Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Scientific Programming

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Frameworks zur plattformübergreifenden Entwicklung mobiler Applikationen

Seminararbeit von

Jannis Konopka

Jülich, Dezember 2016

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema **Frameworks zur plattformübergreifenden Entwicklung mobiler Applikationen** selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Ort und Datum

Unterschrift

Die Arbeit wurde betreut von:

Erstprüfer: Prof. Dr. rer. nat. Volker Sander
Zweitprüfer: Dipl. Phys. Ing. Waldemar Hinz

Diese Arbeit wurde erstellt in der
Zentralbibliothek (ZB)
des Forschungszentrums Jülich

Zusammenfassung

Der Schlüssel zum Erfolg eines jeden Dienstleisters liegt in der Nähe zu seinen Kunden. Heutzutage ist eine App die optimale Lösung, diesen Abstand zu minimieren. Für die Zentralbibliothek des Forschungszentrums ist es folglich eine gute Idee, mithilfe dieser Technologie einen Schritt auf die Nutzer zuzugehen. Die erforderlichen Rahmenbedingungen sind unter anderem plattformübergreifende Nutzbarkeit, einfache Wartbarkeit und Kostenersparnis bei der Entwicklung.
Diese Eingangsvoraussetzungen erfüllen spezielle Frameworks. Mittels ihrer Dienste wird die frameworkspezifische Sprache für jedes Zielsystem (Android, iOS, etc.) in eine native oder hybride App umgesetzt. Dieser Schritt vereinfacht dem Programmierer die Arbeit erheblich. Anstatt für jede Plattform eine separate, native App auf dem klassischen Weg zu entwickeln, muss nur noch einmalig ein Quellcode geschrieben werden, der alle Zielsysteme gleichzeitig bedient.
Die folgende Arbeit befasst sich mit verschiedenen Frameworks zur plattformübergreifenden Entwicklung mobiler Apps. Diese Frameworks werden anhand eines Fallbeispiels miteinander verglichen, um ihre Stärken und Schwächen herauszuarbeiten. Abschließend werden die gesammelten Erkenntnisse eingeordnet und darauf basierend eine Schlussfolgerung für die Zentralbibliothek aufgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Motivation und Einleitung

Smartphones haben unseren Alltag erobert. Für jede Lebenssituation gibt es bereits spezielle Apps, die genau auf ihre Nutzer angepasst sind. Analysen zufolge besitzen 2016 bereits 76% der Deutschen ein Smartphone, sodass über drei Viertel der Bevölkerung über Apps erreicht werden kann [Bit16].

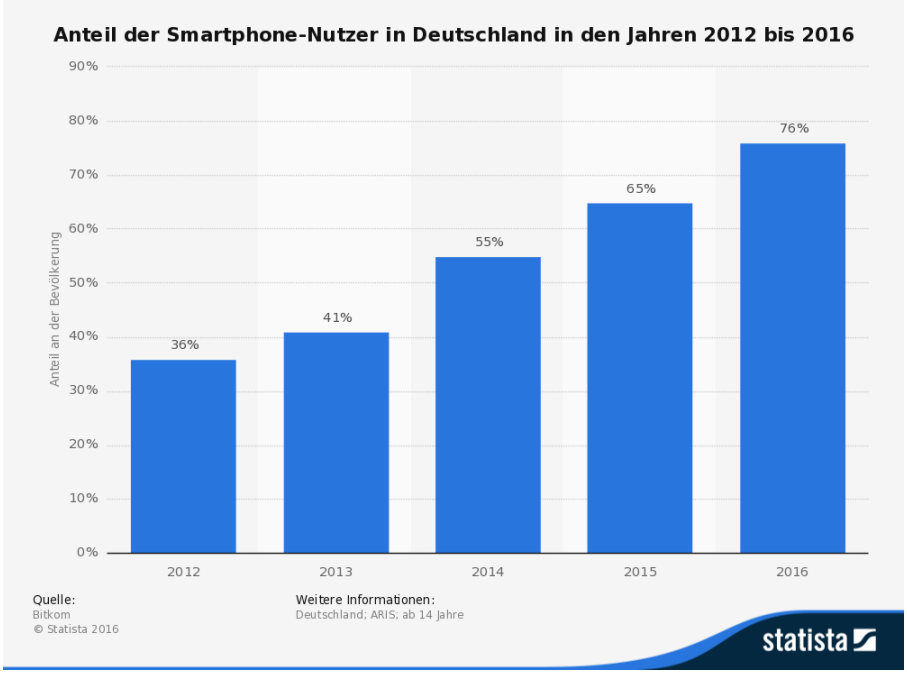

Abbildung 1.1: kontinuierlich ansteigende Nutzung von Smartphones in Deutschland [Bit16]

Für Dienstleistungsanbieter ist die Smartphone-Dichte ein überaus großer Vorteil. Um ihn ausnutzen zu können, muss der jeweilige Dienstleistungsanbieter jedoch eine Applikation entwickeln. An diesem Punkt können ungewollt hohe Kosten entstehen, wenn man sich bei der Entwicklung für die falsche Herangehensweise entscheidet.
Zwar ist die Smartphone-Abdeckung hoch, doch gibt es unterschiedliche Betriebssysteme, für die die native Programmierung von Apps verschieden abläuft. „Nativ“ steht dabei für die Entwicklung für ein konkretes Betriebssystem.

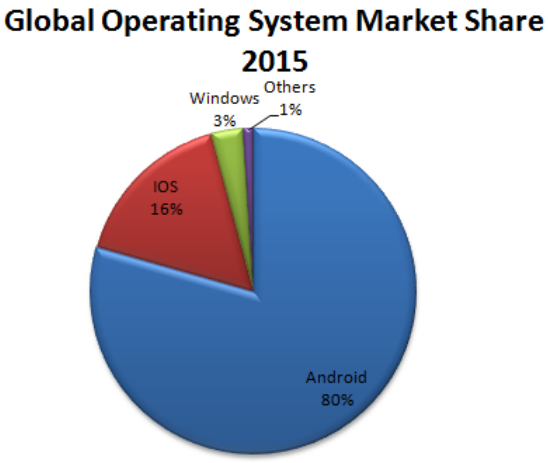

Abbildung 1.2: Marktanteile von Smartphone-Betriebssystemen weltweit [For15]

Wie auf der Abbildung 1.2 zu sehen ist, machen Android und iOS einen Marktanteil von 96 % unter sich aus. Es reichte somit fast aus, nur für diese beiden Zielplattformen zu entwickeln, um weite Teile der Zielgruppe abzudecken.
Nativer Code wird für Android in der Programmiersprache Java geschrieben. Bei iOS Programmen liegt die Sprache Objective-C zugrunde. Hätte der Entwickler kein plattformübergreifendes Framework zur Hand, müsste er sich beide Sprachen, die zugehörigen Entwicklungsumgebungen und die jeweiligen Erweiterungen zur App-Programmierung aneignen. Es wäre also eine überaus intensive Einarbeitung erforderlich, um zufriedenstellende Ergebnisse erzielen zu können. Entwickler von plattformübergreifenden Apps müssen hingegen nur eine Programmiersprache vertieft beherrschen und anwenden können.
Die Aufwendungen des Entwicklers reduzieren sich durch Einsatz solcher Frameworks also erheblich.
Eine Bibliothek wie die Zentralbibliothek des Forschungszentrums Jülich kann nicht alle Kapazitäten auf die Entwicklung einer App konzentrieren. Der Kostenrahmen muss sich deshalb in Grenzen halten. Ebenfalls sollte eine finanziell schonende und wirtschaftliche Wartbarkeit ermöglicht werden, die nicht manuell für jede Plattform vorgenommen werden muss.

Kapitel 2

Grundlagen

In Verbindung mit der Applikationsprogrammierung werden häufig Fachbegriffe verwendet, die einer genaueren Erklärung bedürfen.
So spricht man beispielsweise von mobilen Apps. Das „mobil“ steht dabei lediglich für die Verwendung der Software auf tragbaren Endgeräten, wie einem Smartphone oder einem Tablet-Computer.

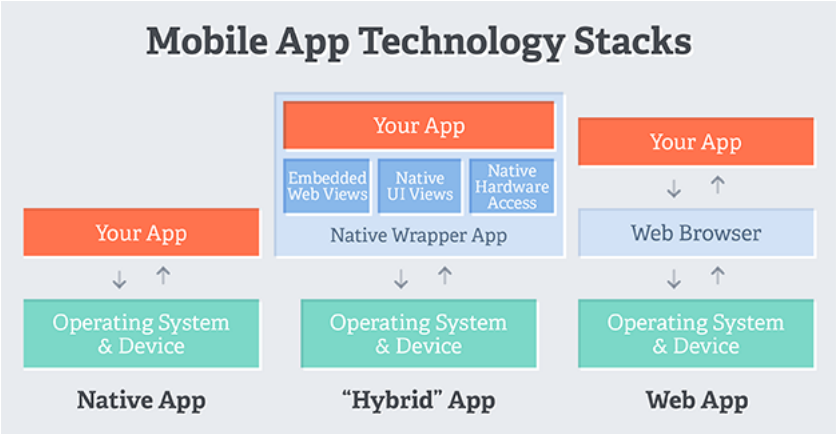

Abbildung 2.1: Funktionsweisen unterschiedlicher Apps [Fre13]

Ferner wird bei mobilen Apps zwischen „nativen“, „web“ und „hybriden“-Apps unterschieden.

2.1 Native Apps

Native Apps sind direkt für ein Betriebssystem programmiert und optimiert. Sie besitzen Zugriff auf alle Elemente der Hardware und haben deswegen ihren größten Nutzen, wenn Daten unterschiedlichster Sensoren ausgelesen und verarbeitet oder aufwändige Berechnungen durchgeführt werden sollen. Darüber hinaus ist die Speicherung von angefallenen Daten auf dem Endgerät leicht zu realisieren. Auch große Datenmengen stellen meist kein Problem dar. Nutzer sind mit dem Installationsverfahren nativer Apps vertraut: Der jeweilige AppStore wird von vielen häufiger aufgesucht, als der Kiosk um die

Ecke. Das ist auch verständlich, denn ein App-Download dauert meistens nur so lange, wie der Weg von der Couch bis zur Haustür.
Nach dem Download erkennt der Nutzer die geladene App am neuen Icon auf seinem Homescreen, weiß wie er sie öffnen kann und ist zufrieden.

