Bachelorarbeit im Studiengang Online-Medien-Management

Evaluation von Technologien zur Reduktion des Entwicklungs- und Wartungsaufwandes von auf Webtechnologien basierenden Anwendungen für die Plattformen Web und Mobile durch Verwendung einer gemeinsamen Codebasis und JavaScript-Transpiler

vorgelegt von

Christoph Strauß

an der

Hochschule der Medien Stuttgart

Januar 2017

zur Erlangung des akademischen Grades eines

Bachelor of Science

Erstprüfer: Prof. Dr.-Ing. Krešimir Vidačković

Zweitprüfer: Prof. Dr. Stephan Wilczek

Ehrenwörtliche Erklärung

Name: Strauß	Vorname: Christoph
Matrikel-Nr.: 27075	Studiengang: Online-Medien-Management (OM7)

Hiermit versichere ich, Christoph Strauß, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: *„Evaluation von Technologien zur Reduktion des Entwicklungs- und Wartungsaufwandes von auf Webtechnologien basierenden Anwendungen für die Plattformen Web und Mobile durch Verwendung einer gemeinsamen Codebasis und JavaScript-Transpiler“* selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), §23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Ort, Datum Unterschrift

Kurzfassung

Die Entwicklung von mobilen Anwendungen und Web-Anwendungen ist aufgrund der großen Zahl von Zielplattformen aufwendig. Gegenstand dieser Arbeit ist die Evaluation von Technologien zur Erstellung von mobilen und Web-Anwendungen aus derselben Codebasis mit Hilfe von Webtechnologien und die Erweiterung der JavaScript-Syntax durch Transpiler. Dabei wird das Ziel verfolgt, Anwendungen aus Perspektive der Wartbarkeit zu konstruieren. Der Begriff der Wartbarkeit im Rahmen der Softwareentwicklung wird geklärt und bildet zusammen mit einem Anforderungskatalog eine Liste von Kriterien, welche zur Auswahl einer geeigneten Technologie verwendet wird. Es wird eine prototypische Anwendung implementiert, anhand der die ausgewählte Technologie hinsichtlich ihrer Eignung evaluiert wird.

Schlagworte: Webtechnologien, JavaScript, TypeScript, Wartbarkeit, mobile Anwendungen, Web-Anwendungen

Abstract

Due to the high number of target platforms development of mobile applications and web applications is expensive. Object of this work is to evaluate technologies which enable the creation of mobile and web applications from the same codebase using web technologies and transpilers to extend the syntax of JavaScript. The overall objective is to construct applications with regard to their maintainability. This work clarifies the term maintainability. Maintainability and a catalog of requirements form a list of criteria, which are used to select an appropriate technology. A prototype application is being implemented, which serves to evaluate the selected technology in regard to their suitability.

Keywords: web technologies, JavaScript, TypeScript, maintainability, mobile applications, web applications

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CSS	Cascading Style Sheets
ES	ECMAScript
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
MVC	Model View Controller
TSC	TypeScript Compiler
UI	User Interface
URL	Uniform Resource Locator
XML	eXtensible Markup Language

1. Einführung

1.1. Motivation und Relevanz

Eine große Zahl an unterschiedlichen Plattformen wie Android, iOS, Windows, Linux, macOS und zusätzlich verschiedene Browser auf jeder Plattform sorgen für einen großen Entwicklungsaufwand bei der Softwareentwicklung. Webtechnologien wie HTML, CSS und JavaScript stehen auf allen Plattformen zur Verfügung und bieten sich daher als universelle Technologien zur Anwendungserstellung an. Webentwickler können ihre Kenntnisse in diesen Technologien nutzen, um ohne großen Lernaufwand Anwendungen für verschiedenen Plattformen zu erstellen. Es bietet sich daher an, Webtechnologien und eine gemeinsame Codebasis zur Erstellung von mobilen Anwendungen und Web-Anwendungen zu verwenden, um den Entwicklungs- und Wartungsaufwand für die jeweilige Plattform zu minimieren.

Mit JavaScript geschriebener Code wird aus verschiedenen Gründen als schwer wartbar wahrgenommen. Der Programmiersprache fehlen Konzepte und Funktionen, welche die Arbeit an komplexen Codebasen erleichtern. Die JavaScript-Sprachversion ECMAScript 2015 definiert unter anderem Sprachkonstrukte einer klassenbasierten Objektorientierung, welche die Strukturierung und Wartung des Quellcodes erleichtern. Da diese Sprachversion bisher von keiner Plattform vollständig unterstützt wird, werden sogenannte JavaScript-Transpiler genutzt, um ECMAScript 2015-Code in die von allen Plattformen unterstützte Sprachversion ECMAScript 5 zu übersetzen. Diese Transpiler bieten zudem die Möglichkeit, die JavaScript-Syntax zu erweitern oder komplett auszutauschen. Dadurch lässt sich JavaScript darüber hinaus um Sprachkonstrukte und Funktionen erweitern, welche die Wartbarkeit der Sprache verbessern.

1.2. Zielsetzung

In dieser Arbeit wird untersucht, mit welchen Technologien und Frameworks es möglich ist, unter Verwendung von Webtechnologien eine Anwendung aus einer gemeinsamen Codebasis zu implementieren, die auf den mobilen Betriebssystemen iOS und Android und im Webbrowser lauffähig ist.

Diese Technologien und Frameworks werden aus der Perspektive der Wartbarkeit betrachtet, um Entwicklungs- und Wartungsaufwand zur Erstellung einer solchen Anwendung niedrig zu halten. Angesichts einer komplexen Codebasis durch die Unterstützung verschiedener Plattformen sollen hierdurch Technologien identifiziert werden, welche die Entwicklung von verständlichem und strukturiertem Quellcode fördern und dabei aus Sicht eines Webentwicklers einen geringen Lernaufwand benötigen.

Im Fokus steht dabei steht vor allem die Programmiersprache JavaScript und die Möglichkeiten, deren Syntax durch Transpiler zu erweitern. Besonders eine Erweiterung der JavaScript-Syntax um Sprachkonstrukte einer klassenbasierten Objektorientierung kann dabei unterstützen wartbaren Quellcode zu erzeugen.

Im Rahmen dieser Arbeit sollen folgende Forschungsfragen beantwortet werden:

1. Mit welchen Technologien lässt sich eine mobile Anwendung und eine Web-Anwendung mit einer möglichst einheitlichen Codebasis umsetzen, mit dem Ziel den Entwicklungs- und Wartungsaufwand möglichst gering zu halten?
2. Mit welchen Technologien lässt sich die JavaScript-Syntax um Sprachkonstrukte der klassenbasierten Objektorientierung erweitern, um die Wartbarkeit der Codebasis zu verbessern?

Zur Beantwortung der Forschungsfragen soll eine prototypische Anwendung implementiert werden, welche als Web-Anwendung im Browser und als mobile Anwendungen auf Geräten oder Emulatoren mit den Betriebssystemen iOS und Android ausführbar ist. Anhand dieser Zielsetzung und den Anforderungen an die Anwendung werden Kriterien zur Auswahl eines geeigneten Frameworks gebildet. Der Anwendungsprototyp dient schließlich zur Validierung der Eignung des Frameworks zur Erstellung von wartbaren mobilen und Web-Anwendungen.

1.3. Vorgehensweise

Das erste Kapitel der Arbeit dient zur Einführung in das Thema. Im zweiten Kapitel werden Grundlagen und Konzepte behandelt. Dazu werden verschiedene Ansätze zur Entwicklung mobiler Anwendungen und deren technische Grundlagen dargestellt. Im Folgenden wird der Begriff der Wartbarkeit bestimmt und die Frage geklärt, auf welche Art und Weise wartbare Software konstruiert wird. In diesem Zusammenhang findet sich eine Reihe von Faktoren, welche die Wartbarkeit von Software beeinflussen. Es werden Eigenschaften von Programmiersprachen sowie Konzepte und Methoden der Softwareentwicklung identifiziert, welche für die Konstruktion wartbarer Software wichtig sind. Zudem werden die Grundlagen der objektorientierten Programmierung und ihre Rolle bei der Erstellung von wartbarer Software dargestellt. Auf Basis dieser Grundlagen wird die Programmiersprache JavaScript hinsichtlich ihrer Mängel bei der Erstellung von wartbarer Software analysiert und bewertet.

Im dritten Kapitel werden funktionale und nicht-funktionale Anforderungen an der Prototypen festgelegt. Diese werden im vierten Kapitel zusammen mit den Forschungsfragen und dem Begriff der Wartbarkeit zur Bildung eines Kriterienkataloges verwendet, der zur Auswahl von geeigneten Frameworks dient.

Anhand dieser Kriterien werden verschiedene Frameworks hinsichtlich ihrer Eignung zur Erstellung der beschriebenen Anwendung analysiert und diskutiert. An dieser Stelle werden auch die vom Framework verwendeten Technologien zur Erweiterung der JavaScript-Syntax betrachtet. Abschließend wird ein Framework zur Implementierung des Prototyps ausgewählt. Die Implementierung des Prototyps dient dazu, die Eignung des gewählten Frameworks zur Erstellung von wartbaren mobilen und Web-Anwendungen aus einer Codebasis detailliert zu evaluieren.

Der Prototyp der Anwendung findet sich für die Plattform Browser auf https://tadeussenf.de/.
Die mobile Anwendung für Android ist im Google Play Store über diesen Link erreichbar:
https://play.google.com/store/apps/details?id=com.ionicframework.thesis652279&hl=en
Die mobile Anwendung für das Betriebssystem iOS ist nicht im Apple AppStore verfügbar, sie kann aus dem Quellcode erstellt und in einem Emulator ausgeführt werden.
Der Quellcode findet sich auf GitHub unter dem Link: https://github.com/tadeussenf/budget-tracker

2. Grundlagen und Konzepte

2.1. Konzepte zur Entwicklung mobiler Anwendungen

Im Folgenden werden verschiedene Konzepte zur Realisierung von Anwendungen für mobile Geräte beschrieben.

2.1.1. Native Apps

Native Apps werden mit den Programmierumgebungen der Smartphonehersteller erstellt und vom Gerät direkt ausgeführt. Native Apps sind die proklamierte Entwicklungsmethode der Betriebssystemanbieter und haben Zugriff auf alle vom Gerät angebotenen Hardwarefunktionen. Die verwendete Programmiersprache und der strukturelle Aufbau der Apps sind vom jeweiligen Gerät abhängig. So wird eine App für Apples iOS in Objective-C entwickelt, während für Android Java verwendet wird (Keist et al., 2016, 111). Native Anwendungen erreichen durch die direkte Ausführung von Code die beste Performance im Vergleich mit den anderen Entwicklungskonzepten, bedeuten aber auch den größten Entwicklungsaufwand, da für jedes Betriebssystem eine eigene Anwendung erstellt werden muss (Heitkötter et al., 2012, 135).

2.1.2. Web-Apps

Bei Web-Apps handelt es sich um Webseiten, die für mobile Geräte optimiert wurden. Sie verfügen über ein responsives Design, welches sich automatisch an die Displaygröße der Zielplattform anpasst und das für eine gute Bedienbarkeit auf mobilen Geräten ausgelegt ist (Keist et al., 2016, 110 f).

Dank den standardisierten Technologien HTML, CSS und JavaScript lassen sich Web-Apps unabhängig von der Plattform ausführen. Dies ermöglicht es, die Anwendung auf allen Zielgeräten mit gleichem Quellcode, mit gleicher Funktionalität und gleichem Aussehen zu realisieren. Web-Apps werden üblicherweise nicht auf dem Zielgerät installiert, sondern werden im Browser des Gerätes über eine URL aufgerufen. Da Web-Apps vom Browser ausgeführt werden, haben sie nur eingeschränkten Zugriff auf gerätespezifische Hardwarefunktionen. (Heitkötter et al., 2012, 122)

Zwar spezifiziert HTML5 Schnittstellen für den Zugriff auf verschiedene Hardwarefunktionen, diese Schnittstellen sind jedoch bisher nicht auf jeder Zielplattform implementiert. Zudem decken diese Schnittstellen nicht die gesamte Bandbreite an Hardware ab, die sich in einem modernen Smartphone finden lässt. West (2014) gibt einen Überblick darüber, welche Hardwarefunktionen im Browser Google Chrome zur Verfügung stehen.

2.2.2.1. Progressive Web-Apps

Progressive Web-Apps sind ein weiteres Konzept zur Erstellung von mobilen Anwendungen (Google, o.J. a). Das Konzept wird bisher nicht in der wissenschaftlichen Literatur behandelt. Marfatiya gibt einen Überblick über die Funktionsweise von Progressive Web-Apps.

Während es sich bei einer Hybrid-App um native Anwendungen handelt, die eine Web-Anwendung, einen nativen Wrapper zum Zugriff auf plattformspezifische Funktionen und alle zur Ausführung benötigten Ressourcen beinhaltet und über den App-Store des Plattformanbieters installiert wird, handelt es sich bei einer Progressive Web-App um eine Web-Anwendung, die sich direkt auf dem Gerät installieren lässt und sich wie eine native Anwendung verhalten soll. (Marfatiya, 2016) Dascalescu (2016) zeigt, dass der Browser Google Chrome Zugriff auf eine Reihe von Gerätefunktionen bietet.

Teilweise noch im Entwurfsstadium befindliche Browser-APIs, das Web App Manifest (W3C, 2017) und die sogenannten Service Worker (W3C, 2015) ermöglichen es, Progressive Web-Apps ohne Umweg durch den App-Store auf einer Zielplattform zu installieren, ohne ständige Internetverbindung auszuführen und auf Funktionen des Zielplattform zuzugreifen. (Marfatiya, 2016)

Die Unterstützung von Web-App-Manifest und Service Workers sind essentielle technische Voraussetzungen für Progressive Web-Apps. Bereits für diese Browser-APIs variiert die Unterstützung der verschiedenen Plattformen erheblich. Während beispielsweise die Browser Google Chrome und Firefox die Service Worker-API nahezu vollständig unterstützen ist dies beim Internet Explorer und dem für die Plattform iOS wichtigen Safari bisher nicht der Fall. (Archibald, o.J.) Die Unterstützung von weiteren APIs zum Zugriff auf Gerätefunktionen variiert stark zwischen den einzelnen Plattformen.

Progressive Web-Apps sind daher eine interessanter Ansatz, um Web-Anwendungen als native Anwendungen auszuführen. In der Praxis ist dieser Ansatz insbesondere durch die fehlende Unterstützung von iOS nicht für die Zielsetzung dieser Arbeit geeignet.

2.1.3. Hybrid-Apps

Beim Konzept der Hybrid-Apps lassen sich zwei Ansätze unterscheiden, WebView-basierte Hybrid-Apps und Runtime-basierte Hybrid-Apps.

.

WebView-basierte Hybrid-Apps kombinieren Webtechnologien und native Funktionalität. Dazu wird ein sogenannter WebView verwendet, die Browserengine der Plattform. WebView-basierte Hybrid-Apps bestehen grundlegend aus einer Webseite, die auf der Browserkomponente des Endgeräts ausgeführt wird, und einem nativen Wrapper, welcher der Webseite den Zugriff auf gerätespezifische APIs ermöglicht. (Heitkötter et al., 2012, 123) Häufig wird dazu Apache Cordova verwendet, welches es erlaubt native Anwendungen mit HTML, CSS und JavaScript zu implementieren und auf native Gerätefunktionen zuzugreifen (Heitkötter et al., 2012, 124).
Eine Reihe von JavaScript-Frameworks baut auf Apache Cordoba auf. Diese stellen für mobile Betriebssysteme optimierte UI-Komponenten bereit und geben eine Struktur zur Erstellung von Anwendungen vor (Smeets und Aerts, 2016, 191).

Runtime-basierte Hybrid-Apps kombinieren native UI-Komponenten mit einer JavaScript-Laufzeitumgebung. Auf diese Weise wird die Anwendungslogik in JavaScript implementiert, während die Benutzeroberfläche und der Zugriff auf gerätespezifische APIs über einen nativen Wrapper angesteuert werden. Dies ermöglicht die Verwendung der nativen UI-Komponenten des jeweiligen mobilen Betriebssystems. Die Benutzeroberfläche wird nicht mittels HTML und CSS beschrieben, was eine Wiederverwendbarkeit von Code zwischen verschiedenen Plattformen erschwert. Beispiele für diesen Entwicklungsansatz ist Appcelerator Titanium Mobile oder NativeScript. (Smeets und Aerts, 2016, 196)

2.2. Wartbarkeit

Der Begriff der Wartbarkeit wird in der Literatur im Rahmen von Software-Qualitätsmodellen beschrieben. Häufige Erwähnung in der Literatur findet die Norm ISO/IEC 9126, welche mittlerweile von der Norm ISO/IEC 25010 ersetzt wurde (ISO/IEC, 2011, IV).

ISO/IEC 25010 stellt dabei kein komplett neues Qualitätsmodell dar, sondern verwendet mit einigen Anpassungen die Merkmale aus ISO/IEC 9126 (ISO/IEC, 2011, V). Viele Begriffe werden daher in beiden Normen identisch oder bedeutungsgleich verwendet. Die nahezu identische Definition des Begriffs Wartbarkeit in den beiden Normen zeigen exemplarisch, dass sich Literatur mit Bezug zu ISO/IEC 9126 sich in vielen Fällen in Bezug zu ISO/IEC 25010 setzen lässt.

ISO/IEC 9126 definiert Wartbarkeit als *"Fähigkeit des Softwareprodukts änderungsfähig zu sein. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen."* (ISO/IEC, 2001, zit. n. Balzert 2011, 116)

ISO/IEC 25010 definiert den Begriff der Wartbarkeit inhaltlich deckungsgleich als
"degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. [...] Modifications can include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications." (ISO/IEC, 2011, 14)

Anhand dieser beiden Definition des Begriffs Wartbarkeit wird deutlich, dass Wartbarkeit als das Ziel, die Durchführungen von Wartungstätigkeiten an Softwareprodukten zu erleichtern, nicht nur im Sinne eines Behebens von Defekten zu verstehen ist. Vielmehr sind neben solchen Reparaturen auch die Anpassung, Veränderung und Optimierung der Software zu verstehen. (Teßmer, 2012, 2 f)

Analog dazu findet sich in ISO/IEC 14764 eine Kategorisierung von Wartungstätigkeiten in erweiternde und korrigierende Maßnahmen. Die erweiternden Maßnahmen werden hierbei unterteilt in adaptive und perfektionierende Wartung, korrigierende Maßnahmen in korrektive und vorbeugende Tätigkeiten. (ISO/IEC, 2006) Riebisch und Bode beschreiben in diesem Zusammenhang mit dem Begriff Software-Evolvability die Zielsetzung Software nicht nur wartbar, sondern auch weiterentwickelbar zu konstruieren. (Riebisch und Bode, 2009, 1)

Der Begriff Wartbarkeit bezieht sich demnach nicht nur auf den Zeitraum nach Auslieferung der Software, sondern umfasst auch bereits frühere Schritte im Entwicklungsprozess. Wartbare Software muss leicht änderbar sein. Sowohl Korrekturen, als auch Erweiterungen der Software

aufgrund Veränderungen in den Rahmenbedingungen, den Anforderungen oder der funktionalen Spezifikation sollen einfach umsetzbar sein. Zukünftige Änderungen der Software müssen daher bereits bei ihrer Planung und Konstruktion antizipiert werden.

Aus diesem Grund verringert eine gute Wartbarkeit nicht nur den Pflegeaufwand einer Software nach ihrer Fertigstellung, sondern auch den gesamten Entwicklungsaufwand zur Herstellung. Entwicklungs- und Wartungsaufwand sind daher im Rahmen dieser Arbeit synonym zu verstehen. Der Begriff Wartbarkeit wird im Rahmen dieser Arbeit im Sinne von Weiterentwickelbarkeit (Balzert, 2011, 119 ff) verwendet.

Im Folgenden wird auf die Frage eingegangen, durch welche Konzepte, Methoden und Techniken ein Softwareprodukt unter der Prämisse einer guten Wartbarkeit konstruiert werden kann.

2.3.1. Subcharakteristika von Wartbarkeit

Die Normen ISO/IEC 9126 und 25010 zerlegen den Begriff der Wartbarkeit in verschiedene Subcharakteristika. ISO/IEC 9126 nennt Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit und Standardkonformität als Subcharakteristika von Wartbarkeit. Diese Subcharakteristika werden in ISO/IEC 25010 umstrukturiert und teilweise genauer definiert. ISO/IEC 25010 nennt die Subcharakteristika Analysierbarkeit, Modifizierbarkeit, Testbarkeit, Modularität und Wiederverwendbarkeit. (ISO/IEC, 2011, 23)

4.2.7	**Maintainability**	**Maintainability**	
4.2.7.1	Modularity		New subcharacteristic
4.2.7.2	Reusability		New subcharacteristic
4.2.7.3	Analysability	Analysability	
4.2.7.4	Modifiability	Stability	More accurate name combining changeability and stability
4.2.7.5	Testability	Testability	

Abbildung 1: Subcharakteristika der Wartbarkeit in ISO/IEC 9216 und ISO/IEC 25010 (ISO/IEC 25010 2011, 23)

Abbildung 1 zeigt, dass die Subcharakteristika Änderbarkeit und Testbarkeit aus ISO/IEC 9126 übernommen wurden und das neue Subcharakteristikum Modifizierbarkeit eine Kombination aus Änderbarkeit und Stabilität ist (Abb. 1). Zusätzlich kommen die Subcharakteristika Modularität und Wiederverwendbarkeit in ISO/IEC 25010 hinzu. Standardkonformität wird

nicht als Anforderung an die Qualität, sondern als allgemeine Anforderung an Systeme betrachtet und aus dem Modell entfernt (ISO/IEC, 2011, V).

Im Rahmen des Qualitätsmanagements von Softwareprojekten werden konstruktive und analytische Maßnahmen zur Sicherstellung einer bestimmten Qualität des Produktes unterschieden. Konstruktive Maßnahmen beinhalten Methoden, Konzepte, Technologien und Richtlinien, welche die Konstruktion von wartbarer Software ermöglichen (Balzert et al., 2008, 476 ff). Analytische Maßnahmen verwenden Metriken zur Vermessung des Quellcodes, wie beispielsweise die Anzahl der Codezeilen und die zyklomatische Komplexität nach McCabe, um diagnostische Aussagen über die Wartbarkeit von Software zu machen (Teßmer, 2012, 20). Analytische Maßnahmen deuten so a posteriori auf problematische Stellen im Quellcode hin, liefern per se aber keine Aussagen darüber, welche Methoden, Konzepte, Sprachen oder Werkzeuge a priori zur Konstruktion wartbarer Software verwendet werden können. (Balzert et al., 2008, 477) Zur Beurteilung von geeigneten Frameworks sollen daher deren Unterstützung für konstruktive Maßnahmen betrachtet werden. Analytische Maßnahmen werden im Rahmen dieser Arbeit nicht betrachtet.

In ISO/IEC 25010 finden sich keine Informationen dazu, auf welche Art und Weise wartbare Software konstruiert werden kann oder welche Merkmale von Software die einzelnen Subcharakteristika der Wartbarkeit beeinflussen. Es ist daher eine nähere Betrachtung der einzelnen Subcharakteristika notwendig, um einzelne Merkmale von wartbarer Software zu formulieren.

Im Folgenden werden die Subcharakteristika der Norm ISO/IEC 25010 anhand von Teßmers Analyse von ISO/IEC 9126 näher beschrieben. Da es sich bei ISO/IEC 25010 nicht um ein komplett neues Qualitätsmodell, sondern vielmehr um eine Umstrukturierung der Qualitätskriterien aus ISO/IEC 9126 handelt, lassen sich Teßmers Überlegungen zu den Subcharakterstika auf ISO/IEC 25010 übertragen.

Von den insgesamt acht Eigenschaftskategorien des ISO/IEC 25010 wird im Rahmen dieser Arbeit nur die Wartbarkeit und deren Untereigenschaften betrachtet. ISO/IEC 25010 beschreibt das Merkmal Wartbarkeit über die Eigenschaften Modularität, Wiederverwendbarkeit, Analysierbarkeit, Änderbarkeit und Testbarkeit (ISO/IEC, 2011, 14 f).

