Philip Tellis
@bluesmoon

Nic Jansma
@nicj

https://github.com/SOASTA/boomerang
http://www.soasta.com/mpulse
http://slideshare.net/nicjansma/measuring-real-user-performance-in-the-browser
Measuring Real User Performance in the Browser

2016-09-20

#VELOCITYCONF NY 2016

http://slideshare.net/nicjansma/measuring-real-user-performance-in-the-browser
Agenda

- Real User Measurement
- Browser Performance APIs
- Visual Experience
- Beaconing
- Single Page Apps
- Continuity
- Nixing Noise
Abbé Jean-Antoine Nollet

1700 - 1770

French Clergyman & Budding Electrician

Invented one of the first Electroscopes

(we now call them beacon collectors)
In 1746, he conducted the first ever RUM Experiment.

He shot an electric current through 200 monks, and checked how quickly they jumped; thereby measuring the latency of an electric signal with...
Fortunately, our methods have gotten far less intrusive...
but first...

Why do we care?
Delight

Or

Frustrate
Experience

Responsiveness

Smoothness

Cognitive Dissonance/Resonance

Delight

Frustration
ok then...

How do we do it?
Performance aware Browser APIs

- **High Resolution Time**: Better `Date.now()`
- **Navigation Timing** (NT): Page load timings
- **Performance Timeline**: Access NT/RT/UT from one API
- **Resource Timing** (RT): Resource load timings
- **User Timing** (UT): Custom site events and measurements
- **Page Visibility**: Visibility state of the document
- **Timing control for script-based animations**: `requestAnimationFrame()`
- **Efficient Script Yielding**: `setImmediate()`
- **Resource Hints**: `dns-prefetch`, `preconnect`, `prefetch`, `prerender`
- **Preload**: Mandatory high-priority fetch for current navigation
- **Cooperative Scheduling of Background Tasks**: `requestIdleCallback()`
- **Beacon**: `sendBeacon()`
DOMHighResTimeStamp

High-resolution, monotonically non-decreasing clock

<table>
<thead>
<tr>
<th></th>
<th>Date</th>
<th>DOMHighResTimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessed via</td>
<td>(new Date).getTime()</td>
<td>performance.now()</td>
</tr>
<tr>
<td>Start</td>
<td>Unix epoch</td>
<td>navigationStart</td>
</tr>
<tr>
<td>Monotonically non-decreasing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Affected by user’s clock</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Resolution</td>
<td>Millisecond</td>
<td>Sub-millisecond</td>
</tr>
<tr>
<td>Example</td>
<td>1420147524606</td>
<td>3392.275999998674</td>
</tr>
</tbody>
</table>

https://w3c.github.io/hr-time/
DOMHighResTimeStamp

Monotonically non-decreasing

<table>
<thead>
<tr>
<th>@ Time</th>
<th>Date.getTime()</th>
<th>performance.now()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation Start</td>
<td>1420147524606</td>
<td>0</td>
</tr>
<tr>
<td>@ 100ms</td>
<td>1420147524706</td>
<td>100.0</td>
</tr>
<tr>
<td>@ 200ms</td>
<td>1420147524806</td>
<td>200.0</td>
</tr>
<tr>
<td>@ 300ms + user's clock moves back 1s</td>
<td>1420147523906</td>
<td>300.0</td>
</tr>
<tr>
<td>@ 400ms</td>
<td>1420147524006</td>
<td>400.0</td>
</tr>
</tbody>
</table>
DOMHighResTimeStamp: Usage

```javascript
var myTime = performance.now();

// 8141.84 -> 8.1 seconds after page load
```
Navigation Timing

Exposes accurate performance metrics describing your visitor's page load

http://www.w3.org/TR/navigation-timing
Navigation Timing: Timestamps

window.performance.timing

Navigation timestamps:

```
window.performance.timing

connectEnd: 1473630941946
connectStart: 1473630941946
domComplete: 1473630946441
domContentLoadedEventEnd: 1473630945308
domContentLoadedEventStart: 1473630945294
domInteractive: 1473630945294
domLoading: 1473630943828
domainLookupEnd: 1473630941946
domainLookupStart: 1473630941946
fetchStart: 1473630941946
loadEventEnd: 1473630946441
loadEventStart: 1473630946441
navigationStart: 1473630941871
redirectEnd: 1473630941946
redirectStart: 1473630941871
requestStart: 1473630941951
responseEnd: 1473630943953
responseStart: 1473630943807
secureConnectionStart: 0
unloadEventEnd: 1473630943813
unloadEventStart: 1473630943811
```
NavigationTiming: Characteristics

window.performance.navigation

Characteristics of the browser navigation

window.performance.navigation.type:
- navigate = 0
- reload = 1
- back / forward = 2
function onLoad() {
  if ('performance' in window && 'timing' in window.performance) {
    setTimeout(function() {
      var t = window.performance.timing;
      var ntData = {
        redirect: t.redirectEnd - t.redirectStart,
        dns: t.domainLookupEnd - t.domainLookupStart,
        connect: t.connectEnd - t.connectStart,
        ssl: t.secureConnectionStart ? (t.connectEnd - t.secureConnectionStart) : 0,
        request: t.responseStart - t.requestStart,
        response: t.responseEnd - t.responseStart,
        dom: t.loadEventStart - t.responseEnd,
        total: t.loadEventEnd - t.navigationStart
      };
      }, 0);
  }
}
Navigation Timing API

API for accessing timing information related to navigation and elements.

- **Global**: 90.73%
  - unprefixed: 90.65%
- **U.S.A.**: 95.63%
  - unprefixed: 95.58%

<table>
<thead>
<tr>
<th>Browser</th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>13</td>
<td>47</td>
<td>51</td>
<td>9.1</td>
<td>39</td>
<td>9.3</td>
<td>all</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>14</td>
<td>48</td>
<td>52</td>
<td>10</td>
<td>40</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: Removed in iOS 8.1 due to poor performance.
Navigation Timing: Integrations

DIY:
• Send this data to your backend for logging
• Show any page’s timings via a bookmarklet: kaaes.github.io/timing
• Boomerang: github.com/SOASTA/boomerang
• Boomcatch: cruft.io/posts/introducing-boomcatch
• BoomerangExpress: github.com/andreas-marschke/boomerang-express
• SiteSpeed.io: sitespeed.io
• Piwik: github.com/piwik/piwik

Commercial:
• SOASTA mPulse, Google Analytics Site Speed, New Relic Browser, NeuStar WPM, SpeedCurve, etc...
NavigationTiming: Tips

- Use `fetchStart` instead of `navigationStart` unless you're interested in redirects, tab initialization time, etc.
- `loadEventEnd` will be 0 until after the body's load event has finished (so you can't measure it in the load event)
- We don't have an accurate way to measure the "request time", as "requestEnd" is invisible to us (the server sees it)
- **Home page** scenarios: Timestamps up through `responseEnd` event may be 0 duration because some browsers speculatively pre-fetch home pages (and don't report the correct timings)
- If possible, do any beaconing of the data as soon as possible. Browser `onbeforeunload` isn't 100% reliable for sending data
- Not suitable for Single-Page Apps (we’ll cover this later)
Coming Soon to Navigation Timing

- Part of the **Performance Timeline**: `performance.getEntries("navigation")`
- Support for **DOMHighResTimeStamp**
- Timing information for **prerender**
- **Protocol** information: `nextHopProtocol`
- **Transfer, encoded body and decoded body** sizes
Navigation Timing: chrome.loadTimes()

