Article

Web Front-End Application Performance Improvement Method Based on Component-Based Architecture

Yifan Yang [1,*]

[1] Viterbi school of Engineering, University of Southern California, CA, 90089, USA
* Correspondence: Yifan Yang, Viterbi school of Engineering, University of Southern California, CA, 90089, USA

Abstract: In the context of the continuous evolution of Web front-end technology, performance optimization has become the core work to improve user experience and enhance product value. This paper focuses on analyzing the performance bottlenecks encountered by current Web front-end applications, such as rendering bottlenecks, network bottlenecks, resource bottlenecks and computing bottlenecks, and introduces optimization suggestions. By designing front-end applications with a block architecture, you can reduce redundant rendering, optimize resource loading, split code and lazy loading, and improve the efficiency of state management to significantly improve application speed. At the same time, the influence of the block structure on the front-end application is analyzed, and how to optimize the development process and performance through good structure design is discussed.

Keywords: Web front-end; componentized architecture; performance improvement

Received: 24 April 2025
Revised: 27 April 2025
Accepted: 21 May 2025
Published: 22 May 2025

1. Introduction

In the Internet era, Web front-end applications have long served as the primary gateway connecting users to online services. As Web application functions become more and more complex, users have higher and higher requirements for Web application performance, so improving front-end performance is the first task developers must face. The performance bottleneck of a Web front-end application usually occurs in the aspects of rendering, networking, resource loading, and computing. Through in-depth analysis of these bottlenecks and reasonable optimization, the response time and user experience of the application can be effectively improved. This paper introduces the causes of Web front-end performance bottlenecks, and gives the corresponding performance optimization scheme through the componentized structure [1].

2. Concept of Componentized Architecture

2.1. Definition of Componentized Architecture

Component-based architecture is currently the most mainstream application organization method in Web front-end development. The core idea is to take components as the basic units of construction. It is applied in front-end frameworks such as React, Vue and Angular. A Component is a UI functional module with independent encapsulation, reusability and self-state management capabilities. It can carry specific interface structures, style definitions and behavioral logic, and can be flexibly combined and called in applications according to requirements.

The componentized architecture divides the entire Web application into multiple highly cohesive and low-coupled functional units. Components interact with information and transfer data through interfaces, achieving the elimination of logical coupling and repetitive coding problems between modules designed from the perspective of pages.

This organizational approach not only enhances the maintainability and scalability of the system, but also supports local rendering and progressive loading, significantly reducing the overall rendering cost and improving the application response efficiency.

In fact, when developing applications, the advantages of component-based architecture can often be realized through a complete set of auxiliary technologies. For example, virtual DOM can be used to accelerate the rendering speed, unidirectional data flow can be used to maintain the consistency of state changes, application composite apes can be used to increase code reusability and functional isolation. In addition, developers can also download components on demand through path configuration. State managers such as Redux and Pinia are utilized to achieve state sharing and centralized management among components, thereby further optimizing the overall performance and user experience [2].

2.2. Advantages of the Componentized Architecture

The component-based architecture in the construction of Web front-end, as a highly structured, low-coupled and highly cohesive development paradigm, can not only simplify the development process of the system, but also significantly improve the development efficiency. It divides the entire program into multiple functional units, each implementing its own user interface, behavior, and state control, and enables collaboration through standard interfaces.

Compared with the traditional "page-centered" construction mode, the component-based design architecture is more likely to achieve code reuse and function combination, reduce the coupling degree of the system, and enhance the operability of the system. In terms of performance, the component architecture supports on-demand rendering and lazy loading mechanisms, which helps to reduce memory usage and rendering overhead during the application startup stage. Especially for the commonly used front-end frameworks such as React and Vue nowadays, virtual DOM, composition APIs, and lifecycle control have all become key elements for optimizing performance. In addition, the component pattern essentially supports the integration of construction tools and automated testing [3]. Combined with modern build tools such as Webpack and Vite, it can effectively enhance the build efficiency, deployment flexibility and testability of the front-end system, laying the foundation for continuous integration. The following Table 1 summarizes the main technical advantages and performance of component-based architecture in the Web front end:

Table 1. The Main Technical Advantages and Performance of Component-Based Architecture in Web Front-End.

