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

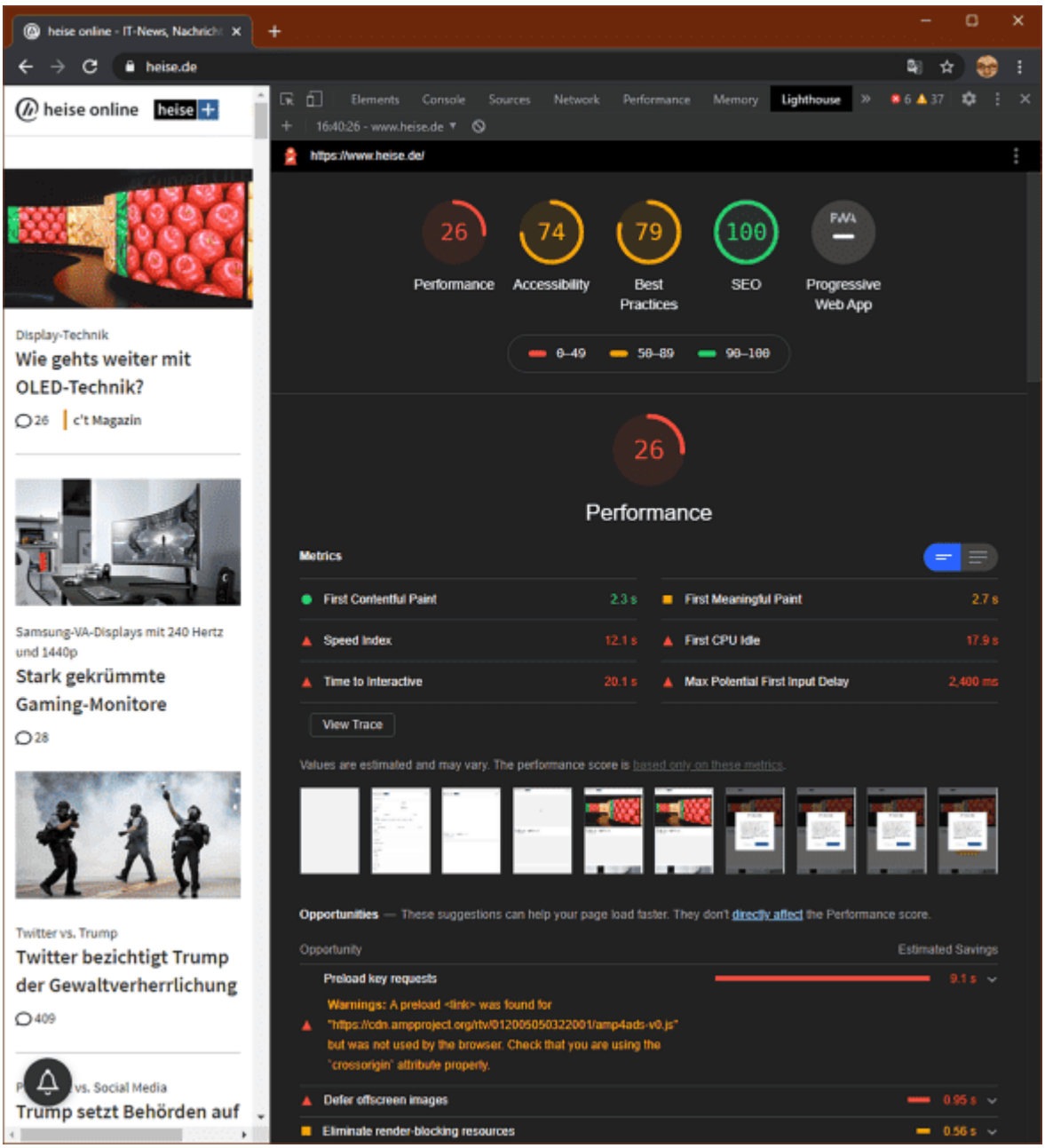
Eine durchschnittliche Webseite wiegt heute zwei MByte, die sich auf 75 HTTP-Requests verteilen (siehe ct.de/yp4b). Fast ein halbes MByte JavaScript-Code hat der Browser dabei zu verdauen. Gleichzeitig sind die Nutzer nicht mehr so geduldig wie zu ISDN-Zeiten: Drei Sekunden leerer Bildschirm sind für 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

