Performantere Webapplikationen entwickeln

Performantere Webapplikationen entwickeln

[expand title="mehr lesen…"]

Performantere Webapplikationen entwickeln

Richtig schnell

Sebastian Springer

Egal, ob klassische Website oder moderne Single-Page-App: In Zeiten schnellen Internets sinkt die Bereitschaft der Anwender, auf Webseiten zu warten. Wer zufriedene Nutzer (und gute Rankings bei Google) haben will, ist daher gut beraten, Ladezeiten zu minimieren und seinen JavaScript-Code auf Trab zu bringen.

iX-TRACT

Trotz guter Inhalte haben viele Websites ein Problem: Die Benutzer müssen zu lange auf die erste Interaktionsmöglichkeit warten.

Das kleine Einmaleins der Performanceoptimierung: erkennen, verstehen und beheben. Dabei helfen die Analysewerkzeuge der Browser.

Die aktuellen, hoch optimierten JavaScript-Engines bieten gerade bei Single-Page-Applikationen großes Potenzial für gute Performance.

Haben Besucher einer Seite früher noch Wartezeiten von einer halben Minute mit der Erklärung „das Internet ist langsam“ verziehen, springen sie mittlerweile schon nach wenigen Sekunden ab. Untersuchungen zeigen: Je länger die Ladezeit einer Seite, desto geringer die Konversionsrate bei E-Commerce-Sites, die Zahl der weiteren Klicks auf der Site, die Zufriedenheit der Besucher und die Wahrscheinlichkeit, dass sie zurückkommen [a]. Auch in das Google-Rating geht die Ladezeit mit ein.

Wenig Geduld mit lahmen Sites ist verstärkt bei Webapplikationen und Websites für Mobilgeräte zu beobachten. Hier erwarten die Benutzer vor allem ein flüssiges und responsives Bedienkonzept.

Will man die Performance einer Webanwendung verbessern, muss man die genutzte Technik verstehen, über Analysewerkzeuge verfügen und wissen, wie man Performanceprobleme behebt. Auch wenn Web-Apps und Websites unterschiedliche Ansätze verfolgen können, die Grundlagen sind doch immer die gleichen: Webserver und Browser kommunizieren über HTTP. Das Frontend ist mit HTML strukturiert, Styling und Animationen erfolgen mit CSS und die Applikationslogik im Frontend ist in JavaScript implementiert. Je nach den Anforderungen an die Applikation muss man in unterschiedlichen Bereichen optimieren. Außerdem gibt es einige Best Practices, die Entwickler kennen sollten.

Die Entwicklerwerkzeuge der Browser helfen dabei, Webapplikationen und -seiten zu analysieren und potenzielle Schwachstellen zu finden. Dieser Artikel konzentriert sich auf die Chrome Developer Tools. Die vorgestellten Funktionen finden sich unter etwas anderer Bezeichnung aber auch in den anderen Browsern.

Der Lebenszyklus einer Webseite

Nach Eingabe der URL in die Adresszeile des Browsers versucht dieser, die angegebene Datei und daraus referenzierte Ressourcen (HTML-, CSS-, JavaScript- und Mediendateien) herunterzuladen, zu verarbeiten und ihren Inhalt darzustellen. Die Gesamtheit der Aktionen, die der Browser bis zur initialen Darstellung der Applikation unternimmt, heißt Critical Rendering Path.

Die Entwicklertools im Browser bieten zahlreiche Werkzeuge zur Analyse der Performance von Webanwendungen. Eine lange Wartezeit bis zum Beginn der Datenübertragung deutet auf ein Problem beim Server hin (Abb. 1).
Der Download der benötigten Ressourcen ist das erste Hindernis auf dem Weg zu einer performanten Applikation. Der Netzwerk-Tab in den Entwicklertools zeigt eine Übersicht über alle Downloads der aktuellen Webseite. Die initial angefragte HTML-Seite kann auf weitere Ressourcen verweisen, die selbst weitere Ressourcen anfordern können – das führt zu einer Baumstruktur von Downloads mit entsprechenden Abhängigkeiten. So kann ein Stylesheet mit *@import* zusätzliche Stylesheets referenzieren (Abbildung 1).

