Performance-Probleme in Websites erkennen und beseitigen

Performance-Probleme in Websites erkennen und beseitigen

[expand title=“mehr lesen…“]

Performance-Probleme in Websites erkennen und beseitigen

Praxis Web-Performance

Bild: Rudolf A. Blaha

Ungebremst

Performance-Probleme in Websites erkennen und beseitigen

Surfer schätzen komplexe Apps, geschmeidige Animationen, Webfonts, Videos und hochauflösende Fotos. Viele Seiten laden, derart aufgemotzt, aber zu langsam. Lahme Websites wieder flott zu machen ist ein Mehrkampf mit vielen Disziplinen – ein Überblick. Von Herbert Braun

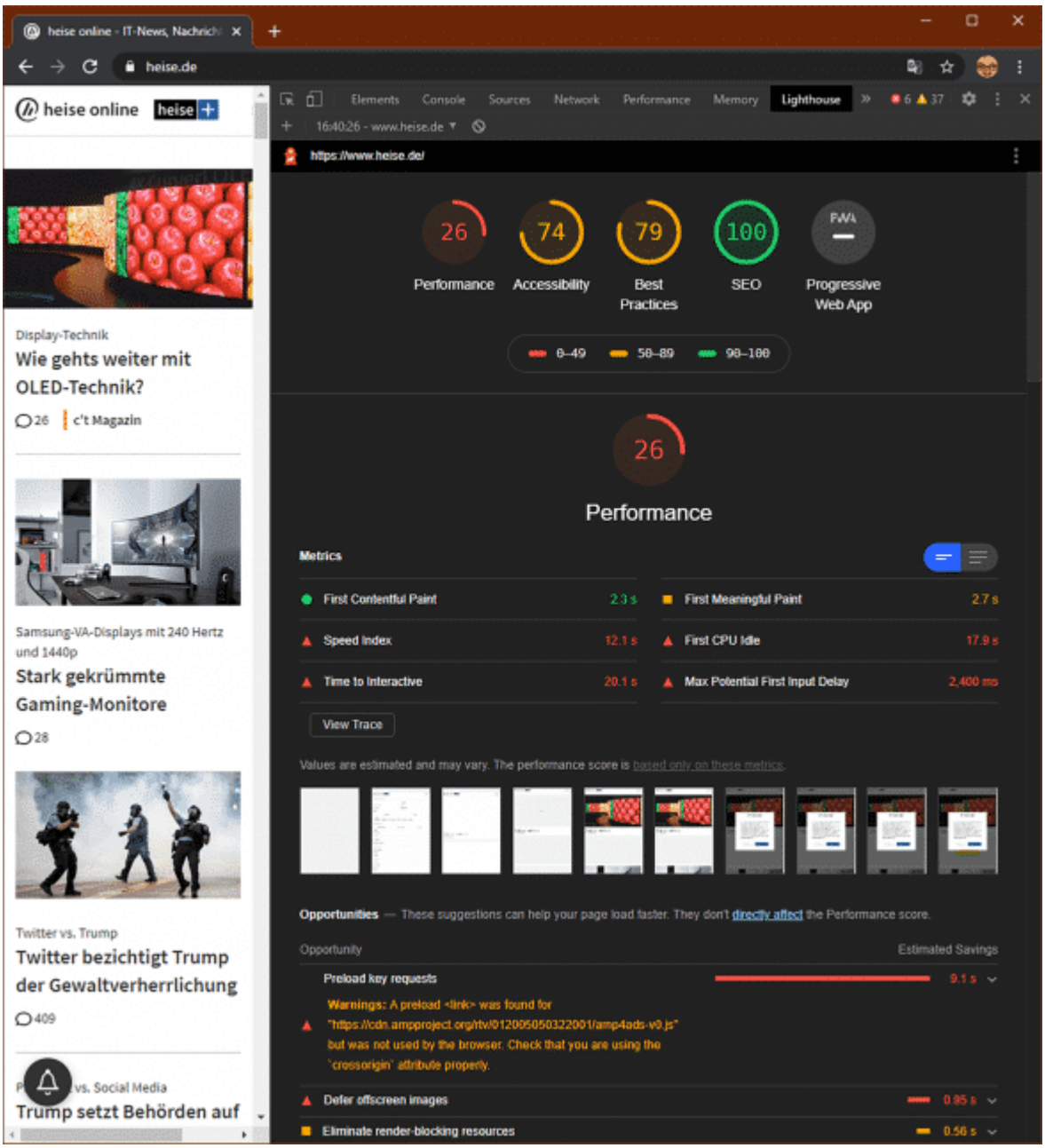
Eine durchschnittliche Webseite wiegt heute zwei MByte, die sich auf 75 HTTP-Requests verteilen (siehe ct.de/yp4b). Fast ein halbes MByte JavaScript-Code hat der Browser dabei zu verdauen. Gleichzeitig sind die Nutzer nicht mehr so geduldig wie zu ISDN-Zeiten: Drei Sekunden leerer Bildschirm sind für manchen Besucher schon zu viel. Eine Website, die nach zehn Sekunden noch nicht geliefert hat, wird den überwiegenden Teil ihrer Besucher verloren haben.

Es gibt viele sehr unterschiedliche Maßnahmen, die Sie als Website-Betreiber umsetzen können, um ihre Seiten flotter zu machen. Dieser Artikel beschreibt Optimierungen für das Frontend. Er gibt einen Überblick über das Spektrum der Möglichkeiten und geht nur vereinzelt in die Tiefe; die Umsetzung im Detail hängt ohnehin stark von den Anforderungen und Problemen der jeweiligen Website ab.