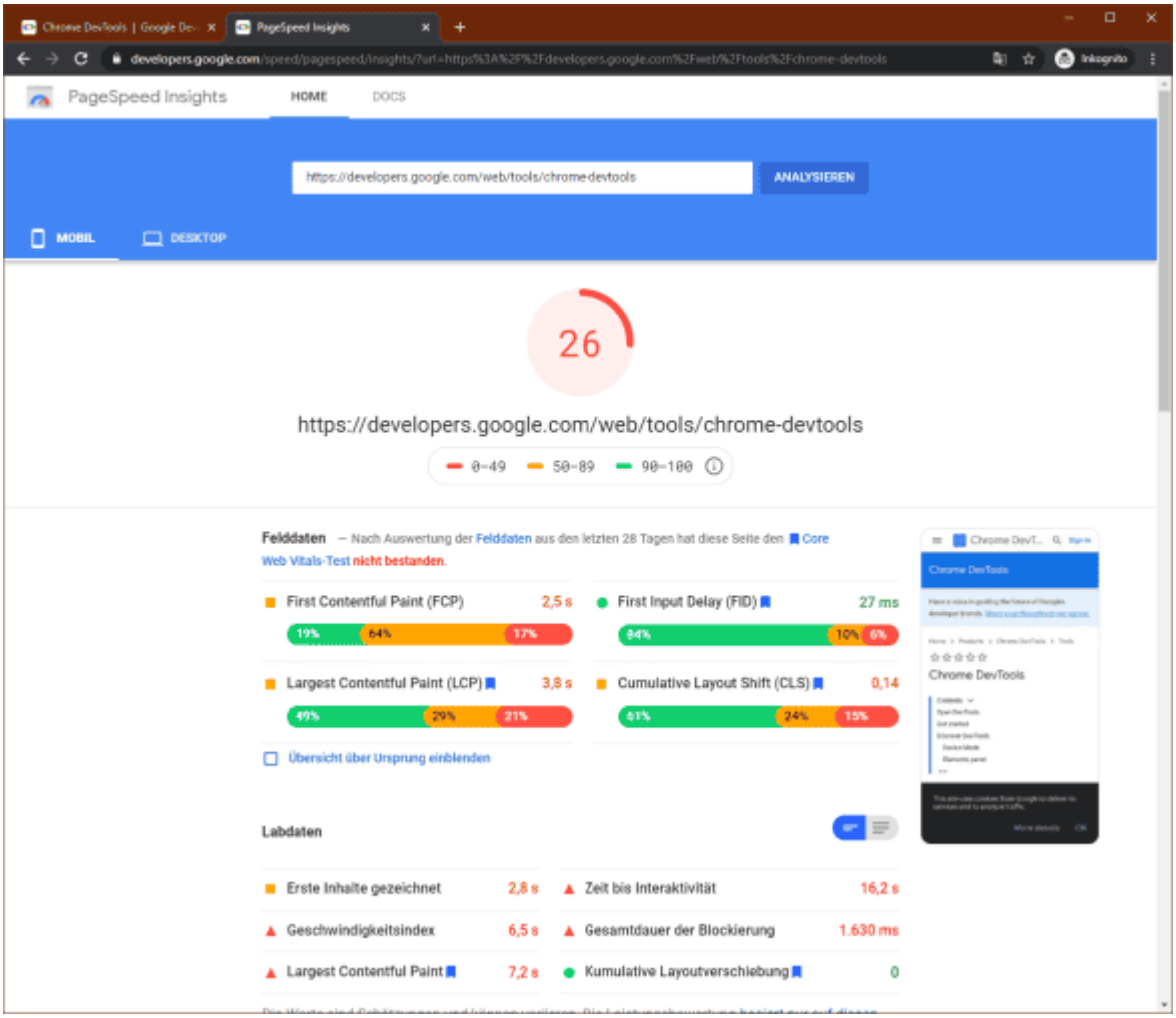
Zunächst gilt es herauszufinden, wo es klemmt. Heute benutzt man für Tests und Tipps meist Google PageSpeed Insights (PSI), Webpagetest.org oder Lighthouse. PSI ist vergleichsweise übersichtlich und eignet sich gut für Einsteiger. Das Open-Source-Projekt Webpagetest.org legt den Fokus mehr auf die Aufbereitung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



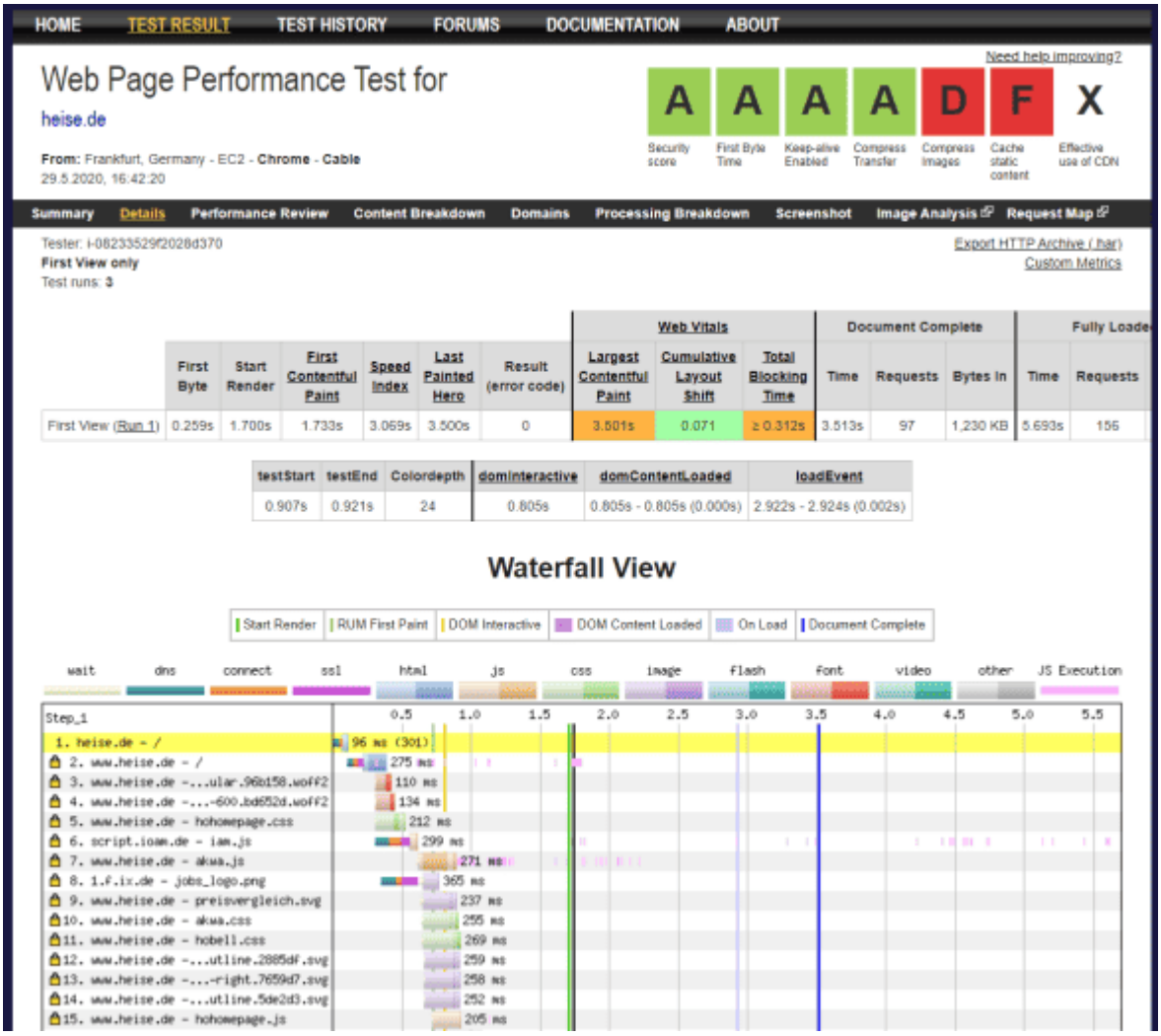
Lighthouse zeigt hübsch gestaltete Messergebnisse und wartet mit konkreten Verbesserungshinweisen auf.

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?