JavaScript-Code kann weitere Downloads initiieren, wenn er beispielsweise ein neues Image-Element in den DOM-Baum einfügt oder ein *script*-Tag für eine weitere JavaScript-Datei erzeugt. Eine weitere, nicht zu unterschätzende Quelle von Downloads sind Mediendateien, Werbung und Tracking. Alle diese Elemente erzeugen zusätzliche Anfragen beim Server und können ein schnelles Laden der Seite verhindern.

In der Liste der Downloads sieht man neben der angeforderten Ressource und dem Initiator der Verbindung auch den Statuscode des Servers. 200 bedeutet, dass der Server die Anfrage ordnungsgemäß verarbeitet und eine Antwort geschickt hat. Ressourcen mit dem Statuscode 304 konnte der Browser aus

seinem Cache laden, sodass sie keine Netzwerklast erzeugt haben. Ebenfalls relevant ist die Anzahl der übermittelten Bytes von Header und Inhalt der Antwort.

Weniger Daten sind schneller geladen

Bei der benötigten Zeit für den Download gibt es zwei Kennzahlen: die Gesamtzeit und den Latenzwert. Erstere gibt die Zeitspanne vom Absenden der Anfrage bis zum kompletten Download der Antwort an. Die Latenz oder Antwortzeit umfasst den Verbindungsaufbau und die Wartezeit bis zum Empfang des ersten Bytes der Antwort. Bei Downloads, die viel Zeit benötigen, lohnt ein Blick in die Details. Eine lange Wartezeit deutet auf ein Serverproblem hin, da hier das Erzeugen der Antwort lange gedauert hat.

Einige Best Practices im Bereich Netzwerk verhindern hier bereits Probleme. Natürlich sollte der Server nur die Informationen an den Client senden, die dieser wirklich benötigt. Die Webanwendung darf nicht zu viele Verbindungen zum Server aufbauen, da der Browser nur eine bestimmte Menge an parallelen Verbindungen zulässt. Weitere Anfragen müssen warten, bis wieder eine Verbindung frei wird, sodass sich die Verbindungen gegenseitig ausbremsen.

Die Lösung liegt im Zusammenfassen von Ressourcen. Werden mehrere JavaScript-Dateien zu einer großen Datei verbunden, fällt der Overhead für den mehrfachen Verbindungsaufbau weg. Auch CSS-Dateien lassen sich zusammenfassen. Für Bilder gibt es eine ähnliche Methode: Sprites kombinieren mehrere Bilder zu einem großen Bild und zeigen dann per CSS nur den relevanten Ausschnitt an.

Sowohl für JavaScript als auch für CSS gibt es Werkzeuge wie UglifyJS und CSSmin, die unnötige Leerzeichen und Kommentare entfernen und so die zu übermittelnde Datenmenge reduzieren. JavaScript-Code lässt sich weiter verkleinern, indem beispielsweise Variablennamen verkürzt werden. UglifyJS kann

JavaScript-Dateien so auf die Hälfte bis ein Drittel eindampfen. Bei Bildern lässt sich die Dateigröße reduzieren, indem eine möglichst geringe Auflösung ausgewählt wird. Mit dem *srcset*-Attribut kann man je nach Bildschirmgröße unterschiedliche Auflösungen ausliefern, ohne mit JavaScript basteln zu müssen.

Schließlich sorgt die Aktivierung von gzip-Komprimierung beim Server für eine weitere Reduzierung der Datenmenge. Die Komprimierung auf dem Server und die Dekomprimierung im Browser erzeugen zwar Rechenlast, die Einsparung auf Netzwerkebene ist allerdings in den meisten Fällen deutlich größer.