Level 0: Testwerkzeuge

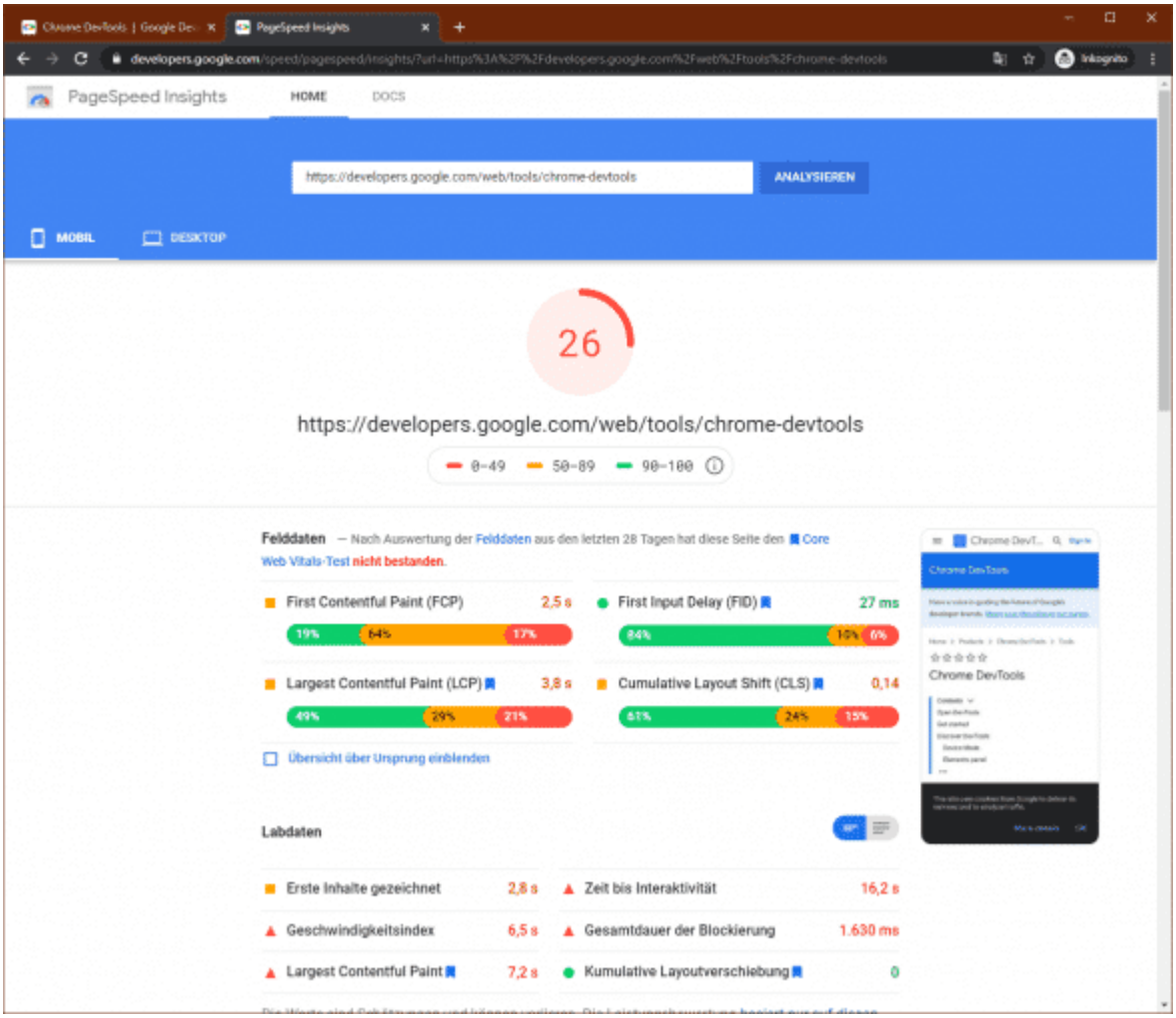
Zunächst gilt es herauszufinden, wo es klemmt. Heute benutzt man für Tests und Tipps meist Google PageSpeed Insights (PSI), Webpagetest.org oder Lighthouse. PSI ist vergleichsweise übersichtlich und eignet sich gut für Einsteiger. Das Open-Source-Projekt Webpagetest.org legt den Fokus mehr auf die Aufbereitung der Rohdaten als auf klare Handlungsanweisungen.

Lighthouse – ebenfalls Open Source – stammt wie PSI von Google, testet aber nicht nur die Performance einer Website, sondern etwa auch SEO und Barrierefreiheit; es steckt hinter den Analysefunktionen von PSI, wertet aber anders aus. Lighthouse ist kein Webdienst: Sie finden es in den Chrome-Entwicklerwerkzeugen, können es aber auch als Node.js-Anwendung installieren.



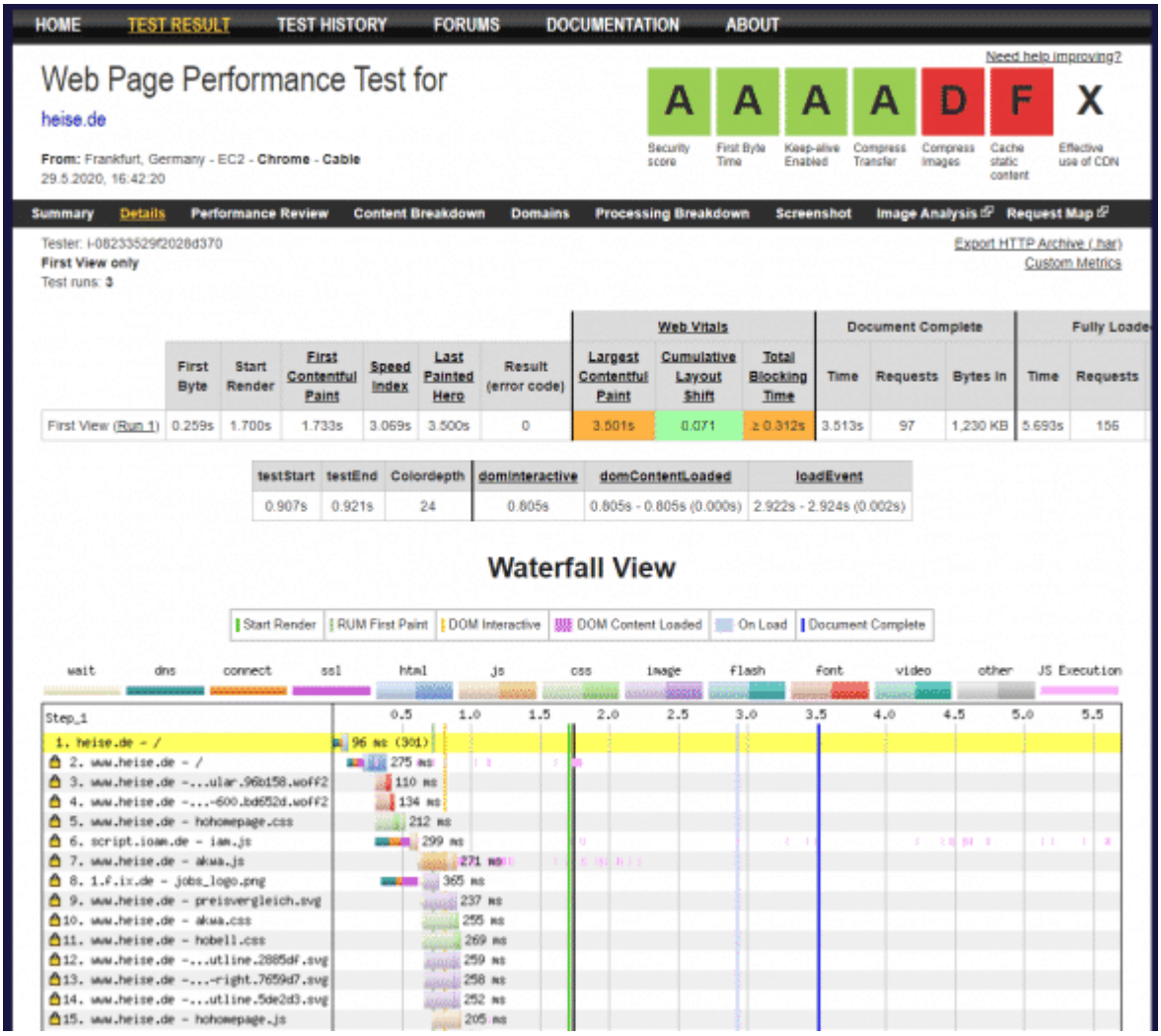
Lighthouse zeigt hübsch gestaltete Messergebnisse und wartet mit konkreten Verbesserungshinweisen auf.

