An Introduction to the Performance API

The Performance API measures the responsiveness of your live web application on real user devices and network connections. It can help identify bottlenecks in your client-side and server-side code with:

- **user timing:** Custom measurement of client-side JavaScript function performance
- **paint timing:** Browser rendering metrics
- **resource timing:** Loading performance of assets and Ajax calls
- **navigation timing:** Page loading metrics, including redirects, DNS look-ups, DOM readiness, and more

The API addresses several problems associated with typical performance assessment:

1. Developers often test applications on high-end PCs connected to a fast network. DevTools can emulate slower devices, but it won't always highlight real-world issues when the majority of clients are running a two year-old mobile connected to airport wifi.

2. Third-party options such as Google Analytics are often blocked, leading to skewed results and assumptions. You may also encounter privacy implications in some countries.

3. The Performance API can accurately gauge various metrics better than methods such as Date().

The following sections describe ways you can use the Performance API. Some knowledge of JavaScript and page loading metrics is recommended.

Performance API Availability

Most modern browsers support the Performance API – including IE10 and IE11 (even IE9 has limited support). You can detect the API's presence using:

```
if ('performance' in window) {
  // use Performance API
}
```

It's not possible to fully Polyfill the API, so be wary about missing browsers. If 90% of your users are happily browsing with Internet Explorer 8, you'd only be measuring 10% of clients with more capable applications.

The API can be used in Web Workers, which provide a way to execute complex calculations in a background thread without halting browser operations.

Most API methods can be used in server-side Node.js with the standard perf_hooks module:

```
// Node.js performance
import { performance } from 'node:perf_hooks';
// or in Common JS: const { performance } = require('node:perf_hooks');

console.log( performance.now() );
```

Deno provides the standard Performance API:

```
// Deno performance
console.log( performance.now() );
```

You will need to run scripts with the **--allow-hrtime** permission to enable high-resolution time measurement:

```
deno run --allow-hrtime index.js
```

Server-side performance is usually easier to assess and manage because it's dependent on load, CPUs, RAM, hard disks, and cloud service limits. Hardware upgrades or process management options such as PM2, clustering, and Kubernetes can be more effective than refactoring code.

The following sections concentrate on client-side performance for this reason.

Custom Performance Measurement

The Performance API can be used to time the execution speed of your application functions. You may have used or encountered timing functions using **Date()**:

```
const timeStart = new Date();
runMyCode();
const timeTaken = new Date() - timeStart;

console.log(`runMyCode() executed in ${ timeTaken }ms`);
```

The Performance API offers two primary benefits:

1. **Better accuracy:** Date() measures to the nearest millisecond, but the Performance API can measure fractions of a millisecond (depending on the browser).
2. **Better reliability:** The user or OS can change the system time so Date()-based metrics will not always be accurate. This means your functions could appear particularly slow when clocks move forward!

The Date() equivalent is **performance.now()** which returns a high-resolution timestamp which is set at zero when the process responsible for creating the document starts (the page has loaded):

```
const timeStart = performance.now();
runMyCode();
const timeTaken = performance.now() - timeStart;

console.log(`runMyCode() executed in ${ timeTaken }ms`);
```

A non-standard performance.timeOrigin property can also return a timestamp from 1 January 1970 although this is not available in IE and Deno.

performance.now() becomes impractical when making more than a few measurements. The Performance API provides a buffer where you can record event for later analysis by passing a label name to performance.mark():

```
performance.mark('start:app');
performance.mark('start:init');

init(); // run initialization functions

performance.mark('end:init');
performance.mark('start:funcX');

funcX(); // run another function

performance.mark('end:funcX');
performance.mark('end:app');
```

An array of all mark objects in the Performance buffer can be extracted using:

```
const mark = performance.getEntriesByType('mark');
```

Example result:

```
[
  {
    detail: null
    duration: 0
    entryType: "mark"
    name: "start:app"
    startTime: 1000
  },
  {
    detail: null
    duration: 0
    entryType: "mark"
    name: "start:init"
```

```
    startTime: 1001
  },
  {
    detail: null
    duration: 0
    entryType: "mark"
    name: "end:init"
    startTime: 1100
  },
...
]
```

The **performance.measure()** method calculates the time between two marks and also stores it in the Performance buffer. You pass a new measure name, the starting mark name (or null to measure from the page load), and the ending mark name (or null to measure to the current time):

```
performance.measure('init', 'start:init', 'end:init');
```