2.2.1. Analysierbarkeit

Analysierbarkeit wird in ISO/IEC 25010 definiert als der *"degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified" (ISO/IEC, 2011, 15).*

Nach Teßmer zielt der Begriff der Analysierbarkeit auf das Systemverstehen durch den Entwickler ab, also darauf, wie leicht ein Entwickler Struktur und Funktionsweise des Softwareprodukts oder von Code-Fragmenten erfassen kann. Diese Sichtweise findet sich auch in anderen Arbeiten. Balzert (2011, 117) nennt neben den Subcharakteristika aus ISO/IEC 9126 ebenfalls Verständlichkeit als wichtigen Faktor von Wartbarkeit. Nitze (2015, 108) verwendet den Begriff Verständlichkeit synonym zu Analysierbarkeit.

Die Analysierbarkeit des Quellcodes ist nach Teßmer *"abhängig von (1) dem Umfang der Entitäten, (2) der verwendeten Programmiersprache und ihren Ausdrucksmöglichkeiten, (3) der Strukturierung des Codes (inkl. der Anzahl von Entitäten und Verbindungen in jeder Sicht), und (4) dem Auftreten von Redundanzen"* (Teßmer, 2012, 70).

Die Verständlichkeit eines Softwaresystems wird dabei durch dem Entwickler bekannte Quellcodestrukturen gestützt (Teßmer, 2012, 69). In diesem Zusammenhang sind Architektur- und Entwurfsmuster relevant. Entwurfsmuster sind nach Balzert (2011, 37) *"[...] bewährte generische Lösungen für häufig wiederkehrende Entwurfsprobleme [...]"*. Architekturmuster beschreiben Prinzipien zur Strukturierung des gesamten Systems. (Balzert, 2011, 37 f)

Die Unterstützung von bestimmten syntaktischen Konstrukten durch die verwendete Programmiersprache ermöglicht es, Sachverhalte explizit auszudrücken. Dies verbessert das Verständnis des Quellcodes, da beim Programmverstehen versucht wird, die Entwurfsentscheidungen nachzuvollziehen, die zu dem vorliegenden System geführt haben. Je größer die Zahl der syntaktischen Sprachkonstrukte ist, die von der Programmiersprache angeboten wird, desto eher lassen sich Hypothesen über die Funktionsweise der Software falsifizieren. Unterstützt eine Programmiersprache solche Sprachkonstrukte, wie Klassen oder Module nicht, sind Entwurfsentscheidungen nicht ohne größeren Aufwand erkennbar. (Teßmer, 2012, 68 ff) Balzert beschreibt in diesem Zusammenhang das Prinzip der

Sichtbarkeit: *"Eine schlechte Sichtbarkeit liegt vor, wenn wichtige Entwurfs- und Fachkonzepte nicht beschrieben oder implizit in der Architektur verborgen sind."* (Balzert, 2011, 34).

Ein Beispiel hierfür ist das Konzept der Kapselung von Daten, auch als "information hiding" bezeichnet, welches zur Modularisierung von Software verwendet wird. In der Programmiersprache Java steht ein syntaktisches Konstrukt *private* zur Verfügung, mit dem angegeben wird ob eine Eigenschaft oder Methode außerhalb einer Klasse zugänglich ist. Die Sprache JavaScript bietet kein explizites Sprachkonstrukt, um den Zugriff zu Eigenschaften oder Methoden zu beschränken. Es ist mit JavaScript dennoch möglich, über spezielle Codestrukturen das Konzept der Kapselung umzusetzen. Dies führt dazu, dass Zugriffsbeschränkungen von Variablen und Methoden nicht direkt gekennzeichnet sind und die Zugriffsbeschränkung aus dem Kontext erschlossen werden muss. Zur Implementierung einer nicht von außen zugänglichen Eigenschaft muss daher neben der Kenntnis des Konzepts der Kapselung zusätzliches Wissen über die Umsetzung mit JavaScript vorhanden sein. Auf diesen Umstand wird in Kapitel 2.4 umfassend eingegangen

2.2.2. Modifizierbarkeit

ISO/IEC 25010 definiert Modifizierbarkeit als *"degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality". (ISO/IEC, 2011, 15)*

ISO/IEC 25010 kombiniert im Begriff der Modifizierbarkeit die Subcharakteristika Änderbarkeit und Stabilität aus ISO/IEC 9126 (ISO/IEC, 2011, 23). In ISO/IEC 25010 wird zusätzlich angemerkt, dass die Modifizierbarkeit durch Modularität und Analysierbarkeit beeinflusst werden kann (ISO/IEC, 2011, 15).

Nach Teßmer ist die Änderbarkeit von Software abhängig von "*(1) der Strukturierung des Quellcodes, (2) seinem Umfang, und (3) seiner Redundanz"* (Teßmer, 2012, 73). Stabilität ist abhängig von "*(1) der Verwendung von Assertions [...] und (2) von der Verwendung von Exceptions zur Fehlerbehandlung" (Teßmer, 2012, 76).*

Der Quellcode sollte demnach auf eine Art und Weise strukturiert werden, welche eine Änderung und Erweiterung der Software erleichtert. (Teßmer, 2012, 71) Dies kann durch die

Verwendung von Architektur- und Entwurfsmustern oder durch das Paradigma der Objektorientierung erreicht werden. Redundanzen im Code erschweren die Änderbarkeit, da jeweils alle betroffenen Stellen angepasst werden müssen und sind daher zu vermeiden.

Teßmer nennt das Konzept der Kapselung und die daraus resultierende Modularität des Quellcodes als wesentliche Voraussetzung von Änderbarkeit. Durch die Kapselung von Daten wird die Entstehung von gekoppelten Code-Strukturen verhindert, da einzelne Komponenten nicht auf die internen Variablen und Methoden anderer Komponenten zugreifen können. Kapselung lässt sich durch verschiedene Strukturen umsetzen, Teßmer nennt hier unter anderem Module, Klassen und Funktionen. (Teßmer, 2012, 71) Neben der genannten Unterstützung der Programmiersprache für Assertions und Exceptions lässt sich die Stabilität auch durch eine statische Typisierung verbessern. Statische Typprüfungen erlauben es eine Reihe syntaktischer Fehler vor der Ausführung des Programms abzufangen und können so unerwartete Auswirkungen von Änderungen verhindern. (Teßmer, 2012, 73)

2.2.3. Modularität

Modularität wird in ISO/IEC 25010 definiert als *"degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components" (ISO/IEC, 2011, 14)*.

Wesentliche Voraussetzung für Modularität ist die Unterstützung der Programmiersprache für das Konzept der Kapselung. Indem der Zugriff auf die Interna eines Bausteins von außen unterbunden wird, wirkt eine modulare Struktur der Entwicklung von eng gekoppelten Code-Strukturen entgegen. Dies führt auch zu einer Trennung von Schnittstelle und Implementierung, die es erlaubt bei Änderungen nur die Implementierung des Bausteins zu ändern, ohne die Funktionalität abhängiger Bausteine zu beeinflussen. (Teßmer, 2012, 70)

Separation of Concerns ist ein Prinzip zur Strukturierung von Quellcode. Jede Komponente eines Systems ist dabei für eine Aufgabe oder einen Aufgabenkomplex zuständig. Aufgaben dürfen nicht über mehrere Komponenten verteilt sein. Verantwortet eine Komponente mehrere Aufgaben, so soll sie für jede Aufgabe eine eigene Schnittstelle anbieten. (Balzert, 2011, 31) Ein Architekturmuster, welches das Prinzip Separation of Concerns unterstützt, ist das Model-View-Controller-Muster (MVC-Muster). Dabei wird eine Anwendung in mehrere Komponenten aufgeteilt: Der View ist verantwortlich für die Darstellung der

Benutzeroberfläche. Das Model enthält Daten und Anwendungslogik. Der Controller fungiert als Bindeglied zwischen View und Model und bildet Ereignisse in der Benutzeroberfläche auf die entsprechende Anwendungslogik ab. (Balzert, 2011, 62-68) Auch die objektorientierte Programmierung ermöglicht eine Aufteilung des Quellcodes nach dem Prinzip Separation of Concerns (Teßmer, 2012, 87).

2.2.4. Wiederverwendbarkeit

Bei Wiederverwendbarkeit handelt es sich nach ISO/IEC 25010 um den *"degree to which an asset can be used in more than one system, or in building other assets". (ISO/IEC, 2011, 15)*

Modularität ist nach Szyperski et al. (2003, 7) die wesentliche Voraussetzung von wiederverwendbarer Software. Die Aufteilung der Codebasis in Module, die jeweils nur für die Erfüllung einer einzigen Aufgabe verantwortlich sind, ermöglicht es diese Module an anderer Stelle erneut zu verwenden (Lahres and Raýman, 2006c).
Meyer sieht in der Wiederverwendung von Code unter anderem den Vorteil eines verringerten Wartungsaufwandes und einer höheren Entwicklungsgeschwindigkeit (Meyer, 1997, 68).

2.2.5. Testbarkeit

Testbarkeit wird bestimmt durch den *"degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met" (ISO/IEC, 2011, 15).*

Nach Teßmer handelt es sich bei Tests um das Aufspüren von Fehlern durch die Ausführung des Quellcodes mit bestimmten Eingabedaten. Grundlage für die Durchführbarkeit solcher Tests ist eine angemessene Modularisierung des Quellcodes. Eine hohe Vererbungstiefe durch objektorientierte Programmierung kann die Testbarkeit negativ beeinflussen, weil jede Klasse innerhalb der Vererbungshierarchie einzeln getestet werden muss. Teßmer weist zudem auf die Wichtigkeit einer automatischen Durchführung der Tests hin, da die Korrektheit umfangreicher Systeme durch manuelle Tests nicht mehr zufriedenstellend geprüft werden kann. (Teßmer, 2012, 77-79)

2.2.6. Zusammenfassung

Zusammenfassend lassen sich mehrere Punkte festhalten, welche die Wartbarkeit eines Softwaresystems und damit den Entwicklungs- und Wartungsaufwand positiv beeinflussen.

Grundlegende Eigenschaft wartbarer Software ist deren modulare Strukturierung nach dem Prinzip Separation of Concerns. Voraussetzung hierfür ist die Unterstützung von information hiding, also der Kapselung von Daten und Methoden. Die Verwendung von Architekturmustern, Design Patterns und objektorientierter Programmierung schafft bekannte Strukturen im Quellcode, welche die Verständlichkeit der Software verbessern. Das Vorhandensein expliziter Sprachkonstrukte für die Strukturierung des Codes erleichtert es, die Funktionsweise der Software zu verstehen. Eine statische Typisierung kann zusätzlich die Änderbarkeit der Software verbessern und zur Vermeidung von Fehlern beitragen. Eine Infrastruktur zur automatischen Durchführung von Testfällen stellt zudem die funktionale Korrektheit der Software sicher.

2.3. Objektorientierte Programmierung

Objektorientierte Programmierung ist eine Methode zur Strukturierung von Programmen, welche zum Ziel hat, die reale Welt innerhalb von Software zu modellieren. Programme werden dabei in Objekte zerlegt, die über Nachrichten miteinander kommunizieren. Angelehnt an Objekte der realen Welt besitzen Objekte in der objektorientierten Programmierung bestimmte Eigenschaften und Methoden, in welches das Verhalten des Objektes beim Empfang einer Nachricht beschrieben wird. (Brauer, 2014, 29 ff).

Grundlegende Konzepte von objektorientierten Programmiersprachen sind Datenkapselung, Vererbung und Polymorphie (Lahres and Raýman, 2006a).
Datenkapselung, auch als information hiding bezeichnet, ist ein Konzept zur Einschränkung des Zugriffs auf die Interna eines Objekts. Der Zugriff auf die internen Datenstrukturen eines Objekts ist damit nur über die Methoden des Objektes möglich. Auf diese Weise wird sichergestellt, dass bei Änderungen am Zustand des Objektes keine Auswirkungen auf andere Teile des Systems auftreten. (Lahres and Raýman, 2006b) Ein Objekt ist also ein in sich gekapselter Baustein, der definierte Schnittstellen zur Kommunikation mit anderen Objekten anbietet.
Vererbung ist ein Konzept zur hierarchischen Strukturierung von Objekten mit ähnlichen Eigenschaften und Methoden. Durch Vererbung kann ein Objekt die Eigenschaften und Methoden eines anderen Objektes erhalten und zusätzlich eigene Eigenschaften und Methoden definieren. (Gumm et al., 2013, 177 f) Objekte innerhalb der so entstandenen

Vererbungshierarchie können unterschiedliche Methoden gleichen Namens definieren, was als Polymorphie bezeichnet wird (Gumm et al., 2013, 254).

Meyer (1997, 16) sieht durch objektorientierte Programmierung wesentliche Verbesserungen bei Faktoren Änderbarkeit, Modularität und Wiederverwendbarkeit .

2.4. JavaScript

JavaScript steht sowohl im Webbrowser als auch auf mobilen Betriebssystemen zur Anwendungsentwicklung zur Verfügung. JavaScript eignet sich daher zur Erstellung von mobilen und Web-Anwendungen aus einer gemeinsamen Codebasis. Um die Eignung von Frameworks für die Erstellung von mobilen und Web-Anwendungen fundiert evaluieren zu können, wird im Folgenden die Programmiersprache JavaScript in Bezug auf Wartbarkeit analysiert. Diese Analyse stellt zudem die Grundlage dar, um Technologien zur Erweiterung der JavaScript-Syntax um explizite Sprachkonstrukte für eine klassenbasierte Objektorientierung zu bewerten.

Bei JavaScript handelt es sich um eine prototypen-basierte, objektorientierte und interpretierte Programmiersprache mit dynamischen Typprüfungen. Obwohl die Sprache gemeinhin als JavaScript bekannt ist, ist sie unter dem Namen ECMAScript standardisiert. Einzelne Versionen des Standards werden beispielsweise als ECMAScript 3, ECMAScript 5 oder kurz ES3 und ES5 bezeichnet. Im Rahmen dieser Arbeit bezieht sich der Begriff JavaScript auf die momentan von allen modernen Browsern unterstützte Sprachversion ES5. Der neue JavaScript-Standard ECMAScript 2015 wird im Rahmen dieser Arbeit als ECMAScript 2015 oder ES6 bezeichnet. (Flanagan, 2011, 1 f)

Nach Veröffentlichung einer neuen Sprachversion müssen die JavaScript-Interpreter von den Herstellern angepasst werden. (Flanagan, 2011, 1 f) Aufgrund der Vielzahl an Plattformen kann nicht davon ausgegangen werden, dass insbesondere neue Sprachfunktionen auf jeder Plattform verfügbar sind. Zaytsev (o.J.) bietet eine Übersicht über die ECMAScript 2015 Unterstützung verschiedener Browser (Abb. 2).

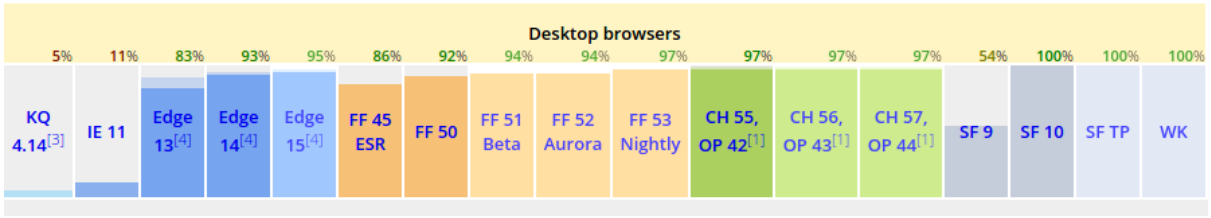

Abbildung 2: Implementierungsgrad von ECMAScript 2015 in Desktop-Browsern (Zaytsev, o.J.)

Zum Zeitpunkt dieser Arbeit bieten die wenigsten Browser eine komplette Unterstützung von ECMAScript 2015.

JavaScript wird oft als eine nicht ernstzunehmende Programmiersprache angesehen, die nur für relativ einfache Aufgaben geeignet ist (Mikkonen und Taivalsaari, 2007, 3). Im Folgenden werden Eigenschaften von JavaScript erörtert, welche die Nutzung von JavaScript in großen Projekten erschweren.

2.4.1. Fehlerbehandlung

Typisch für JavaScript ist die hohe Fehlertoleranz der Sprache. Generell werden Fehler nur gemeldet, wenn es absolut notwendig ist. So führen beispielsweise Tippfehler im Namen von Variablen, die vorher bereits erstellt wurden, nicht zu einer Fehlermeldung:

```
var name = "value";
naem = "new value";
name; // liefert "value"
naem; // liefert "new value"
```

Stattdessen wird die falsch geschriebene Variable implizit erstellt. (Mikkonen und Taivalsaari, 2007, 9) Ähnliches gilt für Eigenschaften von Objekten:

```
var a = "value";
a.property; // liefert undefined
```

Wird auf eine nicht existierende Eigenschaft zugegriffen, wird ohne Fehlermeldung der Wert *undefined* zurückgeliefert. (Ducasse et al., 2012, 6) Wird der Fehler nicht während der Entwicklung bemerkt kann dies zu unerwarteten Konsequenzen führen, da der Wert *undefined* in der weiteren Programmausführung ohne Hinweis weiterverwendet wird.

Auch die interne Code-Vervollständigung von JavaScript kann zu unerwartetem Verhalten führen, wenn nach einem *return*-Statement der Rückgabewert vergessen oder in der Zeile danach angegeben wird:

```
function func() {
  return
  { obj: "obj" }
};
```

Dieser Code wird von JavaScript intern umgewandelt zu:

```
function func() {
  return;
  { obj: "obj" };
};

func(); // liefert undefined
```

Dies führt dazu, dass die aufgerufene Funktion ohne Fehlermeldung den Wert *undefined* zurückgibt. (Nitze, 2015, 113)

2.4.2. Automatische Typkonvertierung

Auch die Durchführung von automatischen Typkonvertierungen ist ein typisches Verhalten von JavaScript. Wird beispielsweise eine Verkettung mit einem String und einer Zahl durchgeführt, ist der Rückgabewert ein String:

```
"1" + 1; // liefert "11"
```

Das Verhalten von JavaScript bei der Evaluation von Bedingungen ist ähnlich:

```
var a = "1";
var b = 1;
if (a == b) { // evaluiert zu true
  //...
}
```

Soll in der Bedingungen auch der Typ der Variablen beachtet werden, muss der sogenannte strikte Gleichheitsoperator verwendet werden (Ducasse et al., 2012, 20 f):

```
var a = "1";
var b = 1;
if (a === b) { // evaluiert zu false
  //...
}
```

Nach Nitze kann JavaScripts flexible Typisierung schwerwiegende Konsequenzen haben. In Kombination mit der permissiven Fehlerbehandlung wird dies noch verstärkt. So kann der Rückgabewert *undefined* einer nicht existierenden Objekteigenschaft ohne eine Fehlermeldung mit anderen Werten verkettet oder anderweitig kombiniert werden:

```
var a = {b: "str"};
var c = "foo";
c + a.d; // liefert "fooundefined"
```

Besonders im Unternehmensumfeld, in dem viele unterschiedlich erfahrene und kompetente Entwickler am Quellcode arbeiten, kann statische Typisierung zur Analysierbarkeit des Programms beitragen und Defekte durch simple Tippfehler verhindern. (Nitze, 2015, 111)

2.4.3. Modularität und der Geltungsbereich von Variablen

Modularität ist eine Eigenschaft von wartbarer Software. Nitze kritisiert das Fehlen eines *import*- oder *include*-Sprachkonstrukts im Sprachstandard, mit dem Quellcode zur Laufzeit dynamisch aus Modulen geladen werden kann. Diese Lücke wird erst von externen Moduleloadern geschlossen, die zusätzlich in das Programm eingebunden werden müssen. (Nitze, 2015, 110 f)

Wie im Kapitel 2.2 festgestellt ist das Konzept der Kapselung die wesentliche Voraussetzung für Modularität, da es erlaubt die Sichtbarkeit von Variablen zu beschränken. Im Folgenden wird gezeigt, dass JavaScript das Konzept der Kapselung nur teilweise unterstützt.

Wird die Variable in einer Bedingung deklariert, kann auf sie von außen zugegriffen werden:

```
if (true) {
  var test = "noop";
}

test; // liefert “noop”
```

Dasselbe gilt für auch für for- und while-Schleifen. Die nächste JavaScript-Sprachversion ES6 führt hier das Sprachkonstrukt *let* ein, mit dem sich die Gültigkeit von Variablen auf Blockebene beschränken lässt (Zakas, 2016, 3):

```
if (true) {
  let test = "noop";
}

test; // Error: test is not defined
```

In der aktuellen Sprachversion ES5 ist nur eine innerhalb einer Funktion deklarierte Variable vor Zugriffen von außen geschützt (Rantanen, 2015, 31) .

```
function func() {
  var test = "noop";
}

func();
test; // liefert undefined
```

Da jede Variable außerhalb eines Funktionsblocks dem globalen Objekt hinzugefügt wird und JavaScript explizite Sprachkonstrukte für Module fehlen, kann es leicht zu Kollisionen kommen, wenn an verschiedenen Stellen im Code derselbe Variablenname verwendet wird. Daher werden sogenannte "Immediately Invoked Function Expression" (IIFE) zur Vermeidung von Kollisionen verwendet (Rantanen, 2015, 32 f):

```
(function(){
  var foobar = "foo";
  foobar; // liefert "foo"
})();

(function(){
  var foobar = "bar";
  foobar; // liefert "bar"
})();
```

Verschärft wird diese Problematik zusätzlich durch implizit globale Variablen. Variablen werden auch innerhalb einer IIFE eindem globalen Objekt hinzugefügt, wenn sie nicht mit dem Keyword *var* deklariert werden (Nitze, 2015, 111):

```
(function(){
  foo = "foo";
  var bar = "bar";
})();

window.foo; // liefert "foo"
window.bar; // liefert undefined
```

Auf diese Weise kann eine Variable auch innerhalb eines Funktionsblocks global gültig sein. Verhindern lässt sich dies durch die Verwendung des "strict mode":

```
(function(){
  "use strict";
  foo = "foo"; // Error: foo is not defined
})();
```

Die kommende Sprachversion ES6 führt ein Modulsystem ein, welches verhindert, dass Variablen außerhalb eines Funktionsblocks zum globalen Objekt hinzugefügt werden (Zakas, 2016, 283 ff). Module können in diesem Fall über die Sprachkonstrukte *export* und *import* aus Dateien geladen werden:

```
// module.js
export let foo = 4;

// main.js
import { foo } from 'module';
foo; // liefert 4;
```

2.4.4. Objektorientierung

Da die meisten Sprachen eine klassenbasierte Objektorientierung verwenden (Crockford, 2008, 29), ist JavaScripts prototypenbasierte Objektorientierung für viele Entwickler ungewohnt (Crockford, 2008, 3). Objektive Untersuchungen zum Verwendungsgrad von prototypenbasierter Objektorientierung in JavaScript finden sich nicht. Die Einführung von Sprachkonstrukten einer klassenbasierten Objektorientierung (Zakas, 2016, 166 f) deutet aber in diesem Zusammenhang auf einen Verbesserungsbedarf hin.

In der klassenbasierter Objektorientierung werden Instanzen mittels Konstruktorfunktionen erzeugt. Eigenschaften, Methoden und Konstruktor eines Objekt werden innerhalb eines *class*-Konstrukts definiert. Auch die Zugriffssteuerung wird üblicherweise hier über bestimmte Konstrukte, wie *private* und *public* geregelt. Vererbung erfolgt durch das Keyword *extends*. Unterklassen müssen ebenfalls explizit definiert werden und erben die Eigenschaften und Methoden der Mutterklasse.

Da JavaScript prototypenbasiert ist existiert weder das Keyword *class* noch ein vergleichbares Konstrukt. Zwar ist es neben der Verwendung von Prototypen in JavaScript möglich, durch die Verwendung einer Konstruktorfunktion das Verhalten von Klassen nachzubilden, das Fehlen von expliziten Sprachkonstrukten für eine klassenbasierte Objektorientierung erschwert aber die objektorientierte Programmierung mit JavaScript. Gama et al. (2013) beobachten verschiedene Ansätze zur Erstellung von Objekten nach dem Vorbild einer klassenbasierten Objektorientierung in JavaScript und sieht diese als hinderlich für das Programmverständnis und Wartbarkeit. Crockford (2008, 46-53) beschreibt ebenfalls verschiedene Methoden zur klassenbasierten, prototypischen und funktionalen Vererbung in JavaScript.

