How To Optimize Inp

- index
- Client-side rendering of HTML and interactivity
- Diagnose slow interactions in the lab
- How large DOM sizes affect interactivity, and what you can do about it
- Find slow interactions in the field
- Optimize input delay
- Script evaluation and long tasks

index

layout: collection override:tags: [] date: 2023-05-09 pathName: how-to-optimize-inp

Client-side rendering of HTML and interactivity

Parsing and rendering of HTML is something that browsers do very well by default for websites that use the browser's built-in navigation logic—sometimes called "traditional page loads" or "hard navigations". Such websites are sometimes called multi-page applications (MPAs).

However, developers may work around browser defaults to suit their application needs. This is certainly the case for websites using the single page application (SPA) pattern, which dynamically creates large parts of the HTML/DOM on the client with JavaScript. Client-side rendering is the name for this design pattern, and it can have effects on your website's Interaction to Next Paint (INP) if the work involved is excessive.

This guide will help you weigh the difference between using HTML sent by the server to the browser versus creating it on the client with JavaScript, and how the latter can result in high interaction latency at crucial moments.

How the browser renders HTML provided by the server

The navigation pattern used in traditional page loads involves receiving HTML from the server on every navigation. If you enter a URL in the address bar of your browser or click on a link in an MPA, the following series of events occurs:

1. The browser sends a navigation request for the URL provided.
2. The server responds with HTML in chunks.

The last step of these is key. It's also one of the most fundamental performance optimizations in the server/browser exchange, and is known as *streaming*. If the server can begin sending HTML as soon as possible, and the browser doesn't wait for the entire response to arrive, the browser can process HTML in chunks as it arrives.

> *A screenshot of parsing of HTML sent by the server visualized in the performance panel of Chrome DevTools. As the HTML streams in, chunks of it are processed across multiple*

shorter tasks, and rendering is incremental.

Parsing and rendering of HTML provided by the server as visualized in the performance panel of Chrome DevTools. The tasks involved in parsing HTML and rendering it are split into chunks.

Like most things that happen in the browser, parsing HTML occurs within tasks. When HTML is streamed from the server to the browser, the browser optimizes parsing of that HTML by doing so a bit at a time as bits of that stream arrive in chunks. The consequence is that the browser yields to the main thread periodically after processing each chunk, which avoids long tasks. This means that other work can occur while HTML is being parsed, including the incremental rendering work necessary to present a page to the user, as well as processing user interactions that may occur during the page's crucial startup period. This approach translates to a better Interaction to Next Paint (INP) score for the page.

The takeaway? When you stream HTML from the server, **you get incremental parsing and rendering of HTML and automatic yielding to the main thread for free.** You don't get that with client-side rendering.

How the browser renders HTML provided by JavaScript

While every navigation request to a page requires some amount of HTML to be provided by the server, some websites will use the SPA pattern. This approach often involves a minimal initial payload of HTML is provided by the server, but then the client will populate the main content area of a page with HTML assembled from data fetched from the server. Subsequent navigations—sometimes referred to as "soft navigations" in this case—are handled entirely by JavaScript to populate the page with new HTML.

Client-side rendering may also occur in non-SPAs in more limited cases where HTML is dynamically added to the DOM through JavaScript.

There are a few common ways of creating HTML or adding to the DOM through JavaScript:

1. The `innerHTML` property allows you to set the content on an existing element via a string, which the browser parses into DOM.
2. The `document.createElement` method allows you to create new elements to be added to the DOM without using any browser HTML parsing.
3. The `document.write` method allows you to write HTML to the document (and the browser parses it, just like in approach #1). Due to a number of reasons, however, **usage of**

`document.write` is strongly discouraged.

A screenshot of parsing of HTML rendered via JavaScript visualized in the performance panel of Chrome DevTools. The work occurs in a single, long task that blocks the main thread.

Parsing and rendering of HTML through JavaScript on the client as visualized in the performance panel of Chrome DevTools. The tasks involved in parsing and rendering it are not chunked up, resulting in a long task that blocks the main thread.

The consequences of creating HTML/DOM through client-side JavaScript can be significant:

- Unlike HTML streamed by the server in response to a navigation request, JavaScript tasks on the client are not automatically chunked up, which may result in long tasks that block the main thread. This means that your page's INP can be negatively affected if you're creating too much HTML/DOM at a time on the client.
- If HTML is created on the client during startup, resources referenced within it will *not* be discovered by the browser preload scanner. This will certainly have a negative effect on a page's Largest Contentful Paint (LCP). While this is not a runtime performance issue (instead it's an issue of network delay in fetching important resources), you don't want your website's LCP to be affected by sidestepping this fundamental browser performance optimization.

What you can do about the performance impact of client-side rendering

If your website depends heavily on client-side rendering and you've observed poor INP values in your field data, you might be wondering if client-side rendering has anything to do with the problem. For example, if your website is an SPA, your field data may reveal interactions responsible for considerable rendering work.

Whatever the cause, here are some potential causes you can explore to help get things back on track.

Provide as much HTML from the server as possible

As mentioned earlier, the browser handles HTML from the server in a very performant way by default. It will break up parsing and rendering of HTML in a way that avoids long tasks, and optimizes the amount of total main thread time. This leads to a lower Total Blocking Time (TBT), and TBT is strongly correlated with INP.

You may be relying on a frontend framework to build your website. If so, you'll want to make sure you're rendering component HTML on the server. This will limit the amount of initial client-side rendering your website will require, and should result in a better experience.

- For React, you'll want to make use of the Server DOM API to render HTML on the server. But be aware: the traditional method of server-side rendering uses a synchronous approach, which can lead to a longer Time to First Byte (TTFB), as well as subsequent metrics such as First Contentful Paint (FCP) and LCP. Where possible, make sure you're using the streaming APIs for Node.js or other JavaScript runtimes so that the server can begin streaming HTML to the browser as soon as possible. Next.js—a React-based framework—provides many best practices by default. In addition to automatically rendering HTML on the server, it can also statically generate HTML for pages that don't change based on user context (such as authentication).
- Vue also performs client-side rendering by default. However, like React, Vue can also render your component HTML on the server. Either take advantage of these server-side APIs where possible, or consider a higher-level abstraction for your Vue project to make the best practices easier to implement.
- Svelte renders HTML on the server by default—although if your component code needs access to browser-exclusive namespaces (`window`, for example), you may not be able to render that component's HTML on the server. Explore alternative approaches wherever possible so that you're not causing unnecessary client-side rendering. SvelteKit—which is to Svelte as Next.js is to React—embeds many best practices into your Svelte projects as possible, so you can avoid potential pitfalls in projects that use Svelte alone.

If your application uses either of these frameworks and requires interactivity on the client, you'll still need to hydrate your components on the client so that they'll be interactive. For a comprehensive evaluation of the different approaches to rendering HTML, and for some advice on how to handle component hydration on the client with performance in mind, read Rendering on the web.

Limit the amount of DOM nodes created on the client

When DOMs are large, the amount of processing time required to render them tends to increase. Whether your website is a full-fledged SPA, or is injecting new nodes into an existing DOM as the result of an interaction for an MPA, consider keeping those DOMs as small as possible. This will help reduce the work required during client-side rendering to display that HTML, hopefully helping to keep your website's INP lower.

Read to learn more: DOM size and interactivity.

Consider a streaming service worker architecture

This is an advanced technique—one that may not work easily with every use case—but it's one that can turn your MPA into a website that feels like it's loading instantly when users navigate from one page to the next. You can use a service worker to precache the static parts of your website in `CacheStorage` while using the `ReadableStream` API to fetch the rest of a page's HTML from the server.