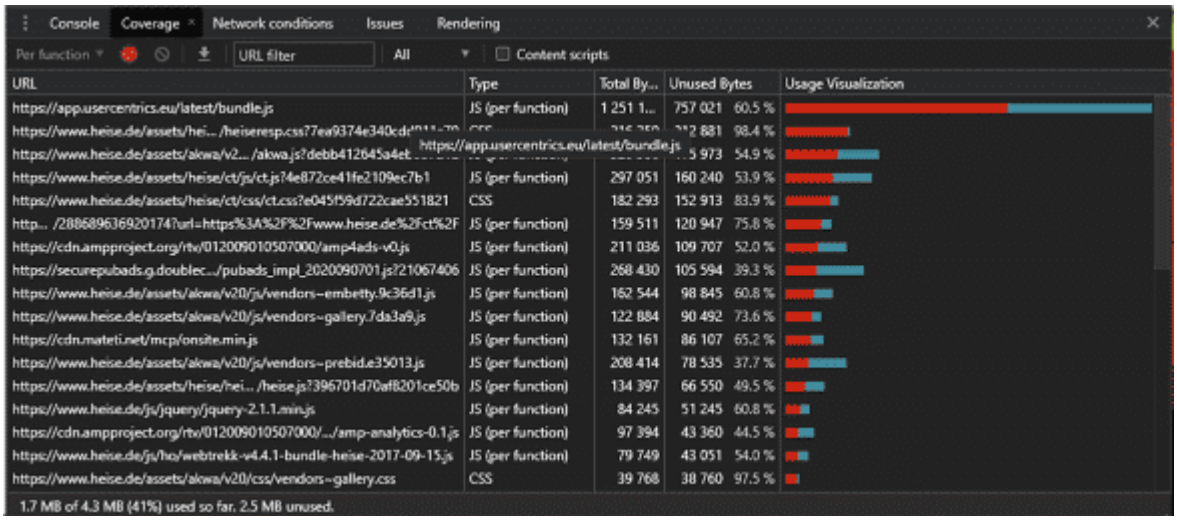
Nicht zu vergessen: Der Browser ist nach dem Download nicht fertig, sondern muss den Code auch noch verarbeiten. Während das etwa bei Bildern eine Frage von Millisekunden ist, leistet er bei JavaScript Schwerarbeit, die den Haupt-Thread oft sekundenlang blockiert. Auf einem leistungsschwachen Gerät kann 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.

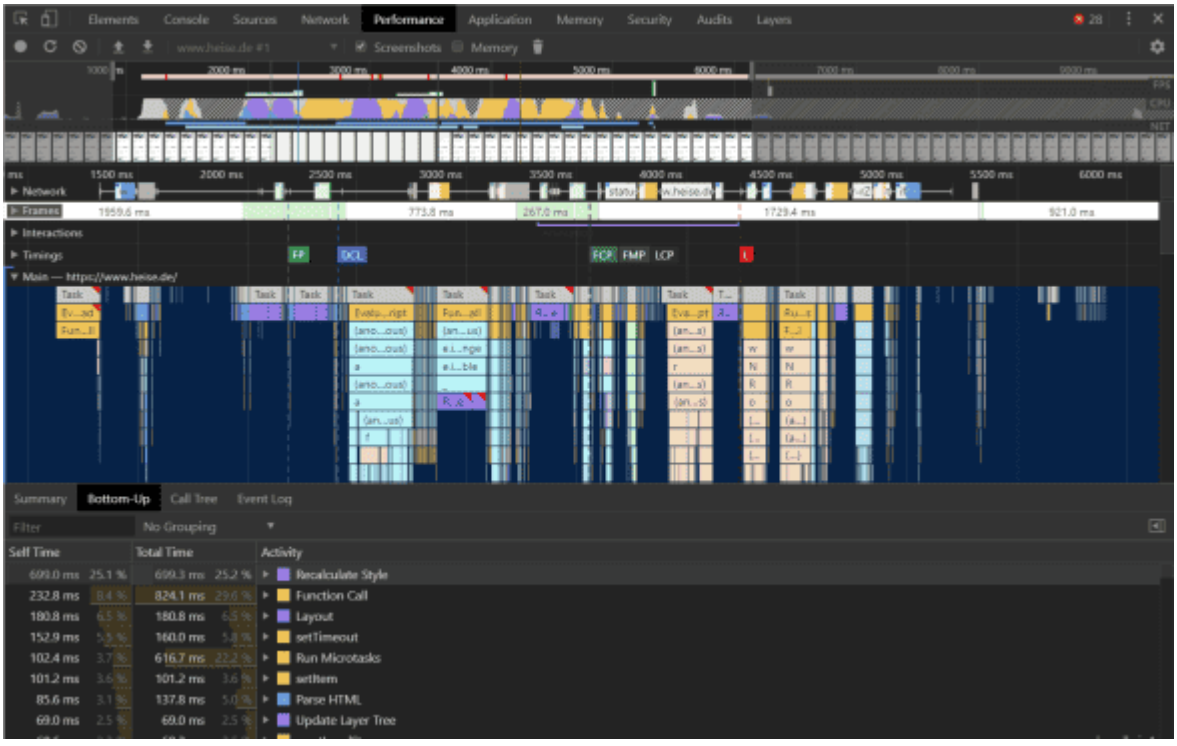
Weniger bekannt ist, dass auch Stylesheets nicht im <head>-Bereich stehen müssen. Sie können beispielsweise das CSS unterhalb des Browserfensters nachladen oder es komponentenweise aufteilen. Wenn Sie Stile für Media-Querys mit <link href="[URL]" rel="stylesheet" media="[Media-Query]"> anfordern, lädt 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-

