ACTA · GRAPHICA
ISSN: 0353-4707
e-ISSN: 1848-3828
ZNANSTVENI ČASOPIS ZA TISKARSTVO I GRAFIČKE KOMUNIKACIJE
JOURNAL FOR PRINTING SCIENCE AND GRAPHIC COMMUNICATIONS

Scientific article

Characteristics and Comparison of Global State Management Tools in React applications

Sanja Brekalo[1], Klaudio Pap[2], Florijan Kos[1]

[1] Međimurje University of Applied Sciences in Čakovec; Bana Josipa Jelačića 22a, 40000 Čakovec, Croatia
[2]University of Zagreb, Faculty of Graphic Arts; Getaldićeva 2, 10000 Zagreb, Croatia
Correspondence: sanja.brekalo@gmail.com

Abstract:

This paper analyses and compares four tools for global state management in React applications, namely Context API, Zustand, Redux and MobX. The aim of the comparison was to define the advantages and limitations of each tool in different application scenarios. The Context API is a built-in tool for managing global application state that solves the problem of passing props through components – prop drilling. Other analysed tools need to be added separately to the project, and they were created as a solution to the limitations of the Context API. The results showed that Context API is best suited for smaller and medium-complex projects, Zustand is optimized for all project sizes, while Redux and MobX are more applicable for large and complex applications with complex global state. The Context API can become limiting in complex applications due to performance issues, context losses, or the lack of a solution to directly prevent the Zombie Child problem. Other tested tools have built-in solutions to the mentioned problems. Zustand has proven to be a comprehensive solution with its simple code base applicable to different sizes of applications. The advantage of Redux is its Flux architecture, which increases the boilerplate code, but also makes application development suitable for large teams and projects with high demands for predictability and strictly defined state management. MobX offers a flexible and reactive approach, which simplifies working with complex states, but in certain cases it can be less predictable compared to Redux. In conclusion, this paper contributes to the understanding of the key features of state management tools and offers guidelines for their selection, enabling developers and development teams to make informed decisions about tool selection based on technical and project requirements.

Keywords:

React, Context API, MobX, Zustand, Redux

1. Introduction

React.js is a popular open-source JavaScript library created by Facebook. It is used for building user interfaces for websites and native applications that run in a Node.js environment. In React, applications are developed using components that construct the interface. The division of an application into components follows the principles of code reusability, with each component serving as a building block of the application. Each component is responsible for rendering a part of the HTML code for the interface it builds. React uses

JavaScript to construct the user interface and for asynchronous communication with the server. Asynchronous communication is used to fetch data from API endpoints, read from a database, or execute code on the server side of the application [1, 2]. As React applications grow in complexity, refactoring becomes an essential practice to improve maintainability and scalability. Research shows that developers frequently restructure component hierarchies and optimize state management to enhance code quality and long-term maintainability [3].

React changes the approach to web development by enabling the creation of Single Page Applications (SPA). A Single Page Application loads only one HTML document and updates parts of the page using JavaScript, without the need to reload the entire page from the server when navigating through links or other interactive elements. All the necessary code, including HTML, JavaScript, and CSS, is downloaded during the initial page load. This approach makes the page more reactive and similar to a desktop application, while also improving performance [4]. Understanding the hierarchy of React components is essential for efficient state management, as it helps developers analyze component relationships and data flow. Visualization tools like React-bratus have been proposed to aid developers in navigating component structures and identifying architectural inefficiencies [5].

React employs DOM manipulation techniques that enable faster rendering of dynamic web applications. It uses a virtual DOM, which is essentially a copy of the browser's real DOM stored in memory, to identify the parts of the page that need to be updated after an event occurs. React maintains two versions of the virtual DOM for comparison: the new DOM created after the event is executed in the application, and the old DOM currently displayed in the browser. React compares these two versions of the virtual DOM in memory, a process that is faster than working directly with the DOM displayed in the browser. This process is known as reconciliation. React identifies the differences between the two versions of the virtual DOM, and if changes are detected in the new virtual DOM, the real DOM is updated only in the places where those changes occurred. This approach enables more efficient and faster updates to the user interface, giving React applications the feel of mobile apps [6, 7].