Der Critical Rendering Path

Um eine Webseite dazustellen, muss der Browser immer die gleichen grundlegenden Schritte abarbeiten:

– HTML-Datei herunterladen und in das DOM umwandeln;

– anhand der Stylesheets das CSSOM erzeugen;

– DOM und CSSOM zum Render Tree kombinieren;

– für jeden sichtbaren Knoten Position und Größe berechnen;

– die Elemente darstellen.

Für eine schnelle Seite muss jeder Abschnitt dieses Critical Rendering Path optimiert werden.

Das Tortendiagramm zeigt, dass der Browser bei dieser Seite die meiste Zeit mit dem Abarbeiten von JavaScript verbringt (Abb. 2).
Der Timeline-Tab der Entwicklertools zeigt die Ereignisse an, die im Browser auftreten. Zur Analyse des Critical Rendering Path startet man eine Aufnahme, lädt seine Webanwendung und beendet die Aufzeichnung, wenn die Seite komplett geladen ist.

Einen ersten Überblick gibt das Tortendiagramm auf dem Summary-Tab unten, das die zeitliche Verteilung der wesentlichen Schritte – Laden des Codes, Ausführen der Skripte, Rendern von DOM und CSSOM, Darstellung der Seite – zusammenfasst (Abbildung 2).

Das DOM bildet die Hierarchie der HTML-Elemente und deren Abhängigkeiten ab; wie lange sein Aufbau gedauert hat, zeigt das Event „Parse HTML“ auf dem Reiter „Event Log“ an. Wenn dabei viel Zeit vergeht, kann dies auf einen großen DOM-Baum hinweisen. Dann stellt sich die Frage, ob zum Start der Applikation bereits alle Knoten nötig sind oder ob diese nachträglich geladen und per JavaScript eingefügt werden können.

Nicht mehr als unbedingt nötig

Für den ersten Kontakt mit einer Seite ist der sichtbare Bereich entscheidend. Dieser muss schnell geladen sein, damit der Benutzer interagieren kann. Nun kann beispielsweise ein Loading-Indicator signalisieren, dass die restlichen Elemente nachgeladen werden. Ziel sollte stets sein, den Benutzer informiert zu halten – und das lässt sich auf keinen Fall mit der weißen Seite erreichen, die der Browser anzeigt, bis der HTML-Code verarbeitet und der DOM-Baum aufgebaut ist.

Auch während des Ladens und Verarbeitens der CSS-Styles zum CSSOM kann der Browser noch nichts anzeigen. In diese Baumstruktur fließen sowohl die definierten Styles als auch die Standard-Styles des Browsers ein, die sogenannten User Agent Styles. Dabei werden zunächst die allgemeinen Styles für ein Element angewandt und diese dann von spezifischeren Styles überschrieben.

Das Parsen des CSS-Quellcodes und der Aufbau des CSSOM tauchen in der Timeline der Browser-Tools zusammengefasst als Ereignis „Recalculate Style“ auf. Auch hier gilt: Je umfangreicher das Stylesheet, desto länger dauert die Verarbeitung – und desto

länger die Wartezeit der Benutzer. Daher sollte das Stylesheet nur die wirklich erforderlichen Angaben enthalten. Außerdem sollte man das Einbinden zusätzlicher Stylesheets über *@import* vermeiden, da das zu zusätzlichen Requests führt. Zeit lässt sich auch durch die Verwendung von Media-Queries und Media-Types sparen, die angeben, für welches Medium welches Stylesheet gilt. Hier lassen sich sowohl Auflösungen als auch Bildschirmausrichtung oder Ausgabetyp wählen.

In Verbindung mit JavaScript entsteht ein weiteres Performanceproblem. *script*-Tags blockieren das Rendering des DOM, bis der Browser dessen Inhalt ausgeführt hat. Das *async*-Attribut im *script*-Tag signalisiert dem Browser, dass DOM-Erzeugung und JavaScript-Code entkoppelt sind, sodass das JavaScript nicht blockiert.