{  
    "requestTime" : 1473093945.032975,  
    "startLoadTime" : 1473093945.129178,  
    "commitLoadTime" : 1473093945.575271,  
    "finishDocumentLoadTime" : 1473093946.872513,  
    "finishLoadTime" : 1473093952.281069,  
    "firstPaintTime" : 1473093945.96769,  
    "firstPaintAfterLoadTime" : 1473093952.316622,  
    "navigationType" : "BackForward",  
    "wasFetchedViaSpdy" : true,  
    "wasNpnNegotiated" : true,  
    "npnNegotiatedProtocol" : "quic/1+spdy/3",  
    "wasAlternateProtocolAvailable" : false,  
    "connectionInfo" : "quic/1+spdy/3"  
}  

what did we do before Navigation Timing?
Prior to NavigationTiming

1. Hook into the `beforeUnload`, `unload` and `pagehide` events to set a cookie with the `timestamp` and `url`.

2. In the `onload` or `pageshow` event, check if the cookie is set and if the `url` in the cookie matches `document.referrer`.

3. If we have a match, calculate the time delta:
   - `beforeUnload` corresponds to `navigationStart`.
   - `unload` and `pagehide` correspond to `responseStart`.
   - We also hook into clicks and form submits just in case the user goes off to a new tab.

*Note:* This doesn't help for external referrers!
Surprisingly, this works all the way back to IE5.5
(insofar as we've tested)
Resource Timing

https://www.w3.org/TR/resource-timing/

Exposes sub-resource performance metrics
ResourceTiming: Inspiration
ResourceTiming: History

How it was done in the old days:

```javascript
var start = new Date().getTime();
var image1 = new Image();
var resourceTiming = function() {
    var now = new Date().getTime();
    var latency = now - start;
    alert("End to end resource fetch: " + latency);
};

image1.onload = resourceTiming;
image1.src = 'http://www.w3.org/Icons/w3c_main.png';
```

(not practical for all types of content -- or a regular HTML website)
Performance Timeline

Unifying interface to access and retrieve performance metrics

window.performance:

- `getEntries()`: Gets all entries in the timeline
- `getEntriesByType(type)`: Gets all entries of the specified type (eg resource, mark, measure)
- `getEntriesByName(name)`: Gets all entries with the specified name (eg URL or mark name)
Performance Timeline: Usage

```javascript
> performance.getEntriesByType("resource")
<string>
[10 entries]
0: PerformanceResourceTiming
1: PerformanceResourceTiming
2: PerformanceResourceTiming
3: PerformanceResourceTiming
4: PerformanceResourceTiming
5: PerformanceResourceTiming
6: PerformanceResourceTiming
7: PerformanceResourceTiming
8: PerformanceResourceTiming
9: PerformanceResourceTiming
10: PerformanceResourceTiming
```
Resource Timing:

```
performance.getEntriesByType("resource")[0]
{
    name : "http://www.foo.com/foo.png",
    initiatorType : "img",
    entryType : "resource",
    startTime : 566.357000003336,
    workerStart : 0,
    redirectStart : 0,
    redirectEnd : 0,
    fetchStart : 566.357000003336,
    domainLookupStart : 566.357000003336,
    domainLookupEnd : 566.357000003336,
    connectStart : 566.357000003336,
    secureConnectionStart : 0,
    connectEnd : 566.357000003336,
    requestStart : 568.4959999925923,
    responseStart : 569.4220000004862,
    responseEnd : 570.6329999957234,
    duration : 4.275999992387369
}
```
ResourceTiming: InitiatorType

- img
- link
- script
- css: url(), @import
- xmlhttprequest
- image (SVG)
- object (Flash)

localName of the element
ResourceTiming: Buffer

- There is a ResourceTiming buffer (per IFRAME) that stops filling after its size limit is reached (default: 150 entries)
- Listen for the `onResourceTimingBufferFull` event
- `setResourceTimingBufferSize(n)` and `clearResourceTimings()` can be used to modify it
- **Do NOT** `setResourceTimingBufferSize(99999999)` as this can lead to browser memory growing unbound
ResourceTiming: Compressing

- Each resource is ~500 bytes `JSON.stringify()`d
- HTTP Archive tells us there's 103 HTTP resources on average, per page, with an average URL length of 85 bytes
- That means you could expect around 45 KB of ResourceTiming data per page load
- For comparison, the default TCP Window size allows 15 KB to go through before requiring an ACK, so do the math.
- Compress it: nicj.net/compressing-resourcetiming
ResourceTiming: Compressing

```javascript
{
  connectEnd: 566.357000003336,
  connectStart: 566.357000003336,
  domainLookupEnd: 566.357000003336,
  domainLookupStart: 566.357000003336,
  duration: 4.275999992387369,
  entryType: "resource",
  fetchStart: 568.357000003336,
  initiatorType: "img",
  name: "http://www.foo.com/foo.png",
  redirectEnd: 0,
  redirectStart: 0,
  requestStart: 568.495999925923,
  responseEnd: 570.632999957234,
  responseStart: 569.4220000004862,
  secureConnectionStart: 0,
  startTime: 566.357000003336,
  workerStart: 0
}

```

Compresses ResourceTiming data down to 15% of original size

https://github.com/nicjansma/resourcetiming-compression.js
ResourceTiming: Timing-Allow-Origin

• By default to protect the user’s privacy, cross-origin resources expose timestamps for only the `fetchStart` and `responseEnd` attributes.

• If you have a CDN, or multiple domains, you can allow access to this data from your primary domain.

• Use the TAO:
  `Timing-Allow-Origin: origin-list-or-null OR *`

• Note: Third-party libraries (ads, analytics, etc) must set this on their servers. 5% do according to HTTP Archive. Google, Facebook, Disqus, mPulse, etc.

What are the others afraid of?
Resource Timing: **Timing-Allow-Origin**

// PHP

```php
<?php
    header('Timing-Allow-Origin: *');
?
```

// Apache .htaccess

```html
<IfModule mod_headers.c>
    Header set Timing-Allow-Origin "*"
</IfModule>
```

// JBoss

```java
protected void doPost(HttpServletRequest req,
                        HttpServletResponse res) {
    res.setHeader("Timing-Allow-Origin", "*");
}
```

// nginx

```nginx
location / {
    add_header 'Timing-Allow-Origin' '*';
}
```

And we can get more creative if we only want to allow specific Origins
Resource Timing: Blocking Time

- Browsers open a **limited number of connections** to each unique origin
- If there are more resources than this number, later resources "block"
- Resource Timing duration includes Blocking time!
- So, **don't use duration**... but this is all you get with cross-origin resources.

```javascript
var waitTime = 0;
if (res.connectEnd && res.connectEnd === res.fetchStart) {
  waitTime = res.requestStart - res.connectEnd;
}
else if (res.domainLookupStart) {
  waitTime = res.domainLookupStart - res.fetchStart;
}
```
ResourceTiming: Cache Hits

- Cached resources will show up along side resources that were fetched from the network
- Due to privacy concerns, no direct indication a resource was fetched from the cache
- In practice, resources with a very short duration are likely cache hits
  - 0 - 2ms → In memory cache
  - 2 - 10ms → Disk cache
  - 10 - 40ms → Cached by Edge Proxy
Resource Timing: Integrations

**DIY:**
- Compress + **send this data** to your backend for logging
- Show any page's resources via a **bookmarklet**: [github.com/andydavies/waterfall](https://github.com/andydavies/waterfall)
- **Heatmap** bookmarklet / Chrome extension: [github.com/zeman/perfmap](https://github.com/zeman/perfmap)
- Nurun's **Performance Bookmarklet**: [github.com/nurun/performance-bookmarklet](https://github.com/nurun/performance-bookmarklet)
- **Boomerang** supports ResourceTiming: [github.com/SOASTA/boomerang](https://github.com/SOASTA/boomerang)

**Pay:**
- SOASTA mPulse, New Relic Browser, SpeedCurve, etc.
Resource Timing: Browser Support

Method to help web developers to collect complete timing information related to resources on a document.

