Fachhochschule Köln
Cologne University of Applied Sciences

Erarbeitung von Best-Practise Lösungen bei der Entwicklung performanter mobiler hybrider Web-Anwendungen mit AngularJS, Ionic und Cordova

Bachelorarbeit

ausgearbeitet von

Marco Busemann

vorgelegt an der

Fachhochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang

Allgemeine Informatik
Matrikelnummer: 11081445

Erster Prüfer:	Professor Dr. Heiner Klocke Fachhochschule Köln
Zweiter Prüfer:	Dr. Hendrik Voigt OPITZ CONSULTING GmbH

Gummersbach, im April 2015

Adressen: Marco Busemann
Im Sohl 33
51643 Gummersbach
marco.busemann@live.com

Professor Dr. Heiner Klocke
Fachhochschule Köln
Fakultät für Informatik
und Ingenieurwissenschaften
Steinmüllerallee 1
51643 Gummersbach
heinrich.klocke@fh-koeln.de

Dr. Hendrik Voigt
OPITZ CONSULTING GmbH
Kirchstraße 6
51647 Gummersbach (Nochen)
hendrik.voigt@opitz-consulting.com

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 mit Best-Practice Lösungen für die Entwicklung von Apps mit diesen Frameworks zu erstellen. Der Begriff App wird in diesem Zusammenhang für so genannte „Productivity“ Apps verwendet. Spiele fallen nicht in diese Kategorie, da sich ihre Anforderungen stark unterscheiden. 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 oder dem Nutzererlebnis getroffen. Im Ergebnis wird deutlich, dass @TODO: Ergebnis ergänzen

Motivation

Das Interesse an Apps für Smartphones oder Tablets ist seit einigen Jahren ungebrochen. Und wohl auch in Zukunft ist mit zunehmendem Interesse zurechnen. Regelmäßig erscheinen neue Betriebssystemen oder Plattformen, die mit einer App abgedeckt werden müssen. Dadurch ergibt sich die Anforderung die jeweilige App auf möglichst vielen dieser Systeme zu positionieren. Unternehmen stehen bei der Entwicklung von neuen Apps 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 oft eine Entscheidung gegen diese Technologien. Dies zeigt auch das Umdenken großer Unternehmen wie Facebook[9] und LinkedIn[28] in diesem Zusammenhang. Die Performance, Usability und Entwicklertools reichen oft nicht an eine native Lösung heran [8]. Moderne Frameworks auf Basis von Webtechnologien wie AngularJS und Ionic versuchen diese Lücke zu schließen und legen bei ihrer Umsetzung viel Wert auf eine performante und ressourcensparende Implementierung. Dadurch soll schon heute die Entwicklung konkurrenzfähiger Apps für Smartphone möglich sein. Um eine gewisse Performance zu gewährleisten reicht es jedoch nicht aus nur diese Frameworks einzusetzen. Die Performance hängt stark von der eigenen Implementierung ab. 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.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Frameworks

In diesem Kapitel werden die verwendeten Frameworks und ihre Versionen beschrieben. Die Frameworks bauen technisch aufeinander auf und können somit auf drei Ebenen angeordnet werden. Auf der untersten Ebene befindet sich Cordova, als Schnittstelle zum Betriebssystem. Darüber ist AngularJS angesiedelt und gibt die Struktur der Anwendung vor. Auf der obersten Ebene befindet sich letztendlich Ionic. Ionic stellt die User Interface (UI) Komponenten bereit, mit denen der Anwender interagiert.

1.1 Cordova

Die Plattform Apache Cordova ermöglicht es die mit Webtechnologien wie Hypertext Markup Language (HTML) , JavaScript und Cascading Style Sheets (CSS) erstellten Anwendung als App zu verpacken. Es bildet dabei die Brücke zwischen dem Betriebssystemen und der mit Webtechnologien entwickelten Anwendung. Die Anwendung wird in einer Komponente mit dem Namen WebView[1] angezeigt. Das WebView wird vom Betriebssystem bereitgestellt und beeinflusst grundlegend die Geschwindigkeit der Anwendung. Über eine von Cordova bereitgestellte Application Programming Interface (API) haben Anwendungen die Möglichkeit auf Funktionen des Betriebssystems sowie sonstige Programmierschnittstellen zuzugreifen [43]. Cordova unterstützt alle gängigen Handy- und Tablet-Betriebssysteme wie Android, iOS und Windows Phone. Zum Zeitpunkt des Schreibens dieser Arbeit liegt Cordova in der Version 4.1.2 vor. In dieser Version können Apps für Android[42] 2.3.x und 4.x sowie für iOS[44] >5.x erstellt werden.

1.2 AngularJS

AngularJS ist ein JavaScript-Framework mit dem dynamische Single Page (SP) Anwendungen erstellt werden können. Es wurde von Google veröffentlicht und erfreut sich derzeit stetig steigender Beliebtheit. Tatsächlich verfügt AngularJS über einige

[1] Ein WebView ist eine UI Komponente, die Webseiten wie in einem Browser darstellen und JavaScript ausführen kann.

Features, die einen positiven Effekt auf die Entwicklung von Web-Apps haben. Einzelne Seiten einer mit AngularJS erstellten Anwendung basieren auf der Model View Controller (MVC) Architektur. In JavaScript erstellte Controller steuern die Logik und vermitteln zwischen dem Model und der Benutzeroberfläche. Diese Benutzeroberfläche wird mit einem Markup[2], das auf HTML basiert deklarativ beschrieben und präsentiert Inhalte aus dem Model. 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 Komponenten ermöglicht AngularJS das Definieren von so genannten Services. Dies sind Objekte, die über die Laufzeit einer Anwendung bestimmte Funktionen übernehmen. 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 ermöglicht die Bildung von wiederverwendbaren Komponenten. AngularJS verwendet das Dependency Injection (DI) [3] Pattern um Abhängigkeiten zwischen Komponenten einer Anwendung zu definieren. Darüber lassen sich Services in beliebigen Komponenten der Anwendung einbinden und nutzen. In Kombination mit der MVC Architektur ist es damit möglich leicht testbare Apps zu entwickeln. Details zu diesen oder weiteren Features folgen in den kommenden Kapiteln dieser Ausarbeitung.[19]

1.3 Ionic

Ionic ist ein Framework zur Entwicklung von grafischen Benutzeroberflächen für Apps auf Basis von Webtechnologien. Es stellt bekannte Komponenten für UIs zur Verfügung um die Entwicklung zu beschleunigen. Auf Wunsch sind diese Komponenten dem nativen „Look and Feel“ der jeweiligen Plattform nachempfunden. Im Kern verwendet Ionic eine bestimmte Version von AngularJS und stellt über AngularJS Direktiven die UI-Komponenten zur Verfügung. Die aktuelle Version von Ionic heißt 1.0.0-beta 13. Dieser Name impliziert bereits, dass sich das Framework noch in der Beta-Phase befindet. In dieser Version von Ionic wird iOS 6+ sowie Android 4.0+ Offiziell unterstützt. Da Ionic auf AngularJS basiert, setzt jede Version von Ionic eine spezielle Version von AngularJS voraus. In diesem Fall ist es die Version 1.2.27.[10]

[2] Der Begriff Markup beschreibt eine Sprache, die zur Gliederung und Formatierung von Texten oder anderen Daten eingesetzt wird und maschinell gelesen werden kann.

[3] „Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. “[2]

2 Plattformen

In diesem Kapitel wird auf die unterstützten Versionen von Android und die daraus resultierenden Probleme eingegangen. Die verwendeten Frameworks erfordern zum Teil eine bestimmte Version von Android. Bei Ionic ist dies eine Version $>$ 4.0 und bei Cordova entweder die Version 2.3.x oder $>$ 4.0. Daraus ergibt sich, dass mindestens Android 4.0 für den Einsatz dieser Frameworks erforderlich ist.