2.2 Web-Apps

Die neben den nativen Apps existierenden Web-Apps funktionieren entschieden anders. Im Wesentlichen sind es Websites, die für mobile Endgeräte optimiert sind. Daraus folgt, dass die Laufzeitumgebung der Internet-Browser des jeweiligen Gerätes ist. Dieser Umstand bringt einige Vor- aber auch manche Nachteile mit sich.
Zum einen wird die Portabilität extrem erhöht. Jedes Smartphone ist vom Hersteller mit einem internetfähigen Browser ausgestattet. Somit kann der Nutzer durch einfaches Aufrufen der Webseite die Web-App ausführen. Zum anderen hat die gewonnene Kompatibilität ihren Preis. Durch den verwendeten Browser ist es nur in wenigen Ausnahmefällen möglich, auf die Hardware des Geräts zuzugreifen. In der Regel werden solche Zugriffe schon vom Betriebssystem verhindert, da ein Browser als sogenannte Sandbox fungiert. Dabei handelt es sich um einen isolierten Bereich, innerhalb dessen jede Maßnahme keinerlei Auswirkung auf die äußere Umgebung hat [Wik16e].
Das Speichern von Daten ist ein weiteres Problem des Sandbox-Systems. Eine Web-App kann lediglich Cookies im Browser speichern oder die Daten in den Speicherbereich des Webservers auslagern. Wenn es zur Datenauslagerung kommen sollte, muss aber auch ein Benutzerverwaltungssystem vorhanden sein, das wiederum an eine Datenbank angeschlossen ist. In jedem Fall entsteht erheblicher Mehraufwand, wenn Daten mittels einer Web-App dauerhaft gespeichert werden sollen.
Das Veröffentlichen von Web-Apps ist jedoch sehr viel einfacher, als das Veröffentlichen von nativen Apps. Während letztere über den jeweiligen AppStore vertrieben werden und dort einen Zulassungsprozess durchlaufen müssen, lassen sich Web-Apps aktualisieren, indem Sie mittels FTP (File Transfer Protocol) auf den Web-Server aufgespielt werden. Dadurch fällt es leichter, Web-Apps aktuell zu halten. Auch etwaige Kosten, die durch den Vertrieb im AppStore entstehen könnten, fallen weg [Ent16].

2.3 Hybride Apps

Hybride Apps vereinen die Vorteile von nativen und Web-Apps. Indem Sprachen der Webprogrammierung zum Einsatz kommen, können Apps erstellt werden. Für Nutzer wirken hybride Apps weitestgehend nativ, funktionieren allerdings anders. Sie laufen in einer nativen Hülle, die HTML, CSS und JavaScript darstellen kann. Praktisch ist es ein WebView in einer nativen App. Passt man die jeweilige native Hülle für jedes Betriebssystem einmalig an, erhält man Anwendungen, die plattformübergreifend auf unterschiedlichen Hard- und Software-Architekturen laufen können. In der Entwickung sind sie weder für eine spezielle Zielplattform, noch für ein spezifisches Betriebssystem

geschrieben. Daraus folgt eine Portabilität, die den Arbeitsaufwand für Programmierer erheblich verringert. Zwar sind hybride Apps nicht so portabel wie Web-Apps, dennoch flächendeckender einsetzbar als native Apps. Das resultiert zum einen aus der Laufzeitumgebung, die bei Web-Apps lediglich der Browser des ausführenden Smartphones ist, und zum anderen aus dem Umstand, dass nativ entwickelte Apps stets nur für ein konkretes Zielsystem existieren.
Der genaue Vorteil hybrider Apps liegt also darin, mittels weit verbreiteter Web-Sprachen über verschiedene Plattformen hinaus einheitlich programmieren zu können. Auf diese Weise werden vor allem Kosten minimiert. Darüber hinaus werden, je nach Framework, essentielle Teile der Geräte-Hardware unterstützt. So kann eine hybride App beispielsweise die Kamera eines Smartphones ansteuern.

2.4 Frameworks

Bei Frameworks handelt es sich um Programmiergerüste, die insbesondere in der objektorientireten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen verwendet werden [Wik16c]. Sie stellen dabei nur den Rahmen zur Verfügung und liefern noch keine fertigen Programme. Es ist möglich, Frameworks anhand ihres Typs zu klassifizieren. So gibt es beispielsweise Komponenten-Frameworks, deren Bestandteile sich vom Entwickler einzeln nutzen lassen können. Mittels Test-Frameworks können Programmstücke systematisch validiert werden, und Application-Frameworks stellen allgemeine Funktionen sowie Programmstrukturen zur Anwendungsentwicklung bereit. Frameworks dienen letztendlich dazu, gleichbleibende und immer wiederkehrende Problemfälle zu parametrisieren und durch Automatisierung möglichst viel für den Entwickler zu vereinfachen.

2.5 Frameworks zur plattformübergreifenden Programmierung

Zu den Application-Frameworks zählen auch spezielle Frameworks, die der plattformübergreifenden App-Entwicklung dienen. Sie wandeln den Code aus einer frameworkspezifischen Sprache in die Sprache der jeweiligen Zielsysteme um. Dies ist eine aufwändige Aufgabe: Der Quellcode muss portiert und mit den nativen Funktionen verknüpft werden. Daraus resultieren auch Nachteile. Gibt es Funktionen, die derart nativ und nah am Gerät arbeiten, kann es sein, dass sie gar nicht über das Framework erreichbar sind. So muss bei der Entwicklung mithilfe eines Frameworks auch auf spezielle Funktionen eines Betriebssystems verzichtet werden.
Auch aufwändige 3D-Spiele oder Anwendungen sollten besser in der nativen Sprache verfasst werden, da der Support der meisten Frameworks nicht darauf ausgelegt ist.
Der überaus große Vorteil solcher Frameworks ist jedoch das einmalige Implementieren. Der Code muss nicht doppelt und dreifach für spezielle Betriebssysteme geschrieben werden, sondern kann nach Belieben für verschiedene Plattformen genutzt werden.
Gute Ergebnisse können beispielsweise bei Datenbankanwendungen, Nutzereingaben und

sogar Kamerazugriffen erzielt werden.
Das Verwenden von Frameworks zur plattformübergreifenden Programmierung spiegelt also einen Tradeoff zwischen Aufwand und Nutzbarkeit spezifischer Funktionen wieder.

Kapitel 3

Marktübersicht

Auf dem Markt existiert eine Vielzahl Frameworks zur plattformübergreifenden Entwicklung mobiler Apps. Nicht alle haben dabei die gleiche Herangehensweise, sodass teilweise große Unterschiede zwischen den Frameworks bestehen.
Durch diese Diversität ist ein heterogener Markt gegeben. Er bietet für jeden Anwendungsfall verschiedene Lösungen, sodass vor der Entwicklung gut evaluiert werden muss, auf welches Framework vertraut werden soll.
Ebenfalls muss in diese Entscheidung der Kenntnisstand des Entwicklungsteams einfließen. Wird ein Framework gewählt, das auf einer dem Team bekannten Sprache aufbaut, sind schneller bessere Ergebnisse zu erwarten, als wenn eine dem Team unbekannte Sprache verwendet wird.
Des Weiteren ist mitzubeachten, wie schnell die App publikationsfähig sein kann und Marktreife erreichen muss.

3.1 Frameworks zur Erzeugung plattformübergreifender hybrider Apps

Hierbei werden aus Sprachen der Web-Programmierung Apps erzeugt, die in einem Web-View in einer nativen Applikation angezeigt werden können. Auf diesem Gebiet der mobilen Anwendungsentwicklung gibt es verschiedene Angebote.
Am häufigsten wird dabei auf Apaches „Cordova“ [Cor16] verwiesen. Dies ist ein Framework, auf das einige speziellere Frameworks wie Adobes „PhoneGap“ [Ado16] oder das kostenpflichtige Framework „Monaca“ [Asi16] aufbauen. Die Beliebtheit ist deshalb so groß, da es die klaffende Lücke (Gap) zwischen unterschiedlichen Plattformen wie iOS, Android und einigen anderen, nicht mobilen Systemen, schließt.

Sieht man Cordova als gegebene Funktionalität, wie Frameworks es tun, die darauf aufbauen, ist plattformübergreifendes Entwickeln auf eine Disziplin herunter gebrochen worden - das Designen des Web-Views. Dabei ist der Kreativität keine Grenze gesetzt, HTML und CSS in Verbindung mit Java-Script Logik nach Belieben auszunutzen.
Abstrahiert man auf die Metaebene der auf Cordova aufbauenden Frameworks, ist es für sie ein Leichtes, den Bedarf an hochwertigen Bedienoberflächen abzudecken.

Seitdem das Nutzen von Internet-Browsern am heimischen PC an Bedeutung gewann, hat man sich mit dem Problem, Inhalte elektronisch ansprechend und strukturiert zu vermitteln, auseinandergesetzt und Lösungen entwickelt. Diese Resultate nun

zur App-Entwicklung einzusetzen, ist der nächste logische Schritt. Da der Web-Markt aber bereits von Design-Frameworks wie beispielsweise „Bootstrap“ [Boo16] oder Logik-Frameworks, wie „AngularJS“ [Ang16], gesättigt ist, fällt der Apfel nicht weit vom Stamm, wenn Anbieter dieser Produkte auch Lösungen für die hybride App-Entwicklung mittels Cordova anbieten. Berühmte Vertreter, die eben dies tun, sind Frameworks wie „Ionic“ [IF16], Adobes „PhoneGap“, „Onsen UI“ [UI16], „Monaca“ und „Sencha Touch“ [Tou16].

Die Facebook-App war zunächst eine hybride App. Es entstanden jedoch Probleme bei der Performance und dem Scroll-Gefühl. Deshalb entschied das Unternehmen 2012 wieder auf die klassische native App-Programmierung umzusteigen [Ola16].
Vielleicht war die hybride Technologie damals noch nicht ausgereift genug. Vielleicht ist das Konzept hybrider Apps aber auch für derart große und umfangreiche Apps, wie die von Facebook, einfach nicht geeignet.
Ganz abgeschrieben hat Facebook die hybride App-Entwicklung nicht. Der übernommene Social-Media Dienst Instagram verwendet für die Timeline ein klassisches Web-View Element, in der die Posts mittels HTML dargestellt werden. Bekannte Unternehmen wie Amazon (Amazon AppStore [Dev16]) setzten diese Web-Views komponentenweise in ihren Apps ein, um Daten anschaulich darzustellen. Da sie ihr Geld verstärkt durch Apps verdienen, liegt es nahe, dass große Summen in die App-Entwicklung reinvestiert werden. Deshalb sind bei berühmten Anwendungsentwicklern (Facebook, Twitter, etc.) verstärkt native Apps anzutreffen.
Andere kleinere Smartphone Anwendungen wie der ADAC Mitfahrclub [Hyb16] oder FanReact, ein Social-Media Dienst zum Teilen von Sportreaktionen [Pho16], sind komplett hybride Apps. Das zeigt zum einen die Aktualität dieser Technologie, zum anderen beweist dieser Umstand aber auch, dass hybride App-Entwicklung nicht unterschätzt werden sollte und insbesondere für überschaubare Budgets mehr als einen Blick wert ist.

3.2 Frameworks zur Erzeugung plattformübergreifender nativer Apps

Neben den Frameworks zur hybriden App-Entwicklung existieren ebenfalls Frameworks, die native Apps für verschiedene Plattformen aus einer gleichen Codebasis bereitstellen können. Sie funktionieren ohne Web-View, arbeiten aber dafür in der Regel auf einer anderen Abstraktionsebene.
Anders als bei hybriden Frameworks, sind Aktualisierungen in der Regel hier schneller verfügbar, da mit den originalen SoftwareDevelopmentKits (SDK) jeder Plattform gearbeitet wird. Erscheint beispielsweise ein neues Android SDK, lässt es sich direkt einbinden. Das ist ein Vorteil gegenüber hybriden Frameworks, bei denen SDK-Aktualisierungen üblicherweise mit verzögerten Aktualisierungen der jeweiligen Plattform-Versionen einhergehen.

