RS Global Journals

Scholarly Publisher
RS Global Sp. z O.O.
ISNI: 0000 0004 8495 2390

Dolna 17, Warsaw, Poland 00-773
Tel: +48 226 0 227 03
Email: editorial_office@rsglobal.pl

JOURNAL	World Science
p-ISSN	2413-1032
e-ISSN	2414-6404
PUBLISHER	RS Global Sp. z O.O., Poland

ARTICLE TITLE	OPTIMIZATION OF CROSS-PLATFORM APPLICATIONS USING THE REACT LIBRARY
AUTHOR(S)	Bezverhiy Olexandr, Kutsenko Olexandr
ARTICLE INFO	Bezverhiy Olexandr, Kutsenko Olexandr. (2024) Optimization of Cross-Platform Applications Using the React Library. *World Science*. 3(85). doi: 10.31435/rsglobal_ws/30092024/8218
DOI	https://doi.org/10.31435/rsglobal_ws/30092024/8218
RECEIVED	01 July 2024
ACCEPTED	14 September 2024
PUBLISHED	16 September 2024
LICENSE	This work is licensed under a **Creative Commons Attribution 4.0 International License.**

OPTIMIZATION OF CROSS-PLATFORM APPLICATIONS USING THE REACT LIBRARY

Bezverhiy Olexandr
doctor science, professor, National Transport University, Ukraine

Kutsenko Olexandr
Postgraduate, National Transport University, Ukraine

DOI: https://doi.org/10.31435/rsglobal_ws/30092024/8218

ARTICLE INFO

Received: 01 July 2024
Accepted: 14 September 2024
Published: 16 September 2024

KEYWORDS

React, React Native, Cross-Platform Applications, Optimization, Software Development.

ABSTRACT

In today's world, with the growing use of mobile devices, the importance of creating cross-platform applications that provide high performance and quality of user experience has increased significantly. Improving the creation of such applications using React and React Native opens up wide opportunities for developers due to the possibility of code reuse, high development speed and ease of innovation, and is especially relevant because it concerns not only the technological aspects of development, but also economic efficiency, quality of user experience and the speed of introducing innovations to the software market. Open source and active community communication allow rapid identification and resolution of issues, development of new features and enhancements, and the sharing of knowledge and experience between developers. This creates a positive environment for innovation and growth, promotes rapid technological progress, and improves the quality of end products.
The paper analyzes the key aspects and challenges associated with the creation of cross-platform applications, analyzes the main types of optimizations at different levels of development, including the component level, the state and data level, the level of working with APIs and external data, as well as the loading and code level. Special attention is paid to the integration of synchronous and asynchronous rendering to achieve optimal performance and user experience. Highlights the benefits of flexible state management using state managers, as well as the importance of image and media optimization to improve overall website performance.
The article notes the role of the developer community in the process of improvement and innovation. Developers who master these technologies and optimize their application can create products that not only meet the requirements of the time, but also form new standards in the field of mobile and web development. Further research and integration of the latest technologies into the development process of cross-platform applications will have a significant impact on the software industry.

Citation: Bezverhiy Olexandr, Kutsenko Olexandr. (2024) Optimization of Cross-Platform Applications Using the React Library. *World Science.* 3(85). doi: 10.31435/rsglobal_ws/30092024/8218

Problem Statement: Using React and React Native libraries for cross-platform app development stems from several trends:

1. Increase in mobile traffic and use of mobile apps: There is a consistent increase in the number of mobile device users worldwide, which compels the development of apps capable of satisfying the diverse needs of this audience. Cross-platform capabilities become a key response to this challenge, allowing developers to create applications that effectively function across different platforms.

2. Need to reduce time and costs of development: Developing separate app versions for each platform is time-consuming and expensive. Using React and React Native allows for significant optimization of resources, as developers can utilize the same code base to create apps that operate on different operating systems.

3. Demand for high performance and quality of applications.

4. Need for continuous updates and support of applications**.** Market conditions and user requirements are constantly changing, hence applications require regular updates and modernization.

Scientific research in optimizing the use of JavaScript, developing new development patterns, improving application architecture, and integrating with other technologies and platforms are critically important for achieving this goal**.** Collaborative efforts of scientists, developers, and the community can lead to significant progress in creating effective, accessible, and high-quality cross-platform applications that meet the needs of modern users and the software market.

Analysis of the main types of optimizations for a React application:

Optimization of React applications is a key element in increasing performance, ensuring interface operation speed, and improving the overall user experience. There are several levels at which React application optimization can be performed, each requiring the application of specific approaches and techniques. Let's consider the main types of optimizations at different levels:

Component Level:

Reusing components helps reduce code duplication and enhances code readability and maintainability.

Using React.memo for class components and React.PureComponent for functional components allows avoiding unnecessary renders by comparing props and state.

If a component displays the same result with the same props and state, it can be wrapped in a call to React.memo to enhance performance in some cases by memorizing the result. React.memo only checks if props have changed. If a function wrapped in React.memo has useState or useContext hooks in its implementation, it will still re-render when state or context changes. By default, it only shallowly compares complex objects in the props object. If you want to control the comparison process, you can also provide a custom comparison function by placing it as the second argument.

