powoduje, że wyniki mogą znacznie różnić się pomiędzy różnymi wersjami systemu.

2.2.2 Lighthouse

Jest to narzędzie open-source stworzone przez firmę Google do badania wydajności stron internetowych oraz aplikacji webowych [10]. Jest ono integralną częścią przeglądarki Chrome, tak więc jest ono niezwykle łatwe w użytkowaniu. Narzędzie to pozwala nam na ocenę naszej aplikacji w kilku wybranych kategoriach w tym wydajności. Dodatkowo, narzędzie pozwala nam na określenie trybu badania gdzie do wyboru mamy tryb mobilny oraz tryb użytkownika komputera domowego. Ma to bardzo duży wpływ na wynik testu, gdyż wydajność urządzeń mobilnych jest z reguły bardzo ograniczona w porównaniu do pełnoprawnej platformy domowej. Metrykami jakie możemy otrzymać to:

- Czas rozpoczęcia pierwszego renderowania
- Czas gdy strona faktycznie została wyrenderowana w całości

- Indeks szybkości - czas, gdy główna treść strony została wyrenderowana
- Czas do interakcji - czyli czas, gdy użytkownik faktycznie może zacząć interakcję ze stroną

Jednak o ile informacje te są jak najbardziej przydatne z punktu widzenia ładowania strony, narzędzie to nie jest w stanie określić faktycznego czasu interakcji ze stroną. Wpływ na taki czas ma wiele czynników z których najważniejszym jest ilość elementów na stronie. Jest to istotne, gdyż JavaScript jest językiem jednowątkowym z mechanizmem event loop [11]. Tak więc jesteśmy w stanie otrzymać informacje o wydajności ładowania aplikacji przez przeglądarkę, jednak narzędzie w żaden sposób nie wspiera pomiarów działania aplikacji po etapie ładowania, a właśnie na tym skupia się ta praca.

2.2.3 Porównanie narzędzi do tworzenia aplikacji typu SPA na przykładzie Angular2 i React

Praca Pań Jadwiga Kalinowska oraz Beata Pańczyk [12] także ma na celu porównanie aplikacji typu SPA, tym razem Angular2 oraz React. Autorzy przedstawiają szereg metryk według których porównywać będą obydwa narzędzia. Pierwszą metryką jest sama struktura aplikacji czyli jak wygląda architektura wewnętrzna rozpatrywanego framework'a. I jak autorki trafnie zauważają, obydwa rozwiązania czerpią garściami z nowoczesnej szkoły tworzenia wysoce skalowalnych i wydajnych rozwiązań implementując naturę tworzenia złożonych aplikacji z małych komponentów. Z komponentów takich składamy coraz to większą i bardziej złożoną aplikację. Drugą metryką rozpatrywaną przez autorki jest metryka kodu przedstawiona na rysunku 2.5. Innymi słowy, jak wiele kodu boilerplate należy napisać, aby osiągnąć ten sam efekt. Ma to wpływ przede wszystkim na końcowy rozmiar pliku który należy przesyłać pomiędzy serwerem a klientem. Im niższa wartość, tym lepiej.

		React	Angular
Komponent dodawania plików graficznych	Rozmiar plików	3,43 KB	3,16 KB
	Liczba plików	1	4
	Liczba linii kodu	103	107
Rozmiar biblioteki		240MB	339MB
Rozmiar całego projektu		241MB	340MB

Rysunek 2.5. Porównanie metryk kodu dla pojedynczego komponentu [12]

Wnioskiem płynącym z otrzymanych danych jest fakt, iż React jest finalnie znacznie mniejszym frameworkiem co bezpośrednie redukuje koszt tworzenia oraz utrzymania kodu. Ostatnim interesującym punktem jest metryka badająca bezpośrednią wydajność aplikacji przedstawiona na rysunku 2.6. Badanie miało na celu zmierzenie czasu, jaki potrzebny jest aby pobrać

2. Część przeglądowa

rekordy z przygotowanej bazy danych.

Liczba rekordów	Angular			React		
	Czas transferu [s]			Czas transferu [s]		
	Google Chrome	Mozilla Firefox	Microsoft Edge	Google Chrome	Mozilla Firefox	Microsoft Edge
1	7,74	12,79	13,48	6,96	7	8,17
5	8,96	13,22	24,73	7,03	7,53	11,35
10	11,78	15,44	35,66	7,08	7,99	14,26
25	15,4	18,75	66,47	12,94	12,88	52,71
50	21,63	23,49	107,15	14,87	18,59	76,1
75	25,34	28,97	152,6	22,88	23,49	94,36

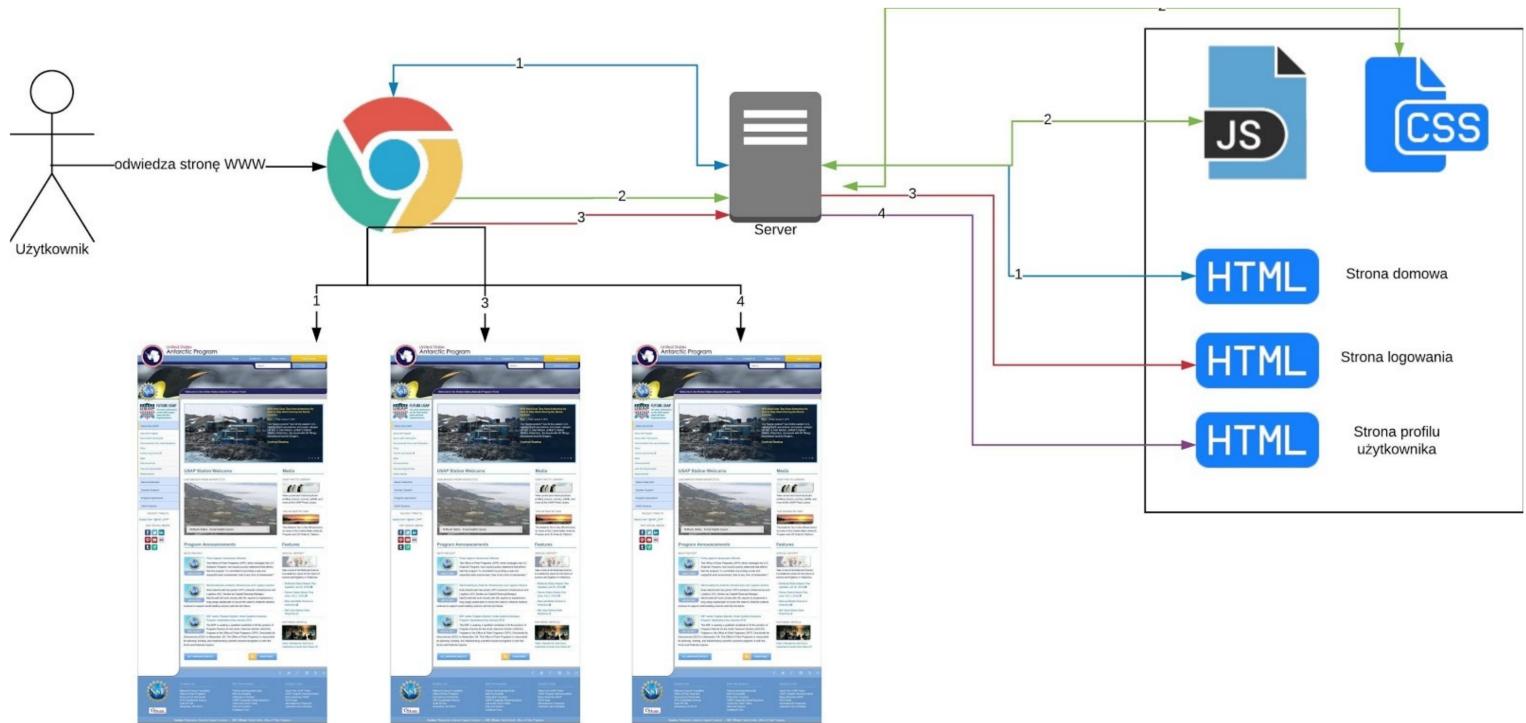
Rysunek 2.6. Porównanie czasów pobrania danych z bazy [12]

Interesującym wynikiem jest fakt, iż React w przeglądarce Google Chrome dla 1,5 oraz 10 wyników otrzymał bardzo zbliżone czasy bliskie błędowi pomiaru. Jednak im większa liczba rekordów pobrana z bazy, tym większą przewagę osiągał React. Wszystkie metryki które zostały zastosowane podczas porównania dobrze obrazują różnicę pomiędzy tymi rozwiązaniami, ale tylko w ujęciu wysoko poziomowym. Badanie zawiera w sobie serwer WWW oraz bazę danych, których czasy stanowią znaczną część czasu uzyskanego w wynikach. Nie pozwala to nam na jednoznaczne stwierdzenie, które narzędzie jest wydajniejsze, gdyż wykonano tylko jedno badanie, które nie pozwala na ocenę tych narzędzi pod kątem różnych przypadków użycia.

Część praktyczna

3. Część praktyczna

Cykl życia standardowej strony internetowej przedstawiono na rysunku 3.1



Rysunek 3.1. Ilustracja przedstawiająca cykl życia statycznej strony internetowej

- Użytkownik otwiera przeglądarkę i odwiedza stronę WWW.
- Przeglądarka wysyła zapytanie do serwera z żądaniem zasobów.
- Serwer odsyła plik HTML w którym znajdują się odnośniki do plików JavaScript oraz CSS.
- Przeglądarka pobiera pliki JavaScript oraz CSS
- Przeglądarka renderuje stronę dla użytkownika.
- Użytkownik kliknie w odnośnik do kolejnej strony (w przykładzie strona logowania)
- Przeglądarka pobiera cały nowy plik HTML
- Przeglądarka renderuje nową stronę od nowa.
- Przeglądarka ponownie pobiera dodatkowe pliki JavaScript oraz CSS (zakładamy, że przeglądarka nie przechowuje plików w pamięci podręcznej).
- Przeglądarka renderuje stronę od nowa.

Jak widzimy, cały proces odwiedzenia dwóch stron w tej samej domenie wymaga każdorazowego przeładowania strony oraz pobrania tych samych danych. Proces ten wymaga ciągłego pobierania plików HTML, pobierania plików dodatkowych co obciąża procesor, pamięć łącze

oraz sam serwer.

Dla porównania, przedstawiona poniżej ilustracja pokazuje cykl działania aplikacji SPA (rysunek 3.2)

- Użytkownik odwiedza tę samą stronę .
- Przeglądarka pobiera plik HTML z definicją potrzebnych zasobów. Zazwyczaj sekcja body pliku jest pusta.
- Przeglądarka pobiera dodatkowe pliki JavaScript oraz CSS.
- Często pliki te są zoptymalizowane poprzez kompresję i sklejone w jeden plik tak, aby wykonać tylko jedno zapytanie zamiast dwóch.
- Pliki JavaScript, generują treść strony HTML.
- Przeglądarka może (ale nie musi) wykonać dodatkowe zapytanie o dane do jednego z endpointów serwera.
- Użytkownik kliką na odnośnik do kolejnej strony. Przeglądarka nie pobiera już dodatkowego pliku HTML, jako, że aplikacja jest dynamiczna. Skrypt JavaScript może, ale nie musi pobrać dodatkowych informacji z serwera.

Jak widzimy, wykonano znacznie mniej zapytań do serwera. Dodatkowe zapytania które skrypt może wykonać w celu pobrania danych zazwyczaj są znacznie mniejsze od pełnego pliku HTML z wybraną treścią. Dodatkowo, przeglądarka nie przeładowuje strony przy każdej zmianie podstrony, co także odciąża procesor jak i łącze sieciowe.

3.1 Problem pomiaru wydajności aplikacji SPA

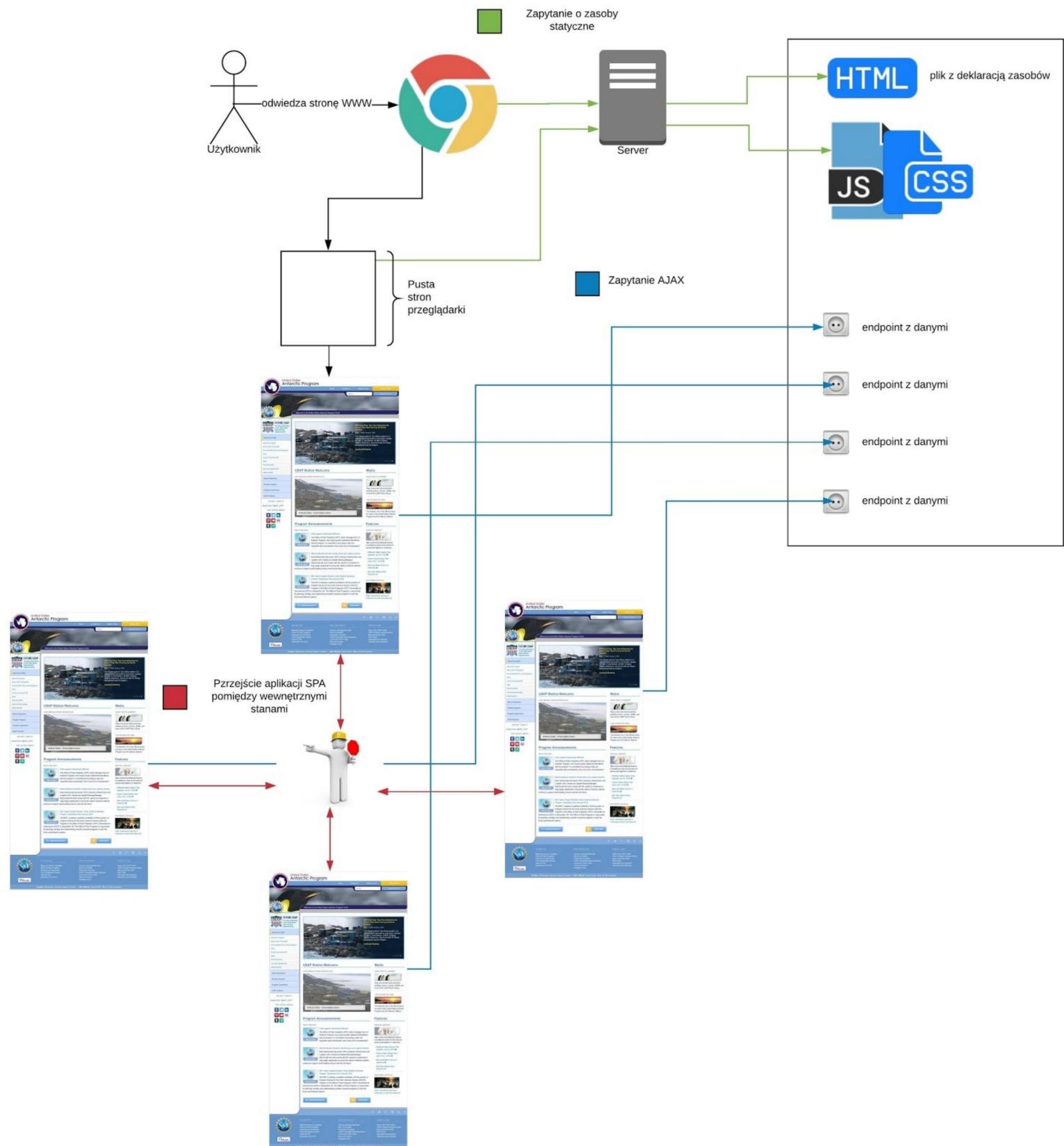
Z racji, że w aplikacjach SPA nigdy nie następuje przeładowanie strony, a ich działanie jest ciągłe pojawia się problem następującej natury.

W celu ułatwienia zrozumienia problemu posłużę się przykładem przedstawionym na rysunku 3.3. Założmy że jesteśmy na platformie Netflix. Otworzyliśmy stronę z listą dostępnych filmów do obejrzenia.

Aplikacja w stanie początkowym nie posiada żadnych filmów. Użytkownik kliką przycisk pobierz listę filmów. Efektem ubocznym (side effect) jest wysłanie informacji o akcji do menedżera stanu. Menedżer stanu wysyła zapytanie do serwera o listę filmów do zaproponowania. Serwer odpowiada z danymi, i menedżera stanu przechodzi w nowy stan. Podstrona otrzymuje informacje o nowym stanie. Pobiera nowy stan z menedżera stanu i prze renderuje widok.