DOM + CSSOM = Render Tree

Aus DOM und CSSOM erzeugt der Browser den Render Tree mit den sichtbaren Knoten. Nicht sichtbare Elemente wie *meta-*, *script*– und *link*-Tags finden sich hier nicht. Außerdem fallen alle Knoten weg, die per CSS ausgeblendet werden. Anhand der Regeln aus dem CSSOM kann der Browser jetzt die Positionen und Größen der Knoten im Render Tree berechnen. Der Aufbau des Render Tree und die Berechnung der Dimensionen ist in der Timeline unter dem Event „Layout“ zusammengefasst. Für die Optimierung des Render Tree gelten die gleichen Regeln wie für DOM und CSSOM: möglichst schlanke HTML- und CSS-Strukturen, die sich auf das beschränken, was zur initialen Darstellung nötig ist.

Schließlich stellt der Browser die Inhalte der Seite für den Benutzer sichtbar dar. Dieser Vorgang triggert das Event „Paint“. Damit ist die initiale Darstellung abgeschlossen, die Applikation befindet sich im nächsten Abschnitt ihres Lebenszyklus.

Damit sich eine Seite möglichst schnell anfühlt, zeigt man zunächst nur so viele Daten an, dass sich der Besucher weiter

mit der Seite beschäftigt. Der Rest lässt sich dann asynchron weiter aufbauen. Wenn beispielsweise am Anfang nur ein Teil des CSS geladen wurde, kann JavaScript-Code jetzt weitere *link*-Tags erzeugen, die zusätzliche Stylesheets nachladen. Sind rechenintensive JavaScript-Funktionen per Callback an das *load*-Event des Browsers gebunden, laufen sie erst, nachdem die Seite inklusive aller weiteren Ressourcen fertig geladen und aufgebaut ist.

Best Practices

Auch die Ausnutzung des Browser-Cache steigert die Performance einer Webseite. Dazu sind serverseitig die korrekten HTTP-Header zu setzen. Häufig reicht es schon, den Cache-Control-Header auf einen passenden Zeitwert zu setzen. Die Feinheiten beim Thema Caching und Webserver-Konfiguration würden allerdings den Umfang dieses Artikels sprengen, daher müssen wir hier auf weiterführende Literatur verweisen [b].

Eine weitere Methode zur Performancesteigerung sind progressive Web-Apps, also Webapplikationen, die mit Progressive Enhancement arbeiten. Progressive Apps starten mit einer sehr einfachen Darstellung, die auch auf Mobilgeräten und älteren Browsern gut funktioniert. Dieses leichtgewichtige Gerüst wird mit asynchron nachgeladenen Inhalten erweitert, die die Fähigkeiten des konkreten Endgeräts möglichst gut ausnutzen. Der Performancegewinn resultiert aus den geringen initialen Dateigrößen, dem Einsatz des Browser-Cache sowie Service Workern [1].

Analyse und Verbesserung zur Laufzeit

Nachdem die Seite im Browser geladen ist, interagieren die Benutzer mit der Oberfläche. Auch hierbei gibt es Optimierungspotenzial – nicht so sehr für traditionelle Multi-Page-Anwendungen, wohl aber für länger laufende Single-Page-Applikationen, die sich ähnlich wie Desktop-Anwendungen anfühlen. Die Navigation zwischen unterschiedlichen Modulen

erfolgt hier, indem lediglich Teile der Anzeige ausgetauscht werden. Das Neuladen kompletter Seiten geschieht nur noch in Ausnahmefällen.