Generell bringen native Anwendungen im Vergleich zu hybriden Apps einen Peformance-Schub. Dafür ist die Kompilierzeit meistens länger, da das Framework Optimierungen,

Datenverknüpfungen und Assemblercode erzeugen muss.
Bekannte Vertreter dieser Framework-Gattung sind „Xamarin“ [Xam16c] oder „Native Script“ [Nat16]. Während Ersteres C# als Code-Quelle hat, verwendet NativeScript JavaScript. Obwohl JavaScript die Grundlage der hybriden Apps ist, erzeugt NativeScript, wie der Name vermuten lässt, native Anwendungen.
Mittlerweile bietet auch Intel eine Lösung zur plattformübergreifenden Entwicklung nativer Apps an. Mittels des XDK-Development-Kit lassen sie sich in HTML5, CSS und JavaScript schreiben [Int16]. Sie werden online mittels Cordova kompiliert und dem Nutzer anschließend bereit gestellt.
Die Kompilation wie Intel in die Cloud auszulagern, ist kein Einzelfall. Der Programmierer muss sich folglich keine Sorgen mehr um die Aktualität seiner verwendeten Software machen, da stets mit der aktuellsten Frameworkversion und den fortschrittlichsten SDKs der jeweiligen Zielplattformen kompiliert wird.
Egal ob hybride oder native plattformübergreifende Entwicklung: Der Markt wächst stetig. Letztendlich werden die Anbieter von Lösungen mit geringster Einarbeitungszeit, schnellsten und besten Ergebnissen, Garantie von Performance, kurzer Kompilierdauer und einfacher Testbarkeit die meisten Kunden für sich gewinnen. Für Anwender wird dieser Überschuss an Frameworks ein Segen sein. Sie können sich Frameworks nach ihrem Bedarf und beherrschten Programmiersprachen aussuchen, müssen allerdings auch Zeit in die Entscheidungsphase investieren.

3.3 Konkrete Frameworkauswahl

Im Rahmen dieser Arbeit werden zwei plattformübergreifende Frameworks ausgewählt, die im Folgenden näher beschrieben werden. Dabei wird auch die konkrete Funktionsweise der Softwarepakete beleuchtet.

3.3.1 Apaches Cordova

Apaches Cordova ist ein Framework zur Erstellung *hybrider* Apps.
Mittels der Hypertext Markup Language (HTML5), den Cascading Style Sheets (CSS 3) und der Skriptsprache JavaScript 6 wird zunächst eine Website programmiert. Im nächsten Schritt wird das entstandene Produkt für Zielplattformen veröffentlicht. Hierbei lassen sich Plattformen modular hinzufügen. Cordova erstellt für jedes gewählte Zielsystem eine native, leblose Verpackung (Wrapper), die ein Web-View-Gerüst enthält [Fou16].
Durch diese „Nativisierung“ einer mobilen App ist es nun möglich, die Hardware anzusteuern. Dazu besteht die Möglichkeit, Hardware ebenso modular hinzuzufügen wie Plattformen. In Cordova wird das durch Plugins realisiert. Einmal hinzugefügt, wird die entsprechende Funktionalität freigeschaltet und automatisch für jede Plattform in das Projekt eingepflegt.

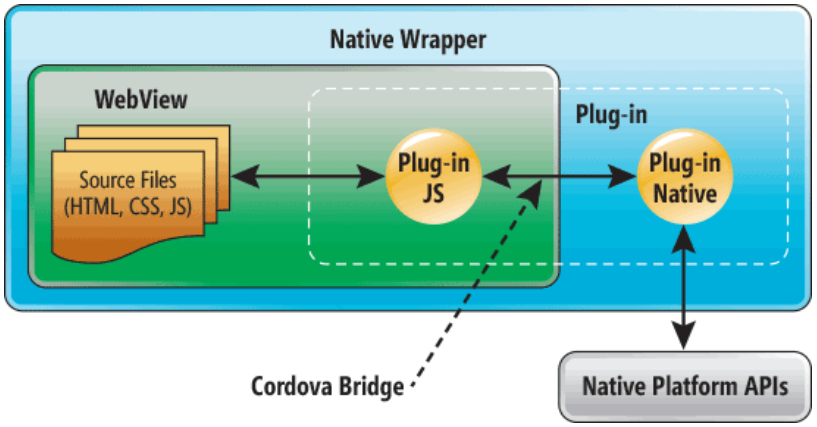

Abbildung 3.1: Die Struktur der Cordova-Device Brücke [KB14]

In dem WebView werden neben HTML, CSS und JavaScript-Code, auch die JavaScript-Plugins ausgeführt. Das geschieht, indem Cordova eine Brücke (Cordova Bridge) zwischen dem JavaScript-Plugin und dem nativen Plugin errichtet. Das native Plugin kommuniziert wiederum mit den nativen APIs (Application Programming Interfaces) der jeweiligen Plattform, um den Hardwarezugriff zu realisieren [KB14]. Abschließend müssen der nativen Hülle durch das Betriebssystem die konkreten Berechtigungen garantiert werden.
Ist der Zugriff schließlich durch den Nutzer autorisiert, kann die Hardwarefunktionalität benutzt und mittels JavaScript nach Belieben programmiert werden.

3.3.2 Microsoft's Xamarin

Das Xamarin Framework ist ein Tool zur plattformübergreifenden Entwicklung *nativer* Apps. Die erzeugten Applikationen sind vollständig nativ und enthalten keine Hülle, in der ein WebView ausgeführt wird, so wie es bei hybriden Apps der Fall ist.
Im Vergleich zur fertigen App ist die Entwicklung keinesfalls nativ. Hierbei können die wichtigsten mobilen Plattformen Android, iOS und WindowsPhone mit C# (C Sharp) bedient werden. Diese Programmiersprache zeichnet sich als typsichere, objektorientierte Allzwecksprache aus, die hauptsächlich für Microsoft's Windows entwickelt wurde [Wik16b]. Mittels Xamarin ist es mit ihr nun möglich, mobile Zielsysteme zu erreichen. Die Entwicklung findet mit Xamarins Produkt Xamarin.Forms [Xam16a] statt. Es bietet die benötigte Funktionalität, einen geteilten Code für Android, iOS und WindowsPhone zu schreiben und plattformübergreifend zu nutzen. Mithilfe des Teilprodukts Xamarin.Forms.Portable [Xam16d] lässt sich ein portables Objekt erstellen, in dem man seine App losgelöst von einer speziellen Plattform programmiert und mithilfe einer Variante der Extensible Markup Language designt (XAML). Die konkrete Benutzeroberfläche auf der jeweiligen Zielplattform wird dann von Xamarin automatisch generiert.
Die Umwandlung in eine native App geschieht im Speziellen bei jeder Plattform anders.

Unter Xamarin.Android wird der C# Quellcode mittels der Mono Virtual Machine (MonoVM [Mon16]) in ein Zwischenprodukt kompiliert. Das entstandene Programm liegt dann in der Common Intermediate Language (CIL) vor. Dies ist eine als Zwischensprache fungierende objektorientierte Assemblersprache, die an keine Hardware gekoppelt ist und dadurch zu anderen Maschinen portabel bleibt. Alle Implementierungen der Common-Language-Infrastructure (CLI) Spezifikation werden letztendlich in die CIL übersetzt. Berühmteste Implementierung dieser CLI Spezifikation ist Microsofts .NET Framework zur Anwendungsentwicklung.
Xamarin verwendet Mono. Das ist eine quelloffene Implementierung eben dieses .NET Frameworks. Die fertige Applikation in der CIL-Zwischensprache wird nun in ein Android-Package (APK) verpackt und auf dem Gerät installiert.

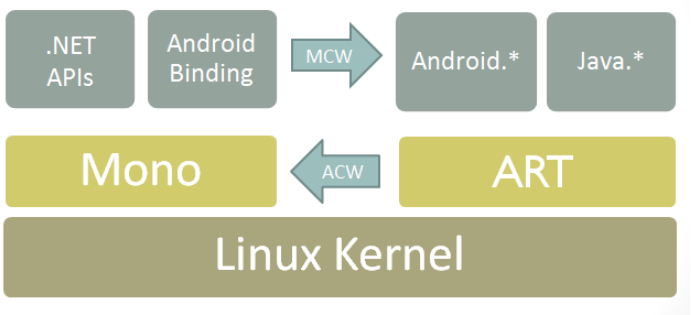

Abbildung 3.2: Betrieb einer Xamarin.Android Applikation [Xam16e]

Xamarin.Android Applikationen laufen in der MonoVM, die parallel zur Android-Laufzeitumgebung (ART) betrieben wird. Beide Laufzeitumgebungen sind wiederum auf dem Linux Kernel aktiv.
Nun wird die Applikation ausgeführt. Wie es bei .NET Anwendungen üblich ist, übersetzt die .NET-Laufzeitumgebung den CIL-Code über einen Just-In-Time-Compiler (JIT) [Wik16d] in eine native Assembler Sprache. Dies geschieht zur Laufzeit in der MonoRuntime. Um die Kommunikation im aktiven Betrieb zwischen Applikation und Hardwarefunktionen zu gewährleisten und die nötigen Verbindungen (Bindings) herzustellen, existieren Managed Callable Wrapper (MCW) und Android Callable Wrapper (ACW). Sie sorgen dafür, dass der dynamisch generierte Assembler-Code zur Hardwarefunktionalität zwischen den verschiedenen Zuständigkeitsbereichen ART und Mono ausgetauscht und ausgeführt wird. [Xam16e].

Unter Xamarin.iOS ist es durch den Apple iOS Kernel verboten, den Quellcode während der Laufzeit zu generieren. Somit muss statt des Just-In-Time-Compilers ein Ahead-Of-Time-Compiler (AOT [Wik16a]) verwendet werden. Diesen Schritt übernimmt das iOS-SDK Tool M-Touch. Es kompiliert aus der CIL ein iOS-Bundle. Dieses Endprodukt lässt sich nun auf iOS Simulatoren oder echten Apple-Geräten installieren.
Wie man Abbildung 3.3 entnehmen kann, bleibt der vorherige Schritt, die Kompilation eines C# Programmes in die Zwischensprache CIL, weitestgehend gleich.

Abbildung 3.3: Kompilationsablauf einer Xamarin.iOS Applikation

Die Verwendung des Ahead-Of-Time-Compilers hat zur Folge, dass sich der Betriebsablauf einer Xamarin.iOS App von der Xamarin.Android App unterscheidet.
So ist eine Kommunikation über Wrapper für spezielle Funktionalitäten nicht mehr erforderlich. Durch das Kompilieren vor der Laufzeit wurden bereits die Verbindungen (Bindings) zwischen der .NET API und der iOS-API hergestellt.
Auf der anderen Seite bestehen auch Gemeinsamkeiten zwischen dem Betrieb einer Xamarin.iOS-App und einer Xamarin.Android-App. So existiert auch unter iOS die Parallelität von zwei Laufzeitumgebungen. Anstelle der AndroidRunTime(ART) wird hier von Apple allerdings die Objective-C Runtime verwendet und für den Mono Just-In-Time-Compiler tritt hier eine Mono Laufzeitumgebung an, die den M-Touch-Ahead-Of-Time-kompilierten Code ausführt.

