

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Informatyki

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Sztuczna Inteligencja

Analiza struktury i przepływu danych w nowoczesnych aplikacjach
przeglądarkowych

Adam Steciuk

Numer albumu 300263

promotor
prof. dr hab. inż. Przemysław Rokita

WARSZAWA 2024

Analiza struktury i przepływu danych w nowoczesnych aplikacjach przeglądarkowych

Streszczenie. Przedstawione w niniejszej pracy zgłębienie złożonego obszaru integracji narzędzi deweloperskich z nowoczesnymi bibliotekami frontendowymi, pozwoliło na zaprojektowanie i implementację uniwersalnego narzędzia, ułatwiającego analizę struktury i przepływu danych między komponentami aplikacji webowych.

Dzięki skrupulatnej analizie istniejących rozwiązań, wyodrębniono podstawowe cechy, które powinno posiadać nowe rozwiązanie oraz wybrano formę implementacji w postaci rozszerzenia przeglądarkowego jako najlepiej wpisującą się w ramy postawionych wymagań.

Dogłębne zbadanie wewnętrznych mechanizmów bibliotek React i Svelte, reprezentatywnych dla dwóch różnych podejść do zarządzania stanem w aplikacjach webowych, umożliwiło stworzenie uniwersalnej architektury narzędzia oraz priorytetyzacji jego elastyczności i łatwości rozbudowywania o obsługę kolejnych bibliotek.

Podczas analizy React Developer Tools, jednego z najpopularniejszych rozszerzeń deweloperskich na rynku, zdobyto spostrzeżenia dotyczące ugruntowanych praktyk, wzorców i metodologii w projektowaniu rozszerzeń przeglądarkowych. Ich wykorzystanie pozwoliło na implementację wydajnego systemu komunikacji, zarówno między poszczególnymi komponentami rozszerzenia, jak i między nim a aplikacjami webowymi.

Pokonanie nieoczekiwanych wyzwań, napotkanych podczas pracy nad projektem, zaowocowało naprawą błędów w otwartoźródłowej bibliotece do testów jednostkowych rozszerzeń korzystających z API rozszerzeń Chrome oraz stworzeniem i publikacją nowej biblioteki, rozbudowującej funkcje narzędzia do budowania aplikacji webowych Vite.

Rygorystyczne testy i porównanie autorskiego rozwiązania z istniejącymi rozszerzeniami, potwierdziły jego stabilność, wydajność i uniwersalność, a także pomogły nakreślić kierunki dalszego rozwoju i potencjalne obszary ulepszeń.

Niniejsza praca stanowi kompleksowe omówienie procesu projektowania i implementacji przeglądarkowego rozszerzenia narzędzi deweloperskich i stanowi fundament dla dalszych badań i innowacji w dziedzinie wspierania procesu tworzenia aplikacji webowych.

Słowa kluczowe: React, Svelte, narzędzia deweloperskie, uniwersalne rozszerzenia przeglądarkowe, analiza stanu aplikacji, Vite, React Developer Tools, Chrome Extension API

Analysis of structure and data flow in modern web applications

Abstract. This paper delves into the complex realm of integrating developer tools with modern frontend libraries, culminating in the creation of a versatile tool aimed at analyzing structure and data flow between components in web applications.

A thorough examination of existing solutions identified essential features for the new tool, leading to its implementation as a browser extension aligned with established requirements.

By delving into the inner workings of React and Svelte libraries, representing different approaches to state management in web applications, a universal tool architecture was devised, prioritizing performance, flexibility, and extensibility to accommodate additional libraries.

The analysis of React Developer Tools, a prominent browser extension, provided valuable insights into established design practices, patterns, and architecture methodologies. Leveraging these insights, an efficient communication system was implemented, facilitating seamless interaction between the extension's components and web applications.

Overcoming unforeseen challenges during the project led to bug fixes in an open-source unit testing library used for Chrome extensions, as well as the creation and release of a new library enhancing the capabilities of the Vite web application building tool.

Thorough testing and comparison with existing extensions confirmed the developed solution's stability, performance, and adaptability while highlighting areas for further development and enhancement.

This paper offers a comprehensive overview of designing and implementing a browser extension for developer tools, setting the stage for future research and innovation in supporting the web application development process.

Keywords: React, Svelte, developer tools, universal browser extensions, application state analysis, Vite, React Developer Tools, Chrome Extension API

Spis treści

1. Wstęp	11
1.1. Kontekst pracy	11
1.2. Motywacja do podjęcia pracy	11
1.3. Cel pracy	12
1.4. Struktura pracy	13
2. Istniejące rozwiązania	14
2.1. Narzędzia dla biblioteki React	14
2.1.1. React Developer Tools	14
2.1.2. Realize	15
2.1.3. React Sight	16
2.2. Narzędzia dla biblioteki Svelte	16
2.2.1. Svelte DevTools	16
2.2.2. Svelte DevTools+	18
2.3. Uniwersalne narzędzia	18
3. Ogólna koncepcja rozwiązania	19
3.1. Narzędzie w formie rozszerzenia przeglądarkowego	19
3.2. Inne możliwe sposoby integracji	19
3.3. Wybór sposobu integracji	20
3.4. Wybór analizowanych bibliotek	20
4. Architektura rozszerzeń przeglądarkowych	22
4.1. Metadane rozszerzenia (manifest)	22
4.2. Anatomia rozszerzeń przeglądarkowych	22
4.2.1. Skrypty tła	22
4.2.2. Skrypty zawartości	23
4.2.3. Okno modalne	24
4.2.4. Strona opcji	24
4.2.5. Narzędzia deweloperskie	24
4.3. API rozszerzeń Chrome	24
4.3.1. Wiadomości jednorazowe	25
4.3.2. Długotrwałe połączenia	25
4.3.3. Persystencja danych w rozszerzeniach	26
5. Analiza działania biblioteki React	27
5.1. Co to jest React?	27
5.2. Komponenty w Reaccie	27
5.3. Właściwości komponentów	28
5.4. JavaScript XML (JSX)	28
5.5. Drzewo aplikacji	28
5.6. Rekonyliacja, czyli wykrywanie zmian	30
5.7. Renderowanie widoku aplikacji	30
5.8. React Fiber	30

5.8.1. Struktura obiektu Fiber	31
5.9. Hooke - reaktywność w komponentach funkcyjnych	31
5.9.1. Hook <code>useState</code>	31
5.9.2. Hook <code>useReducer</code>	32
5.9.3. Hook <code>useContext</code>	32
5.9.4. Hook <code>useRef</code>	33
5.9.5. Hook <code>useEffect</code>	33
5.9.6. Hook <code>useMemo</code>	34
5.9.7. Hook <code>useCallback</code>	34
5.9.8. Hook <code>useDebugValue</code>	34
5.9.9. Inne hooki	34
5.9.10. Niestandardowe hooki	35
6. Analiza działania biblioteki Svelte	37
6.1. Założenia biblioteki	37
6.2. Składnia Svelte	37
6.3. Svelte - inne podejście do reaktywności	39
6.4. Bloki logiczne	39
6.4.1. Blok <code>{#if...}</code>	39
6.4.2. Blok <code>{#each...}</code>	39
6.4.3. Blok <code>{#await...}</code>	40
6.4.4. Blok <code>{#key...}</code>	41
6.5. Inne funkcjonalności	41
7. Analiza implementacji React Developer Tools	42
7.1. Punkty wejścia dla rozszerzenia przeglądarkowego	42
7.1.1. Przygotowanie wstrzyknięcia (<code>prepareInjection.js</code>)	43
7.1.2. Skrypt tła (<code>background.js</code>)	44
7.1.3. Okno modalne	45
7.1.4. Narzędzia deweloperskie (<code>main.html</code>)	46
7.2. Struktura i instalacja hooka w aplikacji (<code>installHook.js</code>)	46
7.2.1. Właściwość <code>renderers</code>	47
7.2.2. Właściwość <code>rendererInterfaces</code>	47
7.2.3. Właściwość <code>supportsFiber</code>	47
7.2.4. Funkcja <code>inject</code>	48
7.2.5. Funkcja <code>checkDCE</code>	48
7.2.6. Funkcja <code>onCommitFiberRoot</code>	48
7.2.7. Funkcja <code>onCommitFiberUnmount</code>	49
7.3. Skrypt <code>renderer.js</code>	49
7.4. Pomost między izolowanym a głównym światem wykonywania (<code>proxy.js</code>)	49
7.4.1. Ograniczenia mechanizmu <code>window.postMessage</code>	50
7.5. Dopasowanie RDT do wersji Reacta (<code>backendManager.js</code>)	50
7.6. Backendy narzędzi deweloperskich	50
7.7. Parsowanie drzewa Fiber	51

7.7.1. Montowanie obiektów struktury Fiber	51
7.7.2. Aktualizacja obiektów struktury Fiber	51
7.8. Optymalizacja przesyłania danych	52
7.8.1. Tabela operacji	52
7.9. Panele narzędzi deweloperskich	53
7.10. Inspekcja hooków Reacta	54
8. Przygotowanie środowiska deweloperskiego	55
8.1. Struktura projektu	55
8.2. Konfiguracja Vite	56
8.3. Rozszerzenie funkcjonalności Vite	56
8.3.1. Interfejs rozszerzeń Vite	57
8.3.2. Działanie stworzonego rozszerzenia	57
8.3.3. Testy rozszerzenia Vite	61
8.3.4. Publikacja rozszerzenia Vite	61
8.4. Uruchomienie głównego projektu	61
8.4.1. Załadowanie rozszerzenia do przeglądarki	61
9. Autorskie rozwiązanie	63
9.1. Główne założenia implementacji	63
9.2. Nazwa rozszerzenia	63
9.3. Punkty wejścia dla rozszerzenia przeglądarkowego	64
9.3.1. Skrypt tła (background/index.ts)	65
9.3.2. Okno modalne (popup/index.html)	65
9.3.3. Główny skrypt zawartości (content/content-main/index.ts)	65
9.3.4. Izolowany skrypt zawartości (content/content-isolated/index.ts)	65
9.3.5. Narzędzia deweloperskie (devtools/index.html)	66
9.4. Abstrakcje nad mechanizmami komunikacji	67
9.4.1. Abstrakcja nad mechanizmem wiadomości jednorazowych (shared/chrome/chrome-message.ts)	67
9.4.2. Abstrakcja nad mechanizmem długotrwałych połączeń (shared/chrome/ChromeBridge.ts)	68
9.4.3. Abstrakcja nad mechanizmem windows.postMessage (pages/content/shared/PostMessageBridge.ts)	69
9.5. Uniwersalna reprezentacja elementów	70
9.6. Łączenie się rozszerzenia z bibliotekami frontendowymi - „adaptery” (pages/content/content-main/Adapter.ts)	70
9.6.1. Abstrakcyjne metody klasy Adapter	71
9.6.2. Dehydratacja danych	72
9.6.3. Podświetlanie elementów drzewa DOM	74
9.7. Implementacja adaptera dla Reacta	74
9.7.1. Mechanizm wstrzykiwania hooka i kompatybilność z RDT	74
9.7.2. Wstrzykiwany hook	76
9.7.3. Łączenie się Reacta z rozszerzeniem	78

9.7.4.	Montowanie i aktualizacja elementów	78
9.7.5.	Odmontowywanie elementów	80
9.7.6.	Inspekcja stanu i właściwości komponentów	81
9.7.7.	Parsowanie szczegółowych danych komponentów	82
9.8.	Implementacja adaptera dla Svelte	84
9.8.1.	Wykrywanie biblioteki Svelte	84
9.8.2.	Nasłuchiwanie na zdarzenia Svelte	84
9.8.3.	Obsługa rejestrowania komponentów	85
9.8.4.	Wstrzykiwanie logiki adaptera do funkcji cyklu życia bloków	86
9.8.5.	Przechowywanie identyfikatora przetwarzanego bloku	86
9.8.6.	Wykrywanie i obsługa montowania bloków	87
9.8.7.	Wykrywanie i obsługa aktualizacji bloków	89
9.8.8.	Wykrywanie i obsługa usuwania bloków	89
9.8.9.	Montowanie elementów DOM	90
9.8.10.	Aktualizacja elementów DOM	90
9.8.11.	Usuwanie elementów DOM	90
9.8.12.	Inspekcja stanu i właściwości komponentów	90
9.9.	Przekazywanie informacji do panelu narzędzi deweloperskich	91
9.9.1.	Aktualizacja przechowywanych w pamięci drzew elementów	92
9.10.	Panel narzędzi deweloperskich - interfejs użytkownika (pages/panel/)	93
9.10.1.	Globalne konteksty dla panelu narzędzi deweloperskich	93
9.10.2.	Struktura interfejsu użytkownika	94
9.10.3.	Menu panelu narzędzi deweloperskich	95
9.10.4.	Filtrowanie wyświetlanych elementów	95
9.10.5.	Widok drzewa elementów	96
9.10.6.	Widok szczegółów elementu	97
9.10.7.	Persystencja ustawień użytkownika	99
9.11.	Prezentacja funkcjonalności	99
10.	Testy przygotowanego rozwiązania	100
10.1.	Testy jednostkowe i integracyjne	100
10.1.1.	Automatyzacja CI/CD	100
10.1.2.	Konieczność stworzenia poprawek dla vitest-chrome	101
10.2.	Testy manualne	101
11.	Porównanie z istniejącymi rozwiązaniami	104
11.1.	Porównanie z narzędziami dla biblioteki React	104
11.1.1.	Realize	104
11.1.2.	React Sight	105
11.1.3.	React Developer Tools	105
11.2.	Porównanie z narzędziami dla biblioteki Svelte	106
11.2.1.	Svelte DevTools	106
11.2.2.	Svelte DevTools+	108
11.3.	Podsumowanie porównania	109

12.Podsumowanie	111
12.1.Kierunek dalszych prac	112
Bibliografia	115
Wykaz symboli i skrótów	122
Spis rysunków	123
Spis załączników	123

1. Wstęp

1.1. Kontekst pracy

Pierwsze strony internetowe, ze względu na ograniczenia technologiczne, były prostymi statycznymi dokumentami. Użytkownik przy użyciu przeglądarki internetowej inicjował żądanie, serwer wykonywał akcję (np. zapytanie do bazy danych) i zwracał gotowy dokument HTML, który wyświetlany był przez przeglądarkę. Każda akcja wymagała przeładowania całej strony a interakcja z witrynami była ograniczona do używania prostych formularzy, klikania w linki i w przyciski [1].

W roku 1995 w przeglądarce Netscape Navigator 2 zadebiutowała pierwsza wersja języka JavaScript będąca przedświtem interaktywnych aplikacji internetowych [2].

Z biegiem czasu, przeglądarki internetowe zyskiwały nowe funkcjonalności a język JavaScript coraz bardziej ewoluował. W pierwszej dekadzie XXI wieku rozwiązania takie jak AJAX (Asynchronous JavaScript and XML) [3] czy jQuery [4] rozpoczęły trend przenoszenia coraz większej ilości logiki aplikacji na stronę klienta. Jednakże prawdziwy przełom nastąpił w 2010 roku, kiedy to Google opublikowało framework AngularJS [5].

AngularJS zyskał ogromną popularność i zapoczątkował erę tworzenia tzw. aplikacji SPA (Single Page Application). Aplikacje takie nie wymagają przeładowania strony przy każdej akcji użytkownika, a także ograniczają wielokrotne pobieranie zewnętrznych zasobów, takich jak obrazy czy pliki CSS (Cascading Style Sheets).

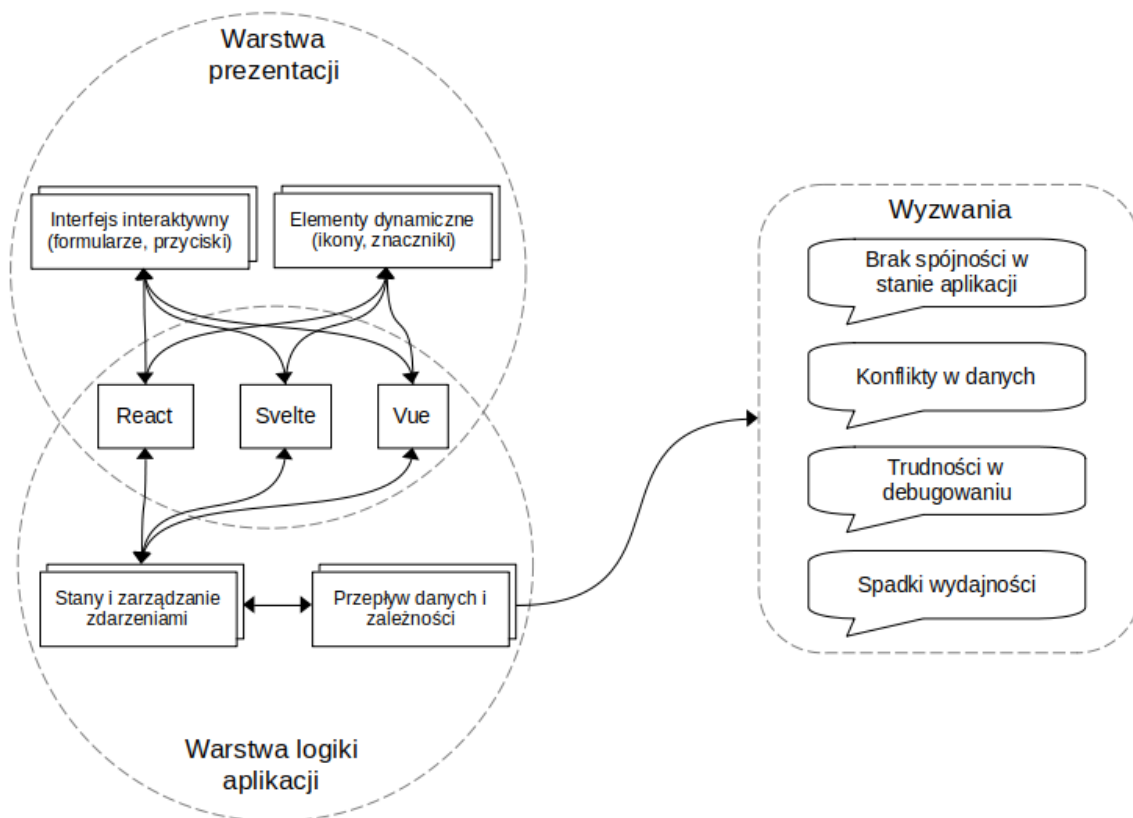
W kolejnych latach powstało wiele innych frameworków i bibliotek, które w coraz większym stopniu ułatwiały budowanie coraz bardziej skomplikowanych aplikacji. W 2013 roku powstał React [6], w 2014 roku Vue.js [7], a w 2016 roku Svelte [8].

1.2. Motywacja do podjęcia pracy

W obecnych czasach, gdy aplikacje przeglądarkowe dorównują (a czasem nawet przerażają) poziomem skomplikowania klasyczne aplikacje „desktopowe”, brak usystematyzowanego sposobu zarządzania stanem i przepływem danych może prowadzić do licznych, bardzo trudnych w wykryciu i rozwiązaniu błędów. Twórcy frameworków, a także niezależni programiści, zauważywszy ten problem, zaczęli tworzyć narzędzia, które miałyby ułatwić przeglądanie oraz debugowanie stanu aplikacji. Rozwiązania takie często jednak pozostawiają wiele do życzenia w kwestii stabilności i wydajności funkcjonowania, a także zwykle są dostosowane do konkretnego frameworku.

Coraz częściej zdarza się, że aplikacje korzystają z wielu różnych bibliotek frontendowych na raz. Część frontendowa aplikacji powstaje z tzw. mikrofrontendów, na wzór backendowych mikroserwisów. Te z kolei często tworzone są przez różne zespoły programistów, nie zawsze korzystające z tych samych technologii [9] [10].

W takich sytuacjach, specjalizacja narzędzi do obsługi jednego, konkretnego frameworku wymaga od programisty instalacji wielu z nich jednocześnie, przyzwyczajania się do różnych interfejsów użytkownika, a także ciągłych zmian kontekstu nawet pracując nad różnymi fragmentami jednej aplikacji.



Rysunek 1.1. Problemy związane z zarządzaniem stanem w aplikacjach przeglądarkowych

Aby rozwiązać ten problem, postanowiono zbadać możliwość stworzenia uniwersalnego narzędzia, które pozwoliłoby na rozbudowywanie go o obsługę dowolnych bibliotek i zapewniałoby ujednolicony interfejs użytkownika niezależnie od używanych frameworków.

1.3. Cel pracy

Celem pracy jest przegląd oraz analiza wad i zalet istniejących rozwiązań pozwalających na debugowanie i wizualizację stanu aplikacji przeglądarkowych, a także poznanie sposobów ich integracji z frameworkami frontendowymi. Aby osiągnąć ten cel, koniecznym jest zagłębienie się w architekturę istniejących narzędzi oraz w sposoby implementacji reaktywności, zarządzania stanem i przepływem danych w analizowanych bibliotekach.

Zwieńczeniem pracy będzie stworzenie narzędzia, które:

- ułatwi proces debugowania i analizy stanu aplikacji przeglądarkowych korzystających z różnych bibliotek frontendowych,
- pozwoli na łatwe rozbudowanie, celem integracji z kolejnymi bibliotekami,
- będzie łatwe w instalacji i użyciu,
- nie będzie wymagało modyfikacji kodu źródłowego aplikacji,
- będzie łatwe w rozbudowie i utrzymaniu,
- nie będzie w znaczącej mierze wpływało na wydajność aplikacji.

Zostaną także osiągnięte następujące cele:

- przetestowanie stworzonego narzędzia,
- porównanie stworzonego narzędzia z istniejącymi rozwiązaniami pod kątem:
 - wydajność,
 - funkcjonalności,
 - stabilność,
 - łatwość użycia,
 - możliwość konfiguracji.

1.4. Struktura pracy

Pracę podzielono na jedenaście rozdziałów. Pierwszy z nich stanowi wstęp, przedstawiający kontekst i motywację do jej podjęcia.

W kolejnych dwóch rozdziałach przedstawiono: przegląd istniejących rozwiązań, możliwe podejścia do rozwiązania postawionego problemu oraz argumentację wyboru rozszerzenia przeglądarkowego jako najbardziej uniwersalnego z nich.

Rozdział czwarty zawiera ogólny opis architektury rozszerzeń dla przeglądarek opartych na silniku Chromium.

W rozdziałach piątym i szóstym zgłębiono sposób działania wybranych bibliotek frontendowych: odpowiednio React i Svelte.

Rozdział siódmy zawiera szczegółowy opis implementacji i działania jednego z najpopularniejszych, „state of the art” narzędzi do debugowania stanu aplikacji webowych - React Developer Tools.

Dwa kolejne rozdziały prezentują: konfigurację środowiska deweloperskiego, implementację stworzonego w ramach pracy narzędzia oraz opis napotkanych problemów i ich rozwiązań.

Rozdziały dziesiąty i jedenasty zawierają opis przeprowadzonych testów i porównanie stworzonego narzędzia z istniejącymi rozwiązaniami.

W ostatnim rozdziale zawarto podsumowanie pracy, wnioski oraz propozycje dalszych kierunków rozwoju stworzonego narzędzia.

Wszystkie załączniki, w tym odnośniki do repozytoriów, które zawierają kod źródłowy stworzonych narzędzi, jak i również dodatkowe materiały, zamieszczono na końcu pracy.

2. Istniejące rozwiązania

2.1. Narzędzia dla biblioteki React

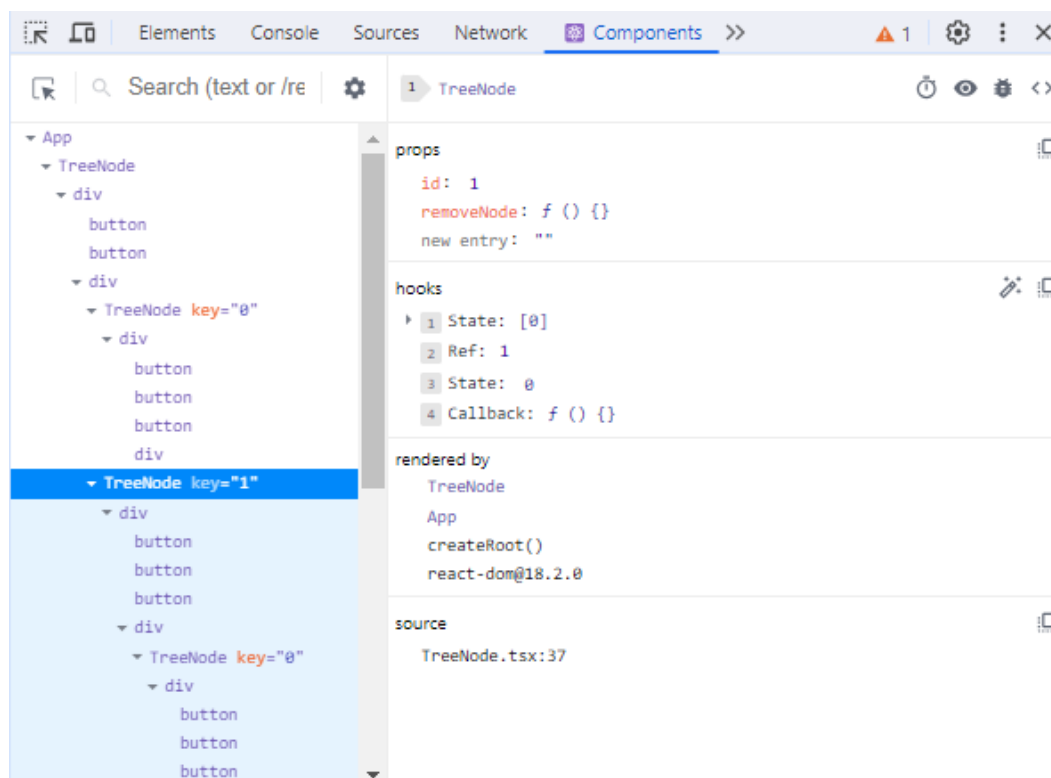
Wszystkie użyte w tym podrozdziale pojęcia dotyczące Reacta zostały wyjaśnione w rozdz. 5.

2.1.1. React Developer Tools

React Developer Tools (RDT) to oficjalny zestaw narzędzi deweloperskich stworzony przez twórców Reacta. Dla aplikacji webowych jest on dostępny jako rozszerzenie do przeglądarek Chrome, Firefox i Edge.

Narzędzie to różni się nieco pod względem sposobu działania i implementacji w zależności od tego czy używane jest z aplikacjami przeglądarkowymi, czy uruchamianymi w innych kontekstach (np. React Native [11]). W tej pracy, opisując RDT, będzie mowa o wersji przeznaczonej do aplikacji webowych (chyba, że zaznaczono inaczej).

Do użycia RDT nie są wymagane żadne modyfikacje kodu źródłowego badanej aplikacji. Zainstalowane rozszerzenie wykrywa, czy obecnie otwarta strona używa Reacta, i jeżeli tak, to w panelu deweloperskim przeglądarki tworzy dwie nowe zakładki: „Components” i „Profiler”.



Rysunek 2.1. Widok zakładki „Components” w React Developer Tools.

Zakładka „Components” wyświetla hierarchicznie zbudowane drzewo komponentów aplikacji. Pozwala na wyświetlenie nazwy komponentu, jego stanu, przekazanych do niego wartości, używanych hooków oraz łańcucha komponentów nadrzędnych. Ponadto, RDT

pozwała na edycję stanu komponentu umożliwiającą szybkie przetestowanie zmian w aplikacji. Wyświetlane komponenty można łatwo filtrować i wyszukiwać. Opcją szczególnie przydatną w przypadku złożonych aplikacji, jest wyświetlanie informacji o komponentach poprzez najechanie kursorem na elementy strony. RDT udostępnia także opcję podświetlania komponentów, które zostały ponownie wyrenderowane w wyniku zmiany stanu aplikacji, co może być przydatne w celu optymalizacji jej wydajności i badania wpływu zmian stanu na konkretne komponenty [12][13].

Zakładka „Profiler” zawiera obszerny zbiór funkcji, które pozwalają na analizę wydajności aplikacji. Nie opisano szczegółów ich funkcjonowania, gdyż profilowanie wykracza poza ramy tej pracy.

React Developer Tools jest narzędziem rozbudowanym, przygotowanym do funkcjonowania z wieloma różnymi wersjami Reacta. Poprawnie obsługuje aplikacje, w których używane są zarówno klasy komponentów, jak i hooki. Obsługuje strony, na których znajduje się kilka instancji Reacta, a także oferuje wsparcie dla aplikacji mobilnych tworzonych przy użyciu React Native.

RDT można potraktować jako „state of the art” w dziedzinie narzędzi do debugowania aplikacji przeglądarkowych, i dlatego na nim, postanowiono oprzeć autorskie rozwiązanie. W ramach pracy dokonano starań, aby odtworzyć istotne funkcjonalności RDT, takie jak wyświetlanie hierarchii komponentów i ich stanu, a następnie rozszerzyć je o funkcje przedstawione w podrozdz. 1.3.

Opis implementacji narzędzia przedstawiony został w rozdz. 7.

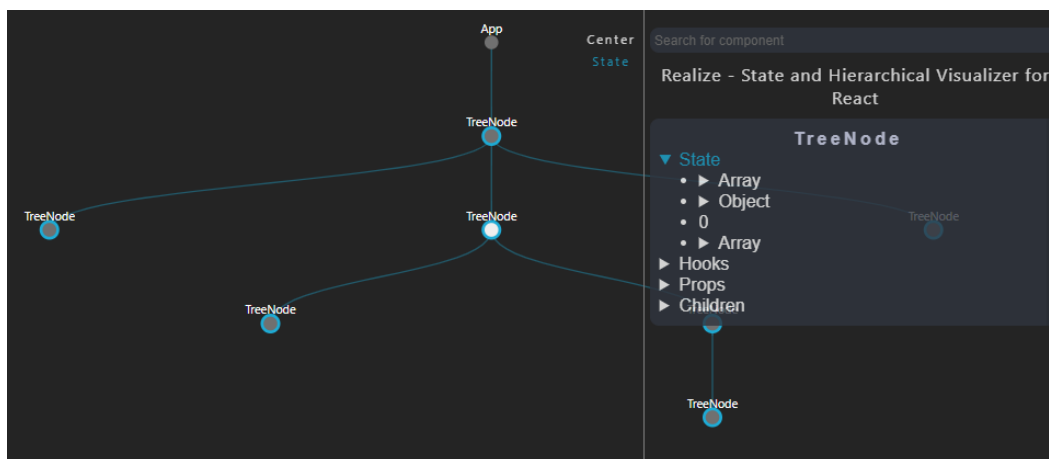
2.1.2. Realize

Realize to narzędzie również dostępne w formie rozszerzenia przeglądarkowego, którego celem jest wyświetlenie hierarchii komponentów aplikacji React. W przeciwieństwie do RDT, Realize nie jest oficjalnym narzędziem Reacta, a jego twórcy nie są związani z Facebookiem.

Realize wyświetla drzewo komponentów przy użyciu biblioteki D3.js [14] i także pozwala na wyświetlenie stanu każdego z komponentów. Twórcy, jak podają na swojej stronie, postawili sobie za cel ułatwienie wizualizacji przepływu stanu w aplikacji [15], jednakże narzędzie jest w stanie jedynie podświetlać węzły, które odpowiadają komponentom definiującym fragmenty stanu i komponentom, do których przekazywane są dane w postaci właściwości [16].

Dodatkową wadą narzędzia jest to, że do prawidłowego działania, wymaga ono dodania do przeglądarki rozszerzenia React Developer Tools. Realize nie łączy się bezpośrednio z Reactem a wykorzystuje hook tworzony przez RDT (rozdz. 7). Wiąże się to z dodatkowym narzutem na wydajności aplikacji, a także może skutkować konfliktami między rozszerzeniami.

Realize przekazuje z kontekstu strony internetowej do panelu rozszerzenia całe drzewo komponentów przy każdym renderowaniu aplikacji. Takie rozwiązanie jest bardzo nieefektywne (wielokrotne przekazywanie danych o komponentach, które nie uległy zmianie), i nawet w przypadku średniej wielkości aplikacji, prowadzi do znacznego spadku wydajno-



Rysunek 2.2. Interfejs narzędzia Realize.

ści. Narzut na wydajności każdorazowego przekazywania całego drzewa komponentów zauważyli również twórcy RDT w jego poprzedniej implementacji [17], czemu poświęcono więcej uwagi w podrozdz. 7.8.

Jak zostało pokazane w podrozdz. 11.1.1, narzędzie jest niewydajne i zawodne, a jego użycie w bardziej złożonych aplikacjach, może nawet doprowadzić do zawieszenia przeglądarki.

2.1.3. React Sight

React Sight jest narzędziem oferującym podobne funkcjonalności do Realize. Posiada ono jednak nieco większe możliwości konfiguracji, wyświetlanego przez D3.js, drzewa komponentów. Niestety, to rozwiązanie również wiąże się z koniecznością instalacji RDT oraz z narzutem, związanym z przekazywaniem całej struktury aplikacji przy każdym renderowaniu. Nie ułatwia także samej wizualizacji przepływu danych między komponentami. Co więcej, React Sight nie jest rozwijane od 2020 roku, a samo rozszerzenie nie jest już dostępne w sklepach przeglądarek [18][19].

2.2. Narzędzia dla biblioteki Svelte

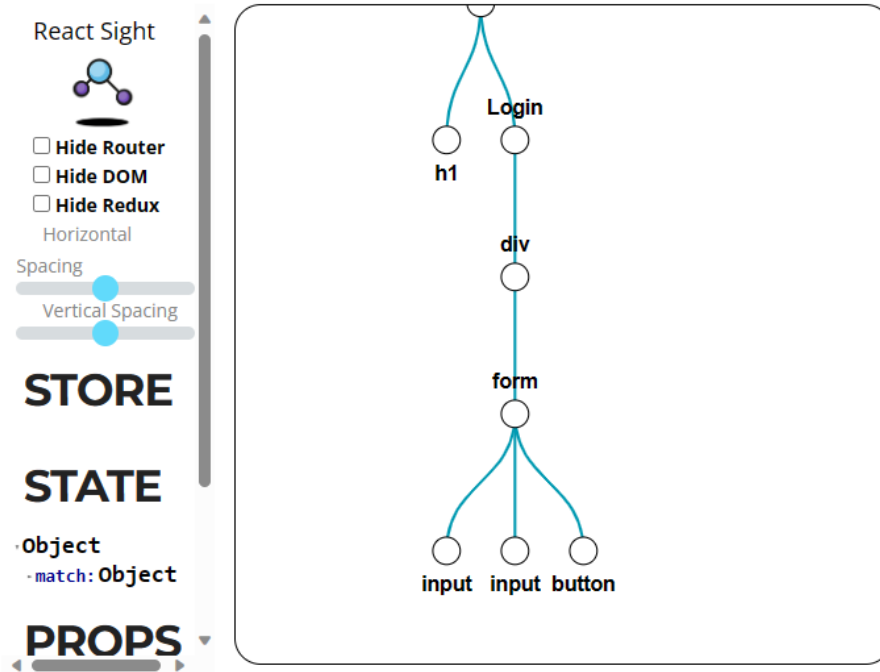
2.2.1. Svelte DevTools

Svelte DevTools to oficjalny zestaw narzędzi deweloperskich pozwalających na analizę aplikacji stworzonych przy użyciu Svelte.

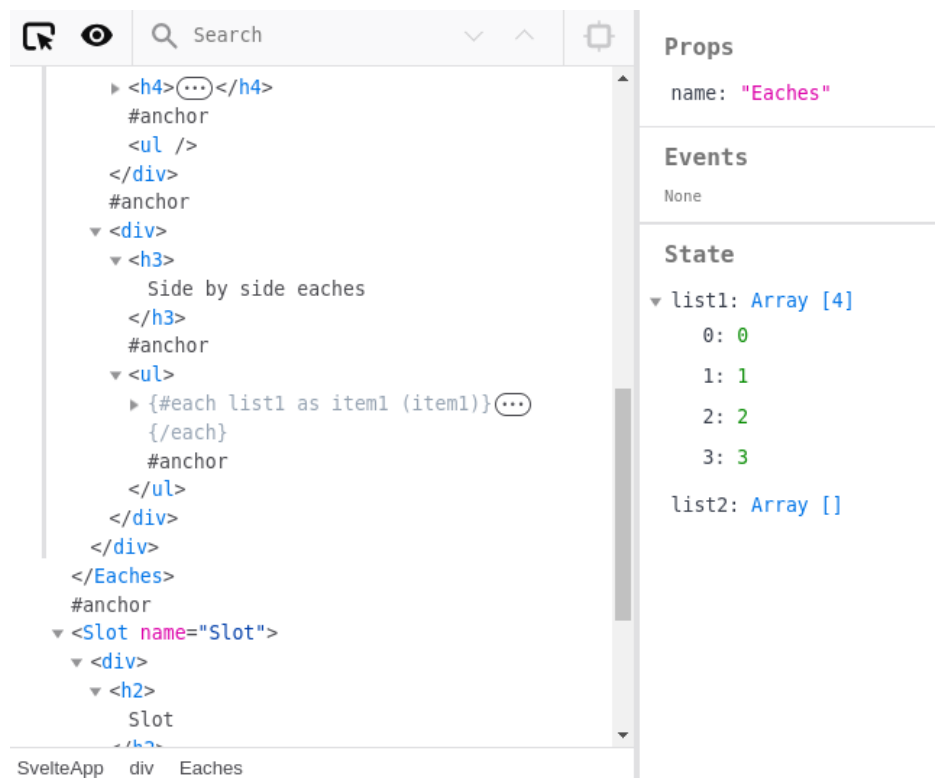
Narzędzie dostępne jest jako rozszerzenie do przeglądarek opartych na chromium i zapewnia podobne funkcjonalności do zakładki „Components” w React Developer Tools. Pozwala zarówno na wyświetlanie, jak i edycję stanu komponentów oraz przekazanych do nich wartości, a także na wizualizację ich hierarchii.

Svelte DevTools pozwala również na proste filtrowanie wyświetlanych elementów w zależności od ich typu i nazwy [20].

Podczas korzystania z narzędzia, można zauważyć, że jest ono mniej rozbudowane niż narzędzia przystosowane do pracy z Reactem. Nie oferuje możliwości połączenia się z aplikacjami zbudowanymi w trybie produkcyjnym, a także relatywnie często pojawiają się



Rysunek 2.3. Interfejs narzędzia React Sight.



Rysunek 2.4. Interfejs narzędzia Svelte DevTools.

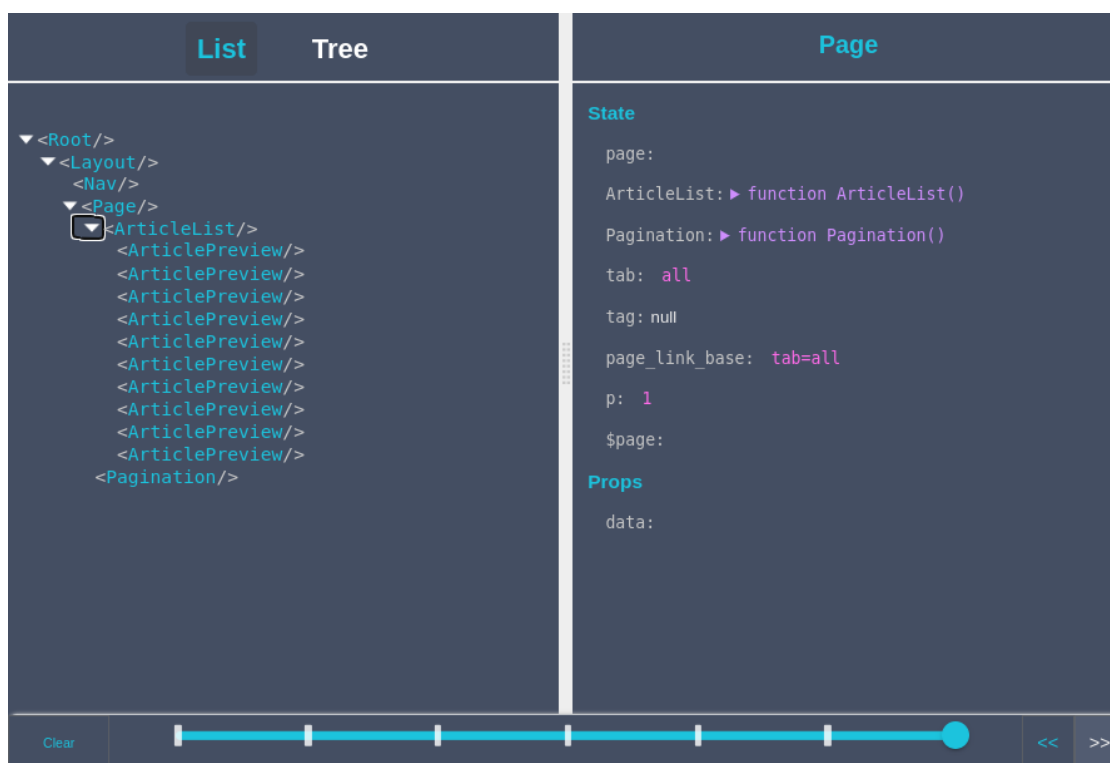
w nim problemy ze stabilnością (np. wymaganie odświeżenia strony w celu połączenia z aplikacją).

