Custom metrics

- APIs to measure custom metrics
  - Performance Observer
    - Observing entries that already happened
    - Legacy performance APIs to avoid
  - User Timing API
  - Long Tasks API
  - Element Timing API
  - Event Timing API
  - Resource Timing API
  - Navigation Timing API
  - Server Timing API

There's a lot of value in having user-centric metrics that you can measure, universally, on any given website. These metrics allow you to:

- Understand how real users experience the web as a whole
- Easily compare your site to a competitor's
- Track useful and actionable data in your analytics tools without needing to write custom code

Universal metrics offer a good baseline, but in many cases you need to measure *more* than just these metrics in order to capture the full experience for your particular site.

Custom metrics allow you to measure aspects of your site's experience that may only apply to your site, such as:

- How long it takes for a single page app (SPA) to transition from one "page" to another
- How long it takes for a page to display data fetched from a database for logged-in users
- How long it takes for a server-side-rendered (SSR) app to hydrate
- The cache hit rate for resources loaded by returning visitors
- The event latency of click or keyboard events in a game

APIs to measure custom metrics

Historically web developers haven't had many low-level APIs to measure performance, and as a result they've had to resort to hacks in order to measure whether a site was performing well.

For example, it's possible to determine whether the main thread is blocked due to long-running JavaScript tasks by running a `requestAnimationFrame` loop and calculating the delta between each frame. If the delta is significantly longer than the display's framerate, you can report that as a long task. Such hacks are not recommended, though, because they actually affect performance themselves (by draining battery, for example).

The first rule of effective performance measurement is to make sure your performance measurement techniques aren't causing performance issues themselves. So for any custom metrics you measure on your site, it's best to use one of the following APIs if possible.

Performance Observer

Understanding the PerformanceObserver API is critical to creating custom performance metrics because it's the mechanism by which you get data from all other performance APIs discussed in this article.

With `PerformanceObserver` you can subscribe passively to performance-related events, which means these APIs generally will not interfere with the performance of the page, as their callbacks are generally fired during idle periods.

You create a `PerformanceObserver` by passing it a callback to be run whenever new performance entries are dispatched. Then you tell the observer what types of entries to listen for via the `observe()` method:

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // Log the entry and all associated details.
      console.log(entry.toJSON());
    }
  });

  po.observe({type: 'some-entry-type'});
} catch (e) {
```

```
  // Do nothing if the browser doesn't support this API.
}
```

The sections below list all the various entry types available for observing, but in newer browsers you can inspect what entry types are available via the static `PerformanceObserver.supportedEntryTypes` property.

The object passed to the `observe()` method can also specify an `entryTypes` array (in order to observe more than one entry type via the same observer). While specifying `entryTypes` is an older option with wider browser support, using `type` is now preferred, as it allows for specifying additional entry-specific observation configuration (such as the `buffered` flag, discussed next).

Observing entries that already happened

By default, `PerformanceObserver` objects can only observe entries as they occur. This can be problematic if you want to load your performance analytics code lazily (to not block higher-priority resources).

To get historical entries (after they've occurred), set the `buffered` flag to `true` when you call `observe()`. The browser will include historical entries from its performance entry buffer the first time that your `PerformanceObserver` callback is called.

```
po.observe({
  type: 'some-entry-type',
  buffered: true,
});
```

To avoid memory issues, the performance entry buffer is not unlimited. For most typical page loads it's unlikely that the buffer will fill up and entries will be missed.

Legacy performance APIs to avoid

Prior to the Performance Observer API, developers could access performance entries using the following three methods defined on the `performance` object:

- `getEntries()`
- `getEntriesByName()`
- `getEntriesByType()`

While these APIs are still supported, their usage is not recommended because they don't allow you to listen for when new entries are emitted. In addition, many new APIs (such as Long Tasks) are not exposed via the `performance` object, they're only exposed via `PerformanceObserver`.

Unless you specifically need Internet Explorer compatibility, it's best to avoid these methods in your code and use `PerformanceObserver` going forward.

User Timing API

The User Timing API is your general purpose measurement API for time-based metrics. It allows you to arbitrarily mark points in time and then later measure the duration between those marks.

