Performance Observer: Efficient Access to Performance Data

By Marc Cohen
Eng Manager, Web Developer Relations

Progressive Web Apps enable developers to build a new class of applications that deliver reliable, high performance user experiences. But to be sure a web app is achieving its desired performance goals, developers need access to high resolution performance measurement data. The W3C Performance Timeline specification defines such an interface for browsers to provide programmatic access to low level timing data. This opens the door to some interesting use cases:

- offline and custom performance analysis
- third party performance analysis and visualization tools
- performance assessment integrated into IDEs and other developer tools

Access to this kind of timing data is already available in most major browsers for navigation timing, resource timing, and user timing. The newest addition is the performance observer interface, which is essentially a streaming interface to gather low level timing information asynchronously, as it's gathered by the browser. This new interface provides a number of critical advantages over previous methods to access the timeline:

- Today, apps have to periodically poll and diff the stored measurements, which is costly. This interface offers them a callback. (In other words, there is no need to poll). As a result apps using this API can be more responsive and more efficient.
- It’s not subject to buffer limits (most buffers are set to 150 items by default), and avoids race conditions between different consumers that may want to modify the buffer.
- Performance observer notifications are delivered asynchronously and the browser can dispatch them during idle time to avoid competing with critical rendering work.

Beginning in Chrome 52, the performance observer interface is enabled by default. Let’s take a look at how to use it.

```
<html>
<head>
```

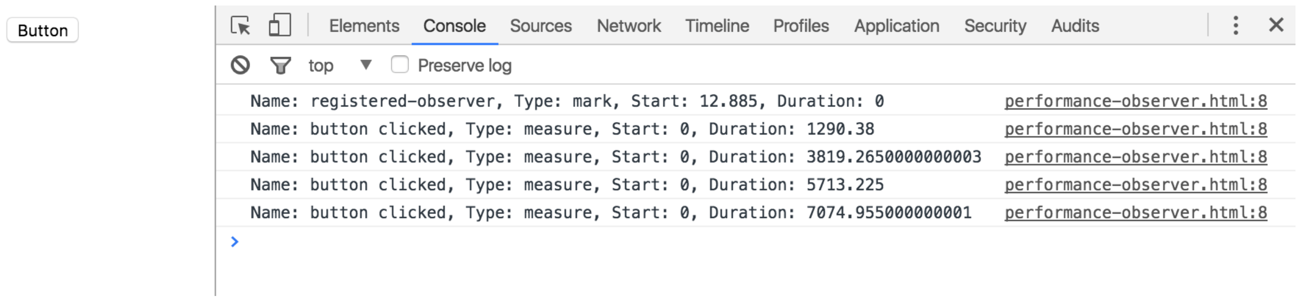
```
  <script>
    var observer = new PerformanceObserver(list => {
      list.getEntries().forEach(entry => {
        // Display each reported measurement on console
        if (console) {
          console.log("Name: "       + entry.name      +
                      ", Type: "     + entry.entryType +
                      ", Start: "    + entry.startTime +
                      ", Duration: " + entry.duration  + "\n");
        }
      })
    });
    observer.observe({entryTypes: ['resource', 'mark', 'measure']});
    performance.mark('registered-observer');

    function clicked(elem) {
      performance.measure('button clicked');
    }
  </script>
</head>
<body>
  <button onclick="clicked(this)">Measure</button>
</body>
</html>
```

This simple page starts with a script tag defining some JavaScript code:

- We instantiate a new `PerformanceObserver` object and pass an event handler function to the object constructor. The constructor initializes the object such that our handler will be called every time a new set of measurement data is ready to be processed (with the measurement data passed as a list of objects). The handler is defined here as an anonymous function that simply displays the formatted measurement data on the console. In a real world scenario, this data might be stored in the cloud for subsequent analysis, or piped into an interactive visualization tool.
- We register for the types of timing events we're interested in via the `observe()` method and call the `mark()` method to mark the instant at which we registered, which we'll consider the beginning of our timing intervals.
- We define a click handler for a button defined in the page body. This click handler calls the `measure()` method to capture timing data about when the button was clicked.

In the body of the page, we define a button, assign our click handler to the `onclick` event, and we're ready to go.

Now, if we load the page and open the Chrome DevTools panel to watch the JavaScript console, every time we click the button a performance measurement is taken. *And because* *we've registered to observe such measurements, they are forwarded to our event handler, asynchronously without the need to poll the timeline*, which displays the measurements on the console as they occur:



```
Name: registered-observer, Type: mark, Start: 12.885, Duration: 0                performance-observer.html:8
Name: button clicked, Type: measure, Start: 0, Duration: 1290.38                 performance-observer.html:8
Name: button clicked, Type: measure, Start: 0, Duration: 3819.2650000000003      performance-observer.html:8
Name: button clicked, Type: measure, Start: 0, Duration: 5713.225                performance-observer.html:8
Name: button clicked, Type: measure, Start: 0, Duration: 7074.955000000001       performance-observer.html:8
```

The `start` value represents the starting timestamp for events of type `mark` (of which this app has only one). Events with type `measure` have no inherent starting time; they represent timing measurements taken relative to the last `mark` event. Thus, the duration values seen here represent the elapsed time between the call to `mark()`, which serves as a common interval starting point, and multiple subsequent calls to `measure()`.

As you can see, this API is quite simple and it offers the ability to gather filtered, high resolution, real time performance data without polling, which should open the door to more efficient performance tooling for web apps.
