Bastian Grimm

Besseres Messen von Webseite-Ladegeschwindigkeit mit Paint- & User-Timings

In der Web-Performance-Optimierung tut sich aufgrund der zunehmenden Komplexität von Webseiten (insbesondere auch durch Single Page Apps sowie Progressive Web Apps) derzeit eine ganze Menge. Das reine Erfassen sowie Optimieren anhand von Legacy-Metriken wie Time to first Byte oder Page Load Time (mittels „window.onload" oder „document complete") sind bei Weitem nicht mehr ausreichend. Auch Googles PageSpeed Insights reflektieren nicht, wie schnell eine Seite gefühlt lädt. Paint- und User-Timings versprechen, hier Abhilfe zu schaffen. Experte Bastian Grimm gibt Tipps für zielführende Messmetriken für Ihren Sitespeed.

DER AUTOR

Bastian Grimm verantwortet bei der Peak Ace AG die Bereiche Suchmaschinenoptimierung sowie Performance-Content-Marketing und blickt dabei auf mehr als 15 Jahre Erfahrung im Performance-Marketing bzw. SEO zurück. Peak Ace ist eine 2008 gegründete, international tätige Performance-Marketing-Agentur mit Sitz in Berlin. Mit mehr als 100 Mitarbeitern realisiert Peak Ace Kampagnen in mehr als 20 Sprachen auf Muttersprachler-Niveau.

Hat man sich in der Vergangenheit bereits einmal mit der Optimierung von Webseite-Ladegeschwindigkeit beschäftigt, wurde meist unweigerlich auch auf Googles kostenloses PageSpeed-Insights-Tool (PSI, *http://einfach.st/gpsi*) zurückgegriffen – sozusagen um auch den „offiziellen" Blick des Suchmaschinengiganten auf die Geschwindigkeit der eigenen Seite mit zu berücksichtigen. Kritiker monieren hier zu Recht, dass – neben der Tatsache, dass hier zum Teil Optimierungsempfehlungen geliefert werden, die schlicht nicht umsetzbar sind (ich als Seitenbetreiber kann ja nicht die Caching-Dauer des Facebook-Pixels beeinflussen) –, beispielsweise eine Bewertung der Form „89/100" relativ wenig über die eigentliche Ladegeschwindigkeit einer Webseite aussagen kann. Und noch viel weniger kann diese Zahl reflektieren, wie schnell ebendiese Webseite gefühlt lädt. Eine einzelne Zahl kann dies schlicht nicht leisten.

Paint-Timings: Mehrere Schritte bis zum vollständigen Seitenaufbau

Die Paint-Timing API *(http://einfach.st/w3paint)*, deren Spezifikation im September 2017 veröffentlicht und durch das Chrome-Browser-Team im März 2018 mit dem Chrome 65 Release implementiert wurde, erlaubt den Zugriff auf verschiedene Paint-Zustände. Darüber hinaus finden aktuell folgende Metriken entsprechend Anwendung:

1. **Time to First Paint (FP)** – der Punkt, an dem der Browser irgendeinen beliebigen Inhalt anzeigt. Dies kann im Grunde jedes Element sein, z. B. eine einzelne Linie, ein Punkt etc. – oder auch das Ausspielen der Hintergrundfarbe.
2. **Time to First Contentful Paint (FCP)** – der Punkt, an dem der Browser einen ersten, wirklichen Inhalt aus dem DOM anzeigt. In der Regel ist das entweder ein Text oder ein Bild.

