Ministry of SMEs and Startups

KOSME Korea SMEs and Startups Agency

LIKELION

JOBKOREA

LESSON 46
Performance Audit with Lighthouse

WEEK 10

Introduction to Performance Auditing

❖ What is performance auditing?

- ➢ Performance auditing involves analyzing how well a web application performs in real-world conditions and identifying areas for improvement.

❖ Why is it important for ReactJS and NextJS applications?

- ➢ ReactJS and NextJS apps often face performance challenges like slow load times or inefficient rendering, which can impact user experience.

❖ Overview of key performance metrics:

- ➢ **Load time:** How quickly the application becomes usable.
- ➢ **First Contentful Paint (FCP):** Time taken to render the first visible element.
- ➢ **Time to Interactive (TTI):** Time taken for the app to become fully interactive.

What is Lighthouse?

- **A tool for measuring web app performance:**
  - Lighthouse is an open-source tool that helps developers assess the quality of web pages.
- **Open-source, automated tool:**
  - It provides an automated way to measure several aspects of web performance and accessibility.
- **Provides insights on various performance metrics:**
  - Lighthouse evaluates the app in areas like performance, SEO, accessibility, and best practices.
- **Generates reports to help improve app performance:**
  - The tool gives actionable insights that developers can use to optimize their web applications.

How Lighthouse Works

- **Performs audits based on real-world conditions:**
  - Lighthouse runs audits that simulate network speeds, device capabilities, and CPU performance to reflect realistic user experience.
- **Simulates load on various devices (mobile, desktop):**
  - The tool provides separate scores for mobile and desktop performance to ensure cross-platform optimization.
- **Provides suggestions for performance improvements:**
  - Based on the audit, Lighthouse suggests steps for optimizing performance, such as reducing JavaScript execution time or optimizing images.

Lighthouse Report Overview

- **Performance score:**
  - The score ranges from 0-100, indicating the performance level of the application.
- **Key metrics reported by Lighthouse:**
  - **FCP (First Contentful Paint), LCP (Largest Contentful Paint), Speed Index**
  - Metrics that directly impact the perceived load time and interactivity of the app.
- **Opportunities and Diagnostics:**
  - Suggestions on how to improve the performance, like eliminating render-blocking resources or reducing JavaScript file sizes.
- **How to read the Lighthouse report:**
  - Each section of the report is clearly laid out to show areas of improvement along with potential fixes.

Setting up Lighthouse

- **Install Lighthouse as Chrome Extension or use CLI:**
  - Lighthouse can be used directly in the Chrome DevTools or via the CLI for automated performance checks.
- **Audit Web app with Lighthouse in DevTools:**
  - Open the Chrome DevTools, navigate to the Lighthouse tab, and click "Generate Report" to audit the Web application.
- **Example: Running Lighthouse on a web page:**
  - Demonstrate the audit process with a simple web page to show the tool in action.

Analyzing the Lighthouse Report

- **FCP (First Contentful Paint):**
  - **What it measures:** Time taken for the first piece of content to appear on the screen.
  - **Impact:** Affects how quickly users see something on the screen, which is crucial for user engagement.
- **TTI (Time to Interactive):**
  - **What it measures:** Time taken for the app to become fully interactive, meaning it responds to user input.
  - **Impact:** Directly impacts the usability of the app, especially for complex web applications.
- **LCP (Largest Contentful Paint):**
  - **What it measures:** The time it takes to render the largest piece of content on the page.
  - **Impact:** A slow LCP can cause frustration as users wait for the main content to appear.
- **CLS (Cumulative Layout Shift):**
  - **What it measures:** The visual stability of the page, indicating how much the content shifts unexpectedly.
  - **Impact:** A high CLS causes poor user experience, especially when buttons or links move unexpectedly.