A PerformanceMeasure object is appended to the buffer with the calculated time duration. To obtain this value, you can either request an array of all measures:

```
const measure = performance.getEntriesByType('measure');
```

or request a measure by its name:

```
performance.getEntriesByName('init');
```

Example result:

```
[
  {
    detail: null
    duration: 99
    entryType: "measure"
    name: "init"
    startTime: 1001
  }
]
```

Using the Performance Buffer

As well as marks and measures, the Performance buffer is used to automatically record navigation timing, resource timing, and paint timing (which we'll discuss later). You can obtain an array of all entries in the buffer:

```
performance.getEntries();
```

By default, most browsers provide a buffer that stores up to 150 resource metrics. This should be enough for most assessments, but you can increase or decrease the buffer limit if needed:

```
// record 500 metrics
performance.setResourceTimingBufferSize(500);
```

Marks can be cleared by name or you can specify an empty value to clear all marks:

```
performance.clearMarks('start:init');
```

Similarly, measures can be cleared by name or an empty value to clear all:

```
performance.clearMeasures();
```

Monitoring Performance Buffer Updates

A **PerformanceObserver** can monitor changes to the Performance buffer and run a function when specific events occur. The syntax will be familiar if you've used **MutationObserver** to respond to DOM updates or **IntersectionObserver** to detect when elements are scrolled into the viewport.

You must define an observer function with two parameters:

1. an array of observer entries which have been detected, and
2. the observer object. If necessary, its **disconnect()** method can be called to stop the observer.

```
function performanceCallback(list, observer) {
  list.getEntries().forEach(entry => {

    console.log(`name     : ${ entry.name }`);
    console.log(`type     : ${ entry.type }`);
    console.log(`start    : ${ entry.startTime }`);
    console.log(`duration: ${ entry.duration }`);

  });

}
```

The function is passed to a new PerformanceObserver object. Its **observe()** method is passed an array of Performance buffer entryTypes to observe:

```
let observer = new PerformanceObserver( performanceCallback );
observer.observe({ entryTypes: ['mark', 'measure'] });
```

In this example, adding a new mark or measure runs the performanceCallback() function. While it only logs messages here, it could be used to trigger a data upload or make further calculations.

Measuring Paint Performance

The Paint Timing API is only available in client-side JavaScript and automatically records two metrics that are important to Core Web Vitals:

1. **first-paint:** The browser has started to draw the page.
2. **first-contentful-paint:** The browser has painted the first significant item of DOM content, such as a heading or an image.

These can be extracted from the Performance buffer to an array:

```
const paintTimes = performance.getEntriesByType('paint');
```

Be wary about running this before the page has fully loaded; the values will not be ready. Either wait for the window.load event or use a PerformanceObserver to monitor 'paint' entryTypes.

Example result:

```
[
  {
    "name": "first-paint",
    "entryType": "paint",
    "startTime": 812,
    "duration": 0
  },
  {
    "name": "first-contentful-paint",
    "entryType": "paint",
    "startTime": 856,
    "duration": 0
  }
]
```

A slow first-paint is often caused by render-blocking CSS or JavaScript. The gap to the first-contentful-paint could be large if the browser has to download a large image or render complex elements.

Resource Performance Measurement

Network timings for resources such as images, stylesheets, and JavaScript files are automatically recorded to the Performance buffer. While there is little you can do to solve

network speed issues (other than reducing file sizes), it can help highlight issues with larger assets, slow Ajax responses, or badly-performing third-party scripts.

An array of PerformanceResourceTiming metrics can be extracted from the buffer using:

```
const resources = performance.getEntriesByType('resource');
```

Alternatively, you can fetch metrics for an asset by passing its full URL:

```
const resource = performance.getEntriesByName('https://test.com/script.js');
```

Example result:

```
[
  {
    connectEnd: 195,
    connectStart: 195,
    decodedBodySize: 0,
    domainLookupEnd: 195,
    domainLookupStart: 195,
    duration: 2,
    encodedBodySize: 0,
    entryType: "resource",
    fetchStart: 195,
    initiatorType: "script",
    name: "https://test.com/script.js",
    nextHopProtocol: "h3",
    redirectEnd: 0,
    redirectStart: 0,
    requestStart: 195,
    responseEnd: 197,
    responseStart: 197,
    secureConnectionStart: 195,
    serverTiming: [],
    startTime: 195,
    transferSize: 0,
    workerStart: 195
  }
]
```

The following properties can be examined:

- **name**: Resource URL
- **entryType**: "resource"
- **initiatorType**: How the resource was initiated, such as "script" or "link"
- **serverTiming**: An array of PerformanceServerTiming objects passed by the server in the HTTP Server-Timing header (your server-side application could send metrics to the client for further analysis)
- **startTime**: Timestamp when the fetch started

