Inhalt

Montag, 19. Januar 2015 14:09

Abstract

Die vorliegende Arbeit beschäftigt sich mit der Performance-Optimierung von Apps, die mit Webtechnologien und auf Basis der Frameworks AngularJS, Ionic und Cordova entwickelt werden. Es wird der Frage nachgegangen, welche Implementierungen unterschiedlicher Anwendungsbestandteile die beste Performance ermöglichen. Ziel ist es eine allgemeine Erkenntnissammlung für die Entwicklung von Apps mit diesen Frameworks zu erstellen. Der Begriff App wird in diesem Zusammenhang nicht für Spiele verwendet, diese Kategorie hat noch einmal ganz andere Anforderungen. Die untersuchten Anwendungsbestandteile wurden im Vorfeld festgelegt. Es werden jeweils verschiedene Implementierungen getestet und hinsichtlich ihrer Performance untersucht. Die Entscheidung für oder gegen eine Implementierung wird auf Basis der Performance oder weiterer Anhaltspunkte wie der gefühlten Performance getroffen.
Im Ergebnis wird deutlich, dass … {TODO}

Motivation

Das Interesse an Apps für Smartphones oder Tablets ist seit einigen Jahren ungebrochen und auch in der Zukunft ist wohl mit zunehmendem Interesse zurechnen. Die Art der App-Fähigen Geräte oder ihrer Betriebssysteme steigt stetig und somit auch die Anforderung die jeweilige App auf möglichst vielen dieser Geräte und Betriebssysteme zu positionieren. Daher stehen Unternehmen bei der Entwicklung von neuen Apps schon heute vor der Frage, welche Technologien sie einsetzen um möglichst effektiv alle Plattformen abzudecken. In diesem Zusammenhang treten Webtechnologien immer mehr in den Fokus. Entwickler können meist auf jahrelange Erfahrung mit Webtechnologien zurückgreifen und im besten Fall alle Plattformen mit einer einzigen App abdecken. Doch trotzdem fällt meistens eine Entscheidung gegen diese Technologien, weil Ihre Performance und Usability oft nicht an eine native Lösung heranreicht. Moderne Web-Frameworks wie AngularJS und Ionic versuchen jedoch diese Lücke zu schließen und legen bei ihrer Umsetzung viel Wert auf eine performante und ressourcensparende Implementierung, um schon heute die Entwicklung konkurrenzfähiger Apps für Smartphone zu ermöglichen. Entwickler die diese Frameworks einsetzen stellen jedoch schnell fest, dass eine gute Performance nicht automatisch gewährleistet ist. Vielmehr müssen sich die Entwickler der leistungsschwächen bewusst sein, entsprechend performanten Code schreiben und die Frameworks richtig einsetzen. Aus diesem Grund werde ich in dieser Arbeit so genannte Best-Practice Lösungen für die Entwicklung von mobilen hybriden Web-Anwendungen mit AngularJS und Ionic erarbeiten. Dabei beschränke ich mich aus Zeitlichen Gründen auf die Android Plattform. Erfahrungen aus vorrangegangenen Projekten haben gezeigt, dass hier ein besonders großer Bedarf an Optimierungen vorhanden ist.

Umgebung -> Frameworks

In diesem Abschnitt wird auf die verwendeten Frameworks und ihre Versionen eingegangen. Die Frameworks können auf drei Ebenen angeordnet werden. Auf der untersten Ebene befindet sich Cordova, als Schnittstelle zum Betriebssystem. Darüber ist AngularJS einzuordnen, also Strukturgeber für die Anwendung. Auf der obersten Ebene letztendlich Ionic. Es stellt die UI-Komponenten bereit, mit denen der Anwender interagiert.

Umgebung -> Frameworks -> Cordova

Die Plattform Apache Cordova ermöglicht es die mit Webtechnologien wie HTML, JavaScript und CSS erstellten Anwendung als App zu verpacken. Es bildet dabei die Brücke zwischen dem Betriebssystemen und der mit Webtechnologien entwickelten Anwendung. Die Anwendung wird dabei in einem so genannten WebView angezeigt. Das WebView wird vom Betriebssystem bereitgestellt und beeinfluss grundlegend die Geschwindigkeit der Anwendung. Über Cordova APIs haben Anwendungen die Möglichkeit auf Betriebssystems-Features zuzugreifen [0.1].

Cordova unterstützt alle gängigen Handy- und Table-Betriebssysteme wie Android, iOS und Windows Phone. Zum Zeitpunkt an dem diese Arbeit erstellt wurde liegt Cordova in der Version 4.1.2 vor. In dieser Version können Apps für Android 2.3.x oder 4.x erstellt werden, wobei das "x" für eine beliebige verfügbare Versionsnummer steht [0.2].