When you use this technique successfully, you aren't creating HTML on the client, but the instant loading of content partials from the cache will give the impression that your site is loading quickly. Websites using this approach can feel almost like an SPA, but without the downfalls of client-side rendering. It also reduces the amount of HTML you're requesting from the server.

In short, a streaming service worker architecture doesn't *replace* the browser's built-in navigation logic—it *adds* to it. For more information on how to achieve this with Workbox, read Faster multipage applications with streams.

Conclusion

How your website receives and renders HTML has an impact on performance. When you rely on the server to send all (or the bulk of) the HTML required for your website to function, you're getting a lot for free: incremental parsing and rendering, and automatic yielding to the main thread to avoid long tasks.

Client-side HTML rendering introduces a number of potential performance issues that can be avoidable in many cases. Due to each individual website's requirements, however, it's not entirely

avoidable 100% of the time. To mitigate the potential long tasks that can result from excessive client-site rendering, make sure you're sending as much of your website's HTML from the server whenever possible, keep your DOM sizes as small as possible for HTML that must be rendered on the client, and consider alternative architectures to speed the delivery of HTML to the client while also taking advantage of the incremental parsing and rendering the browser provides for HTML loaded from the server.

If you can get your website's client-side rendering to be as minimal as possible, you'll improve not just your website's INP, but other metrics such as LCP, TBT, and possibly even your TTFB in some cases.

Hero image from Unsplash, by Maik Jonietz.

Diagnose slow interactions in the lab

One challenging aspect of optimizing Interaction to Next Paint (INP) is figuring out what's causing poor INP. There's a large variety of potential causes: third-party scripts that schedule many tasks on the main thread, large DOM sizes, expensive event callbacks, and so on.

Finding ways to fix poor INP can be difficult. To start, you have to know which interactions tend to be responsible for a page's INP. If you don't know this already, start by reading Find slow interactions in the field. Once you have that field data and know what interactions to test, you can do so in lab tools to work out why those interactions are slow.

What if you don't have field data?

Ideally, you'll want field data, as it saves you a lot of time trying to figure out which interactions need to be optimized. You might be in a position where you don't have field data, though. If that's your situation, you can still find interactions to improve: you'll just have to take a different approach.

Total Blocking Time (TBT) is a metric that assesses page responsiveness during load. It correlates very well with INP, and can give you an idea if there might be interactions you can profile while the page is loading.

You can use either Lighthouse or PageSpeed Insights to measure your page's TBT. If your TBT is either poor or needs improvement, there's a good chance there are interactions that might be slow during page load.

To find slow interactions after the page has loaded, you might need to rely on other types of data, such as common user flows that you may already have in your website's analytics. If you work on an ecommerce website, for example, a common user flow would be the actions users take when they're adding items to an online shopping cart or going through an online checkout.

Whether or not you have field data, the next step involves reproducing that interaction—because it's only when you're able to conclusively identify a slow interaction that you'll be able to fix it.

Reproducing slow interactions in the lab

Once you've identified a slow interaction, the next step is to test it in the lab to see if it's reliably slow.

Don't reach for the performance profiler right away

Chrome performance profiler—while invaluable—doesn't provide a live view while interacting with the page. It's more efficient to test interaction latency in a much faster way first, so you can quickly assess whether a given interaction is reliably slow, and then reach for the performance profiler when you need more information.

Use the Web Vitals Chrome Extension

The Web Vitals Chrome Extension involves the lowest amount of effort in testing interaction latency. Once installed, the extension will display interaction data in the console if you do the following:

1. In Chrome, click the extensions icon to the right of the address bar.
2. Locate the Web Vitals extension in the drop-down menu.
3. Click the icon at the right to open the extension's settings.
4. Click **Options**.
5. Enable the **Console logging** checkbox in the resulting screen, and then click **Save**.

Once this has been done, open the console in Chrome DevTools, and begin testing the desired interactions on your website. As you interact with the page, you'll receive useful console logs giving you detailed diagnostic information for the interaction.

> *A screenshot of the console logging that the Web Vitals extension provides for interactions. The logging contains details such timings and other contextual information.*
>
> A console entry from the Web Vitals extension when console logging is turned on. Each qualifying interaction will log interaction data to the console.

Use a JavaScript snippet

Using the Web Vitals extension may not be a viable option for a number of reasons. Extensions can be blocked in some cases, and they also can't be installed on mobile devices. The latter is problematic if you want to test on a physical Android device with remote debugging.

An alternate method involves copying and pasting some JavaScript into the console of Chrome DevTools. The following code yields the same console output as the Web Vitals extension for every interaction:

```
let worstInp = 0;

const observer = new PerformanceObserver((list, obs, options) => {
  for (let entry of list.getEntries()) {
    if (!entry.interactionId) continue;

    entry.renderTime = entry.startTime + entry.duration;
    worstInp = Math.max(entry.duration, worstInp);

    console.log('[Interaction]', entry.duration, `type: ${entry.name} interactionCount:
${performance.interactionCount}, worstInp: ${worstInp}`, entry, options);
  }
});

observer.observe({
  type: 'event',
  durationThreshold: 0, // 16 minimum by spec
  buffered: true
});
```

A more convenient way of using the preceding code is to use the snippets feature in Chrome DevTools.

Once you have determined that the interaction is reliably slow, you can then profile the interaction to get more detailed information on *why* that interaction is slow.

What if you can't reproduce a slow interaction?

What if you've managed to find something in your field data that suggests a particular interaction is slow, but you can't reproduce it in the lab?

For one, this is a common challenge in troubleshooting performance issues of any kind. Your point of view when testing is entirely relative and dependent on the hardware you're using. After all, you may be on a fast device with a fast internet connection—but that doesn't mean your users are too.

For this situation you can do one of three things:

1. If you have a physical Android device, use remote debugging to open a Chrome DevTools instance on your host machine and try to reproduce slow interactions there.
2. If you don't have a physical device, enable the CPU throttling feature in Chrome DevTools.
3. Follow both steps 1 and 2, as you can also enable CPU throttling on the DevTools instance for a physical Android device.

Another cause could be that you're waiting for a page to load before you interact with it, but your users are not. If you're on a fast network, enable network throttling and interact with the page as soon as it paints. You should do this because the main thread is often busiest during startup, and testing at that time might help reveal what your users are actually experiencing.

Record a trace

To get more information on why an interaction is slow, you'll need to take things to the next level by using the performance profiler in Chrome DevTools. To profile an interaction in Chrome's performance profiler, do the following:

1. With the page you need to profile loaded and ready for interactions, open Chrome DevTools and go to the **Performance** panel.
2. Click the **Record** button at the upper left of the panel to start tracing.
3. Perform the desired interaction.
4. Click the **Record** button again to stop tracing.

When the profile populates, the first place to look should be the activity summary at the top of the profiler. The activity summary will show red bars at the top where long tasks occurred in the recording. This allows you to quickly zoom in on problem areas.

> *A screenshot of the activity summary in the performance panel of Chrome DevTools. The activity displayed is mostly from JavaScript that causes a long task, which is highlighted in red above the flame chart.*
>
> The activity summary at the top of Chrome's performance profiler. Long tasks are highlighted in red above the activity flame chart. In this case, significant scripting work was responsible for most of the work in the long task.