Advantage category	Specific manifestations	The impact on performance improvement
Module reuse and improvement of development efficiency	Independently encapsulated, with a clear structure, it can be reused in multiple pages or projects	Reduce redundant code, lower maintenance costs and improve development efficiency
Decoupling and maintainability have been enhanced	Each component has independent functions and communicates through props or events	Reduce global dependencies, simplify the debugging process, and enhance system stability
Support for local updates and rendering optimization	Support virtual DOM difference comparison and minimum rendering unit control (such as React's diff algorithm)	Reduce invalid redrawing and rearrangement to improve page response speed
Fine control of status management	Support local state isolation and centralized processing of global states (such as Vuex, Redux, etc.)	Avoid repetitive rendering to enhance interaction performance and resource utilization efficiency

The integration of construction and testing processes is convenient	It is easy to access build tools and automated processes, and supports on-demand loading, unit testing, etc.	Enhance the controllability of the module, accelerate the iteration speed, and facilitate regression and release
Adapt to the requirements of large-scale system expansion	Support modular organization, asynchronous components, and micro-front-end architecture	Meet the comprehensive requirements of structural layering and performance optimization for large-scale SPA projects

In conclusion, the componentized architecture not only enhances the engineering management capabilities of the front-end system, but also effectively supports the performance and user experience of web applications in complex environments through precise rendering management and modular management.

3. Web Front-End Performance Bottleneck Analysis

3.1. Render Bottle Pre

Web front-end rendering bottlenecks usually refer to performance issues that occur when browsers translate HTML, CSS, and JavaScript into visual content for users. Rendering includes parsing, DOM building, style calculation, layout, drawing, and merge steps. Each of these steps can lead to a display rendering bottleneck, resulting in slow page loading or long response times. Common rendering bottlenecks include complex CSS selectors or too many DOM elements that make style calculations and layouts expensive. Too much use of JavaScript can cause the rendering process to be blocked, especially if the JavaScript script to be executed when the website is launched can cause subsequent rendering to be delayed. In addition, unnecessary Reflow and Repaint can cause poor performance by repeatedly triggering the browser to rearrange the work. To remove the display rendering bottleneck, developers need to reduce the number of DOM elements, improve the selection efficiency of CSS selectors, prevent scripts that hinder rendering, and minimize unwanted Reflow and Repaint operations [4].

3.2. Network Bottleneck

Network bottleneck means that in Web front-end applications, due to the combined impact of Internet traffic restrictions, slow demand response, and lagging communication, the request for front-end data becomes slow, which hinders the opening time of web pages and customer experience. The network bottleneck is closely related to the response time and resolution time of the resource to be queried, especially when the user's network status is poor, the problem is more serious. Common problems such as too many HTTP requests, too much request data, and incorrect buffering applications can easily lead to network bottlenecks. Each HTTP request involves DNS resolution, link creation, and message sending, which increases the page opening time. Through reasonable buffering, merging requests, reducing data files and using CDN (Content distribution system), we can solve the problems caused by network bottlenecks. At the same time, HTTP/2 protocol and lazy loading can further improve network efficiency and reduce startup delay.

3.3. Resource Bottleneck

Web front-end resource bottlenecks are often caused by the excessive size or number of external resources such as images, videos, fonts, and JavaScript files, resulting in long page loading time, poor user experience. With the increasing dependence of network applications on various media and dynamic content, resource bottleneck has become one of the factors affecting performance. Common problems include: too large resource files causing content to fail to load in time, uncompressed and unoptimized images and videos, and the wrong download order [5]. A number of measures can be taken to optimize, such as loading images in gradient mode, compressing resources, using the appropriate image format (Web P), merging and separating JS/CSS files, etc. The use of SVG graphics, some font files, delayed loading/on-demand loading method can greatly reduce the burden of web resources and improve the performance of web pages.

3.4. Computing Bottleneck

In the front-end stage of the Web, if the JavaScript operation overhead is too large, it may lead to the phenomenon of slow loading or stagnation of the page, which is called the computation bottleneck. Computing bottleneck generally occurs in the need for a large number of data processing, a large number of DOM structure changes, a large number of animation calculations, etc. Large-scale computing operations on user devices may cause the web thread to freeze or not respond for a long time. Common reasons are: high-frequency DOM query, frequent event monitoring, heavy numerical operations and drawing pictures. The methods that can reduce the computation bottleneck include algorithm optimization, Web Worker sharing time-consuming computation to background threads, reducing DOM query operations, reducing repetitive operations, and using request Animation Frame to achieve better animation quality. In addition, minimizing the use of synchronous JavaScript and reasonably setting the execution sequence of tasks are of great help to prevent computing bottlenecks, and can maintain the fluency of the webpage rendering process, so as to ensure the good effect of the entire front-end program.