[0.1] http://cordova.apache.org/
[0.2] http://cordova.apache.org/docs/en/4.0.0/guide_platforms_android_index.md.html#Android%20Platform%20Guide

Umgebung -> Frameworks -> AngularJS

AngularJS ist ein JavaScript-Framework mit dem dynamischen Single-Page Anwendungen erstellt werden können. Es wurde von Google veröffentlicht und erfreut sich derzeit stetig steigender Beliebtheit. Tatsächlich verfügt AngularJS über einige Features, die das Entwickeln sehr angenehm gestalten. Einzelne Seiten einer Anwendung die mit AngularJS erstellt werden, basieren auf einer Model-View-Controller-Architektur (MVC). In JavaScript erstellte Controller steuern die Logik und kommunizieren über ein Model mit der Benutzeroberfläche. Diese Benutzeroberfläche wird mittels HTML erstellt und präsentiert Inhalte aus dem Model. Dadurch sind Anwendungen automatisch Strukturiert und einfach zu testen. Durch das von AngularJS bereitgestellte Two-Way-Databinding wird die Benutzeroberfläche automatisch mit dem Model synchronisiert. Änderungen im Model werden somit direkt in der Benutzeroberfläche sichtbar und umgekehrt. Zusätzlich zu den MVC-Elementen ermöglicht AngularJS das definieren von so genannten Services. Dies sind Objekte, die über die Laufzeit einer Anwendung bestimmte Funktionen übernehmen. Mittels Dependency-Injection können diese Services beliebig in Controllern oder anderen Stellen eingebunden werden. Ein weiteres Feature von AngularJS sind Direktiven. Durch Direktiven lassen sich eigene HTML-Elemente oder -Attribute definieren und in die Benutzeroberflächen einbinden. Dies fördert die Übersichtlichkeit und ermöglich die Bildung von wiederverwendbaren Komponenten. Details zu diesen oder weiteren Features folgen bei Bedarf in den folgenden Kapiteln [0.3].

[0.3] https://docs.angularjs.org/guide/introduction

Umgebung -> Frameworks -> Ionic

Ionic ist ein Framework zur Entwicklung von grafischen Benutzeroberflächen für Apps auf Basis von Webtechnologien. Es stellt bekannte UI-Komponenten zur Verfügung um schöne, schnelle und vor allem native Benutzeroberflächen zu gestalten. Im Kern verwendet Ionic AngularJS und stellt beispielsweise über AngularJS Direktiven seine UI-Komponenten zur Verfügung. Die aktuelle Version von Ionic ist 1.0.0-beta 13. Dieser Name impliziert bereits, dass sich das Framework noch in der Beta-Phase befinden. Dennoch wird es schon in Produktivumgebungen eingesetzt. In dieser Version von Ionic wird iOS 6+ sowie Android 4.0+ Offiziell unterstützt. Da Ionic auf AngularJS basiert, setzt jede Ionic-Version eine spezielle AngularJS Version voraus. In diesem Fall ist es die Version 1.2.27 [0.4].

[0.4] http://ionicframework.com/docs/overview/#download

Umgebung -> Plattformen

In diesem Abschnitt wird auf die notwendige Android Version und die daraus resultierenden Probleme eingegangen. Die verwendeten Frameworks erfordern zum Teil eine bestimmte Version von Android.

- Ionic: 4.0+
- Cordova: 2.3.x und 4.0+

Daraus ergibt sich, dass mindestens Android 4.0 für den Einsatz mit diesen Technologien erforderlich ist.

Wie bereits beschrieben verwendet Cordova eine WebView Komponente um die Web-App darzustellen. Diese WebView Komponente basierte seit dem Bestehen von Android auf der WebKit Engine [0.9.2]. Seit Android Version 4.4 wird jedoch Chromium verwendet, eine von Google entwickelte Browser-Engine [0.9.1]. Der Grund für den Wechsel war zum einen die bessere Performance und zum anderen eine bessere Unterstützung von Standards [0.5][0.9]. Bei der Entwicklung von Apps muss man sich also bewusst sein, dass je nach Android Version von Cordova eine unterschiedliche Browser-Engine verwendet wird. Dadurch kann es zu Unterschieden in der Performance kommen. Außerdem muss

sichergestellt sein, dass verwendete Standards in beiden Versionen funktionieren. Abbildung {TODO} zeigt die aktuelle Verteilung der Android Versionen. Laut dieser Übersicht verwenden derzeit 52,7% aller Android Geräte noch Versionen zwischen 4.0 bis 4.3. Die neue und für Web-App günstigere Version 4.4 wird jedoch erst von 39,1% aller Geräte verwendet [0.6]. Dies bedeutet, je größer die Zielgruppe an Geräte, die man mit seiner App abdecken möchte, desto mehr Performance-Optimierungen sind notwendig.

Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	7.8%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	6.7%
4.1.x	Jelly Bean	16	19.2%
4.2.x		17	20.3%
4.3		18	6.5%
4.4	KitKat	19	39.1%

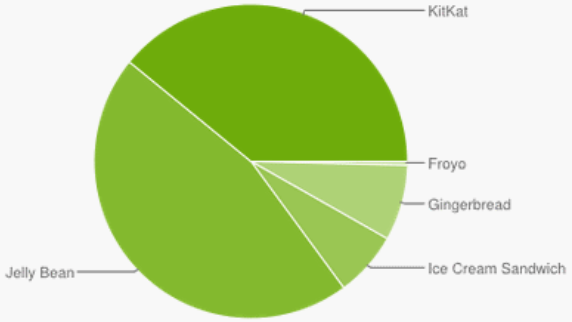

Abb. 0.1 (https://developer.android.com/about/dashboards/index.html)

[0.5] https://developer.android.com/guide/webapps/migrating.html
[0.6] https://developer.android.com/about/dashboards/index.html
[0.9] https://developer.chrome.com/multidevice/webview/overview
[0.9.1] http://www.chromium.org/Home
[0.9.2] https://developer.android.com/reference/android/webkit/WebView.html

Tim Roes ein Computer Scientist hat auf seinem Blog Benchmarks durchgeführt und die beiden Engins verglichen [0.7]: "The new (chromium based) WebView is faster – so far no surprise. But looking at the numbers, the performance has really increased in several areas (like up to 354% for HTML5 Canvas or 358% for some Javascript test)."

[0.7] https://www.timroes.de/2013/11/23/old-webview-vs-chromium-webview/

Dabei hat er die gleichen Benchmarks auf demselben Gerät jeweils mit Android 4.3 und Android 4.4 durchgeführt. Abgesehen von performance Optimierungen, die durch den Versionssprung an sich ausgelöst wurden, zeigt dieser Vergleich dennoch ganz gut, dass sich der Wechsel lohn.

Verschiedene Projekte versuchen diesen Vorteil auch für ältere Android Versionen zugänglich zu machen. Eines davon nennt sich Crosswalk, ein von der Intel's Open Source Technology Center veröffentlichtes Open Source Projekt. Crosswalk basiert auf Chromium und liefert für jede App eine eigene Browser-Engine mit. Eine mit Cordova entwickelte App läuft demnach nicht in der Android WebView, sondern in einer von Crosswalk bereitgestellten Komponente. Dadurch laufen Apps für Android 4.x immer in der gleichen Browser-Engine und haben gleiche Voraussetzungen bezüglich der Performance und den verwendbaren Standards. Ein erster Schritt um die Performance für ältere Android Versionen zu verbessern wäre also der Einsatz solch eines Projektes. Natürlich hat der Einsatz auch Nachteile. Durch das Bereitstellen einer auf Chromium basierenden Engine müssen einige Daten mit der App mitgeliefert werden [0.8]. Dadurch beträgt die Größe der App ohne weitere Inhalte bereits ca. 20mb. Jeder Entwickler muss also für sich selbst die Entscheidung treffen, ob er diese Maßnahme für sinnvoll erachtet.

[0.8] https://crosswalk-project.org/documentation/about/faq.html

Performance

Der Begriff Performance wurde in einschlägigen Literaturen bereits ausgiebig diskutiert. Die folgende Definition soll daher in dieser Arbeit als Basis herangezogen werden [1]:

„Performance is the degree to which a software system or component meets its objectives for timeliness“.
Mit "timeliness" ist dabei das Antwortzeitverhalten, also die Geschwindigkeit des Systems aus Sicht des Anwenders gemeint. Das Antwortzeitverhalten ist ein wichtiger Aspekt für die Beurteilung der Qualität des Softwaresystems durch den Anwender. Verzögerungen bei Eingaben oder lange Wartezeiten wirken sich demnach negativ auf seine Erfahrungen aus. Studien haben gezeigt, dass {TODO: Studie finden}% aller Anwender Apps aufgrund mangelnder Performance deinstallieren. Neben anderen Aspekten ist es für den Erfolg einer App also wichtig ein gutes Antwortzeitverhalten aufzuweisen.

[1]: Smith, C.U.: Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software. Addison-Wesley (2002)

Performance -> Kategorien

Eine schlechtes Antwortzeitverhalten kann viele Ursachen haben. Um diesen Ursachen bereits bei der Entwicklung einer App entgegenzuwirken, ist es erforderlich sie zu kennen. Die Codecentric AG hat auf Basis von Großprojekten und Performance-Analysen in diesem Zusammenhang acht Kategorien von Ursachen erarbeitet: [2]