You can quickly focus on the problem area by dragging and selecting a region in the activity summary. Once you've focused to where the interaction occurred, the **Interactions** track will help you line up the interaction and the activity that occurred in the main thread track below it:

> *A screenshot of an interaction as visualized in the performance panel of Chrome DevTools. An interactions track above the main thread track shows the duration of an*

interaction, which can be lined up with main thread activity.

An interaction profiled in the performance profiler in Chrome's DevTools. The **Interactions** track shows a series of events that amount to a click interaction. The Interactions track entries span across the tasks responsible for driving the interaction.

From here, it's a matter of observing what the problem with the interaction might be. There are many things that can contribute to high interaction latency, so let's go through what some of the culprits could be.

Use Lighthouse timespans as an alternative to tracing

Chrome's performance profiler—while rich with diagnostic information—can be a bit intimidating to the uninitiated. One alternative to the performance profiler is Lighthouse's timespan mode. To use this mode, do the following:

1. With DevTools open, go to the **Lighthouse** tab in DevTools.
2. Under the section labeled **Mode**, select the **Timespan** option.
3. Select the desired device type under the section labeled **Device**.
4. Ensure at least the checkbox labeled **Performance** is selected under the **Categories** label.
5. Click the **Start timespan** button.
6. Test the desired interaction(s) on the page.
7. Click the **End timespan** button and wait for the audit to appear
8. Once the audit populates in the Lighthouse tab, you can filter the audits by INP by clicking the **INP** link next to the label which reads **Show audits relevant to**.

At this point, you'll see a drop-down list for audits that have failed or passed. When you expand that drop-down, you'll probably see a breakdown of time spent during the interaction.

A screenshot of a Lighthouse audit provided by its timespan mode. The audit is specific to INP, and shows details for an interaction, including a screenshot of the element that triggered it, and a table beneath detailing where time was spent processing the interaction.

An interaction profiled in Lighthouse's timespan mode. When interactions are made with the page, Lighthouse provides an audit detailing where the time during an interaction was spent, and breaks it down by input delay, processing delay, and presentation delay.

If you want to further investigate the cause behind a slow interaction, you can do so by clicking the **View Trace** button just above the filmstrip. This will take you to the performance profiler that shows interactions details for the interactions that you tested in the timespan mode.

How to identify long input delays

One thing that could be contributing to high interaction latency is input delay. The *input delay* is the first phase of an interaction. This is the period of time from when the user action is first received by the operating system until the browser is able to start processing the first event triggered by that input. The input delay ends right as the event callbacks for the interaction begin to run.

Identifying input delay in Chrome's performance profiler can be done by finding the start of an interaction in the interactions panel, and then finding the beginning of when the event callbacks for that interaction start to run.

You'll always incur at least some input delay, as it takes some time for the operating system to pass the input event to the browser—but you do have some control over how long the input delay is. **The key is to figure out if there is work running on the main thread that's preventing your callbacks from running.**

> *A depiction of input delay in Chrome's performance panel. The start of the interaction comes significantly before the event callbacks because of increased input delay due to a timer firing from a third-party script.*
>
> Input delay caused by a task fired by a timer from a third-party script.

In the previous figure, a task from a third-party script is running as the user attempts to interact with the page, and therefore extends the input delay. The extended input delay affects the interaction's latency, and could therefore affect the page's INP.

For more information on how you can resolve long input delays, read about how you can identify and reduce input delay.

How to identify expensive event callbacks

Event callbacks occur immediately after the input delay. If an event callback runs too long, it delays the browser from presenting the next frame, and can add significantly to an interaction's total latency. Event callbacks can often run for too long, whether they run as the result of first-party or third-party JavaScript—and in some cases, both.

A depiction of event callback tasks in Chrome's performance panel. The event callbacks occur for the pointerdown and click events, which occur in a long task.

The event callbacks that run in response to a click interaction, as shown in the performance profiler in Chrome DevTools. Note the red triangle in the upper right corner of both the **Event: pointerdown** and **Event: click** entries, which identifies expensive event callbacks.

Finding expensive event callbacks can be done by observing the following in a trace for a specific interaction:

1. Determine whether the task associated with the event callbacks is a long task. To reveal long tasks in a lab setting more reliably, you may need to enable CPU throttling in the performance panel, or connect a low to mid-tier Android device and use remote debugging.
2. If the task that runs the event callbacks is a long task, look for event handler entries (entries with names such as **Event: click**, for example) in the call stack that have a red triangle at the upper right corner of the entry. These are expensive event callbacks.

To address expensive event callbacks, try one of the following strategies:

1. **Do as little work as possible.** Is everything that happens in an expensive event callback strictly necessary? If not, consider removing that code altogether if you can, or deferring its execution to a later point in time if you can't. You can also take advantage of framework features to help. For example, React's `PureComponent` class and memoization feature can skip unnecessary rendering work when props and state haven't changed for a component.
2. **Defer non-rendering work in the event callback to a later point in time.** Long tasks can be broken up by yielding to the main thread. Whenever you yield to the main thread, you're ending execution of the current task and breaking up the remainder of the work into a separate task. This gives the renderer a chance to process updates to the user interface that were processed earlier in the event callback. If you happen to be developing in React, its transitions feature will do this for you.

By employing these strategies, you should be able to get your event callbacks in a place where they're responding more quickly to user input.

How to identify presentation delays

Long input delays and expensive event callbacks aren't the only possible culprits of poor INP. Sometimes the rendering updates that occur in response to even small amounts of event callback code can be expensive. The time it takes for the browser to render visual updates to the user interface to reflect the result of an interaction is known as *presentation delay*.

The rendering performance guide is a good primer on understanding how the browser renders a page. Life of a pixel goes further in depth on how rendering works in Chromium-based browsers.

> *Rendering work as visualized in the performance panel of Chrome DevTools. The rendering work occurs after the event callback in order to paint the next frame.*
>
> Rendering tasks as shown in Chrome's performance profiler. The rendering work is shown in purple, with paint work in green.

When profiling slow interactions due to expensive rendering work, you should enable the **Screenshots** checkbox at the top of the performance profiler. This will help you to line up user interface updates with the corresponding rendering work on the main thread.

Of all the possible causes of high interaction latency, rendering work can be the most difficult to troubleshoot and fix, but the result is worth the effort. Excessive rendering work could be caused by any of the following:

- **Large DOM sizes.** Rendering work often increases along with the size of the page's DOM. For more information, read How large DOM sizes affect interactivity—and what you can do about it.
- **Forced reflows.** This happens when you apply style changes to elements in JavaScript, and *then* query the results of that work. The result is that the browser has to perform the layout work before doing anything else, so that the browser can return the updated styles. For more information and tips on avoiding forced reflows, read Avoid large, complex layouts and layout thrashing.
- **Excessive or unnecessary work in `requestAnimationFrame` callbacks.** `requestAnimationFrame()` callbacks are run during the rendering phase of the event loop, and must complete before the next frame can be presented. If you're using `requestAnimationFrame()` to do work that doesn't involve changes to the user interface, understand that you could be delaying the next frame.
- `ResizeObserver` **callbacks.** Such callbacks run prior to rendering, and may delay presentation of the next frame if the work in them is expensive. As with event callbacks, defer any logic not needed for the next frame.

Troubleshooting might be made easier by enabling the **Timeline: event initiators** experiment in Chrome DevTools. Enabling this experiment will draw an arrow from any style recalculation task to the JavaScript task that kicked it off. To enable this experiment, go into the Chrome DevTools settings, click on **Experiments** in the left hand menu, the click the checkbox labeled **Timeline: event initiators**, and restart DevTools.