Dadurch lässt sich einiges an Overhead sparen, da der Browser zur Laufzeit nur die Daten laden muss, die dem Benutzer gerade angezeigt werden sollen. Die statischen Bestandteile sind meist bereits geladen; es ist nicht nötig, weitere HTML-, CSS- und JavaScript-Dateien zu laden. Zudem greifen einige Optimierungen der JavaScript-Engine erst, wenn bestimmte Strukturen häufiger verwendet werden. Die folgenden Ausführungen beziehen sich vor allem auf die V8-Engine des Chrome-Browsers. Andere JavaScript-Engines verfügen jedoch über ähnliche Optimierungen.

Listing 1: JavaScript-Optimierung

```
console.time('first');
let arr2 = ['Mary', 'Peter', 'Paul', 'Henry'];
for (let i = 0; i < arr2.length; i++) {
  console.log(arr2[i]);
}
console.timeEnd('first');

console.time('second');
let arr = ['Peter', 'Paul', 'Mary'];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
console.timeEnd('second');
```

Wie deutlich sich diese Optimierung auf die Performance auswirkt, lässt sich an einem sehr einfachen Beispiel demonstrieren. Der Code in Listing 1 iteriert zweimal über ein JavaScript-Array. Die Iteration über das erste Array dauert 1,7 Millisekunden, beim zweiten Array aber nur noch 0,08 Millisekunden.

Hidden Classes beschleunigen den Speicherzugriff

Eine häufige Operation in JavaScript ist der Zugriff auf Eigenschaften und Methoden von Objekten. Dazu muss die JavaScript-Engine den passenden Speicherbereich im Objekt finden. Da diese Adressauflösung relativ kostspielig ist, kommen sogenannte Hidden Classes zum Einsatz. Hidden Classes sind Verzeichnisse für den Speicher eines Objekts und helfen der Engine bei der Lokalisierung von Eigenschaften des Objekts. Listing 2 zeigt, wie die Hidden Classes für ein Objekt funktionieren.

Listing 2: Hidden Classes

```
class User {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}

console.time('first object');
let klaus = new User('Klaus', 'Müller');
console.log(klaus.firstname);
console.timeEnd('first object');
console.time('second object');
let peter = new User('Peter', 'Meier');
console.log(peter.firstname);
console.timeEnd('second object');
```

Wenn der Code eine erste Instanz *klaus* der Klasse *User* anlegt, erzeugt die JavaScript-Engine eine initiale Hidden Class für das *User*-Objekt. Werden die Eigenschaften *firstname* und *lastname* gesetzt, entstehen zwei weitere Hidden Classes, die auf der ersten basieren und den Speicher-Offset dieser Eigenschaft enthalten. Außerdem erhält die initiale Hidden Class Verweise auf die weiteren Hidden Classes. Wenn man jetzt, wie im Beispiel, auf eine Eigenschaft des Objekts

zugreift, findet die Engine mithilfe der Hidden Classes die Speicherstelle wesentlich schneller.

Beim Anlegen des zweiten *User*-Objekts *peter* erkennt die JavaScript-Engine, dass es sich um den gleichen Objekttyp handelt. Sie verwendet die gleichen Hidden Classes wie beim ersten Objekt, sodass keine neuen Hidden Classes mehr erzeugt werden müssen. Der Overhead für das Anlegen der Hidden Class, der beim ersten Objekt anfiel, entfällt also, das Objekt ist schneller angelegt. Entwickler sollten daher für ähnliche Objekte eine gemeinsame Klasse definieren, um der Engine die Optimierung der Applikation zu ermöglichen. Als Nebeneffekt wird so auch der Quellcode besser strukturiert.

Garbage Collector

Die relativ lange Laufzeit von Single-Page-Applikationen ohne Neuladen von Seiten hat allerdings nicht nur Vorteile. Die Objekte der Applikation können – je nachdem, in welchem Gültigkeitsbereich sie sich befinden – mitunter recht lange im Speicher liegen. Ein stetig ansteigender Speicherverbrauch führt dazu, dass die Applikation immer langsamer wird und irgendwann nicht mehr benutzbar ist – der Anwender muss die Browser-Sitzung beenden.