4. Methods to Improve the Performance of Web Front-End Applications Based on the Component-Based Architecture

4.1. Reduce Redundant Rendering

Redundant rendering is a common performance problem in front-end web applications, especially when large amounts of data change frequently or when there are too many components in the page. Redundant rendering refers to unnecessary components or parts of a page that are rendered again after their state has changed, consuming system resources and reducing response efficiency. To this end, developers can minimize the number of component updates and avoid prerendering the entire page every time the data changes. For example, should Component Update in React allows developers to determine whether to update components by detecting changes in state and properties. Pure Component reduces rendering by means of depth comparison, while the memo function is specifically designed for functional components to prevent duplicate rendering of components with similar properties and states. Virtual DOM technologies, such as those in React, reduce DOM changes by calculating changes in the user interface for comparison, and thus reduce the cost of redrawing, which is important for some complex and frequently changing pages. It is also possible to break up components so that each component only cares about its own state changes to reduce over rendering. This will not only greatly improve the performance of your application, but also improve the maintainability and reusability of your code.

4.2. Resource Loading Optimization

Resource load optimization is a key part of Web front-end performance improvement. Resource load optimization is a critical aspect of improving the overall performance of web front-end applications. The first is to reduce the number of resource requests, merge CSS and JS files to reduce the number of HTTP requests, to improve the page launch speed, especially for a large number of small files. Compression techniques such as grip and Bortle are used to effectively reduce the size of the document, thereby reducing the pressure of data transfer to speed up the reading of the file. This approach is usually only used for optimizing static data such as JavaScript and CSS. A Content Delivery Network (CDN) hosts resources on geographically distributed servers, reducing latency by minimizing the physical distance between users and servers, reduces the physical distance between the customer and the server, speeds up the reading of resources, and can provide a better experience for users around the world. HTTP/2 protocol through the advantages of multi-path sharing and header information compression, so that the client delay is reduced, and increase the efficiency of parallel requests, speed up the response speed of the page. Based on these optimization techniques, you can reduce page load time and improve the performance of web applications (see Table 2).

Table 2. Resource Loading Optimization Methods.

Optimization method	Specific description
Reduce the number of resource requests	Reduce the number of HTTP requests and speed up resource loading by merging CSS and JavaScript files
Use compression techniques	The use of grip, Bortle and other compression technologies can reduce the size of the file and improve the speed.
Use a CDN	Using a CDN to distribute resources across multiple server nodes reduces physical distance and speeds up loading
Use the HTTP/2 protocol	The HTTP/2 protocol supports multiplexing, header compression, and more to reduce latency and speed up page response times

As can be seen from the above table, the methods to optimize Web front-end performance mainly focus on reducing the number of resource requests, improving loading efficiency and reducing resource consumption. The integration of these methods can effectively improve the loading speed and user experience of web applications.

4.3. Code Splitting and Lazy Loading

Code splitting and lazy loading are currently a common means of Web front-end performance optimization, its general idea is to split the application code into a fragment on demand, not at the beginning to load all the code of the application at once, but the first user to enter the website will only get the skeleton part of the application, that is, the code required on the user's visible page. The other functions will be loaded according to the user's request, which greatly reduces the amount of code required by the browser to load at a time, and realizes the fast startup of the application. With this solution, the user experience can be improved by reducing the amount of code used for initial application loading, improving the response speed and reducing the waiting time of users. On the other hand, with the deepening of application development and the increase of complexity, the code is divided into various functional units, and each module can be independently developed, and will not affect other functional codes due to the development of a part of the functional module, which is essential for the application with a large number of complex pages and more related functions. Lazy loading is often used in conjunction with code splitting. While code splitting divides an application into smaller chunks, lazy loading ensures that these chunks are only loaded when required by the user. Lazy loading is great for JavaScript modules, images, and multimedia applications such as video. Downloading such resources can consume significant bandwidth and processing power, potentially slowing down page initialization. In lazy loading mode, resources are downloaded only after the user moves the cursor in a browser window to a certain area or performs an action, rather than requesting the data in the first place. In this way, you can reduce the initial page loading network pressure and redundant data waste. Build tools such as webpack have powerful and customizable code splitting capabilities, and also support dynamic loading and lazy loading modules, which can greatly improve the efficiency of large and medium-sized projects.

4.4. Status Management Optimization

For Web applications, good state management is one of the most effective ways to improve performance and user experience. Good state management can avoid unnecessary rendering, improve response rates, and keep data flowing (Figure 1).