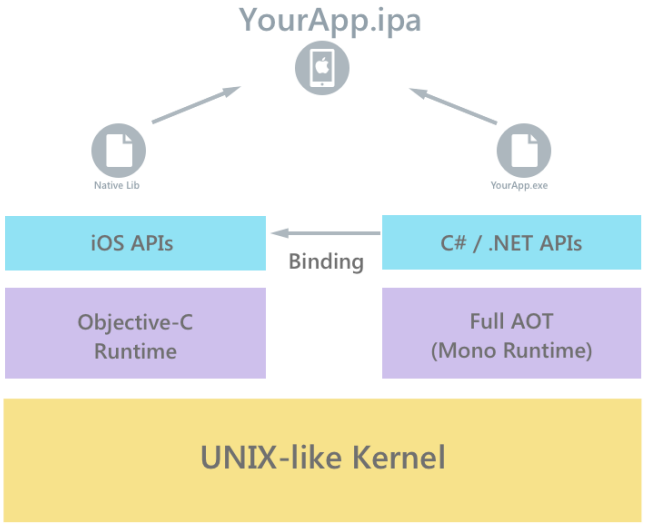

Abbildung 3.4: Betrieb einer Xamarin.iOS Applikation [Xam16b]

Kapitel 4

Implementierung einer Beispielanwendung

In diesem Kapitel wird auf die konkrete Entwicklung einer Anwendung eingegangen. Mittels den eingangs beschriebenen plattformübergreifenden Frameworks Cordova und Xamarin soll eine Beispielanwendung entstehen, die als Grundlage für den Frameworkvergleich dienen soll. Da die Frameworks auf unterschiedlichen Sprachen basieren, handelt es sich um die Implementierung zweier von Grund auf verschiedener Apps.

4.1 Das Beispielszenario: Eine Barcodescanner-App

Als Testszenario dient die Entwicklung einer Barcodescanner-App. Sie soll Zugriff auf die Kamera gewährleisten und dadurch das Scannen des Barcodes eines Buchs ermöglichen. Mit Hilfe der ermittelten ISBN Nummer lässt sich das Buch anschließend im Bibliotheks-eigenen Katalog JuLib eXtended suchen. Nutzer können dann nähere Informationen zum Exemplar erhalten oder auch ähnliche Bücher finden, die ihren Wünschen entsprechen. In Zukunft kann die App um die Funktionalität des Verleihens erweitert werden. Dadurch besteht die Möglichkeit, auch in der Nacht, wenn der Lesesaal nicht personell besetzt ist, aber mittels Nachtausweis zugänglich bleibt, Bücher auszuleihen.

Den Algorithmus des Barcode-Scannens zu implementieren ist nicht Teil dieser Arbeit. Hier wurde auf frei verfügbare Bibliotheken vertraut, die bereits vollkommen optimiert und zuverlässig den handelsüblichen Buch-Barcode-Standard EAN-13 erkennen können. Im Folgenden werden die Entwicklungsschritte frameworkspezifisch aufgezeigt.

4.2 Xamarin

Im Zuge dieser Arbeit wurde Xamarin 4.2 für Microsoft Visual Studio Professional in der Version 14.0 unter dem Betriebssystem Microsoft Windows 8.1 benutzt. Bei Visual Studio handelt es sich um eine integrierte Entwicklungsumgebung (IDE) der Firma Microsoft, die viele Funktionen der modernen Programmierung unterstützt. So ist beispielsweise das Debuggen zur Laufzeit möglich, indem während der Programmausführung der Wert von Variablen ausgelesen und der Aufrufstapel abgerufen werden kann. Zusätzlich ist ein Versionskontrollsystem fest integriert, sodass optimal im Team gearbeitet werden kann.

Xamarin lässt sich in Visual Studio einfach als Plugin hinzufügen.
Darüber hinaus ist auch eine Verwendung über die Xamarin-Studio IDE möglich. Hiermit ist es allerdings schwieriger für iOS zu entwickeln, sofern eine Windows-Maschine benutzt wird.

4.2.1 Entwurf der Applikation

Unter Xamarin findet die Entwicklung in Xamarin.Forms.Portable statt. Dieses Unterprodukt bietet die Möglichkeit der Entwicklung mittels einer .NET-Teilmenge, die intern vollständig kompatibel zu Android, iOS und WindowsPhone ist.

Verwendung finden außerdem zwei externe Open-Source Bibliotheken, die sich über das .NET-Paketverwaltungssystem NuGet [NuG16] problemlos in VisualStudio hinzufügen lassen.
Um die Scanner-Funktionalität zu gewährleisten, wird ZXing [ZXi16] eingebunden. Damit kann das Live-Bild der Kamera ausgewertet werden, sodass Barcodes direkt gefunden werden können.
Damit der in alphanumerische Zeichen aufgelöste 13-stellige Barcode im bibliothekseigenen Katalog JulibExtended gesucht werden kann, wird das HTMLAgilityPack [HTM16] genutzt. Es kommuniziert über das HTTP-Protokoll mit Websites. So können einfache GET-Anfragen verschickt und entsprechende Antworten verarbeitet werden.

In Xamarin wurde mit dem Model-ViewModel-View (MVVM) Programmiermuster gearbeitet. Damit lässt sich die Applikation klar strukturieren.

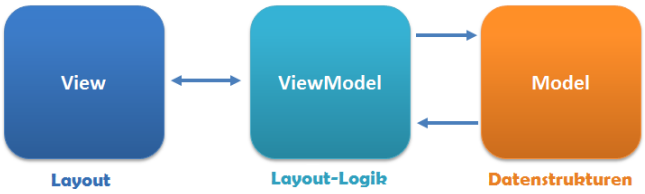

Abbildung 4.1: Programmiermuster Model-ViewModel-View (MVVM) [Bil16].

Das Model beschreibt hierbei die Backend-Logik der Anwendung, beispielsweise eine Webanfrage. Die View stellt die Benutzeroberfläche dar. Dies beinhaltet alle plattformspezifischen Anzeigeelemente.
Das ViewModel ist die Verbindung zwischen View (Benutzeroberfläche) und Model (Geschäftslogik). Es ruft Funktionen der Model-Schicht auf und stellt alle öffentlichen Eigenschaften und Befehle der View Schicht zur Verfügung.

Der typische Ablauf des Barcode-Scannens läuft wie folgt ab:

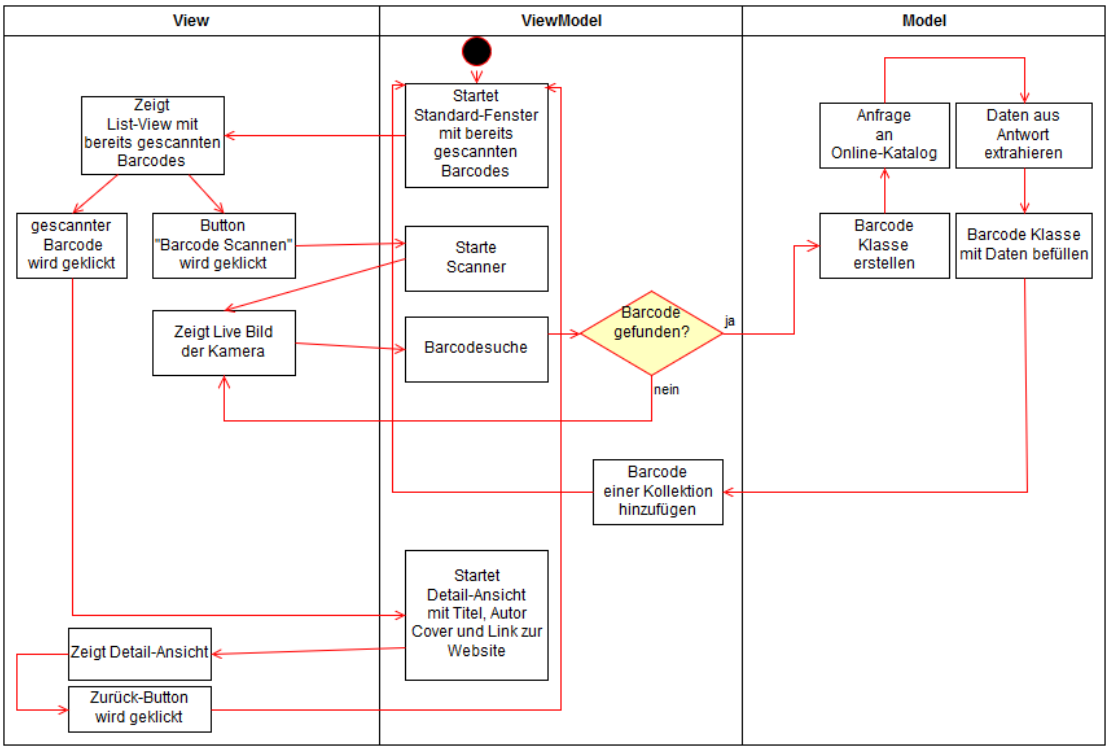

Abbildung 4.2: Aktivitätsdiagramm für die Barcodescanner Applikation unter Xamarin

Der Nutzer startet die App (ViewModel) und befindet sich in der Hauptansicht (View). Hier werden alle Barcodes, die bereits gescannt wurden, in einer Liste angezeigt. Logischerweise ist die Liste mit Barcodes beim Start der App leer.

Der Nutzer kann nun auf den Button „Barcode scannen“ klicken (View). Der Befehl wird weitergereicht (ViewModel) und der eigentliche Scanner wird gestartet (ViewModel). Nun öffnet sich die Kamera (View). Der Algorithmus zur Barcodesuche wertet die Live-Bilder solange aus, bis ein Barcode gefunden wurde (ViewModel). Tritt das Ereignis ein, wird der Scanner geschlossen (View) und eine Anfrage zum bibliothekseigenen Katalog gesendet (Model). Die Antwort der Website wird extrahiert und mittels einer Barcode Klasse (Model) in einer Kollektion gespeichert (ViewModel). Die Kollektion wird wiederum in einer ListView für den Nutzer dargestellt (View). Somit kann er seine bereits gescannten Barcodes betrachten.
Der Nutzer hat außerdem die Möglichkeit, durch Klicken auf einen ListView-Eintrag (View) in eine Detail-Ansicht zu gelangen (View). Hier erhält er Informationen wie Titel und Autor, kann das Buchcover betrachten und erreicht über einen Link die spezifische Website des gescannten Elements im Bibliothekskatalog.

4.2.2 Weg zur Android-App

Um die Applikation für Android zu vervollständigen, muss im Android Projektverzeichnis eine Kleinigkeit hinzugefügt werden. Das Paket, das es ermöglicht, Barcodes zu erkennen, erfordert eine Initialisierung in der MainActivity. Dazu muss eine einzelne Zeile Code im Android-spezifischen Projektverzeichnis hinzugefügt werden. Das ist schade, denn somit hat sich die App nicht zu 100% plattformübergreifend entwickeln lassen.

Das Testen unter Android funktioniert hingegen einwandfrei. Aus Visual Studio heraus ist es möglich, Android SDKs zu verwalten oder einen Emulator zu starten. Über integrierte Bedienelemente lässt sich die geschriebene App wahlweise direkt auf ein emuliertes Smartphone oder ein reales Gerät installieren. Der Emulator bietet die Möglichkeit, über die Webcam des PCs eine Smartphone-Kamera zu simulieren, sodass auch im Testmodus am PC Barcodes gescannt werden können.