Web-Apps tendieren dazu, mit der Zeit immer mehr Speicher zu belegen. Von Zeit zu Zeit räumt der Garbage Collector nicht mehr benötigte Objekte aus dem Speicher (Abb. 3).
Die Speichernutzung lässt sich ebenfalls mit den Entwicklertools des Browsers überwachen. Einen ersten Überblick liefert die Timeline, die neben den bereits erwähnten Events auch den Speicherverbrauch aufzeichnet und als hellblaue Fläche darstellt (Abbildung 3). Meist wächst sie im zeitlichen Verlauf, bricht aber immer wieder ein.

Dafür ist der Garbage Collector der JavaScript-Engine verantwortlich, der nicht mehr benötigte Objekte – solche, auf

die keine Referenz mehr verweist – aus dem Speicher entfernt. Referenzen lassen sich explizit löschen, beispielsweise mit dem *delete*-Operator, und werden automatisch entfernt, wenn der Gültigkeitsbereich eines Objekts verlassen wird: Eine mit *var* oder *let* in einer Funktion definierte Variable existiert nur innerhalb einer Funktion oder eines Blocks. Sobald die Abarbeitung der Funktion beendet ist, wird der dafür allozierte Speicher wieder freigegeben.

Der Garbage Collector wird immer wieder aktiv; seine Aktivität ist in der Timeline als „Minor GC“ und „Major GC“ verzeichnet. Wie die unterschiedlichen Bezeichnungen nahelegen, gibt es verschiedene Arten der Garbage Collection. Die V8-Engine von Chrome teilt den Speicher in zwei Bereiche auf: einen für kurzlebige und einen für langlebige Objekte. Neue Objekte landen zunächst im Speicherbereich für kurzlebige Objekte, der in einzelne Sektionen unterteilt ist.

Ist eine Sektion voll, verschiebt die JavaScript-Engine die noch verwendeten Objekte in eine neue Sektion und leert die vollgelaufene Sektion. Wird ein Objekt zum zweiten Mal verschoben, verlagert es die Engine in den Speicher für langlebige Objekte. Dieser Vorgang taucht als Minor GC im Event Log auf. Der auch als Scavenge bezeichnete Minor-GC-Durchlauf eignet sich gut für kleine Speicherbereiche, da er zwar sehr schnell ist, aber relativ viel Speicher benötigt: Kurzzeitig werden ja der alte und der neue Speicherbereich belegt.

Der Speicher für langlebige Objekte wird durch einen sogenannten Mark-and-Sweep-Algorithmus aufgeräumt: Noch verwendete Objekte werden markiert, alle Objekte ohne Markierung anschließend gelöscht. Diese Major Garbage Collection passiert wesentlich seltener als die Minor GC, da sie die Ausführung des Skripts kurz unterbricht. Das führt bei JavaScript-Animationen zu unschönem Ruckeln und unruhigem Verlauf, weshalb die Garbage Collection Cycles möglichst in die Idle Time der CPU verlagert werden.

Speicher sparen

Wenn möglich sollten Anwendungen Objekte wiederverwenden, um dem Garbage Collector Arbeit zu ersparen, und Referenzen auf Objekte nur solange halten wie nötig, damit der Speicher schnell wieder freigegeben werden kann. Nicht freigegebener Speicher führt zu einem Memory Leak. Einer der häufigsten Gründe ist die Verwendung von (bewusst oder versehentlich erzeugten) globalen Variablen: Da diese überall in der Applikation erreichbar sind, kann der Garbage Collector ihren Speicher nicht freigeben.

DOM-Knoten sind eine weitere Ursache für Speicherverlust. Normalerweise werden nicht mehr benötigte Knoten gelöscht. Allerdings können Applikationen Referenzen auf DOM-Knoten in Variablen speichern, um sie beispielsweise aus dem DOM auszuhängen, zu bearbeiten und an einer anderen Stelle wieder einzuhängen. Bleiben die Variablen allerdings nach dieser Operation gültig, kann der durch die DOM-Knoten belegte Speicher nicht freigegeben werden. Eine ebenfalls nicht ganz offensichtliche Ursache für Memory Leaks sind verkettete Listen von Referenzen. Listing 3 gibt hierfür ein Beispiel.