Performance-Problemkategorien

Problemkategorie	Beschreibung
Ineffiziente Algorithmen	Es kommt schon bei geringer Last zu schlechtem Antwortzeitverhalten, weil umständliche Algorithmen implementiert wurden.
Ineffiziente Zugriffspfade	Ein Spezialfall ineffizienter Algorithmen, bei dem der Datenspeicher eine suboptimale Lösung für die Durchführung einer Datenabfrage ermittelt.
Speicherleck (engl. „memory leak“)	Es kommt zu einer Verschlechterung des Antwortzeitverhaltens bei längerer Laufzeit, verursacht durch mangelhaftes Speichermanagement.
Reinitialisierungsproblem	Das System verhält sich bei zunehmender Last unberechenbar, weil Speicherbereiche vor der Verwendung nicht sauber initialisiert werden.
Flaschenhälse	Es kommt zu exponentiell steigenden Antwortzeiten, weil an bestimmten Systemstellen eine Warteschlange entsteht. Dabei werden pro Zeiteinheit mehr Anfragen an die Systemstelle herangetragen als verarbeitet werden können.
Intermittierende Probleme	Das System reagiert plötzlich mit sehr schlechten Antwortzeiten, weil zum Beispiel die Netzwerkverfügbarkeit durch andere Prozesse gestört wird.
Ressourcenauslastung	Es kommt zu einer exponentiellen Verschlechterung des Antwortzeitverhaltens bei Erhöhung der Last, weil sich die CPU-Auslastung an bestimmten Systemknoten der Auslastungsgrenze nähert.
Ressourcenkonflikte (engl. „deadlocks“)	Es kommt bei zunehmender Last zu Konfliktsituationen durch parallelen Zugriff auf gemeinsame Ressourcen wie Datenbanktabellen.

Abb. 0.2 (https://www.codecentric.de/files/2011/06/performance-problemkategorien.png)

Diese Kategorien sind sehr allgemein gehalten und haben keinen direkten Zusammenhang zu den Betrachteten Webtechnologien HTML, JavaScript und CSS. Für eine Einordnung ist es daher notwendig zunächst die Abläufe in einem Browser zu betrachten.

[2]: https://www.codecentric.de/kompetenzen/publikationen/performance-analyse-und-optimierung-in-der-softwareentwicklung/

Performance -> Funktionsweise einer Browser-Engine

Der konkrete Ablauf in einem Browser hängt natürlich von der jeweiligen Implementierung ab. Jeder Browser funktioniert hier etwas anders. Jedoch sind grundlegenden Abläufe und Strukturen meist gleich und zum Teil auch durch Standards definiert [10][11]. Eine grobe Betrachtung der Abläufe ist also ohne Fokussierung auf einen speziellen Browser möglich.

Als Grundlage dieser Betrachtung wird eine Zusammenfassung der Funktionsweise aktueller Browser

von Tali Garsiel herangezogen. Sie hat über einige Jahre Implementierungen von Oben-Source-Browsern analysiert und Ihre Erkenntnisse zusammengefasst. [3]

Ein Browser besteht demnach aus den in Abb. 1 dargestellten Komponenten. Das User Interface stellt dem Anwender alle Komponenten für die Interaktion mit der Webseite zur Verfügung. Ein Beispiel dafür sind die Vor-/Zurück-Schaltflächen oder Lesezeichen. Eine Ebene tiefer befindet sich die Browser-Engine. Sie dient als Schnittstelle zwischen dem User Interface und der Rendering Engine. Der wichtigste Bestandteil ist die Rendering Engine. Sie ist für die Anzeige von HTML-Datei zuständig und kommuniziert dabei mit einem JavaScript Interpreter und einer Netzwerkschnittstelle. Ein Beispiel für eine Rendering Engine ist WebKit in Safari oder Blink in Chrome und Opera. [12][13]

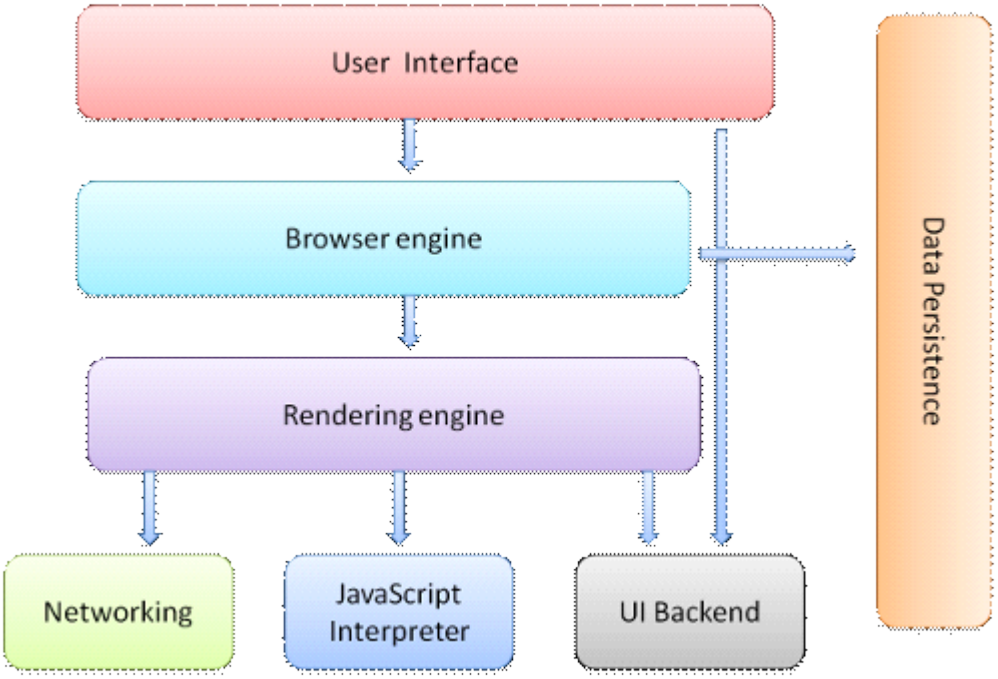

Abb. 1
(http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/layers.png)