Die Messergebnisse geben Anhaltspunkte, doch sollten Sie sie nicht überbewerten: Sie hängen oft von Zufällen ab und weichen zum Beispiel zwischen Lighthouse und PSI ab. Selbst bei Google-eigenen Seiten fällt der Geschwindigkeitsindex mitunter schlecht aus. Nützlicher sind die Ratschläge („Nicht genutztes JavaScript entfernen“, „Bilder richtig dimensionieren“ etc.), verbunden mit konkreten Angaben zu den betroffenen Dateien und Zeilen.



Googles PageSpeed Insights verwendet eine ähnliche Technik wie Lighthouse, kommt aber zu anderen Ergebnissen.

Unabhängig von Lighthouse finden Sie in den Entwicklerwerkzeugen der gängigen Browser Werkzeuge zum Messen von Netzwerkzugriffen, zur Rendering-Performance und zum Ressourcenverbrauch. Diese zeichnen nach der Aktivierung große Mengen an Daten auf, die Sie anschließend studieren können, um Performance-Engpässe auszumachen. Allerdings sind sie für Website-Tuning-Einsteiger kaum geeignet.



Webpagetest.org ist eine Alternative zu Googles Performance-Werkzeugen.

Level 1: Abspecken

Browser-Entwicklerwerkzeuge erlauben es, den Datendurchsatz zu drosseln, beispielsweise, um die Nutzung im Mobilfunk nachzustellen. Wer das einmal ausprobiert hat, wird sich mit mehr Engagement dem Entrümpeln und Komprimieren der Website widmen.

Das größte Einsparpotenzial haben meist die Bilder – keine andere Maßnahme wirkt so schnell wie deren Optimierung. Klar, dass ein Bild nicht größer sein sollte als das Maximum der Anzeigebreite. Was die Sache kompliziert macht, sind „Retina“-Displays, die Bilder höher auflösen können. Ein iPhone etwa stellt in der Standardskalierung jedes CSS-Pixel mit 2 × 2 Gerätepixeln dar; eine 500 × 300 Pixel große Bilddatei wird in

einem entsprechend großen CSS-Container okay aussehen, aber das Gerät könnte auf dieser Fläche auch 1000 × 600 Pixel unterbringen – ein Bild sieht so einfach schärfer aus.

Um solche Fälle und unterschiedliche Bildgrößen durch responsives Layout abzufangen, stehen Frontend-Entwicklern CSS-Media-Querys und insbesondere die HTML-Attribute srcset und sizes zur Verfügung. Der Browser ermittelt anhand dieser Angaben, welche Bilddatei am besten passt, und lädt nur diese herunter, zum Beispiel:

```
<img alt="Bild" srcset=
  "standard.jpg 1x, retina.jpg 2x">
```

Eine JPEG-Qualitätsstufe von mehr als 80 oder eine verlustfrei komprimierte PNG-Grafik sind im Web meist Bandbreitenverschwendung. Auch das Entfernen von Metadaten oder effizientere Komprimierung holen etliche KByte heraus. Umsetzen lässt sich so was mit üblicher Bildbearbeitungs- und Betrachtungs-Software oder mit Konsolen-Tools wie jpegtran, jpegoptim oder optipng. Diese Tools verarbeiten große Mengen an Bildern und lassen sich in die Build-Pipeline integrieren. Die folgende Anweisung schrumpft manche Fotos auf ein Zehntel ihrer Dateigröße (Achtung, überschreibt Quelldateien!):

```
jpegoptim -o -m75 --strip-all  --all-progressive *.jpg
```

Bei JPEGs empfiehlt sich das progressive Rendering, bei dem das Bild von Anfang an in voller Größe erscheint und während des Ladens immer detailgenauer wird – das fühlt sich für den Benutzer schneller an. Für Icons kommen heute Vektorgrafiken in Form von SVGs oder Iconfonts zum Einsatz. PNGs sind vor allem bei Transparenzen interessant. Das neue WebP-Format wiegt nur etwa 80 bis 90 Prozent einer gleichwertigen JPEG-Datei, aber Sie brauchen gegebenenfalls ein Fallback für Internet Explorer. Bislang nur in Chrome läuft AVIF, das seine Stärken bei hoher Kompressionsrate ausspielt und GIF-ähnliche Animationen erlaubt.

Für Videos setzen viele Websites auf externe Dienstleister, die beim Streamen die Wiedergabequalität an die Bandbreite anpassen. Wo aber ein <video> oder <audio> zum Einsatz kommt, das eine Mediendatei anfordert, kann die richtige Komprimierung Megabytes an Daten einsparen. Tools wie ffmpeg erledigen diesen Job zuverlässig. Leider gibt es keine Entsprechung zu srcset für gestreamte Medien.

Auch den Website-Code sollten Sie zusammenstauchen. Code-Minifizierungswerkzeuge gibt es für CSS und HTML, aber mehr holen Sie bei JavaScript heraus. Das bekannteste Tool dafür heißt „Uglify“ – sein Output ist für den Menschen kaum leserlich, doch der Maschine ist das egal.