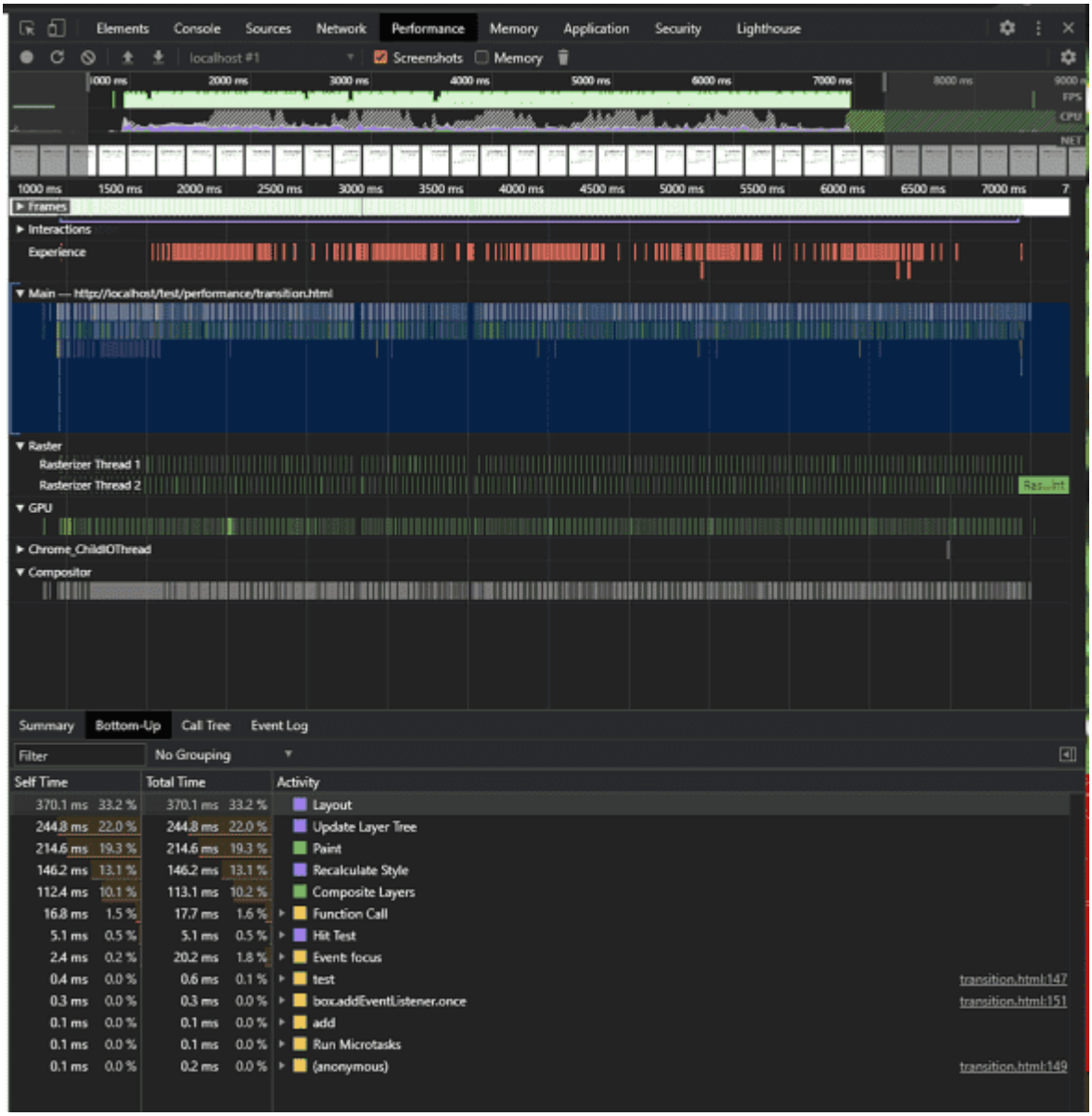
Animationen und -Übergänge sowie JavaScript-Aktionen können das verursachen.

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.

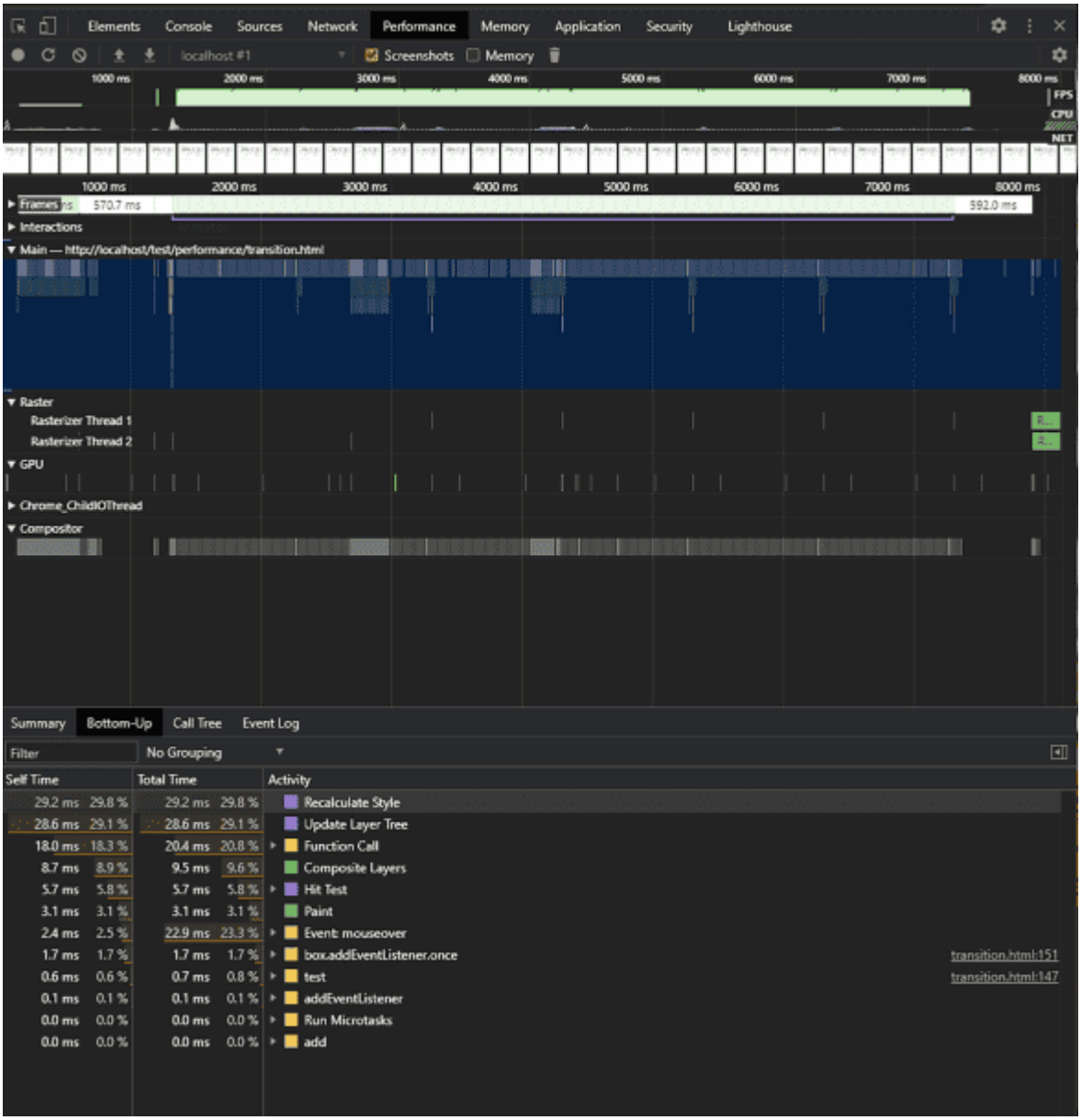
Selbst auf Mobilgeräten laufen derartige Animationen in der Regel flüssig. Wenn Sie Ihren Augen nicht trauen, können Sie die Framerate mit den Entwicklerwerkzeugen messen – in Chrome geht das mit der Kommandopalette (Strg+Umschalt+P) unter „Show frames per second meter“. Maximal erreichbar sind 60 fps.