[10] http://www.w3.org/DOM/DOMTR
[11] https://html.spec.whatwg.org/multipage/syntax.html#parsing
[12] http://www.chromium.org/blink
[13] http://www.webkit.org/

Für das Anzeigen einer URL ruft die Rendering Engine zunächst die HTML-Datei von der Netzwerkschnittstelle ab. Diese Schnittstelle liefert das Dokument in Blöcken und die Rendering Engine beginnt mit der Verarbeitung.

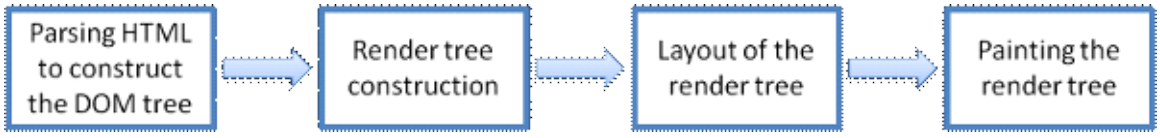

Abb. 2
(http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/flow.png)

Zu Beginn wird die HTML-Datei geparst und einzelne HTML-Element in DOM-Elemente umgewandelt. Aus diesen DOM-Elementen entsteht der DOM-Tree. Verweise auf CSS-Dateien werden aufgelöst und sobald möglich ebenfalls geparst. Es entstehen CSSOM-Elemente die den CSSOM-Tree ergeben. Beide Bäume werden im nächsten Schritt zu einem Render-Tree zusammengefasst. Er enthält nur sichtbare Elemente inklusive ihrer Formatierungseigenschaften. Durch den Layout-Prozess erhält anschließend jedes Element des Render-Tree eine feste Position auf dem Bildschirm und kann dadurch im letzten Schritt gezeichnet werden. [15][16]

Der Begriff Single-Page Application (SPA) beschreibt im Bezug zu Webtechnologien eine Website die aus einer einzigen HTML-Datei besteht. Der Inhalt dieser Datei ändert sich dynamisch zur Laufzeit der Anwendung [17]. Bei dieser Art von Anwendung fällt Schritt 1 in Abb. 2 nur beim ersten Aufruf ins Gewicht. Der DOM- und CCOM-Tree wird nur einmalig erstellt und zur Laufzeit der Anwendung lediglich geändert. Durch Aktionen in JavaScript kann auf den DOM zugegriffen werden. Das Abfragen oder Setzen von Werten in einem DOM-Element kann den Effekt haben, dass die Rendering-Engine einen Reflow durchführen muss. Ein Reflow bezeichnet das neuberechnen der Positionen und zeichnen aller Elemente. In Abb. 2 bezeichnet dies die Schritte drei und vier. Reflows sollten möglichst vermieden werden, da es sich um eine blockierende Aktion handelt und es das Antwortzeitverhalten der Anwendung beeinträchtigen kann [18].

Das Parsen eines HTML-Dokumentes kann bereits durchgeführt werden, wenn erste Teile des Dokumentes über die Netzwerkschnittstelle eingetroffen sind. Dadurch verkürzt sich die Wartezeit für den Anwender. [15]Problematisch wird es, wenn JavaScript-Verweise in einem Dokument vorhanden sind. Prinzipiell verzögern sich dadurch das Parsen und damit auch die schnelle Anzeige eines Dokumentes. Denn JavaScript wird ausgeführt, sobald es beim Parsen gefunden wird.[JSHP, Seite 1]

Externe Ressourcen (JavaScript, CSS, Bilder, etc.) werden beim Parsen über die Netzwerkschnittstelle heruntergeladen. Um diesen Vorgang zu beschleunigen öffnen die verschiedenen Browser eine unterschiedliche Anzahl von parallelen Verbindungen. In Chrome werden beispielsweise maximal 6 parallele Verbindungen zugelassen [5]. Befinden sich mehr externe Ressourcen in einem Dokument, verzögert sich demnach das Herunterladen und Ausführen, bis erneut freie Verbindungen vorhanden sind. Zu dieser Problematik zählen auch XHR-Abfragen. [Selbst getestet]

Performancetechnisch lassen sich aus diesen Vorgängen bereits grundlegende Regeln ableiten.

