Wrocław University of Science and Technology
Faculty of Information and Communication Technology

Field of study: **Computer Engineering**
Speciality: **Computer Graphics and Multimedia Systems**

MASTER THESIS

Comparison of the state management libraries for React application

Michał Droń

Supervisor
dr inż. Tomasz Szandała

Keywords: frontend, react, state management

WROCŁAW 2024

ABSTRACT

State management is a very important part of the process of building React applications, but research on how different state management libraries compare in controlled environments appears to be lacking. This study helps to fill that gap by comparing how effective the different state management libraries are in a controlled environment. In this thesis, a React e-commerce store was developed using a Saleor API as a back-end. This base application was copied six times, and each of them used a different state management library while keeping the functionality and experience consistent. The comparison of the libraries was based on important metrics: performance, package size, memory consumption, and ecosystem support. The results obtained lend valuable insights to the developers and practical recommendations they can rely upon to make decisions towards the best state management library that can be appropriate for specific project requirements.

STRESZCZENIE

Zarządzanie stanem jest bardzo ważną częścią procesu budowy aplikacji w React, jednakże badania na temat porównania różnych bibliotek zarządzania stanem w kontrolowanych warunkach wydają się być niewystarczające. Niniejsza praca pomaga wypełnić tę lukę poprzez porównanie skuteczności różnych bibliotek do zarządzania stanem w kontrolowanym środowisku. W ramach tego badania stworzono sklep e-commerce wykorzystujący bibliotekę React, korzystający z Saleor API jako back-end. Ta podstawowa aplikacja została skopiowana sześć razy, a następnie wykorzystano różną bibliotekę do zarządzania stanem w każdej kopii, utrzymując spójność funkcjonalności i doświadczenia użytkownika. Porównanie bibliotek opierało się na ważnych metrykach: wydajności, rozmiarze paczki, zużyciu pamięci oraz wsparciu ekosystemu. Otrzymane wyniki dostarczają cennych informacji programistom oraz praktycznych rekomendacji, na podstawie których mogą polegać przy podejmowaniu decyzji o wyborze najlepszej biblioteki zarządzania stanem, odpowiedniej dla specyficznych wymagań projektu.

CONTENTS

Introduction 4

Objective of the thesis 4

Scope of the thesis 4

1. Background and Literature Review 5

1.1. Introduction to React 5

1.2. Introduction to state management in front-end development 7

1.3. Overview of existing research and comparisons 8

1.4. Selected solutions for comparison 9

2. Methodology 10

2.1. Test application 10

2.2. Metrics for evaluation 10

2.2.1. Bundle size 10

2.2.2. Performance 10

2.2.3. Memory consumption 12

2.2.4. Ecosystem 13

2.3. Hardware 13

2.4. API selection and justification 13

2.5. Tools and technologies used 14

2.5.1. Bundler 14

2.5.2. TypeScript 14

2.5.3. Linter 14

2.5.4. Routing 14

2.5.5. Data fetching layer 15

2.5.6. UI 15

2.5.7. Forms 15

3. E-commerce store implementation 16

3.1. Common boilerplate 16

3.2. Setup 17

3.2.1. Running Saleor API 17

3.2.2. Store configuration 17

3.3. Routing & pages 17

3.4. Data fetching 19

3.5. Authentication . . . 20
Available actions . . . 20
3.6. Product list . . . 21
3.6.1. Filtering . . . 21
3.6.2. Searching . . . 23
3.6.3. Sorting . . . 23
3.7. Checkout . . . 23
Adding products to cart . . . 23
Finalizing the checkout . . . 24
3.8. State delegated to libraries . . . 25

4. Experiment . . . 28

4.1. Redux . . . 28
4.1.1. Overview of Redux . . . 28
4.1.2. Implementation details . . . 28
4.1.3. Results . . . 28
4.2. MobX . . . 32
4.2.1. Overview of MobX . . . 32
4.2.2. Implementation details . . . 32
4.2.3. Results . . . 34
4.3. Context API . . . 36
4.3.1. Overview of Context API . . . 36
4.3.2. Implementation details . . . 36
4.3.3. Results . . . 37
4.4. Zustand . . . 41
4.4.1. Overview of Zustand . . . 41
4.4.2. Implementation details . . . 41
4.4.3. Results . . . 41
4.5. Recoil . . . 44
4.5.1. Overview of Recoil . . . 44
4.5.2. Implementation details . . . 45
4.5.3. Results . . . 45
4.6. Jotai . . . 48
4.6.1. Overview of Jotai . . . 48
4.6.2. Implementation details . . . 48
4.6.3. Results . . . 49

5. Discussion . . . 52

5.1. Comparison of bundle size . . . 52
5.2. Comparison of performance . . . 53
5.3. Comparison of memory consumption . . . 54
5.4. Comparison of ecosystem . . . 55

5.5. Recommendations 56
6. Conclusions 64
6.1. Limitations 64
Limited scenarios for state updates 64
Fixed API choice 64
Limited ecosystem evaluation 64
Hardware dependence 65
Focus on Single-Page Applications 65
Time relevance 65
6.2. Future work 65
Diverse hardware testing 65
Testing on different architectures 65
Developer experience 65
Code maintainability 66
Testability 66
Bibliography 67

INTRODUCTION

State management system of an application defines how data is controlled, distributed and moved within the application [7].

Nowadays there are multiple ways to handle state in React applications. Often it is difficult to choose adequate solution for a given project. Developers rely on their personal experience, online recommendations and anecdotal reports. There is however little research supporting usage of a particular solution.

OBJECTIVE OF THE THESIS

Aim of this thesis is to perform comprehensive analysis and comparison of various state management solutions, in context of their usage in React applications. The objective is to evaluate which of the researched solutions performs best in different scenarios, including aspects like performance, ecosystem or bundle size.

SCOPE OF THE THESIS

The research is going to be based on creating a test application which contains features common in React applications like forms, complex state update, data fetching, sorting, filtering. The test application is going to be an e-commerce store, which uses existing open source API, which allows focusing on front-end aspects. The base application is going to be then duplicated six times. Each copy will use different state management library: Redux, MobX, React useContext, Zustand, Jotai, Recoil. Functionality and user interface is going to stay consistent between applications, aiming to minimize code differences which could affect metrics.

1. BACKGROUND AND LITERATURE REVIEW

1.1. INTRODUCTION TO REACT

React is a widely used JavaScript library for building interfaces, developed by Facebook back in 2011, with initial release in 2013. The library itself was started by Facebook engineer Jordan Walke to help fix some of the struggles Facebook was having at the time of supporting their complex user interfaces for their growing suite of web applications, notably Facebook Ads [13]. In 2013, React was open-sourced to the public. The primary focus of React was to provide developers with the ability to build large dynamic web applications in which changes could be rendered fast and effectively without reloading the page. React introduced the concept of a "virtual DOM" that makes the updating and rendering of components very effective in terms of performance and thus facilitates an easier way of developing interactive UIs. Since its release, React has been one of the most widely adopted tools in front-end development, influencing a lot of the design behind many other frameworks and libraries.

As of 2024, React is still the most popular front-end library for building applications which is supported by data from Google Trends (figure 1.1) and Stack Overflow Developer Survey from 2024 (figure 1.2).

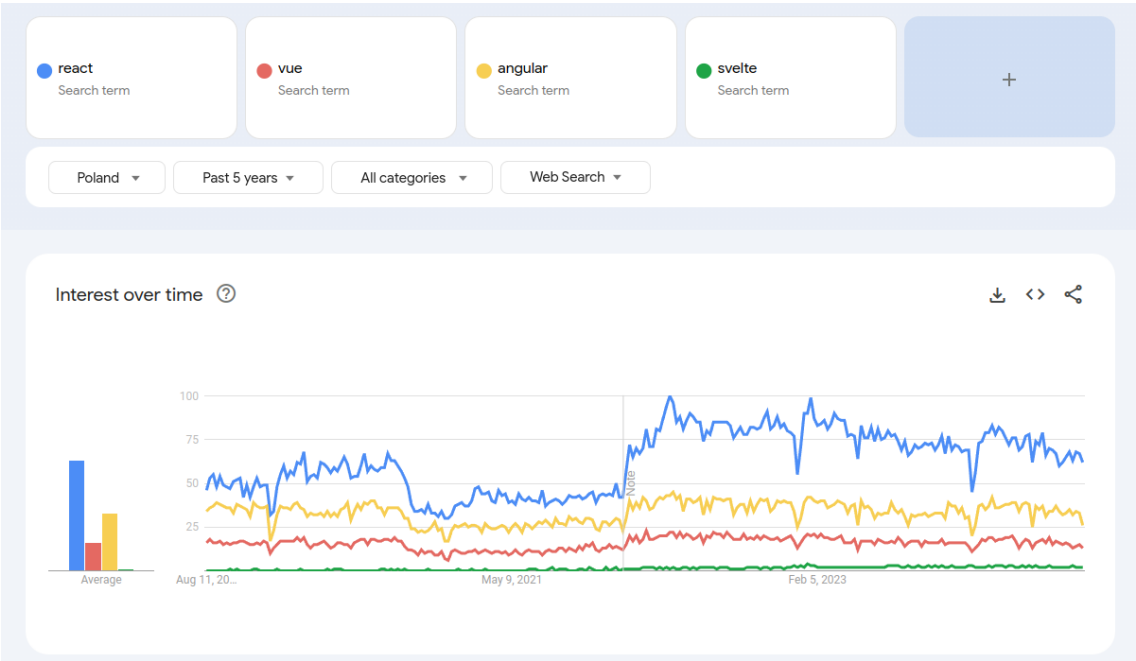

Fig. 1.1. Popularity of front-end frameworks in Google Trends

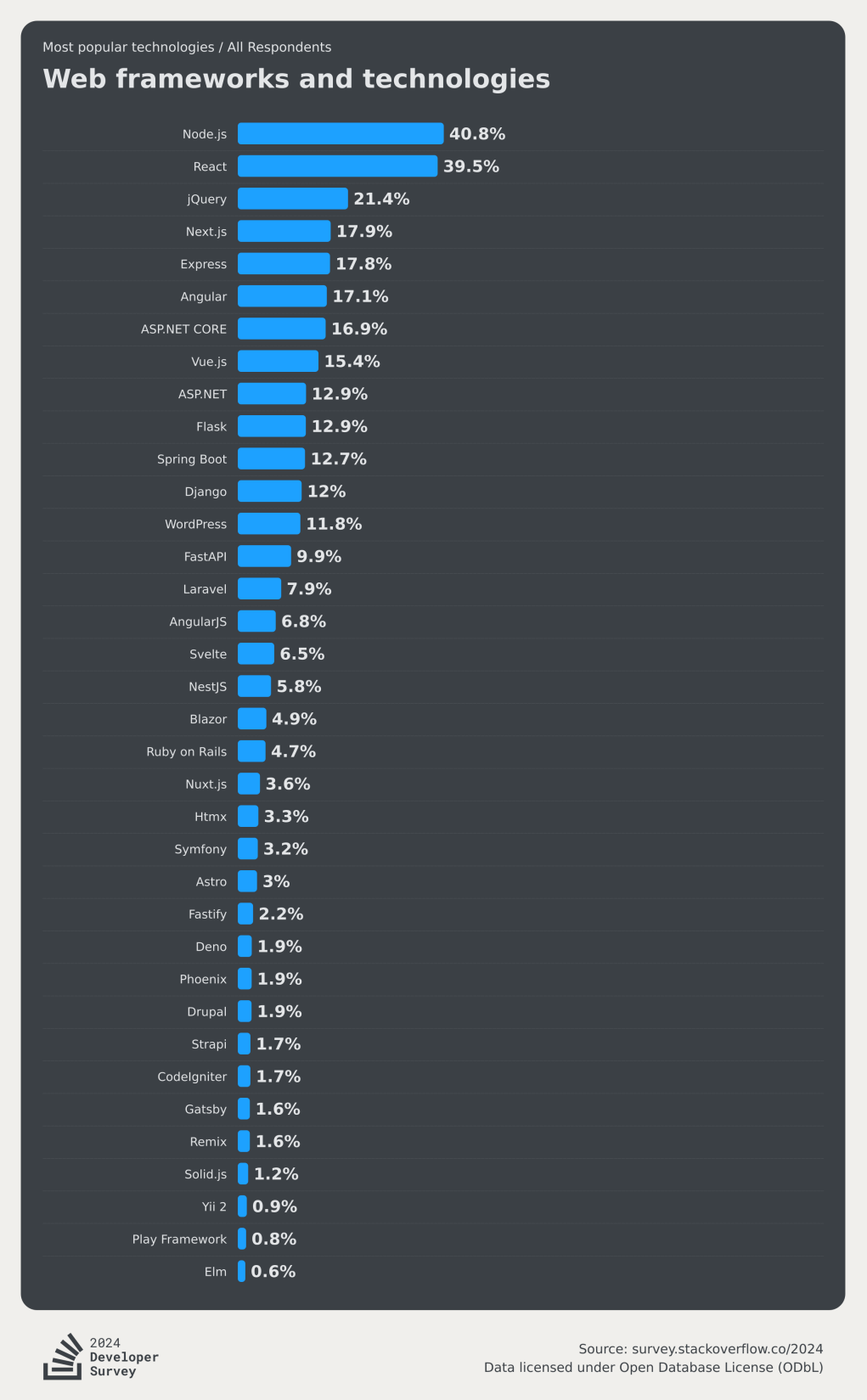

Fig. 1.2. Popularity of web frameworks in Stack Overflow Developer Survey 2024 [23]

1.2. INTRODUCTION TO STATE MANAGEMENT IN FRONT-END DEVELOPMENT

State management has a long history that tracks back to creation of React itself. From the very beginning it allowed front-end developers to create interactive UIs and seamless data exchange with databases.

Initially, when React only supported class components, state was handled using `this.state` (attribute of Component class) and `this.setState()` (method of Component class). This approach - local state management - is still used today, albeit due to decreasing popularity of class components the hooks version is favoured, which we are going to discuss later on.

In 2014 Flux architecture was proposed by creators of React [10], which was not a library but rather a set of recommendations, that allowed developers to reason about state changes more easily in complex applications. Flux proposed unidirectional data flow, which is also an important feature of React. There is a central **dispatcher**, implemented as a singleton, that handles user actions. Moreover, this pattern introduced **stores**, which hold the data of an application (figure 1.3). These inventions were the foundation inspiring later generations of state management solutions.

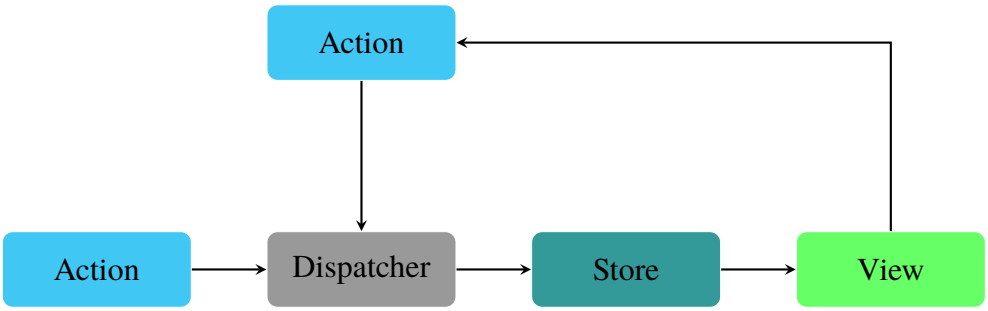

Fig. 1.3. Flux architecture diagram from Flux Concepts repository [8]

In 2015, Dan Abramov and Andrew Clark introduced Redux, hugely inspired by Flux, but they refined its architecture in a way that facilitates easier management of the application. The main change was introduction of a single centralized store. This helped centrally locate the state management in the application and approach it in a more predictable manner. The unidirectional data flow of data was used. Additionally, immutability and pure functions were really brought out with respect to how the state changed in Redux [6].

With the release of React 16.8 in 2019, Hooks were introduced by the React team as the new way in which developers could fundamentally manage state and side effects within functional components. Introduction of useState and useReducer Hooks enabled the developers to manage local component state without necessarily relying on the class components; it brought a more functional approach towards state management. It has made state management more intuitive and cut the need for large external libraries to manage state in smaller applications [18].

Modern state management libraries such as Zustand and Jotai have been created with the development of React to provide simpler and more flexible alternatives to Redux. For example, Zustand aims to be minimalistic, therefore offering a much simpler API for managing global state with less boilerplate. Jotai, in contrast, is much more atomic in nature with state handling; it allows developers to build self-reliant pieces of state that can later be composed into flexible and modular systems [27].

1.3. OVERVIEW OF EXISTING RESEARCH AND COMPARISONS

In 2022, Lorenzo Ventura made a comparison of Flutter framework's state management solutions [28]. It focuses on architecture of shared state, synchronizing it with the UI, and sharing information between components. The comparison of these approaches is conducted through conceptual analysis and practical implementation in two applications with the same functionality but of different complexity levels. Results: both Redux and BLoC introduce a lot of boilerplate, especially into complex applications. MobX reduces the boilerplate by its code generator, making it closer to basic features in Flutter. There is not much difference in performance between the solutions. The paper concluded that Redux and BLoC are powerful but boilerplate-heavy, whereas MobX is concise but potentially less predictable.

Thai Le, in the paper "Comparing State Management Between Redux and Zustand in React" [15] gets into the differences between the two most popular state management libraries in React: Redux and Zustand. Their implementation, performance, and scalability are compared using implementation of a "To-Do List" application. This approach involved setting up each library, implementing their respective state management logic, and measuring performance using tools like Chrome DevTools. The results suggest that using Redux with extensive tooling, like Redux DevTools, is in fact more performant and scalable. It makes it suitable for more complex projects. Zustand, on the other hand, is lightweight and much more comfortable to set up, making it more suitable for smaller projects. It, however, lacks sophistication around tracking and scaling around which Redux is built. The paper concludes that project requirements should be the basis upon which a choice of whether to use Redux or Zustand is made.

Thanh Le's paper "Comparison of State Management Solutions between Context API and Redux Hook in ReactJS" [16] discusses the differences and similarities between two popular state management solutions within React -Context API and Redux Hook. The study is based on the development of a portfolio application using Context API and Redux Hook, Then, they are compared under seven criteria: implementation complexity, state change tracking, additional package requirements, codebase complexity, resource consumption, processing speed, and scalability. Results have shown that the use of Context API is more appropriate and easier for smaller projects since it is already built within the structure and

less consuming in memory. On the other hand, Redux Hook shows better state change tracking, performance, and scalability -larger, more complex applications are better handled with it. Finally, the paper concludes that the choice of using either Context API or the Redux Hook should be guided by the specific needs of a project: the Context API suffices for small-scale applications, while Redux Hook is ideal for bigger and more demanding ones.

Similar findings were obtained by Daria Pronina and Iryna Kyrychenko in their "Comparison of Redux and React Hooks Methods in Terms of Performance" [21], suggesting use of React Hooks for applications where performance is critical. They found that Redux not only results in larger codebases and production bundles, but also takes longer to perform basic state operations.

1.4. SELECTED SOLUTIONS FOR COMPARISON

In the assortment of state management libraries for React, Redux, MobX, useContext, Zustand, Jotai, and Recoil are quite remarkable as they have different ways of managing state yet they are mostly preferred by many people. All of these libraries are open-source and have documentation available.

Redux is an extensive state container designed for reliability and predictability that valuably maintains immutability via a unidirectional data flow; this has been adopted in several large scale systems [12].

MobX uses a more adaptable and functional approach to managing application state. This means that when one property changes, the other related properties track observables thus updating themselves after every change [19].

State management can also be performed natively with React Hooks and **Context API** without the use of external libraries through useContext alongside useReducer making it less complicated and more self-contained especially for tiny apps [18].

Zustand, **Recoil** and **Jotai** represent modern light state management libraries improving upon the shortcomings of their predecessors. Zustand is oriented to simplicity and minimal boilerplate, Jotai is the most primitive and atomic way to manage state, and Recoil offers a brand new way of looking at managing state with fine-grained updates and further compatibility with concurrent mode [27].

These libraries were chosen for their groundbreaker approach, high adoption in the React community, and the tendency to service small and large applications alike in their complexity.

2. METHODOLOGY

2.1. TEST APPLICATION

In order to test different state managers a test application is going to be created, specifically an e-commerce store with common features like product list, filtering, searching and authentication.

2.2. METRICS FOR EVALUATION

Properly comparing state management solutions requires defining metrics. In front-end projects, there are various ways to test whether one application is better built than another one.

2.2.1. Bundle size

Bundle size is one of the most important metrics in the web development. Many Internet users do not have access to fast connection, and quick download of the application is crucial. This metric is easily testable, because each library provides information about its size. In this paper there we are going to focus on two bundle size metrics:

— Bundle size of specific library used
— Bundle size of test project with the specific library installed

Such an approach provides benefits because of the bundler's tree shaking feature. It automatically removes dead code, which is not used in actual application, so that bundle size can be compared in practical applications.

2.2.2. Performance

Measuring performance in web applications can be approached in different ways. One of the best metrics are core web vitals used often in field called search engine optimization (SEO) [5]. Excellent scores in these metrics lead to higher placement in search engines, which in turn attracts more website users.

Metrics that are going to be analyzed in this study are generated in a Lighthouse report, which is an open-source automated tool for measuring performance of web applications available in Google Chrome web browser. Production build of the application will be used for this test.

Largest Contentful Paint (LCP) is a metric that reports the render time of the largest image or text block visible in the viewport, relative to when the user first navigated to the page. It is one of the best numbers to measure website speed, which was historically difficult to determine [30].

Total Blocking Time (TBT) is a metric that measures the total amount of time after First Contentful Paint (FCP) where the main thread was blocked for long enough to prevent input responsiveness [29].

State update time

This metric directly shows responsiveness of the application. Whenever users interact with it, we can measure how long it takes to re-render the view. This is going to be measured in the following scenarios:

— Modifying a filter on the homepage
— Changing sort order on the homepage
— Saving the address on the checkout page

Test 1 is going to be performed by changing filters' settings. Firstly, the following filters are going to be set:

— Stock availability: true
— Price range: 0-250 $
— Category: Juices

Then, additional category will be added - T-Shirts.
This test is designed as a state update to a moderately-sized object.

Test 2 is going to be performed by changing sort order on the homepage from ascending to descending. No additional filters are applied. This test is designed as a state update to a small object.

Test 3 is going to be performed on checkout page by changing Street field in the address form and then saving it. Rest of the address fields should be previously set. This test is designed as a state update to a large object, which is also updated upon receiving API response.

These scenarios allow for reproducibility in each copy of the application. All of the tests are going to be performed in Google Chrome using React Profiler, which is a tool commonly used to measure performance. It is a part of React Dev Tools browser extension. **6x CPU Throttling will be applied** to simulate behaviour on lower-end devices, which are greatest impacted by application performance.