Wie bereits im vorherigen Kapitel beschrieben, verwendet Cordova eine WebView Komponente um die App darzustellen. Diese Komponente basierte seit dem Bestehen von Android auf der WebKit Engine [14]. Erst seit Android Version 4.4 wird Chromium verwendet, eine von Google entwickelte Browser-Engine [17]. Der Grund für den Wechsel war zum einen die bessere Performance und zum anderen eine bessere Unterstützung von Standards [16][18]. Bei der Entwicklung von Apps mit Cordova wird demnach je nach Android Version eine unterschiedliche Browser-Engine für die Darstellung verwendet. Dadurch kann es zu Unterschieden in der Performance kommen. Außerdem muss sichergestellt sein, dass verwendete Standards in beiden Versionen funktionieren. Die Tabelle 2.1 zeigt die aktuelle Verteilung der eingesetzten Versionen von Android. Laut dieser Übersicht verwenden derzeit 52,7% aller Android Geräte Versionen zwischen 4.0 und 4.3. Die neue und für die hier betrachtete Art von Anwendung günstigere Version ab 4.4 wird erst von 39,1% aller Geräte eingesetzt [12]. Daraus lässt sich ableiten, dass die Zielgruppe einer App die erforderliche Optimierung der Performance maßgeblich beeinflusst. Tim Roes, ein Computer Scientist, hat Benchmarks mit beiden Engines durchgeführt und die Ergebnisse auf seinem Blog veröffentlicht [37]:

> "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)."

Er hat die gleichen Benchmarks auf demselben Gerät jeweils mit Android 4.3 und Android 4.4 durchgeführt. Abgesehen von Optimierungen der Performance, die durch den Versionssprung ausgelöst wurden, zeigt dieser Vergleich dennoch, dass vor allem für Versionen $<$4.4 von Android eine Optimierung der Performance relevant ist.

Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	6.9%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.9%
4.1.x	Jelly Bean	16	17.3%
4.2.x		17	19.4%
4.3		18	5.9%
4.4	KitKat	19	40.9%
5.0	Lollipop	21	3.3%

Tabelle 2.1: Android Versionsverteilung [12]

Verschiedene Projekte versuchen diesen Vorteil auch für ältere Versionen von Android zu nutzen. Crosswalk ist ein vom Intel Open Source Technology Center veröffentlichtes Open Source Projekt. Es basiert auf Chromium und liefert für jede App eine eigene Browser-Engine mit. Eine mit Cordova entwickelte App verwendet dabei nicht die WebView Komponente von Android, sondern eine von Crosswalk bereitgestellte Komponente. Somit basieren Apps ab Android 4.x auf der gleichen Browser-Engine und haben dadurch gleiche Voraussetzungen bezüglich der Performance und den verwendbaren Standards. Ein erster Schritt um die Performance für ältere Versionen von Android zu verbessern wäre demnach der Einsatz solch eines Projektes. Natürlich gibt es auch Nachteile. Durch das Bereitstellen einer auf Chromium basierten Engine müssen einige Software-Komponenten, Bibliotheken und Binärdaten mitgeliefert werden [21]. Dadurch beträgt die Größe der App ohne weitere Inhalte bereits ~20mb. Es ist daher abhängig von der jeweiligen App zu entscheiden, ob solch ein Verfahren Sinn macht.

3 Performance

Der Begriff Performance wurde in einschlägigen Literaturen bereits ausgiebig diskutiert. Die folgende Definition soll daher in dieser Arbeit als Basis herangezogen werden [38]:

> "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 vorzuweisen.

3.1 Kategorien

Eine schlechtes Antwortzeitverhalten kann viele Ursachen haben. Die Codecentric AG hat auf Basis von Großprojekten und Performance-Analysen in diesem Zusammenhang acht allgemeine Kategorien von Ursachen schlechter Performance erarbeitet (Tabelle 3.1)[23]. Diese allgemeinen Kategorien sollen in dieser Arbeit für SP Anwendungen konkretisiert werden. Dafür ist jedoch zunächst ein tieferes Verständnis der Funktionsweise dieser Art von Anwendungen und von Browser-Engines erforderlich. Aus diesem Grund wird in den folgenden Abschnitten zunächst die Funktionsweise einer Browser-Engine analysiert um anschließend diese Kategorien auf SP Anwendungen zu übertragen.

3.2 Browser-Engines

Der konkrete Ablauf in einem Browser hängt natürlich von der jeweiligen Implementierung ab. Jeder Browser funktioniert hier etwas anders. Dennoch sind grundlegenden Abläufe und Strukturen meist gleich und zum Teil auch durch Standards definiert[48][50]. Eine grobe Betrachtung der Abläufe ist daher ohne Fokussierung auf einen speziellen Browser möglich. Als Grundlage dieser Betrachtung wird eine Zusammenfassung der

Problemkategorien	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 längeren Laufzeiten, 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 ist.
Ressourcenauslastung	Es kommt zu einer exponentiellen Verschlechterung des Antwortzeitverhaltens bei Erhöhung der Last, weil sich die CPU-Auslastung an bestimmten Systemknoten der Auslastungsgrenze nähert.

Tabelle 3.1: Kategorien von Performance Problemen [23]

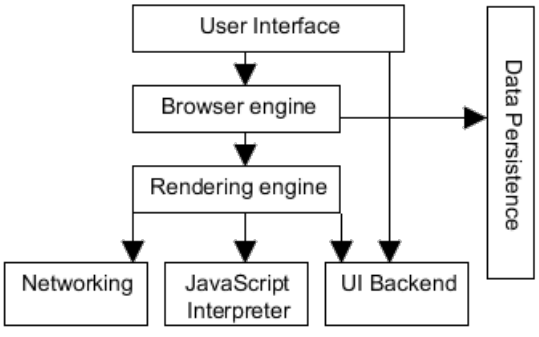

Abbildung 3.1: Schichten eines Browsers [41]

Funktionsweise aktueller Browser von Tali Garsiel herangezogen. Sie hat über einige Jahre Implementierungen von Oben-Source Browsern analysiert und Ihre Erkenntnisse zusammengefasst[41].

Ein Browser besteht demnach aus den in Abbildung 3.1 dargestellten Komponenten. Das UI 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.[13][4]

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. Zu Beginn wird die HTML-Datei interpretiert und in einzelne Elemente aufgeteilt. Aus diesen Elementen entsteht das Document Object Model (DOM) [1] oder auch DOM-Tree genannt. Verweise auf CSS-Dateien werden aufgelöst und sobald möglich ebenfalls interpretiert. Es entstehen weitere Elemente die das CSS Object Model (CSSOM) [2], oder auch CSSOM-Tree genannt, bilden. 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 (Abbildung 3.2).[41][15]

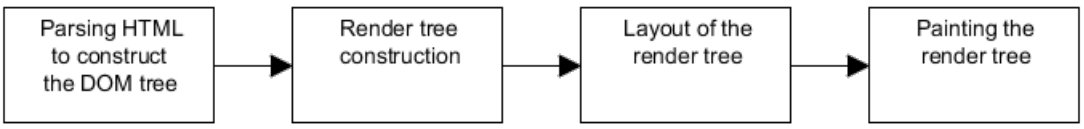

Abbildung 3.2: Workflow eines Browsers [41]

Das Parsen eines HTML-Dokumentes kann bereits durchgeführt werden, wenn erste

[1] "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. This is an overview of DOM-related materials here at W3C and around the web."[47]

[2] "CSSOM defines APIs (including generic parsing and serialization rules) for Media Queries, Selectors, and of course CSS itself."[46]

Teile des Dokumentes über die Netzwerkschnittstelle eingetroffen sind. Dadurch verkürzt sich die Wartezeit für den Anwender[41]. 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 [51, S. 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 Chromium werden beispielsweise maximal 6 parallele Verbindungen zugelassen [39]. Befinden sich mehr externe Ressourcen in einem Dokument, verzögert sich demnach das Herunterladen und Ausführen, bis freie Verbindungen vorhanden sind. Zu dieser Problematik zählen auch Abfragen über XMLHttpRequest (XHR) [3].