Anstrengender, aber lohnender ist es, unnötigen Code komplett rauszuwerfen. JavaScript-Bibliotheken lassen den Code-Umfang enorm anwachsen. Daher sollte sich der Entwickler bei jedem Third-Party-Skript fragen: Brauche ich das wirklich? Muss ich moments.js einbinden, wenn ich einmal ein Datum umrechne? Lohnt sich das Karussell-Plug-in, benötige ich jQuery, weil $(...) so schön kurz ist?

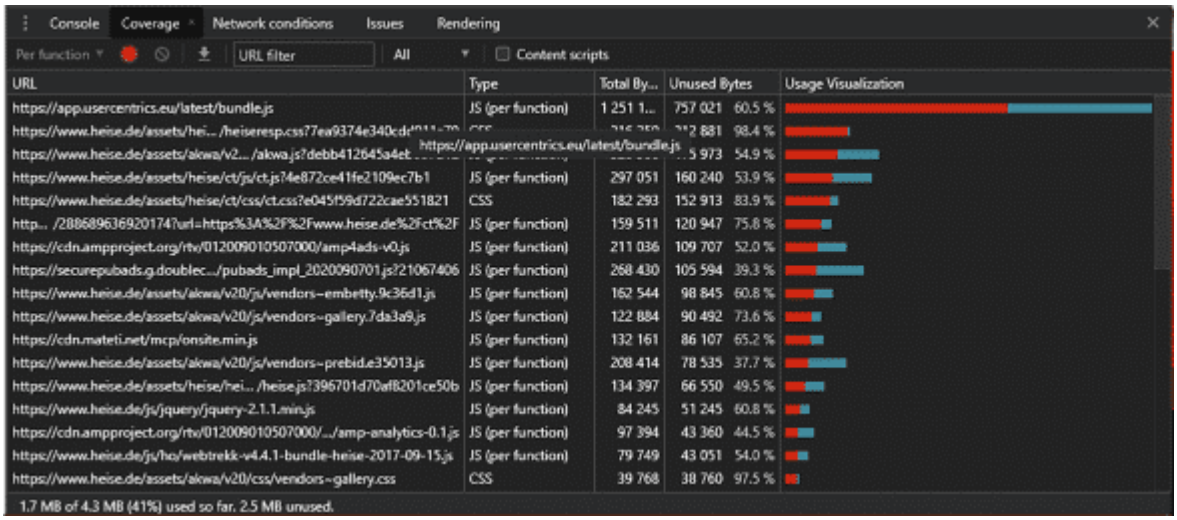
Nicht zu vergessen: Der Browser ist nach dem Download nicht fertig, sondern muss den Code auch noch verarbeiten. Während das etwa bei Bildern eine Frage von Millisekunden ist, leistet er bei JavaScript Schwerarbeit, die den Haupt-Thread oft sekundenlang blockiert. Auf einem leistungsschwachen Gerät kann das Kompilieren und Ausführen länger dauern als der Download. Tests mit realer Hardware bei unterschiedlicher Netzqualität fördern dabei mitunter Überraschendes zu Tage, sind aber aufwendig.

In gewachsenen Projekten findet sich oft erstaunliches Code-Gerümpel wie unterschiedliche jQuery-Versionen oder Polyfills, die seit Jahren keiner mehr braucht. Aber testen Sie die Seite gründlich und schmeißen Sie Code nicht vorschnell raus! Eine JavaScript-Exception stoppt nämlich die weitere Code-Ausführung, und wenn nichts mehr geht, nützt die schönste

Performance-Optimierung nichts mehr.

Chrome hat einen „Coverage“-Reiter (unter „More Tools“), der nicht benutzte CSS-Selektoren und JavaScript-Funktionen rot markiert. Bei den meisten Webseiten liegt deren Anteil bei weit über 50 Prozent. Das Node.js-Werkzeug UnCSS gibt das tatsächlich benutzte CSS aus – auch übergreifend für mehrere Seiten und Bildschirmgrößen.

Beim Import von Modulen ist es oft möglich, sich auf einzelne Komponenten zu beschränken. Moderne Bundler wie webpack oder Rollup beherrschen dieses „Tree-Shaking“ und kopieren mit `import {func1} from 'bigFile.js'` nur den zu func1 gehörenden Code ins Projekt statt der gesamten Skriptdatei.



Wie Chromes „Coverage“-Werkzeug zeigt, braucht man viele eingebundene Skripte und Stile nicht auf der aktuellen Seite.

Level 2: Ausliefern

Trotz Detailverbesserungen hat das Netzwerkprotokoll HTTP seine Wurzeln in den frühen 90er-Jahren, und TCP, auf dem es aufsetzt, ist noch älter. Beide erledigen den Job solide, aber ein bisschen umständlich – für heute übliche Szenarien mit oft mehr als hundert Requests pro Seitenaufruf waren sie jedenfalls nicht gedacht.

Der Overhead, den beide Protokolle verursachen, macht besonders die Übertragung kleiner Dateien teuer. Deshalb gilt

es als Performance-Optimierung, kleine Datenpäckchen zu größeren zusammenzufassen – etwa durch das Bündeln mehrerer Skript- und Stylesheet-Dateien („Bundling“) oder durch Tricks wie CSS-Sprites, bei denen man alle Icon-Grafiken in ein Bild stopft, um mithilfe von CSS das passende herauszufischen.

Außerdem beschränken Browser gemäß der HTTP-Spezifikation die Zahl der gleichzeitigen Verbindungen zu einem Host; typischerweise erlauben sie sechs gleichzeitige Downloads. Um das zu umgehen, setzen manche Websites „Domain-Sharding“ ein – die Aufteilung der Ressourcen auf mehrere Subdomains.

HTTP/2 macht solche Hacks überflüssig. Es benötigt nur eine TCP-Verbindung, um beliebig viele HTTP-Antworten zu liefern – auch solche, die der Client noch gar nicht angefragt hat (Server-Push). HTTP/2 ist inzwischen ein etablierter Standard, der laut W3Techs in 45 Prozent aller Websites zum Einsatz kommt [1].

Tatsächlich findet man das Protokoll bei internationalen Websites wie Google, Facebook, Amazon, eBay, LinkedIn beziehungsweise bei deren Content Delivery Networks (CDN), die diese Technik allesamt beherrschen. Auch manche Shared--Hosting-Angebote von der Stange liefern mit HTTP/2 aus, während andere Hoster den Umstieg bisher gescheut haben. - Wunderdinge sollte man von HTTP/2 allerdings nicht erwarten.