Troubleshooting INP is an iterative process

Finding out what's causing high interaction latency that contributes to poor INP takes a lot of work—but if you can pin down the causes, you're halfway there. By following a methodical approach to troubleshooting poor INP, you can reliably pin down what's causing a problem, and arrive more quickly to the right fix. To review:

- Rely on field data to find slow interactions.
- Test problematic field interactions in the lab to see if they're reproducible.
- Identify whether the cause is due to long input delay, expensive event callbacks, or expensive rendering work.
- Repeat.

The last of these is the most important. Like most other work you must do to improve page performance, troubleshooting and improving INP is a cyclical process. When you fix one slow interaction, you'll need to move onto the next, and continue to do so until you start to see results. Stay vigilant!

Hero image from Unsplash, by Louis Reed.

How large DOM sizes affect interactivity, and what you can do about it

There's no way around it: when you build a web page, that page is going to have a Document Object Model (DOM). The DOM represents the structure of your page's HTML, and gives JavaScript and CSS access to a page's structure and contents.

The problem, however, is that the *size* of the DOM affects a browser's ability to render a page quickly and efficiently. Generally speaking, the larger a DOM is, the more expensive it is to initially render that page and update its rendering later on in the page lifecycle.

This becomes problematic in pages with very large DOMs when interactions that modify or update the DOM trigger expensive layout work that affects the ability of the page to respond quickly. Expensive layout work can affect a page's Interaction to Next Paint (INP); If you want a page to respond quickly to user interactions, it's important to ensure your DOM sizes are only as large as necessary.

When is a page's DOM *too* large?

It's important to know the difference between DOM elements and DOM nodes. A DOM *element* refers to a specific HTML element in the DOM tree. A DOM *node* has overlapping meaning with the DOM element term, but its definition is expanded to include comments, whitespace, and text. While the Lighthouse DOM size audit refers to DOM nodes, this guide will refer to DOM elements over nodes whenever possible.

According to Lighthouse, a page's DOM size is excessive when it exceeds 1,400 nodes. Lighthouse will begin to throw warnings when a page's DOM exceeds 800 nodes. Take the following HTML for example:

```
<ul>
  <li>List item one.</li>
  <li>List item two.</li>
```

```
  <li>List item three.</li>
</ul>
```

In the above code, there are four DOM elements: the `<ul>` element, and its three `<li>` child elements. Your web page will almost certainly have many more nodes than this, so it's important to understand what you can do to keep DOM sizes in check—as well as other strategies to optimize the rendering work once you've gotten a page's DOM as small as it can be.

How do large DOMs affect page performance?

Large DOMs affect page performance in a few ways:

1. During the page's initial render. When CSS is applied to a page, a structure similar to the DOM known as the CSS Object Model (CSSOM) is created. As CSS selectors increase in specificity, the CSSOM becomes more complex, and more time is needed to run the necessary layout, styling, compositing, and paint work necessary to draw the web page to the screen. This added work increases interaction latency for interactions that occur early on during page load.
2. When interactions modify the DOM, either through element insertion or deletion, or by modifying DOM contents and styles, the work necessary to render that update can result in very costly layout, styling, compositing, and paint work. As is the case with the page's initial render, an increase in CSS selector specificity can add to rendering work when HTML elements are inserted into the DOM as the result of an interaction.
3. When JavaScript queries the DOM, references to DOM elements are stored in memory. For example, if you call `document.querySelectorAll` to select all `<div>` elements on a page, the memory cost could be considerable if the result returns a large number of DOM elements.

   A screenshot of a long task caused by excessive rendering work in the performance panel of Chrome DevTools. The long task's call stack shows significant time spent recalculating page styles, as well as pre-paint.

   A long task as shown in the performance profiler in Chrome DevTools. The long task shown is caused by inserting DOM elements into a large DOM via JavaScript.

All of these can affect interactivity, but the second item in the list above is of particular importance. If an interaction results in a change to the DOM, it can kick off a lot of work that can contribute to a poor INP on a page.

How do I measure DOM size?

You can measure DOM size in a couple of ways. The first method uses Lighthouse. When you run an audit, statistics on the current page's DOM will be in the "Avoid an excessive DOM size" audit under the "Diagnostics" heading. In this section, you can see the total number of DOM elements, the DOM element containing the most child elements, as well as the deepest DOM element.

A simpler method involves using the JavaScript console in the developer tools in any major browser. To get the total number of HTML elements in the DOM, you can use the following code in the console after the page has loaded:

```
document.querySelectorAll('*').length;
```

Note that the code snippet above only includes the number of HTML *elements* in the DOM. It doesn't include all the *nodes* in the DOM.

If you want to see the DOM size update in realtime, you can also use the performance monitor tool. Using this tool, you can correlate layout and styling operations (and other performance aspects) along with the current DOM size.

> *A screenshot of the performance monitor in Chrome DevTools. At left, there are various aspects of page performance that can be continuously monitored during the life of the page. In the screenshot, the number of DOM nodes, layouts per second, and style recalculations per section are actively being monitored.*
>
> The performance monitor in Chrome DevTools. In this view, the page's current number of DOM nodes is charted along with layout operations and style recalculations performed per second.

If the DOM's size is approaching Lighthouse DOM size's warning threshold—or fails altogether—the next step is to figure out how to reduce the DOM's size to improve your page's ability to respond to user interactions so that your website's INP can improve.

How can I measure the number of DOM elements affected by an interaction?

If you're profiling a slow interaction in the lab that you suspect might have something to do with the size of the page's DOM, you can figure out how many DOM elements were affected by selecting any piece of activity in the profiler labeled "Recalculate Style" and observe the contextual data in the bottom panel.

> *A screenshot of selected style recalculation activity in the performance panel of Chrome DevTools. At top, the interactions track shows a click interaction, and the majority of the work is spent doing style recalculation and pre-paint work. At the bottom, a panel shows more detail for the selected activity, which reports that 2,547 DOM elements were affected.*
>
> Observing the number of affected elements in the DOM as the result of style recalculation work. Note that the shaded portion of the interaction in the interactions track represents the portion of the interaction duration that was over 200 milliseconds, which is the designated "good" threshold for INP.

In the above screenshot, observe that the style recalculation of the work—when selected—shows the number of affected elements. While the above screenshot shows an extreme case of the effect of DOM size on rendering work on a page with many DOM elements, this diagnostic info is useful in any case to determine if the size of the DOM is a limiting factor in how long it takes for the next frame to paint in response to an interaction.

How can I reduce DOM size?

Beyond auditing your website's HTML for unnecessary markup, the principal way to reduce DOM size is to reduce DOM depth. One signal that your DOM might be unnecessarily deep is if you're seeing markup that looks something like this in the **Elements** tab of your browser's developer tools:

```
<div>
  <div>
    <div>
      <div>
        <!-- Contents -->
      </div>
    </div>
  </div>
</div>
```

When you see patterns like this, you can probably simplify them by flattening your DOM structure. Doing so will reduce the number of DOM elements, and likely give you an opportunity to simplify page styles.

DOM depth may also be a symptom of the frameworks you use. In particular, component-based frameworks—such as those that rely on JSX—require you to nest multiple components in a parent container.

However, many frameworks allow you to avoid nesting components by using what are known as fragments. Component-based frameworks that offer fragments as a feature include (but are not limited to) the following:

- React
- Preact
- Vue
- Svelte

By using fragments in your framework of choice, you can reduce DOM depth. If you're concerned about the impact flattening DOM structure has on styling, you might benefit from using more modern (and faster) layout modes such as flexbox or grid.