Der Begriff SP Anwendung 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 [32]. Bei dieser Art von Anwendung fällt Schritt 1 in Abbildung 3.2 nur beim ersten Aufruf ins Gewicht. Der DOM- und CSSOM-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 oder das Entfernen und Hinzufügen neuer DOM-Elemente kann den Effekt haben, dass die Rendering-Engine einen Reflow[4] durchführen muss. Der Begriff Reflow bezeichnet das neu berechnen der Positionen und das zeichnen aller Elemente. In Abbildung 3.2 bezeichnet dies die Schritte drei und vier. Reflows sind aufgrund des Single-Thread Models von JavaScript blockierend und sollten daher vermieden werden [25]. Kritisch für die Performance einer SP Anwendung ist demnach das Layouten und Rendern, sowie die generelle JavaScript Ausführung.

Performance technisch lassen sich aus diesen Vorgängen bereits grundlegende Regeln ableiten:

- Die Größe des HTML-Dokument bestimmt die Zeit für das Erstellen des DOM-Tree.
- Die Anzahl der CSS- und JavaScript-Verweise beeinflusst die Zeit für das Erstellen des DOM-Tree.

[3] "The XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server."[49]

[4] Der Begriff „Reflow “ bezeichnet einen Prozess der Browser-Engine, der die Positionen und Größen von Elementen im Dokument neu berechnet. Dies findet statt, wenn das gesamte Dokument oder teile daraus neu gezeichnet werden müssen.

- Die Anzahl der CSS-Regeln und deren Komplexität bestimmt die Zeit für die Erstellung des CSSOM-Tree.
- Die Größe des DOM- und CSSOM-Tree bestimmen die Zeit für das Erstellen des Render-Tree.
- Die Anzahl der Elemente im Render-Tree bestimmt die Zeit fürs Layouten
- Die Anzahl der Elemente im Render-Tree bestimmt die Zeit fürs Zeichnen
- Die Komplexität des HTML/CSS beeinflusst demnach das Antwortzeitverhalten
- Das Management externer Ressourcen beeinflusst das Antwortzeitverhalten

3.3 Kategorien im Web

In diesem Abschnitt werden die zuvor genannten Kategorien auf SP Anwendungen übertragen. Das Ergebnis sind generelle Schwachstellen, die in den folgenden Kapiteln berücksichtigt werden müssen und schon erste Erkenntnisse für das Ergebnis dieser Arbeit darstellen. Zum Vergleich mit den allgemeinen Kategorien kann die Tabelle 3.1 herangezogen werden.

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 im Bezug zu JavaScript den Zugriff auf klassische Speichermöglichkeiten wie den Session- oder Local Storage[5] verstehen. Jedoch auch der DOM als persistenter Speicher für den aktuellen Status der View fällt in diese Kategorie. Wie in Abschnitt 3.2 beschrieben können Zugriffe auf Elemente im DOM einen Reflow auslösen. Dies wirkt sich entsprechend negativ auf die Performance aus. Daher ist bei solch einem Zugriff immer darauf zu achten auf welche Attribute der Zugriff erfolgt. Eine Liste aller betroffenen Attribute kann dem Anhang @TODO: Anhang festlegen und erstellen entnommen werden.

[5] Mittels dem Local Storage und Session Storage können JavaScript Anwendungen Daten persistent oder für die Laufzeit einer Session speichern. Die Größe des verfügbaren Speichers ist je nach Browser unterschiedlich, beträgt jedoch mindestens 5mb.[11, S. 589ff.]

Speicherleck (engl. „memory leak“)
Unter einem Speicherleck versteht man einen Bereiche bzw. ein Objekt im Speicher, zu dem es keine Referenz mehr gibt. In JavaScript soll sich der Garbage Collector (GC) [6], um das Freigeben solcher Bereiche kümmern. Dennoch kann es auch in heutigen Browsern noch vorkommen, dass es dem GC 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 lange Zeit genutzt werden. Herkömmliche Webseiten, also Anwendungen mit häufigen Page-Reloads[7], hatten hingegen meist eine kurze Laufzeit, und waren dadurch nicht von diesen Problemen betroffen. Spätestens beim schließend der Seite wurde der gesamte verwendete Speicher freigegeben. Im Hinblick auf SP Anwendungen ist dies jedoch ein größeres Problem. Durch das dynamische Ändern der Seite im Gegensatz zum Aufrufen einer neuen Seite, haben SP Anwendungen meist eine lange Laufzeit. Gelangt man mit solch einer Anwendung an die Grenzen des verfügbaren Speichers, wird der GC zu nehmend aktiv, was sich negativ auf das Antwortzeitverhalten auswirkt. Außerdem können dynamische Inhalte nicht erstellt werden, wenn der verfügbare Arbeitsspeicher vollständig belegt ist.[22][5]

Reinitialisierungsproblem
Dieses Problem bezieht sich auf die Verwendung von Variablen, die nicht initialisiert wurden. Aufgrund von daraus resultierenden Fehlern können Probleme mit der Performance entstehen. Bei Programmiersprachen ohne automatische Speicherverwaltung 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 dem Wert *undefined* initialisiert werden [11, S. 41ff.].

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 gehört der asynchrone XHR-Zugriff in diese Kategorie. Wie in Abschnitt 3.2 beschrieben öffnet jeder Browser nur eine maximale Anzahl

[6] Der Garbage Collector ist in speicherverwalteten Sprachen für das Freigegeben von nicht mehr verwendeten Speicherbereichen zuständig.
[7] Herkömmliche Webseiten rufen bei Änderungen die komplette Seite neu auf, anstatt nur den betroffenen Bereich zu aktualisieren.

paralleler Verbindungen. Werden mehr Anfragen gestellt als Antworten eintreffen, bildet sich eine Warteschlange und das Antwortzeitverhalten verschlechtert sich.

Ein weiteres Problem in dieser Kategorie ist das Single-Thread Model von JavaScript. Lange Algorithmen oder synchrone XHR-Zugriffe blockieren diesen einen Thread, wodurch die Anwendung einfriert.

Intermittierende Probleme
Durch die Auslastung von Systemressourcen erhöht sich das Antwortzeitverhalten. Im Bezug zu Webtechnologien passt der Reflow-Prozess in diese Kategorie. Werden durch Aktionen in JavaScript Reflows ausgelöst, leidet darunter die Reaktionsfähigkeit der Anwendung. Anstatt auf Nutzerinteraktionen zu reagieren, ist die Rendering-Engine mit dem Layout und Zeichnen der Oberfläche beschäftigt.

3.4 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 Wartezeit des Anwenders zu minimieren und die Anwendung bei Animationen flüssig wirken zu lassen. Flüssig in diesem Zusammenhang bedeutet bei einer konstanten Bildwiederholungsrate von 60 Frames Per Second (FPS) . Bei zu niedrigen 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 sowie das Layouten und Zeichnen des Render-Tree statt. Für die eigentliche Geschäftslogik bleibt demnach nicht viel Zeit. Es gilt daher Prozesse und Implementierungen bestmöglich zu optimieren.[29]

4 Vorbereitung

4.1 Performancemessung

Wenn es um den Vergleich verschiedener Implementierungsansätze geht, sind Messungen der Performance ein wichtiges Kriterium. In diesem Abschnitt wird daher die Herangehensweise und Methodik bei Messungen in dieser Arbeit beschrieben. Grundsätzlich werden aufgrund der Problematiken in Abschnitt 2 für alle Messungen zwei Testgeräte verwendet. Ein Gerät mit Android in der Version 4.4 und eines mit Android in der Version 4.3. Dadurch werden die Unterschiede noch einmal klar sichtbar. Aufgrund fehlender Ressourcen erfolgt kein Test mit iOS Geräten.