1. Größe des HTML-Dokument <-> Zeit für DOM-Tree Erstellung
2. Anzahl CSS-/JavaScript-Verweise und Netzwerkgeschwindigkeit <-> Zeit für die DOM-Tree Erstellung.
3. Anzahl/Komplexität des CSS <-> Zeit für CSSOM-Tree Erstellung
4. Größe des DOM-/CSSOM-Tree <-> Zeit für Render-Tree Erstellung
5. Anzahl Elemente im Render-Tree <-> Zeit fürs Layouten
6. Anzahl Elemente im Render-Tree <-> Zeit fürs Zeichnen
7. => Komplexität des HTML/CSS beeinfluss Antwortzeitverhalten
8. => Management externer Ressourcen beeinfluss Antwortzeitverhalten

{Gloassar}
DOM = Document Object Model (http://www.w3.org/DOM/DOMTR)
CSSOM = Cascading Style Sheet Object Model (http://dev.w3.org/csswg/cssom/)
XHR = XmlHTTPRequest (http://www.w3.org/TR/XMLHttpRequest/)
+ Kurze Beschreibung
DOM-Element
Reflow
SPA = Single Page Application

[3] http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/
[4] https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=de
[5] http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/
[15] http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Rendering_engines
[16] https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=de
[17] http://www.johnpapa.net/pageinspa/
[18] https://developers.google.com/speed/articles/reflow

Performance -> Web

In diesem Abschnitt wird nun darauf eingegangen, wie sich die in Abschnitt {TODO} genannten Kategorien auf die betrachteten Webtechnologien anwenden lassen.

Ineffiziente Algorithmen:

Dieses Problem hat wenig mit der verwendeten Technologie zu tun. Denn egal in welcher Sprache, langsame Algorithmen können immer der Grund für eine schlechte Performance sein. Analysen im Vorfeld oder Code-Reviews können solche Probleme erst gar nicht entstehen lassen.

Ineffiziente Zugriffspfade:

Der Zugriff auf einen Datenspeicher wird auf eine langsame Art und Weise durchgeführt. Unter diesem Problem kann man in Bezug zu JavaScript den Zugriff auf den DOM verstehen. Wie beschrieben können Zugriffe auf Elemente im DOM einen Reflow auslösen, was sich entsprechend negativ auf die Performance auswirkt. Daher ist bei solch einem Zugriff immer darauf zu achten WIE und auf welche Attribute man zugreifen möchte.

Speicherleck:

Unter einem Speicherleck versteht einen Bereiche bzw. ein Objekt im Speicher, zu dem es keine Referenz mehr gibt. In JavaScript soll sich der Garbage-Collector, um das Freigeben solcher Bereiche kümmert. Dennoch kann es auch in heutigen Browsern noch vorkommen, dass es dem Garbage-Collector nicht möglich ist festzustellen, ob noch Referenzen auf ein Objekt vorhanden sind und es daher freigeben werden muss. Speicherlecks sorgen dafür, dass die Speicherauslastung zur Laufzeit der Anwendung immer weiter ansteigt, bis sie letztendlich einfriert. Relevant ist dies nur bei Anwendungen, die über eine längere Zeit genutzt werden. Herkömmliche Webseiten hatten hingegen meist eine kurze Laufzeit und daher waren Speicherlecks weniger von Bedeutung. Spätestens beim schließend der Seite konnte dann der gesamte verwendete Speicher freigegeben werden. Im Hinblick auf SPA ist dies jedoch ein größeres Problem. Gelangt man mit solch einer Anwendung an die Grenzen des verfügbaren Speichers, wird der Garbage-Collector zu nehmend aktiv, was sich auf das Antwortzeitverhalten auswirkt. Außerdem können dynamische Inhalte nicht erstellt werden, wenn der nötige Platz im Arbeitsspeicher fehlt. [19][20]

Reinitialisierungsproblem:

Dieses Problem bezieht sich auf die Verwendung von Variablen, die nicht initialisiert wurden. Bei Programmiersprachen wie C++ haben solche Variablen einen Wert, der dem Entspricht, was sich zuletzt an der jeweiligen Speicheradresse befunden hat. Eine Verwendung dieses Wertes kann entsprechende Fehler verursachen. Unter JavaScript spielt dieses Problem keine Rolle, da neue Variablen standardmäßig mit undefined initialisiert werden.[21]

Flaschenhälse:

Flaschenhälse sind Komponenten einer Anwendung, die durch Anfragen so sehr belastet werden, dass Warteschlangen entstehen. Der Zugriff auf diese Komponenten dauert entsprechend lange. In JavaScript zählt der XHR-Zugriff dazu. Wie in Abschnitt {TODO} beschrieben öffnet jeder Browser nur eine maximale Anzahl paralleler Verbindungen. Werden mehr Anfragen gestellt als Antworten eintreffen, bildet sich eine Warteschlange und der Anwender wartet.

Intermittierende Probleme:

Durch die Auslastung von Systemressourcen erhöhen sich die Antwortzeiten. Im Bezug zu Webtechnologien kann damit der Reflow-Prozess gemeint sein. Werden durch Aktionen in JavaScript Reflows ausgelöst, leidet darunter die Reaktionsfähigkeit der Anwendung. Anstatt auf Nutzerinteraktionen zu reagieren, ist die Rendering-Engine mit dem Layouten und Zeichnen der Anwendung beschäftigt.