Re-renders

Re-renders in React are a fundamental part of how React maintains and updates the user interface efficiently. Understanding re-renders is crucial for optimizing performance and ensuring that applications are responsive and behave as expected. React triggers a re-render every time one of the following happens:

— State changes
— Component parent re-rerenders
— Context changes
— Hook changes

In figure 2.1 it is shown how React makes a re-render between Virtual DOM and real displayed DOM. Analysis is going to be performed on how many re-rerenders are there in each solution. If a component too high in the tree is updated too frequently, that can cause performance issues. This metric is going to be determined using React Profiler, together with state update time. Re-renders related to UI animations (e.g. ripple effects on click) are going to be ignored and not measured in the final results.

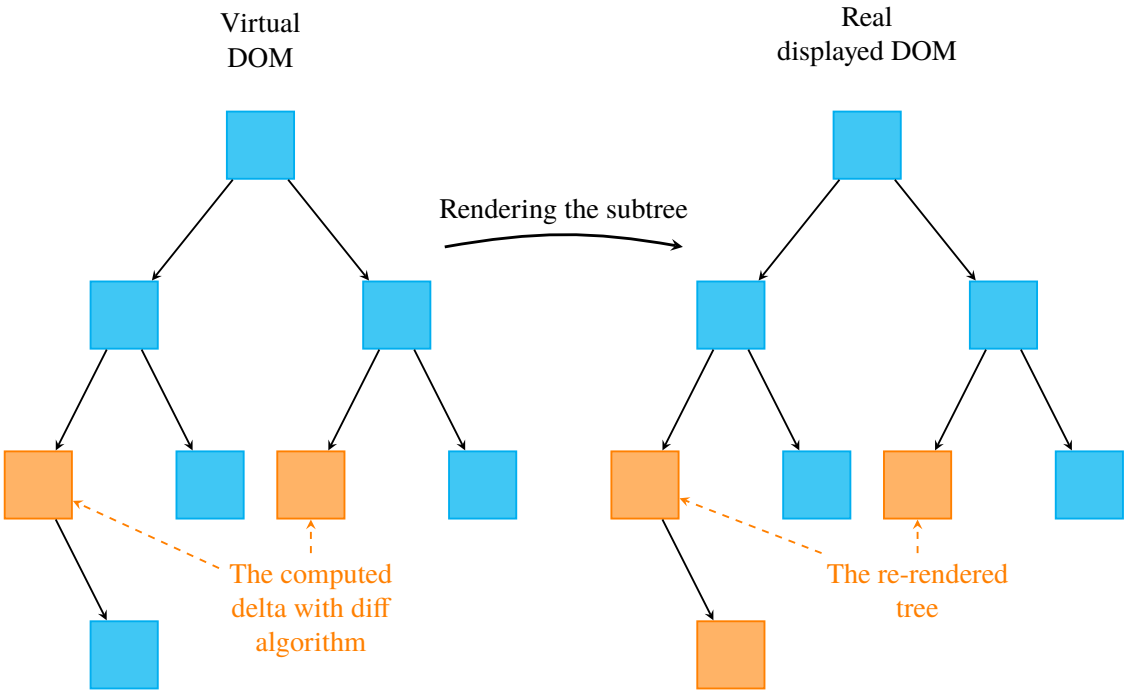

Fig. 2.1. Re-rendering of child nodes in React

2.2.3. Memory consumption

One of the most important metrics used in web applications is their memory consumption. Developers often disregard it, with the rise of interpreted languages like Python and JavaScript we tend to see more and more applications which are not memory-optimized

[14] [32]. The consumption is browser dependent. In this paper analysis is going to be performed on two most popular web browsers as of August 2024 - Google Chrome and Safari [24]. Both applications provide advanced feature called Developer tools, which allows investigating memory load. Memory consumption is going to be tested in the following scenario:

— Filters set to the same configuration as in Test 1 of state update time
— Sort order set to ascending

2.2.4. Ecosystem

Every library analyzed - Redux, MobX, useContext, Zustand, Jotai, and Recoil is unique and represents a unique way to deal with state management in React. Each of these solutions has developed its ecosystem around community support, extensions, middleware, and integration tools that it allows for extra functionality and ease of use. These should be effectively contrasted by analyzing their documentation, active vs. resolved issues, and overall popularity measured by GitHub Stars. End users can assess usability, stability, and community engagement through this lens. Libraries can also be compared by their TypeScript support.

2.3. HARDWARE

Some of the metrics (e.g. performance, memory consumption) might be hardware dependent. Experiments are going to be conducted on **Macbook M3 Pro** (14-inch, Nov 2023, 18 GB Memory) running operating system MacOS Sonoma 14.5.

2.4. API SELECTION AND JUSTIFICATION

There are multiple e-commerce APIs to consider when building an e-commerce store. In this project, after thorough analysis, Saleor was chosen. Saleor is a great choice for the e-commerce project, having its source code open for an unlimited set of customizations and flexibility. It uses a GraphQL API so you can request whatever data you want, which in turn reduces the traffic of network data and makes integrations across other libraries easy for state management. It comes pre-populated with a sample database, providing instant development and testing without the initial overhead of setting up data. Saleor is a headless commerce platform, separating the front end from its backend, hence perfectly suited for concentrating on different state management solutions. Another reason is that it comes with detailed documentation and a strong community; this is important when troubleshooting and having smooth development when working with various state management libraries [33] [22].

2.5. TOOLS AND TECHNOLOGIES USED

2.5.1. Bundler

Many frontend development tools are available these days, such as Webpack and Create React App, but Vite is the new star among them. Written by Evan You, the creator of Vue.js, and developed with performance and speed in mind, Vite provides an ultra-fast development experience for JavaScript/TypeScript apps. Also, it is platform-agnostic, meaning that it could be used with whatever framework. Other Vite features are fast compilation, hot module replacement, lazy-loading of modules, tree-shaking, and code splitting, including a built-in development server. These all add up to provide faster build times, smaller bundle sizes, and improved performance. All things considered, Vite was chosen to be the builder for this project [4].

2.5.2. TypeScript

TypeScript is a superset of JavaScript language, which enhances it by allowing static typing. It gives developers opportunity to design software of higher quality, that is easier to understand and refactor [3]. Currently, it becomes evident that TypeScript has become a standard in web development and an obvious choice for the vast majority of software projects using JavaScript [25] [23].

2.5.3. Linter

A linter is a static analysis tool that is used to warn software developers about possible errors or bad practices. In this study, ESLint was used to impose some coding standards and reduce early potential errors in development, ultimately minimizing the cost and effort of fixing issues at later stages. ESLint is the most widely used JavaScript linter, offering a powerful tool with both default and customizable rules for quality and consistency in coding. In addition, ESLint is highly configurable; hence, we can tune the linting process in the best possible way for our project, where the most relevant coding standards and practices will be applied while avoiding common pitfalls [26].

2.5.4. Routing

Most of front-end applications require routing, in order to accommodate native browser features like history. It also allows sharing a hyperlink to a particular page of an application, which is a pitfall of Single-Page Applications (SPA). In this study, React router has been used which allows handling routes declaratively [11].

2.5.5. Data fetching layer

In order to communicate with Saleor's GraphQL API, there is a need to use some library responsible for data fetching. In theory, it is possible to make GraphQL calls using browser's native Fetch API. In practice, it is not common, because we lose many of the quality of life features that are provided by using GraphQL in the first place. In this study, `urql` is used, which is a minimalistic alternative to large GraphQL libraries like Apollo [31]. Together with `graphql-codegen` package, it allows us to generate React hooks from GraphQL queries and Saleor GraphQL schema with a single command, greatly reducing development cost of the application.

2.5.6. UI

User interface is likely one of the most important aspects of front-end applications. They are designed to be pretty, easy to use and provide great user experience (UX). An arbitrary choice was made to use Material UI component library, which is a popular option for developers. It provides many components used in typical e-commerce stores (sliders, buttons, navigation). Thanks to it, project scope can be reduced, by eleminating the need to write custom CSS. Material UI also has great documentation, helping incorporate it in a project of any size and purpose [17].

2.5.7. Forms

Some projects might opt for using a dedicated form library. There are many to choose from, like react-hook-form or Formik, which simplify validation and ease of form creation. In this project, there are not many forms to implement - the largest one is the address form on the checkout page. Having considered that, a decision was made to not include such a library. Consequently, this creates a smaller bundle size.

3. E-COMMERCE STORE IMPLEMENTATION

3.1. COMMON BOILERPLATE

Firstly, the project was build disregarding any of the state updates. It consists of the following features:

— Product list
— Collection list
— Filtering
— Searching
— Sorting
— Authentication
— Checkout flow

Material UI has provided components which were used to create user interface of the application, which can be seen in figure 3.1.

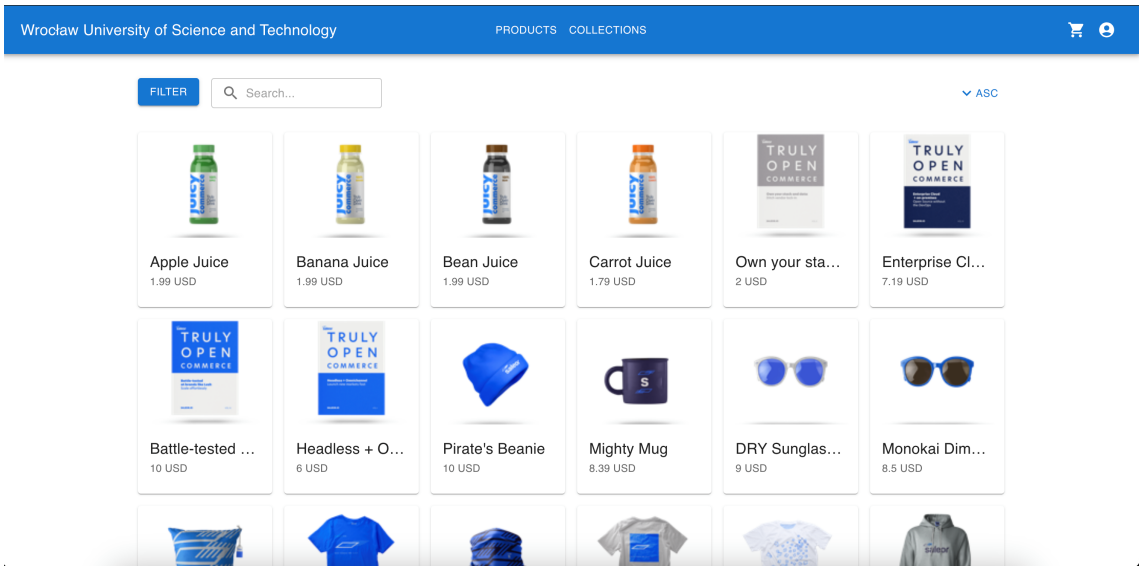

Fig. 3.1. Homepage of the application

3.2. SETUP

3.2.1. Running Saleor API

In order to start to application, one has to have running Saleor API. The easiest way is to install `saleor-platform` which is a dockerized package containing core API, Saleor Dashboard and other necessary services like the PostgreSQL database. After cloning the repository series of docker commands are required which are shown in listing 3.1. First

```
docker compose build
docker compose run --rm api python3 manage.py migrate
docker compose run --rm api python3 manage.py populatedb --createsuperuser
docker compose up
```

Listing 3.1: Docker commands required to run Saleor Platform

command is used to build the docker image. Then, we apply Django migrations. Third command creates an admin user with credentials `admin@example.com` and `admin`. Finally, we run the application with the fourth command.

GraphQL API is available by default on `http://localhost:8000/graphql/` and Saleor Dashboard is reachable on `http://localhost:9000/`.

3.2.2. Store configuration

Some configuration is required for particular features. In order for user registration to work, e-mail confirmation for new users has to be disabled. It can be done via Saleor API using `shopSettingsUpdate` mutation or more conveniently in Saleor Dashbord by unchecking the box "Require email confirmation view", as visible in figure 3.2. This view can be accessed via Configuration (on the sidebar) $\rightarrow$ Site settings.

In order to complete checkouts, payment application has to be configured. Although not necessary to perform experiments in this paper, `saleor/dummy-payment-app` can be used to make checkouts fully functional. This app can be run locally and exposed to web via a tunneling service, for example `ngrok`. Then, the app can be installed in Saleor Dashboard in the app view (reachable from the sidebar). Manifest URL has to be provided which in this case is going to look like `{tunnel_url}/api/manifest`. After successful installation Dummy Payment App should be visible as one of the payment gateways available in the checkout.

3.3. ROUTING & PAGES

Thanks to react-router, application has ease of navigation, allowing possibility to access specific page by URL. Component tree is defined in the App.tsx file (3.2).

```
const App: React.FC = () => {
  return (
    <Box sx={{ display: "flex", flexDirection: "column", minHeight: "100vh"
    ↪  }}>
      <Router>
        <CssBaseline />
        <Navbar />
        <Box sx={{ flex: 1 }}>
          <Routes>
            <Route path="/" element={<Homepage />} />
            <Route path="/product/:productId" element={<ProductPage />} />
            <Route
              path="/collection/:collectionId"
              element={<CollectionPage />}
            />
            <Route path="/checkout" element={<CheckoutPage />} />
            <Route path="/user" element={<UserPage />} />
            <Route path="/login" element={<LoginPage />} />
            <Route path="/register" element={<RegisterPage />} />
            <Route path="/reset-password" element={<PasswordResetPage />}
            ↪  />
          </Routes>
        </Box>
        <Footer />
      </Router>
    </Box>
  );
};
```

Listing 3.2: Routes in the application

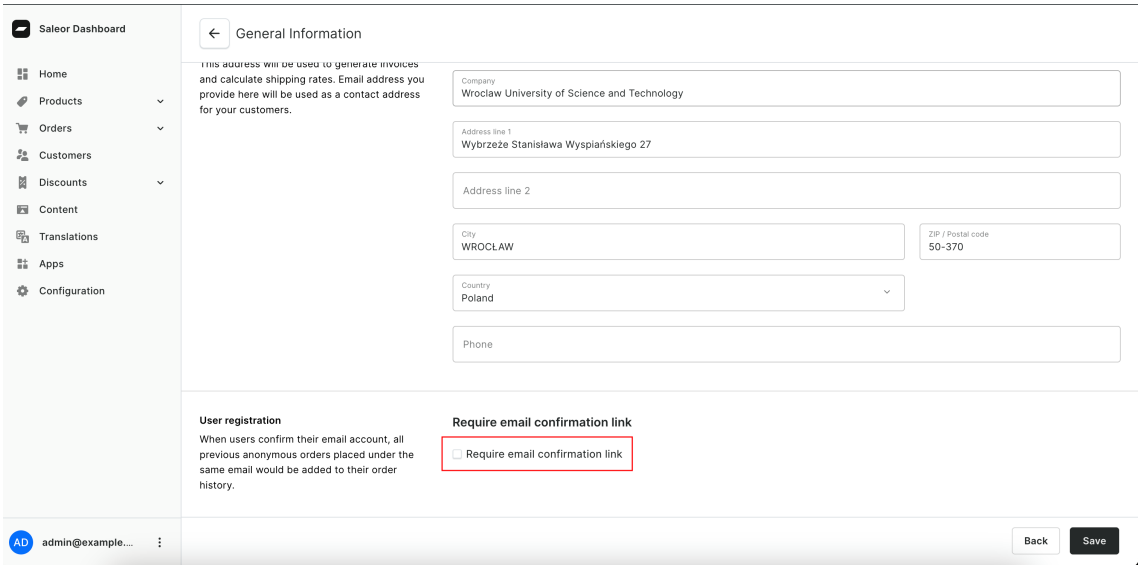

Fig. 3.2. Disabling user email confirmation in Saleor Dashboard

3.4. DATA FETCHING

In order to fetch data from Saleor API, the project requires GraphQL Client. In theory, it would be possible to write it by hand using browser's native Fetch API, however, it is impractical. GraphQL allows developers to create TypeScript interfaces using automated tooling, which greatly reduces boilerplate and development cost. In this project `urql` was used together with `graphql-codegen`. This tooling allows automated hook creation of queries and mutations using build command `npm run generate`. In the `/graphql` directory, fragment, query or mutation is specified. One example is the categories query

```
query GetCategories {
  categories(first: 30) {
    edges {
      node {
        id
        name
      }
    }
  }
}
```

Listing 3.3: GraphQL Query for fetching categories

(3.3). It makes a call to Saleor API and retrieves categories defined in e-commerce store. It contains `first` parameter which specifies how many categories should be fetched. `edges` and `node` are typically used to help with pagination - GraphQL clients can be easily configured to fetch additional page of data (in this case, next 30 categories).

Thanks to libraries used, writing such a query auto-generates `useGetCategoriesQuery` hook which is ready to use in React components, as seen in listing 3.4.

```
const [{ data: dataCategories, fetching, error }] =
↪  useGetCategoriesQuery();
const categories = dataCategories?.categories?.edges ?? [];
```

Listing 3.4: Auto-generated custom hook used to retrieve data from the API

We can use `dataCategories` object to display the data, as well as `fetching` and `error` fields for loading states and error handling respectively.

3.5. AUTHENTICATION

Authentication in Saleor is JWT-based (JSON Web Tokens signed with RS256). After executing a mutation with valid login and password, API returns access token and refresh token. Access tokens are short-lived and require refreshing often. This is handled by `@urql/exchange-auth` library, but custom logic has been written for creating `urql` client. It works by detecting GraphQL "Expired signature" error. After the error is received, another mutation for token refresh is called and access token is replaced. The tokens are stored in local storage, to avoid losing authentication on website refresh, however this logic is specific to a particular state manager library.

Available actions

Users are able to:

— Register
— Log in
— Log out
— Reset password

Registering and logging in are handled by `accountRegister` and `tokenCreate` mutations respectively. Logging out does not use additional mutations, tokens are simply cleared from local storage and state. Users are also able to reset password using `passwordChange` mutation as depicted in figure 3.3. Saleor has been configured so that it accepts accounts without additional e-mail verification in order to avoid unnecessary complexity during authentication process.

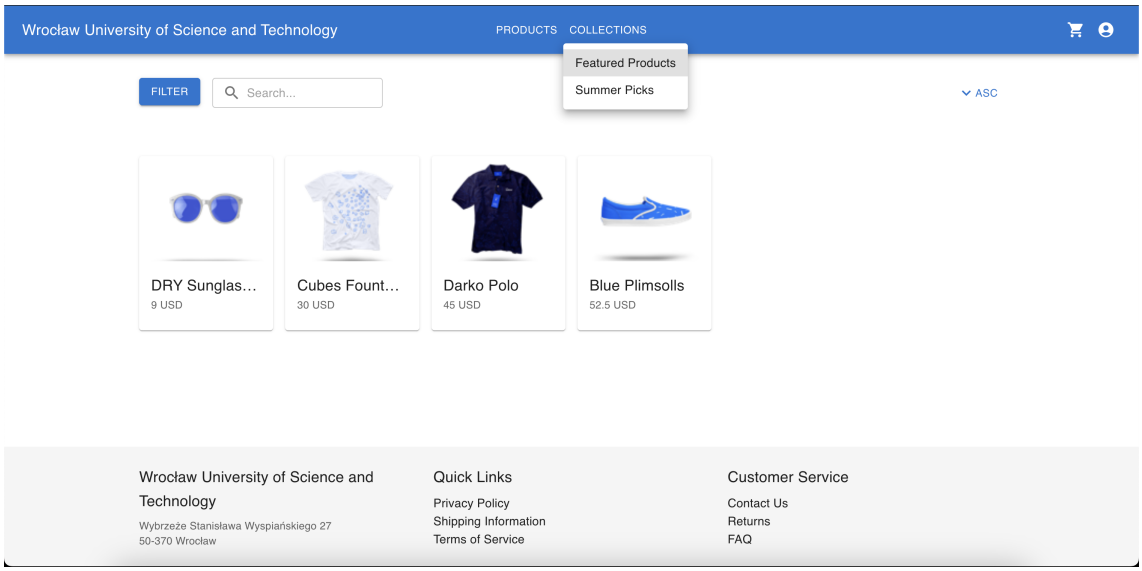

Fig. 3.3. Password reset page

3.6. PRODUCT LIST

The appplication contains a navbar with products and collection routes. Navigating between them does not modify filter, search or sort state, because it is shared. Example of collection page can be seen in 3.4

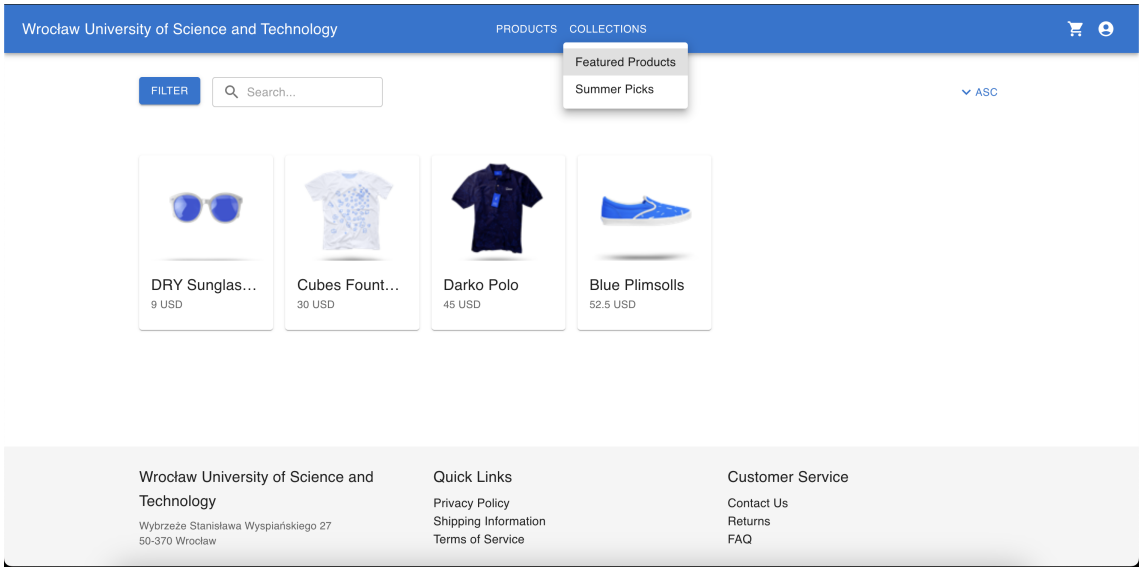

Fig. 3.4. Product collections

3.6.1. Filtering

Filtering is handled by Material UI Popover component. It is possible to filter by category, stock availability, price range and custom attributes of the products (which are specified in the API - can be changed programmaticaly or in Saleor Dashboard). Design

of the filters is visible in figure 3.5. Modifying the filters changes `$where` input on the `products` query, which returns appropriate items from the store. Filters are shared across home and collection views, which make them ideal candidate for state to be handled by one of the management libraries.

