Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Eugen Gez

Entwicklung einer multiplattformfähigen Reiseführeranwendung für Smartphones

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Eugen Gez

Entwicklung einer multiplattformfähigen Reiseführeranwendung für Smartphones

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Birgit Wendholt
Zweitgutachter: Prof. Dr.-Ing. Olaf Zukunft

Eingereicht am: 19. September 2014

Eugen Gez

Thema der Arbeit
Entwicklung einer multiplattformfähigen Reiseführeranwendung für Smartphones

Stichworte
Cordova / PhoneGap, multiplattformfähige Anwendungen, Smartphone, Reiseführer

Kurzzusammenfassung
Dieses Dokument befasst sich mit der Erarbeitung eines Entwurfs und der anschließenden Entwicklung des Prototyps einer multiplattformfähigen Reiseführeranwendung mithilfe des Frameworks Cordova / PhoneGap.

Eugen Gez

Title of the paper
Developing of multiplatform-capable travel guide application for smartphones

Keywords
Cordova / PhoneGap, cross-platform applications, smartphone, travel guide

Abstract
This document deals with the design and development of the prototype of a multi-platform travel guide application using the Cordova / PhoneGap framework.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Heute besitzt fast jeder ein Smartphone in der Tasche. Sie sind so verbreitet, dass sich für jeden eins nach Budget, Geschmack und Maß findet. Die Smartphones verschiedener Hersteller verfügen über Marktanteile die sich ständig und sehr schnell ändern. (Sieh Abb. 1.1)

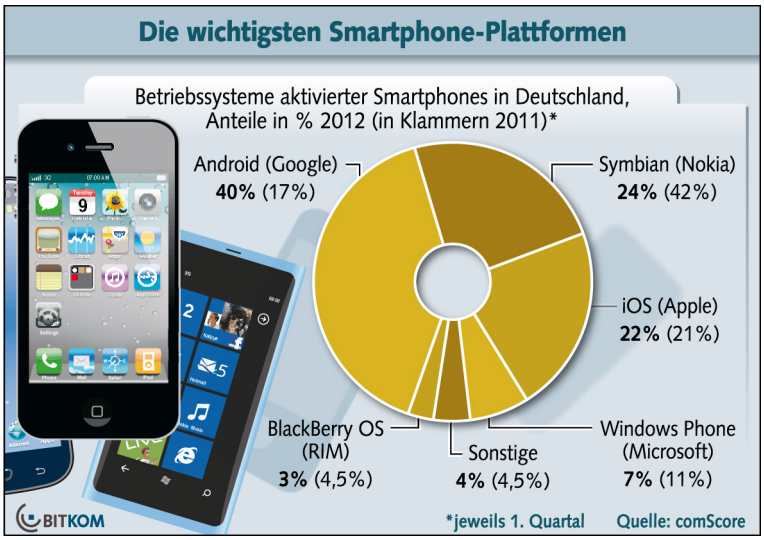


Abbildung 1.1: Marktanteilveränderung 2011-2012 (BITKOM, 2012)

Sie besitzen schon jetzt Kapazitäten die vergleichbar sind mit den Rechnern von gestern. Ein zwei-Kern-Prozessor mit jeweils 1 GHz Taktrate in einem Handy ist heute keine Utopie mehr. Und wo es Rechenpower gibt, da wird sie auch genutzt.

Eine sehr hohe Verbreitung und die Vielfalt technischer Ausstattung dieser Geräte schafft eine hohe Nachfrage für Apps, die zur Lösung verschiedenster Aufgaben verwendet werden und somit das mobile Telefon zu einem Werkzeug oder sogar zu einem Spielzeug macht.

Um dieser Nachfrage gerecht zu bleiben, müssen viele Entwickler beschäftigt werden. Einige beschränken sich aus verschiedenen Gründen auf einzelne Plattformen, die anderen müssen mit einem sehr hohen Aufwand für mehrere Plattformen gleichzeitig entwickeln. Die letzten haben dabei das Problem ihre Software für verschiedene Plattformen ständig anpassen oder für neue komplett neu entwickeln zu müssen.

Früher, um diesem Problem entgegen zu wirken, haben sich die Handy- bzw. Betriebssystemhersteller darauf geeinigt gemeinsame Standards einzusetzen. Sie haben versucht von der Hardware und vor allem von der Plattform insgesamt zu abstrahieren. Als Beispiel schwebt einem der Einsatz von J2ME in den klassischen Handys vor. In den heutigen Smartphones wird es nicht mehr eingesetzt, sodass jeder Hersteller seinen eigenen Weg geht und die Entwickler weiterhin mit dem Problemen der Multi-Plattform Entwicklung konfrontiert sind.

Aufgrund dessen, dass die Smartphones sehr verbreitet und immer dabei sind, dadurch dass sie Kameras, Internetverbindung und Ortungsmöglichkeiten haben, eignen sie sich hervorragend für eine Reiseführeranwendung, eine Anwendung mit der man eigene Reiseführer selbst aufnehmen und zusammenstellen kann. Diese Anwendung sollte multiplattformfähig sein, damit sie auf einem möglichst hohen Anteil der Geräte ausgeführt werden kann.

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht darin eine Anwendung zu entwerfen und zu entwickeln mit deren Hilfe man auf einem Smartphone neben üblicher Reiseführung eigene Reiseführer aufnehmen und zusammenstellen kann, indem man die technischen Möglichkeiten des Smartphones nutzt um interessante Orte bzw. Objekte aufzunehmen, zu beschreiben, deren Position zu bestimmen und sie ggf. als Tour oder einzeln mit anderen zu teilen.

Außerdem ist eine möglichst einfache Lösung zu finden oder zu erarbeiten mit deren Hilfe die Anwendung auf verschiedenen Plattformen eingesetzt werden kann ohne sie aufgrund derer Besonderheiten aufwendig anpassen oder sogar neu entwickeln zu müssen.

Darüber hinaus ist anhand eines Prototyps zu untersuchen, welche Vor- und Nachteile der Einsatz eines Cross-Plattform Frameworks für die Entwicklung mit sich bringt. Es gilt dabei den Aufwand für die Implementierung sowie die Performance zu beurteilen. Anhand geeigneter Kriterien soll die gewählte Technologie mit Alternativen verglichen werden.

1.3 Gliederung

Im Kapitel 1 *Einleitung* wird die Motivation vorgestellt und das Ziel dieser Arbeit definiert.

Anschließend beschäftigt sich das Kapitel 2 *Grundlagen und vergleichbare Arbeiten* mit den grundlegenden Ansätzen zum Entwickeln für mobile Plattformen und vergleicht einige ähnliche Anwendungen. In diesem Kapitel werden mehrere Ansätze miteinander verglichen und eine Auswahl getroffen.

Das Kapitel 3 beschäftigt sich mit der *Analyse*. Hier werden funktionale und nicht-funktionale Anforderungen an die Anwendung erarbeitet und anschließend zusammengefasst. Die Funktionalität wird anhand eines beispielhaften Anwendungsszenario und der Anwendungsfälle vorgestellt.

Im Kapitel 4 *Design* werden zunächst technische Voraussetzungen definiert. Der sich anschließende Entwurf wird anhand der Komponenten-, Klassen- und Sequenzdiagramme vorgestellt. In der sich anschließenden prototypischen Realisierung wird auf einige Lösungskonzepte und die während der Entwicklung aufgetretenen Probleme und deren Lösungen eingegangen.

Das Kapitel 5 fasst die Arbeit zusammen und gibt einen Ausblick auf zukünftige Entwicklungen.

2 Grundlagen und vergleichbare Arbeiten

Dieses Kapitel beschäftigt sich mit den Grundlagen und einigen Technologien, die im Rahmen dieser Arbeit verwendet werden. Anschließend werden Parallelen zu anderen ähnlichen Konzepten gezogen und einige vergleichbare Anwendungen vorgestellt und erläutert.

2.1 Grundlagen

In diesem Kapitel werden einige Ansätze miteinander verglichen, deren Stärken und Schwächen inklusive ihrer Auswirkungen auf die Entwicklung und die Laufzeit der Anwendungen analysiert und anhand dessen eine Auswahl getroffen.

Es gibt derzeit mehrere relevante Ansetze, um Anwendungen für mobile Geräte zu entwickeln. Diese können wie folgt klassifiziert werden.

Die nachfolgende Klassifizierung wurde aus folgender Quelle Erarbeitet: (Heitkötter u. a., 2013)

Nativ: Es sind Anwendungen, die in den jeweiligen plattformspezifischen Programmiersprachen geschrieben sind und für jede Plattform separat entwickelt werden müssen. Der Entwickler muss sich dabei mit jeder diesen Programmiersprachen und Besonderheiten der einzelnen Plattformen auseinander setzen. Solchen Anwendungen stehen alle plattformspezifischen Schnittstellen und Features zur Verfügung.

VM-interpretiert: Diese Anwendungen werden in einer bestimmten Programmiersprache geschrieben. Sie laufen in einer virtuellen Maschine, um von der Plattform komplett zu abstrahieren. Der Entwickler hat sich dabei nur auf die einzelne Programmiersprache zu konzentrieren und kann von den Besonderheiten der einzelnen Plattformen abstrahieren. Die virtuelle Maschine selbst ist jedoch eine native Anwendung und wird ebenfalls für jede Plattform separat entwickelt. Sie wird üblicherweise in die Anwendung mit eingebettet und stellt eine sogenannte *self-contained* Laufzeitumgebung dar. Aufgrund der kompletten Abstraktion haben solche Anwendungen nur einen limitierten Zugriff auf die plattformspezifischen Schnittstellen und Features.

Web App: Es sind reine Webanwendungen, die üblicherweise als Webseiten implementiert und im Browser ausgeführt werden. Es ist der einzige Ansatz, bei dem die Anwendungen weder kompiliert noch installiert werden müssen. Sie werden einfach über das Internet geladen und sofort ausgeführt. Der Entwickler muss sich nur mit der Webentwicklung beschäftigen. Die Webanwendungen haben keinen Zugriff auf die plattformspezifischen Schnittstellen und Features wie z. B. Kamera oder Mikrofon. Ein natives Look and Feel ist bei solchen Anwendungen nicht möglich.