Ressourcenauslastung

//TODO: Equivalent in JavaScript finden
Vielleicht Netzwerk + Objekt Management (GC muss zu viel aufräumen)

Ressourcenkonflikte

//TODO: Equivalent in JavaScript finden

All diese Punkte sind also bei der Entwicklung einer performanten Web-App grundlegend zu beachten. In den Best-Practice Lösungen wird darauf aufbauend die Performance bei der Verwendung der genannten Frameworks betrachtet und analysiert.

{Glossar}
Garbage-Collector

[19] http://javascript.info/tutorial/memory-leaks
[20] http://www.ibm.com/developerworks/library/wa-jsmemory/
[21] http://zentrum.virtuos.uni-osnabrueck.de/wikifarm/fields/webtech11/index.php?n=Themen.Javascript

Performance -> Ziele

Zusammenfassend lassen sich durch die vorherigen Betrachtungen die Ziele hinsichtlich der Performance definieren. Prinzipiell soll das Antwortzeitverhalten mit den Best-Practice Lösungen optimiert werden. Dazu gehört es die Wartezeiten der Anwender zu minimieren und zusätzlich die Anwendung bei Animationen flüssig wirken zu lassen. Flüssig in diesem Zusammenhang bedeutet bei einer konstanten Bildwiederholungsrate von 60 FPS. Bei zu wenigen oder nicht konstanten FPS würde die Anwendung bei Animationen nicht flüssig wirken. Bei einer Bildwiederholungsrate von 60 FPS bleiben der Anwendung 16,7 ms pro Frame. Aufgrund der Leistungsschwäche von mobilen Geräten sollte man jedoch eher von einem Budget von etwa 8-10 ms ausgehen. In jedem Frame findet die JavaScript Ausführung, das layouten des Render-Tree und das Zeichnen des Render-Tree statt. Also ziemlich viel Arbeit für eine so kleine Zeitspanne. Daher gilt es Zeit einzusparen wo es nur geht. [22]

[22] http://www.smashingmagazine.com/2013/06/10/pinterest-paint-performance-case-study/

Vorbereitung -> Performancemessungen

Wenn es um den Vergleich verschiedener Implementierungsansätze geht, sind Performance-Messungen ein wichtiges Kriterium. In diesem Abschnitt wird daher auf die Einzelheiten der Performance-Messungen eingegangen, wie sie in dieser Arbeit angewendet werden. Grundsätzlich werden aufgrund der Problematiken in Abschnitt {TODO} für alle Messungen zwei Testgeräte verwendet. Ein Android-Gerät in der Version 4.4 und eines in der Version 4.3. Dadurch werden die Unterschiede noch einmal klar sichtbar.

Die Performance-Messung von JavaScript-Code ist keine einfache Angelegenheit. Die Aussagekraft einer Messung kann durch viele, vom Entwickler nicht steuerbare Faktoren, beeinfluss werden. Dazu zählen folgende Punkte:

- Garbage Collection
  Der Garbage Collector sorgt für das Freigeben nicht mehr verwendeter Speicherbereiche. Da eine JavaScript-Engine nur auf einem Prozess läuft, verzögert der Garbage Collector Prozess die aktuelle JavaScript Ausführung. Der Entwickler hat keine Möglichkeit zu steuern wann der Prozess seine Arbeit beginnt. Zeitmessungen können durch diesen Prozess also verfälscht werden.[25][26]

- Optimierungen durch JavaScript-Engines
  Aktuelle JavaScript-Engines wie Googles V8 optimieren den Quellcode zur Laufzeit. Bei Zeitmessungen kann dies dazu führen, dass Wiederholungen ab einem bestimmten Zeitpunkt durch die Optimierung sehr viel schnell ausgeführt werden als zuvor. Die V8 Engine setzt zu diesem Zweck zwei verschiedene Compiler ein. Der generische Compiler übersetzt den kompletten JavaScript Quelltext in Maschinencode. Ein Profiler analysiert den Quellcode und sucht nach Stellen, die eine besondere Optimierung erfordern. Diese Optimierung übernimmt dann der zweite Compiler. Durch optimierten Maschinencode können so sehr schnelle Ausführungsgeschwindigkeiten erreicht werden.[27]

- Andere Hintergrundprozesse

All diese Probleme führen dazu, dass ein eine einfache Messung der Zeit für einen Vorgang mit n Wiederholungen und einem Durchschnitt nicht ausreicht. [23][24]

Die JavaScript-Library BenchmarkJS wurde aufgrund all dieser Probleme entwickelt und liefert für Zeitmessungen statistisch relevante Ergebnisse. Eine einfache Schnittstelle ermöglicht es dem Entwickler Performance-Tests durchzuführen, ohne Kenntnisse über diese Problematiken zu benötigen [28]. {TODO: Beschreiben wie BenchmarkJS intern funktioniert um statistisch relevante Ergebnisse zu erhalten}

Für die betrachteten Implementierungen sind zwei verschiedene Arten von Zeitmessungen erforderlich. Um mit BenchmarkJS einen synchronen Vorgang zu messen kann der Code in Abbildung {TODO} verwendet werden. Für das Messen eines asynchronen Vorgangs hingegen der Code in Abbildung {TODO}.