Der schnellste Download ist natürlich der, der nicht stattfindet. Geschicktes Caching kann wiederholte Seitenaufrufe enorm beschleunigen und sogar dafür sorgen, dass der Besucher etwas sieht, wenn er offline ist. Dafür setzt man die HTTP-Header Cache-Control oder Expires ein. Bei heise online zum Beispiel darf der Browser ein Bild für einen Monat im Cache behalten, während das Stylesheet nur zwei Stunden gültig bleibt; die Startseite muss er dagegen schon nach 30 Sekunden neu anfordern. Ist zusätzlich ein ETag-Header gesetzt, können Browser und Server abgleichen, ob sie beide die gleiche Dateiversion haben; in diesem Fall antwortet der

Server mit einem 304-Code, ohne Daten zu übertragen.

Einen Schritt weiter geht der Frontend-seitig programmierbare Cache, der mit Progressive Web Apps (PWA) möglich ist. Hauptzweck ist es, Websites auf Mobilgeräten offline verfügbar zu machen, aber Performance-Optimierung für den Desktop-Browser funktioniert damit ebenso gut. Doch egal, ob PWA oder Cache-Control: Übertreiben Sie nicht, sonst sieht der Besucher zu lange eine veraltete Version der Website!

Level 3: Vor- und Nachliefern

Wenn Sie die Größe des Downloads verringert haben, können Sie darüber nachdenken, wann Sie bestimmte Ressourcen benötigen. Das Standardverhalten – eine HTML-Datei saugt beim Laden sämtliche dazugehörigen Skripte, Stile und Bilder aus dem Netz – ist meistens nicht das schnellste: Manches fordert man besser vorher schon an, anderes erst später.

Aber was heißt eigentlich „Schnelligkeit“ bei einer Webseite? Man kann die Zeit messen, die vom ersten Request bis zum Eintreffen des letzten Bits vergeht, aber das ist nicht unbedingt die relevante Größe. Den Nutzer interessieren eher drei andere Ereignisse: dass irgendetwas auf dem Bildschirm erscheint, dass er im Browser-Viewport ein halbwegs fertiges Layout sieht und dass er mit dieser Ansicht interagieren kann.

Diese Ereignisse sind der „First Contentful Paint“ (FCP), der „Largest Content Paint“ (LCP) oder der „First Meaningful Paint“ (FMP) – sowie die „Time to Interactive“ (TTI).

Wenn also das Laden einer Seite fünf Sekunden dauert, sollte der Benutzer bis zu diesem Zeitpunkt nicht auf einen weißen Bildschirm starren müssen. Idealerweise sieht er innerhalb einer Sekunde relevante Inhalte, die sich anschließend nur noch wenig verändern, und kann die Seite bereits bedienen, während der Browser noch unterhalb des Fensterausschnitts liegende Bilder, Videos und Interaktionen nachlädt.

Meistens stellen Bilder den größten Datenanteil, und so hat sich Lazy Loading etabliert – der Browser fordert die Bilder erst an, wenn er Zeit hat oder sie benötigt. Moderne Browser (mit Ausnahme von Safari) brauchen dafür kein JavaScript mehr: Ein loading="lazy" im <img> genügt. Auch für IFrames funktioniert dies.

Problematisch sind vor allem die Inhalte, die das initiale Rendern blockieren: im Head eingebundene JavaScript- und Stylesheet-Dateien. Trifft der Browser auf solche Inhalte, stoppt er den Seitenaufbau, lädt die Datei herunter und parst sie beziehungsweise führt sie aus, bevor er das Rendern fortsetzt.

Die wenigsten Skripte müssen laufen, bevor die Seite gerendert wurde. Oft verschiebt man daher <script>-Elemente ans Ende des <body>. Den gleiche Effekt erzielen Sie, wenn Sie das <script>-Element im Head lassen und mit dem Attribut defer versehen – allerdings startet der Browser den Download früher, was meist wünschenswert ist. Wenn die Reihenfolge der Skripte egal ist, können Sie stattdessen mit dem Attribut async arbeiten.

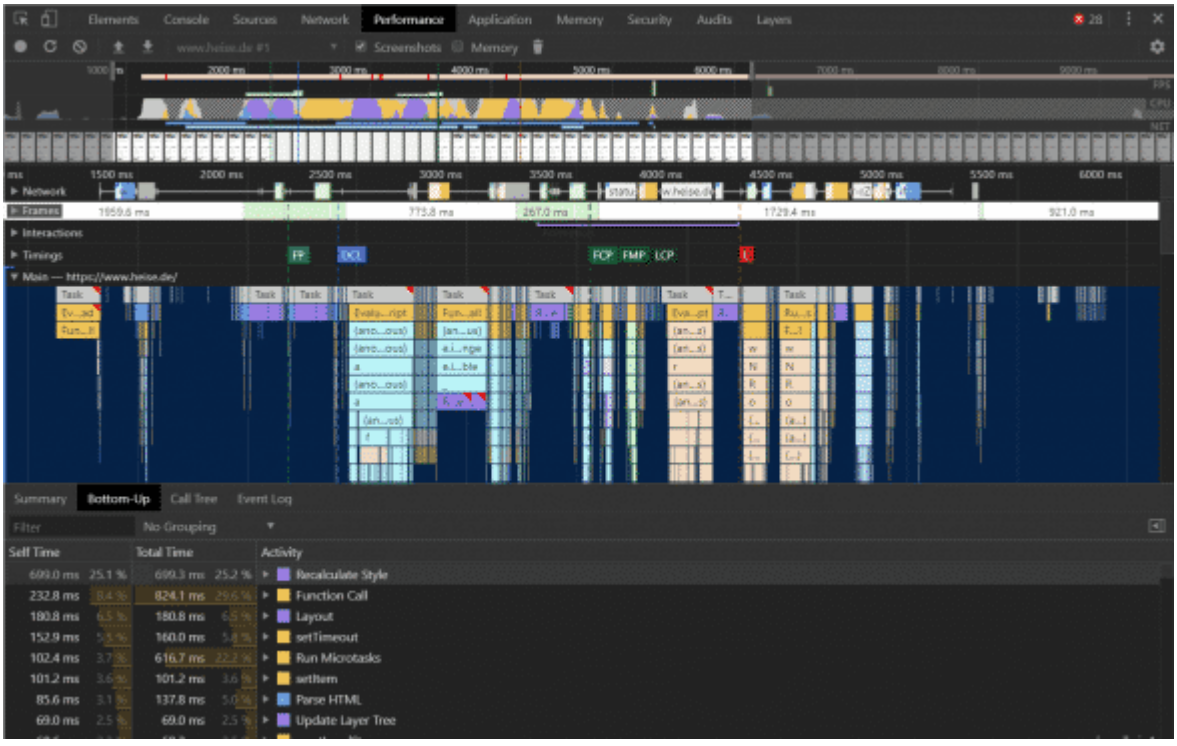
Weniger bekannt ist, dass auch Stylesheets nicht im <head>-Bereich stehen müssen. Sie können beispielsweise das CSS unterhalb des Browserfensters nachladen oder es komponentenweise aufteilen. Wenn Sie Stile für Media-Querys mit <link href="[URL]" rel="stylesheet" media="[Media-Query]"> anfordern, lädt der Browser sie nur herunter, falls er sie braucht. Das Tool Critical extrahiert die sofort benötigten Stile aus dem Stylesheet und fügt sie inline ins HTML-Dokument ein.