2.2.2. Svelte DevTools+

Svelte DevTools+ to nieoficjalne rozszerzenie przeglądarkowe pozwalające na analizę struktury, stanu i właściwości komponentów.

Narzędzie to oferuje ograniczone funkcjonalności względem Svelte DevTools, bowiem widok struktury uwzględnia jedynie komponenty i nie obejmuje ani bloków Svelte (podrozdz. 6.4), ani elementów drzewa DOM.

Jak zostało pokazane w podrozdz. 11.2.2, interfejs rozszerzenia również pozostawia wiele do życzenia. Narzędzie to wyróżnia się jednak, na tle innych rozwiązań, swoją unikatową funkcjonalnością, która umożliwia zapisywanie stanu komponentów i ich właściwości w celu późniejszego odtworzenia zarejestrowanych zmian [21] [22].



Rysunek 2.5. Interfejs narzędzia Svelte DevTools+.

2.3. Uniwersalne narzędzia

W ramach przeprowadzonych poszukiwań nie, udało się znaleźć narzędzi, które pozwalałyby na integrację z więcej niż jedną biblioteką frontendową. Wszystkie, z wyżej przedstawionych, skupiają się na jednej z nich. Analiza aplikacji kompozytowej, wykorzystującej wiele różnych bibliotek frontendowych, wymagałaby zatem korzystania z kilku różnych narzędzi jednocześnie.

3. Ogólna koncepcja rozwiązania

3.1. Narzędzie w formie rozszerzenia przeglądarkowego

Wszystkie wymienione w podrozdz. 2.1.1 rozwiązania zostały zaimplementowane jako rozszerzenia przeglądarkowe. Do zalet takiego podejścia, niewątpliwe, należy zaliczyć:

- brak konieczności modyfikacji kodu źródłowego aplikacji,
- łatwość instalacji i konfiguracji narzędzia,
- możliwość analizy aplikacji w czasie rzeczywistym,
- możliwość integracji z wdrożonymi już aplikacjami,
- znikomy wpływ na bezpieczeństwo aplikacji,
- łatwość włączenia/wyłączenia narzędzia w dowolnym momencie,
- łatwość dystrybucji narzędzia (poprzez sklepy przeglądarek),
- intuicyjność interfejsu dla deweloperów zaznajomionych z przeglądarkowymi narzędziami deweloperskimi.

Rozwiązanie w postaci rozszerzenia przeglądarkowego ma jednak swoje wady. Między innymi:

- mała kontrola nad sposobem integracji, ograniczona do komunikacji z interfejsami udostępnionymi przez twórców frameworków/bibliotek,
- funkcjonalności ograniczone przez możliwości dostępne dla rozszerzeń przeglądarkowych.

Pierwsza z wad nie stanowi większego problemu w przypadku oficjalnych narzędzi deweloperskich, zwykle tworzonych i rozwijanych przez samych twórców bibliotek, ponieważ mogą oni swobodnie modyfikować sposób działania biblioteki w celu umożliwienia integracji i udostępnienia potrzebnych interfejsów. Jednak w przypadku narzędzi tworzonych przez niezależnych deweloperów, integracja z aplikacją, jedynie poprzez rozszerzenie przeglądarkowe, może być problematyczna, a nawet niemożliwa.

3.2. Inne możliwe sposoby integracji

Mając na uwadze powyższe wady, rozważyć można inne możliwe sposoby integracji narzędzia deweloperskiego z aplikacją webową. Wyodrębniłem dwa inne główne podejścia:

1. Integracja z kodem źródłowym aplikacji.

- Zalety:
 - Pełna kontrola nad kodem źródłowym aplikacji.
 - Możliwość przechwytywania dowolnych danych.
- Wady:
 - Dodatkowa zależność w projekcie.
 - Potencjalne zwiększenie rozmiaru wynikowych plików aplikacji.
 - Konieczność instalacji i konfiguracji narzędzia.
 - Konieczność modyfikacji kodu źródłowego aplikacji.
 - Możliwe problemy z kompatybilnością z różnymi konfiguracjami projektu.

- Brak możliwości integracji z wdrożonymi już aplikacjami.
 - Potencjalne problemy z bezpieczeństwem.
 - Potencjalne problemy z wydajnością.
2. Statyczna analiza kodu źródłowego aplikacji.
- Zalety:
 - Brak konieczności modyfikacji kodu źródłowego aplikacji.
 - Duża kontrola nad sposobem integracji.
 - Wady:
 - Kosztowna implementacja parsera kodu źródłowego, charakterystycznego składnią dla danego frameworka/biblioteki.
 - Konieczność instalacji i konfiguracji narzędzia.
 - Brak możliwości integracji z wdrożonymi już aplikacjami.
 - Brak możliwości analizy aplikacji w czasie rzeczywistym.

3.3. Wybór sposobu integracji

Uwzględniając przedstawione powyżej zalety i wady poszczególnych podejść, zdecydowałem się na rozwiązanie w postaci rozszerzenia przeglądarkowego. Rozwiązanie to jest najbardziej uniwersalne i łatwe w użyciu. Głównie jego ograniczenie, czyli brak pełnej kontroli nad sposobem integracji, mogłoby być, w razie konieczności, zniwelowane poprzez zastosowanie rozwiązania hybrydowego, które łączyłoby rozszerzenie przeglądarkowe z jednym z pozostałych podejść.

Według serwisu StatCounter [23], w 2023 roku, najpopularniejszymi przeglądarkami internetowymi na komputerach osobistych były:

1. Google Chrome (64.73%),
2. Safari (18.56%),
3. Microsoft Edge (4.97%),
4. Firefox (3.36%),
5. Opera (2.86%).

Z tych przeglądarek, Google Chrome, Microsoft Edge oraz Opera, oparte są na silniku Chromium [24]. Firefox w znacznym stopniu kompatybilny jest z rozszerzeniami stworzonymi dla Chromium i, po drobnych modyfikacjach kodu rozszerzeń, powinien poprawnie obsługiwać większość z nich [25].

Mając to na uwadze, zdecydowałem się na stworzenie rozszerzenia przeglądarkowego dla silnika Chromium i zgodnego z API rozszerzeń Chrome (podrozdz. 4.3).

3.4. Wybór analizowanych bibliotek

Według niemal 40000 odpowiedzi, zgromadzonych w ankiecie przeprowadzonej przez State of JS w 2022 roku [26], najpopularniejszym frontendowym frameworkiem jest React. Jego użycie zadeklarowało 81.8% ankietowanych. Zdecydowanie rzadziej używanym (21.2%), jednak cieszącym się dużo większym zainteresowaniem wśród programistów, jest

Svelte (69.8% ankietowanych chce go poznać lub używać). Z uwagi na fakt ten, w pracy skupiono się na tych dwóch bibliotekach.

4. Architektura rozszerzeń przeglądarkowych

Jak wspomniano w podrozdz. 3.3, pracę skupiono na stworzeniu rozszerzenia przeglądarkowego dla silnika Chromium, toteż przedstawiona architektura odnosi się do tego środowiska.

Rozszerzenia przeglądarkowe to programy, które dodają funkcjonalności do przeglądarki lub do aplikacji w przeglądarkach internetowych. Zwykle mają za zadanie spełniać jedną konkretną funkcję i są napisane w językach takich jak: HTML, CSS, JavaScript.

4.1. Metadane rozszerzenia (manifest)

Plikiem metadanych dla każdego rozszerzenia jest plik `manifest.json`, który określa kluczowe dla przeglądarki informacje o rozszerzeniu. Między innymi:

- nazwę,
- wersję,
- autora,
- opis,
- ikonę,
- uprawnienia rozszerzenia,
- skrypty, które mają być uruchamiane w różnych kontekstach przeglądarki,
- dostępne dla rozszerzenia zasoby,
- strony internetowe, na których rozszerzenie może być aktywne [27].

Przy załadowaniu rozszerzenia, przeglądarka odczytuje plik manifestu i na jego podstawie ładuje odpowiednie skrypty, zasoby, konfiguruje rozszerzenie i uruchamia je.

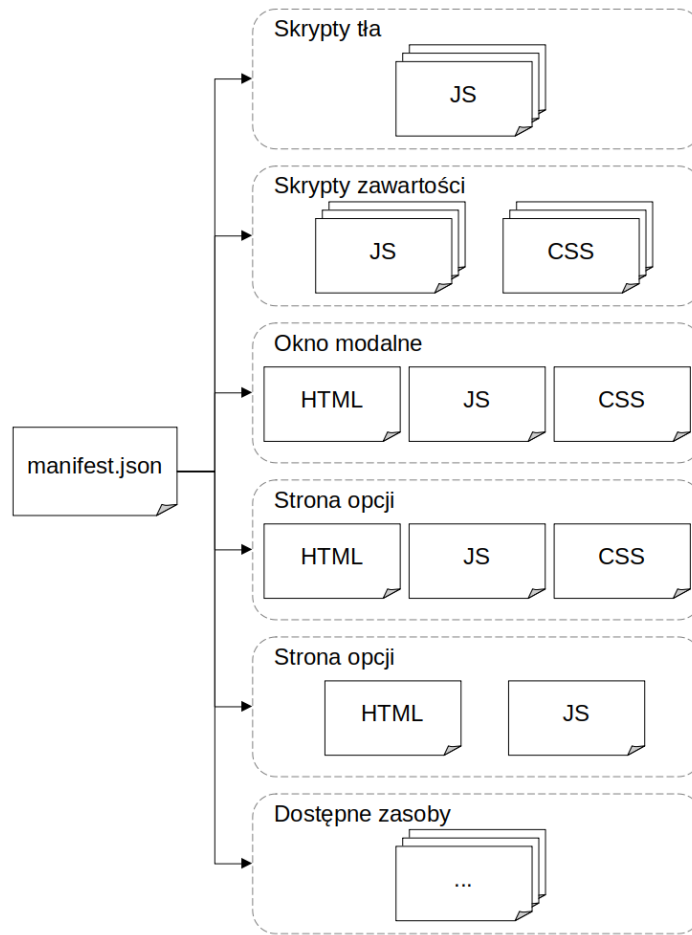
4.2. Anatomia rozszerzeń przeglądarkowych

Każde rozszerzenie przeglądarkowe składa się z kilku niezależnych skryptów, bądź dokumentów HTML (mogących zawierać skrypty), które uruchamiane są w różnych kontekstach, posiadają dostęp do różnych funkcjonalności API rozszerzeń Chrome (podrozdz. 4.3), interfejsów przeglądarki i odpowiadają za różne aspekty działania rozszerzenia. Wy różnić można:

- **skrypty tła** (ang. „background scripts”),
- **skrypty zawartości** (ang. „content scripts”),
- **okno modalne** (ang. „popup”),
- **strona opcji** (ang. „options page”),
- **narzędzia deweloperskie** (ang. „devtools”).

4.2.1. Skrypty tła

Skrypty tła uruchamiane są w momencie otwarcia przeglądarki i działają na podobnych zasadach do skryptów „service workers”. Operują w kontekście całej przeglądarki i nie mają bezpośredniego dostępu do wyświetlanych stron internetowych. Zwykle używane są do



Rysunek 4.1. Fragment anatomii rozszerzenia przeglądarkowego.

zarządzania komunikacją między innymi częściami rozszerzenia i wykonywania operacji, które nie wymagają bezpośredniej interakcji z użytkownikiem.

Ich działanie opiera się na reagowaniu na określone zdarzenia, takie jak np.: uruchomienie rozszerzenia, zamknięcie przeglądarki, czy wysłanie wiadomości z innego skryptu [28].

4.2.2. Skrypty zawartości

Skrypty zawartości uruchamiane są w kontekście strony internetowej, na której rozszerzenie jest aktywne. Mogą czytać i modyfikować zawartość strony internetowej korzystając ze standardowego DOM API oraz wpływać na wygląd i zachowanie strony internetowej. Skrypty te przeważnie uruchamiane są w momencie załadowania strony i działają do jej zamknięcia.

Skrypty zawartości mogą zostać uruchomione w dwóch różnych kontekstach - „światach wykonywania” (ang. „execution worlds”) [29]:

- **izolowanym** (ang. „isolated”) (domyślny) - skrypty te nie mają dostępu do tych samych właściwości globalnych obiektów strony (takich jak np. window czy document). Mogą natomiast komunikować się z resztą rozszerzenia za pomocą ograniczonego

API rozszerzeń przeglądarki. Także strona internetowa nie ma dostępu do samego skryptu zawartości,

- **głównym** (ang. „main”) - skrypty te mają pełny dostęp do obiektów globalnych strony, nie mają natomiast możliwości użycia interfejsu rozszerzeń [30].

4.2.3. Okno modalne

Okno modalne to dokument HTML, wyświetlany w formie wyskakującego okna, po kliknięciu ikony rozszerzenia w pasku przeglądarki. Przeważnie zawiera on interfejs użytkownika, który pozwala na interakcję z rozszerzeniem lub informujący o jego działaniu [31].

4.2.4. Strona opcji

Strona opcji to dokument HTML, wyświetlany w nowej karcie po przejściu do ustawień rozszerzenia. Przeważnie zawiera on interfejs użytkownika, który pozwala na konfigurację rozszerzenia [32].

4.2.5. Narzędzia deweloperskie

Narzędzia deweloperskie to dokument HTML używany w momencie otwierania okna narzędzi deweloperskich. Może zawierać odnośniki do skryptów w języku JavaScript, tworzących dodatkowe panele czy zakładki. Panele te i zakładki mogą wyświetlać swoje, niezależne dokumenty HTML. Przeważnie używane są do prezentacji interfejsu umożliwiającego inspekcję, czy manipulowanie elementami wyświetlonej strony internetowej [33].

4.3. API rozszerzeń Chrome

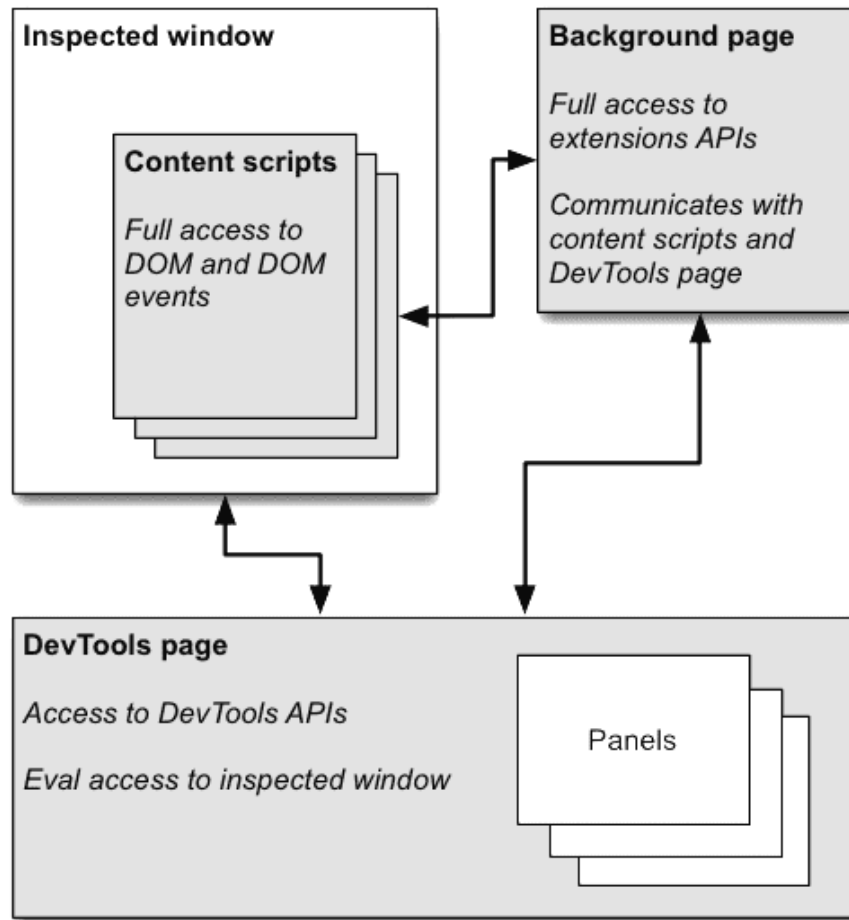
API rozszerzeń Chrome to przestrzeń nazw zawierająca liczne metody i właściwości pozwalające rozszerzeniom między innymi na interakcję z przeglądarką, stronami internetowymi, komunikację między skryptami rozszerzenia, zarządzanie pamięcią, czy dostęp do zasobów.

Funkcje i metody dostępne w API rozszerzeń Chrome są zróżnicowane i zależą od wielu czynników, takich jak: uprawnienia rozszerzenia zdefiniowane w manifeście (podrozdz. 4.1), czy kontekst w jakim jest wykonywanie wywołanie. Przykładowo, skrypty zawartości nie mogą zmienić ikony rozszerzenia poprzez API, ponieważ nie mają dostępu do `chrome.browserAction`, ale skrypt tła już tak [34].

Ze względu na fakt, że poszczególne skrypty rozszerzenia są uruchamiane w różnych, izolowanych kontekstach, koniecznym jest wykorzystanie mechanizmu, który umożliwia przesyłanie informacji między nimi.

W kontekście jednego rozszerzenia, sposoby komunikacji można podzielić na dwie kategorie:

- wiadomości jednorazowe,
- długotrwałe połączenia.



Rysunek 4.2. Dostępne interfejsy i sposoby komunikacji między fragmentami rozszerzenia [33].

4.3.1. Wiadomości jednorazowe

Wiadomości jednorazowe wysyłane są wywołując `chrome.runtime.sendMessage`, w przypadku gdy adresatem wiadomości jest skrypt uruchomiony w kontekście przeglądarki (np. skrypt tła, okno modalne), lub za pomocą funkcji `chrome.tabs.sendMessage`, gdy adresatem jest skrypt uruchomiony w kontekście określonej karty przeglądarki.

W drugim przypadku koniecznym jest podanie identyfikatora karty, do której ma zostać wysłana wiadomość.

Wysyłane w ten sposób wiadomości mogą dodatkowo zawierać funkcję wywołania zwrotnego (ang. *callback*), która może zostać wywołana przez adresata, w celu odesłania odpowiedzi.

Odbiorca, oczekujący na wiadomości, musi zarejestrować nasłuchiwanie na przychodzące wiadomości za pomocą funkcji `chrome.runtime.onMessage.addListener`. Próba wysłania wiadomości, gdy żadne nasłuchiwanie nie zostało zarejestrowane, skutkuje błędem.

4.3.2. Długotrwałe połączenia

Długotrwałe połączenia mogą być nawiązywane poprzez stworzenie kanału komunikacyjnego między dwoma skryptami za pomocą wywołań funkcji `chrome.runtime.connect` i

`chrome.tabs.connect`, odpowiednio dla połączeń ze skryptami uruchomionymi w kontekście przeglądarki i karty.

Jeżeli dojdzie do poprawnego nawiązania połączenia, powyższe funkcje zwracają obiekt `chrome.runtime.Port`, który umożliwia kontrolę nad kanałem (np. rozłączenie) oraz przesyłanie i nasłuchiwanie na nadchodzące wiadomości.

Odbiorca musi zarejestrować nasłuchiwanie na nadchodzące połączenie poprzez wywołanie funkcji `chrome.runtime.onConnect.addListener`, przekazując, jako argument, funkcję obsługującą nawiązanie połączenia. Funkcja ta zostanie wywołana przy każdym nawiązanym połączeniu z argumentem również będącym obiektem `chrome.runtime.Port` [35].

4.3.3. Persystencja danych w rozszerzeniach

Rozszerzenia przeglądarkowe posiadają możliwość zapisywania danych na kilka różnych sposobów:

- **lokalnie** (ang. „local”) - dane przechowywane są na urządzeniu, dostępne tylko dla danego rozszerzenia i usuwane w momencie odinstalowania rozszerzenia,
- **w sesji** (ang. „session”) - dane przechowywane są w pamięci przeglądarki i dostępne tylko w trakcie trwania sesji przeglądarki, czyli do momentu jej zamknięcia,
- **w sposób synchronizowany** (ang. „sync”) - dane przechowywane są na serwerach Google i dostępne są dla danego użytkownika na wszystkich urządzeniach, na których jest on zalogowany do przeglądarki Chrome,
- **w sposób zarządzany** (ang. „managed”) - opcja ta przeznaczona jest dla rozszerzeń zarządzanych przez administratora organizacji. Dane są dostępne tylko do odczytu. Sposób ten pozwala na skonfigurowanie rozszerzeń dla wszystkich użytkowników w organizacji.

Dostęp do przechowywanych danych uzyskuje się za pomocą API `chrome.storage`. Wymaga on zdefiniowania uprawnień w pliku manifestu, w sekcji `permissions`.

5. Analiza działania biblioteki React

Jak wspomniano w podrozdz. 3.4, jedną z analizowanych w pracy bibliotek jest React. Aby móc stworzyć narzędzie, pozwalające na analizę aplikacji stworzonych z jego użyciem, nieodzowne jest dokładne zrozumienie sposobu jego działania.

W tym rozdziale przedstawiony zostanie rezultat analizy funkcjonalności i mechanizmów działania biblioteki, wraz z najważniejszymi pojęciami i mechanizmami, które będą wykorzystane w dalszych częściach pracy.

Przedstawiony opis dotyczy obecnie najnowszej wersji Reacta, czyli 18.2.0, wydanej 14 czerwca 2022 roku [36]. Wszystkie nawiązania do kodu źródłowego zarówno Reacta, jak i React Developer Tools, dotyczą gałęzi „main”, aktualnych na dzień 18 sierpnia 2023 (ostatni „commit” do repozytorium oznaczony jest haszem 98f3f14 [37]).

5.1. Co to jest React?

React to biblioteka służąca do tworzenia interfejsów użytkownika. Dzięki swojej wydajności i prostocie użycia stała się, i przez długi czas pozostaje, jednym z najpopularniejszych narzędzi swojego rodzaju.

React opiera swoje działanie na komponentach, które wydzielają logikę, funkcjonalności i wygląd poszczególnych fragmentów aplikacji. Z tych komponentów, finalnie budowana jest cała struktura aplikacji. Takie podejście pozwala na ponowne używanie fragmentów kodu, większą modularność, a w efekcie łatwiejsze zarządzanie i rozwijanie złożonych projektów [6].

5.2. Komponenty w Reaccie

Komponenty to podstawowe, pozwalające na wielokrotne używanie, elementy budulcowe aplikacji React. Każdy komponent reprezentuje określony fragment interfejsu użytkownika. Komponenty mogą być wzajemnie zagnieżdżane w celu budowania bardziej złożonych komponentów, definiować swój stan (patrz podrozdz. 5.9.1) oraz przyjmować dane wejściowe w postaci właściwości (patrz podrozdz. 5.3).

Komponenty w React mogą być definiowane na dwa sposoby: jako funkcje (tzw. „komponenty funkcyjne”) lub jako klasy (tzw. „komponenty klasowe”). Choć oba sposoby są wspierane, obecnie twórcy Reacta zalecają stosowanie prostszych w użyciu komponentów funkcyjnych. Z tego powodu, w tej pracy, wszystkie opisy dotyczyć będą komponentów funkcyjnych, chyba że zaznaczono inaczej.

Komponenty funkcyjne definiowane są jako funkcje JavaScript, zwracające elementy JSX (patrz podrozdz. 5.4). Przykładem takiego komponentu może być:

Listing 1. Definicja przykładowego komponentu funkcyjnego w React.

```
function ExampleComponent() {  
  return <div>Hello, World!</div>;  
}
```

Taki komponent może być wykorzystany w innych komponentach, budując drzewiastą strukturę aplikacji.

Listing 2. Użycie komponentu w innym komponencie.

```
1 function MyComponent() {
2   return (
3     <div>
4       <h1>My Component</h1>
5       <ExampleComponent />
6     </div>
7   );
8 }
```

5.3. Właściwości komponentów

Właściwości (ang. „props”) pozwalają na przekazywanie informacji z komponentu nadrzędnego do komponentu podrzędnego. Każdy komponent może definiować przyjmowane właściwości jako parametry funkcji, będące dowolnymi wartościami JavaScript [38].

5.4. JavaScript XML (JSX)

JSX to rozszerzenie składni języka JavaScript, pozwalające na definiowanie struktury drzewa DOM. Dzięki JSX, kod źródłowy aplikacji jest bardziej czytelny i łatwiejszy w utrzymaniu.

JSX nie jest jednak zrozumiały dla przeglądarek internetowych. W procesie budowania aplikacji, kod źródłowy jest transpilowany do zagnieżdżonych wywołań funkcji JavaScript. Tak na przykład, wartość zwracana przez przedstawiony w list. 1 komponent, zostanie ostatecznie przekształcona do postaci:

Listing 3. Przekształcenie elementu JSX.

```
React.createElement("div", {}, "Hello, World!");
```

Natomiast, użyty w list. 2 w linii 5 komponent `ExampleComponent`, zostanie przekształcony do:

Listing 4. Przekształcenie komponentu JSX.

```
React.createElement(ExampleComponent, {});
```

[39]

5.5. Drzewo aplikacji

Opisane w podrozdz. 5.4 wywołania `React.createElement`, tworzą obiekty JavaScript, używane przez Reacta do określenia struktury drzewa DOM. Tak na przykład, wywołanie z list. 3, stworzy obiekt (pominięte zostały nieistotne dla przykładu właściwości):

Listing 5. Obiekt reprezentujący element `<div>` (przykład uproszczony).

```
{
  type: "div",
  props: {
    children: "Hello, World!"
  }
}
```

Natomiast wywołanie z list. 4 stworzy obiekt:

Listing 6. Obiekt reprezentujący komponent `ExampleComponent` (przykład uproszczony).

```
{
  type: f ExampleComponent(),
  props: {}
}
```

Aby wyświetlić aplikację Reacta należy „zaczepić” ją do określonego elementu w drzewie DOM.

Listing 7. Zaczepienie aplikacji Reacta do elementu DOM.

```
ReactDOM.createRoot(document.getElementById("root")).render(
  <MyComponent />
);
```

Po wywołaniu `render`, podając jako argument korzeń drzewa aplikacji, React stopniowo tworzy strukturę z obiektów zwróconych przez `React.createElement`. Jeżeli typem danego elementu jest funkcja (w przypadku komponentów funkcyjnych), bądź klasa (w przypadku komponentów klasowych), React wywołuje tę funkcję, bądź tworzy instancję klasy (i wywołuje funkcję `render`), a wynik tych operacji dodaje do listy dzieci elementu nadrzędnego.

Obiektem utworzonym finalnie przy wywołaniu `render` z przykładu list. 7 będzie:

Listing 8. Obiekt reprezentujący strukturę aplikacji (przykład uproszczony).

```
{
  type: "div",
  props: {
    children: [
      {
        type: "h1",
        props: {
          children: "MyComponent"
        }
      },
      {
        type: "div",
        props: {
```

```
      children: "Hello ,␣World!"  
    }  
  }  
]  
}  
}
```

Proces budowania takiego drzewa zaczyna się przy wywołaniu `render` i jest powtarzany przy zmianach stanów (podrozdz. 5.9) oraz właściwości (podrozdz. 5.3) komponentów [40].

Dla uproszczenia, przedstawione w powyższych przykładach obiekty przypominają bardziej te, używane przez Reacta przed wersją 16. Obecnie wykorzystywane struktury są bardziej złożone i oparte o architekturę nazwaną „Fiber” (podrozdz. 5.8).

5.6. Rekonceiliacja, czyli wykrywanie zmian

Po zbudowaniu drzewa aplikacji (podrozdz. 5.5), React zapisuje wynik w pamięci i wysyła do właściwego środowiska zajmującego się renderowaniem (podrozdz. 5.7). Po zmianach stanów komponentów, React ponawia proces budowania drzewa i porównuje wynik z poprzednim, w celu określenia w jaki sposób zaktualizować widoczną aplikację (w przypadku przeglądarki - drzewo DOM). Proces ten nazywany jest rekonceiliacją (ang. „reconciliation”), a zapisywane w pamięci drzewo, często określane jest jako wirtualny DOM (ang. „virtual DOM”). Rekonceiliacja pozwala uniknąć każdorazowego renderowania całej aplikacji [41].

5.7. Renderowanie widoku aplikacji

Procesy rekonceiliacji (podrozdz. 5.6) i renderowania, są w Reaccie od siebie oddzielone. Rekonceiliacja niejako określa co uległo zmianie, a renderowanie - w jaki sposób te zmiany mają zostać zastosowane. Rozwiązanie to pozwala na użycie wspólnego rdzenia Reacta dla różnych platform, takich jak: przeglądarki internetowe, aplikacje mobilne, czy aplikacje desktopowe, zmieniając jedynie warstwę renderującą [41].

Część Reacta, odpowiedzialna za opisany proces, z powodu braku odpowiedniego tłumaczenia, nazywana będzie „rendererem”.

W pozostałej części tej pracy pojęcie „renderowanie komponentu” nie będzie odnosiło się do właściwego procesu opisanego w tym podrozdziale, lecz będzie oznaczało każdorazowe wywołanie funkcji definiującej komponent.

5.8. React Fiber

Przedstawiony wyżej opis działania Reacta został znacząco uproszczony. W rzeczywistości, React korzysta z bardziej złożonego mechanizmu opartego o strukturę danych i architekturę nazwaną „Fiber”. Pozwala ona na reprezentację komponentów w charakterze ramek odkładanych i ściąganych ze stosu w zależności od ich priorytetu. Dzięki temu

React może decydować o odłożeniu niektórych operacji na później, bądź całkowitemu ich pominięciu, jeżeli nie są one konieczne [41].

5.8.1. Struktura obiektu Fiber

Obiekty Fiber są złożone z wielu właściwości. Część z nich zostanie opisana w dalszych częściach tej pracy. Najważniejsze są jednak:

- `type: string | Function | Symbol | null` - podobnie jak w podrozdz. 5.5, powiązanie z danym węzłem drzewa aplikacji. Może to być nazwa elementu HTML czy funkcja komponentu,
- `child: Fiber | null` - referencja do pierwszego dziecka danego węzła (np. obiektu Fiber reprezentującego element zwracany przez funkcję komponentu),
- `sibling: Fiber | null` - referencja do kolejnego węzła na tym samym poziomie drzewa,
- `alternate: Fiber | null` - referencja do obiektu Fiber reprezentującego ten sam węzeł podczas poprzedniego cyklu budowania drzewa,
- `stateNode` (obiekt opisany w podrozdz. 9.7.4) - referencja do rzeczywistego elementu DOM, bądź instancji komponentu,
- `tag: number` - określa typ węzła (np. komponent funkcyjny, klasowy, element HTML),
- `memoizedState` (obiekt opisany w podrozdz. 9.7.7) - buforowany stan komponentu,
- `memoizedProps` (obiekt opisany w podrozdz. 9.7.7) - buforowane właściwości komponentu,
- `dependencies` (obiekt opisany w podrozdz. 9.7.7) - lista wiązana wykorzystywanych w komponencie wartości kontekstów.

Istotnym dla późniejszej implementacji rozwiązania jest fakt, że obiekty Fiber nie przechowują listy dzieci w tablicy, lecz w postaci listy jednokierunkowej. Uproszczona struktura z przykładu list. 2, jako drzewo Fiber, przedstawiona została na rys. 5.1.

Na podstawie porównania danego obiektu Fiber i jego `alternate`, możliwym jest określenie zmian dokonanych dla konkretnego węzła w danym cyklu budowania drzewa [42].

5.9. Hooki - reaktywność w komponentach funkcyjnych

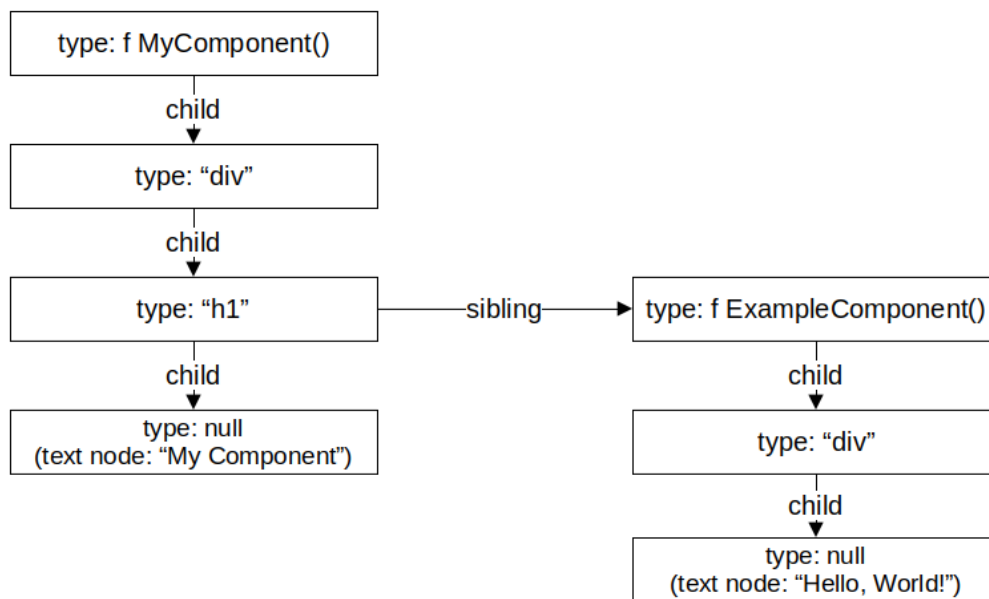
Hooki (ang. „hooks”) to mechanizm pozwalający na wykorzystanie w komponentach funkcyjnych różnych funkcjonalności Reacta. Są to funkcje wbudowane w bibliotekę, których użycie polega na wywołaniu ich w ciele funkcji definiującej komponent [43].

5.9.1. Hook `useState`

Jednym z najczęściej używanych hooków jest `useState`. Używany jest do „zapamiętywania” danych pomiędzy kolejnymi renderowaniami (podrozdz. 5.7) komponentu - zdefiniowania lokalnego stanu. Przykładem użycia może być:

Listing 9. Przykład użycia hooka `useState` [44].

```
export function Counter() {
  const [count, setCount] = useState(0);
```



Rysunek 5.1. Przykładowa struktura drzewa Fiber.

```

return (
  <div>
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>
      Increment
    </button>
  </div>
);
}

```

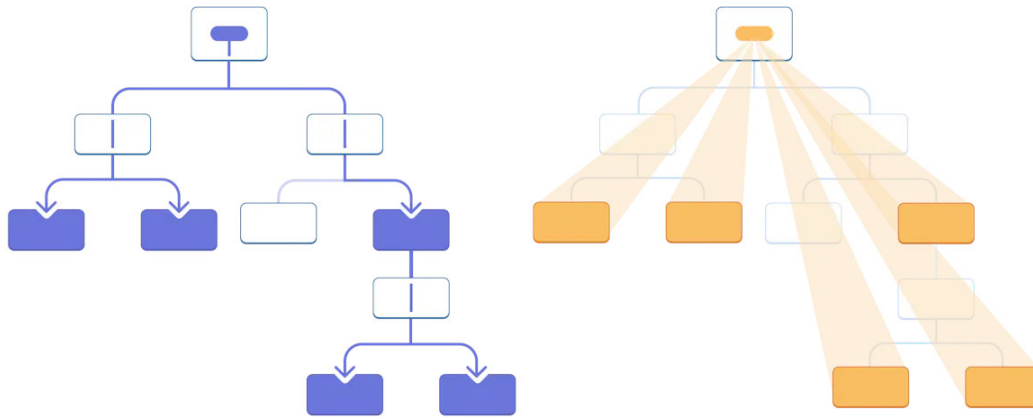
Wywołanie `useState` zwraca aktualną wartość danego fragmentu stanu oraz funkcję pozwalającą na zmianę tej wartości [44].

5.9.2. Hook `useReducer`

`useReducer` podobnie jak `useState` pozwala na zarządzanie stanem komponentu. Wywołanie tego hooka nie zwraca jednak funkcji pozwalającej na bezpośrednie ustawianie wartości stanu, a funkcję redukującą, która umożliwia wysłanie (ang. „dispatch”) uprzednio zdefiniowanych akcji. Logika zmian stanu w reakcji na konkretne zdarzenia zostaje wówczas skonsolidowana i wyniesiona poza ciało komponentu. Rozwiązanie to jest przeważnie stosowane w przypadku bardziej złożonych stanów [45].

5.9.3. Hook `useContext`

`useContext` pozwala na subskrybowanie do wartości kontekstu, zdefiniowanego w którymkolwiek z komponentów nadrzędnych. Takie rozwiązanie pozwala na uniknięcie tzw. „prop drilling”, czyli zbędnego przekazywania danych przez niewykorzystujące ich komponenty pośrednie [46][47].



Rysunek 5.2. Porównanie przekazywania danych przez „prop drilling” (po lewej) i przez kontekst (po prawej) w Reaccie [47].

5.9.4. Hook useRef

Podobnie jak `useState`, `useRef` umożliwia zapamiętywanie danych pomiędzy kolejnymi renderowaniami komponentu. Różnica polega jednak na tym, że zmiana wartości referencji nie powoduje ponownego renderowania komponentu. Z tego powodu hook ten przeważnie wykorzystywany jest do przechowywania danych, które nie są bezpośrednio potrzebne do renderowania (np. referencje do elementów DOM), bądź przy korzystaniu z innych wbudowanych interfejsów przeglądarki [48].

5.9.5. Hook useEffect

`useEffect` służy do obsługi „efektów ubocznych” w komponentach funkcyjnych, takich jak np. pobieranie danych z serwera czy obsługa nasłuchiwanie na zdarzenia przeglądarki. Hook ten przyjmuje dwa argumenty: funkcję, która będzie wywoływana oraz tablicę zależności. Przekazana funkcja wywoływana jest za każdym razem, gdy którakolwiek z wartości w tablicy zależności ulegnie zmianie. Podanie pustej tablicy skutkuje jednokrotnym wywołaniem funkcji po pierwszym renderowaniu komponentu. `useEffect` pozwala również na zwrócenie funkcji czyszczącej, która wywoływana jest przed kolejnym wywołaniem funkcji efektu, bądź po odmontowaniu komponentu [49]. Przykładem użycia może być:

Listing 10. Przykład użycia hooka `useEffect` [49].

```
function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(
      "https://localhost:1234",
      roomId
    );
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [roomId]);
}
```

```
// ...  
}
```

5.9.6. Hook `useMemo`

Hook `useMemo` pozwala na uniknięcie ponownego obliczania określonej wartości, jeżeli wartości, od których zależy, nie uległy zmianie. Hook ten jest najczęściej wykorzystywany w celu optymalizacji wydajności aplikacji, gdy obliczenia są kosztowne, lub gdy wynik używany jest jako zależność użycia innego hooka. W drugim przypadku, `useMemo` może zapobiec ponownemu wywołaniu innych hooków (lub renderowaniu komponentu), jeżeli jego zależności się nie zmieniły [50]. Przykładem użycia może być:

Listing 11. Przykład użycia hooka `useMemo` [50].

```
function TodoList({ todos, tab }) {  
  const visibleTodos = useMemo(  
    () => filterTodos(todos, tab),  
    [todos, tab]  
  );  
  // ...  
}
```

W powyższym przykładzie (list. 11) `visibleTodos` zostanie ponownie obliczone tylko wtedy, gdy zmieni się `todos` lub `tab`.

5.9.7. Hook `useCallback`

Hook `useCallback` działa podobnie do `useMemo` i posiada analogiczne zastosowanie. Różnica polega na tym, że `useCallback` używany jest do buforowania funkcji, a nie wartości.

5.9.8. Hook `useDebugValue`

`useDebugValue` pozwala na dostosowanie etykiety wyświetlanej w React Developer Tools (rozdz. 7) dla niestandardowych hooków. Przykład użycia pokazany zostanie w podrozdz. 5.9.10.

5.9.9. Inne hooki

Oprócz wymienionych powyżej, React oferuje także inne (rzadziej używane) hooki. Są to między innymi:

- `useImperativeHandle` - pozwala na dostosowanie interfejsu udostępnianego przez „ref” do komponentu nadrzędnego,
- `useLayoutEffect` - działa podobnie do `useEffect`, ale wywoływany jest przed i synchronicznie blokuje renderowanie komponentu,
- `useInsertionEffect` - pozwala na dodawanie elementów do drzewa DOM (przeważnie styli CSS) i wywoływany jest przed innymi efektami,

- `useTransition` - pozwala na nieblokujące aktualizowanie stanu komponentu. Używany, gdy jakaś operacja jest czasochłonna, bądź po to, by pozwolić na przerwanie aktualizacji stanu (często używany w animacjach),
- `useDeferredValue` - pozwala na opóźnienie aktualizacji niekrytycznej części interfejsu aplikacji. React zwraca poprzednią wartość, następnie aktualizuje ją w tle i po zakończeniu operacji renderuje komponent ponownie,
- `useId` - generuje dla komponentu unikalny identyfikator (przeważnie używany dla celów zwiększenia dostępności w aplikacji),
- `useSyncExternalStore` - pozwala na subskrypcję do zewnętrznego magazynu danych [43].