In React applications, rendering parts of a page and creating a new virtual DOM is possible only when the component's properties (props) or state changes. Props are values passed to a component from its parent component, and changes to their values trigger the re-rendering of the component. Components in an application have hierarchical relationships. The root component of an application includes all other components as children and descendants. Components receive external data via props from parent components. React's data flow between components is one-directional (from parent to child only). Props are read-only and cannot be modified by the component that receives them. State, on the other hand, allows components to manage their own data. State data can be modified only by the component that defines it and is private (it cannot be directly accessed externally). A state change can also be triggered by a child component but only through methods defined in the parent component [8]. In essence, props are used for passing data, while state is used for managing component's data [9]. State changes typically occur based on user input, event triggers, and similar interactions. When the state changes, React is notified, and the DOM is re-rendered—not the entire DOM, only the components with the updated state. Since the introduction of React Hooks , state can be used in both class and functional components [10].

State management is a crucial concept in modern application development, particularly in React applications, where it plays a key role in optimizing performance

and enhancing the user experience. As the complexity of an application grows, so does the complexity of managing its state. In such cases, it is advisable to use tools that simplify state management, improve application scalability, and facilitate maintenance. Poorly structured React applications can accumulate technical debt, making state management more error-prone and harder to maintain. Studies have identified common 'code smells' that arise in React-based web applications due to suboptimal component design and improper state handling [11]. There are several tools focused on state management that can be implemented in React projects. When choosing among these tools, the main challenge is assessing which tool will provide optimal application performance while maintaining ease of use. This research focuses on identifying and specifying the characteristics of individual tools to simplify the selection of tools and approaches for state management in React applications.

For the purposes of this research, four state management tools were selected: Context API, Zustand, Redux, and MobX. The paper analyses their key features to provide guidelines for selecting the most suitable tool based on the specific needs of projects. The study aims to answer the question: Which state management tool offers the optimal combination of performance, simplicity, and flexibility for React applications of various sizes and complexities? The goal is to explore the characteristics of these tools through comparison and define recommendations for their optimal use in React projects. It is assumed that each of the analysed state management tools—Context API, Zustand, Redux, and MobX—has specific advantages and limitations, and their applicability depends on the size, complexity, and requirements of React applications. The goal is to explore the characteristics of these tools and determine their optimal use cases to enable the selection of a tool that best balances performance, simplicity, and scalability across different projects.

[1]*The Document Object Model (DOM) in a browser represents the structure and content of a web document. DOM is a programming interface for web documents.*

[2]*Hooks are a feature in React introduced in version 16.8, which allow the use of state and other React functionalities within functional components.*

1.1. Managing state in React applications

With React, the user interface cannot be directly modified from the code and is updated only by changing the state in response to user input. Sometimes, it is necessary to update the state of two components simultaneously. To achieve this, the state is moved from the components to their closest common parent component and then passed down via props. This process is known as "lifting state". Passing props from lifted state can become challenging if a prop needs to be passed through many components or if multiple components require the same information [12]. Figure 1 illustrates this scenario, where props are passed through the hierarchy to components that utilize the prop from the lifted state.

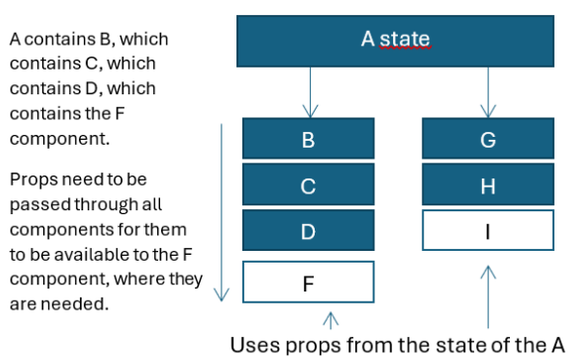

Figure 1. Prop drilling through the hierarchy

Context API, Zustand, Redux and MobX enable lifting state and subscribing components to the state, removing the need to pass props through child components until it reaches the one that needs it. This makes the state accessible to any component in the tree below without explicitly passing it through props and thus eliminating the need to refresh the entire tree of components, so the ones that do not consume the props don't need to be changed. Refreshing components consumes

significant time and device resources, reducing unnecessary component updates (caused by passing props) can significantly improve performance. Today, many popular state management libraries are available for React projects. They all aim to improve state management, but each focuses on one or more specific areas. Therefore, it's not about choosing the best library for everything but selecting the one that best suits the project's needs and characteristics.