Dieses „Code-Splitting“ widerspricht der obigen Forderung nach möglichst großen Datenpaketen. Sie können diesen Widerspruch durch Abwägen und Messen auflösen – oder durch den Umstieg auf HTTP/2. Die technische Seite des Code-Splittings übernehmen gängige Bundler wie webpack, Rollup oder Parcel.js.

HTTP/2-Server-Push ist ein nettes Feature, aber die Frontend-seitigen Möglichkeiten sind flexibler und schicken nicht stumpf Daten durch die Leitung, die der Browser längst im Cache hat. In JavaScript laden Sie mit XMLHttpRequest oder fetch() Dateien, in HTML nutzen Sie das Tag <link href="[URL]" rel="[Typ]">, das besonders differenzierte Optionen bietet.

So spart der Typ dns-prefetch die Zeit für den DNS-Lookup, während preconnect zusätzlich TCP-Verbindung und Verschlüsselung erledigt. preload und prefetch laden eine Datei, allerdings für unterschiedliche Zwecke: prefetch hat niedrige Priorität und eignet sich für noch zu besuchende Seiten, preload dagegen – verpflichtend mit einem as-Attribut, zum Beispiel as="script" – lädt schneller und ist für die aktuelle Seite gedacht. prerender hat den Effekt, als würde man eine Seite im Hintergrund-Tab laden. Aber so mächtig diese Werkzeuge sind: Wie beim PWA-Cache ist Zurückhaltung gegenüber den Ressourcen des Nutzers angezeigt.

Level 4: Code-Feinschliff



Die Performance-Analysewerkzeuge der Browser machen Unmengen von Daten zugänglich, die sich aber erst nach längerer Beschäftigung mit dem Thema erschließen.

Das Laden ist der engste Flaschenhals im Web, deshalb kommt diesem Bereich bei der Performance-Optimierung ein besonderer Stellenwert zu – aber nach dem initialen Laden des HTML und der Render-blockenden Ressourcen muss der Browser binnen Sekundenbruchteilen ein paar Textdateien in Bildschirmpixel verwandeln. Diese Schwerstarbeit heißt „Critical Rendering Path“.

Dahinter verbergen sich mehrere Aufgaben. Der Browser wandelt HTML und CSS in Baumstrukturen (DOM und CSSOM) und führt beide im Rendering-Baum zusammen. Nun wühlt er sich durch alle DOM-Knoten und errechnet für jeden das Layout, also Größe und Position der Inhaltsboxen. In der Paint- oder Raster-Phase füllt der Browser diese Boxen mit Pixeln und ermittelt schließlich in der Compositing-Phase die Anordnung.

Eine offensichtliche Performance-Optimierung ist also, den Arbeitsaufwand überschaubar zu halten, indem man die Zahl der DOM-Knoten drosselt; Lighthouse meckert bei 1500 Elementen.

Der Umfang des CSS ist (abgesehen vom Laden) weniger problematisch, da sich dieses simple Format sehr effektiv verarbeiten lässt. Auch zusammengesetzte CSS-Selektoren (wie nav li:first-child a) ändern daran nichts: Zwar hält sich hartnäckig die Legende, dass diese die Performance beeinträchtigen, aber die Effekte bewegen sich knapp an der Messbarkeitsgrenze.

Eine spürbare Bremswirkung hingegen haben „Reflows“ in umfangreichen Dokumenten – so nennt man es, wenn bereits gerenderte Elemente erneut die Phasen Layout, Paint und Compositing durchlaufen müssen.

In der Praxis passiert dies oft durch nachgeladene Inhalte, zum Beispiel Bilder ohne vorher bekannte Dimensionen oder Webfonts, die nach dem initialen Rendern zur Verfügung stehen. Die dadurch ausgelösten Größenänderungen können eine ganze Kaskade von Reflows hinter sich herziehen. Auch CSS-