Hybrid: Diese Anwendungen vereinen zwei der hier beschriebenen Ansätze in sich. Sie stellen native Anwendungen dar, führen ihre Geschäftslogik jedoch in Form einer Webanwendung innerhalb ihrer WebView aus und stellen ihr erweiterte Schnittstellen zur Verfügung. Dadurch haben solche Anwendungen einen limitierten Zugriff auf die plattformspezifischen Schnittstellen und Features. Der Entwickler muss sich auch bei diesem Ansatz fast nur mit der Webentwicklung beschäftigen. Ein natives Look and Feel ist bei solchen Anwendungen ebenfalls nicht möglich.

Im Rahmen dieser Bachelorarbeit wird nach einer Lösung gesucht, mit der eine multiplattformfähige Anwendung mit möglichst wenigem Aufwand gebaut werden kann. Aus diesem Grund sind die Ansätze *Nativ* und *Web App* nicht verwendbar, da sie entweder nicht multiplattformfähig sind oder keinen Zugriff auf die nativen Schnittstellen erlauben. Sie werden nicht mehr weiter im Detail verfolgt.

Die zwei anderen Ansätze *Hybrid* und *VM-interpretiert* stellen ein Interesse dar und werden nachfolgend anhand einiger konkreten Frameworks miteinander verglichen.

2.1.1 Cordova / PhoneGap vs Titanium Mobile

Cordova / PhoneGap und Titanium sind ähnlich in der Hinsicht, dass sie dem selben Zweck dienen, es den Entwicklern möglich zu machen von der Plattform zu abstrahieren und sich auf die Entwicklung eigentlicher Anwendung zu konzentrieren. Die beiden nutzen JavaScript, um die Geschäftslogik der Anwendungen auszudrücken, verfolgen jedoch verschiedene Ziele und haben völlig unterschiedliche Ansätze.

Wenn Titanium das Ziel verfolgt mit einer gewissen Abstraktion von den jeweiligen Plattformen native Anwendungen entwickeln zu können, so bewegt sich Cordova / PhoneGap in die Richtung hybride Webanwendungen entwickeln und diese als native Apps ausführen zu können. Jeder dieser Ansätze hat seine eigenen Stärken und Schwächen.

Im Folgenden wird die Funktionsweise dieser beiden Technologien etwas detaillierter erläutert und miteinander verglichen.

Cordova / PhoneGap

Cordova / PhoneGap ist ein Framework, welches eine Möglichkeit bietet *hybride Webanwendungen* für die gängigsten Plattformen mithilfe von JavaScript, HTML5 und CSS zu bauen. Um von der Plattform abstrahieren zu können, wird die Laufzeitumgebung des Browsers benutzt. Das Framework stellt unter Anderem eine erweiterte Schnittstelle bereit, um auf die Geräte der Smartphones aus JavaScript zugreifen zu können. Aus der Entwicklersicht ist es nichts anderes, als eine ganz normale JavaScript-Bibliothek, die je nach Funktionalität in Module aufgeteilt ist. Diese werden in Form von Plugins nach Bedarf zugeschaltet. Im übrigen unterscheidet sich die Entwicklung selbst nicht von der Entwicklung einer gewöhnlichen Webseite oder einer Webanwendung.

Eine Ausnahme ist jedoch die Tatsache, dass die Anwendung noch kompiliert werden muss, um auf den Geräten als eine App ausgeführt werden zu können. Dafür stellt das Framework für jede unterstützte Plattform eine sogenannte Wrapper-App oder Hülle bereit.

Eine Wrapper-App ist eine Vorlage, die den nativen Teil der Anwendung darstellt und mithilfe eines vom Framework bereitgestellten command line interface automatisch erzeugt und konfiguriert wird. Sie ist in der jeweiligen nativen Programmiersprache der Zielplattform geschrieben und kann ohne weiteres mit einer entsprechenden SDK gebaut werden. Diese Wrapper-App realisiert nativ die in JavaScript bereitgestellten Schnittstellen und erledigt somit die Zugriffe auf die entsprechenden Schnittstellen der Plattform.

Beim Kompilieren wird die vom Entwickler geschriebene Webanwendung als Assets bzw. Ressourcen eingebettet, sodass sie beim Ausführen einfach in einer WebView, einem Bestandteil des nativen UI-Frameworks, geladen und auf diese Art ausgeführt wird.

Da es sich hierbei um eine Webtechnologie handelt, ist Cordova / PhoneGap sehr flexibel hinsichtlich der benutzten Bibliotheken und anderer Frameworks. Außerdem sind solche Anwendungen sehr leicht portierbar. Dieser Ansatz wirkt sich jedoch negativ auf die Performance und das native Look and Feel der Anwendungen aus.

Das Funktionsprinzip von Cordova / PhoneGap basiert darauf, dass die native Anwendung, der Browser, in der Lage ist mit dem in seiner WebView ausgeführten JavaScript-Code asynchron Nachrichten auszutauschen. Diese Technologie wird üblicherweise als *bridge* bezeichnet. (Whinnery, 2014)

Die Kernfunktionalität des Frameworks besteht bis auf einiges darin, sowohl in JavaScript als auch nativ eine einheitliche Basisschnittstelle zur Kommunikation mit dem jeweils anderen Teil des Frameworks zur Verfügung zu stellen. Die eigentliche Funktionalität wird erst durch die entsprechenden Plugins hinzugefügt.

Jedes Plugin, wie auch der Kern des Frameworks, besteht ebenfalls aus einem JavaScript- und einem nativen Teil. Ein Plugin stellt in JavaScript seine Schnittstelle zur Verfügung und muss sie nativ für alle unterstützten Plattformen einzeln realisieren. Diese Schnittstelle benutzt die von Cordova / PhoneGap bereitgestellte JavaScript-Bibliothek, um mit dem entsprechenden nativen Teil kommunizieren zu können. Die native Realisierung erledigt anschließend die Zugriffe auf die entsprechenden Schnittstellen der Plattform.

Folgendes Komponentendiagramm sollte einen groben Überblick über den Aufbau einer Cordova / PhoneGap Anwendung zur Laufzeit verschaffen.

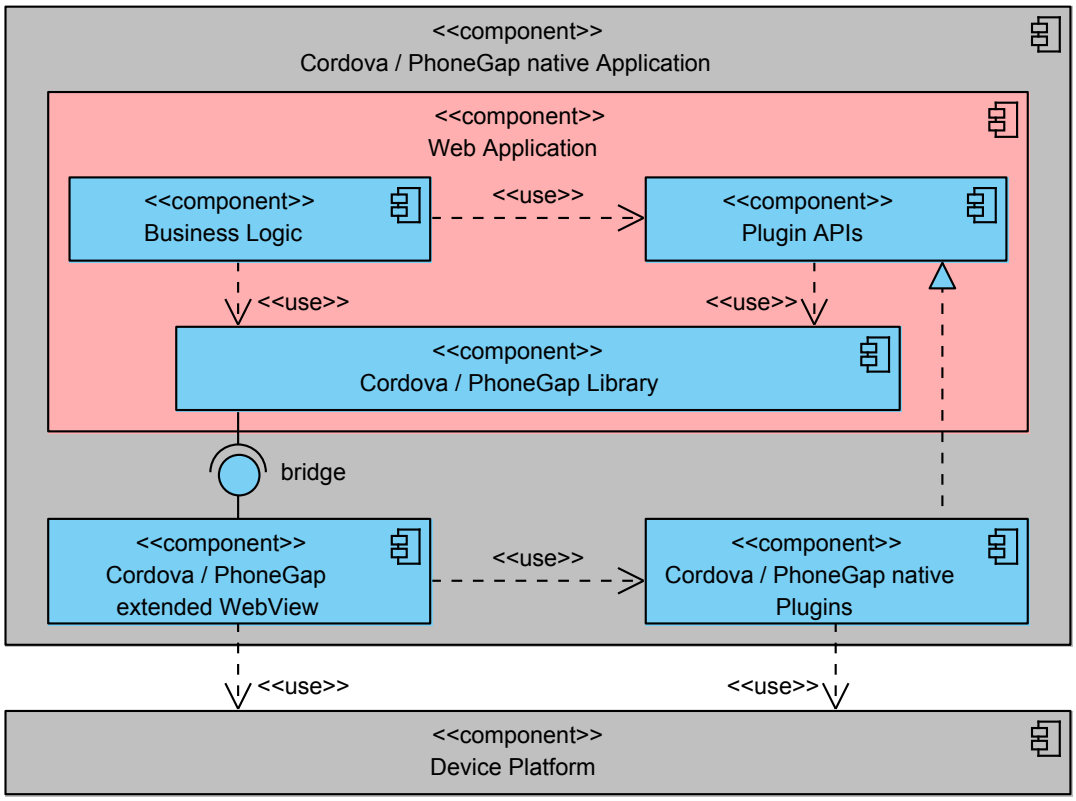


Abbildung 2.1: Aufbau - Cordova / PhoneGap

Wie das Komponentendiagramm 2.1 zeigt kann die Business Logic mithilfe der Plugin APIs und der Cordova / PhoneGap Library, die in JavaScript geschrieben sind, über die bridge mit der erweiterten WebView kommunizieren. Diese kann die Anfragen direkt an die Schnittstellen der Geräteplattform oder an die nativen Teile der Plugins delegieren.

Titanium Mobile

Titanium, im Gegensatz zu Cordova / PhoneGap, bietet eine Möglichkeit mithilfe einer plattformübergreifend einheitlichen Schnittstelle *native bzw. VM-interpretierte Anwendungen* in JavaScript zu entwickeln. Um eine Abstraktionsebene herzustellen, wird eine virtuelle Maschine benutzt. Das Framework stellt dabei neben einer unifizierten und von mehreren Plattformen gemeinsam unterstützten Kernschnittstelle auch einige plattformspezifischen Schnittstellen zur Verfügung. (Powell, 2012) Für den Entwickler ist Titanium ebenfalls eine JavaScript-Bibliothek. Es wird dabei allein in JavaScript entwickelt, ohne dass er sich mit anderen Technologien wie HTML oder CSS beschäftigen muss.