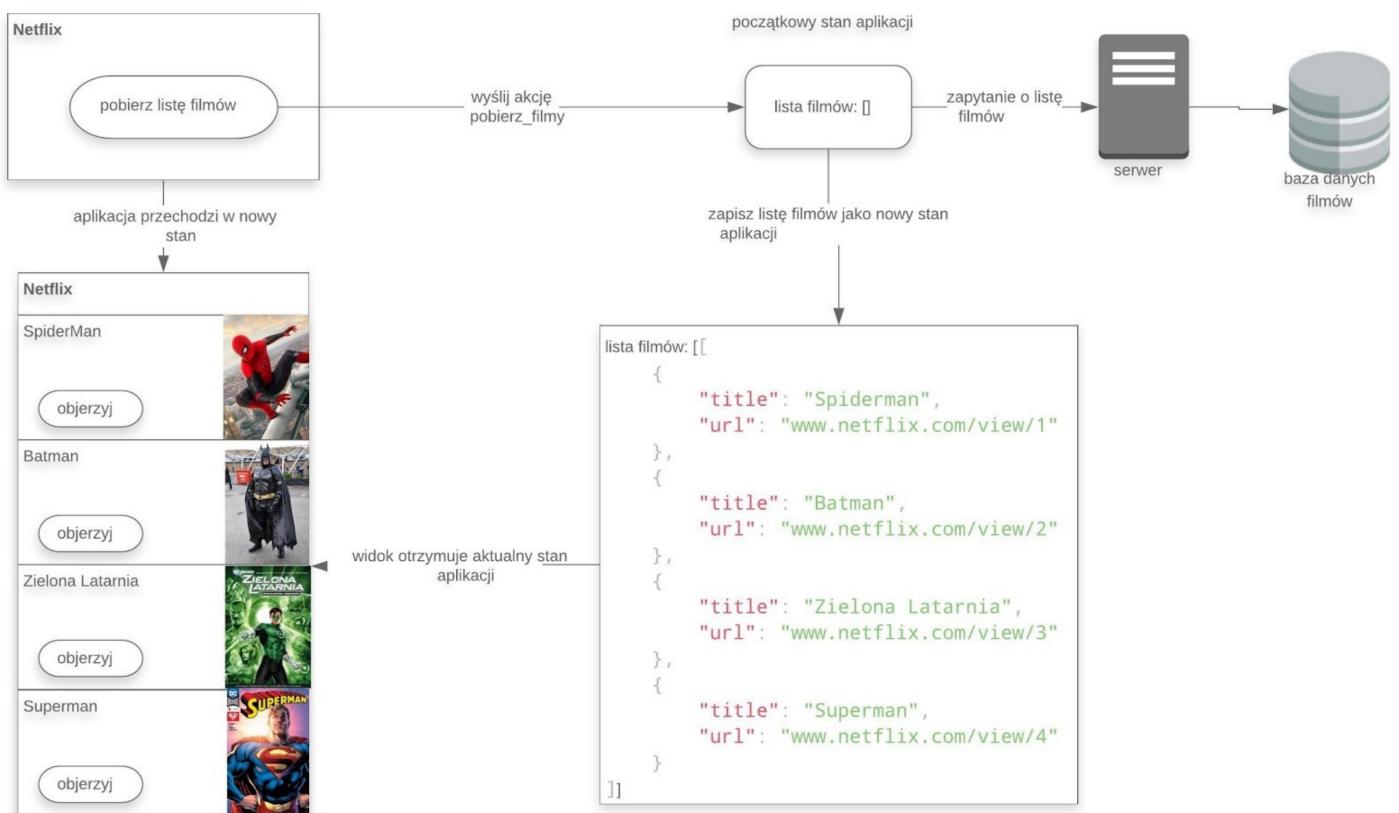
Problem pojawia się gdy chcemy określić kiedy cała procedura została zakończona. Samo sprawdzenie czy przycisk został naciśnięty nie jest wystarczające, gdyż z perspektywy użytkownika, procedura obejmuje także rezultat efektu ubocznego. Zapytanie do serwera zajmie od kilkudziesięciu milisekund do nawet i kilku czy kilkunastu sekund. Samo renderowanie także nie jest deterministyczny w kontekście czasu operacji gdyż wpływ na nie mają inne usługi korzystające z procesora czy pamięci komputera. Z racji, iż frameworki internetowe muszą być

3. Część praktyczna



Rysunek 3.2. Ilustracja przedstawiająca cykl działania aplikacji SPA

3. Część praktyczna



Rysunek 3.3. Grafika przedstawiająca mechanizm działania aplikacji dynamicznych

3. Część praktyczna

generyczne, nikomu jeszcze nie udało się stworzyć idealnego i jednolitego mechanizmu wyznaczania, czy dana akcja została zakończona czy nie.

Tak więc na potrzeby pracy istotnym jest, wprowadzenie jednoznacznej definicji akcji w aplikacji.

Akcja aplikacji - jest to zbiór procedur oraz zdarzeń i ich obsługi następujący w chwili interakcji użytkownika z aplikacją.

3.2 Specyfikacja wymagań

W celu przeprowadzenia doświadczenia mającego na celu pomiar wydajności framework'ów, należy posiadać odpowiednie środowisko badawcze. Jak pokazano w przeglądzie istniejących rozwiązań, na rynku brak jest narzędzi pozwalających na przeprowadzenie pełnego i dokładnego badania.

Na potrzeby pracy stworzono narzędzie do rozwiązania zadanego problemu badawczego. W ramach projektu, założono co następuje:

- Narzędzie jest generyczne - nie zakładamy konkretnej technologii aplikacji tak, że każdy istniejący jak i przyszły framework może zostać przebadany.
- Stworzona zostanie ujednolicona lecz generyczna metoda dodawania nowej aplikacji do puli już istniejących.
- Narzędzie musi posiadać obsługę różnych typów akcji aplikacji.
- Narzędzie musi potrafić działać w odizolowanym środowisku (docker)
- Narzędzie będzie zautomatyzowane w znacznym stopniu.
- Narzędzie będzie składać się z modułów tak, aby było łatwo rozwijalne w przyszłości.
- Narzędzie pozwala na stworzenie różnych zestawów testów tak, aby użytkownik mógł dokładnie przebadać różne warianty akcji aplikacji.
- Stworzony zostanie moduł parsowania danych z badania tak, aby można było otrzymać ujednolicony format wyniku.
- Stworzony zostanie moduł wizualizujący wyniki.

3.3 Architektura rozwiązania

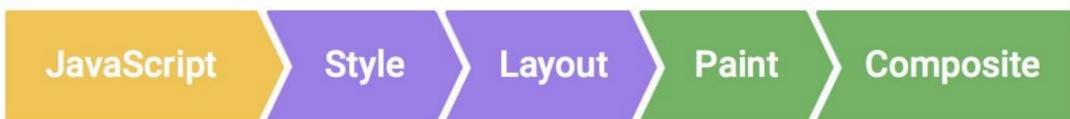
Narzędzie które powinno powstać w celu rozwiązania tak skomplikowanego problemu nie jest łatwym zadaniem. Na wstępie przedstawiono ogólny zarys architektury rozwiązania które spełnia wymogi przedstawione powyżej. Z racji na obszerność rozwiązania, przedstawione zostanie ono w małych rozdziałach.

3.3.1 Projekt testu wydajności

Z założenia każda aplikacja którą chcemy porównać musi być zbudowana w możliwie zbliżony do siebie sposób. Z racji, iż aplikacje internetowe są tworzone w sposób generyczny, istnieje mnogość możliwości implementacji każdego wymagania. Innymi słowy, każde zadanie można wykonać na wiele różnych sposobów w ramach jednego narzędzia. Każde rozwiązanie posiada swoje wady oraz zalety. Niektóre narzędzia pozwalają na dodatkową optymalizację kodu pod konkretne zadanie, pozwalając uzyskać jeszcze lepsze wyniki. Niestety, mechanizmy takie polegają w znacznej mierze na zdolnościach i wiedzy programisty. W ramach badania, chcemy porównać wydajność frameworków, a nie zdolności programisty, dlatego też aplikacje powinny zostać zaprojektowane w taki sposób, aby:

- Były zdolne wykonać te same zadania
- Były możliwie proste
- Nie wymagały dogłębnej wiedzy programisty
- Posiadały ujednoliczoną strukturę projektu
- Z racji, iż dla przeglądarki najtrudniejszym zadaniem jest renderowanie elementów, Test który zaprojektowano skupia się dokładnie na porównaniu wydajności renderowania dużej liczby elementów.

W tym miejscu warto odnieść się do świetnego artykułu autorstwa Pana Paula Lewisa [13] na temat optymalizacji renderowania w przeglądarce. Proces ten składa się z pięciu kroków przedstawionych na rysunku 3.4.



Rysunek 3.4. Grafika przedstawiająca kolejność faz renderowania w przeglądarce

Na początku JavaScript wykonuje zmianę na dokumencie HTML. Następnie przeglądarka wylicza na podstawie arkusza stylów (CSS) pozycję elementów na płótnie przeglądarki. Następnie znając rozmieszczenie elementów na stronie, przeglądarka wylicza ich wzajemne pozycje oraz rozmiar. Elementy na stronie mają na siebie wzajemny wpływ na przykład dla dwóch bloków . W zależności od wysokości pierwszego bloku, drugi blok może zostać wyrenderowany, bądź też nie jeżeli wykracza on poza widoczny obszar rysowania, a także możemy wyliczyć dokładną pozycję na płótnie. Znając pozycję oraz rozmiary elementów, przechodzimy do fazy malowania. Pixele zostają malowane na wielu warstwach. W ostatniej fazie kompozycji, warstwy wyświetlane są na ekranie w konkretnej kolejności (odpowiada za to między innymi atrybut z-index w języku CSS).

Jak widzimy, renderowanie pojedynczej klatki na stronie jest bardzo skomplikowanym procesem, który w idealnym przypadku powinien następować około 60 razy na sekundę (w zależności od częstotliwości odświeżania ekranu). Bardzo istotnym mechanizmem który przeglądarki stosuje w celu optymalizacji ilości pracy jest pomijanie niepotrzebnych fragmentów. Jeżeli zmianie uległa tylko pewna część strony, na przykład kolor tekstu, możemy ominąć etap wyliczania layoutu strony. Jeżeli nastąpi zmiana stanu, w której nie wystąpiła zmiana ani layoutu ani nie występuje potrzeba ponownego malowania, przeglądarka używa poprzedniego zestawu gotowych warstw. Zrozumienie tego mechanizmu jest istotnym czynnikiem na którym oparłem konstrukcję testów wydajnościowych.

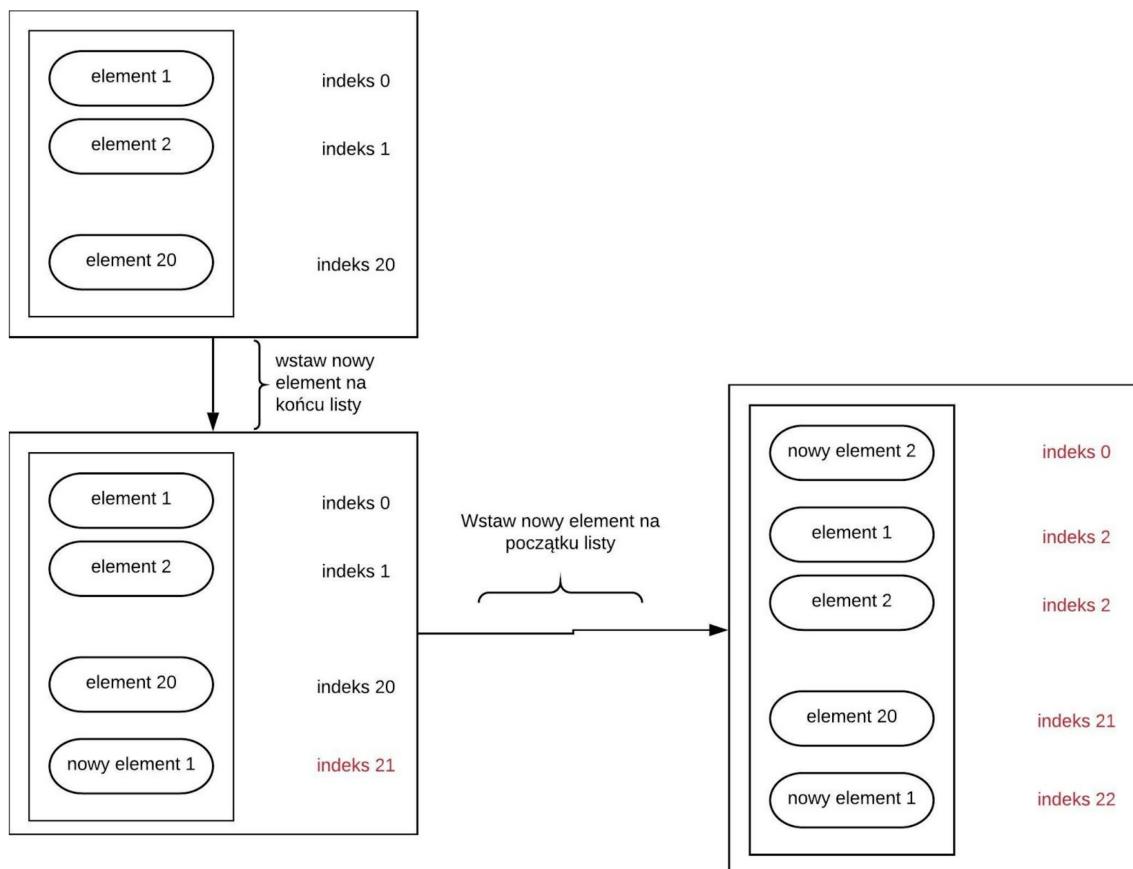
Wracając do samej konstrukcji testu. Znając już mechanizm omijania czynności podczas renderowania obrazu w przeglądarce możemy przedstawić problem renderowania list. Założmy, że mamy listę 20 wyrenderowanych elementów na stronie. Do listy możemy dodać nowe wartości albo na początku albo na końcu. W pierwszym kroku, dodać jeden nowy element do listy, na końcu co przedstawiono na rysunku 3.5. Każdy element na liście ma przypisany indeks. Dodać nowy element do listy, w optymalnym przypadku, powinienem wyrenderować tylko jeden nowy element i dokładnie tak się stanie. Co stanie się, gdy dopiszę nowy element na początku listy? Otóż wszystkie indeksy zostaną przesunięte o 1. React wykryje zmianę jednego elementu na liście, i przeglądarka wyrenderuje tylko jeden nowy element. Widzimy także, że znaczna część fazy layoutu będzie taka sama, gdyż na ekranie zmieni się tylko jeden widoczny element.

Co stanie się w przypadku dopisania jednego nowego elementu na początku listy? Problem ten zaprezentowano na rysunku 3.6. Każdy indeks przypisany do elementu zostanie przesunięty o jeden. Z racji tej, React ponownie wyrenderuje każdy element (gdyż uzna, iż jest to nowy element). Spowoduje to zmianę kodu HTML która zmusi przeglądarkę do pełnego prze renderowania wszystkich warstw oraz wyliczenia całego layoutu na nowo. Oczywiście w skali 20 elementów które nie posiadają skomplikowanej struktury HTML i stylowania CSS nie powinno to robić problemu. Jednak w dobie nowoczesnych aplikacji, listy (np lista filmów netflixa) posiadają masę małych elementów przez co lista nawet 50 elementów może spowodować zauważalne opóźnienia podczas użytkowania aplikacji.

W jaki sposób możemy poradzić sobie z tym problemem? Istnieją dwa sposoby dzięki którym React obiecuje znacznie wyższą wydajność w porównaniu do czystego kodu JavaScript. Pierwszym z nich jest unikalny indeks elementu [14]. Zamiast stosować klucz równego indeksowi elementu, musimy podać klucz który jest unikalny w ramach elementu. Przykładowo dla każdego elementu stworzymy parę (klucz, hasz). W tym momencie, jeżeli dodamy nowy element na początku listy, React spojrzy na listę kluczy, i zobaczy, że nie zmieniły się one, a dodano tylko jeden nowy element. Zamiast 22 renredowań, otrzymamy tylko jedno.