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]

Webanwendungen beschleunigen mit CQRS

Webanwendungen beschleunigen mit CQRS

[expand title=“mehr lesen…“]

Webanwendungen beschleunigen mit CQRS

Getrennt siegen

Arne Blankerts, Sebastian Heuer

Typische PHP-Anwendungen verbringen einen Großteil ihrer Zeit damit, Daten aus einer Datenbank abzurufen, zu verarbeiten und als HTML auszugeben. Command Query Responsibility Segregation

(CQRS) kann die Zahl der Datenbankabrufe drastisch reduzieren und so die Anwendung erheblich beschleunigen.

iX-TRACT

CQRS (Command Query Responsibility Segregation) trennt lesende und schreibende Zugriffe auf den Datenbestand strikt voneinander. Dadurch lassen sich datenbankbasierte Webanwendungen deutlich beschleunigen.

Der für Schreibzugriffe zuständige Prozess erzeugt bei Änderungen eine neue HTML-Repräsentation des aktuellen Zustands der geänderten Daten.

Bei Lesezugriffen muss die Webanwendung lediglich die vorproduzierten HTML-Schnipsel ausliefern, statt die Daten aus der Datenbank abzufragen und selbst für die Darstellung im Browser aufzubereiten.

Da das zeitaufwendige Generieren der HTML-Repräsentation nicht mehr bei jedem Zugriff stattfindet, sondern nur noch bei Änderungen der Daten, profitieren vor allem Anwendungen mit deutlich mehr Lese- als Schreibzugriffen.

PHP gehört nach wie vor zu den gängigsten Sprachen bei der Webentwicklung. Stand früher besonders die einsteigerfreundliche Lernkurve im Vordergrund, werden inzwischen auch umfangreiche Webanwendungen mit der Skriptsprache entwickelt. Die aktuelle Version 7 brachte PHP eine deutlich optimierte Laufzeitumgebung – Grund genug, sich näher anzusehen, wie sich heutzutage hochperformante Webseiten mit PHP umsetzen lassen.

Anders als Java oder Node.js folgt PHP dem „shared nothing“-Prinzip: Die einzelnen Requests an den Server teilen sich erst einmal keinerlei Daten. Auch wenn moderne PHP-Frameworks komplexe Implementierungen möglich machen, läuft es technisch doch erschreckend häufig auf die folgenden Schritte hinaus: den für die URL zuständigen Controller ermitteln, mit SQL aus

der Datenbank die für das Model notwendigen Daten abfragen, den View mit HTML erzeugen und das Ergebnis ausliefern.

Was hier so einfach klingt, ist in der Praxis natürlich deutlich aufwendiger und wird bei komplexen Datenbankabfragen und steigender Systemlast schnell zum Problem. Auf den ersten Blick wäre dann ein dauerhaft laufender Application Server wie bei Java oder Node.js von Vorteil, der bereits geholte Daten im Speicher hält. Ob und wann geänderte Daten in der Persistenz aktualisiert werden, wäre dann bloß ein Implementierungsdetail.

Problemfall zentrale Datenbank

Der so erzielte Performancegewinn wird jedoch im wahrsten Sinne des Wortes teuer erkauft: Nimmt der Besucherandrang zu, muss man den Server immer weiter aufrüsten. Irgendwann geht es nicht mehr sinnvoll weiter und es bleibt nur eine horizontale Skalierung. War bislang ein einziger Serverprozess für die Datenhaltung verantwortlich, müssen sich jetzt mehrere Prozesse und Maschinen synchronisieren, was entweder eine ausgefeilte Interprozess-Kommunikation oder doch wieder eine zentrale Persistenz etwa in Form einer permanent aktualisierten Datenbank erfordert.

Bei einer derartigen Architektur, egal ob mit PHP oder einer anderen Plattform genutzt, hat selbst eine deutliche Beschleunigung der Laufzeitumgebung nur eine marginale Auswirkung auf die Gesamtperformance: Wenn die Anwendung einen signifikanten Anteil der Zeit mit Warten auf Antworten der Datenbank verbringt, beschleunigt das, nüchtern betrachtet, lediglich das Warten.

Vielleicht liegt die Lösung des Problems also weniger im Beschleunigen der Ausführung als im Vermeiden derselben. Beachtet man zudem, dass die meisten Webseiten deutlich häufiger gelesen als neu geschrieben werden, bietet es sich an, hier zuerst anzusetzen.

Mehr Tempo durch Caching

Wenig überraschend ist die gängige Reaktion auf Performanceprobleme bei Lesezugriffen: die Einführung von Caches. Sie sollen das erneute Generieren einer Antwort unnötig machen, sofern die Anfrage innerhalb eines bestimmten Zeitfensters schon einmal beantwortet wurde.

Naheliegend und technisch am einfachsten umzusetzen ist das Vorschalten eines Reverse Proxy wie Varnish, der die HTML-Ausgabe der Applikation zumeist im Arbeitsspeicher zwischenspeichert und sie bei ähnlichen Requests direkt an den Client zurücksendet, ohne dass die Applikation Arbeit damit hat. Das sorgt zwar für eine sehr schnelle Beantwortung von Requests, für die bereits eine passende Antwort im Cache liegt, bringt aber neue Probleme mit sich.