<table>
<thead>
<tr>
<th>Browser</th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>13</td>
<td>47</td>
<td>51</td>
<td>9.1</td>
<td>39</td>
<td>9.2</td>
<td>all</td>
<td>4.4</td>
<td>4.4.4</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>14</td>
<td>48</td>
<td>52</td>
<td>10</td>
<td>40</td>
<td>41</td>
<td></td>
<td>51</td>
<td>51</td>
</tr>
</tbody>
</table>

Notes: Can be enabled in Firefox using the `dom.enable_resource_timing` flag.
ResourceTiming isn't yet available on iOS, but you can *polyfill* it using `MutationObserver`:

1. Start a `MutationObserver` listening for new nodes with a `src` or `href`
2. Add `load` & `error` event listeners & a `timeout` to deal with cached resources
3. Once the `load` (or `error`) event has fired, you have the *total load time* for the resource (keep in mind that an error event might also fire on Network Error)

In addition, you'll want to instrument `XMLHttpRequest` (which won't be captured by `MutationObserver`):

1. *Proxy* the `XMLHttpRequest` object
2. Hook into `.open()` and `.send()` and add `onreadystatechange` listeners


**Note**: This doesn't give you detailed information such as DNS & TCP timings
ResourceTiming: Tips

- Ensure your CDNs and third-party libraries send Timing-Allow-Origin
- What isn't included in ResourceTiming:
  - The root HTML page (get this from window.performance.timing)
  - HTTP code (privacy concerns)
  - Content that loaded with errors (eg 404s) (browser inconsistencies)
- If you're going to be managing the ResourceTiming buffer, make sure no other scripts are managing it as well
- Each IFRAME will have its own ResourceTiming data, and those resources won't be included in the parent FRAME/document. So you'll need to traverse the document frames to get all resources
- about:blank, javascript: URLs will be seen in RT data
- You may see browser extensions fetching resources in RT data
ResourceTiming²: Coming Soon

Available in recent Firefox, Chrome:

- **nextHopProtocol**: ALPN Protocol ID (e.g. quic+http2)
- **transferSize**: Bytes transferred for HTTP header and response
- **decodedBodySize**: Size of the body after removing any applied content-codings
- **encodedBodySize**: Size of the body after prior to removing any applied content-codings
User Timing

Measuring in-page scripts and other things that don’t fire events
UserTiming

https://www.w3.org/TR/user-timing/

Standardized interface to note timestamps ("marks") and durations ("measures")
UserTiming: History

How it was done before:

```javascript
var start = new Date().getTime();
// do stuff
var now = new Date().getTime();
var duration = now - start;
```

UserTiming is a **better** way of doing this!
UserTiming: Marks & Measures

- **Mark**: A timestamp
- **Measure**: The delta between two timestamps
UserTiming: Usage

Creating:
- `window.performance.mark(name)`
- `window.performance.measure(name, [start], [end])`

Clearing:
- `window.performance.clearMarks([name])`
- `window.performance.clearMeasures([name])`

Querying:
- `window.performance.getEntriesByType("mark")`
- `window.performance.getEntriesByType("measure")`
UserTiming: Mark

// mark
performance.mark("start");
performance.mark("end");

performance.mark("another");
performance.mark("another");
performance.mark("another");

// retrieve
performance.getEntriesByType("mark");

[  
  
  
  
  
  
  
]
UserTiming: Measure

```javascript
// measure
performance.mark("start");

// do work
performance.mark("start2");

// measure from "now" to the "start" mark
performance.measure("time to do stuff", "start");

// measure from "start2" to the "start" mark
performance.measure("time from start to start2", "start", "start2");

// retrieval - specific
performance.getEntriesByName("time from start to start2", "measure");

[{
  "duration":4809.890999997151,
  "startTime":145287.66500000347,
  "entryType":"measure",
  "name":"time from start to start2"
}]
```
UserTiming: Benefits

- Uses the **PerformanceTimeline**, so marks and measures are in the PerformanceTimeline along with other events.
- Uses **DOMHighResTimestamp** instead of **Date** so timestamps are sub-millisecond, monotonically non-decreasing, etc.
- **Browsers** and **third-party** tools can find your performance events easily.
UserTiming: Polyfill

- It's easy to add a Polyfill that adds UserTiming support to browsers that do not natively support it.
- UserTiming is accessed via the PerformanceTimeline, and requires window.performance.now() support, so UserTiming.js adds a limited version of these interfaces if the browser does not support them.
- github.com/nicjansma/usertiming.js
UserTiming: Compressing

Compresses `performance.getEntriesByName("mark"):

```javascript
[{
  "duration": 0,
  "entryType": "mark",
  "name": "mark1",
  "startTime": 100.0,
},
{
  "duration": 0,
  "entryType": "mark",
  "name": "mark2",
  "startTime": 150.0,
},
{
  "duration": 0,
  "entryType": "mark",
  "name": "mark3",
  "startTime": 500.0,
},
{
  "duration": 0,
  "entryType": "mark",
  "name": "measure1",
  "startTime": 100.0,
},
{
  "duration": 100,
  "entryType": "mark",
  "name": "measure2",
  "startTime": 150.0,
},
{
  "duration": 200,
  "entryType": "mark",
  "name": "measure3",
  "startTime": 500.0,
}]
```

Down to something more reasonable:

```
~(m~(ark~(1~'2s~2~'5k~3~'8c)~easure~(1~'2s_2s~2~'5k_5k~3~'8c_8c)))
```

[nicj.net/compressing-usertiming/](nicj.net/compressing-usertiming/)

[github.com/nicjansma/usertiming-compression.js](github.com/nicjansma/usertiming-compression.js)
User Timing API

Method to help web developers measure the performance of their applications by giving them access to high precision timestamps.
UserTiming: Dev Tools

UI Responsiveness

Diagnostic session: 14.954 seconds (3.82 seconds selected)

CPU utilization (%)
- Loading
- Scripting
- GC
- Styling
- Rendering
- Image decoding

Visual throughput (FPS)
- Frames per second

Timeline details
Sort by: Start time
Filter events

Event name
- User measure (box cycler)
- DOM event (click)
- Layout
- Paint
- Style calculation
- Timer (colorCycle)
- Paint
- Layout
- Timer (colorCycle)
- Paint
- Layout

User measure
- Duration (inclusive): 3.82 s
- Duration (exclusive): 3.82 s
- Start time: 1.54 s
- Name: box cycler
- Start mark: Begin Rotation
- End mark: End Rotation

An app-specific scenario was measured using the performance.measure() method.
UserTiming: Dev Tools
Other Useful APIs

Visibility and Painting
Page Visibility

Lets you know when a webpage is *visible* or *in focus*.

document.visibilityState:

- **hidden**
  - Browser is minimized
  - Background tab
  - About to unload or traverse session history
  - OS lock screen
- **visible**
- **prerender**
  - Being speculatively pre-rendered
  - Important for analytics!
- **unloaded**
Page Visibility: Usage

// query the current state
var state = document.visibilityState;

// listen for state change events
document.addEventListener("visibilitychange", function() {
    if (document.visibilityState === "hidden") {
        // stop doing something
    } else if (document.visibilityState === "hidden") {
        // restart doing something
    }
});
# Page Visibility: Browser Support

**Page Visibility**

JavaScript API for determining whether a document is visible on the display.