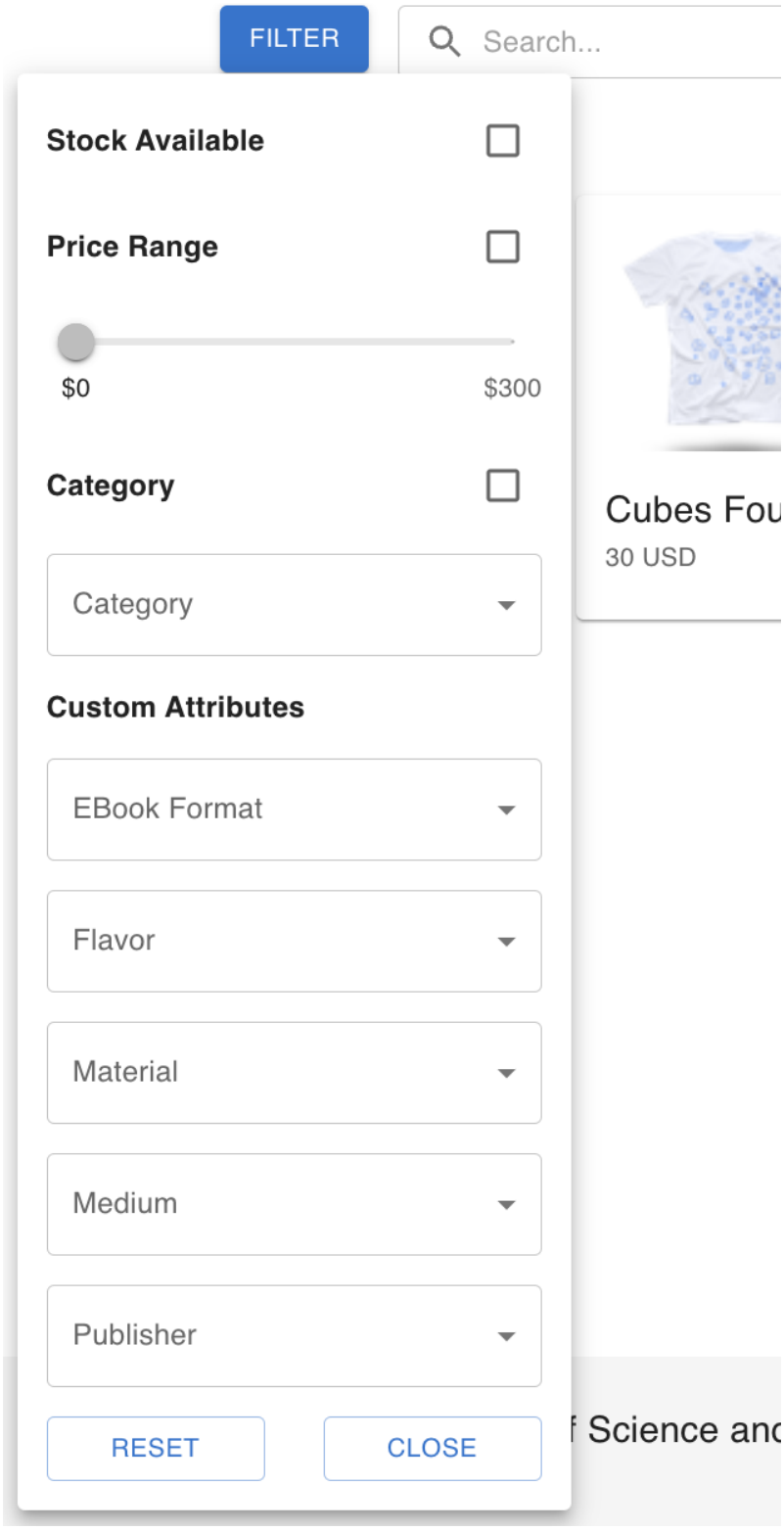

Fig. 3.5. Filter component

3.6.2. Searching

Searching is possible with the use of Material UI TextField component. It is shared across home and collection views, so similarly to filters it can be held in state manager. Changing the query makes an API call to Saleor with newly set `$search` variable.

3.6.3. Sorting

Sorting is possible with Button component in top right button visible in figures 3.4 and 3.1. Clicking on the button changes the parameter in product or collection query (depending on the view). Products are sorted by price in either ascending or descending order. What is important, sort state is reset every time search state is changed, because whenever user searches through products, they are sorted by rank (best match is displayed first). It is impractical to sort products from worst match to best, so clicking on sorting button whenever the search is active resets the search state. Popover component visible on hover has been added to inform user about this, which is visible in figure 3.6. Sort state is persisted across home and collection views.

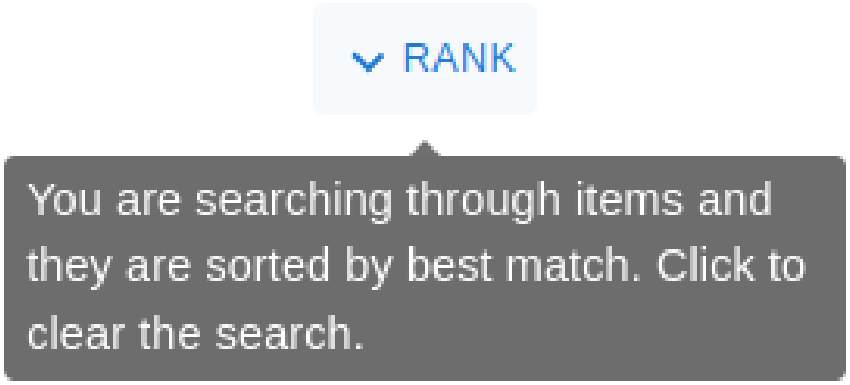

Fig. 3.6. State of sort field when user searches through products

3.7. CHECKOUT

Adding products to cart

When clicking on a product card from either homepage or collection page, users are redirected to product page visible in figure 3.7.

From here, user is able to select product variant and add it to cart. Checkout logic in Saleor allows for anonymous orders, so no login is required. Firstly, it is checked if checkout had been previously created and saved in local storage. If so, item is added to

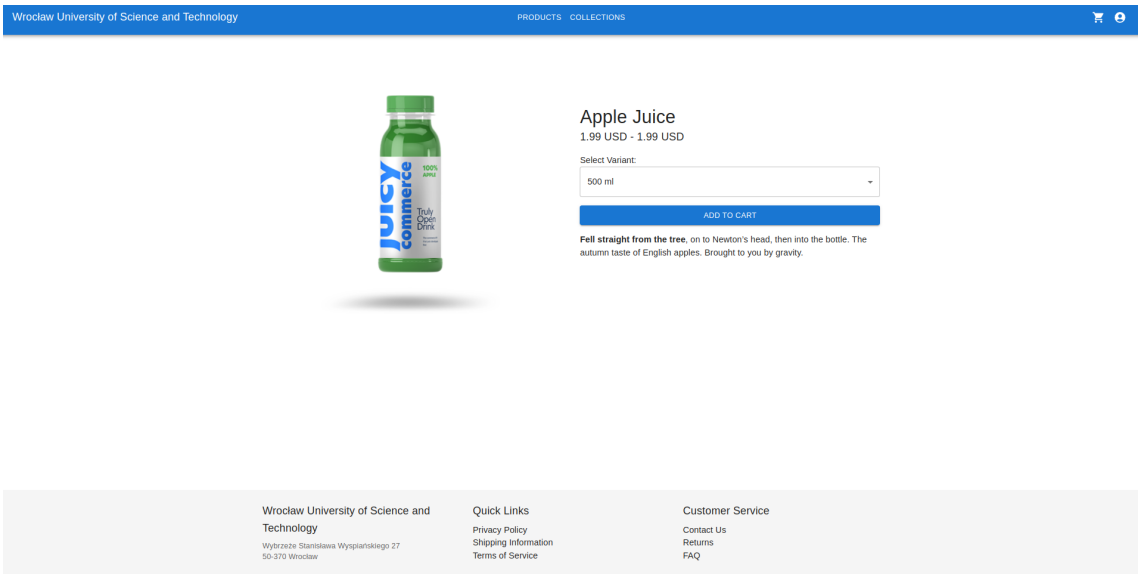

Fig. 3.7. Product page, visible after clicking on product card

checkout using `checkoutLinesAdd` mutation. Otherwise, if there is no checkout or it has expired, a new checkout is created and saved in LS using `checkoutCreate` mutation. This flow is depicted in figure 3.10.

Finalizing the checkout

After the user is ready to make an order, they can proceed to checkout by clicking on shopping cart icon visible in top-right corner. This view is depicted in figure 3.8.

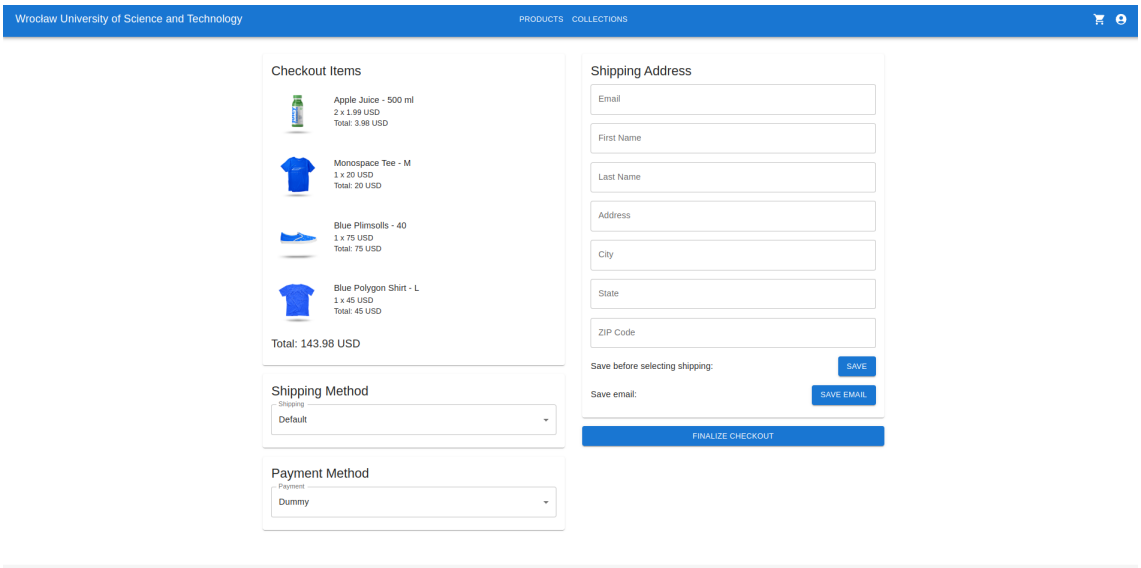

Fig. 3.8. Product page, visible after clicking on product card

In order to create an order, address has to be set. E-mail has to be provided, unless the user is logged in - in this case it is optional. After selecting shipping method and Dummy

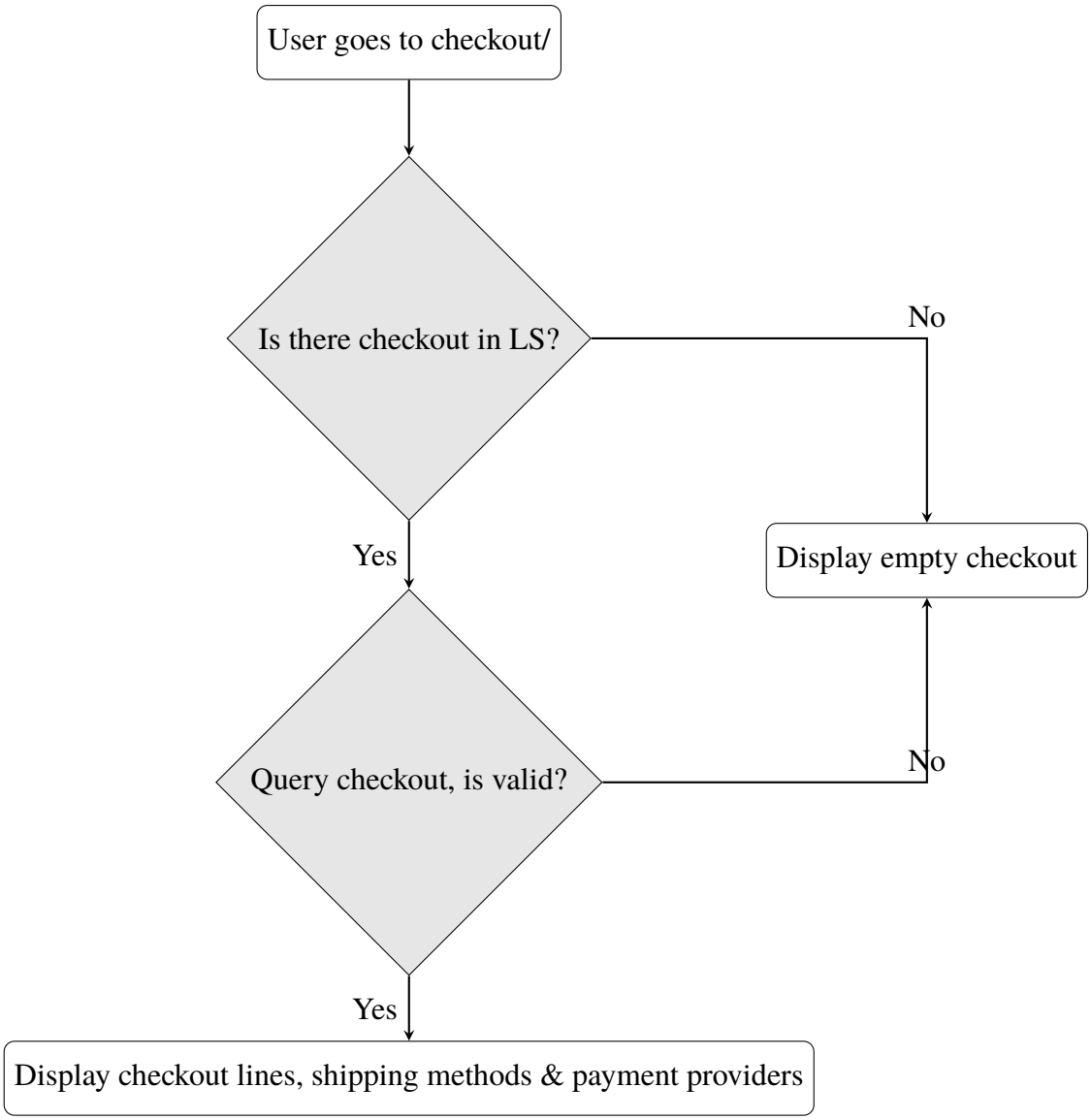

Fig. 3.9. Behaviour of the application after adding product to cart

Payment App as payment provider, checkout can be finalized and transformed into order visible in Saleor Dashboard. Flow of the checkout view is presented in figure 3.9.

3.8. STATE DELEGATED TO LIBRARIES

The boilerplate was written without state management. This means features don't work, because they rely on it. This part is delegated to each state management library. Parts of the projects for which state solutions are responsible are shown in the diagram 3.11.

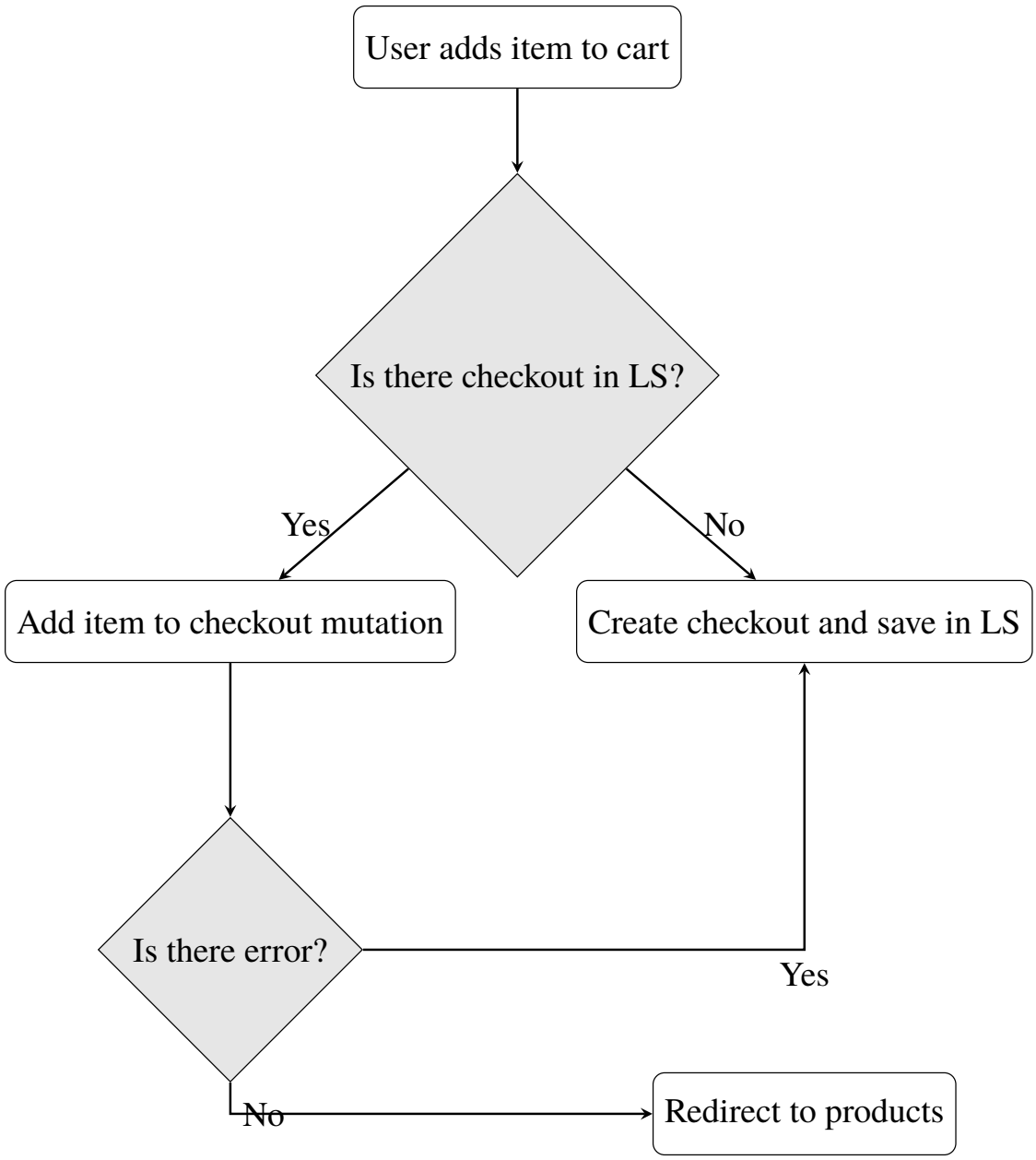

Fig. 3.10. Behaviour of the application after adding product to cart

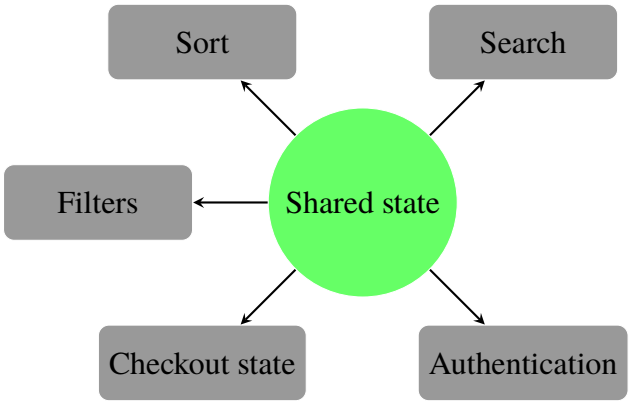

Fig. 3.11. Features delegated to state management library

4. EXPERIMENT

4.1. REDUX

4.1.1. Overview of Redux

Redux is a JavaScript library released in 2015 by Dan Abramov. The first introduction was in his speech at React Europe, "Live React: Hot Reloading with Time Travel" [1]. This library was created to solve certain problems of state maintenance within complicated JavaScript applications, in particular with regard to those built using React. Redux aimed to be predictable in changing the state of an app so that the state could only change in well-defined ways, making it easier to read and debug.

The main features of Redux include its single source of truth and that this state cannot be changed directly; in general, actions describe changes in plain JavaScript objects. The changes are then processed by pure functions called reducers, which receive the current state and an action as arguments and return a new state without modifying the given one. This brings unidirectional data flow into an application and makes it much easier to manage its state, which were the concepts introduced by Flux architecture. It is for the reason of such apparent simplicity of Redux and accompanying them with developer tools of unbelievable power that the framework became so widely spread for managing states inside applications [12]. Current version of Redux Toolkit is 2.2.7.

4.1.2. Implementation details

Redux is implemented using slices, which are part of a single central store. Example implementation of the search slice is present in listing 4.1.

This slice can be later accessed using redux selector logic, for example in the search component, which is shown in listing 4.2.

4.1.3. Results

Bundle size

Bundle size was measured for two libraries installed: `@reduxjs/toolkit` and `react-redux` which are 5.46 MB and 740 kB respectively, as seen in table 4.1.