Diese Flexibilität führt zu einer Vielzahl von Vorgehensweisen zur Erzeugung von Objekten, die insbesondere in Bezug auf das Konzept der Kapselung jeweils eigene Implikationen haben. Im Folgenden werden verschiedene Möglichkeiten zur Erzeugung von Objekten in JavaScript analysiert. Dabei wird jeweils auf die Möglichkeiten zur Kapselung von Daten Bezug genommen.

Ein Objekt lässt sich wie folgt erzeugen (Crockford, 2008, 50):

```
var obj = {
  value: "public", // von außen zugänglich
  show: function () {
  return this.value
  }
}

obj.show(); // liefert "public"
obj.value; // liefert "public"
```

In diesem Fall sind alle Eigenschaften und Methoden des Objekt von außen zugänglich, die Kapselung von Informationen ist nicht möglich.

Ein Objekt lässt sich auch mittels einer Konstruktorfunktion erzeugen:

```
function Class () {
  this.public = "public"; // von außen zugänglich
  var private = "private"; // nicht von außen zugänglich
  this.show = function () {
       return private
  }
}
var obj = new Class();
obj.private; // liefert undefined
obj.show(); // liefert "private"
obj.public; // liefert "public"
```

Die innerhalb der Funktion deklarierte Variable ist nicht von außen zugänglich. Auf dieselbe Weise lassen sich auch private Methoden erstellen. Wird die Variable mit *this* direkt an das aufrufende Objekt gebunden, kann sie von außen abgerufen werden.

Auch zur Programmlaufzeit lässt sich ein Objekt dynamisch ändern (Crockford, 2008, 47 f):

```
function Class () {
  this.public = "public"; // von außen zugänglich
  var private = "private"; // nicht von außen zugänglich

}
var obj = new Class();

Class.prototype.showPublic = function () {
       return this.public
  }

Class.prototype.showPrivate = function () {
       return private
  }

obj.private; // liefert undefined
obj.showPublic(); // liefert "public"
obj.showPrivate(); // Error: private is not defined
```

Mit dieser Methode kann aber die Funktion *showPublic()* nicht auf gekapselte Variable *private* zugreifen. Private Eigenschaften und Methoden können also nur angelegt werden, wenn das Objekt erstellt wird. Öffentliche Eigenschaften und Methoden können jederzeit hinzugefügt werden. (Crockford, 2001)

Da sich die Konstruktorfunktion auch als reguläre Funktion aufrufen lässt, muss das Keyword *new* nicht verwendet werden. Dann wird kein neues Objekt erzeugt, sondern die Funktion *Class* wird dem globalen *window*-Objekt zugeordnet. (Crockford, 2008, 46)

```
function Class() {
    return this
};

Class(); // Window {top: Window, window: Window, location: Location,...}
var obj = Class();
obj      // Window {top: Window, window: Window, location: Location,...}
```

Dies lässt sich vermeiden, indem programmatisch die Verwendung des Keywords *new* erzwungen wird (Schmid, 2013):

```
var Class = function () {
    if (!(this instanceof Class)) {
        return new Class();
    }
    //...
};

Class(); // liefert Class
var obj = Class();     // liefert Class
```

Vererbung lässt sich wie folgt realisieren (Crockford, 2008, 47 f):

```
function Class () {
  this.public = "public"; // von außen zugänglich
  var private = "private"; // nicht von außen zugänglich
  this.show = function () {
       return private
  }
}

var SubClass = function () {
  var private = "different variable";
}

SubClass.prototype = new Class();
SubClass.prototype.showPublic = function () {
  return 'inherited value: ' + this.public;
}
var obj = new SubClass();
obj.showPublic(); // liefert "inherited value: public"
obj.show(); // liefert "private"
```

Die Konstruktorfunktion *Class* wird als Prototyp für *SubClass* verwendet. Die Variable *public* wird vererbt, was in diesem Beispiel durch das Anfügen eines zusätzlichen Strings verdeutlicht wird. Ebenso erbt das mit der Konstruktorfunktion *SubClass* erzeugte Objekt die Methode *show()*. Die Variable *private* wird allerdings nicht vererbt, stattdessen greift die Methode *show()* auf die Variable *private* aus der Konstruktorfunktion *Class* zurück. Es lässt sich schließen, dass private Variablen nicht vererbt werden. Damit der neue Wert der Variablen *private* in *SubClass* ausgegeben wird, muss auch die Methode *show()* in der Konstruktorfunktion *SubClass* explizit implementiert werden.

Es existieren weitere Ansätze zur Erzeugung von Objekten, die hier nicht weiter betrachtet werden. Gama et al. identifizieren fünf verbreitete Ansätze zur Nachbildung einer klassenbasierten Objektorientierung (Gama et al., 2012, 1 f).

Die exemplarisch dargestellten Ansätze zur Erstellung von Objekten zeigen, dass weder eine einheitliche Methode zur Erzeugung von Objekten noch explizite Sprachkonstrukte zur Zugriffssteuerung existieren. Die Möglichkeit Objekte zur Laufzeit zu verändern, führt dazu, dass das Verhalten von Objektmethoden an beliebigen Stellen im Quellcode verändert werden kann. Die Funktionsweise von Codestrukturen zur Erzeugung von Objekten und die Verhaltensweise von Objekten selbst ist daher nicht leicht erkennbar und muss in vielen Fällen aus dem Kontext erschlossen werden. Dies erschwert die Analysierbarkeit und Änderbarkeit des Quellcodes.

Da jeder Ansatz eigene Vor- und Nachteile bietet, ist eine detaillierte Kenntnis der Sprache notwendig, um die Funktionsweise der Ansätze selbst und deren Implikationen insbesondere in Bezug auf das Konzept der Kapselung zu verstehen. Die prototypische Objektorientierung von JavaScript erschwert daher die Analysierbarkeit des Quellcodes und wirkt sich durch das Fehlen von Sprachkonstrukten zur Kapselung auch negativ auf die Modularität des Quellcodes aus.

ECMAScript 2015 führt das Sprachkonstrukt *class* ein, das eine klare Syntax zur Erzeugung von Objekten bietet (Zakas, 2016, 189 f). Das Konstrukt *class* ist dabei nur “syntactic sugar”, die Klassendefinition wird von JavaScript intern in eine IIFE-gekapselte Konstruktorfunktion umgewandelt, die prototypische Objektorientierung wird beibehalten. Die Vorgehensweise zur Kapselung von Eigenschaften und Methoden bleibt unverändert:

```
class Class {
  constructor() {
    let hidden = "private";
    this.visible = "public";

    this.getHidden = function () {
      return hidden;
    }
  }

  print() {
    return this.visible;
  }
}

let obj = new Class();
obj.hidden; // liefert undefined
obj.visible; // liefert "public"
obj.getHidden(); // liefert "private"
```

Die Methode zum Zugriff auf die private Variable *hidden* muss wie in ES5 im Konstruktor deklariert werden. Wird die Klasse ohne das Keyword *new* aufgerufen wird ein Fehler geworfen. Zudem werden alle Anweisungen innerhalb einer Klassendefinition automatisch im "strict mode" ausgeführt. (Zakas, 2016, 167 f)

Zur Vererbung führt ES6 das Keyword *extend* ein, mit dem eine Klasse Eigenschaften und Methoden von einer anderen Klasse erben kann (Zakas, 2016, 178 f). In diesem Beispiel wird von der Klasse aus dem vorherigen Beispiel geerbt:

```
class Child extends Class {
       constructor() {
         super(); // super ruft den Konstruktor der vererbenden Klasse auf
       }
       setVisible(param) {
         this.visible = param;
       }
}

let obj = new Child();
obj.setVisible("visible");
obj.print(); // liefert "visible"
```

Das *class*-Konstrukt in ES6 bietet eine einheitliche Syntax, die Entwicklern mit einem Hintergrund in klassenbasierter Objektorientierung vertraut ist. Somit lässt sich die Analysierbarkeit und Änderbarkeit des Quellcodes erhöhen. Explizite Sprachkonstrukte zur Kapselung von Eigenschaften und Methoden werden in ES6 aber weiterhin nicht definiert.

2.4.5. Bewertung

Zusammenfassend lassen sich mehrere Punkte festhalten, welche die Konstruktion von wartbarer Software mit JavaScript erschweren. Die Sprache weist an vielen Stellen nicht auf Fehler im Code hin, was die Fehlersuche und damit auch Modifizierbarkeit und Analysierbarkeit der Anwendung erschwert.

Es fehlt ein Modulsystem, mit dem sich einzelne Bausteine unabhängig voneinander erstellen lassen. Dies erschwert eine klare Strukturierung des Quellcodes. Die Kapselung von Informationen ist nur unter Verwendung nicht intuitiver Codestrukturen möglich. Es ist für Entwickler nicht explizit erkennbar, welche Absicht hinter einer solchen Codestruktur steht. Diese Information muss aus dem Kontext des betrachteten Quellcodes erschlossen werden, was die Analysierbarkeit des Quellcodes erschwert. Das Fehlen von expliziten Sprachkonstrukten zur Zugriffssteuerung kann außerdem zu eng gekoppelten Code-Strukturen führen, welche sich nachteilig auf die Modularität des Quellcodes auswirken.

JavaScript bietet eine mächtige und flexible Unterstützung für Objektorientierung, deren Verwendung aufgrund der vielen Handlungsoptionen aber einen großen Lernaufwand und Detailwissen erfordert. Da wie im Kapitel 2.2 ausgeführt, dem Entwickler bekannte Code-Strukturen die Analysierbarkeit der Software stützen, kann sich diese Flexibilität negativ auf die Analysierbarkeit von Software auswirken. Zudem ist die prototypenbasierte Objektorientierung und die Vorgehensweise zur Kapselung von Eigenschaften und Methoden für viele Entwickler ungewohnt.

Eine einheitliche Syntax zur Erzeugung von Objekten und zur Vererbung kann die Verständlichkeit von JavaScript verbessern und Fehler verhindern, die durch Unkenntnis eines Ansatzes entstehen können. Dasselbe gilt für das Konzept der Kapselung. Private und öffentliche Eigenschaften oder Methoden sind nicht leicht erkennbar, sondern müssen aus dem jeweiligen Programmkontext hergeleitet werden. Rantanen (2015, 31 f) weist darauf hin, dass explizite Sprachkonstrukte für Kapselung von Variablen in größeren Projekten ein Mittel sind, um anderen Entwicklern mitzuteilen, dass bestimmte Code-Fragmente Implementierungsdetails sind und nicht darauf zugegriffen werden soll .

Die kommende JavaScript-Sprachversion ES6 verbessert einige dieser Punkte durch die Einführung eines Modulsystems, Variablen mit Blockgeltungsbereich und einem *class*-Sprachkonstrukt. Automatische Typumwandlungen, die komplexe Vorgehensweise zur Kapselung und die an vielen Stellen nicht vorhandenen Fehlermeldungen beeinflussen die Wartbarkeit von JavaScript-Code aber weiterhin negativ.

Aufgrund der bisher fehlenden Unterstützung von ES6 durch die verschiedenen Plattformen ist der Einsatz der neuen Sprachversion bisher nicht ohne weiteres möglich. Nitze (2015, 155 ff) schlägt daher neben der Etablierung von bestimmten Konventionen über die Entwicklung von Software die Verwendung von Programmiersprachen wie TypeScript, CoffeeScript oder Dart vor, die in JavaScript übersetzt werden.

Bei diesem Ansatz ermöglichen sogenannte Transpiler (oder source-to-source-compiler) eine Programmiersprache in eine andere zu übersetzen. Dies ermöglicht es, wie im Fall von TypeScript, die JavaScript-Syntax zu erweitern (TypeScript, o.J. a) oder wie bei CoffeeScript (CoffeeScript, o.J.) und Dart (Dart, o.J.) gänzlich zu ersetzen. Zusätzlich kann mit dem Transpiler Babel auch ES6 in die aktuelle JavaScript-Version übersetzt werden (Babel, o.J.). Eine Umfrage unter JavaScript-Entwicklern (Abb. 3) legt nahe, dass ca. 85% der Entwickler Transpiler verwenden, um ECMAScript 2015 zu ES5 zu übersetzen (Bevacqua, 2015).

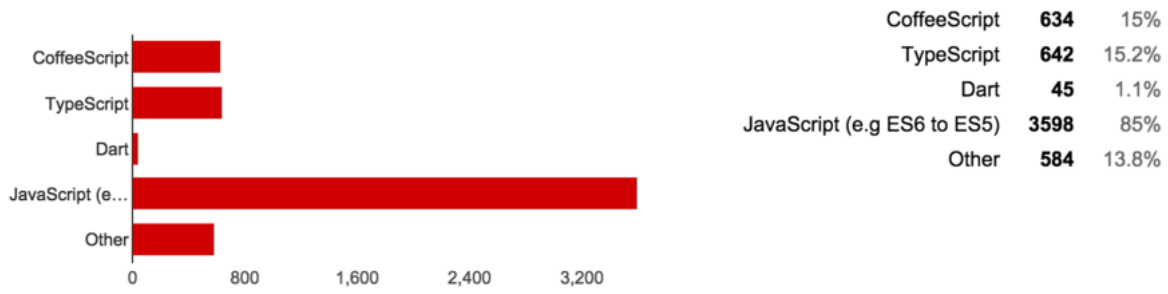

Abbildung 3: Nutzung von JavaScript-Transpilern (Bevacqua, 2015)

Neben den genannten Sprachen findet sich eine große Anzahl weiterer Sprachen, die in JavaScript übersetzt werden können (Ashkenas, 2016). Aufgrund der großen Beliebtheit von JavaScript-Transpilern wird JavaScript von Hanselmann (2013) als "*Assembly Language of the Web*" bezeichnet.

3. Anforderungsanalyse

Zur Evaluation von geeigneten Technologien wird mit den im Folgenden ausgewählten Technologien eine prototypische Anwendung entwickelt. Bei der Anwendung handelt es sich um einen Budget-Tracker, der es ermöglicht getätigte Einkäufe strukturiert abzuspeichern, um einen Überblick über die persönlichen Ausgaben zu erhalten. Die gespeicherten Einkäufe können Kategorien zugeordnet werden, um einen Überblick über die Ausgaben für bestimmte Warengruppen, beispielsweise Nahrungsmittel, Genusswaren, Kleidung, Reisekosten, u.ä. zu erhalten. Die Daten sollen sich in aggregierter Form als Statistik darstellen lassen, anhand derer sich erkennen lässt, welcher Geldbetrag für die einzelnen Kategorien ausgegeben wurde.

Im Folgenden werden die konkreten Anforderungen an die Anwendung unterteilt in funktionale und nicht-funktionale Anforderungen aufgelistet.

3.1. Funktionale Anforderungen

Allgemeine Anforderungen

- Die Anwendung lässt sich im Webbrowser Google Chrome und Mozilla Firefox und auf mobilen Geräten oder Emulatoren mit den Betriebssystemen iOS und Android ausführen.

Eingabe von Einkäufen

- Ein Einkauf besteht aus einem einzelnen Produkt, das über einen frei wählbaren Namen und einen Preis verfügt.
- Einkäufe können einer Kategorie zugeordnet werden.
- Es können neue Kategorien erstellt werden.
- Bei der Eingabe des Einkaufs wird wenn möglich anhand des aktuellen GPS-Koordinaten des Gerätes über die Google Places API (Google, o.J. b) das Geschäft zu bestimmen versucht, in dem das Produkt gekauft wurde. Bei Erfolg werden Name und Adresse des Geschäfts neben den anderen Daten des Einkaufs gespeichert.
- Nach der Eingabe bleiben die Einkäufe auf dem Gerät gespeichert.

Darstellung der getätigten Einkäufe

- Die Anwendung stellt eine Liste der bereits getätigten Einkäufe dar.

- Name, Preis und Kategorie von bereits getätigten Einkäufen können zur Erstellung von neuen Einkäufen wiederverwendet werden.

Statistik

- In einem Kuchendiagramm werden die Anteile der einzelnen Kategorien am insgesamt ausgegebenen Betrag dargestellt.
- In einem Kuchendiagramm werden die Anteile der einzelnen Geschäfts am insgesamt ausgegebenen Betrag dargestellt.
- In einem Balkendiagramm werden die Anteile der einzelnen Kategorien an dem ausgegebenen Betrag pro Monat dargestellt.
- Die Diagramme können auf mobilen Betriebssystemen mit anderen Anwendungen geteilt werden, um diese beispielsweise per E-Mail zu versenden.

3.2. Nicht-funktionale Anforderungen

- Die Anwendung soll aus einer technologisch möglichst einheitlichen Codebasis bestehen.
- Die Anwendung soll aus möglichst wenig plattformspezifischem Code bestehen, das bedeutet, dass möglichst viel Quellcode auf allen Zielplattformen ausgeführt werden kann.
- Das Look & Feel der Anwendung soll sich in das der jeweiligen Plattform einfügen.

4. Kriterienkatalog und Technologieauswahl

Heitkötter et al. schlagen eine Reihe von Kategorien zur Evaluation von Ansätzen zur plattformübergreifenden Entwicklung von mobile Anwendungen vor (Heitkötter et al., 2012).

Die verschiedenen Ansätze werden dabei aus zwei Perspektiven bewertet: Der Infrastrukturperspektive, welche sich auf die Verwendung, Betrieb und Funktionalität des Entwicklungsansatzes bezieht, und der Entwicklungsperspektive, deren Kategorien in Verbindung zum Entwicklungsprozess der App stehen. (Heitkötter et al., 2012, 124)

Infrastrukturperspektive	**Entwicklungsperspektive**
1. Lizenz und Kosten 2. Unterstützte Plattformen 3. Zugriff auf plattformspezifische Funktionen 4. Zukunftssicherheit 5. Look & Feel 6. Anwendungsperformance 7. Distribution	1. Entwicklungsumgebung 2. GUI Design 3. Einfache Entwicklung 4. Wartbarkeit 5. Skalierbarkeit 6. Möglichkeiten zur Weiterentwicklung 7. Entwicklungsgeschwindigkeit

Kategorien nach Heitkötter et al. (2012, 125 f)

Diese Kategorien bieten einen geeigneten Grundlage zur Entwicklung von Kriterien, anhand derer sich geeignete Technologien zur Erstellung von nativen Apps und Web-Apps unter dem Gesichtspunkt der Wartbarkeit evaluieren lassen.

4.1. Kategorienbildung und Definition der Auswahlkriterien

Heitkötter et al. (2012) geben mit ihrer Arbeit einen allgemeinen Leitfaden zur Evaluation von Entwicklungsansätzen zur Entwicklung von mobilen Anwendungen vor und gehen nicht spezifisch auf die Entwicklung von mobilen Anwendungen und Webanwendungen mit gemeinsamer Codebasis ein.

Die vorgeschlagenen Kategorien sind nicht immer trennscharf formuliert, sondern weisen teilweise inhaltliche Überschneidungen auf. Beispielsweise nennen Heitkötter et al. die Modularität des Frameworks als Bewertungskriterium für die Kategorie Skalierbarkeit. In dieser Arbeit wird jedoch Modularität in der Kategorie Wartbarkeit betrachtet. Anforderungen zur Erfüllung der Kategorien sind oft nur beispielhaft aufgelistet und bedürfen einer Konkretisierung und Anpassung auf den jeweiligen Anwendungsfall.

Aufgrund dieses allgemeinen Charakters ist es notwendig, die von Heitkötter et al. vorgeschlagenen Kategorien an die Zielsetzung dieser Arbeit anzupassen. Zudem sind nicht alle von Heitkötter et al. vorgeschlagenen Kategorien für die Zielsetzung dieser Arbeit relevant oder anwendbar. So ist beispielsweise die Anwendungsperformance nicht Untersuchungsgegenstand dieser Arbeit. Es werden daher nicht alle Kategorien zur Evaluation von geeigneten Frameworks verwendet.

Aus den gebildeten Kategorien und den im vorigen Kapitel festgelegten funktionalen und nicht-funktionalen Anforderungen werden anschließend konkrete Kriterien abgeleitet, anhand derer eine Evaluation von Technologien durchgeführt werden kann. Dazu werden die Kriterien in Muss- und Soll-Kriterien aufgeteilt. Muss-Kriterien sind Anforderungen, welche die betrachteten Frameworks zwingend erfüllen müssen. Die Muss-Kriterien werden tabellarisch ausgewertet. Soll-Kriterien sind Anforderungen, die nicht unbedingt erforderlich aber vorteilhaft sind, oder die auf unterschiedliche Arten realisiert werden können, sodass eine ausführliche Diskussion notwendig wird.

Die Nichterfüllung eines Muss-Kriteriums führt zum Ausschluss der Frameworks aus der weiteren Evaluation. Die verbleibenden Frameworks werden anschließend anhand der Soll-Kriterien detailliert evaluiert. Die Nichterfüllung eines Soll-Kriteriums führt nicht zum Ausschluss des Frameworks, sondern es wird anhand aller Soll-Kriterien das am besten geeignete Framework diskutiert und ausgewählt.

4.1.1. Lizenz und Kosten

Dieses Kriterium bezieht sich auf die Lizenzierung des Frameworks als freie Software, Open-Source oder als kommerzielle Software und die mit der Nutzung verbundenen Kosten (Heitkötter et al., 2012, 125). Im Rahmen dieser Arbeit werden nur Technologien mit Open-Source-Lizenzen nach der Open Source Initiative (o.J.) betrachtet, welche kostenfrei nutzbar sind und auch eine kommerzielle Nutzung zulassen.

Muss-Kriterien

Das Framework muss

1. Unter einer Open Source Lizenz nach der Open Source Initiative veröffentlicht sein
2. Ohne Kosten kommerziell nutzbar sein

4.1.2. Unterstützte Plattformen

Dieses Kriterium betrachtet die unterstützten mobilen Plattformen und ob die verschiedenen Plattformen gleichermaßen gut unterstützt werden (Heitkötter et al., 2012, 125).

Im Rahmen dieser Arbeit ist außerdem relevant, ob es mit dem Framework möglich ist, mobile und Web-Anwendungen zu erstellen. Zudem sollte das Framework ermöglichen, die Anwendung mit möglichst wenig redundantem, plattformspezifischem Code zu erstellen.

Da nicht alle Funktionen auf allen Plattformen zur Verfügung stehen, ist plattformspezifischer Code in einigen Fällen unumgänglich. Beispielsweise existiert im Browser kein Konzept des Teilens von Informationen mit anderen installierten Anwendungen, wie es bei iOS und Android der Fall ist. In solchen Fällen sollte das Framework eine Möglichkeit bieten, die Plattform zu erkennen, auf der die Anwendung ausgeführt wird, um sich angemessen verhalten zu können.

Muss-Kriterien

Das Framework muss

1. es ermöglichen Anwendungen zu erstellen, die auf den Betriebssystemen iOS, Android und im Browser lauffähig sind.

Soll-Kriterien

Das Framework soll

1. es ermöglichen, Anwendungen mit möglichst wenig plattformspezifischem Code zu erstellen
2. in der Lage sein, die Plattform zu identifizieren auf der die Anwendung ausgeführt wird

4.1.3. Zugriff auf plattformspezifische Funktionen

Der Zugriff auf plattformspezifische Funktionen bezieht sich sowohl auf Gerätehardware, wie die Kamera oder den GPS-Sensor, als auch Funktionalität, die von der Plattform angeboten wird, wie Zugriff auf Kontakte oder System-Benachrichtigungen (Heitkötter et al., 2012, 125).

Die Betriebssysteme iOS und Android verfügen über ein Konzept zum Teilen von Informationen mit anderen Anwendungen. Hierüber lassen sich unter anderem Bilder oder Texte an andere Anwendungen übertragen, um sie beispielsweise als E-Mail zu versenden. (Apple, o.J. a und Android, o.J. a)

Diese Funktionalität wird im Folgenden kurz als “Teilen von Informationen” bezeichnet. Im Browser wird dieses Konzept nicht unterstützt. Ausgehend von den im vorigen Kapitel festgelegten Anforderungen ist es relevant, dass das Framework den Zugriff auf den GPS-

Sensor, das Teilen von Informationen und das Speichern von Informationen auf dem Gerät ermöglicht.