Drugim sposobem poradzenia sobie z minimalizacją ilości renderowania jest koncepcja wirtualnego modelu DOM [15]. Jest to mechanizm, który tworzy reprezentację interfejsu użyt-

3. Część praktyczna



Rysunek 3.5. Ilustracja przedstawiająca problem dopisywania elementu na koniec listy

3. Część praktyczna



Rysunek 3.6. Ilustracja przedstawiająca problem dopisywania elementów na początek listy

3. Część praktyczna

kownika w pamięci i synchronizuje ją z modelem DOM przeglądarki. Proces ten nazywa się rekoncyliacją [16]. Pozwala to na wyrenderowanie tylko tych elementów, które różnią się między reprezentacją w pamięci a reprezentacją faktyczną drzewa DOM.

Znając już konkretny problem, z którym aplikacje SPA muszą się mierzyć, postanowiono zaprojektować szereg akcji manipulujących listą elementów oraz zbadać, ile czasu zajmie konkretnego frameworkowi wykonanie takiej czynności.

3.3.2 Budowa aplikacji

Każda aplikacja będzie składać się z szeregu przycisków. W wewnętrznym stanie aplikacji, będziemy trzymać listę wartości do wyrenderowania. Wartości które będziemy renderować będą stringiem o długości 10 znaków. Proces zmiany aplikacji prezentuję na poniższym rysunku.

Na początku stan wewnętrznego aplikacji będzie pustą tablicą. Spowoduje to wyrenderowanie stanu minimalnego, czyli bloku . Akcja naciśnięcia przycisku spowoduje wygenerowanie 250 wpisów zawierających wylosowane alfanumeryczne wartości. Wartości te muszą być, w miarę możliwości unikatowe. Zmiana stanu z kolei spowoduje ponowne wyrenderowanie komponentu wyświetlającego listę. Kolejne akcje będą działać w myśl tej samej zasady.

Na rysunku poniżej przedstawiam listę akcji jakie aplikacja musi implementować:

- *Append 250 rows start* - dodaj 250 wierszy na początku listy.
- *Append 250 rows end* - dodaj 250 wierszy na końcu listy.
- *Replace all rows* - wygeneruj na nowo pełną listę wartości
- *Update N rows* - wybierz losowo 250 wierszy i zaktualizuj ich wartości
- *Swap rows* - zamień miejscami 2 wiersze
- *Remove row* - usuń jeden wiersz
- *Clear rows* - usuń wszystkie wiersze

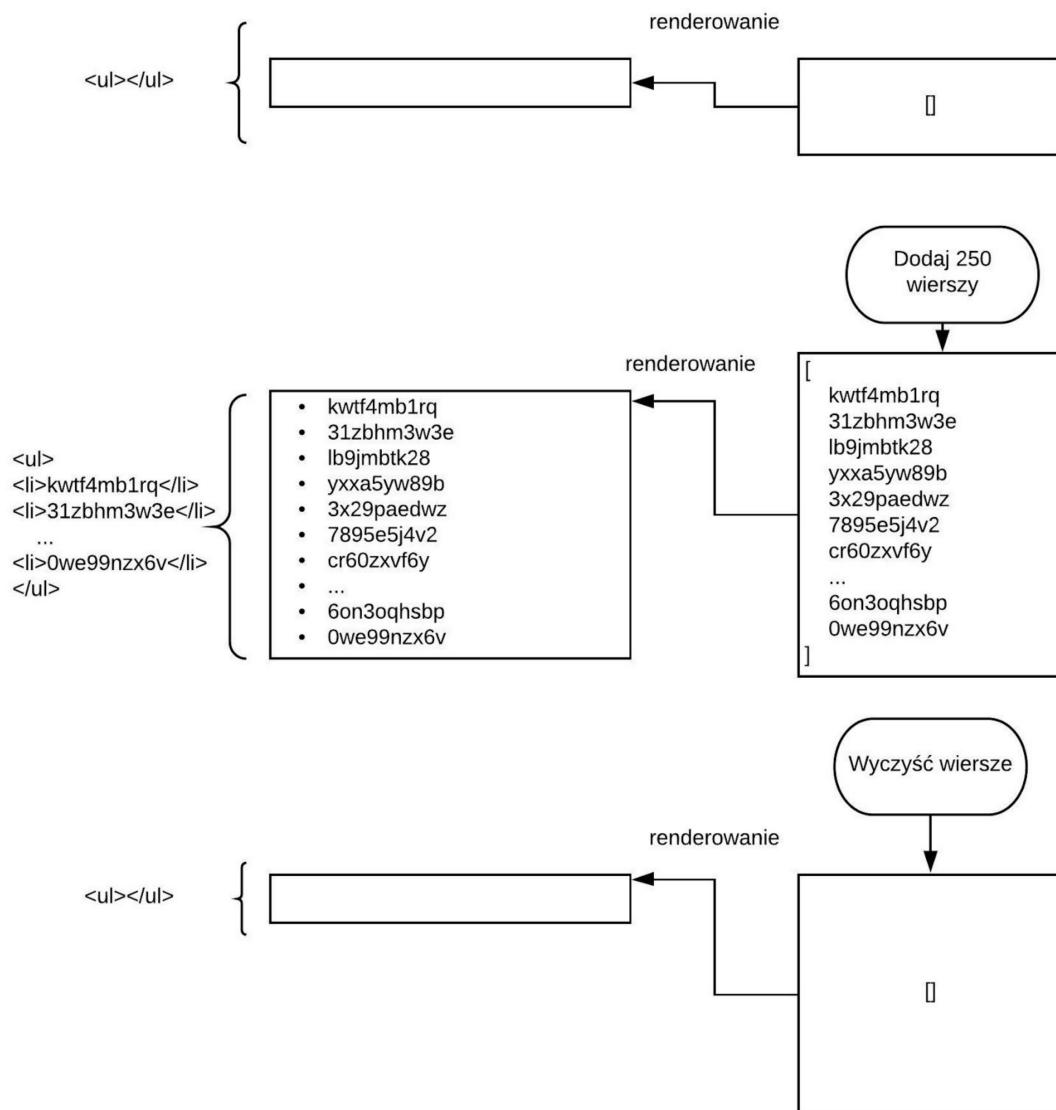
Na początku części praktycznej wprowadzono definicję akcji aplikacji. Widzimy tutaj, iż samo naciśnięcie przycisku dodania aplikacji nie jest wystarczające, gdyż przed wyrenderowaniem listy musimy wygenerować listę wpisów. Renderowanie aplikacji także jest nie deterministyczne, i wpływ na nie ma szereg czynników niezależnych od przeglądarki. Spowoduje to znaczne różnice w wynikach pomiaru czasu akcji aplikacji, z tego też powodu istotnym jest upewnienie się, iż kolejne iteracje badania są od siebie możliwie niezależne.

Ostatnim elementem składającym się na aplikację jest jej infrastruktura. Każda aplikacja tworzona jest w całkowicie inny sposób, dlatego istotnym jest, aby ujednolicić proces budowania i pakowania plików wynikowych. Z racji tego, każda aplikacja musi posiadać w sobie plik Makefile [17] z dwoma możliwymi celami:

- install
- Production

Cel *install* jak sama nazwa wskazuje, przygotuje folder *node_modules* zawierający pobrane biblioteki z platformy NPM [18] Drugi cel *production* zajmie się procesem pakowania aplikacji

3. Część praktyczna



Rysunek 3.7. Ilustracja mechanizmu przebiegu badania

Append 250 rows start | Append 250 rows end | replaceAllRows | updateNRows | Swap Rows | Remove Row | Clear rows

Rysunek 3.8. Grafika przedstawiająca zaimplementowane przyciski w aplikacji

i optymalizacji kodu w celach produkcyjnych [19].

3.3.3 Przygotowanie aplikacji do badania

Kolejnym krokiem podczas przygotowania aplikacji do badania, jest zautomatyzowanie procesu budowania i transferowania aplikacji. Przygotowujemy je tym samym do konteneryzacji [20]. W ramach tej części procesu, stworzono 3 skrypty.

- Compile all applications - skrypt ten ma na celu przejście po wszystkich folderach aplikacji, i wykonanie komend install oraz production. Komendy wykonują szereg intensywnych zadań polegających na pobraniu dużej ilości danych a następnie transpilacji [2] i optymalizacji kodu które są zadaniami wysoce obciążającymi procesor. Dlatego też przygotowanie aplikacji sekwencyjnie zajmuje dużo czasu i jest nieefektywne. Skrypt ten został zoptymalizowany aby wykonywał procesowanie współbieżne za pomocą mechanizmu obietnic [21] w Javascript, dzięki temu w pełni wykorzystuje możliwości procesora.
- Copy released apps - prosty skrypt mający na celu przekopiowanie folderu dist z każdej aplikacji do głównego folderu "releases".
- Create index html file - skrypt ten ma na celu stworzenie pliku index.html. W pliku tym zawrzemy informację o przygotowanych aplikacjach. Plik ten będzie dla nas istotny w dalszej części procesowania, gdy automat będzie wykrywać dostępne aplikacje do badania.

Cały proces przedstawiony został na rysunku 3.9.

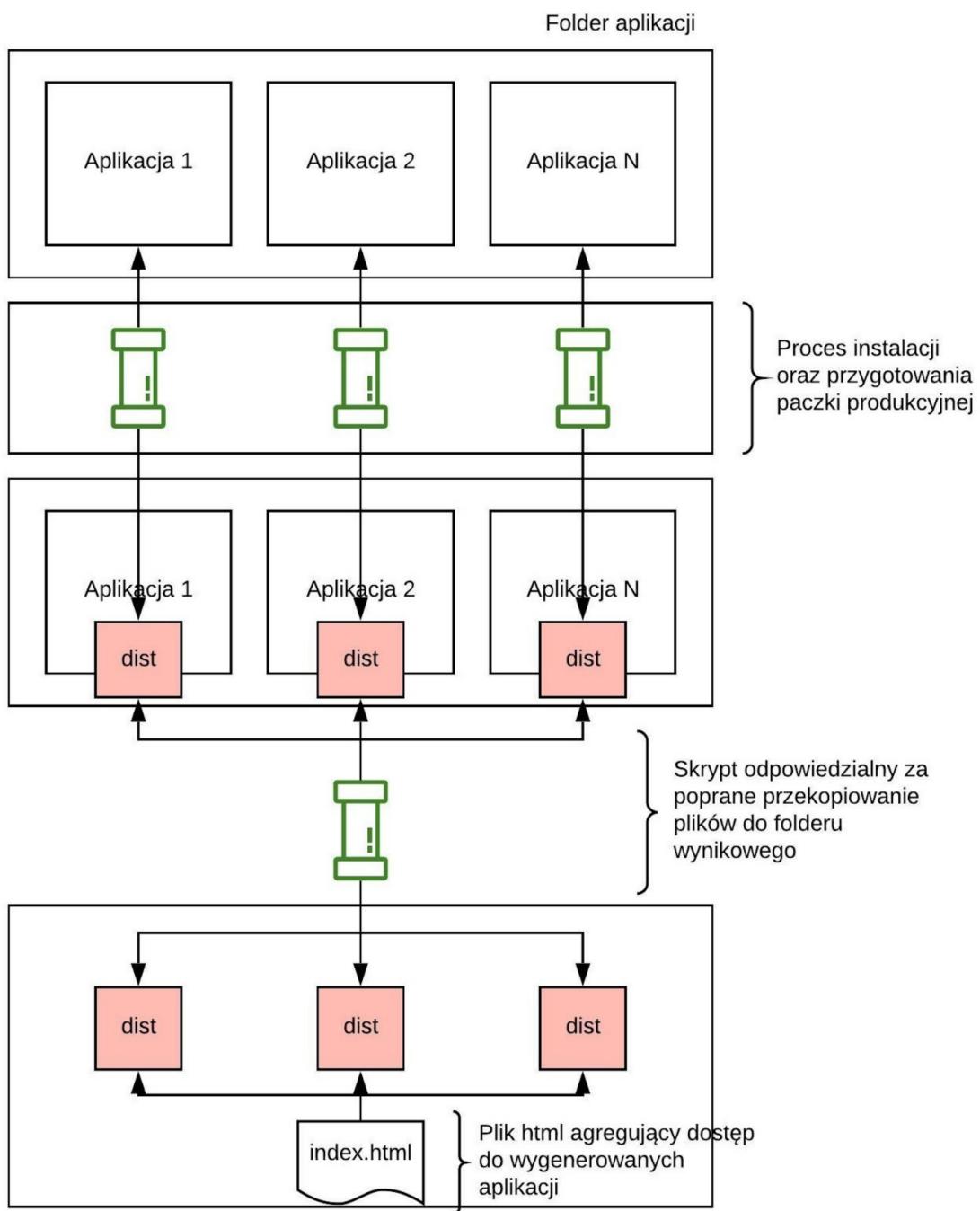
3.3.4 Konteneryzacja aplikacji

Na tym etapie, dysponuje już zbudowanymi paczkami aplikacji. Aplikacje zostały zoptymalizowane do produkcyjnego użytku. Dodatkowo posiadam plik index.html stanowiący punkt wejścia do dalszego procesowania.

Kolejnym krokiem jest stworzenie mechanizmu umożliwiającego dostęp do aplikacji. W normalnym środowisku służą do tego serwery aplikacji [22]. Przykładem takiego serwera jest Nginx lub Apache. Z racji, iż instalacja takich serwerów wiąże się z pobraniem dużej ilości danych oraz każdorazową konfiguracją środowiska, skonteneryzowane rozwiązania znaczco przyspieszy i uprości pracę.

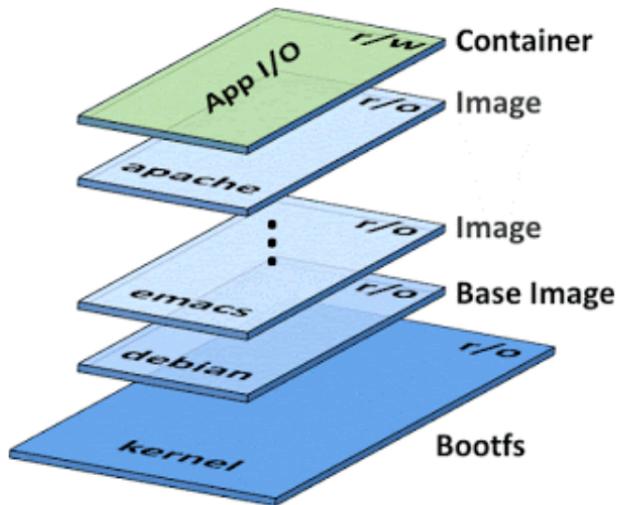
Jednym z założeń narzędzia jest, aby całość działała w odizolowanym środowisku przy użyciu platformy docker [23]. W zależności od systemu operacyjnego, zestaw instrukcji potrzebnych do instalacji serwera może być całkowicie inny [24][25]. Dlatego też wykorzystanie odizolowanej warstwy abstrakcji nad systemem operacyjnym jest istotnym zabiegiem.

Istotnym detalem dotyczącym platformy docker jest jej zasada działania. Aplikacje składają się z warstw obrazów, co przedstawiono na rysunku 3.10. Aby rozpocząć nasz własny obraz, należy wybrać obraz bazowy. W celu zmniejszenia wagi obrazów, zazwyczaj stosuje się obrazy czystych, okrojonych wersji systemu Linux. Przykładem takiego obrazu jest obraz alpine



Rysunek 3.9. Ilustracja procesu przygotowania aplikacji do konteneryzacji

[26], który jest systemem Linux o wadze 5 MB. Następny obraz jako swoją bazę wykorzysta obraz alpine, i doda kolejną warstwę oprogramowania. Możliwym jest także jako bazę nowego obrazu, wybranie pełnoprawnej wersji systemu Linux na przykład bazą może być Ubuntu [27]. Dzięki wykorzystaniu podejścia komponentowego, rozwiązanie to jest bardzo generyczne i pozwala na publikację tego samego oprogramowania na wiele różnych sposobów.