Optimizing FCP (First Contentful Paint)

- **What is FCP?**
  - FCP measures the time from when a user navigates to a page to when the browser renders the first piece of content.
- **How to reduce FCP:**
  - **Code splitting:** Break the app into smaller chunks that load on demand.
  - **Lazy loading:** Load non-essential resources only when required.
  - **Server-side rendering (SSR):** Render content on the server before sending it to the client.
- **Example: ReactJS component lazy loading:**
  - Implement lazy loading in ReactJS using React.lazy() to only load components when needed.

Optimizing LCP (Largest Contentful Paint)

- **What is LCP?**
  - LCP measures how long it takes for the largest piece of content on the screen to load.
- **How to reduce LCP:**
  - **Image optimization:** Compress images and use modern formats like WebP.
  - **Preloading resources:** Preload important resources such as fonts or images that will render above the fold.
  - **Reducing server response times:** Optimize your backend to serve content faster.
- **Example: Optimizing images with NextJS Image component:**
  - Use NextJS's built-in <Image> component to automatically optimize images for different screen sizes and formats.

Optimizing CLS (Cumulative Layout Shift)

- **What is CLS?**
  - CLS measures unexpected layout shifts during the page load process.
- **How to prevent CLS:**
  - **Set size for images:** Always specify width and height for images to prevent shifts.
  - **Avoid dynamic content shifts:** Ensure that dynamically injected content doesn't affect the layout unexpectedly.
- **Example: Fixing layout shifts in ReactJS components:**
  - Use fixed or responsive layout styles to ensure elements don't shift unexpectedly during rendering.

Server-Side Rendering (SSR) with NextJS

- **Benefits of SSR for performance:**
  - SSR allows content to be rendered on the server, which can speed up the time it takes for users to see content on the screen.
- **How to implement SSR in NextJS:**
  - Use NextJS's getServerSideProps function to fetch data on the server and send it to the client as part of the initial render.
- **Lighthouse report improvements with SSR:**
  - Server-side rendering improves FCP and LCP by reducing the amount of client-side rendering required.

Optimizing TTI (Time to Interactive)

- **What is TTI?**
  - TTI measures the time it takes for a page to become fully interactive, meaning it can respond to user input without delay.
- **Why is TTI important?**
  - High TTI can cause users to abandon the app because it feels unresponsive.
- **How to reduce TTI:**
  - **Minimize JavaScript execution time:** Reduce the amount of JavaScript running on page load. Use code splitting and dynamic imports to load only the required code.
  - **Defer non-critical scripts:** Load non-essential scripts after the page becomes interactive.
  - **Use Web Workers:** Offload heavy tasks to background threads using Web Workers to keep the UI responsive.
- **Example: Deferring non-critical scripts in NextJS:**
  - Implement next/script with the strategy="afterInteractive" attribute to load non-essential scripts after the page has loaded and become interactive.

Optimizing Speed Index

- **What is Speed Index?**
  - Speed Index measures how quickly the visible parts of the page are populated during the page load process.
- **Why is Speed Index important?**
  - A low “Speed Index” means that the user sees content quickly, even before the entire page has fully loaded.
- **How to reduce Speed Index:**
  - **Prioritize above-the-fold content:** Load and render the critical content that appears first on the screen before non-critical resources.
  - **Lazy loading images and assets:** Load images and other assets only when they come into the viewport to avoid blocking the rendering process.
  - **Critical CSS:** Inline or preload the critical CSS needed for above-the-fold content to render faster.
- **Example: Prioritizing above-the-fold content in NextJS:**
  - Use next/head to include critical CSS and preload important resources like fonts to ensure above-the-fold content loads faster.