Listing 3: Verkettete Listen (HTML)

```
<html>
  <head>
    <script>
      let value;

      function leak() {
        let prevValue = value;
        function debug() {
          if (prevValue) {
            console.log('already hava a value');
          }
        }
        value = {
          val: new Array(10000).join('Memory'),
```

```
          getVal: function () {
            return this.val;
          }
        }
      }

      let count = 0;
      let interval = setInterval(() => {
        leak();
        if (count > 20) {
          clearInterval(interval);
        }
      }, 500);
    </script>
  </head>

  <body>
    <p>A real memory leak!</p>
  </body>
</html>
```

Die globale Variable *value* soll als Zwischenspeicher dienen. Die Funktion *leak()* speichert den aktuellen Inhalt von *value* in der temporären Variablen *prevValue*, auf die wiederum die Hilfsfunktion *debug()* zugreift. Letzteres sorgt dafür, dass der Garbage Collector den Speicher nicht ohne Weiteres freigeben kann. Schließlich wird der globalen Variablen in *leak()* ein neuer Wert zugewiesen.

Wird diese Funktion nun in einer Schleife oder wie im Beispiel über einen Timer aufgerufen, sieht man in der Timeline einen stetigen Anstieg der Speichernutzung. Ohne ein Abbruchkriterium (Variable *count*) würde der Speicher mit der Zeit volllaufen und der Browser abstürzen.

Die Grafik des Speicherverbrauchs ist ein erster Schritt in der Analyse von Memory Leaks. Genauere Informationen bieten die Speicherprofile, die sich im Profiles-Tab anfertigen lassen. Sie zeigen, welche Objekte wie viel Speicherplatz belegen und welche Funktion für die Belegung verantwortlich

ist. Mit diesen Informationen lässt sich der Quellcode so optimieren, dass der Speicher korrekt freigegeben wird.

Performante Animationen

Moderne Webapplikationen nutzen Animationen, um die Aufmerksamkeit des Benutzers an eine bestimmte Stelle zu lenken und ihn bei der Interaktion zu unterstützen. JavaScript-Code kann Bewegungen über Modifikationen des CSSOM erzeugen: Erhöht man den *left*-Wert eines Objekts kontinuierlich, wandert es von links nach rechts über den Bildschirm. Nachteil: Die Rendering-Engine kann solche Animationen nicht optimieren, da sie von außerhalb – mit JavaScript – erzeugt werden. Sie belasten daher die CPU.

Listing 4: CSS-Animation

```
<style type="text/css">
.outer {
  position: relative;
  width: 510px;
  height: 10px;
  border: 1px solid black;
}
.inner {
  transition: all 5s;
  position: absolute;
  top: 0;
  left: 0;
  width: 10px;
  height: 10px;
  background-color: red;
}
.move {
  transform: translateX(500px);
}
</style>

<div id="outer" class="outer">
  <div id="inner" class="inner"></div>
```

```
</div>

<script>
  document.getElementById('outer').onclick =  function () {
    document.getElementById('inner').classList.add ('move');
  };
</script>
```

Wesentlich eleganter und auch performanter ist die Verwendung von CSS-Animationen. Für eine so einfache Animation wie das Verschieben eines Elements reicht eine Transformation aus. Die Angabe *transform: translateX(500px)* sorgt dafür, dass das Element durch die Veränderung seiner x-Koordinate nach rechts verschoben wird. Die Animation wird dabei nicht durch die CPU ausgeführt, sondern auf der Grafikkarte. CSS-Animationen sind daher ruckelfreier und um einiges ressourcenschonender als ihre Gegenstücke in JavaScript. Listing 4 zeigt ein Beispiel für eine solche CSS-Animation.