Other strategies to consider

Even if you take pains to flatten your DOM tree and remove unnecessary HTML elements to keep your DOM as small as possible, it can still be quite large and kick off a lot of rendering work as it changes in response to user interactions. If you find yourself in this position, there are some other strategies you can consider to limit rendering work.

Consider an additive approach

You might be in a position where large parts of your page aren't initially visible to the user when it first renders. This could be an opportunity to lazy load HTML by omitting those parts of the DOM on startup, but add them in when the user interacts with the parts of the page that require the initially hidden aspects of the page.

This approach is useful both during the initial load and perhaps even afterwards. For the initial page load, you're taking on less rendering work up front, meaning that your initial HTML payload will be lighter, and will render more quickly. This will give interactions during that crucial period more opportunities to run with less competition for the main thread's attention.

If you have many parts of the page that are initially hidden on load, it could also speed up other interactions that trigger re-rendering work. However, as other interactions add more to the DOM, rendering work will increase as the DOM grows throughout the page lifecycle.

Adding to the DOM over time can be tricky, and it has its own tradeoffs. If you're going this route, you're likely making network requests to get data to populate the HTML you intend to add to the page in response to a user interaction. While in-flight network requests are not counted towards INP, it can increase perceived latency. If possible, show a loading spinner or other indicator that data is being fetched so that users understand that something is happening.

Limit CSS selector complexity

When the browser parses selectors in your CSS, it has to traverse the DOM tree to understand how—and if—those selectors apply to the current layout. The more complex these selectors are, the more work the browser has to do in order to perform both the initial rendering of the page, as well as increased style recalculations and layout work if the page changes as the result of an interaction.

Read to learn more: Reduce the scope and complexity of style calculations.

Use the `content-visibility` property

CSS offers the `content-visibility` property, which is effectively a way to lazily render off-screen DOM elements. As the elements approach the viewport, they're rendered on demand. The benefits of `content-visibility` don't just cut out a significant amount of rendering work on the initial page render, but also skip rendering work for offscreen elements when the page DOM is changed as the result of a user interaction.

Read to learn more: `content-visibility`: the new CSS property that boosts your rendering performance.

Conclusion

Reducing your DOM size to only what is strictly necessary is a good way to optimize your website's INP. By doing so, you can reduce the amount of time it takes for the browser to perform layout and rendering work when the DOM is updated. Even if you can't meaningfully reduce DOM size, there are some techniques you can use to isolate rendering work to a DOM subtree, such as CSS containment and the `content-visibility` CSS property.

However you go about it, creating an environment where rendering work is minimized—as well as reducing the amount of rendering work your page does in response to interactions—the result will be that your website will feel more responsive to users when they interact with them. That means you'll have a lower INP for your website, and that translates to a better user experience.

Hero image from Unsplash, by Louis Reed.

Find slow interactions in the field

Field data is *the* authoritative source when it comes to how actual users are experiencing your website. It teases out issues you may not see in lab data alone. While interactions can be simulated in lab-based tools, you're not going to be able to reproduce every single interaction in the lab in the way that users in the field experience them. Gathering field data for Interaction to Next Paint (INP) is critical to understanding how responsive your page is to real users, and it contains the clues to make their experience even better.

What you should collect in the field and why

When collecting INP data in the field, you'll want to capture the following to give context to why interactions were slow:

- **The INP value.** This is the key piece of data you'll need to collect. The distribution of these values will determine whether the page meets the INP thresholds.
- **The element selector string responsible for the page's INP.** Knowing a page's INP is good, but not good enough by itself. Without knowing what element was responsible for it, you'll be in the dark. By logging element selector strings, you'll know exactly what elements are responsible for interactions.
- **The loading state of the page for the interaction that is the page's INP.** When a page is loading, it's reasonable to assume that there's more main thread activity occurring that could result in higher interaction latency. During page load, there's HTML parsing, stylesheet parsing, and JavaScript evaluation and execution going on. Knowing whether an interaction has taken place during page load or afterwards helps you to figure out if you need to optimize for faster startup so interactions have more room on the main thread to run quickly, or if the interaction responsible for the page's INP in itself is slow regardless.
- **The interaction's `startTime`.** Logging the interaction's `startTime` lets you know exactly when the interaction occurred on the performance timeline.
- **The event type.** The event type is good to know, as it will tell you whether the interaction was the result of a `click`, `keypress`, or other eligible event type. A user

interaction may contain multiple callbacks, and can help you to pinpoint exactly which event callback in the interaction took the longest to run.

While this all seems like a lot to take into account, you don't have to reinvent the wheel to get there. Thankfully, this data is exposed in the `web-vitals` JavaScript library, and you'll learn how to gather it in the next section.

Measure interactions in the field with the `web-vitals` JavaScript library

The `web-vitals` JavaScript library is a great way to find slow interactions in the field, thanks in large part to its ability to provide attribution for what's causing them. INP can be collected in browsers that support the Event Timing API and its `interactionId` property.

Using a Real User Monitoring (RUM) provider to get INP is most convenient, but not always an option. If that's your case, for example, you can use the `web-vitals` JavaScript library to collect and transmit INP data to Google Analytics for later evaluation:

```
// Be sure to import from the attribution build:
import {onINP} from 'web-vitals/attribution';

function sendToGoogleAnalytics ({name, value, id, attribution}) {
  // Destructure the attribution object:
  const {eventEntry, eventTarget, eventType, loadState} = attribution;

  // Get timings from the event timing entry:
  const {startTime, processingStart, processingEnd, duration, interactionId} = eventEntry;

  const eventParams = {
    // The page's INP value:
    metric_inp_value: value,
    // A unique ID for the page session, which is useful
    // for computing totals when you group by the ID.
    metric_id: id,
    // The event target (a CSS selector string pointing
```

```
    // to the element responsible for the interaction):
    metric_inp_event_target: eventTarget,
    // The type of event that triggered the interaction:
    metric_inp_event_type: eventType,
    // Whether the page was loaded when the interaction
    // took place. Useful for identifying startup versus
    // post-load interactions:
    metric_inp_load_state: loadState,
    // The time (in milliseconds) after page load when
    // the interaction took place:
    metric_inp_start_time: startTime,
    // When processing of the event callbacks in the
    // interaction started to run:
    metric_inp_processing_start: processingStart,
    // When processing of the event callbacks in the
    // interaction finished:
    metric_inp_processing_end: processingEnd,
    // The total duration of the interaction. Note: this
    // value is rounded to 8 milliseconds of granularity:
    metric_inp_duration: duration,
    // The interaction ID assigned to the interaction by
    // the Event Timing API. This could be useful in cases
    // where you might want to aggregate related events:
    metric_inp_interaction_id: interactionId
  };

  // Send to Google Analytics
  gtag('event', name, eventParams);
}

// Pass the reporting function to the web-vitals INP reporter:
onINP(sendToGoogleAnalytics);
```

Where element attribution is concerned, not every interaction can be attributed to a specific element. This is due to a limitation of the underlying Event Timing API, which is used to calculate the page's INP. In this API, the event target may be absent if the element responsible for the interaction was disconnected from the DOM, or if the element was in the shadow DOM.

If you have Google Analytics and run the preceding code on your website, you'll get a detailed reporting of not just the page's INP, but also useful contextual information that can give you a

better sense of where to begin optimizing slow interactions.

Monitor full session duration, not just up to `onload`

Using the `web-vitals` JavaScript library, as previously mentioned, may result in multiple analytics events being sent to Google Analytics. This is fine, as the ID `web-vitals` generates for the current page will stay the same, and Google Analytics will allow you to overwrite previous data.