Global state is used in applications to share state and data that need to be accessible across multiple components. This is particularly useful in scenarios where different parts of the application—or multiple components—require the same data or state. Examples of such use cases include:

• Tracking global application information: Refers to global state that must be accessible to various parts of the application, such as a shopping cart in an e-commerce app or the currently playing song in a music app.

• User authentication: Tracks user authentication status to determine which parts of the application are displayed and what management rights the user has.

• Managing themes and visual appearance: Enables toggling between light and dark modes, changing fonts, and other theme-related settings.

• Localization: Tracks the selected language and localization settings.

• User preferences: Monitors user settings such as notification status or other preferences.

• Modal states: Manages the opening and closing of modals across different parts of the application.

• Managing API requests: Stores data or loading statuses (e.g., loading or error states) to display responses in multiple places in the application.

• Breadcrumb navigation: Displays breadcrumbs to help users navigate back to previous pages.

•Step progress tracking: Tracks progress through steps in an application (e.g., registration steps or the checkout process).

1.1.1. React Context API

React Context API was designed to simplify working with state and eliminate the need to pass props through the component tree from higher-level components to lower-level components where they are needed. By using the Context API, state management is simplified as it removes the need for prop drilling, allowing components to directly access the data they need from the state. This makes sharing data across the component tree easier and reduces the likelihood of errors [13] .

In the Context API, a context is first created using the createContext() method. A Provider is then defined as a component that utilizes the context, and it is set to wrap a parent component of the tree section where the context should be made available. The Provider component contains a value property that holds the data to be shared across components. When the Provider's value changes, all descendants using the context are re-rendered. A Consumer component enables any descendant to use the context. In modern React, the useContext hook is commonly used instead of the Consumer component [14].

1.1.2. Zustand

Zustand is a library developed by the creators of the popular Immer library. Zustand is an external library built on Context API and hooks. It stands out for its simplicity, featuring a minimal API that makes it easy to learn and use, while requiring very little boilerplate code for setup. Based on React Hooks, Zustand integrates naturally into modern React applications and offers excellent TypeScript support with precise type inference. Zustand supports middleware and enables easy integration with middleware functions for additional functionalities through its zustand/middleware library. Some supported middleware includes devtools, which logs

state changes and helps track state updates, as well as integration with the Redux DevTools browser extension for monitoring application state and enabling Time-Travel Debugging . Another middleware, persist, allows state to be stored in local storage (e.g., LocalStorage) or other storage systems. Zustand also supports creating custom middleware for specific application needs, such as state protection or action customization. Zustand automatically handles state updates, which can be partial and mutable, as well as subscriptions and efficient re-rendering of components. It uses selector functions, which allow precise retrieval of specific parts of the state within components rather than accessing the entire state. Selectors optimize performance by enabling components to subscribe only to specific parts of the state, avoiding unnecessary re-renders when other parts of the state change [15, 16].

1.1.3. Redux and Redux Toolkit

Redux is an external library for managing and updating the state of an application, which can be used not only with React but also with other JavaScript frameworks. It defines rules that ensure the centralized state store can be updated only in a predictable way. Compared to Context API, Redux requires more boilerplate code and has a steeper learning curve. However, the introduction of Redux Toolkit—a simpler and easier version of Redux—has significantly reduced this complexity. The Redux library has the most established ecosystem and provides a wide variety of middleware. It also offers features like Time-Travel Debugging with Redux DevTools and tools for handling side effects. It is often considered more suitable for larger applications with complex state management needs.

React is based on the Flux architecture, which enables a predictable and consistent flow of data throughout the application. Flux relies on a unidirectional data flow, meaning data moves in a single direction without feedback loops. The main components of Flux architecture include:

• Actions - operations that trigger changes in the application and are created by application events using action creators, which are functions that return action objects.

• Dispatcher - manages the actions by forwarding them to the appropriate reducer.

• Reducers - are pure functions that take the current state and an action as input and return an updated state. In Redux, reducers create a new copy of the state rather than modifying the existing one. They must not produce side effects or perform asynchronous tasks, like accessing a database or fetching API data.

• Stores - store the application state, while Consumers are the components that use the stored data [17].