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>13</td>
<td>47</td>
<td>51</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.4</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>14</td>
<td>48</td>
<td>52</td>
<td>9.1</td>
<td>39</td>
<td></td>
<td>9.3</td>
<td>all</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>53</td>
<td>50</td>
<td>54</td>
<td>10</td>
<td>40</td>
<td></td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>51</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

- Known issues (1)
- Resources (8)
- Feedback

**No notes**
requestAnimationFrame

Tells the browser you wish to run a function prior to the next repaint:

```javascript
var last = performance.now();

function raf(timestamp) {
  var now = performance.now();
  var diff = last - now;

  // update the UI based on the difference in time
  last = now;
  requestAnimationFrame(raf);
}

requestAnimationFrame(raf);

More examples when we talk about measuring continuity.
```
**requestAnimationFrame: Browser Support**

API allowing a more efficient way of running script-based animation, compared to traditional methods using timeouts. Also covers support for cancelAnimationFrame.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Global Support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90.81%</td>
</tr>
<tr>
<td></td>
<td>90.32%</td>
</tr>
<tr>
<td></td>
<td>96.58%</td>
</tr>
<tr>
<td></td>
<td>96.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Browser</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.4</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.4</td>
<td>51</td>
</tr>
</tbody>
</table>

**Notes:**

1. Partial support refers to lacking cancelAnimationFrame support.
2. Supports webkitCancelAnimationFrame rather than `webkitCancelAnimationFrame`. 
Page Load Milestones

- **First Byte**
  - First *content* was received from the server
  - $= \text{responseStart}$
- **onload**
  - Once all content *statically* included or injected before onload has been fetched
  - $= \text{loadEventStart}$
- **Fully Loaded**
  - Once all *static & dynamic* content has been fetched
  - No browser event!
Visual Experience

When does the user **feel like** they can **use** the app?

Network timings != visual experience
Visual Experience

**Milestones:**
- First Paint
- First *Contentful* Paint
- First *Meaningful* Paint
- Visually Complete

**Metrics:**
- Visual Progress
- Speed Index
Visual Experience: First Paint

What was the first thing the user saw?
Visual Experience: First Paint

What was the first thing the user saw?
Visual Experience: First Paint

• Not an industry standard metric!
• The first paint of the browser might show zero content (all white)

// IE 9+ only
window.performance.timing.msFirstPaint
// -> 1473640901

// Chrome only
window.chrome.loadTimes().firstPaintTime;
// -> 1473640917.063874
Visual Experience: First Contentful Paint

- First time a "contentful" thing is painted:
  - text
  - image
  - canvas
  - SVG

- Could still be just a minor page element
  - e.g. just a navigation bar
Visual Experience: First Meaningful Paint

• Page’s primary content appears on screen
• Primary content differs for each page
• Definition still being developed, but this could be a heuristic, guided by hints from developers
Visual Experience: Visually Complete

- All content has been **displayed** on the screen
- Might be hard to measure with **animations**, ads, etc
- Not the same as **onload**! Content can load after **onload**.
Visual Experience

• Besides First Paint, none of these are available in browsers today:
  • First Contentful Paint
  • First Meaningful Paint
  • Visually Complete
• Currently being developed into industry-standard definitions
• Also options:
  • First Non-White (non-background) Paint
  • First Non-Blank Paint
Progressive Web Metrics

Via Paul Irish: github.com/paulirish/pwmetrics
**Visual Progress**

**Percentage** of screen drawn over time (relative to last frame)
Speed Index

- **Average time** at which visible parts of the page are displayed
- Expressed in **milliseconds**
- Area above the curve
- **Lower the better**

\[
\text{Speed Index} = \int_{0}^{\text{end}} 1 - \frac{VC}{100}
\]

end = end time in milliseconds
VC = % visually complete

\[\text{SpeedIndex} = 11581\]
Speed Index

Downsides:

• Not very well understood
• Can be hard to describe (even to techies! let alone marketing)
• Can only be captured accurately in a lab (synthetic testing)
RUM Speed Index

Calculate Speed Index measurements from the field using Resource Timings

- Depends on ResourceTiming support
- Still being developed
  - Needs better support for IFRAMES, SVGs, etc

github.com/WPO-Foundation/RUM-SpeedIndex
[ 30 minute break ]

http://slideshare.net/nicjansma/measuring-real-user-performance-in-the-browser
Single Page Apps
Single Page Apps (SPAs)

- Run on a single page, dynamically bringing in content as necessary
- Frameworks such as AngularJS, Ember.js, Backbone.js, React, etc.

Definitions

- **Hard Navigation**: The first page load, which will include all static HTML, JavaScript, CSS, the SPA framework itself (e.g. angular.js), plus showing the initial route
- **Soft Navigation**: Any subsequent route (address bar) change
- Any URL might be loaded via either hard or soft navigation
SPAs...

3 Challenges
Challenge 1: The onload Event No Longer Matters

Traditional Websites:

- On navigation, the browser begins downloading all of the JavaScript, CSS, images and other static resources.
- Once all static resources are fetched, the body's onload event will fire.
- This is traditionally what websites consider as page load complete.
Challenge 1: The onload Event No Longer Matters
Challenge 1: The onload Event No Longer Matters

Single Page Apps:

- Load all static content like a traditional website
- The frameworks' code will also be fetched (e.g. angular.js)
- (the onload event fires here)
- Once the SPA framework is loaded, it starts looking at routes, fetching views and data
- All of this content is fetched after the onload event
Challenge 1: The onload Event No Longer Matters

<table>
<thead>
<tr>
<th>Load Time</th>
<th>First Byte Time</th>
<th>Start Render Time</th>
<th>Visually Complete Time</th>
<th>Speed Index</th>
<th>DOM Elements</th>
<th>Result (error code)</th>
<th>Document Complete Time</th>
<th>Fully Loaded Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.225s</td>
<td>0.204s</td>
<td>1.396s</td>
<td>1.900s</td>
<td>1794</td>
<td>155</td>
<td>99699</td>
<td>1.225s</td>
<td>2.609s</td>
</tr>
</tbody>
</table>

### Performance Metrics

- **RUM First Paint**
- **DOMContentLoaded**
- **loadEvent**

---

**Page Load Complete (?)**

---

![Image showing a chart with performance metrics and a note about page load complete.](chart.png)
Challenge 1: The onload Event No Longer Matters

- Browser fires `onload` at 1.225 seconds
- All visual resources (.jpegs) aren't complete until after 1.7 seconds
- Filmstrip confirms nothing is shown until around 1.7 seconds
- `onload` fired 0.5 seconds too early!
Challenge 2: Soft Navs are not Real Navigations

- This is great for performance
- The browser is **no longer re-rendering** the same header, footer or common components
- The browser is **no longer re-parsing** the same HTML, JavaScript and CSS