Muss-Kriterien

Das Framework muss

1. Zugriff auf den GPS-Sensor des Gerätes ermöglichen
2. das Speichern von Daten auf dem Gerät unterstützen
3. das Teilen von Informationen mit anderen Anwendungen unterstützen, wenn die Plattform diese Möglichkeit bietet

4.1.4. Look & Feel

Heitkötter et al. betrachten in dieser Kategorie inwiefern das Framework dem typischen Look & Feel der Plattform entspricht.

Dazu gehören auch plattformtypische Bedienkonventionen. Zudem weisen Heitkötter et al. darauf hin, dass eine App jederzeit minimiert oder beendet werden kann. Beim erneuten Starten erwartet der Nutzer, dass er dort weitermachen kann, wo vorher aufgehört wurde (2012, 125). Android teilt Anwendungen über den sogenannten Activity Lifecycle, wenn sie minimiert oder beendet werden (Android, o.J. b), iOS bietet mit dem App Lifecycle ein vergleichbares Konzept an (Apple, o.J. b).

Da es die Aufgabe der Anwendung ist, auf diese Lifecycle-Events angemessen zu reagieren und beispielsweise Benutzereingaben vor dem Beenden zu speichern, muss es das Framework ermöglichen, auf die Lifecycle-Events der jeweiligen Plattform innerhalb der Anwendung zu reagieren.

Muss-Kriterien

Das Framework muss

1. über UI-Komponenten verfügen, die sich in das Look & Feel der jeweiligen Plattform einfügen
2. die Lifecycle-Events der Plattform unterstützen

4.1.5. Zukunftssicherheit

Die Zukunftssicherheit des Frameworks zeigt sich nach Heitkötter et al. an folgenden Punkten: Kurze Update-Intervalle, regelmäßige Bugfixes, die Unterstützung der neuesten mobilen

Betriebssysteme und die ständige Weiterentwicklung durch eine aktive und große Entwicklergemeinde und möglichst mehrere Unternehmen (Heitkötter et al., 2012, 125).

Da sich der Erfüllungsgrad dieser Punkte schwer messen lässt - welche Größe ist beispielsweise für die Entwicklergemeinde angemessen und wie kann deren Aktivität objektiv bewertet werden? - werden im Rahmen dieser Arbeit nur Frameworks betrachtet, die innerhalb der letzten drei Monate ein Update veröffentlicht haben und von denen in der Vergangenheit bereits eine stabile Version veröffentlicht wurde. Sind diese Punkte gegeben werden auch neuere Versionen eines Frameworks, die beispielsweise im Status eines Release Candidate kurz vor der Veröffentlichung als stabile Version stehen, betrachtet. Alpha- und Beta-Versionen werden nicht betrachtet.

Muss-Kriterien

Vom Framework muss

1. innerhalb der letzten 3 Monaten eine neue Version veröffentlicht worden sein
2. wenigstens ein stabiles Release veröffentlicht worden sein

4.1.6. Entwicklungsumgebung

In dieser Kategorie werden Ausgereiftheit und Funktionalität der typischerweise mit dem Framework genutzten Entwicklungsumgebung bewertet. Heitkötter et al. nennen hier insbesondere Werkzeugunterstützung, wie Vorhandensein einer IDE, eines Debugger und eines Emulators und Funktionen, wie Auto-Vervollständigung von Code und automatische Durchführung von Tests. Zudem wird eine einfache Installation im Sinne eines geringen Aufwands zur Erstellung einer vollständig nutzbaren Entwicklungsumgebung als positiv bewertet (2012, 126).

Das Kriterium der einfachen Installation bedeutet im Rahmen dieser Arbeit, dass das Framework keine manuelle Integration mit zusätzlicher Software benötigt, um voll funktionsfähige mobile Anwendungen zu erstellen.

Es ist zudem von Relevanz, ob das Framework den Softwareentwicklungsprozess in Gänze unterstützt. Dies bedeutet, dass nach jeder Änderung automatisch der aktuelle Quellcode kompiliert oder ausgeführt wird, um das manuelle Testen der Anwendung zu ermöglichen. Diese Funktionalität wird im Folgenden als Live-Reload bezeichnet. Die Funktionalität der

typischerweise mit dem Framework genutzten IDE oder Editors wird wegen der Vielzahl von Editoren und IDEs nicht evaluiert. Im Rahmen der Zielsetzung JavaScript-Frameworks zu evaluieren, wird hier zudem die typischerweise mit dem Framework genutzte Programmiersprache betrachtet. Diese muss entweder JavaScript sein, oder das Framework muss die Transpilierung der verwendeten Programmiersprache zu JavaScript unterstützen.

Muss-Kriterien

Das Framework muss

1. die Programmiersprache JavaScript verwenden, oder die verwendete Sprache zu JavaScript transpilieren.
2. nach Änderungen am Code die Anwendung automatisch kompilieren und ausführen (Live-Reload).
3. die Erstellung von auf mobilen Betriebssystemen lauffähigen Anwendungen ohne manuelle Integration von zusätzlicher Software ermöglichen

Soll-Kriterien

Das Framework soll

1. die automatische Durchführung von Tests unterstützen
2. über einen Debugger und Emulator verfügen

4.1.7. Einfache Entwicklung

Diese Kategorie bezieht sich auf die Qualität der Dokumentation des Frameworks und auf den damit verbundenen Lernaufwand, der nötig ist, um Anwendungen mit dem Framework zu erstellen. Nach Heitkötter et al. ist das Kriterium der Dokumentationsqualität gegeben, wenn Codebeispiele, Links zu ähnlichen Problemen und Nutzerkommentare verfügbar sind (2012, 126).

Das Kriterium Lernaufwand wird darüber bestimmt, ob sich bekannte Programmier-Paradigmen in den Konzepten des Frameworks wiederfinden und wieviel zusätzliches, frameworkspezifisches Wissen sich ein Entwickler aneignen muss. Dabei ist insbesondere relevant, ob das Framework bekannte Technologien, wie HTML, CSS und JavaScript unterstützt, oder ob die Aneignung neuer Konzepte oder Technologien notwendig ist. Eine Möglichkeit, Dritt-Software wie häufig verwendete JS- oder CSS-Bibliotheken einzubinden, ermöglicht die Wiederverwendung von Code und verringert den Lernaufwand.

Soll-Kriterien

Das Framework soll

1. über eine Dokumentation mit Codebeispiele verfügen
2. über Kanäle zur Kommunikation zwischen Anwendern verfügen
3. auf bekannten Paradigmen und Konzepten aufbauen
4. wenig frameworkspezifisches Wissen erfordern
5. die Verwendung von Dritt-Software erlauben

4.1.8. Wartbarkeit

Heitkötter et al. schlagen als Metrik für Wartbarkeit die Zahl der Codezeilen vor (2012, 126). Anstelle dieser Metrik wird an dieser Stelle der im Rahmen dieser Arbeit erarbeitete Begriff von Wartbarkeit mit den Subcharakteristika Analysierbarkeit, Modifizierbarkeit, Modularität, Wiederverwendbarkeit und Testbarkeit verwendet. Die Wartbarkeit des Frameworks wird vor allem darüber bestimmt, welche Möglichkeiten zur Strukturierung und Wiederverwendung des Quellcodes vorhanden sind. Aufgrund der vorangegangenen Analyse von JavaScript wird eine statische Typisierung genauso als vorteilhaft betrachtet, wie unterstützende Hinweise auf Programmierfehler oder unerwartete Verhaltensweisen, wie sie im Kapitel 2.4 genannt wurden. Die Verwendung von Architekturmustern zur Strukturierung des Quellcodes und das Vorhandensein eines Modulsystems, sowie explizite Sprachkonstrukte zur Kapselung von Informationen und zur objektorientierten Programmierung tragen ebenfalls zur Wartbarkeit des Frameworks bei. Das Vorhandensein einer Infrastruktur zur automatischen Durchführung von Tests wird bereits in der Kategorie Entwicklungsumgebung bewertet und wird hier nicht erneut betrachtet.

Soll-Kriterien

Das Framework soll

1. Architekturmuster zur Strukturierung verwenden
2. über ein Modulsystem verfügen
3. Konzepte zur modularen Strukturierung des Quellcodes anbieten
4. Möglichkeiten zur Wiederverwendung von Codestrukturen bieten
5. über syntaktische Konstrukte zur Kapselung von Daten und zur objektorientierten Programmierung verfügen
6. über eine statische Typisierung verfügen

4.2. Auswahl anhand der Muss-Kriterien

Im Folgenden werden die festgelegten Muss-Kriterien tabellarisch ausgewertet.

	Ionic	**NativeScript**
Lizenz und Kosten - Open Source-Lizenz - Ohne Kosten nutzbar	- Ja - Keine Kosten (Ionic, o.J. a)	- Ja - keine Kosten (NativeScript, o.J. a)
Unterstützte Plattformen - unterstützte Plattformen	- iOS, Android, Browser (Ionic, o.J. e)	- iOS, Android, Browser (NativeScript, o.J. g und Stoychev 2016)
Zugriff auf plattformspezifische Funktionen - GPS-Sensor - Speichern von Daten - Teilen von Informationen	- Ja (Ionic, o.J. i) - Ja (Ionic, o.J. b) - Ja (Ionic, o.J. b)	- Ja (NativeScript, o.J. b) - Ja (NativeScript, o.J. c) - Ja (NativeScript, o.J. d)
Look & Feel - UI-Komponenten fügen sich in das Look & Feel der Plattform ein - Lifecycle-Events der Plattform werden unterstützt	- Ja (Ionic, o.J. c) - Ja (Ionic, o.J. f)	- Ja (NativeScript, o.J. e) - Ja (NativeScript, o.J. f)
Entwicklungsumgebung - Programmiersprache - JavaScript-Transpiler - Erstellung von lauffähigen Anwendungen	- TypeScript - Ja (Ionic, o.J. j) - Ja (Ionic, o.J. k)	- JavaScript oder TypeScript - Ja (NativeScript, o.J. g) - Ja (NativeScript, o.J. i)
Zukunftssicherheit - Release in den letzten 3 Monaten - Bereits ein stabiles Release veröffentlicht	- Ja: 11.1.17 - Ja (Ionic, o.J. d)	- Ja: 8.12.16 - Ja (NativeScript, o.J. k)

	React Native	**Appcelerator Titanium**
Lizenz und Kosten - Open Source-Lizenz - Ohne Kosten nutzbar	 - Ja - Keine Kosten (React Native, o.J. a)	 - kommerziell - kostenpflichtig (Appcelerator, o.J. a)
Unterstützte Plattformen - unterstützte Plattformen	 - iOS, Android, Browser (React Native, o.J. k und Gallagher o.J. a)	 - iOS, Android (Appcelerator, o.J. b)
Zugriff auf plattformspezifische Funktionen - GPS-Sensor - Speichern von Daten - Teilen von Informationen	 - Ja (React Native, o.J. b) - Ja (React Native, o.J. c) - Ja (React Native, o.J. d)	 - Ja (Appcelerator, o.J. c) - Ja (Appcelerator, o.J. d) - Ja (Appcelerator, o.J. e)
Look & Feel - UI-Komponenten fügen sich in das Look & Feel der Plattform ein - Lifecycle-Events der Plattform werden unterstützt	- Ja (React Native, o.J. e) - Ja (React Native, o.J. f)	- Ja (Appcelerator, o.J. f) - Nein
Entwicklungsumgebung - Programmiersprache - JavaScript-Transpiler - Erstellung von lauffähigen Anwendungen	- ECMAScript 2015 - vorhanden (React Native, o.J. g) - Ja (React Native, o.J. i)	- JavaScript (Appcelerator, o.J. b) - nicht notwendig - Ja (Appcelerator, o.J. h)
Zukunftssicherheit - Release in den letzten 3 Monaten - Bereits ein stabiles Release veröffentlicht	- Ja: 4.1.17 - Ja ((React Native, o.J. j))	- Ja - Ja (Appcelerator, 2016)

	Xamarin Plattform	**Flutter**
Lizenz und Kosten - Open Source-Lizenz - Ohne Kosten nutzbar	- kommerziell - kostenlos nur für Open Source Projekte (Xamarin, o.J. a)	- keine Open Source Institute-Lizenz - Keine Kosten (Flutter, o.J. a)
Unterstützte Plattformen - unterstützte Plattformen	- iOS, Android (Xamarin, o.J. b)	- iOS, Android (Flutter, o.J. b)
Zugriff auf plattformspezifische Funktionen - GPS-Sensor - Speichern von Daten - Teilen von Informationen	- Ja - Ja - Ja (Xamarin, o.J. c)	- Nein - Nein - Nein (Flutter, o.J. c)
Look & Feel - UI-Komponenten fügen sich in das Look & Feel der Plattform ein - Lifecycle-Events der Plattform werden unterstützt	- Ja (Xamarin, o.J. c) - Ja (Xamarin, o.J. d und Xamarin, o.J. e)	- Nein (Flutter, o.J. d)
Entwicklungsumgebung - Programmiersprache - JavaScript-Transpiler - Erstellung von lauffähigen Anwendungen	- C# (Xamarin, o.J. c) - Nein - Ja (Xamarin, o.J. f)	- Dart (Flutter, o.J. e) - Nein (Flutter, o.J. f) - Ja (Flutter, o.J. h)
Zukunftssicherheit - Release in den letzten 3 Monaten - Bereits ein stabiles Release veröffentlicht	- Ja - Ja (Boggan, 2016)	- Nein: 7.12.15 - Nein, Pre-Alpha (Flutter, o.J. b)

Die Frameworks Appcelerator Titanium und Xamarin Plattform sind nur kostenpflichtig nutzbar und erfüllen daher die Kriterien nicht. Sie unterstützen zudem nur die Plattformen iOS und Android, nicht den Browser. Das Framework Flutter wird nicht unter einer Open Source-Lizenz nach der Open Source Initiative veröffentlicht. Es ist zudem explizit nicht für produktive Anwendungen geeignet. Das letzte Release liegt mehr als ein Jahr zurück. Damit erfüllt auch dieses Framework die Kriterien nicht. Die Frameworks Ionic, NativeScript und React Native erfüllen alle Anforderungen und werden im nächsten Kapitel detailliert evaluiert.

4.3. Auswahl anhand der Soll-Kriterien

4.3.1. Ionic

Ionic ist ein WebView-basiertes Framework zur Erstellung von Hybrid-Apps. Die erstellten Anwendungen lassen sich als Web-Anwendung im Browser (Ionic, o.J. e) und mittels Apache Cordova auf den Plattformen iOS und Android ausführen (Ionic, o.J. e). Das Framework stellt eine Reihe von UI-Komponenten zur Verfügung, die sich in das Look & Feel der jeweiligen Plattform einfügen (Ionic, o.J. e) und verwendet das JavaScript-Framework Angular2 zur Implementierung der Anwendungslogik (Ionic, o.J. e). Mit Ionic erstellte Anwendungen werden in TypeScript geschrieben und zu JavaScript transpiliert (Ionic, o.J. j)..

4.3.1.1. Plattformunterstützung

Da die Anwendung auf Smartphones in einem WebView ausgeführt wird, ist sowohl das Benutzerinterface als auch die Anwendungslogik ohne weiteres in einem Browser lauffähig. Native Plugins stellen eine einheitliche API zum Zugriff auf plattformspezifische Funktionen bereit. Bei der Ausführung als Web-App müssen plattformspezifische Funktionen, die im Browser nicht zur Verfügung stehen, deaktiviert werden. Das ist beispielsweise beim Teilen von Informationen mit anderen Anwendungen der Fall. Dies wird von iOS und Android unterstützt, im Browser existiert aber kein vergleichbares Konzept. Ionic stellt hier detaillierte Möglichkeiten zur Verfügung die Plattform zu identifizieren, auf der die Anwendung ausgeführt wird (Ionic, o.J. f). Vorteilhaft ist, dass die API einiger Plugins mit der offiziellen W3C-Spezifikation identisch ist und so beispielsweise die Abfrage von GPS-Koordinaten sowohl bei der Ausführung als Hybrid-App, als auch als Web-App mit identischem Code möglich ist (Ionic, o.J. i). Im Vergleich mit den anderen Frameworks verwendet Ionic daher am wenigsten plattformspezifischen Code und ein einheitliches Set von Technologien..

4.3.1.2. Entwicklungsumgebung

Ionic lässt sich schnell und unkompliziert über die Paketverwaltung npm installieren. Ein Kommandozeilentool, die Ionic CLI erleichtert das Erstellen von neuen Projekten auf Basis vorgefertigter Templates. (Ionic, o.J. l) Während der Entwicklung übernimmt die CLI die automatische Neukompilierung und Darstellung des aktuellen Entwicklungsstandes per Live-Reload im Browser, einem Emulator oder auf einem mobilen Gerät. Mit Ionic lassen sich Anwendungen erstellen, die als Hybrid-Apps auf mobilen Betriebssystemen lauffähig sind. (Ionic, o.J. g)

Obwohl das Ionic zugrundeliegende Framework Angular2 generell für die automatische Durchführung von Testfällen ausgelegt ist (Angular, o.J. b), findet sich im Framework Ionic keine Infrastruktur zum automatischen Testen. Es ist möglich diese Funktionalität mit zusätzlicher Software nachzurüsten. (Morony, 2016)

4.3.1.3. Einfache Entwicklung

Ionic bietet eine ausführliche Dokumentation des Frameworks mit Code-Beispielen (Ionic, o.J. p). Es stehen verschiedene Kanäle zur Kommunikation mit anderen Nutzern des Frameworks zur Verfügung (Ionic, o.J. e).

Durch die Verwendung von HTML und CSS zur Implementierung des Benutzerinterfaces (Ionic, o.J. h) verfügt Ionic über die größte Flexibilität und Erweiterbarkeit im Vergleich zu den anderen Frameworks. Dieses einheitliche Set aus Webtechnologien erleichtert den Einstieg ins Framework und erfordert vom Entwickler einen vergleichsweise geringen Lernaufwand. Da Ionic in einem Browser oder WebView ausgeführt wird, lassen sich prinzipiell alle existierenden JavaScript-Bibliotheken verwenden, auch solche für die der Zugriff auf das DOM des Browsers notwendig ist (Chart.js, o.J.).

4.3.1.4. Wartbarkeit

Das Ionic zugrundeliegende Framework Angular2 verfügt über ein eigenes Modulsystem und bietet mehrere Konzepte zur Strukturierung des Quellcodes, wie Templates, Services, Directives und Components, welche die Wiederverwendung von Anwendungslogik ermöglichen. Insbesondere ermöglichen Components auch die Wiederverwendung von HTML-Code.

Diese von Angular2 vorgegebenen Bausteine entsprechen dem MVC-Architekturmuster und folgen damit dem Prinzip Separation of Concerns. Das Template ist für die Darstellung der Benutzeroberfläche verantwortlich und entspricht dem View. Es ist einem Component zugeordnet, der für die Reaktion auf Benutzeraktionen verantwortlich ist und die Funktion des Controllers erfüllt. Services enthalten die Anwendungslogik und stellen damit das Model dar. (Angular, o.J. a)

Die von Ionic eingesetzte Sprache TypeScript, ist auf die Erstellung und Wartung großer JavaScript-Codebasen ausgelegt (TypeScript, o.J. a). TypeScript ist eine Obermenge von ECMAScript 2015, welche durch den TypeScript-Compiler zu JavaScript übersetzt wird. Der generierte JavaScript-Code soll der Sprachspezifikation zufolge dem ursprünglichen TypeScript-Code weitestgehend entsprechen. Dabei ist jeder JavaScript-Code funktionsfähiger TypeScript-Code. (Microsoft, 2016, 1).
Mittels TypeScript lassen sich Sprachkonstrukte der kommenden JavaScript-Version ES6 verwenden. TypeScript unterstützt zusätzlich eine statische Typisierung. (Ionic, o.J. j) Die Sprache bietet Sprachkonstrukte zur Kapselung von Daten und Methoden (Microsoft 2016, 120 f).

Die von Ionic angebotenen Konzepte zur modularen Strukturierung des Quellcodes und die Funktionen von TypeScript ermöglichen daher die Konstruktion von wartbarer Software, wie in Kapitel 2.2 erörtert wurde..

4.3.2. Nativescript

NativeScript ist ein Runtime-basiertes Framework zur Erstellung von Hybrid-Apps für die Plattformen iOS und Android. Zur Implementierung der Anwendungslogik kann entweder JavaScript oder Angular2 in Kombination mit TypeScript verwendet werden. (NativeScript, o.J. r)

NativeScript ermöglicht den Zugriff auf native APIs aus der Hybrid-App heraus, wodurch der Zugriff auf alle plattformspezifischen Funktionen ermöglicht wird. Zusätzlich wird eine API angeboten, die eine Reihe von nativen API-Aufrufen abstrahiert und vereinheitlicht, beispielsweise ein http-Modul das sich plattformunabhängig nutzen lässt.

Zur Implementierung des Benutzerinterfaces verwendet NativeScript XML-Markup, wodurch sich die nativen UI-Komponenten der jeweiligen Plattform nutzen lassen. (Atanasov, 2014) Um das Aussehen von Elementen zu beschreiben verwendet NativeScript eine Untermenge von CSS (NativeScript, o.J. o).

Die Erstellung von Web-Apps ist nicht Ziel des Projektes, allerdings werden Methoden vorgeschlagen, wie die Erstellung von Hybrid-Apps und Web-Apps aus einer gemeinsamen Codebasis realisiert werden kann (Stoychev, 2016). Ein Beispielprojekte unterstützt bei der Erstellung von Hybrid- und Web-Anwendungen aus einer gemeinsamen Codebasis (Walker, o.J. a). NativeScript plant in Zukunft, die Möglichkeiten zur Wiederverwendung von Code für Hybrid- und Web-Anwendungen auszubauen (NativeScript, o.J. r).

4.3.2.1. Plattformunterstützung

Das von NativeScript verwendete XML-Markup zur Implementierung von Benutzerinterfaces kann nicht im Browser wiederverwendet werden. Das Benutzerinterface muss daher für den Browser gänzlich oder in großen Teilen erneut implementiert werden. Im Beispielprojekt findet sich HTML-Code für die Ausführung im Browser und XML-Markup für die Ausführung auf mobilen Betriebssystemen (Walker, o.J. b).

Die Möglichkeit des direkten Zugriffs auf native APIs führt kann unter Umständen zu redundantem plattformspezifischem Code führen, da sich diese APIs je nach Plattform unterscheiden. (NativeScript, o.J. n). Dies ist nur in den Fällen relevant, in denen NativeScript keine einheitliche API zum Zugriff auf diese Funktionen bereitstellt (Atanasov, 2014). NativeScript bietet Methoden, mit denen die Plattform zur Laufzeit identifiziert werden kann (NativeScript, o.J. k).

4.3.2.2. Entwicklungsumgebung

Eine Infrastruktur zur automatischen Durchführung von Tests ist vorhanden (NativeScript, o.J. l). Ein Kommandozeilentool übernimmt das erstmalige Aufsetzen von Projekten und die Erstellung von lauffähigen Anwendungen für mobile Betriebssysteme (NativeScript, o.J. m). Das Beispielprojekt zur Erstellung von Web-Apps auf Basis von NativeScript verfügt über einen Live-Reload zur automatischen Darstellung des Entwicklungsstands im Browser (Walker, o.J. a).

4.3.2.3. Einfache Entwicklung

Auf der Webseite des Frameworks findet sich eine ausführliche Dokumentation mit Code-Beispielen (NativeScript, o.J. k). Es stehen mehrere Kanäle für die Kommunikation zwischen Anwendern des Frameworks zur Verfügung (NativeScript, o.J. q).