```
// Record the time immediately before running a task.
performance.mark('myTask:start');
await doMyTask();
// Record the time immediately after running a task.
performance.mark('myTask:end');

// Measure the delta between the start and end of the task
performance.measure('myTask', 'myTask:start', 'myTask:end');
```

While APIs like `Date.now()` or `performance.now()` give you similar abilities, the benefit of using the User Timing API is it integrates well with performance tooling. For example, Chrome DevTools visualizes User Timing measurements in the Performance panel, and many analytics providers will also automatically track any measurements you make and send the duration data to their analytics back end.

To report User Timing measurements, you can use PerformanceObserver and register to observe entries of type `measure`:

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // Log the entry and all associated details.
      console.log(entry.toJSON());
    }
  });
  // Start listening for `measure` entries to be dispatched.
  po.observe({type: 'measure', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

Long Tasks API

The Long Tasks API is useful for knowing when the browser's main thread is blocked for long enough to affect frame rate or input latency. Currently the API will report any tasks that execute for longer than 50 milliseconds (ms).

Anytime you need to run expensive code (or load and execute large scripts) it's useful to track whether or not that code blocked the main thread. In fact, many higher-level metrics are built on top of the Long Tasks API themselves (such as Time to Interactive (TTI) and Total Blocking Time (TBT)).

To determine when long tasks happen, you can use PerformanceObserver and register to observe entries of type `longtask`:

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // Log the entry and all associated details.
      console.log(entry.toJSON());
    }
  });
  // Start listening for `longtask` entries to be dispatched.
  po.observe({type: 'longtask', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

Element Timing API

The Largest Contentful Paint (LCP) metric is useful for knowing when the largest image or text block was painted to the screen, but in some cases you want to measure the render time of a different element.

For these cases, you can use the Element Timing API. In fact, the Largest Contentful Paint API is actually built on top of the Element Timing API and adds automatic reporting of the largest contentful element, but you can report on additional elements by explicitly adding the `elementtiming`

attribute to them, and registering a PerformanceObserver to observe the element entry type.

```
<img elementtiming="hero-image" />
<p elementtiming="important-paragraph">This is text I care about.</p>
...
<script>
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((entryList) => {
    for (const entry of entryList.getEntries()) {
      // Log the entry and all associated details.
      console.log(entry.toJSON());
    }
  });
  // Start listening for `element` entries to be dispatched.
  po.observe({type: 'element', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
</script>
```

The types of elements considered for Largest Contentful Paint are the same as those observable via the Element Timing API. If you add the `elementtiming` attribute to an element that isn't one of those types, the attribute will be ignored.

Event Timing API

The First Input Delay (FID) metric measures the time from when a user first interacts with a page to the time when the browser is actually able to begin processing event handlers in response to that interaction. However, in some cases it may also be useful to measure the event processing time itself as well as the time until the next frame can be rendered.

This is possible with the Event Timing API (which is used to measure FID) as it exposes a number of timestamps in the event lifecycle, including:

- `startTime`: the time when the browser receives the event.
- `processingStart`: the time when the browser is able to begin processing event handlers for the event.

- `processingEnd`: time when the browser finishes executing all synchronous code initiated from event handlers for this event.
- `duration`: the time (rounded to 8ms for security reasons) between when the browser receives the event until it's able to paint the next frame after finishing executing all synchronous code initiated from the event handlers.

The following example shows how to use these these values to create custom measurements:

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  const po = new PerformanceObserver((entryList) => {
    const firstInput = entryList.getEntries()[0];

    // Measure First Input Delay (FID).
    const firstInputDelay = firstInput.processingStart - firstInput.startTime;

    // Measure the time it takes to run all event handlers
    // Note: this does not include work scheduled asynchronously using
    // methods like `requestAnimationFrame()` or `setTimeout()`.
    const firstInputProcessingTime = firstInput.processingEnd - firstInput.processingStart;

    // Measure the entire duration of the event, from when input is received by
    // the browser until the next frame can be painted after processing all
    // event handlers.
    // Note: similar to above, this value does not include work scheduled
    // asynchronously using `requestAnimationFrame()` or `setTimeout()`.
    // And for security reasons, this value is rounded to the nearest 8ms.
    const firstInputDuration = firstInput.duration;

    // Log these values the console.
    console.log({
      firstInputDelay,
      firstInputProcessingTime,
      firstInputDuration,
    });
  });

  po.observe({type: 'first-input', buffered: true});
} catch (error) {
  // Do nothing if the browser doesn't support this API.
```

```
}
```

