.html

Front-End Performance Optimization Checklist

This checklist is designed to help you systematically apply critical front-end performance optimizations, including critical CSS, lazy loading, and script deferring. Use it as a guide to audit your existing projects or plan new implementations.

Part 1: Critical CSS Optimization

Goal: Ensure the "above the fold" content renders as fast as possible by inlining essential styles.

Tasks:

- **Identify Critical CSS:**
  - Determine the content visible within the initial viewport for common screen sizes (desktop, tablet, mobile).
  - Use an automated tool (e.g., Penthouse, Critical, `critical` npm package) to extract CSS rules specific to this "above the fold" content.
  - Verify the extracted critical CSS size is minimal (ideally < 14 KB uncompressed).
  - ***Check***: Is critical CSS extracted correctly for all key page templates?
- **Inline Critical CSS:**
  - Embed the extracted critical CSS directly into a `<style>` block within the `<head>` of your HTML document.
  - Ensure this `<style>` block appears as early as possible within the `<head>`, before any external stylesheet links if possible.
  - ***Check***: Is critical CSS inlined in the HTML head?
- **Asynchronously Load Remaining CSS:**
  - Modify the link to your full, external CSS stylesheet to load asynchronously.
  - Common methods include using `rel="preload"` with a `onload="this.rel='stylesheet'"` fallback or using JavaScript to dynamically create the `<link>` tag.
  - ***Check***: Is the main CSS stylesheet loaded non-render-blocking?
- **Test and Validate:**
  - Use Lighthouse or WebPagetest to measure metrics like First Contentful Paint (FCP) and Largest Contentful Paint (LCP).
  - Visually inspect the page load to ensure there's no "flash of unstyled content" (FOUC) before the main stylesheet loads.
  - ***Check***: Have FCP and LCP improved significantly after critical CSS implementation?

Part 2: Lazy Loading Techniques

Goal: Defer the loading of off-screen images, iframes, and videos until they are needed, reducing initial page weight.

Tasks:

- **Images (`<img>` tags):**
  - For all images that are not immediately visible in the initial viewport, add the `loading="lazy"` attribute to their `<img>` tags.
  - Provide `width` and `height` attributes or use CSS `aspect-ratio` to prevent layout shifts (CLS).
  - Consider using responsive images (`<picture>` and `srcset`) for optimal image delivery based on device.
  - ***Check***: Are all off-screen images using `loading="lazy"`?
- **Iframes (`<iframe>` tags):**
  - Apply `loading="lazy"` to all `<iframe>` tags that are not immediately visible (e.g., embedded maps, YouTube videos).
  - ***Check***: Are off-screen iframes using `loading="lazy"`?
- **Videos (`<video>` tags):**
  - For `<video>` elements, use the `preload="none"` attribute to prevent the browser from preloading the entire video file until the user initiates playback.
  - Consider `poster` attributes for a static image fallback.
  - ***Check***: Are off-screen videos optimized for lazy loading?
- **JavaScript-based Lazy Loading (for older browsers or custom control):**
  - If native lazy loading is not sufficient or for elements like CSS background images, implement Intersection Observer API.
  - For images, store the actual `src` in a `data-src` attribute and swap it when the element enters the viewport.
  - ***Check***: Is a robust lazy loading solution in place for all deferred media?

Part 3: Deferring & Async Scripts

Goal: Prevent JavaScript from blocking HTML parsing and rendering, improving initial page load speed.

Tasks:

- **Audit All Scripts:**
  - Identify all JavaScript files linked in your HTML, both internal and third-party.
  - Categorize them by their necessity for initial page render (e.g., essential UI, analytics, ads, interactivity).

    - ***Check***: Is there a clear understanding of what each script does and when it's needed?
- **Apply `defer` Attribute:**
    - For scripts that depend on the DOM being fully parsed or need to execute in a specific order (but are not render-blocking), add the `defer` attribute.
    - Place `<script defer src="..."></script>` towards the end of the `<head>` or just before the closing `</body>` tag.
    - ***Check***: Are most non-critical scripts using `defer`?
- **Apply `async` Attribute:**
    - For independent scripts where execution order doesn't matter and they don't modify the DOM structure significantly (e.g., analytics, some tracking scripts), use the `async` attribute.
    - Place `<script async src="..."></script>` as early as possible in the `<head>` if no DOM dependencies.
    - ***Check***: Are independent, non-DOM-dependent scripts using `async`?
- **Inline Small, Essential Scripts:**
    - For very small scripts (< 1 KB) that are absolutely critical for initial UI functionality and cannot be deferred, consider inlining them directly into the HTML.
    - ***Check***: Are only truly essential and tiny scripts inlined?
- **Minimize and Bundle Scripts:**
    - Combine multiple small JavaScript files into a single, larger file (bundling) to reduce HTTP requests.
    - Minify all JavaScript files to remove unnecessary characters and reduce file size.
    - ***Check***: Are JavaScript files bundled and minified?

Part 4: Measuring & Testing Performance

Goal: Continuously monitor and evaluate performance improvements, ensuring optimizations are effective.

Tasks:

- **Establish Baselines:**
    - Before implementing any changes, run performance tests to get baseline scores using tools like Lighthouse, WebPagetest, or GTmetrix.
    - Record key metrics: FCP, LCP, Total Blocking Time (TBT), Cumulative Layout Shift (CLS), Speed Index.
    - ***Check***: Are baseline performance metrics documented?
- **Regular Audits with Lighthouse:**
    - Run Lighthouse audits frequently (e.g., weekly or with each major deployment) in Chrome DevTools or via CI/CD.

.html

    - Pay close attention to "Opportunities" and "Diagnostics" sections for actionable advice.
    - ***Check***: Is Lighthouse integrated into the development workflow?
- **Advanced Testing with WebPagetest:**
    - Use WebPagetest to simulate real-world conditions (different locations, network speeds, devices).
    - Analyze waterfall charts to identify render-blocking resources, bottlenecks, and optimization opportunities.
    - ***Check***: Are performance tests conducted under various network conditions?
- **Monitor Core Web Vitals:**
    - Keep track of your site's Core Web Vitals (LCP, FID, CLS) in Google Search Console.
    - Address any reported issues to ensure a good user experience and SEO ranking.
    - ***Check***: Are Core Web Vitals consistently monitored and within healthy thresholds?
- **Continuous Improvement Loop:**
    - Treat performance optimization as an ongoing process, not a one-time fix.
    - Re-evaluate optimizations as content, features, or dependencies change.
    - ***Check***: Is there a plan for ongoing performance monitoring and refinement?