Daten in einem Cache sind immer potenziell veraltet, da sie der Reverse Proxy lediglich anhand ihres Alters aus dem Cache löschen kann – die Anwendung soll ja gerade nicht in die Beantwortung eingebunden werden. Zudem kommt es schnell zu Inkonsistenzen, wenn verschiedene Ansichten der gleichen Daten zu unterschiedlichen Zeiten abgerufen wurden und so in unterschiedlichen Ständen im Cache zwischengespeichert sind.

Auch die Auslieferung personalisierter Inhalte unter der gleichen URL ist ein Problem, da die HTML-Ausgabe dann eben gerade nicht identisch ist. Um dennoch vom Caching zu profitieren, müsste die Antwort in personalisierte und nicht personalisierte Fragmente zerlegt werden. Die personalisierten Bereiche müssten dabei für jeden eingehenden Request von der Applikation neu geliefert werden.

Varnish bietet mit Edge Side Includes (ESI) eine in diese Richtung gehende Option. Allerdings bringt das einen zusätzlichen Frontend-Layer außerhalb der Applikation mit sich, was zu neuen Problemen führt. Und gerade die Fragmente mit dynamischen Inhalten, deren Erzeugung teuer ist, werden

nicht vom Cache abgedeckt, sodass das Performanceproblem letztlich bestehen bleibt.

Nicht zuletzt erschwert die Abhängigkeit der Anwendung von einem gefüllten Cache auch noch das Deployment neuer Versionen. In der Praxis erweist es sich als sehr kompliziert, zu ermitteln, welche Daten veraltet sind; daher löscht man meist einfach alle Daten aus dem Cache. Das bedeutet allerdings, dass die Anwendung jetzt wieder alle Anfragen bearbeiten muss, was zwangsläufig zu einem Performanceengpass führt.

Caching kann nicht die Antwort sein

Caching lindert also nur Symptome, ohne jedoch die eigentliche Ursache von Performanceproblemen zu adressieren. Die vermeintlich einfache und schnelle Verbesserung der Antwortzeiten wird mit zusätzlicher Komplexität, neuen Ausfallszenarien und der Auslieferung potenziell veralteter Daten erkauft. Es lohnt sich daher, einige Schritte zurückzutreten und erneut einen Blick auf die Softwarearchitektur zu werfen.

Eines der wichtigsten Ziele jeder Website ist die schnellstmögliche Beantwortung von Anfragen. Um das zu erreichen, muss der Server mit der ohnehin schon knappen Ressource Zeit extrem sparsam umgehen. Geht es um Antwortzeiten von 100 Millisekunden und weniger, wird die Luft beim Einsatz gängiger Fullstack-Frameworks auch und gerade im PHP-Umfeld schnell dünn: Die schiere Größe der Codebasis, die zur Laufzeit viele Entscheidungen treffen muss, zusammen mit nur selten benötigter Flexibilität fordert ihren Preis.

Doch auch ein bewusst schlankes, im Zweifel selbst geschriebenes Framework verschenkt noch viel Potenzial. Betrachtet man beispielsweise die Produktseite zu einem Artikel in einem E-Shop, beginnt deren Aufbau meist damit, die Artikelstammdaten – Bilder, Texte, tagesaktuelle Preise –

unter Einsatz eines ORMs wie Doctrine aus der Datenbank zu ziehen und die zugehörigen Models zu instanziieren. Die so vorbereiteten Daten gehen dann an eine Template-Engine zur Umwandlung in HTML.

Aber sind all diese Schritte überhaupt notwendig? Aus Sicht des Browsers hat das empfangene HTML mit dem Model „Produkt“ nichts mehr zu tun. Nimmt man hinzu, dass sich die Artikelstammdaten nicht bei jedem Request ändern, folgt daraus: Das Erzeugen der HTML-Repräsentation muss gar nicht während der Beantwortung der Anfrage geschehen, sondern nur dann, wenn sich die zugrunde liegenden Daten ändern.

CQRS to the Rescue

CQRS (Command Query Responsibility Segregation) beschreibt ein Architekturmuster, das lesende und schreibende Operationen strikt voneinander trennt. Konkret bedeutet das, dass ein generisches Model aufgeteilt wird in eines für die lesende und eines für die schreibende Seite. Aus einer Klasse, die eine API zum Abrufen und eine zum Verändern von Daten anbietet, werden also zwei Klassen mit spezialisierten APIs. Das Read-Model ist dann nur noch eine unveränderbare Repräsentation eines Zustandes, während das Write-Model Methoden zu dessen Veränderung bereitstellt.

Diese explizite Trennung eröffnet spannende Möglichkeiten für die Performanceoptimierung. Da lediglich Schreiboperationen den Zustand verändern können, reicht es, wenn sie die Neugenerierung der Zustandsrepräsentation triggern. Bei Abfragen muss lediglich die bei der letzten Änderung erzeugte Repräsentation ausgeliefert werden.

Interessanterweise findet sich diese konzeptuelle Trennung schon im HTTP-Standard: Der lesende *GET*-Request verändert den Zustand der Applikation nicht, während die schreibenden Request-Methoden wie *DELETE, PATCH, POST* und *PUT* eine Zustandsänderung bewirken. CQRS und das Web scheinen also

bestens zusammenzupassen.

Für einen E-Shop heißt das: Die HTML-Repräsentation einer Produktseite muss nur dann neu generiert werden, wenn beispielsweise neue Artikeldaten vom ERP-System kommen. Daher kann ein von den eingehenden Requests unabhängiger Prozess diese Daten entgegennehmen und neue HTML-Repräsentationen erzeugen. Listen mehrerer Artikel beispielsweise einer Kategorie kann man aus HTML-Schnipseln für die einzelnen Artikel zusammensetzen.

Key-Value-Store statt SQL

Diese Schnipsel müssen nun so persistiert werden, dass sie sich bei der Verarbeitung eines Requests möglichst schnell laden lassen. Dafür eignet sich ein Key-Value-Store, der beliebige Daten unter einem eindeutigen Schlüssel ablegt. Statt eine SQL-Query an die Datenbank zu schicken, lädt die Anwendung die Daten über den Schlüssel, was viel schneller geht.

Der quelloffene Key-Value-Store Redis beispielsweise beantwortet bei geringem Ressourcenverbrauch mühelos mehrere Tausend Requests pro Sekunde. Dass der Key-Value-Store die generierten Daten dabei nicht wie eine relationale Datenbank in normalisierter Form, sondern im Gegenteil mehrfach in verschiedenen Varianten speichert, gehört dabei zum Konzept. Als Datenquelle für die Generierung kann natürlich weiterhin eine relationale Datenbank dienen.