Rysunek 3.10. Ilustracja przedstawiająca warstwy składające się na przykładowy obraz dockera

W celu stworzenia kontenera, należy utworzyć plik Dockerfile który jest plikiem typu YAML. W pliku tym precyzujemy, jaki obraz dockera ma być bazą naszego nowego obrazu. Podczas pisania implementacji narzędzia, użyto obrazu *nginx:1.17.6-alpine*. Następnie kopowane są przygotowane w poprzednich krokach aplikacje do folderu */var/www*. Jest to folder który NGINX wykorzystuje do serwowania plików. Kolejny krokiem jest przygotowanie wcześniej stworzonego pliku *nginx.conf* jako pliku konfiguracyjnego serwera NGINX. Prezycje się w nim wiele istotnych opcji, lecz dla nas jedyną interesującą opcja jest ścieżka do folderu aplikacji, port oraz plik *index.html*. Następnie następuje etap budowy nowego kontenera.

Z racji, iż nowy kontener musi być zbudowany od nowa za każdym razem, gdy zmienimy kod aplikacji, stworzono plik *Makefile* mający na celu usprawnienie całego procesu. Należy wykonać komendę **make build_nginx_image**. Po zbudowaniu obrazu, możemy uruchomić nasz serwer za pomocą komendy **make start_nginx_docker**. Będzie on dostępny pod adresem <http://localhost:8085>.

Ostatnim elementem jest opisanie faktycznego wyniku całego procesu. W tym celu, wchodzimy na stronę <http://localhost:8085>. Naszym oczom ukaże się bardzo prosta strona startowa przedstawiona na rysunku 3.11. Strona ta będzie potrzebna kolejnemu modułowi badawczemu opartemu na Selenium, w celu wykrycia, jakie aplikacje są dostępne do badania.

List of available apps:

- [angular2](#)
- [react](#)
- [vue](#)

Rysunek 3.11. Grafika przedstawiająca plik index.html wraz z dostępnymi aplikacjami do badania

Możemy kliknąć na odnośnik przenoszący nas do konkretnej aplikacji. Należy tylko pamiętać o poprawieniu portu, czyli aby przejść do implementacji reacta, należy wejść w link <http://localhost:8085/react/>.

3.3.5 Moduł badania

Przechodzimy teraz do najbardziej skomplikowanego modułu całej pracy. Mając już przygotowane aplikacje, wystawione do badania za pomocą serwera webowego, możemy w tym momencie zacząć interakcję z aplikacjami. Nie mniej pojawia się duży problem pod tytułem, w jaki sposób możemy zacząć interakcję z aplikacjami? Jak już wspomniano w poprzednich rozdziałach, akcje aplikacji są nie deterministyczne, co znaczco utrudnia możliwość badania takich aplikacji. Ale na czym dokładnie polega problem?

W ramach pracy, posłużono się narzędziami powszechnie stosowanymi w celach testów end-to-end. Ale najpierw, spójrzmy na problem z którym mamy do czynienia aby zrozumieć, jaki wpływ ma on na wyniki badania.

W statycznych aplikacjach internetowych, interakcja użytkownika ze stroną jest zazwyczaj bardzo uproszczona. Znacząca większość akcji, będzie prowadziła do przeładowania treści strony (o czym mówiliśmy już w poprzednich rozdziałach pracy). Zobrazowano to na przykładzie języka PHP na rysunku 3.12.

Oto najprostszy, wiekowy już przykład aplikacji kalendarza napisanej w PHP. Gdy będziemy chcieli zmienić miesiąc z marca na kwiecień, nastąpi przeładowanie całej strony.

Teraz, z punktu badawczego chcemy ustalić ile trwała cała akcja. Prosimy nasz automat o naciśnięcie przycisku. Poszukujemy odpowiedzi na pytanie, ile czasu upłynie, od momentu naciśnięcia przycisku do momentu wyrenderowania nowej treści. Przeanalizujmy więc cały proces ukazany na rysunku 3.13.

Z punktu widzenia automatu do testów end-to-end, cały proces, o ile skomplikowany ma jasno zdefiniowany koniec. Końcem całego procesu jest przeładowanie strony przez przeglądarkę oraz wyrenderowanie treści. Przeglądarka posiada wbudowane w swoje API szereg zdarzeń

3. Część praktyczna

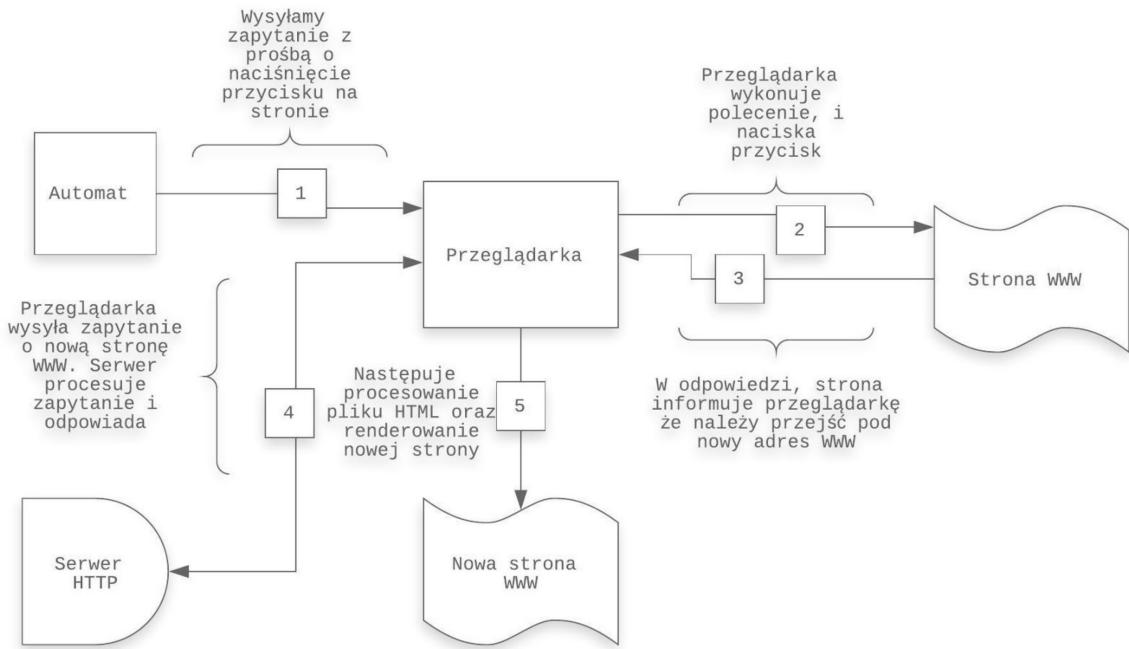
Zmiana miesiąca nie spowodowała dynamicznej zmiany

Dopiero zatwierdzenie spowoduje przeładowanie strony

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
09	28	29	1	2 (1 event) • Verton Art Gallery	3 (1 event) • Naming ceremonies	4	5
10	6	7 (2 events) • QA work • QA Testing	8 (2 events) • Football Awards 2009 • Repatriation	9	10 (1 event) • International Trade	11	12 (1 event) • Marketing and Design
11	13 (1 event) • Verton Art Gallery	14 (7 events) • Walking • Recycling • Water Testing • Age Sales • Repatriation • Graffiti • QA work <small>Markations were show all »</small>	15	16 (3 events) • Economic info • Teaching • DENTISTA	17	18 (1 event) • Economic info	19
12	20	21	22 (2 events) • Naming ceremonies • Walking	23	24	25 (6 events) • Bin Collections • Golf • Recycling • Outdoor activity • Teaching • International Trade <small>show all »</small>	26
13	27 (1 event) • Bin Collections	28 (2 events) • Graffiti • QA work	29	30 (1 event) • Age Sales	31 (1 event) • Catalogues	1	2

Przeładowanie strony

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
13	27	28	29	30	31	1	2 (1 event) • Verton Art Gallery
14	3 (1 event) • Naming ceremonies	4	5	6	7 (2 events) • QA work • QA Testing	8 (2 events) • Football Awards 2009 • Repatriation	9
15	10 (1 event) • International Trade	11	12 (1 event) • Marketing and Design	13 (1 event) • Verton Art Gallery	14 (7 events) • Walking • Recycling • Water Testing • Age Sales • Repatriation • Graffiti • QA work <small>Markations were show all »</small>	15	16 (3 events) • Economic info • Teaching • DENTISTA
16	17	18 (1 event) • Economic info	19	20	21	22 (2 events) • Naming ceremonies • Walking	23
17	24	25 (6 events) • Bin Collections • Golf • Recycling	26	27 (1 event) • Bin Collections	28 (2 events) • Graffiti • QA work	29	30 (1 event) • Age Sales



Rysunek 3.13. Grafika przedstawiająca proces przeładowania strony statycznej

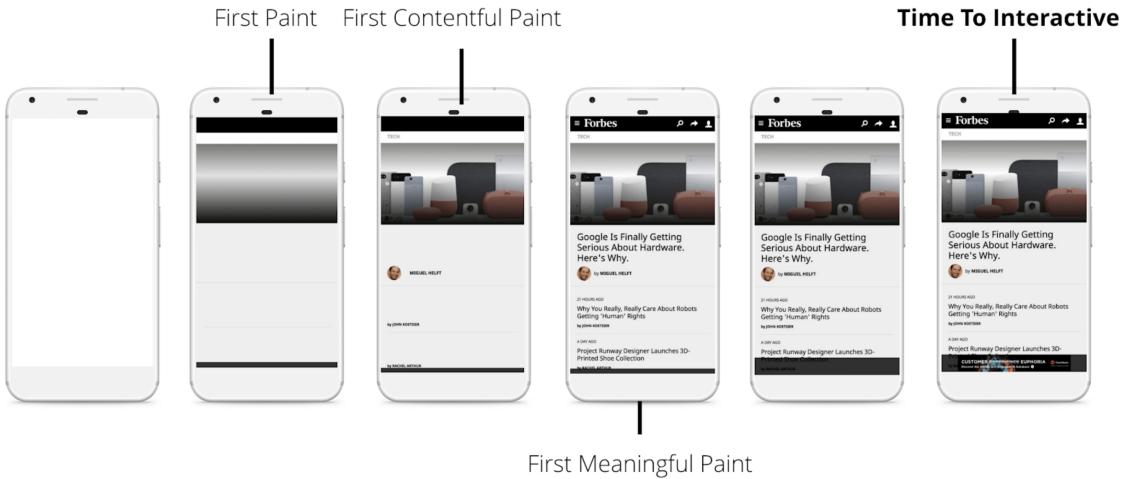
które deweloperzy, a więc także automaty, mogą nasłuchiwać. W naszym przypadku, będzie to First Meaningful Paint [28] które oznacza, kiedy pierwsza treść dostępna dla użytkownika zostaje w pełni namalowana. Kolejność zdarzeń ukazano na rysunku 3.14.

Tak więc widzimy, że w procesie interakcji ze stroną statyczną, czas jaki będzie potrzebny na przeładowanie strony to nie tylko czas kliknięcia na przycisk, ale także:

- Czas potrzebny na wysłanie zapytania z automatu do przeglądarki z żądaniem wykonania akcji.
- Czas, jaki przeglądarka potrzebuje na naciśnięcie przycisku.
- Czas wysłania zapytania do serwera.
- Czas, jaki serwer WWW potrzebuje na skonstruowanie odpowiedzi z plikiem HTML dla przeglądarki. Dla nietrywialnych stron, może to stanowić większą część czasu całej akcji. Przykładem takiego mechanizmu są strony napisane przy użyciu Ruby on Rails gdzie do skonstruowania nowej strony HTML, należy przekształcić wzorce HAML [29] na plik HTML, często po drodze wykonując zapytanie do bazy (może wielu baz?) danych.
- Czas ponownego przesłania treści nowej strony do przeglądarki.
- Czas procesowania nowego pliku HTML oraz wyrenderowanie nowej strony.

Możemy wyróżnić 3 bazowe serwisy stanowiące rdzenie całej operacji. Jest to przeglądarka, serwer HTTP oraz automat. W zależności od maszyny jaką dysponujemy, przeprowadzając takie badanie, wpływ na czas każdej z operacji pośrednich ma to co dzieje się aktualnie na komputerze. I tak, wpływ będzie miał na przykład program antywirusowy działający w tle, lub inna

3. Część praktyczna



Rysunek 3.14. Ilustracja procesu ładowania aplikacji oraz zdarzenia rejestrowane przez przeglądarkę

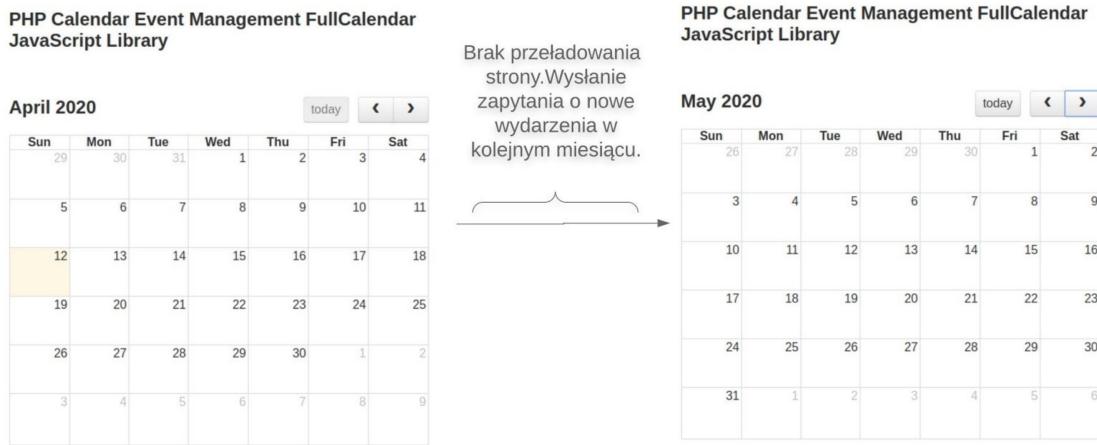
przeglądarka otworzona obok. Oczywiście im więcej rdzeni [30] posiada nasz procesor, tym mniejszy wpływ na badanie będą miały programy współdziałające. Z tego względu podczas badania należy wykonać nie tylko pojedynczy pomiar, ale pomiary należy wykonywać wielokrotnie tak, aby wyliczyć na przykład odchylenie standardowe czasów od uśrednionej wartości.

Mając już wiedzę, jak wygląda akcja naciśnięcia przycisku na stronie statycznej, możemy porównać ją do strony aplikacji SPA. Jako przykład mamy podobny kalendarz jak na stronie statycznej, tym razem z użyciem dynamicznej biblioteki JavaScript. Na rysunku 3.15 przedstawiono zmianę daty aplikacji dynamicznej z punktu widzenia użytkownika. Na rysunku 3.16 przedstawiono rozpisaną kolejność zdarzeń z punktu widzenia komputera.

