Custom Metrics

There are times when it's good to have a sense of how long your code takes to execute. Maybe you're troubleshooting performance or you're conceerned that parts of your code are not as fast as they could be.

In the beginning you could do something like this:

```
function myHeavyTask() {
  for (let i = 1; i < 11; i++) {
    console.log(i, i*2, i*3, i*Math.sqrt(2), i*Math.sqrt(3));
  }
};

let startDate = new Date().getTime();
myHeavyTask();
let endDate = new Date().getTime();
let duration = (endDate - startDate);
console.log(duration / 1000 + ' milliseconds');
```

But it's cumbersome and error prone.

The User Timing API allows for custom logging of activity in your page. It also provides a series of predefined events that we can measure against.

The basic tools that we get from the User Timing API provides two basic tools we can use to measure performance

- `performance.mark` gives us a way to indicate when an event has started or finished
- `performance.measure` creates a result between two given marks

Different APIs and specifications give you different performance events that you can capture and use as starting points for measurements.

- `domLoading` fires the browser is about to start parsing the first received bytes of the HTML document
- `domInteractive` triggers when the browser has finished parsing all of the HTML and DOM construction is complete

- `domContentLoaded` marks the point when both the DOM is ready and there are no stylesheets that are blocking JavaScript execution - we may be able construct the render tree.
    - Many JavaScript frameworks wait for this event before they start executing their own logic. For this reason the browser captures the EventStart and EventEnd timestamps to allow us to track how long this execution took.
- `domComplete` indicates all of the processing is complete and all of the resources on the page (images, etc.) have finished downloading
- `loadEvent` as the final step in every page load the browser fires an onload event that can trigger additional application logic

The HTML specification dictates specific conditions for each and every event: when it should be fired, which conditions should be met, and so on. We'll focus on a few key milestones related to the critical rendering path:

- `domInteractive` marks when DOM is ready
- `domContentLoaded` typically marks when both the DOM and CSSOM are done loading
    - If there is no parser blocking JavaScript then DOMContentLoaded will fire immediately after domInteractive
- `domComplete` marks when the page and all of its subresources are ready.

The image below illustrates some of the events available and at what point in the load process they happen.

Figure 1:

Different events triggered

Taken from Measuring the Critical Rendering Path

so we can use some of these measurements to create more descriptive names for the events that happen during the page rendering process.

The example below shows how we can measure some events and log the results to the console. It performs the following tasks

1. Set a variable to hold windows.performance.timing
2. Measure from when content begins to load to DomInteractive
3. Measure from when content begins to load to when the domContentLoaded event start
4. Measure from when content begins to load to when it's complete
5. Log the results to console

```
let t = window.performance.timing; // 1
let interactive = t.domInteractive - t.domLoading; // 2
let dcl = t.domContentLoadedEventStart - t.domLoading; // 3
let complete = t.domComplete - t.domLoading; // 4
console.log('interactive: ' + interactive / 1000 +  ' seconds'); 'interac
console.lo' + dcl / 1000 + '// 5ete / 1000 + ' seconds'); // 5
' seconds'// 5
```

Furthermore we can create custom marks to measure our code's performance. This is more complex than using the pre-defined benchmarks.

We'll use the following function as the starting point for explaining performance.

```
function myFunction(val) {
  for(i = 0; i < val; i++) {};
  console.log('done');
}
```

We wrap our function executions in two `performance.mark` with a name. The name is important for the next step. In this case we've named them startTime and endTime.

```
performance.mark('startTime1');
myFunction(10000);
performance.mark('endTime1');

performance.mark('startTime2');
myFunction(100000);
performance.mark('endTime2');
```

The next step is to create a measure using `performance.measure`to display the duration of a performance event.

This method takes three arguments:

- The name of the measure
- The beginning mark name
- The ending mark name

Using the marks we defined in the previous section I've created two measures; one for each set of marks.

```
performance.measure('duration1', 'startTime1', 'endTime1');
performance.measure('duration2', 'startTime2', 'endTime2');
```

So far we've worried about setting up the performance instrumentation for our scripts, now we can look at how to report it using `performance.getEntriesByType`.

The first example finds all the performance entries of type **mark**, loops through them and logs their start time.

The second example does the same thing for all measures (performance entries of type **measure**).

These are trivial examples with few entries. For large codebases this becomes significantly more useful.

We can also get entries by name, which would allow us to identify individual marks or measures to do further analysis of the application's performance.

```
// Grab all the mark entries
perfMarks = performance.getEntriesByType('mark');
'mark'// For each mark log name and duration to console
for (i = 0; i < perfMarks.length; i++) {
  console.log(`Name: ${perfMarks[i].name}
  Start Time: ${perfMarks[i].startTime}`)
}
```

```
`Name: ${perfMarks[i].name}
  Start Time: ${perfMarks[i].startTime}`// Grab all the measure entries
perfMeasures = performance.getEntriesByType('measure');
// For each measure log name and duration to console
for (i = 0; i < perfMeasures.length; i++) {
  console.log(`
    Name:  ${perfMeasures[i].name}
    Duration: ${perfMeasures[i].duration}`);
}
```

Finally, we need to cleanup all the marks and measures we create. This is good practice to make sure that we don't get stale data for our measurements.

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

Links and Resources

- Measuring the Critical Rendering Path
- User Timing and Custom Metrics
- User Timing Marks and Measures (Lighthouse)
- Using the User Timing API
- Discovering the User Timing API
- User Timing API
- Specs
    - Navigation Timing Recommendation
    - User Timing Level 2 (editor's draft)