NativeScript baut auf dem JavaScript-Framework Angular2 auf. Mit NativeScript entwickelte Anwendungen sind daher nach dem MVC-Architekturmuster strukturiert. Da das Framework auf mobilen Geräten nicht in einem Browser oder WebView ausgeführt wird können für die native Ausführung auf mobilen Geräten nur Dritt-Bibliotheken eingebunden werden, die nicht auf den DOM zugreifen. Beispielsweise benötigt chart.js, eine Bibliothek zur Erstellung statistischen Schaubildern, einen Browser, bzw. das DOM zur Ausführung (Chart.js, o.J.). NativeScript erlaubt es außerdem native Dritt-Bibliotheken zu verwenden, die für die jeweilige Zielplattform zur Verfügung stehen. (Martin, 2016)

Zur Erstellung von Benutzerinterfaces verwendet NativeScript eine Untermenge von CSS (NativeScript, o.J. o) und XML-Markup (NativeScript, o.J. s), mit welchem sich die nativen UI-Komponenten des jeweiligen mobilen Betriebssystems nutzen lassen. Dies bietet eine hohe Anwendungsperformance und Anpassung an das Look & Feel der Zielplattform, führt aber dazu, dass die Aneignung von frameworkspezifischem Wissen notwendig ist. Die Möglichkeit auf native APIs zuzugreifen und native Dritt-Bibliotheken zu verwenden erfordert zudem ein detailliertes Wissen über die jeweiligen Zielplattformen. Es wird also Kenntnisse über die Funktionsweise der Plattform Browser und auch über die Betriebssysteme iOS und Android benötigt.

4.3.2.4. Wartbarkeit

Da NativeScript, wie auch das Framework Ionic, Angular2 und TypeScript verwendet, ist die Wartbarkeit von mit diesem Framework erstellten Anwendungen als gut zu bewerten (NativeScript, o.J. q). Die Anforderungen an die Verwendung von Architekturmustern, ein Modulsystem, die Wiederverwendbarkeit von Codestrukturen, Sprachkonstrukte zur Kapselung von Informationen und eine statische Typisierung sind in gleichem Maße, wie beim Framework Ionic gegeben. Die Separierung der Benutzerinterfaces nach mobilen Betriebssystemen und Browser führt allerdings zu redundantem Code, der aus Sicht der Wartbarkeit zu vermeiden ist. Der Einsatz von plattformspezischem Plugins und der Zugriff auf native APIs von mobilen Betriebssystemen führt zu plattformspezifischem Code und

dadurch ebenfalls zu Redundanzen, welche die Analysierbarkeit und Änderbarkeit der Anwendung beeinträchtigen.

4.3.3. React Native

React Native ist ein Runtime-basiertes Framework zur Erstellung von Hybrid-Apps (React Native, o.J. m) für die Plattformen iOS und Android (React Native, o.J. u). Es basiert auf dem Framework React, das auf die Entwicklung von reinen Web-Anwendung ausgelegt ist (React Native, o.J. u). React Native verwendet ES6 und JSX, eine Syntax um XML-Markup innerhalb von JavaScript zu verwenden. Ähnlich wie NativeScript verwendet React Native eine XML-basierte Syntax zur Beschreibung des Benutzerinterfaces. Diese wird auf dem mobilen Betriebssystem mit nativen UI-Komponenten dargestellt. (React Native, o.J. q) Zur Anpassung der Darstellung von Elementen wird bei React Native ebenfalls eine Untermenge von CSS verwendet. Ein Unterschied zu im Browser verwendeten CSS findet sich darin, dass die Angaben für Höhe und Breite ohne Einheit angegeben werden (React Native, o.J. l), dies wird im Browser nicht unterstützt (w3schools.com, o.J.).
Eine detaillierte Auflistung der Unterschiede zu im Browser verwendeten CSS findet sich nicht, React Native gibt aber an, dass die Funktionsweise *"usually match[es] how CSS works on the web"* (React Native, o.J. o).

Es existiert zusätzliche Software, welche die Verwendung der React Native UI-Komponenten im Webbrowser ermöglicht (Gallagher, o.J. a). Allerdings stehen nicht alle von React Native angebotenen UI-Komponenten auch für den Browser zur Verfügung (Gallagher, o.J. b). Daneben findet sich auch eine Methode, die ähnlich wie der Ansatz von NativeScript die Benutzeroberfläche für den Browser mit HTML und für mobile Betriebssysteme mit XML-Markup implementiert (Vallon, o.J.).

4.3.3.1. Plattformunterstützung

Das von React Native verwendete XML-Markup zur Implementierung von Benutzerinterfaces kann ohne Weiteres im Browser wiederverwendet werden. Es finden sich zwei Methoden, um mit React Native erstellte Anwendungen auch im Browser auszuführen: Die erste Methode ermöglicht die Verwendung des XML-Markup von React Native im Browser. Es stehen allerdings nicht alle UI-Komponenten auch im Browser zur Verfügung. (Gallagher, o.J. a) Bei der zweiten Methode werden die Benutzerinterfaces jeweils separat für mobile Betriebssysteme und den Browser implementiert (Vallon, o.J.). Bei beiden Methoden muss

daher das Benutzerinterfaces für den Browser teilweise oder komplett neu implementiert werden. React Native bieten Methoden um die ausführende Plattform zur Laufzeit zu identifizieren (React Native, o.J. n).

4.3.3.2. Entwicklungsumgebung

Ein Kommandozeilentool, die React Native-CLI, setzt das Projekt auf, bietet einen Live-Reload und erzeugt lauffähige Anwendungen (React Native, o.J. u). Eine Infrastruktur zur Durchführung von automatischen Tests ist in das Framework integriert (React Native, o.J. p).

4.3.3.3. Einfache Entwicklung

React Native verfügt über eine ausführliche Dokumentation mit Code-Beispielen (React Native, o.J. u). Es stehen verschiedene Kanäle für die Kommunikation zwischen Anwendern des Frameworks zur Verfügung (React Native, o.J. t). Da React Native auf mobilen Geräten nicht in einem WebView ausgeführt wird, können auf mobilen Geräten nur Dritt-Bibliotheken verwendet werden, die keinen Zugriff auf das DOM benötigen. XML-Markup und die abweichende Funktionsweise von CSS zur Erstellung von Benutzerinterfaces erfordern die Aneignung von frameworkspezifischem Wissen. Bestehende Erfahrungen mit HTML und CSS lassen sich nur teilweise anwenden.

4.3.3.4. Wartbarkeit

React, bzw. React Native basiert nicht auf dem MVC-Architekturmuster. Während mit Angular2 erstellte Anwendungen der Aufteilung in Model, View und Controller folgen, existiert diese Aufteilung bei React Native nicht. (Hunt, 2013) Zudem werden bei React Native Anwendungslogik und Code zur Beschreibung des Benutzerinterfaces vermischt (React Native, o.J. s). Dadurch wird die sonst übliche Trennung zwischen HTML, CSS und JavaScript aufgehoben.

Zentrales Element von React Native sind sogenannte Components. Diese ermöglichen es, modulare Codestrukturen zu schaffen, die innerhalb der Anwendung wiederverwendet werden können (React Native, o.J. r).

React Native verwendete JSX, eine Erweiterung von ECMAScript 2015 mit der sich XML-Markup in JavaScript verwenden lässt (React Native, o.J. q). Es stehen daher die Sprachkonstrukte von ES6 und dessen Modulsystem während der Entwicklung zur Verfügung.

Die Erstellung von wartbaren Anwendungen mit React Native ist durch die Verwendung von ECMAScript 2015 grundlegend ermöglicht. Eine statische Typisierung und Sprachkonstrukte für Kapselung von Informationen würde sich zusätzlich positiv auf die Wartbarkeit der Anwendung auswirken. Zusätzlich erschwert das Fehlen eines bekannten Architekturmusters die Analysierbarkeit des Quellcodes.

4.4. Auswahl eines geeigneten Frameworks

4.4.1. Unterstützte Plattformen

Nach der Evaluation der drei Frameworks Ionic, NativeScript und React Native zeichnen sich zwei Konzepte zur Erstellung von mobilen Anwendungen und Web-Anwendungen aus gemeinsamer Codebasis ab. Erstens die Wiederverwendung des für das Benutzerinterface verantwortlichen Codes für alle Plattformen. Zweitens voneinander getrennte Implementierungen des Benutzerinterfaces, einmal für die Plattform Web und einmal für die Plattformen Android und iOS. Bei beiden Konzepten lässt sich dabei die Anwendungslogik zwischen allen Plattformen grundlegend wiederverwenden.

Da sich mobile Plattformen von der Plattform Web sowohl hinsichtlich Bildschirmgröße, als auch Bedienkonventionen teilweise stark unterscheiden, ermöglicht das Konzept der getrennten Implementierung des Benutzerinterfaces eine gezielte Ausrichtung auf die technischen Eigenschaften und Bedienkonventionen der jeweiligen Plattform. Die Implementierung und Pflege von separaten Benutzerinterfaces für mobile Betriebssysteme und den Browser bedeutet allerdings einen zusätzlichen Aufwand und führen zu Redundanzen im Quellcode des Benutzerinterfaces. Auch ist zu befürchten, dass separate Benutzerinterfaces zu redundanter Anwendungslogik führen, beispielsweise wenn für einzelne Entwurfsprobleme im Detail unterschiedliche Lösungsansätze benötigt werden. Unter der Prämisse einer guten Wartbarkeit, wie sie im Rahmen dieser Arbeit beschrieben wird, ist von der Erstellung redundanten Codes generell abzusehen.

Dagegen bieten HTML und CSS - insbesondere durch die von Ionic bereitgestellten Anpassungen an die Bedienkonzepte mobiler Plattformen - Möglichkeiten auf die jeweilige Plattformen und Bildschirmgrößen unterschiedlich zu reagieren, ohne separate Benutzerinterfaces zu implementieren und zu pflegen. Das erste Konzept, die

Wiederverwendung des Benutzerinterfaces für alle Plattformen, ist daher aus der Perspektive der Wartbarkeit vorzuziehen.

Die einzelnen Frameworks unterscheiden sich in ihrem Umgang mit plattformspezifischen Funktionen. Wird beispielsweise der von NativeScript angebotene Zugriff auf native APIs des jeweiligen mobilen Betriebssystems genutzt, um auf den GPS-Sensor zuzugreifen, führt dies zu zwei verschiedenen Code-Strukturen für iOS und Android und einer dritten Struktur für den Browser. Das Framework Ionic bietet durch die Ausführung der Anwendung in einem WebView oder Browser, abgesehen von Funktionen, die nicht im Browser zur Verfügung stehen, eine plattformunabhängige und einheitliche Umgebung und Programmierschnittstellen. Plattformspezifischer Code lässt sich so vermeiden.

4.4.2. Entwicklungsumgebung

Es finden sich keine nennenswerten Unterschiede in dieser Kategorie. Alle Frameworks bieten ein Kommandozeilentool zur einfachen Installation des Frameworks und zur Erstellung von lauffähigen mobilen Anwendungen. Eine Infrastruktur für automatisches Testen ist mit Ausnahme von Ionic bei allen Frameworks vorhanden. Für Ionic kann diese Funktionalität vergleichsweise einfach nachgerüstet werden.

4.4.3. Einfache Entwicklung

Dritt-Bibliotheken können mit dem Framework Ionic uneingeschränkt verwendet werden. NativeScript und React Native erlauben dies nur eingeschränkt. Diese beiden Frameworks erfordern zudem die Aneignung von zusätzlichem, frameworkspezifischem Wissen und teilweise auch Kenntnisse über die nativen APIs von mobilen Betriebssystemen. Ionic bietet hier mit HTML und CSS ein einheitliches Set an bekannten Technologien und erfordert so einen geringen Lernaufwand. Zusätzliche Funktionen von TypeScript, wie statische Typisierung und nicht in ECMAScript 2015 vorgesehene Sprachkonstrukte können bei Bedarf komplett ignoriert werden, sodass die Verwendung von TypeScript wenig Zusatzwissen erfordert.

4.4.4. Wartbarkeit

Die Frameworks Ionic und NativeScript verwenden beide das JavaScript-Framework Angular2 zur Implementierung der Anwendungslogik. Angular2 folgt dem MVC-Architekturmuster zur Strukturierung des Quellcodes und bietet eine Reihe von Konzepten zur Wiederverwendung

und Modularisierung von Quellcode. Zusätzlich setzen beide Frameworks TypeScript ein, um ECMAScript 2015 um statische Typisierung und Sprachkonstrukte zu erweitern.

React Native verwendet Components, um wiederverwendbare, modulare Codestrukturen zu schaffen, die sowohl Anwendungs- als auch Darstellungslogik enthalten. Diese Auflösung der Trennung zwischen Darstellungslogik und Anwendungslogik entspricht nicht dem Prinzip Separation of Concerns und ist daher als schlecht wartbar zu bewerten. Neben den Components werden in der Dokumentation keine weiteren Konzepte zur Wiederverwendung oder Modularisierung der Software aufgezeigt. React Native folgt explizit nicht dem MVC-Architekturmuster, auf die Strukturierung von Anwendungen wird in der Dokumentation des Frameworks nicht eingegangen. Die Verwendung von ECMAScript 2015 erhöht im Vergleich zu JavaScript die Wartbarkeit der Anwendung, weitere Sprachkonstrukte zur Kapselung oder statische Typisierung werden aber nicht unterstützt.

Aus Perspektive der Wartbarkeit sind die beiden Frameworks, die eine Kombination aus dem MVC-Framework Angular2 und TypeScript verwenden vorzuziehen. Zwischen den beiden Frameworks Ionic und NativeScript gibt es wegen des hohen Anteils gleicher Technologien keine wartungsrelevanten Unterschiede. Die bei NativeScript entstehenden Redundanzen durch den Zugriff auf native APIs und durch das verwendete Konzept der Trennung des Benutzerinterfaces nach Plattformen wurden im Unterpunkt “Unterstützte Plattformen” behandelt. Diese beiden Punkte wirken sich negativ auf die Wartbarkeit des Frameworks NativeScript aus.

4.4.5. Fazit

Zur Implementierung des Prototyps wird das Framework Ionic ausgewählt. Im Vergleich mit den anderen Optionen ist bei diesem Framework am wenigsten plattformspezifischer Code notwendig und der Quellcode kann mit einem vergleichsweise geringen Redundanzen für alle Plattformen verwendet werden. Das Framework ermöglicht es, wartbare Anwendungen zu konstruieren. frameworkspezifisches Wissen ist nur in geringem Maße notwendig, da vorhandenes Wissen aus dem Bereich der Webtechnologien angewendet werden kann, ohne neue Konzepte oder Technologien zu erlernen. Da das Framework auf die Ausführung in einem WebView oder Browser ausgelegt ist, bietet es die höchste Portabilität auf andere Frameworks und Entwicklungsansätze. Auch ist es mit zusätzlichem Aufwand möglich, die Abhängigkeit des Quellcodes von Ionic, mit dem Ziel separate Codebasen für mobile und Web-

Anwendungen zu schaffen, aufzuheben. Zudem unterstützt das Framework ausdrücklich sowohl die Entwicklung von mobilen Anwendungen, als auch von Web-Anwendungen.

5. Evaluation des Frameworks Ionic

In diesem Kapitel wird zunächst die prototypische Implementierung der Anwendung dargestellt. Basierend darauf werden die aufgestellten Kriterien anhand der prototypisches Implementierung validiert. Dabei wird nur auf diejenigen Muss- und Soll-Kriterien eingegangen, die durch den Prototypen validiert werden können. Beispielsweise sind die Existenz von Kanälen zur Kommunikation zwischen Anwendern des Frameworks oder die Lizenzierung des Frameworks keine technischen Fragestellungen, deren Erfüllung sich durch einen funktionierenden Prototyp nachweisen ließe. Auf solche Kriterien wurde im vierten Kapitel bereits hinreichend eingegangen.

5.1. Implementierung der Anwendung

Die Anwendung ist in drei Navigationspunkte unterteilt. Jeder Navigationspunkt besteht aus einer Ansicht (View), über die sich weitere Views aufrufen lassen. Die Navigation zwischen den einzelnen Views erfolgt über eine ein- und ausklappbare Seitenleiste. Ab einer Bildschirmbreite von 1200 Pixeln wird die Seitenleiste dauerhaft im ausgeklappten Zustand eingeblendet (Abb. 4).

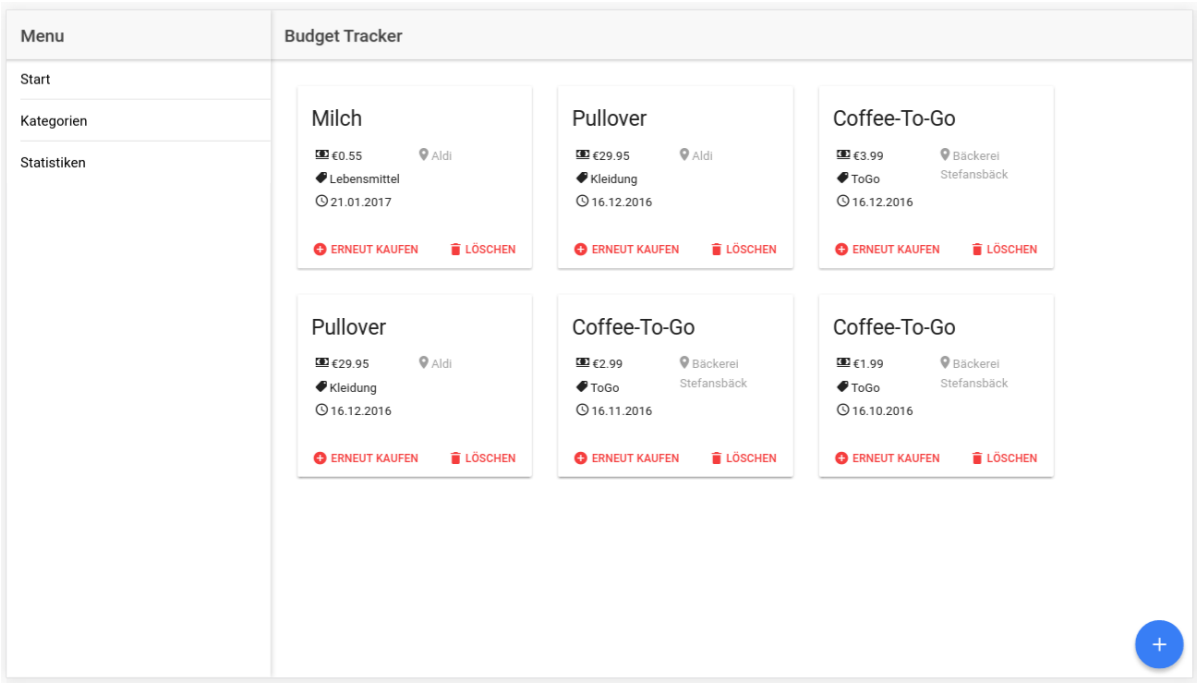

Abbildung 4: Dauerhaft ausgeklappte Seitenleiste

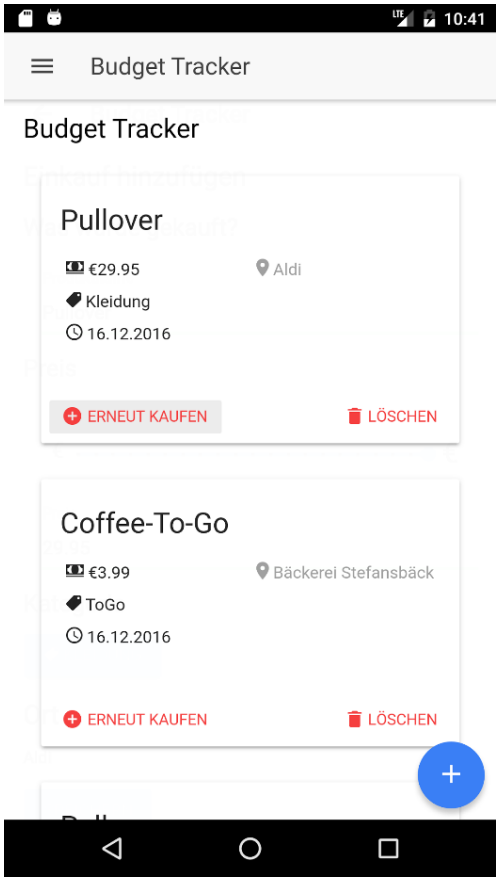

Abbildung 5: Geschlossene Seitenleiste

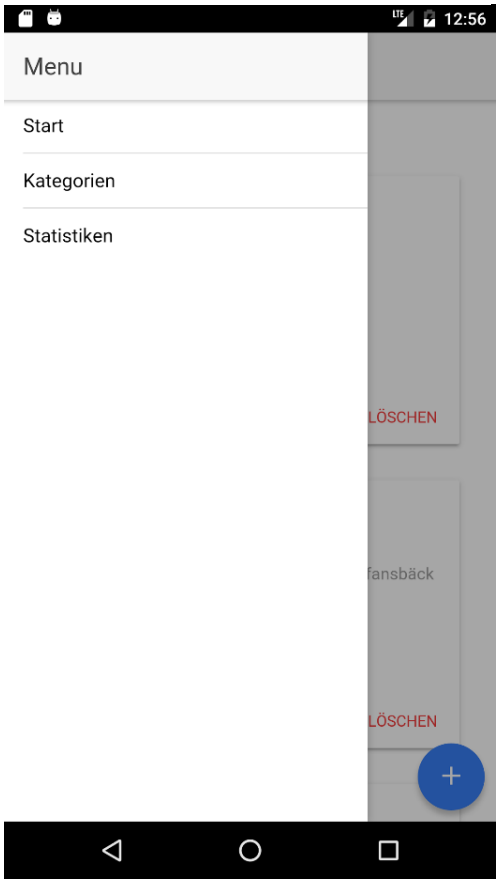

Abbildung 6: Geöffnete Seitenleiste

Bei einer geringeren Bildschirmbreite wird die Seitenleiste nicht angezeigt und kann über einen Menu-Button geöffnet werden (Abb. 5). Wird die Seitenleiste geöffnet, legt sie sich über die geöffnete Ansicht (Abb. 6).

Diese Funktionalität ist nicht Teil von Ionic, sondern wurde im Rahmen der Erstellung des Prototypen zusätzlich implementiert.

Die drei zentralen Navigationspunkte sind die Views Start, Kategorien und Statistiken. Beim ersten Start der Anwendung wird eine Reihe von beispielhaften Einträgen und Kategorien erzeugt.

Der View “Start” ist der Einstiegspunkt in die Anwendung. Hier werden die bereits getätigten Einkäufe aufgelistet. Es können neue Einträge hinzugefügt und bestehende Einträge gelöscht werden. Über die Funktion “Erneut kaufen” können bestehende Einträge dazu verwendet werden, neue Einträge zu erstellen. Dies ist dann hilfreich, wenn ein einzelnes Produkt zu einem anderen Preis, oder an einem anderen Ort erneut gekauft wurde, die Eigenschaften Name und Kategorie aber gleich bleiben.