**Bad** for measuring:

- Browser events (readyState, onload) and metrics (NavigationTiming) are all geared toward a single load event
- Won't run again until the next time it loads on a **full navigation**
Challenge 3: Browser Won't Tell You When All Downloads Have Completed

- The browser fires **onload only once**
- The onload event helps us know when all **static content** was fetched
- In a **soft navigation** scenario, the browser does not fire the onload event again, so we don't know when its content was fetched
Challenge 3: Browser Won't Tell You When All Downloads Have Completed

SPA soft navigations may fetch:

- Templates
- Images
- CSS
- JavaScript
- XHRs
- Videos
How to Measure SPAs

We need to figure out at what point the navigation started (the start event), through when we consider the navigation complete (the end event).
How to Measure SPAs: Start Event

Hard navigations:

- Same as a traditional app - navigationStart

Soft navigations:

- We need to figure out when the user's view is going to significantly change
- The browser history is changing
- SPA framework routing events can give us an indicator that the view will be changing
- Other important events that might indicate a view change are a user click, or an XHR that triggers DOM changes
How to Measure SPAs: Start Event

SPA frameworks fire routing events when the view is changing:

- AngularJS: `$rootScope.$on("$routeChangeStart")`
- Ember.js: `beforeModel` or `willTransition`
- Backbone.js: `router.on("route")`
How to Measure SPAs: Start Event

Clicks & XHRs:

• To determine if a user click or XHR is really triggering a navigation, we can listen to what happens next.

• If there was a lot of subsequent network activity, we can keep on listening for more events.

• If history (address bar) changed, we can consider the event the start of a navigation.

• If the DOM was updated significantly, we can consider the event the start of a navigation.

• If nothing else happened, it was probably just an insignificant interaction.
How to Measure SPAs: Start Event

- Browser navigates (hard nav)
- SPA route change
- User click/Interaction
- XHR activity
How to Measure SPAs: End Event

When do we consider the SPA navigation complete?

- When all networking activity has completed
- When the UI is visually complete (above-the-fold)
- When the user can interact with the page

Remember: onload doesn't work:

- Only tracks static resources
- SPA frameworks dynamically load other content
- onload doesn't fire for Soft Navs
How to Measure SPAs: End Event

Let's make our own SPA complete event:

- Similar to the body `onload` event, let's wait for all network activity to complete.
- This means we will have to intercept both implicit resource fetches (e.g. from new DOM elements) as well as programmatic (e.g. XHR) resource fetches.
How to Measure SPAs: Monitoring XHRs

- XHRs are used to fetch HTML, templates, JSON, XML, data and other assets
- We should monitor to see if any XHRs are occurring
- The XMLHttpRequest object can be proxied
- Intercept the `.open()` and `.send()` methods to know when an XHR is starting
- Listen to `onreadystatechange` events to know when it's complete

How to Measure SPAs: Monitoring DOM Fetches

• XHR is the main way to fetch resources via JavaScript

• What about Images, JavaScript, CSS and other HTML elements that trigger resource fetches?

• We can't proxy the Image object as that only works if you create a new Image() in JavaScript

• If only we could listen for DOM changes…
MutationObserver provides developers a way to react to changes in a DOM

- `observe()` for specific events
- Get a callback when mutations for those events occur
How to Measure SPAs: Monitoring DOM Fetches

- Start listening when an XHR, click, route change or other interesting navigation-like event starts
- Use `MutationObserver` to listen for DOM mutations
- Attach `load` and `error` event handlers and set timeouts on any `IMG`, `SCRIPT`, `LINK` or `FRAME`
- If an interesting element starts fetching keep the navigation "open" until it completes
- After the last element's resource has been fetched, wait a few milliseconds to see if it kicked off anything else
- If not, the navigation completed when the last element's resource was fetched

How to Measure SPAs: Monitoring DOM Fetches

What's interesting?

- **Internal** and **cached resources** may not fetch anything, so you have to inspect elements first

- **IMG** elements that haven't already been fetched (naturalWidth==0), have external URLs (e.g. not data-uri:) and that we haven't seen before

- **SCRIPT** elements that have a **src** set

- **IFRAMEs** elements that don't have javascript: or about: protocols

- **LINK** elements that have a **href** set
How to Measure SPAs: Monitoring DOM Fetches

Why not ResourceTiming?

• ResourceTiming events are only added to the buffer after they complete

• In order to extend the SPA navigation end time, we have to know if any resource fetches are outstanding

• We can use ResourceTiming later to supplement the data we get from XHR+MO
How to Measure SPAs: Front-End vs. Back-End

Traditional apps:
How to Measure SPAs: Front-End vs. Back-End

**Traditional apps:**

- **Back-End:** HTML fetch start to HTML response start
- **Front-End:** Total Time - Back-End

**Single Page Apps:**

- **Back-End:** Any time slice with an XHR or SCRIPT outstanding
  - Since these are most likely critical path resources
- **Front-End:** Total Time - Back-End
Accelerated Mobile Pages
AMP: Accelerated Mobile Pages

What is AMP?
• A way to build web pages for improved performance
• Restricts what you can put in your site to achieve this

Components:
• AMP HTML: Similar to HTML5 with restrictions
• AMP JavaScript: JavaScript library you include
• Google AMP Cache: Free CDN

Restrictions
• Cannot include any first- or third-party JavaScript
AMP: RUM

<amp-pixel src="http://...">

- GET query URL
- **Substitution variables** to gather metrics:
  - Document info (URL, Canonical URL, Title, Referer)
  - NavigationTiming (TCP, DNS, SSL, Page Load, etc)
  - Navigation Type and Redirect Count
  - Persisted Client ID
  - Total Engaged Time
  - Screen/Viewport dimensions

Example:

<amp-pixel src="http://myserver.com/beacon?u=AMPDOC_URL&t=PAGE_LOAD_TIME">
AMP: RUM

<amp-analytics>

- AMP extension
- Built in vendor configs (> 25)
  - Easy to configure
  - Predefined list of metrics is sent to vendor
Continuity
• We measure everything up to navigation complete (page load or SPA nav)
• We measure whether users bounce or convert

But

• The bulk of user interaction and experience happens after navigation has completed
Which continuous variables can we measure and how?
Developer Tools
Developer Tools

“The fact that something is possible to measure, and may even be highly desirable and useful to expose to developers, does not mean that it can be exposed as runtime JavaScript API in the browser, due to various privacy and security constraints”

– Performance APIs, Security and Privacy

https://w3c.github.io/perf-security-privacy/
Continuity Metrics
FPS - Frames Per Second

- requestAnimationFrame(callback)
- Callback is run before the next paint

```javascript
// total frames seen this second
var frames = 0;

function measureFps() {
  frames++;

  // request a callback before the next frame
  window.requestAnimationFrame(measureFps);
}

// start measuring
window.requestAnimationFrame(measureFps);

// report on frame rate (FPS) once a second
setInterval(function() {
  console.log("FPS: " + frames);
  frames = 0;
}, 1000);
```
FPS - Long Frames