- **nextHopProtocol**: Network protocol used
- **workerStart**: Timestamp before starting a Progressive Web App Service Worker (0 if the request is not intercepted by a Service Worker)
- **redirectStart**: Timestamp when a redirect started
- **redirectEnd**: Timestamp after the last byte of the last redirect response
- **fetchStart**: Timestamp before the resource fetch
- **domainLookupStart**: Timestamp before a DNS lookup
- **domainLookupEnd**: Timestamp after the DNS lookup
- **connectStart**: Timestamp before establishing a server connection
- **connectEnd**: Timestamp after establishing a server connection
- **secureConnectionStart**: Timestamp before the SSL handshake
- **requestStart**: Timestamp before the browser requests the resource
- **responseStart**: Timestamp when the browser receives the first byte of data
- **responseEnd**: Timestamp after receiving the last byte or closing the connection
- **duration**: The difference between startTime and responseEnd
- **transferSize**: The resource size in bytes including the header and compressed body
- **encodedBodySize**: The resource body in bytes before uncompressing
- **decodedBodySize**: The resource body in bytes after uncompressing

This example script retrieves all Ajax requests initiated by the Fetch API and returns the total transfer size and duration:

```
const fetchAll = performance.getEntriesByType('resource')
  .filter( r => r.initiatorType === 'fetch')
  .reduce( (sum, current) => {
    return {
      transferSize: sum.transferSize += current.transferSize,
      duration: sum.duration += current.duration
    }
  },
  { transferSize: 0, duration: 0 }
);
```

Navigation Performance Measurement

Network timings for unloading the previous page and loading the current page are automatically recorded to the Performance buffer as a single PerformanceNavigationTiming object.

Extract it to an array using:

```
const pageTime = performance.getEntriesByType('navigation');
```

…or by passing the page URL to .getEntriesByName():

```
const pageTiming = performance.getEntriesByName(window.location);
```

The metrics are identical to those for resources but also includes page-specific values:

- **entryType**: E.g. "navigation"
- **type**: Either "navigate", "reload", "back_forward," or "prerender"
- **redirectCount**: The number of redirects
- **unloadEventStart**: Timestamp before the unload event of the previous document
- **unloadEventEnd**: Timestamp after the unload event of the previous document
- **domInteractive**: Timestamp when the browser has parsed the HTML and constructed the DOM
- **domContentLoadedEventStart**: Timestamp before document's DOMContentLoaded event fires
- **domContentLoadedEventEnd**: Timestamp after document's DOMContentLoaded event completes
- **domComplete**: Timestamp after DOM construction and DOMContentLoaded events have completed
- **loadEventStart**: Timestamp before the page load event has fired
- **loadEventEnd**: Timestamp after the page load event and all assets are available

Typical issues include:

- A long delay between **unloadEventEnd** and **domInteractive**. This could indicate a slow server response.
- A long delay between **domContentLoadedEventStart** and **domComplete**. This could indicate that page start-up scripts are too slow.
- A long delay between **domComplete** and **loadEventEnd**. This could indicate the page has too many assets or several are taking too long to load.

Performance Recording and Analysis

The Performance API allows you to collate real-world usage data and upload it to a server for further analysis. You *could* use a third-party service such as Google Analytics to store the data, but there's a risk the third-party script could be blocked or introduce new performance problems. Your own solution can be customized to your requirements to ensure monitoring does not impact other functionality.

Be wary of situations in which statistics cannot be determined — perhaps because users are on old browsers, blocking JavaScript, or behind a corporate proxy. Understanding what data is missing can be more fruitful than making assumptions based on incomplete information.

Ideally, your analysis scripts won't negatively impact performance by running complex calculations or uploading large quantities of data. Consider utilizing web workers and minimizing the use of synchronous localStorage calls. It's always possible to batch process raw data later.

Finally, be wary of outliers such as very fast or very slow devices and connections that adversely affect statistics. For example, if nine users load a page in two seconds but the tenth experiences a 60 second download, the average latency comes out to nearly 8

seconds. A more realistic metric is the median figure (2 seconds) or the 90th percentile (9 in every 10 users experience a load time of 2 seconds or less).

Summary

Web performance remains a critical factor for developers. Users expect sites and applications to be responsive on most devices. Search Engine Optimization can also be affected as slower sites are downgraded in Google.

There are plenty of performance monitoring tools out there, but most assess server-side execution speeds or use a limited number of capable clients to judge browser rendering. The Performance API provides a way to collate real user metrics that it would not be possible to calculate any other way.