Foto: AzFree / thinkstockphotos.de

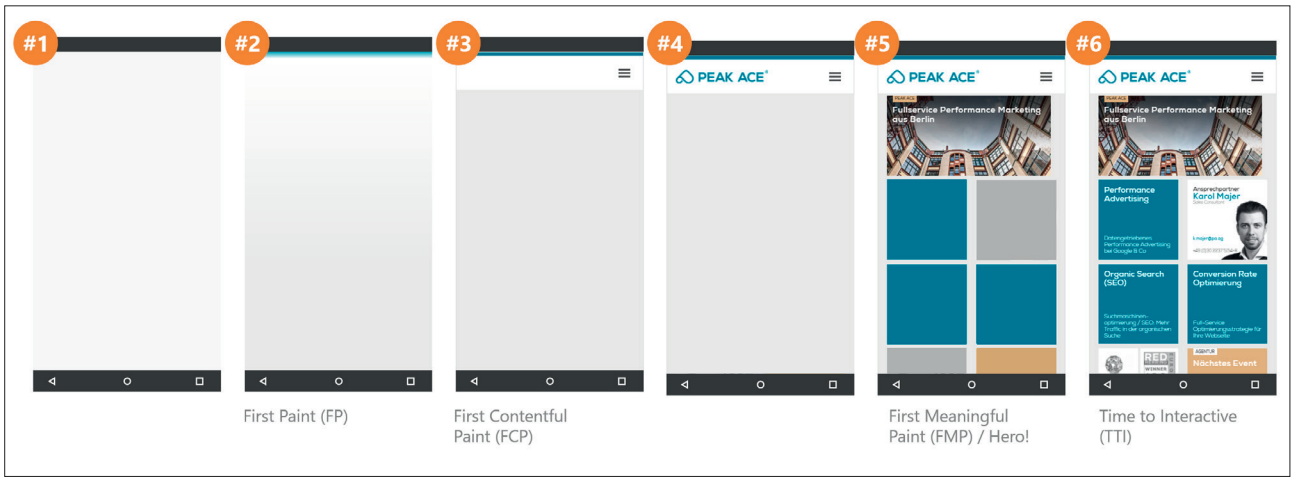

Abb.1: Browser Paint-Timings im zeitlichen Verlauf (Quelle: Peak Ace AG)

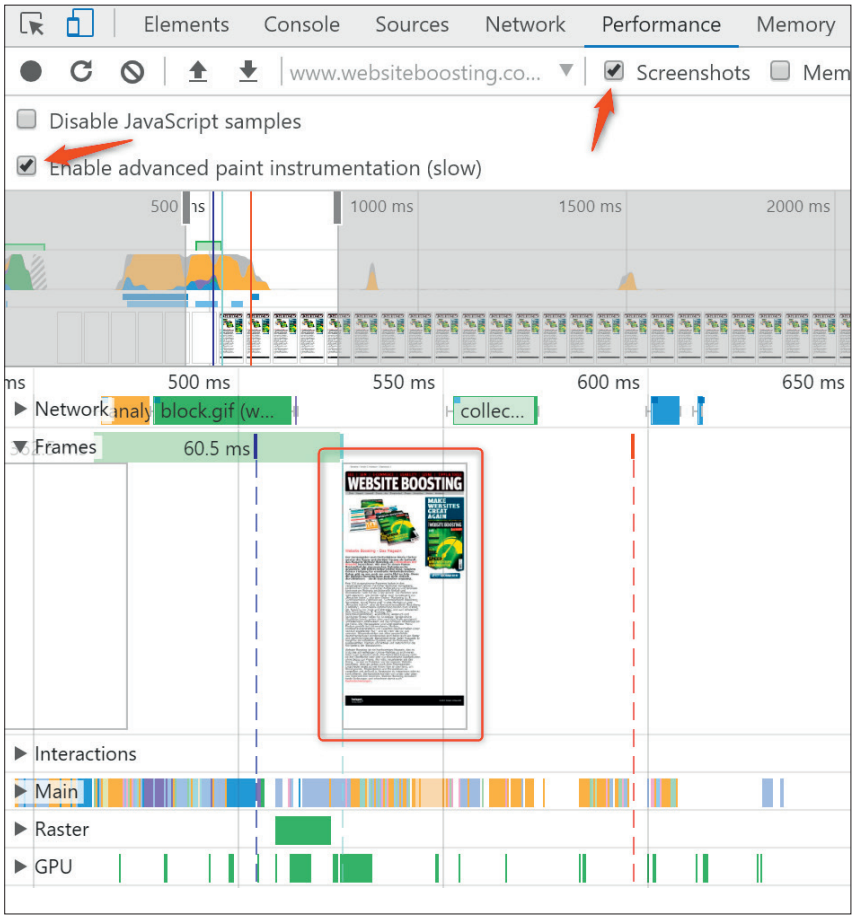

Abb.2: Paint-Events im Zeitverlauf für www.websiteboosting.com via Chrome Developer Console