Wird ein neuer Einkauf angelegt öffnet sich die Ansicht “Einkauf hinzufügen” (Abb. 7). Dieser View dient dem Erfassen von neuen Einkäufen. Es muss ein Produktname und ein Preis angegeben werden. Über den aktuellen Aufenthaltsortes des Gerätes wird versucht zu bestimmen, in welchem Geschäft der Einkauf getätigt wurde. Dazu fragt die Anwendung vom GPS-Sensor des Endgerätes die aktuellen GPS-Koordinaten ab. Über die Google Places API lassen sich

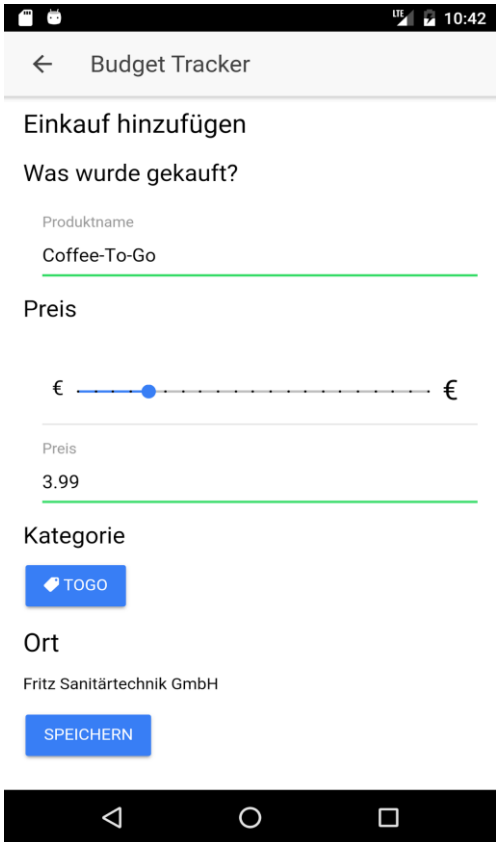

Abbildung 7: Ansicht "Einkauf hinzufügen"

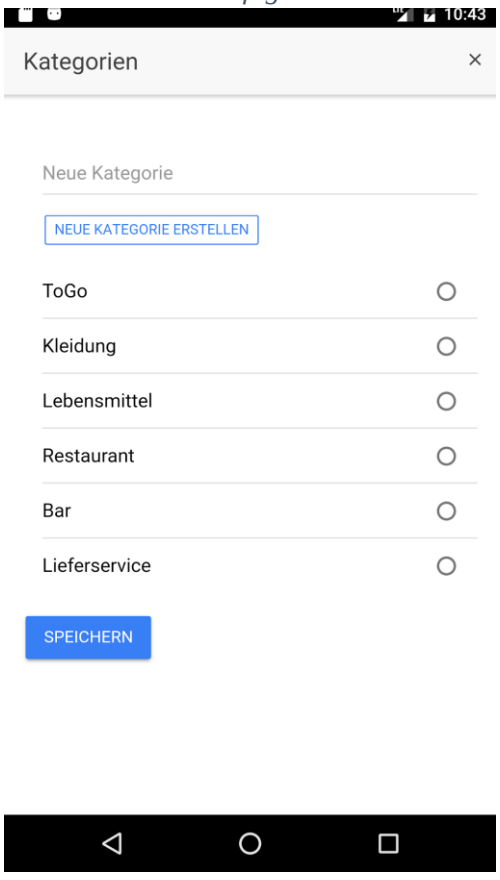

Abbildung 8: Liste aller Kategorien

mittels der Koordinaten die umliegenden Karteneinträge abrufen und nach Typ filtern. Die Anwendung übernimmt Bezeichnung und Adresse des nächstgelegenen Eintrags vom Typ Geschäft. Der Google Places API Web-Service lässt sich nicht client-seitig verwenden (Google, o.J. b). Daher wurde ein Web-Service auf Basis von Node.js entwickelt, dessen einzige Aufgabe es ist, mit den vom Zielgerät übermittelten Koordinaten die Google Places API zu kontaktieren und die Antwort an die Anwendung weiterzuleiten.

Wird die Funktionen “Erneut kaufen” genutzt erscheint ebenfalls diese Ansicht. Name, Preis und Kategorie des vorangegangenen Produkts werden dann in die Ansicht übernommen und können abgeändert werden. Zudem wird erneut das Geschäft über die aktuellen GPS-Koordinaten bestimmt. Die Angabe einer Kategorie ist optional.

Bei der Eingabe einer Kategorie öffnet sich eine weitere Ansicht, die eine Liste aller Kategorien anzeigt (Abb. 8). Hier kann dem Einkauf eine Kategorie zugeordnet werden. Es können außerdem neue Kategorien angelegt werden.

Der Navigationspunkt “Kategorien” listet alle Kategorien und die Zahl der Einkäufe pro Kategorie auf (Abb. 9).

Durch den Klick auf eine Kategorie wird eine neue Ansicht geöffnet, in welcher die in der Kategorie enthaltenen Einkäufe angezeigt werden (Abb. 10). Diese Ansicht verfügt über eine eigene

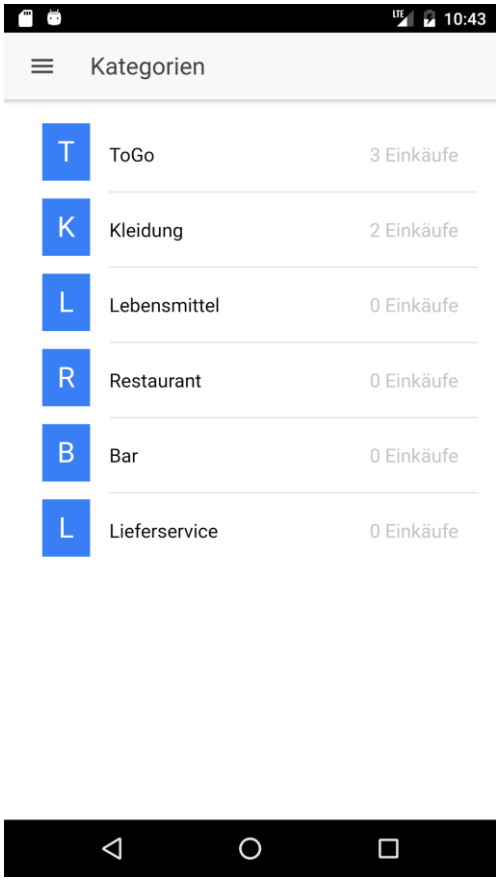

Abbildung 10: Ansicht "Kategorien"

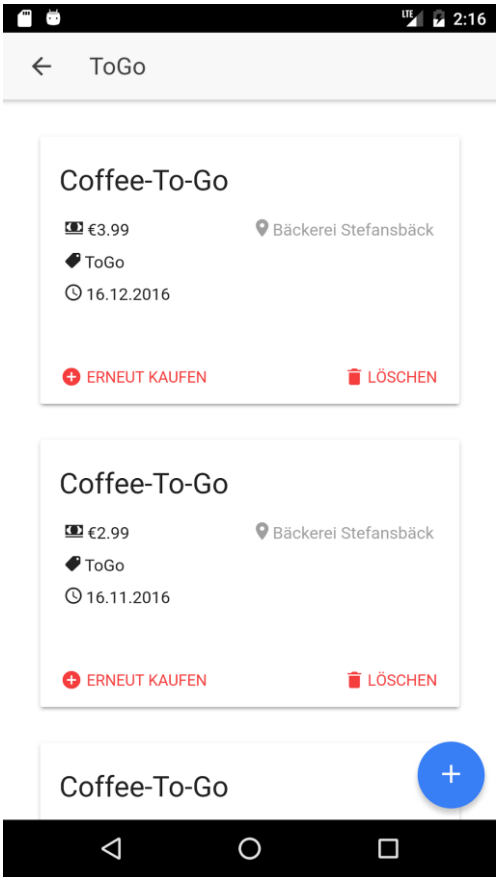

Abbildung 9: Detailansicht einer Kategorie

Anwendungslogik, verwendet zur Darstellung der Benutzeroberfläche aber das Template der Ansicht “Start”. Da die Aufgaben beider Ansichten identisch sind und sich nur die Auswahl, nicht aber die Art der dargestellten Daten ändert, lässt sich so redundanter Code vermeiden. Wird das Template angepasst, ändert sich auch die Darstellung beider Views.

Der Navigationspunkt “Statistiken” (Abb. 11) zeigt Diagramme über die gekauften Produkte an. Es wird die Gesamtsumme der Einkäufe nach Kategorien und nach Geschäften dargestellt. Ein weiteres Diagramm stellt die Ausgabenverteilung auf einzelne Kategorien innerhalb eines Jahres dar. Zur Darstellung wird die JavaScript-Bibliothek Chart.js verwendet.

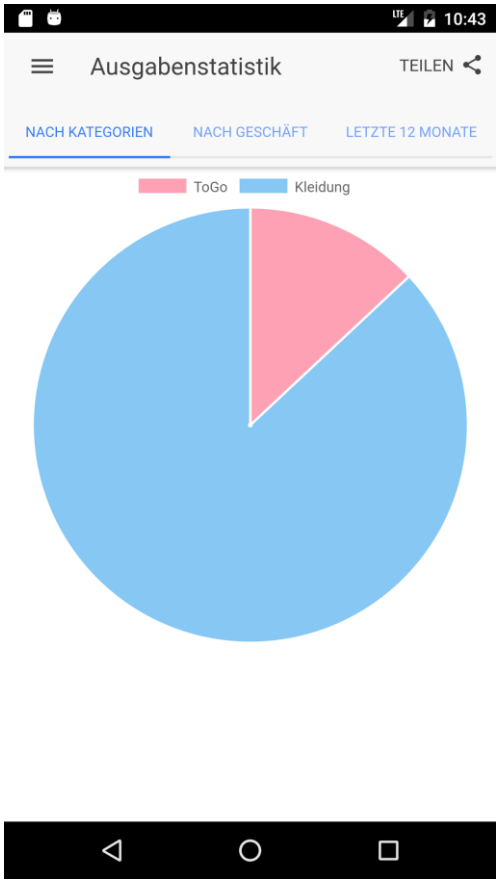

Abbildung 11: Ansicht Statistiken

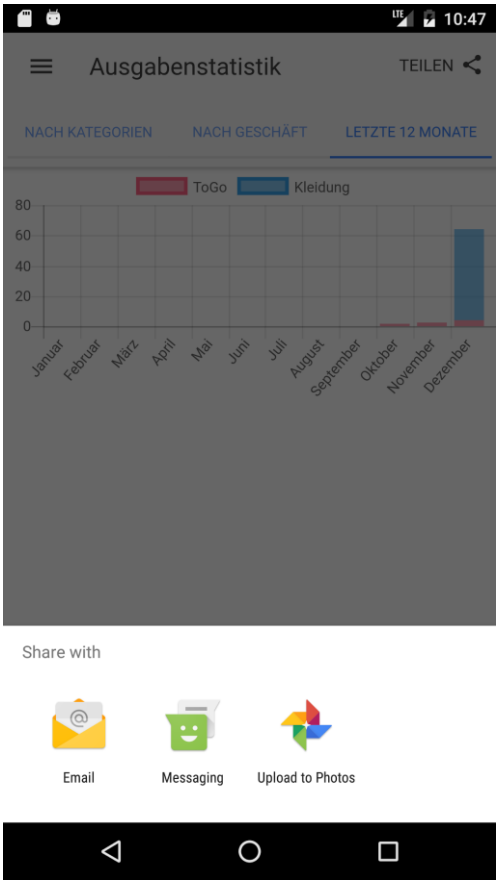

Abbildung 12: Teilen der Statistiken mit anderen Anwendungen

Die Diagramme können über den Button “Teilen” als Bilddatei mit anderen Anwendungen geteilt werden, um sie beispielsweise in einem Chat oder einer E-Mail zu versenden (Abb. 12). Da diese Funktionalität im Browser nicht unterstützt wird, wird der Button “Teilen” auf dieser Plattform nicht angezeigt (Abb. 13).

5.2. Unterstützte Plattformen und Zugriff auf plattformspezifische Funktionen

Das Kommandozeilentool Ionic-CLI ermöglicht es über die folgenden Befehle lauffähige Anwendungen für alle drei Zielplattformen zu erstellen:

```
ionic build <platform> // kann sein: ios, android oder browser
```

Die generierten Anwendungen lassen sich im Emulator oder auf den entsprechenden Plattformen ausführen. Die Web-Anwendung muss im Anschluss über einen Webserver distribuiert werden.

Durch den Einsatz von Cordova steht eine einheitliche API zum Zugriff auf plattformspezifische Funktionen zur Verfügung. So entspricht die API zum Zugriff auf den GPS-Sensor der API im Browser und der entsprechende Programmcode lässt sich für alle Plattformen verwenden:

```
if (navigator.geolocation) {
   Geolocation
   .getCurrentPosition()
   .then((result: any) => {
       result.coords.latitude // liefert den Breitengrad
       result.coords.longitude // liefert den Längengrad
   })
}
```

Zum Speichern von Daten auf dem Gerät stellt Ionic eine Bibliothek bereit, die den Zugriff auf die auf der jeweiligen Plattform verfügbare Methode zum Speichern von Daten abstrahiert. Unter iOS und Android werden die Daten in einer SQLite-Datenbank gespeichert, im Browser wird je nach Verfügbarkeit IndexedDB, WebSQL oder LocalStorage genutzt. (Ionic, o.J. o)

Auch zum Teilen von Informationen steht unter iOS und Android eine einheitliche API zur Verfügung. Da das Konzept des Teilens von Informationen im Browser nicht existiert, ist es notwendig, die aktuelle Plattform zu identifizieren, um diese Funktion dem Nutzer entsprechend anzubieten. Ionic bietet eine API um die ausführende Plattform zur Laufzeit zu identifizieren. Über diese API wird beispielsweise bei Ausführung auf dem Betriebssystem Android eine Liste von Werten geliefert:

```
this.platform.platforms() // liefert ["cordova", "mobile", "android", "phablet"]
```

Analog wird bei Ausführung im Browser der Wert “core” zurückgegeben, wobei dies den Browser bezeichnet:

```
this.platform.platforms() // liefert "core"
```

Zur Anpassung des Benutzerinterfaces an verschiedene Plattformen bietet Ionic ebenfalls Methoden an. Abhängig von der jeweiligen Plattform wird im HTML eine CSS-Klasse gesetzt, mit der sich das Aussehen der Benutzeroberfläche mittels CSS plattformspezifisch anpassen lässt (Ionic, o.J. p). Zudem bietet Ionic eine Directive “showWhen”, die genutzt werden kann, um UI-Elemente je nach Plattform anzuzeigen oder auszublenden:

```
<button showWhen="cordova" ion-button (click)="onShare()">Share</button>
```

Diese showWhen-Directive wird im Prototyp genutzt (Abb. 13), um den Button zum Teilen der Statistiken nur bei nativer Ausführung auf Mobilgeräten anzuzeigen. (Ionic, o.J. f)

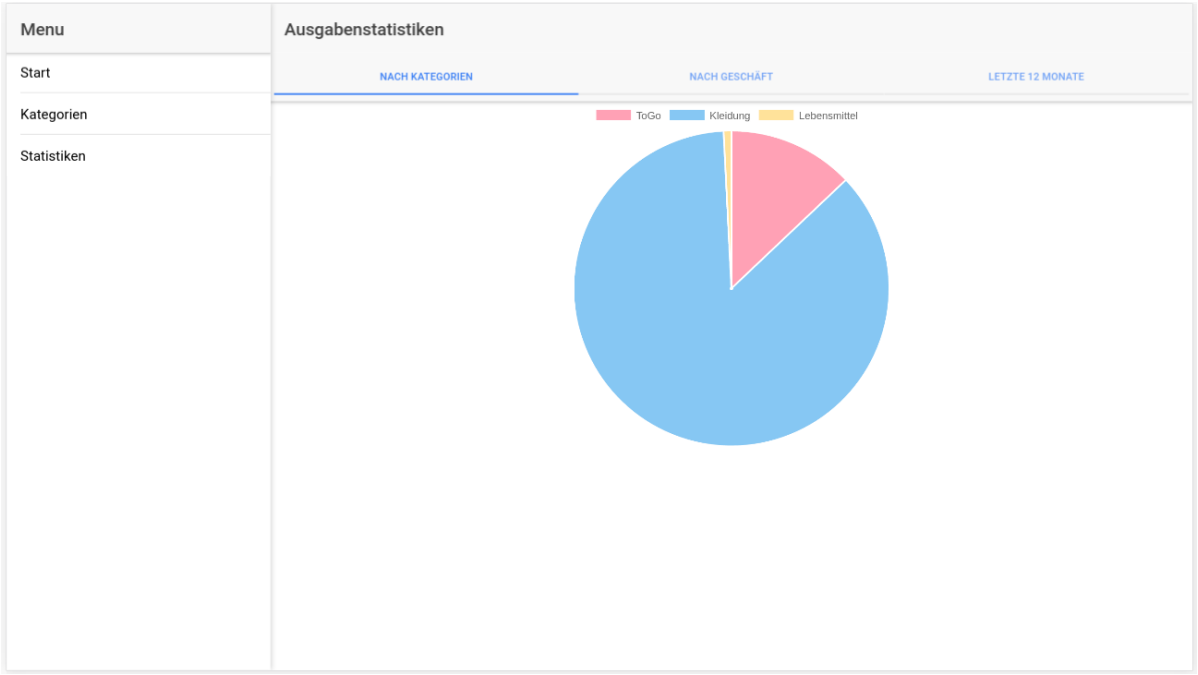

Abbildung 13: Verbergen des Button "Teilen" auf der Plattform Browser

Ionic bietet zudem die Möglichkeit Spalten zu definieren, die zur Darstellung von UI-Elementen verwendet werden (Ionic, o.J. q). Die Anzahl der Spalten kann abhängig von der Bildschirmgröße angepasst werden. Dies wird interessanterweise nicht in der Ionic Dokumentation erwähnt, sondern in einem Pull-Request im zugehörigen GitHub-Repository des Projekts (Agius, 2016). Im Prototyp wird diese Funktionalität verwendet, um die Zahl der Spalten in denen die Einkäufe in der Ansicht “Start” angezeigt werden abhängig von der Bildschirmgröße anzupassen.

Bei großen Bildschirmauflösungen wird auf diese Weise ein dreispaltiges Layout zur Darstellung genutzt (Abb. 4). Auf Mobilgeräten wird ein einspaltiges Layout verwendet (Abb. 14).

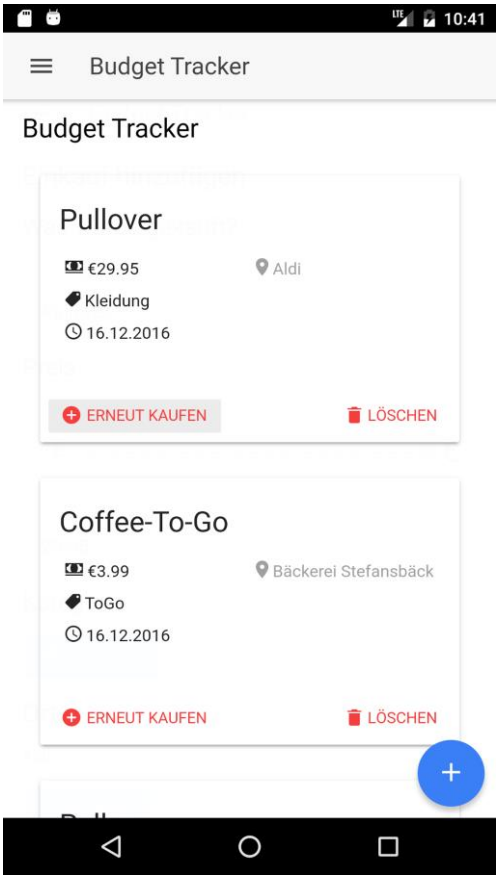

Abbildung 14: Einspaltiges Layout

5.3. Look & Feel

Die UI-Elemente von Ionic sind den nativen Elementen der mobilen Betriebssysteme nachempfunden. Die Darstellung der UI-Komponenten passt sich automatisch der jeweiligen Plattform an (Ionic, o.J. p, Abb. 11 und Abb. 14). Im Browser wird das auch für Android verwendete "Material"-Design genutzt. (Ionic, o.J. p)

Die Lifecycle-Events der mobilen Betriebssysteme werden unterstützt. Über die Events *pause* und *resume* ist es möglich bei der Minimierung und Wiederherstellung der Anwendung Code auszuführen (Ionic, o.J. f):

```
platform.ready().then(() => {
    this.platform.pause.subscribe(() => {
        //...
    });
    this.platform.resume.subscribe(() => {
        //...
    });
});
```

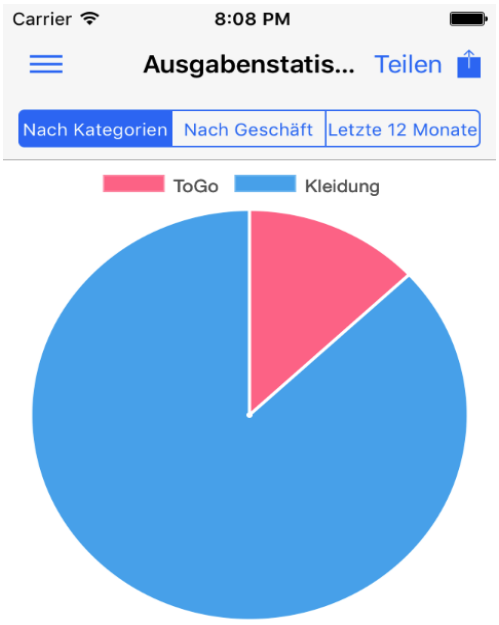

Abbildung 15: Darstellung der Anwendung unter iOS

Beim Minimieren und darauffolgenden Wiederherstellen der Anwendung wird die zuletzt geöffnete Ansicht mit allen eingegebenen Daten geöffnet. Wird die Anwendung beendet und dann erneut geöffnet wird allerdings nicht die zuletzt geöffnete Ansicht geöffnet. Eventuell eingegebene Daten sind gelöscht. Eine eigene Implementierung ist hier notwendig, um die zuletzt geöffnete Ansicht und alle Daten wiederherzustellen. Dies ist nicht Teil der Anforderungen und wurde daher im Prototyp nicht umgesetzt. Im Sinne einer umfassenden Unterstützung von mobilen Betriebssystemen wäre eine vorgefertigte Lösung seitens des Frameworks wünschenswert gewesen.

5.4. Entwicklungsumgebung

Die Ionic CLI bietet einen Live-Reload, um nach Änderungen am Code automatisiert den geänderten Projektstand darzustellen. Mit dem Befehl *ionic serve* startet die Anwendung in einem lokalen Webserver. Die Anwendung kann dann im Browser über die URL

localhost:8100 aufgerufen werden. Bei Änderungen am Code wird die Anwendung automatisch neu geladen.

Der Befehl *ionic emulate* lädt und startet die Anwendung automatisch in einem Emulator mit dem angegebenen mobilen Betriebssystem. Mittels *ionic run* wird die Anwendung direkt in einem verbundenen Gerät gestartet. Wird zusätzlich an einen der beiden Befehle "--livereload" angehängt, wird die Anwendung bei Änderungen am Code automatisch neu auf das Gerät oder den Emulator geladen und gestartet. (Ionic, o.J. n) Über den Befehl *ionic build* werden lauffähige Anwendungen für erstellt. Anwendungen für iOS lassen sich nur auf einem macOS-Betriebssystem erstellen. Diese Einschränkung trifft auf alle Frameworks zu (React Native, o.J. u und NativeScript, o.J. i)

5.5. Einfache Entwicklung

Das Framework Ionic verwendet HTML, CSS und TypeScript zur Erstellung von Anwendungen. HTML und CSS sind für Webentwickler bekannte Technologien. TypeScript an sich ist zwar eine neue Technologie, entspricht aber der ECMAScript 2015 Syntax, welche um optionale Sprachkonstrukte erweitert wird. Wegen der hohen Ähnlichkeit zu ES6 und damit zu JavaScript ist davon auszugehen, dass die Anwendung in TypeScript für einen Webentwickler mit JavaScript-Erfahrung ohne großen Lernaufwand realisierbar ist. Zwar erfordert Ionic ein gewisses Grundverständnis des Frameworks selbst und von der Funktionsweise von mobilen Betriebssystemen, es ist aber kein Wissen über native APIs oder tiefergehendes Wissen über die Entwicklung von nativen Anwendungen notwendig. Plattformspezifisches Wissen, wie es beispielsweise zur Konfiguration von Berechtigungen notwendig ist, findet sich in der Dokumentation des Frameworks. Zur Entwicklung von Anwendungen mit dem Framework Ionic ist daher, vor allem im Vergleich zu den anderen Frameworks, mit einem geringen Maß an framework- und plattformspezifischem Wissen möglich.

Es ist möglich, Dritt-Bibliotheken in das Framework einzubinden. Der Prototyp nutzt die JavaScript-Bibliothek Chart.js zur Darstellung von statistischen Diagrammen.

5.6. Wartbarkeit

Zunächst wird das Framework Ionic selbst hinsichtlich seiner Eignung zur Erstellung von wartbarer Software betrachtet. Danach wird TypeScript im Hinblick auf die im zweiten Kapitel diskutierten Schwächen von JavaScript analysiert.

5.6.1. Ionic

Ionic baut auf dem JavaScript-Framework Angular2 auf und entspricht in Struktur und Konzepten diesem in weiteren Teilen. Ionic ist daher ebenso wie Angular2 nach dem MVC-Architekturmuster strukturiert. Es stehen mehrere Konzepte zur modularen Strukturierung und zur Wiederverwendung von Quellcode zur Verfügung: Components, Directives und Services (Angular o.J. a).