5.9.10. Niestandardowe hooki

Niestandardowe hooki to funkcje, które umożliwiają wydzielenie fragmentu logiki komponentu i użycie jej jako hooka w wielu miejscach aplikacji. Zawierają one wywołania innych wbudowanych hooków [51]. Przykładem może być:

Listing 12. Przykład definicji niestandardowego hooka `useToggle`.

```
1 function useToggle(initialValue = false) {
2   const [value, setValue] = useState(initialValue);
3   useDebugValue(value ? "on" : "off");
4   const toggle = useCallback(() => setValue(v => !v), []);
5   return [value, toggle];
6 }
```

Listing 13. Przykład użycia niestandardowego hooka `useToggle`.

```
function MyComponent() {
  const [isOn, toggle] = useToggle();
  return (
    <button onClick={toggle}>
      {isOn ? "ON" : "OFF"}
    </button>
  );
}
```

Powyższy przykład pokazuje uproszczenie (i tak już prostego) mechanizmu przełączania wartości logicznej w komponencie. Użyty w list. 12 w linii 3 `useDebugValue` (opisany w podrozdz. 5.9.8), nie wpływa na zachowanie hooka, ale modyfikuje sposób w jaki hook jest wyświetlany w RDT (rys. 5.3).

HooksComponent	HooksComponent
props new entry: ""	props new entry: ""
hooks ▶ Toggle: "off"	hooks ▶ Toggle:
rendered by ReactApp createRoot() react-dom@18.2.0	rendered by ReactApp createRoot() react-dom@18.2.0

Rysunek 5.3. Wyświetlanie niestandardowego hooka w React Developer Tools z `useDebugValue` (po lewej) i bez (po prawej).

6. Analiza działania biblioteki Svelte

Svelte podobnie do Reacta jest biblioteką służącą do tworzenia interaktywnych aplikacji frontendowych, zbudowanych z komponentów. Stworzony został przez Richa Harrisa, a jego pierwsza wersja została wydana w listopadzie 2016 roku.

6.1. Założenia biblioteki

Svelte w przeciwieństwie do innych popularnych bibliotek frontendowych, takich jak React czy Vue, opiera się na kompilatorze, który przekształca specyficzny dla siebie dialekt HTML (podrozdz. 6.2) w JavaScript, który bezpośrednio manipuluje DOM-em, zapewnia obsługę zdarzeń wymagających ponownego przeliczenia wartości i aktualizacji widoku. W rezultacie, stworzona aplikacja, uruchomiona w przeglądarce, nie wymaga importowania logiki z samej biblioteki.

Biblioteka Svelte nie używa pośrednich reprezentacji stanu aplikacji, takich jak wirtualny DOM. Przenosi ona większość wykonywanej pracy ze środowiska uruchomieniowego (ang. „runtime”) na etap kompilacji. W efekcie, aplikacje napisane w Svelte są zwykle bardziej wydajne i zajmują mniej miejsca w pamięci, niż te stworzone przy użyciu „klasycznych” bibliotek [52].

6.2. Składnia Svelta

Język Svelte to HTML rozszerzony o składnie szablonów opartych o JavaScript. Swoją strukturą przypomina JSX (podrozdz. 5.4) znany z Reacta.

Listing 14. Przykładowa definicja komponentu w Svelte.

```
1  <!-- App.svelte -->
2  <script>
3    import Example from "../Example.svelte"
4
5    let count = 1;
6    $: doubled = count * 2;
7  </script>
8
9  <p>{count} * 2 = {doubled}</p>
10 <button on:click={() => count = count + 1}>Count</button>
11 <Example name="world" count={count}/>
12
13 <style>
14   p {
15     background-color: goldenrod;
16   }
17 </style>
```

Listing 15. Przykładowa definicja komponentu w Svelte.

```
1  <!-- Example.svelte -->
2  <script>
3    export let name;
4    export let count;
5  </script>
6
7  <p>Hello {name}!</p>
8  {#if count >= 5}
9    <p>Count is high</p>
10 {:else}
11   <p>Count is low</p>
12 {/if}
```

W list. 14 i w list. 15 przedstawiono przykładowe definicje komponentów w Svelte. Komponenty w Svelte składają się z trzech części, a każda z nich jest opcjonalna. Są to:

1. **Szablon** - zawiera definicję struktury elementów HTML, które mają zostać wyrenderowane w przeglądarce. Wyróżnić można kilka elementów:
 - list. 14 (linia 9) - definicja paragrafu używającego wartości zmiennej `count` oraz `doubled`. Użycie nawiasów klamrowych `{}` pozwala na wstawienie wartości zmiennej do szablonu,
 - list. 14 (linia 10) - definicja przycisku, który po kliknięciu zwiększa wartość zmiennej `count` o 1. Atrybut `on:click` pozwala na zdefiniowanie obsługi zdarzenia kliknięcia,
 - list. 14 (linia 11) - użycie komponentu `Example` z przekazaniem wartości `name` oraz `count`,
 - list. 15 (linia 8) - specyficzne dla Svelte bloki logiczne (podrozdz. 6.4) pozwalające na warunkowe wyświetlenie elementów. W tym przypadku, w zależności od wartości zmiennej `count` wyświetlany jest odpowiedni paragraf.
2. **Skrypt** - zawarty jest między znacznikami `<script>` i `</script>` i definiuje logikę komponentu. Można zwrócić uwagę na:
 - list. 14 (linia 3) - import komponentu `Example` z pliku `Example.svelte`,
 - list. 14 (linia 5) - definicja lokalnego, reaktywnego stanu poprzez zmienną `count`,
 - list. 14 (linia 6) - definicja reaktywnego wyrażenia pochodnego, które przeliczane jest za każdym razem, gdy wartość reaktywnej zmiennej `count` ulegnie zmianie. W Svelte każde takie wyrażenie poprzedzone jest etykietą `$` [53],
 - list. 15 (linia 3 i 4) - definicja właściwości komponentu, których wartości mogą być przekazane z komponentu nadrzędnego. W Svelte właściwości komponentu definiowane są za pomocą słowa kluczowego `export`.
3. **Styl** - zawarty jest między znacznikami `<style>` i `</style>` i pozwala na zdefiniowanie stylów CSS dla komponentu.

[54]

6.3. Svelte - inne podejście do reaktywności

Zmiana stanu w Svelte i uruchomienie procesu aktualizacji widoku, w przeciwieństwie do Reacta, nie wymaga od programisty użycia hooków czy innych funkcji importowanych z biblioteki.

Dzięki wykorzystaniu kompilatora, Svelte transformuje kod źródłowy, wstrzykując obsługę reaktywności w odpowiednie miejsca. W efekcie, zwyczajne przypisanie nowej wartości do zmiennej, powoduje automatyczne przeliczenie odpowiednich wartości i aktualizację zależnych fragmentów widoku [54].

6.4. Bloki logiczne

Svelte pozwala na wykorzystywanie specjalnych bloków logicznych, które wpływają na sposób wyświetlania elementów w szablonie. Użycie bloków logicznych składa się z co najmniej dwóch znaczników: otwierającego `{#...}` i zamykającego `{\...}` oraz ewentualnych bloków kontynuacji `{:...}` (w miejscu kropek użyta zostaje odpowiednia nazwa bloku) [55].

6.4.1. Blok `{#if...}`

Blok `{#if...}` pozwala na warunkowe wyświetlanie elementów w zależności od wartości wyrażenia.

Listing 16. Składnia bloku `{#if...}` w Svelte.

```
{#if expression}...{/if}
{#if expression}...{:else}...{/if}
{#if expression}...{:else if expression}...{/if}
```

W miejscu `expression` użyte może być dowolne wyrażenie, które zostanie przeliczone na wartość logiczną. Jeżeli wartość ta będzie prawdziwa, elementy znajdujące się wewnątrz bloku zostaną wyświetlone. W przeciwnym wypadku, zostanie obliczona wartość kolejnych wyrażeń w blokach `{:else if...}` (jeżeli są obecne). Jeżeli żadne z wyrażeń nie będzie prawdziwe, wyświetlona zostanie zawartość bloku `{:else}` (jeżeli jest obecny) [55].

6.4.2. Blok `{#each...}`

Blok `{#each...}` pozwala na wyrenderowanie określonych elementów dla każdego elementu w danej kolekcji.

Listing 17. Składnia bloku `{#each...}` w Svelte.

```
{#each expression as item}...{/each}
{#each expression as item, index}...{/each}
{#each expression as item (key)}...{/each}
{#each expression as item, index (key)}...{/each}
{#each expression as item}...{:else}...{/each}
```

W miejscu `expression` użyte powinno zostać wyrażenie, które obliczone zostanie do „tablico-podobnej” (ang. „array-like”) wartości. Wewnątrz bloku dostępne są zmienne `item` (wartość danego elementu kolekcji) oraz opcjonalnie `index` (indeks danego elementu).

W przypadku zmian w kolekcji, Svelte domyślnie dodaje i usuwa elementy na końcu listy. Takie rozwiązanie zwiększa wydajność i pozwala na ponowne używanie elementów między przerysowaniami komponentu. Aby uniknąć tego zachowania, możliwym jest przekazanie dodatkowo klucza (`key`), który unikatowo identyfikuje każdy z elementów w kolekcji. Na podstawie jego wartości, Svelte jest w stanie efektywnie wykrywać i odpowiednio reagować na zmiany w dowolnym miejscu listy.

Jeżeli kolekcja jest pusta, wyświetlona zostanie zawartość bloku `{:else}` (jeżeli jest obecny) [55].

Listing 18. Przykład użycia bloku `{#each...}` w Svelte.

```
<script>
  let items = [
    {name: "A", id: 0},
    {name: "B", id: 1},
    {name: "C", id: 2}
  ];
</script>

{#each items as item, i (item.id)}
  <p>{i}: {item.name}</p>
{/each}
```

Listing 19. Rezultat użycia bloku `{#each...}` w drzewie DOM.

```
<p>0: A</p>
<p>1: B</p>
<p>2: C</p>
```

6.4.3. Blok `{#await...}`

Blok `{#await...}` pozwala na warunkowe wyświetlanie elementów w zależności od stanu obietnicy (ang. „promise”).

Listing 20. Składnia bloku `{#await...}` w Svelte.

```
{#await expression}...{:then value}...{:catch error}...{/await}
{#await expression}...{:then value}...{/await}
{#await expression then value}...{/await}
{#await expression catch error}...{/await}
```

W miejscu `expression` użyte powinno zostać wyrażenie, którego wynikiem jest obietnica. Obietnica w JavaScript może być jednym z trzech stanów:

- oczekująca (ang. „pending”) - obietnica nie została jeszcze zakończona ani odrzucona. W takim wypadku wyświetlona zostanie zawartość bloku zaraz po rozpoczynającym znaczniku `{#await...}`,
- spełniona (ang. „fulfilled”) - obietnica została zakończona sukcesem. W takim wypadku wyświetlona zostanie zawartość bloku `{:then...}`,
- odrzucona (ang. „rejected”) - obietnica zakończyła się niepowodzeniem. W takim wypadku wyświetlona zostanie zawartość bloku `{:catch...}`.

W blokach `{:then...}` oraz `{:catch...}` dostępne są odpowiednio zmienne: `value` (wartość zwrócona przez obietnicę) oraz `error` (błąd zwrócony przez obietnicę).

Warto zauważyć, że trzeci oraz czwarty wariant składni, przedstawiony w list. 20, nie definiuje elementów, które miałyby być wyświetlone w czasie oczekiwania na zakończenie obietnicy. Wówczas wyświetlane są elementy tylko w przypadku spełnienia lub odrzucenia obietnicy [55] [56].

6.4.4. Blok `{#key...}`

Blok `{#key...}` pozwala na kontrolowane niszczenie i tworzenie elementów w zależności od zmiany wartości określonego klucza.

Listing 21. Składnia bloku `{#key...}` w Svelte.

```
{#key expression}...{/key}
```

Zawartość bloku `{#key...}` zostanie zniszczona i ponownie utworzona za każdym razem, gdy zmieni się wartość wyrażenia `expression`. Blok ten pozwala między innymi na ponowne uruchamianie animacji wejścia określonych elementów [55] [57].

6.5. Inne funkcjonalności

Svelte posiada także wiele innych funkcjonalności, takich jak funkcje umożliwiające reakcję na zdarzenia związane z cyklem życia komponentów, wiązanie danych z atrybutami elementów drzewa DOM, dynamiczne komponenty czy wbudowane mechanizmy do obsługi animacji. Są to jednak funkcje, których nie opisano w tej pracy, gdyż nie są one szczególnie istotne w kontekście analizowanego problemu.

7. Analiza implementacji React Developer Tools

Jak już wspomniano w podrozdz. 2.1.1, React Developer Tools (RDT) to zestaw narzędzi deweloperskich stworzonych przez twórców Reacta. W tym rozdziale opisana zostanie implementacja narzędzia, które posłużyło za inspirację do stworzenia rozwiązania opisanego w tej pracy.

Całość opisu implementacji RDT oparta jest na autorskiej analizie kodu źródłowego narzędzia, dostępnego publicznie na platformie GitHub. Najnowszy „commit” analizowanego kodu pochodzi z dnia 18 sierpnia 2023 i oznaczony jest haszem 98f3f14 [37].

Jako że RDT jest narzędziem rozbudowanym, przystosowanym do pracy z aplikacjami Reacta uruchamianymi w różnych kontekstach (klasyczne aplikacje przeglądarkowe, React Native, przeglądarki mobilne czy aplikacje desktopowe stworzone np. przy użyciu Electron [58]), kod źródłowy RDT został podzielony na kilka modułów.

- `react-devtools-core` [59] - moduł zawierający niskopoziomą logikę narzędzi deweloperskich, przygotowaną do obsługi innych kontekstów uruchomienia, niż przeglądarka internetowa (np. React Native),
- `react-devtools-extension` [60] - moduł zawierający logikę specyficzną dla RDT uruchamianego jako rozszerzenie przeglądarkowe dla Chrome, Firefox i Edge,
- `react-devtools-inline` [61] - moduł pozwalający na uruchomienie RDT w tzw. „placach zabaw kodu źródłowego” (ang. „source code playgrounds”) takich jak CodeSandbox [62], StackBlitz [63] czy Replay [64],
- `react-devtools-shared` [65] - moduł zawierający kod wspólny dla wyżej wymienionych modułów,
- `react-devtools-shell` [66] - moduł pozwalający na uruchomienie testów dla `react-devtools-inline` i `react-devtools-shared`,
- `react-devtools` [67] - moduł wykorzystujący logikę `react-devtools-core` i oferujący interfejs użytkownika dla RDT uruchamianego poza przeglądarką,
- `react-debug-tools` [68] - moduł zawierający eksperymentalne narzędzia wykorzystywane np. w RDT do analizy hooków (podrozdz. 5.9) komponentów funkcyjnych.

Z uwagi na fakt, że przedmiotem tej pracy było stworzenie rozszerzenia przeglądarkowego, to analizę kodu źródłowego RDT rozpoczęto od punktów wejścia dla modułu `react-devtools-extension`, zagłębiając się w kolejnych krokach w kod źródłowy innych części rozwiązania. W podrozdziałach, przedstawionych w dalszej części pracy, opisano implementację RDT w analogicznej sekwencji.

Jeżeli nie zaznaczono inaczej, wszystkie opisy dotyczą implementacji RDT w wersji przeznaczonej dla klasycznych aplikacji webowych, uruchamianych w przeglądarce Chrome.

7.1. Punkty wejścia dla rozszerzenia przeglądarkowego

Jak wspomniano w podrozdz. 4.1, metadane o rozszerzeniach przeglądarkowych znajdują się w pliku `manifest.json`. W pliku tym muszą być zdefiniowane punkty wejścia dla rozszerzenia, które przeglądarka wykorzystuje w celu ich uruchomienia.

Listing 22. Fragment manifestu rozszerzenia React Developer Tools.

```

{
  // ...
  "action": {
    // ...
    "default_popup": "popups/disabled.html"
  },
  "devtools_page": "main.html",
  "background": {
    "service_worker": "build/background.js"
  },
  "content_scripts": [{
    // ...
    "js": [
      "build/prepareInjection.js"
    ],
    "run_at": "document_start"
  }]
}

```

Zgodnie z zawartością pliku, punktami wejścia dla rozszerzenia są:

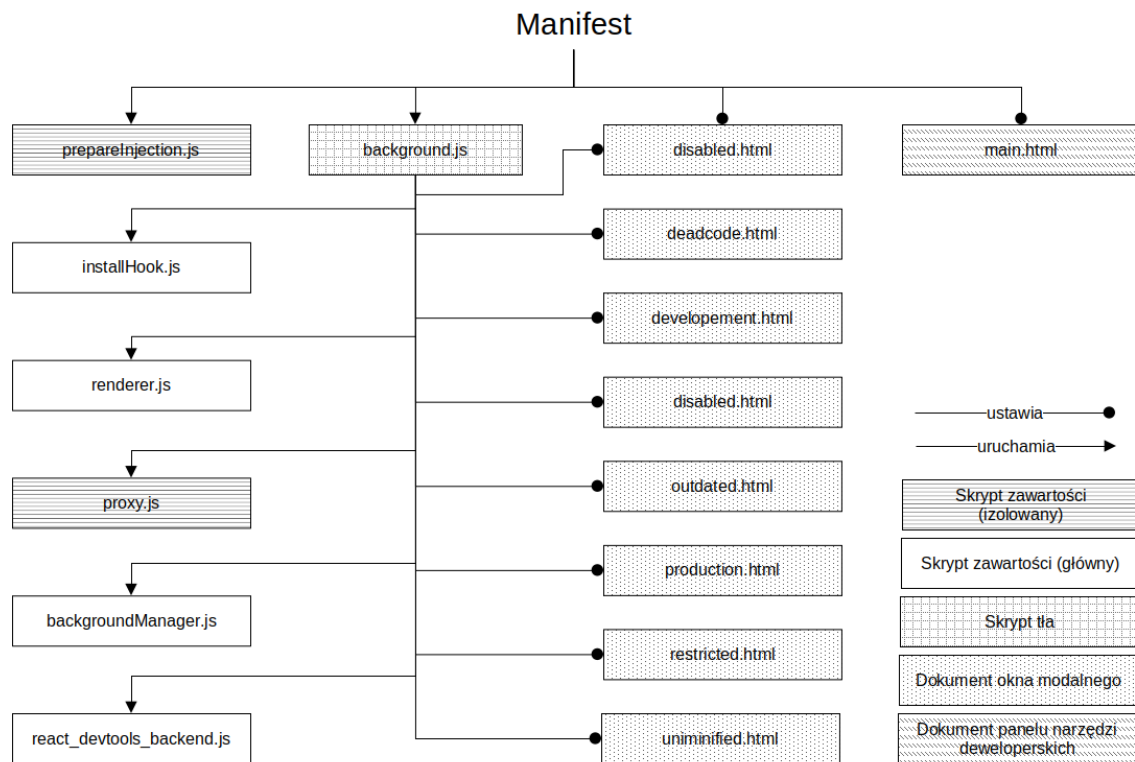
- `popups/disabled.html` - dokument wyświetlany jako okno modalne po kliknięciu w ikonę rozszerzenia w pasku narzędzi przeglądarki,
- `main.html` - dokument wyświetlany w panelu deweloperskim przeglądarki,
- `build/background.js` - skrypt „service worker” uruchamiany w tle przeglądarki,
- `build/prepareInjection.js` - skrypt uruchamiany w kontekście strony internetowej, na której rozszerzenie jest aktywne.

7.1.1. Przygotowanie wstrzyknięcia (`prepareInjection.js`)

Na dzień 16.04.2024 r., Firefox nie obsługuje skryptów zawartości uruchamianych w „głównym świecie wykonywania” (opcja `ExecutionWorld: "MAIN"`, podrozdz. 4.2.2) [69]. Twórcy RDT zdecydowali się obejść to ograniczenie wykorzystując starszą wersję API rozszerzeń (Manifest V2) [70]. Wersja ta pozwala skryptom zawartości na dodawanie do drzewa DOM wbudowanych skryptów (ang. „inline scripts”), które zachowują się podobnie jak te uruchomione z opcją `ExecutionWorld: "MAIN"` w nowszej wersji API.

Dla przeglądarek innych niż Firefox, które obsługują nowsze API, takie wstrzykiwanie skryptów nie jest konieczne, i obsługa dodawania skryptów zawartości z odpowiednimi parametrami, jest w RDT wykonywana przez `background.js` (podrozdz. 7.1.2). Niemal cały kod zawarty w `prepareInjection.js`, uruchamiany jest zatem tylko dla przeglądarek Firefox i stanowi jedynie obejście istniejących ograniczeń.

Uwzględniając fakt, że w ramach tej pracy, projekt był przygotowywany z myślą o przeglądarce Chrome, zawartość `prepareInjection.js` nie jest szczególnie istotna.



Rysunek 7.1. Architektura punktów wejścia dla rozszerzenia React Developer Tools.

7.1.2. Skrypt tła (background.js)

Pierwszą akcją, wykonywaną przez skrypt tła, jest wstrzyknięcie dwóch skryptów uruchamianych w głównym świecie uruchomienia:

- `installHook.js` - skrypt odpowiedzialny za instalację, hooka z którym łączy się aplikacja Reacta (podrozdz. 7.2),
- `renderer.js` - skrypt odpowiedzialny za połączenie RDT z aplikacją w momencie uruchomienia trybu profilowania (podrozdz. 7.3).

Następną akcją jest rozpoczęcie nasłuchiwanie na połączenia przychodzące poprzez interfejs rozszerzeń przeglądarki `chrome.runtime` (podrozdz. 4.3). Skrypt dla każdej karty oczekuje na dwa połączenia:

- przychodzące z panelu deweloperskiego,
- przychodzące ze skryptu zawartości `proxy.js` (podrozdz. 7.4). Skrypt ten uruchamiany jest w kontekście strony dopiero po nawiązaniu połączenia z panelem deweloperskim.

Po nawiązaniu obydwu połączeń dla danej karty, skrypt tła tworzy pomost komunikacyjny między panelem deweloperskim a wstrzykniętymi skryptami zawartości przekazując między nimi wszystkie wiadomości. Jako, że skrypt tła uruchamiany jest raz w kontekście całej przeglądarki, a niezależne aplikacje Reacta i odpowiadające im panele deweloperskie mogą być uruchomione w wielu kartach, skrypt tła zachowuje w pamięci odpowiednią

mapę identyfikatorów kart z odpowiadającymi im połączeniami. Obsługuje także zamykanie pomostu komunikacyjnego w momencie zakończenia któregośkolwiek z połączeń.


W następnym kroku skrypt tła zaczyna nasłuchiwanie na gotowość panelu narzędzi deweloperskich na wstrzykiwanie backendów (podrozdz. 7.6). Po otrzymaniu odpowiedniej wiadomości, skrypt tła uruchamia skrypt `backendManager.js` (podrozdz. 7.5) w kontekście właściwej karty przeglądarki. Na podstawie informacji odebranych od menadżera backendów, skrypt tła wstrzykuje odpowiednie skrypty zawartości backendów (podrozdz. 7.6), a następnie przekazuje te informacje do panelu (podrozdz. 7.9).

Ostatnią istotną funkcją skryptu tła jest zarządzanie ikoną rozszerzenia w pasku narzędzi przeglądarki i wyskakującym oknem (widocznym po jej kliknięciu). Skrypt nasłuchuje na zdarzenia przeglądarki, takie jak utworzenie nowej karty, zmiana aktywnej czy nawigacja do innej strony, i na ich podstawie, oraz korzystając z informacji przekazywanych przez zainstalowanego hooka (podrozdz. 7.2), aktualizuje ikonę i okno (podrozdz. 7.1.3).

7.1.3. Okno modalne

Okno modalne oraz ikona rozszerzenia aktualizowane są przez skrypt tła w zależności od wyświetlonej strony i właściwości uruchomionej aplikacji Reacta. Twórcy RDT wyszczególnili następujące warianty okna, informujące użytkownika o stanie rozszerzenia:

- **disabled** - okno wyświetlane, gdy na stronie nie udało się wykryć aplikacji Reacta (domyślnie ustawiane przy starcie przeglądarki),
- **deadcode** - wyświetlane, gdy wykryto, że uruchomiona aplikacja Reacta zawiera zarówno produkcyjną i deweloperską wersję Reacta oraz ostrzega o potencjalnych problemach z wydajnością,
- **development** - wyświetlane, gdy aplikacja Reacta uruchomiona jest w trybie deweloperskim,
- **outdated** - wyświetlane, gdy wykryto przestarzałą wersję Reacta i zawiera komunikat, zalecający aktualizację biblioteki,
- **production** - wyświetlane, gdy aplikacja Reacta uruchomiona jest w trybie produkcyjnym,
- **restricted** - wyświetlane w przypadku, gdy obecnie wyświetlana strona nie pozwala na dostęp rozszerzenia (np. strony `chrome://settings/` czy `chrome://history/`),
- **unminified** - wyświetlane, gdy rozszerzenie wykryje niezminifikowaną wersję Reacta i ostrzega użytkownika o potencjalnych problemach z wydajnością.

This page is using the production build of React. 
Open the developer tools, and "Components" and "Profiler" tabs will appear to the right.

Rysunek 7.2. Okno modalne „production” rozszerzenia React Developer Tools.

7.1.4. Narzędzia deweloperskie (main.html)

Dokument `main.html` jest punktem wejścia dla panelu narzędzi deweloperskich przeglądarki. Dokument ten nie zawiera żadnych elementów, a jedynie importuje i uruchamia skrypt JavaScript `main.js`.

Uruchomiony skrypt, co sekundę aż do skutku, sprawdza, czy na stronie została wykryta aplikacja Reacta i czy hook (podrozdz. 7.2) został poprawnie wstrzyknięty. W przypadku spełnienia obydwu warunków, tworzy nowe panele w narzędziach deweloperskich: „Components” i „Profiler”. Następnie uruchamia w nich odpowiednie aplikacje Reacta (podrozdz. 7.9).

Oprócz tego, skrypt zarządza także komunikacją z innymi częściami rozszerzenia po stronie narzędzi deweloperskich i inicjuje wymagane magazyny danych, jednakże opis użytych rozwiązań pominięto, ponieważ w tych aspektach podjęto decyzję o stworzeniu własnej implementacji niebazującej na RDT.

7.2. Struktura i instalacja hooka w aplikacji (installHook.js)

Z racji, że skrypt ten uruchomiony jest w głównym świecie wykonywania, ma on pełny dostęp do obiektu `window`.

Skrypt wykorzystuje tę możliwość poprzez dodanie do globalnego obiektu `window` obiektu przedstawionego w list. 23.

Listing 23. Instalacja hooka przez RDT.

```
window.__REACT_DEVTOOLS_GLOBAL_HOOK__ = {
  rendererInterfaces ,
  listeners ,
  backends ,
  renderers ,
  emit ,
  getFiberRoots ,
  inject ,
  on ,
  off ,
  sub ,
  supportsFiber: true ,
  checkDCE ,
  onCommitFiberUnmount ,
  onCommitFiberRoot ,
  onPostCommitFiberRoot ,
  setStrictMode ,
  getInternalModuleRanges ,
  registerInternalModuleStart ,
  registerInternalModuleStop ,
};
```

W poniższych podrozdziałach opisane zostały te właściwości obiektu, które są istotne dla zrozumienia implementacji projektu realizowanego w ramach tej pracy.

7.2.1. Właściwość `renderers`

```
renderers: Map<number, ReactRenderer>;
```

Pod kluczem `renderers` przypisywana jest mapa zawierająca unikatowe identyfikatory oraz odpowiadające im renderery (podrozdz. 5.7). Mapa wypełniana jest konkretnymi wartościami w momencie podłączania się aplikacji Reacta do narzędzi deweloperskich (podrozdz. 7.2.4). Według dokumentacji RDT, mapę tę wykorzystuje mechanizm „Fast Refresh” [71]. Nie został on, co prawda, użyty w autorskim projekcie, jednakże, podczas pracy nad rozwiązaniem, zauważono problemy z działaniem samego Reacta, gdy mapa ta nie była obecna w hooku (podrozdz. 9.7.2).

7.2.2. Właściwość `rendererInterfaces`

```
rendererInterfaces: Map<number, RendererInterface>;
```

Mapa ta, wprawdzie nie jest używana w implementacji autorskiego rozwiązania, jednak fragmenty logiki obiektów w niej przechowywanych są, więc warto o niej krótko wspomnieć, gdyż ułatwi to zrozumienie działania innych właściwości obiektu. Przechowuje on identyfikatory rendererów (podrozdz. 7.2.1) oraz odpowiadające im interfejsy renderera. Interfejsy te są złożonymi obiektami (około 5000 linii kodu dla analizowanego interfejsu), zawierającymi całą logikę potrzebną do interpretacji wykonywanych w Reaccie operacji, takich jak np.:

- parsowanie drzewa komponentów,
- profilowanie wydajności aplikacji,
- inspekcja właściwości komponentów,
- analiza hooków (podrozdz. 7.10).

Takie zmodularyzowane rozwiązanie pozwala na implementację charakterystycznej dla danego renderera logiki i poprawne działanie RDT np. z różnymi środowiskami uruchomieniowymi (przeglądarki czy React Native).

Mapa ta wypełniana jest przez zainicjowane backendy (podrozdz. 7.6) dla każdego renderera, który zarejestruje się przez wywołanie funkcji `inject` (podrozdz. 7.2.4).

7.2.3. Właściwość `supportsFiber`

```
supportsFiber: boolean;
```

Właściwość `supportsFiber` ustawiona na `true` informuje Reacta, że RDT są przystosowane do pracy z architekturą Fiber (podrozdz. 5.8). Brak tej flagi powoduje wyświetlenie przez Reacta błędu w konsoli przeglądarki, który informuje o niezgodności wersji RDT z wersją biblioteki.

7.2.4. Funkcja inject

```
inject: (renderer: ReactRenderer) => number;
```

Każdy z obecnych na stronie rendererów (podrozdz. 5.7) wywołuje tę funkcję, aby połączyć się z RDT i zarejestrować się jako dostępny do analizy.

Funkcja generuje unikatowy identyfikator dla przekazanego jako argument renderera, i używając go jako klucza, dodaje ów renderer do mapy renderers (podrozdz. 7.2.1).

Następnym krokiem, wykonywanym przez funkcję, jest łącanie konsoli (ang. „console patching”). Polega to na podmienieniu funkcji `console.log`, `console.warn` i `console.error` na zaimplementowane przez RDT odpowiedniki. Dzięki temu, RDT dodaje dodatkowe funkcjonalności, takie jak:

- wyświetlanie struktury komponentów w ostrzeżeniach i informacjach o błędach,
- wyłączanie, bądź oznaczanie podwójnych komunikatów w konsoli, generowanych przez Reacta uruchomionego w „strict mode” [72],
- wyłączenie komunikatów w konsoli, generowanych przez Reacta przy ponownym renderowaniu komponentów w trakcie inspekcji hooków (podrozdz. 7.10).

Funkcja `inject` zwraca identyfikator przypisany do renderera, który służy całemu rozszerzeniu do rozróżniania poszczególnych rendererów działających na stronie. Identyfikator ten, przekazywany jest np. do funkcji `onCommitFiberUnmount` (podrozdz. 7.2.7) i `onCommitFiberRoot` (podrozdz. 7.2.6).

Pozostałe działania, wykonywane przez funkcję `inject`, nie są istotne z punktu widzenia tej pracy.

7.2.5. Funkcja checkDCE

```
checkDCE: (function: Function) => void;
```

React wywołuje tę funkcję po to, żeby określić czy w wersji produkcyjnej aplikacji zostały poprawnie usunięte wszystkie fragmenty kodu potrzebne jedynie w trybie deweloperskim.

Implementacja tej funkcji nie jest istotna z perspektywy rozważanego problemu, jak opisano w podrozdz. 9.7.2.

7.2.6. Funkcja onCommitFiberRoot

```
onCommitFiberRoot: (  
  rendererID: number,  
  root: FiberRoot,  
  priorityLevel?: number,  
  didError?: boolean  
) => void;
```

React wywołuje funkcję `onCommitFiberRoot` za każdym razem, gdy drzewo komponentów zostanie zaktualizowane. Wraz z `onCommitFiberUnmount` (podrozdz. 7.2.7) są to dla rozszerzenia, jedyne źródła informacji o stanie aplikacji Reacta.

Hook zachowuje informacje o nowo zamontowanych korzeniach drzewa komponentów, a następnie, na podstawie przekazanego jako argument identyfikatora renderera, pobiera z mapy `rendererInterfaces` (podrozdz. 7.2.2) odpowiedni interfejs i oddaje mu sterowanie przekazując do niego obiekt `Fiber`.

Interfejs renderera analizuje przekazane mu drzewo komponentów, i na jego podstawie, generuje informacje o zmianach, które nastąpiły w aplikacji. Opis implementacji wykrywania tych zmian przedstawiono w podrozdz. 7.7.

7.2.7. Funkcja `onCommitFiberUnmount`

```
onCommitFiberUnmount: (  
  rendererID: number,  
  fiber: Fiber  
) => void;
```

Funkcja `onCommitFiberUnmount` każdorazowo wywoływana jest przez Reacta, gdy jakiś komponent jest odmontowywany. W przypadku odmontowania komponentu-rodzica, funkcja ta wywoływana jest również dla wszystkich jego potomków. Jako argumenty, przekazywane są do niej identyfikator renderera oraz obiekt `Fiber` (podrozdz. 5.8) danego komponentu. Funkcja, podobnie jak `onCommitFiberRoot` (podrozdz. 7.2.6), przekazuje informacje do odpowiedniego interfejsu renderera.

Interfejs natomiast, generuje identyfikator dla danego obiektu `Fiber` i zapisuje go w liście oczekujących na przesłanie do panelu narzędzi deweloperskich, identyfikatorów komponentów do odmontowania (podrozdz. 7.8.1).

7.3. Skrypt `renderer.js`

Zgodnie z dokumentacją RDT, skrypt ten służy do obsługi połączenia Reacta z RDT jedynie w przypadku uruchomienia trybu profilowania aplikacji Reacta, więc nie jest istotny z punktu widzenia tej pracy.

7.4. Pomost między izolowanym a głównym światem wykonywania (`proxy.js`)

Jak wspomniano w podrozdz. 4.2.2, skrypty zawartości uruchomione w głównym świecie wykonywania, mają pełny dostęp do właściwości obiektów globalnych (tj. `window` czy `document`), jednakże nie mają dostępu do API rozszerzeń Chrome.

W celu umożliwienia komunikacji hooka (uruchomionego w głównym świecie) z resztą rozszerzenia, skrypt `proxy.js` (uruchomiony w środowisku izolowanym) wykorzystuje mechanizm wysyłania wiadomości poprzez `window.postMessage` (podrozdz. 7.4.1).

Po uruchomieniu, skrypt nawiązuje połączenie ze skryptem tła (podrozdz. 7.1.2) poprzez `chrome.runtime.connect` (podrozdz. 4.3) oraz rozpoczyna nasłuchiwanie na wiadomości wysłane poprzez `window.postMessage`. Skrypt przekazuje wiadomości przychodzące jednym kanałem do drugiego, łącząc w ten sposób oba światy wykonywania.

Ostatnim zdaniem skryptu jest powiadomienie skryptu `backendManager.js` (podrozdz. 7.5) o gotowości do pracy.

Skrypt zapewnia także zakończenia nasłuchiwanie na wiadomości `postMessage` po zamknięciu połączenia ze skryptem tła.

7.4.1. Ograniczenia mechanizmu `window.postMessage`

Mechanizm ten używany jest w przeglądarkach do między-źródłowej komunikacji (ang. „cross-origin communication”), np. w przypadku osadzania na stronie ramek „`iframe`” z treścią pochodzącą z innej domeny.

Strona wysyłająca wiadomość, wywołuje metodę `postMessage` na obiekcie `window`, przekazując jako argument treść wiadomości. Odbiorca musi zarejestrować nasłuchiwanie na zdarzenie „`message`” poprzez `window.addEventListener` [73].

Wszystkie wiadomości wysyłane poprzez mechanizm są rekursywnie serializowane przy użyciu ustrukturyzowanego algorytmu klonowania (ang. „structured clone algorithm”). Algorytm ten, dzięki przechowywanej mapie wcześniej odwiedzonych wartości, unika nieskończonego zapętlenia. Mechanizm serializacji nie obsługuje jednak niektórych typów danych, jak np. funkcje i węzły drzewa DOM [74].

Serializacja złożonych obiektów może być także kosztowna obliczeniowo. Fakt ten jest istotny w kontekście narzędzi deweloperskich, które mogą wymagać częstego przesyłania dużych ilości danych (podrozdz. 7.8).

7.5. Dopasowanie RDT do wersji Reacta (`backendManager.js`)

Skrypt nasłuchuje na wiadomość o gotowości do pracy od skryptu `proxy.js` (podrozdz. 7.4) i po jej otrzymaniu określa wymagane wersje backendów wymaganych do obsługi każdego z zarejestrowanych w `renderers` (podrozdz. 7.2.1) rendererów.

Informacje te są następnie przekazywane do `background.js` (podrozdz. 7.1.2), który wstrzykuje odpowiednie skrypty zawartości backendów (podrozdz. 7.6).

Proces jest ponawiany dla pojedynczych, z opóźnieniem zarejestrowanych w narzędziach deweloperskich rendererów (o ile odpowiednie wersje backendów nie zostały już zainstalowane) w sposób, którego nie opisano w tej pracy.

Skrypt następnie nasłuchuje na wiadomość o gotowości wstrzykniętych backendów i po jej otrzymaniu inicjuje ich działanie.

7.6. Backendy narzędzi deweloperskich

Backendy są złożonymi konstruktami obsługującymi całą logikę komunikacji oraz interakcji narzędzi deweloperskich z Reactem i stroną internetową, na której jest uruchomiony. Opis ich implementacji nie jest szczególnie istotny z punktu widzenia tej pracy. Ważne jest jednak to, że backendy przy inicjacji uzupełniają mapę `rendererInterfaces` (podrozdz. 7.2.2) odpowiednimi interfejsami renderera, fragmenty logiki których (opisane np. w podrozdz. 7.7 i podrozdz. 7.8) posłużyły do implementacji autorskiego rozwiązania.

7.7. Parsowanie drzewa Fiber

Proces parsowania drzewa rozpoczyna się od przekazanego z `onCommitFiberRoot` (podrozdz. 7.2.6) korzenia.

W pierwszej kolejności sprawdzane jest czy korzeń drzewa posiada referencje do swojej wersji alternatywnej (właściwość `alternate`, podrozdz. 5.8). Jeżeli nie, to znaczy, że pierwszy raz natrafiamy na dany korzeń i należy zgłosić zamontowanie całego drzewa (podrozdz. 7.7.1).

Jeżeli korzeń drzewa posiada wersję alternatywną, na podstawie wartości właściwości `memoizedState` określane jest czy korzeń jest i czy był w wcześniej zamontowany. Następnie rozpatrywane są rozłączne przypadki:

- jeżeli nie był zamontowany a teraz jest, należy zgłosić zamontowanie całego drzewa;
- jeżeli był zamontowany a teraz nie jest, należy zgłosić odmontowanie drzewa podobnie jak w podrozdz. 7.2.7;
- jeżeli był i nadal jest zamontowany, przeprowadzana jest analiza drzewa w celu określenia wymagających zgłoszenia różnic (podrozdz. 7.7.2).

7.7.1. Montowanie obiektów struktury Fiber

Proces montowania polega na rekurencyjnym przemieszczaniu się wгłęb drzewa obiektów Fiber (właściwość `child`) iterując po obiektach na każdym z poziomów (`sibling`).

Listing 24. Uproszczony algorytm przemieszczania się po drzewie Fiber.

```
function traverseFiberTree(fiber: Fiber) {
  let current: Fiber | null = fiber;
  while (current !== null) {
    recordMount(current);
    if (current.child !== null) {
      traverseFiberTree(current.child);
    }
    current = current.sibling;
  }
}
```

Identyfikator każdego z odwiedzonych obiektów Fiber dodawany jest do listy operacji (podrozdz. 7.8.1) jako identyfikator zamontowanego elementu do zgłoszenia.

7.7.2. Aktualizacja obiektów struktury Fiber

Proces aktualizacji opiera się na podobnym, jak w podrozdz. 7.7.1, algorytmie przemieszczania się po drzewie Fiber. Jeżeli jakiś z napotkanych obiektów nie posiada wersji alternatywnej, to znaczy, że on i całe jego poddrzewo zostało dodane w aktualizacji i należy zgłosić jego zamontowanie.

Algorytm pomija (odcina) gałęzie drzewa, jeżeli dziecko danego obiektu Fiber i jego wersji alternatywnej, są referencjami do tego samego obiektu. Oznacza to, że w całym poddrzewie nie zostały zamontowane żadne nowe elementy.

7.8. Optymalizacja przesyłania danych

Twórcy RDT zauważyli, że najpoważniejszym problemem związanym z wydajnością narzędzi deweloperskich jest przesyłanie dużej ilości danych między skryptami zawartości uruchomionymi w różnych światach wykonywania (podrozdz. 7.4.1). W celu zminimalizowania ruchu, zdecydowali się zmodyfikować logikę tak, aby przesyłane były jedynie informacje niezbędne dla panelu deweloperskiego do przedstawienia drzewiastej struktury komponentów. Dodatkowe informacje (np. stan komponentów czy właściwości) są przesyłane „na żądanie”, dopiero w momencie interakcji użytkownika z panelem [17].