3. **Time to First Meaningful Paint (FMP)** – bezeichnet den Zeitpunkt, an dem der Benutzer den wichtigsten Inhalt (das „Hero-Element“) einer jeweiligen URL als fertig geladen empfindet.
4. **Time to Interactive (TTI)** – der Zeitpunkt, an dem die Webseite gemeinhin als bedienbar gilt.

Die spannendste Render-Metrik ist derzeit wohl die Time to First Meaningful Paint, reflektiert sie doch am besten, wie schnell – pro Seitentyp/Template – das für den Besucher wichtigste Element angezeigt wird. Hier wird häufig von dem sogenannten Hero-Element gesprochen, also dem für den Besucher wichtigsten Element. Besucht jemand beispielsweise YouTube, so ist das für ihn mit Abstand wichtigste Element das jeweilige Video; alle anderen Elemente sind weniger relevant und könnten demnach auch mit Zeitverzug geladen werden.

Paint-Timings in der Chrome Developer Console darstellen

Die in den Chrome Browser integrierte Developer Console (unter Windows z. B. erreichbar mit der Tastenkombination Strg + Shift + i) erlaubt eine komfortable Darstellung der oben genannten Messpunkte. Dazu wechseln Sie einfach in den Tab „Performance“, wählen dort die Optionen „Screenshots“ sowie „Enable advanced paint instrumentation“ an und laden im Anschluss die gewünschte Seite neu.

Nach Abschluss des Ladevorgangs navigieren Sie nun in der linken Navigation zum Punkt „Frames“ und klappen diesen auf. Dort finden sich nun, anlog zur Zeitachse oben im Bild, die einzelnen Messpunkte grafisch aufbereitet.

PerformanceObserver im Chrome Browser

Für eine kontinuierliche und nicht nur punktuelle Messung von Paint-Timings kann beispielsweise Google Analytics verwendet werden. Dabei ist der gemeinhin bekannte Google-Analytics-Standardcode um den sogenannten PerformanceObserver zu erweitern, was in der Praxis aussieht wie in Abbildung 3 zu sehen ist.

Was geschieht hier im Detail?

1. Einbinden des asynchronen Google-Analytics-Standard-Snippets (inkl. Spezifizierung der Account-ID in der Form UA-XXXX-Y sowie Messen eines Seitenaufrufs).
2. Registrierung einer Instanz des PerformanceObservers, welcher in der Folge auf die Standardevents „First Paint“ sowie „First Contentful Paint“ reagieren und diese als Events in der Kategorie „Performance Metrics“ speichern wird.
3. Das alles geschieht, bevor das erste Stylesheet (CSS) aufgerufen wird, hier exemplarisch in der vorletzten Zeile dargestellt.

Tipp: Zum aktuellen Zeitpunkt ist es essenziell, dass der PerformanceObserver im ‹head› und vor dem ersten CSS-File registriert wird. Nur so kann sicher gewährleistet werden, dass dieser auch wirklich vor dem ersten Paint-Event ausgeführt wird. In einer zukünftigen Implementation soll ein interner Buffer implementiert werden, sodass dies nicht mehr zwingend notwendig ist; aktuell führt aber kein Weg drum herum.

Ein kurzer Hinweis zum Thema Kompatibilität: Das Chrome-Team hat den PerformanceObserver ab Version 64 implementiert, sprich Vorgängerversionen können hier keine Daten erfassen. Die Browser von Microsoft (Internet Explorer sowie Edge) bieten (noch) keinerlei Unterstützung. Im Firefox 57 (Quantum) funktioniert das Ganze auch, in den Vorversionen wiederum nicht.