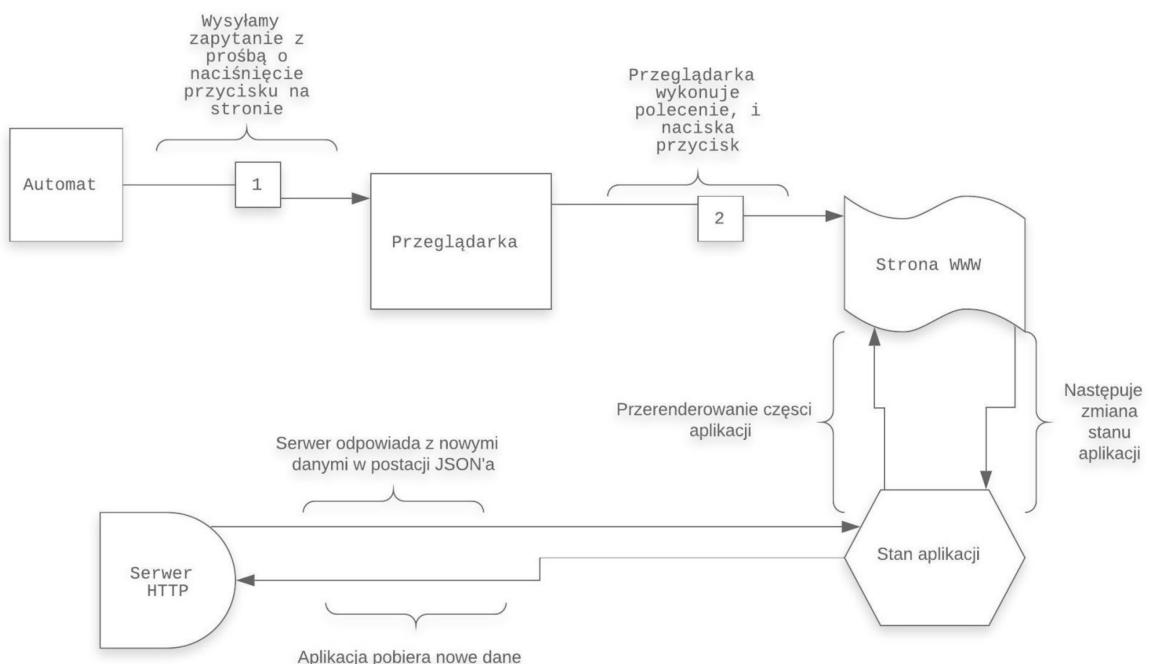
Początek procesu jest taki sam, lecz zarządcą stanu aplikacji widząc prośbę o zmianę miesiąca, wysyła zapytanie do serwera. Serwer przygotuje odpowiedź ale zamiast strony HTML, serwer odpowie ze strukturą danych typu JSON. Następnie zarządcą stanu zmieni stan i poprosi nasz framework o prze renderowanie strony. Teraz, z punktu widzenia automatu, nie ma tutaj jednoznacznego punktu końca całej operacji. Tak jak na stronie statycznej mogliśmy oczekiwać na przeładowanie strony oraz zdarzenie First Meaningful Paint, tak w tym przypadku nie istnieje żadne API pozwalające na precyzyjne określenie czy cała akcja została zakończona. Jest to bardzo kosztowny problem z którym branża informatyczna zmaga się od bardzo dawna. Wiodącym jak na razie rozwiązaniem tego problemu, które i tak nie jest idealne jest sposób testowania narzędzie Cypress.io.

- Sprawdza, czy w przeglądarce istnieją nierozerwane zapytania HTTP
- Sprawdza, czy aktualnie renderowana jest jakaś nowa zawartość
- Wykonuje serię screenshotów i porównuje, czy zawartość zmienia się w czasie.
- Periodycznie próbuje, czy zawartość DOM została już zmieniona.

3. Część praktyczna



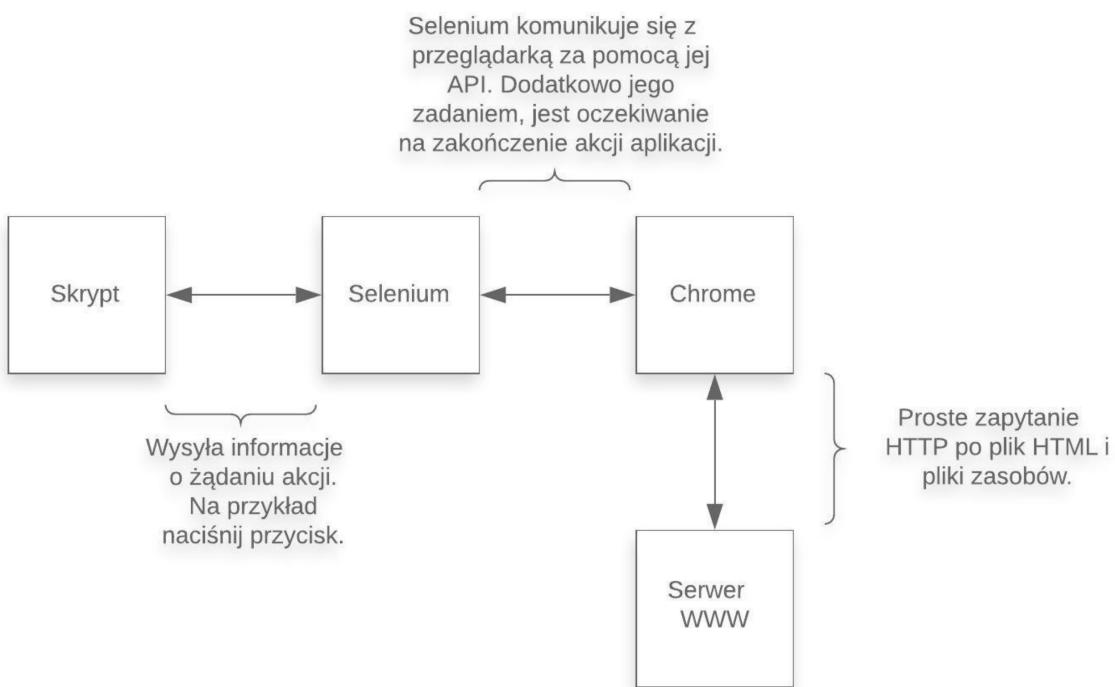
Rysunek 3.15. Ilustracja mechanizmu zmiany daty w przypadku aplikacji dynamicznej z punktu widzenia użytkownika



Rysunek 3.16. Ilustracja przedstawiająca proces zmiany treści strony w przypadku aplikacji dynamicznej z punktu widzenia komputera

Jednak nawet nowoczesne metody zastosowane w narzędziu Cypress nie są wystarczające aby zapewni, iż nawet poprawnie napisane testy zawsze zakończą się sukcesem. Zjawisko to nazywamy testami flaky, gdyż ze względu na czynniki zewnętrzne takie jak na przykład test A/B, mimo, iż sama implementacja testu jest poprawna, nadal może on zakończyć się niepowodzeniem [31]. Ze względu na opisane powyżej problemy, postanowiono zbudować moduł badania strony w oparciu o narzędzie Selenium [32]. Jest to najbardziej dojrzałe narzędzie na rynku, które stanowi bazę całego badania.

Pierwszym krokiem potrzebnym do zbudowania modułu, jest konteneryzacja samego Selenium. W tym celu, wykorzystam narzędzie docker compose [33]. Pozwala ono na uruchomienie wielu obrazów dockera, oraz połączyć je ze sobą na przykład za pomocą zwirtualizowanej sieci [34]. Bazowym obrazem będzie *selenium/hub:3.141.59-zinc*. Selenium hub, posiada w sobie selenium driver, jednak naszym zadanie jest podłączenie do niego wybranej przez nas przeglądarki. Drugim elementem jest sama przeglądarka, w tym celu wykorzystam obraz *selenium/node-chrome:3.141.59-zinc*. Ostatnim elementem jest skrypt który wykorzystując stworzony serwis Selenium, wyśle on odpowiednie informacje do przeglądarki, zagreguje dane oraz zachowa je na dysku do dalszego procesowania i analizy. Proces ten zobrazowano na rysunku 3.17.



Rysunek 3.17. Ilustracja procesu współpracy pomiędzy Selenium a przeglądarką

Skrypt do obsługi Selenium składa się z kilku głównych elementów. Pierwszym z nich jest moduł odpowiadający za oczekивание на готовность systemu. Z racji, iż Docker compose stara

3. Część praktyczna

się uruchomić wszystkie obrazy na raz, nie możemy od razu zacząć badania gdyż w pierwszej kolejności, Selenium hub musi zostać zainicjalizowane, oraz przeglądarka Chrome musi zarejestrować się w systemie. Z tego powodu stworzono funkcje które pingują Selenium oraz Serwer HTTP w oczekiwaniu na odpowiedź z kodem 200. Próbujemy połączyć się trzykrotnie z jednosekundowym odstępem czasu. Obydwa mechanizmy pracują asynchronicznie. Jeżeli którykolwiek z nich po 3 próbach nadal nie będzie mógł się połączyć, cały skrypt zwróci stosowną wiadomość która pomoże nam określić przyczynę problemu. Próbę uzyskania połączenia zobrazowano na rysunku 3.18. Z kolei na rysunku 3.19 widzimy, iż pomimo początkowych problemów z uzyskaniem połączenia, mechanizm zadziałał poprawnie i udało się nawiązać połączenie.

```
chrome_1 2020-04-13 10:48:35,508 INFO Included extra file "/etc/supervisor/conf.d/selenium.conf" during configuration
chrome_1 2020-04-13 10:48:35,508 INFO supervisord started with pid 7
selenium-hub 2020-04-13 10:48:35,197 INFO Included extra file "/etc/supervisor/conf.d/selenium-hub.conf"
selenium-hub 2020-04-13 10:48:35,198 INFO supervisord started with pid 7
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO Waiting for Web Server
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO Waiting for Selenium Server ← Wysyła informacje
master_web_server_1 2020-04-13 10:48:35,200 INFO 172.18.0.5 - - [13/Apr/2020:10:48:35 +0000] "GET / HTTP/1.1" 200 470 "-" "axios/0.19.2" "-"
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO Fail nr 1/3 Error: Error: connect ECONNREFUSED 172.18.0.2:4444 ←
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO at connect to page (/app/src/utils.js:23:11)
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO at processTicksAndRejections (internal/process/task_queues.js:97:5)
selenium_benchmark_1 2020-04-13 10:48:35,200 INFO ✓ Connected successfully to Web Server!
selenium-hub 2020-04-13 10:48:36,200 INFO spawned: 'selenium-hub' with pid 10
selenium-hub 2020-04-13 10:48:36,200 INFO Starting Selenium Hub with configuration:
selenium-hub 2020-04-13 10:48:36,209 INFO success: selenium-hub entered RUNNING state, process has stayed
Kały wpis ma jasno określone źródło z którego pochodzi
Sukces połączenia do serwera HTTP
Widzimy próbę połączenia
Serwer HTTP odpowiada z kodem 200
Nie udało się połączyć jednak do serwera selenium
```

Rysunek 3.18. Wycinek wpisów skryptu przeprowadzającego badanie w środowisku docker-compose. Ilustruje on inicjalizację skryptu oraz mechanizm uzyskiwania połączenia pomiędzy Skryptem - Przeglądarką - Selenium

```
chrome_1 2020-04-13 10:48:35,508 INFO Included extra file "/etc/supervisor/conf.d/selenium.conf" during configuration
chrome_1 2020-04-13 10:48:35,508 INFO supervisord started with pid 7
selenium-hub:208 2020-04-13 10:48:35,197 INFO Included extra file "/etc/supervisor/conf.d/selenium-hub.conf"
selenium-hub:208 2020-04-13 10:48:35,198 INFO supervisord started with pid 7
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO Waiting for Web Server
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO Waiting for Selenium Server
master_web_server_1 172.18.0.5 - - [13/Apr/2020:10:48:35 +0000] "GET / HTTP/1.1" 200 470 "-" "axios/0.19.2" "-"
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO Fail nr 1/3 Error: Error: connect ECONNREFUSED 172.18.0.2:4444
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO     at connect to page (/app/src/utils.js:23:11)
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO     at processTicksAndRejections (internal/process/task_queues.js:97:5)
selenium-benchmark_1 2020-04-13 10:48:35,200 INFO ✓ Connected successfully to Web Server!
selenium-hub:208 2020-04-13 10:48:36,200 INFO spawned: 'selenium-hub' with pid 10
selenium-hub:208 2020-04-13 10:48:36,200 INFO Starting Selenium Hub with configuration:
selenium-hub:208 2020-04-13 10:48:36,209 INFO success: selenium-hub entered RUNNING state, process has stayed
      up for > 0 seconds (startsecs)

Widzimy próbę połączenia
Wysyła informacje
do serwera HTTP
Nie udaje się połączyć jednak do serwera selenium
Każdy wpis ma jasno określone źródło z którego pochodzi
Sukces połączenia do serwera HTTP
Serwer HTTP odpowiada z kodem 200
```

Rysunek 3.19. Wycinek skryptu ukazujący uzyskanie połączenia do Selenium pomimo początkowych problemów

Następnie łączymy się do serwera WWW. Odczytujemy z niego listę dostępnych aplikacji.

Dzięki temu, nie musimy zmieniać implementacji kodu testu przy dodawaniu kolejnych implementacji lub badań. Przed każdym badaniem chcemy jak najbardziej odizolować od siebie testy, z tego też powodu wysyłamy komendę przeładowania strony oraz wyczyszczenia pamięci cache oraz wołamy komendę garbage collection [35]. Wreszcie, mając już tak przygotowane środowisko oraz połączenie do Selenium możemy przejść do faktycznej implementacji testu badania.

Obsługa testów została zaprojektowana tak, aby można było dodać dowolną ilość testów oraz dowolnie sprecyzować ilość powtórzeń. Powtórzenia są dla nas istotne gdyż musimy wykonać ten sam test wielokrotnie w celach porównania wyników i analizy różnic w czasach w poszczególnych obiegach. Następnie zostanie omówiona struktura badania którą napisano stricte na potrzeby analizy czasów akcji aplikacji w frameworkach aplikacji SPA. Test ten otwiera stronę WWW, i dla każdej kolejnej funkcji na początku dopisuje 1000 elementów (tak, aby kolejne komendy miały materiał do pracy) i po zakończeniu testu danej funkcji czyści wszystkie wiersze. W badaniu następują kolejne zdarzenia:

- Wykonuje akcje dopisania 1000 elementów na początku listy, i powtarza ją 100 razy.
- Wykonuje akcje dopisania 1000 elementów na końcu listy, i powtarza ją 100 razy.
- Wykonuje akcje zamiany wszystkich wartości na liście, i powtarza ją 100 razy.
- Wykonuje akcje podmiany wartości dla 500 wierszy, i powtarza ją 100 razy.
- Wykonuję akcję zamiany wiersza 0 z 1, i powtarzam ją 100 razy.
- Wykonuję akcję usunięcia pierwszego w kolejności wiersza, i powtarzam ją 100 razy.
- Wykonuję akcję usunięcia wszystkich wierszy, i powtarzam ją 100 razy.

Każda akcja jest stworzona jako funkcja, która w momencie uruchomienia tworzy wpis w przeglądarce używając wysoce precyzyjnego API wbudowanego w przeglądarkę jakim jest Performance API [36]. Wpis taki, posiada precyzję sięgającą 5 mikrosekund [37]. Tak więc dla jednego obiegu takiego testu mamy 700 wpisów. Całą operację przedstawiono na rysunku 3.20.

Po zakończeniu testu (rysunek 3.21), skrypt komunikuje się z Selenium i pobiera listę wpisów zarchiwizowanych po badaniu. Wpisy te zostają zapisane w postaci pliku JSON w folderze results. Dla każdej badanej aplikacji i każdego testu, zostaną zapisane 3 pliki wynikowe, po jednym dla każdego z dostępnych logów.

3.3.6 Moduł analizy

W module tym, zajmiemy się przygotowaniem zebranych danych i skondensowaniem ich do postaci pojedynczego pliku JSON zawierającego istotne dla nas informacje. Skrypt ten jest napisany w sposób generyczny, tak więc dla każdego nowego rodzaju testu powinniśmy przygotować nowy skrypt parsujący. Podstawowa struktura pliku jest jednak ujednolicona i pozwala nam na generyczne procesowanie pliku. Struktura danych przedstawiona została na grafice 3.22:

Dla każdego z dostępnych wpisów, przygotowano po jednej funkcji pozwalającej na ekstrakcję wpisów. Do celów badania, jedynie wpisy przeglądarki zawierały istotne dane (rysunek 3.23).