To handle asynchronous operations in Redux, middleware tools like Redux Thunk or Redux Saga are commonly used. These tools allow asynchronous code to run outside reducers, and once the data is retrieved or processed, they dispatch actions to inform reducers about state changes. This separation ensures that reducers remain pure functions.

Redux is typically used in applications with many state variables shared across different components, where state changes frequently and the logic for updating the state is complex. Redux is especially suitable for medium to large applications and for teams of developers [18].

1.1.4. MobX

MobX is a state management library that can simplify state management, improve performance, and enhance the scalability of React applications. The key concepts in MobX relate to the application state, where parts of the state are defined as observables—variables that automatically generate derivations,

[3] *Time-Travel Debugging is a technique that allows navigating through the history of state changes in an application. Using Redux DevTools, developers can "travel through time" by rewinding or replaying state changes, returning the application to previous states. This is highly useful for debugging, as it enables a detailed review of every step the application took to reach its current state.*

or computed values, based on the state. Other important concepts in MobX include reactions, which are functions wrapped in the autorun() function and automatically triggered when changes occur in observable values, and actions, which modify the state. Derivations are values automatically calculated from the application's state. Reactions are similar to derivations, but the main difference is that they do not produce values; instead, they automatically perform a specific task. Actions modify the state and ensure that derivations and reactions are automatically processed after changes. To enable the automatic execution of derivations and reactions, state variables must be defined as observables. MobX integrates well with existing React Hooks code, allowing MobX state to be combined with Hooks state [19-22].

2. Methodology

A qualitative research approach was applied, enabling a detailed analysis of the features of state management tools in React applications. Tools - Context API, Zustand, Redux and MobX - were selected based on their popularity, available documentation, and frequency of use in React projects and analysed based on their technical features and practical application. Key aspects of state management were analysed to assess the applicability of these tools in various application development scenarios. The study also considers previous work on React component hierarchy visualization, as an effective way to understand how state flows through an application and optimize its management [5]. The tools were examined from multiple perspectives, including how their implementation affects application performance, ease of use in terms of required boilerplate code, handling complex states in terms of partial updates and state mutability, the complexity of managing asynchronous operations, and the suitability of the tool for different project sizes and levels of state complexity. Recent research highlights the importance of identifying structural weaknesses in React applications to improve maintainability and prevent common pitfalls associated with large-scale state management [11]. Apart from managing client-side state, real-world applications often require synchronization between the client and server. Research has identified frameworks that help address this challenge by ensuring consistency between UI state and backend data [23]. State management decisions often influence how frequently a React application requires refactoring. Recent studies have identified common patterns in React refactoring, such as restructuring component hierarchies and optimizing state flow to improve code sustainability [3].

The analysis was based on data collected from technical documentation and usage guides for each tool. Practical implementations of each tool were also conducted in a simulated development environment, focusing on real-world use cases. As a result of the research, a tabular analysis was created, highlighting the advantages and disadvantages of each tool.

The limitation of this study is in the fact that it does not compare all available state management tools but focuses only on the most used and popular ones. Additionally, performance was not empirically tested on large-scale projects but analysed through simpler examples and documentation. Future research should expand the analysis to include less popular tools and incorporate empirical performance testing in large and complex projects to provide additional insights into their scalability and efficiency.

3. Results

Table 1. provides an overview of the key features of the four analysed tools for managing state in React applications. The features of each tool describe the programming capabilities used for managing the application's global state. These listed characteristics aim to achieve the main goal of this study: simplifying the selection of

the tool that best suits the project based on technical requirements for state management. As shown in Table 1, the tools differ in terms of flexibility and adaptability to specific application development scenarios.

Table 1. *Comparison of tools for managing global state*

No.	Feature	Context API	Zustand	Redux + Middleware	MobX
1	Suitable for application sizes	Small and medium	Small, medium, large	Medium and large	Medium and large
2	Additional installation required in React	No	Yes	Yes	Yes
3	Flux architecture	No	No	Yes	No
4	Solution for context loss	No	Yes	Yes	Yes
5	Automatic prevention of Zombie Child issue	No	Yes	Yes	Yes
6	Partial state updates (merged state)	No	Yes	No	Yes
7	Mutable state updates	No	Yes	No	Yes
8	Handling asynchronous operations	Manual	Yes	Middleware	Manual
9	Boilerplate code size	Medium	Smallest	Largest	Larger
10	Saving state to local storage	Manual	Middleware persist	With redux-persist	Manual