Kolejnym usprawnieniem było użycie tabeli operacji. Mechanizm ten nie został wykorzystany w implementacji autorskiego rozwiązania, jednakże rozwiązanie jest na tyle ciekawe, że warto o nim wspomnieć i potraktować jako możliwe rozszerzenie w przyszłości (podrozdz. 12.1).

7.8.1. Tabela operacji

Tabela operacji jest listą zakodowanych w postaci liczb operacji, które mają zostać wykonane na drzewie komponentów. Dwa pierwsze wpisy w tabeli identyfikują, którego renderera i korzenia drzewa dotyczą operacje.

Dla każdego elementu, którego zamontowanie należy zgłosić, przekazywane są jedynie informacje takie jak:

- typ elementu,
- identyfikator,
- identyfikator rodzica,
- identyfikator właściciela,
- nazwa,
- klucz,

przy czym wszystkie napisy również są kodowane w postaci tabel napisów (ang. „string table”, list. 25), która zawiera kolejno:

1. Długość całej tabeli;
2. Dla każdego napisu w tabeli:
 - a) długość napisu,
 - b) listę kodów znaków reprezentujących napis.

Listing 25. Przykładowa tabela napisów [17].

```
[  
  // ...  
  8,    // długość całej tabeli  
  3,    // długość napisu  
  70,   // "F"  
  111,  // "o"  
  111,  // "o"  
  3,    // długość napisu  
  66,   // "B"
```

```

97,    // "a"
114,   // "r"
// ...
]

```

Późniejsze operacje mogą odwoływać się do napisów w tabeli poprzez ich indeksy, zaczynając od 1. Dla przykładu z list. 25, napis „Foo” ma indeks 1, a „Bar” 2. Indeks 0 zawsze oznacza wartość null i nie jest bezpośrednio przekazywany w tabeli napisów.

Każda z dokonywanych operacji ma swój unikatowy kod, np. dodanie nowego elementu ma kod 1, usunięcie 2, a zmiana kolejności elementów 3.

Przykładem może posłużyć tablica operacji dodająca komponent funkcyjny o nazwie Foo jako bezpośrednie dziecko korzenia o identyfikatorze 1, dla renderera o identyfikatorze 1 oraz usuwająca 2 elementy przedstawiona w list. 26.

Listing 26. Przykładowa tabela operacji [17].

```

[
  1,    // identyfikator renderera
  1,    // identyfikator korzenia
  4,    // długość całej tabeli napisów
  3,    // długość napisu
  70,   // "F"
  111,  // "o"
  111,  // "o"
  1,    // operacja dodania nowego elementu
  2,    // identyfikator nowego elementu
  0,    // typ: komponent funkcyjny
  1,    // identyfikator rodzica
  0,    // identyfikator właściciela
  1,    // indeks "Foo" w tabeli napisów
  0,    // brak klucza (null)
  2,    // operacja usunięcia elementów
  2,    // liczba elementów do usunięcia
  35,   // identyfikator pierwszego elementu do usunięcia
  21,   // identyfikator drugiego elementu do usunięcia
]

```

7.9. Panele narzędzi deweloperskich

Zarówno w panelu komponentów, jak i w panelu profilowania, tworzonych przez RDT, uruchamiane są aplikacje Reacta, zapewniające interfejs użytkownika, który pozwala na analizę i interakcję z badaną aplikacją.

W podrozdz. 2.1.1 wspomniano, że panel profilowania nie ma znaczenia z perspektywy niniejszej pracy, a także opisano już funkcjonalności panelu komponentów.

Zaimplementowane w rozwiązaniu funkcje i interfejs użytkownika były, co prawda, bazowane na tych dostępnych w RDT, jednakże implementacja jest autorska i nie jest

oparta na kodzie źródłowym RDT. Z tego powodu opis działania samych paneli zostanie pominięty.

7.10. Inspekcja hooków Reacta

Inspekcja hooków, według twórców RDT, okazała się wyjątkowym wyzwaniem ze względu na możliwość wykorzystywania hooków niestandardowych (podrozdz. 5.9.10).

W związku z tym, że są to zwykle funkcje wywoływane w ciele komponentu, RDT w celu ich identyfikacji tymczasowo nadpisuje wbudowane hooki, a następnie ponownie renderuje komponent.

Podmienione implementacje wbudowanych hooków, umyślnie zgłaszają błędy (ang. „throw error”), które następnie są przechwytywane i analizowane pod kątem śladu stosu (ang. „stack trace”). Na podstawie śladu, RDT, bez analizy kodu źródłowego komponentu, jest w stanie określić wszystkie wywołania funkcji pomiędzy ciałem komponentu a wbudowanym hookiem, który zgłosił błąd, i w rezultacie, zidentyfikować niestandardowe hooki i sposób ich zagnieżdżeń [17].

Twórcy tego rozwiązania wykazali się dużą kreatywnością i pomysłowością, jednakże ze względu na jego złożoność, w autorskim rozwiązaniu podjęto decyzję, aby, w ramach uproszczenia, nie implementować funkcjonalności pozwalającej na wykrywanie niestandardowych hooków. Rozwiązanie to można potraktować jako dalsze rozwinięcie pracy w przyszłości (podrozdz. 12.1).

8. Przygotowanie środowiska deweloperskiego

Jak wspomniano w rozdz. 4, rozszerzenie składa się ze statycznych plików HTML, JavaScript i ewentualnie CSS. Jednak tworzenie kodu źródłowego z użyciem TypeScript, pozwala na zwiększenie kontroli błędów oraz udostępnia bardziej zaawansowane podpowiedzi auto-uzupełniania tworzonego kodu. Z tego powodu podjęto decyzję o skonfigurowaniu projektu tak, aby maksymalnie uprościć proces deweloperski.

Założeniem w autorskim projekcie było również to, aby poszczególne części rozszerzenia mogły być tworzone z użyciem frontendowych frameworków takich jak np. React, co umożliwiłoby łatwiejsze rozwijanie rozszerzenia. Każda z części rozszerzenia mogłaby być stworzona jako osobny projekt, jednak biorąc pod uwagę fakt, że części te są od siebie zależne, rozsądniejszym wydawało się skonfigurowanie jednego projektu umożliwiające budowanie całego rozszerzenia na raz.

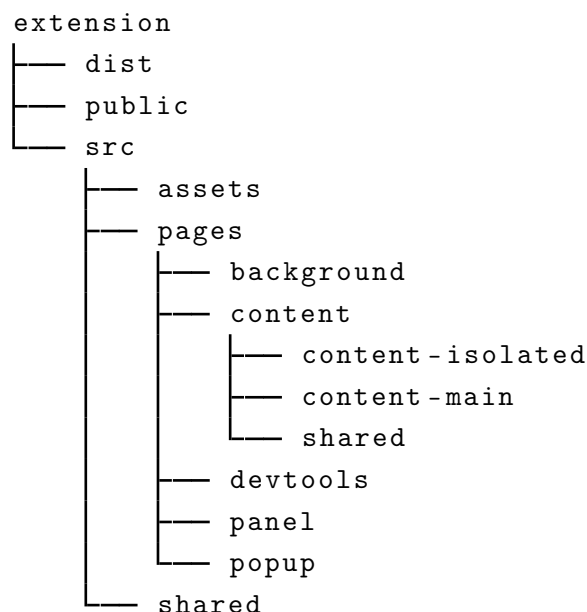
Bazując na projekcie „chrome-extension-boilerplate-react-vite” [75] i wprowadzając drobne modyfikacje, udało się osiągnąć wszystkie z tych wymagań. Jest to projekt szablonowy, który już zawiera podstawową konfigurację pozwalającą na tworzenie rozszerzeń do przeglądarki Chrome z użyciem Reacta i Vite. Zawiera on również bazową konfigurację takich narzędzi jak ESLint, TypeScript, Prettier czy SASS.

8.1. Struktura projektu

Repozytorium projektu zawiera dwa katalogi:

- `extension` - zawierający kod źródłowy rozszerzenia,
- `test-apps` - zawierający kod źródłowy przykładowych aplikacji, używanych do testowania rozwiązań podczas implementacji.

Listing 27. Uproszczona struktura katalogów projektu rozszerzenia.



W list. 27 przedstawiona została uproszczona struktura katalogów w folderze `extension`. Pełną architekturę katalogów repozytorium można znaleźć w zał. 4.

Jeżeli nie zaznaczono inaczej, wymienione w tym rozdziale ścieżki do plików podawane są względem katalogu `extension`.

- `dist` - zawiera pliki wynikowe zbudowanego rozszerzenia,
- `public` - pliki statyczne, bezpośrednio kopiowane do katalogu `dist`,
- `src` - kod źródłowy rozszerzenia,
- `src/assets` - pliki statyczne, możliwe do importowania w kodzie źródłowym,
- `src/pages` - pliki źródłowe poszczególnych części rozszerzenia,
- `src/pages/background` - kod źródłowy skryptu tła,
- `src/pages/content` - kod źródłowy skryptu zawartości,
- `src/pages/content/content-isolated` - kod źródłowy skryptu zawartości, uruchamianego w środowisku izolowanym podrozdz. 4.2.2,
- `src/pages/content/content-main` - kod źródłowy skryptu zawartości, uruchamianego w głównym świecie wykonywania,
- `src/pages/content/shared` - kod źródłowy skryptów współdzielonych pomiędzy skryptami zawartości,
- `src/pages/devtools` - kod źródłowy narzędzi deweloperskich,
- `src/pages/panel` - kod źródłowy panelu rozszerzenia,
- `src/pages/popup` - kod źródłowy okna modalnego rozszerzenia,
- `src/shared` - kod współdzielony pomiędzy różnymi częściami rozszerzenia.

8.2. Konfiguracja Vite

Vite to narzędzie do budowania projektów webowych, stworzone przez Evana You, twórcę frameworka Vue.js [76]. Vite stawia dużą wagę na szybkość działania, zapewnia natywną obsługę ESM [77] oraz efektywną obsługę HMR (Hot Module Replacement) [78], co zdecydowanie przyspiesza i ułatwia proces tworzenia aplikacji.

Dużą zaletą Vite jest również, łatwość dodawania nowych funkcjonalności poprzez rozszerzenia kompatybilne z używanym wewnątrz silnikiem budującym - Rollup [79]. W projekcie, niezbędnym okazało się wykorzystanie tej funkcjonalności, co zostało opisane w podrozdz. 8.3.

Konfiguracja narzędzia Vite i jego rozszerzeń znajduje się w pliku `vite.config.ts`.

8.3. Rozszerzenie funkcjonalności Vite

Podczas tworzenia rozszerzenia problemem okazał się fakt, że Vite automatycznie optymalizuje importowane przez różne części projektu pliki. Jeżeli dany plik importowany jest tylko w jednym miejscu, Vite wstawia jego zawartość w miejsce importu. Jeżeli jednak plik jest importowany w wielu miejscach, Vite tworzy dla niego osobny plik (tzw. „chunk”) i pozostawia jako import w miejscach użycia. Takie rozwiązanie pozwala na zmniejszenie wielkości wynikowych plików, jednak w tym przypadku nie pozwalało na poprawne działanie rozszerzenia.

Skrypty zawartości z zasady nie mogą importować innych modułów, więc niemożliwym okazało się współdzielenie pomiędzy nim a innymi częściami rozszerzenia żadnych funkcjonalności. Prowadziło to trudności w utrzymaniu spójności pomiędzy wymagającymi zduplikowania fragmentami kodu źródłowego.

W internecie dostępne jest rozszerzenie do Vite, które w teorii powinno rozwiązywać opisany problem („vite-plugin-force-inline-module” [80]), jednak nie spełniało ono w pełni wymagań projektowych. Nie pozwalało na wystarczająco granularne sterowanie tym, które z plików mają być importowane a które nie. Poza tym, konfliktowało z, obecnym w projekcie, automatycznym restartowaniem serwera deweloperskiego po wykryciu zmian w kodzie źródłowym. Utworzenie nowego rozszerzenia, było więc w konsekwencji koniecznością i elementem umożliwiającym pracę nad głównym rozszerzeniem, będącym przedmiotem tej pracy.

8.3.1. Interfejs rozszerzeń Vite

Rozszerzenia Vite są funkcjami, które mogą przyjmować parametry konfiguracyjne i zwracają obiekt z kluczami odpowiadającymi nazwami konkretnym hookom Vite i wartościami, będącymi funkcjami, które zostają wywołane w odpowiednich momentach budowania projektu [81].

8.3.2. Działanie stworzonego rozszerzenia

Kod źródłowy rozszerzenia widoczny jest w list. 28. Rozszerzenie przyjmuje dwa parametry:

- `rules` - tablica obiektów opisujących reguły, według których pliki mają być importowane,
- `inlineTag` - opcjonalny ciąg znaków, wykorzystywany do oznaczania plików, które mają zostać wbudowane w miejsce importu. Brak podania argumentu skutkuje użyciem domyślnej wartości `"___!inline!___"`, ale wykorzystany może być dowolny ciąg znaków, co do którego mamy pewność, że nie będzie on występował w żadnej ścieżce pliku.

Listing 28. Kod źródłowy stworzonego rozszerzenia.

```

1 export function inlineImports({
2   rules = [],
3   inlineTag = "___!inline!___",
4 }): {
5   rules: InlineImportsRule[];
6   inlineTag?: string;
7 }): PluginOption {
8   const codeMap = new Map<string, string>();
9
10  return {
11    name: "vite-plugin-inline-imports",
12    enforce: "pre",

```

```
13     async resolveId(source, importer, options) {
14         if (!importer || options.isEntry) return;
15
16         let inlineRoot = "";
17         let realImporter = importer;
18         if (importer.includes(inlineTag)) {
19             realImporter = importer.split(inlineTag)[2];
20             inlineRoot = importer.split(inlineTag)[1];
21         }
22
23         const resolution = await this.resolve(
24             source,
25             realImporter, {
26                 skipSelf: true,
27                 ...options,
28             }
29         );
30
31         if (!resolution || resolution.external) return;
32
33         const isInline = shouldBeInlined(
34             rules,
35             resolution.id,
36             realImporter,
37             inlineRoot
38         );
39
40         if (isInline) {
41             const code = await this.load({
42                 id: resolution.id
43             });
44             inlineRoot = inlineRoot
45                 ? inlineRoot
46                 : realImporter;
47
48             if (!code.code) return;
49             const fakeId = `
50                 ${randomVarName()}${inlineTag}${inlineRoot}
51                 ${inlineTag}${resolution.id}
52             `;
53             codeMap.set(fakeId, code.code);
54
55             return { id: fakeId };
56         }
57     }
```

```

58         if (inlineRoot !== "") {
59             return { id: resolution.id };
60         }
61     },
62     load(id) {
63         return codeMap.get(id);
64     },
65 };
66 }

```

Reguły (list. 29) są obiektami zawierającymi trzy pola:

- `for` - tablica wyrażeń regularnych, które muszą pasować do ścieżki pliku, w którym znajduje się import,
- `inline` - tablica wyrażeń regularnych, które muszą pasować do ścieżki pliku, który ma zostać wbudowany w miejsce importu,
- `recursively` - opcjonalne pole logiczne, które określa czy pliki importowane przez plik, który ma zostać wbudowany, również powinny zostać wbudowane.

Listing 29. Struktura obiektu opisującego regułę.

```

1  export type InlineImportsRule = {
2    for: RegExp[];
3    inline: RegExp[];
4    recursively?: boolean;
5  };

```

Działanie rozszerzenia opiera się na dwóch hookach Vite. Dla każdego importowanego pliku, Vite wywołuje hook `resolveId` (linia 13) z parametrami `source` - ścieżką importowanego pliku, `importer` - ścieżką pliku, w którym znajduje się import oraz `options` - obiektem dodatkowych opcji. W linii 14 sprawdzane jest, czy dany moduł nie jest wejściem do programu. Jeżeli tak jest, kontrola przekazywana jest do domyślnego systemu rezolucji Vite poprzez zwrócenie wartości `undefined`. W przeciwnym wypadku, w linii 18 sprawdzane jest, czy `importer` zawiera zdefiniowany znacznik. Jeżeli tak jest, to znaczy, że `importer` jest plikiem, który został wbudowany w miejsce innego importu. W takim przypadku wydzielane są z niego dwie ścieżki: `realImporter` - ścieżka pliku, w którym znajduje się import, oraz `inlineRoot` - ścieżka do pliku, który był korzeniem od którego rozpoczął się łańcuch wbudowywania (w przypadku użycia flagi `recursively`). Następnie dokonywana jest próba rezolucji importowanego pliku (linia 23) z użyciem flagi `skipSelf: true`. Dzięki temu, Vite używa domyślnego mechanizmu rezolucji z pominięciem rozszerzenia. Jeżeli rezolucja się powiedzie, w linii 33 wywoływana jest funkcja `shouldBeInlined` (list. 30), która na podstawie zdefiniowanych reguł określa, czy dany moduł powinien zostać wbudowany w miejsce importu. W przypadku zwrócenia prawdziwej wartości logicznej, w linii 41 wczytywana jest zawartość pliku i zapisywana w mapie `codeMap`. Kluczem w mapie (linia 45) jest sztucznie tworzony identyfikator modułu złożony, według poniższej kolejności, z:

- 16 losowych znaków, które mają na celu zapobiec kolizjom z innymi identyfikatorami,

- zdefiniowanego znacznika,
- ścieżki pliku, który był korzeniem, od którego rozpoczął się łańcuch wbudowywania,
- zdefiniowanego znacznika,
- ścieżki pliku, który ma zostać wbudowany w miejsce importu.

Zwrócenie z funkcji obiektu z kluczem `id` i wartością będącą sztucznym identyfikatorem, powoduje oszukanie mechanizmu zapamiętywania importowanych modułów przez Vite. Dzięki temu, przy kolejnych importach Vite nie uzna wbudowywanych modułów za już wcześniej wykorzystane i wbuduje je w miejsce importu.

Następnym etapem jest hook `load` (linia 58), wywoływany dla każdego modułu, który został rozwiązany przez hook `resolveId`. W linii 59 pobierany jest z `codeMap` kod źródłowy modułu i zwracany jako wynik działania hooka.

Listing 30. Funkcja decydująca czy kod z danego pliku powinien zostać wbudowany w miejsce importu.

```
1 function shouldBeInlined(  
2   rules: InlineImportsRule[],  
3   resolvedId: string,  
4   importer: string,  
5   inlineRoot: string  
6 ): boolean {  
7   const rulesForInlineRoot = rules.filter(  
8     (rule) => rule.for.some(  
9       (pattern) => pattern.test(inlineRoot)  
10    )  
11  );  
12  
13  if (rulesForInlineRoot.some(  
14    (rule) => rule.recursively  
15  )) return true;  
16  
17  const matchingRules = rules.filter(  
18    (rule) => rule.for.some(  
19      (pattern) => pattern.test(importer)  
20    )  
21  );  
22  if (matchingRules.length === 0) return false;  
23  
24  return matchingRules.some(  
25    (rule) => rule.inline.some(  
26      (pattern) => pattern.test(resolvedId)  
27    )  
28  );  
29 }
```

Decyzja o tym, czy dany moduł powinien zostać wbudowany w miejsce importu, podejmowana jest za pomocą funkcji `shouldBeInlined` (listing list. 30). W pierwszej kolejności (linia 7) wyszukiwane są wszystkie reguły pasujące do potencjalnego korzenia, od którego zaczęliśmy rekursywne wbudowywanie. Następnie (linia 14), jeżeli którakolwiek z tych reguł zawiera flagę `recursively`, zwracana jest wartość prawdziwa. W przeciwnym wypadku (linia 17) wyszukiwane są wszystkie reguły dopasowane do ścieżki importera. Jeżeli którakolwiek z nich określa, że obecnie przetwarzany moduł powinien zostać wbudowany, zwracana jest wartość prawdziwa (linia 26), w przeciwnym wypadku zwracana jest wartość fałszywa.

8.3.3. Testy rozszerzenia Vite

Dla rozszerzenia zostały stworzone testy jednostkowe z użyciem biblioteki Vitest [82]. Uruchomienie ich jest możliwe poprzez wykonanie polecenia `npm run test` w głównym katalogu projektu.

Testy sprawdzają poprawność działania rozszerzenia w przypadku różnych kombinacji reguł oraz różnych ścieżek importowanych plików.

8.3.4. Publikacja rozszerzenia Vite

Rozszerzenie zostało opublikowane jako publicznie dostępny pakiet pod licencją MIT [83] w repozytorium npm.

Do dnia 14.05.2024 r., rozszerzenie zostało pobrane łącznie 215 razy [84].

Adresy do repozytorium w serwisie GitHub, zawierającego kod źródłowy rozszerzenia, oraz do pakietu w serwisie npm znajdują się w zał. 2.

8.4. Uruchomienie głównego projektu

W pliku `package.json` skonfigurowane zostały polecenia pozwalające na uruchomienie projektu w trybie deweloperskim oraz budowanie gotowego rozszerzenia. Wywołanie polecenia `npm run dev` uruchamia serwer deweloperski Vite, nasłuchujący na zmiany w plikach źródłowych i automatycznie przebudowujący projekt.

Polecenie `npm run build` jednorazowo buduje gotowe rozszerzenie, a wynikowe pliki są zapisywane w katalogu `dist`.

W projekcie skonfigurowane zostało również rozszerzenie Vite, które przy każdym budowaniu projektu tworzy plik `manifest.json` na podstawie pliku `manifest.js` i umieszcza go w katalogu `dist`.

8.4.1. Załadowanie rozszerzenia do przeglądarki

Aby załadować zbudowane rozszerzenie do przeglądarki Chrome należy:

1. Otworzyć przeglądarkę Chrome.
2. Przejść do panelu rozszerzeń (`chrome://extensions/`).
3. Uruchomić tryb deweloperski (przełącznik w prawym górnym rogu).
4. Kliknąć przycisk „Załaduj rozpakowane” i wybrać katalog `dist` ze zbudowanym rozszerzeniem.

8. Przygotowanie środowiska deweloperskiego

Po wykonaniu tych kroków, rozszerzenie powinno pojawić się na liście zainstalowanych rozszerzeń i być gotowe do użycia.

9. Autorskie rozwiązanie

Całość kodu źródłowego rozwiązania dostępna jest w repozytorium projektu (zał. 1).

W celu umożliwienia łatwiejszego porównania autorskiego narzędzia z React Developer Tools, dołożono starań, aby zachować strukturę opisu podobną do tej z rozdz. 7, zaczynając od punktów wejścia dla uruchamianego narzędzia, a następnie przechodząc do bardziej szczegółowych opisów poszczególnych części rozwiązania.

W zał. 5 znajduje się diagram sekwencji obrazujący komunikację między poszczególnymi częściami rozszerzenia. Śledzenie diagramu równolegle z lekturą niniejszego rozdziału, może pomóc w zrozumieniu opisywanych procesów.

Wszystkie ścieżki w tym rozdziale podawane są względem katalogu `extension/src` (chyba, że zaznaczono inaczej).

9.1. Główne założenia implementacji

Analizując implementację React Developer Tools (rozdz. 7), najtrudniejszym do zrozumienia aspektem projektu, wydaje się komunikacja między poszczególnymi częściami rozszerzenia.

Dodatkowo wysoki poziom skomplikowania sposobu komunikacji, naraża rozwiązanie na problemy związane z jego utrzymaniem. Problematicznym może być dostosowanie rozszerzenia do różnic między implementacjami API rozszerzeń w różnych przeglądarkach, a także do zmian samego standardu (np. migracje z manifestu w wersji 2 na wersję 3).

W związku z powyższym, główną ideą przyświecającą implementacji autorskiego rozszerzenia było ograniczenie złożoności komunikacji między poszczególnymi częściami rozszerzenia.

Kolejnym założeniem było umożliwienie łatwej adaptacji narzędzia do pracy z różnymi bibliotekami frontendowymi. W związku z tym, podjęto starania, aby uczynić większość funkcjonalności rozszerzenia uniwersalnymi, a te, zależne od konkretnej biblioteki, jasno oddzielić i integrować z rozszerzeniem za pomocą ściśle określonych interfejsów (podrozdz. 9.6).

Ostatnim, ale nie mniej ważnym założeniem, było umożliwienie funkcjonowania rozszerzenia bez dodatkowych zewnętrznych zależności, takich jak React Developer Tools w przypadku innych nieoficjalnych narzędzi (podrozdz. 2.1.2, podrozdz. 2.1.3).

9.2. Nazwa rozszerzenia

Aby odzwierciedlić funkcjonalności i istotę tworzonego rozszerzenia, jako jego nazwę wybrano „StateViz”. Składa się ona z dwóch członów:

- „state” - odnosi się do stanu komponentów, który jest kluczowym aspektem utworzonego narzędzia,
- „viz” - jako skrót od „visualization”, jednak z przegłosem „z” zamiast „s”, co nawiązuje do słowa „wizard” (czarodziej), sugerując, że narzędzie jest w stanie „działać w sposób magiczny”.

Wybrana nazwa jest krótka, łatwa do zapamiętania i uniwersalna (nie jest związana z żadną konkretną technologią frontendową).

9.3. Punkty wejścia dla rozszerzenia przeglądarkowego

Jak wspomniano w podrozdz. 4.1 i podrozdz. 7.1, punkty wejścia dla rozszerzeń przeglądarkowych definiowane są za pomocą pliku manifestu (`manifest.json`).

Listing 31. Fragment manifestu tworzonego narzędzia.

```
{
  // ...
  "background": {
    "service_worker": "src/pages/background/index.js",
    // ...
  },
  "action": {
    "default_popup": "src/pages/popup/index.html",
    // ...
  },
  "content_scripts": [
    {
      // ...
      "js": [
        "src/pages/content-main/index.js"
      ],
      "run_at": "document_start",
      "world": "MAIN"
    },
    {
      // ...
      "js": [
        "src/pages/content-isolated/index.js"
      ],
      "run_at": "document_start",
      "world": "ISOLATED"
    }
  ],
  "devtools_page": "src/pages/devtools/index.html",
}
```

Zauważalną różnicą względem RDT, jest zdefiniowanie wszystkich używanych skryptów zawartości w pliku manifestu. W przypadku RDT, takie skrypty były wstrzykiwane dynamicznie, bądź rejestrowane w rozszerzeniu przez skrypt tła (podrozdz. 7.1.2). Zastosowane rozwiązanie zapewnia większą klarowność i przejrzystość wykonywanego kodu.

W list. 31 wynikowe pliki skryptów są plikami JavaScript, jednak (jak wspomniano we wstępie tego rozdziału) kod źródłowy jest pisany w TypeScript. Z tego powodu, w kolejnych podrozdziałach, w których omawiane są konkretne skrypty, odwołania do nich zawierają rozszerzenie „.ts”.

Zaznaczone w kolejnych podrozdziałach ścieżki do plików podawane są względem katalogu `extension/src/pages`.

9.3.1. Skrypt tła (`background/index.ts`)

Biorąc pod uwagę założenia opisane w podrozdz. 9.1, zrezygnowano z wykorzystania skryptu tła jako pośrednika w komunikacji między skryptami rozszerzenia (tak jak to miało miejsce w RDT). W związku z tym, skrypt ten odpowiada jedynie za zmianę ikony rozszerzenia w zależności od tego, czy na danej stronie wykryte zostały obsługiwane biblioteki czy też nie.

Możliwe warianty ikony przedstawione zostały na rys. 9.1.

9.3.2. Okno modalne (`popup/index.html`)

Okno modalne rozszerzenia jest wykorzystywane do wyświetlania podstawowych informacji o rozszerzeniu, takich jak:

- nazwa rozszerzenia,
- wersja,
- link do repozytorium projektu,
- informacje o autorze

oraz informacji, czy i jakie obsługiwane biblioteki zostały wykryte na obecnie otwartej stronie internetowej.

Okno modalne ma charakter informacyjny i mogłoby być prostym dokumentem HTML, reagującym jedynie na informacje o podłączonych bibliotekach, jednak zdecydowano się na zastosowanie frameworka React, aby umożliwić potencjalnie łatwiejsze rozbudowanie okna w przyszłości.

Warianty okna modalnego przedstawione zostały na rys. 9.1.

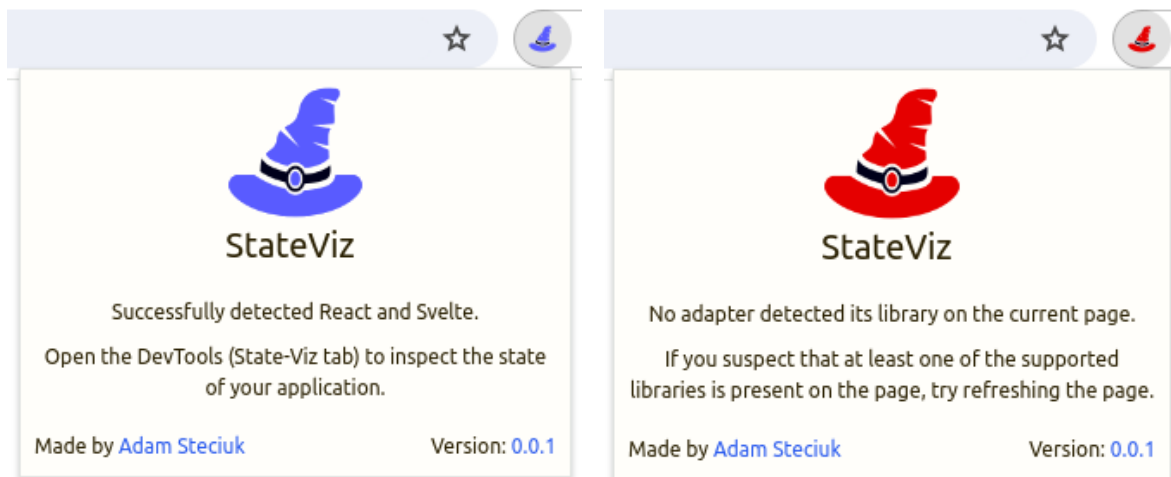
9.3.3. Główny skrypt zawartości (`content/content-main/index.ts`)

Główny skrypt zawartości, jak już wspomniano między innymi w podrozdz. 4.2.2 oraz podrozdz. 7.4, ma pełny dostęp do kontekstu uruchomionej strony internetowej. Jest zatem odpowiedzialny za komunikację z bibliotekami frontendowymi i reagowanie na wszelkie zdarzenia związane z ich użyciem.

Skrypt ten uruchamia tzw. „adaptery”, z których każdy odpowiedzialny jest za integrację z konkretną biblioteką. Proces ten opisany został w podrozdz. 9.6.

9.3.4. Izolowany skrypt zawartości (`content/content-isolated/index.ts`)

Izolowany skrypt zawartości, w przeciwieństwie do skryptu zawartości uruchomionego w głównym świecie wykonywania, posiada ograniczone możliwości interakcji z kontekstem strony internetowej.



(a) Okno modalne i ikona rozszerzenia w przypadku wykrycia obsługiwanych bibliotek.

(b) Okno modalne i ikona rozszerzenia w przypadku braku wykrycia obsługiwanych bibliotek.

Rysunek 9.1. Okno modalne i ikona rozszerzenia.

Jego zadaniem jest pośredniczenie w komunikacji między głównym skryptem zawartości a panelem narzędzi deweloperskich.

Proces ten realizowany jest poprzez klasę `ContentIsolated` i został omówiony w podrozdz. 9.9.

9.3.5. Narzędzia deweloperskie (`devtools/index.html`)

Dokument HTML narzędzi deweloperskich ładowany jest w momencie ich otwarcia. Jego zawartość nie jest widoczna dla użytkownika, jednak skrypty w nim zawarte mają mają zarządzania narzędziami deweloperskimi przegłdarki.

Wraz z otwarciem narzędzi uruchamiany jest skrypt `devtools/index.ts`, którego zadaniem jest stworzenie nowego panelu narzędzi deweloperskich, w przypadku wykrycia na stronie internetowej obsługiwanej biblioteki.

Jak pokazano w na przykładzie w zał. 5, narzędzia deweloperskie mogą zostać zainicjowane zarówno przed, jak i po załadowaniu strony internetowej. W związku z tym, tuż po uruchomieniu skryptu, rozpoczynane jest nasłuchiwanie na wiadomość od skryptu zawartości, informującą o wykryciu biblioteki. Jeżeli jednak, narzędzia deweloperskie zostały otwarte po załadowaniu strony, taka wiadomość nie zostanie odebrana. Skrypt zatem, równoległe z nasłuchiowaniem, wysyła wiadomość z zapytaniem o wykryte biblioteki.

Jeżeli skrypt otrzyma wiadomość lub pozytywną odpowiedź na zapytanie, tworzony jest nowy panel narzędzi deweloperskich (podrozdz. 9.10) przy użyciu funkcji `chrome.devtools.panels.create`.

Takie podejście, zapewnia większą niezawodność i responsywność względem rozwiązania opartego na aktywnym, zapętlonym w nieskończoność odpytywaniu skryptu zawartości, jak to ma miejsce w RDT (podrozdz. 7.1.4).

9.4. Abstrakcje nad mechanizmami komunikacji

API rozszerzeń Chrome pozwala na wysyłanie wiadomości o dowolnej strukturze. Daje to programiście dużą swobodę w określeniu sposobu komunikacji, jednak bezpośrednie użycie wywołań API w kodzie źródłowym może prowadzić do błędów związanych z rozbieżnością typów przesyłanych i oczekiwanych wiadomości.

Jako że projekt przygotowywany jest z wykorzystaniem języka TypeScript, powzięto decyzję o stworzeniu abstrakcji nad mechanizmami komunikacji, które pozwolą na ścisłą kontrolę typów przesyłanych treści.

9.4.1. Abstrakcja nad mechanizmem wiadomości jednorazowych

(shared/chrome/chrome-message.ts)

Dla mechanizmu opisanego w podrozdz. 4.3.1, w pierwszej kolejności, zdefiniowane zostały typy wiadomości, które mogą być przesyłane między skryptami. Obiekty zgodne z definicjami tych typów zawierają następujące pola:

- type - typ wiadomości,
- source - nadawca wiadomości,
- content - opcjonalna treść wiadomości,
- responseCallback - opcjonalna funkcja wywołania zwrotnego.

Określone zostały następujące typy typy wiadomości:

- LIBRARY_ATTACHED,
- WHAT_LIBRARIES_ATTACHED.

Następnie stworzone zostały funkcje pomocnicze, które umożliwiają wysyłanie i nasłuchiwanie na wiadomości zgodnych z określonymi typami.

Listing 32. Sygnatury funkcji abstrakcji nad mechanizmem wiadomości jednorazowych.

```
function sendChromeMessage(message: ChromeMessage): void;
function sendChromeMessageToTab(
  tabId: number,
  message: ChromeMessage
): void;
function onChromeMessage(
  callback: (
    message: ChromeMessage,
    sender: chrome.runtime.MessageSender
  ) => void
): () => void;
```

W list. 32 typ ChromeMessage jest dyskryminującą unią (ang. „discriminating union”, [85]) między typami wiadomości po polu type.

Funkcja onChromeMessage zwraca funkcję, której wywołanie kończy nasłuchiwanie na wiadomości.

9.4.2. Abstrakcja nad mechanizmem długotrwałych połączeń

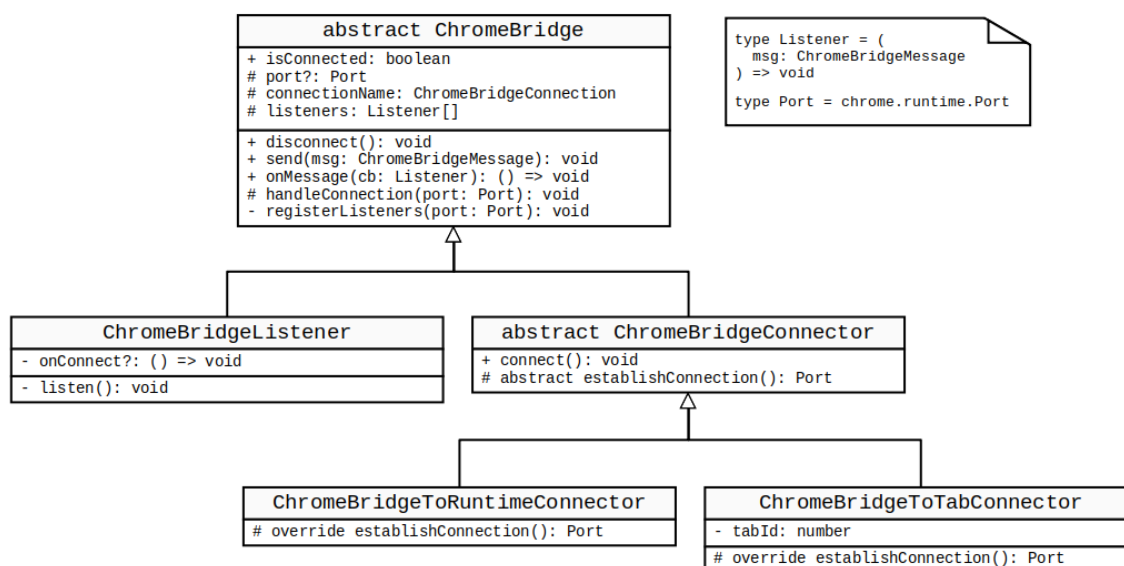
(shared/chrome/ChromeBridge.ts)

Dla mechanizmu opisanego w podrozdz. 4.3.2, podobnie jak w podrozdz. 9.4.1, zdefiniowane zostały następujące typy wiadomości:

- FULL_SKELETON,
- INSPECTED_DATA,
- INSPECT_ELEMENT,
- HOVER_ELEMENT.

Definicja abstrakcji nad długotrwałymi połączeniami nie składa się jednak z funkcji a z klas:

- ChromeBridge - abstrakcyjna klasa, po której dziedziczą pozostałe, wymienione poniżej,
- ChromeBridgeConnector - abstrakcyjna klasa, po której dziedziczą klasy reprezentujące połączenia ze strony nawiązującej połączenie,
- ChromeBridgeToRuntimeConnector - klasa reprezentująca połączenie ze strony nawiązującej połączenie do strony nasłuchującej, uruchomionej w kontekście przeglądarki (np. do skryptu tła),
- ChromeBridgeToTabConnector - klasa reprezentująca połączenie ze strony nawiązującej połączenie do strony nasłuchującej, uruchomionej w kontekście karty przeglądarki,
- ChromeBridgeListener - klasa reprezentująca połączenie ze strony nasłuchującej.



Rysunek 9.2. Diagram UML klas abstrakcji nad mechanizmem połączeń długotrwałych.

W celu nawiązania połączenia obiekty klas dziedziczących po ChromeBridgeConnector wymagają wywołania metody connect. ChromeBridgeListener natomiast, rozpoczyna nasłuchiwanie zaraz po utworzeniu obiektu. Jego konstruktor może przyjąć jako argument, funkcję wywołania zwrotnego, wywoływaną przy nawiązaniu połączenia.

Wszystkie klasy umożliwiają „dodanie” do ich obiektów funkcji nasłuchujących na wiadomości (`onMessage`) jeszcze przed nawiązaniem połączenia. Są one przechowywane w tablicy `listeners` i rejestrowane w momencie połączenia.

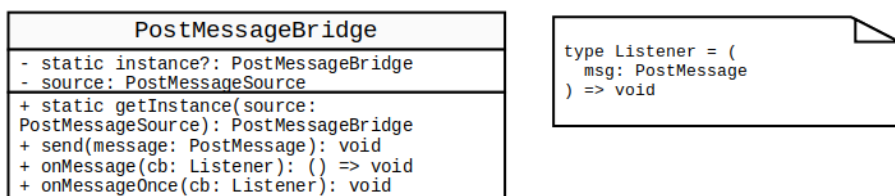
Funkcja `onMessage` zwraca funkcję, której wywołanie kończy nasłuchiwanie na wiadomości oraz usuwa funkcję z tablicy `listeners`.

9.4.3. Abstrakcja nad mechanizmem `windows.postMessage` (`pages/content/shared/PostMessageBridge.ts`)

Podobnie jak dla opisanych wcześniej mechanizmów, zdefiniowane zostały typy wiadomości:

- `LIBRARY_ATTACHED`,
- `MOUNT_ROOTS`,
- `MOUNT_NODES`,
- `UPDATE_NODES`,
- `UNMOUNT_NODES`,
- `INSPECTED_DATA`,
- `INSPECT_ELEMENT`,
- `HOVER_ELEMENT`.

Mechanizm abstrakcji nad `windows.postMessage` jest realizowany przez klasę singletonową `PostMessageBridge`.



Rysunek 9.3. Diagram UML klasy abstrakcji nad mechanizmem `windows.postMessage`.

Mechanizm `windows.postMessage`, w przeciwieństwie do mechanizmów API rozszerzeń Chrome, powoduje wywołanie wszystkich zarejestrowanych funkcji nasłuchujących, nawet w tym samym skrypcie, w którym została wysłana wiadomość. W autorskim rozwiązaniu, transfer wiadomości poprzez `windows.postMessage` jest wykorzystywany do przekazywania informacji między skrypcem zawartości uruchomionym w głównym świecie wykonywania a skrypcem zawartości uruchomionym w izolowanym świecie wykonywania (podobnie jak w RDT; podrozdz. 7.4).