3. Część praktyczna

```
chrome_1 | 10:48:38.438 INFO [RemoteSession$Factory.lambda$open$0] Skrypt
selenium_benchmark_1 | ★ Driver created
selenium_benchmark_1 | ○ Get list of links
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET /"
selenium_benchmark_1 | ○ Found: [ 'angular2', 'react', 'vue' ]
selenium_benchmark_1 | ✨ Start Benchmark for angular2
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET //"
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET //"
(master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET //")
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET //"
Kit/537.36 (KHTML, like Gecko) HeadlessChrome/79.0.3945.117 Safari/537.36"
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:48:38 +0000] "GET //"
it/537.36 (KHTML, like Gecko) HeadlessChrome/79.0.3945.117 Safari/537.36"
selenium_benchmark_1 | result_foler ./results/angular2/single_action/0
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:50:18 +0000] "GET //"
rome/79.0.3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/angular2/single_action/1
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:51:55 +0000] "GET //"
rome/79.0.3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/angular2/single_action/2
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:53:34 +0000] "GET //"
rome/79.0.3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/angular2/single_action/3
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:55:10 +0000] "GET //"
rome/79.0.3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/angular2/single_action/4
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:10:56:45 +0000] "GET //"
rome/79.0.3945.117 Safari/537.36" "-"
```

Skrypt pobiera listę dostępnych aplikacji

Web serwer odpowiada z plikiem index.html

Następuje rozpoczęcie badania

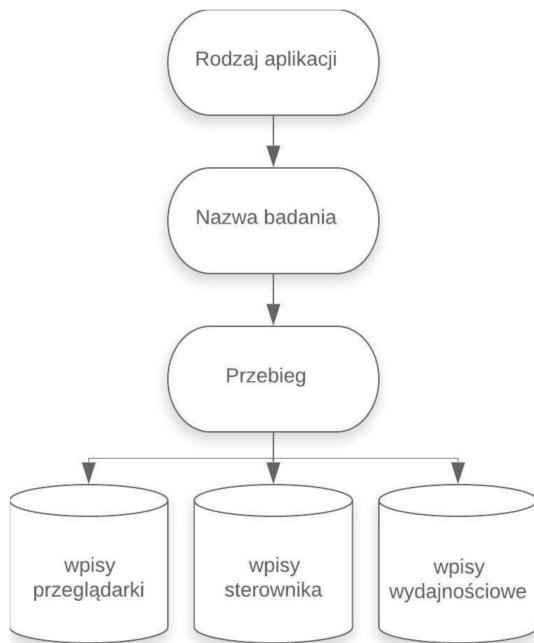
Wyniki poszczególnych badań zostają zapisane na dysku

Rysunek 3.20. Grafika przedstawia rozpoczęcie badania

```
selenium_benchmark_1 | result_foler ./results/vue/single_action/3
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:11:10:55 +0000] "GET
3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/vue/single_action/4
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:11:12:28 +0000] "GET
3945.117 Safari/537.36" "-"
selenium_benchmark_1 | result_foler ./results/vue/simple_benchmark/0
master_web_server_1 | 172.18.0.4 - - [13/Apr/2020:11:12:30 +0000] "GET
3945.117 Safari/537.36" "-"
selenium_benchmark_1 | 🎉 Finished benchmark successfully for vue
selenium_benchmark_1 | ❤️ Finished all benchmarks!
selenium_benchmark_1 | 💀 Killing driver
chrome_1 | 11:12:30.877 INFO [ActiveSessions$1.onStop] - Rem
```

Rysunek 3.21. Grafika przedstawia skrypt zakańczający badanie po zapisaniu zebranych danych na dysk - widzimy, że ważnym elementem jest zamknięcie połączenia do Selenium

3. Część praktyczna



Rysunek 3.22. Ilustracja struktury wyniku badania na które zostanie poddane dalszej obróbce

Wynikiem całego skryptu jest pojedyńczy plik index.json zawierający taki sam podział na aplikacje, badania i przebiegi. Atomowym wpisem dla każdego badania jest pomiar. Pomiar zawiera w sobie nazwę akcji jaka została wykonana oraz zmierzoną wartość.

```
{  
    "name": "measure_replace_all_click-1586714207563}",  
    "entryType": "measure",  
    "startTime": 61.564999996335246,  
    "duration": 49.28500000096392  
},  
[  
    {"  
        "name": "measure_updateNRows_click-1586714207631}",  
        "entryType": "measure",  
        "startTime": 130.34499999776017,  
        "duration": 48.45000000204891  
],  
}
```

Rysunek 3.23. Grafika przedstawiająca wpis zebranego pomiaru

3.4 Analiza wyników

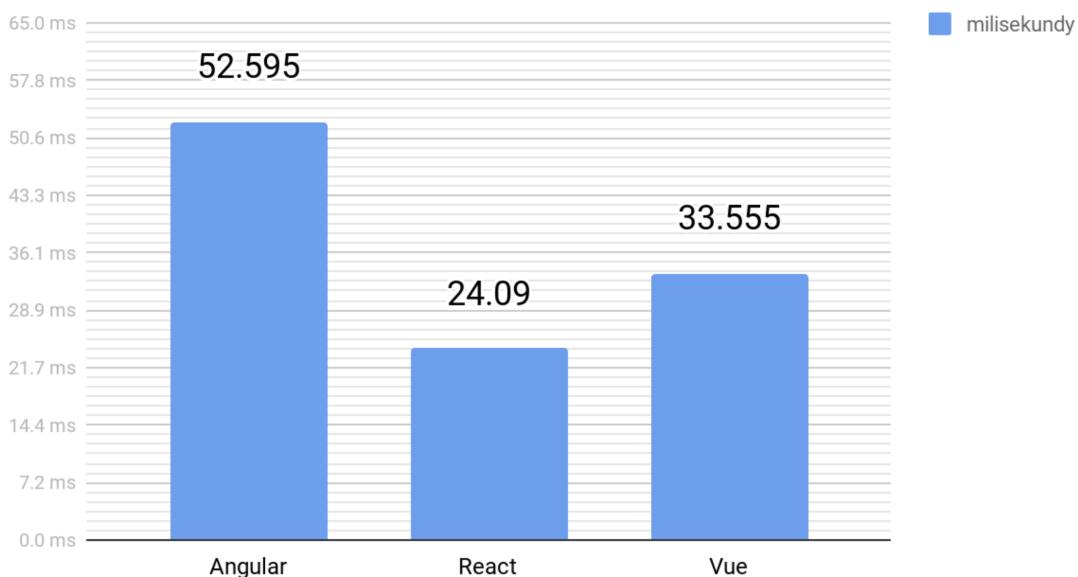
Finalny zestaw badań jakie wykonano objęły następujące parametry:

- Test wykonano dla 20 powtórzeń dla każdej aplikacji, czyli łącznie 60 powtórzeń
- Każda akcja została powtórzona 100 razy w ramach jednego testu, czyli 2000 razy w ramach pojedynczego badania.
- Łączna liczba punktów pomiarowych wyniosła 42000 pomiarów.
- Na potrzeby pracy przyjęto błąd pomiarowy równy ± 1 milisekunda.

3.4.1 Czas załadowania aplikacji

Pierwsze badanie polega na zmierzeniu czasu potrzebnego do rozpoczęcia malowania strony. Oczekujemy tutaj na zdarzenie first paint [38]. Wyniki zaprezentowano na rysunku 3.24.

Pomiar czasu potrzebnego do rozpoczęcia malowania strony



Rysunek 3.24. Diagram kolumnowy ukazujący wynik badania czasu potrzebnego do rozpoczęcia malowania strony

Każdy pomiar miał miejsce w czasie odświeżenia strony. Podane wartości są średnimi wartościami z 20 przeładowań. Widzimy ogólną różnicę w otrzymanych wynikach. React jest o 9.465 milisekundy szybszy, czyli o 39.290% szybszy od Vue. Vue jest także o 19.04 milisekundy szybsze od Angulara co stanowi 56.743% lepszy wynik.

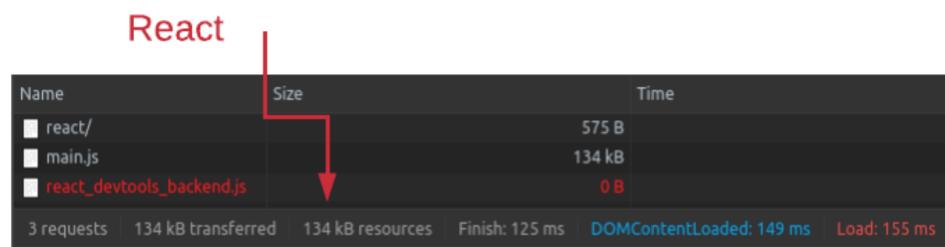
Główny wpływ na otrzymane wyniki będą mieć waga narzędzia oraz ilość kodu który musi być pobrany oraz uruchomiony zanim narzędzie zacznie renderować wymaganą treść. Dodatkowe informacje możemy wyłuskać patrząc na ilość bajtów pobranych przez przeglądarkę co

3. Część praktyczna

ukazano na rysunku 3.25 - Angular2, rysunek 3.26 - React oraz rysunek 3.27 - Vue.



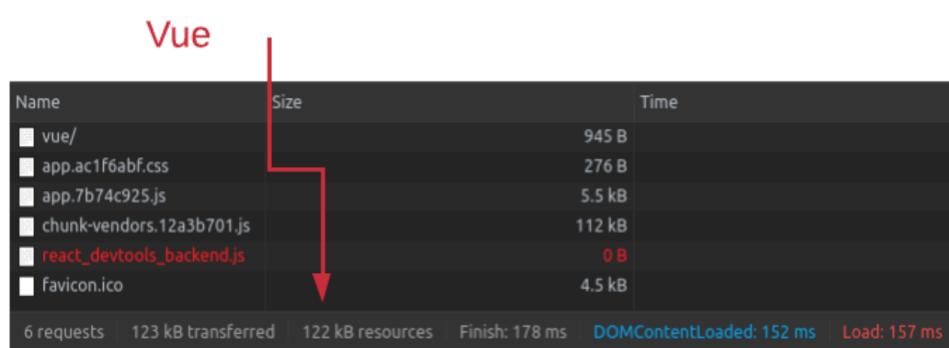
Rysunek 3.25. Grafika ukazująca rozmiar plików zasobów dla aplikacji Angular2



Rysunek 3.26. Grafika ukazująca rozmiar plików zasobów dla aplikacji React

Wniosek nasuwa się następujący. React który potrzebuje 134 kB danych potrzebuje 24.09 milisekund na wyrenderowanie pierwszego piksela. Vue który pobiera 122 kB czyli o 8.955% mniej danych, ładuje się w czasie 33.555 milisekund. Wskazuje to jednoznacznie, że kod Reacta pomimo większej wagi, wykonuje mniej operacji przed pierwszym renderowaniem. Na ostatnim miejscu w tym badaniu znajduje się Angular2 który potrzebuje 185 kB oraz 52.595 milisekund na załadowanie pierwszego piksela.

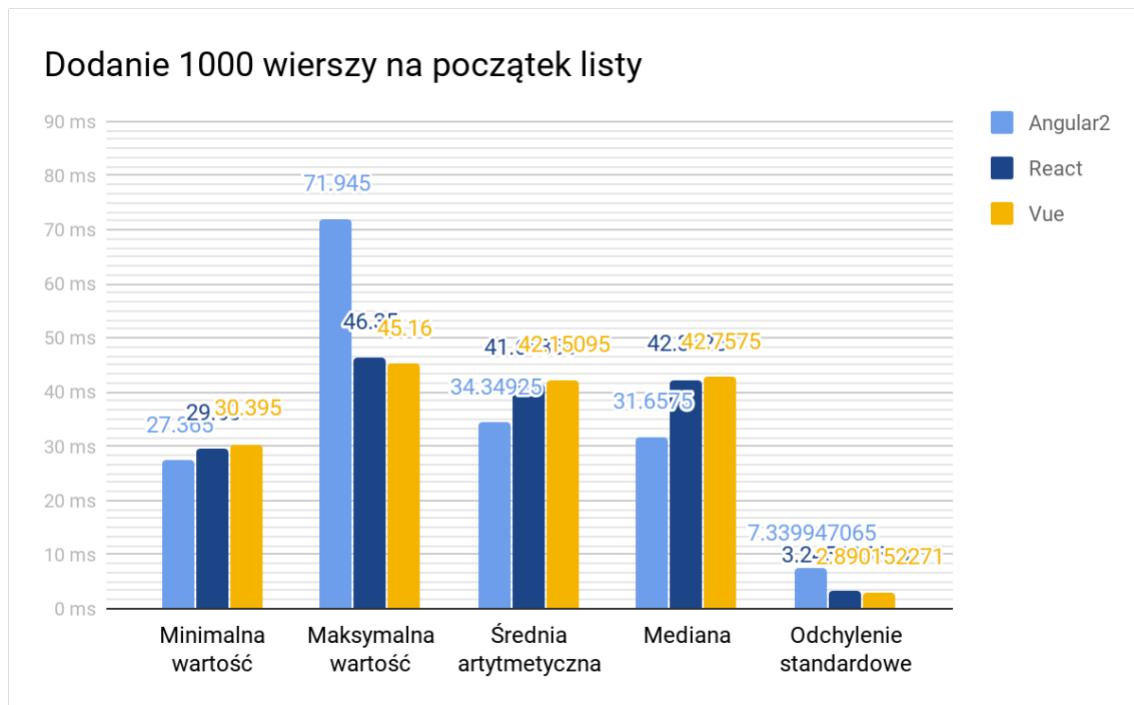
3. Część praktyczna



Rysunek 3.27. Grafika ukazująca rozmiar plików zasobów dla aplikacji Vue

3.4.2 Dodanie 1000 wierszy na początek listy

W badaniu tym do istniejącej listy 1000 elementów dodajemy kolejne 1000 elementów na początek listy. Akcję tą powtarzamy 100 krotnie, czyli zaczynając od 1000 elementów aż do 101 000 kończąc. Problem badawczy tego zadania opisano w rozdziale Projekt testu wydajności. Wyniki badania zobrazowano na rysunku 3.28.

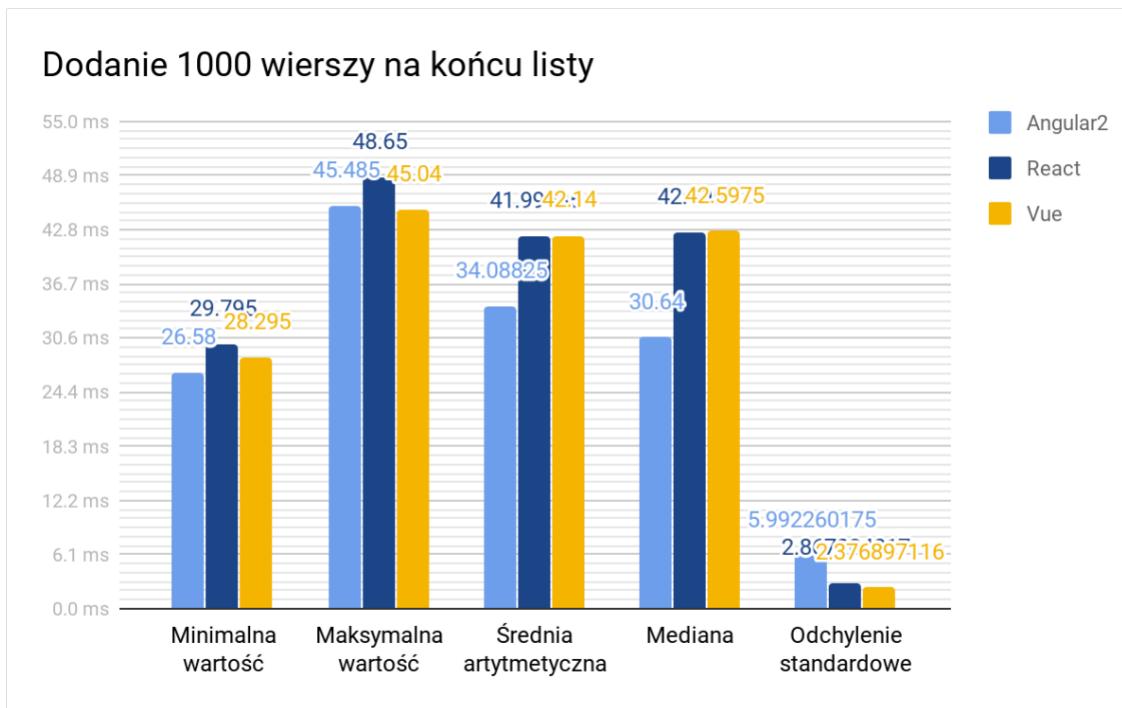


Rysunek 3.28. Diagram kolumnowy obrazujący wynik badania pomiaru czasu dodania 1000 elementów na początek listy

Z wykresu wartości, możemy odczytać, iż wartości brzegowe Angular2 wyznaczają dolną oraz górną granicę otrzymanych wartości czasu. Także odchylenie standardowe jest ponad dwukrotnie wyższe niż kolejny po nim React. Jednak średnia i mediana wskazują, że w uśrednionym przypadku, dopisywanie nowych elementów do listy jest najszybsze w frameworku Angular2. React oraz Vue w przypadku uśrednionym, plasują się na drugiej pozycji, gdyż ich wartości mieścią się w wartości błędów pomiaru. Najmniejsze rozproszenie wyników prezentuje Vue, którego średnia i mediana są bardzo zbliżone, a odchylenie standardowe najniższe spośród badanych narzędzi, co świadczy o stabilności rozwiązania.