Optimizing FID (First Input Delay)

- **What is FID?**
  - FID measures the time it takes for the browser to respond to the first interaction (click, tap, etc.) after the page is loaded.
- **Why is FID important?**
  - High FID indicates that the page is slow to respond to user input, which can negatively impact the user experience, especially in interactive apps.
- **How to reduce FID:**
  - **Minimize JavaScript execution on initial load:** Reduce the complexity of the JavaScript required for the initial interaction by using code splitting and deferring non-essential JavaScript.
  - **Optimize event handlers:** Ensure that event handlers are quick to execute by keeping them lightweight and avoiding blocking operations during page load.
  - **Prioritize critical scripts:** Load only essential scripts before allowing the page to respond to user input.
- **Example: Optimizing event handlers in ReactJS:**
  - Use React.memo and useCallback to avoid unnecessary re-renders and optimize event handling for interactive elements.

Optimizing TBT (Total Blocking Time)

- **What is TBT?**
  - TBT measures the total amount of time that the page is blocked from responding to user input, caused by long-running JavaScript tasks.
- **Why is TBT important?**
  - High TBT indicates that the page is not interactive during long periods, causing a poor user experience.
- **How to reduce TBT:**
  - **Break up long tasks:** Split long-running JavaScript tasks into smaller tasks that run asynchronously to prevent blocking the main thread.
  - **Use requestIdleCallback:** Offload non-critical tasks to idle time when the main thread is free.
  - **Optimize third-party scripts:** Minimize the use of third-party scripts or make sure they load asynchronously without blocking the page.
- **Example: Splitting long tasks in ReactJS:**
  - Use setTimeout or requestIdleCallback to break long-running calculations into smaller chunks, ensuring the main thread is not blocked.

Optimizing Time to First Byte (TTFB)

- **What is TTFB?**
  - TTFB is the time it takes from sending the request to receiving the first byte of data from the server.
- **Why is TTFB important?**
  - High TTFB means that the server is slow to respond, leading to longer wait times for users.
- **How to reduce TTFB:**
  - **Optimize server performance:** Use faster server-side technology, caching mechanisms, and databases to reduce response times.
  - **Use Content Delivery Networks (CDN):** Distribute content closer to users to reduce the distance and time required to fetch resources.
  - **Implement server-side caching:** Cache commonly requested resources to avoid repetitive computations on the server.
- **Example: Implementing CDN in NextJS:**
  - Use a CDN like Vercel or Cloudflare to deliver static assets faster by caching them at edge locations closer to the user.

Optimizing Image Performance

- **Why optimize images?**
  - Large or unoptimized images can slow down page load times, particularly for mobile users.
- **How to optimize images:**
  - **Use modern image formats (WebP):** WebP offers better compression and quality than traditional formats like JPEG and PNG.
  - **Lazy load images:** Only load images when they are about to appear in the viewport.
  - **Resize images for different screen sizes:** Ensure that images are served in appropriate sizes for different devices.
- **Example: Using NextJS Image component for optimization:**
  - NextJS's <Image> component automatically optimizes images by serving them in the correct size, format, and quality for each device.

Optimizing Fonts

- **Why optimize fonts?**
  - Fonts can block rendering and delay page load times if not properly optimized.
- **How to optimize fonts:**
  - **Preload critical fonts:** Use font-display: swap and preload important fonts to avoid rendering delays.
  - **Limit the number of fonts and font weights:** Reduce the number of font files to improve load speed.
  - **Use system fonts:** Where possible, use system fonts to avoid the overhead of downloading custom fonts.
- **Example: Preloading fonts in NextJS:**
  - Use next/head to preload critical fonts and avoid blocking the rendering process while waiting for font files.

Optimizing Critical Rendering Path

❖ What is Critical Rendering Path?

- ➢ The sequence of steps that the browser follows to render the content of a web page.

❖ Why is it important?

- ➢ By optimizing the critical rendering path, you can ensure that the user sees content as quickly as possible.

❖ How to optimize:

- ➢ **Eliminate render-blocking resources:** Ensure CSS and JavaScript don't block the rendering of critical content by using async/defer attributes.
- ➢ **Minimize critical CSS:** Extract only the critical CSS needed for the first render and inline it in the HTML.
- ➢ **Reduce the number of requests:** Consolidate CSS and JavaScript files where possible.