PerformanceObserver mittels Google-Tag-Manager

Viele Webseiten-Betreiber verwenden mittlerweile allerdings gar keine native Template-Integration von Google Analytics mehr, sondern setzen vielmehr auf eine zentralisierte Tag-Management-Lösung, wie zum Beispiel den Google-Tag-Manager. Von dieser Lösung ausgehend werden dann, meist abhängig von URLs oder anderen, internen Regeln, alle notwendigen Tracking-Pixel ins Template integriert und entsprechend ausgelöst. Mit ein wenig Überzeugungsarbeit bzw. einigen Code-Anpassungen kann der PerformanceObserver natürlich auch über den Google-Tag-Manager ausgespielt werden (siehe Abbildung 4).

Der versierte Tag-Manager-Benutzer weiß natürlich schon, dass neben dem Code dennoch, wie auch für alle anderen Pixel-Events entsprechend, die Tag-, Variable- und Trigger-Configuration weiterhin notwendig ist.

```
<head>
  <!-- Add the async Google Analytics snippet first. -->
  <script>
  window.ga=window.ga||function(){(ga.q=ga.q||[]).push(arguments)};ga.l=+new Date;
  ga('create', 'UA-XXXXX-Y', 'auto');
  ga('send', 'pageview');
  </script>
  <script async src='https://www.google-analytics.com/analytics.js'></script>

  <!-- Register the PerformanceObserver to track paint timing. -->
  <script>
  const observer = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // `name` will be either 'first-paint' or 'first-contentful-paint'.
      const metricName = entry.name;
      const time = Math.round(entry.startTime + entry.duration);

      ga('send', 'event', {
        eventCategory: 'Performance Metrics',
        eventAction: metricName,
        eventValue: time,
        nonInteraction: true,
      });
    }
  });
  observer.observe({entryTypes: ['paint']});
  </script>

  <!-- Include any stylesheets after creating the PerformanceObserver. -->
  <link rel="stylesheet" href="...">
</head>
```

Abb.3: Beispielcode zur Einbindung von Google Analytics inklusive des PerformanceObservers *(Quelle: Google; http://einfach.st/gdevperform)*

```
<script>
if( typeof( PerformanceObserver ) != 'undefined' ){
    var dataLayer = dataLayer || [];
    const observer = new PerformanceObserver( list => {
        for( const entry of list.getEntries() ){
            var time = Math.round( entry.startTime + entry.duration );
            var actionTime = {};
            actionTime[ entry.name ] = time;
            dataLayer.push( actionTime );
        }
    });
    observer.observe({entryTypes: ['paint']});
}</script>
```

Abb.4: Beispielcode zu Verwendung des PerformanceObservers via GTM *(Quelle: Peak Ace AG)*

Verwenden von individuellen Messpunkten

Die W3C User Timing-Spezifikation (*http://einfach.st/w3user*) bietet zudem eine API, um benutzerdefinierte Metriken in Templates zu integrieren und damit noch granularer zu messen. Dies geschieht über zwei Hauptfunktionen:

» **performance.mark** zeichnet die Zeit (in Millisekunden) seit navigationStart auf
» **performance.measure** erfasst das Delta zwischen zwei einzelnen Markern

Auch hier ein kurzer Hinweis zum Thema Kompatibilität: Die User Timing

```
<link rel="stylesheet" href="/mein-stylesheet-1.css" />
<link rel="stylesheet" href="/mein-stylesheet-2.css" />
<link rel="stylesheet" href="/mein-stylesheet-3.css" />
<script>
performance.mark("css blockieren beendet");
</script>
```

Abb.5: Beispielcode eines Markers zur Erfassung, wann sämtliche CSS nicht mehr den Seitenaufbau blockieren

```
<img src="/bilder/das-hero-element.jpg"
     onload="performance.clearMarks('hero img angezeigt');
             performance.mark('hero img angezeigt');"
     alt="" />
<script>
performance.clearMarks("hero img angezeigt");
performance.mark("hero img angezeigt");
</script>
```

Abb.6: Beispielcode eines doppelten Markers zur Erfassung, wann ein Hero-Element angezeigt wurde

API ist allen aktuellen Browsern verfügbar. In den folgenden Beispielen wird daher die native API verwendet, im Live-Code sollte dennoch immer auf Kompatibilität geprüft oder ein Wrapper verwendet werden. Eine gute Polyfill gibt es von Pat Meenan unter *http://einfach.st/github4*.