3.4.3 Dodanie 1000 wierszy na końcu listy

Dopisywanie elementów do końca listy jest mało obciążającym zadaniem dla naszych narzędzi. Nie musimy przesuwać żadnych elementów, jedynie wyrenderować nowo dodane. Oczekujemy, iż wyniki (rysunek 3.29) będą miały małe odchylenie standardowe (gdyż ilość poprzednich elementów nie powinna mieć wpływu na dodanie nowych).

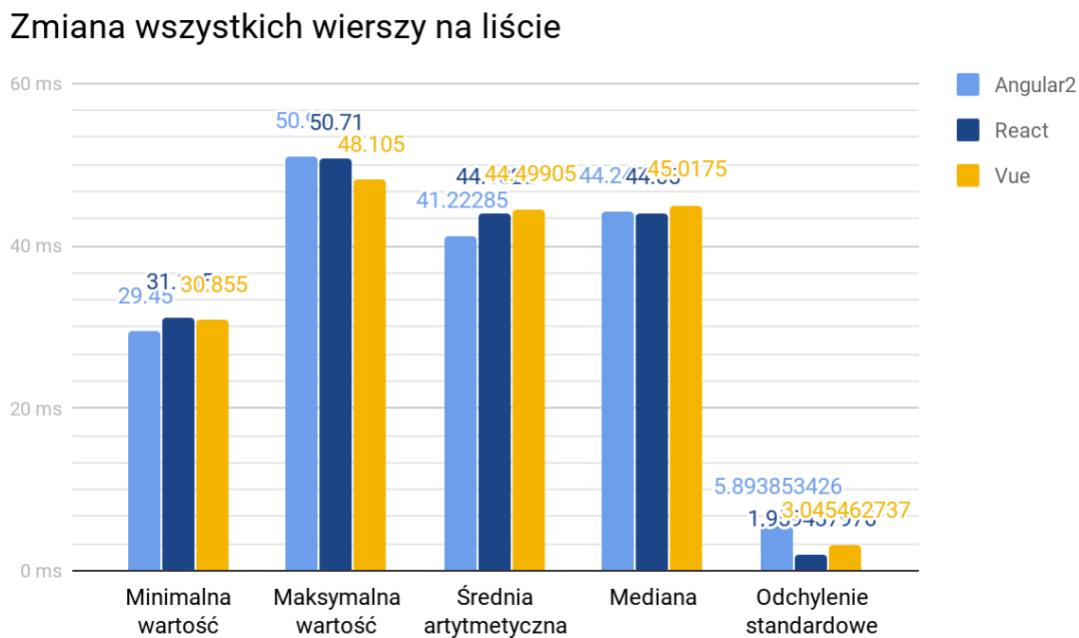


Rysunek 3.29. Diagram kolumnowy obrazujący wynik badania pomiaru czasu dodania 1000 elementów na koniec listy

Pierwszym, co można zauważyć, jest fakt iż dla przypadku minimalnej wartości oraz maksymalnej wartości, wyniki pomiędzy narzędziami są zbliżone. Jednak warto zaznaczyć, iż React jest naj wolniejszy w skrajnych przypadkach. W uśrednionym przypadku, Angular2 jest najszybszym narzędziem bijąc React oraz Vue o 8 milisekund w przypadku średniej a w przypadku mediany o aż 12 milisekund. Jednak warto zaznaczyć, iż odchylenie standardowe tego badania wskazuje, że Angular2 jest najmniej przewidywalnym z całej trójki. Ciekawą sytuację natomiast możemy zaobserwować dla wyników średniej oraz mediany pomiędzy React oraz Vue. Wyniki są bardzo zbliżone, jednak odchylenie standardowe wskazuje, że to Vue jest stabilniejszym rozwiązaniem.

3.4.4 Zamiana wszystkich wierszy na liście

Badanie to wskazuje, jak szybko jesteśmy w stanie przerenderować listę 1000 elementów. Wyniki ukazano na rysunku 3.30.

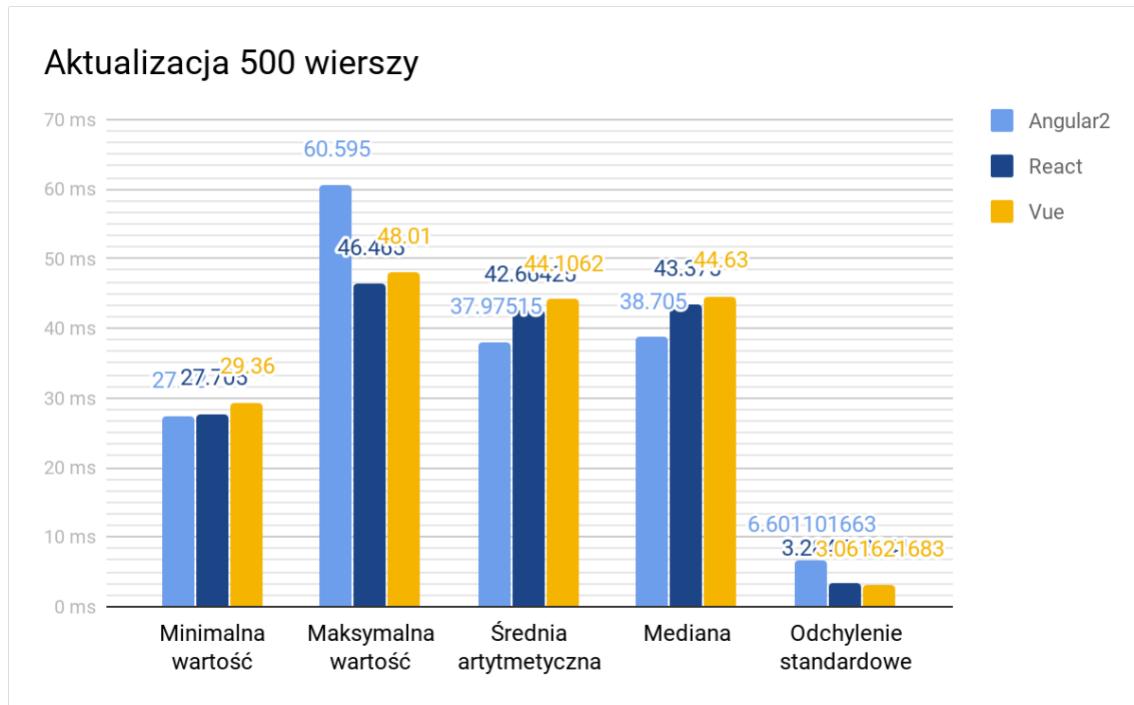


Rysunek 3.30. Diagram kolumnowy obrazujący wynik badania pomiaru czasu zmiany wszystkich wartości listy dla 1000 elementów

Minimalna wartość jest bardzo zbliżona dla każdego z badanych narzędzi. Maksymalna wartość wskazuje nieznaczną przewagę około 2 milisekund na korzyść Vue. Średnia arytmetyczna wskazuje nam, iż Angular2 jest najszybszym narzędziem, jednak odchylenie standarde we wskazuje, iż jest także najbardziej niestabilnym. W tym badaniu React udało się uzyskać odchylenie poniżej 2 milisekund, co jest wynikiem bardzo dobrym.

3.4.5 Aktualizacja 500 wierszy

W badaniu tym, aktualizujemy tylko połowę wyrenderowanych wierszy. Wyniki ukazano na rysunku 3.31. Wartości minimalne są bardzo zbliżone dla Reacta oraz Angular2. Vue odbiega



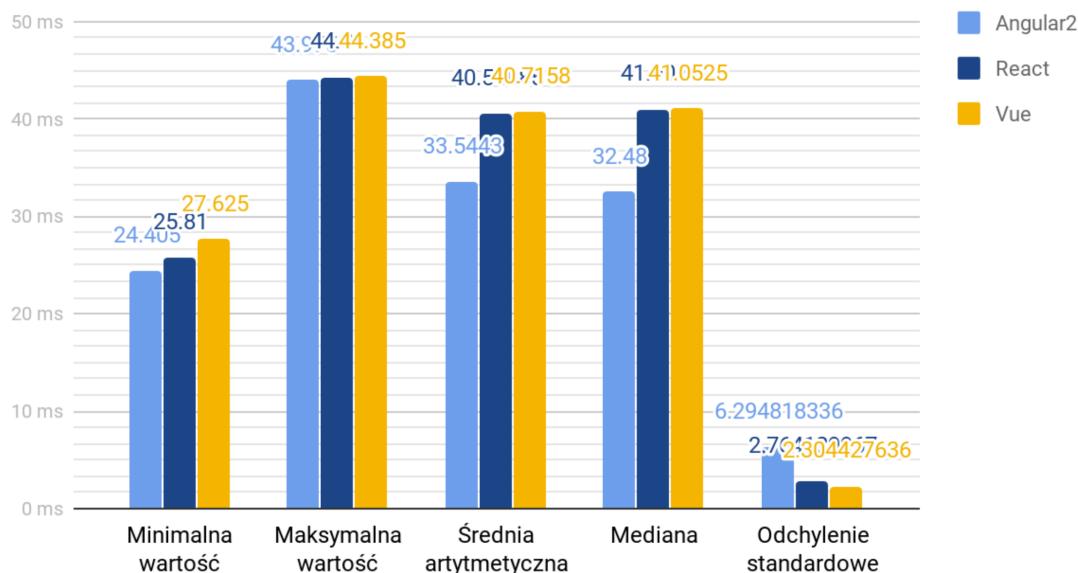
Rysunek 3.31. Diagram kolumnowy obrazujący wynik badania pomiaru czasu zmiany wartości dla 500 elementów listy

tylko nieznacznie mniej niż 2 milisekundy. W przypadku wartości maksymalnych, Angular2 znaczco odbiega od reszty narzędzi wykazując różnicę 12 milisekund do kolejnego na liście Vue. W uśrednionym przypadku, widzimy zgodność średniej arytmetycznej i mediany, które wskazują, iż Angular2 jest najszybszym narzędziem. Ponownie już odchylenie standardowe wskazuje iż jest także najmniej stabilnym czasowo narzędziem.

3.4.6 Zamiana miejscami dwóch wierszy

Zamiana miejscami dwóch wierszy jest najmniej wymagającym badaniem z całej pracy. Z tego powodu oczekujemy niskich wartości odchylenia standardowego. Wyniki ukazano na rysunku 3.32.

Zamiana miejscami dwóch wierszy

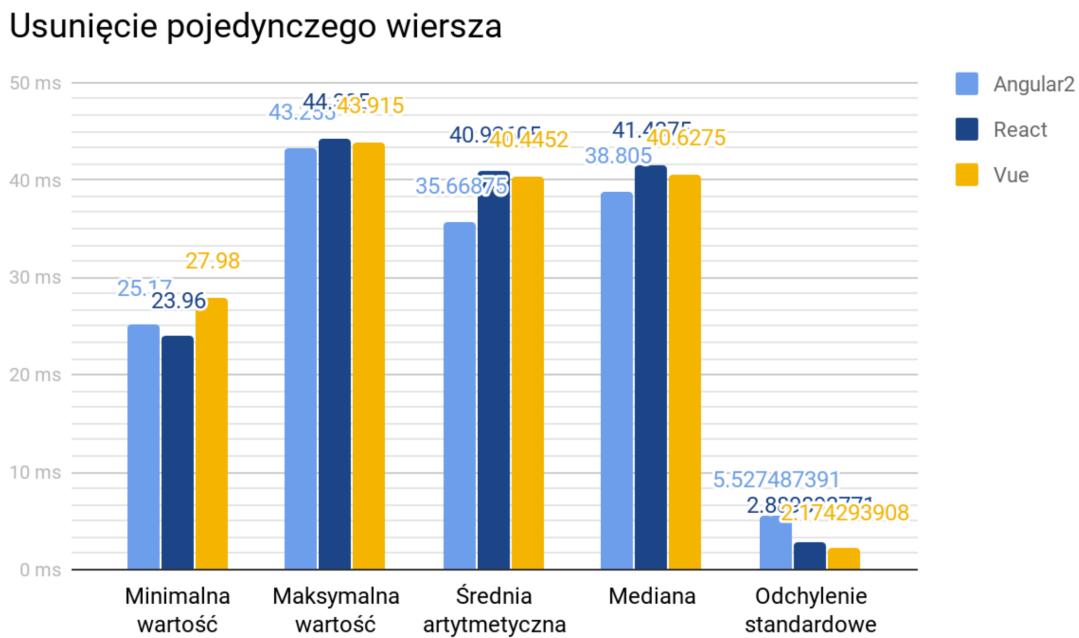


Rysunek 3.32. Diagram kolumnowy obrazujący wynik badania pomiaru czasu zamiany miejscami dwóch wartości na liście elementów

I tak wartość minimalna wskazuje, iż Angular2 jest najszybszy. Wartości maksymalne są prawie identyczne. Są one w granicy błędu pomiarowego. Średnia arytmetyczna i mediana zgodnie wskazują, iż Angular2 jest najszybszy w średnim przypadku, znowu z zastrzeżeniem, gdyż odchylenie standardowe wynosi aż 6.3 milisekundy.

3.4.7 Usunięcie pojedynczego wiersza

Badanie to polega na usunięciu pierwszego w kolejności wiersza na początku listy. Powoduje to przesunięcie całej listy o jeden wiersz. Wyniki ukazano na rysunku 3.33.

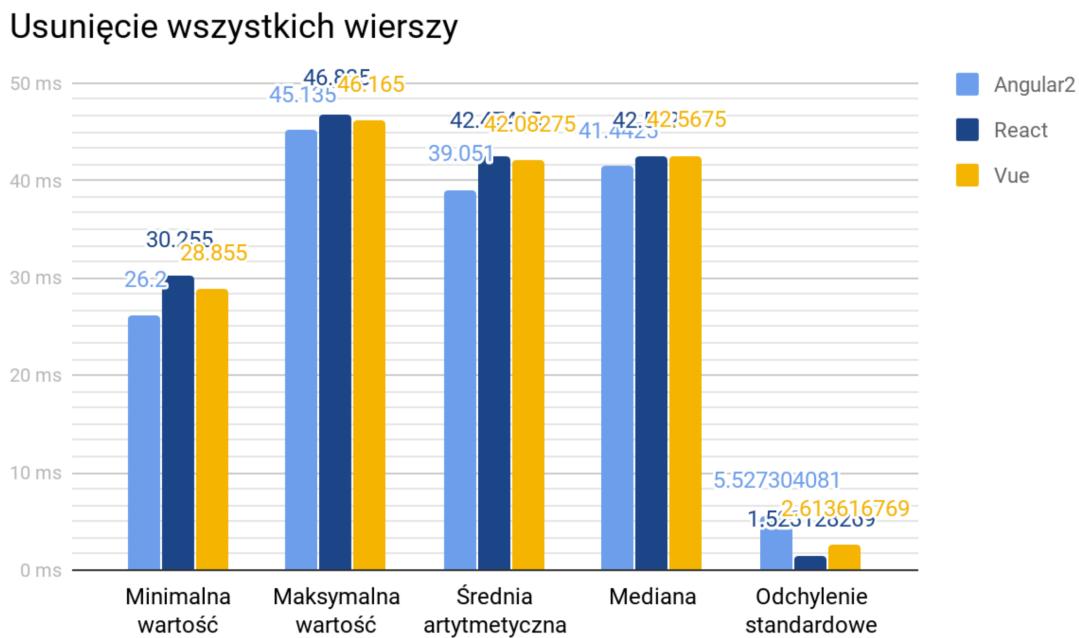


Rysunek 3.33. Diagram kolumnowy obrazujący wynik badania pomiaru czasu usunięcia pojedynczej wartości z listy elementów

Po raz pierwszy obserwujemy Reacta jako najszybsze narzędzie w przypadku wartości minimalnej. Wartości maksymalne mieścią się w granicy błędu pomiarowego dla każdego z badanych narzędzi. Średnia arytmetyczna i mediana ponownie wskazują na dominację Angulara 2. Jednak ponownie odchylenie standardowe wskazuje na znaczące wahania wartości wokół średniej wartości dla Angular2. Vue oraz React poza wartością minimalną mieścią się w granicy błędu pomiarowego.

