horizontal line

Queries & Compiled Search Results

Performance Data Extraction

**─**

# What are the different Page Level Metrics?

1. **Page Load Time:** This metric measures the time it takes for the entire web page to load, starting from the initial request sent by the browser until all the resources (HTML, CSS, JavaScript, images, etc.) have finished downloading and rendering.
2. **Time to First Byte (TTFB):** TTFB measures the time taken for the browser to receive the first byte of data from the server after sending the request. It indicates the responsiveness of the server and network latency.
3. **DNS Lookup Time:** This metric measures the time it takes to resolve the domain name (URL) of the web page to its corresponding IP address. It includes the time spent on DNS resolution and network latency.
4. **Connection Time:** Connection time measures the time taken to establish a connection between the browser and the web server. It includes the time spent on DNS lookup, establishing a TCP handshake, and negotiating a secure SSL/TLS connection.
5. **First Contentful Paint (FCP):** FCP measures the time it takes for the first piece of content (text, image, or DOM element) to be rendered on the web page. It gives an indication of how quickly users perceive that the page is loading.
6. **Time to Interactive (TTI):** TTI measures the time it takes for the web page to become fully interactive, meaning users can interact with the page and all the JavaScript event handlers are registered. It's an important metric for measuring user experience.
7. **Total Download Size:** This metric measures the total size of all the resources (HTML, CSS, JavaScript, images, etc.) downloaded by the browser to render the web page. A smaller download size generally leads to faster page load times.
8. **Number of Requests:** This metric counts the total number of HTTP requests made by the browser to fetch all the resources required for the web page. Reducing the number of requests can improve page load performance.
9. **Error Rate:** Error rate measures the percentage of requests that resulted in errors, such as HTTP status codes 4xx or 5xx. A high error rate indicates potential issues with the server or resource availability.
10. **Render Time:** Render time measures the time taken by the browser to process and render the HTML, CSS, and JavaScript code, transforming them into a visual representation of the web page on the user's screen.

# What are the different Element Level Metrics?

1. **Time to First Byte (TTFB):** TTFB measures the time it takes for the browser to receive the first byte of data for a specific element/component after sending the request. It indicates the responsiveness of the server and network latency for that particular element.
2. **Resource Load Time:** This metric measures the time it takes for a specific resource (such as an image, CSS file, or JavaScript file) to be loaded and available for use by the element. It helps determine the efficiency of resource delivery.
3. **Time to First Paint (TTP):** TTP measures the time it takes for the first pixels of a specific element to be rendered on the screen. It indicates how quickly users perceive visual progress for that particular element.
4. **Time to Interactive (TTI):** TTI measures the time it takes for a specific element or component to become fully interactive. It signifies when users can actively engage with or manipulate the element/component.
5. **DOMContentLoaded:** DOMContentLoaded measures the time it takes for the HTML document to be parsed and the DOM (Document Object Model) to be fully loaded for the specific element. It reflects when the element is ready for JavaScript manipulation.
6. **Render Start Time:** This metric indicates when the rendering process begins for a specific element. It helps assess the rendering performance and visual experience of that element.
7. **Layout Time:** Layout time measures the time taken to calculate the position and size of a specific element on the page. It indicates how efficiently the browser can perform layout operations.
8. **Script Execution Time:** This metric measures the time taken to execute JavaScript code related to a specific element or component. It helps identify any script-related performance bottlenecks.
9. **Memory Usage:** Memory usage refers to the amount of memory allocated and used by a specific element or component. High memory usage can indicate potential performance issues or memory leaks.
10. **Repaint Frequency:** Repaint frequency measures how often a specific element triggers repaints or reflows on the page. Frequent repaints can impact the overall rendering performance and user experience.