However, not all RUM providers operate this way, so if you're building your own RUM collection solution, you'll need to take this into account. If your current analytics provider won't allow overwrites of existing records, you'll need to record all the `delta` values—that is, the difference between a metric's past and current states— for a metric and transmit them using the same ID provided by the `web-vitals` library; then you can sum those changes by grouping on the ID. For more more information, consult the `web-vitals` documentation's section on handling deltas.

Get field data quickly with CrUX

The Chrome UX Report (CrUX) is the official dataset of the Web Vitals program. While data from CrUX alone doesn't give you all the information you need to troubleshoot specific INP issues, it does let you know whether you have a problem in the first place. Even if you're already collecting field data through a RUM provider, consider contrasting it with CrUX data for your website (if available), as there are differences in the methodologies they use.

You can evaluate your website's INP and view its CrUX data using PageSpeed Insights (PSI). PageSpeed Insights may provide page-level field data for websites that are included in the CrUX dataset. To audit a URL with PageSpeed Insights, go to https://pagespeed.web.dev/, enter a URL to test, and click the **Analyze** button.

The URL you enter may not be eligible for CrUX. If this is the case, you'll need to collect your own field data, which is explained in the previous section. Also, where page-level data is not available for the given URL in PageSpeed Insights, the provided field data will be aggregated over the entire website.

Once the assessment begins, CrUX data will be available instantly as Lighthouse (a lab tool) runs.

> *A screenshot of INP field data in a distribution. The distribution aligns with the INP thresholds, and in this example, the field INP value is 545 milliseconds, which lands in*

the poor range.

The distribution of INP experiences as seen in PageSpeed Insights.

When Lighthouse has finished running its assessment, PSI will populate the assessment with Lighthouse audits.

Because PageSpeed Insights is not scripted to interact with the page, there will be no audits relevant to INP. However, Total Blocking Time (TBT) is highly correlated with INP. Therefore, you can filter audits by TBT to find opportunities to reduce the amount of time a page blocks the main thread during the test.

A screenshot of Lighthouse audits as seen in PageSpeed Insights. The audits are filtered by the TBT metric, showing tips for minimizing main thread work, avoiding an excessive DOM size, and avoiding long main thread tasks.

Audits for Total Blocking Time

What if I don't have field data?

You might be in a situation where you don't have or can't even collect field data. If this describes your situation, then you'll be entirely dependent on lab tools in order to find slow interactions. For more information on lab testing, read How to diagnose what's causing poor INP in the lab.

Conclusion

Field data is the best source of information you can draw on when it comes to understanding which interactions are problematic for actual users in the field. By drawing on information available in PageSpeed Insights, or relying on field data collection tools such as the `web-vitals` JavaScript library (or your RUM provider), you can be more confident about which interactions are most problematic, and then move on to reproducing problematic interactions in the lab and then go about fixing them.

Hero image from Unsplash, by Federico Respini.

Optimize input delay

Interactions on the web are complicated things, with all sorts of activity occurring in the browser to drive them. What they all have in common, though, is that they incur some input delay before their event callbacks begin to run. In this guide, you'll learn what input delay is, and what you can do to minimize it so your website's interactions run faster.

What is input delay?

Input delay is the period of time beginning from when the user first interacts with a page—such as tapping on a screen, clicking with a mouse, or pressing a key—up to when the event callbacks for the interaction begin to run. Every interaction begins with some amount of input delay.

> *A simplified visualization of input delay. At left, there is line art of a mouse cursor with a starburst behind it, signifying the start of an interaction. To the right is line art of a cogwheel, signifying when the event handlers for an interaction begin to run. The space in between is noted as the input delay with a curly brace.*
>
> The mechanics behind input delay. When an input is received by the operating system, it must be passed to the browser before the interaction starts. This takes a certain amount of time, and can be increased by existing main thread work.

Some portion of the input delay is unavoidable: it always takes some amount of time for the operating system to recognize an input event and pass it to the browser. However, that portion of the input delay is often not even noticeable, and there are other things that happen on the page itself that can make input delays long enough to cause problems.

How to think about input delay

Generally speaking, you want to keep every part of an interaction as short as possible so that your website has the best chance of meeting the Interaction to Next Paint (INP) metric's "good" threshold, regardless of the user's device. Keeping input delay in check is just one part of meeting that threshold.

You might be tempted to look to the First Input Delay (FID) thresholds to determine an allowance for input delays—but the "good" threshold for FID is **100 milliseconds or less**. If you go by this threshold, you'd be allotting half of your budget for INP to input delay alone. This is inadvisable

when you consider that an interaction also requires time to run event callbacks and for the browser to paint the next frame.

To meet INP's "good" threshold, you'll want to aim for the shortest input delay possible, but you shouldn't expect to eliminate it entirely, as that is impossible. As long as you're avoiding excessive main thread work while users are attempting to interact with your page, your input delay should be low enough to avoid problems.

How to minimize input delay

As said previously, some input delay is unavoidable, but on the other hand, some input delay *is* avoidable. Here are some things to consider if you're struggling with long input delays.

Avoid recurring timers that kick off excessive main thread work

There are two commonly used timer functions in JavaScript that can contribute to input delay: `setTimeout` and `setInterval`. The difference between the two is that `setTimeout` schedules a callback to run after a specified time. `setInterval`, on the other hand, schedules a callback to run every *n* milliseconds in perpetuity, or until the timer is stopped with `clearInterval`.

`setTimeout` is not problematic in itself—in fact, it can be helpful in avoiding long tasks. However, it depends on *when* the timeout occurs, and whether the user attempts to interact with the page when the timeout callback runs.

Additionally, `setTimeout` can be run in a loop or recursively, where it acts more like `setInterval`, though preferably not scheduling the next iteration until the previous one is completed. While this means the loop will yield to the main thread every time `setTimeout` is called, you should take care to ensure its callback doesn't end up doing excessive work.

`setInterval` runs a callback on an interval, and is therefore much more likely to get in the way of interactions. This is because—unlike a single instance of a `setTimeout` call, which is a one-off callback that *may* get in the way of a user interaction—`setInterval`'s recurring nature makes it much more likely that it *will* get in the way of an interaction, thus increasing the interaction's input delay.

> *A screenshot of the performance profiler in Chrome DevTools demonstrating input delay. A task fired by a timer function occurs just before a user initiates a click interaction. However, the timer extends the input delay, causing the interaction's event callbacks to run later than they otherwise would.*

A timer registered by a previous `setInterval` call contributing to input delay as depicted in the performance panel of Chrome DevTools. The added input delay causes the event callbacks for the interaction to run later than they otherwise could.

If timers are occurring in first-party code, then you have control over them. Evaluate whether you need them, or do your best to reduce the work in them as much as possible. However, timers in third-party scripts are a different story. You often don't have control over what a third-party script does, and fixing performance issues in third-party code often involves working with stakeholders to determine whether a given third-party script is necessary, and if so, establish contact with a third-party script vendor to determine what can done to fix performance issues they may cause on your website.

Avoid long tasks

One way to mitigate long input delays is to avoid long tasks. When you have excessive main thread work that blocks the main thread during interactions, that will add to the input delay for them before the long tasks have had a chance to finish.

A visualization of how long tasks extend input delay. At top, an interaction occurs shortly after a single long task runs, causing significant input delay that causes event callbacks to run much later than they should. At bottom, an interaction occurs at roughly the same time, but the long task is broken up into several smaller ones by yielding, allowing the interaction's event callbacks to run much sooner.