Using shouldComponentUpdate allows control over the rendering process by comparing current and next props and state.

Asynchronous Rendering with React.Suspense and Lazy Loading:

React.Suspense and React.lazy offer a mechanism for asynchronous component rendering, allowing components to wait for necessary data or other components before they are displayed. This is particularly useful for improving performance through lazy loading of components that are not needed immediately after the app loads.

React.lazy takes a function that must call a dynamic import(). It must return a promise that resolves to a module with a default export containing a React component.

A lazy component should then be rendered within a Suspense component body. This allows us to display fallback content (e.g., a loading indicator) while we wait for the lazy component to load.

Using React.memo and React.PureComponent to prevent unnecessary renders and the shouldComponentUpdate method to control component updates.

State and Data Level:

State Management: Efficient state management using contexts or state managers like Redux or MobX can significantly reduce the number of renders and simplify data flow.

Redux library is used to manage the state of the application, which allows reducing the amount of code and making the application easier to understand and extend.

Lazy Data Loading: Applying lazy loading for data and components can improve app load time and resource use efficiency.

API Level and External Data:

Caching Responses: Caching server responses can reduce the number of requests to the server and speed up data display to the user.

Optimizing Requests: Avoid excessive or unnecessary server requests by aggregating data or using debounce/throttle for event handling.

Loading and Code Level:

Code Splitting: Using code splitting allows dividing the code into smaller parts that can be loaded as needed, reducing app load time.

Lazy Loading Components: React.lazy and Suspense allow organizing the lazy loading of components that are not critical for the initial rendering.

Using Web Workers for Heavy Task Processing: For processing complex calculations without blocking the main UI thread, Web Workers can be used, which helps improve interface responsiveness.

Image and Media Optimization:

Image compression, using formats optimized for the web (such as WebP), and lazy loading of media can significantly improve performance by reducing the volume of data transferred.

Optimizing React applications at all these levels allows achieving significant performance improvements and ensuring a better user experience. It is important to note that optimization is a process that requires continuous analysis, testing, and refinement.

Application of Synchronous and Asynchronous Rendering for optimization:

Using synchronous and asynchronous rendering in React should be balanced to provide optimal performance and user experience. Synchronous rendering can be effective for displaying content that is immediately available or for very critical parts of the interface, while asynchronous rendering is ideal for optimizing the loading of additional data, modules, or components that are not necessary for the initial display.

In React, both synchronous and asynchronous rendering can be used for different parts of your application. This is possible thanks to the state management and rendering mechanisms that React provides, allowing developers to optimize performance and user experience.

Synchronous Rendering: This standard approach in React involves components being rendered one after another in the main execution thread. When a component receives new props or its state updates, React re-renders the component and its children synchronously. This means that the user interface is blocked until the entire rendering process is completed.

Asynchronous Rendering: Allows components to await the loading of data or other resources without blocking the user interface. This can be implemented using features like React.lazy for lazy loading components and React.Suspense, which allows components to "wait" for necessary content before rendering. Concurrent Mode is another feature that allows React to work on multiple tasks asynchronously, improving the responsiveness of the app.

Examples of using synchronous and asynchronous rendering for optimization: Synchronous rendering is used for rendering the main skeleton of the app or critically important components that must be immediately available to the user. Asynchronous rendering is used for loading large components, modules that require additional data from the server, or functionality that is not used immediately after the app loads (e.g., modal windows, additional pages).

By wisely using these methods, significant improvements in app performance and user experience can be achieved, reducing load times and interface responsiveness. React provides flexible tools for optimizing rendering, and effectively using them allows achieving high performance even in large and complex applications.