Resource Timing API

The Resource Timing API gives developers detailed insight into how resources for a particular page were loaded. Despite the name of the API, the information it provides is not just limited to timing data (though there's plenty of that). Other data you can access includes:

- initiatorType: how the resource was fetched: such as from a `<script>` or `<link>` tag, or from `fetch()`
- nextHopProtocol: the protocol used to fetch the resource, such as `h2` or `quic`
- encodedBodySize/decodedBodySize]: the size of the resource in its encoded or decoded form (respectively)
- transferSize: the size of the resource that was actually transferred over the network. When resources are fulfilled via the cache, this value can be much smaller than the `encodedBodySize`, and in some cases it can be zero (if no cache revalidation is required).

Note, you can use the `transferSize` property of resource timing entries to measure a *cache hit rate* metric or perhaps even a *total cached resource size* metric, which may be useful in understanding how your resource caching strategy affects performance for repeat visitors.

The following example logs all resources requested by the page and indicates whether or not each resource was fulfilled via the cache.

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // If transferSize is 0, the resource was fulfilled via the cache.
      console.log(entry.name, entry.transferSize === 0);
    }
  });
  // Start listening for `resource` entries to be dispatched.
  po.observe({type: 'resource', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

Navigation Timing API

The Navigation Timing API is similar to the Resource Timing API, but it reports only navigation requests. The `navigation` entry type is also similar to the `resource` entry type, but it contains some additional information specific to only navigation requests (such as when the `DOMContentLoaded` and `load` events fire).

One metric many developers track to understand server response time (Time to First Byte) is available via the Navigation Timing API—specifically it's entry's `responseStart` timestamp.

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // If transferSize is 0, the resource was fulfilled via the cache.
      console.log('Time to first byte', entry.responseStart);
    }
  });
  // Start listening for `navigation` entries to be dispatched.
  po.observe({type: 'navigation', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

Another metric developers who use service worker may care about is the service worker startup time for navigation requests. This is the amount of time it takes the browser to start the service worker thread before it can start intercepting fetch events.

The service worker startup time for a particular navigation request can be determined from the delta between `entry.responseStart` and `entry.workerStart`.

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      console.log('Service Worker startup time:',
```

```
          entry.responseStart - entry.workerStart);
    }
  });
  // Start listening for `navigation` entries to be dispatched.
  po.observe({type: 'navigation', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

Server Timing API

The Server Timing API allows you to pass request-specific timing data from your server to the browser via response headers. For example, you can indicate how long it took to lookup data in a database for a particular request—which can be useful in debugging performance issues caused by slowness on the server.

For developers who use third-party analytics providers, the Server Timing API is the only way to correlate server performance data with other business metrics that these analytics tools may be measuring.

To specify server timing data in your responses, you can use the `Server-Timing` response header. Here's an example.

```
HTTP/1.1 200 OK


Server-Timing: miss, db;dur=53, app;dur=47.2
```

Then, from your pages, you can read this data on both `resource` or `navigation` entries from the Resource Timing and Navigation Timing APIs.

```
// Catch errors since some browsers throw when using the new `type` option.
// https://bugs.webkit.org/show_bug.cgi?id=209216
try {
  // Create the performance observer.
  const po = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // Logs all server timing data for this response
      console.log('Server Timing', entry.serverTiming);
    }
  });
  // Start listening for `navigation` entries to be dispatched.
```

```
  po.observe({type: 'navigation', buffered: true});
} catch (e) {
  // Do nothing if the browser doesn't support this API.
}
```

□□□ #3
□ □□□ □□□ 2026-10-31 22:57:31 UTC
□ cocobook-sync □□□ 2026-11-04 17:39:35 UTC