A visualization of what happens to interactions when tasks are too long and the browser can't respond quickly enough to interactions, versus when longer tasks are broken up into smaller tasks.

Besides minimizing the amount of work you do in a task—and you should *always* strive to do as little work as possible on the main thread—you can improve responsiveness to user input by breaking up long tasks.

Yielding isn't foolproof, as tasks from third-party JavaScript can still make their way in between first-party tasks after yielding. This can still contribute to input delay for future interactions, and you'll need to be mindful of third-party tasks that may run on an interval, as described in the previous section. This is a problem that the proposed `scheduler.yield` API is looking to solve.

Be mindful of interaction overlap

A particularly challenging part of optimizing INP can be if you have interactions that overlap. Interaction overlap means that after you've interacted with one element, you make another interaction with the page before the initial interaction has had a chance to render the next frame.

A depiction of when tasks can overlap to produce long input delays. In this depiction, a click interaction overlaps with a keydown interaction to increase the input delay for the keydown interaction.

A visualization of two concurrent interactions in the performance profiler in Chrome's DevTools. The rendering work in the initial click interaction causes an input delay for the subsequent keyboard interaction.

Sources of interaction overlap may be as simple as users making many interactions in a short period of time. This can occur when users type in form fields, where many keyboard interactions can occur in a very short period of time. If the work on a key event is especially expensive—such as in the common case of autocomplete fields where network requests are made to a back end—you have a couple of options:

- Consider debouncing inputs to limit the amount of times an event callback executes in a given period of time.
- Use `AbortController` to cancel outgoing `fetch` requests so the main thread doesn't become congested handling `fetch` callbacks. Note: an `AbortController` instance's `signal` property can also be used to abort events.

Another source of increased input delay due to overlapping interactions can be expensive animations. In particular, animations in JavaScript may fire many `requestAnimationFrame` calls, which may get in the way of user interactions. To get around this, use CSS animations whenever possible to avoid queueing potentially expensive animation frames—but if you do this, make sure you avoid non-composited animations so that animations run mainly on the GPU and compositor threads, and not on the main thread.

Conclusion

While input delays may not represent the majority of the time your interactions take to run, it's important to understand that every part of an interaction takes up an amount of time that you can reduce. If you're observing long input delay, then you have opportunities to reduce it. Avoiding recurring timer callbacks, breaking up long tasks, and being aware of potential interaction overlap can all help you to reduce input delay, leading to faster interactivity for your website's users.

Hero image from Unsplash, by Erik Mclean.

Script evaluation and long tasks

When it comes to optimizing Interaction to Next Paint (INP), most of the advice you'll encounter is to optimize interactions themselves. For example, in the optimize long tasks guide, techniques such as yielding with `setTimeout`, `isInputPending`, and so forth are discussed. These techniques are beneficial, as they allow the main thread some breathing room by avoiding long tasks, which can allow more opportunities for interactions and other activity to run sooner, rather than if they had to wait for a single long task.

However, what about the long tasks that come from loading scripts themselves? These tasks can interfere with user interactions and affect a page's INP during load. This guide will explore how browsers handle tasks kicked off by script evaluation, and look into what you may be able to do to break up script evaluation work so that your main thread can be more responsive to user input while the page is loading.

What is script evaluation?

If you've profiled an application that ships a lot of JavaScript, you may have seen long tasks where the culprit is labeled **Evaluate Script**.

> *Script evaluation work as visualized in the performance profiler of Chrome DevTools. The work causes a long task during startup, which blocks the main thread's ability to respond to user interactions.*
>
> Script evaluation work as shown in the performance profiler in Chrome DevTools. In this case, the work is enough to cause a long task that blocks the main thread from taking on other work—including tasks that drive user interactions.

Script evaluation is a necessary part of executing JavaScript in the browser, as JavaScript is compiled just-in-time before execution. When a script is evaluated, it is first parsed for errors. If the parser doesn't find errors, the script is then compiled into bytecode, and then can continue onto execution.

Though necessary, script evaluation can be problematic, as users may try to interact with a page shortly after it initially renders. However, just because a page has *rendered* doesn't mean that the page has finished *loading*. Interactions that take place during load can be delayed because the

page is busy evaluating scripts. While there's no guarantee that the desired interaction can take place at this point in time—as a script responsible for it may not have loaded yet—there could be interactions dependent on JavaScript that *are* ready, or the interactivity doesn't depend on JavaScript at all.

One metric that might give you insight into whether you have excessive script evaluation occurring during page load is Total Blocking Time (TBT), as it is a load responsiveness metric. Because TBT correlates well with INP, a page with a high TBT is a reasonable indicator that there may be high INP values during load that may be tied to script evaluation work.

The relationship between scripts and the tasks that evaluate them

How tasks responsible for script evaluation are kicked off depends on whether the script you're loading is loaded via a regular `<script>` element, or if a script is a module loaded with the `type=module`. Since browsers have the tendency to handle things differently, how the major browser engines handle script evaluation will be touched upon where script evaluation behaviors across them vary.

Loading scripts with the `<script>` element

This section deals with loading scripts using the `<script>` element *without* the `type=module` attribute.

The number of tasks dispatched to evaluate scripts generally has a direct relationship with the number of `<script>` elements on a page. Each `<script>` element kicks off a task to evaluate the requested script so it can be parsed, compiled, and executed. **This is the case for Chromium-based browsers, Safari, *and* Firefox.**

Why does this matter? Let's say you're using a bundler to manage your production scripts, and you've configured it to bundle everything your page needs to run into a single script. If this is the case for your website, you can expect that there will be a single task dispatched to evaluate that script. Is this a bad thing? Not necessarily—unless that script is *huge*.

You can break up script evaluation work by avoiding loading large chunks of JavaScript, and load more individual, smaller scripts using additional `<script>` elements.

As devices vary in their capability, it's very difficult to define a set limit for how large individual scripts should be. To achieve a decent balance between compression efficiency, download time, and script evaluation time, a limit of 100 kilobytes per individual script is a good target.

While you should always strive to load as little JavaScript as possible during page load, splitting up your scripts ensures that, instead of one large task that may block the main thread, you have a greater number of smaller tasks that won't block the main thread at all—or at least less than what you started with.

> *Multiple tasks involving script evaluation as visualized in the performance profiler of Chrome DevTools. Because multiple smaller scripts are loaded instead of fewer larger scripts, tasks are less likely to become long tasks, allowing the main thread to respond to user input more quickly.*
>
> Multiple tasks spawned to evaluate scripts as a result of multiple `<script>` elements present in the page's HTML. This is preferable to sending one large script bundle to users, which is more likely to block the main thread.

You can think of breaking up tasks for script evaluation as being somewhat similar to yielding during event callbacks that run during an interaction. However, with script evaluation, the yielding mechanism breaks up the JavaScript you load into multiple smaller scripts, rather than a smaller number of larger scripts than are more likely to block the main thread.

Currently, Chromium-based browsers will execute all loaded scripts with the `defer` attribute in the same task as the `DOMContentLoaded` event. While this minimizes overall layout work, the cost is the increased likelihood of a longer task, which may cause other performance problems. A potential solution is currently being explored. This behavior also takes place with scripts loaded using the `type=module` attribute, as such scripts are deferred by default.

Loading scripts with the `<script>` element and the `type=module` attribute

If you're not bundling ES modules and loading them using the `type=module` attribute, you could actually end up slowing down page startup. For more information, read the trade-offs and considerations section later on in this guide.