Das Framework stellt ebenfalls eine plattformspezifische Implementierung zur Verfügung. Die Schnittstellen und der JavaScript-Interpreter sind für jede unterstützte Plattform in der jeweiligen nativen Programmiersprache implementiert. Diese stellen eine virtuelle Maschine dar, die beim Kompilieren der ebenfalls vom Framework bereitgestellten Vorlage der Anwendung mit eingebettet wird. Die in JavaScript geschriebene Geschäftslogik wird dabei nicht kompiliert, sondern in Form eines nativen String-Objekts eingebettet und erst zur Laufzeit vom Interpreter evaluiert.

In mehreren Quellen werden Titanium-Anwendungen entweder als *nativ* (Whinnery, 2014) oder als *self-contained VM-interpretiert* (Heitkötter u. a., 2013) klassifiziert. Dabei wird im ersten Fall nicht zwischen *nativ* und *VM-interpretiert* unterschieden. Obwohl die Geschäftslogik solcher Anwendungen in JavaScript geschrieben ist und erst zur Laufzeit evaluiert wird, arbeiten sie mithilfe der sogenannten Proxy-Objekte direkt auf den nativen Objekten und somit in der nativen Laufzeitumgebung.

Aufgrund dessen, dass Titanium neben einer einheitlichen Schnittstelle noch einige plattformspezifischen zur Verfügung stellt, sind die Anwendungen nicht ohne weiteres portierbar. Daher müssen sich die Entwickler etwas mit den Besonderheiten der einzelnen Zielplattformen auseinandersetzen. Außerdem ist Titanium hinsichtlich der benutzten Bibliotheken nicht so flexibel wie Cordova / PhoneGap, bringt jedoch nahezu volle Performance und ein natives Look and Feel mit sich.

Das Funktionsprinzip von Titanium besteht darin, dass der JavaScript-Interpreter und die Schnittstelle zusammen eine virtuelle Maschine bilden, welche die in JavaScript geschriebene Geschäftslogik zur Laufzeit evaluiert. Für jedes neu erstellte Objekt wie Button oder Socket in der nativen Umgebung wird zur Laufzeit ein Proxy-Objekt in der JavaScript-Umgebung erstellt. Jedes dieser Proxy-Objekte stellt die Schnittstelle des mit ihm gepaarten nativen Objektes in JavaScript zur Verfügung, sodass die Geschäftslogik mithilfe dieser Proxy-Objekte direkt auf den Objekten der nativen Umgebung arbeitet.

Wenn man in JavaScript einen neuen Button erstellt, so wird eine native Methode aufgerufen, die in der nativen Umgebung mithilfe des plattformspezifischen UI-Frameworks einen neuen Button und in der JavaScript-Umgebung ein neues Proxy-Objekt für diesen Button erstellt. (Whinnery, 2014) Dieser Konzept könnte man mit Java RMI vergleichen.

Folgendes Komponentendiagramm sollte einen groben Überblick über den Aufbau einer Titanium-Anwendung zur Laufzeit verschaffen.

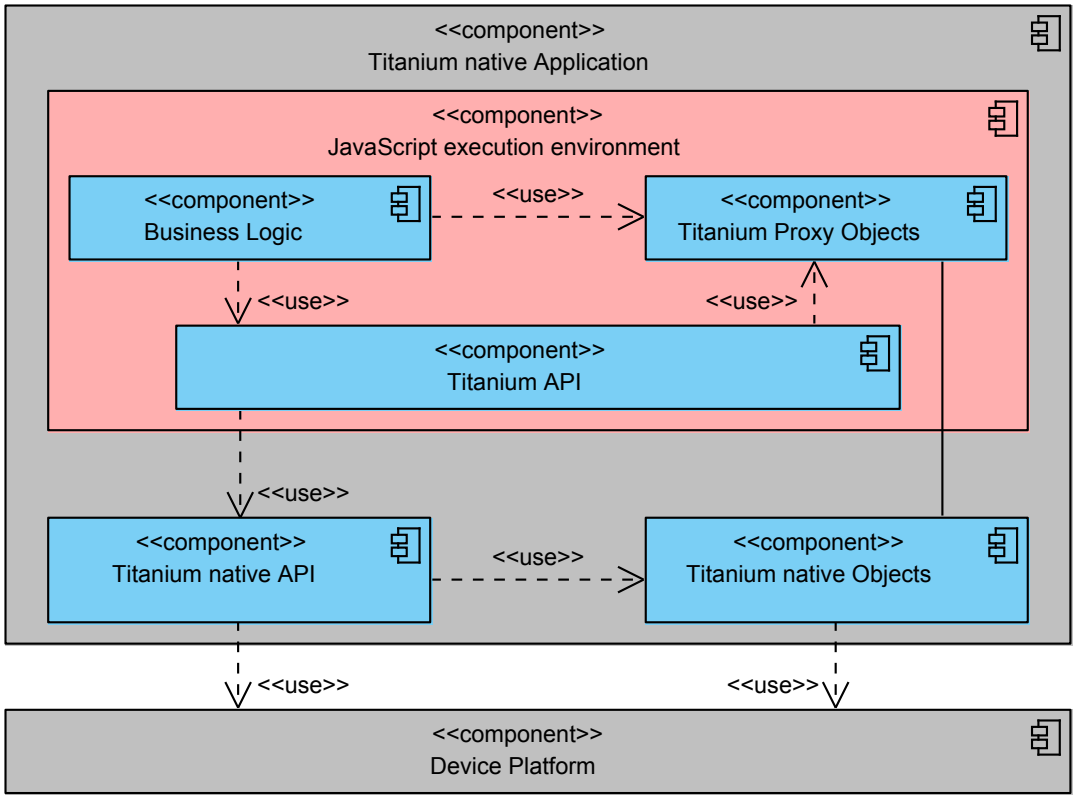


Abbildung 2.2: Aufbau - Titanium

Wie das Komponentendiagramm 2.2 zeigt arbeitet die in JavaScript geschriebene Business Logic mit der Titanium API und den Proxy-Objekten, die ebenfalls ein Teil dieser API sind. Jeder Aufruf an so einem Proxy-Objekt wird an das entsprechende native Objekt delegiert, sodass die Geschäftslogik mithilfe dieser Proxy-Objekte immer auf den nativen Objekten arbeitet. Die Aufrufe an der Titanium API werden an deren nativen Teil entsprechend weitergeleitet.

Fazit

Cordova / PhoneGap benutzt eine erweiterte WebView als Laufzeitumgebung, um von der Plattform zu abstrahieren. Dies führt dazu, dass die Performance der Anwendungen ggf. nachlässt und das native Look and Feel nicht mehr möglich ist.

Dieser Ansatz ist jedoch sehr flexibel hinsichtlich der Benutzung anderer Frameworks und Bibliotheken. Außerdem ist der Entwicklungsaufwand relativ gering und die Anwendungen sind hoch portierbar.

Titanium liefert eine eigene virtuelle Maschine, um von der Plattform zu abstrahieren. Dennoch ist nur eine Untermenge der vom Framework angebotenen Schnittstellen allgemeingültig. Dies führt dazu, dass die Anwendungen nicht ohne weiteres portierbar sind und die Entwickler sich trotzdem mit den einzelnen Besonderheiten der Plattformen auseinander setzen müssen.

Die virtuelle Maschine des Frameworks ist so konzipiert, dass die Geschäftslogik in die native Umgebung eingreifen und auf diese Art, beschränkt durch den Schnittstellenumfang, direkt mit der Plattform arbeiten kann. Dadurch kann der Benutzer nahezu volle Performance und das native Look and Feel genießen.

Der Umfang der Schnittstellen der beiden Frameworks ist annähernd gleich. Somit ergibt sich daraus kein Vor- bzw. Nachteil für das eine oder das andere Framework.

Aufgrund dessen, dass im Rahmen dieser Arbeit nach einer Lösung gesucht wird, mit der multiplattformfähige Anwendungen mit wenigem Aufwand gebaut werden können, fällt die Entscheidung wegen der hohen Portierbarkeit und des niedrigen Entwicklungsaufwandes zugunsten von Cordova / PhoneGap. Titanium Mobile wird nicht mehr weiter verfolgt.

Im nächsten Kapitel werden einige ähnliche Anwendungen vorgestellt und erläutert.

2.2 Vergleichbare Arbeiten

Dieses Kapitel beschäftigt sich mit vergleichbaren Arbeiten. Hier werden ähnliche Anwendungen vorgestellt und kurz erläutert.

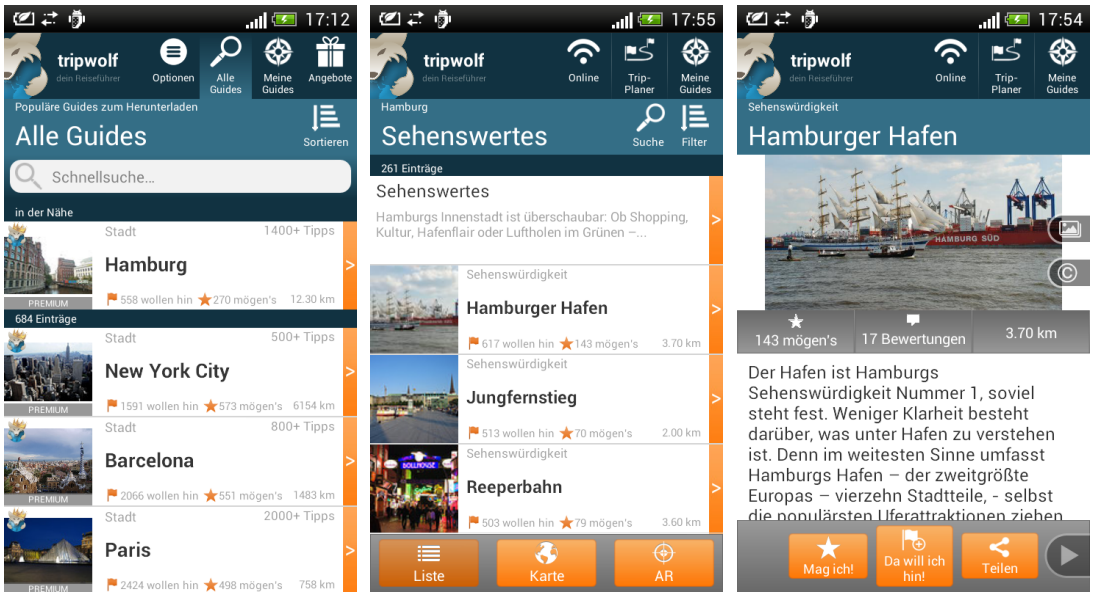
2.2.1 tripwolf

tripwolf ist eine native Anwendung für Android. Sie ermöglicht vordefinierte Reiseführer noch vor dem Reiseantritt inklusive der Karten, Verkehrspläne und aller Sehenswürdigkeiten herunter zu laden und ganz individuelle Touren zu planen.