4. Conclusion

4.1. Suitability of Tools for Different Application Sizes

State management tools differ in their suitability depending on the size and complexity of the application. Context API provides a native solution suitable for small and medium applications. It is lightweight for implementations in smaller applications and straightforward to use. However, in more complex projects, a limitation of Context API is the re-rendering of all components consuming a specific context, which can affect performance. Zustand is designed to be lightweight and easy to use. Its flexibility and scalability support make it an excellent choice for projects of various sizes. The ability to perform partial and mutable state updates, middleware support, and performance optimizations make it more suitable for larger applications compared to Context API. Redux and MobX are more appropriate for large projects due to their robust handling of complex states. In addition to handling state locally, applications that rely on frequent interactions with a backend require efficient state synchronization mechanisms. Research has demonstrated that structured synchronization frameworks improve data consistency and reduce re-rendering issues caused by inconsistent client-server states [23]. Redux's Flux architecture strictly defines state management, which is beneficial for collaborative team work on the same application. MobX, being more flexible than Redux, works well in large projects due to its built-in reactivity and support for observable variables.

4.2. Ease of Implementation in Projects

One critical factor in choosing a state management tool is the complexity of its implementation. Context API differs from

other tools in its simple implementation and built-in support without requiring additional installations. Zustand needs to be installed and is similarly easy to integrate with minimal boilerplate code, utilizing React hooks. Redux and MobX, however, require installation and significantly more setup code, which can be justified for large teams and complex states.

4.3. Flux Architecture

Redux is the only tool among those analysed that is based on the Flux architecture. Flux enables unidirectional data flow, reducing the likelihood of unexpected states and errors during application operation. This predictable architecture is particularly applicable to applications with complex states and larger team environments. MobX enforces rules and methods for changing state but does not follow the strict principles of the Flux architecture, making it more flexible but potentially less predictable. Context API and Zustand have no direct connection to Flux architecture.

4.4. Addressing Context Loss

A drawback of Context API can be context loss in React applications with different rendering trees. This occurs when data shared via Context API cannot be shared between different renderers or separate component trees. Context API operates within a single rendering tree, meaning data accessible through a Context Provider in one part of the application will not automatically be available to components in another tree. For example, if an application uses different renderers, such as React DOM for web and another renderer (e.g., React Native, Canvas, React Three Fiber), each has its own component tree. Consequently, Context API cannot share data across these trees. For this reason, Context API is not suitable for sharing global state in complex applications with multiple renderers. In such cases, Zustand, Redux or MobX are recommended as they allow state sharing across multiple application trees or renderers, ensuring state availability throughout the application.

4.5. Preventing the Zombie Child Problem

With React, the user interface cannot be directly modified from the code and is updated only by changing the state in response to user input. Sometimes, it is necessary to update the state of two components simultaneously. To achieve this, the state is moved from the components to their closest common parent component and then passed down via props. This process is known as “lifting state”. Passing props from lifted state can become challenging if a prop needs to be passed through many components or if multiple components require the same information [12]. Figure 1 illustrates this scenario, where props are passed through the hierarchy to components that utilize the prop from the lifted state.

4.6. Partial State Updates

Partial state updates refer to the ability to update only a part of the state without replacing the entire state object. This approach can improve application performance, as only components dependent on the changed part of the state are updated rather than the entire application. Since a new copy of the entire state is not created, but only a part is updated, memory usage is optimized, and code handling becomes simpler especially in applications that use complex state objects.

When using Context API, changes within a context trigger re-rendering of all components connected to it. Splitting state into multiple contexts can achieve a similar effect, improving performance by updating only components using a specific context. However, splitting state across multiple contexts can introduce additional complexity in larger applications. Redux does not directly support partial state updates but achieves a similar effect through reducer composition. State changes affect only components subscribed to a specific state slice. The Immer library, included with Redux Toolkit, simplifies working with immutable state by allowing code that appears to mutate

state, while Redux Toolkit handles creating new copies in the background. Zustand and MobX support partial state updates. Components are updated only when the part of the state they are subscribed to changes, improving application performance. This enables flexible and optimized control over complex states.