It's now possible to load ES modules natively in the browser with the `type=module` attribute on the `<script>` element. This approach to script loading carries some developer experience benefits, such as not having to transform code for production use—especially when used in combination with import maps. However, loading scripts in this way schedules tasks that differ from browser to browser.

Chromium-based browsers

In browsers such as Chrome—or those derived from it—loading ES modules using the `type=module` attribute produces different sorts of tasks than you'd normally see when not using `type=module`. For example, a task for each module script will run that involves activity labeled as **Compile module**.

> *Module compilation work in multiple tasks as visualized in Chrome DevTools.*
>
> Module loading behavior in Chromium-based browsers. Each module script will spawn a **Compile module** call to compile their contents prior to evaluation.

Once the modules have compiled, any code that subsequently runs in them will kick off activity labeled as **Evaluate module**.

> *Just-in-time evaluation of a module as visualized in the performance panel of Chrome DevTools.*
>
> When code in a module runs, that module will be evaluated just-in-time.

The effect here—in Chrome and related browsers, at least—is that the compilation steps are broken up when using ES modules. This is a clear win in terms of managing long tasks; however, the resulting module evaluation work that results still means you're incurring some unavoidable cost. While you should strive to ship as little JavaScript as possible, using ES modules—regardless of the browser—provides the following benefits:

- All module code is automatically run in strict mode, which allows potential optimizations by JavaScript engines that couldn't otherwise be made in a non-strict context.
- Scripts loaded using `type=module` are treated as if they were deferred by default. It's possible to use the `async` attribute on scripts loaded with `type=module` to change this behavior.

Safari and Firefox

When modules are loaded in Safari and Firefox, each of them is evaluated in a separate task. This means you could theoretically load a single top-level module consisting of only static `import` statements to other modules, and every module loaded will incur a separate network request and task to evaluate it.

Loading scripts with dynamic `import()`

Dynamic `import()` is another method for loading scripts. Unlike static `import` statements that are required to be at the top of an ES module, a dynamic `import()` call can appear anywhere in a script to load a chunk of JavaScript on demand. This technique is called code splitting.

Dynamic `import()` has two advantages when it comes to improving INP:

1. Modules which are deferred to load later reduce main thread contention during startup by reducing the amount of JavaScript loaded at that time. This frees up the main thread so it can be more responsive to user interactions.
2. When dynamic `import()` calls are made, each call will effectively separate the compilation and evaluation of each module to its own task. Of course, a dynamic `import()` that loads a very large module will kick off a rather large script evaluation task, and that can interfere with the ability of the main thread to respond to user input if the interaction occurs at the same time as the dynamic `import()` call. **Therefore, it's still very important that you load as little JavaScript as possible.**

Dynamic `import()` calls behave similarly in all major browser engines: the script evaluation tasks that result will be the same as the amount of modules that are dynamically imported.

Loading scripts in a web worker

Web workers are a special JavaScript use case. Web workers are registered on the main thread, and the code within the worker then runs on its own thread. This is hugely beneficial in the sense that—while the code that registers the web worker runs on the main thread—the code within the web worker doesn't. This reduces main thread congestion, and can help keep the main thread more responsive to user interactions.

In addition to reducing main thread work, web workers *themselves* can load external scripts to be used in the worker context, either through `importScripts` or static `import` statements in browsers that support module workers. The result is that any script requested by a web worker is evaluated off the main thread.

Read to learn more: Use web workers to run JavaScript off the browser's main thread.

Trade-offs and considerations

While breaking up your scripts into separate, smaller files helps limit long tasks as opposed to loading fewer, much larger files, it's important to take some things into account when deciding how to break scripts up.

Compression efficiency

Compression is a factor when it comes to breaking up scripts. When scripts are smaller, compression becomes somewhat less efficient. Larger scripts will benefit much more from compression. While increasing compression efficiency helps to keep load times for scripts as low as possible, it's a bit of a balancing act to ensure that you're breaking up scripts into enough smaller

chunks to facilitate better interactivity during startup.

Bundlers are ideal tools for managing the output size for the scripts your website depends on:

- Where webpack is concerned, its `SplitChunksPlugin` plugin can help. Consult the `SplitChunksPlugin` documentation for options you can set to help manage asset sizes.
- For other bundlers such as Rollup and esbuild, you can manage script file sizes by using dynamic `import()` calls in your code. These bundlers—as well as webpack—will automatically break off the dynamically imported asset into its own file, thus avoiding larger initial bundle sizes.

Cache invalidation

Cache invalidation plays a big role in how fast a page loads on repeat visits. When you ship large, monolithic script bundles, you're at a disadvantage when it comes to browser caching. This is because when you update your first-party code—either through updating packages or shipping bug fixes—the entire bundle becomes invalidated and must be downloaded again.

By breaking up your scripts, you're not just breaking up script evaluation work across smaller tasks, you're also increasing the likelihood that return visitors will grab more scripts from the browser cache instead of from the network. This translates into an overall faster page load.

In order for caching to be both efficient and to avoid stale resources from being served from the cache, be sure that your bundler is generating resources with a hash in the file name.

Nested modules and loading performance

If you're shipping ES modules in production and loading them with the `type=module` attribute, you need to be aware of how module nesting can impact startup time. Module nesting refers to when an ES module statically imports another ES module that statically imports another ES module:

```
// a.js
import {b} from './b.js';

// b.js
import {c} from './c.js';
```

If your ES modules are not bundled together, the preceding code results in a network request chain: when `a.js` is requested from a `<script>` element, another network request is dispatched for `b.js`, which then involves *another* request for `c.js`. One way to avoid this is to use a bundler—but be sure you're configuring your bundler to break up scripts to spread out script evaluation work.

If you don't want to use a bundler, then another way to get around nested module calls is to use the `modulepreload` resource hint, which will preload ES modules ahead of time to avoid network request chains. Beware, however: this hint is currently only supported in Chromium-based browsers.

Conclusion

Optimizing evaluation of scripts in the browser is no doubt a tricky feat. The approach depends on your website's requirements and constraints. However, by splitting up scripts, you're spreading the work of script evaluation over numerous smaller tasks, and therefore giving the main thread the ability to handle user interactions more efficiently, rather than blocking the main thread.

To recap, here are some things you can to do to break up large script evaluation tasks:

- When loading scripts using the `<script>` element without the `type=module` attribute, avoid loading scripts that are very large, as these will kick off resource-intensive script evaluation tasks that block the main thread. Spread out your scripts over more `<script>` elements to break up this work.
- Using the `type=module` attribute to load ES modules natively in the browser will kick off individual tasks for evaluation for each separate module script.
- Reduce the size of your initial bundles by using dynamic `import()` calls. This also works in bundlers, as bundlers will treat each dynamically imported module as a "split point," resulting in a separate script being generated for each dynamically imported module.
- Be sure to weigh trade-offs such as compression efficiency and cache invalidation. Larger scripts will compress better, but are more likely to involve more expensive script evaluation work in fewer tasks, and result in browser cache invalidation, leading to overall lower caching efficiency.
- If using ES modules natively without bundling, use the `modulepreload` resource hint to optimize the loading of them during startup.
- As always, ship as little JavaScript as possible.

It's a balancing act for sure—but by breaking up scripts and reducing initial payloads via dynamic `import()`, you can achieve better startup performance and better accommodate user interactions during that crucial startup period. This should help you score better on the INP metric, thus delivering a better user experience.

Hero image from Unsplash, by Markus Spiske.