Performance and re-renders

Lighthouse results for Redux implementation can be seen in figure 4.1.

```typescript
import { createSlice, PayloadAction } from "@reduxjs/toolkit";

interface SearchState {
  query: string;
}

const initialState: SearchState = {
  query: "",
};

const searchSlice = createSlice({
  name: "search",
  initialState,
  reducers: {
    setSearchQuery: (state, action: PayloadAction<string>) => {
      state.query = action.payload;
    },
    resetSearchQuery: (state) => {
      state.query = "";
    },
  },
});

export const { setSearchQuery, resetSearchQuery } = searchSlice.actions;

export const selectSearchQuery = (state: SearchState) => state.query;

export default searchSlice.reducer;
export type { SearchState };
```

Listing 4.1: Search slice implementation in Redux

```typescript
  const dispatch = useDispatch();
  const searchQuery = useSelector((state: RootState) =>
  ↪  state.search.query);

  const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>)
  ↪  => {
    dispatch(setSearchQuery(event.target.value));
  };
```

Listing 4.2: Updating search query in Redux

Tests 1-3 for performance were repeated 5 times and average time of render time was

Libraries size	Project bundle size
6.2 MB	552 kB

Table 4.1. Comparison of Libraries Size and Project Bundle Size for Redux

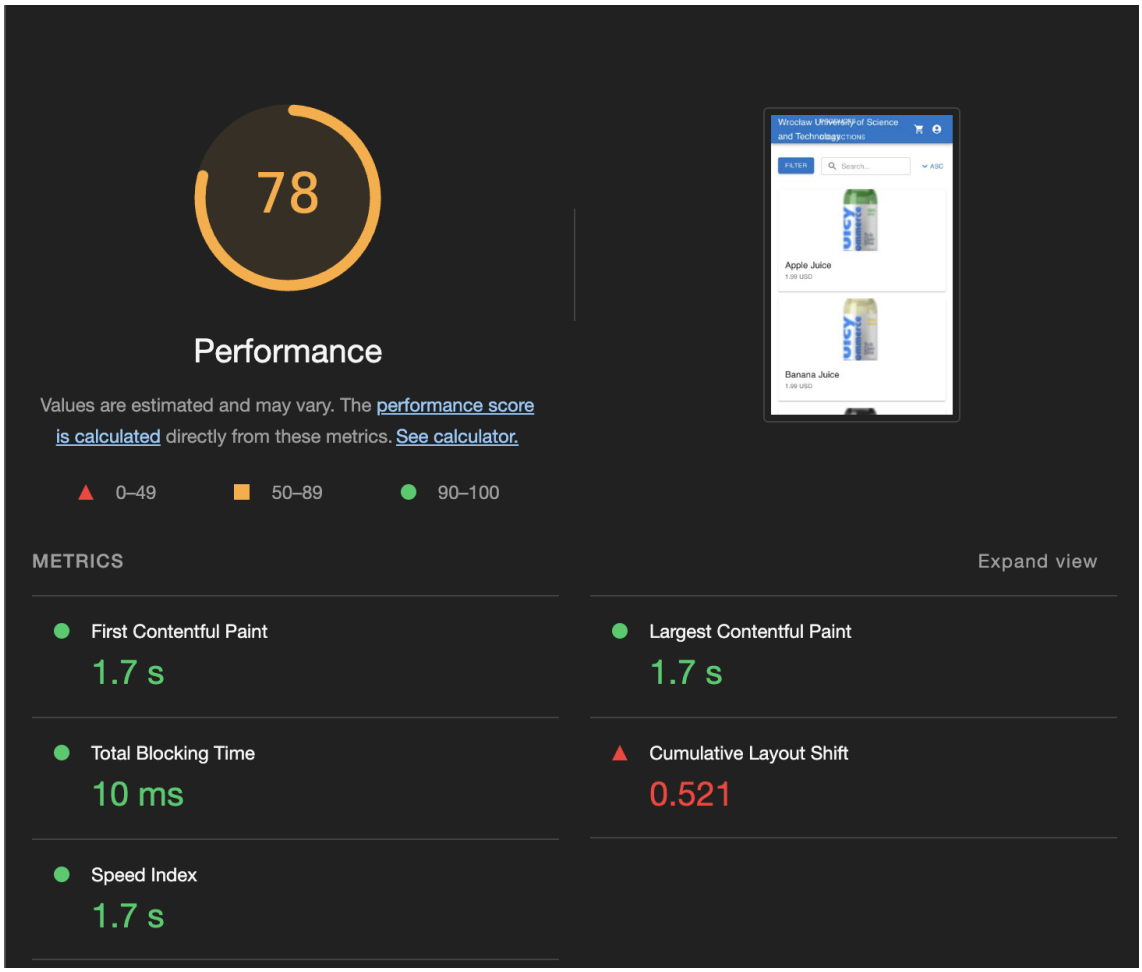

Fig. 4.1. Lighthouse scores for Redux implementation

calculated. Render times are displayed in table 4.2. Example profiler results can be seen in figure 4.2.

Scenario	Average Time ± Uncertainty (ms)	Rerenders
Test 1	110.8 ± 7.0	1
Test 2	67.4 ± 6.0	1
Test 3	420.0 ± 20.0	9

Table 4.2. Average state update time and re-renders for Redux

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.3

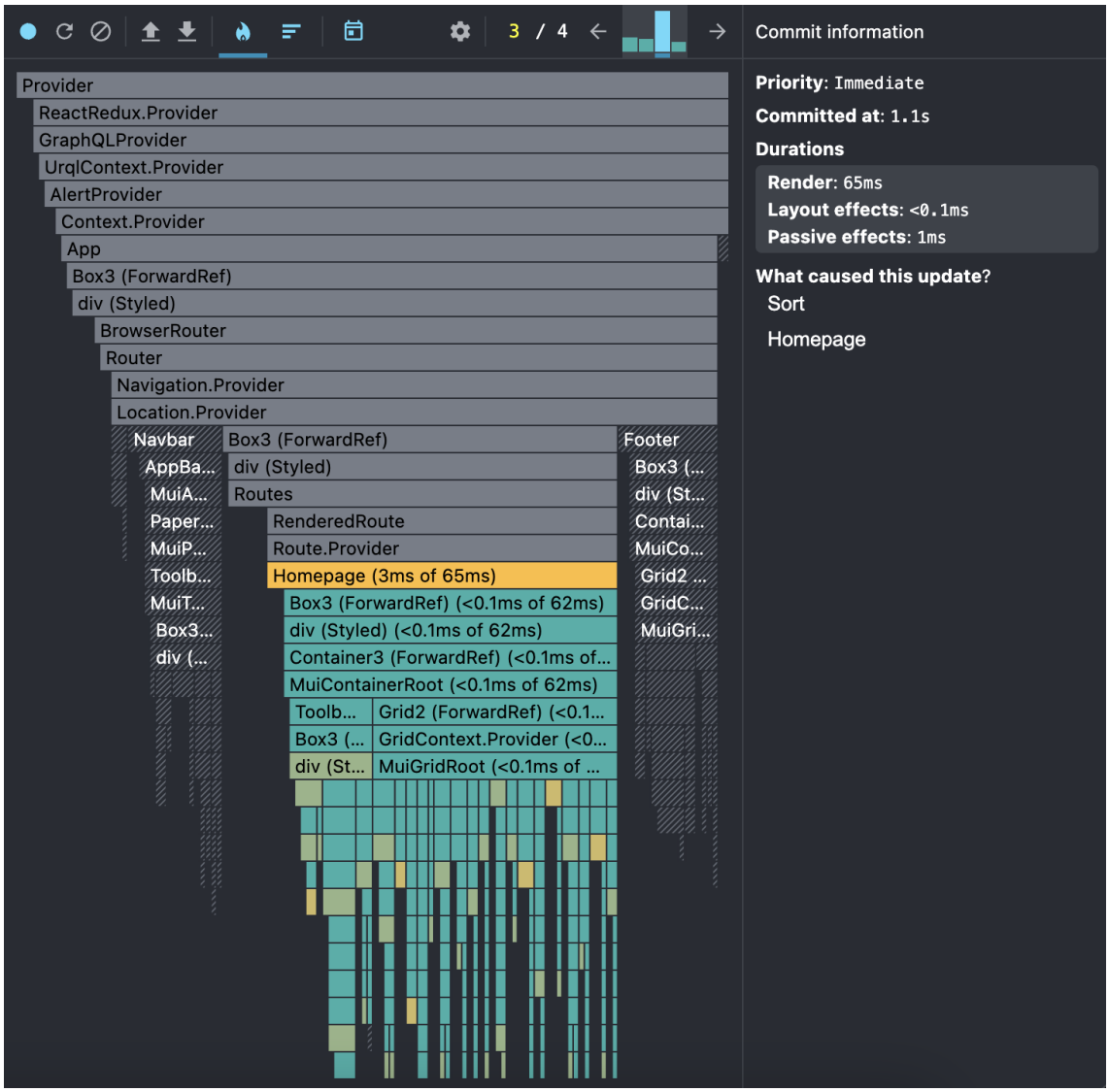

Fig. 4.2. React Profiler after Test 2 in Redux implementation

Browser	Memory Usage (MB)
Safari	21.37
Chrome	7.59

Table 4.3. Memory Usage for Redux

Ecosystem

Redux popularity might be difficult to measure because of two libraries that it consists of. Comparison is made on base library, not its bindings to React. Its GitHub statistics can be seen in table 4.4. Redux has extensive documentation, due to its maturity. It is beginner-friendly, but covers many edge cases and more advanced topics. In terms of TypeScript, Redux has excellent support. Libraries provide various templates and utilities which allow easier creation of reducers, actions and store configurations.

Stars	Active Issues	Closed Issues	Commits in last year
60.7k	37	2008	215

Table 4.4. GitHub statistics for Redux as of 08-08-2024

4.2. MOBX

4.2.1. Overview of MobX

Another strong state management library is MobX. Created by Michel Weststrate and firstly released in 2014, it implements the reactivity programming paradigm for the management of client state in JavaScript applications, in particular those built with React. In fact, MobX will allow you to create observable state which would automatically track changes and propagate them to any observers, like UI components. This reactivity makes state management much simpler and more intuitive, with less opportunity for error than classical state management with Redux. MobX introduces what it calls "automatic derivations," in that anything that can be derived from the state should be done automatically. This gets rid of the concept of boilerplate code and reduces common errors, leading to improved maintainability of the code. It gives an elegant API in order to manage observables, actions, and reactions so that they create a unidirectional data flow, keeping the UI in sync with the underlying state in a performant and scalable way. It has been designed with minimalism but also power in mind and is easy to learn and integrate into any JavaScript project [19]. Current version of MobX is 6.13.1.

4.2.2. Implementation details

Contrary to Redux, MobX does not provide any hooks. There are multiple stores instead of slices and they are directly imported in components. In order for component to re-render based on external change in the store, `observer` utility function is used, from MobX React bindings library called `mobx-react-lite`. Stores are implemented as classes, which might be considered a drawback for JavaScript developers who prefer to use functional programming style instead of object-oriented. Example store (search) is displayed in listing 4.3. We also use `makeAutoObservable` function, which automatically infers all class fields and makes them observable.

After component is wrapped in `observer` util, it can use values from the directly imported store, which is shown in listing 4.4. If the component does not display any data from the store and just makes the updates, wrapping it in `observer` is not required.

```
import { makeAutoObservable } from "mobx";

class SearchStore {
  query: string = "";

  constructor() {
    makeAutoObservable(this);
  }

  setSearchQuery(query: string) {
    this.query = query;
  }

  resetSearchQuery() {
    this.query = "";
  }

  get searchQuery() {
    return this.query;
  }
}

const searchStore = new SearchStore();
export default searchStore;
```

Listing 4.3: Search store implementation in MobX

```
import { observer } from "mobx-react-lite";
import searchStore from "../../mobx/searchStore";

export const Search: React.FC = observer(() => {
  const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>)
  ↪  => {
    searchStore.setSearchQuery(event.target.value);
  };
  /** ... Rest of the component  */
});
```

Listing 4.4: Updating search query in MobX

4.2.3. Results

Bundle size

Bundle size was measured for two libraries installed: `mobx` and `mobx-react-lite` which are 4.31 MB and 408 kB respectively, as seen in table 4.5.

Libraries size	Project bundle size
4.718 MB	593 kB

Table 4.5. Comparison of Libraries Size and Project Bundle Size for MobX

Performance and re-renders

Lighthouse results for MobX implementation can be seen in figure 4.3.

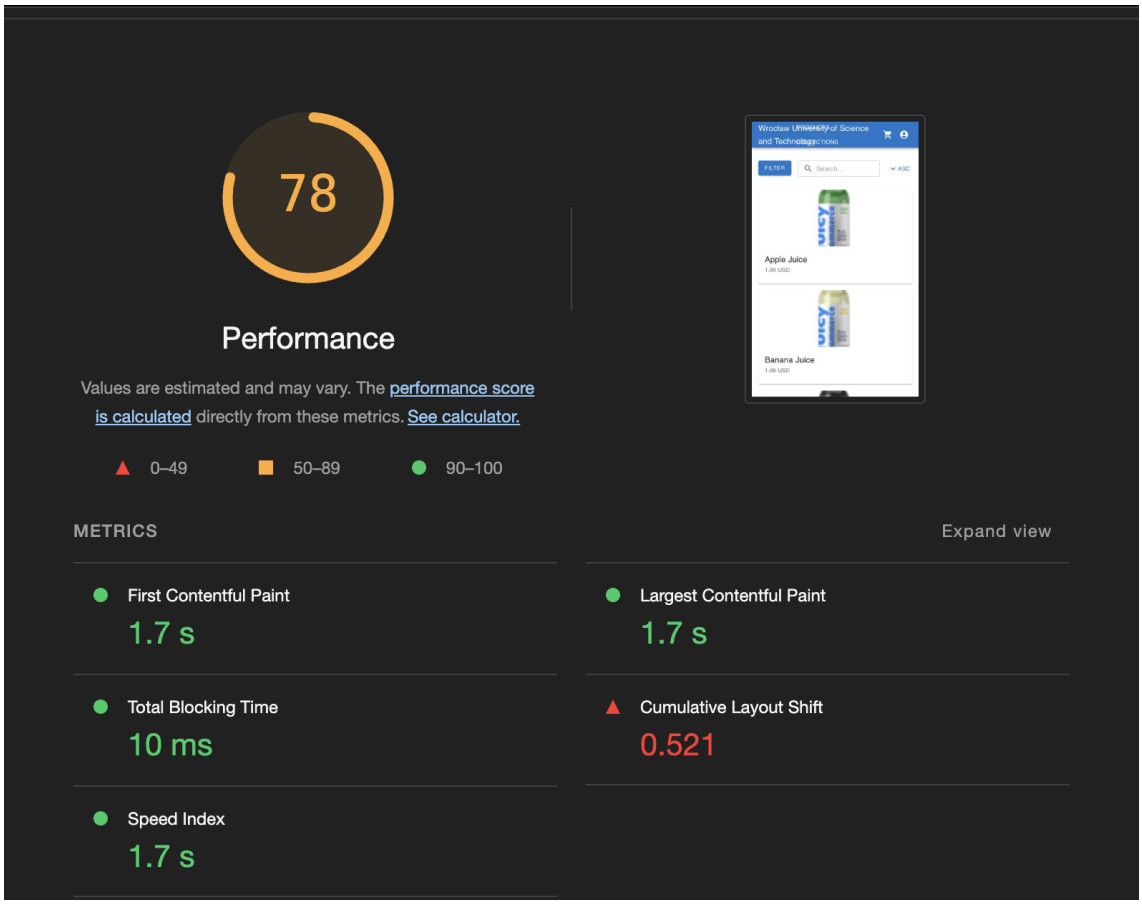

Fig. 4.3. Lighthouse scores for MobX implementation

Tests 1-3 for performance were repeated 5 times and average time of render time was calculated. Render times are displayed in table 4.6. Example profiler results can be seen in figure 4.4.

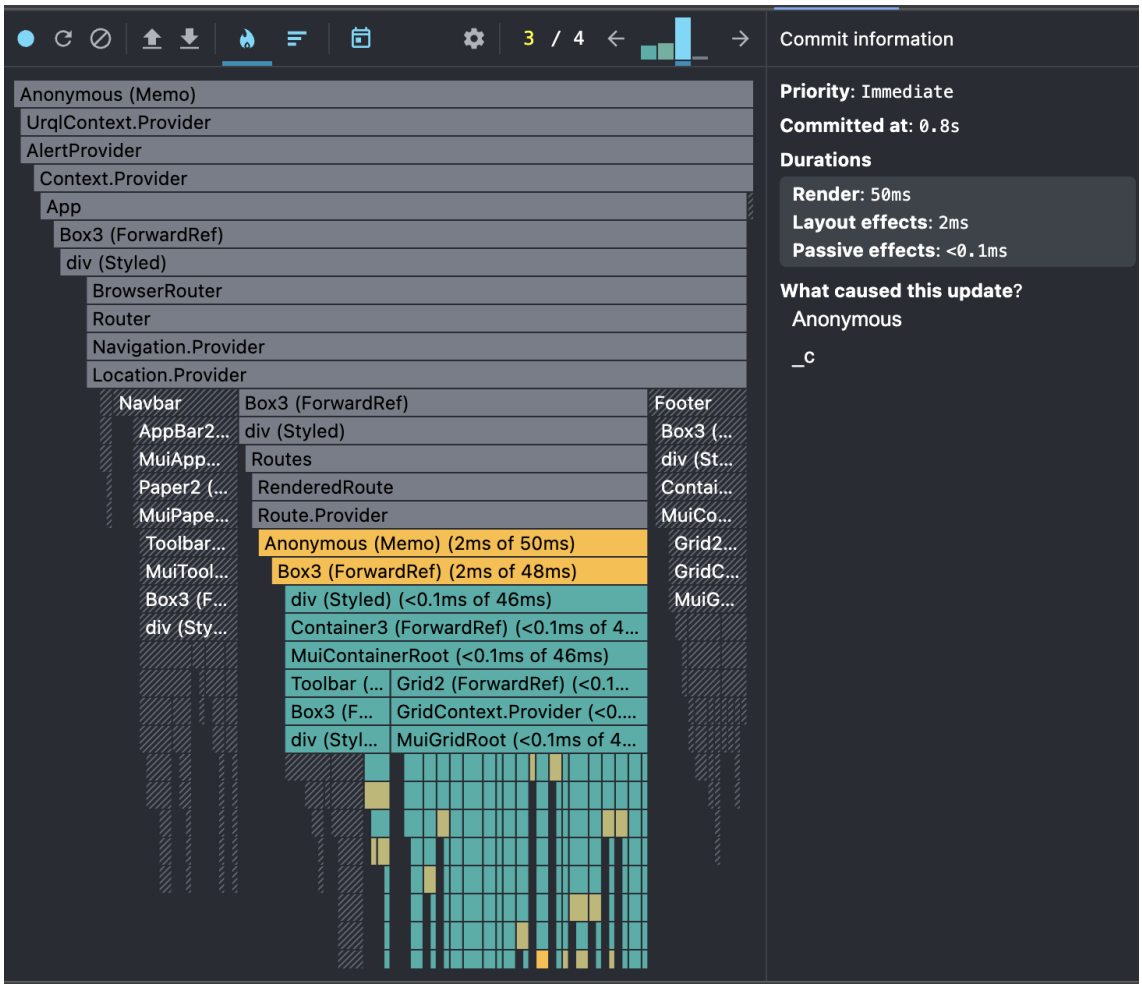

Fig. 4.4. React Profiler after Test 2 in MobX implementation

Scenario	Average Time ± Uncertainty (ms)	Rerenders
MobX Test 1	114.6 ± 3.0	1
MobX Test 2	57.8 ± 6.0	1
MobX Test 3	426.2 ± 20.0	9

Table 4.6. Average state update time and re-renders for MobX

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.7.

Browser	Memory Usage (MB)
Safari	23.97
Chrome	7.72

Table 4.7. Memory Usage for MobX

Ecosystem

It might be difficult to measure MobX's popularity, because similarly to Redux it consists of core library itself and additional bindings for React. GitHub statistics for core library can be seen in table 4.8. It offers documentation that is both concise and friendly to novices. On some advanced topics of fine-tuning, it is extremely exhaustive, for instance, on performance optimization with decorators. Strong TypeScript definitions are good measure of the TypeScript support. Utilities help make type safety very strong across observables, computed values, and reactions, so that it can be a reliable choice for any developer who is looking for a reactive state management solution.

Stars	Active Issues	Closed Issues	Commits in last year
27.4k	46	1899	68

Table 4.8. GitHub statistics for MobX as of 08-08-2024

4.3. CONTEXT API

4.3.1. Overview of Context API

The Context API for React provides a built-in way to manage global state in React applications without prop drilling by sharing state across components without the need to pass props manually. Different from third-party libraries like Redux or MobX, the Context API does not need more setup or packages, so it is really easy and intuitive for small or moderately complex apps. While both Redux and MobX come with powerful, large-scale application features such as middleware and fine-grained reactivity, they come with increased complexity and boilerplate. On the other hand, the Context API is pretty much for simple cases in order to have a lightweight, native, and wholly within-the React-ecosystem way to manage state [2].

4.3.2. Implementation details

React Context allows use to use providers, which can be used over the component that consume the context, however in this comparison there is a need for global solution. This leads to more complex provider tree in `main.tsx` file apart from GraphQL and Alert providers which are present in other implementations. This is shown in listing 4.5. Stores are replaced with contexts, reducers with native React hooks - `useContext` and `useReducer`. Example implementation of search context is shown in listing 4.6.

After contexts are properly set up, we can consume them and use reducers to update the state in the components. Example of this in the search component is shown in listing 4.7.

```
ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <AuthProvider>
      <CheckoutProvider>
        <FilterProvider>
          <SearchProvider>
            <SortProvider>
              <GraphQLProvider>
                <AlertProvider>
                  <App />
                </AlertProvider>
              </GraphQLProvider>
            </SortProvider>
          </SearchProvider>
        </FilterProvider>
      </CheckoutProvider>
    </AuthProvider>
  </React.StrictMode>
);
```

Listing 4.5: Providers used in Context API implementation.

4.3.3. Results

Bundle size

Bundle size for context is measured only for the project size, because we do not use any additional libraries, which is displayed in the table 4.9.

Libraries size	Project bundle size
0 MB	533 kB

Table 4.9. Comparison of Libraries Size and Project Bundle Size for Context API

Performance and re-renders

Lighthouse results for Context API implementation can be seen in figure 4.5.

Tests 1-3 for performance were repeated 5 times and average time of render time was calculated. Render times are displayed in table 4.10. Example profiler results can be seen in figure 4.6.