Die Reiseführer stehen zwar nicht kostenlos zur Verfügung, können jedoch einige Zeit getestet werden. Ein Reiseführer für Hamburg kostet z. B. 4,49 € und kann direkt in der Anwendung gekauft werden. Die Informationen scheinen sehr sorgfältig vorbereitet zu sein, sodass z. B. neben Eckdaten wie die Einwohnerzahl oder die Fläche der Stadt auch eine kurze Beschreibung vorhanden ist. Außerdem scheinen die meisten POIs auch recht gut beschrieben zu sein.

Darüber hinaus bietet die Anwendung noch eine Community-Anbindung mit einem Bewertungssystem mit deren Hilfe die Kunden die Sehenswürdigkeiten bewerten und Reisetipps geben können, sodass anhand dessen auch noch einige Empfehlungen von den Betreibern gemacht werden können. (Tripwolf GmbH, 2014)

Folgende Abbildungen sind während der Evaluierungsphase der Anwendung entstanden und sollten einen kurzen Überblick verschaffen.



(a) Startbildschirm (b) Sehenswürdigkeiten (c) Detailansicht

Abbildung 2.3: Abbildungsreihe - tripwolf

In der Abbildung 2.3 sind drei Screenshots abgebildet. Auf dem Screenshot (a) ist der Startbildschirm der Anwendung zu sehen. Hier werden alle zur Verfügung stehenden Reiseführer aufgelistet. Unter *Meine Guides* sind die gekauften zu finden.

Auf dem Screenshot (b) sind die Sehenswürdigkeiten des gewählten Reiseführers aufgelistet. Hier kann der Benutzer die einzelnen POIs auf drei Arten darstellen lassen: Auf einer Karte, in einer Liste und durch den Kamerasucher als augmented reality.

Der Screenshot (c) zeigt einen POI in der Detailansicht. In dieser Ansicht kann der Benutzer die Beschreibung lesen, sich Fotos und Bewertungen des POIs anschauen, ihn selbst bewerten oder mit anderen teilen.

2.2.2 Field Trip

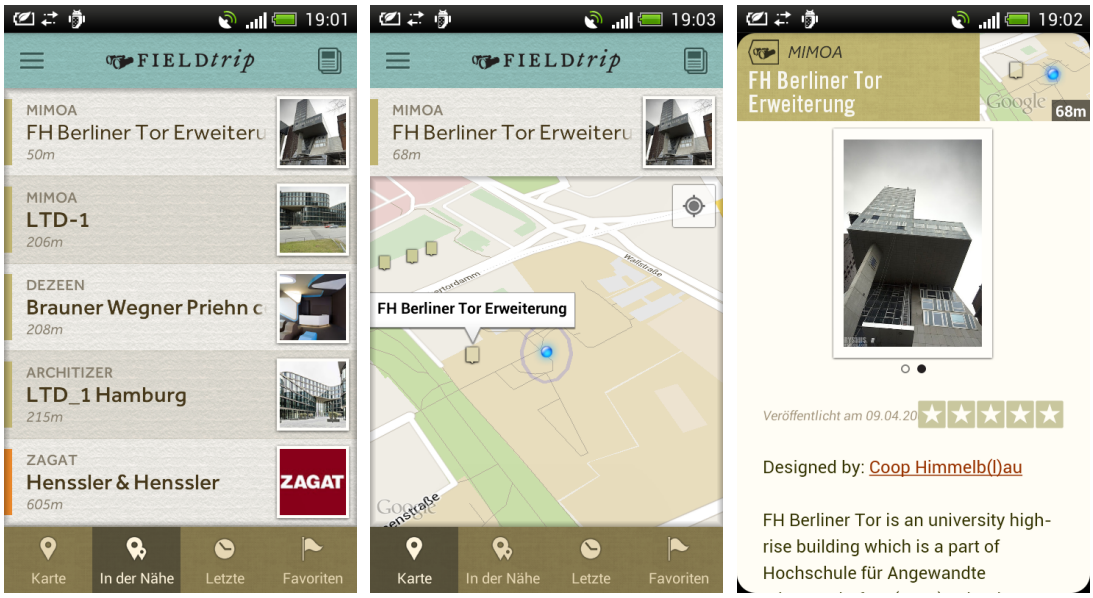
Field Trip ist ebenfalls eine native Anwendung für Android. Sie ist jedoch für diejenigen gedacht, die sich gern überraschen lassen. Daher gibt es hier keinen Planer oder vorgefertigten Reiseführer.

Field Trip ist so konzipiert, dass der Benutzer auf seiner Reise gelegentlich mit den Informationen über die sich in der Nähe befindlichen Sehenswürdigkeiten oder anderen POIs einfach spontan überrascht wird. Dabei läuft die Anwendung im Hintergrund und meldet

einige interessante Orte in der Nähe aus den zuvor ausgewählten Kategorien. Die Auswahl ist einstellbar, sodass es möglich ist die POIs von der einen oder der anderen Kategorie mehr oder weniger anzeigen zu lassen.

Die Kategorien sind recht vielfältig. Hier gibt es sowohl Einkaufsmöglichkeiten als auch Museen oder andere Orte, die mit der Geschichte oder Architektur im Zusammenhang stehen. Die Beschreibungen zu den POIs werden von verschiedenen Projekten oder Experten bereitgestellt. Außerdem wird an die entsprechenden Artikel verwiesen, sodass es Möglichkeiten für weitere Recherchen zum Thema gibt. Ein Kundenbewertungssystem ist ebenfalls vorhanden. (NianticLabs@Google, 2014)

Folgende Abbildungen sind während der Evaluierungsphase der Anwendung entstanden und sollten einen kurzen Überblick verschaffen.



(a) Startbildschirm (b) Kartenansicht (c) Detailansicht

Abbildung 2.4: Abbildungsreihe - Field Trip

In der Abbildung 2.4 sind ebenfalls drei Screenshots abgebildet. Der Screenshot (a) zeigt den Startbildschirm der Anwendung, auf dem POIs aufgelistet sind, die sich in der Nähe befinden. Diese Liste ist nach Entfernung sortiert, sodass die am nächsten liegenden POIs oben sind.

Auf dem Screenshot (b) ist eine Kartenansicht abgebildet. Hier sieht der Benutzer seinen aktuellen Standort und die umliegenden POIs. Oben wird eine Zusammenfassung zum ausgewählten POI angezeigt.

Der Screenshot (c) zeigt einen POI in der Detailansicht. Hier kann der Benutzer ebenfalls eine kurze Beschreibung lesen, sich Abbildungen dazu anschauen, den POI bewerten oder mit den anderen Teilen. Es wird außerdem auf die entsprechenden Artikel verwiesen, sodass der Benutzer ggf. auch mehr zum Thema erfahren kann.

2.3 Zusammenfassung

In diesem Kapitel wurden einige Grundlagen vorgestellt, Technologien miteinander verglichen und ähnliche Anwendungen aufgeführt. Die Entscheidung ist aufgrund der hohen Portierbarkeit und des relativ wenigen Aufwandes zugunsten des Frameworks Cordova / PhoneGap gefallen. Das nachfolgende Kapitel beschäftigt sich mit der Analyse der Anforderungen an die Anwendung.

3 Analyse

In diesem Kapitel geht es um die Analyse der Anforderungen an die Anwendung und deren Funktionalität. Im ersten Abschnitt werden funktionale Anforderungen festgestellt und mithilfe der Anwendungsfälle sowie Anwendungsfalldiagramme dargelegt. Der zweite Abschnitt befasst sich mit den nicht-funktionalen Anforderungen. Die Zusammenfassung führt die wesentlichen Aspekte dieses Kapitels nochmal auf und leitet das nächste Kapitel ein.

3.1 Funktionale Anforderungen

3.1.1 Funktionalität

Diese Anwendung stellt einen persönlichen Reiseführer dar. Sie soll neben üblicher Navigation zwischen Sehenswürdigkeiten außerdem noch eine Funktionalität bieten mit der auch eigene, individuelle Reiseführer, im Folgenden Touren genannt, erstellt und ggf. sogar mit anderen geteilt werden können. Dabei nutzt Sie verschiedene Sensoren und andere technische Ausstattung der Smartphones, um z. B. Fotos oder Videos von Sehenswürdigkeiten aufzunehmen, deren Standort und den des Benutzers zu bestimmen, Touren oder einzelne *Points of Interest*, nachfolgend POIs, mit anderen zu teilen oder abzurufen.

Etwas näher und detaillierter soll die Funktionalität am folgenden beispielhaften Anwendungsszenario erläutert werden:

> *Erika Mustermann studiert Angewandte Informatik an der HAW-Hamburg. Sie hat endlich ihren wohlverdienten Urlaub und geht auf Reisen. Auf ihrem Smartphone ist eine nagelneue Software mit der sie ihren eigenen persönlichen Reiseführer erstellen und mit anderen teilen möchte. Es geht nach Frankreich.*
>
> *Im Zielland legt sie als Erstes eine neue Tour an, beschriftet diese, macht ein Paar kurze Notizen als Beschreibung dazu und nimmt Kurs auf den Eiffelturm.*
>
> *Sobald sie am Ziel angekommen ist und sich eine gute Perspektive gefunden hat, erfasst sie eine neue Sehenswürdigkeit, die sie ebenfalls beschriftet und ggf. kurz beschreibt. Während dessen hat die Anwendung mittels GPS den aktuellen Standort*

von Erika ermittelt und die entsprechende Eigenschaft des POIs bereits vorbelegt. Erika schießt jetzt nur noch ein Paar schöne Fotos bei Nacht, nimmt ein kurzes Video auf, macht noch eine Audionotiz und speichert. Die Anwendung übernimmt noch vor dem Speichern die Bearbeitungszeit als voraussichtliche Aufenthaltsdauer. Erika setzt nun die Besichtigung fort.