# Difference ways to compute time?

1. **Server-Side Timing:** Time can be computed on the server side, where the web server measures the time taken to process a request and generate a response. This includes measuring the time for server-side scripting, database queries, and other server-side operations.
2. **Client-Side Timing:** Time can be computed on the client side, within the user's browser. Browser-based timing methods provide insights into the time taken for various activities, such as resource loading, rendering, and JavaScript execution.
3. **Network Timing:** Network timing measures the time it takes for network-related operations, such as DNS lookup, establishing a connection, and transferring data over the network. These timings are typically available through browser APIs, like the Navigation Timing API.
4. **User Timing API:** The User Timing API allows developers to instrument their JavaScript code and create custom timing measurements. This API enables you to measure specific operations, events, or segments of code, providing more granular insights into performance.
5. **Navigation Timing API:** The Navigation Timing API is a browser API that provides timing-related information for a web page's navigation process. It offers metrics like page load time, TTFB, DOMContentLoaded, and more.
6. **Resource Timing API:** The Resource Timing API provides detailed timing information about individual resources (e.g., images, scripts, CSS files) that are loaded on a web page. It enables measuring resource-specific timings, such as DNS lookup time, connect time, and response time.
7. **Performance Mark and Measure API:** The Performance Mark and Measure APIs allow developers to mark specific points in code and measure the time between those marks. This enables fine-grained performance measurements for specific sections or events.
8. **Real User Monitoring (RUM):** RUM involves capturing and analyzing timing data from actual user interactions on a live website. It collects data from real users' browsers, providing insights into actual user experience and performance.
9. **Synthetic Monitoring:** Synthetic monitoring involves simulating user interactions and measuring performance in a controlled testing environment. It allows you to measure performance metrics consistently and repeatedly from different locations.
10. **Performance Profiling:** Performance profiling tools, such as browser developer tools, enable capturing and analyzing the execution time of JavaScript code. Profiling helps identify performance bottlenecks and optimize code for better performance.

# Commonly used Open Source Tools available for Web Page Performance

1. **Lighthouse:** Developed by Google, Lighthouse is a widely-used tool that audits web pages for performance, accessibility, SEO, and best practices. It provides comprehensive reports with actionable suggestions for improving page performance.
2. **WebPageTest:** WebPageTest allows you to test and analyze the performance of web pages from multiple locations and different browsers. It provides detailed metrics, waterfall charts, and video recordings of the page loading process.
3. **GTmetrix:** GTmetrix is a tool that analyzes the performance of web pages and provides actionable recommendations. It generates a performance report with various metrics, such as page load time, page size, and requests.
4. **PageSpeed Insights:** Another tool developed by Google, PageSpeed Insights provides insights into the performance of web pages on both desktop and mobile devices. It offers suggestions for improving performance and assigns a score based on performance metrics.
5. **Sitespeed.io:** Sitespeed.io is an open-source tool that helps you analyze and optimize website performance. It provides detailed reports with various metrics, such as page load time, requests, and asset sizes. It can be integrated into a continuous integration pipeline for automated performance testing.
6. **Apache JMeter:** Apache JMeter is a Java-based open-source tool used for load testing and performance measurement of web applications. It allows you to simulate multiple users, send HTTP requests, and analyze response times and throughput.
7. **WebPagetest:** WebPagetest is an open-source project that allows you to test web page performance from various locations and browsers. It provides detailed waterfall charts, filmstrip views, and performance metrics for analysis.
8. **SpeedTracker:** SpeedTracker is an open-source tool developed by Facebook for monitoring and analyzing web page performance. It provides metrics like FCP, TTI, and visualizes performance trends over time.
9. **Boomerang:** Boomerang is an open-source JavaScript library that enables you to measure web page performance from the end user's perspective. It can be easily integrated into websites to collect performance metrics and send them to various analytics platforms.
10. **Calibre:** Calibre is an open-source tool that helps you measure, track, and report web page performance. It provides insights into various performance metrics and allows for continuous monitoring of performance regressions.

# Commonly used Open Source Tools available for Element Performance

1. **Chrome DevTools:** Chrome DevTools is a powerful set of web development and debugging tools built into the Google Chrome browser. It provides a Performance panel that allows you to record and analyze the performance of individual elements on a web page.
   * Performance Recording: Use the "Record" button in the Performance panel to start recording the performance timeline. Interact with the web page to capture the specific element/component's performance.
   * Waterfall Analysis: After recording, you can examine the waterfall chart to see the timings for various events like network requests, script execution, and rendering. This helps identify performance bottlenecks for specific elements.
   * User Timings: You can instrument your JavaScript code to measure custom user timings for specific element-related operations. This can provide insights into the performance of those elements during different user interactions.
2. **Firefox Developer Tools:** Firefox Developer Tools also offer similar performance analysis capabilities.
   * Waterfall Analysis: The Network panel in Firefox Developer Tools provides a waterfall view of network requests, including individual element/component requests.
   * Performance Analysis: The Performance panel allows you to record and analyze the performance of specific elements or components on the web page.
3. **WebPagetest:** In addition to overall page-level performance analysis, WebPagetest also allows you to analyze specific elements or components on a web page.
   * Element Timing: WebPagetest provides an "Element Timing" feature that allows you to specify specific elements using CSS selectors. It measures the time taken to load and render those elements individually.
4. **Custom JavaScript Performance Monitoring:** You can use custom JavaScript code to monitor the performance of specific elements or components.
   * Performance APIs: The Navigation Timing API and the User Timing API provide methods for measuring and analyzing the performance of specific elements in your JavaScript code.
   * Custom Timers: You can use custom timers in your JavaScript code to measure the time taken for specific operations related to elements or components.