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.11.

```tsx
import React, { createContext, useReducer, useContext, ReactNode } from
↪  "react";

interface SearchState {
  query: string;
}

interface SearchAction {
  type: "SET_SEARCH_QUERY" | "RESET_SEARCH_QUERY";
  payload?: string;
}

const initialSearchState: SearchState = {
  query: "",
};

const SearchContext = createContext<
  { state: SearchState; dispatch: React.Dispatch<SearchAction> } |
  ↪  undefined
>(undefined);

const searchReducer = (
  state: SearchState,
  action: SearchAction
): SearchState => {
  switch (action.type) {
    case "SET_SEARCH_QUERY":
      return { ...state, query: action.payload ?? "" };
    case "RESET_SEARCH_QUERY":
      return initialSearchState;
    default:
      return state;
  }
};

export const SearchProvider = ({ children }: { children: ReactNode }) => {
  const [state, dispatch] = useReducer(searchReducer, initialSearchState);
  return (
    <SearchContext.Provider value={{ state, dispatch }}>
      {children}
    </SearchContext.Provider>
  );
};
```

Listing 4.6: Search context implementation in Context API.

```
const { state: searchState, dispatch: searchDispatch } = useSearch();

const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>)
↪  => {
  searchDispatch({ type: "SET_SEARCH_QUERY", payload: event.target.value
  ↪  });
};
```

Listing 4.7: Usage of search context

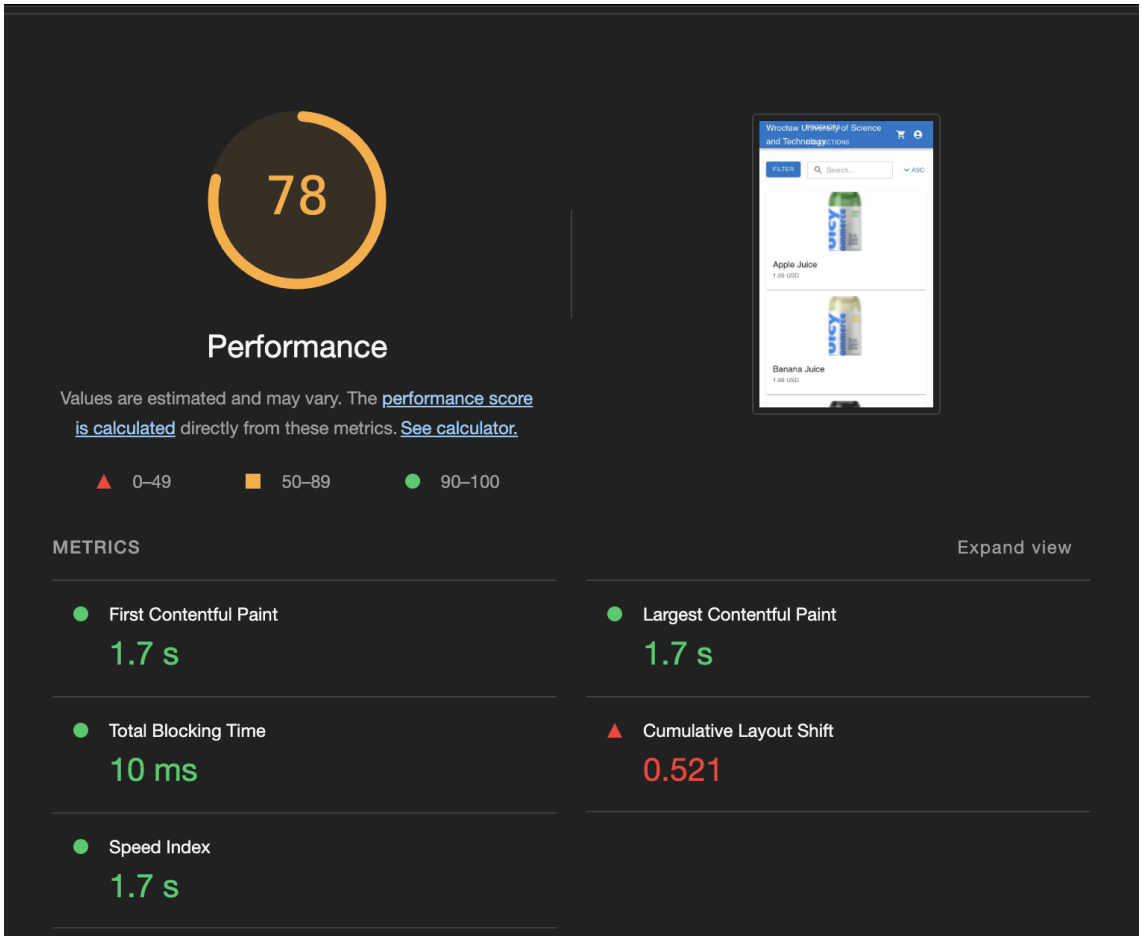

Fig. 4.5. Lighthouse scores for Context API implementation

Scenario	Average Time ± Uncertainty (ms)	Rerenders
Test 1	115.0 ± 5.0	1
Test 2	64.8 ± 7.0	1
Test 3	417.0 ± 20.0	9

Table 4.10. Average state update time and re-renders for Context API

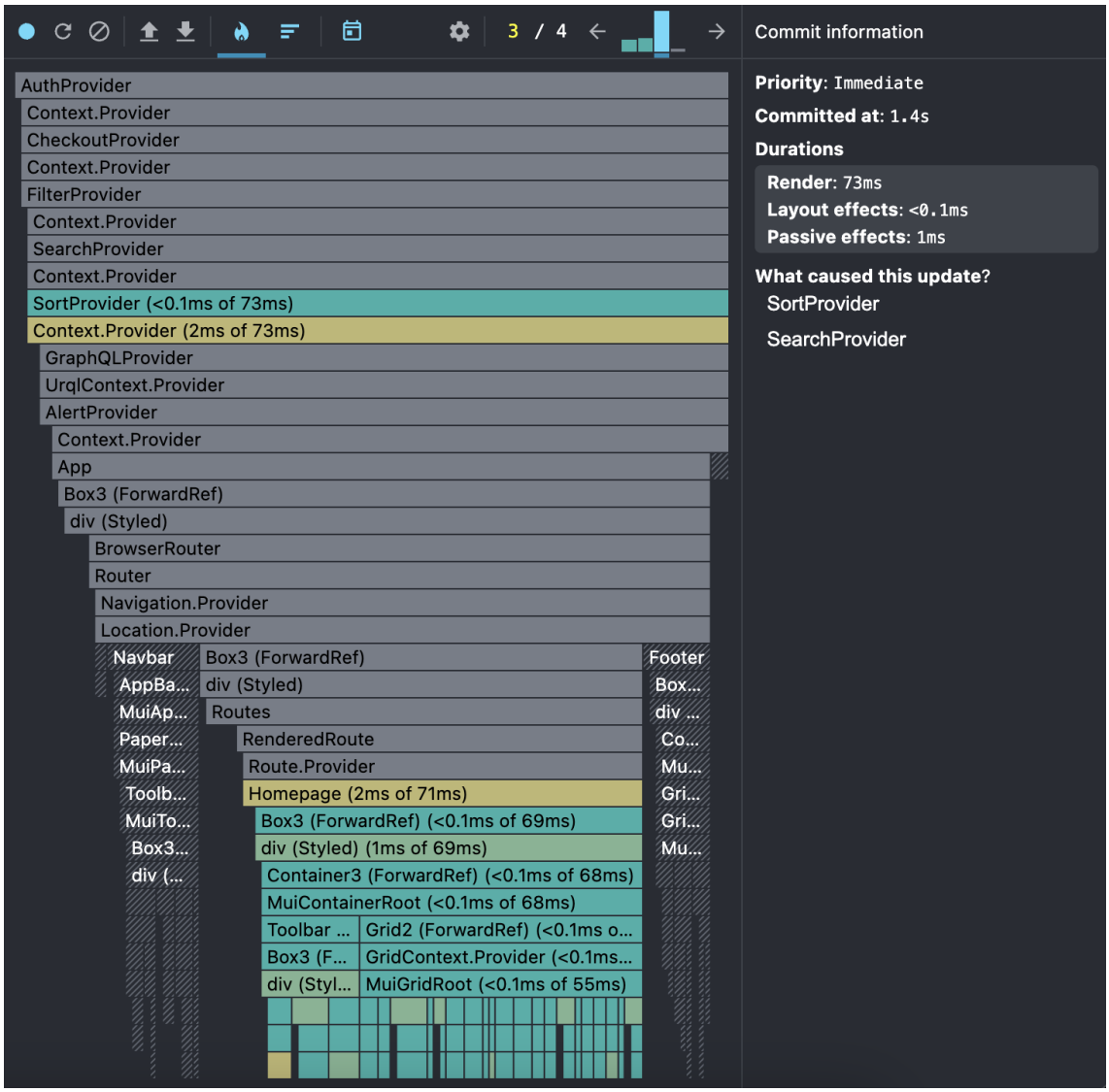

Fig. 4.6. React Profiler after Test 2 in Context API implementation

Browser	Memory Usage (MB)
Safari	23.54
Chrome	8.07

Table 4.11. Memory Usage for Context API

Ecosystem

Popularity of React Context is difficult to measure, because it is not a 3rd party library. In table 4.12 there are statistics for React itself, however this might not reflect actual popularity of using Context API as a state management solution. For this reason they are not going to be used in direct comparison.

Documentation of React explains in depth usage of Context and `useReducer` hook. It is beginner-friendly, contains tutorials and troubleshooting tips.

In terms of TypeScript support Context API performs well, however in case of this

implementation, additional boilerplate code was required to make typing strict on custom reducers.

Stars	Active Issues	Closed Issues	Commits in last year
226k	670	12411	2273

Table 4.12. GitHub statistics for React as of 14-08-2024

4.4. ZUSTAND

4.4.1. Overview of Zustand

Zustand is a lightweight, quick, and flexible state library for React that many developers prefer because of its simplicity and small size. Released in 2019 by development team which included Daishi Kato, it aimed to be a more lightweight alternative to complicated solutions for state management like Redux or MobX, using simplified Flux principles but not requiring any providers or complex setup. Zustand lets you manage global state with hooks and makes sure only the components dependent on state change will be re-rendered. It is tiny and perfect for small apps, but it scales great for larger applications - especially in micro-frontend architectures. It produces re-renders by directly subscribing components to specific state slices [27]. Current version of Zustand is 4.5.4.

4.4.2. Implementation details

Zustand architecture can be implemented in one, centralized, Redux-like stores or multiple stores. In this implementation second option was chosen, as it follows seperation of concerns principle. There is no need for providers, so boilerplate code is minimal. Example implementation of search store is visible in listing 4.8.

`create` function produces a hook using which we select properties from the store. Example usage is visible in listing 4.9.

4.4.3. Results

Bundle size

Bundle size is easily measured because there is only one library, no additional bindings for React are required. This makes it really lightweight, as visible in table 4.13.

Libraries size	Project bundle size
325 kB	538 kB

Table 4.13. Comparison of Libraries Size and Project Bundle Size for Zustand

```tsx
import { create } from "zustand";

interface SearchState {
  query: string;
  setSearchQuery: (query: string) => void;
  resetSearchQuery: () => void;
}

const useSearchStore = create<SearchState>((set) => ({
  query: "",
  setSearchQuery: (query) => set({ query }),
  resetSearchQuery: () => set({ query: "" }),
}));

export const selectSearchQuery = (state: SearchState) => state.query;

export default useSearchStore;
```

Listing 4.8: Search store implementation in Zustand.

```tsx
  const searchQuery = useSearchStore((state) => state.query);
  const setSearchQuery = useSearchStore((state) => state.setSearchQuery);

  const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>)
  ↪  => {
    setSearchQuery(event.target.value);
  };
```

Listing 4.9: Search state update in Zustand.

Performance and re-renders

Lighthouse results for Zustand implementation can be seen in figure 4.7.

Tests 1-3 for performance were repeated 5 times and average time of render time was calculated. Render times are displayed in table 4.14. Example profiler results can be seen in figure 4.8.

Scenario	Average Time ± Uncertainty (ms)	Rerenders
Test 1	113.4 ± 5.0	1
Test 2	65.0 ± 4.0	1
Test 3	473.6 ± 30.0	9

Table 4.14. Average state update time and re-renders for Zustand

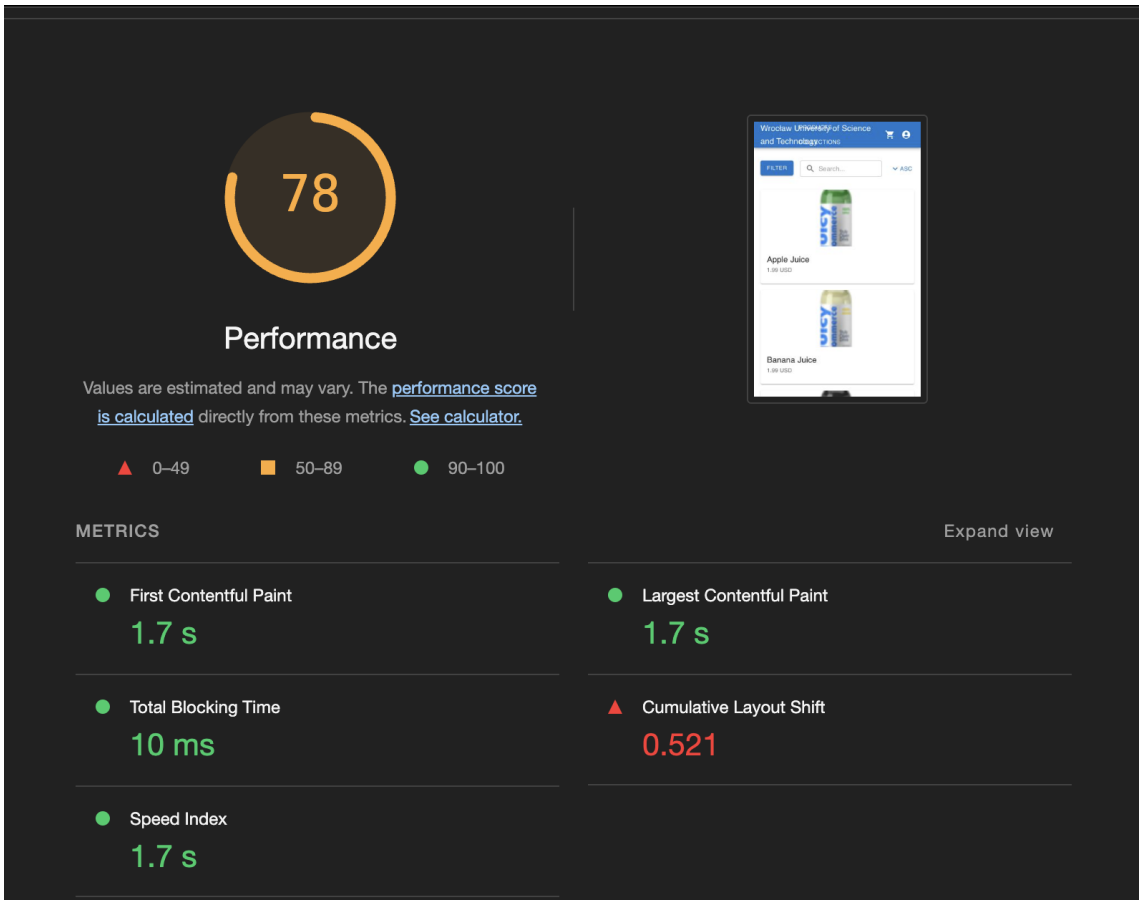

Fig. 4.7. Lighthouse scores for Zustand implementation

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.15.

Browser	Memory Usage (MB)
Safari	20.18
Chrome	7.71

Table 4.15. Memory Usage for Zustand

Ecosystem

Zustand popularity is rapidly rising. Without additional bindings library we can relay on statistics from GitHub visible in table 4.16. In terms of documentation, it explains concepts in an understandable way. There are many details regarding integrations with other libraries and beginner-friendly guides.

TypeScript support is great, Zustand automatically generates all neccesary definitions.

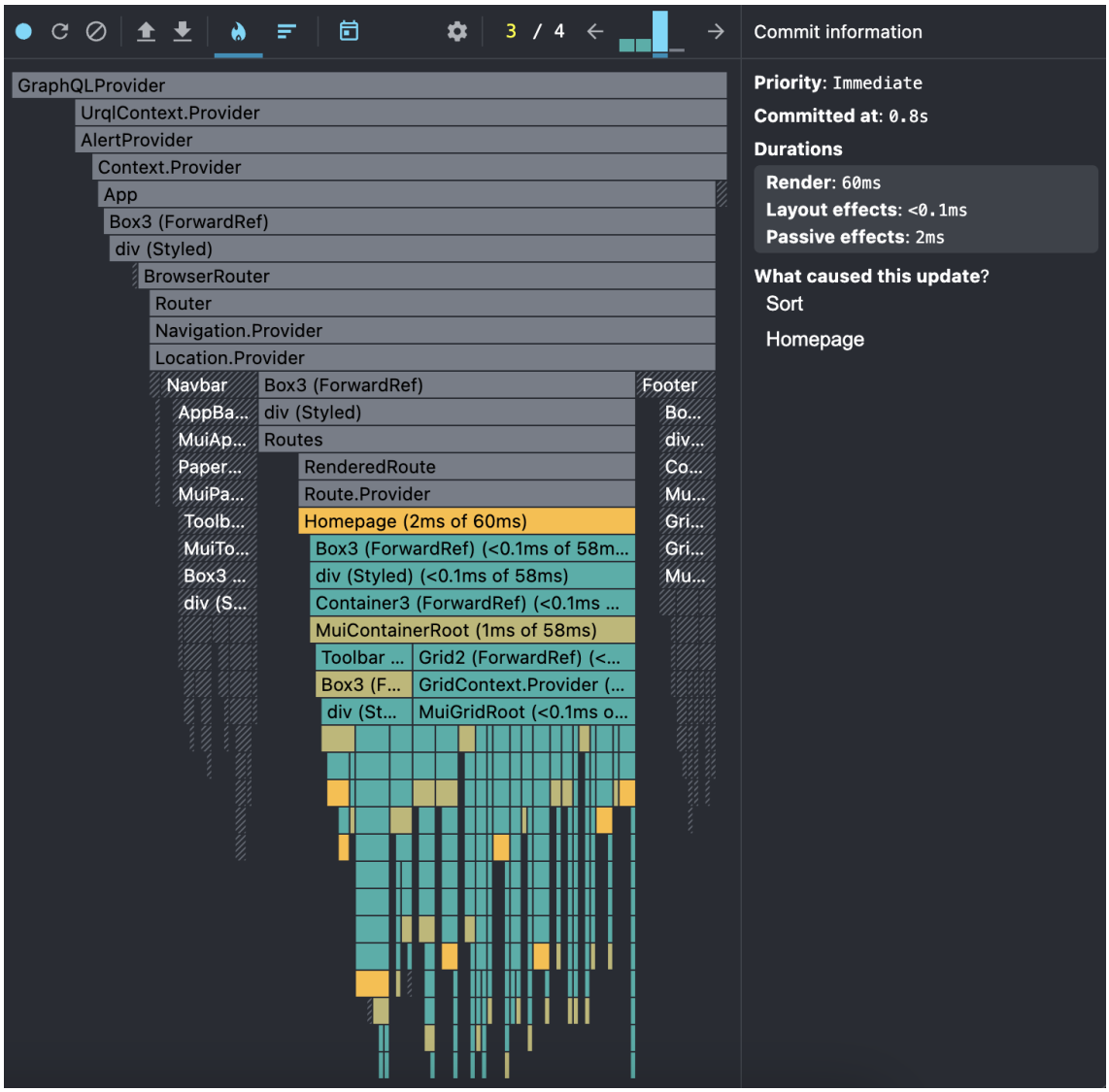

Fig. 4.8. React Profiler after Test 2 in Zustand implementation

4.5. RECOIL

4.5.1. Overview of Recoil

Recoil was developed by Facebook as a state management library, keeping in mind the intention of working very well with React. It was firstly released in 2020. The motivations behind it were some limitations in the built-in state management features of React: for example, an uncomfortable system for sharing component state without lifting it up to a common ancestor, and the limitation within React Context, which can only store one value. Recoil tackles these problems by giving a more flexible and efficient means to manage state.

Key features of Recoil include a directed graph architecture with the React component tree. It lets state changes have flow through pure functions that are called selectors. It also distributes the changes across components, for both synchronous and asynchronous

Stars	Active Issues	Closed Issues	Commits in last year
45.5k	5	661	120

Table 4.16. GitHub statistics for Zustand as of 14-08-2024

data. In addition, Recoil offers a very simple API that mirrors the local state management in React. It further supports advanced features like compatibility with the Concurrent Mode, code-splitting, and the application state persistence surviving across updates of the application [9]. Current version of Recoil is 0.7.7.

4.5.2. Implementation details

Implementation of Recoil does not provide us with actions, which might lead to additional overhead when using in components, however each state definitions are very short. Example for search state is visible in listing 4.10.

```
import { atom } from "recoil";

interface SearchState {
  query: string;
}

export const searchState = atom<SearchState>({
  key: "searchState",
  default: {
    query: "",
  },
});

export const selectSearchQuery = (state: SearchState) => state.query;
```

Listing 4.10: Search state implementation in Recoil.

Example usage in Search component is shown in listing 4.11. It is very similar to usage of state in pure React, however contrary to local state it is available from multiple components.

4.5.3. Results

Bundle size

Recoil is comprised of only one library, with no additional bindings. This in theory should lead to small size, however project size with Recoil turned out to be quite large, as seen in table 4.17.

```
const [searchQuery, setSearchQuery] = useRecoilState(searchState);

const handleSearchChange = (event: React.ChangeEvent<HTMLInputElement>) =>
↪ {
setSearchQuery({ query: event.target.value });
};
```

Listing 4.11: Search usage in Recoil.

Libraries size	Project bundle size
2.21 MB	609 kB

Table 4.17. Comparison of Libraries Size and Project Bundle Size for Recoil

Performance and re-renders

Lighthouse results for Recoil implementation can be seen in figure 4.9.

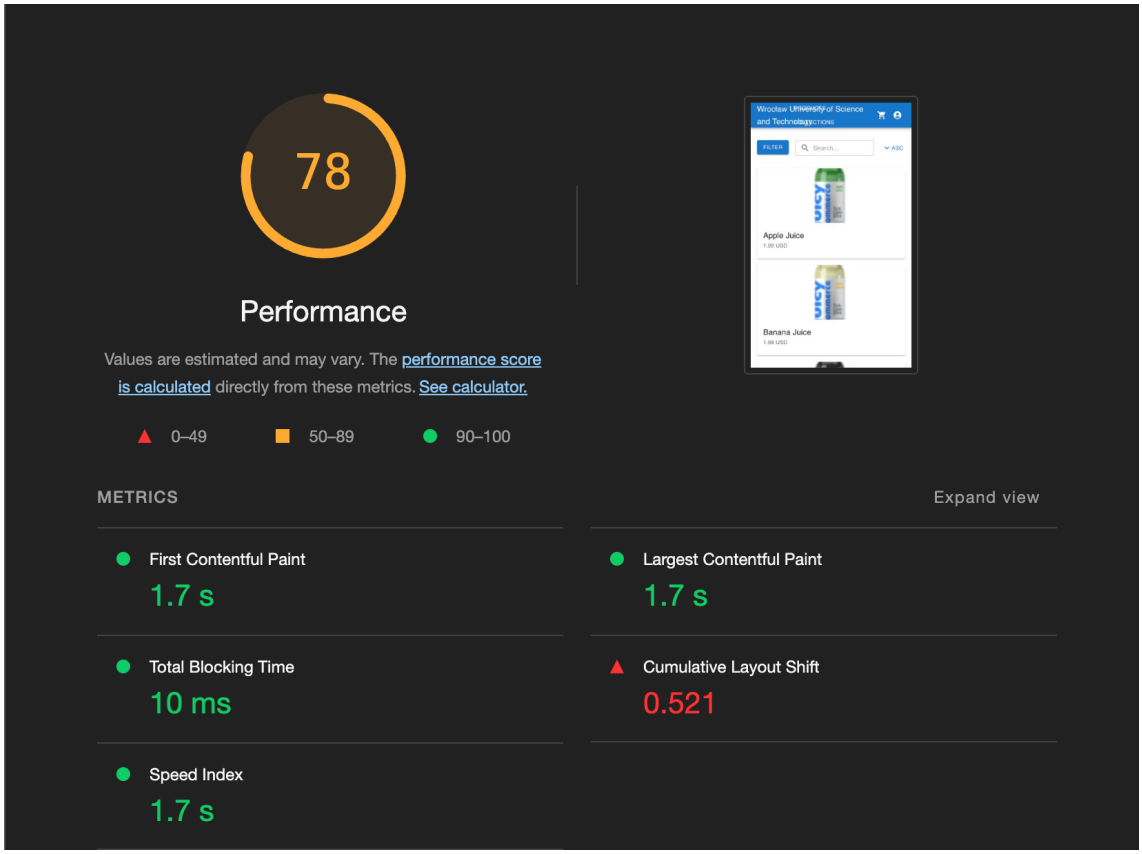

Fig. 4.9. Lighthouse scores for Recoil implementation

Tests 1-3 for performance were repeated 5 times and average time of render time was calculated. Render times are displayed in table 4.18. Example profiler results can be seen in figure 4.10.