Mając na uwadze powyższe, użycie klasy singletonowej (zgodnie z wzorcem projektowym *Singleton*) pozwala na zachowanie jednej instancji obiektu `PostMessageBridge` dla każdego kontekstu (głównego i izolowanego). Umożliwia także odfiltrowanie wiadomości wysyłanych w obrębie tego samego kontekstu (na podstawie wartości enum `PostMessageSource`).

Konstruktor klasy `PostMessageBridge` jest prywatny, a ponowne wywołanie funkcji `getInstance` w tym samym kontekście oraz z inną wartością `PostMessageSource`, kończy się błędem.

9.5. Uniwersalna reprezentacja elementów

Każda biblioteka frontendowa wykorzystuje inne struktury danych do przechowywania informacji o elementach. W związku z tym, koniecznym jest stworzenie uniwersalnej reprezentacji elementów, która jednak pozwoli na ewentualne rozszerzenie o specyficzne dla danej biblioteki informacje.

Autorskie narzędzie opiera się na pracy z obiektami `ParsedNode`, które muszą zawierać następujące pola:

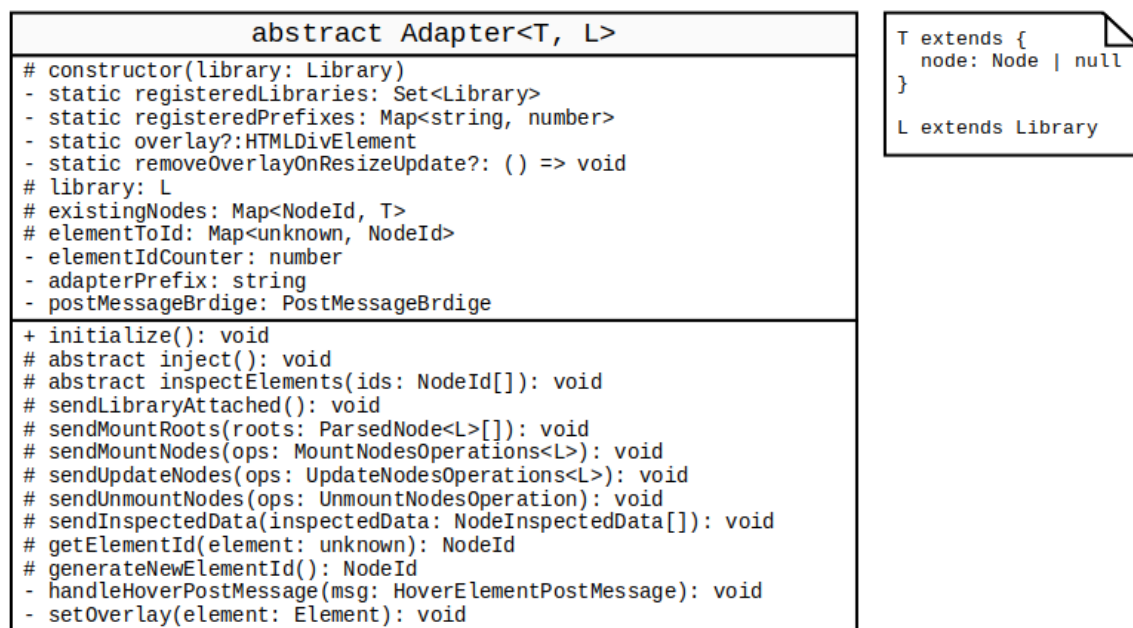
- `id` - unikalny identyfikator elementu,
- `name` - informatywna nazwa elementu,
- `children` - tablica obiektów `ParsedNode`, reprezentujących dzieci elementu.

Obiekty te dodatkowo mogą zawierać pola charakterystyczne dla elementów danej biblioteki. Zdecydowana większość funkcjonalności narzędzia nie jest zależna od nich zależna, jednak fragmenty panelu narzędzi deweloperskich pozwalają na ich wykorzystanie w specjalnie do tego przeznaczonych miejscach (podrozdz. 9.10).

9.6. Łączenie się rozszerzenia z bibliotekami frontendowymi - „adaptery”

(`pages/content/content-main/Adapter.ts`)

Za obsługę komunikacji z bibliotekami frontendowymi odpowiedzialne są adaptery, czyli klasy dziedziczące po klasie abstrakcyjnej `Adapter`, przedstawionej na rys. 9.4.



Rysunek 9.4. Diagram UML klasy abstrakcyjnej `Adapter`.

Klasy dziedziczące po klasie `Adapter` muszą do jej konstruktora przekazać wartość typu `enum Library`, reprezentującą bibliotekę, z którą adapter ma się komunikować. Klasa zapewnia, żeby dla danej biblioteki nie został utworzony więcej niż jeden adapter.

Dodatkowo, klasa dla każdego stworzonego adaptera tworzy prefiks, składający się z dwóch pierwszych liter nazwy biblioteki, a w razie konfliktów między nazwami - także liczby je poprzedzającej (np. `re`, `1re`, `2re`). Prefiks ten jest wykorzystywany w metodzie `generateNewElementId` do generowania unikalnych identyfikatorów elementów.

Metoda `getElementId` pozwala uzyskać identyfikator elementu, który był już wcześniej przypisany do elementu, a jeżeli element nie posiada identyfikatora, generuje nowy.

Klasa implementuje abstrakcję nad wysyłaniem informacji do izolowanego skryptu zawartości (podrozdz. 9.9) poprzez metody odpowiadające poszczególnym typom wiadomości `PostMessageBridge`:

- `sendLibraryAttached`,
- `sendMountRoots`,
- `sendMountNodes`,
- `sendUpdateNodes`,
- `sendUnmountNodes`,
- `sendInspectedData`.

W ramach pracy przygotowano adaptory dla wybranych w podrozdz. 3.4 bibliotek frontendowych (React i Svelte). Szczegóły ich implementacji zostaną przedstawione w kolejnych podrozdziałach (podrozdz. 9.7, podrozdz. 9.8).

9.6.1. Abstrakcyjne metody klasy `Adapter`

Każdy adapter musi zaimplementować metody `inject` oraz `inspectElements`.

Pierwsza z wymienionych, wywoływana jest w momencie inicjalizacji adaptera (przy wywołaniu `initialize()`) i powinna zawierać kod odpowiadający za połączenie się z biblioteką frontendową.

Druga zaś, wywoływana jest w reakcji na wiadomość `INSPECT_ELEMENT` z panelu narzędzi deweloperskich. Jako argument przyjmuje identyfikatory elementów, które mają zostać zbadane. Oczekuje się od poszczególnych adapterów, że odpowiednio obsługują przekazane identyfikatory i w reakcji na zmiany wyszczególnionych elementów, wywołują metodę `sendInspectedData`, przekazując ich aktualne dane.

Przy otrzymaniu wiadomości `INSPECT_ELEMENT` dla każdego z adapterów, wyfiltrowywane są identyfikatory elementów, obsługiwanych przez konkretny adapter, toteż otrzymanie w wiadomości identyfikatorów należących tylko do jednej z bibliotek skutkować będzie przekazaniem do pozostałych adapterów pustych tablic, które mogą być zinterpretowane jako zakończenie procesu nasłuchiwanie na zmiany właściwości elementów.

Struktura danych przesyłanych w wiadomości `INSPECTED_DATA` (przedstawiona na list. 33) została tak zaprojektowana, aby być zupełnie niezależną od obsługiwanych bibliotek frontendowych, np. komponenty funkcyjne w React mogą wymagać przesłania danych o używanych hookach i właściwościach, podczas gdy pojęcie hooków w Svelte zupełnie nie występuje.

Listing 33. Struktura danych przesyłanych w wiadomości INSPECTED_DATA.

```
type NodeInspectedData = {  
  library: Library;  
  id: NodeId;  
  name: string;  
  nodeInfo: Array<{ label: string; value: string }>;  
  nodeData: NodeDataGroup[];  
};  
  
type NodeDataGroup = {  
  group: string;  
  data: Array<{ label: string; value: InspectData }>;  
};
```

Dane przekazywane w właściwości `nodeData` to tablica obiektów, której każdy reprezentuje grupę danych, posiadającą swoją nazwę (np. „Props” czy „Hooks”) oraz tablicę obiektów, będących poszczególnymi wartościami dla danej grupy. Każda wartość posiada swoją etykietę oraz wartość typu `InspectData`, będącej wynikiem funkcji pomocniczej `dehydrate`, opisanej w podrozdz. 9.6.2.

9.6.2. Dehydratacja danych

Tak jak wspomniano w podrozdz. 7.4.1, mechanizm `windows.postMessage` wymaga serializacji i deserializacji przesyłanych danych. Nie dość, że dla złożonych obiektów może to być czasochłonne, to dodatkowo, próba serializacji niektórych obiektów, takich jak funkcje czy obiekty drzewa DOM, kończy się błędem.

Stan czy właściwości komponentów, mogą zawierać dowolne typy danych, często posiadając setki właściwości i przechowując wielokrotnie złożone obiekty czy nawet odwołania cykliczne.

Mając na uwadze powyższe obserwacje, koniecznym było stworzenie mechanizmu, który pozwoli na transformację i zoptymalizowanie ilości przesyłanych między skryptami danych, jednak w taki sposób, aby przekazać jak najistotniejsze informacje.

W tym celu powstała funkcja `dehydrate`.

Listing 34. Sygnatura funkcji dehydratującej dane.

```
function dehydrate(  
  value: unknown, depth: number = 0  
) : InspectData
```

Wartości w JavaScript mogą być jednym z 7 podstawowych typów:

- `string`,
- `number`,
- `boolean`,
- `null`,
- `undefined`,

- symbol,
- bigint.

Wszystkie pozostałe wartości są obiektami, w tym także, tablice i funkcje [86]. Niepraktycznym z punktu widzenia użytkownika, oraz nieefektywnym byłoby jednak przesyłanie np. wszystkich właściwości obiektu `Date` czy `RegExp`. Dużo lepszym rozwiązaniem zdaje się być przesłanie wartości, która dla tego rodzaju obiektów, wystarczy dla ich identyfikacji i analizy (np. `23 May 2023 00:00:00 GMT` dla obiektu `Date`).

Z tego powodu, w oparciu o rozwiązanie zastosowane w RDT, zdefiniowane zostały warianty typów danych przedstawione w list. 35.

Listing 35. Warianty typów danych rozpatrywane w procesie dehydratacji.

```
enum DataType {
  NULL = "NULL",
  UNDEFINED = "UNDEFINED",
  HTML_ELEMENT = "HTML_ELEMENT",
  HTML_ALL_COLLECTION = "HTML_ALL_COLLECTION",
  BOOLEAN = "BOOLEAN",
  FUNCTION = "FUNCTION",
  STRING = "STRING",
  SYMBOL = "SYMBOL",
  BIGINT = "BIGINT",
  INFINITY = "INFINITY",
  NAN = "NAN",
  NUMBER = "NUMBER",
  REACT_ELEMENT = "REACT_ELEMENT",
  ARRAY = "ARRAY",
  TYPED_ARRAY = "TYPED_ARRAY",
  ARRAY_BUFFER = "ARRAY_BUFFER",
  DATA_VIEW = "DATA_VIEW",
  REGEXP = "REGEXP",
  DATE = "DATE",
  ITERATOR = "ITERATOR",
  OPAQUE_ITERATOR = "OPAQUE_ITERATOR",
  CLASS_INSTANCE = "CLASS_INSTANCE",
  OBJECT = "OBJECT",
  UNKNOWN = "UNKNOWN",
}
```

Funkcja `dehydrate` analizuje przekazaną wartość i na podstawie ustalonego typu, tworzy obiekt `InspectData`, który zawiera pole `type` oraz dla niektórych typów dodatkowo pole `value`. Pierwsze z nich przechowuje jedną z przedstawionych w list. 35 wartości enum, a drugie - potencjalną reprezentację wartości. Na podstawie pola `type`, panel narzędzi deweloperskich może odpowiednio dobrać wykorzystany sposób prezentacji dla określonej danej.

Przy tworzeniu obiektu `InspectData` dla niektórych typów, takich jak np. `ARRAY` czy `OBJECT`, funkcja `dehydrate` wywoływana jest rekurencyjnie dla każdego elementu tablicy czy właściwości obiektu, z ograniczeniem głębokości do 5. W przypadku przekroczenia tej wartości, zwrócony zostaje obiekt `{type: 'MAX_DEPTH'}`.

Potencjalne rozszerzenie rozwiązania w przyszłości mogłoby pozwalać użytkownikowi na sterowanie limitem głębokości rekurencyjnej dehydratacji i samodzielne znalezienie odpowiedniego kompromisu między wydajnością a szczegółowością otrzymywanych danych (podrozdz. 12.1)

9.6.3. Podświetlanie elementów drzewa DOM

Od adapterów, dziedziczących po klasie `Adapter`, oczekiwane jest, że będą aktualizować mapę wszystkich istniejących w aplikacji w danym momencie elementów w mapie `existingNodes`. Typ mapy, wymaga aby wartości w niej przechowywane posiadały właściwość `node`, zawierającą referencję do odpowiadającego danemu elementowi biblioteki, węzła w drzewie DOM.

W reakcji na żądanie `HOVER_ELEMENT`, przychodzące z panelu narzędzi deweloperskich, klasa dodaje do strony półprzezroczysty element `div` o położeniu i rozmiarach odpowiadających elementowi w drzewie DOM oraz rejestruje nasłuchiwanie na zdarzenie `resize` okna w celu aktualizacji podświetlenia.

9.7. Implementacja adaptera dla Reacta

Adapter dla Reacta, dziedziczący po klasie `Adapter`, łączy się z Reactem wykorzystując przewidziany dla RDT hook (podrozdz. 7.2).

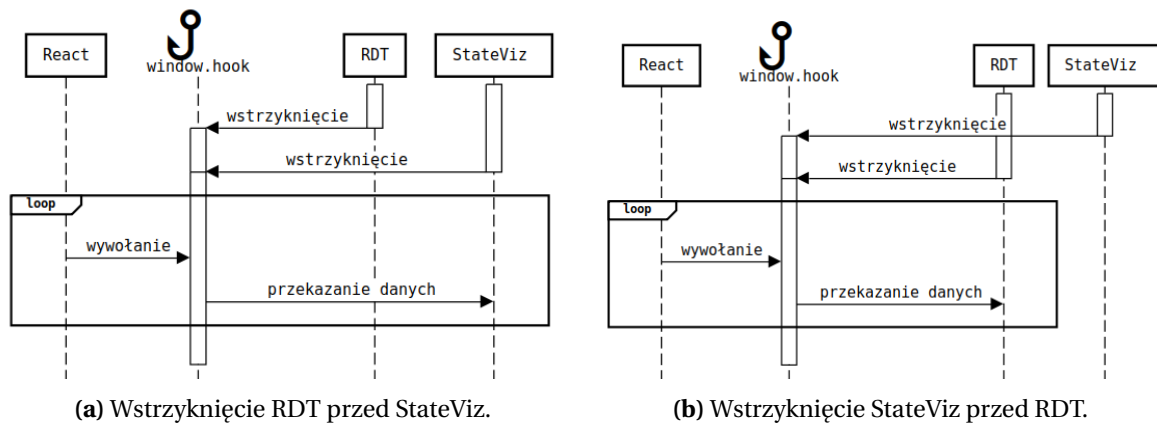
9.7.1. Mechanizm wstrzykiwania hooka i kompatybilność z RDT

Inne nieoficjalne rozwiązania, takie jak `Realize` (podrozdz. 2.1.2) czy `React Sight` (podrozdz. 2.1.3), wymagają do działania równoległego uruchomienia RDT, który zajmuje się wstrzyknięciem hooka do aplikacji. Rozwiązania te oczekują przez z góry określony czas, zakładając, że wystarczy on na uruchomienie RDT, a następnie wykorzystują tzw. „monkey patching” [87] do nadpisania wywoływanych przez Reacta funkcji.

`StateViz`, zgodnie z założeniami opisanymi w podrozdz. 9.1, nie wymaga dodatkowych zależności, dlatego adapter dla Reacta musi samodzielnie wstrzyknąć hook do aplikacji. Wiąże się to jednak z możliwością konfliktu autorskiego rozszerzenia z RDT w sytuacji uruchomienia obydwu rozszerzeń jednocześnie.

Jak widać na rys. 9.5, właściwość obiektu `window` zostanie nadpisana przez to rozszerzenie, które wstrzyknie hooka później, co uniemożliwi poprawne działanie wcześniej zainicjowanego rozszerzenia.

Problem ten mógłby zostać poniekąd rozwiązany poprzez zastosowanie mechanizmu podobnego do używanego we wspomnianych nieoficjalnych rozszerzeniach (`Realize` i `React Sight`), rozwiniętego o możliwość samodzielnego wstrzyknięcia hooka, gdy nie uczyni tego RDT. Diagram sekwencji dla takiego rozwiązania przedstawiono na rys. 9.6.



Rysunek 9.5. Możliwe wyniki wyścigu między RDT a StateViz przy wstrzykiwaniu hooka.

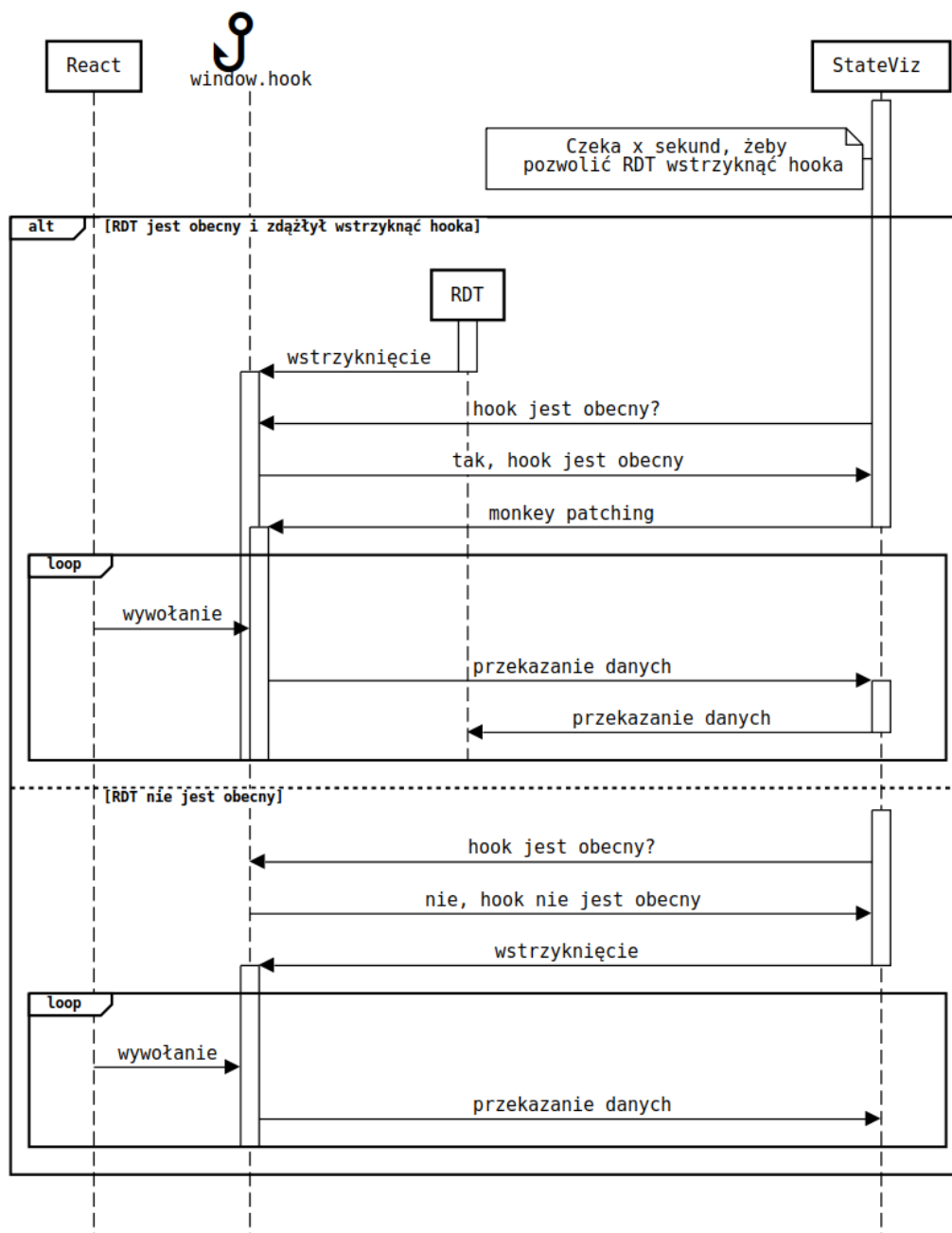
Mimo, że rozwiązanie pozwoliłoby na jednoczesne działanie obydwu rozszerzeń, to wymuszenie na użytkowniku oczekiwania na aktywację StateViz, mogłoby być dla niego uciążliwe. Jednocześnie nie zagwarantowałoby ono poprawnego działania w przypadku, gdy czas oczekiwania byłby zbyt krótki i RDT nie zdążyłby wstrzyknąć hooka (sytuacja przedstawiona na rys. 9.7).

Co więcej, sytuacja, w której użytkownik potrzebowałby jednocześnie korzystać z obydwu rozszerzeń, zdaje się być mało prawdopodobna. Z tego względu zdecydowano się na zastosowanie prostszego rozwiązania.

Adapter sprawdza, czy właściwość `__REACT_DEVTOOLS_GLOBAL_HOOK__` obecna jest w globalnym obiekcie `window`. Jeżeli tak, w konsoli narzędzi deweloperskich wyświetlany jest błąd informujący użytkownika o konieczności dezaktywacji RDT. W przeciwnym wypadku, adapter wstrzykuje hooka do aplikacji.

Dodatkowym zabezpieczeniem, przed nadpisaniem wstrzykiwanego hooka w przypadku późniejszej aktywacji RDT, jest dodanie do obiektu hooka (poza właściwościami opisanymi w podrozdz. 9.7.2), właściwości `stateViz` i sprawdzenie, po upływie 5 sekund, czy właściwość wciąż jest obecna. Jeżeli nie (hook został nadpisany), również wyświetlony zostanie błąd.

Potencjalnie najlepszym rozwiązaniem byłoby wykorzystanie podobnego, do przedstawionego na rys. 9.6, warunkowego wstrzykiwania, bądź używania mechanizmu monkey patching. Jednak, zamiast oczekiwania przez określoną ilość czasu podczas inicjalizacji adaptera, można by było wykorzystać możliwości komunikacji między rozszerzeniami (oferowaną przez API rozszerzeń Chrome) w celu wykrycia czy w przeglądarce aktywne jest RDT. Przy niewykryciu instalacji RDT można byłoby wstrzyknąć hooka, a w przypadku przeciwnym - z ustalonym interwałem sprawdzać czy hook został wstrzyknięty i ostatecznie zastosować monkey patching. Funkcjonalność tę pozostawiono jednak jako potencjalne rozszerzenie rozwiązania w przyszłości (podrozdz. 12.1).



Rysunek 9.6. Powodzenie w przypadku oczekiwania StateViz na wstrzyknięcie hooka przez RDT.

9.7.2. Wstrzykiwany hook

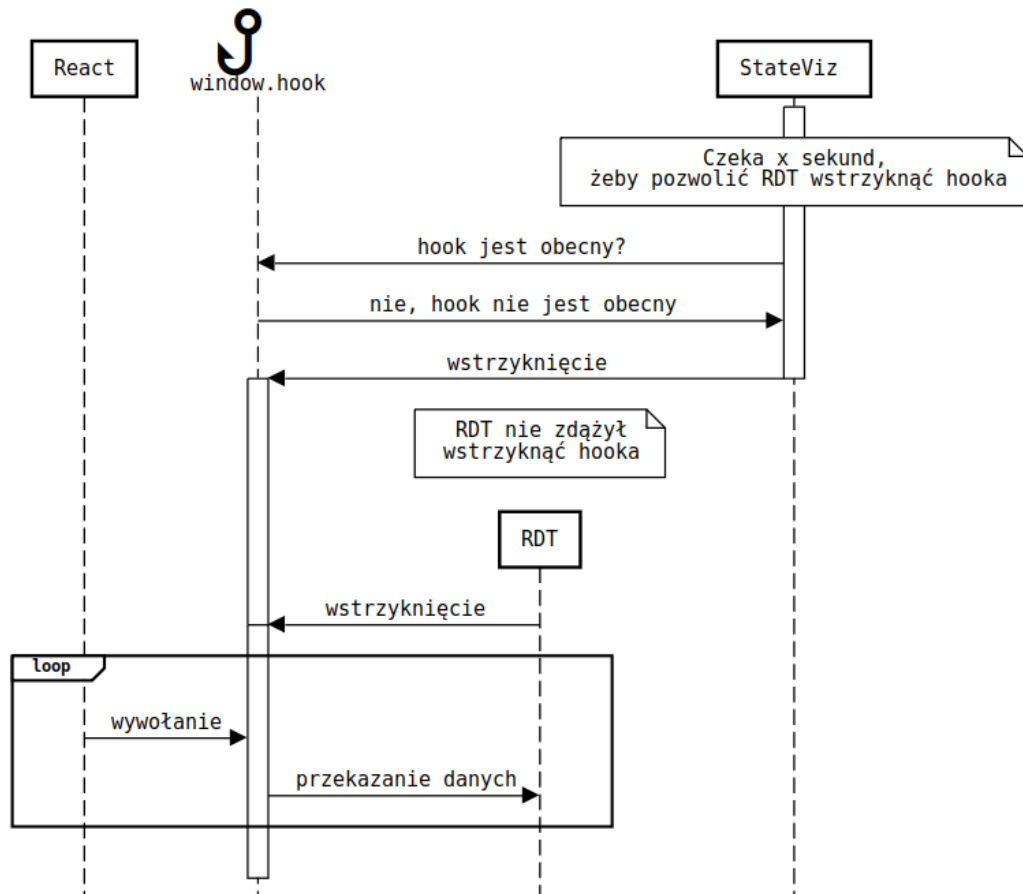
Biorąc pod uwagę fakt, że komunikacja z Reactem odbywa się poprzez przygotowany dla RDT hook, jego struktura jest analogiczna do tej przedstawionej w list. 23, jednak z ograniczoną do niezbędnych liczbą właściwości.

Listing 36. Instalacja hooka dla Reacta przez adapter.

```

window.__REACT_DEVTOOLS_GLOBAL_HOOK__ = {
  stateViz: true,
  renderers: this.renderers,
  supportsFiber: true,

```



Rysunek 9.7. Niepowodzenie w przypadku zbyt krótkiego oczekiwania StateViz na wstrzyknięcie hooka przez RDT.

```

inject: (renderer) => this.handleInject(renderer),
checkDCE: () => {},
onCommitFiberUnmount: (_, fiber) => this.unmountFiber(fiber),
onCommitFiberRoot: (...args) =>
  this.handleCommitFiberRoot(...args)
};
  
```

Właściwości wstrzykiwanego hooka to kolejno:

- `stateViz` - flaga informująca o aktywacji StateViz. Jej zastosowanie opisane zostało w podrozdz. 9.7.1;
- `renderers` - mapa podłączonych rendererów (podrozdz. 7.2.1). Zasadniczo nie jest ona wykorzystywana przez autorskie narzędzie, jednak jej brak powoduje błąd przy inicjacji mechanizmu „Fast Refresh” Reacta (list. 37), skutkujący zatrzymaniem działania aplikacji;
- `supportsFiber` - flaga informująca Reacta o gotowości rozszerzenia do pracy z Fiber (podrozdz. 7.2.3);
- `inject` - funkcja wywoływana przez Reacta w momencie podłączenia nowego renderera (podrozdz. 7.2.4). Obsługa danych przekazywanych w tej funkcji została opisana w podrozdz. 9.7.3;

- `checkDCE` - funkcja wywoływana przez Reacta, aby rozszerzenie mogło określić tryb uruchomionej aplikacji (np. produkcyjny, deweloperski, podrozdz. 7.2.5). W autorskim rozwiązaniu funkcjonalność ta nie jest wykorzystywana (przekazanie pustej funkcji), jednak jej brak skutkuje komunikatem w konsoli zalecającym instalację React DevTools (list. 38);
- `onCommitFiberUnmount` - funkcja wywoływana przez Reacta w momencie odmontowywania elementu (podrozdz. 7.2.7). Obsługę danych przekazywanych w tej funkcji opisano w podrozdz. 9.7.5;
- `onCommitFiberRoot` - funkcja wywoływana przez Reacta przy każdej aktualizacji drzewa Fiber (podrozdz. 7.2.6). Obsługę danych przekazywanych w tej funkcji opisano w podrozdz. 9.7.4.

Listing 37. Błąd przy inicjacji mechanizmu „Fast Refresh” Reacta, gdy w zainstalowanym hooku brakuje mapy podłączonych rendererów.

```
Uncaught TypeError: Cannot read properties of undefined
(reading 'forEach')
    at Object.injectIntoGlobalHook (@react-refresh:451:20)
    at (index):6:16
```

Listing 38. Komunikat wyświetlany przez Reacta w konsoli przeglądarki, gdy w zainstalowanym hooku brakuje właściwości `checkDCE`.

```
Download the React DevTools for a better development experience:
https://reactjs.org/link/react-devtools
```

9.7.3. Łączenie się Reacta z rozszerzeniem

Adapter, w reakcji na przekazaną przez Reacta do funkcji `inject` referencję do renderera, w pierwszej kolejności sprawdza jego wersję. Jeżeli przekazany obiekt nie posiada właściwości `version` lub jest ona mniejsza niż 16 (React 16 wprowadził Fiber), to adapter wyświetla błąd w konsoli informujący o niekompatybilności wersji Reacta z autorskim narzędziem.

W przeciwnym wypadku, adapter generuje identyfikator dla przekazanego renderera, zapisuje go w mapie `renderers` i wysyła do izolowanego skryptu zawartość wiadomości `LIBRARY_ATTACHED`, informując, że podłączona została biblioteka Reacta.

9.7.4. Montowanie i aktualizacja elementów

Podobnie jak w RDT (podrozdz. 7.7), parsowanie drzewa Fiber rozpoczyna się od korzenia przekazywanego przez Reacta poprzez `onCommitFiberRoot`.

W autorskim rozwiązaniu nie są jednak wykorzystywane tablice operacji (podrozdz. 7.8.1), a rekursywnie budowane są obiekty `ParsedNode` (podrozdz. 9.5) reprezentujące nowozamontowane poddrzewa Fiber.

Jeżeli zamontowania wymaga całe drzewo Fiber, to w wyniku wywołania przez Reacta `onCommitFiberRoot`, zostanie zbudowany obiekt `ParsedNode` reprezentujący korzeń drzewa,

a poprzez właściwość `children`, rekursywnie, również wszyscy jego potomkowie. Korzeń jest następnie wysyłany jako zawartość wiadomości `MOUNT_ROOTS`.

Jeżeli korzeń drzewa Fiber nie zmienia się, w funkcji `getNodesUpdates` rozpoczęty zostaje proces wykrywania zmian. Algorytm podobnie jak w przypadku RDT, rekursywnie przemieszcza się po drzewie, sprawdzając czy poszczególne węzły posiadają wersję alternatywną i jeżeli nie, to od danego węzła, ponownie budowane jest poddrzewo `ParsedNode`. Funkcja zwraca tablicę zawierającą wszystkie wymagające zamontowania poddrzewa. Tablica ta, wraz z innymi wymaganymi informacjami (list. 43), wysyłana jest jako zawartość wiadomości `MOUNT_NODES`.

W procesach zapoczątkowanych przez wywołanie `onCommitFiberRoot`, nie jest wykonywane wykrywanie odmontowywania elementów. Usunięcie elementów z drzewa jest osobno zgłaszane przez Reacta poprzez wywołanie `onCommitFiberUnmount` (podrozdz. 9.7.5).

Nazwy przesyłanych elementów `ParsedNode`, tworzone są tak, aby w jak najbardziej intuicyjny sposób przedstawić użytkownikowi najważniejsze informacje o danym węźle, np.:

- `ExampleComponent` - nazwa funkcji lub klasy dla komponentów,
- `<div>` - nazwa elementu, otoczonego nawiasami trójkątnymi dla elementów DOM,
- `"Hello, World!"` - zawartość, otoczona cudzysłowami dla węzłów tekstowych.

Wadą autorskiego rozwiązania w porównaniu do RDT, jest brak pomijania trawersowania niezmiennych poddrzew Fiber przy wykrywaniu zmian (podrozdz. 7.7.2). W przypadku dużych aplikacji, w których zmiany dotyczą niewielkiej części drzewa, takie podejście może wpłynąć na wydajność systemu. Funkcjonalność ta nie została zaimplementowana ze względu na różnicę między sposobem funkcjonowania systemu odpowiedzialnego za wykrywanie i przekazywanie zmian w stanie i właściwościach komponentów (podrozdz. 9.7.6).

Każdy nowo zamontowany obiekt Fiber zapisywany jest w mapie `existingNodes`, zawierającej obiekty o strukturze przedstawionej w list. 39. Poszczególne właściwości obiektu to:

- `parentId` - identyfikator rodzica węzła. Jego użycie zostało opisane w podrozdz. 9.7.5;
- `fiber` - referencja do obiektu Fiber, pozwalająca na późniejszą analizę stanu i właściwości komponentów (podrozdz. 9.7.6);
- `node` - referencja do odpowiadającego elementu drzewa DOM, zapisywana w mapie, zgodnie z wymaganiem przedstawionym w podrozdz. 9.6.3.

Listing 39. Struktura danych przechowywanych w mapie `existingNodes` dla Reacta.

```
{
  parentId: NodeId | null;
  fiber: Fiber;
  node: Node | null;
}
```

W przypadku Reacta, referencja do obiektu `Node` uzyskiwana jest na podstawie właściwości obiektu `Fiber` `stateNode`. Bazą dla tego rozwiązania była samodzielna analiza obiektów `Fiber` i w konsekwencji - zauważenie, że dla obiektów typu `HostComponent`, czyli odpowiadających elementom DOM, właściwość `stateNode` zawiera referencję do odpowiednich węzłów w aktualnym drzewie dokumentu.

Niektóre obiekty `Fiber`, takie jak np. `FunctionComponent`, nie posiadają właściwej reprezentacji w drzewie DOM (czyli też fizycznego położenia i rozmiaru na stronie). W takim wypadku logika podpowiada, że podświetlony powinien zostać pierwszy potomek danego węzła, który takową posiada. W tym celu zaimplementowana została funkcja przedstawiona w list. 40.

Listing 40. Funkcja znajdująca najbliższy element DOM dla obiektów `Fiber`

```
function getNearestStateNode(fiber: Fiber): Node | null {
  let current: Fiber | null = fiber;
  let stateNode = fiber.stateNode;
  while (current !== null && !(stateNode instanceof Node)) {
    current = current.child;
    stateNode = current?.stateNode;
  }
  if (!(stateNode instanceof Node)) {
    return null;
  }
  return stateNode;
}
```

Rozwiązanie to nie zawsze skutkuje oczekiwanym rezultatem, jednak w większości przypadków, pozwala na poprawną identyfikację odpowiedniego elementu. Poprawa tego mechanizmu (potencjalnie oparta na analizie kodu źródłowego Reacta) została pozostawiona jako ewentualne rozszerzenie rozwiązania w przyszłości (podrozdz. 12.1).

9.7.5. Odmontowywanie elementów

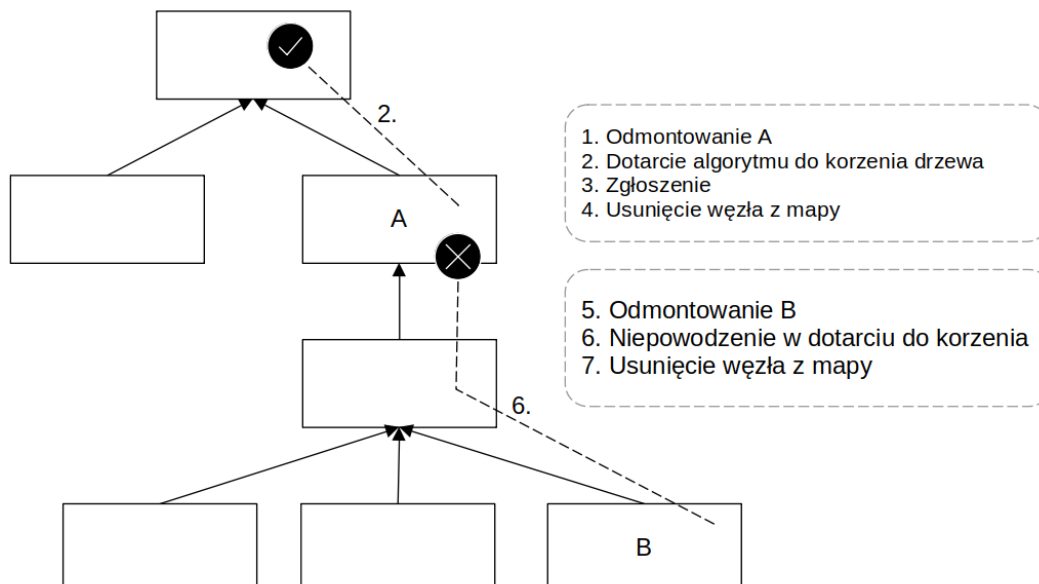
React wywołuje funkcję `onCommitFiberUnmount` dla każdego odmontowanego obiektu `Fiber` przekazując jego referencję.

Początkowo zaimplementowane, naiwne rozwiązanie polegało na bezpośrednim przekazywaniu do izolowanego skryptu zawartości, identyfikatorów wszystkich odmontowanych obiektów. W przypadku jednak, gdy usuwany z drzewa jest dany element, oczywistym jest, że wszyscy jego potomkowie również zostaną usunięci. Nie ma zatem potrzeby przesyłania informacji o każdym z nich.

Dostrzeżono również, że w ogromnej większości przypadków, choć nie zawsze (analizę dlaczego tak się dzieje pozostawiono jako aspekt do przeanalizowania w przyszłości, podrozdz. 12.1), React zgłasza odmontowanie węzłów w kolejności od najpłycej zagnieżdżonych.

Mając na uwadze te względy, zastosowano mechanizm, który dla każdego odmontowywanego węzła, na podstawie identyfikatorów rodziców przechowywanych w mapie

`existingNodes`, podąża do korzenia drzewa. Jeżeli algorytmowi nie uda się do niego dotrzeć, to znaczy, że któryś z przodków został już wcześniej odmontowany. W takiej sytuacji, informacja o usunięciu danego węzła nie jest przekazywana do izolowanego skryptu zawartości. Diagram obrazujący ten proces przedstawiony został na rys. 9.8.



Rysunek 9.8. Optymalizacja procesu wykrywania niezbędnych do przesłania danych o odmontowywanych węzłach.

Na koniec, każdy zgłoszony przez Reacta węzeł usuwany jest z mapy `existingNodes`.

Rozwiązanie to pozwala na znaczące ograniczenie ilości przesyłanych danych w przypadku odmontowywania dużych poddrzew, co ma znaczący wpływ na wydajność działania rozszerzenia, np. w przypadku nawigacji na stronie, skutkującej usunięciem i ponownym zamontowaniem niemal całego drzewa Fiber. Jednocześnie, zastosowane podejście, nie skutkuje utratą danych w momencie, gdy zgłoszenia odmontowania węzłów zdarzą się w innej kolejności, niż od najpłycej zagnieżdżonych.

9.7.6. Inspekcja stanu i właściwości komponentów

Proces analizowania stanu i właściwości komponentów jest relatywnie kosztowny obliczeniowo, dlatego jest on wykonywany tylko i wyłącznie dla elementów, których dane zostały zażądane przez panel narzędzi deweloperskich.

Po wywołaniu, wspomnianej w podrozdz. 9.6.1, metody `inspectElements`, z przekazanych identyfikatorów tworzony jest set i zapisywany w pamięci adaptera.

Następnie dla każdego obiektu Fiber odpowiadającego przekazanym identyfikatorom, dokonywane jest parsowanie stanu i właściwości. Proces ten omówiony został w podrozdz. 9.7.7. Po zakończeniu analizy dla wszystkich identyfikatorów, dane są przekazywane do izolowanego skryptu w wiadomości `INSPECTED_DATA`.

Zgodnie z opisem przedstawionym w podrozdz. 9.7.4, przy każdorazowym wywołaniu `onCommitFiberRoot`, odwiedzany jest każdy element drzewa Fiber. Pozwala to, na sprawdzenie dla wszystkich węzłów, czy zgłoszone zostało żądanie przekazania szczegółowych

informacji o ich stanach i właściwościach. Jeżeli dany komponent jest „obserwowany”, wykonywane jest odświeżenie przechowywanych o nim danych (podrozdz. 9.7.7). Po zakończeniu przetwarzania całego drzewa, podobnie jak po pierwotnym otrzymaniu żądania inspekcji, dane są wysyłane w wiadomości `INSPECTED_DATA`.

Rozwiązanie to, oparte na strategii „push” (w przeciwieństwie do „pull”, zastosowanej w RDT; podrozdz. 7.8), pozwala na, niemal natychmiastowe, odświeżenie wyświetlanych w panelu narzędzi deweloperskich danych. Jest to szczególnie ważne w przypadku dynamicznie zmieniających się aplikacji, jednak skutkuje brakiem możliwości odcinania drzew (podrozdz. 9.7.4).