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

- **Hintergrundprozesse**
- **Garbage Collection**
  Der Garbage Collector sorgt für das Freigeben nicht mehr verwendeter Speicherbereiche. Da eine JavaScript-Engine auf dem Single-Thread Modell basiert, verzögert der Garbage Collector die aktuelle JavaScript Ausführung. Der Entwickler hat keine Möglichkeit zu steuern wann der Garbage Collector seine Arbeit beginnt. Zeitmessungen können dadurch verfälscht werden.[30]
- **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 und zulassen. Diese Optimierung übernimmt im Anschluss der zweite Compiler. Durch optimierten Maschinencode können so sehr schnelle Ausführungsgeschwindigkeiten erreicht werden.[24]

Diese Probleme führen dazu, dass ein eine einfache Messung der Zeit für einen Vorgang mit n Wiederholungen und einem Durchschnitt nicht ausreicht [7][35]. Die JavaScript-Library BenchmarkJS wurde aufgrund dieser Probleme entwickelt und liefert für Zeitmessungen statistisch relevante Ergebnisse. Eine einfache Schnittstelle ermöglicht es dem Entwickler Tests der Performance durchzuführen, ohne Kenntnisse über diese Problematiken oder statistische Messverfahren zu besitzen [26]. Um Laufzeiten zu Messen führt man in BenchmarkJS einen Benchmark durch. Innerhalb dieses Benchmarks finden mehrere Test-Zyklen inklusive Berechnung aktueller statistischer Werte statt. Es werden so lange Test-Zyklen durchgeführt, bis eine maximale Zeitgrenze überschritten wird. Diese Grenze ist standardmäßig auf 5 Sekunden festgelegt, kann aber durch den Entwickler geändert werden [27]. Hat der zu testende Code eine lange Laufzeit, werden weniger Test-Zyklen durchgeführt als bei kurzen Laufzeiten.[6]

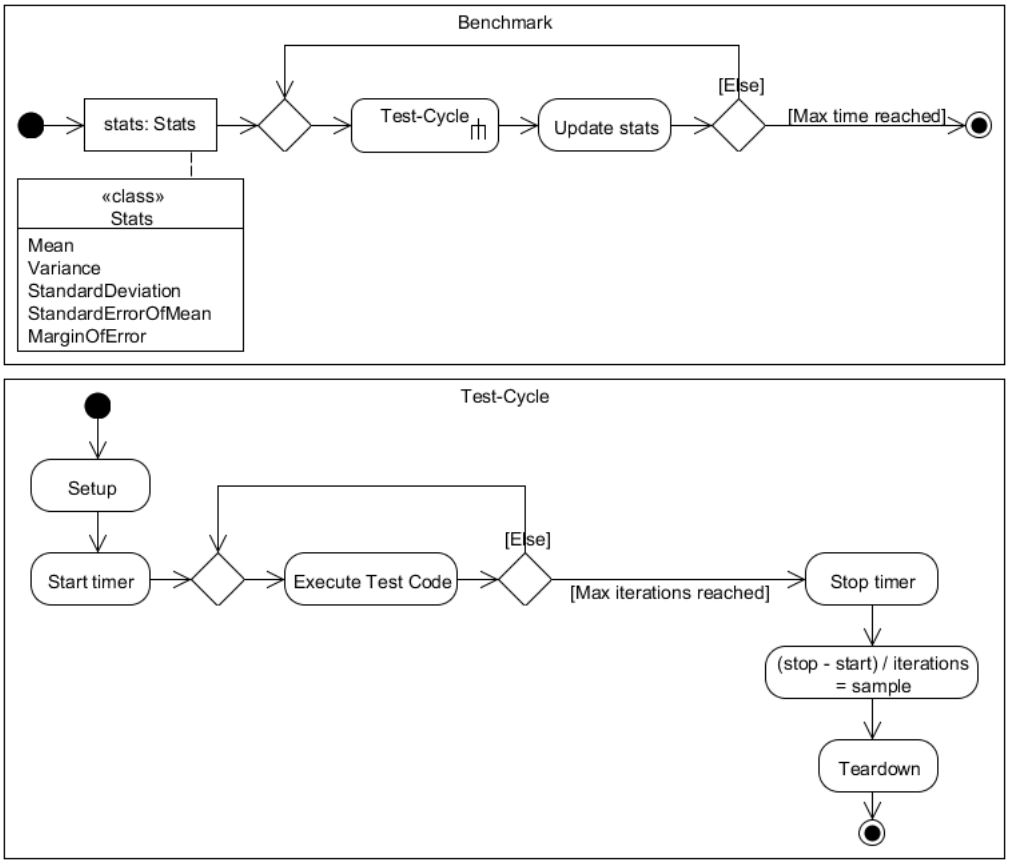

Abbildung 4.1: BenchmarkJS Benchmark Prozess[20]

Innerhalb jedes Test-Zyklus wird der zu testende Code in einer Schleife durchlaufen.

Vor der Codeausführung findet eine Setup-Phase zum initialisieren des Tests und im Anschluss eine Teardown-Phase zum Aufräumen des Tests statt. Das Ergebnis eines Test-Zyklus ist ein sogenanntes Sample. Das Sample berechnet sich durch die Formel $\frac{Ende-Start}{Iterationen}$. Die Anzahl der Iterationen wird von BenchmarkJS dynamisch angepasst. Nach jedem Test-Zyklus wird das Sample in einer Liste gespeichert um anschließend statistische Werte auf Basis dieser Liste zu berechnen (Tabelle 4.1) [6][36]. Abbildung 4.1 zeigt schematisch den Ablauf von BenchmarkJS.

Durch die Berechnung des Durchschnitts und der Standard-Abweichung aller Samples ergibt sich der relative Fehler mittels einer T-Verteilung [33, S. 440ff.]. Ist dieser relative Fehler größer als 1%, erhöht BenchmarkJS die Anzahl der Iterationen in einem Test-Zyklus um bessere Ergebnisse zu erzielen [6]. Störungen durch Hintergrundprozesse fallen dadurch weniger ins Gewicht.

Bezeichnung	Formel	
Durchschnitt	$mean = \frac{\sum_{i=0}^{array.length} array[i]}{array.length}$	(4.1)
Standard Abweichung	$sd = \sqrt{\frac{\sum_{i=0}^{array.length} (array[i] - mean)^2}{array.length - 1}}$	(4.2)
Standard Fehler des Durchschnitts	$sem = \frac{sd}{\sqrt{array.length}}$	(4.3)
Freiheitsgrad	$df = array.length - 1$	(4.4)
Fehler Rahmen	$moe = sem * tDistribution(df)$	(4.5)
Relativer Fehler Rahmen	$rme = \frac{moe * 100}{mean}$	(4.6)

Tabelle 4.1: Formeln der statistischen Berechnungen von BenchmarkJS[6]

In den folgenden Kapiteln werden zum Teil sehr unterschiedliche Implementierungen

durch Zeitmessungen untersucht. Diese Implementierungen unterscheiden sich durch Asynchrones- und Synchrones-Verhalten. Für beide Varianten bietet BenchmarkJS eine Schnittstelle an. Listing 4.1 und 4.2 zeigen jeweils die Verwendung von BenchmarkJS für einen synchronen und einen asynchronen Vorgang. Der asynchrone Vorgang unterscheidet sich dadurch, dass BenchmarkJS das Ende der Codeausführung nicht selbst feststellen kann. Aus diesem Grund muss bei der asynchronen Variante nach der Codeausführung ein Promise[1] abgeschlossen werden. Somit lassen sich alle in den folgenden Kapiteln durchgeführten Tests hinsichtlich ihrer Performance messen.