Das Frontend liefert lediglich HTML-Snippets aus, die beim Anlegen und Ändern von Einträgen in der Produktdatenbank generiert werden. Die Suchmaschine ermöglicht beliebige Kombinationen der Schnipsel (Abb. 1).
Erfahrungen aus Projekten der Autoren zeigen, dass auch der Arbeitsspeicherbedarf geringer ausfällt, als man vielleicht vermuten würde. Und wenn der Key-Value-Store in einer sehr

großen Anwendung einmal mehr als ein paar Gigabyte belegt, lässt sich der HTML-Code leicht komprimiert ablegen, was den Speicherbedarf drastisch verringert – um den Preis einer etwas höheren CPU-Belastung durch das Dekomprimieren beim Lesen.

Ein Beispiel aus der Praxis

Im Folgenden zeigen wir die Funktionsweise von CQRS in PHP am Beispiel der E-Commerce-Plattform hinter www.kartenmacherei.de, einem Shop für personalisierbare Print-Produkte wie Einladungskarten oder Fotokalender. Das Team der kartenmacherei hat 2016 ein komplett neues Shop-Frontend vor eine existierende Standardsoftware gesetzt, das konsequent den Grundsätzen von CQRS folgt. Die Anwendung auf Grundlage dieser Architektur erreicht ohne vorgeschaltetes Caching Antwortzeiten von unter 35 Millisekunden und ermöglicht eine problemlose horizontale Skalierung.

Die Applikation ist in mehrere Komponenten aufgeteilt. Das in PHP geschriebene Frontend ist für die Beantwortung von Requests zuständig und soll daher möglichst wenig zu tun haben. Bei Lesezugriffen ermittelt es anhand des Request-URI die XHTML-Snippets, die zum Aufbau der Seite erforderlich sind, und lädt sie aus dem Key-Value-Store. Anschließend werden sie nach einer vorgegebenen Logik in ein XHTML-Template eingefügt und als HTML an den Client geschickt.

Eine Kategorienseite ist aus fertigen HTML-Snippets zusammengesetzt (Abb. 2).

Listing 1: Kategorienseite

```
<?php
class CategoryPage {
  private $template;
  private $searchEngine;
  private $snippetStore;
```

```
   public function __construct(SearchEngine $searchEngine,
SnippetStore $snippetStore,  PageTemplate $template) {
    $this->searchEngine = $searchEngine;
    $this->snippetStore = $snippetStore;
    $this->template = $template;
  }

   public function asHtml(CategoryName $categoryName,
FilterCollection $filters): HTML {
                                    $productSkus        =
$this->searchEngine->getProductsInCategory($categoryName,
$filters);
                                       $snippets        =
$this->snippetStore->getListTiles($productSkus);
     $template = $this->template->inject('category-items',
$snippets);
    return $template->asHtml();
  }
}
```

Die zu ladenden Produkt-Snippets beispielsweise für eine Kategorieseite im Shop ermittelt eine Suchmaschine. Bei kartenmacherei.de ist das Elasticsearch, eine Alternative wäre Apache Solr. Beide Open-Source-Projekte basieren auf der Lucene-Bibliothek und beantworten Anfragen auch unter hoher Last innerhalb weniger Millisekunden. Elasticsearch gibt für eine Kategorie unter Berücksichtigung von Filterkriterien („nur die Farben Rot und Pink“) eine Liste von Artikelnummern zurück, mit denen das Frontend die HTML-Snippets aus dem Key-Value-Store lädt. Listing 1 zeigt, wie wenig Code im Frontend dafür nötig ist.

Es mag so klingen, als sei der Key-Value-Store nur ein besserer Cache, doch es gibt fundamentale Unterschiede. Aus Sicht des Frontend ist der Store als primäre Datenquelle an die Stelle der zuvor genutzten relationalen Datenbank getreten. Fehlt dort ein Eintrag, verhält sich die Anwendung nicht anders, als wenn ein Eintrag in einer MySQL-Tabelle fehlt. Somit haben die Daten im Key-Value-Store auch keine TTL, da sie per Definition immer den aktuellen Datenstand

repräsentieren. Dass eine separate Komponente diese Daten von Zeit zu Zeit durch neuere Versionen ersetzt, weiß das Frontend nicht. Auch das bei Caches übliche Verdrängen älterer Einträge durch neuere gibt es nicht: Geht dem Key-Value-Store der Arbeitsspeicher aus, quittiert er schlichtweg den Dienst oder muss sich mit Swappen behelfen.

Um das Frontend möglichst einfach zu halten, werden schreibende Requests aufgrund von Benutzeraktionen im Frontend an Microservices delegiert. Diese kapseln die gesamte Geschäftslogik und sind unter anderem auch für die inhaltliche Validierung zuständig. So prüft der Warenkorb-Service beispielsweise, ob ein Artikel überhaupt verfügbar ist, bevor er im Warenkorb landet. Diese Zustandsänderungen werden in der Regel synchron ausgeführt, das Frontend wartet also, bis der Service die Aufgabe erledigt hat. Da Nutzer bei solchen Aktionen eine gewisse Verarbeitungszeit erwarten, sind hier etwas längere Antwortzeiten vertretbar. Tatsächlich gibt es Fälle, in denen eine zu schnelle Beantwortung einer abgesendeten Bestellung zu der Annahme führte, etwas habe nicht funktioniert.

Das Backend erzeugt die HTML-Snippets

Listing 2: Einfügen eines neuen Produkts

```
class ProductSnippetRenderer {
  private $snippetStore;

  public function __construct(SnippetStore $snippetStore)  {
    $this->snippetStore = $snippetStore;
  }

  public function render(Product $product) {
    $this->snippetStore->startTransaction();
$this->snippetStore->addProductListSnippet($this->renderProduc
tListSnippet($product));
$this->snippetStore->addProductDetailPageSnippet($this->render
ProductDetailPageSnippet($product));
```

```
$this->snippetStore->addCartItemSnippet($this->renderCartItemS
nippet($product));
    // (...)
    $this->snippetStore->commit();
  }
}
```

Listing 3: Schnittstelle zu Elasticsearch

```
class ElasticsearchProductSearchIndexer {
  private $elasticsearchClient;

    public function __construct(ElasticsearchClient
$elasticsearchClient) {
      $this->elasticsearchClient = $elasticsearchClient;
  }

  public function index(Product $product) {
$this->elasticsearchClient->index($this->mapProductToElasticSe
archDocument($product));
  }
}
```

Das ebenfalls in PHP geschriebene Backend befüllt den Key-Value-Store mit den vom Frontend benötigten HTML-Snippets und sonstigen Datenstrukturen (Listing 2). Dazu nimmt es neue Produktdaten entgegen (Push) oder fragt regelmäßig nach Änderungen etwa im ERP-System (Pull). Zusätzlich wird die Suchmaschine mit neuen Daten versorgt: Neue Artikeldaten gehen als JSON-Objekte an die REST-API von Elasticsearch (Listing 3).