Potencjalnym usprawnieniem mógłby być mechanizm, który włącza odcinanie, gdy żadne elementy nie są obserwowane lub rozwiązanie, które wykrywałoby poddrzewa niezawierające żadnych poddanych inspekcji węzłów. Propozycje te pozostawiono do rozważenia jako ewentualne rozszerzenia rozwiązania w przyszłości (podrozdz. 12.1).

9.7.7. Parsowanie szczegółowych danych komponentów

Dla każdego obiektu `Fiber`, dla którego zgłoszono żądanie inspekcji, wywoływana jest funkcja `getNodeData` zwracająca tablicę obiektów `NodeDataGroup` przedstawioną w list. 33.

Zwracana tablica zawiera nie więcej niż trzy grupy danych:

1. **Właściwości komponentu** („Props”) - zawierającą informacje o przekazanych do komponentu właściwościach podrozdz. 5.3,
2. **Konteksty komponentu** („Contexts”) - zawierającą informacje o wykorzystywanych w komponencie kontekstach podrozdz. 5.9.3,
3. **Stan komponentu**, w zależności od typu komponentu:
 - „State” - dla komponentów klasowych,
 - „Hooks” - dla komponentów funkcyjnych wykorzystujących hooki (podrozdz. 5.9).

Proces parsowania właściwości opiera się na analizie pola `memoizedProps` obiektów `Fiber`. Jest on jednak pomijany dla niektórych elementów wbudowanych w Reacta, których analiza właściwości nie dostarczyłaby użytkownikowi użytecznych informacji (np. `Fragment` czy `HostRoot`).

Pole `memoizedProps` może zawierać obiekt, z kluczami będącymi nazwami konkretnych właściwości i ich wartościami, bądź wartość prymitywną. W obydwu przypadkach, wartości są dehydratowane (podrozdz. 9.6.2) i w przypadku obiektu - zwracane wraz z nazwami kluczy.

Analiza stanu komponentów klasowych, podobnie jak w przypadku właściwości, polega na dehydratacji wartości przechowywanych w polu `memoizedState` obiektów `Fiber`.

Jak opisano w podrozdz. 7.10, proces analizowania hooków w RDT jest dość skomplikowany i kosztowny obliczeniowo (wymaga ponownego renderowania danego komponentu). Z tego powodu, w autorskim rozwiązaniu zastosowano prostsze rozwiązanie.

Wartości hooków (np. zachowana w `useState` czy `useRef` wartość) używanych w komponentach funkcyjnych również zapisane są we właściwości `memoizedState` obiektów `Fiber`. W przeciwieństwie do komponentów klasowych, nie jest to obiekt, a łączona lista

wartości, bez informacji o tym do jakiego hooka się odnoszą. Stwarza to problem, gdyż przedstawienie użytkownikowi samych wartości, bez wskazania źródła ich pochodzenia, nie byłoby dla niego zbyt informatywne.

Podczas poszukiwań rozwiązania i dogłębnej analizy struktury obiektów Fiber, zauważono, że jeżeli React uruchomiony jest w trybie deweloperskim, obiekty Fiber posiadają dodatkowe pole `_debugHookTypes`, zawierające nazwy wszystkich hooków użytych w danym komponencie funkcyjnym. Co więcej, kolejność ich jest zgodna z kolejnością wykorzystania w komponencie oraz z kolejnością wartości przechowywanych w łączonej liście `memoizedState` z pewnymi zaobserwowanymi wyjątkami.

Lista `memoizedState` nie zawiera wartości dla hooków:

- `useDebugValue`,
- `useContext`.

Zdecydowałem się rozwiązać wspomniany problem poprzez próbę dopasowania wartości do wspomnianej listy nazw hooków.

Rozwiązanie to, opiera się niejako na „inżynierii wstecznej” obiektów Fiber i mimo, że dla przeanalizowanych przypadków, sprawdza się, to, bez dogłębnej analizy pochodzenia wartości zawartych w `_debugHookTypes`, nie można zagwarantować poprawnego funkcjonowania dla wszystkich możliwych sytuacji. Dodatkową wadą jest to, że w przypadku, produkcyjnego trybu działania Reacta, pole to nie jest dostępne, co w obecnej implementacji skutkuje wyświetleniem napisu „Unknown” zamiast nazwy hooka.

Niewątpliwą zaletą rozwiązania jest natomiast, jego wydajność. Brak konieczności ponownego renderowania komponentów, pozwala na zdecydowanie szybsze uzyskanie danych o hookach.

Analiza wartości, wykorzystywanych w komponencie kontekstów, polega na dehydratacji wszystkich wartości znajdujących się w łączonej liście właściwości `dependencies` obiektu Fiber.

Przy testach rozwiązania zauważono jednak, że w przypadku komponentów funkcyjnych i Reacta uruchomionego w trybie deweloperskim, wartości zawarte w `dependencies` są zduplikowane. Nasunęło się więc podejrzenie, że jest to skutek podwójnego renderowania komponentów w trybie „strict mode” [72]. Problem ten dotyczy jednak tylko komponentów funkcyjnych (to znaczy, kontekstów używanych przez `useContext`) i tylko wtedy, gdy React jest w trybie deweloperskim. Wówczas jednak, jak wspomniano wcześniej, dostępna jest tablica `_debugHookTypes`.

Problem duplikacji kontekstów udało się zatem rozwiązać, zliczając wystąpienia nazwy hooka `useContext` w tablicy `_debugHookTypes` a następnie dehydratując i zwracając tylko odpowiednią liczbę wartości z `dependencies`.

Wspomniane problemy i wątpliwości pozostawiono do analizy i ewentualnej poprawy w przyszłości (podrozdz. 12.1).

9.8. Implementacja adaptera dla Svelte

W ramach pracy, oprócz adaptera dla Reacta, zaimplementowany został również adapter dla Svelte.

Implementacja sposobu łączenia się adaptera z biblioteką Svelte, opiera się na kodzie źródłowym oficjalnych narzędzi deweloperskich Svelte DevTools (podrozdz. 2.2.1).

Warto na wstępie nadmienić, że ze względu na konieczność ograniczenia się w adapterze do użycia interfejsów przygotowanych przez twórcę biblioteki (podrozdz. 3.1), które nie są aktywne w trybie produkcyjnym, adapter działa jedynie dla aplikacji Svelte uruchamianych w trybie deweloperskim. Problem ten dotyczy również oficjalnego narzędzia.

Analizę możliwości połączenia autorskiego rozszerzenia z produkcyjnymi aplikacjami Svelte zostały pozostawione jako aspekt do rozważenia w przyszłości (podrozdz. 12.1).

9.8.1. Wykrywanie biblioteki Svelte

Przy inicjalizacji adaptera, skrypt, w pierwszej kolejności, rejestruje nasłuchiwanie na zdarzenia emitowane przez bibliotekę Svelte (podrozdz. 9.8.2) w celu zapewnienia, że adapter nie utraci żadnych informacji przekazywanych z biblioteki.

Następnie, rejestrowane jest nasłuchiwanie na zdarzenie `DOMContentLoaded`, które sygnalizuje, że strona została załadowana i jest gotowa do interakcji. W momencie jego wywołania, adapter sprawdza obecność właściwości `__svelte` w obiekcie globalnym okna przeglądarki. Obiekt pod tym kluczem, dodawany jest przez Svelte i zawiera informacje o wersji używanej biblioteki.

W przypadku, gdy wspomnianego obiektu brak, lub numer wersji biblioteki jest mniejszy niż 4, adapter, w celu ograniczenia wpływu na wydajność systemu, usuwa nasłuchiwanie na zdarzenia emitowane przez bibliotekę i kończy swoje działanie, wyświetlając odpowiedni komunikat w konsoli przeglądarki. W przeciwnym wypadku wysyłana jest wiadomość `LIBRARY_ATTACHED` do izolowanego skryptu zawartości, informująca o wykryciu biblioteki Svelte.

Wsparcie dla wcześniejszych wersji biblioteki Svelte nie zostało zaimplementowane ze względu na ograniczenia czasowe, jednak potencjalnie możliwym jest rozszerzenie o nie rozwiązania w przyszłości (podrozdz. 12.1).

9.8.2. Nasłuchiwanie na zdarzenia Svelte

Komunikacja ze Svelte odbywa się poprzez nasłuchiwanie na zdarzenia emitowane przez bibliotekę.

1. `SvelteRegisterComponent` - emitowane, gdy komponent Svelte zostaje zarejestrowany.
2. `SvelteRegisterBlock` - emitowane, gdy blok Svelte (podrozdz. 6.4) zostaje zarejestrowany;
3. `SvelteDOMInsert` - emitowane, gdy element DOM zostaje wstawiony do drzewa dokumentu.
4. `SvelteDOMRemove` - emitowane, gdy element DOM zostaje usunięty z drzewa dokumentu.

5. `SvelteDOMSetData` - emitowane, gdy dla elementu drzewa DOM, zmieniane są dane (np. zawartość tekstowa).
6. `SvelteDOMAddEventListener` - emitowane, gdy dla elementu drzewa DOM, zostaje zarejestrowane nasłuchiwanie na zdarzenia.
7. `SvelteDOMRemoveEventListener` - emitowane, gdy dla elementu drzewa DOM, nasłuchiwanie na zdarzenia zostaje usunięte.
8. `SvelteDOMSetProperty` - emitowane, gdy dla elementu drzewa DOM, zmieniane są właściwości.
9. `SvelteDOMSetAttribute` - emitowane, gdy dla elementu drzewa DOM, zmieniane są atrybuty.
10. `SvelteDOMRemoveAttribute` - emitowane, gdy dla elementu drzewa DOM, usuwane są atrybuty.

W autorskim rozwiązaniu wykorzystywanych jest tylko pierwszych 5. Pozostałe mogłyby zostać użyte w celu dodania dodatkowych funkcjonalności, takich jak np. wyświetlanie atrybutów elementów w panelu narzędzi deweloperskich (podrozdz. 12.1).

9.8.3. Obsługa rejestrowania komponentów

Tworząc implementację adaptera dla Svelte, przeszukano dokumentację biblioteki w celu pozyskania odpowiedzi na pytania: czym właściwie jest i kiedy odbywa się proces rejestrowania komponentów w bibliotece. Niestety nie znaleziono informacji na ten temat.

Fakt ten wymógł dokonanie analizy wzajemnej chronologii zdarzeń emitowanych przez Svelte i treści przekazywanych wraz z nimi. W rezultacie dociekań zauważono, że zdarzenie `SvelteRegisterComponent` emitowane jest zaraz po zdarzeniu `SvelteRegisterBlock` dla bloku odpowiadającego danemu komponentowi, ale zaraz przed samym jego zamontowaniem (wy tłumaczenie tego procesu przedstawiono w podrozdz. 9.8.6).

Zauważono, że od tej reguły odbiega jeden wyjątek, którym jest komponent będący korzeniem całej aplikacji. W tym przypadku, zdarzenie `SvelteRegisterComponent` dla tego elementu emitowane jest na samym końcu procesu inicjalizacji aplikacji.

Treść, przekazywana wraz ze zdarzeniem `SvelteRegisterComponent`, zawiera nazwę komponentu oraz referencję do bloku wewnętrznie używanego przez Svelte do jego reprezentacji. Na podstawie tej referencji, adapter tworzy unikalny identyfikator dla bloku komponentu.

Następnie, adapter w mapie `componentsCaptureStates`, pod kluczem będącym wygenerowanym identyfikatorem, zapisuje obiekt wykorzystywany później przy analizie stanu i właściwości komponentów. Obiekt ten zawiera dwa pola (list. 41), których wartości możliwe są do uzyskania jedynie w czasie rejestracji komponentu. Wykorzystanie tych wartości omówiono w podrozdz. 9.8.12.

Listing 41. Typ mapy zachowującej dane używane przy inspekcji komponentów Svelte.

```
componentsCaptureStates: Map<NodeId, {
  captureState: () => unknown;
  propsKeys: string[];
}>;
```

Nazwa komponentu nie jest dostępna w momencie jego montowania, dlatego też, adapter zachowuje ją w mapie `pendingComponents`, w celu wykorzystania jej później, przy montowaniu bloku odpowiadającego danemu komponentowi.

Wyjątkiem jest wspomniany wcześniej korzeń aplikacji. W momencie jego montowania, nazwa komponentu nie znajduje się jeszcze w mapie `pendingComponents`, dlatego też, adapter zgłasza zamontowanie z nazwą „Unknown”, a następnie dodaje identyfikator bloku do mapy.

Jeżeli w procesie rejestracji komponentu, adapter we wspomnianej mapie znajdzie jego identyfikator, znaczy to, że zgłoszenie zamontowanego bloku zostało już wysłane (podrozdz. 9.8.6), jednak bez (niedostępnej wówczas) nazwy komponentu. W takim przypadku, adapter usuwa z mapy `pendingComponents` identyfikator, a następnie wysyła właściwą nazwę w wiadomości `UPDATE_NODES`.

9.8.4. Wstrzykiwanie logiki adaptera do funkcji cyklu życia bloków

Zdarzenie `SvelteRegisterBlock` emitowane jest dla wszystkich bloków Svelte, zarówno tym odpowiadającym blokom logicznym (opisanym w podrozdz. 6.4), jak i komponentom.

Treść przekazywana wraz ze zdarzeniem zawiera referencję do bloku oraz jego typ.

Adapter, stosując „monkey patching” [87], wstrzykuje swoją logikę do funkcji obecnych w obiekcie bloku, odpowiedzialnych za montowanie, aktualizację i usuwanie bloku. Są to odpowiednio funkcje:

- `m: (target: Node, anchor: Node | null) => void` - montująca blok,
- `p: (changed: boolean, ctx: unknown) => void` - aktualizująca blok,
- `d: (detaching: boolean) => void` - usuwająca blok.

Logikę adaptera wstrzykiwaną do tych funkcji przedstawiono w kolejnych podrozdziałach.

9.8.5. Przechowywanie identyfikatora przetwarzanego bloku

Aby zrozumieć mechanizm działania adaptera dla Svelte, istotnym jest zrozumienie sposobu, w jaki biblioteka Svelte przetwarza bloki i jak dokładnie działa „monkey patching”.

„Monkey patching” polega na zachowaniu referencji do oryginalnej funkcji, następnie zastąpieniu jej nową funkcją, zawierającą wstrzykiwaną logikę, a na końcu, wywołaniu oryginalnej funkcji z odpowiednimi argumentami. Mechanizm pozwala zatem, na przechwycenie argumentów przekazywanych do oryginalnej funkcji i na ich analizę.

Svelte natomiast, w momencie montowania i aktualizacji bloku, wywołuje funkcje `m` i `p` z odpowiednimi argumentami. Wywołanie ich, skutkować może wywołaniami tych funkcji dla kolejnych bloków, co w efekcie, prowadzi do rekurencyjnego przetwarzania całego drzewa bloków.

Argumenty przekazane do funkcji `m` (`target` i `anchor`) to referencje, odpowiednio, do elementu DOM, pod którym oraz elementu DOM, przed którym (na tym samym poziomie zagnieżdżenia) ma zostać zamontowany blok. Bardziej szczegółowy opis zamieszczono w podrozdz. 9.8.6.

Istotnym jest, że `target` zawiera referencję do pierwszego przodka bloku, będącego

elementem DOM, a nie jego bezpośredniego rodzica. Rodzi to problem w momencie, gdy jakiś blok montowany jest pod innym blokiem, co w bibliotekach, opierających się na budowie aplikacji z komponentów, jest przypadkiem powszechnym.

W związku z powyższym, koniecznym było stworzenie mechanizmu pozwalającego na śledzenie kolejno przetwarzanych bloków.

W tym celu, adapter przechowuje identyfikator analizowanego (montowanego, lub aktualizowanego) bloku w zmiennej `currentBlockId`. Początkowa wartość tej zmiennej jest ustawiona na `null`.

W ciele funkcji `m` oraz `p`, wykorzystanych w „monkey patching”, adapter, przed wywołaniem oryginalnej funkcji, zapamiętuje w lokalnej zmiennej wartość `currentBlockId`, a następnie ustawia ją na identyfikator obecnie przetwarzanego bloku. Po zakończeniu wywołania oryginalnej funkcji, wartość `currentBlockId` przywracana jest do poprzedniej (list. 42).

Listing 42. Mechanizm przechowywania identyfikatora przetwarzanego bloku.

```
// ...
const originalM = block.m;
block.m = (target, anchor) => {
  // ...
  const prevBlockId = this.currentBlockId;
  this.currentBlockId = block.id;
  originalM(target, anchor);
  this.currentBlockId = prevBlockId;
};
// ...
```

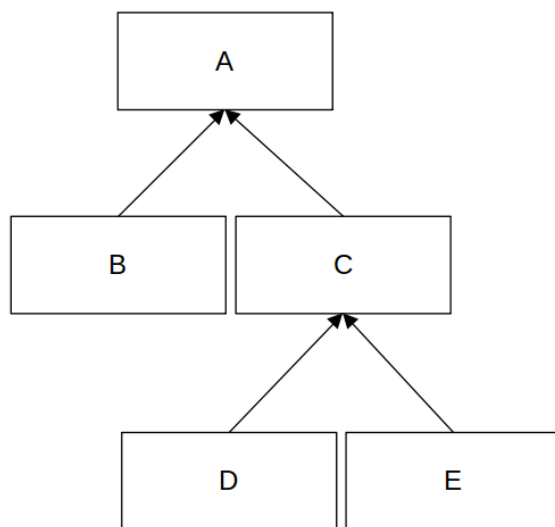
W rezultacie, kolejny blok, dla którego w wyniku wywołania oryginalnych funkcji zostanie wywołana funkcja `m` lub `p`, będzie mógł odwołać się do zmiennej `currentBlockId`, która zawiera identyfikator rodzica.

Mechanizm ten jest specyficzną implementacją stosu, na którym, w kolejnych zagnieżdżonych wywołaniach przechwyconych funkcji, odkładane są identyfikatory przetwarzanych bloków, a następnie zdejmowane w odpowiedniej kolejności. Ustawienie wartości `currentBlockId` na `null` po zakończeniu przetwarzania pierwszego bloku, jest równoznaczne z opróżnieniem stosu. Diagram obrazujący ten proces przedstawiony został w postaci śladu wywołań pseudo-kodu na rys. 9.9.

9.8.6. Wykrywanie i obsługa montowania bloków

Kod adaptera, wstrzykiwany do wspomnianej w podrozdz. 9.8.4 funkcji `m`, w pierwszej kolejności generuje identyfikator montowanego bloku.

Następnie, jeżeli blok jest komponentem, adapter sprawdza w mapie `pendingComponents`, czy pod kluczem będącym identyfikatorem bloku znajduje się nazwa komponentu. Jeżeli tak, nazwa jest zapisywana w tworzonym obiekcie `ParsedNode` (podrozdz. 9.5) i usuwana z mapy. W przeciwnym wypadku, adapter zapisuje identyfikator bloku w mapie `pendingComponents` (jak wspomniano w podrozdz. 9.8.3).



```

currentBlockId = null
A.m(){
  A.parentId = currentBlockId // null
  prevBlockId = currentBlockId // null
  currentBlockId = A.id // "A"
  // ...
  B.m(){
    B.parentId = currentBlockId // "A"
    prevBlockId = currentBlockId // "A"
    currentBlockId = B.id // "B"
    // ...
    currentBlockId = prevBlockId // "A"
  }
  C.m(){
    C.parentId = currentBlockId // "A"
    prevBlockId = currentBlockId // "A"
    currentBlockId = C.id // "C"
    // ...
    D.m(){
      D.parentId = currentBlockId // "C"
      prevBlockId = currentBlockId // "C"
      currentBlockId = D.id // "D"
      // ...
      currentBlockId = prevBlockId // "C"
    }
    E.m(){
      E.parentId = currentBlockId // "C"
      prevBlockId = currentBlockId // "C"
      currentBlockId = E.id // "E"
      // ...
      currentBlockId = prevBlockId // "C"
    }
    currentBlockId = prevBlockId // "A"
  }
  currentBlockId = prevBlockId // null
}

```

Rysunek 9.9. Uproszczony ślad wywołań funkcji `m` ze wstrzykniętym kodem adaptera dla zagnieżdżonych bloków Svelte.

W trakcie pracy nad rozwiązaniem i analizy kodu źródłowego Svelte DevTools, zauważono potrzebę zaimplementowania specjalnej obsługi montowania bloków `each` (podrozdz. 6.4.2). Wynika to z faktu, że Svelte przy każdej zmianie zawartości bloku (np. dodaniu czy usunięciu elementu z listy), zdaje się tworzyć nowy blok `each`.

Przeszukując dokumentację, nie udało się ustalić z czego wynika to zachowanie. Z powodu ograniczeń czasowych, analizę jego przyczyny postanowiono pozostawić jako aspekt do rozważenia w przyszłości (podrozdz. 12.1).

W pierwszej implementacji adaptera, wspomniany problem prowadził do błędów skutkujących wielokrotnym zamontowaniem bloków `each` dla jednego użycia bloku w komponencie. Rozwiązaniem okazało się stworzenie mapy `eaches`, która jako klucze przechowuje identyfikatory stworzone z konkatenacji `currentBlockId` i, przekazywanego przez Svelte, numeru porządkowego bloku `each` w obecnie przetwarzanym bloku. Wartościami w tej mapie jest liczba zamontowanych dla pojedynczego użycia, bloków `each` oraz prawdziwy identyfikator tego z nich, który został zamontowany jako pierwszy.

W momencie montowania bloku `each`, adapter sprawdza, czy w mapie `eaches` znajduje się już identyfikator dla danego bloku. Jeżeli tak, adapter nie zgłasza zamontowania bloku, a jedynie zwiększa w mapie `eaches` liczbę zamontowań i podmienia wartość `currentBlockId` na identyfikator pierwszego z zamontowanych bloków iteracji (dla zachowania poprawności działania mechanizmu opisanego w podrozdz. 9.8.5). W przeciwnym wypadku do mapy `eaches` dodawany jest nowy wpis.

Na samym końcu przetwarzania informacji o zamontowaniu bloku, wymagane jest ustalenie bezpośredniego rodzica. Jak wspomniano w podrozdz. 9.8.5, jako argument

`target`, przekazywana do funkcji `m` jest referencja do pierwszego przodka bloku, będącego elementem DOM. Należy jednak pamiętać, że w funkcji `m` są przetwarzane tylko bloki Svelte (nie elementy DOM), oraz, że adapter dla Svelte zachowuje identyfikator poprzednio przetwarzanego bloku w zmiennej `currentBlockId`.

Mając to na uwadze, adapter, jest w stanie ustalić bezpośredniego rodzica danego bloku (lub elementu DOM, podrozdz. 9.8.9), uwzględniając poniższe przypadki.

1. Jeżeli `currentBlockId` ma wartość `null` - przetwarzany blok jest korzeniem aplikacji. W takim przypadku, adapter zgłasza zamontowanie bloku poprzez wiadomości `MOUNT_ROOTS`.
2. Jeżeli `currentBlockId` nie jest `null` oraz `currentBlockId` różni się od identyfikatora bloku, pod którym został zamontowany pierwszy przodek będący elementem DOM (`target`), znaczy to, że między nimi musi znajdować się inny blok. Może to być tylko pierwszy przodek będący blokiem, czyli ten o identyfikatorze `currentBlockId`. W takim przypadku, adapter zgłasza zamontowanie pod owym blokiem poprzez wysłanie wiadomości `MOUNT_NODES`.
3. W pozostałych przypadkach, przetwarzany blok jest bezpośrednim dzieckiem elementu `target` i, jako taki, zostaje zgłoszony do zamontowania (również w wiadomości `MOUNT_NODES`).

W wiadomości `MOUNT_NODES` przesyłany jest także identyfikator elementu `anchor` (jeżeli istnieje), który jest elementem DOM, przed którym (na tym samym poziomie zagnieżdżenia) ma zostać zamontowany blok. Wykorzystanie tej informacji zostało opisane w podrozdz. 9.9.

Po zgłoszeniu zamontowania, adapter zapisuje dane bloku w mapie `existingNodes`.

9.8.7. Wykrywanie i obsługa aktualizacji bloków

Przy wywołaniu metody `p`, poza zapamiętywaniem identyfikatora przetwarzanego bloku (podrozdz. 9.8.5), adapter jedynie sprawdza czy panel narzędzi deweloperskich zażądał przekazywania szczegółowych informacji o danym bloku. W takim przypadku, adapter dokonuje analizy stanu i właściwości komponentu, co opisano w podrozdz. 9.8.12.

9.8.8. Wykrywanie i obsługa usuwania bloków

Wstrzykiwany do funkcji `d` kod adaptera usuwa zgłoszony blok z mapy `existingNodes` oraz zgłasza odmontowanie do izolowanego skryptu zawartości za pośrednictwem wiadomości `UNMOUNT_NODES`.

Wyjątkiem są wspomniane w podrozdz. 9.8.6 bloki `each`. Adapter zgłasza odmontowanie bloku `each` jedynie wtedy, gdy liczba zamontowań przechowywana w mapie `eaches` dla danego bloku wynosi 1. W przeciwnym wypadku, adapter zmniejsza liczbę zamontowań i nie zgłasza odmontowania.

Ponadto, w adapterze zaimplementowany został mechanizm optymalizacji ilości zgłaszanych wiadomości odmontowania. Działa on tak samo, jak ten zaimplementowany dla adaptera Reacta (podrozdz. 9.7.5), dlatego też, nie zostanie opisany ponownie.

9.8.9. Montowanie elementów DOM

Montowanie elementów DOM odbywa się w reakcji na zdarzenie `SvelteDOMInsert`. W treści zdarzenia przekazywane są referencje do elementu DOM, który został wstawiony oraz, analogicznie do montowania bloków, referencje do `target` i `anchor`.

Dla przekazanego elementu i jego dzieci (właściwość `childNodes`), adapter tworzy drzewo elementów `ParsedNode` (podrozdz. 9.5) i podobnie jak dla bloków, zgłasza zamontowanie i zapisuje informacje w mapie `existingNodes`.

Różnicą względem montowania bloków jest to, że w przypadku elementów DOM, adapter zapisuje w mapie `existingNodes` również referencję do elementu DOM (zgodnie z wymaganiem przedstawionym w podrozdz. 9.6.3).

Niestety, dla adaptera Svelte nie udało mi się zaimplementować mechanizmu podświetlania bloków nieposiadających bezpośredniej reprezentacji w drzewie DOM (np. komponentów). W przypadku adaptera dla Reacta, zaznaczany jest pierwszy potomek, posiadający taką reprezentację (list. 40). W przypadku Svelte, z uwagi na brak dostępu do drzewa elementów, takie rozwiązanie nie było możliwe. Analizę innych potencjalnych rozwiązań pozostawiono jako aspekt do rozważenia w przyszłości (podrozdz. 12.1).

9.8.10. Aktualizacja elementów DOM

Aktualizacja elementów DOM odbywa się w reakcji na zdarzenie `SvelteDOMSetData`. W treści zdarzenia przekazywana jest referencja do zmienianego elementu DOM oraz jego nowe dane.

W ramach testów nie zaobserwowano przypadku, w którym zmieniane byłyby inne dane, niż zawartość tekstowa elementu będącego węzłem tekstowym DOM. W związku z tym, adapter aktualizuje przechowywaną w mapie `existingNodes` nazwę elementu oraz wysyła do izolowanego skryptu zawartości wiadomość `UPDATE_NODES`, informując o zmianach.

Głębsza analiza innych przypadków aktualizacji elementów DOM pozostaje jako aspekt do rozważenia w przyszłości (podrozdz. 12.1).

9.8.11. Usuwanie elementów DOM

Obsługa zdarzenia `SvelteDOMRemove` polega jedynie na usunięciu elementu z mapy `existingNodes` oraz zgłoszeniu odmontowania do izolowanego skryptu zawartości, za pośrednictwem wiadomości `UNMOUNT_NODES`, z uwzględnieniem mechanizmu optymalizacji ilości zgłaszanych wiadomości, analogicznym do tego zaimplementowanego w adapterze dla Reacta (podrozdz. 9.7.5).

9.8.12. Inspekcja stanu i właściwości komponentów

Po wywołaniu wspomnianej w podrozdz. 9.6.1 metody `inspectElements`, z przekazanych identyfikatorów tworzony jest set i zapisywany w pamięci adaptera.

Następnie, dla każdego elementu odpowiadającego przekazanym identyfikatorom, z mapy `existingNodes` pobierane są podstawowe dane. Dodatkowo dla komponentów dokonywane jest parsowanie stanu i właściwości.

Wykorzystywane w tym celu są, zapisane uprzednio w mapie `componentsCaptureStates` (podrozdz. 9.8.3) wartości.

Wywołanie metody `captureState` zwraca obiekt, którego klucze zawierają nazwy właściwości i stanu komponentu, a wartości - aktualne, odpowiadające im wartości. Na podstawie tablicy `propsKeys`, zawierającej nazwy samych właściwości, adapter dzieli dane na dwie grupy - właściwości i stan.

Następnie poszczególne wartości są dehydratowane (podrozdz. 9.6.2) i przekazywane (wraz z danymi z mapy `existingNodes`) do izolowanego skryptu zawartości, w wiadomości `INSPECTED_DATA` (list. 33).

9.9. Przekazywanie informacji do panelu narzędzi deweloperskich

Początkowa implementacja izolowanego skryptu zawartości, która miała polegać na bezpośrednim przekazywaniu wiadomości, okazała się niewystarczająca, ze względu na to, że narzędzia deweloperskie przeglądarki mogą być otwierane i zamykane wielokrotnie. Przy każdej takiej ponownej inicjalizacji, główny skrypt zawartości musiałby przysyłać całą strukturę drzewa elementów poprzez mechanizm `window.postMessage`, a jak już wspomniano w podrozdz. 7.4.1, wiąże się to z dużym narzutem na wydajności systemu.

Z tego względu zastosowano mechanizm polegający na analizowaniu wiadomości przychodzących z głównego skryptu zawartości i, na ich podstawie, odbudowywaniu całego drzewa elementów w pamięci skryptu. Takie podejście pozwala na zachowanie całej struktury drzewa i przesyłania go do panelu narzędzi jedynie przez wydajniejszy mechanizm komunikacji API rozszerzeń Chrome (podrozdz. 9.4.2).

Funkcjonalności skryptu realizowane są poprzez klasę singletonową `ContentIsolated`. W momencie inicjalizacji, klasa ta zaczyna oczekiwanie na połączenie z panelem narzędzi deweloperskich (podrozdz. 9.10) tworząc obiekt `ChromeBridgeListener`. Jednocześnie inicjuje kanał połączeń ze skryptem zawartości, uruchomionym w głównym świecie wykonywania, poprzez `PostMessageBridge`, rozpoczynając nasłuchiwanie na wiadomości.

Obsługa poszczególnych wiadomości przychodzących z głównego skryptu zawartości, w zależności od typu wiadomości polega na:

- dla `LIBRARY_ATTACHED` - zapisaniu informacji o podłączonej bibliotece i przekazaniu jej do narzędzi deweloperskich i skryptu tła,
- dla `INSPECTED_DATA` - przekazaniu informacji o analizowanym elemencie do panelu narzędzi deweloperskich,
- aktualizacji przechowywanych w pamięci drzew elementów (podrozdz. 9.9.1), dla wszystkich pozostałych typów wiadomości.

Skrypt nasłuchuje również na jednorazową wiadomość `WHAT_LIBRARIES_ATTACHED`, zawierającą funkcję wywołania zwrotnego pozwalającą pozostałym skryptom, uzyskać informacje o podłączonych bibliotekach („na żądanie”).

Wiadomości `INSPECT_ELEMENT` oraz `HOVER_ELEMENT`, przychodzące z panelu narzędzi deweloperskich są jedynie przekazywane do głównego skryptu zawartości.

9.9.1. Aktualizacja przechowywanych w pamięci drzew elementów

Mechanizm odbudowywania drzew elementów, opiera się na dwóch strukturach danych:

- `nodes` - mapy, w której kluczami są identyfikatory elementów, a wartościami obiekty reprezentujące elementy (podrozdz. 9.5),
- `roots` - tablicy, w której, dla każdej podłączonej instancji biblioteki, przechowywane są informacje o bibliotekach i korzenie odpowiadających im drzew elementów.

Wiadomość `MOUNT_ROOTS` powoduje zapisanie referencji do korzeni drzew elementów w tablicy, a następnie dodanie korzeni i, rekursywnie, wszystkich ich dzieci, do mapy `nodes`.

Wiadomość `MOUNT_NODES` zawiera tablicę obiektów o strukturze przedstawionej w list. 43.

Listing 43. Struktura obiektu przekazywanego w wiadomości `MOUNT_NODES`.

```
{
  parentId: NodeId;
  anchor: {
    type: "before" | "after";
    id: NodeId | null;
  };
  node: ParsedNode;
}
```

Dla każdego elementu w tablicy, obiekt reprezentujący element dodawany jest do tablicy dzieci elementu o identyfikatorze `parentId` w mapie `nodes`. Pozycja w tablicy wyznaczana jest na podstawie wartości pola `anchor`.

Następnie, rekursywnie, wszystkie elementy i ich dzieci dodawane są do mapy `nodes`.

Pierwotna implementacja mechanizmu opierała się jedynie na tablicy `roots`, przechowującej korzenie drzew. Przychodzące wiadomości `MOUNT_NODES` zawierały tablicę `pathFromRoot`, składającą się z kolejnych identyfikatorów elementów, od korzenia do dodawanego elementu. Takie rozwiązanie wymagało każdorazowego trawersowania drzewa od korzenia, skutkując złożonością obliczeniową równą wysokości drzewa. Nowe rozwiązanie, oparte na mapie `nodes`, pozwala na dostęp do dowolnego elementu w czasie stałym. Dodatkowy narzut na wydajności, generowany jest przez potrzebę rekursywnego dodawania wszystkich dzieci elementu do mapy `nodes`. Testy jednak wykazały, że dla analizowanych bibliotek częściej zdarzają się przypadki dodawania pojedynczych elementów, niż całych poddrzew.

Wiadomość `UPDATE_NODES` zawiera identyfikatory elementów oraz nowe dane, które mają zostać zaktualizowane (np. nazwę elementu). Obsługa wiadomości polega na zaktualizowaniu reprezentacji elementu w mapie `nodes`.

Wiadomość `UNMOUNT_NODES` zawiera identyfikatory elementów do usunięcia i ich rodziców. Obsługa wiadomości polega na usunięciu elementów z tablicy dzieci ich rodziców, a także rekursywnym usunięciu wszystkich dzieci elementów z mapy `nodes`. W przypadku, gdy element do usunięcia jest korzeniem drzewa, usuwany jest on także z tablicy `roots`.

Po każdej zmianie w strukturze drzewa, jeżeli połączenie z panelem narzędzi deweloperskich zostało nawiązane, wysyłana jest wiadomość `FULL_SKELETON`, zawierająca całą strukturę drzewa. Rozwiązanie takie nie jest optymalne pod względem wydajności, jednak gwarantuje spójność między stanem drzewa w pamięci skryptu a stanem wyświetlanym w panelu narzędzi deweloperskich. Rozwój mechanizmu, polegający na niezależnym odbudowywaniu drzewa zarówno w skrypcie zawartości, jak i w panelu narzędzi deweloperskich, umożliwiającą przesyłanie jedynie aktualizacji i pełnej synchronizacji stanu „na żądanie”, pozostaje jednym z potencjalnych kierunków rozwoju narzędzia (podrozdz. 12.1).

Wiadomość `FULL_SKELETON` jest wysyłana również w momencie nawiązania połączenia z panelem narzędzi deweloperskich.

9.10. Panel narzędzi deweloperskich - interfejs użytkownika (pages/panel/)

Panel narzędzi deweloperskich, spełnia rolę interfejsu użytkownika, zapewniając warstwę prezentacji informacji o strukturze analizowanej aplikacji webowej.

Jak wspomniano w rozdz. 8, projekt został skonfigurowany w taki sposób, aby poszczególne części rozszerzenia, wymagające graficznego interfejsu, mogły być budowane z użyciem bibliotek frontendowych. Panel narzędzi deweloperskich nie jest wyjątkiem i został stworzony przy użyciu biblioteki React.

9.10.1. Globalne konteksty dla panelu narzędzi deweloperskich

Dla aplikacji, uruchomionej w panelu narzędzi deweloperskich, zdefiniowane zostały poniższe, globalne konteksty (podrozdz. 5.9.3) zapewniające dostęp do danych i funkcji w całej aplikacji:

- `ChromeBridgeContext` - kontekst zapewniający dostęp do obiektu `ChromeBridge` (podrozdz. 9.4.2), który umożliwia komunikację z izolowanym skrypcie zawartości. Przy inicjalizacji, kontekst ten, nawiązuje połączenie oraz zapewnia odświeżenie połączenia w przypadku nawigacji na nową stronę internetową (skutkujące ponownymi inicjalizacjami skryptów zawartości);
- `FilterContext` - kontekst zapewniający dostęp do funkcji pozwalającej na określenie, które z elementów powinny być wyświetlane w panelu narzędzi deweloperskich oraz na aktualizację stanu filtrów. Implementacja i sposób działania opisano w podrozdz. 9.10.4;
- `SelectedNodeContext` - kontekst zapewniający dostęp do informacji o aktualnie zaznaczonym elemencie oraz funkcji pozwalającej na zmianę zaznaczenia.
- `InspectedDataContext` - kontekst nasłuchujący na wiadomości `INSPECTED_DATA` i zapewniający dostęp do danych odbieranych w tych wiadomościach (list. 33);
- `ExpandContext` - kontekst pozwalający na rozwijanie i zwijanie całego drzewa wyświetlonych elementów. Mechanizm ten omówiono w podrozdz. 9.10.5.

Architektonicznie prawdopodobnie lepszą decyzją byłoby udostępnienie kontekstów niżej w drzewie komponentów, tam, gdzie są one faktycznie wykorzystywane. Jednak z uwagi na stosunkowo niewielkie rozmiary aplikacji, zastosowano konteksty globalne.

Znaczący wzrost złożoności aplikacji, w przyszłości, może wymagać refaktoryzacji w tym zakresie.

9.10.2. Struktura interfejsu użytkownika



Rysunek 9.10. Struktura interfejsu użytkownika panelu narzędzi deweloperskich.

Interfejs użytkownika składa się z trzech głównych części (zaznaczonych na rys. 9.10 numerami 1-3):

1. **Nagłówek** - zawierający nazwę aplikacji, przyciski pozwalające na rozwijanie i zwijanie całego drzewa elementów, suwak pozwalający na zmianę wielkości wcięcia elementów drzewie, oraz przycisk otwierający menu narzędzia (podrozdz. 9.10.3).
2. **Drzewo elementów** - widok drzewa elementów, pokazujący strukturę elementów na analizowanej stronie (podrozdz. 9.10.5).
3. **Szczegóły elementu** - widok szczegółów wybranego elementu, zawierający informacje o jego nazwie, typie oraz wszystkie inne informacje, które zostały przekazane z izolowanego skryptu zawartości (podrozdz. 9.10.6).

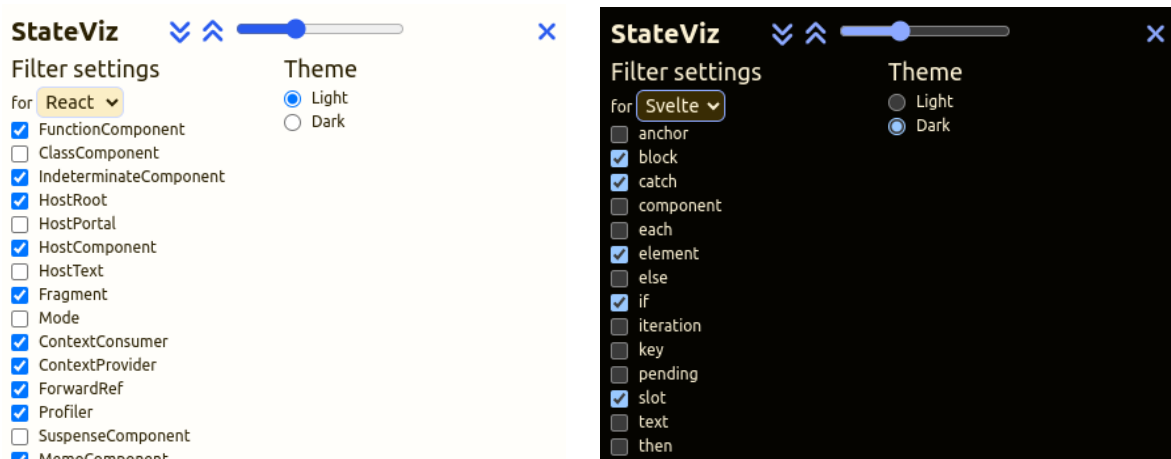
Widok szczegółów elementu wyświetlany jest tylko wtedy, gdy użytkownik zaznaczy jakiś element w drzewie elementów.

W momencie wyświetlenia obydwu widoków, użytkownik może sterować szerokością każdego z nich, poprzez przeciąganie myszą separatora znajdującego się między nimi. Pozwala to na dostosowanie interfejsu do indywidualnych potrzeb użytkownika.