Am Ende ihrer Tour prüft sie nochmal, ob die Reihenfolge der Sehenswürdigkeiten stimmt, tippt ggf. noch einige Audionotizen ab, korrigiert die letzten Schreibfehler, speichert die Tour und teilt sie mit der Community.

Max Mustermann, der jüngere Bruder von Erika, studiert BWL ebenfalls an der HAW-Hamburg. Die Erzählungen ihrer älteren Schwester haben ihn sehr beeindruckt, sodass er diese Sehenswürdigkeiten selbst sehen möchte. Erika, als gute Informatikerin, hat ihm dabei geholfen die Anwendung auf seinem Smartphone zu installieren. Er hat die von ihrer Schwester zuvor veröffentlichte Tour abgerufen.

Als er im Zielland angekommen ist, startet er die Tour. Die Anwendung hat dabei seinen Standort ermittelt, den Weg über die Sehenswürdigkeiten berechnet und auf einer Karte angezeigt. Während der Bewegung verfolgt die Anwendung den aktuellen Standort und gibt ggf. Meldungen aus, wenn Max sich zu weit von seinem Weg entfernt.

Sobald Max sich einer Sehenswürdigkeit nähert, wird er darüber ebenfalls mit einer Meldung benachrichtigt. Die Anwendung zeigt den Titel und die Beschreibung der Sehenswürdigkeit mit allen dazugehörigen Medien an die der Autor der Tour aufgenommen hat.

Nachdem alles besichtigt und die Tour ggf. um einige Medien erweitert wurde, speichert Max die Sehenswürdigkeit und die Anwendung führt ihn zur nächsten.

Die Sehenswürdigkeiten werden dabei zu Datensammlungen zusammengefasst die *Points of Interest* (kurz POIs) genannt werden. Sie beinhalten in der Regel mindestens einen Namen sowie die Koordinaten des Ortes, den sie beschreiben, können jedoch auch weitere zusätzliche Informationen wie z. B. Bilder, Videos oder Notizen enthalten.

Mithilfe dieses Szenario lassen sich folgende Hauptanforderungen ableiten:

Eigene Touren erstellen: Der Benutzer soll eine Möglichkeit haben persönliche Reiseführer, sogenannte Touren, zu erstellen indem er interessante Orte oder Objekte als POIs erfasst oder bereits vorhandene wählt und diese ggf. mit zusätzlichen Infos in Form von Texten, Fotos und anderen Medien verseht.

Sehenswürdigkeiten vorschlagen: Die Anwendung soll beim Erfassen eines neuen POIs anhand des aktuellen Standortes nach bereits vorhandenen in der nächsten Nähe suchen und diese dem Benutzer zur Übernahme vorschlagen.

Automatische Tourführung: Die Anwendung soll den Benutzer dabei unterstützen die POIs der aktuell gewählten Tour zu besichtigen indem sie Navigationsfunktion bietet und Ortsabhängige oder den POIs zugehörige Infos anzeigt.

Touren teilen und abrufen: Es soll möglich sein eigene Touren auf einer Internetplattform mit der Community zu teilen oder von den anderen Mitgliedern geteilte Touren abzurufen und zu verwenden.

Abgerufene Touren erweitern: Es soll möglich sein von der Community bezogene Touren zu bearbeiten oder zu ergänzen.

im Nächsten Kapitel wird die Funktionalität der Anwendung anhand der Anwendungsfälle detaillierter dargelegt.

3.1.2 Anwendungsfälle

Anwendungsfalldiagramme

Folgende Anwendungsfalldiagramme geben einen groben Überblick über die wichtigsten Abläufe, deren Zusammenhang sowie die Abhängigkeiten der weiter unten detaillierter beschriebenen Anwendungsfälle.

Die Abbildung 3.1 zeigt Anwendungsfälle die mit dem Erstellen bzw. Bearbeiten der Touren im Zusammenhang stehen. Wie das Erstellen bzw. Bearbeiten der einzelnen POIs abläuft, zeigt die Abbildung 3.2. Das Abspielen sowie das Veröffentlichen und Abrufen der Touren wird entsprechend in den Abbildungen 3.3 und 3.4 gezeigt.

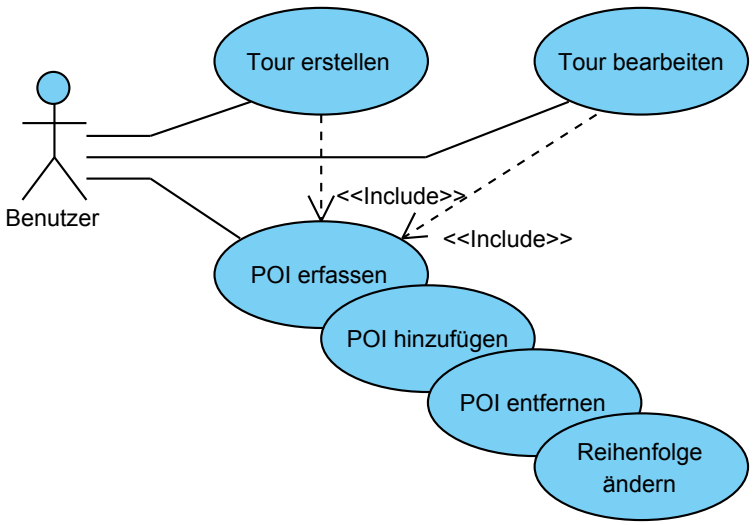


Abbildung 3.1: Anwendungsfalldiagramm - Tour erstellen/bearbeiten.

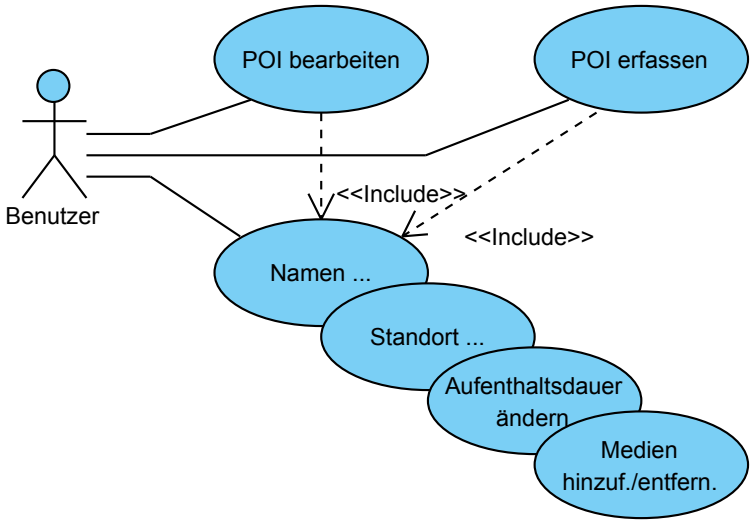


Abbildung 3.2: Anwendungsfalldiagramm - POI erfassen/bearbeiten.

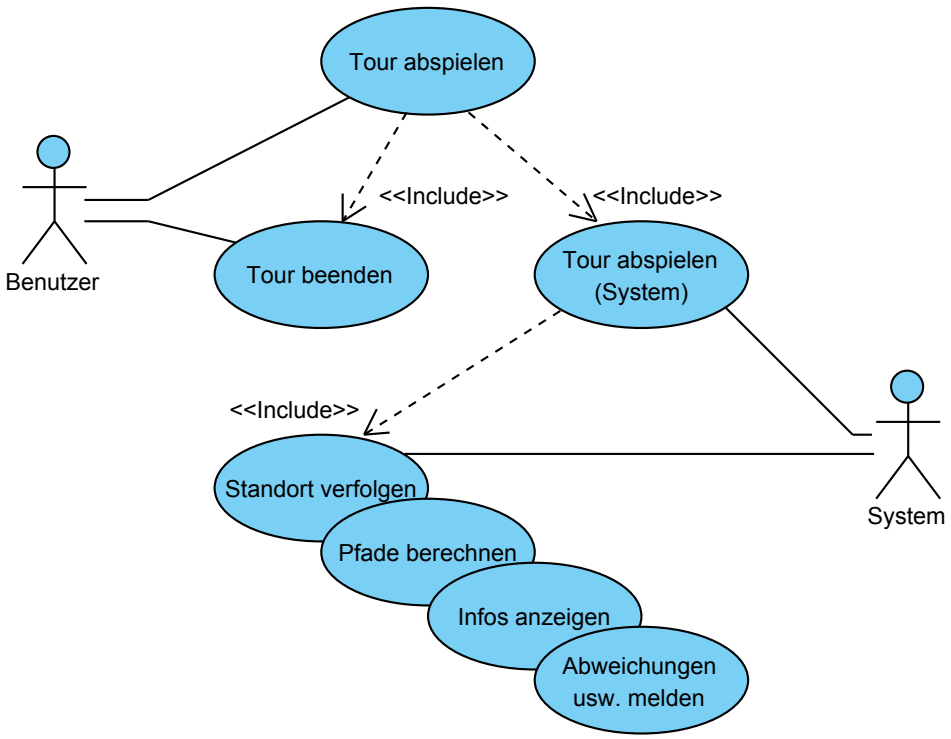


Abbildung 3.3: Anwendungsfalldiagramm - Tour abspielen.