Die User-Timing API ist insgesamt relativ einfach gehalten, dennoch kann es manchmal schwierig sein zu wissen, wann entsprechende Marker Sinn machen. Das liegt primär an der komplexen Funktionsweise des jeweiligen Browsers und vor allem an der Art und Weise, wie beispielsweise CSS und synchrone JavaScripte das Parsen und Rendern möglicherweise blockieren. Es soll an dieser Stelle auch nicht unerwähnt bleiben, dass die folgenden Beispiele insgesamt deutlich vereinfacht wurden. In vielen Fällen ist nicht sofort offensichtlich, warum eine Webseite beispielsweise erst mit einer Verzögerung beginnt, entsprechende Inhalte zu rendern. Der Teufel steckt häufig sprichwörtlich im Detail.

Anwendungsfall #1: Stylesheets blocken nicht mehr die Ausgabe

Das Laden eines Stylesheets blockiert das Rendern der gesamten Seite. Für den Besucher passiert zu diesem Zeitpunkt rein gar nichts. Dies gilt sowohl für alle gängigen Browser als auch für dynamisch geladene CSS. Daher empfiehlt es sich zu messen, wann alle Stylesheets (sofern mehr als eines vorhanden ist) fertig prozessiert und den „Blocking"-Zustand verlassen haben. Dies kann folgendermaßen passieren: siehe Abbildung 5.

Ein ähnliches Vorgehen ist beispielsweise auch für JavaScript denkbar, hier ist allerdings zu beachten, dass Messpunkte durch etwaige asynchrone Skripte oder solche, die im Footer (nachträglich) geladen werden, möglicherweise nicht mehr akkurat ausfallen.

Anwendungsfall #2: Das Hero-Element wird dem Besucher angezeigt

Wie eingangs erwähnt, konzentrieren sich Besucher in der Regel auf ein oder evtl. mehrere kritische Designelemente auf der Seite – das sogenannte Hero-Element. Auch die zugehörige Ladezeit lässt sich mittels Markern entsprechend erfassen (Abbildung 6).

Die Änderung hier ist die kombinierte Verwendung des onload-Events innerhalb des <img>-Tags gemeinsam mit dem inline-Skript. Nur dadurch wird genau erfasst, wann das Bild tatsächlich gerendert wird. Das onload-Event liefert die Renderzeit, wenn der Bilddownload länger dauert als die blockierenden Ressourcen. Der inline-Marker spiegelt die Renderzeit wider, wenn das Bild schnell heruntergeladen wird, aber blockierende Ressourcen (wie zuvor genannte CSS/JS) verhindern, dass das Bild angezeigt wird.

Tipp: Jedes Template bzw. jeder Seitentyp hat in der Regel sein eigenes Hero-Element. Nehmen Sie sich die Zeit und definieren Sie mindestens einmal, was für die Startseite, Kategorienseiten sowie Artikel-/Produktdetailseiten das jeweils wichtigste Element ist. Und wenn Sie schon dabei sind, statten Sie diese doch direkt mit entsprechenden User-Timing-Markern aus.

Fazit

Insbesondere auf mobilen Endgeräten – denn hier ist die Bereitschaft, zu warten, begrenzter denn je und gleichermaßen die Konnektivität nicht selten schwach – und spätestens mit der Ankündigung seitens Google, Performance nunmehr auch „offiziell" als Rankingkriterium zu betrachten (siehe Website Boosting Ausgabe #49), ist es nunmehr absolut unumgänglich, sich gezielt mit dem Thema Web Performance auseinanderzusetzen. Doch nur mit einer guten Datenbasis können fundierte Entscheidungen getroffen und Optimierungspotenziale abgeleitet werden; die neuen Browser APIs bieten hier großartige Möglichkeiten, wenngleich der Implementationsaufwand aufgrund der Tatsache, dass viele dieser Funktionalitäten noch relativ neu sind, aktuell an einigen Stellen noch etwas für Kopfschmerzen sorgt. ¶