Frames > 16.6 ms lead to < 60 FPS

```javascript
var lastFrame = performance.now();
var longFrames = 0;

function measureFps() {
  var now = performance.now();

  // calculate how long this frame took
  if (now - lastFrame >= 18) { longFrames++;
}

  lastFrame = now;

  window.requestAnimationFrame(measureFps);
}

window.requestAnimationFrame(measureFps);

// report on long frames once a second
setInterval(function() {
  console.log("Long frames: " + longFrames);
  longFrames = 0;
}, 1000);
```
var latestFrame = 0;
var latestReportedFrame = 0;

setInterval(function() {
    // find the first VIDEO element on the page
    var vids = document.getElementsByTagName("video");
    if (vids && vids.length) {
        var vid = vids[0];
        if (vid.webkitDecodedFrameCount || vid.mozPaintedFrames) {
            latestFrame = vid.webkitDecodedFrameCount || vid.mozPaintedFrames;
        }
    }

    console.log("Video FPS: ",
                + Math.max(latestFrame - latestReportedFrame, 0));

    // reset count
    latestReportedFrame = latestFrame;
}, 1000);
CPU - Page Busy

- Browser doesn’t expose **CPU metrics** directly
- Detect **Busy** by running a function at a **regular interval**
- See if the callback runs at the time we **expect**
- If the callback was **delayed**, the page was Busy
- Busy can be **caused** by other **JavaScript, layout, render**, etc
setInterval(function() {
    var now = performance.now();
    var delta = now - last;
    last = now;

    // if we are more than 2x the poll
    // + deviation, we missed one period completely
    while (delta > ((POLLING_INTERVAL * 2) + ALLOWED_DEVIATION_MS)) {
        total++;
        late++;
        delta -= POLLING_INTERVAL; // adjust, try again
    }

    total++;

    if (delta > (POLLING_INTERVAL + ALLOWED_DEVIATION_MS)) {
        late++;
    }
}, POLLING_INTERVAL);
NET - Resources

- **ResourceTiming**
- **Bytes** available in ResourceTiming

```javascript
var resources =
    window.performance.getEntriesByType("resource");

// number of resources fetched
var resourceCount = resources.length;

// number of bytes
var bytesOverWire = 0;
resources.forEach(function(res) {
    bytesOverWire +=
        res.transferSize ? res.transferSize : 0;
});

console.log("Resources: " + resourceCount
    + " " + bytesOverWire + "b");
```
**HEAP - Memory Usage**

- Non-standard (Chrome only)
- **Reduced precision** to avoid privacy concerns

```javascript
// report on JS object memory once a second
setInterval(function() {
    var mem = window.performance && window.performance.memory && window.performance.memory.usedJSHeapSize;

    console.log("Memory usage: " + mem);
}, 1000);
```
**Battery**

- **Monitor** your visitor’s battery state
- **Reduce work** on low battery

```javascript
setInterval(function() {
    navigator.getBattery().then(function(batt) {
        console.log(batt.level);
    });
}, 1000);
```
Interactions
Interactions - User Input

- scroll
-mousemove
- click
- keydown
Interactions - Visibility

Window’s visibility state

document.addEventListener("visibilitychange", function() {
    console.log(document.hidden ? "hidden" : "visible");
}, false);

Also look at the IntersectionObserver

Interactions - Orientation

How the device is being held

```javascript
window.addEventListener("orientationchange", function() {
    console.log("orientation: " + screen.orientation.angle);
});
```
Size Metrics
Size - Nodes

- HTML size (bytes)
- Overall Node count
- IFRAME, IMG, SCRIPT, etc., node count
MutationObserver == change over time

```javascript
var d = document;
var mutationCount = 0;
var domLength =
    d.getElementsByTagName('*').length;

// create an observer instance
var observer = new MutationObserver(function(mutations) {
    mutations.forEach(function(mutation) {
        if (mutation.type !== 'childList') {
            return;
        }
        for (var i = 0; i < mutation.addedNodes.length; i++) {
            var node = mutation.addedNodes[i];
            mutationCount++;
            mutationCount += node.getElementsByTagName ?
                node.getElementsByTagName('*').length : 0;
        }
    });
});

// configure the observer
observer.observe(d, { childList: true, subtree: true });
```
var errorCount = 0;

window.onerror = function () {
    errorCount++;
}

setInterval(function () {
    console.log("Errors: " + errorCount);
    errorCount = 0;
}, 1000);
So what?

- Raw data != useful metrics
- Let’s measure the user experience
  - Smoothness
  - Responsiveness
  - Reliability
  - Emotion
Smoothness - FPS during scroll

Blanket your home in fast, reliable WiFi
Smoothness - FPS after interaction
Responsiveness

- How long it takes for the site to respond to input?
  - `requestAnimationFrame` to detect next paint
  - `MutationObserver` to detect DOM changes
- `UserTiming` to monitor your own code
- SPA instrumentation via `boomerang`
- Strive to give feedback within **100 milliseconds** of a user interaction!
Responsiveness

document.addEventListener("click", function(e) {
    var start = performance.now();
    requestAnimationFrame(function() {
        var delta = performance.now() - start;
        console.log("Click responsiveness: " + delta);
    });
}, false);
Responsiveness: Long Task API

- [https://github.com/spanicker/longtasks](https://github.com/spanicker/longtasks)
- Call a callback whenever a task takes too long to complete
Reliability

- JavaScript errors
- Leaks:
  - JavaScript memory usage over time
  - DOM size increase over time
Tracking Emotion
Rage Clicks

_Rage clicks_ are series of clicks in which your users are _pummeling_ their mouse buttons in _frustration_. It’s like punching your site in the face, usually because it’s _not doing what the user wants_ or expects it to.

– Caitlin Brett, FullStory
Rage Clicks

```javascript
var same = 0, x = 0, y = 0, targ = null;

document.addEventListener("click", function(e) {
    var nX = e.clientX; var nY = e.clientY;

    // calculate number of pixels moved
    var pixels = Math.round(
        Math.sqrt(Math.pow(y - nY, 2) +
        Math.pow(x - nX, 2)));

    if (targ == e.target || pixels <= 10) {
        same++;
    } else {
        same = 0;
    }

    console.log("Same area clicked: " + same);

    x = nX; y = nY; targ = e.target;
}, false);
```
Dead Clicks

- Clicking without any meaningful visual (DOM) change
- Might happen during (or right after) page load due to delayed JavaScript
Dead Clicks
Missed Clicks

user clicks near an element, but misses it
“People who are angry are more likely to use the mouse in a jerky and sudden, but surprisingly slow fashion.

People who feel frustrated, confused or sad are less precise in their mouse movements and move it at different speeds.”

– Inferring Negative Emotion from Mouse Cursor Movements

Martin Hibbeln, Jeffrey L. Jenkins, Christoph Schneider, Joseph S. Valacich, and Markus Weinmann
T rusting Your D ata

T rusting Your Data
Avoiding the Observer Effect

- JavaScript is **Single Threaded** (per domain)
- Unless the browser is idle, anything you do in JavaScript will slow down some other JavaScript
- So how do we measure performance without affecting performance?
Avoiding the Observer Effect

Use the IFrame Loader Technique to load measurement code outside the critical path

OR

Load measurement code after the onload event (but then you can't measure things that happen before onload)
Avoiding the Observer Effect

- Do as little as possible in event handlers, e.g., read a timestamp or save state to a variable.

- Do more expensive processing of this data via a `requestIdleCallback` that runs when the browser is idle.