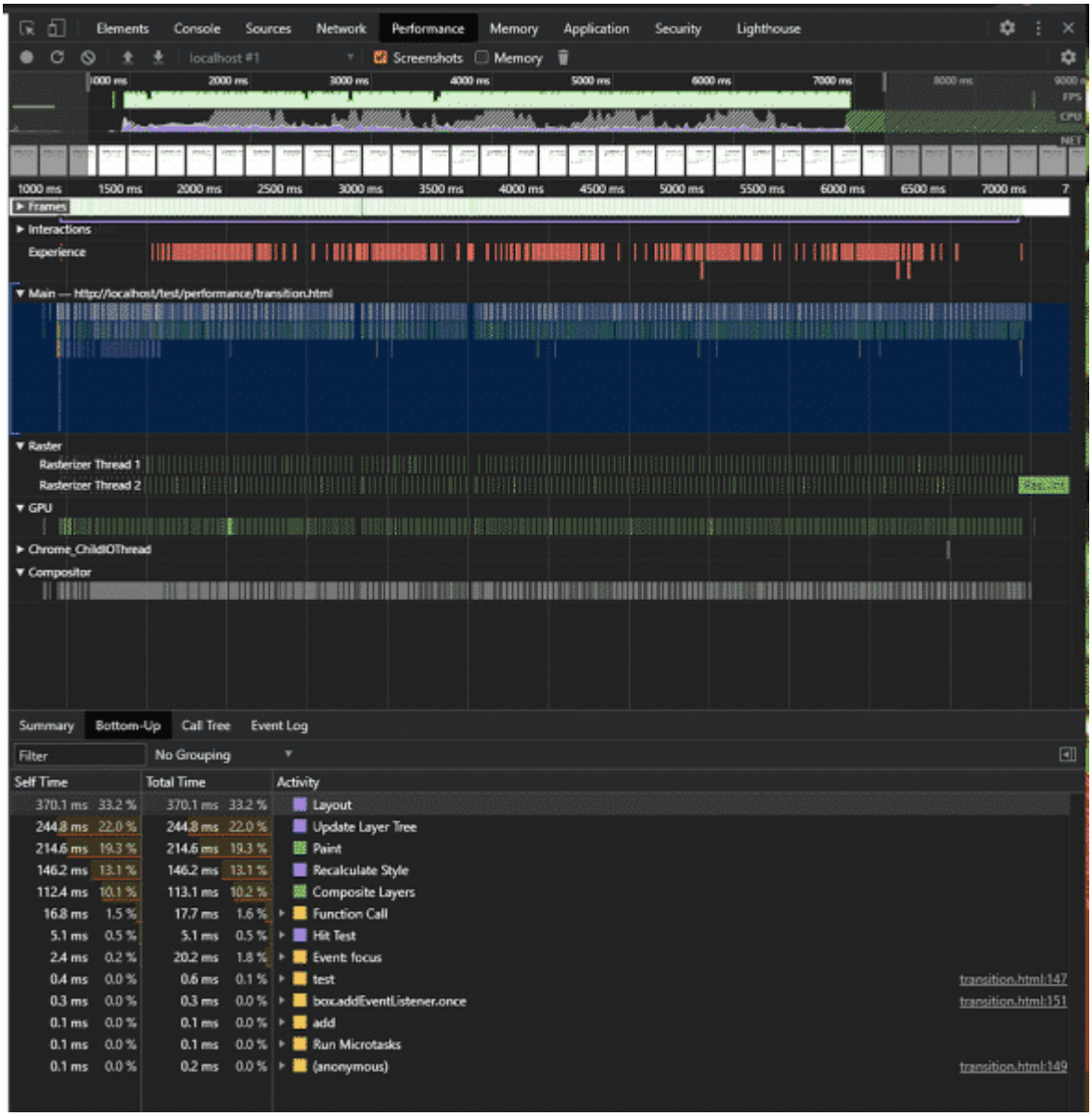
Animationen und -Übergänge sowie JavaScript-Aktionen können das verursachen.

Je nach Art der Änderung und Intelligenz des Browsers muss es nicht immer ein komplettes „Reflow“ sein. Um etwa ein Element animiert zu vergrößern und zu verschieben, bieten sich die CSS-Eigenschaften top, left, width und height an. Wo es möglich ist, sollten Sie dafür jedoch die transform-Eigenschaft mit den Funktionen translate() und scale() verwenden. Die meisten Browser überspringen dann Layout und Paint und gehen gleich zur Compositing-Phase über: Der Grafikprozessor manipuliert die Pixel des schon gerenderten Elements binnen Millisekunden.

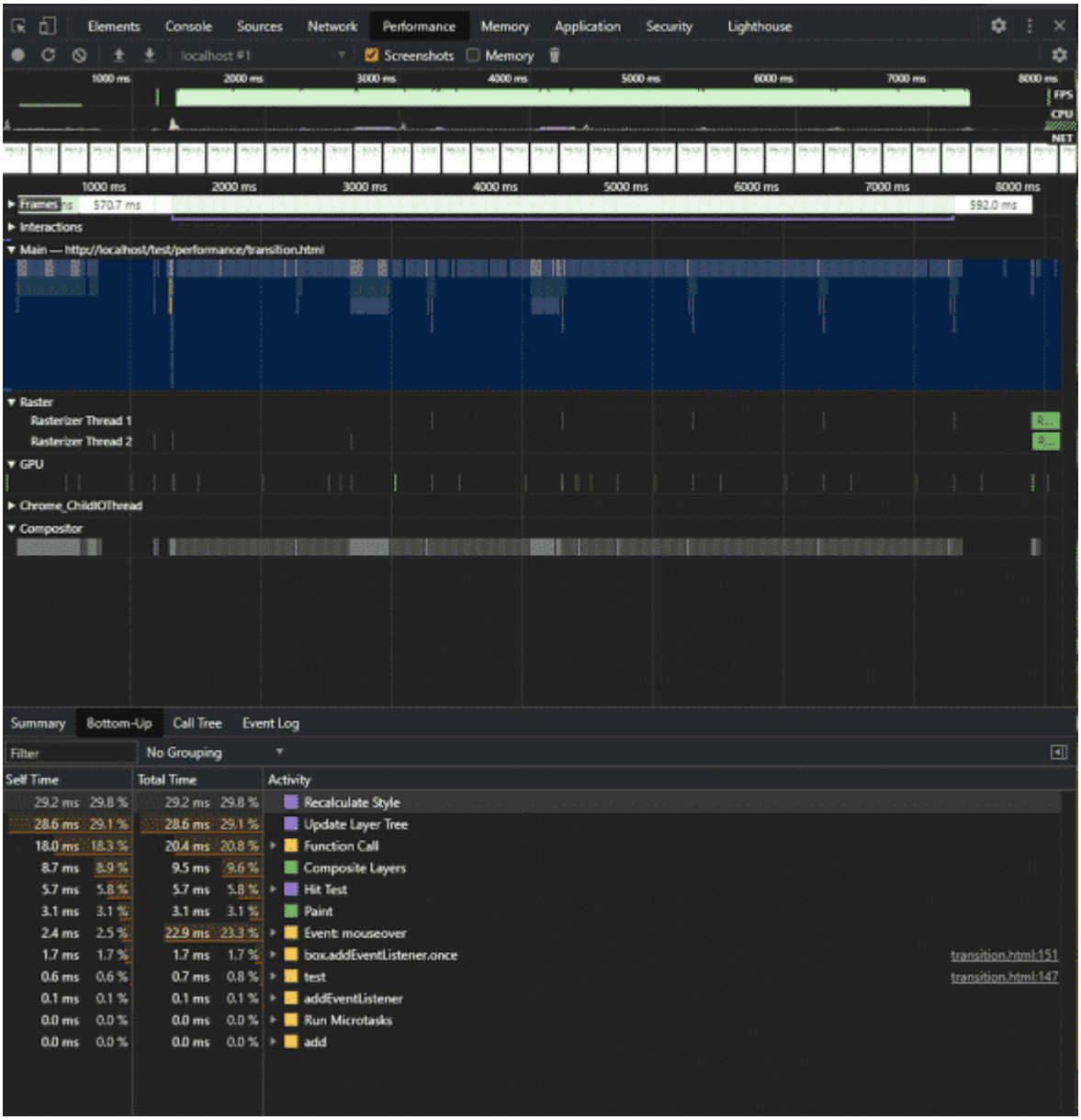
Selbst auf Mobilgeräten laufen derartige Animationen in der Regel flüssig. Wenn Sie Ihren Augen nicht trauen, können Sie die Framerate mit den Entwicklerwerkzeugen messen – in Chrome geht das mit der Kommandopalette (Strg+Umschalt+P) unter „Show frames per second meter“. Maximal erreichbar sind 60 fps.