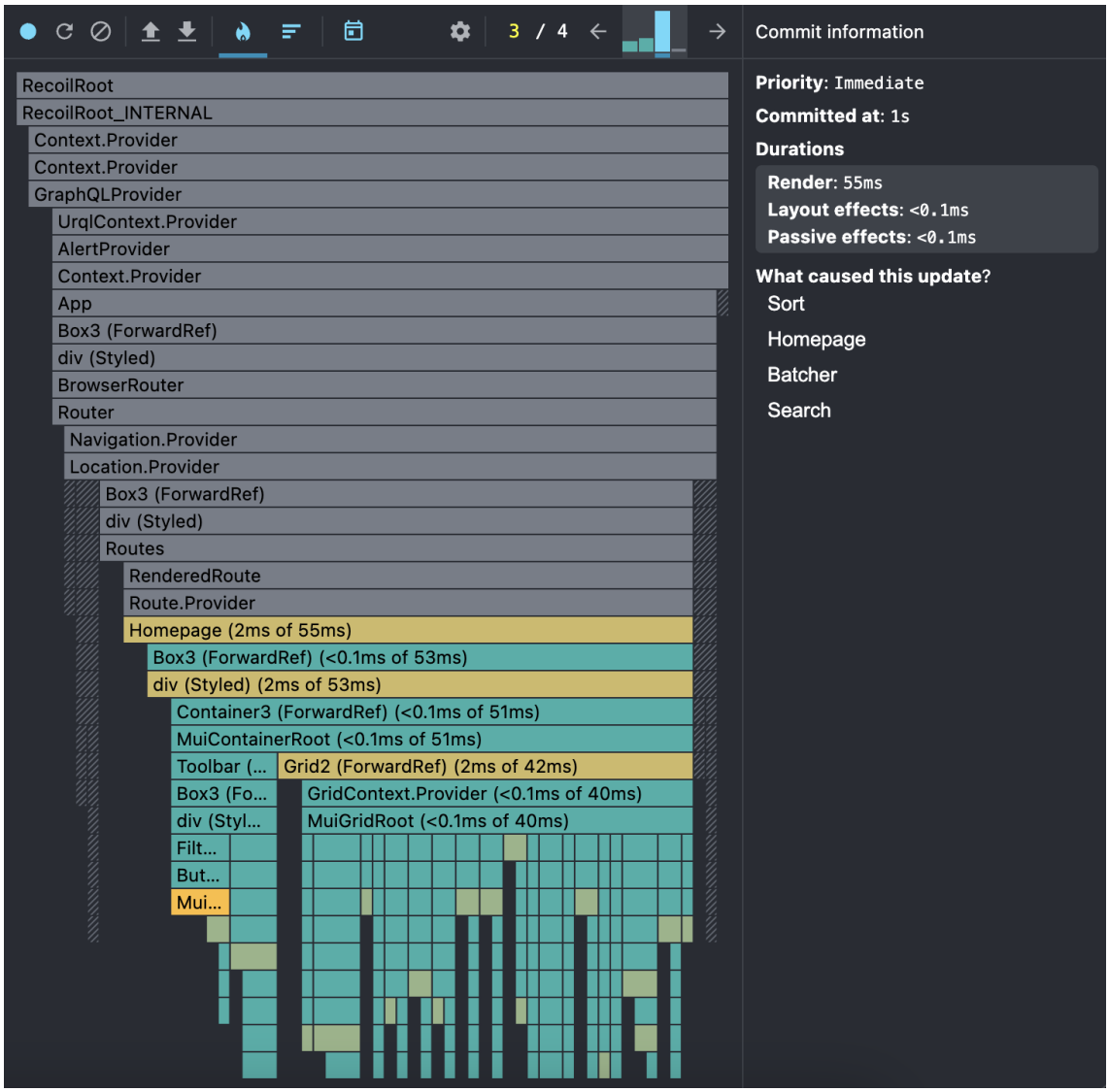

Fig. 4.10. React Profiler after Test 2 in Recoil implementation

Scenario	Average Time ± Uncertainty (ms)	Rerenders
Test 1	113.2 ± 4.0	1
Test 2	55.8 ± 4.0	1
Test 3	497.2 ± 10.0	10

Table 4.18. Average state update time and re-renders for for Recoil

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.19.

Browser	Memory Usage (MB)
Safari	22.73
Chrome	7.43

Table 4.19. Memory Usage for Recoil

Ecosystem

Recoil's popularity is relatively easy to measure since it does not rely on other additional binding libraries; it is a standalone state management solution in the React ecosystem. The comparison with Redux is based on the core library itself. The GitHub statistics of Recoil are presented in table 4.20. While it is still newer and less mature than established libraries like Redux or MobX, Recoil is a framework that gives you a modern, intuitive API natively built with React. Documentation is straightforward, so it really is accessible for newbies to veterans.

Recoil also shines through its TypeScript support. The library has quite strong TypeScript typings, which make it easy for one to define and manage atoms, selectors, and all other components in a type-safe manner. This allows developers to take full advantage of TypeScript features while working with Recoil, like type inference and autocompletion.

Stars	Active Issues	Closed Issues	Commits in last year
19.5k	255	778	2

Table 4.20. GitHub statistics for Recoil as of 15-08-2024

4.6. JOTAI

4.6.1. Overview of Jotai

Jotai is a primitive state management library, designed to simplify the process of managing global state. It was created by Daishi Kato (who is author of Zustand as well). Firstly released in 2021, it draws inspiration from Recoil's atomic model. Developers can build complex models using derived atoms, which are computed based on values of other atoms [20]. Current version of Jotai is 2.9.3.

Compared to Recoil, Jotai is focused on more primitive API that is unopinionated and easy to learn, without additional features like caching strategies available in Recoil.

4.6.2. Implementation details

Jotai is implemented using atoms, similar to Recoil. Developer familar with the latter library should be able to quickly understand the concepts. The selector functions used on homepage or collection pages can be implemented as derived atoms, which are ready to use. Example implementation of search atom is visible in listing 4.12.

```ts
import { atom } from "jotai";

interface SearchState {
  query: string;
}

const initialSearchState: SearchState = {
  query: "",
};

export const searchAtom = atom(initialSearchState);

export const selectSearch = atom((get) => {
  const state = get(searchAtom);
  return state.query;
});
```

Listing 4.12: Search atom in Jotai

4.6.3. Results

Bundle size

Jotai is designed as a lightweight library, similar to Zustand. Its package size is very small, ensuring quick initial load times, as seen in table 4.21.

Libraries size	Project bundle size
428 kB	537 kB

Table 4.21. Comparison of Libraries Size and Project Bundle Size for Jotai

Performance and re-renders

Lighthouse results for Jotai implementation can be seen in figure 4.11.

Tests 1-3 for performance were repeated 5 times and average time of render time was calculated. Render times are displayed in table 4.22. Example profiler results can be seen in figure 4.12.

Scenario	Average Time ± Uncertainty (ms)	Rerenders
Test 1	108.8 ± 3.0	1
Test 2	55.8 ± 1.0	1
Test 3	414.8 ± 10.0	9

Table 4.22. Average state update time and re-renders for Jotai

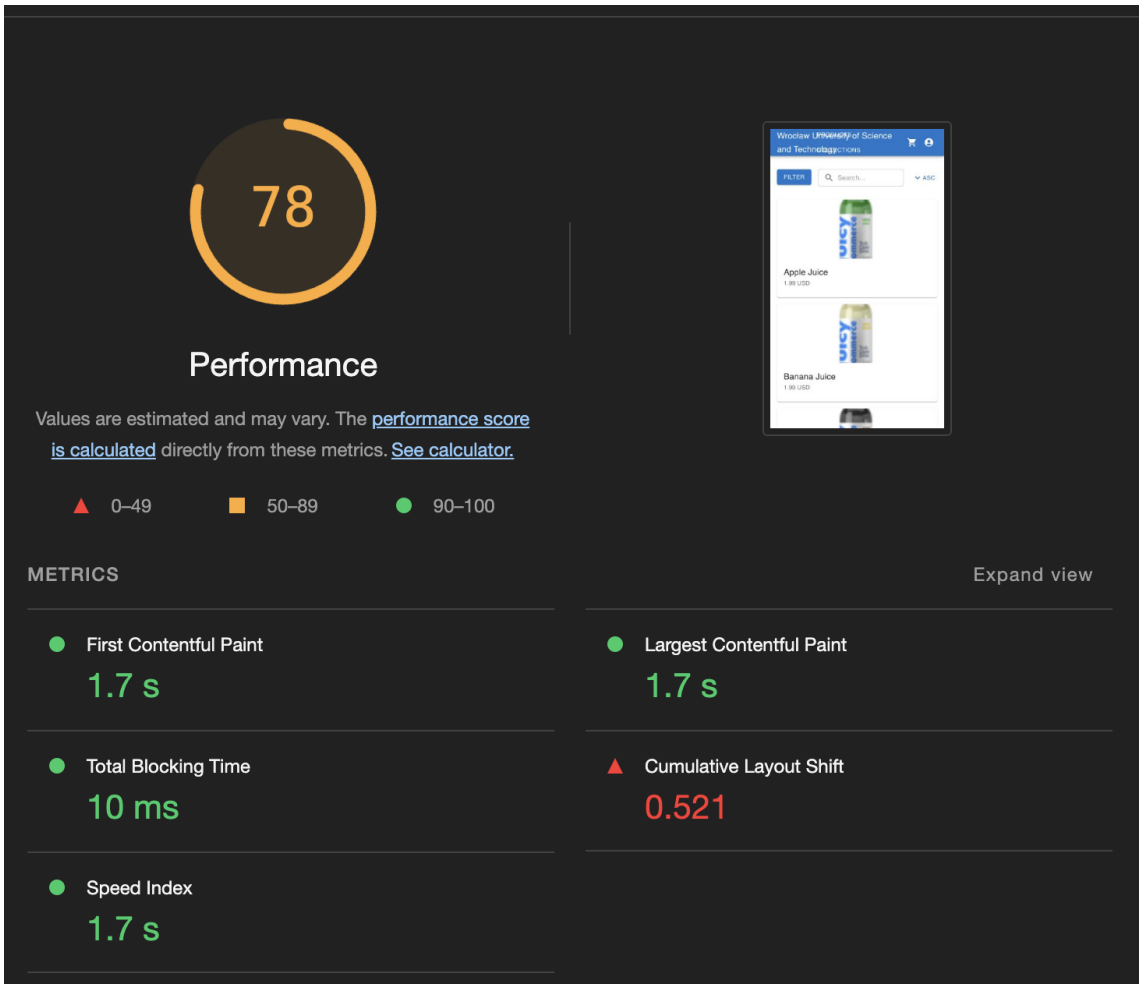

Fig. 4.11. Lighthouse scores for Jotai implementation

Memory consumption

Memory consumption of the application was tested and results are visible in table 4.23.

Browser	Memory Usage (MB)
Safari	22.32
Chrome	8.21

Table 4.23. Memory Usage for Jotai

Ecosystem

Ecosystem for Jotai is rapidly growing, as it is the newest library in comparison. Its GitHub statistics are visible in 4.24. It already achieved sufficient documentation, which compares it to other libraries, provides guides for basic and more advanced concepts and includes API reference.

Jotai's TypeScript support is also excellent, requiring strict null checks enabled for optimal usage. Derived atoms can be inferred or explicitly typed.

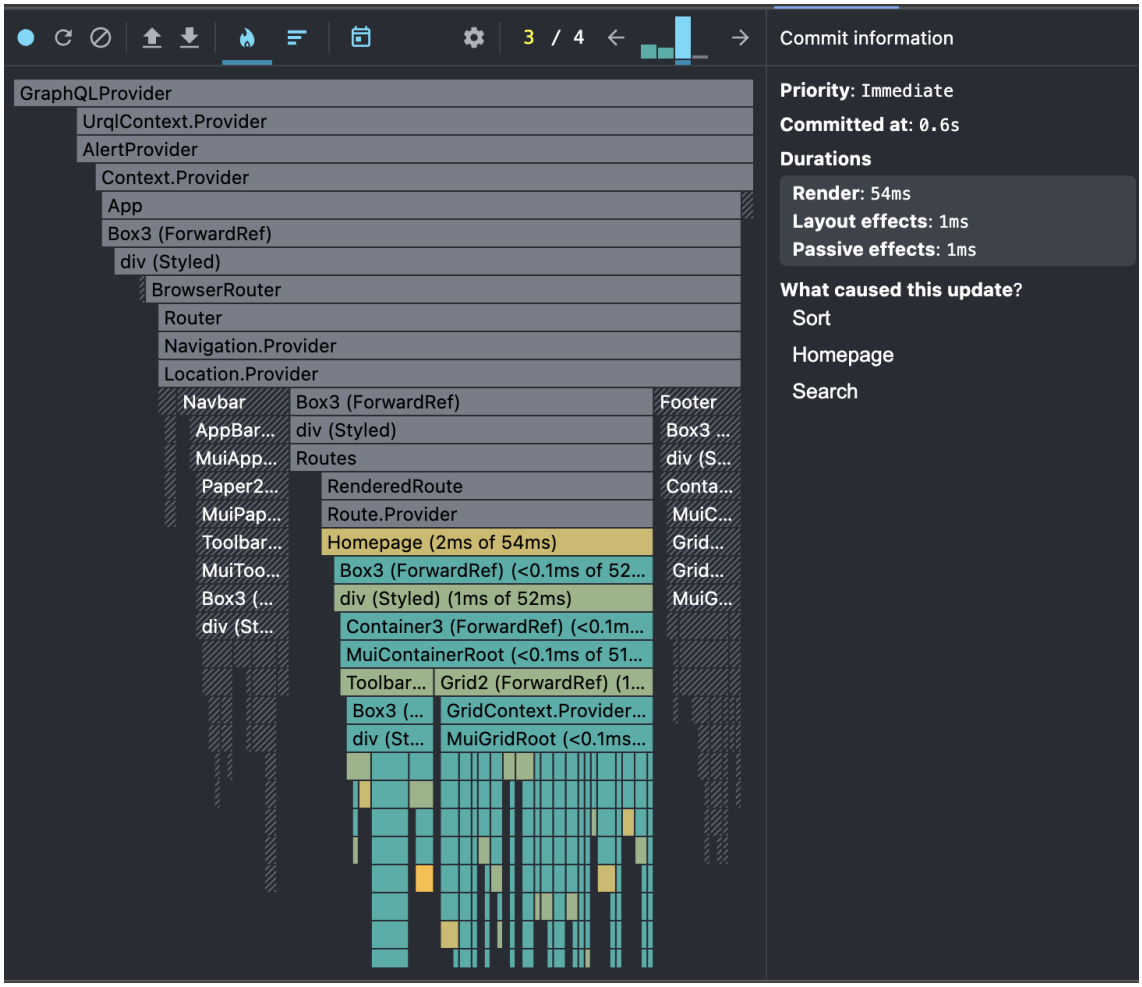

Fig. 4.12. React Profiler after Test 2 in Jotai implementation

Stars	Active Issues	Closed Issues	Commits in last year
18.1k	2	638	234

Table 4.24. GitHub statistics for Jotai as of 16-08-2024

5. DISCUSSION

5.1. COMPARISON OF BUNDLE SIZE

As seen in figure 5.1, state management solutions differ substantially in their size. The biggest library is Redux weighing 6.2 MB, Context API is part of React, so effectively its weight can be measured as zero. Apart from Context, the smallest library is Zustand, achieving impressive 325 kB.

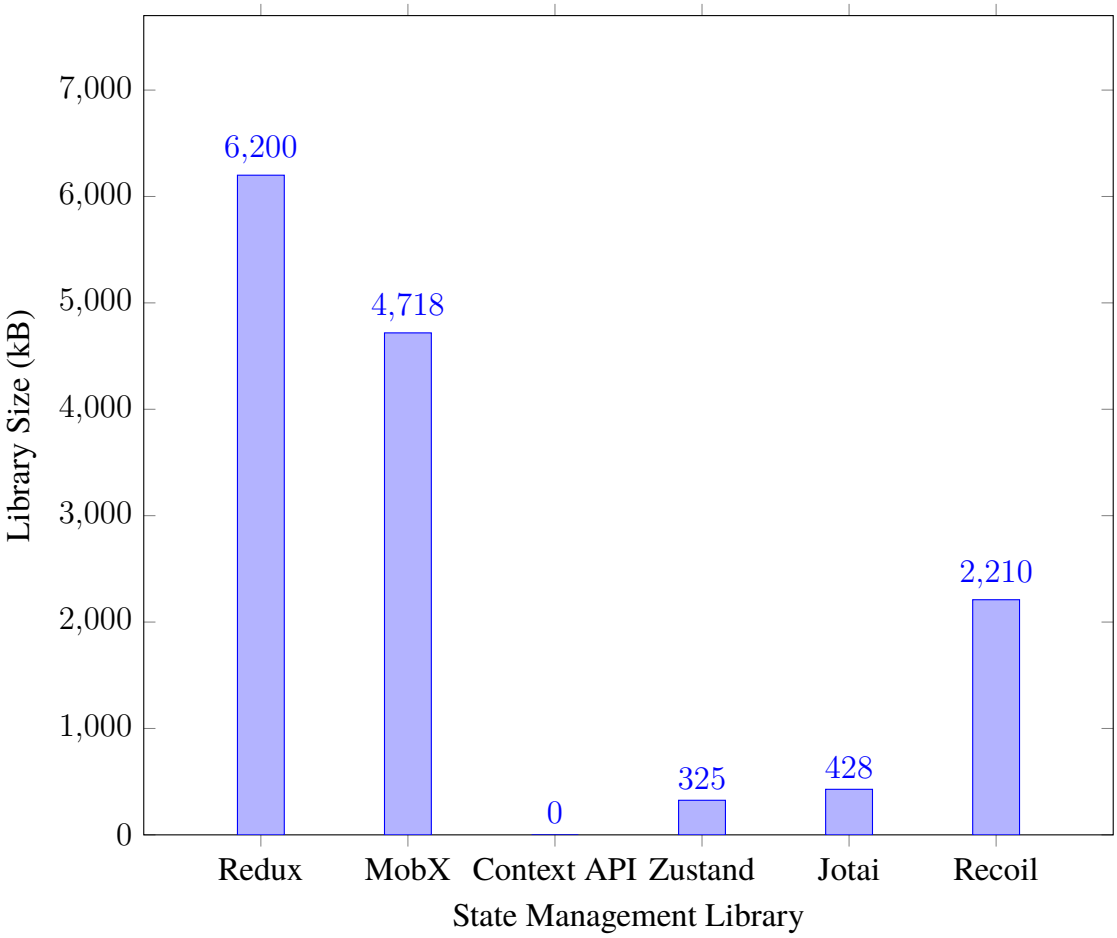

Fig. 5.1. Library sizes of various state management solutions (in kB)

In practice, due to tree-shaking process introduced by JavaScript bundlers, state management solutions influence package size of front-end projects in a way that is not that significant. There is correlation - libraries with larger sizes influence project bundle size more. As visible in figure 5.2, the baseline here is Context API with 533 kB. Jotai ended up being the smallest 3rd party library adding mere 4 kB to the bundle size. Surprisingly,

neither Redux nor Mobx but Recoil emerged as the largest, adding 76 kB with total size of 609 kB.

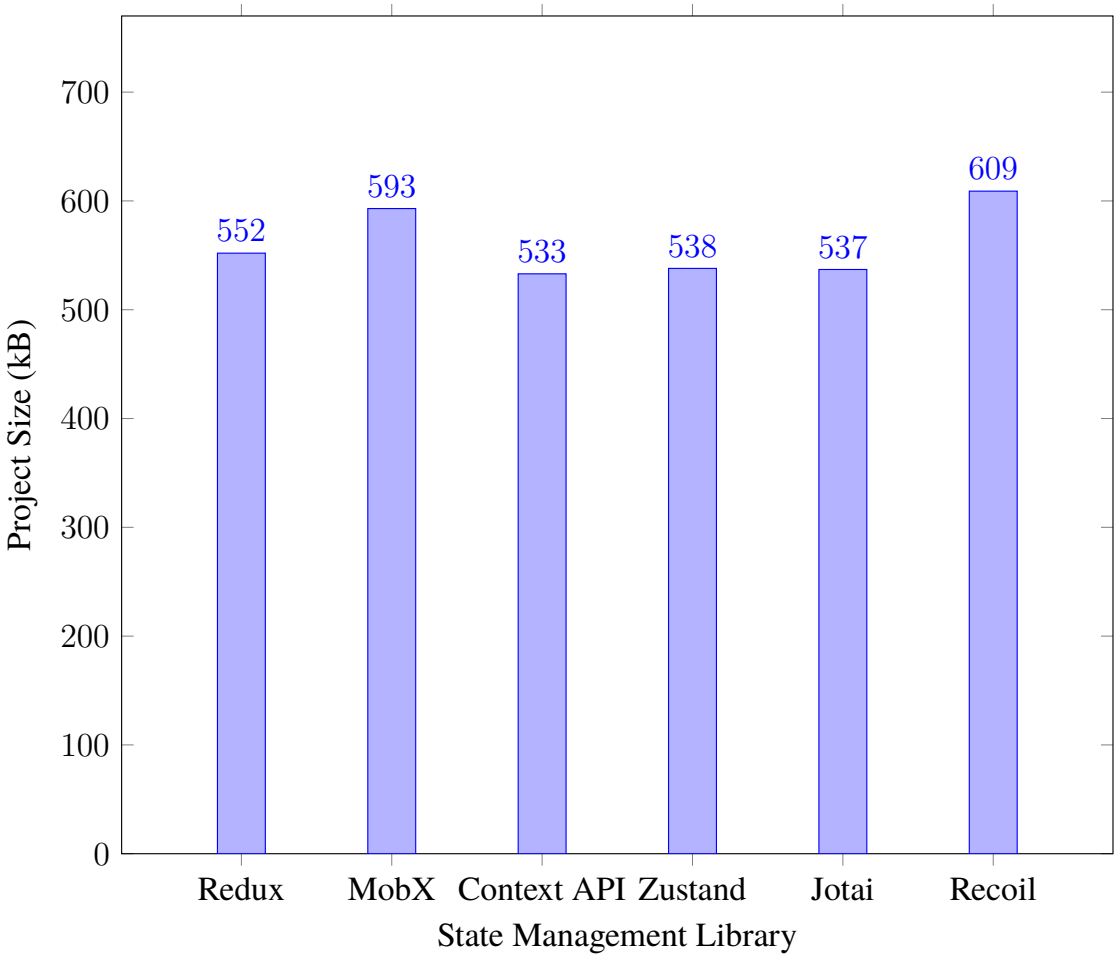

Fig. 5.2. Project sizes of various state management solutions (in kB)

5.2. COMPARISON OF PERFORMANCE

In terms of Lighthouse scores, the tests were identical. TBT and LCP values were the same for all state management solutions, suggesting that selection of a particular library has little to none influence on SEO perormance, as visible in figures 5.3 and 5.4.