Synchron:

```
var suite = new Benchmark.Suite;

suite.add('Sync test', function () {

   // Test code

}).on('complete', function () {
   // Check test results
   var times = this[0].times;
   var stats = this[0].stats;
})
.run();
```

Asynchron:

```
var suite = new Benchmark.Suite;

suite.add('Async test', {
   defer: true,
   fn: function (deferred) {

      // Test code
      setTimeout(function () {
         deferred.resolve();
      }, 1000);

   }
}).on('complete', function () {
   // Check test results
   var times = this[0].times;
   var stats = this[0].stats;
})
.run();
```

[23] https://mathiasbynens.be/notes/javascript-benchmarking
[24] http://ejohn.org/blog/javascript-benchmark-quality/
[25] http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/
[26] http://javascript.info/tutorial/events-and-timing-depth
[27] http://blog.chromium.org/2012/05/better-code-optimization-decisions-for.html
[28] http://benchmarkjs.com/

Vorbereitung -> Aufbau einer Beispielanwendung

Das Ziel ist es die Implementierungsansätze in einer einzigen App zu testen. Dadurch können auf Anhieb alle Möglichkeiten ohne großen Aufwand auf verschiedenen Geräten ausprobiert werden. Diese App muss daher so aufgebaut sein, dass neue Implementierungen (Module) ohne viel Aufwand hinzugefügt werden können.

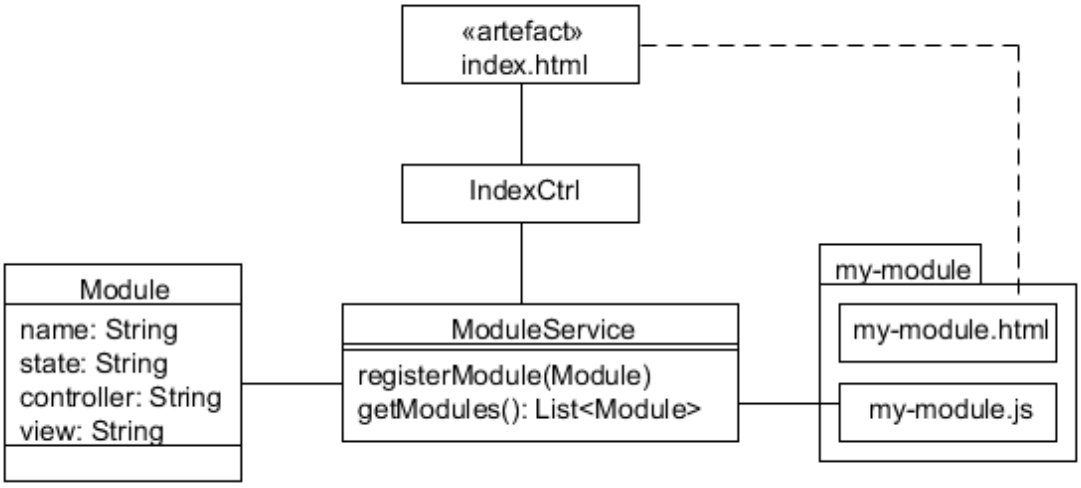

Abb 3.

Um dies zu ermöglichen werden Module in einem Unterordner der Anwendung abgelegt. Jedes Modul besteht mindestens aus einer JavaScript-Datei und einer HTML-Datei. Die JavaScript-Datei muss in der index.html als Skript referenziert werden. In der JavaScript-Datei wird das Modul in einem globalen Service registriert. Wird nun die index.html angezeigt, ruft sie alle registrierten Module vom globalen Service ab und zeigt sie mit ihrem Namen in einer Liste an. Klickt der Anwender auf ein Modul, wechselt die Anwendung zur HTML-Datei des Moduls. Hier können nun Interaktionen oder Informationen für die jeweilige Implementierung angezeigt werden. Für das Routing ist es bei der Registrierung eines Modules erforderlich neben dem Namen des Modules auch noch den Name des Zustands, den Name des Controllers und den Pfad zur HTML-Datei anzugeben. Alles weitere übernimmt der ModuleService.

Die Anwendung basiert grundsätzlich auf den gleichen Technologien und Frameworks wie sie in dieser Arbeit betrachtet werden. Zusätzlich wird neben der bereits beschriebene Library BenchmarkJS noch LoDash eingesetzt. Letzteres stellt viele Funktionen für die Interaktion mit Arrays und Objekten zur Verfügung [29].

[29] https://lodash.com/