Folgende Ergebnisse konnten unter Android erzielt werden. Die Screenshots stammen von einem Samsung S5 Smartphone, das mit Android 6.0.1 betrieben wird.

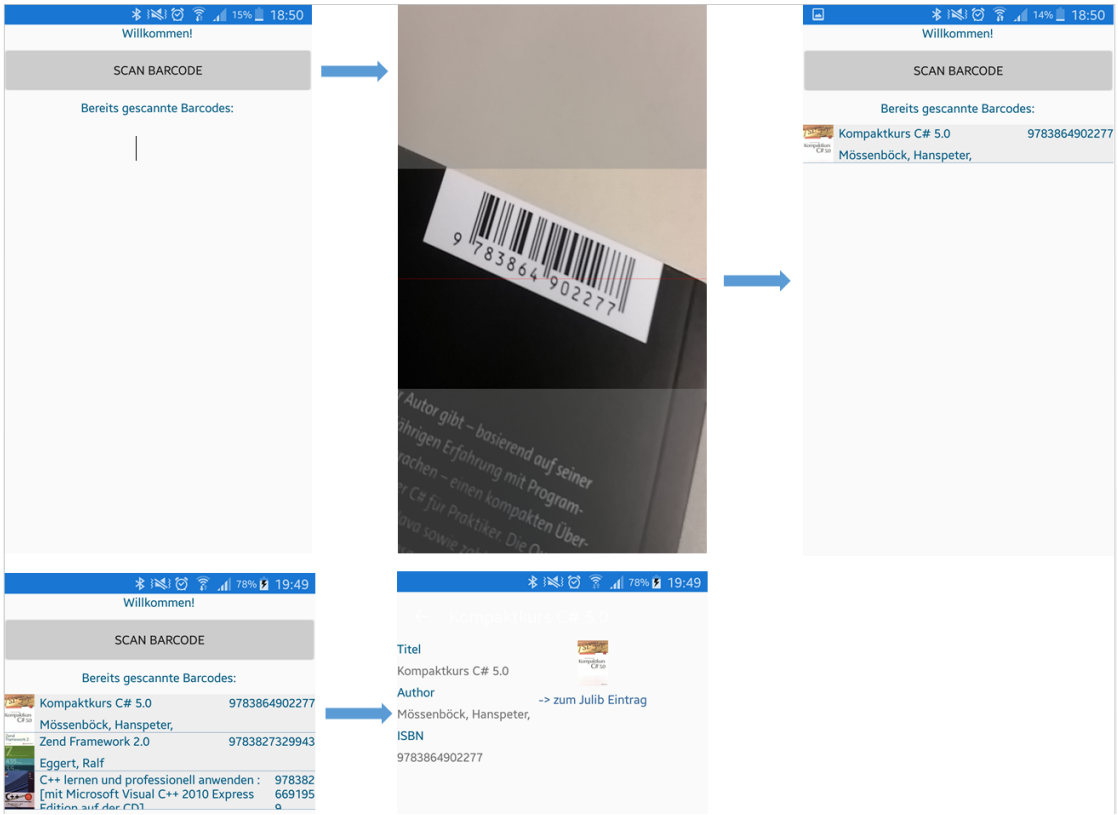

Abbildung 4.3: Screenshots der Xamarin Android App

4.2.3 Weg zur iOS-App

In Xamarin besteht die Möglichkeit, für iOS zu entwickeln. Das ist keineswegs trivial, denn iOS Software lässt sich nur auf einem Mac kompilieren.
Visual Studio bietet nun die Möglichkeit, einen Mac über das Netzwerk hinzuzufügen. Xamarin erhält über Fernanmeldung Zugriff auf den Mac und kann dort die installierten Provisioning Profiles (Bereitstellungs-Profile) verwenden. Diese Provisioning Profiles dienen der Verifikation des Entwicklers und spiegeln eine Sicherheitsstufe in Apples System wieder. Zusätzlich zum Mac benötigt jedes Testgerät das entsprechende Provisioning Profile. Es lässt sich einfach über Xcode installieren.

Seit der Veröffentlichung von Xcode 7 im Sommer 2015 gibt es zwei Arten von Provisioning Konten. Free Provisioning heißt die hinzugekommene Kontenart, mit der es kostenlos möglich ist, selbst geschriebene Apps für Testzwecke auf realen Geräten zu installieren. In früheren Versionen konnte man diese Funktion nur erwerben, indem man Apples Developer Program beitrat. Die Teilnahme ist immer noch Pflicht, um Apps für Apples Appstore zu veröffentlichen und zu vermarkten. Durch die Eintragung in das Programm entstehen jährliche Kosten in Höhe von circa 100 Euro.

Da es nicht Ziel dieser Arbeit ist, eine marktreife App zu entwickeln und zu vermarkten, fiel die Entscheidung für das Free Provisioning Konto. Um diese Möglichkeit zu nutzen, ist es notwendig, eine Apple ID zu erstellen und anschließend zum Developer Account kostenlos aufzuwerten.
Mit diesen Daten meldet man sich nun auf dem Mac in Apples integrierter Entwicklungsumgebung Xcode an. Nachdem man ein Testprojekt erstellt hat, sollte ein sogenannter Bundle Identifier festgelegt werden. Dieser muss bei eigenständig programmierten iOS oder Mac Apps angegeben werden, um die Software auf einem echten Gerät testen zu können. Er ist außerdem einmalig und verbindet die App eindeutig mit dem verwendeten Provisioning Profile.

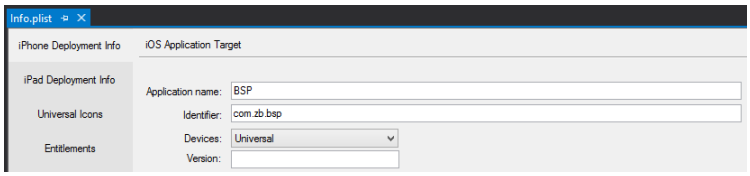

Abbildung 4.4: Ausschnitt aus der Info.plist mit Bundle Identifier in Visual Studio 2015

Das verwendete Provisioning Profile enthält Zertifikate, die immer dann Verwendung finden, wenn der Mac eine App für ein Testgerät signiert.

Abbildung 4.5: Apples ausgestellte Zertifikate zur App-Signierung

Zum Testen einer Xamarin App muss im Xamarin Projektverzeichnis unter iOS der zuvor angelegte Bundle Identifier eingetragen und ein iOS-Endgerät mit bereits installiertem Provisioning Profile an den Mac angeschlossen werden. Außerdem muss die Mono-Laufzeitumgebung vorhanden sein, die Bestandteil von XamarinStudio für Mac ist. Nachdem diese Installation vollendet ist, sind alle Vorkehrungen getroffen. Die App lässt sich nun durch Klick auf integrierte Bedienelemente in Visual Studio auf dem an den Mac angeschlossen Gerät starten, sofern eine VPN Verbindung zwischen den beiden Geräten besteht.

Folgende Ergebnisse konnten unter iOS erzielt werden. Die Screenshots stammen von einem iPad Air 2 Tablet, das mit iOS 10.1 betrieben wird.

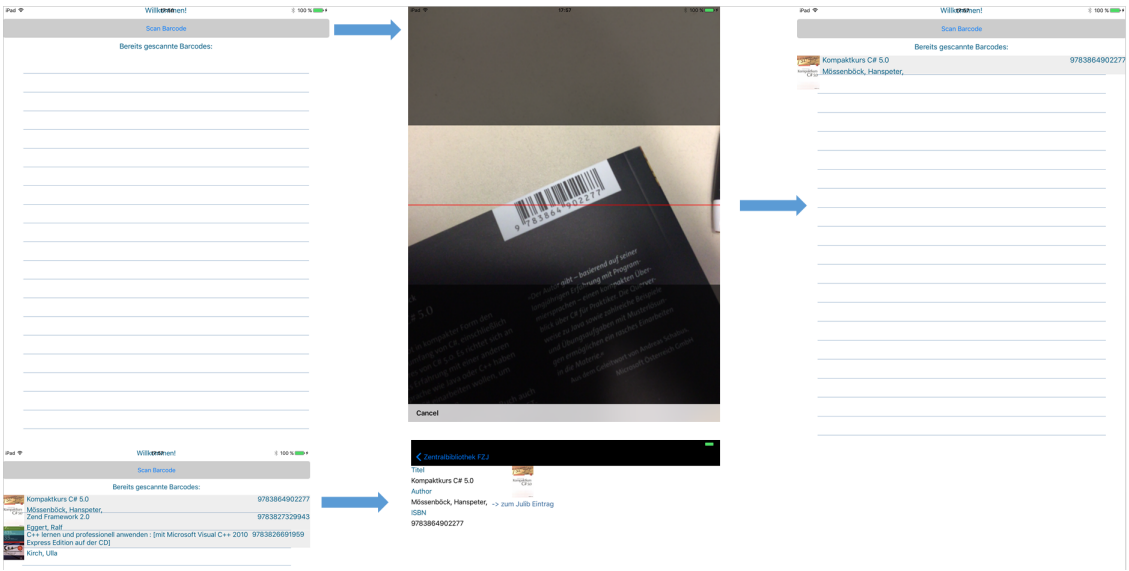

Abbildung 4.6: Screenshots der Xamarin iOS App

4.3 Cordova

Die Implementierung der Barcodescanner-App findet unter der Cordova-Version 6.3.1 statt. Dazu wird Cordova auf einer Windows 8.1 Maschine und einem MacBook Pro mit macOS Sierra 10.12 zugänglich gemacht. Die Installation erfolgt auf beiden Betriebssystemen kommandozeilenbasiert über den Node Package Manager (npm). Da dieser ein Paketmanager für die JavaScript-Laufzeitumgebung Node.js ist, muss eben jene Laufzeitumgebung vorab eingerichtet werden.
Der folgende Befehl führt letztendlich die Installation durch, sobald er im betriebssystemspezifischen Kommandozeilentool abgesetzt wurde. Für Windows ist das die Eingabeaufforderung, während für iOS ein Terminal das Standardwerkzeug ist.

```
npm install -g cordova
```

Um nun das Projektverzeichnis einer Cordova-Applikation anzulegen, muss die nachstehende Zeile in der Kommandozeile abgesetzt werden. BSP steht hierbei für den Projektnamen BarcodeScannerProject.

```
cordova create BSP com.zb.bsp bsp
```

Es wird ein Projekt mit Namen bsp im Ordner BSP mit der Kennung com.zb.bsp angelegt. Diese Kennung wird unter iOS als Bundle Identifier verwendet. Anschließend navigiert man in den angelegten Ordner hinein.