JavaScript-Code läuft in einem einzelnen Thread. Daher kann eine langwierige Aktion den ganzen Browser zum Stehen bringen. Die Lösung für dieses Problem sind asynchrone Funktionen, was JavaScript in Form von Callbacks, Promises und async/await-Funktionen erlaubt.

Um die Vermeidung unnötiger Wartezeiten geht es auch bei passiven Event-Handlern, die für Scroll- und Touch-Events wichtig sind. Da das Scrolling in einem separaten Thread passiert, kann es auch während aufwendiger Berechnungen flüssig laufen – müsste der Browser nicht vorher prüfen, ob der Code nicht mit preventDefault() das Scrollen stoppt. Mit der Option {passive: true} in addEventListener() verspricht der Entwickler, genau das nicht zu tun.



Eine CSS-Transition, die angrenzenden Text zur Seite schiebt, zwingt den Browser zu harter Arbeit, …



… während ihn eine ähnliche Transition mit Transform-Eigenschaften keine Mühe kostet.
WebWorker können aufwendige Berechnungen in separate Threads auslagern. Das lässt sich gut mit WebAssembly kombinieren, eine auf Performance getrimmte Untermenge von JavaScript, die aus Sprachen wie C++ transpiliert wird. Und schließlich bringt WebGL die Grafikausgabe direkt auf die GPU.

Allerdings braucht man diesen Performance-Turbo außer für Spieleentwicklung nur selten, und für typische Webseiten-Aufgaben nützt er auch nicht viel – denn meistens sind Skripte auf einer Webseite mit DOM-Manipulationen beschäftigt, die mit WebWorkern, WebAssembly und WebGL nicht möglich sind.

Bei DOM-Zugriffen kann es erstaunliche Performance-

Unterschiede geben. Der wohl gängigste Weg, ins Dokument zu schreiben, ist, über die Eigenschaft innerHTML HTML-Quelltext einzufügen:

```
someData.forEach(data => {
  document.querySelector('.my-list').
  innerHTML += `<li>${data}</li>`;
});
```

Umständlicher sind die altmodischen DOM-Methoden wie document.createElement() und appendChild():

```
const list = document.
  querySelector('.my-list');
for (let i = 0;
       i < someData.length; i++) {
  const li = document.
    createElement('li');
  li.textNode = someData[i];
  list.appendChild(li);
}
```

Der Code cacht das Listenelement und hängt neue Elemente erst ins DOM ein, wenn Attribute und Inhalte vollständig sind. Die klassische for-Schleife ist minimal schneller als Array-Methoden wie forEach(). Während Letzteres jedoch wie auch das Caching nur minimale Verbesserungen bringt, beschleunigen die DOM-Methoden das Skript massiv – bei großen Listen bis um das Tausendfache.

Aber wie relevant ist das in der Praxis? „Voreilige Optimierung ist die Wurzel allen Übels“, schrieb Programmiergott Donald Knuth. Tatsächlich wird kaum ein Programmierprojekt am Performance-Unterschied zwischen for und forEach() leiden, während Wartbarkeit und Lesbarkeit vitale Bedeutung haben. Wenn Sie fünf Listenpunkte einfügen, ist es egal, welche Variante Sie wählen. Andererseits häufen sich viele kleine Performance-Sünden an, und das Bewusstsein für -effizienten Code kann entscheiden, ob eine Anwendung benutzbar ist oder nicht.

Gut studieren lässt sich dieser Effekt anhand bekannter Algorithmen, etwa für die Berechnung von Fibonacci-Zahlen – eine Reihe von Zahlen, die aus der Summe der vorherigen zwei gebildet werden (0, 1, 1, 2, 3, 5, 8, …). So könnte man die ersten n Fibonacci-Zahlen wie folgt berechnen:

```
const fib = n => n < 2?
  n : fib(n - 1) + fib(n - 2);
```

Der Algorithmus ruft sich rekursiv selbst auf, um bis zu den ersten Zahlen der Reihe zurückzugehen, die er dann addiert. Simpel, elegant – und extrem ineffizient; irgendwo bei n = 60 wird sich der Browser verabschieden. Der Rechenaufwand steigt mit jeder Iteration exponentiell an, während schlauere Algorithmen das Ergebnis in Sekundenbruchteilen liefern.

Rekursionen und verschachtelte Schleifen können die stärksten CPUs in die Knie zwingen. Wer öfter an komplexen Skripten arbeitet, sollte sich mit der O-Notation („Big O“) vertraut machen, die den Blick für solche Performance-Fallen schärft.

Fazit

Heutige Webanwendungen neigen zum Übergewicht. Aus Frameworks und Bibliotheken kommen Megabytes an oft ungenutztem Code, native Webtechniken wie Buttons, Eingabefelder oder Scrolling werden mit JavaScript nachgebaut. Kann man alles machen, solange die Seite schnell lädt und ruckelfrei läuft – nicht nur auf dem gut ausgerüsteten Entwickler-Laptop, sondern auch auf dem drei Jahre alten Billig-Handy.

Oft sind die naheliegenden Maßnahmen besonders effektiv, aber wer mehr rausholen will, muss tiefer einsteigen – und stößt dabei auf immer mehr Feinheiten beim Laden, Kompilieren und Rendern. JavaScript ist Fluch und Segen zugleich: Es trägt häufig zu Performance-Problemen bei. Funktionen wie Lazy Loading oder Service Worker können aber auch für flüssigeres Surfen sorgen. (*jo@ct.de*)

1. Literatur
2. Jan Mahn, Web-Beschleunigung, Das neue Webprotokoll HTTP/2 in der Praxis, c’t 20/2018, S. 162

Weiterführende Informationen: ct.de/yp4b

[/expand]