`requestIdleCallback` is only available on Chrome and Opera at the moment, so use a shim for other browsers:

  - `requestIdleCallback` API Spec: MDN://docs/Web/API/Window/requestIdleCallback
  - Complete, spec compliant SHIM: github://aFarkas/requestIdleCallback
  - Minimal Google SHIM (not entirely compliant): github://github/requestIdleCallback
Avoiding the Observer Effect

- The `unload` event is a bit problematic because nothing can be deferred from it, so KiSS*, like writing to `localStorage`
- Anything else should be run deferred from an `onBeforeUnload` handler (which is non-standard and not supported everywhere)
- Also stay away from the `scroll` event or debounce/throttle

https://css-tricks.com/debouncing-throttling-explained-examples/

KiSS: Keep it Short & Simple
and whatever you do...

Never Instrument Flash!
Beaconing

the mechanics of getting performance data back to you
How to Beacon

There are several methods for sending ("beaconing") data:

- Image beacon
- Form beacon
- XMLHttpRequest beacon
- Beacon API
Create an `<IMG>` element from JavaScript with your query string data

```html
<script>
var img = new Image();
img.src = "http://myserver.com/beacon?pageLoad=100&dns=30";
// your data has been sent!
</script>
```

Pros:

- Easy and lightweight!
- 100% browser support

Hint: Return a 204 No Content HTTP response
Cons:

- Must put data on the query string (no POST)
- **URL length** (payload) limitation:
  - Windows IE 6-8: 2083 b
  - Windows Chrome, Safari, Opera, Firefox, IE9+: >100 kb
  - Mac: Chrome, Opera, Firefox, Safari: >100 kb
  - Android Chrome, Opera: 81,500 b
  - iOS Safari, Chrome: >100 kb
- Proxies? Beware
Need to update your server config config for >8 KB URLs:

// Apache
// https://httpd.apache.org/docs/2.2/mod/core.html#limitrequestline
// Default: LimitRequestLine 8190
LimitRequestLine 16380

// nginx
// Default: large_client_header_buffers 4 8k;
large_client_header_buffers 4 16k;

// JBoss
// https://docs.jboss.org/jbossweb/2.1.x/config/http.html
// Default: maxHttpHeaderSize="8192"
<Connector ... maxHttpHeaderSize="16384"/>
Create a `<FORM>` element, POST it to a hidden IFRAME

```javascript
var iframe, name = "beacon-" + Math.random();
try {
    // IE <= 8
    iframe = document.createElement('iframe name="' + name + '"');
} catch (ignore) {
    // everything else
    iframe = document.createElement("iframe");
}
form.action = "https://my-server.com/beacon";
form.target = iframe.name = iframe.id = name;
iframe.style.display = form.style.display = "none";
iframe.src = "javascript:false";
remove(iframe.id);
remove(form.id);

document.body.appendChild(iframe);
document.body.appendChild(form);

try { form.submit(); } catch (ignore) {}
```

```javascript
function remove(id) {
    var el = document.getElementById(id);
    if (el) {
        el.parentNode.removeChild(el);
    }
}

// cleanup
setTimeout(function() { remove(iframe.id); }, 10000);

https://github.com/SOASTA/boomerang/blob/master/boomerang.js#L710-L761
```
Form Beacon

Server Implementation:

- IE 10 *can hang* if given a 200 response. For best compat:
  - 204 No Content
  - Content-Length: 0
  - Content-Type: image/gif
  - X-XSS-Protection: 0

- Timing-Allow-Origin: *
  - To be able to capture ResourceTiming data

- Access-Control-Allow-Origin: [*|domain]
  - If sent from another origin
Form Beacon

Pros:

• POST data > 2,083 bytes (works in IE <= 8)

Cons:

• Complex JavaScript

• Less efficient than an Image beacon

• Lots of potential for browser bugs and incompatibilities! We've seen browser hangs, beacons opening in new windows, beacon URL in the status bar, etc. Use the boomerang.js code to avoid these.
Create an XMLHttpRequest object

```javascript
<script>
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://my-server.com/beacon?pageLoad=100", true);
xhr.send();
</script>

Pros:

- Easy and relatively lightweight!
- GET and POST support
- Large payload support

Hint: Return a 204 No Content HTTP response
XMLHttpRequest Beacon

Server Implementation:

- For **best performance**:
  - 204 No Content
  - Content-Length: 0
- **Timing-Allow-Origin**: *
  - To be able to capture **ResourceTiming** data
- **Access-Control-Allow-Origin**: [*/domain]
  - If sent from **another origin**
XMLHttpRequest Beacon

Cons:

- Not supported in IE 8/9. Requires XDomainRequest
  - GET/POST only
  - No custom headers
  - Only text/plain requests
  - No authentication or cookies
  - Restricted to same scheme as host page
Beacon API

How do we guarantee an beacon is sent when the user is leaving the page?

```javascript
window.addEventListener('unload', logData, false);

function logData() {
    var client = new XMLHttpRequest();
    client.open("POST", "/log", false); // third parameter indicates sync xhr. :(
    client.setRequestHeader("Content-Type", "text/plain; charset=UTF-8");
    client.send(analyticsData);
}
```

This is bad because it is synchronous and blocks the browser UI.

Async XHRs and Images can be cancelled in unload. onbeforeunload is not supported by Safari.
Beacon API

Beacon API requests are:

- **Prioritized** to avoid competition with other UI and higher-priority network requests
- **Optimized on mobile devices** (may be coalesced)
- **Guaranteed** to be initiated before page is unloaded
Beacon API: Usage

Simple API:

```javascript
window.addEventListener("visibilitychange", logData, false);