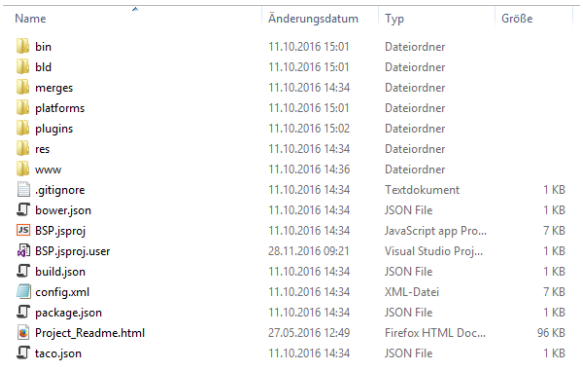

Abbildung 4.7: Ordnerstruktur eines Cordova Projekts

Nun lassen sich komponentenweise Plattformen hinzufügen, für die entwickelt werden soll. Im Rahmen dieser Arbeit wurden iOS und Android als Zielplattformen ausgewählt. Um sie hinzuzufügen, sind die nachfolgenden Zeilen abzusetzen.

```
cordova platform add android
cordova platform add ios
```

Ähnlich wie Plattformen lassen sich Plugins modular hinzufügen. Für die angestrebte Barcodescanner-Applikation werden die Plugins für Kamerazugriff, In-App-Browser und den Barcodescanner benötigt. Diesem Zweck dienen folgende Kommandos:

```
cordova plugin add cordova-plugin-camera
cordova plugin add cordova-plugin-inappbrowser
cordova plugin add phonegap-plugin-barcodescanner
```

Alternativ lässt sich eine Cordova-Applikation unter Windows ebenfalls in Visual Studio erstellen. Hier werden automatisch Aktualisierungen von Cordova heruntergeladen und Plugins installiert. Dadurch verliert der Programmierer aber möglicherweise den Überblick über den Aufbau seiner App, sodass es im weiten Verlauf der Programmierung und insbesondere beim Entwickeln für iOS zu Verständnisproblemen kommen kann.

4.3.1 Entwurf der Applikation

Von der verwendeten Installationsart unabhängig ist die Programmierung der App. Sie findet ausschließlich im www-Ordner statt. Dort liegen die Web-Dateien mit den Endungen .html .css und .js in den jeweiligen Unterordnern.

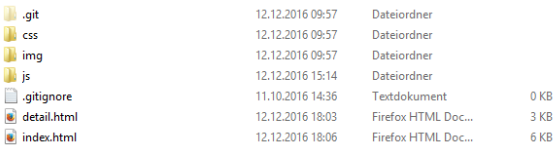

Abbildung 4.8: Ordnerstruktur eines Cordova www-Verzeichnisses

Während man in einer HTML-Datei das Layout festlegt und mittels CSS-Dateien den Stil der App bestimmt, sind die JavaScript-Dateien für die Logik und Dynamik der entstehenden Applikation zuständig. Diese enthalten Skripte, die von HTML-Dateien ereignisorientiert aufgerufen werden.

4.3.2 Weg zur Android-App

Unter Android lassen sich Builds und Tests mittels den unterhalb stehenden Befehlen durchführen.

```
cordova build android
cordova run android
```

Cordova geht dann von dem Android SDK aus, das als Umgebungsvariable ANDROID_HOME definiert ist. Wenn ein Android-Gerät an den PC angeschlossen ist, wird dieses zur Ausführung bevorzugt. Sollte dies nicht der Fall sein, wird der Default-Emulator gestartet. Dieser ist standardmäßig im verwendeten Android SDK enthalten.
Visual Studio bietet auch hier wieder die Möglichkeit, mit integrierten Bedienelementen

für wahlweise Emulator oder reales Gerät zu testen.

Folgende Ergebnisse konnten unter Android erzielt werden. Testgerät war auch hier wieder ein Samsung S5 Smartphone mit Android 6.0.1.

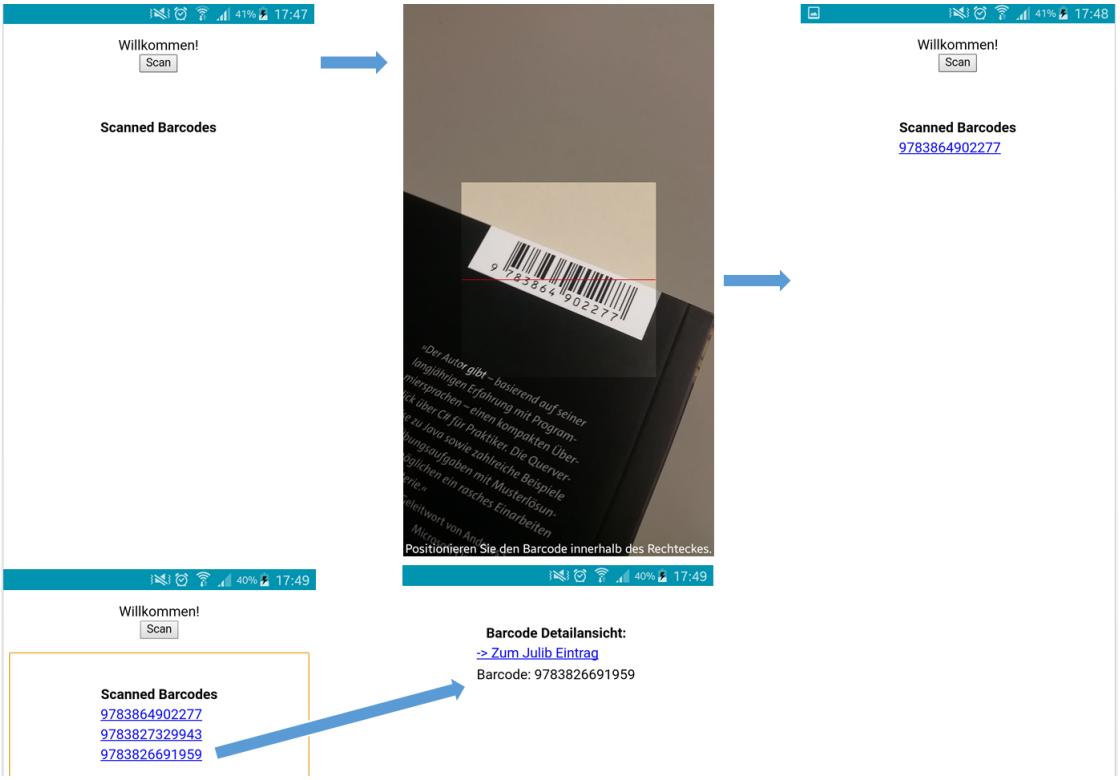

Abbildung 4.9: Screenshots der Cordova Android App

4.3.3 Weg zur iOS-App

Um die entwickelte mobile Anwendung auch für iOS zu erstellen, ist bei Cordova ein Mac erforderlich, der die App kompiliert. Dazu muss auf dem Apple-PC das gleiche Projektverzeichnis angelegt werden wie unter Windows. Wichtig ist das Hinzufügen der Plattform iOS und der gleichen Plugins, die auch unter Windows verwendet wurden.
Der www-Ordner mit geschriebenem Programm kann mittels eines Versionsverwaltungssystems wie GIT zentral gespeichert und auf dem Mac wiederverwendet werden.
Sobald die Plattform iOS hinzugefügt wurde, ist eine Xcode Projektdatei vorhanden. Nach dem Öffnen mit Xcode, verläuft das Erstellen und Testen der Applikation wie bei einer nativen App für iOS.

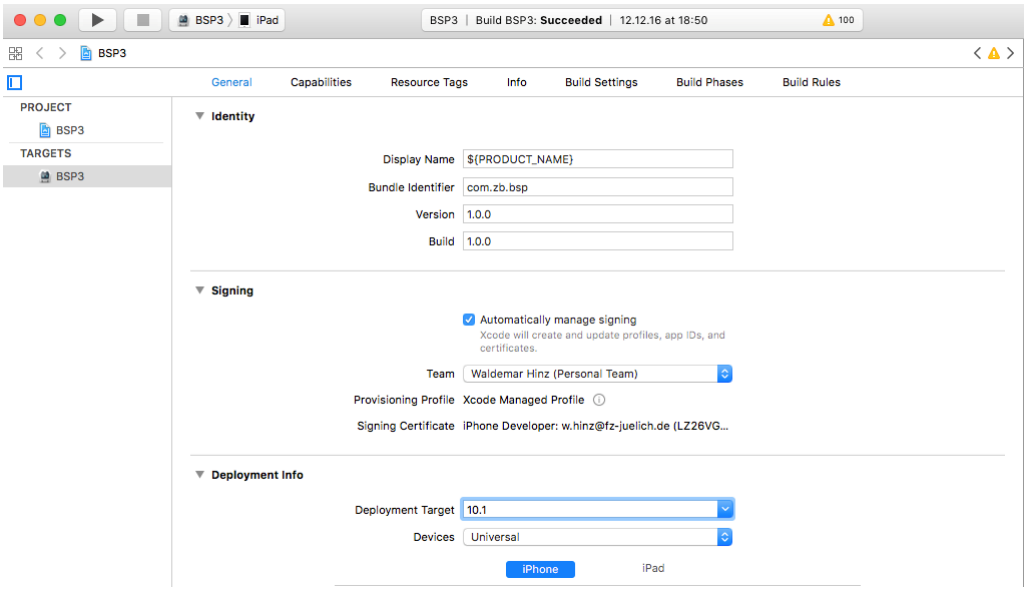

Abbildung 4.10: Screenshot der Info.plist in Xcode auf einem Mac für Cordova

Über die Datei Info.plist lassen sich letzte Einstellungen vornehmen, ein Bundle-Identifier final festlegen und die verwendete iOS-Version zum Kompilieren bestimmen. Nun besteht die Möglichkeit, die App wahlweise auf Simulator oder realem Gerät mit dem Deploy-Button oben links zu testen.

Die nachstehenden Ergebnisse konnten unter iOS erzielt werden. Die Screenshots stammen auch hier von einem iPad Air 2 Tablet mit installiertem iOS 10.1.

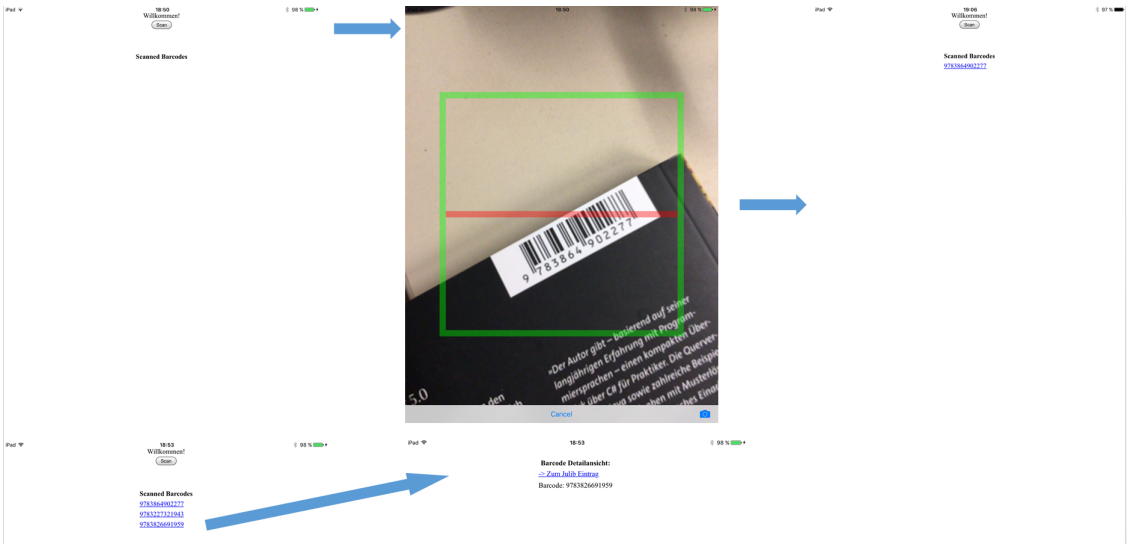

Abbildung 4.11: Screenshots der Cordova iOS App

Kapitel 5

Vergleich

Während der Implementation der Barcodescanner-Applikationen mit beiden Frameworks, konnten einige Unterschiede, Vorteile und Nachteile festgestellt werden. In diesem Abschnitt werden deshalb Kriterien festgelegt, die im Anschluss für die Bewertung der beiden Frameworks genutzt werden.