Komplexere Animationen sollte man allerdings nicht mit Transitionen umsetzen, sondern lieber auf *@keyframes* zurückgreifen. Damit lassen sich wesentlich mehr grafische Aspekte steuern. Das bedeutet allerdings auch mehr an Aufwand beim Programmieren der Animation. Es gibt jedoch Generatoren, die den Quellcode fürs Stylesheet erzeugen, zum Beispiel cssanimate.com.

Reflow und Repaint

Jede Manipulation von DOM und CSSOM mit JavaScript führt dazu, dass der Render Tree angepasst und die Seite teilweise neu dargestellt werden muss. Einzelne Eingriffe in diese Strukturen stellen kein Problem dar. Wenn man jedoch in einer Schleife neue DOM-Knoten einfügt, müssen in jedem Schleifendurchlauf Teile der Seite neu dargestellt werden. Man spricht von Reflow (Neuberechnung des Render Tree sowie der Abmessungen und Position der Knoten) und Repaint (Berechnung der visuellen Eigenschaften und Darstellung der Elemente).

Wie auch der initiale Seitenaufbau tauchen Reflows in der Timeline als Layout-Events auf. Jeder Reflow zieht einen Repaint nach sich. Nicht nur das Einfügen neuer Knoten in das DOM, sondern auch zahlreiche weitere Aktionen wie das Verändern des Inhalts der Seite, das Ändern der Fenstergröße, Scrollen durch den Benutzer und sogar das Auslesen und Schreiben bestimmter CSS-Eigenschaften lösen einen Reflow aus. Unverhältnismäßig viele Layout- und Paint-Events in der Timeline sind ein Hinweis auf Performanceengpässe in einer Anwendung.

Einige Best Practices können die Zahl der Reflows und Repaints reduzieren. Statt CSS-Eigenschaften einzeln zu ändern, fasst man besser mehrere Änderungen in CSS-Klassen zusammen. Die Browser-Engine kann so die Layoutänderungen optimieren – ein Reflow reicht für mehrere Änderungen aus. Für Anpassungen am DOM sollte man den betroffenen Teil des Baums aushängen, die Änderungen abseits des sichtbaren Bereichs durchführen und den veränderten Subbaum in einer Aktion wieder einhängen. So wird lediglich je ein Reflow für das Ein- und Aushängen ausgelöst. Beim Bearbeiten der Styles von Elementen mit JavaScript empfiehlt es sich, diese in Variablen zu speichern, um das wiederholte Auslesen von Werten zu vermeiden.

Fazit

Das Thema Performance in Webapplikationen ist vielfältig, sowohl was die Problemstellungen angeht als auch hinsichtlich der Lösungsansätze und Hilfsmittel. Es empfiehlt sich, gängigen Best Practices zu folgen, von denen einige hier vorgestellt wurden.

Allerdings sollten Entwickler ihre Webanwendung nicht zu früh einseitig auf Performance optimieren: Eine verlässliche Analyse lässt sich erst im Betrieb mit realen Daten durchführen. Eine vorzeitige Optimierung bedeutet möglicherweise viel Aufwand an der falschen Stelle. Es kann auch durchaus in Ordnung sein, bei selten verwendeten Features

nicht die letzte Millisekunde herauszuholen: Letztlich muss man immer Kosten und Nutzen von Verbesserungen abwägen. (odi) Sebastian Springer versucht als Dozent für JavaScript, Sprecher auf zahlreichen Konferenzen und Autor, die Begeisterung für professionelle Entwicklung mit JavaScript zu wecken.

Literatur

- [1] Sebastian Springer; Dienst am Kunden; Progressive Web Apps: Service Worker und mehr; iX 6/2016, S. 120

Richtig schnell

- Caching
- Site Performance For Webmasters

[/expand]