4.7. Mutable State Updates

Mutable state updates involve directly modifying the existing state object without creating a new copy for every change. This approach can simplify code and improve performance by avoiding additional data copying. Speed improvements can be achieved, especially when many small state changes occur, as mutable state reduces memory allocations. However, immutability has advantages in more complex systems, as it ensures state consistency and predictability.

Zustand and MobX support mutable state updates. They manage state outside React's render cycle and notify components when updates occur, facilitating mutable state updates without significant risk. Context API and Redux require immutable state to ensure consistency and predictability. However, tools like Immer allow writing code as if state is being updated mutably while maintaining immutability in the background.

4.8. Handling Asynchronous Operations

Managing asynchronous code in JavaScript ensures tasks run in the background without blocking the application, enhancing user interaction. However, asynchronous state changes in React applications can cause issues as updates are not immediate, and the order of changes is not guaranteed. Tools like Zustand, Redux, and MobX offer optimized solutions for managing asynchronous operations, whereas Context API requires manual handling.

4.9. Boilerplate Code Size

Compared to Context API, Redux and MobX, Zustand requires the least code for setup and working with a store. While Context API is simpler than Redux and MobX, it typically requires additional setup, such as creating contexts and provider components, as well as manually defining functions for state updates, especially for more complex states or asynchronous management. Redux has the most boilerplate code due to its strict structure, which involves creating actions, reducers, configuring the store, and often additional middleware (e.g., Redux Thunk for asynchronous tasks). Although Redux Toolkit reduces the amount of boilerplate, it is still more complex than Zustand. MobX, being flexible and reactive, requires setting up "observable" objects, actions, and sometimes decorators to track changes, adding complexity compared to Zustand and Context API.

4.10. Storing State in Local Storage

State persistence is particularly useful for applications that need to maintain state between sessions or in environments with limited network access. Zustand has built-in support for state persistence through its persist middleware, which allows quick and easy state preservation between page reloads or reopening the application. It is configured with minimal additional code and requires no extra libraries. Context API, Redux, and MobX lack built-in functionality for state persistence, requiring manual handling. In Redux, the redux-persist library can be used for state persistence, offering a simple integration with additional configuration.

5. Conclusion

This paper analysed and compared four tools for managing global state in React applications, highlighting their advantages and disadvantages. The tools differ in their approach to implementing global state management, making them optimally suited for projects of varying sizes and levels of state complexity. Context API and Zustand are suited for smaller projects, all tested tools are optimized for medium-sized applications,

while Zustand, Redux and MobX are well-suited for scalability and managing complex states in larger applications.

Context API stands out for its simplicity of implementation, as it is a built-in feature of the core React library. The other tools require installation within the project. However, Context API can become limiting in complex applications due to performance issues, context loss, or the lack of built-in solutions for preventing the Zombie Child problem. The other tested tools provide built-in solutions for these issues. Zustand and MobX support partial and mutable state updates. Zustand further requires minimal boilerplate code, making it flexible and easy to use. Additionally, it offers a significant advantage over the other tools with its built-in support for managing asynchronous code and a simplified method for persisting state to local storage. Redux, thanks to its Flux architecture and rich ecosystem, provides the best support for teams working on large and complex projects. Solving certain problems with Redux often requires additional tools and middleware, and it has the largest amount of boilerplate code among the tested tools. MobX offers an alternative to Redux for applications that require high flexibility in state management, with automatic reactivity and slightly less boilerplate code. Persisting state to local storage is simplified in Redux and Zustand with the use of additional tools.

The hypothesis has been confirmed, as the analysis successfully identified the unique strengths and limitations of each state management tool, providing clear recommendations for their optimal use based on the size, complexity, and requirements of React applications. This study provides guidelines for choosing a state management tool for React applications based on the size and complexity of the project as well as its technical requirements. Future research could include performance testing on large-scale projects and an analysis of less commonly used state management tools.

Literature

[1] HubSpot. What is React.js? Uses, Examples, & More. HubSpot [Internet]. 2024 [Accessed July 2, 2024]. Available from: https://blog.hubspot.com/website/react-js.

[2] Gackenheimer C. Introduction to React. Apress; 2015.

[3] Ferreira F., Borges H.S., Valente MT. Refactoring React-based Web Apps. J Syst Softw. 2024;192:111362.