No significant differences in state update times were found as well. In test 1, which was designed to mimic change of object of moderate size (in this case filters), all solutions ended up with similar times. Jotai emerged as minimally better than other options with average time of 108.8 ms, whereas Context API was the slowest with average of 115 ms. Other solutions' times were similar, as seen in figure 5.5.

In test 2 which changes a low complexity field - sort order, more variance was found. Jotai and Recoil were the fastest libraries of all with average state update time of 55.8 ms.

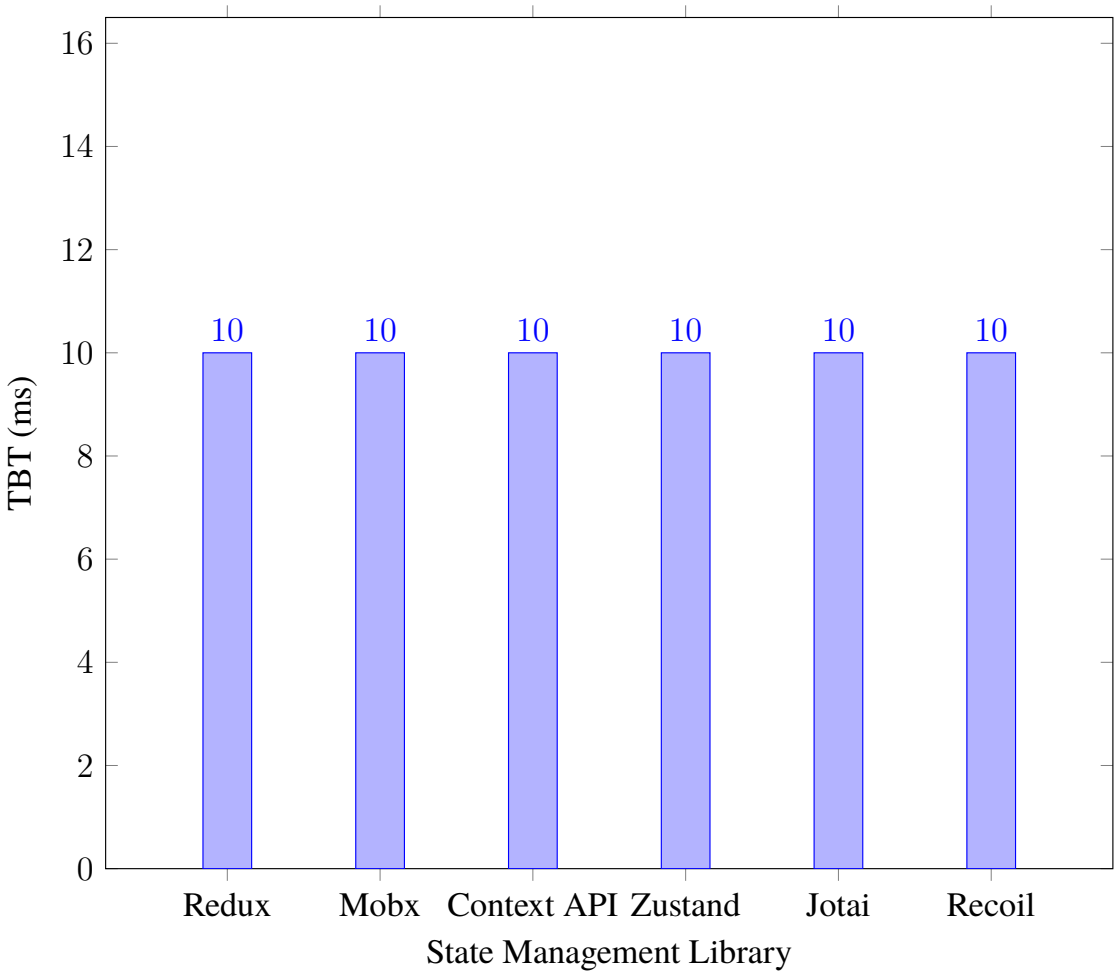

Fig. 5.3. TBT values of particular state management solutions

The slowest was Redux with 67.4 ms. MobX achieved a good time as well with average of 57.8 ms, as shown in figure 5.6.

Recoil is the only library which caused 1 more re-render, only in the most complex Test 3, as visible in figure 5.8. This directly caused its worst performance in Test 3, as demonstrated in figure 5.7. Jotai once again turned out to be the fastest with average time of 414.8 ms. Zustand had a particularly bad performance in this test with time of 473.6 ms, even though it achieved 9 re-renders unlike Recoil.

5.3. COMPARISON OF MEMORY CONSUMPTION

Due to the existence of Garbage Collector in JavaScript, memory consumption of JavaScript is unstable. This is important in the context of interpreting results provided in figure 5.9. Considering minimal differences between the applications, it is safe to assume that choice of state management solution has minuscule effect on memory consumption.

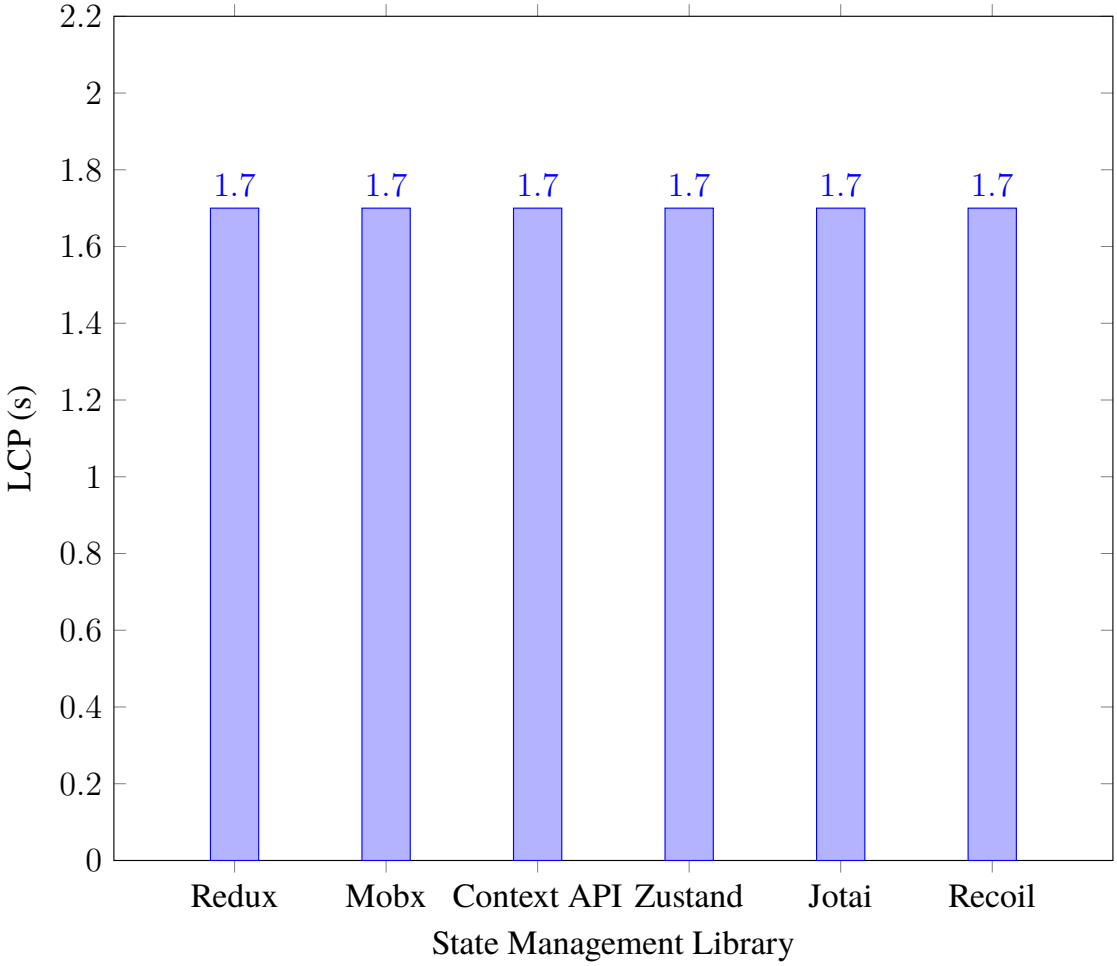

Fig. 5.4. LCP values of particular state management solutions

5.4. COMPARISON OF ECOSYSTEM

In terms of ecosystem, all libraries have excellent documentation which provides guides for beginner developers, API reference and explanations of more advanced concepts. The same can be said about TypeScript support - every library has its own type definitions, making development faster and more reliable.

What can be effectively compared is each solution is popularity. As shown in figure 5.10, Redux remains the most popular solution. Zustand gained massive amount of GitHub stars, overtaking MobX which is as old as Redux. The least popular libraries at this moment are Recoil and Jotai, with less than 20000 stars.

When comparing GitHub issues, time frame should also be taken into account. As seen in figure 5.11, all libraries have low amounts of open issues apart from Recoil. Adding to this fact that latest Recoil release is still major version 0, which is considered for initial development, one can draw conclusions that this library is not actively maintained. Low amount of commits in the last year which is visible in figure 5.12, also supports this idea.

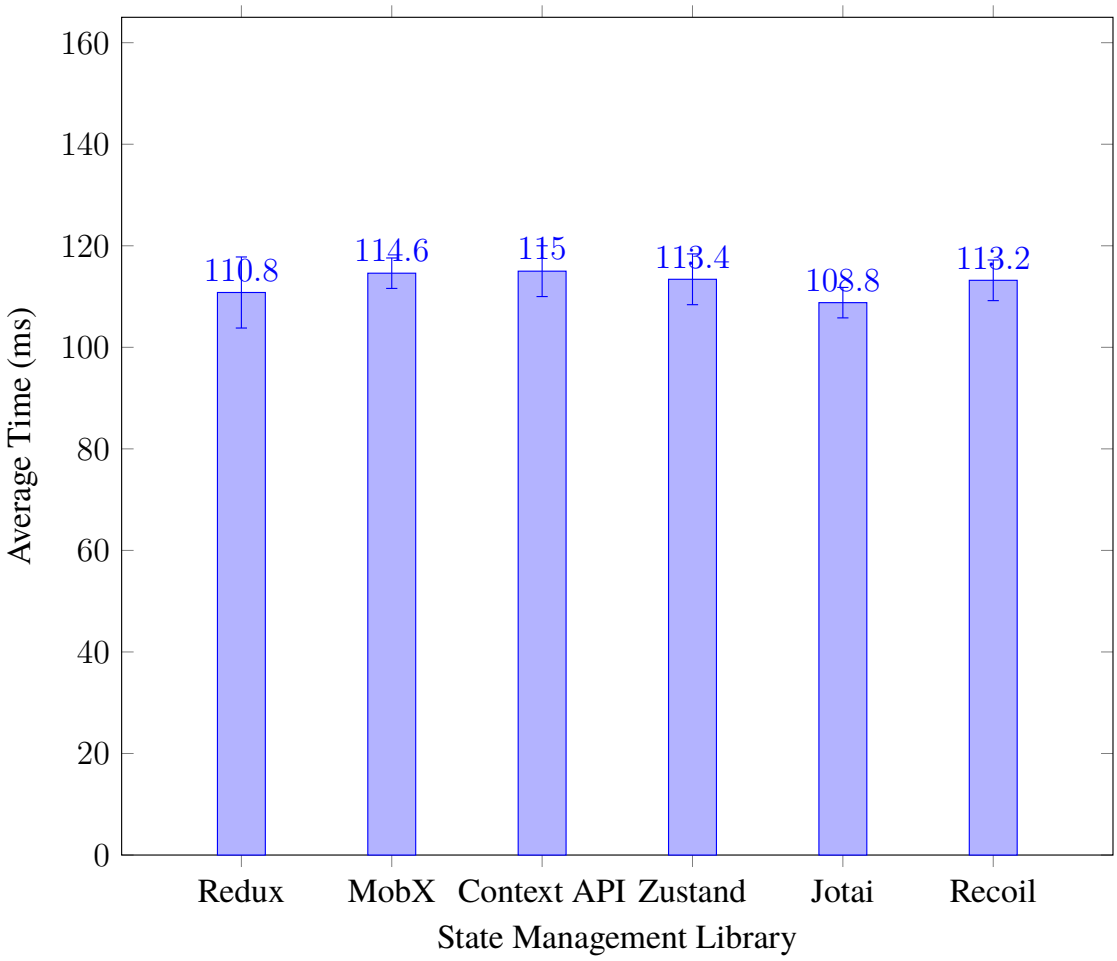

Fig. 5.5. Average time for Test 1 with uncertainties

5.5. RECOMMENDATIONS

Based on the above findings, it is safe to assume that all libraries have their place and most important aspect for selection is developer's preference. A set of recommendations can be created:

1. Redux should be chosen by developers who prefer mature, tried and tested solutions, as well as for large scale applications.
2. MobX should be chosen by developers who prefer object-oriented style of writing code, because of class-based syntax.
3. Context API should be chosen for small to moderate-sized applications which also aim for small bundle size. Rapidly growing boilerplate when scaled makes this solution unsuitable for large applications.
4. Zustand should be chosen by developers who prefer simplicity and balance between popularity and maturity, but also want to reduce their bundle size. It is a great choice for people who previously used Redux and want to try something new.

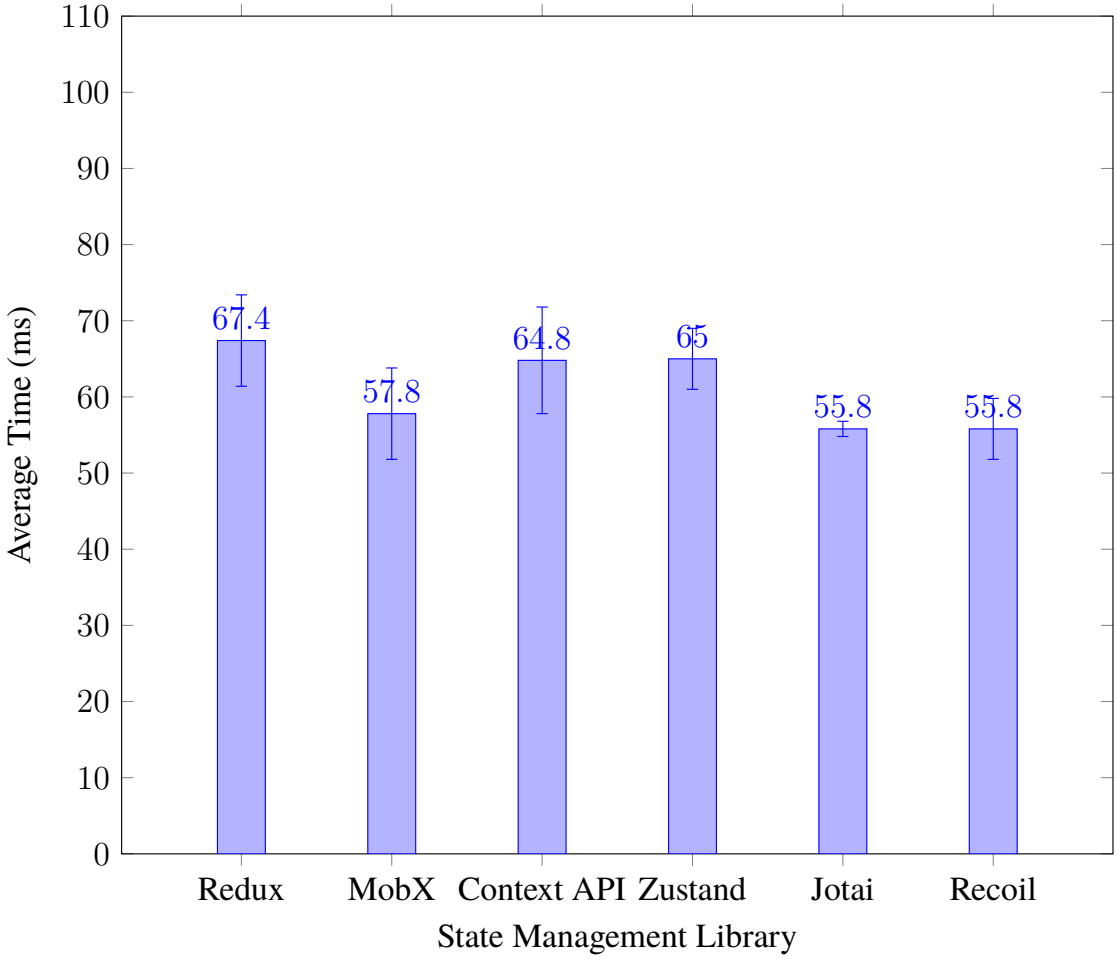

Fig. 5.6. Average time for Test 2 with uncertainties

5. Jotai should be chosen by developers who enjoy trying new solutions, for performance-oriented applications which also need small bundle size.
6. Recoil is not recommended because it is likely not maintained. Developers who prefer Recoil's atom syntax should achieve similar results using Jotai with much smaller bundle size.