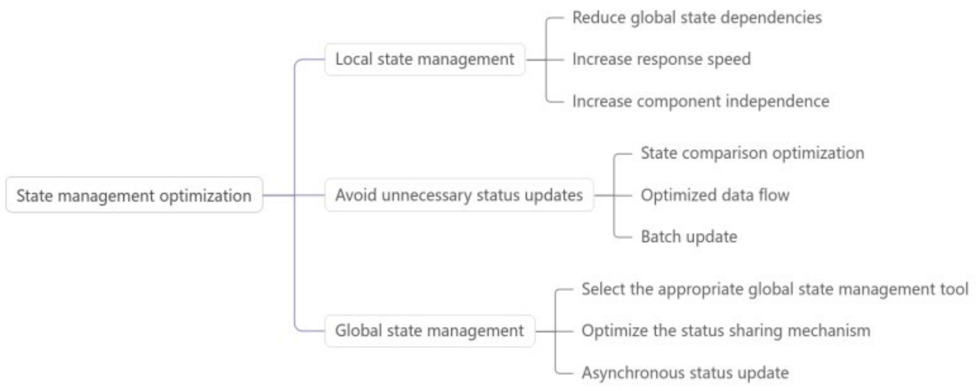
State management optimization
Local state management
Reduce global state dependencies
Increase response speed
Increase component independence
Avoid unnecessary status updates
State comparison optimization
Optimized data flow
Batch update
Global state management
Select the appropriate global state management tool
Optimize the status sharing mechanism
Asynchronous status update

Figure 1. State Management Optimization.

First, local state management is key to improving performance. Because each module handles only its own state, it reduces its dependence on global state, which increases the module's responsiveness and enhances its reusability and maintainability. Second, reducing unwanted status updates is also an effective way to improve performance. With state comparison optimization, update the old state only when it is inconsistent with the new state, avoiding meaningless re-rendering. Third, optimizing the data flow to ensure that the state changes when and only when it is needed can also greatly improve performance. The use of batch update technology can reduce the redrawing cost caused by multiple status updates during the status update process, and therefore improve the flow of the program. Finally, global state management is important for states that need to be shared among a large number of components. Global state can be easily controlled by selecting the appropriate state management module (Redux or Vuex), but it is important to store the required state in the entire system, because too much global state can degrade program performance. Finally, the processing of asynchronous status updates ensures that the status can be updated in time and no empty updates are generated during asynchronous operations. By combining the above improvement suggestions, web application state management becomes clearer, more powerful and more efficient, and has a good user experience.

5. Conclusion

Web front-end performance optimization is a lot of content, involving many aspects of improvement, such as rendering optimization, resource loading, code splitting, lazy loading and state management. Reasonable block structure design and performance optimization methods are beneficial to improve the software running speed and user satisfaction. Reducing redundant rendering, optimizing resource loading, precise code splitting and lazy loading, and efficient state management are all important ways to optimize website front-end performance. With the continuous development of Web technology, front-end applications become more and more complex, and users' requirements for front-end quality continue to improve, so we should continue to learn and adopt new optimization technologies, and choose reasonable optimization solutions for practical problems.

References

1. A. Siddiqui, R. Potoff, and Y. Huang, "Sustainability metrics and technical solution derivation for performance improvement of electroplating facilities," Clean Technol. Environ. Policy, vol. 26, pp. 1825–1842, 2024, doi: 10.1007/s10098-023-02696-9.

2. L. A. Gólcher-Barguil, S. P. Nadeem, J. A. Garza-Reyes, A. Samadhiya, and A. Kumar, "Measuring the financial impact of equipment performance improvement: ISB and IEB metrics," Benchmarking Int. J., vol. 30, no. 7, pp. 2408–2431, 2023, doi: 10.1108/BIJ-09-2021-0559.
3. Y. Zhao, U. De Silva, S. B. Venkatakrishnan, D. Psychogiou, G. Larkins, and A. Madanayake, "STAR Front-End Using Two Circulators in a Differential Connection," IEEE J. Microw., vol. 4, no. 2, pp. 253–263, Apr. 2024, doi: 10.1109/JMW.2024.3372855.
4. M. Vaquero, P. Mestres, and J. Cortés, "Resource-Aware Discretization of Accelerated Optimization Flows: The Heavy-Ball Dynamics Case," IEEE Trans. Autom. Control, vol. 68, no. 4, pp. 2109–2124, Apr. 2023, doi: 10.1109/TAC.2022.3171307.
5. H. Saffari, M. Abbasi, and J. Gheidar-Kheljani, "The design of a sustainable-resilient forward-reverse logistics network considering resource sharing and using an accelerated Benders decomposition algorithm," Int. J. Ship. Transp. Logist., vol. 19, no. 4, pp. 444–481, 2024, doi: 10.1504/IJSTL.2024.144020.