[4] BairesDev. React Single Page Application. BairesDev [Internet]. 2024 [Accessed July 2, 2024]. Available from: https://www.bairesdev.com/blog/react-spa-single-page-application/.

[5] Boersma S., Lungu M. React-bratus: Visualising React Component Hierarchies. In: VISSOFT 2021 - Working Conference on Software Visualization. IEEE; 2021. p. 130-134.

[6] GeeksforGeeks. ReactJS Virtual DOM. GeeksforGeeks [Internet]. 2023 [Accessed July 2, 2024]. Available from: https://www.geeksforgeeks.org/reactjs-virtual-dom/.

[7] Maratkar PS, Pratibha A. React JS–An Emerging Frontend JavaScript Library. Iconic Res Eng J. 2021;12(4):99-102.

[8] Shaik V. Understanding the Concept of “State” in React. Medium [Internet]. 2024 [Accessed July 2, 2024]. Available from: https://medium.com/@vaheedsk36/understanding-the-concept-of-state-in-react-4f3461a1c7c4.

[9] Eygi C. React.js for Beginners—Props and State Explained. freeCodeCamp [Internet]. 2024 [Accessed July 2, 2024]. Available from: https://www.freecodecamp.org/news/react-js-for-beginners-props-state-explained/.

[10] GeeksforGeeks. Differences between Functional Components and Class Components. GeeksforGeeks [Internet]. 2023 [Accessed July 3, 2024]. Available from: https://www.geeksforgeeks.org/differences-between-functional-components-and-class-components/.

[11] Ferreira F, Valente M.T. Detecting Code Smells in React-based Web Apps. Inf Softw Technol. 2023;155.

[12] React Documentation. Managing State. React [Internet]. 2024 [Accessed July 3, 2024]. Available from: https://react.dev/learn/managing-state.

[13] Matéu.sh. React Context API Explained with Examples. freeCodeCamp [Internet]. 2024 [Accessed October 28, 2024]. Available from: https://www.freecodecamp.org/news/react-context-api-explained-with-examples/.

[14] Thanh L. Comparison of State Management Solutions between Context API and Redux Hook in ReactJS. Metropolia University of Applied Sciences, Bachelor's Thesis; 2021.

[15] Onix. Zustand State Management for React. Medium [Internet]. 2022 [Accessed July 11, 2024]. Available from: https://medium.com/@onix_react/zustand-state-management-for-react-feef64b2555e.

[16] Anujkumarsinh D., Apeksha J., Pradeep K.S. Application State Management (ASM) in the Modern Web and Mobile Applications: A Comprehensive Review. arXiv Preprint arXiv:2407.19318; 2024.

[17] Kumah E.F. How to manage state in a React app using Redux. DEV Community [Internet]. 2023 [Accessed November 8, 2024]. Available from: https://dev.to/efkumah/how-to-manage-state-in-a-react-app-using-redux-5pc.

[18] Banks A., Porcello E. Learning React: Functional Web Development with React and Redux. O'Reilly Media; 2017.

[19] Maurya H. MobX with React: A Comprehensive Guide. Medium [Internet]. 2023 [Accessed November 8, 2024]. Available from: https://harish-git.medium.com/mobx-with-react-a-comprehensive-guide-23598bfa54f2.

[20] MobX Documentation. Ten-minute Introduction to MobX and React. MobX [Internet]. 2024 [Accessed November 13, 2024]. Available from: https://mobx.js.org/getting-started.

[21] Pavan P., Weststrate M. MobX Quick Start Guide: Supercharge the Client State in Your React Apps with MobX. Packt Publishing Ltd; 2018.

[22] Lu R. How to Improve State Management in React with MobX. Whitespectre [Internet]. 2024 [Accessed July 3, 2024]. Available from: https://medium.com/whitespectre/how-to-improve-state-management-in-react-with-mobx-568808ff86a4.

[23] Tagdiwala V., Bharoliya A., Aibin M. Robust Client and Server State Synchronisation Framework for React Applications: react-state-sync. In: Proc IEEE Can Conf Electr Comput Eng (CCECE). 2023. p. 1-6.

[24] React-Redux Documentation. Stale Props and Zombie Children. React-Redux [Internet]. 2024 [Accessed November 7, 2024]. Available from: https://react-redux.js.org/api/hooks#stale-props-and-zombie-children.