3.4.8 Usunięcie wszystkich wierszy

Ostatnie badanie skupia się na usunięciu wszystkich wierszy. Spowoduje to przerenderowanie całej listy. Wyniki ukazano na rysunku 3.34.



Rysunek 3.34. Diagram kolumnowy obrazujący wynik badania pomiaru czasu usunięcia wszystkich wartości z listy elementów

Dla minimalnej wartości Angular2 wykazuje 2 milisekundy przewagi nad kolejnym Vue które z kolei wykazuje około 1.5 milisekundy przewagi nad React. Wartości maksymalne dla każdego badanego narzędzia mieszczą się w granicy błędu pomiarowego, z delikatnym wskazaniem przewagi dla Angular2. W średnim przypadku Angular2 jest ponownie najszybszy, lecz odchylenie standardowe wskazuje prawie 5.5 milisekundy zabiegu. W tym przypadku należy zauważyć, iż React ponownie uzyskał odchylenie standardowe na poziomie 1.5 milisekundy.

4. Podsumowanie

Celem pracy było stworzenie narzędzia pozwalającego na pomiar akcji aplikacji zachodzących już po załadowaniu strony oraz pomiar różnych interakcji ze stroną na przykładzie trzech wiodących rozwiązań na rynku. Stworzenie narzędzia było niemałym wyzwaniem, gdyż problem pomiaru dynamicznych aplikacji nie jest problemem trywialnym i wymaga szeregu narzędzi współpracujących ze sobą. Powoduje to także, iż sam pomiar będzie zanieczyszczony przez niedokładności narzędzie Selenium. Jesteśmy w stanie jednak przeciwdziałać niektórym czynnikom mającym negatywny wpływ na pomiar, dzięki zastosowaniu warstwy izolacji systemu badającego od faktycznego systemu operacyjnego hosta badania dzięki użyciu platformy Docker. Także poprawne przygotowanie badania na przykład poprzez przeładowanie strony i wyczyszczenie pamięci podręcznej przeglądarki poprawia stabilność i dokładność testów. Istotnym podczas testów jest wielokrotny pomiar wartości tak, abyśmy mogli wyliczyć graniczne wartości pomiarów. Dzięki temu i wiedzy uzyskanej podczas studiów możemy przeprowadzić poprawną analizę wyników.

Same badania wykazały, iż Angular2 najczęściej jest najszybszym narzędziem, jednak bardzo często odchylenie standardowe wskazywało, iż także istnieje duży rozrzut wartości wokół średniej. React jest najbardziej stabilnym rozwiązaniem, gdyż w dwóch badaniach uzyskał odchylenie standardowe na poziomie poniżej 2 milisekund. W reszcie przypadków, różnica pomiędzy Vue i Reactem mieściła się w granicy błędu pomiarowego.

Podsumowując -najważniejszym wnioskiem który udało się potwierdzić dzięki przeprowadzonym badaniom jest wykazanie, iż różnica pomiędzy przedstawionymi rozwiązaniami nie przekracza 20 milisekund dla pojedynczego zadania. Jest to istotny argument podczas dyskusji nad przewagą konkretnych narzędzi pomiędzy sobą. Jest to wniosek zgodny także z pracami cytowanych autorów, iż najistotniejsze jest środowisko programistyczne dostarczone wraz z narzędziem oraz popularność narzędzia która bezpośrednio ma wpływ na przykład na ilość informacji dostępnych w internecie na temat częstych problemów niżeli wydajność pojedynczego narzędzia w skali mikro.

Bibliografia

- [1] JSON.org, <https://www.json.org/>
- [2] Peleke Sengstacke *JavaScript Transpilers: What They Are and Why We Need Them*, Kwiecień 25, 2016
<https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them>
- [3] Sławomir Kołodziej *What Are Single Page Applications? What Is Their Impact on Users' Experience and Development Process?*, 3 Lipiec 2019
<https://www.netguru.com/blog/what-are-single-page-applications>
- [4] Eric Molin, *Comparison of Single-Page Application Frameworks*, Instytut KTH w Sztokholmie
<https://pdfs.semanticscholar.org/fa9f/f75f32de61cddafa8805ea433d4d8a0e20da.pdf>
- [5] <https://octoverse.github.com/>
- [6] Maja Nowak, Reasons, *Why Vue.js Is Getting More Traction Every Month*, 19 Grudzień 2018,
<https://www.monterail.com/blog/reasons-why-vuejs-is-popular>
- [7] <https://github.com/angular/angular/blob/master/CHANGELOG.md>
- [8] <https://github.com/facebook/react/blob/master/CHANGELOG.md>
- [9] *Probabilistyczny opis błędu jako podstawa definiowania niepewności pojedynczego wyniku pomiaru*
<http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-article-BSW4-0034-0011>
- [10] Google Developers, *Lighthouse*
<https://developers.google.com/web/tools/lighthouse>
- [11] Kyle Simpson *You Don't Know JS: Async & Performance*
http://cdn.lxqnsys.com/05_You_Don't_Know_JS_Async_&_Performance.pdf

- [12] Jadwiga Kalinowska, Beata Pańczyk, *Porównanie narzędzi do tworzenia aplikacji typu SPA na przykładzie Angular2 i React*, Politechnika Lubelska, Instytut Informatyki
<http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-5c6271b4-27e3-42d0-9762-a240dc3a9973>
- [13] Paul Lewis, *Google Developers, Rendering Performance*,
<https://developers.google.com/web/fundamentals/performance/rendering>
- [14] *ReactJS, Lists and Keys*,
<https://reactjs.org/docs/lists-and-keys.html>
- [15] *ReactJS, Virtual DOM and Internals*,
<https://reactjs.org/docs/faq-internals.html>
- [16] ReactJS, Reconciliation,
<https://reactjs.org/docs/reconciliation.html>
- [17] GNU.org, *Makefile*,
https://www.gnu.org/software/make/manual/html_node/Introduction.html
- [18] NpmJs.com, *npm | build amazing things*,
<https://www.npmjs.com/>
- [19] ReactJS, *Optimizing Performance*,
<https://reactjs.org/docs/optimizing-performance.html>
- [20] magnifier.pl, *Konteneryzacja - czym jest i dlaczego staje się tak popularna?*, 24 Październik. 2019,
<https://magnifier.pl/konteneryzacja-docker-kubernetes/>
- [21] MDN web docs, *Promise - JavaScript*,
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise
- [22] MDN web docs, *What is a web server?*, 18 Stycznia 2019,
https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server
- [23] Docker Docs, *Docker*,
<https://docs.docker.com/get-started/#docker-concepts>

- [24] Nginx, *Nginx for Windows*,
<http://nginx.org/en/docs/windows.html>
- [25] Nginx, *Linux packages*,
http://nginx.org/en/linux_packages.html
- [26] Wiki Alpine Linux, *Docker*,
<https://wiki.alpinelinux.org/wiki/Docker>
- [27] Docker Hub, *Ubuntu*,
https://hub.docker.com/_/ubuntu
- [28] Meggin Kearney, Addy Osmani, Kayce Basques, Jason Miller, *Measure Performance with the RAIL Model | Web Fundamentals*, 12 Luty. 2019,
<https://developers.google.com/web/fundamentals/performance/rail>
- [29] Jesus Castello, *Ruby Templating Engines: ERB, HAML & Slim - RubyGuides*,
<https://www.rubyguides.com/2018/11/ruby-erb-haml-slim/>
- [30] Whatsabyte, *What Are Threads in a Processor?*, 24 Sierpień 2018,
<https://whatsabyte.com/blog/processor-threads/>
- [31] Dimiter Petrov, *A tale of flaky Cypress tests*, 24 Październik 2019,
<https://dimiterpetrov.com/blog/a-tale-of-flaky-cypress-tests/>
- [32] Selenium Dev, *The Selenium project and tools*,
https://www.selenium.dev/documentation/en/introduction/the_selenium_project_and_tools/
- [33] Docker Docs, *Overview of Docker Compose*,
<https://docs.docker.com/compose/.>
- [34] Docker Documentation, *Networking in Compose*.
<https://docs.docker.com/compose/networking/>
- [35] MDN - Mozilla, *Memory Management*, 4 Marca 2020,
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management
- [36] MDN - Mozilla, *Performance - Web APIs*, 30 Styczeń 2020,
<https://developer.mozilla.org/en-US/docs/Web/API/Performance>

- [37] MDN - Mozilla, *DOMHighResTimeStamp - Web APIs*, 19 Lusty 2020,
<https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp>
- [38] MDN - Mozilla, *First paint - MDN Web Docs Glossary*, 10 Marca. 2020,
https://developer.mozilla.org/en-US/docs/Glossary/First_paint

Spis rysunków

2.1	Grafika przedstawiająca ocenę wpływu wybranego elementu na atrakcyjność danego narzędzia [4]	7
2.2	Grafika przedstawiająca wzrost czasu ładowania elementów od 10 do 5000 przy użyciu AngularJS Angular2 oraz React [4]	8
2.3	Grafika przedstawiająca wykres zależności czasu do ilości edytowanych elementów na stronie [4]	8
2.4	Grafika przedstawiająca wykres czasu ładowania aplikacji [4]	9
2.5	Porównanie metryk kodu dla pojedynczego komponentu [12]	10
2.6	Porównanie czasów pobrania danych z bazy [12]	11
3.1	Ilustracja przedstawiająca cykl życia statycznej strony internetowej	13
3.2	Ilustracja przedstawiająca cykl działania aplikacji SPA	15
3.3	Grafika przedstawiająca mechanizm działania aplikacji dynamicznych	16
3.4	Grafika przedstawiająca kolejność faz renderowania w przeglądarce	18
3.5	Ilustracja przedstawiająca problem dopisywania elementu na koniec listy	20
3.6	Ilustracja przedstawiająca problem dopisywania elementów na początek listy	21
3.7	Ilustracja mechanizmu przebiegu badania	23
3.8	Grafika przedstawiająca zaimplementowane przyciski w aplikacji	23
3.9	Ilustracja procesu przygotowania aplikacji do konteneryzacji	25
3.10	Ilustracja przedstawiająca warstwy składające się na przykładowy obraz dockera	26
3.11	Grafika przedstawiająca plik index.html wraz z dostępnymi aplikacjami do badania	27
3.12	Ilustracja mechanizmu działania strony statycznej na przykładzie aplikacji kalendarza przy użyciu języka PHP	28
3.13	Grafika przedstawiająca proces przeładowania strony statycznej	29
3.14	Ilustracja procesu ładowania aplikacji oraz zdarzenia rejestrowane przez przeglądarkę	30
3.15	Ilustracja mechanizmu zmiany daty w przypadku aplikacji dynamicznej z punktu widzenia użytkownika	31
3.16	Ilustracja przedstawiająca proces zmiany treści strony w przypadku aplikacji dynamicznej z punktu widzenia komputera	31
3.17	Ilustracja procesu współpracy pomiędzy Selenium a przeglądarką	32
3.18	Wycinek wpisów skryptu przeprowadzającego badanie w środowisku docker-compose. Ilustruje on inicjalizację skryptu oraz mechanizm uzyskiwania połączenia pomiędzy Skryptem - Przeglądarką - Selenium	33

3.19	Wycinek skryptu ukazujący uzyskanie połączenia do Selenium pomimo początkowych problemów	33
3.20	Grafika przedstawia rozpoczęcie badania	35
3.21	Grafika przedstawia skrypt zakańczający badanie po zapisaniu zebranych danych na dysk - widzimy, że ważnym elementem jest zamknięcie połączenia do Selenium	35
3.22	Ilustracja struktury wyniku badania na które zostanie poddane dalszej obróbce .	36
3.23	Grafika przedstawiająca wpis zebranego pomiaru	37
3.24	Diagram kolumnowy ukazujący wynik badania czasu potrzebnego do rozpoczęcia malowania strony	38
3.25	Grafika ukazująca rozmiar plików zasobów dla aplikacji Angular2	39
3.26	Grafika ukazująca rozmiar plików zasobów dla aplikacji React	39
3.27	Grafika ukazująca rozmiar plików zasobów dla aplikacji Vue	40
3.28	Diagram kolumnowy obrazujący wynik badania pomiaru czasu dodania 1000 elementów na początek listy	41
3.29	Diagram kolumnowy obrazujący wynik badania pomiaru czasu dodania 1000 elementów na koniec listy	42
3.30	Diagram kolumnowy obrazujący wynik badania pomiaru czasu zmiany wszystkich wartości listy dla 1000 elementów	43
3.31	Diagram kolumnowy obrazujący wynik badania pomiaru czasu zmiany wartości dla 500 elementów listy	44
3.32	Diagram kolumnowy obrazujący wynik badania pomiaru czasu zamiany miejscowości dwóch wartości na liście elementów	45
3.33	Diagram kolumnowy obrazujący wynik badania pomiaru czasu usunięcia pojedynczej wartości z listy elementów	46
3.34	Diagram kolumnowy obrazujący wynik badania pomiaru czasu usunięcia wszystkich wartości z listy elementów	47

Wrocław, dnia 2020-05-08

Wydział Informatyki, Administracji i Fizjoterapii

Kierunek studiów: **informatyka (INF)**

Daniel Słaby

.....
(imię i nazwisko studenta)

6781

.....
(nr albumu)

**OŚWIADCZENIE O UDOSTĘPNIANIU PRACY
DYPLOMOWEJ**

Tytuł pracy dyplomowej: Analiza porównawcza nowoczesnych rozwiązań technologicznych aplikacji SPA za pomocą jednakowej aplikacji zaimplementowanej w każdym z wybranych rozwiązań

Wyrażam zgodę (nie wyrażam zgody)¹ na udostępnianie mojej pracy dyplomowej.

.....
(podpis studenta)

¹Niepotrzebne skreślić.

Wrocław, dnia 2020-05-08

Wydział Informatyki, Administracji i Fizjoterapii

Kierunek studiów: **informatyka (INF)**

Daniel Słaby

.....
(imię i nazwisko studenta)

6781

.....
(nr albumu)

OŚWIADCZENIE AUTORSKIE

Oświadczam, że niniejszą pracę dyplomową pod tytułem:

Analiza porównawcza nowoczesnych rozwiązań technologicznych aplikacji SPA za pomocą jednakowej aplikacji zaimplementowanej w każdym z wybranych rozwiązań

napisałem/am samodzielnie. Nie korzystałem/am z pomocy osób trzecich, jak również nie dokonałem/am zapożyczeń z innych prac.

Wszystkie fragmenty pracy takie jak cytaty, rycinę, tabele, programy itp., które nie są mojego autorstwa, zostały odpowiednio zaznaczone i zamieszczono w pracy źródła ich pochodzenia. Treść wydrukowanej pracy dyplomowej jest identyczna z wersją pracy zapisaną na przekazywanym nośniku elektronicznym.

Jednocześnie przyjmuję do wiadomości, że jeżeli w przypadku postępowania wyjaśniającego zebrany materiał potwierdzi popełnienie przeze mnie plagiatu, skutkować to będzie niedopuszczeniem do dalszych czynności w sprawie nadania mi tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz złożenie zawiadomienia o popełnieniu przestępstwa.

.....

(podpis studenta)