9.10.3. Menu panelu narzędzi deweloperskich

Kliknięcie w ikonę menu znajdującą się z prawej strony nagłówka, rozwija w nim ustawienia panelu narzędzi deweloperskich.

Menu pozwala na wyświetlenie i zmianę aktualnych ustawień filtrowania elementów dla każdej z obsługiwanych bibliotek oraz na zmianę motywu rozszerzenia (jasny/ciemny).



(a) Fragment ustawień filtrowania dla biblioteki React w motywie jasnym.

(b) Ustawienia filtrowania dla biblioteki Svelte w motywie ciemnym.

Rysunek 9.11. Widok ustawień panelu narzędzi deweloperskich.

Jak pokazano na rys. 9.11, użytkownik wykorzystując listę rozwijaną, może wybrać dla której z bibliotek chce zmienić ustawienia filtrowania, a następnie włączyć lub wyłączyć filtrowanie konkretnych typów elementów.

Komponent, odpowiadający za wyświetlanie ustawień filtrowania, wykorzystuje kontekst `FilterContext` (podrozdz. 9.10.1) do odczytania i aktualizacji stanu filtrów.

Zarówno dla ustawień filtrowania, jak i motywu aplikacji, dane są zachowywane w pamięci lokalnej przeglądarki (podrozdz. 9.10.7).

9.10.4. Filtrowanie wyświetlanych elementów

Mechanizm filtrowania elementów w panelu narzędzi deweloperskich jest jedynym, który zawiera logikę specyficzną dla każdej z bibliotek. Wynika to z faktu, że każda z nich, z oczywistych względów, operuje na innym zbiorze typów elementów oraz, ewentualnie, na innych właściwościach.

Wspomniano o tym na samym początku tego podrozdziału, ponieważ uznano, iż jest to jedna z największych wad obecnego rozwiązania. W przyszłości, warto byłoby zastanowić się nad możliwością przeniesienia informacji o typach elementów i używanych w filtrowaniu właściwościach, do adapterów, całkowicie oddzielając logikę specyficzną dla bibliotek od reszty rozwiązania. Panel narzędzi deweloperskich mógłby wtedy dynamicznie tworzyć

interfejs użytkownika, na podstawie przekazanych informacji, a dodanie obsługi kolejnej biblioteki do rozwiązania, wymagałoby jedynie stworzenia odpowiedniego adaptera (podrozdz. 12.1).

W obecnej implementacji `FilterContext` określa nazwy właściwości obiektu `ParsedNode` (podrozdz. 9.5) używanych do filtrowania oraz domyślne ustawienia filtrów dla każdej z bibliotek.

Kontekst udostępnia globalnie dwie funkcje, których sygnatury przedstawione są na list. 44.

Listing 44. Sygnatury funkcji udostępnianych przez kontekst filtrowania.

```
function updateFilter<T extends Library>(
  library: T,
  key: SettingIdentifier<T>,
  value: boolean
): void;

function shouldDisplayNode(
  nodeAndLibrary: FilterDifferentiator
): boolean;
```

Funkcja `updateFilter` pozwala na zmianę wartości filtra dla danej wartości właściwości, używanej do filtrowania dla konkretnej biblioteki.

Funkcja `shouldDisplayNode` wymaga przekazania jako argument obiektu zawierającego pole określające z jakiej biblioteki pochodzi dany element oraz wartości właściwości, która dla tej biblioteki używana jest do filtrowania elementów. Funkcja, na podstawie aktualnych ustawień, zwraca wartość logiczną, określającą czy dany element powinien być wyświetlany w panelu narzędzi deweloperskich czy nie.

Implementacje typów `SettingIdentifier` i `FilterDifferentiator` nie są istotne dla zrozumienia działania mechanizmu. Warto jedynie wiedzieć, że zostały stworzone po to, aby zapewnić ścisłe typowanie dla udostępnianych funkcji (np. nie można podać typu właściwości, który nie istnieje dla danej biblioteki, bądź nie jest używany do filtrowania).

Ustawienia filtrowania zapisywane są w pamięci lokalnej przeglądarki (podrozdz. 9.10.7). Przy inicjalizacji kontekstu, dokonuje on próby odczytania zapisanych ustawień. W przypadku ich braku, ustawienia są inicjowane wartościami domyślnymi.

9.10.5. Widok drzewa elementów

Korzeniem widoku drzewa elementów jest komponent `Roots`. Przy inicjalizacji, wykorzystuje on kontekst `ChromeBridgeContext` w celu rozpoczęcia nasłuchiwanie na wiadomość `FULL_SKELETON`, przekazywaną z izolowanego skryptu zawartości (podrozdz. 9.9).

Wiadomość ta zawiera tablicę obiektów reprezentujących korzenie drzew elementów oraz nazwy każdej z podłączonych bibliotek.

Po otrzymaniu wiadomości, komponent na podstawie ustawień filtrowania (podrozdz. 9.10.4), tworzy drzewa składające się tylko z elementów spełniających kryteria filtrowania.

Następnie, każdy z korzeni nowo utworzonych drzew, przekazywany jest do komponentu `Row`. Warto zauważyć, że w wyniku filtrowania, może powstać większa liczba korzeni, niż w otrzymanej wiadomości (np. poprzez odfiltrowanie korzenia zawierającego kilkoro dzieci).

Komponent `Row` przyjmuje jako właściwości obiekt `ParsedNode`, reprezentujący konkretny element oraz głębokość w drzewie, w którym się znajduje. Na podstawie tych danych, komponent wyświetla wiersz drzewa zawierający nazwę elementu.

Na podstawie głębokości w drzewie oraz ustalonej suwakiem, wspomnianym w podrozdz. 9.10.2, wartości wcięcia, dla komponentu ustalany jest odpowiedni styl, pozwalający na łatwiejsze odczytanie struktury drzewa.

Komponent dla wszystkich dzieci przekazanego elementu, rekursywnie tworzy kolejne komponenty `Row`, inkrementując przy tym przekazywaną głębokość. Tworzy w ten sposób całe drzewo elementów.

Jeżeli element posiada dzieci, przy nazwie komponentu wyświetlany jest przycisk, pozwalający na rozwinięcie i zwinięcie danego poziomu drzewa.

Kliknięcia w przyciski zwiłania i rozwijania, wspomniane w podrozdz. 9.10.2, powodują zmianę stanu kontekstu `ExpandContext` (podrozdz. 9.10.1). Kontekst ten udostępnia obiekt, który zawiera jedno pole z wartością logiczną, określającą czy elementy drzewa powinny zostać rozwinięte (wartość prawdziwa) czy zwinięte (wartość fałszywa). Wartość kontekstu zachowywana jest w obiekcie zamiast wartości prymitywnej po to, by nawet przy braku zmiany samej wartości, uruchamiać ponownie wszystkie hooki zależne od tego kontekstu.

Komponent `Row` ignoruje pierwszą wartość otrzymaną z kontekstu `ExpandContext`, aby przy inicjalnym renderowaniu nie reagować na wcześniejsze kliknięcia przycisków. Przy każdej kolejnej zmianie wartości, komponent w zależności od wartości, rozwija lub zwiłaja swoje dzieci.

Dodatkowo, komponent reaguje na najechania myszą i kliknięcia w wiersz. Pierwsze ze zdarzeń powoduje wysłanie wiadomości `HOVER_ELEMENT` do izolowanego skryptu zawartości z identyfikatorem elementu, nad którym znajduje się kursor. Adapter obsługuje otrzymaną wiadomość, podświetlając odpowiedni element na stronie internetowej (podrozdz. 9.6.3). Drugie zdarzenie powoduje aktualizację zaznaczonego elementu w kontekście `SelectedNodeContext` (podrozdz. 9.10.1).

9.10.6. Widok szczegółów elementu

Widok szczegółów elementu subskrybuje konteksty:

- `SelectedNodeContext`,
- `InspectedDataContext`.

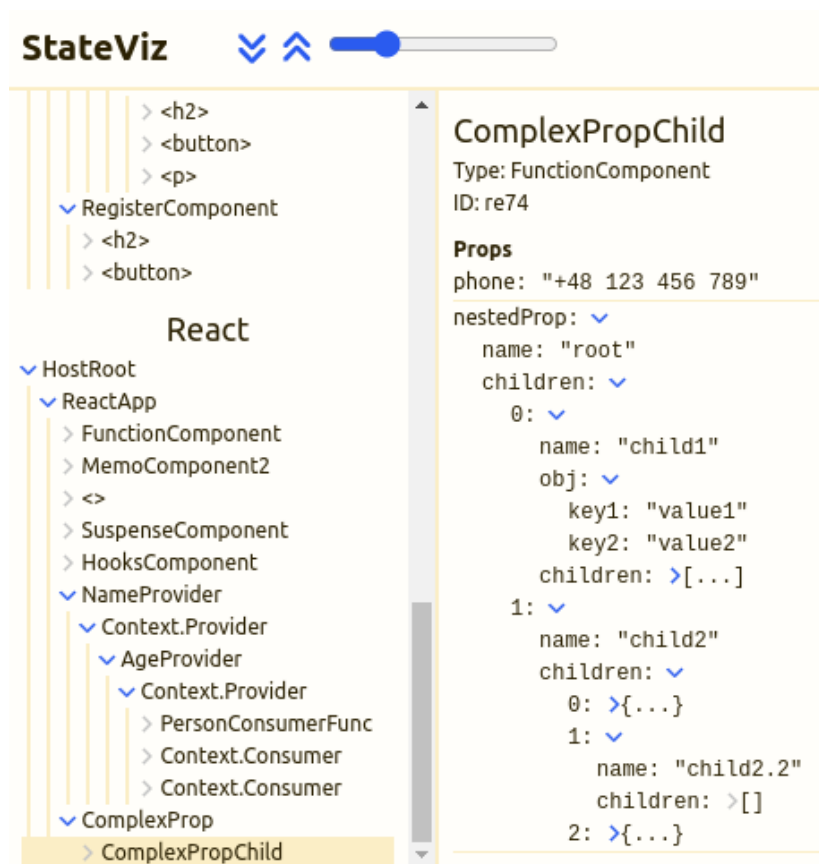
W momencie wybrania lub zmiany zaznaczenia elementu, poprzez `ChromeBridge`, wysyłana jest wiadomość `INSPECT_ELEMENT`, zawierającą identyfikator zaznaczonego elementu.

Po otrzymaniu danych o zaznaczonym elemencie, komponent wyświetla jego podstawowe informacje, takie jak nazwa i typ. Natomiast wszystkie dehydratowane dane znajdujące się w obiektach `InspectedDataContext` (list. 33) przekazywane są do komponentu `NodeInspectData`.

Komponent ten, dla każdej grupy danych, wyświetla jej nagłówek, a następnie listę elementów znajdujących się w grupie.

Za wyświetlanie wartości, będących wynikami działania funkcji `dehydrate` (podrozdz. 9.6.2), odpowiedzialny jest komponent `NodeStateValue`.

Dla każdego z typów danych, przedstawionych w list. 35, komponent używa odpowiednio przygotowanej reprezentacji. Dla wartości mogących zawierać inne, zagnieżdżone wartości (np. `ARRAY` czy `OBJECT`), komponent rekursywnie tworzy kolejne komponenty `NodeStateValue` i na każdym poziomie rekurencji umieszcza przycisk, pozwalający na rozwijanie oraz zwijanie drzewa obiektów.



Rysunek 9.12. Widok szczegółów komponentu ze złożoną wartością właściwości.

Na rys. 9.12 przedstawiono przykład widoku szczegółów komponentu ze złożoną wartością właściwości. Warto zwrócić uwagę na sposób wyświetlania:

- wartości prostej - tekstowa wartość właściwości „phone”;
- rozwiniętego obiektu - wartość właściwości „nestedProp” oraz jego właściwości:
 - nierozwiniętej, niepustej tablicy - wartość „children” węzła o nazwie „child1”. Podświetlona strzałka w prawo oznacza, że tablica nie jest, ale może zostać rozwinięta. Dodatkowo „...” oznacza, że tablica zawiera elementy. Analogicznie wyświetlane są obiekty (element pod indeksem „2”, właściwości „children” węzła o nazwie „child2”);

- rozwiniętej tablicy - wartość „children” węzła o nazwie „root”. Strzałka w dół oznacza, że tablica jest rozwinięta. Dodatkowo, każdy element tablicy jest wyświetlany w osobnym wierszu, z etykietą odpowiadającą indeksowi elementu;
- pustej tablicy - wartość „children” węzła o nazwie „child2.2”. Niepodświetlona strzałka w prawo oznacza, że lista nie jest rozwinięta oraz brak możliwości jej rozwinięcia.

9.10.7. Persystencja ustawień użytkownika

Wszystkie ustawienia rozszerzenia zapisywane są w pamięci lokalnej, przy wykorzystaniu API rozszerzeń chrome (podrozdz. 4.3.3).

Dla łatwiejszego zarządzania zapisywanymi danymi, posłużyła abstrakcja nad API, przygotowana przez twórców „chrome-extension-boilerplate-react-vite” [75].

Na abstrakcję składają się:

- `createStorage` - funkcja przyjmująca jako argumenty:
 - klucz, pod którym mają być zapisywane dane,
 - wartość domyślną, zwracaną w przypadku braku zapisanych danych,
 - typ magazynu, określający w jaki sposób dane mają być zapisywane (np. `local` dla pamięci lokalnej, podrozdz. 4.3.3).

Zwraca ona obiekt, reprezentujący stworzony magazyn. Obiekt ten udostępnia metody pozwalające na odczytanie, zapisanie i usunięcie danych.

- `useStorage` - niestandardowy hook (podrozdz. 5.9.10), wymagający przekazania obiektu magazynu jako argumentu. Hook zwraca aktualną, reaktywną wartość zapisaną w magazynie.

Przykładowa definicja lokalnego magazynu, przechowującego ustawienia motywu aplikacji, przedstawiona została na list. 45.

Listing 45. Przykładowa definicja magazynu z ustawieniami motywu aplikacji.

```
const themeStorage = createStorage<"light" | "dark">(
  "StateViz-theme",
  "light",
  {
    storageType: StorageType.Local,
  }
);
```

9.11. Prezentacja funkcjonalności

Prezentacja funkcjonalności rozszerzenia, w formie filmu zamieszczonego w serwisie YouTube, dostępna jest w zał. 1.

10. Testy przygotowanego rozwiązania

Dla projektu skonfigurowane zostało środowisko testowe przy użyciu narzędzia Vitest [82] oraz biblioteki „react-testing-library” [88].

10.1. Testy jednostkowe i integracyjne

Dla kluczowych komponentów aplikacji oraz funkcji pomocniczych, stworzonych zostało 249 testów jednostkowych i integracyjnych, pokrywających 91.83% kodu odpowiadającego za warstwę logiki aplikacji.

Odnosnik do wszystkich scenariuszy testowych oraz aktualnych wyników testów znajduje się w zał. 3.

Testy pokrywają między innymi:

- logikę klasy abstrakcyjnej Adapter (podrozdz. 9.6),
- logikę adaptera dla Reacta (podrozdz. 9.7),
- logikę adaptera dla Svelte (podrozdz. 9.8),
- logikę abstrakcji nad połączeniami między częściami rozszerzenia dla:
 - wiadomości jednorazowych (podrozdz. 9.4.1),
 - połączeń długotrwałych (podrozdz. 9.4.2),
 - wiadomości postMessage (podrozdz. 9.4.3),
- logikę dehydratacji danych (podrozdz. 9.6.2),
- logikę funkcji pomocniczych wykorzystywanych przez adaptery.

Niestety, ze względu na ograniczenia czasowe, nie udało się stworzyć automatycznych testów jednostkowych dla warstwy prezentacji aplikacji, jednakże przygotowane testy manualne (podrozdz. 10.2), weryfikujące poprawność działania panelu narzędzi deweloperskich, pokazały, że aplikacja działa zgodnie z oczekiwaniami.

Stworzenie testów jednostkowych dla warstwy prezentacji, pozostaje jednym z kierunków dalszych prac nad projektem (podrozdz. 12.1).

10.1.1. Automatyzacja CI/CD

W celu automatyzacji procesu testowania, skonfigurowany został proces CI/CD w usłudze GitHub Actions [89].

Każdy stworzony „pull request”, przed połączeniem z główną gałęzią, musi pozytywnie przejść wszystkie przygotowane testy.

Definicja procesu znajduje się w pliku `.github/workflows/extension-tests.yml` w repozytorium projektu i składa się z siedmiu etapów:

1. Stworzenie kontenera z systemem operacyjnym Ubuntu.
2. Pobranie kodu źródłowego projektu.
3. Skonfigurowanie środowiska Node.js.
4. Zainstalowanie zależności projektu.
5. Uruchomienie testów jednostkowych i integracyjnych z włączoną opcją raportowania pokrycia kodu.

6. Stworzenie odznaki „.svg” przy użyciu „vitest-badge-action” [90], na podstawie raportu pokrycia kodu.
7. Dodanie „commita” z odznaką i wynikami testów przy użyciu „git-auto-commit-action” [91].

10.1.2. Konieczność stworzenia poprawek dla `vitest-chrome`

W celu przetestowania poprawności działania funkcji korzystających z API rozszerzeń Chrome, użyta została biblioteka „vitest-chrome” [92], pozwalająca na tworzenie atrap (ang. „mocks”) obiektów globalnych, takich jak `chrome.runtime` czy `chrome.devtools`.

Biblioteka umożliwia definiowanie obiektów globalnych, które w środowisku testowym pozwalają, na przykład, na przekazywanie wiadomości między częściami rozszerzenia poprzez `chrome.runtime` bez konieczności uruchamiania całego środowiska przeglądarki.

W definicji biblioteki `package.json`, brakowało jednak poprawnej definicji eksportowanych plików, co skutkowało użyciem wersji skompilowanej dla modułów CommonJS w projekcie wykorzystującym moduły ES [93].

Problem ten udało się naprawić lokalnie posiłkując się biblioteką „patch-package” [94]. Pozwala ona na lokalne modyfikowanie zależności, a następnie zapisanie zmian w postaci łatki, która jest automatycznie uwzględniana po procesie instalacji używanych bibliotek.

Obecność problemu oraz propozycję rozwiązania zgłoszono autorowi biblioteki [95], jednak na dzień 14.05.2024 r. propozycja nie została jeszcze zaakceptowana.

10.2. Testy manualne

Aby w trakcie pracy nad projektem weryfikować poprawność działania rozszerzenia, stworzone zostały aplikacje testowe znajdujące się w katalogu `test-apps`.

Aplikacje te pozwalają na przetestowanie działania rozszerzenia pod kątem różnych scenariuszy, takich jak:

- poprawna obsługa większości typów elementów Fiber w Reaccie i różnorodnych ich interakcji,
- prawidłowe parsowanie i wyświetlanie informacji o wykorzystywanych w Reaccie:
 - hookach (w komponentach funkcyjnych),
 - kontekstach,
 - właściwościach,
 - stanach (w komponentach klasowych),
 - komponentach wyższego rzędu,
 - „forward refach” [96], itp.,
- właściwa obsługa wszystkich bloków w Svelte,
- poprawne parsowanie i wyświetlanie informacji o stanie i właściwościach komponentów w Svelte,
- prawidłowa obsługa dodawania, usuwania i aktualizacji elementów dla obu bibliotek,
- poprawne wyświetlanie struktury i informacji o zmianach w aplikacjach, które wykorzystują dwie instancje bibliotek jednocześnie (React - React lub React - Svelte),

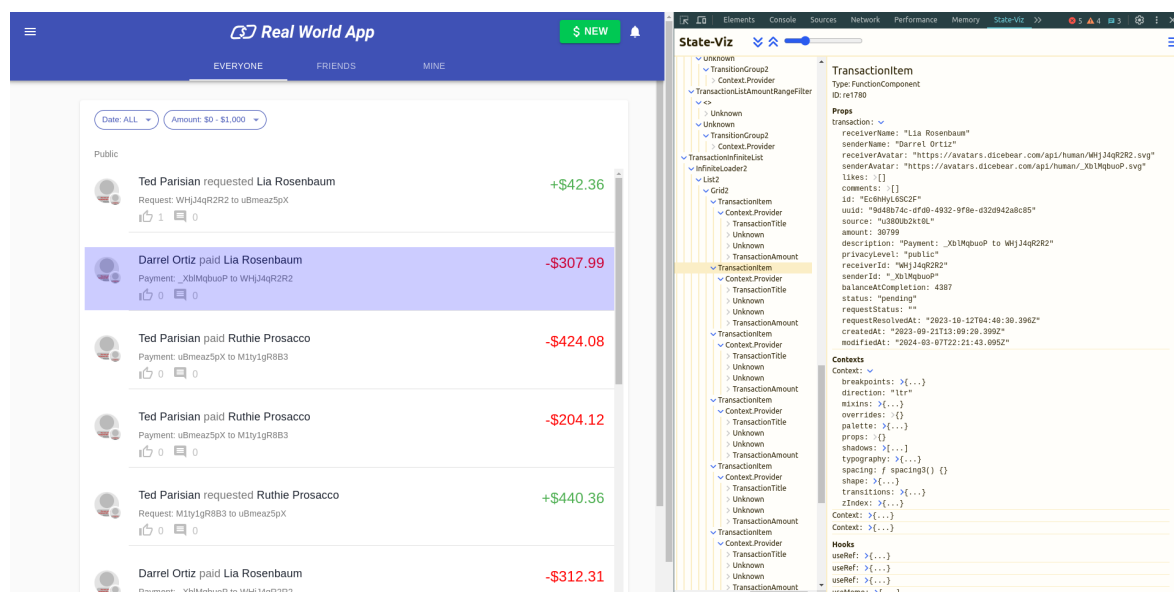
10. Testy przygotowanego rozwiązania

- właściwe odświeżanie widoku, ikony i okna modalnego rozszerzenia w przypadku nawigacji między różnymi stronami (zarówno używającymi obsługiwane biblioteki, jak i nie),
- odpowiednie działanie rozszerzenia przy zamykaniu i otwieraniu narzędzi deweloperskich przeglądarki,
- poprawne działanie interfejsu użytkownika panelu narzędzi deweloperskich,
- odpowiednie podświetlanie elementów na stronie, odpowiadających wybranym, w panelu narzędzi deweloperskich, komponentom,
- prawidłowe filtrowanie elementów w panelu narzędzi deweloperskich,
- poprawna reakcja na zmianę motywu rozszerzenia,
- właściwe zapamiętywanie ustawień rozszerzenia między odświeżeniami strony czy sesjami przeglądarki.

Ponadto, stworzone rozszerzenie zostało przetestowane na różnych dużych i złożonych aplikacjach Reacta:

- „jira_clone” [97] [98] (w trybie produkcyjnym),
- „cypress-realworld-app” [99] (w trybie deweloperskim),
- „simorgh” [100] [101] (w trybie produkcyjnym),
- „QRnamic” [102] (w trybie deweloperskim i produkcyjnym).

W trakcie testów, na wyżej wymienionych aplikacjach nie napotkano żadnych problemów a rozszerzenie działało zgodnie z oczekiwaniami.



Rysunek 10.1. Zrzut ekranu, przedstawiający działanie rozszerzenia w złożonej aplikacji Reacta - „cypress-realworld-app”.

Jak wspomniano w podrozdz. 9.8, ze względu na implementację interfejsów Svelte wykorzystywanych przez rozszerzenie do łączenia się z biblioteką, nie jest możliwe przetestowanie rozwiązania na produkcyjnych aplikacjach Svelte.

Z tego powodu, działanie rozszerzenia przetestowano na wersji deweloperskiej aplikacji „realworld” [103], przygotowanej przez twórców biblioteki Svelte.

Ponieważ aplikacja ta wykorzystuje metaframework SvelteKit [104], pozwoliła na wykrycie oraz rozwiązanie problemów, które nie zostały zauważone podczas testów na aplikacjach testowych.

Okazało się, że oczekiwanie na zdarzenie `DOMContentLoaded` (podrozdz. 9.8.1) w przypadku dużych aplikacji jest zbyt krótkie i Svelte nie zdąża dodać wymaganych danych do globalnego obiektu `window`. W celu rozwiązania problemu, po zdarzeniu `DOMContentLoaded` dodano dodatkowe, sekundowe opóźnienie, które pozwoliło na poprawne wykrycie aplikacji Svelte. Rozwiązanie to potraktowano jako tymczasowe, a problem ten pozostał jednym z kierunków dalszych prac nad projektem (podrozdz. 12.1).

Problemem okazało się również, wielokrotne zgłaszanie dodania tych samych elementów DOM do drzewa aplikacji, skutkujące powieleniem elementów wyświetlanych w panelu narzędzi deweloperskich oraz pojawianiem się błędów w przypadku prób ich filtrowania i podświetlania. Rozwiązanie problemu było trywialne, bowiem sprowadzało się do dodatkowego sprawdzenia obecności zgłaszanych elementów w mapie `existingNodes` w funkcji obsługującej zdarzenie `SvelteDOMInsert` (podrozdz. 9.8.9), i wysyłaniu wiadomości o zamontowaniu wyłącznie dla nowych elementów.

11. Porównanie z istniejącymi rozwiązaniami

Porównania między rozwiązaniami zostały przeprowadzone przy użyciu aplikacji wspomnianych w podrozdz. 10.1, zarówno tych przygotowanych przez autora, jak i tych dostępnych w internecie.

Kryteria porównania obejmują:

- wydajność,
- funkcjonalności,
- stabilność,
- łatwość użycia,
- możliwość konfiguracji.

11.1. Porównanie z narzędziami dla biblioteki React

11.1.1. Realize

Jak wspomniano w podrozdz. 2.1.2, Realize pozwala na wizualizację drzewiastej struktury komponentów aplikacji.

Niestety, implementacja opierająca się na parsowaniu i przekazywaniu całej struktury aplikacji przy każdej zmianie stanu oraz konieczność równoległego uruchomienia React Developer Tools, mocno wpływają na wydajność narzędzia.

Nawet w przypadku prostej aplikacji znajdującej się w katalogu `test-apps/react/tree`, szybkie zmiany struktury drzewa (np. poprzez wielokrotne dodawanie i usuwanie komponentów) prowadziły do błędów i niewyświetlania się wszystkich utworzonych węzłów.

Mimo, że narzędzie prezentuje informacje o stanie komponentów, dane te nie są automatycznie aktualizowane. W przypadku każdej zmiany stanu, konieczne jest ponowne zaznaczenie danego komponentu i (w przypadku obiektów bądź tablic) ręczne rozwinięcie ich widoku w panelu narzędziowym. Jest to rozwiązanie niepraktyczne i w przypadku szybko zmieniających się stanów, uniemożliwia śledzenie zmian.

Pozostałe wady względem autorskiego rozwiązania obejmują:

- brak możliwości wyświetlania innych elementów niż komponenty,
- brak możliwości wyświetlania wartości wykorzystywanych w komponentach hooków,
- brak możliwości filtrowania wyświetlanych elementów,
- brak możliwości identyfikowania komponentów wyższego rzędu i „forwardRefów”,
- brak podświetlania elementów na stronie odpowiadających wybranym, w panelu narzędzi deweloperskich, komponentom,
- brak możliwości zwijania i rozwijania widoku drzewa elementów,
- konieczność manualnej zmiany stanu aplikacji w celu wyświetlenia struktury drzewa po każdym otwarciu narzędzi deweloperskich.

Próby użycia narzędzia w bardziej złożonych aplikacjach, takich jak „jira-clone”, czy „simorgh”, nie przyniosły pozytywnych rezultatów. W obydwu przypadkach, żadne z komponentów nie zostały wyświetlone w panelu narzędziowym, a w przypadku „cypress

-real-world-app”, uruchomienie narzędzia powodowało zawieszenie się całej strony internetowej.

11.1.2. React Sight

Niestety, w momencie tworzenia tego porównania, rozszerzenie React Sight (podrozdz. 2.1.3) nie było już dostępne w sklepie rozszerzeń Chrome. Kod źródłowy narzędzia, co prawda, można znaleźć w serwisie GitHub [18], jednakże nie jest on rozwijany od 2020 roku i nie udało się zbudować działającej wersji rozszerzenia. Z tego powodu React Sight nie został uwzględniony w porównaniu.

11.1.3. React Developer Tools

React Developer Tools, jako oficjalne i cieszące się ogromną popularnością narzędzie dla biblioteki React, oferuje najwięcej funkcjonalności ze wszystkich porównywanych rozwiązań.

Poza osobnym panelem, pozwalającym na profilowanie i badanie wydajności aplikacji, RDT względem autorskiego rozwiązania oferuje dodatkowo możliwości:

- bezpośredniej edycji stanu i właściwości komponentów z poziomu narzędzi deweloperskich,
- wyszukiwania poszczególnych elementów po ich nazwie lub używając wyrażeń regularnych,
- wyszukiwania elementów w panelu narzędzi deweloperskich, odpowiadających elementom strony, które zostały zaznaczone kursorem,
- przejścia do definicji komponentu w kodzie źródłowym aplikacji,
- możliwość wyświetlenia wybranych elementów drzewa DOM we wbudowanych narzędziach deweloperskich przeglądarki,
- bardziej przejrzystego sposobu oznaczania komponentów wyższego rzędu i „forwardRefów” w panelu narzędziowym,
- wyświetlania nazw zmiennych, przechowywanych w hookach i kontekstach (podrozdz. 7.10),
- podświetlania na stronie elementów, które zostały ponownie wyrenderowane w wyniku zmiany stanu aplikacji.

Wszystkie te funkcjonalności mogą posłużyć jako inspiracja do dalszego rozwoju autorskiego narzędzia w przyszłości (podrozdz. 12.1).

Mimo swoich zalet, RDT ma też swoje wady. Najbardziej oczywistą, względem autorskiego rozwiązania, jest kompatybilność jedynie z aplikacjami Reacta. Twórcy, dostosowali i zoptymalizowali narzędzie do jak najwydajniejszej pracy z tą biblioteką, co sprawia, że rozbudowanie narzędzia o wsparcie dla innych bibliotek, takich jak Svelte, Vue.js czy Angular, byłoby bardzo czasochłonne i wymagałoby gruntownych zmian w całej (bardzo złożonej) architekturze narzędzia.

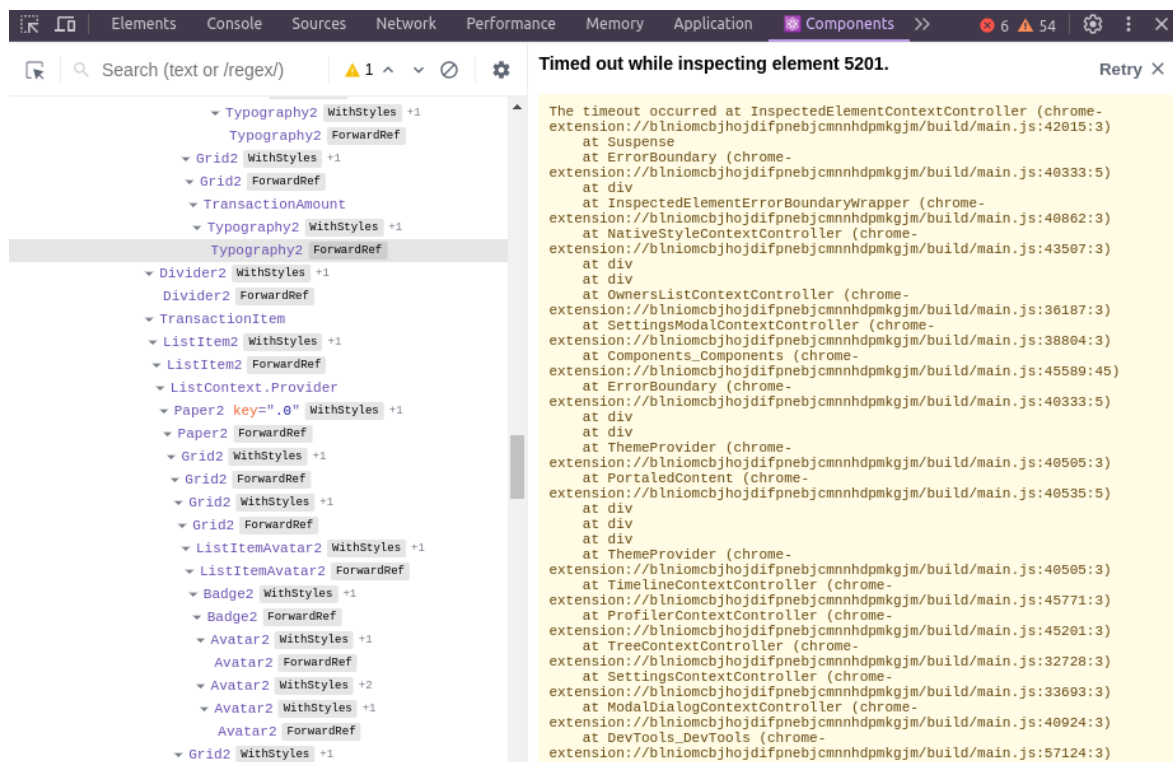
Dodatkowo, w autorskim rozwiązaniu, informacje o zmianach stanu są przesyłane natychmiastowo, co pozwala na śledzenie zmian w czasie rzeczywistym. W RDT natomiast, panel narzędzi periodycznie odpytuje skrypt zawartości o zmiany stanu komponentów

11. Porównanie z istniejącymi rozwiązaniami

(podrozdz. 7.8), co skutkuje około jednosekundowymi opóźnieniami w wyświetlaniu informacji, a w przypadku częstych zmian - prowadzi do ich utraty.

Zastosowanie w RDT podejścia „pull” zamiast „push”, zgodnie z dokumentacją narzędzia [17] było techniką optymalizacyjną, jednak, mimo niezastosowania jej w autorskim rozwiązaniu, nawet w przypadku bardzo często zmieniających się stanów, nie zaobserwowano znaczącego spadku wydajności narzędzia StateViz.

Co więcej, zaistniało podejrzenie, że różnica w podejściach prowadzi do częstych problemów ze stabilnością działania widoku szczegółów komponentów w RDT. W czasie testów wielokrotnie napotkano komunikat o przekroczeniu limitu czasu na odpowiedź (rys. 11.1).



Rysunek 11.1. Komunikat o przekroczeniu limitu czasu na odpowiedź w React Developer Tools.

Zauważono, że problem pojawia się dla różnych wersji Reacta, o czym świadczą wątki zamieszczone przez użytkowników na forach dyskusyjnych w Internecie [105] [106] [107]. Sugerowanymi rozwiązaniami były zazwyczaj: odświeżenie strony, zamknięcie i ponowne otwarcie narzędzi deweloperskich, czy nawet restart przeglądarki i reinstalacja rozszerzenia.

Podobne problemy nie pojawiały się w przypadku autorskiego rozwiązania, a rozszerzenie StateViz działało bez zarzutu, nawet w przypadku aplikacji o złożonych strukturach (podrozdz. 10.2).

11.2. Porównanie z narzędziami dla biblioteki Svelte

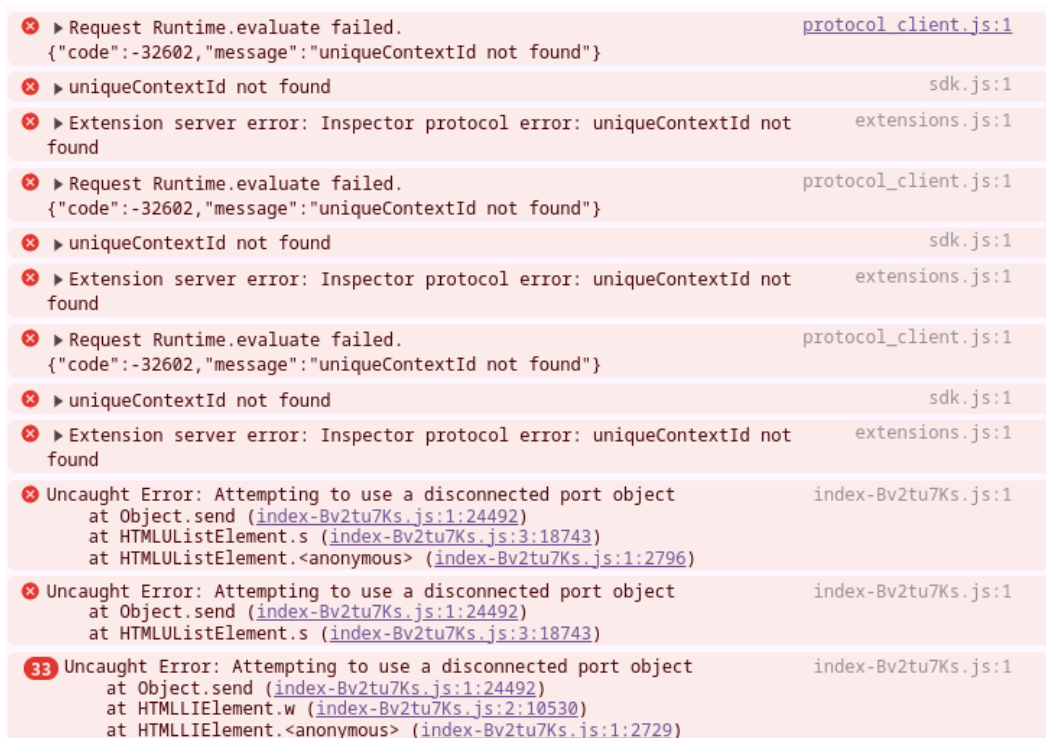
11.2.1. Svelte DevTools

Svelte DevTools, jest oficjalnym narzędziem dla biblioteki Svelte, jednak nie oferuje ono takiej ilości funkcjonalności, jak RDT dla Reacta.

W stosunku do autorskiego rozwiązania, Svelte DevTools oferuje dodatkowo możliwości:

- bezpośredniej edycji stanu i właściwości komponentów z poziomu narzędzi deweloperskich,
- wyszukiwania poszczególnych elementów po ich nazwie,
- wyszukiwania elementów w panelu narzędzi deweloperskich, odpowiadających elementom strony, które zostały zaznaczone kursorem (teoretycznie, ponieważ w czasie testów funkcjonalność ta nie działała),
- wyświetlania nasłuchiwań na zdarzenia zarejestrowanych na poszczególnych elementach,
- wyświetlania atrybutów elementów.

Zasadniczą wadą narzędzia, względem autorskiego rozwiązania, jest jednak stabilność jego działania. Rozszerzenie, szczególnie w czasie pracy ze złożonymi aplikacjami, bardzo często się zawiesza i wymaga ponownego uruchomienia. W czasie testów na aplikacji „realworld”, średnio co kilka minut, rozszerzenie przestawało odpowiadać, a w konsoli deweloperskiej pojawiały się błędy przedstawione na rys. 11.2.



Rysunek 11.2. Błędy w konsoli deweloperskiej narzędzia Svelte DevTools.

Ponadto, Svelte DevTools nie łączy się z aplikacją automatycznie i po otwarciu panelu narzędzi deweloperskich, konieczne jest odświeżenie strony (rys. 11.3).

Co więcej, zdarza się, że przedstawiony na rys. 11.3, przycisk odświeżenia nie działa, a więc konieczne jest zamknięcie narzędzi deweloperskich i ponowne ich otwarcie.

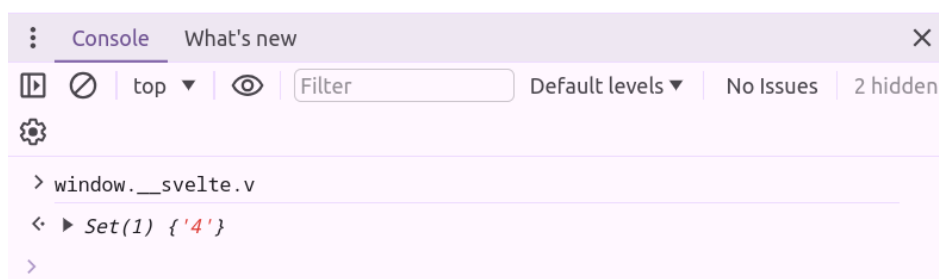
Problemy z działaniem narzędzia są na tyle częste, że praktycznie uniemożliwiają korzystanie z niego w przypadku złożonych aplikacji.

Svelte DevTools

No Svelte app detected reload

Not working? Did you...

- Build with dev mode enabled?
- Use Svelte version 4.0.0 or above?



Rysunek 11.3. Komunikat o niewykryciu biblioteki w narzędziu Svelte DevTools.

Autorskie rozwiązanie, za każdym razem automatycznie wykrywa obecne na stronie instance bibliotek i łączy się z nimi bez konieczności odświeżania karty. Wyniki przeprowadzonych testów wykazały także zdecydowanie większą stabilność jego działania.

11.2.2. Svelte DevTools+

Svelte DevTools+, pomimo, że jest nieoficjalnym rozszerzeniem, w czasie testów okazało się oferować większą niezawodność, niż oficjalne narzędzie.

Rozwiązanie to jest jednak ubogie w funkcjonalności, zarówno względem Svelte DevTools, jak i względem autorskiego rozwiązania. Pozwala ono na wyświetlenie jedynie hierarchii komponentów, ich stanu i właściwości (nie uwzględnia bloków Svelte ani elementów DOM).

Ponadto, interfejs narzędzia wymaga ręcznego rozwijania wszystkich węzłów prezentowanego drzewa, co w przypadku dużych aplikacji, może być bardzo niewygodne i czasochłonne.