Listing 4.1 BenchmarkJS synchrone Benchmarks

```
// Synchrone Implementierung
var bench = new Benchmark('', {
  'fn': function() {
    // Test code
  },
  'onComplete': function() {
    var meanInMs = this.stats.mean * 1000;
  }
});
bench.run();
```

Listing 4.2 BenchmarkJS asynchrone Benchmarks

```
// Asynchrone implementierung
var bench = new Benchmark('', {
  'defer': true,
  'fn': function(deferred) {
    doAsyncWork(function(){
      // Benchrichtigt BenchmarkJS darüber,
      // dass der Durchlauf fertig ist.
      deferred.resolve();
    });
  },
  'onComplete': function() {
    var meanInMs = this.stats.mean * 1000;
  }
});
bench.run();
```

[1] Promise bezeichnet ein Pattern um Ergebnisse eine asynchronen Vorgangs abzufragen.

4.2 Testanwendung

In den folgenden Kapiteln werden Implementierungen für verschiedene Anwendungsszenarien beschrieben und verglichen. All diese verschiedenen Implementierungen sollen in einer einzigen App zusammengefasst und dadurch leicht auf verschiedenen Geräten getestet werden können. Aus diesem Grund wird in diesem Abschnitt eine App beschrieben, die Modular aufgebaut ist und es somit ermöglicht leicht alle Implementierungen in einer App zu testen. Im Folgenden wird jeder Implementierungsansatz als Modul bezeichnet. Die App basiert grundsätzlich auf den gleichen Technologien und Frameworks wie sie in dieser Arbeit betrachtet werden. Zusätzlich wird neben der bereits beschriebene Library BenchmarkJS, die Library LoDash[2] verwendet. LoDash enthält viele Funktionen für die Interaktion mit Arrays und Objekten.

Um die Navigation zu vereinfachen folgt die Oberfläche der App einem Master-Detail Ansatz. Nach dem Start wird dem Anwender eine Liste aller Module angezeigt. Jedes Modul dieser Liste besitzt einen eindeutigen Namen und verweist auf eine Seite, die den Test für das jeweilige Modul beinhaltet. Die Seiten der einzelnen Module werden individuell und dem Test entsprechend gestaltet. Über einen Button in der linken oberen Ecke kann der Anwender zurück zur Liste der Module navigieren. Thematisch zusammenhängende Module werden über eine Kodierung im Namen Gruppiert. Um

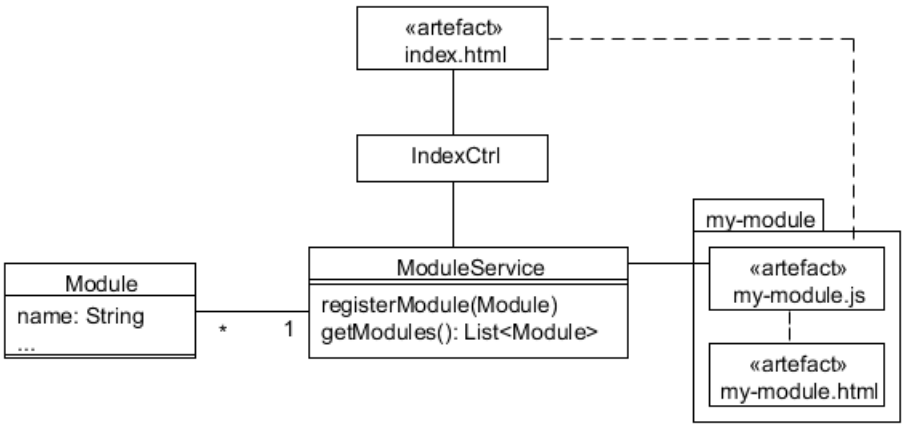

Abbildung 4.2: Struktur der Testanwendung

die Ergebnisse eines Tests nicht zu beeinflussen, müssen Tests isoliert von einander ausgeführt werden. Dies setzt voraus, dass die einzelnen Module untereinander keine und mit der App möglichst wenige Schnittpunkte besitzen. Außerdem ist es einfacher

[2] https://lodash.com/

neue Module hinzuzufügen, wenn die App an sich nicht geändert werden muss. Damit die App diesen Anforderungen genügt, 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 App als Skript referenziert werden (*index.html*). Dies ist der einzige Eingriff in die Implementierung der App. In der JavaScript-Datei, wird das Modul in einem globalen Service registriert und dabei einem Zustandsgraphen für das Routing[3] in der Anwendung hinzugefügt. Beim Start der Anwendung ruft die App, alle registrierten Module diesem globalen Service ab und zeigt sie mit ihrem Namen in der Liste an. Klickt der Anwender auf ein Modul, wechselt die App über den Routing-Mechanismus zur dessen HTML-Datei. Dort werden Interaktionen und Informationen für den jeweilige Test angezeigt. Für das erstellen neuer Module, reicht es demnach aus die beiden Datei anzulegen, in der *index.html* zu referenzieren und per JavaScript zu registrieren. Module haben keine implizite Verknüpfung untereinander oder mit der Anwendung und werden daher grundsätzlich isoliert voneinander ausgeführt. Um weitere Seiteneffekte zu vermeiden werden in der App sämtliche von AngularJS durchgeführte Optimierungen und Animationen ausgeschaltet. Dazu zählt das Zwischenspeichern bereits besuchter Seiten, sowie das vorzeitige Laden von Templates.

[3] Der Begriff Routing bezeichnet die Navigationshierarchie einer Anwendung.

5 Anwendungsbestandteile

In diesem Kapitel werden nacheinander verschiedene Implementierungsansätze verglichen, analysiert und ihr bestmöglicher Einsatzzweck definiert. Die Entscheidung für oder gegen eine Implementierung erfolgt dabei auf Basis der Performance, des Nutzererlebnisses und anderen spezifischen Vor- und Nachteilen.

5.1 Dynamische Views

Mit dynamischen Views sind Seiten einer Anwendung gemeint, deren Inhalt sich zur Laufzeit aufgrund von Nutzerinteraktionen ändert. In AngularJS wird solch eine dynamische Verhaltensweise über das in Abschnitt 1 beschriebene Two-Way Databinding realisiert. Dieses Konzept ermöglicht das automatische Synchronisieren von Views und Models. Durch das setzten von Eigenschaften in einem Model ändert sich die Anzeige in der View. Umgekehrt werden Änderungen in der View, beispielsweise durch Benutzereingaben, automatisch ins Model übertragen. Diese beiden Komponenten existieren unabhängig voneinander. Die View ist somit eine Projektion der Daten im Model. Dies fördert zum einen die Testbarkeit und zum anderen die Komponentenbildung[1]. Um solch eine Synchronisation zu realisieren ist natürlich ein gewisser Overhead erforderlich. Dies führt dazu, dass sich bei zu intensiver Nutzung des Two-Way Databinding das Antwortzeitverhalten der Anwendung in bestimmten Situationen verschlechtert. Aus diesem Grund werden im folgenden Herangehensweisen betrachtet, wie dynamische Views möglichst effizient implementiert werden können. Doch dazu ist es zunächst notwendig die Technik hinter diesem Konzept näher zu betrachten.