Components sind modulare Bausteine, die aus einem Template und Programmcode bestehen. Sie entsprechen Controller und View im MVC-Muster. Im Folgenden wird ein minimalistisches Beispiel für einen Component aufgezeigt:

```
// example.ts
import {Component} from '@angular/core';

@Component({
 selector: 'example-view',
 templateUrl: 'example.html'
})
export class Example {
 constructor() {
// ...
}
 someMethod() {
    // ...
 }
}

// example.html
<ion-content>
   <p>Example template</p>
</ion-content>
```

Dieser Component zeigt nur eine Zeile Text an. Über *import* wird das Modul *Component* aus dem Paket *@angular/core* importiert. Bei *ion-content* handelt es sich um eine Directive von Ionic, die für die Darstellung des enthaltenen p-Tags in der plattformspezifischen Darstellung

sorgt (Ionic, o.J. m). Der Component *Example* kann entweder über Ionics Navigationssystem als einzelne Seite angezeigt werden, oder über die Verwendung des Selektors *example-view* in einem anderen Template wiederverwendet werden:

```
//example2.html
<example-view></example-view>
```

An der Stelle des Selektors wird das Template des Components Example eingebunden. Dies ermöglicht die Wiederverwendung von HTML-, CSS und TypeScript-Code, da auf diese Weise komplexer Code aus Gründen der Übersichtlichkeit abstrahiert und redundanter Code wiederverwendet werden kann.

Directives dienen der Wiederverwendung von häufig genutzten Aufgaben. Angular 2 bietet eine Reihe von vorgegebenen Directives an. Daneben lassen sich eigene Directives implementieren:

```
import {Directive, ElementRef} from '@angular/core';

@Directive({selector: '[yellow]'})
export class HighlightDirective {
   constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
   }
}
```

Diese Directive wird als HTML-Attribut wie folgt verwendet:

```
<p yellow>Example template</p>
```

Das Attribut *yellow* sorgt in diesem Fall dafür, dass die Hintergrundfarbe des p-Elements auf die Farbe Gelb gesetzt wird. (Angular, o.J. c)

Services dienen zur Wiederverwendung von reinem Anwendungscode, sie entsprechen dem Model im MVC-Muster.

Der Prototyp verwendet einen Service, um den Zugriff auf lokal gespeicherte Daten zu abstrahieren:

```
import {Injectable} from '@angular/core';

@Injectable()
export class StorageService {
   constructor() {
   }
   saveData() {
       // … (Code zum Speichern der Daten auf der Zielplattform)
   }
}
```

Der Service kann innerhalb eines Components verwendet werden:

```
import {Component} from '@angular/core';
import {StorageService} from "../../../providers/storage-service";

@Component({
   selector: 'example-view',
   templateUrl: 'example.html'
})
export class Example {
   constructor(private storage: StorageService) {
   }
   saveDataToDevice(data) {
       this.storage.saveData(data);
   }
}
```

5.6.2. TypeScript

TypeScript ermöglicht die Nutzung von Funktionen aus ECMAScript 2015 und erweitert die Syntax um weitere Sprachkonstrukte zur statischen Typisierung und Kapselung von Informationen. Der TypeScript-Compiler (TSC) führt bei der Übersetzung zu JavaScript eine Reihe von Prüfungen durch, wodurch viele Fehler im Code bereits zum Zeitpunkt der Übersetzung erkannt und dem Entwickler gemeldet werden. Diese Fehlermeldungen werden von der Ionic-CLI während der Entwicklung ausgegeben. Zudem unterbricht die Ionic-CLI die Übersetzung der Anwendung, sodass ein Weiterarbeiten nicht möglich ist, ohne den Fehler zu beheben.

6.2.2. Fehlerbehandlung

TypeScript verhindert die implizite Erstellung von Variablen.

```
var name = "value";
naem = "new value"; // TSC: Cannot find name 'naem'.
```

Wird auf eine bisher nicht deklarierte Variable verwiesen, gibt der TypeScript-Compiler eine Fehlermeldung aus. Ein Zugriff auf nicht existierende Eigenschaften wird ebenfalls verhindert:

```
var a = "value";
a.property; // TSC: Property 'property' does not exist on type 'string'.
```

Unerwartete Rückgabewerte durch die automatische Code-Vervollständigung von JavaScript werden verhindert, indem auf nicht erreichbaren Code hingewiesen wird:

```
function func() {
  return
  { obj: "obj" } // TSC: Unreachable code detected.
};
```

6.2.1. Automatische Typkonvertierungen

TypeScript ergänzt JavaScript um eine optionale, statische Typisierung. Die statische Typüberprüfungen werden nur zum Zeitpunkt der Übersetzung ausgeführt und werden nicht in den generierten Code übernommen. (Microsoft, 2016, 25) Die Deklaration von Typen folgt folgendem Schema:

```
var a: string = "1";
var b: number = 1;
```

Der Quellcode muss nicht durchgängig mit Typdeklarationen versehen werden, es ist möglich nur einzelne Codefragmente explizit zu typisieren. Durch Typinferenz können Typen aus dem Inhalt von Variablen abgeleitet werden und müssen nicht zwingend explizit deklariert werden (Microsoft, 2016, 1 f) So wird bei der Evaluation von Bedingungen eine Fehlermeldung ausgegeben, wenn zwei verschiedene Typen miteinander verglichen werden, auch wenn diese nicht explizit deklariert wurden:

```
var a = "1";
var b = 1;
if (a === b) { // TSC: Operator '===' cannot be applied to types 'string' and 'number'.
  //...
}
```

TypeScript liefert Typdefinitionen für die JavaScript-API. Auf diese Weise werden in der JavaScript-Spezifikation vorgesehenen Funktionen und Objekte mit Typen versehen. (Microsoft, 2016, 3) Typdefinitionen für Drittbibliotheken lässt sich über die Paketverwaltung “npm” zusätzlich installieren (Rosenwasser, 2016). Ionic liefert die notwendigen Typdefinitionen mit (Ionic, o.J. m). So wird beispielsweise das in JavaScript enthaltene *Math*-

Objekt typisiert. Wird in JavaScript einer Methode von *Math* ein Parameter mit falschem Typ übergeben, so versucht JavaScript diesen automatisch in einen geeigneten Typ zu konvertieren:

```
//JavaScript
Math.ceil("1.2"); // liefert 2
```

TypeScript gibt an dieser Stelle eine Fehlermeldung aus:

```
// TypeScript
Math.ceil("1.2") // TSC: Argument of type 'string' is not assignable to parameter of type
                    'number'.
```

Auf diese Weise werden unerwartete Rückgabewerte verhindert, die bei der Übergabe eines nicht sinnvoll umwandelbaren Parameters entstehen und unter Umständen nicht entdeckt werden, beispielsweise:

```
// JavaScript
Math.ceil("foo") // liefert NaN
```

6.2.2. Modularität und der Geltungsbereich von Variablen

Das in ECMAScript 2015 eingeführte Keyword *let* wird ebenfalls in TypeScript unterstützt und ermöglicht die Erstellung von Variablen mit Block-Scope:

```
if (true) {
  let test = "noop";
}

test; // TSC: Error: test is not defined
```

TypeScript bietet ein Modulsystem, mit dem sich über die Konstrukte *import* und *export* Code aus einzelnen Dateien laden lässt (Microsoft, 2016, 16). Wird das Modulsystem verwendet, wird im generierten JavaScript-Code automatisch der Strict-Mode aktiviert, um die implizite Erstellung globaler Variablen zu verhindern:

```
// TypeScript
export class Class {
   constructor() {
       var foo = "foo";
   }
}
// generiertes JavaScript
"use strict";
var Class = (function () {
   function Class() {
       var foo = "foo";
   }
   return Class;
}());
exports.Class = Class;
```

5.6.2. Objektorientierung

TypeScript erweitert die Syntax von ECMAScript 2015 um Sprachkonstrukte zur Kapselung von Informationen. Variablen und Methoden einer Klasse können als privat markiert werden:

```
class Class {
 public visible: string;
 private hidden: string;

 constructor() {
   this.hidden = "private";
   this.visible = "public";
 }

 getHidden() {
   return this.hidden;
 }

 getVisible() {
   return this.visible;
 }
}

let obj = new Class();
obj.hidden; // TSC: Property 'hidden' is private and only accessible within class 'Class'.
obj.visible; // liefert "public"
obj.getHidden(); // liefert "private"
```

Beim Zugriff auf private Variablen oder Methoden liefert der TypeScript-Compiler eine Fehlermeldung. Neben den Konstrukten *public* und *private* existieren noch die Konstrukte *static* und *protected*. Eine als *static* gekennzeichnete Methode kann nicht von Instanzen der Klasse, sondern nur von der Klasse selbst verwendet werden. Eine als *protected* gekennzeichnete Methode kann nur von Instanzen der Klasse oder Instanzen von Sub-Klassen verwendet werden. Wird keine Kennzeichnung vorgenommen, wird die Methode oder Variable als *public* behandelt. (Microsoft, 2016, 120)

Wird beim obigen Beispiel das Keyword *new* vergessen, liefert der TypeScript-Compiler ebenfalls eine Fehlermeldung:

```
let obj = Class(); // TSC: Value of type 'typeof Class' is not callable. Did you mean to
                      include 'new'?
```

Wird der vom TypeScript-Compiler generierte JavaScript-Code betrachtet, lassen sich mehrere Punkte festhalten:

```
var Class = (function () {
   function Class() {
       this.hidden = "private";
       this.visible = "public";
   }
   Class.prototype.getHidden = function () {
```

```
        return this.hidden;
    };
    Class.prototype.getVisible = function () {
        return this.visible;
    };
    Class.prototype.getPrivate = function () {
        return "private method";
    };
    Class.getStatic = function () {
        return "static method";
    };
    return Class;
}());
```

Der generierte JavaScript-Code ist in einer IIFE gekapselt, um Namenskonflikte und die Erstellung von globalen Variablen zu vermeiden. Mit *private* gekennzeichnete Variablen werden im generierten Code nicht gekapselt, sondern sind frei zugänglich. Auch existiert keine programmatische Sicherstellung, dass das Keyword *new* verwendet wird. Dies verdeutlicht die Funktionsweise von TypeScript als "Syntactic Sugar". Die Funktionen von TypeScript zur Kapselung und statischen Typisierung existieren nur während der Entwicklung und nicht zur Laufzeit. Der generierte Code wird nicht um Code-Strukturen erweitert, die beispielsweise die Kapselung von Variablen auch zur Laufzeit sicherstellen. Eine Änderung des Codes zur Laufzeit ist allerdings auch nicht notwendig; die im zweiten Kapitel angesprochene Kommunikationswirkung von expliziten Sprachkonstrukten wird dennoch erreicht. Nach Meyer (1997, 52) zielt das Konzept der Kapselung nicht auf die Geheimhaltung von Codestrukturen ab, sondern auf die Etablierung einer verlässlichen Übereinkunft, auf welche Variablen und Methoden eines Objekts von außen zugegriffen werden darf.

TypeScript bietet ein weiteres Sprachkonstrukt, das *interface*. Ein Interface ermöglicht es Variablen und Methoden festzulegen, die eine Klasse implementieren muss.

Das folgende Interface beschreibt die Klasse aus dem ersten Beispiel dieses Kapitels:

```
interface Interface {
 public visible: string;
 private hidden: string;

 getHidden(): string;
 getVisible(): string;
}
```

Über das Keyword *implements* wird angegeben, dass eine Klasse ein bestimmtes Interface implementiert (Microsoft, 2016, 4-6). Implementiert die Klasse die vom Interface vorgegebenen Eigenschaften nicht korrekt, liefert der TypeScript-Compiler eine Fehlermeldung:

```
class Class implements Interface {
 visible: string;     // TSC: Class 'Class' incorrectly implements interface 'Interface'.
                         Types have separate declarations of a private property 'hidden'.
 private hidden: string;

 constructor() {
   this.hidden = "private";
   this.visible = "public";
 }

 getHidden() {
   return this.hidden;
 }

 getVisible() {
   return this.visible;
 }
}
```

Durch das Sprachkonstrukt Interface lassen sich verbindliche Anforderungen an Klassen und andere Typen definieren, die vom Entwickler eingehalten werden müssen. Interfaces existieren nur im TypeScript-Code und werden nicht in den generierten JavaScript-Code übertragen. Im Prototyp existiert ein Interface namens *Product*, das die Eigenschaften eines einzelnen Einkaufs beschreibt:

```
export interface Product {
 productName: string,
 price: number,
 category: string,
 date: Date,
 shopAdress: any,
 shopName: any,
 latitude: any,
 longitude: any
}
```

Dieses Interface wird genutzt, um sicherzustellen, dass alle Objekte vom Typ *Product* über die festgelegten Eigenschaften verfügen. So erwartet beispielsweise der Service zur Speicherung der Einkäufe auf dem Endgerät ein Objekt vom Typ *Product* als zweiten Parameter:

```
export class StorageService {
//…
   pushItem(key: string, obj: Product) {
//…
   }
//…
}
```

Erkennt der TypeScript-Compiler, dass ein Objekt mit falschem Typ als Parameter übergeben wird, wird eine Fehlermeldung ausgegeben. Auf diese Weise wird sichergestellt, dass der Methode *pushItem* nur Objekte übergeben werden, mit denen diese Methode umgehen kann. Beim Interface handelt es sich demnach ebenfalls um eine Form der Kommunikation, wie einzelne Code-Fragmente funktionieren und verwendet werden sollen. Dieser Punkt zeigt gleichzeitig die Einschränkungen von TypeScript auf:

```
export class CategoriesModal implements OnInit {
   //...
   addCategory(category) {
       this.storage.pushItem('categories', category).then(() => { // keine Fehlermeldung
           //...
       });
   }
   //...
}
```

Der Methode *addCategory* wird eine Variable *category* übergeben, bei der es nicht um ein *Product*, sondern um einen String handelt. Da TypeScript den Wert des Parameters nicht kennt, wird dieser als Parameter für die Methode *pushItem* ohne Fehlermeldung akzeptiert, obwohl der Typ *Product* als Parameter gefordert ist. Typinferenz ist an dieser Stelle nicht möglich, da die Methode *addCategory* aus dem HTML-Code aufgerufen wird, welcher nicht in die Typprüfungen von TypeScript einbezogen wird.

Nur wenn der Typ des an *addCategory* übergebenen Parameters explizit angegeben wird, wird eine Fehlermeldung ausgegeben:

```
export class CategoriesModal implements OnInit {
   //...
   addCategory(category: string) {
       this.storage.pushItem('categories', category)
       .then(() => { // TSC: Argument of type 'string' is not assignable to parameter
                     of type 'Product'.
           //...
       });
   }
   //...
}
```

Dieser Umstand ist darin begründet, dass die statische Typprüfungen von TypeScript nur für den Anwendungscode selbst ausgeführt werden und nicht zur Laufzeit existieren. Eine durchgängige, Typisierung des Quellcodes wirkt dem beschriebenen Problem entgegen. Die Konfigurationsoption *noImplicitAny* erzwingt die explizite Typisierung von allen Variablen (TypeScript, o.J. b). In Ionic ist diese Option allerdings standardmäßig nicht aktiviert.

5.7. Zusammenfassung

Das Framework Ionic unterstützt die Erstellung von Web-Anwendungen und Hybrid-Apps aus einer Codebasis. Der Quellcode kann dabei nahezu vollständig für alle Zielplattformen ausgenommen werden. Ausgenommen sind Funktionen, die auf der Zielplattform nicht unterstützt werden.

Die von Ionic gebotenen UI-Komponenten könnten eine bessere Unterstützung für die Darstellung auf Geräten mit hohen Bildschirmauflösungen bieten. Zur Umsetzung einer dauerhaft geöffneten Navigationsleiste ab einer bestimmten Bildschirmgröße war zusätzlicher Implementierungsaufwand notwendig. Zudem sind einige Funktionen, wie die Anpassung des Spalten-Layouts an die jeweilige Bildschirmgröße, nicht in der offiziellen Dokumentation des Frameworks aufgeführt. Insgesamt lässt sich festhalten, dass der Code für das Benutzerinterface für alle Zielplattformen ohne redundanten Code wiederverwendet werden kann und Ionic die Möglichkeit bietet, das Benutzerinterface in Einzelheiten an die Zielplattform anzupassen.

Da JavaScript-Code gültiger TypeScript-Code ist, erlaubt es das Framework Ionic Anwendungen komplett mit den Webtechnologien HTML, CSS und JavaScript zu entwickeln. Für die Nutzung des Frameworks ist daher ein vergleichsweise geringer Lernaufwand notwendig. Das Framework Ionic baut zudem auf dem bekannten MVC-Architekturmuster auf und bietet Strukturen und Technologien, welche die Wartbarkeit des Quellcodes verbessern. Der Entwicklungs- und Wartungsaufwand zur Erstellung von Hybrid-Apps und Web-Anwendungen lässt sich daher durch das Framework Ionic verringern.

6. Fazit und Ausblick

Die Erstellung von mobilen Anwendungen und Web-Anwendungen aus einer gemeinsamen Codebasis lässt sich mit dem Framework Ionic umsetzen. Die Umsetzung mit diesem Framework erlaubt im Vergleich mit den anderen Optionen den größten Grad der Wiederverwendung des Quellcodes für die drei Plattformen iOS, Android und den Webbrowser. Zur Implementierung wird ein einheitliches Set an Technologien verwendet, plattformspezifische Technologien, wie beispielsweise XML-Markup werden nicht benötigt, die Codebasis erreicht daher den größtmöglichen Grad an Einheitlichkeit.

Durch die Wiederverwendung des Quellcodes für mobile Anwendungen und Web-Anwendungen fällt insgesamt ein geringerer Entwicklungs- und Wartungsaufwand an. Das Framework Ionic bietet zudem eine Reihe von Konzepten zur Strukturierung des Quellcodes, welche sich über die verbesserte Modularität und Wiederverwendbarkeit positiv auf die Wartbarkeit der Codebasis auswirken. Neben diesen Konzepten gibt auch das MVC-Architekturmuster eine bestimmte Struktur für den Quellcode vor. Dadurch müssen von Entwicklerseite keine eigenen Konzepte zur Strukturierung des Quellcodes entwickelt und eingehalten werden, sondern es wird eine bestimmte Anwendungsarchitektur vorgegeben.

Durch die Verwendung von ECMAScript 2015 mittels Transpiler lassen sich viele Probleme der Programmiersprache JavaScript beheben und die Syntax unter anderem um Sprachkonstrukte für klassenbasierte Objektorientierung erweitern. Dies verringert den Wartungsaufwand von JavaScript-Anwendungen und ermöglicht Entwicklern einen leichteren Zugang zur objektorientierten Entwicklung mit JavaScript und damit eine bessere Modularität und Wiederverwendbarkeit der Codebasis.

TypeScript verbessert die Wartbarkeit von JavaScript zusätzlich, indem dieses um eine statische Typisierung ergänzt wird. Ein zusätzlicher Kompilierschritt ermöglicht die Prüfung des Quellcodes, um den Entwickler im Umgang mit fehlerträchtigen Eigenschaften der Programmiersprache zu unterstützen, beispielsweise der permissiven Fehlerbehandlung oder automatischen Typkonvertierungen. Dies verbessert in erster Linie die Analysierbarkeit und dadurch auch die Änderbarkeit des Quellcodes. Statische Typprüfungen machen auch die Verwendung von weiteren Sprachkonstrukten, wie Interfaces möglich und prüfen die Einhaltung von Zugriffsbeschränkungen, wie *private*, *public* oder *protected*. Dies verbessert

ebenfalls die Analysierbarkeit und Modifizierbarkeit und trägt zudem zu einer modularen Strukturierung des Quellcodes bei.

Die Technologien Ionic und TypeScript erfüllen damit die zu Beginn der Arbeit formulierte Zielsetzung, vor dem Hintergrund eines verringerten Entwicklungs- und Wartungsaufwandes mobile Anwendungen und Web-Anwendungen aus einer gemeinsamen Codebasis zu erstellen und die JavaScript-Syntax um Sprachkonstrukte einer klassenbasierten Objektorientierung zu erweitern.

Obwohl sie aus der Wartbarkeitsperspektive dieser Arbeit nicht geeignet sind, bieten die Runtime-basierten Ansätze, wie sie von NativeScript oder React Native verfolgt werden über die Trennung des Benutzerinterfaces eine höhere Anpassung an die jeweilige Zielplattform. Welche Implikationen dieses Konzept in Bezug auf Wartbarkeit mit sich bringt kann in weiteren Arbeiten untersucht werden. Angesichts der Unzulänglichkeiten von JavaScript in Bezug auf Wartbarkeit ist eine detaillierte Analyse der Wartbarkeit von JavaScript zwar relevant, jedoch muss ein Entscheidungsprozess zur Auswahl von geeigneten Technologien immer mehrere Perspektiven berücksichtigen. Mit der weiteren Verbreitung von ECMAScript 2015 und TypeScript nimmt die Relevanz der Wartungsperspektive zugunsten anderer Perspektiven, wie der Benutzerfreundlichkeit oder Anwendungsperformance ab.

Durch die Positionierung von Ionic als Framework zur Erstellung von mobilen und Web-Anwendungen und die Ankündigung von NativeScript, in Zukunft weitere Möglichkeiten zur Wiederverwendung von Code für mobile Anwendungen und Web-Anwendungen zu bieten (NativeScript, o.J. r), zeichnet sich insgesamt eine Entwicklung ab, in der die Erstellung von Anwendungen für verschiedene Plattformen aus gemeinsamer Codebasis zunehmend vorangetrieben wird.

Literaturverzeichnis

Agius, Alan (2016): „feat(grid): responsive grid columns widths by alan-agius4 · Pull Request #7714 · driftyco/ionic“. GitHub. Abgerufen am 23.01.2017 von https://github.com/driftyco/ionic/pull/7714.

Android (o.J. a): „Sharing Simple Data | Android Developers“. Abgerufen am 20.01.2017 von https://developer.android.com/training/sharing/index.html.

Android (o.J. b): „Activities | Android Developers“. Abgerufen am 20.01.2017 von https://developer.android.com/guide/components/activities/index.html.

Angular (o.J. a): „Architecture Overview - ts - GUIDE“. Architecture Overview - ts - GUIDE. Abgerufen am 12.01.2017 von https://angular.io/docs/ts/latest/guide/architecture.html.

Angular (o.J. b): „Testing - ts - GUIDE“. Testing - ts - GUIDE. Abgerufen am 12.01.2017 von https://angular.io/docs/ts/latest/guide/testing.html.

Angular (o.J. c): „Angular“. Abgerufen am 20.01.2017 von https://angular.io/docs/ts/latest/guide/attribute-directives.html.

Appcelerator (2016): „GA Releases for 6.0.0 of SDK, CLI & 4.8.0 of Studio“. Appcelerator. Abgerufen am 20.01.2017 von http://www.appcelerator.com/blog/2016/11/ga-releases-for-6-0-0-of-sdk-cli-4-8-0-of-studio/.

Appcelerator (o.J. a): „Platform Plans and Pricing“. Appcelerator. Abgerufen am 20.01.2017 von http://www.appcelerator.com/pricing/.

Appcelerator (o.J. b): „Appcelerator Developer“. Abgerufen am 20.01.2017 von https://developer.appcelerator.com/.

Appcelerator (o.J. c): „Location Services - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/Location_Services.

Appcelerator (o.J. d): „Working with Local Data Sources - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/Working_with_Local_Data_Sources.

Appcelerator (o.J. e): „Android Intents - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/Android_Intents.

Appcelerator (o.J. f): „OptionDialog - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/OptionDialog.

Appcelerator (o.J. g): „LiveView - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/LiveView.

Appcelerator (o.J. h): „Deploying to Android devices - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/Deploying_to_Android_devices-section-src-29004923_DeployingtoAndroiddevices-DeploytheapplicationusingStudio.

Appcelerator (o.J. i): „Contents - Appcelerator Platform - Appcelerator Docs“. Abgerufen am 20.01.2017 von https://docs.appcelerator.com/platform/latest/#!/guide/About_Content_Assist.

Apple (o.J. a): „UIActivityViewController - UIKit | Apple Developer Documentation“. Abgerufen am 20.01.2017 von https://developer.apple.com/reference/uikit/uiactivityviewcontroller.

Apple (o.J. b): „The App Life Cycle“. Abgerufen am 20.01.2017 von https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html.