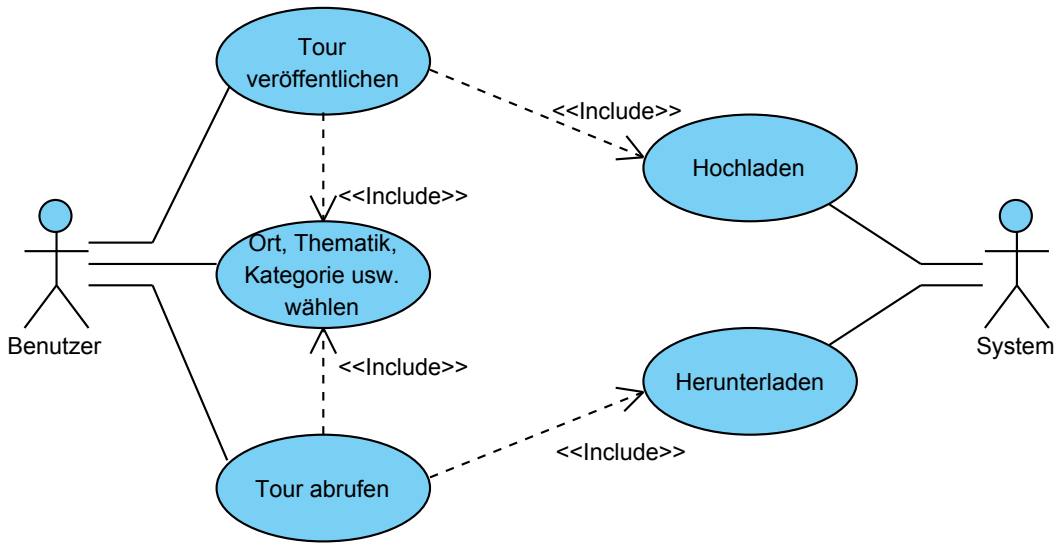


Abbildung 3.4: Anwendungsfalldiagramm - Tour veröffentlichen/abrufen.

Anwendungsfälle: Touren

Folgende Tabellen beschreiben etwas detaillierter die Abläufe, Vor- und Nachbedingungen der einzelnen Anwendungsfälle hinsichtlich der Touren.

Tabelle 3.1: Anwendungsfall - Tour erstellen

Ziel:	Erstellen einer neuen Tour.
Akteur:	Benutzer
Vorbedingung:	Keine.
Ablauf:	
	1. Hauptbildschirm (Touren) aufrufen.
	2. Neue Tour hinzufügen.
	3. Einen Namen vergeben.
optional:	Beschreiben.
entweder:	POIs offline hinzufügen/entfernen. (Tab. 3.3: POIs offline hinzufügen/entfernen)
oder:	Aufzeichnungsmodus starten. (Tab. 3.4: Tour Aufzeichnen)
optional:	Reihenfolge der POIs ändern bzw. optimalen Pfad berechnen.
	4. Speichern.
Nachbedingung:	Eine neue Tour wurde erstellt und gespeichert.

Tabelle 3.2: Anwendungsfall - Tour bearbeiten

Ziel:	Bearbeiten einer vorhandenen Tour.
Akteur:	Benutzer
Vorbedingung:	Mindestens eine Tour wurde zuvor erstellt.
Ablauf:	
	1. Hauptbildschirm (Touren) aufrufen.
	2. Eine Tour auswählen.
	optional: Umbenennen.
	optional: Beschreibung ändern.
	optional: POIs offline hinzufügen/entfernen. (Tab. 3.3: POIs offline hinzufügen/entfernen)
	optional: Reihenfolge der POIs ändern.
	optional: Aufzeichnungsmodus starten. (Tab. 3.4: Tour Aufzeichnen)
	3. Speichern.
Nachbedingung:	Die Tour wurde bearbeitet, Änderungen sind gespeichert.

Tabelle 3.3: Anwendungsfall - POIs offline hinzufügen/entfernen

Ziel:	POIs offline, d.h. nicht vor Ort, einer Tour hinzufügen bzw. entfernen.
Akteur:	Benutzer
Vorbedingung:	Eine Tour wurde angelegt (Tab. 3.1: Tour erstellen) oder zum Bearbeiten geöffnet. (Tab. 3.2: Tour bearbeiten)
Ablauf:	
optional:	POIs zuweisen:
entweder:	Neuen POI erfassen. (Tab. 3.10: POI erfassen)
oder:	Vorhandenen POI zuweisen:
i.	Zum POI Pool wechseln.
ii.	Einen POI aus der Karte auswählen oder suchen.
optional:	POI bearbeiten. (Tab. 3.11: POI bearbeiten)
iii.	Zuweisen.
optional:	POIs entfernen:
a)	Zugewiesene POIs auswählen.
b)	Zuweisung aufheben.
Nachbedingung:	Keine.

Tabelle 3.4: Anwendungsfall - Tour Aufzeichnen

Ziel:	Eine Tour online aufzeichnen.
Akteur:	Benutzer
Vorbedingung:	Eine Tour ist gerade im Aufzeichnungsmodus. (Tab. 3.1: Tour erstellen, Tab. 3.2: Tour bearbeiten)
Ablauf:	
	1. POI hinzufügen. (wiederholbar)
	System: Kartendienst nach vorhandenen POIs in der Nähe abfragen und zur Auswahl anbieten.
	entweder: Vorgeschlagenen POI übernehmen.
	oder: Neuen POI hinzufügen.
	i. Einen Namen vergeben.
	ii. Ggf. beschreiben.
	iii. Ggf. Standort auf der Karte wählen.
	optional: Medien hinzufügen. (Fotos, Videos, Audionotizen)
Nachbedingung:	Keine.

Tabelle 3.5: Anwendungsfall - Tour entfernen

Ziel:	Entfernen einer zuvor erstellten Tour.
Akteur:	Benutzer
Vorbedingung:	Mindestens eine Tour wurde zuvor erstellt.
Ablauf:	
	1. Hauptbildschirm (Touren) aufrufen.
	2. Eine Tour auswählen.
	3. Entfernen.
Nachbedingung:	Gewählte Tour wurde entfernt.

Tabelle 3.6: Anwendungsfall - Tour abspielen

Ziel:	Starten einer Tourführung.
Akteur:	Benutzer
Vorbedingung:	Mindestens eine nicht leere Tour ist vorhanden.
Ablauf:	1. Hauptbildschirm (Touren) aufrufen. 2. Eine Tour auswählen. 3. Führung starten.
Nachbedingung:	Tourführung ist mit der ausgewählten Tour gestartet und der Benutzer sieht POIs auf der Karte, sofern diese in Reichweite sind. Die Anwendung hat die Führung übernommen (Tab. 3.7: Tour abspielen (System))

Tabelle 3.7: Anwendungsfall - Tour abspielen (System)

Ziel:	Automatische Tourführung.
Akteur:	System
Vorbedingung:	Eine Tour wurde vom Benutzer gestartet. (Tab. 3.6: Tour abspielen)
Ablauf:	

1. Aktuellen Standort bestimmen.
2. Eine Route über die POIs berechnen.
3. Kartenansicht zum aktuellen Standort zentrieren.
4. Navigation zum nächsten POI starten.

bedingt: Falls vom Pfad abgekommen:

a) Den Benutzer benachrichtigen.

b) Den Pfad ggf. neu berechnen.

bedingt: Falls nächster POI erreicht:

a) Den Benutzer benachrichtigen.

b) Informationen zum POI anzeigen.

- Name.
- Ein Paar Zeilen der Beschreibung.
- Ein Paar Bilder, wenn Platz vorhanden.
- ”Mehr...”-Button für Detailansicht.
- Ggf. ”Weiter”-Button.

bedingt: Wenn aktueller POI abgeschlossen: Weiter bei 4.

bedingt: Wenn letzter POI abgeschlossen: Tour beenden.

Nachbedingung:	Keine.

Tabelle 3.8: Anwendungsfall - Tour veröffentlichen

Ziel:	Veröffentlichen einer Tour auf einer Internetseite.
Akteur:	Benutzer
Vorbedingung:	Mindestens eine Tour wurde zuvor erstellt.
Ablauf:	1. Hauptbildschirm (Touren) aufrufen. 2. Eine Tour zum Veröffentlichen auswählen. 3. Ort, Thematik, Kategorie und ggf. andere Kriterien angeben. 4. Tour mit POIs und Medien wird zum Webdienst hochgeladen.
Nachbedingung:	Die ausgewählte Tour wurde veröffentlicht und steht zum Abruf bereit.

Tabelle 3.9: Anwendungsfall - Tour online abrufen

Ziel:	Abrufen einer veröffentlichten Tour.
Akteur:	Benutzer
Vorbedingung:	Mindestens eine Tour wurde zuvor veröffentlicht und steht zum Abruf bereit.
Ablauf:	1. Zu Veröffentlichten Touren wechseln. 2. Anhand der Kriterien wie Ort, Thematik, Kategorie usw. suchen. 3. Gewünschte Tour aus den Ergebnissen auswählen. 4. Tour mit POIs und Medien wird vom Webdienst heruntergeladen.
Nachbedingung:	Die Tour wurde abgerufen und steht lokal zur Verfügung.

Anwendungsfälle: POIs

Die nachfolgenden Tabellen beschreiben etwas detaillierter die Abläufe, Vor- und Nachbedingungen der einzelnen Anwendungsfälle hinsichtlich der POIs.

Tabelle 3.10: Anwendungsfall - POI erfassen

Ziel:	Erfassung eines Ortes bzw. Objektes.
Akteur:	Benutzer
Vorbedingung:	Keine.
Ablauf:	
	1. Einen Namen vergeben.
optional:	Beschreiben.
	2. Standort auf der Karte wählen.
optional:	Voraussichtliche Aufenthaltsdauer einstellen.
optional:	Medien hinzufügen/entfernen. (Fotos, Videos, Audionotizen)
	3. Speichern.
Nachbedingung:	POI ist gespeichert und wird ggf. zum Webdienst hochgeladen.

Tabelle 3.11: Anwendungsfall - POI bearbeiten

Ziel:	Ändern eines vorhandenen POIs.
Akteur:	Benutzer
Vorbedingung:	Mindestens ein POI wurde zuvor erfasst.
Ablauf:	
optional:	Umbenennen.
optional:	Beschreibung ändern.
optional:	Standort ändern.
optional:	Voraussichtliche Aufenthaltsdauer ändern.
optional:	Medien hinzufügen/entfernen. (Fotos, Videos, Audionotizen)
	1. Speichern.
Nachbedingung:	Der POI ist wurde gespeichert.

Tabelle 3.12: Anwendungsfall - POI entfernen

Ziel:	Entfernen eines vorhandenen POIs.
Akteur:	Benutzer
Vorbedingung:	Mindestens ein POI wurde zuvor erfasst.
Ablauf:	1. Zu POIs wechseln. 2. Einen POI auswählen. 3. Entfernen. 4. ggf. bestätigen.
Nachbedingung:	POI wurde entfernt.