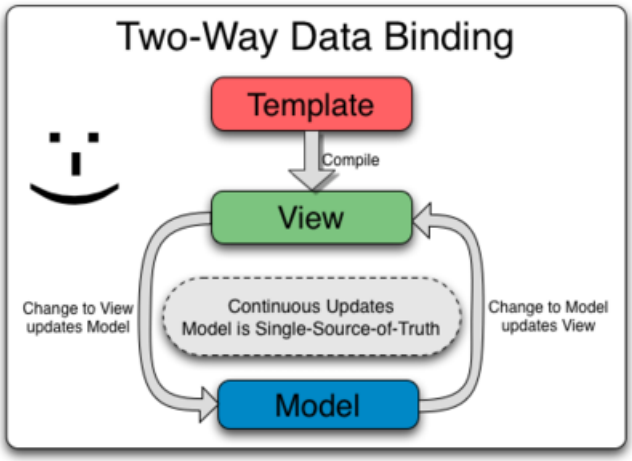

Abbildung 5.1: AngularJS Two-Way Databinding[1]

Prinzipiell gibt es derzeit zwei verschiedene Konzepte, wie ein Two-Way Databinding implementiert werden kann[40]. Das erste Konzept arbeitet mit Gettern und Settern und basiert auf dem Observer-Pattern. Eine Änderung an einem JavaScript Objekt wird dabei nur über eine entsprechende Setter-Methode durchgeführt. Registrierte Observer können dadurch bei Änderungen über diese Methode benachrichtigt werden. Das zweite Konzept verwendet hingegen externe Monitoring, auch „Dirty-Checking“ genannt. Dieses Konzept wird in AngularJS eingesetzt. Auch hier registrieren sich Observer um

über Änderungen an einem Objekt informiert zu werden. Die Prüfung ob sich ein Wert geändert hat, erfolgt jedoch nicht im Objekt selbst, sondern in einem Prüfmechanismus. Zu bestimmten Zeitpunkten wird dieser Mechanismus ausgeführt. Dabei werden alle beobachteten Eigenschaften auf Änderungen geprüft. In AngularJS wird dieser Mechanismus in einer Methode namens *$digest()* ausgeführt und trägt daher den Namen $digest-Zyklus [40]. Über die Methode *$watch()* registrieren sich Observer um Eigenschaften zu überwachen. In der Regel werden Eigenschaften eines Models überwacht. In AngularJS trägt solch ein Model den Namen *$scope*. Unterschiedliche Direktiven, die in einem HTML-Template platziert werden, erzeugen automatisch solch ein Model. Dadurch entsteht eine Hierarchie von Model Objekten entlang der HTML Hierarchie [3]. Jedes dieser Models besitzt eine eigene *$watch*, *$digest* und *$apply* Methode. Die Methode *$digest* löst dabei den Prüfmechanismus für sich selbst und seine Kinder aus. Die MEthode *$apply* hingegen löst den Prüfmechanismus für das in der Hierarchie oberste Model aus und ist demnach langsamer. [31]

Generell ruft AngularJS die Methode *$apply* automatisch auf, wenn über entsprechende Direktiven wie *ng-click* auf Nutzerinteraktionen reagiert wurde. Gerade dies ist jedoch ein kritischer Moment, da Nutzerinteraktionen meist Animationen oder Änderungen an dem UI zur Folge haben.

Aufgrund dieser Vorgehensweise lässt sich demnach die Problematik bei dynamischen Views in AngularJS ableiten. Denn je mehr Eigenschaften beobachtet werden, desto länger dauert der Prüfvorgang. Flüssige Animationen oder Änderungen an dem UI können dadurch beeinträchtigt werden. Das Ziel ist es daher zur selben Zeit möglichst wenig Eigenschaften zu beobachten.

5.1.1 Analyse

Es gilt nun zu untersuchen, wie sich der Einsatz des Two-Way Databinding auf das Antwortzeitverhalten einer App auswirkt. Da der Prüfvorgang hauptsächlich bei Interaktionen mit der Anwendung durchgeführt wird, sind flüssige Animationen besonders von den Problemen betroffen. In diesem Abschnitt werden daher Richtwerte ermittelt, ab denen bestimmte Optimierungen durchgeführt werden sollten.

Im ersten Schritt wird gemessen, wie viel Zeit ein Prüfvorgang bei einer unterschiedlichen Anzahl von Beobachteten Werten in Anspruch nimmt. Entsprechend der Struktur der Testanwendung in Abschnitt 4.2 wird für diesen Testfall ein Modul angelegt. Für die Laufzeitmessung befindet sich in der View eine Liste mit Beispieldaten. Im oberen

Teil kann über ein Eingabefeld die Anzahl der Elemente eingegeben und die Liste damit befüllt werden. Anschließend wird die Laufzeit des Prüfvorgangs für diese View über BenchmarkJS ermittelt und ausgegeben (Listing 5.1). Bei den Beispieldaten handelt es sich um eine einfache Bücherliste mit einem Bild, dem Titel, den Autoren, der Beschreibung, dem Preis und der Verfügbarkeit (Abbildung **??**). Wenn das Buch lieferbar ist, wird über CSS der Titel grün hinterlegt. Insgesamt werden durch dieses Template für jedes Buch der Liste sechs Eigenschaften beobachtet.

Listing 5.1 Dynamische Views, Laufzeitmessung mit BenchmarkJS

```
1  function calculate($scope, callback) {
2    var bench = new Benchmark('', {
3      'fn': function() {
4        this.injectedScope().$digest();
5      },
6      'onComplete': function() {
7        callback(this.stats.mean * 1000)
8      }
9    });
10   Benchmark.prototype.injectedScope = function() {
11     return $scope;
12   };
13   bench.run();
14 }
```

In Kombination mit der Anzahl an Elementen in der Liste ergeben sich dadurch die Laufzeiten der Prüfvorgänge in Tabelle 5.1. Die Spalte „% Frame“ gibt den Anteil der Laufzeiten an einem Frame mit der Zeitspanne 10ms an (Siehe Abschnitt 3.4).

# Listen Elemente	# Beobachtete Elemente	Android 4.3		Android 4.4	
		Laufzeiten (ms)	% Frame	Laufzeiten (ms)	% Frame
1	6				
10	60				
20	120				
30	180				
50	300				
100	600				
150	900				
200	1200				
500	3000				

Tabelle 5.1: Dynamische Views, BenchmarkJS Laufzeitanalyse

@TODO: Ergebnis beschreiben

5.1.2 Lösungsansatz

Um die Durchführung des Prüfvorgans zu beschleunigen, muss die Anzahl der Beobachteten Eigenschaften minimiert werden. Ein erster Schritt in diese Richtung ist das betrachten der Daten. Man kann Daten an dieser Stelle in statische und dynamische Daten einteilen. Statischen Daten sind Informationen, die sich zur Laufzeit der Anwendung nicht ändern. Sie dienen als reine Informationsquelle für den Anwender. Dynamische Daten hingegen können sich durch Interaktionen des Anwenders mit dem UI, oder durch andere Ereignisse ändern. Aufgrund dieser Eigenschaft ist ein Two-Way Databinding für statische Daten überflüssig. Das beobachten dieser Daten bremst den Prüfvorgang unnötig aus. Um eine View mit vielen statischen Daten zu optimieren, ist es demnach notwendig das Two-Way Databinding zu entfernen.

Das Konzept für diese Umsetzung nennt sich One-Time Binding. Eigenschaften werden dabei weiterhin beobachtet, bis die Daten einmalig in der View angezeigt wurden. Anschließend wird die Beobachtung entfernt und somit folgende Prüfvorgänge nicht länger beeinflusst. Der Vorteil ist, dass weiterhin die AngularJS Syntax in Templates verwendet werden kann und trotzdem die Performance verbessert wird. AngularJS unterstützt ab Version 1.3 nativ eine Syntax für One-Time Bindings [34]. Listing 5.3 zeigt dessen Verwendung für die zuvor beschriebene Bücherliste. Entscheidend dabei sind zwei Doppelpunkte zu beginn einer AngularJS AngularJS Expression[1]. Bei der Verwendung einer AngularJS Version < 1.3 kann die Library Bindonce eingesetzt werden [45]. Listing 5.2 zeigt eine entsprechende Implementierung. Bindonce setzt dabei auf eigene Direktiven und erzeugt bei der Verwendung keinen nennenswerten Overhead.