5.1 Kriterien

In Anlehnung an die Norm ISO/IEC 25000 [ISO05] kann die Qualität von Software anhand der Qualitätsmerkmale Änderbarkeit, Effizienz, Übertragbarkeit, Zuverlässigkeit, Funktionalität und Benutzbarkeit bewertet werden. Für den Vergleich der Frameworks werden diese Merkmale an wichtige Eckpunkte der plattformübergreifenden Programmierung angepasst und stellen sich wie folgt zusammen:

Tabelle 5.1: Bewertungskriterien für Frameworks zur plattformübergreifenden Applikationsentwicklung für mobile Zielsysteme

Kriterien
Installierbarkeit und Nutzbarkeit
Installationsaufwand
Hauptprogramme Mac
Hauptprogramme Windows
iOS Entwicklung unter Windows
Test von iOS Versionen unter Windows
Kosten
Fixkosten
Android-Extrakosten
iOS-Extrakosten
Dokumentation & Community
Dokumentation
Online-Referenzen
Umfang
umfangreicher Hardwarezugriff
mobile Zielsystemabdeckung
Einarbeitung
Programmiersprachen
Schwierigkeit der Einarbeitung
Erstelldauer
Kompilierdauer
Deploymentdauer
Programmverifizierung
integrierte Testverfahren
Ergebnisse
Performance
Look and Feel

5.2 Installierbarkeit und Nutzbarkeit

Für die Nutzung von Cordova und Xamarin müssen viele kleine Teil- und Unterprogramme installiert werden. Xamarin erledigt dies jedoch alles im Hintergrund, während einer einmaligen Installation. Bei Cordova entsteht an dieser Stelle größerer Aufwand, da die grundlegenden Pakete zumeist per Hand über die Kommandozeile installiert werden müssen. Hierbei kann es auch zu Problemen kommen. Sollten beispielsweise schon veraltete Versionen von verwendeten Paketen installiert sein, könnte die weitere Nutzung durch das Framework zu fehlerhaftem Verhalten führen.

Die iOS Entwicklung ist bei beiden Frameworks über Windows möglich. Xamarin bietet hier zusätzlich die Möglichkeit des Testens über eine Netzwerkverbindung zu einem Mac. Das ist ein starker Vorteil gegenüber Cordova, da nicht die Notwendigkeit besteht, zwei Installationen durchzuführen und den Code auf der Windows Maschine als auch dem Mac aktuell zu halten.

5.3 Kosten

Cordova ist ein OpenSource Projekt und somit in vollem Umfang kostenlos nutzbar.

Xamarin ist seit März 2016 ein kostenloser Bestandteil von Visual Studio, nachdem Microsoft zuvor die Firma gekauft und als Tochterfirma eingegliedert hat. Seitdem ist Xamarin kostenlos über den in Visual Studio integrierten Paketmanager NuGet installier- und frei nutzbar. Auch Visual Studio ist in der Community-Version kostenlos verfügbar, sodass ohne finanziellen Aufwand die volle Funktionsfähigkeit genutzt werden kann. Die kostenpflichtigen Visual-Studio-Versionen „Professional“ und „Enterprise“ bieten zwar bessere Unterstützung von UML-Diagrammen, Architekturüberprüfungen und Debug Werkzeugen, sind aber nicht zwingend erforderlich.

Die Android-spezifische Entwicklung zieht keine Kosten nach sich.

Für das Vermarkten und Veröffentlichen sowie für die Nutzung von Apples App-Analysen fallen jährliche Kosten in Höhe von circa 100 Euro pro Entwicklerteam an.

5.4 Dokumentation und Community

Beide Frameworks verfügen über eine umfangreiche und strukturierte Dokumentation, die auf den jeweiligen Websites zur Verfügung steht. Bei Cordova besteht allerdings das Problem, durch Nutzung von Suchmaschinen auf veraltete Dokumentationsseiten zu gelangen. Unter Xamarin kann es durch den enormen Umfang der Dokumentation und unterschiedlicher Layouts der Unterseiten vorkommen, das eigentliche Ziel aus den Augen zu verlieren.

Sollten sich Probleme ergeben, die nicht durch die Dokumentation abgedeckt sind, gibt es für beide Frameworks Forenbeiträge, die die meisten Probleme lösen. Für Xamarin existiert ein einschlägiges Forum, das über qualifizierte Kommentare von Xamarin-Entwicklern verfügt.

5.5 Umfang

Während Cordova Kompatibilität für aktuell fünf verschiedene mobile Zielsysteme anbietet (Android, iOS, WindowsPhone, Blackberry, Fire OS), bedient Xamarin nur die wichtigsten drei mobilen Plattformen (Android, iOS, WindowsPhone). Unterschiede bezüglich des Umfangs der Hardwarezugriffe konnten nicht festgestellt werden.

5.6 Einarbeitung

Unter Xamarin fällt die Einarbeitung nicht leicht aus. Der typische Zyklus einer App und die Handhabung der Interaktion zwischen einzelnen Klassen ist gerade am Anfang schwer nachzuvollziehen. Die vielen Möglichkeiten von Visual Studio wirken zu Beginn eher erschlagend als hilfreich. Die Programmiersprache C# ist je nach Vorkenntnissen auch nicht direkt begreifbar, es lässt sich jedoch die syntaktische Ähnlichkeit zu Java nutzen, um gerade am Anfang wirkungsvolle Ergebnisse zu erzielen.

Das Programmieren unter Cordova ist übersichtlich gehalten. In einem Ordner existieren die drei Dateien, die man zu Beginn benötigt. Wer schon einmal Websites entwickelt hat, der wird sich sofort zurechtfinden.
Natürlich gibt es Abweichungen zur Webprogrammierung, wie beispielsweise die Zuweisung von Funktionalität an den Zurück-Hardware-Button oder die Implementierung der Datenübertragung zwischen einzelnen Fenstern, doch sind diese Veränderungen und Neuerungen relativ leicht zu erlernen.
Was jedoch ein großes Problem darstellt, sind die weitestgehend fehlenden Fehlermeldungen, falls der Code nicht richtig ausgeführt werden konnte. Fehler werden schlichtweg ignoriert und somit bekommt man keine Auskunft über fehlerhafte Programmzeilen im Projekt.

5.7 Erstelldauer

Die Kompilationsdauer ist bei beiden Frameworks nicht gerade gering. Dieser Umstand ist aufgrund des Umfangs von Bibliotheksverknüpfungen zwar verständlich, jedoch wirkt er während einer intensiven Testphase oder Fehlersuche ermüdend. Kleine Änderungen im Code ziehen lange Kompilationsphasen nach sich. Besonders lange dauert die erste Kompilation.

Tabelle 5.2: Kompilationsdauern für Android im Vergleich

Kompilationsart	Apache's Cordova	Microsofts Xamarin
erstes Kompilieren (m:ss)	2:52	1:21
durchschnittliche Kompilationsdauer (m:ss)	0:27	0:15
durchschnittliche Deploymentdauer auf Endgerät/Emulator mit Kompilation (m:ss)	0:44	0:19

Wie man den Testdaten entnehmen kann, schneidet Xamarin erheblich besser ab, als Cordova. Insbesondere die durchschnittliche Deploymentdauer ist mit 19 Sekunden mehr als doppelt so schnell wie der Vergleichswert von Cordova (44 Sekunden).
Ärgerlich ist, dass man lange Wartezeiten in Kauf nehmen muss, bis die Auswirkungen der eigenen Arbeit begutachtet werden können. Zum Vergleich liegt das Starten einer kleinen Desktopanwendung in Visual Studio bei circa einer Sekunde. Dieser Unterschied ist also gewaltig und somit ein Nachteil der plattformübergreifenden Programmierung.

5.8 Programmverifizierung

Xamarin bietet Testverfahren direkt out-of-the-Box an. Es lassen sich UI- und Unit Tests einbinden. UI Tests schließen dabei Fehlerquellen in der Benutzeroberfläche aus, während Unit Tests die Richtigkeit von Methoden anhand vorgegebener Paare von Eingabewerten und erwarteten Ausgabewerten zeigen.

Für Cordova gibt es keine integrierten Testverfahren, es können jedoch externe Tools wie Calabash [Cal16] verwendet werden. Außerdem bieten sich für die programmierte Website Frameworks an, die eigentlich zur Verifizierung von Websites verwendet werden.

5.9 Ergebnisse

Mit beiden Frameworks ließ sich die angestrebte App realisieren. Unter Xamarin war es zudem möglich, durch die Nutzung eines HTML Pakets eine Website effizient auszulesen und entsprechende Daten in der Anwendung darzustellen.

Die Xamarin-App hat außerdem die Erzeugung einer ListView zugelassen, die sich nicht nur nativ anfühlt, sondern es auch ist. Im Gegensatz dazu konnte in Cordova nur eine scrollbare Seite erreicht werden, die über Barcodelinks in die Detailansicht wechselt.

Performance-Unterschiede konnten zwischen den beiden BarcodeScanner-Applikationen nicht festgestellt werden. Die Barcodeerkennung funktioniert ähnlich schnell.

5.10 Abschließende Bewertung

Die zu Beginn des Kapitels festgelegten Kriterien lassen sich nun mit Bewertungen für die beiden Frameworks Cordova und Xamarin auffüllen.

Tabelle 5.3: Bewertung der Frameworks auf einen Blick

Kriterien	Apache's Cordova	Microsofts Xamarin
Installierbarkeit und Nutzbarkeit	☆☆☆	☆☆☆
Installationsaufwand	mittel (viele Abhängigkeiten)	gering
Hauptprogramme Mac	Cordova, Xcode	Xamarin Studio, Xcode
Hauptprogramme Windows	Cordova & Editor / Visual Studio	Visual Studio
iOS Enticklung unter Windows	✓	✓
Test von iOS Versionen unter Windows	✗	✓(Mit Mac im Netzwerk)
Kosten	☆☆☆	☆☆☆
Fixkosten	kostenlos	kostenlos
Android-Extrakosten	kostenlos	kostenlos
iOS-Extrakosten	ca. 100 Euro pro Jahr	ca. 100 Euro pro Jahr
Dokumentation & Community	☆☆☆	☆☆☆
Dokumentation	✓	✓
Online-Referenzen	✓	✓
Umfang	☆☆☆	☆☆☆
umfangreicher Hardwarezugriff?	✓	✓
mobile Zielsystemabdeckung	Android, iOS, WindowsPhone, Blackberry,Fire OS	Android, iOS, WindowsPhone
Einarbeitung	☆☆☆	☆☆☆
Programmiersprachen	HTML, CSS & JavaScript	C# & AXML
Schwierigkeit der Einarbeitung	mittel	schwer
Erstelldauer	☆☆☆	☆☆☆
Kompilierdauer	lang	mittel
Deploymentdauer	lang	mittel
Programmverifizierung	☆☆☆	☆☆☆
integrierte Testverfahren	✗	✓(Unit, UI)
Ergebnisse	☆☆☆	☆☆☆
Performance	schnell	schnell
Look and Feel	ok	sehr gut
Insgesamt	☆☆☆	☆☆☆

Wie zu sehen ist, schneidet Xamarin durchweg besser ab als der Konkurrent Cordova. Gerade die unter Xamarin entwickelte Applikation konnte am besten optimiert werden. Auch das Testen der App hat schneller funktioniert, als unter Cordova.
Nicht zu vernachlässigen ist jedoch die größere Einarbeitungszeit, die in das Erlernen von Xamarin investiert werden musste. Dieser Punkt ist jedoch stark subjektiv, da generell das Vorwissen in den verwendeten Programmiersprachen darüber entscheidet, wie gut sich der Programmierer zu Beginn innerhalb eines Frameworks zurechtfindet.
Insbesondere, seitdem Xamarin kostenlos zur Verfügung steht, sind auch die finanziellen Gründe, die für Cordova sprachen, gegenstandslos geworden.