# Capturing the performance and impact of third-party frameworks and libraries

1. **Performance Profiling:**
   * Use browser developer tools or dedicated profiling tools to analyze the performance of the web page.
   * Profile the page with and without the third-party frameworks/libraries to identify any performance differences.
   * Analyze the profiling results to identify functions, scripts, or components from the third-party frameworks/libraries that contribute to performance bottlenecks.
2. **Network Monitoring:**
   * Utilize browser developer tools or network monitoring tools to inspect network requests made by the third-party frameworks/libraries.
   * Examine the size, number, and duration of network requests generated by the frameworks/libraries.
   * Analyze the impact of these requests on the overall page load time and network performance.
3. **Resource Monitoring:**
   * Monitor the resources (e.g., CSS, JavaScript files) loaded by the third-party frameworks/libraries.
   * Measure the size and impact of these resources on the page's load time and rendering performance.
   * Compare the performance with and without loading the third-party resources to assess their impact.
4. **Dependency Analysis:**
   * Identify the dependencies of the third-party frameworks/libraries (e.g., other libraries, scripts, CSS files) and their impact on the overall page performance.
   * Determine if there are any redundant or unnecessary dependencies that can be removed or optimized.
   * Assess the impact of dependencies on script execution, rendering, and overall page load time.
5. **User Experience Testing:**
   * Conduct user experience testing with and without the third-party frameworks/libraries to gather feedback and assess any noticeable differences.
   * Solicit user feedback on perceived performance, responsiveness, and overall experience when using the page with the frameworks/libraries.
6. **A/B Testing:**
   * Perform A/B testing by creating multiple versions of the web page, one with the third-party frameworks/libraries and one without.
   * Collect performance metrics such as page load time, time to interact, and conversion rates to compare the impact of the frameworks/libraries.
7. **Real User Monitoring (RUM):**
   * Implement Real User Monitoring tools to collect performance data from actual users interacting with the web page.
   * Analyze the performance metrics specifically associated with the use of the third-party frameworks/libraries.
   * Identify any performance issues reported by real users and their correlation with the presence of the frameworks/libraries.

**Capturing the performance and impact wrt main and worker thread**

1. **Performance Profiling:**
   * Utilize browser developer tools or dedicated profiling tools to capture and analyze the performance of the main and worker threads.
   * Use the profiling tools to measure the execution time and performance characteristics of code running on each thread.
   * Identify any long-running or blocking operations that may impact the responsiveness of the threads.
   * Look for performance bottlenecks, such as excessive JavaScript execution, rendering operations, or network requests.
2. **JavaScript Timers:**
   * Utilize JavaScript timers to measure and compare the execution times of specific operations on the main and worker threads.
   * Use timers, such as performance.now() or Date.now(), to calculate the time taken for specific tasks on each thread.
   * Compare the execution times to identify any significant differences between the main and worker threads.
3. **Worker Thread Monitoring:**
   * Implement custom monitoring within the worker thread to capture performance metrics.
   * Measure the time taken for specific tasks, message passing latency, and resource utilization within the worker thread.
   * Use custom timers or performance monitoring libraries specific to the worker thread environment (e.g., self.performance in Web Workers).
4. **Network Monitoring:**
   * Utilize browser developer tools or network monitoring tools to assess the impact of network requests on the main and worker threads.
   * Analyze the network activity on both threads, including request and response times, to identify any potential performance issues.
   * Consider offloading network-intensive tasks to worker threads to prevent blocking the main thread.
5. **Rendering Performance:**
   * Analyze rendering performance and its impact on the main thread.
   * Use browser developer tools to inspect rendering times, layout operations, and paint times.
   * Identify any rendering-related bottlenecks that could impact the main thread's performance.
6. **Load Testing:**
   * Conduct load testing scenarios to simulate high user traffic and measure the performance impact on both the main and worker threads.
   * Monitor key performance metrics, such as response times, CPU and memory utilization, and thread concurrency during the load testing process.
   * Assess how the main and worker threads handle increased workload and identify any performance degradation.

Reference

<https://developers.google.com/web/fundamentals/performance>

<https://developer.mozilla.org/en-US/docs/Learn/Performance>

<https://webpagetest.org/>

<https://gtmetrix.com/>

<https://www.w3.org/>

<https://www.smashingmagazine.com/>

<https://css-tricks.com/>