Listing 5.2 Dynamische Views, One-Time Binding mit Bindonce

```
1 <ion-item class="..." ng-repeat="book in books">
2   <img bo-src="book.image"></img>
3   <h2 bo-class="{ onStock: book.onStock > 0, notOnStock: book.onStock
        <= 0 }"
4     bo-text="book.name"></h2>
5   <h3 bo-text="book.author"></h3>
6   <p bo-text="book.description"></p>
7   <p style="..." bo-text="book.price"></p>
8 </ion-item>
```

[1] Durch AngularJS Expressions lassen sich Daten aus einem Model in das Template einer View einfügen. Standardmäßig findet dabei automatisch eine Verknüpfung mit dem Model über ein Two-Way Databinding statt. AngularJS ersetzt die Expressions dabei in seinem Kompilier-Prozess durch konkrete Werte.

Listing 5.3 Dynamische Views, One-Time Binding mit AngularJS

```
<ion-item class="..." ng-repeat="book in books">
<img ng-src="{{::book.image}}"></img>
<h2 ng-class="::{ onStock: book.onStock > 0, notOnStock: book.onStock
    <= 0 }">{{::book.name}}</h2>
<h3>{{::book.author}}</h3>
<p>{{::book.description}}</p>
<p style="...">{{::book.price}}</p>
</ion-item>
```

Befinden sich viele dynamische Daten in einer View, ist die Optimierung nicht ganz so einfach. Dennoch gibt es auch hier verschiedene Möglichkeiten eine Optimierung vorzunehmen. Generell gilt, die Anzahl der DOM-Elementen, die sich zur selben Zeit in einer View befinden, zu minimieren. Wie in Abschnitt 3.2 festgestellt hat dies einen positiven Effekt auf die Performance. Dies gilt im Bezug zum Two-Way Databinding vor allem für DOM-Elemente, die davon Gebrauch machen.

Ein klassisches Verfahren um die Anzahl vom DOM-Elementen in einer View zu verringern ist der Einsatz eines ausklappbaren Bereiches. Der Inhalt dieses Bereiches kann beim Aufklappen in den DOM eingefügt werden und beeinflusst erst ab diesem Zeitpunkt den Prüfvorgang beim Two-Way Databinding. Alternativ können Inhalte auf Detail-Seiten ausgelagert werden um die Komplexität zu verringern. Bei Listen empfiehlt sich der Einsatz der Virtual-Scrolling Technik, wie es in Kapitel @TODO: Kapitel ergänzen beschrieben ist.

Eine weitere Microoptimierung bei dynamischen Daten kann durch die Begrenzung des Prüfvorgangs auf lokale Model erreicht werden. Wie zuvor beschrieben besitzt jedes Model die Methode *$apply* um den Prüfvorgang für die gesamte Modellhierarchie durchzuführen. Bei ausschließlich lokalen Änderungen ist dies ein Overhead, der eingespart werden kann.

Literatur

[1] AngularJS. *Data Binding*. URL: https://docs.angularjs.org/guide/databinding (besucht am 16. 03. 2015).

[2] AngularJS. *Dependency Injection*. URL: https://docs.angularjs.org/guide/di (besucht am 16. 03. 2015).

[3] AngularJS. *Understanding Scopes*. 22. Dez. 2014. URL: https://github.com/angular/angular.js/wiki/Understanding-Scopes (besucht am 16. 03. 2015).

[4] Apple Inc. *WebKit*. URL: http://www.webkit.org/ (besucht am 16. 03. 2015).

[5] Dolmar Ben. *Understand memory leaks in JavaScript applications*. IBM. 6. Nov. 2012. URL: http://www.ibm.com/developerworks/library/wa-jsmemory/ (besucht am 16. 03. 2015).

[6] *BenchmarkJS Quelltext*. URL: https://github.com/bestiejs/benchmark.js/blob/2c1d5d4cda2e58fc29da1293e8d6cb92d453983d/benchmark.js#L2837 (besucht am 16. 03. 2015).

[7] Mathias Bynens. *Bulletproof JavaScript benchmarks*. 23. Dez. 2010. URL: https://mathiasbynens.be/notes/javascript-benchmarking (besucht am 16. 03. 2015).

[8] Drew Crawford. *Why mobile web apps are slow*. 9. Juli 2013. URL: http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/ (besucht am 16. 03. 2015).

[9] Jonathan Dann. *Under the hood: Rebuilding Facebook for iOS*. Facebook. 23. Aug. 2012. URL: https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-ios/10151036091753920 (besucht am 16. 03. 2015).

[10] Drifty. *Ionic Documentation Overview*. URL: http://ionicframework.com/docs/overview/ (besucht am 16. 03. 2015).

[11] David Flanagan. *JavaScript - The Definitive Guide*. 6. Aufl. Sebastopol: Ö'Reilly Media, Inc.", 2011. ISBN: 978-0-596-80552-4.

[12] Google. *Android Plattform Versionsverteilung*. URL: https://developer.android.com/about/dashboards/index.html#Platform (besucht am 16. 03. 2015).

[13] Google. *Blink*. URL: http://www.chromium.org/blink (besucht am 16. 03. 2015).

[14] Google. *Class WebView*. URL: https://developer.android.com/reference/android/webkit/WebView.html (besucht am 16. 03. 2015).

[15] Google. *Erstellung der Rendering-Baumstruktur, Layout, und Paint*. URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=de (besucht am 16. 03. 2015).

[16] Google. *Migrating to WebView in Android 4.4*. URL: https://developer.android.com/guide/webapps/migrating.html (besucht am 16. 03. 2015).

[17] Google. *The Chromium Projects*. URL: http://www.chromium.org/Home (besucht am 16. 03. 2015).

[18] Google. *WebView for Android*. URL: https://developer.chrome.com/multidevice/webview/overview (besucht am 16. 03. 2015).

[19] Google. *What Is Angular?* URL: https://docs.angularjs.org/guide/introduction (besucht am 16. 03. 2015).

[20] Monsur Hossain. *benchmark.js: how it works*. URL: http://monsur.hossa.in/2012/12/11/benchmarkjs.html (besucht am 16. 03. 2015).

[21] Intel Open Source Technology Center. *Frequently-asked questions*. URL: https://crosswalk-project.org/documentation/about/faq.html (besucht am 16. 03. 2015).

[22] Ilya Kantor. *Memory leaks*. URL: http://javascript.info/tutorial/memory-leaks (besucht am 16. 03. 2015).

[23] Mirko Novakovic and Robert Spielmann and Tobias Knierim. *PERFORMANCE ANALYSE UND OPTIMIERUNG IN DER SOFTWAREENTWICKLUNG*. codecentric AG. 1. Apr. 2007. URL: https://www.codecentric.de/kompetenzen/publikationen/performance-analyse-und-optimierung-in-der-softwareentwicklung/ (besucht am 16. 03. 2015).

[24] Jakob Kummerow. *Better code optimization decisions for V8*. 1. Mai 2012. URL: http://blog.chromium.org/2012/05/better-code-optimization-decisions-for.html (besucht am 16. 03. 2015).

[25] Lindsey Simon, UX Developer. *Minimizing browser reflow*. 28. Aug. 2014. URL: https://developers.google.com/speed/articles/reflow (besucht am 16. 03. 2015).

[26] Mathias Bynens, John-David Dalton. *BenchmarkJS*. URL: http://benchmarkjs.com/ (besucht am 16. 03. 2015).

[27] Mathias Bynens, John-David Dalton. *Benchmark.js Dokumentation.* URL: `http://benchmarkjs.com/docs` (besucht am 16. 03. 2015).