Ciekawą funkcjonalnością jednak, której nie posiada żadne inne z analizowanych narzędzi, jest możliwość „cofania się” w historii aplikacji. Rozszerzenie wyświetla suwak, na którym, po każdej zmianie stanu w aplikacji, dodawany jest jeden segment (dolna część rys. 11.4). Przesunięcie suwaka w lewo, powoduje cofnięcie stanu aplikacji do momentu, w którym został dodany dany segment. Mimo, że funkcjonalność ta okazała się działać

Zważywszy, że RDT rozwijane jest od wielu lat przez zespół programistów stojących za samym Reactem, jest jednym z najpopularniejszych narzędzi tego typu i uchodzi za „state of the art” w swojej dziedzinie, to wynik ten można uznać za satysfakcjonujący.

12. Podsumowanie

W niniejszej pracy dokonano wszechstronnego zgłębienia obszaru integracji narzędzi deweloperskich z nowoczesnymi aplikacjami przeglądarkowymi. Pozwoliło to na zaprojektowanie i stworzenie uniwersalnego rozszerzenia umożliwiającego analizę struktury i przepływu danych między komponentami badanych aplikacji webowych.

Dzięki starannemu przebadaniu funkcjonalności i sposobu dystrybucji istniejących rozwiązań (rozdz. 2), udało się określić podstawowe wymagania funkcjonalne, które musi spełniać nowe narzędzie oraz wybrać implementację w postaci rozszerzenia przeglądarkowego jako najbardziej uniwersalną i dostępną dla użytkowników (podrozdz. 3.1).

Ponadto, przegląd rynku narzędzi deweloperskich pokazał, że nie istnieją rozwiązania uniwersalne, które pozwalałyby na analizę przepływu danych w więcej niż jednej bibliotece, nie mówiąc już o ich integracji z aplikacjami wykorzystującymi wiele bibliotek jednocześnie. Fakt ten potwierdził zasadność podjętych prac nad stworzeniem innowacyjnego narzędzia, będącego przedmiotem tej pracy.

Dogłębne badania nad mechanizmami działania bibliotek React (rozdz. 5) i Svelte (rozdz. 6), reprezentatywnych dla dwóch różnych podejść do zarządzania stanem w aplikacjach webowych, pozwoliły na opracowanie architektury rozszerzenia (podrozdz. 9.3) i na zaprojektowanie implementacji mechanizmów komunikacji (podrozdz. 9.4), w sposób wydajny i przejrzyste oddzielający logikę uniwersalną (np. podrozdz. 9.9, podrozdz. 9.5) od specyficznej dla każdej z bibliotek (podrozdz. 9.6).

Wnikliwe przeanalizowanie kodu źródłowego React Developer Tools (rozdz. 7) umożliwiło poznanie nietypowych problemów związanych z komunikacją między poszczególnymi skryptami w środowisku rozszerzeń przeglądarkowych (podrozdz. 7.4), z wydajnością (podrozdz. 7.8), a także z badaniem aplikacji w czasie ich uruchomienia (podrozdz. 7.10). Przystudiowanie ugruntowanych metodologii, wzorców architektonicznych i rozwiązań wykorzystanych w tym „state of the art” narzędziu, dało solidne podstawy do stworzenia stabilnego i wydajnego rozwiązania, a także pozwoliło poznać i zrozumieć wiele nieudokumentowanych mechanizmów działania samej biblioteki React.

Co więcej, w trakcie prac nad projektem napotkano dwa nieoczekiwane problemy, których rozwiązanie zaowocowało wniesieniem wkładu w naprawienie błędu w otwartoźródłowym projekcie „vitest-chrome” (podrozdz. 10.1.2) oraz stworzeniem i publikacją rozszerzenia do jednego z najpopularniejszych narzędzi, służących do budowania projektów webowych (podrozdz. 8.3, zał. 2).

Podsumowując, udało się zrealizować wszystkie założenia pracy, a niektóre z nich nawet z naddatkiem. Wyniki stworzonych testów jednostkowych i integracyjnych (rozdz. 10) oraz porównanie z istniejącymi rozwiązaniami (rozdz. 11), potwierdzają stabilność i wydajność autorskiego rozszerzenia, a testy manualne (podrozdz. 10.2) wykazują jego zgodność z wymaganiami i poprawność działania nawet w nietypowych scenariuszach.

Dowodem na kompletność pracy jest publikacja rozwiązania w sklepie rozszerzeń przeglądarek Chrome oraz w serwisie GitHub (zał. 1), jednak w świetle refleksji nad przyszłością rozwoju narzędzia (podrozdz. 12.1), można uznać, że prace nad nim dopiero się rozpoczy-

nają. Będąc świadomym iteracyjnego charakteru postępu technologicznego, mam nadzieję, że praca ta stwarza podwaliny pod dalsze prace badawcze i stanowić będzie katalizator kolejnych innowacji i doskonalenia w dziedzinie rozwoju narzędzi deweloperskich.

12.1. Kierunek dalszych prac

W trakcie prac nad projektem, udało się zrealizować wiele założeń, jednakże badania, eksperymenty i zauważone problemy odkryły wiele nowych możliwości i fascynujących pomysłów na dalszy rozwój narzędzia. Poniżej przedstawiono kierunki, które uznano za najbardziej obiecujące i wartościowe. W każdym z punktów podano odnośnik do sekcji, w której, z większą szczegółowością opisany został dany problem lub pomysł.

- **Optymalizacja wydajności** poprzez:
 - zastosowanie, podobnego do użytego w RDT, mechanizmu kodowania wiadomości o wykonywanych w aplikacji operacjach (podrozdz. 7.8),
 - zastosowanie odcinania niezmienionych poddrzew przy analizie obiektów Fiber (podrozdz. 9.7.6),
 - zaimplementowanie mechanizmu niezależnego odbudowywania drzewa elementów zarówno w kontekście panelu narzędzi deweloperskich, jak i w izolowanym skrypcie zawartości, pozwalającego na znaczne ograniczenie ilości przesyłanych danych i wykonywanie pełnej synchronizacji struktury drzewa, jedynie „na żądanie” panelu, przy każdorazowym jego otwarciu (podrozdz. 9.9.1).
- **Dodanie nowych funkcjonalności**, takich jak:
 - możliwość wykrywania i analizy zagnieżdżeń niestandardowych hooków w Reactie (podrozdz. 7.10),
 - pozwolenie użytkownikowi na sterowanie limitem głębokości dehydratacji właściwości i stanu komponentów (podrozdz. 9.6.2),
 - umożliwienie równoległego użycia StateViz z React Developer Tools, poprzez zastosowanie warunkowego wstrzykiwania hooka, lub „łatania” tego wstrzykiwanego przez RDT (podrozdz. 9.7.1),
 - wyświetlanie faktycznego rozmiaru elementów zaznaczonych w panelu narzędzi deweloperskich,
 - wyświetlanie atrybutów HTML i zarejestrowanych nasłuchiwań na zdarzenia dla elementów drzewa DOM pokazywanych w panelu narzędzi (podrozdz. 9.8.2),
 - obsługa innych przeglądarek niż Chrome, takich jak Firefox, czy Safari,
 - możliwość bezpośredniej edycji stanu i właściwości komponentów z poziomu narzędzi deweloperskich (podrozdz. 11.1.3, podrozdz. 11.2.1),
 - możliwość wyszukiwania poszczególnych elementów według ich nazwy, lub używając wyrażeń regularnych (podrozdz. 11.1.3, podrozdz. 11.2.1),
 - możliwość wyszukiwania elementów w panelu narzędzi deweloperskich, odpowiadających elementom strony, które zostały zaznaczone kursorem (podrozdz. 11.1.3, podrozdz. 11.2.1),

- możliwość przejścia do definicji komponentu w kodzie źródłowym aplikacji (podrozdz. 11.1.3),
 - podświetlanie na stronie i w panelu elementów, które zostały ponownie wyrenderowane w wyniku zmiany stanu aplikacji (podrozdz. 11.1.3),
 - możliwość zapamiętywania historii zmian struktury drzewa komponentów, ich stanu i właściwości, pozwalającego na „cofanie się w czasie” i porównywanie różnic między kolejnymi stanami aplikacji (podobnie jak robi to narzędzie Svelte-Devtools+; podrozdz. 11.2.2),
 - obsługa innych bibliotek, takich jak Angular, Vue, czy Preact, poprzez zaimplementowanie adapterów,
 - umożliwienie użytkownikowi wczytywania własnych adapterów i dynamiczne ładowanie ich kodu w trakcie działania rozszerzenia, co pozwoliłoby na rozszerzanie funkcjonalności rozwiązania bez konieczności ingerencji w kod źródłowy rozszerzenia. Takie rozwiązanie oferowałoby ogromne możliwości, jednak wymagałoby dogłębnej analizy potencjalnych zagrożeń związanych z bezpieczeństwem i stabilnością działania rozszerzenia.
- **Poprawa istniejących aspektów narzędzia**, takich jak:
 - wykrywanie i podświetlanie węzłów drzewa DOM odpowiadających poszczególnym elementom, wyświetlanym w panelu narzędzi deweloperskich dla Reacta (podrozdz. 9.7.4) i dla Svelte (podrozdz. 9.8.9),
 - sposób oczekiwania na załadowanie Svelte w złożonych aplikacjach przeglądarkowych (podrozdz. 10.2),
 - przejrzystość i czytelność danych wyświetlanych w panelu narzędzi deweloperskich poprzez zastosowanie bardziej intuicyjnych nazw, etykiet i kolorów,
 - dostępność i użyteczność narzędzia dla osób z niepełnosprawnościami poprzez zastosowanie zgodnych z WCAG standardów, takich jak kontrast kolorów, czy obsługa czytników ekranowych [108],
 - enkapsulacja i izolacja logiki uniwersalnej od tej specyficznej dla każdej z bibliotek poprzez przeniesienie z panelu narzędzi deweloperskich logiki, odpowiedzialnej za wybór sposobu filtrowania elementów dla konkretnych bibliotek, do poszczególnych adapterów (podrozdz. 9.10.4).
 - **Dalsza analiza**, takich aspektów jak:
 - chronologia zgłaszania odmontowań elementów w Reaccie (podrozdz. 9.7.5),
 - przyczyna zduplikowania wartości kontekstów w obiektach Fiber, reprezentujących komponenty funkcyjne Reacta, uruchomionego w trybie deweloperskim (podrozdz. 9.7.7),
 - możliwość połączenia rozszerzenia z aplikacjami Svelte uruchomionymi w trybie produkcyjnym (podrozdz. 9.8),
 - możliwość połączenia rozszerzenia z aplikacjami używającymi starszych wersji bibliotek niż React 16 i Svelte 4 (podrozdz. 9.8.1),

- sposób implementacji montowania bloków `each` w bibliotece Svelte (podrozdz. 9.8.6),
- istnienie możliwości aktualizacji innych danych w elementach DOM oprócz ich zawartości tekstowej w Svelte (podrozdz. 9.8.10).
- **Utrzymanie jakości i niezawodności rozszerzenia**, poprzez:
 - stworzenie testów jednostkowych dla warstwy prezentacji rozwiązania (podrozdz. 10.1),
 - stworzenie testów end-to-end, pozwalających na automatyczne zadbanie o poprawność działania rozwiązania, w przypadku potencjalnych zmian w obsługiwanych bibliotekach, sposobie funkcjonowania przeglądarki oraz API rozszerzeń Chrome.

Bibliografia

- [1] A. Volle, *Web application*. adr.: <https://www.britannica.com/topic/Web-application> (term. wiz. 03.01.2024).
- [2] „A Brief History of JavaScript”, w *DOM Scripting: Web Design with JavaScript and the Document Object Model*. Berkeley, CA: Apress, 2005, s. 3–10, ISBN: 978-1-4302-0062-8. DOI: 10.1007/978-1-4302-0062-8_1. adr.: https://doi.org/10.1007/978-1-4302-0062-8_1.
- [3] J. J. Garrett i in., „Ajax: A new approach to web applications”, 2005.
- [4] OpenJS Foundation - openjsf.org, *jQuery*. adr.: <https://jquery.com/> (term. wiz. 03.01.2024).
- [5] *AngularJS — Superheroic JavaScript MVW Framework*, mar. 2022. adr.: <https://angularjs.org/> (term. wiz. 03.01.2024).
- [6] *React*. adr.: <https://react.dev/> (term. wiz. 03.01.2024).
- [7] *Vue.js - The Progressive JavaScript Framework | Vue.js*. adr.: <https://vuejs.org/> (term. wiz. 03.01.2024).
- [8] *Svelte • Cybernetically enhanced web apps*. adr.: <https://svelte.dev/> (term. wiz. 03.01.2024).
- [9] M. Geers, „Micro Frontends”, *Micro Frontends*, sierp. 2017. adr.: <https://micro-frontends.org> (term. wiz. 02.06.2024).
- [10] A. Partovi, „Easily integrate multiple JavaScript frameworks into one web page”, *Medium*, lut. 2023. adr.: <https://medium.com/geekculture/easily-integrate-multiple-javascript-frameworks-into-one-web-page-8a785ec3a74c> (term. wiz. 03.01.2024).
- [11] *React DevTools • React Native*. adr.: <https://reactnative.dev/docs/react-devtools> (term. wiz. 20.01.2024).
- [12] *React Developer Tools – React*. adr.: <https://react.dev/learn/react-developer-tools> (term. wiz. 03.01.2024).
- [13] K. Aman, „How to Use React Dev Tools – With Example Code and Videos”, *FreeCodeCamp*, sty. 2023. adr.: <https://www.freecodecamp.org/news/how-to-use-react-dev-tools> (term. wiz. 20.01.2024).
- [14] *D3 by Observable | The JavaScript library for bespoke data visualization*. adr.: <https://d3js.org/> (term. wiz. 20.01.2024).
- [15] Czer. 2020. adr.: <https://www.realizeforreact.com> (term. wiz. 20.01.2024).
- [16] *Realize*. adr.: <https://github.com/oslabs-beta/Realize> (term. wiz. 03.01.2024).
- [17] *react/packages/react-devtools/OVERVIEW.md*. adr.: <https://github.com/facebook/react/blob/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools/OVERVIEW.md> (term. wiz. 20.01.2024).
- [18] *React-Sight/React-Sight: Visualization tool for React, with support for Fiber, Router (v4), and Redux*. adr.: <https://github.com/React-Sight/React-Sight> (term. wiz. 03.01.2024).
- [19] *React Sight*. adr.: <https://www.reactstight.com/> (term. wiz. 20.01.2024).

- [20] *svelte-devtools*, kw. 2024. adr.: <https://github.com/sveltejs/svelte-devtools/tree/aea036f42a6e2783f43fe7fb5080fcc574a5aba1> (term. wiz. 25.04.2024).
- [21] *Svelte-DevTools-Plus*, maj 2024. adr.: <https://github.com/oslabs-beta/Svelte-DevTools-Plus> (term. wiz. 14.05.2024).
- [22] *Svelte DevTools+*, maj 2024. adr.: <https://www.sveltedevtools.com> (term. wiz. 14.05.2024).
- [23] *Browser Market Share Worldwide | Statcounter Global Stats*. adr.: <https://gs.statcounter.com/browser-market-share/> (term. wiz. 20.01.2024).
- [24] *Chromium (web browser) - Wikipedia*. adr.: [https://en.wikipedia.org/wiki/Chromium_\(web_browser\)](https://en.wikipedia.org/wiki/Chromium_(web_browser)) (term. wiz. 20.01.2024).
- [25] *Browser extensions - Mozilla | MDN*. adr.: <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions> (term. wiz. 20.01.2024).
- [26] *State of JavaScript 2022: Front-end Frameworks*, list. 2023. adr.: <https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/> (term. wiz. 20.01.2024).
- [27] *Manifest file format | Chrome for Developers*. adr.: <https://developer.chrome.com/docs/extensions/reference/manifest?hl=en> (term. wiz. 20.01.2024).
- [28] *Events in service workers | Extensions | Chrome for Developers*. adr.: <https://developer.chrome.com/docs/extensions/develop/concepts/service-workers/events?hl=en> (term. wiz. 20.01.2024).
- [29] *scripting.ExecutionWorld - Mozilla | MDN*, maj 2024. adr.: <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/scripting/ExecutionWorld> (term. wiz. 03.05.2024).
- [30] *Content scripts - Mozilla | MDN*. adr.: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts (term. wiz. 20.01.2024).
- [31] *Add a popup | Extensions | Chrome for Developers*. adr.: <https://developer.chrome.com/docs/extensions/develop/ui/add-popup?hl=en> (term. wiz. 20.01.2024).
- [32] *Options page - Mozilla | MDN*. adr.: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/user_interface/Options_pages (term. wiz. 20.01.2024).
- [33] *Extend DevTools | Extensions | Chrome for Developers*. adr.: <https://developer.chrome.com/docs/extensions/how-to/devtools/extend-devtools?hl=en> (term. wiz. 20.01.2024).
- [34] *API reference | Chrome for Developers*. adr.: <https://developer.chrome.com/docs/extensions/reference/api?hl=en> (term. wiz. 20.01.2024).
- [35] *Message passing*, wrz. 2012. adr.: <https://developer.chrome.com/docs/extensions/develop/concepts/messaging> (term. wiz. 26.04.2024).
- [36] *react/CHANGELOG.md*. adr.: <https://github.com/facebook/react/blob/main/CHANGELOG.md#1820-june-14-2022> (term. wiz. 03.02.2024).

-
- [37] *[Fizz] Track Key Path (#27243) · facebook/react*, sierp. 2024. adr.: <https://github.com/facebook/react/commit/98f3f14d2e06fb785103296318204cd154d5d0ed> (term. wiz. 03.02.2024).
 - [38] *Passing Props to a Component – React*, angielski. adr.: <https://react.dev/learn/passing-props-to-a-component> (term. wiz. 03.02.2024).
 - [39] M. Mohan, „How React works under the hood”, angielski, *FreeCodeCamp*, wrz. 2019. adr.: <https://www.freecodecamp.org/news/react-under-the-hood>.
 - [40] *React Components, Elements, and Instances – React Blog*, kw. 2024. adr.: <https://legacy.reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html> (term. wiz. 10.04.2024).
 - [41] *react-fiber-architecture*, kw. 2024. adr.: <https://github.com/acdlite/react-fiber-architecture> (term. wiz. 10.04.2024).
 - [42] K. Kalyanaraman, „A deep dive into React Fiber - LogRocket Blog”, *LogRocket Blog*, kw. 2022. adr.: <https://blog.logrocket.com/deep-dive-react-fiber> (term. wiz. 21.04.2024).
 - [43] *Built-in React Hooks – React*, angielski. adr.: <https://react.dev/reference/react/hooks> (term. wiz. 03.02.2024).
 - [44] *useState – React*, kw. 2024. adr.: <https://react.dev/reference/react/useState> (term. wiz. 09.04.2024).
 - [45] *Extracting State Logic into a Reducer – React*, kw. 2024. adr.: <https://react.dev/learn/extracting-state-logic-into-a-reducer> (term. wiz. 09.04.2024).
 - [46] GfG, „What is prop drilling and how to avoid it?”, *GeeksforGeeks*, grud. 2023. adr.: <https://www.geeksforgeeks.org/what-is-prop-drilling-and-how-to-avoid-it> (term. wiz. 09.04.2024).
 - [47] *Passing Data Deeply with Context – React*, kw. 2024. adr.: <https://react.dev/learn/passing-data-deeply-with-context> (term. wiz. 09.04.2024).
 - [48] *useRef – React*, kw. 2024. adr.: <https://react.dev/reference/react/useRef> (term. wiz. 09.04.2024).
 - [49] *useEffect – React*, kw. 2024. adr.: <https://react.dev/reference/react/useEffect> (term. wiz. 09.04.2024).
 - [50] *useMemo – React*, kw. 2024. adr.: <https://react.dev/reference/react/useMemo#skipping-re-rendering-of-components> (term. wiz. 09.04.2024).
 - [51] *Reusing Logic with Custom Hooks – React*, kw. 2024. adr.: <https://react.dev/learn/reusing-logic-with-custom-hooks#extracting-your-own-custom-hook-from-a-component> (term. wiz. 09.04.2024).
 - [52] Contributors to Wikimedia projects, *Svelte - Wikipedia*, lut. 2024. adr.: <https://en.wikipedia.org/w/index.php?title=Svelte&oldid=1210470585> (term. wiz. 15.04.2024).
 - [53] *Labeled statement - JavaScript | MDN*, kw. 2024. adr.: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label> (term. wiz. 15.04.2024).

- [54] *Svelte components • Docs • Svelte*, kw. 2024. adr.: <https://svelte.dev/docs/svelte-components> (term. wiz. 15.04.2024).
- [55] *Logic blocks • Docs • Svelte*, kw. 2024. adr.: <https://svelte.dev/docs/logic-blocks> (term. wiz. 15.04.2024).
- [56] *Promise - JavaScript | MDN*, kw. 2024. adr.: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (term. wiz. 15.04.2024).
- [57] *svelte/transition • Docs • Svelte*, kw. 2024. adr.: <https://svelte.dev/docs/svelte-transition> (term. wiz. 15.04.2024).
- [58] *Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron*, kw. 2024. adr.: <https://www.electronjs.org> (term. wiz. 16.04.2024).
- [59] *react/packages/react-devtools-core*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools-core> (term. wiz. 16.04.2024).
- [60] *react/packages/react-devtools-extensions*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools-extensions> (term. wiz. 16.04.2024).
- [61] *react/packages/react-devtools-inline*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools-inline> (term. wiz. 16.04.2024).
- [62] Kw. 2024. adr.: <https://codesandbox.io> (term. wiz. 16.04.2024).
- [63] *StackBlitz | Instant Dev Environments | Click. Code. Done.* Kw. 2024. adr.: <https://stackblitz.com> (term. wiz. 16.04.2024).
- [64] *Replay - The time-travel debugger from the future.* Kw. 2024. adr.: <https://www.replay.io> (term. wiz. 16.04.2024).
- [65] *react/packages/react-devtools-shared*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools-shared> (term. wiz. 16.04.2024).
- [66] *react/packages/react-devtools-shell*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools-shell> (term. wiz. 16.04.2024).
- [67] *react/packages/react-devtools*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-devtools> (term. wiz. 16.04.2024).
- [68] *react/packages/react-debug-tools*, kw. 2024. adr.: <https://github.com/facebook/react/tree/98f3f14d2e06fb785103296318204cd154d5d0ed/packages/react-debug-tools> (term. wiz. 16.04.2024).
- [69] *1736575 - Support execution world MAIN in 'scripting.executeScript()' and content scripts*, kw. 2024. adr.: https://bugzilla.mozilla.org/show_bug.cgi?id=1736575 (term. wiz. 16.04.2024).
- [70] *About Manifest V2*, list. 2020. adr.: <https://developer.chrome.com/docs/extensions/mv2> (term. wiz. 16.04.2024).

-
- [71] *What is React Fast Refresh?*, kw. 2024. adr.: <https://www.netlify.com/blog/2020/12/03/what-is-react-fast-refresh> (term. wiz. 17.04.2024).
 - [72] *– React*, kw. 2024. adr.: <https://react.dev/reference/react/StrictMode> (term. wiz. 18.04.2024).
 - [73] *Window: postMessage() method - Web APIs | MDN*, mar. 2024. adr.: <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage> (term. wiz. 18.04.2024).
 - [74] *The structured clone algorithm - Web APIs | MDN*, kw. 2024. adr.: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm (term. wiz. 18.04.2024).
 - [75] *Jonghakseo/chrome-extension-boilerplate-react-vite: Chrome Extension Boilerplate with React + Vite + Typescript*. adr.: <https://github.com/Jonghakseo/chrome-extension-boilerplate-react-vite> (term. wiz. 20.01.2024).
 - [76] *Vite (software) - Wikipedia*. adr.: [https://en.wikipedia.org/wiki/Vite_\(software\)](https://en.wikipedia.org/wiki/Vite_(software)) (term. wiz. 20.01.2024).
 - [77] *JavaScript modules - JavaScript | MDN*. adr.: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules> (term. wiz. 20.01.2024).
 - [78] *Features | Vite*. adr.: <https://vitejs.dev/guide/features%5C#hot-module-replacement> (term. wiz. 20.01.2024).
 - [79] *Why Vite | Vite*. adr.: <https://vitejs.dev/guide/why.html%5C#why-not-bundle-with-esbuild> (term. wiz. 21.01.2024).
 - [80] *soultice/vite-plugin-force-inline-module: Tricks vite into inlining modules, helpful when code should be centralized but cannot be imported during runtime due to arbitrary constraints*. adr.: <https://github.com/soultice/vite-plugin-force-inline-module> (term. wiz. 20.01.2024).
 - [81] *Plugin API | Vite*. adr.: <https://vitejs.dev/guide/api-plugin> (term. wiz. 20.01.2024).
 - [82] *Vitest | Next Generation testing framework*. adr.: <https://vitest.dev/> (term. wiz. 21.01.2024).
 - [83] *MIT License*. adr.: <https://mit-license.org/> (term. wiz. 21.01.2024).
 - [84] *npm-stat: vite-plugin-inline-imports*. adr.: <https://npm-stat.com/charts.html?package=vite-plugin-inline-imports> (term. wiz. 21.01.2024).
 - [85] *Handbook - Unions and Intersection Types*, angielski, kw. 2024. adr.: <https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html#discriminating-unions> (term. wiz. 26.04.2024).
 - [86] *JavaScript data types and data structures - JavaScript | MDN*, maj 2024. adr.: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures (term. wiz. 05.05.2024).
 - [87] *Contributors to Wikimedia projects, Monkey patch - Wikipedia*, mar. 2024. adr.: https://en.wikipedia.org/w/index.php?title=Monkey_patch&oldid=1216352798 (term. wiz. 04.05.2024).

- [88] *react-testing-library*, maj 2024. adr.: <https://github.com/testing-library/react-testing-library#readme> (term. wiz. 11.05.2024).
- [89] *Features • GitHub Actions*, maj 2024. adr.: <https://github.com/features/actions> (term. wiz. 11.05.2024).
- [90] *vitest-badge-action*, maj 2024. adr.: <https://github.com/wjervis7/vitest-badge-action/tree/v1.0.0> (term. wiz. 11.05.2024).
- [91] *git-auto-commit-action*, maj 2024. adr.: <https://github.com/stefanzweifel/git-auto-commit-action/tree/v5> (term. wiz. 11.05.2024).
- [92] *probil/vitest-chrome: A complete mock of the Chrome API for Chrome extensions for use with Vitest*. adr.: <https://github.com/probil/vitest-chrome> (term. wiz. 21.01.2024).
- [93] *How vitest resolves dependency exports • vitest-dev/vitest • Discussion #4233*. adr.: <https://github.com/vitest-dev/vitest/discussions/4233> (term. wiz. 21.01.2024).
- [94] *patch-package*, maj 2024. adr.: <https://www.npmjs.com/package/patch-package> (term. wiz. 11.05.2024).
- [95] *Pull requests • probil/vitest-chrome*. adr.: <https://github.com/probil/vitest-chrome/pull/1> (term. wiz. 21.01.2024).
- [96] *forwardRef – React*, maj 2024. adr.: <https://react.dev/reference/react/forwardRef> (term. wiz. 11.05.2024).
- [97] *Jira Clone*, mar. 2021. adr.: <https://jira.ivorreic.com/project/board> (term. wiz. 12.05.2024).
- [98] *jira_clone*, maj 2024. adr.: https://github.com/oldboyxx/jira_clone (term. wiz. 12.05.2024).
- [99] *cypress-realworld-app*, maj 2024. adr.: <https://github.com/cypress-io/cypress-realworld-app> (term. wiz. 12.05.2024).
- [100] *BBC News Thai*, maj 2024. adr.: <https://www.bbc.com/thai> (term. wiz. 12.05.2024).
- [101] *simorgh*, maj 2024. adr.: <https://github.com/bbc/simorgh> (term. wiz. 12.05.2024).
- [102] *QRnamic*. adr.: <https://qrnamic.steciuk.dev> (term. wiz. 24.05.2024).
- [103] *realworld*, maj 2024. adr.: <https://github.com/sveltejs/realworld> (term. wiz. 12.05.2024).
- [104] *SvelteKit • Web development, streamlined*, maj 2024. adr.: <https://kit.svelte.dev> (term. wiz. 12.05.2024).
- [105] *react-devtools : Timeout unable to inspect element*, maj 2024. adr.: <https://stackoverflow.com/questions/70897741/react-devtools-timeout-unable-to-inspect-element> (term. wiz. 13.05.2024).
- [106] *react-devtools : TimeoutError: Timed out while inspecting element 2*, maj 2024. adr.: <https://stackoverflow.com/questions/74835267/react-devtools-timeouterror-timed-out-while-inspecting-element-2> (term. wiz. 13.05.2024).
- [107] *[DevTools Bug] Could not inspect element with id 10 • Issue #21579*, maj 2024. adr.: <https://github.com/facebook/react/issues/21579> (term. wiz. 13.05.2024).

- [108] W3c, „Introduction to Web Accessibility”, *Web Accessibility Initiative (WAI)*, mar. 2024. adr.: <https://www.w3.org/WAI/fundamentals/accessibility-intro> (term. wiz. 13.05.2024).

Wykaz symboli i skrótów

AJAX – Asynchronous JavaScript and XML
API – Application Programming Interface
CSS – Cascading Style Sheets
DCE – Dead Code Elimination
DOM – Document Object Model
ESM – ECMAScript Modules
GMT – Greenwich Mean Time
HMR – Hot Module Replacement
HTML – HyperText Markup Language
JS – JavaScript
JSX – JavaScript XML
list. – listing
MIT – Massachusetts Institute of Technology
npm – Node Package Manager
podrozdz. – podrozdział
RDT – React Developer Tools
RDT – React Developer Tools
rozdz. – rozdział
rys. – rysunek
SASS – Syntactically Awesome Style Sheets
SPA – Single Page Application
SSG – Static Site Generator
tab. – tabela
UML – Unified Modeling Language
WCAG – Web Content Accessibility Guidelines
zał. – załącznik

Spis rysunków

1.1	Problemy związane z zarządzaniem stanem w aplikacjach przeglądarkowych .	12
2.1	Widok zakładki „Components” w React Developer Tools.	14
2.2	Interfejs narzędzia Realize.	16
2.3	Interfejs narzędzia React Sight.	17
2.4	Interfejs narzędzia Svelte DevTools.	17
2.5	Interfejs narzędzia Svelte DevTools+.	18
4.1	Fragment anatomii rozszerzenia przeglądarkowego.	23
4.2	Dostępne interfejsy i sposoby komunikacji między fragmentami rozszerzenia [33].	25
5.1	Przykładowa struktura drzewa Fiber.	32

5.2	Porównanie przekazywania danych przez „prop drilling” (po lewej) i przez kontekst (po prawej) w Reaccie [47].	33
5.3	Wyświetlanie niestandardowego hooka w React Developer Tools z <code>useDebugValue</code> (po lewej) i bez (po prawej).	36
7.1	Architektura punktów wejścia dla rozszerzenia React Developer Tools.	44
7.2	Okno modalne „production” rozszerzenia React Developer Tools.	45
9.1	Okno modalne i ikona rozszerzenia.	66
9.2	Diagram UML klas abstrakcji nad mechanizmem połączeń długotrwałych. . .	68
9.3	Diagram UML klasy abstrakcji nad mechanizmem <code>windows.postMessage</code> . . .	69
9.4	Diagram UML klasy abstrakcyjnej <code>Adapter</code>	70
9.5	Możliwe wyniki wyścigu między RDT a StateViz przy wstrzykiwaniu hooka. . .	75
9.6	Powodzenie w przypadku oczekiwania StateViz na wstrzyknięcie hooka przez RDT.	76
9.7	Niepowodzenie w przypadku zbyt krótkiego oczekiwania StateViz na wstrzyknięcie hooka przez RDT.	77
9.8	Optymalizacja procesu wykrywania niezbędnych do przesłania danych o odmontowywanych węzłach.	81
9.9	Uproszczony ślad wywołań funkcji <code>m</code> ze wstrzykniętym kodem adaptera dla zagnieżdżonych bloków Svelte.	88
9.10	Struktura interfejsu użytkownika panelu narzędzi deweloperskich.	94
9.11	Widok ustawień panelu narzędzi deweloperskich.	95
9.12	Widok szczegółów komponentu ze złożoną wartością właściwości.	98
10.1	Zrzut ekranu, przedstawiający działanie rozszerzenia w złożonej aplikacji Reacta - „cypress-realworld-app”.	102
11.1	Komunikat o przekroczeniu limitu czasu na odpowiedź w React Developer Tools.	106
11.2	Błędy w konsoli deweloperskiej narzędzia Svelte DevTools.	107
11.3	Komunikat o niewykryciu biblioteki w narzędziu Svelte DevTools.	108
11.4	Interfejs narzędzia Svelte DevTools+.	109

Spis załączników

1.	Rozszerzenie StateViz	124
2.	Rozszerzenie vite-plugin-inline-imports	124
3.	Scenariusze i aktualne wyniki testów jednostkowych i integracyjnych rozszerzenia StateViz	124
4.	Struktura katalogów projektu rozszerzenia StateViz	124
5.	Diagram sekwencji obrazujący komunikację między częściami rozszerzenia StateViz	127

Załącznik 1. Rozszerzenie StateViz

Kod źródłowy rozszerzenia dostępny jest pod adresem: <https://github.com/steciuk/StateViz>.

Rozszerzenie jest dostępne do zainstalowania pod adresem: <https://chromewebstore.google.com/detail/stateviz/fckfpalhoncacipdeapifoaljchalin>.

Film, prezentujący działanie rozszerzenia, dostępny jest pod adresem: <https://www.youtube.com/watch?v=QPvJOuqo52k>.

Załącznik 2. Rozszerzenie vite-plugin-inline-imports

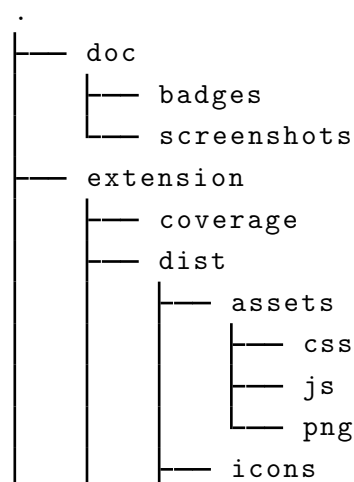
Kod źródłowy rozszerzenia dostępny jest pod adresem: <https://github.com/steciuk/vite-plugin-inline-imports>.

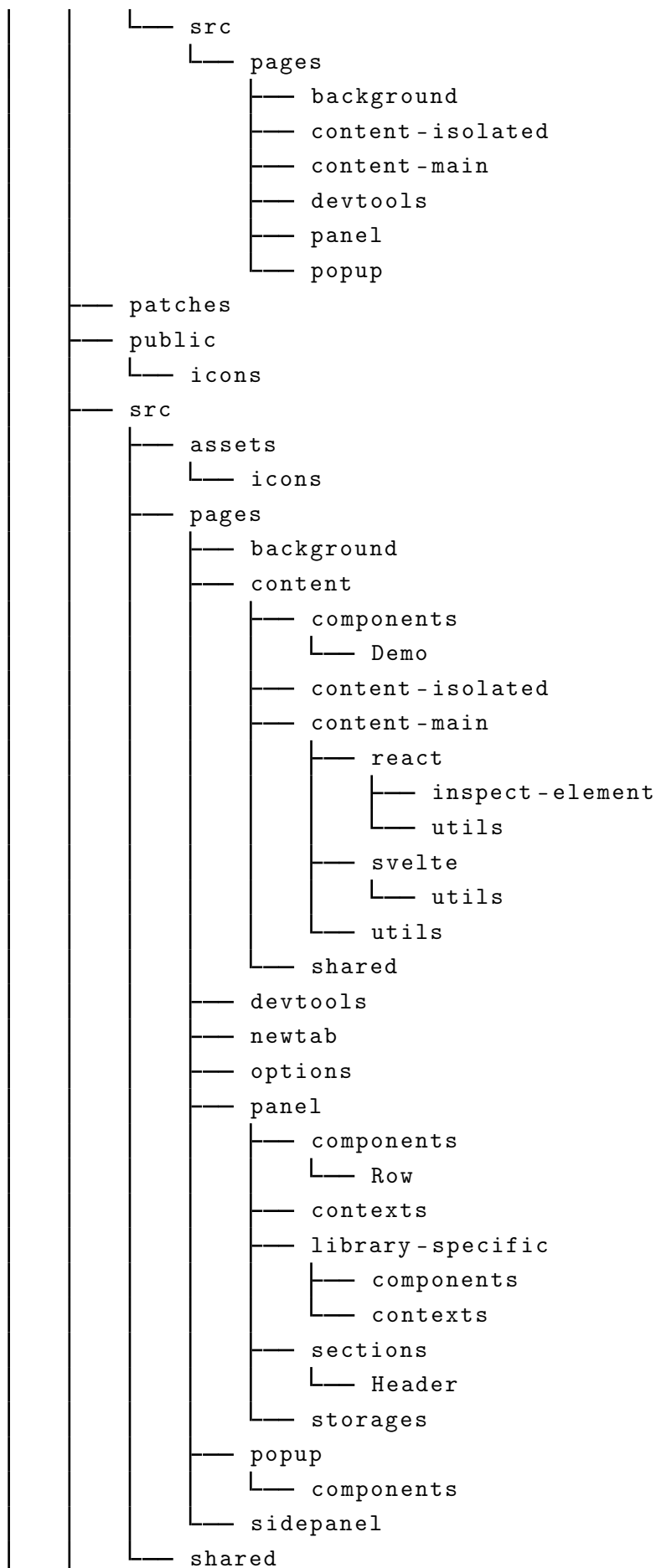
Rozszerzenie dostępne jest w formie pakietu npm pod adresem: <https://www.npmjs.com/package/vite-plugin-inline-imports>.

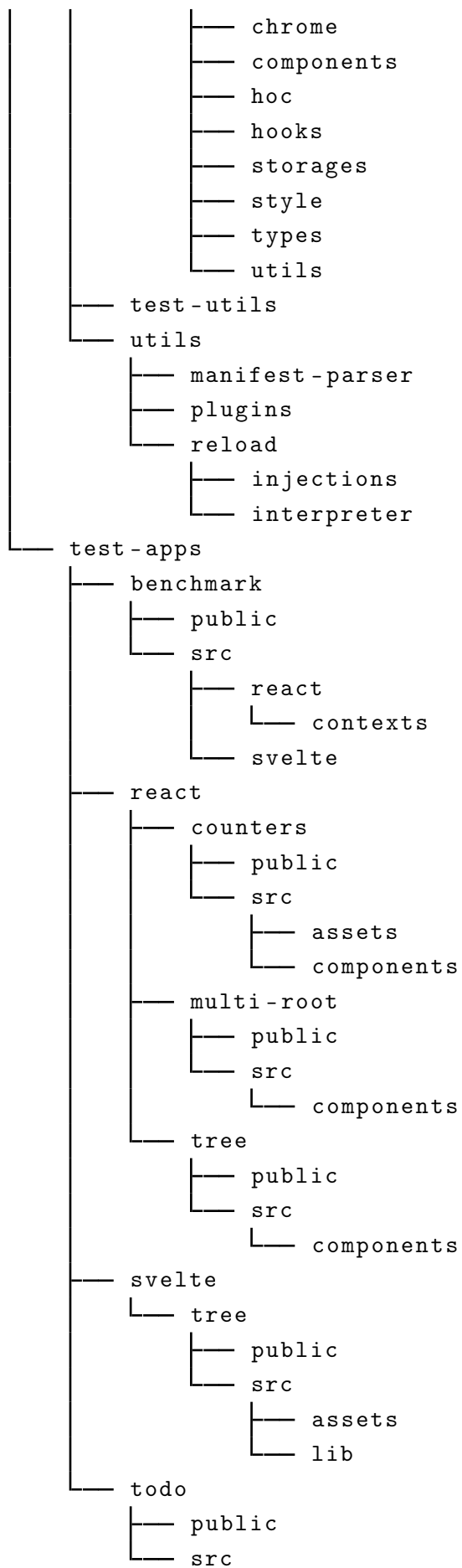
Załącznik 3. Scenariusze i aktualne wyniki testów jednostkowych i integracyjnych rozszerzenia StateViz

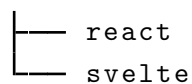
Scenariusze i wszystkie aktualne wyniki testów rozszerzenia dostępne są pod adresem: <https://github.com/steciuk/StateViz/blob/main/extension/test-output.json>.

Załącznik 4. Struktura katalogów projektu rozszerzenia StateViz









Załącznik 5. Diagram sekwencji obrazujący komunikację między częściami rozszerzenia StateViz

Ze względu na rozmiar diagramu, został on załączony na kolejnych stronach. Jest on także dostępny (w wersji elektronicznej) pod adresem: <https://raw.githubusercontent.com/steciuk/StateViz/main/doc/communication-UML.png>.

Nazwy wiadomości pisane wielkimi literami (np. `EXAMPLE_MESSAGE`) oznaczają konkretne typy wiadomości zdefiniowane w kodzie źródłowym rozszerzenia (podrozdz. 9.4.1).