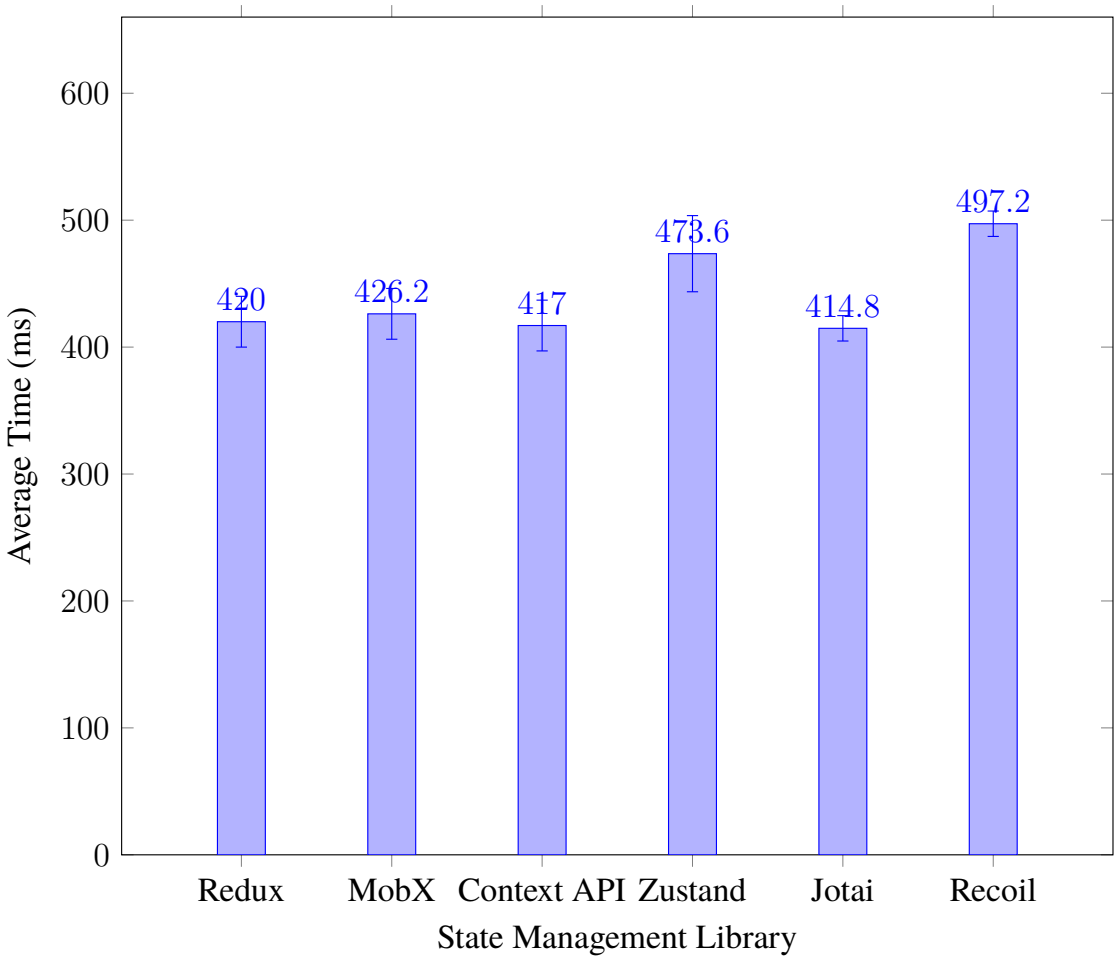

Fig. 5.7. Average time for Test 3 with uncertainties

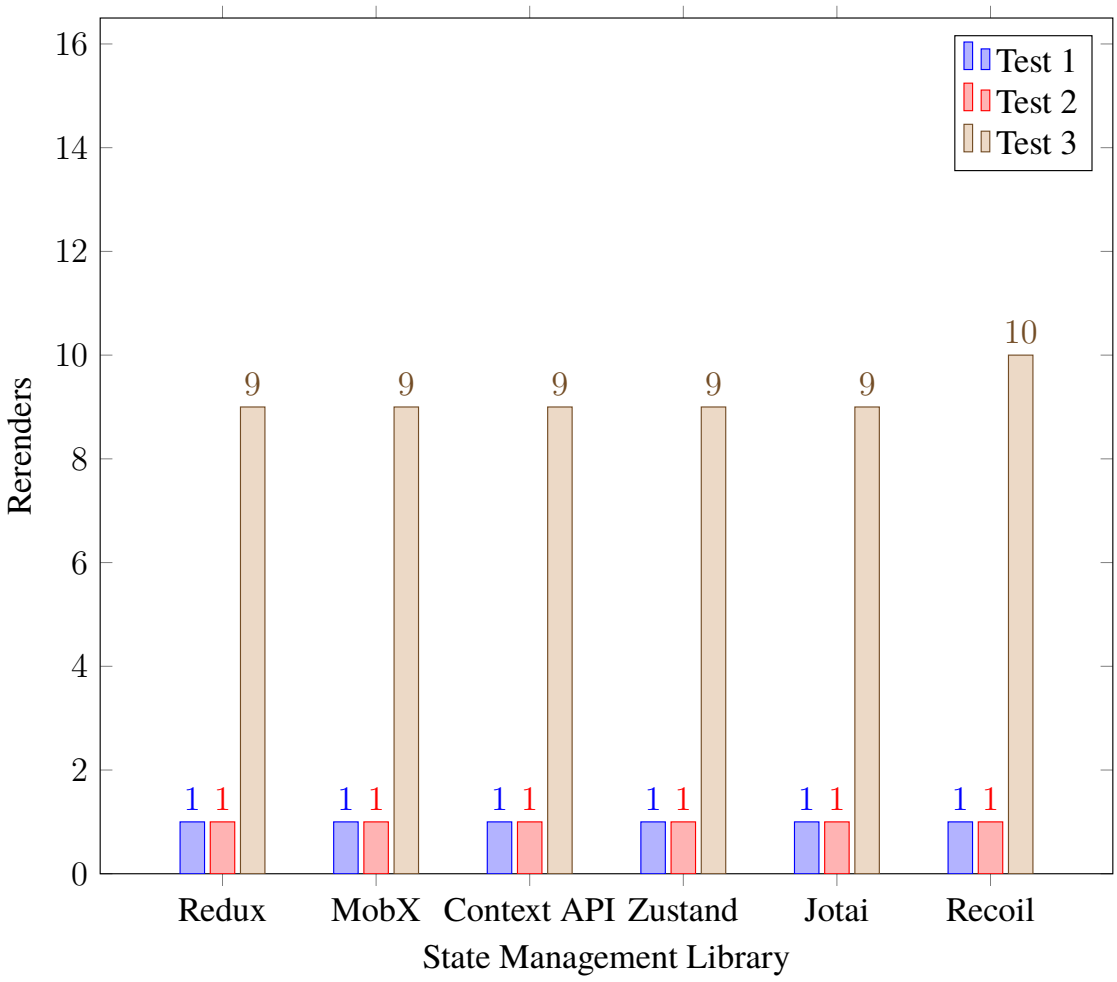

Fig. 5.8. Re-renders for each performance test

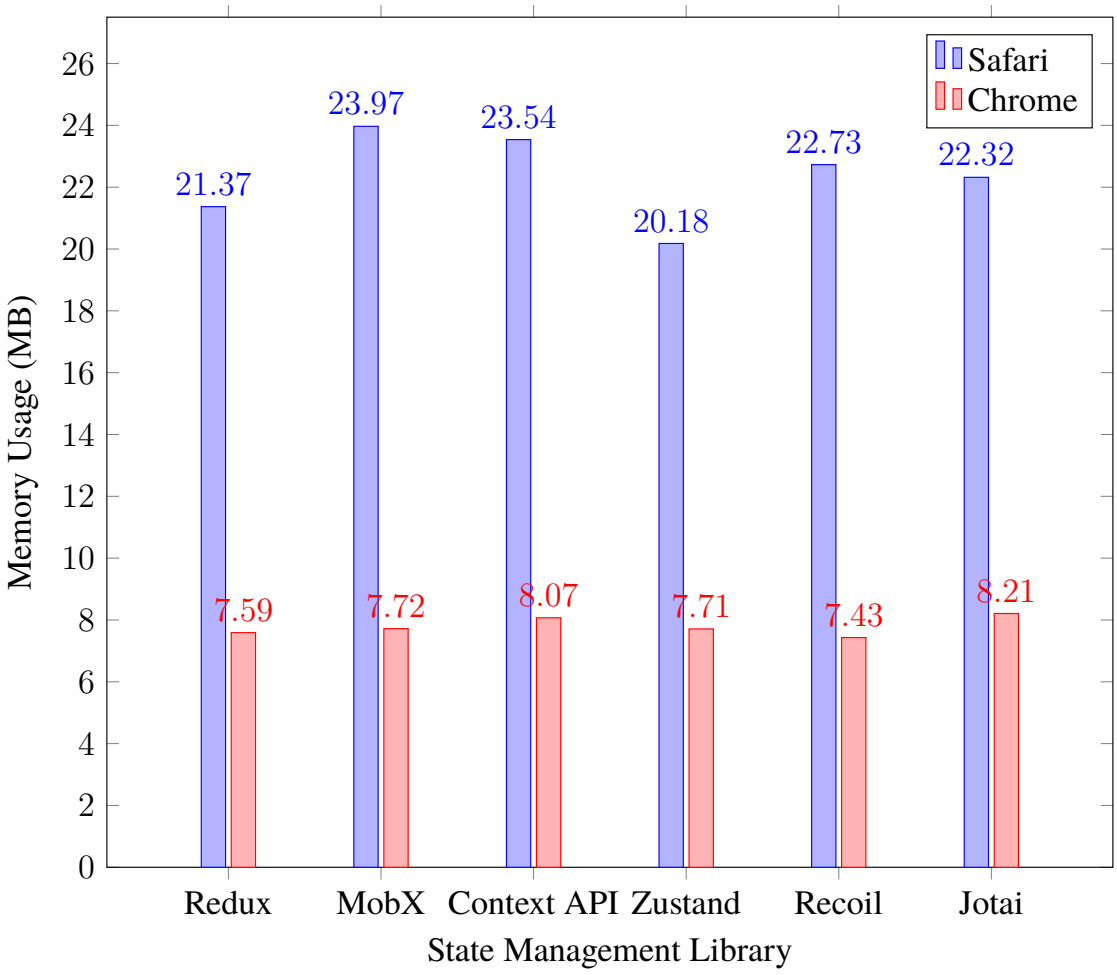

Fig. 5.9. Memory usage per state management library on Safari and Chrome

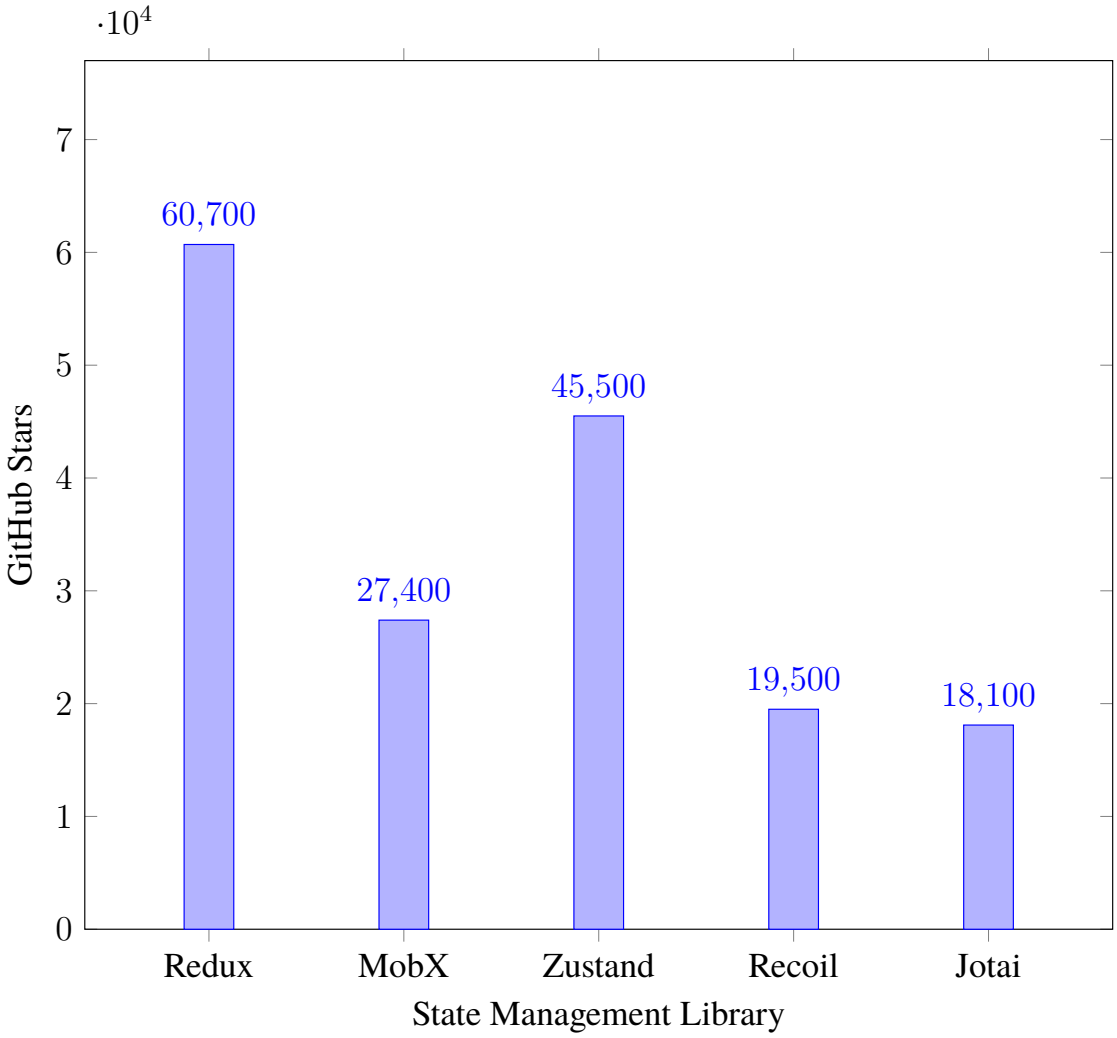

Fig. 5.10. GitHub Stars for state management libraries

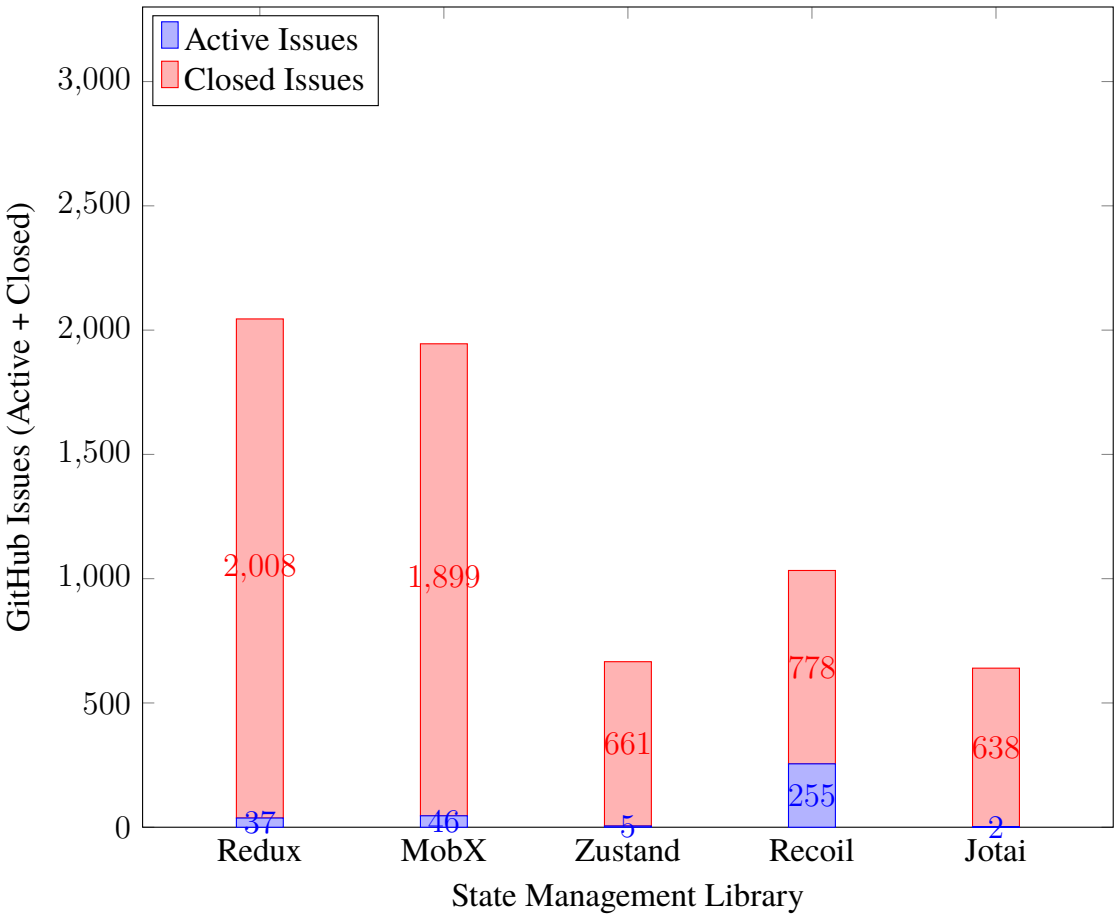

Fig. 5.11. Active and closed issues for state management libraries

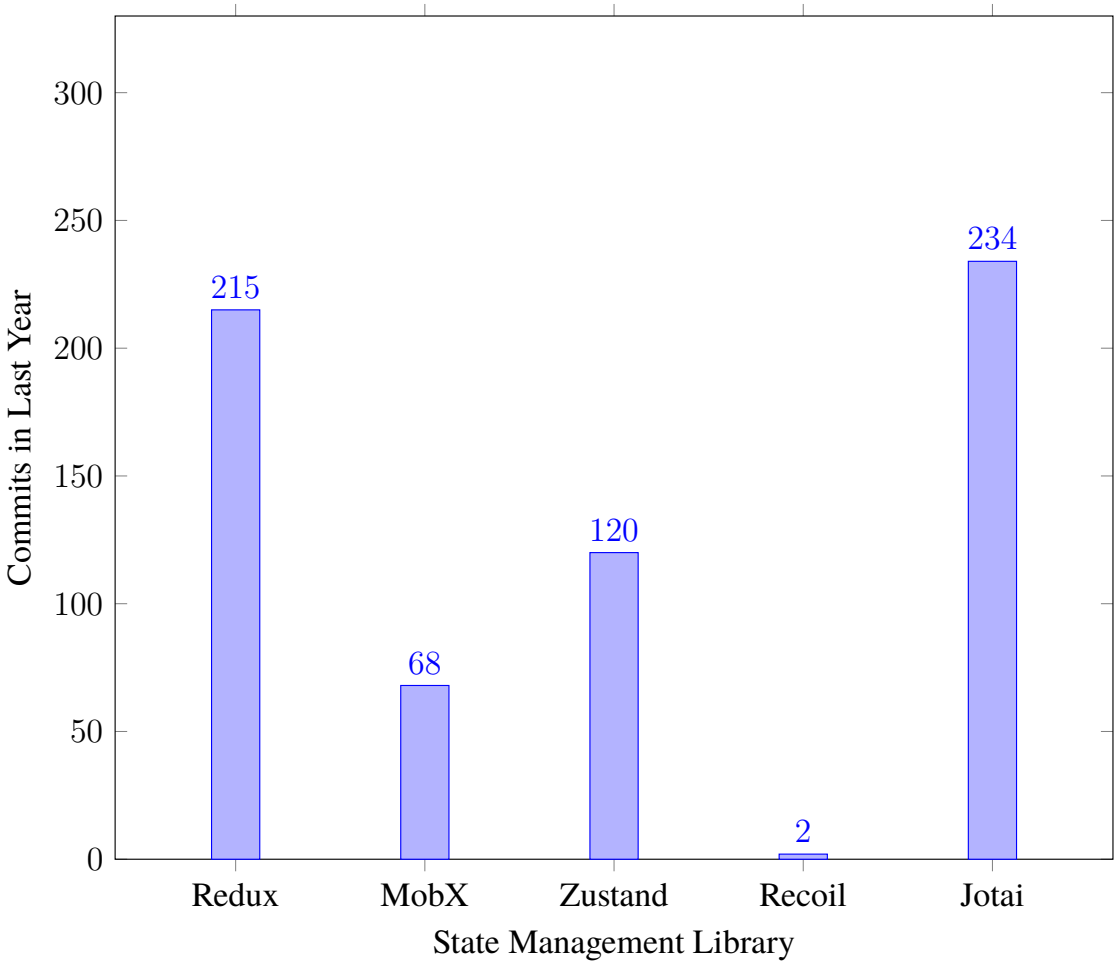

Fig. 5.12. Commits in the last year for different state management libraries

6. CONCLUSIONS

In this thesis, an e-commerce store was implemented six times. Each copy has the same functionality and user interface but uses a different state management solution under the hood: Redux, MobX, Context API, Zustand, Recoil, and Jotai. Applications were thoroughly analyzed and compared across different metrics, more specifically bundle size, performance, memory consumption, and ecosystem. Based on the findings, a set of practical recommendations for front-end developers was created.

6.1. LIMITATIONS

This study has several limitations that should be taken into account when interpreting the results.

Limited scenarios for state updates

Performance tests include few predefined scenarios which were chosen arbitrarily. It is possible that state libraries perform differently in particular situations. Even though these scenarios provide baseline for comparison, they might not capture all use cases of a particular library, especially in applications with more complex interactions.

Fixed API choice

In this study Saleor API was chosen, which is suitable and provides enough flexibility to conduct performance tests. Nonetheless, it is possible that different APIs might yield different results, as they provide different data structures and their query logic can be different.

Limited ecosystem evaluation

The ecosystem analysis is based on GitHub statistics, documentation presence and TypeScript support. It is also possible to verify other important factors like ease of integration, developer experience or 3rd party integration. These might influence eventual choice of a particular solution.

Hardware dependence

Tests were performed on single piece of high-end hardware, and even though CPU throttling was used, performance tests might not fully reflect real-world scenarios, where users' devices differ in computing power.

Focus on Single-Page Applications

With the rise of frameworks like Next.js, SPAs are not sole option for building web applications. Using different architectures like server-site rendered apps or multi-page applications might lead to different conclusions when comparing state managers.

Time relevance

This study was performed in 2024. Trends in front-end development change rapidly and measures like popularity might change drastically in two years from now. Given development of JavaScript ecosystem, new features and optimizations it is possible that conclusions from this study will not be able to stand the test of time.

6.2. FUTURE WORK

This field can be further explored in various ways, firstly addressing concepts outlined in limitations.

Diverse hardware testing

Performance tests can be performed on different types of hardware. Testing on lower-end devices, especially mobile will give insights about validity of results in real-world scenarios.

Testing on different architectures

State managers are commonly used with frameworks supporting architectures like server-site rendering (SSR) or static site generation (SSG). One example of such framework which gained a lot of popularity is Next.js. It would be interesting to verify if state managers behave differently in such architectures.

Developer experience

Surveys can be conducted to analyze subjective experience of people who actually use state managers in day to day work. Engineers tend to use tools that are enjoyable and easy to use.

Code maintainability

When projects grow in complexity, management of state is getting harder to maintain. It would be interesting how fast boilerplate code is changing with the development of the project.

Testability

This study did not cover if different state management solutions provide testing tools. In many companies unit and integration testing is important part of development and ease of testing reducers, actions and stores can be driving factor in choosing a state management library.

BIBLIOGRAPHY

[1] Abramov, D., *Live react: Hot reloading with time travel*, https://www.youtube.com/watch?v=mpbWQbkl8_g. React Europe 2015. Accessed: 19-08-2024.

[2] Bugl, D., *Learn React Hooks: Build and refactor modern React. js applications using Hooks* (Packt Publishing Ltd, 2019).

[3] Choi, D., *Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL* (Packt Publishing Ltd, 2020).

[4] Davidson, T., *What is Vite?*, https://cleancommit.io/blog/what-is-vite/. Accessed: 10-08-2023.

[5] Dobbala, M.K., Lingolu, M.S.S., *Web performance tooling and the importance of web vitals*, Journal of Technological Innovations. 2022, volume 3, 3.

[6] Erikson, M., *The History of Redux*, https://redux.js.org/understanding/history-and-design/history-of-redux. Accessed: 08-08-2024.

[7] Evergreen, P., *Selecting a State Management Strategy for Modern Web Frontend Applications*, Master thesis, Tampere University. 2023.

[8] Facebook, *Flux Concepts*, https://github.com/facebookarchive/flux/tree/main/examples/flux-concepts. Accessed: 08-08-2024.

[9] Facebook, *Motivation: Recoil*, https://recoiljs.org/docs/introduction/motivation. Accessed: 15-08-2024.

[10] Fischer, B., *Flux: Actions and the dispatcher – react blog*, https://legacy.reactjs.org/blog/2014/07/30/flux-actions-and-the-dispatcher.html. 2014. Accessed: 08-08-2024.

[11] Ganatra, S., *React Router Quick Start Guide: Routing in React Applications Made Easy* (Packt Publishing Ltd, 2018).

[12] Garreau, M., i in., *Redux in action* (Simon and Schuster, 2018).

[13] Hámori, F., *The History of React.js on a Timeline*, https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/. Accessed: 08-08-2024.

[14] Ilyushin, E., Namiot, D., *On memory management in javascript applications*, International Journal of Open Information Technologies. 2015, volume 3, 10, pages 11–15.

[15] Le, T., *Comparing state management between redux and zustand in react*, LAB University of Applied Sciences. 2023.

[16] Le, T., *Comparison of state management solutions between context api and redux hook in reactjs*, Metropolia University of Applied Sciences. 2021.

[17] MUI, *Overview - Material UI*, https://mui.com/material-ui/getting-started/. Accessed: 12-08-2024.

[18] Patil, K., Javagal, S.D., *React state management and side-effects–a review of hooks*, Yearbook of the Association for Computational Linguistics. 2022.

[19] Podila, P., Weststrate, M., *MobX Quick Start Guide: Supercharge the client state in your React apps with MobX* (Packt Publishing Ltd, 2018).

[20] Poimandres, *Comparison - Jotai, Primitive and Flexible State Management for React*, `https://jotai.org/docs/basics/comparison`. Accessed: 16-08-2024.

[21] Pronina, D., Kyrychenko, I., *Comparison of redux and react hooks methods in terms of performance*, Proceedings http://ceur-ws. org ISSN. 2022, volume 1613, page 0073.

[22] Saleor, *Saleor Documentation*, `https://docs.saleor.io/`. Accessed: 10-08-2024.

[23] StackOverflow, *Popularity of Web Frameworks and Technologies*, `https://survey.stackoverflow.co/2024/technology/#1-web-frameworks-and-technologies`. 2024. Accessed: 08-08-2024.

[24] StatCounter, *Browser Market Share Worldwide*, `https://gs.statcounter.com/browser-market-share`. Accessed: 08-08-2024.

[25] ThemeSelection, *JavaScript Usage Statistics 2023*, `https://medium.com/quick-code/javascript-usage-statistics-2023-thatll-blow-your-mind-2af8bce5a4f7`. Accessed: 08-08-2023.

[26] Tómasdóttir, K.F., Aniche, M., Van Deursen, A., *The adoption of javascript linters in practice: A case study on eslint*, IEEE Transactions on Software Engineering. 2018, volume 46, 8, pages 863–891.

[27] Tran, K., *State management in react*, Vaasa University of Applied Sciences. 2023.

[28] Ventura, L., *Analysis of redux, mobx and bloc and how they solve the state management problem*, Polytechnic University of Milan. 2021.

[29] Walton, P., *Total Blocking Time (TBT) : Articles : web.dev*, `https://web.dev/articles/tbt`. Accessed: 12-08-2024.

[30] Walton, P., Pollard, B., *Largest Contentful Paint (LCP) : Articles : web.dev*, `https://web.dev/articles/lcp`. Accessed: 12-08-2024.

[31] Wieruch, R., *The Road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React. js and Node. js* (Robin Wieruch, 2018).

[32] Zehra, F., Javed, M., Khan, D., Pasha, M., *Comparative analysis of c++ and python in terms of memory and time*, Preprints. 2020.

[33] Zetterlund, L., Tiwari, D., Monperrus, M., Baudry, B., *Harvesting production graphql queries to detect schema faults*, w: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)* (IEEE, 2022), pages 365–376.