PageSpeed Rules and Recommendations

- Eliminate render-blocking JavaScript and CSS
- Optimize JavaScript use
  - Prefer asynchronous JavaScript resources
  - Avoid synchronous server calls
  - Defer parsing JavaScript
  - Avoid long running JavaScript
- Optimize CSS Use
  - Put CSS in the document head
  - Avoid CSS imports
  - Inline render-blocking CSS
- Feedback {: #feedback }

This guide examines PageSpeed Insights rules in context: what to pay attention to when optimizing the critical rendering path, and why.

Eliminate render-blocking JavaScript and CSS

To deliver the fastest time to first render, minimize and (where possible) eliminate the number of critical resources on the page, minimize the number of downloaded critical bytes, and optimize the critical path length.

Optimize JavaScript use

JavaScript resources are parser blocking by default unless marked as `async` or added via a special JavaScript snippet. Parser blocking JavaScript forces the browser to wait for the CSSOM and pauses construction of the DOM, which in turn can significantly delay the time to first render.

Prefer asynchronous JavaScript resources

Asynchronous resources unblock the document parser and allow the browser to avoid blocking on CSSOM prior to executing the script. Often, if the script can use the `async` attribute, it also means it is not essential for the first render. Consider loading scripts asynchronously after the initial render.

Avoid synchronous server calls

Use the `navigator.sendBeacon()` method to limit data sent by XMLHttpRequests in `unload` handlers. Because many browsers require such requests to be synchronous, they can slow page transitions, sometimes noticeably. The following code shows how to use `navigator.sendBeacon()` to send data to the server in the `pagehide` handler instead of in the `unload` handler.

```
<script>
  function() {
    window.addEventListener('pagehide', logData, false);
    function logData() {
      navigator.sendBeacon(
        'https://putsreq.herokuapp.com/Dt7t2QzUkG18aDTMMcop',
        'Sent by a beacon!');
    }
  }();
</script>
```

The new `fetch()` method provides an easy way to asynchronously request data. Because it is not available everywhere yet, you should use feature detection to test for its presence before use. This method processes responses with Promises rather than multiple event handlers. Unlike the response to an XMLHttpRequest, a fetch response is a stream object starting in Chrome 43. This means that a call to `json()` also returns a Promise.

```
<script>
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Looks like there was a problem. Status Code: ' +  response.status);
```

```
          return;
        }
        // Examine the text in the response
        response.json().then(function(data) {
          console.log(data);
        });
      }
    )
    .catch(function(err) {
      console.log('Fetch Error :-S', err);
    });
  </script>
```

The `fetch()` method can also handle POST requests.

```
  <script>
  fetch(url, {
    method: 'post',
    headers: {
      "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
    },
    body: 'foo=bar&lorem=ipsum'
  }).then(function() { // Additional code });
  </script>
```

Defer parsing JavaScript

To minimize the amount of work the browser has to perform to render the page, defer any non-essential scripts that are not critical to constructing the visible content for the initial render.

Avoid long running JavaScript

Long running JavaScript blocks the browser from constructing the DOM, CSSOM, and rendering the page, so defer until later any initialization logic and functionality that is non-essential for the first render. If a long initialization sequence needs to run, consider splitting it into several stages to allow the browser to process other events in between.

Optimize CSS Use

CSS is required to construct the render tree and JavaScript often blocks on CSS during initial construction of the page. Ensure that any non-essential CSS is marked as non-critical (for example, print and other media queries), and that the amount of critical CSS and the time to deliver it is as small as possible.

Put CSS in the document head

Specify all CSS resources as early as possible within the HTML document so that the browser can discover the `<link>` tags and dispatch the request for the CSS as soon as possible.

Avoid CSS imports

The CSS import (`@import`) directive enables one stylesheet to import rules from another stylesheet file. However, avoid these directives because they introduce additional roundtrips into the critical path: the imported CSS resources are discovered only after the CSS stylesheet with the `@import` rule itself is received and parsed.

Inline render-blocking CSS

For best performance, you may want to consider inlining the critical CSS directly into the HTML document. This eliminates additional roundtrips in the critical path and if done correctly can deliver a "one roundtrip" critical path length where only the HTML is a blocking resource.

Feedback {: #feedback }

□□□ #2
□ □□□ □□□ 2026-10-31 21:46:44 UTC
□ □□□ □□□ 2026-10-31 21:46:46 UTC