Kapitel 6

Fazit

6.1 Zusammenfassung

Im Rahmen dieser Seminararbeit wurden verschiedene Wege der Entwicklung für mobile Zielsysteme analysiert. Darüber hinaus konnten zwei unterschiedlich funktionierende Frameworks zur plattformübergreifenden Programmierung mobiler Applikationen erklärt und miteinander verglichen werden.
Als Ergebnis ist festzustellen, dass das einmalige Entwickeln für alle relevanten Zielplattformen ohne Verlust an Funktionalität möglich ist. Dabei sind Gemeinsamkeiten und Unterschiede in den Frameworks deutlich geworden, die anhand entscheidender Kriterien bewertet wurden. Als überzeugenderes Werkzeug ging das Framework Xamarin zur plattformübergreifenden Entwicklung *nativer* Apps vor dem Framework Cordova zur Erstellung plattformübergreifender *hybrider* Apps hervor. Die Integration von Xamarin in Visual Studio 2015 rundet die Benutzung des Frameworks zudem sehr gut ab und lässt alles „wie aus einem Guss“ wirken.
Diese Einordnung kann jedoch nicht als pauschale Empfehlung für Xamarin gesehen werden. Anzuraten ist, stets den Kenntnisstand des Teams zu evaluieren und darauf aufbauende Frameworks für die plattformübergreifende Programmierung mobiler Apps mit in die nähere Auswahl zu ziehen.
Der Markt hat sich in den letzten Jahren stark gewandelt und ist konsequent gewachsen, sodass es ständig neue Lösungen gibt, die möglicherweise besser zu der sich stellenden Aufgabe passen.

6.2 Ausblick

Auf Grundlage dieser Arbeit wird sich nun mit großer Wahrscheinlichkeit innerhalb der Zentralbibliothek des Forschungszentrums für die mobile Applikationsentwicklung mit Xamarin entschieden. Die Umstände, dass die Entwicklungsumgebung Visual Studio bereits zum Einsatz kommt und das Team fortgeschrittene Kenntnisse in C# besitzt, werden in den Entscheidungsprozess mit einfließen.
Die Zentralbibliothek bietet mehrere Einsatzgebiete mobiler Anwendungen. Ein Anwendungsfall könnte in Zukunft die Erweiterung der im Rahmen dieser Arbeit entwickelten Barcodescanner-Applikation sein. Im Zuge dessen könnte die Selbstausleihe implementiert werden, die in Verbindung mit einem integrierten Benutzerverwaltungssystem Buchausleihen konsistent in bereits existierenden Verwaltungssystemen verbucht.

Literaturverzeichnis

[Ado16] Adobe: *PhoneGap*. http://phonegap.com/, 2016. – [Online; Entnommen: 8-November-2016]

[Ang16] AngularJS: *AngularJS — Superheroic JavaScript MVW Framework*. https://angularjs.org, 2016. – [Online; Entnommen: 16-November-2016]

[Asi16] Asial: *Monaca*. https://monaca.io/de/, 2016. – [Online; Entnommen: 8-November-2016]

[Bil16] Bilgin, Jonathan Peppers; George Taskos; C.: *Xamarin: Cross-Platform Mobile Application Development*. Packt Publishing, 2016. – ISBN 978–1–78712–795–1. – Kapitel 3.1

[Bit16] Bitkom: *Anteil der Smartphone-Nutzer in Deutschland in den Jahren 2012 bis 2016* . https://de.statista.com/statistik/daten/studie/585883/umfrage/anteil-der-smartphone-nutzer-in-deutschland/, 2016. – [Online; Entnommen: 14-October-2016]

[Boo16] Bootstrap: *Bootstrap, The world's most popular mobile-first and responsive front-end framework*. http://getbootstrap.com/, 2016. – [Online; Entnommen: 16-November-2016]

[Cal16] Calabash: *Calabash*. http://calaba.sh/, 2016. – [Online; Entnommen: 15-Dezember-2016]

[Cor16] Cordova: *Apache Cordova*. https://cordova.apache.org/, 2016. – [Online; Entnommen: 20-Dezember-2016]

[Dev16] Developer, Amazon A.: *Amazon Appstore app is a single WebView*. https://twitter.com/AmazonAppDev/status/276374405066670082, 2016. – [Online; Entnommen: 21-November-2016]

[Ent16] Entwickler, App: *Unterschiede und Vergleich native Apps vs. Web Apps*. http://www.app-entwickler-verzeichnis.de/faq-app-entwicklung/11-definitionen/107-unterschiede-und-vergleich-native-apps-vs-web-apps, 2016. – [Online; Entnommen: 17-October-2016]

[For15] Forbes: *Global Operating System Market Share*. http://www.forbes.com/sites/greatspeculations/2015/07/14/reasons-why-nokia-may-be-planning-to-re-enter-the-smartphone-business/, 2015. – [Online; Entnommen: 14-October-2016]

[Fou16] Foundation, Apache S.: *Embed Cordova in native apps*. https://cordova.apache.org/docs/de/latest/guide/hybrid/webviews/, 2016. – [Online; Entnommen: 8-November-2016]

[Fre13] Frey, Andy: *There's More Than One Way to Build Mobile Apps (Part 1)*. http://blog.meltmedia.com/2013/05/theres-more-than-one-way-to-build-mobile-apps/, 2013. – [Online; Entnommen: 14-October-2016]

[HTM16] HTMLAgilityPack: *Html Agility Pack*. https://htmlagilitypack.codeplex.com/, 2016. – [Online; Entnommen: 5-Dezember-2016]

[Hyb16] HybridHeroes: *Die besten deutschsprachigen Hybrid Apps*. hybridheroes.de/blog/2015-11-12-die-besten-hybrid-apps-deutschlands/, 2016. – [Online; Entnommen: 21-November-2016]

[IF16] Ionic-Framework: *Build Amazing Native Apps and Progressive Web Apps with Ionic Framework and Angular*. http://ionicframework.com/, 2016. – [Online; Entnommen: 16-November-2016]

[Int16] Intel: *Intel XDK*. https://software.intel.com/en-us/intel-xdk, 2016. – [Online; Entnommen: 21-November-2016]

[ISO05] ISO/IEC: ISO/IEC 25000 - Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE. ISO/IEC, 2005. – Forschungsbericht

[KB14] Kraig Brockschmidt, Mike J.: *Die Struktur von Cordova-Plug-Ins, wenn eine Überbrückung erforderlich ist*. https://msdn.microsoft.com/de-de/magazine/dn879349.aspx, 2014. – [Online; Entnommen: 8-November-2016]

[Mon16] Mono: *The Mono Runtime — Mono* . http://www.mono-project.com/docs/advanced/runtime/, 2016. – [Online; Entnommen: 16-November-2016]

[Nat16] NativeScript: *Welcome to NativeScript*. http://docs.nativescript.org/, 2016. – [Online; Entnommen: 21-November-2016]

[NuG16] NuGet: *NuGet Package Manager*. https://www.nuget.org/, 2016. – [Online; Entnommen: 5-Dezember-2016]

[Ola16] Olanoff, Drew: *Mark Zuckerberg: Our Biggest Mistake Was Betting Too Much On HTML5*. https://techcrunch.com/2012/09/11/mark-zuckerberg-our-biggest-mistake-with-mobile-was-betting-too-much-on-htm 2016. – [Online; Entnommen: 21-November-2016]

[Pho16] PhoneGap, Adobe: *Ever wondered who is building amazing apps using PhoneGap?* http://phonegap.com/app/fanreact/, 2016. – [Online; Entnommen: 21-November-2016]

[Tou16] Touch, Sencha: *Cross-platform Mobile Web App Development Framework for HTML5 and JS — Sencha* . https://www.sencha.com/products/touch/#overview, 2016. – [Online; Entnommen: 16-November-2016]

[UI16] UI, Onsen: *Onsen UI 2: Beautiful HTML5 Hybrid Mobile App Framework and Tools* . https://onsen.io/, 2016. – [Online; Entnommen: 16-November-2016]

[Wik16a] Wikipedia: *Ahead-Of-Time-Compiler* . https://de.wikipedia.org/wiki/Ahead-of-time-Compiler, 2016. – [Online; Entnommen: 21-November-2016]

[Wik16b] Wikipedia: *C-Sharp*. https://de.wikipedia.org/wiki/C-Sharp, 2016. – [Online; Entnommen: 3-November-2016]

[Wik16c] Wikipedia: *Frameworks*. https://de.wikipedia.org/wiki/Framework, 2016. – [Online; Entnommen: 17-October-2016]

[Wik16d] Wikipedia: *Just-In-Time-Kompilierung* . https://de.wikipedia.org/wiki/Just-in-time-Kompilierung, 2016. – [Online; Entnommen: 21-November-2016]

[Wik16e] Wikipedia: *Sandbox*. https://de.wikipedia.org/wiki/Sandbox, 2016. – [Online; Entnommen: 17-October-2016]

[Xam16a] Xamarin: *Build native UIs for iOS, Android and Windows from a single, shared C# codebase*. https://www.xamarin.com/forms, 2016. – [Online; Entnommen: 9-November-2016]

[Xam16b] Xamarin: *iOS Architecture Xamarin*. https://developer.xamarin.com/guides/ios/under_the_hood/architecture/, 2016. – [Online; Entnommen: 21-November-2016]

[Xam16c] Xamarin: *Mobile App Development App Creation Software - Xamari*. https://www.xamarin.com/, 2016. – [Online; Entnommen: 20-Dezember-2016]

[Xam16d] Xamarin: *Portable Class Libraries*. https://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/, 2016. – [Online; Entnommen: 9-November-2016]

[Xam16e] Xamarin: *Xamarin Android Architecture*. https://developer.xamarin.com/guides/android/under_the_hood/architecture/, 2016. – [Online; Entnommen: 9-November-2016]

[ZXi16] ZXing: *ZXing Zebra Crossing*. https://github.com/zxing/zxing, 2016. – [Online; Entnommen: 5-Dezember-2016]