Tabelle 3.13: Anwendungsfall - POI auf einer Karte anzeigen

Ziel:	Anzeigen eines POI auf der Karte.
Akteur:	Benutzer
Vorbedingung:	Mindestens ein POI wurde zuvor erfasst.
Ablauf:	1. Zu POIs wechseln. 2. Einen POI auswählen. 3. Auf der Karte anzeigen lassen.
Nachbedingung:	POI wird auf der Karte angezeigt.

Tabelle 3.14: Anwendungsfall - POI veröffentlichen

Ziel:	Veröffentlichen einzelner POIs.
Akteur:	Benutzer
Vorbedingung:	Mindestens ein POI wurde zuvor erfasst.
Ablauf:	1. Zu POIs wechseln. 2. Einen POI zum Veröffentlichen auswählen. 3. Ort, Thematik, Kategorie und ggf. andere Kriterien angeben. 4. Zum Webdienst hochladen.
Nachbedingung:	Der ausgewählte POI wurde veröffentlicht und steht zum Abruf bereit.

Tabelle 3.15: Anwendungsfall - POI online abrufen

Ziel:	Abrufen der veröffentlichten POIs.
Akteur:	Benutzer
Vorbedingung:	Mindestens ein POI wurde zuvor veröffentlicht und steht zum Abruf bereit.
Ablauf:	1. Zu Veröffentlichten POIs wechseln. 2. Anhand der Kriterien wie Ort, Thematik, Kategorie usw. suchen. 3. Einen POI aus den Ergebnissen auswählen. 4. Vom Webdienst herunterladen.
Nachbedingung:	Der POI wurde abgerufen und steht lokal zur Verfügung.

3.2 Nicht-funktionale Anforderungen

Neben den funktionalen Anforderungen sind für diese Arbeit folgende nicht-funktionale Anforderungen besonders nennenswert:

Portierbarkeit: Da die Anwendung auf möglichst vielen Geräten laufen soll, muss sie ohne Entwicklungsaufwand, d. h. ohne deren Geschäftslogik anpassen zu müssen, auf andere Plattformen portiert werden können.

Benutzerfreundlichkeit: Eine möglichst intuitive Benutzeroberfläche, sowie für einen nicht technisch versierten Benutzer verständliche Texte und Meldungen sollten vorhanden sein. Ausreichende Reaktionszeit der Anwendung ist ebenfalls nicht unbedeutend.

Erweiterbarkeit: Das System sollte gut strukturiert sein damit es erweitert werden kann ohne dabei einen sehr hohen Aufwand betreiben zu müssen.

Wartbarkeit: Ähnlich wie bei der Erweiterbarkeit sollte der Code gut strukturiert und verwaltet sein, damit ggf. Anpassungen oder Korrekturen leichter durchgeführt werden können.

Für die serverseitige Komponente des Systems gilt insbesondere folgendes:

Verfügbarkeit: Die serverseitige Komponente sollte ständig verfügbar sein, um rund-um-die-Uhr-Zugriffe zu ermöglichen. Heimrechner sind dafür nicht geeignet, ein entsprechender Hosting-Service wird daher benötigt.

Skalierbarkeit: Es soll möglich sein die serverseitige Komponente bei Bedarf z. B. in Clustern einzusetzen, um ggf. wachsender Benutzeranzahl gerecht zu bleiben.

Sicherheit: Sensible Benutzerdaten sollen geschützt und entsprechend behandelt werden.

3.3 Zusammenfassung

Anhand der in diesem Kapitel festgestellten Anforderungen lassen sich unter anderem folgende wesentliche Funktionen ableiten:

Eigene Touren: Erstellen von eigenen, personalisierten Touren indem man sie entweder direkt, z.B. während einer Stadtrundfahrt, aufzeichnet oder später offline zusammenstellt.

Touren abspielen: Die Anwendung soll zum Teil die Rolle eines Navigationsgerätes übernehmen, sodass sie den Benutzer vom POI zum POI führt und mit den entsprechenden Infos dazu versorgt.

Mit der Community teilen: Teilen bzw. Abrufen der Touren oder einzelner POIs mit bzw. von der Community auf einer speziell dafür entworfenen Webplattform.

Mit der Hinsicht auf die technischen Aspekte und Möglichkeiten der Zielplattformen und mit Berücksichtigung der funktionalen und nicht-funktionalen Anforderungen wird im nächsten Kapitel auf die technischen Voraussetzungen eingegangen und ein Entwurf der Lösung erarbeitet.

4 Design

Dieses Kapitel befasst sich mit der Erarbeitung eines Entwurfs.

Im Kapitel 4.1 geht es um technische Voraussetzungen für die Realisierung, sowohl Software- als auch Hardwaretechnisch.

Das Kapitel 4.2 zeigt zunächst einen groben Überblick mit der Systemarchitektur und stellt anschließend im Kapitel 4.2.2 mit der Softwarearchitektur den technischen Entwurf im Detail vor. Hier werden Komponenten-, Klassen- und Sequenzdiagramme aufgeführt, um die Zusammenhänge und entsprechende Prozesse in der Anwendung beschreiben zu können.

Anschließend wird im Kapitel 4.3 eine prototypische Realisierung vorgestellt. Hier wird auf die Entwicklungskonzepte und einige Probleme während der Entwicklung eingegangen sowie deren Lösungen erläutert.

4.1 Technische Voraussetzungen

Da die Anwendung für mobile Geräte gedacht ist, sollten diese einigen Voraussetzungen hinsichtlich der Hardware und der Software genügen.

4.1.1 Hardware

Um Bild- bzw. Video- oder Tonaufnahmen zu ermöglichen, müssen die Geräte über die entsprechende Hardware verfügen. Dies wäre in den ersten beiden Fällen die Kamera sowie das Mikrofon im letzten Fall.

Für die Ortung sowie die Erfassung neuer POIs als auch für die Tourführung ist ein Ortungsmodul erforderlich.

Da die Darstellung der POIs, Berechnung der Pfade und das Anzeigen der Karten mithilfe eines Onlinekartendienstes erfolgt, muss das Gerät entsprechend über eine Internetverbindung verfügen. Dasselbe gilt auch für das Veröffentlichen der POIs auf der Community-Plattform.

4.1.2 Software

Softwaretechnisch ist es unter Anderem erforderlich, dass die Plattformen entsprechende Schnittstellen für den Zugriff auf deren Hardwarekomponenten wie Ortungsmodul, Kamera usw. bieten. Außerdem wird eine Netzwerkschnittstelle benötigt, um auf ein Onlinekartendienst zugreifen zu können.

4.2 Architektur

Dieses Kapitel befasst sich mit der Architektur des gesamten Systems. Es wird einzeln auf die Systemarchitektur sowie auf die Softwarearchitektur eingegangen und das Zusammenspiel der beiden Welten Software und Hardware in sich selbst und miteinander gezeigt.

4.2.1 Systemarchitektur

Dieses Kapitel zeigt die Zusammenarbeit der einzelnen Hardwarekomponenten des gesamten Systems und gibt einen groben Überblick über deren Funktionalität und Aufgaben.

Systemdiagramm

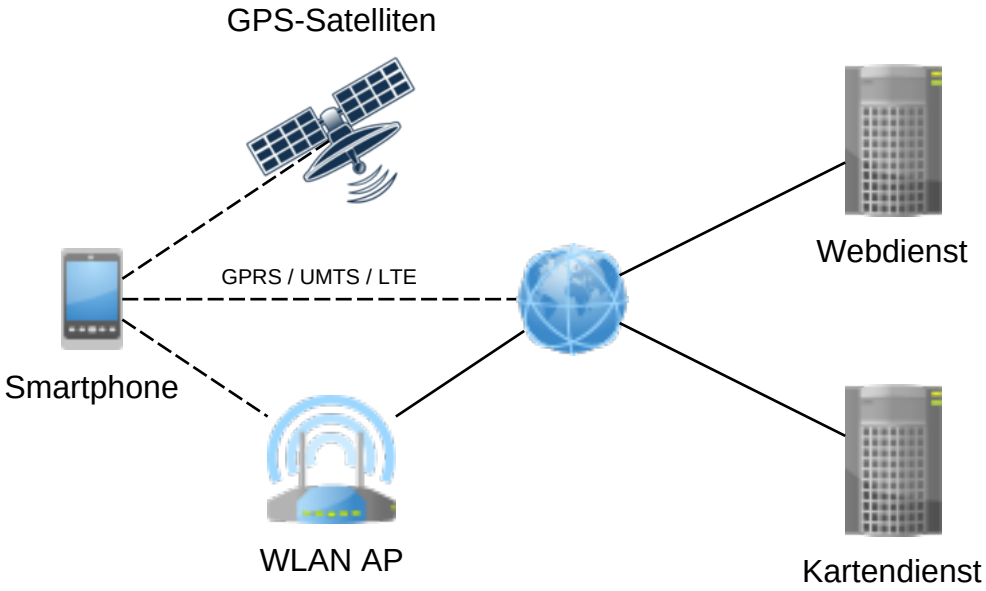


Abbildung 4.1: Systemdiagramm

Erläuterung

Wie das Systemdiagramm in der Abbildung 4.1 zeigt besteht das System unter Anderem aus drei Hauptkomponenten: Einem Smartphone, einem Webdienst auf dem eine Community-Plattform basiert und einem Kartendienst. Deren Aufgaben werden im Folgenden etwas detaillierter Beschrieben.

Smartphone: Das Endgerät auf dem die mobile Anwendung läuft. Diese kommuniziert über das Internet mit den beiden Diensten.

Kartendienst: Liefert der Anwendung abhängig vom Standort des Benutzers entsprechenden Kartenausschnitt zur Orientierung, sowie die POIs und andere Infos über die Umgebung. Berechnet Pfade.

Webdienst: Dient als Webplattform auf der die Community-Mitglieder untereinander Touren, POIs und andere relevante Informationen teilen können.

WLAN AP: Der AccessPoint dient als Alternative zur mobilen Internetverbindung.

GPS-Satelliten: Liefern dem Smartphone Informationen zur Berechnung der aktuellen Position.