Archibald, Jake (o.J.): „Is ServiceWorker ready?“. Abgerufen am 15.09.2016 von https://jakearchibald.github.io/isserviceworkerready/.

Ashkenas, Jeremy (2016): „jashkenas/coffeescript“. GitHub. Abgerufen am 28.11.2016 von https://github.com/jashkenas/coffeescript.

Atanasov, Georgi (2014): „NativeScript – a Technical Overview“. Telerik Developer Network. Abgerufen am 12.01.2017 von http://developer.telerik.com/featured/nativescript-a-technical-overview/.

Babel (o.J.): „Babel · The compiler for writing next generation JavaScript“. Abgerufen am 20.01.2017 von https://babeljs.io/.

Balzert, Helmut (2009): Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements-Engineering. 3. Aufl. Heidelberg: Spektrum, Akad. Verl (Lehrbücher der Informatik). — ISBN: 978-3-8274-1705-3

Balzert, Helmut; Ebert, Christof (2008): Softwaremanagement. 2. Aufl. Heidelberg: Spektrum, Akad. Verl (Lehrbuch der Softwaretechnik). — ISBN: 978-3-8274-1161-7

Balzert, Helmut; Liggesmeyer, Peter (2011): Lehrbuch der Softwaretechnik. [2]: Entwurf, Implementierung, Installation und Betrieb. 3. Aufl. Heidelberg: Spektrum, Akad. Verl (Lehrbücher der Informatik). — ISBN: 978-3-8274-1706-0

Bevacqua, Nicolás (2015): „JavaScript Developer Survey Results“. Pony Foo. Abgerufen am 28.11.2016 von https://ponyfoo.com/articles/javascript-developer-survey-results.

Boggan, Pierce (2016): „Introducing Xamarin.Forms 2.3.3: Native View Declaration and Platform Specifics |“. Xamarin Blog.

Brauer, Johannes (2014): Grundkurs Smalltalk - Objektorientierung von Anfang an: eine Einführung in die Programmierung. 4., und überarb. Aufl. Wiesbaden: Springer Vieweg. — ISBN: 978-3-658-00630-3

Chart.js (o.J.): „Chart.js | Documentation“. Abgerufen am 12.01.2017 von http://www.chartjs.org/docs/#notes-browser-support.

CoffeeScript (o.J.): „CoffeeScript“. Abgerufen am 28.11.2016 von http://coffeescript.org/.

Crockford, Douglas (2001): „Private Members in JavaScript“. Abgerufen am 18.11.2016 von http://www.crockford.com/javascript/private.html.

Crockford, Douglas (2008): JavaScript: the good parts ; [unearthing the excellence in JavaScript]. 1. ed. Beijing: O'Reilly. — ISBN: 978-0-596-51774-8

Dart (o.J.): „dart2js: The Dart-to-JavaScript Compiler“. Abgerufen am 28.11.2016 von https://webdev.dartlang.org/tools/dart2js.

Dascalescu, Dan (2016): „google chrome - What features do Progressive Web Apps have vs. native apps and vice-versa, on Android - Stack Overflow“. Abgerufen am 15.09.2016 von http://stackoverflow.com/questions/35504194/what-features-do-progressive-web-apps-have-vs-native-apps-and-vice-versa-on-an/39027789.

Ducasse, Stéphane; Petton, Nicolas; Polito, Guillermo; u. a. (2012): „Semantics and Security Issues in JavaScript“. In: arXiv:1212.2341 [cs].

Flanagan, David (2011): JavaScript: the definitive guide. 6th ed. Beijing ; Sebastopol, CA: O'Reilly. — ISBN: 978-0-596-80552-4

Flutter (o.J. a): „flutter/LICENSE at master · flutter/flutter“. GitHub. Abgerufen am 20.01.2017 von https://github.com/flutter/flutter/blob/master/LICENSE.

Flutter (o.J. b): „Welcome to Flutter! - Flutter“. Abgerufen am 20.01.2017 von https://flutter.io/.

Flutter (o.J. c): „Can I access native services and APIs like sensors and local storage?“. Abgerufen am 20.01.2017 von https://flutter.io/faq/#can-i-access-native-services-and-apis-like-sensors-and-local-storage.

Flutter (o.J. d): „Flutter FAQ - Flutter“. Abgerufen am 20.01.2017 von https://flutter.io/faq/#does-flutter-come-with-widgets.

Flutter (o.J. e): „What language is Flutter written in?“. Abgerufen am 20.01.2017 von https://flutter.io/faq/#what-language-is-flutter-written-in.

Flutter (o.J. f): „Does Flutter run on the web?“. Abgerufen am 20.01.2017 von https://flutter.io/faq/#does-flutter-run-on-the-web.

Flutter (o.J. g): „Viewing source code changes with 'hot reload'“. Abgerufen am 20.01.2017 von https://flutter.io/getting-started/#viewing-source-code-changes-with-hot-reload.

Flutter (o.J. h): „Running your Flutter app“. Abgerufen am 20.01.2017 von https://flutter.io/getting-started/#running-your-flutter-app.

Flutter (o.J. i): „Developing apps in the IntelliJ IDE - Flutter“. Abgerufen am 20.01.2017 von https://flutter.io/intellij-ide/#editing-code-and-viewing-code-problems.

Gallagher, Nicolas (o.J. a): „React Native for Web“. GitHub. Abgerufen am 12.01.2017 von https://github.com/necolas/react-native-web.

Gallagher, Nicolas (o.J. b): „Known issues“. react-native-web/known-issues.md at master · necolas/react-native-web. Abgerufen am 12.01.2017 von https://github.com/necolas/react-native-web/blob/master/docs/guides/known-issues.md.

Gama, W.; Alalfi, M. H.; Cordy, J. R.; u. a. (2012): „Normalizing object-oriented class styles in JavaScript“. In: 2012 14th IEEE International Symposium on Web Systems Evolution (WSE)., DOI: 10.1109/WSE.2012.6320536.

Google (o.J. a): „Progressive Web Apps“. Google Developers. Abgerufen am 16.01.2017 von https://developers.google.com/web/progressive-web-apps/.

Google (o.J. b): „Google Places API Web Service | Google Places API Web Service“. Google Developers. Abgerufen am 23.01.2017 von https://developers.google.com/places/web-service/intro?hl=de.

Gumm, Heinz Peter; Sommer, Manfred; Hesse, Wolfgang (2013): Einführung in die Informatik. 10., vollst. überarb. Aufl. München: Oldenbourg. — ISBN: 978-3-486-70641-3

Hanselman, Scott (2013): „JavaScript is Web Assembly Language and that’s OK.“. Abgerufen am 28.11.2016 von http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx.

Heitkötter, Henning; Hanschke, Sebastian; Majchrzak, Tim A. (2012): „Evaluating Cross-Platform Development Approaches for Mobile Applications“. In: Cordeiro, José; Krempels, Karl-Heinz (Hrsg.) Web Information Systems and Technologies. Springer Berlin Heidelberg (Lecture Notes in Business Information Processing), S. 120–138. — ISBN: 978-3-642-36607-9

Hunt, Pete (2013): „Why did we build React? - React Blog“. Abgerufen am 12.01.2017 von https://facebook.github.io/react/blog/2013/06/05/why-react.html.

Ionic (o.J. a): „ionic/LICENSE at master · driftyco/ionic“. GitHub. Abgerufen am 20.01.2017 von https://github.com/driftyco/ionic/blob/master/LICENSE.

Ionic (o.J. b): „Social Sharing“. Ionic Native: Harness the power of Native APIs. Abgerufen am 20.01.2017 von https://ionicframework.com/docs/v2/native/social-sharing/.

Ionic (o.J. c): „Segment“. Ionic Framework. Abgerufen am 20.01.2017 von https://ionicframework.com/docs/v2/components/.

Ionic (o.J. d): „Release 2.0.0-rc.5 · driftyco/ionic“. GitHub. Abgerufen am 20.01.2017 von https://github.com/driftyco/ionic/releases/tag/v2.0.0-rc.5.

Ionic (o.J. e): „Build Amazing Native Apps and Progressive Web Apps with Ionic Framework and Angular“. Build Amazing Native Apps and Progressive Web Apps with Ionic Framework and Angular. Abgerufen am 12.01.2017 von https://ionicframework.com/.

Ionic (o.J. f): „Platform - Ionic API Documentation - Ionic Framework“. Platform - Ionic API Documentation - Ionic Framework. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/api/platform/Platform/.

Ionic (o.J. g): „Ionic CLI Documentation“. Ionic CLI Documentation. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/cli/.

Ionic (o.J. h): „Ionic Component Documentation“. Ionic Component Documentation. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/components/.

Ionic (o.J. i): „Ionic Native: Harness the power of Native APIs“. Ionic Native: Harness the power of Native APIs. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/native/geolocation/.

Ionic (o.J. j): „Benefits of TypeScript“. Benefits of TypeScript. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/resources/typescript/.

Ionic (o.J. k): „Deploying“. Deploying. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/setup/deploying/.

Ionic (o.J. l): „Installing Ionic“. Installing Ionic. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/setup/installation/.

Ionic (o.J. m): „Content - Ionic API Documentation - Ionic Framework“. Content - Ionic API Documentation - Ionic Framework. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/api/components/content/Content/.

Ionic (o.J. n): „Run“. Run. Abgerufen am 12.01.2017 von https://ionicframework.com/docs/v2/cli/run/.

Ionic (o.J. o): „Storage - Ionic Framework“. Storage - Ionic Framework. Abgerufen am 13.01.2017 von https://ionicframework.com/docs/v2/storage/.

Ionic (o.J. p): „Ionic Documentation“. Ionic Documentation. Abgerufen am 12.01.2017a von https://ionicframework.com/docs/.

Ionic (o.J. p): „Platform Specific Styles“. Ionic Framework. Abgerufen am 20.01.2017b von https://ionicframework.com/docs/v2/theming/platform-specific-styles/.

Ionic (o.J. q): „Ionic Component Documentation“. Ionic Framework. Abgerufen am 23.01.2017 von https://ionicframework.com/docs/v2/components/#grid.

ISO/IEC (2006): „Software Engineering -- Software Life Cycle Processes -- Maintenance“. ISO. Abgerufen am 18.11.2016 von https://www.iso.org/obp/ui/#!iso:std:39064:en.

ISO/IEC (2011): „Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models“. ISO. Abgerufen am 18.11.2016.

Keist, Nikolai-Kevin; Benisch, Sebastian; Müller, Christian (2016): „Möglichkeiten und Grenzen der plattformübergreifenden App-Entwicklung“. In: Barton, Thomas; Müller, Christian; Seel, Christian (Hrsg.) Mobile Anwendungen in Unternehmen. Springer Fachmedien Wiesbaden (Angewandte Wirtschaftsinformatik), S. 109–119. — ISBN: 978-3-658-12009-2

Lahres, Bernhard; Raýman, Gregor (2006a): „Rheinwerk Computing :: Praxisbuch Objektorientierung – 1 Einleitung“. Abgerufen am 12.01.2017 von http://openbook.rheinwerk-verlag.de/oo/oo_01_einleitung_000.htm#Rxxob01einleitung0000400145b1f03712f.

Lahres, Bernhard; Raýman, Gregor (2006b): „Rheinwerk Computing :: Praxisbuch Objektorientierung – 2.2 Die Kapselung von Daten“. Abgerufen am 12.01.2017 von http://openbook.rheinwerk-verlag.de/oo/oo_02_basisderobjektorientierung_001.htm#Rxxob02basisderobjektorientierung001040014a91f03e130.

Lahres, Bernhard; Raýman, Gregor (2006c): „Rheinwerk Computing :: Praxisbuch Objektorientierung – 3 Die Prinzipien des objektorientierten Entwurfs“. Abgerufen am 12.01.2017 von http://openbook.rheinwerk-verlag.de/oo/oo_03_prinzipien_000.htm#Rxxob03prinzipien000040014c11f014137.

Marfatiya, Shoaib (2016): „Should Businesses Run After Progressive Mobile Web App Development“.

Martin, Brad (2016): „Using Native Libraries in NativeScript“. Telerik Developer Network. Abgerufen am 12.01.2017 von http://developer.telerik.com/featured/using-native-libraries-in-nativescript/.

Meyer, Bertrand (1997): Object-oriented software construction. 2nd ed. Upper Saddle River, N.J: Prentice Hall PTR. — ISBN: 978-0-13-629155-8

Microsoft (2016): „TypeScript Language Specification Version 1.8“.

Mikkonen, Tommi; Taivalsaari, Antero (2007): Using JavaScript As a Real Programming Language. Mountain View, CA, USA: Sun Microsystems, Inc.

Morony, Josh (2016): „How to Unit Test an Ionic 2 Application“. joshmorony - Build Mobile Apps with HTML5. Abgerufen am 12.01.2017 von http://www.joshmorony.com/how-to-unit-test-an-ionic-2-application/.

NativeScript (o.J. a): „NativeScript/LICENSE.md at master · NativeScript/NativeScript“. GitHub. Abgerufen am 20.01.2017 von https://github.com/NativeScript/NativeScript/blob/master/LICENSE.md.

NativeScript (o.J. b): „Location“. Abgerufen am 20.01.2017 von https://docs.nativescript.org/hardware/location.
NativeScript (o.J. c): „application-settings“. Abgerufen am 20.01.2017 von https://docs.nativescript.org/cookbook/application-settings.
NativeScript (o.J. d): „Chapter 5—Plugins and npm Modules“. Abgerufen am 20.01.2017 von https://docs.nativescript.org/tutorial/chapter-5#52-using-nativescript-plugins.
NativeScript (o.J. e): „Components“. Abgerufen am 20.01.2017 von https://docs.nativescript.org/ui/components#segmentedbar.
NativeScript (o.J. f): „Application Lifecycle“. Abgerufen am 20.01.2017 von https://docs.nativescript.org/core-concepts/application-lifecycle.
NativeScript (o.J. g): „Transpilers“. Abgerufen am 20.01.2017 von http://docs.nativescript.org/angular/tooling/transpilers.
NativeScript (o.J. h): „Chapter 1—Getting Up and Running“. Abgerufen am 20.01.2017 von http://docs.nativescript.org/angular/tutorial/ng-chapter-1#15-development-workflow.
NativeScript (o.J. i): „Chapter 1—Getting Up and Running“. Abgerufen am 20.01.2017 von http://docs.nativescript.org/angular/tutorial/ng-chapter-1#14-running-your-app.
NativeScript (o.J. k): „NativeScript/NativeScript“. GitHub. Abgerufen am 20.01.2017a von https://github.com/NativeScript/NativeScript.
NativeScript (o.J. k): „Platform“. Abgerufen am 12.01.2017b von http://docs.nativescript.org/angular/code-samples/platform.
NativeScript (o.J. l): „Unit Testing“. Abgerufen am 12.01.2017 von http://docs.nativescript.org/angular/tooling/testing.
NativeScript (o.J. m): „Chapter 1—Getting Up and Running“. Abgerufen am 12.01.2017 von http://docs.nativescript.org/angular/tutorial/ng-chapter-1#11-install-nativescript-and-configure-your-environment.
NativeScript (o.J. n): „Chapter 6—Accessing Native APIs“. Abgerufen am 12.01.2017 von http://docs.nativescript.org/angular/tutorial/ng-chapter-6#62-accessing-android-apis.
NativeScript (o.J. o): „Styling“. Abgerufen am 12.01.2017 von http://docs.nativescript.org/ui/styling.
NativeScript (o.J. p): „Welcome“. Abgerufen am 12.01.2017 von https://docs.nativescript.org/.
NativeScript (o.J. q): „NativeScript“. NativeScript.org. Abgerufen am 12.01.2017 von https://www.nativescript.org/.
NativeScript (o.J. r): „About NativeScript Open Source Cross Platform Framework“. NativeScript.org. Abgerufen am 12.01.2017 von https://www.nativescript.org/about.
NativeScript (o.J. s): „2.3: Adding UI elements“. Abgerufen am 22.01.2017a von http://docs.nativescript.org/angular/tutorial/ng-chapter-2#23-adding-ui-elements.

NativeScript (o.J. s): „NativeScript Advanced Setup—Windows“. Abgerufen am 12.01.2017b von http://docs.nativescript.org/angular/start/ns-setup-win.

Nitze, André (2015): „Evaluation of JavaScript Quality Issues and Solutions for Enterprise Application Development“. In: Winkler, Dietmar; Biffl, Stefan; Bergsmann, Johannes (Hrsg.) Software Quality. Software and Systems Quality in Distributed and Mobile Environments. Springer International Publishing (Lecture Notes in Business Information Processing), S. 108–119. — ISBN: 978-3-319-13250-1

Open Source Initiative (o.J.): „Licenses by Name | Open Source Initiative“. Abgerufen am 21.01.2017 von https://opensource.org/licenses/alphabetical.

Rantanen, Jarno (2015): „Isolation Mechanisms for Web Frontend Application Architectures“. In.:

React Native (o.J. a): „react-native/LICENSE at master · facebook/react-native“. GitHub. Abgerufen am 20.01.2017 von https://github.com/facebook/react-native/blob/master/LICENSE.

React Native (o.J. b): „Geolocation“. React Native. Abgerufen am 20.01.2017 von https://facebook.github.io/react-native/index.html.

React Native (o.J. c): „AsyncStorage“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/asyncstorage.html.

React Native (o.J. d): „Share“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/share.html.

React Native (o.J. e): „Button“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/button.html.

React Native (o.J. f): „AppState“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/appstate.html.

React Native (o.J. g): „JavaScript Environment“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/javascript-environment.html#javascript-syntax-transformers.

React Native (o.J. h): „Debugging“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/debugging.html.

React Native (o.J. i): „Running On Device“. React Native. Abgerufen am 20.01.2017 von http://facebook.github.io/react-native/releases/next/docs/running-on-device.html.

React Native (o.J. j): „Release December 2016 · facebook/react-native“. GitHub. Abgerufen am 20.01.2017 von https://github.com/facebook/react-native.

React Native (o.J. k): „Getting Started“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/getting-started.html.

React Native (o.J. l): „Height and Width“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/height-and-width.html.

React Native (o.J. m): „JavaScript Environment“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/javascript-environment.html.

React Native (o.J. n): „Platform Specific Code“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/platform-specific-code.html#platform-module.

React Native (o.J. o): „Style“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/style.html.

React Native (o.J. p): „Testing“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/testing.html.

React Native (o.J. q): „Tutorial“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/docs/tutorial.html.

React Native (o.J. r): „Props“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/releases/0.40/docs/props.html#props.

React Native (o.J. s): „Using a ListView“. React Native. Abgerufen am 12.01.2017 von http://facebook.github.io/react-native/releases/next/docs/using-a-listview.html.

React Native (o.J. t): „Help“. React Native. Abgerufen am 12.01.2017 von https://facebook.github.io/react-native/support.html.

React Native (o.J. u): „facebook/react-native“. GitHub. Abgerufen am 12.01.2017 von https://github.com/facebook/react-native.

Riebisch, Matthias; Bode, Stephan (2009): „Software-Evolvability“. In: Informatik-Spektrum. 32 (4), S. 339–343, DOI: 10.1007/s00287-009-0349-2.

Ritterbach, Beate (2014): „Werttypen in objektorientierten Programmiersprachen“. Von-Melle-Park 3, 20146 Hamburg: Universität Hamburg.

Rosenwasser, Daniel (2016): „The Future of Declaration Files“. Abgerufen am 20.01.2017 von https://blogs.msdn.microsoft.com/typescript/2016/06/15/the-future-of-declaration-files/.

Schmid, Norbert (2013): „JavaScript Prototypen statt Klassen- Vererbung im Rahmen von JavaScript“. Mayflower Blog.

Smeets, Ruben; Aerts, Kris (2016): „Trends in Web Based Cross Platform Technologies“. In: International Journal of Computer Science and Mobile Computing. 5 (6), S. 190–199.

Stoychev, Valio (2016): „Sharing code between web and native apps with Angular 2 and NativeScript“. NativeScript.org. Abgerufen am 12.01.2017 von https://www.nativescript.org/blog/sharing-code-between-web-and-native-apps-with-angular-2-and-nativescript.

Szyperski, Clemens; Gruntz, Dominik; Murer, Stephan (2003): Component software: beyond object-oriented programming. 2nd ed. London: Addison-Wesley (Addison-Wesley component software series). — ISBN: 978-0-201-74572-6

Teßmer, Meik (2012): „Architekturbezogene Qualitätsmerkmale für die Softwarewartung : Entwurf eines Quellcode basierten Qualitätsmodells [überarb. Ausg.]“. In.:

TypeScript (o.J. a): „TypeScript - JavaScript that scales.“. Abgerufen am 28.11.2016 von https://www.typescriptlang.org/.

TypeScript (o.J. b): „Compiler Options · TypeScript“. Compiler Options · TypeScript. Abgerufen am 13.01.2017 von https://www.typescriptlang.org/docs/handbook/compiler-options.html.

Vallon, Benoit (o.J.): „benoitvallon/react-native-nw-react-calculator“. GitHub. Abgerufen am 12.01.2017 von https://github.com/benoitvallon/react-native-nw-react-calculator.

W3C (2015): „Service Workers“. Abgerufen am 16.01.2017 von https://www.w3.org/TR/service-workers/.

W3C (2017): „Web App Manifest“. Abgerufen am 16.01.2017 von https://www.w3.org/TR/appmanifest/.

w3schools.com (o.J.): „CSS Units“. Abgerufen am 12.01.2017 von http://www.w3schools.com/cssref/css_units.asp.

Walker, Nathan (o.J. a): „NathanWalker/angular-seed-advanced: An advanced Angular2 seed project with support for ngrx/store, ngrx/effects, ng2-translate, angulartics2, lodash, NativeScript (*native* mobile), Electron (Mac, Windows and Linux desktop) and more.“. NathanWalker/angular-seed-advanced: An advanced Angular2 seed project with support for ngrx/store, ngrx/effects, ng2-translate, angulartics2, lodash, NativeScript (*native* mobile), Electron (Mac, Windows and Linux desktop) and more. Abgerufen am 12.01.2017 von https://github.com/NathanWalker/angular-seed-advanced.

Walker, Nathan (o.J. b): „angular-seed-advanced/src/client/app/components/home at master · NathanWalker/angular-seed-advanced“. angular-seed-advanced/src/client/app/components/home at master · NathanWalker/angular-seed-advanced. Abgerufen am 12.01.2017 von https://github.com/NathanWalker/angular-seed-advanced/tree/master/src/client/app/components/home.

West, Matt (2014): „Exploring the JavaScript Device APIs“. Treehouse Blog. Abgerufen am 16.01.2017 von http://blog.teamtreehouse.com/exploring-javascript-device-apis.

Xamarin (o.J. a): „Store - Xamarin“. Abgerufen am 20.01.2017 von https://store.xamarin.com/.

Xamarin (o.J. b): „Mobile App Development & App Creation Software - Xamarin“. Abgerufen am 20.01.2017 von https://www.xamarin.com/.

Xamarin (o.J. c): „Mobile Application Development to Build Apps in C# - Xamarin“. Abgerufen am 20.01.2017 von https://www.xamarin.com/platform.

Xamarin (o.J. d): „Activity Lifecycle - Xamarin“. Abgerufen am 20.01.2017 von https://developer.xamarin.com/guides/android/application_fundamentals/activity_lifecycle/.

Xamarin (o.J. e): „Part 2 - Application Lifecycle Demo - Xamarin“. Abgerufen am 20.01.2017 von https://developer.xamarin.com/guides/ios/application_fundamentals/backgrounding/part_2_application_lifecycle_demo/.

Xamarin (o.J. f): „Debug on Device - Xamarin“. Abgerufen am 20.01.2017 von https://developer.xamarin.com/guides/android/deployment,_testing,_and_metrics/debug-on-device/.

Xamarin (o.J. g): „Code Completion“. Abgerufen am 20.01.2017 von https://developer.xamarin.com/guides/cross-platform/xamarin-studio/ide-tour/#Code_Completion.

Zakas, Nicholas C. (2016): Understanding ECMAScript 6: the definitive guide for JavaScript developers. San Francisco: No Starch Press. — ISBN: 978-1-59327-757-4

Zaytsev, Juriy (o.J.): „ECMAScript 6 compatibility table“. Abgerufen am 16.01.2017 von https://kangax.github.io/compat-table/es6/.