[28] J. O'Dell. *Why LinkedIn dumped HTML5 and went native for its mobile apps.* LinkedIn. 17. Apr. 2013. URL: `http://venturebeat.com/2013/04/17/linkedin-mobile-web-breakup/` (besucht am 16. 03. 2015).

[29] Addy Osmani. *Gone In 60 Frames Per Second: A Pinterest Paint Performance Case Study.* Pinterest. 10. Juni 2013. URL: `http://www.smashingmagazine.com/2013/06/10/pinterest-paint-performance-case-study/` (besucht am 16. 03. 2015).

[30] Addy Osmani. *Writing Fast, Memory-Efficient JavaScript.* 5. Nov. 2012. URL: `http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/` (besucht am 16. 03. 2015).

[31] Sandeep Panda. *Understanding Angular's apply()anddigest().* 7. Mai 2014. URL: `http://www.sitepoint.com/understanding-angulars-apply-digest/` (besucht am 16. 03. 2015).

[32] John Papa. *SPA and the Single Page Myth.* 30. Nov. 2013. URL: `http://www.johnpapa.net/pageinspa/` (besucht am 16. 03. 2015).

[33] Lothar Papula. *Mathematik für ingenieure und naturwissenschaftler* -. 4. Aufl. Wiesbaden: Vieweg, 2001. ISBN: 978-3-528-34937-0.

[34] Tilman Potthof. *AngularJS 1.3 Performance - Einmaliges (One-Time) Databinding.* URL: `https://angularjs.de/blog/angularjs-one-time-binding` (besucht am 16. 03. 2015).

[35] John Resig. *JavaScript Benchmark Quality.* 6. Sep. 2008. URL: `http://ejohn.org/blog/javascript-benchmark-quality/` (besucht am 16. 03. 2015).

[36] Prof. Michael Richmond. *Examples of Uncertainty calculations.* Rochester Institute of Technology. 17. Juli 2003. URL: `http://www.smashingmagazine.com/2013/06/10/pinterest-paint-performance-case-study/` (besucht am 16. 03. 2015).

[37] Tim Roes. *Old WebView vs. Chromium backed WebView Benchmark.* 23. Nov. 2013. URL: `https://www.timroes.de/2013/11/23/old-webview-vs-chromium-webview/` (besucht am 16. 03. 2015).

[38] Connie Smith, Lloyd G. Williams und Lloyd Williams. *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software.* Revised 2003. Amsterdam: Addison-Wesley, 2001. ISBN: 978-0-201-72229-1.

[39] Steve Souders. *Roundup on Parallel Connections*. 20. März 2008. URL: http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/ (besucht am 16. 03. 2015).

[40] Boris Staal. *2-Way Data Binding under the Microscope*. 5. Feb. 2014. URL: http://staal.io/blog/2014/02/05/2-way-data-binding-under-the-microscope/ (besucht am 16. 03. 2015).

[41] Tali Garsiel, Paul Irish. *How Browsers Work: Behind the scenes of modern web browsers*. 5. Aug. 2011. URL: http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/ (besucht am 16. 03. 2015).

[42] The Apache Software Foundation. *Android Platform Guide*. URL: http://cordova.apache.org/docs/en/4.0.0/guide_platforms_android_index.md.html#Android%20Platform%20Guide (besucht am 16. 03. 2015).

[43] The Apache Software Foundation. *Cordova*. URL: http://cordova.apache.org/docs/en/4.0.0/guide_overview_index.md.html#Overview (besucht am 16. 03. 2015).

[44] The Apache Software Foundation. *iOS Platform Guide*. URL: http://cordova.apache.org/docs/en/4.0.0/guide_platforms_ios_index.md.html#iOS%20Platform%20Guide (besucht am 16. 03. 2015).

[45] Pasquale Vazzana. *Github: High performance binding for AngularJs*. URL: https://github.com/Pasvaz/bindonce (besucht am 16. 03. 2015).

[46] W3C. *CSS Object Model (CSSOM)*. 20. Feb. 2015. URL: http://dev.w3.org/csswg/cssom/ (besucht am 16. 03. 2015).

[47] W3C. *Document Object Model (DOM)*. 19. Jan. 2005. URL: http://www.w3.org/DOM/ (besucht am 16. 03. 2015).

[48] W3C. *Document Object Model (DOM) Technical Reports*. URL: http://www.w3.org/DOM/DOMTR (besucht am 16. 03. 2015).

[49] W3C. *XMLHttpRequest Level 1*. 30. Jan. 2014. URL: http://www.w3.org/TR/XMLHttpRequest/ (besucht am 16. 03. 2015).

[50] WhatWG. *HTML Living Standard - Parsing HTML documents*. 14. März 2015. URL: https://html.spec.whatwg.org/multipage/syntax.html#parsing (besucht am 16. 03. 2015).

[51] Nicholas C. Zakas. *High Performance JavaScript*. Sebastopol: Ö'Reilly Media, Inc.", 2010. ISBN: 978-1-449-38874-4.

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat nach meinem Wissen mit gleichem Inhalt noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 14. April 2015

Marco Busemann

Acronyms

API Application Programming Interface 1

CSS Cascading Style Sheets 1, 7–9, 21

CSSOM CSS Object Model 7–9, *Glossar:* CSSOM

DI DI 2, *Glossar:* DI

DOM Document Object Model 7–9, 23, *Glossar:* DOM

FPS Frames Per Second 11

GC Garbage Collector 10, *Glossar:* GC

HTML Hypertext Markup Language 1, 2, 7–9, 17, 20

MVC Model View Controller 2

SP Single Page 1, 5, 8–10

SPA Single Page Application *Glossar:* SPA

UI User Interface 1, 2, 7, 20, 22

XHR XMLHttpRequest 8, 10, 11, *Glossar:* XHR

Glossar

AngularJS Expression Durch AngularJS Expressions lassen sich Daten aus einem Model in das Template einer View einfügen. Standardmäßig findet dabei automatisch eine Verknüpfung mit dem Model über ein Two-Way Databinding statt. AngularJS ersetzt die Expressions dabei in seinem Kompilier-Prozess durch konkrete Werte. 22

CSSOM "CSSOM defines APIs (including generic parsing and serialization rules) for Media Queries, Selectors, and of course CSS itself."[46] 7

DI „Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. “[2] 2

DOM "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. This is an overview of DOM-related materials here at W3C and around the web."[47] 7

GC Der Garbage Collector ist in speicherverwalteten Sprachen für das Freigegeben von nicht mehr verwendeten Speicherbereichen zuständig. 10

Local Storage Mittels dem Local Storage und Session Storage können JavaScript Anwendungen Daten persistent oder für die Laufzeit einer Session speichern. Die Größe des verfügbaren Speichers ist je nach Browser unterschiedlich, beträgt jedoch mindestens 5mb.[11, S. 589ff.] 9

Markup Der Begriff Markup beschreibt eine Sprache, die zur Gliederung und Formatierung von Texten oder anderen Daten eingesetzt wird und maschinell gelesen werden kann. 2

Page-Reloads Herkömmliche Webseiten rufen bei Änderungen die komplette Seite neu auf, anstatt nur den betroffenen Bereich zu aktualisieren. 10

Promise Promise bezeichnet ein Pattern um Ergebnisse eine asynchronen Vorgangs abzufragen. 15

Reflow Der Begriff „Reflow “ bezeichnet einen Prozess der Browser-Engine, der die Positionen und Größen von Elementen im Dokument neu berechnet. Dies findet statt, wenn das gesamte Dokument oder teile daraus neu gezeichnet werden müssen. 8, 9, 11

Routing Der Begriff Routing bezeichnet die Navigationshierarchie einer Anwendung. 17

WebView Ein WebView ist eine UI Komponente, die Webseiten wie in einem Browser darstellen und JavaScript ausführen kann. 1, 3, 4

XHR "The XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server."[49] 8