Asynchronous component rendering in React is performed using a combination of React.lazy and <Suspense>. This allows delaying the loading of a component until it is truly needed, for example, during routing or deferred rendering of parts of the interface.

```
import React, { Suspense } from 'react';
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```

Synchronous rendering is more straightforward and easy to understand, but it can lead to delays in interface responsiveness, especially during large calculations.

Asynchronous rendering (Concurrent Mode) provides better interface responsiveness but requires more careful management of states and side effects in components.

Batching Updates**:** In synchronous rendering, state updates often occur immediately and one at a time. Asynchronous rendering allows more efficiently grouping updates, making rendering more efficient and less costly in terms of performance.

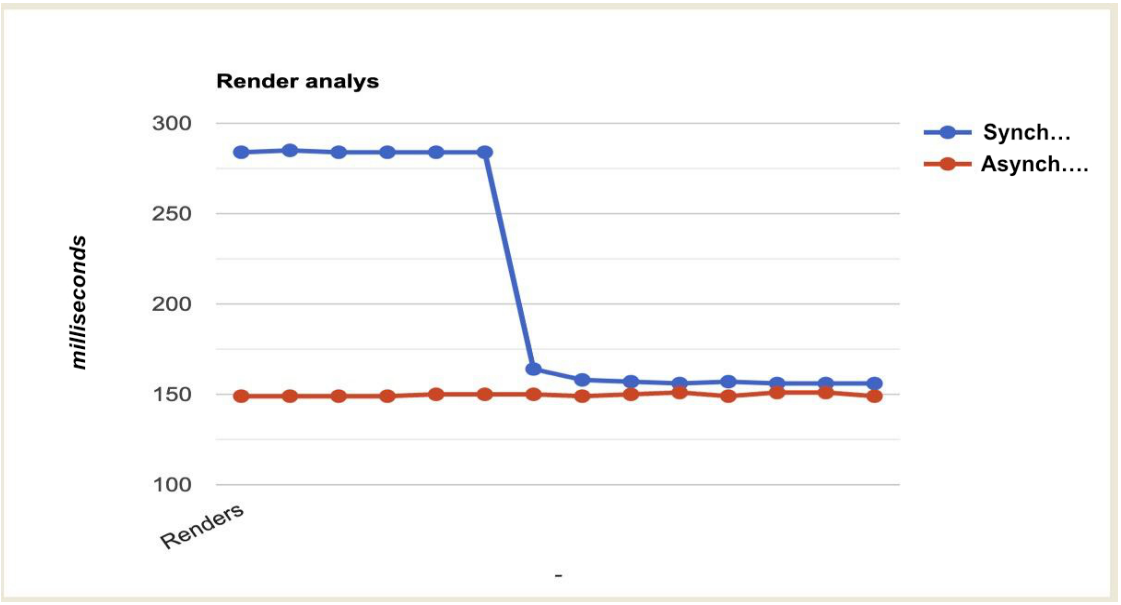

Figure 1. Comparison of Performance Between Synchronous and Asynchronous Rendering.

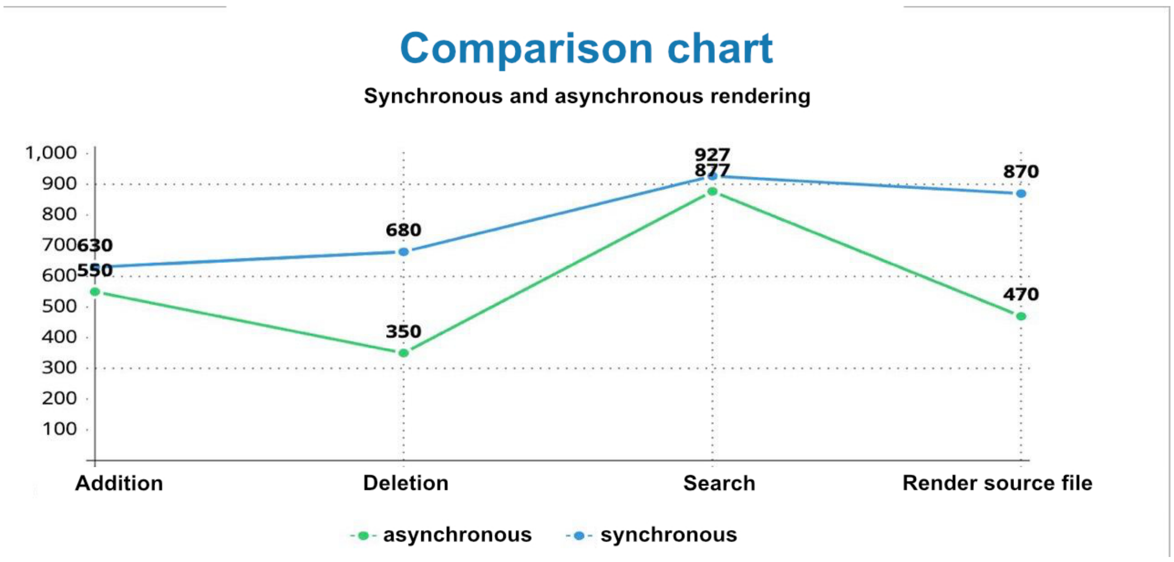

Figure 2. Comparison of Performance Between Synchronous and Asynchronous Rendering in Different Application Operating Modes.

Integrating synchronous and asynchronous rendering enhances development efficiency and enables responsiveness to rapidly changing market demands. The implementation of innovative approaches such as React's Concurrent Mode, deep integration with PWAs, and the expansion of <Suspense> capabilities opens new horizons for enhancing the functionality and accessibility of applications.

Based on the analysis, it can be concluded that further research and integration of the latest technologies in the development process of cross-platform applications will have a significant impact on the software industry. Developers who master these technologies and optimize their application can create products that not only meet the demands of the times but also set new standards in the field of mobile and web development.

REFERENCES

1. React.memo. Access mode: URL: https://uk.legacy.reactjs.org/docs/react-api.html#reactmemo.
2. React.lazy. Access mode: URL: https://uk.legacy.reactjs.org/docs/code-splitting.html#reactlazy.
3. Abramov D. (2015). Getting Started with Redux. URL access mode: https://egghead.io/courses/getting-started-with-redux.
4. Kang, J., Kim, Y., & Kim, D. (2017). A study on the implementation of cross-platform mobile application using react-native.// Journal of the Korea Academia-Industrial cooperation Society, 18(4), P.155-167.
5. UseEffect. Access mode: URL: https://uk.legacy.reactjs.org/docs/hooks-reference.html#useeffect.