function logData() {
    if (document.visibilityState === "hidden") {
        navigator.sendBeacon("/log", analyticsData);
    }
}
```

Note: Use `visibilitychange` event as `unload` will not fire whenever a page is hidden and the process is terminated.
Beacon API: Browser Support

Beacon API

Allows data to be sent asynchronously to a server with `navigator.sendBeacon`, even after a page was closed. Useful for posting analytics data the moment a user was finished using the page.

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current aligned</td>
<td>8</td>
<td>13</td>
<td>47</td>
<td>49</td>
<td>51</td>
<td>52</td>
<td>9.2</td>
<td>9.3</td>
<td>4.4</td>
<td>4.4</td>
</tr>
<tr>
<td>Usage relative</td>
<td>11</td>
<td>14</td>
<td>48</td>
<td>53</td>
<td>9.1</td>
<td>39</td>
<td>10</td>
<td>all</td>
<td>52</td>
<td>51</td>
</tr>
<tr>
<td>Notes</td>
<td>49</td>
<td>54</td>
<td>10</td>
<td>40</td>
<td>41</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No notes
How to Beacon

We recommend:

- Use `sendBeacon()` if available
- If not, use Image beacon if < 2,000 bytes
- If not, use `XMLHttpRequest` if available and > 2,000 bytes
- If not, consider using FORM beacons or just shrug and move on
When to Beacon

Depends on your use-case:

- **As soon as you can** to be the most **reliable**

- **For general analytics:**
  - **As soon as you load** if you're not waiting for perf metrics

- **For performance analytics:**
  - **After onload** or SPA complete to gather all relevant performance metrics

- **For continuous metrics** or session length:
  - **On pagehide** if supported
  - **On beforeunload** if supported
  - **On unload** as a **last resort** (avoid using sync XHR)
Nixing Noise

Getting rid of obviously absurd data
Getting Rid of Noise

• Look at simple things like data type & range of all the data you collect

• Don't trust client timestamps, only trust deltas

• Check for a reasonable rate of data input per client

• Validate that your data collector isn't CSRFed

• Segregate known bot traffic (well behaved bots)
Some common **bogus** things we see

- All timer values are set to **9999999999**
- Client timestamps are more than a day, year or **30+** years in the past
- Requests that **do not change** over time
- Requests that are very **regularly spaced**
- Client makes exactly the same high valued purchase repeatedly
- Page Load time is **negative** or more than a week long
- Measured bandwidth in Terabits/second
Getting Rid of Noise

• We also use statistical methods to identify Outliers

• Use MAD or Tukey's method to identify data that is outside the expected range

• Nelson Rules to check for sufficient randomness

• $3\sigma$-Smoothing to compare actual values with expected values

• Don't throw away outliers, analyze them separately

MAD: Median Absolute Deviation: [wikipedia://Median_absolute_deviation](https://en.wikipedia.org/wiki/Median_absolute_deviation)
Offline First

measuring without network connectivity
ResourceTiming includes `workerStart` that tells us when a ServiceWorker that intercepted a request started.

Our measurement code should also run as a ServiceWorker, queuing up beacons while offline...

But how do we distinguish these queued beacons from forged beacons without an unexpired anti-CSRF token?

This is something we’re still experimenting with, so we don’t have any concrete recommendations, but we invite you to join the experiment.
W3C WebPerf Working Group

www.w3.org/2010/webperf

Founded in 2010 to give developers the ability to assess and understand performance characteristics of their web apps:

“The mission of the Web Performance Working Group is to provide methods to measure aspects of application performance of user agent features and APIs”

Microsoft, Google, Mozilla, Opera, Facebook, Netflix, Akamai, SOASTA, etc
**API Reference**

- **Hi Res Timer**  
  [https://w3c.github.io/hr-time/](https://w3c.github.io/hr-time/)

- **Navigation Timing**  
  [http://www.w3.org/TR/navigation-timing](http://www.w3.org/TR/navigation-timing)  

- **Resource Timing**  
  [https://www.w3.org/TR/resource-timing/](https://www.w3.org/TR/resource-timing/)  

- **Resource Timing Compressor**  
  [http://nicj.net/compressing-resourcetiming](http://nicj.net/compressing-resourcetiming)

- **User Timing**  
  [https://www.w3.org/TR/user-timing/](https://www.w3.org/TR/user-timing/)  

- **User Timing Polyfill**  
  [https://github.com/nicjansma/usertiming.js](https://github.com/nicjansma/usertiming.js)

- **User Timing Compressor**  
  [http://nicj.net/compressing-usertiming](http://nicj.net/compressing-usertiming)

- **Page Visibility**  
  [https://w3c.github.io/page-visibility/](https://w3c.github.io/page-visibility/)  
  [developer.mozilla.org/.../Web/API/Page_Visibility_API](https://developer.mozilla.org/.../Web/API/Page_Visibility_API)

- **Service Workers**  
  [https://www.w3.org/TR/service-workers/](https://www.w3.org/TR/service-workers/)  
  [developer.mozilla.org/.../Web/API/Service_Worker_API](https://developer.mozilla.org/.../Web/API/Service_Worker_API)

- **requestAnimationFrame**  
  [https://www.w3.org/TR/animation-timing/](https://www.w3.org/TR/animation-timing/)  
  [developer.mozilla.org/.../Web/API/window/requestAnimationFrame](https://developer.mozilla.org/.../Web/API/window/requestAnimationFrame)

- **requestIdleCallback**  
  [https://www.w3.org/TR/requestidlecallback/](https://www.w3.org/TR/requestidlecallback/)  
  [developer.mozilla.org/.../Web/API/Window/requestIdleCallback](https://developer.mozilla.org/.../Web/API/Window/requestIdleCallback)

- **Spec compliant requestIdleCallback SHIM**  
  [github://aFarkas/requestIdleCallback](https://github://aFarkas/requestIdleCallback)

- **Minimal requestIdleCallback SHIM**  
  [github://github/requestIdleCallback](https://github://github/requestIdleCallback)

- **Mutation Observer**  
  [developer.mozilla.org/.../Web/API/MutationObserver](https://developer.mozilla.org/.../Web/API/MutationObserver)

- **Performance Observer**  
  [https://www.w3.org/TR/performance-timeline-2/](https://www.w3.org/TR/performance-timeline-2/)  
  [developer.mozilla.org/.../Web/API/PerformanceObserver](https://developer.mozilla.org/.../Web/API/PerformanceObserver)

- **Intersection Observer**  
  [https://wicg.github.io/IntersectionObserver/](https://wicg.github.io/IntersectionObserver/)  
  [developer.mozilla.org/.../Web/API/Intersection_Observer_API](https://developer.mozilla.org/.../Web/API/Intersection_Observer_API)

- **Beacon API**  
  [https://w3c.github.io/beacon/](https://w3c.github.io/beacon/)  
  [developer.mozilla.org/.../Web/API/Navigator/sendBeacon](https://developer.mozilla.org/.../Web/API/Navigator/sendBeacon)

- **W3C Web Performance Working Group**  
  [http://www.w3.org/2010/webperf](http://www.w3.org/2010/webperf)
Further Reading

- Boomerang
  https://github.com/SOASTA/boomerang
- Andy Davies' Waterfall Bookmarklet
  https://github.com/andydavies/waterfall
- Mark Zeman's Heatmap
  https://github.com/zeman/perfmap
- Progressive Web Metrics
  https://github.com/paulirish/pwmetrics
- Accelerated Mobile Pages
  https://www.ampproject.org/
- The IFrame Loader Technique
  http://www.lognormal.com/blog/2012/12/12/the-script-loader-pattern/
- Affectiva Emotion Analyzer
  https://github.com/affectiva/youtube-demo
- MAD: Median Absolute Deviation
  https://en.wikipedia.org/wiki/Median_absolute_deviation
- John Tukey's fences
  datapiqtechnologies.com/.../highlighting-outliers-...-with-the-tukey-method/
- Nelson Rules
- Exponential Smoothing
  grisha.org/blog/2016/01/29/triple-exponential-smoothing-forecasting/
- Rage Clicking
- Inferring Emotion from Mouse Movements
  telegraph://technology/.../Websites-could-read-emotions-by-...-move-your-mouse.html
-Scroll Behaviour
  http://blog.chartbeat.com/2013/08/12/scroll-behavior-across-the-web/
- WebGazer: Eye tracking in JavaScript
  http://webgazer.cs.brown.edu/
- What JavaScript knows about you
  http://webkay.robinlinus.com/
- Video Metrics
  https://wiki.whatwg.org/wiki/Video_Metrics
- The Runtime Performance Checklist
  http://calendar.perfplanet.com/2013/the-runtime-performance-checklist/
- Jank Meter
  https://webperf.ninja/2015/jank-meter/
- RAIL Performance Audit of SFGate.com
  https://docs.google.com/document/d/1K-mKOqUijSjqZTExcBljtjd6E67oiK8H2ztOiq5tigk
- Debouncing and Throttling Events
  https://css-tricks.com/debouncing-throttling-explained-examples/
Photo Credits

Angel Delight by Auntie P
https://www.flickr.com/photos/auntiep/360764980/

Frustrated by Kevin Lawver
https://www.flickr.com/photos/kplawver/1903240219/
Thank You

http://slideshare.net/nicjansma/measuring-real-user-performance-in-the-browser
Philip Tellis
@bluesmoon

Nic Jansma
@nicj

https://github.com/SOASTA/boomerang

http://www.soasta.com/mpulse

http://slideshare.net/nicjansma/measuring-real-user-performance-in-the-browser