Die Idee der vorgenerierten HTML-Snippets lässt sich auf die Pflege von Content in einem CMS übertragen: Beim Veröffentlichen von Inhalten generiert das CMS als Backend die passenden Snippets und schreibt sie in den zentralen Key-Value-Store. Das Frontend muss nur noch wissen, unter welcher URL welche Snippets auszuliefern sind. Diese Information kann das CMS als zusätzlichen Eintrag in den Key-Value-Store schreiben.

Dabei muss das CMS gar nicht über das Internet erreichbar sein, da es an der Auslieferung von Seiten nicht beteiligt ist – in Anbetracht der häufigen Sicherheitslücken in den populären Content-Management-Systemen dürfte das manchem Administrator einen ruhigeren Schlaf bescheren. Zudem lässt sich das CMS als interne Komponente leichter austauschen: Die einzige Anforderung ist die Fähigkeit, veröffentlichte Seiten oder Seitenbestandteile als (X)HTML zu rendern und inklusive der dazugehörigen URL zu exportieren.

Da Frontend und Backend nicht voneinander abhängig sind, lassen sie sich problemlos getrennt skalieren. So genügt es, bei steigender Besucherzahl einfach zusätzliche Frontend-Instanzen hochzufahren.

Jeder Onlineshop hantiert mit Session-abhängigen Daten wie der Anzahl der Artikel im Warenkorb eines Benutzers oder personalisierten Preisen. Solche Informationen lassen sich nicht ohne Weiteres als fertiges HTML ablegen: Sämtliche HTML-Schnipsel mit allen möglichen Preisen für jeden existierenden Benutzer zu rendern, ist nicht praktikabel. Daher kann es notwendig sein, einige Informationen weiterhin während der Verarbeitung eines Requests zu ermitteln. Doch auch hier können optimierte Datenstrukturen, etwa einfache Listen mit sämtlichen Preisen je User, vorbereitet und im Key-Value-Store bereitgestellt werden, um dem Frontend die Arbeit zu erleichtern.

Personalisierung im Shop

Listing 4: Personalisierte Preise

```
class PriceInjector {
  private $priceStore;
  private $template;

  public function __construct(PriceStore $priceStore, Template
$template) {
    $this->priceStore = $priceStore;
```

```
    $this->template = $template;
  }

  public function updatePrices(User $user) {
    if (!$user->hasCustomPrices()) {
      return;
    }
                                  $updatedTemplate       =
$this->template->inject($this->priceStore->getUserPrices($user
));
    return $template;
  }
}
```

Personalisierte Preise kann das Frontend nach dem Laden der HTML-Snippets abfragen und mittels DOM-Operationen anstelle der Standardpreise in die Snippets einsetzen. Der Aufwand dafür ist minimal und fällt bei einer Performancemessung kaum ins Gewicht. Da das HTML-Snippet die Standardpreise enthält, ist die zusätzliche Arbeit nur erforderlich, wenn der Benutzer eingeloggt ist und abweichende Preise haben kann. Darüber hinaus kann es keine fehlerhafte Artikeldarstellung geben, die wegen einer fehlgeschlagenen Ersetzung keinen Preis enthält: Im Zweifelsfall bleibt einfach der Standardpreis stehen (Listing 4).

Auch bei sich schnell ändernden Daten wie Lagerbeständen kann es sinnvoll sein, diese dynamisch in die Snippets zu injizieren, wenn das Backend ansonsten die Snippets zu häufig neu generieren müsste. Hier ist eine Abwägung auf Basis der spezifischen Eigenheiten der Anwendung nötig.

Fazit

Die hier vorgestellte Softwarearchitektur ist nicht auf den Einsatz von PHP beschränkt, sondern mit jeder Sprache umsetzbar. Sogar die Verwendung eines Key-Value-Store wie Redis ist im Grunde ein Implementierungsdetail. PHP mit seinem „shared nothing“-Prinzip eignet sich jedoch besonders gut für

die Trennung der Verarbeitung von Requests und Änderungen an den zugrunde liegenden Daten nach dem CQRS-Ansatz.

Patterns wie CQRS sind dann besonders vorteilhaft, wenn ein starkes Ungleichgewicht zwischen lesenden und schreibenden Operationen herrscht, da sich jede der beiden Seiten unabhängig von der anderen optimieren lässt. In typischen Webanwendungen gilt es, die Performance bei Lesezugriffen zu verbessern, während schreibende Requests deutlich seltener vorkommen und nicht so zeitkritisch sind. Zustandsrepräsentationen vorzugenerieren, um das Frontend zu entlasten, ist keine neue Idee: Konzerne wie Facebook oder Twitter nutzen dieses Prinzip schon lange, um User Generated Content an Millionen Nutzer in aller Welt ausliefern zu können.

Die Anwendung von CQRS ist in der Regel mit einem Umdenken verbunden, das einige Zeit brauchen kann. Auch einige gewohnte Entwicklungs-Workflows ändern sich. So werden Änderungen an Template-Dateien erst nach dem Neugenerieren der Snippets im Frontend sichtbar. Dafür erhält man eine modular skalierbare, hochperformante Applikation, die durch eine konsequente Aufteilung in verschiedene Komponenten auch langfristig beherrsch- und wartbar bleibt. (odi) Arne Blankerts ist Mitbegründer und Principal Consultant der thePHP.cc Consulting Company. Er berät kleine und große Unternehmen unter anderem bei Fragen zum Aufbau und Betrieb von hochperformanten PHP-Umgebungen. Sebastian Heuer ist Developer Advocate bei der kartenmacherei GmbH. Zusätzlich unterstützt er als freiberuflicher Consultant Teams bei der Entwicklung langlebiger und wartbarer Software.

[/expand]