4.2.2 Softwarearchitektur

In diesem Kapitel wird auf das Zusammenspiel der einzelnen Softwarekomponenten eingegangen und deren Funktionalität mithilfe eines Komponenten-, einiger Klassen- und Sequenzdiagramme mit begleitender Beschreibung erläutert.

Komponentendiagramm

Das Komponentendiagramm in diesem Abschnitt veranschaulicht wie und in welche Komponenten sich die Anwendung modular aufteilt.

Das gesamte System ist ein verteiltes System und ist in drei der folgenden Hauptkomponenten unterteilt:

Anwendung (Smartphone): Die eigentliche mobile Anwendung, die auf dem Endgerät des Benutzers ausgeführt wird und auch den Kern des gesamten Systems darstellt.

Web Service (Community Platform): Eine Webplattform im Internet für Community-Mitglieder, auf der sie unter Anderem ihre Touren und POIs untereinander teilen können.

Map Service (external service): Ein von Drittanbietern zur Verfügung gestellter Webdienst im Internet, der unter Anderem Kartenmaterial liefert und Routenplanung anbietet.

Im Folgenden wird das Komponentendiagramm vorgestellt und anschließend auf die einzelnen Komponenten im Detail eingegangen.

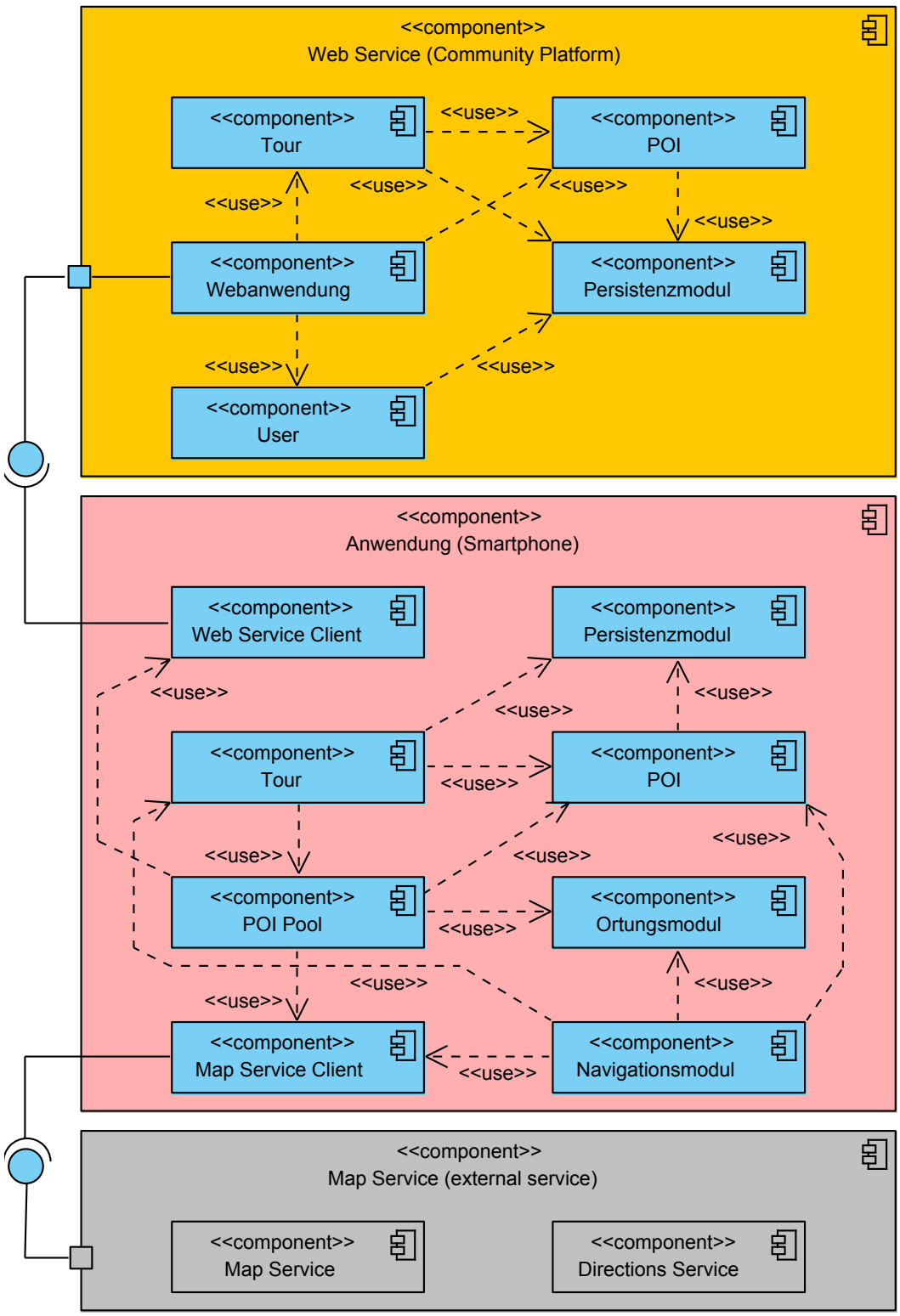


Abbildung 4.2: Komponentendiagramm

Erläuterung

Das Komponentendiagramm in der Abbildung 4.2 zeigt die wesentlichen Komponenten der Anwendung und deren Zusammenhang. Im Folgenden werden die einzelnen Komponenten etwas detaillierter Erläutert.

Web Service (Community Platform)

Tour: Repräsentiert die eigentliche Tour serverseitig. Diese Komponente hat im Grunde die selbe Kernfunktionalität wie die *Tour* auf dem Smartphone.
Sie Benutzt die Komponenten *POI* und *Persistenzmodul*.

POI: Repräsentiert den eigentlichen POI serverseitig. Diese Komponente hat im Grunde auch die selbe Kernfunktionalität wie der *POI* auf dem Smartphone.
Sie Benutzt die Komponente *Persistenzmodul*.

Webanwendung Stellt den serverseitigen Teil der Anwendung dar. Diese Komponente repräsentiert eine Webplattform um die sich eine Community bildet, sodass ihre Mitglieder Touren und POIs untereinander teilen können.
Sie benutzt die Komponenten *Tour*, *POI* und *User*.

User: Implementiert den Benutzer auf der Community-Webplattform. Diese Komponente beinhaltet z. B. Methoden für das An- bzw. Abmelden, Passwort ändern und andere typische Benutzeraktionen.
Sie benutzt die Komponente *Persistenzmodul*.

Persistenzmodul: Stellt die Persistenzschicht des serverseitigen Anwendungsteils dar.

An dieser Stelle ist anzumerken, dass die Hauptkomponente *Web Service (Community Platform)* inklusive ihrer Unterkomponenten nicht mehr weiter im Detail verfolgt wird, da dies sonst den Umfang einer Bachelorarbeit bei weitem übersteigt. Im Folgenden wird lediglich eine grobe Skizze einer möglichen Umsetzung vorgestellt.

Es war ursprünglich geplant die Community-Plattform als eine Webseite zu entwerfen, auf der sich die Community-Mitglieder mit ihren Geräten registrieren und ggf. Freundschaftskreise bilden können, um in diesem sozialen Netzwerk ihre ganzen Touren oder einzelne POIs untereinander zu teilen.

Um das realisieren zu können wäre eine Webanwendung in einer der gängigsten Programmiersprachen bzw. Technologien für serverseitige Anwendungen nötig. Diese Webanwendung

müsste mindestens die Geschäftsobjekte *Tour*, *Poi* und *Media* abbilden und sie inklusive Mediendateien persistent speichern können. Beispielsweise könnte es ein Apache Server mit einer MySQL-Datenbank sein, welcher eine in PHP geschriebene Webanwendung ausführt.

Um die Kommunikation mit der Community-Plattform zu ermöglichen, könnte man eine Schnittstellenarchitektur wie REST verwenden. Diese Architektur hat sich im Internet in Form von Webseiten bereits etabliert und bewehrt. (Bayer, 2002) Als Übertragungsformat wäre JSON ein guter Kandidat, da seine Struktur sehr kompakt ist. Außerdem wird JSON mittlerweile von allen modernen Browsern und vielen Programmiersprachen nativ unterstützt.

Die mobile Webanwendung auf dem Smartphone könnte auf diesem Wege URLs auf dem Server der Community-Platform mittels AJAX mit HTTP POST-Befehl aufrufen, um z. B. ein als JSON serialisiertes Tour- bzw. POI-Objekt inklusive Mediendateien an den Server zu schicken oder von ihm zu erhalten. Außerdem kann man auf diese Art auch auf die vom Browser bereitgestellten Verschlüsselungstechniken wie HTTPS/SSL setzen.

Anwendung (Smartphone)

Tour: Stellt die Tour selbst dar. Diese Komponente implementiert unter Anderem z. B. Methoden für Instanziierung und persistentes Speichern der Touren oder auch Hinzufügen und Entfernen der POIs.
Sie benutzt die Komponenten *POI* als ihren Bestandteil, *POI Pool*, um nach POIs suchen zu können, und *Persistenzmodul*.

POI: Stellt den POI selbst dar. Diese Komponente implementiert unter Anderem eigene Funktionalität für Instanziierung und persistentes Speichern der POIs, bietet außerdem noch Methoden zum Verwalten von Medien.
Sie benutzt die Komponente *Persistenzmodul*.

POI Pool: Verwaltet POIs aus unterschiedlichen Quellen vereint an einem einzigen Ort und stellt sie dem Benutzer transparent in einer Liste dar, unabhängig davon, ob es lokal gespeicherte, von der Community oder vom *Map Service* bezogene POIs sind.
Diese Komponente benutzt die Komponenten *Ortungsmodul*, *Web Service Client*, *Map Service Client* und *POI*, um POIs von den entsprechenden Quellen ggf. auch im Geografischen Kontext zu beziehen.

Ortungsmodul: Ist für die Benutzung und Verwaltung des Ortungsmoduls auf dem Gerät zuständig, sei es GPS-, Netzwerk-basiert oder andere Ortung.

Navigationsmodul: Diese Komponente übernimmt die Führung des Benutzers entlang einer zur gegebenen Tour berechneten Route. Ihre Funktionalität ist ähnlich den herkömmlichen Navigationsgeräten.
Sie benutzt die Komponenten *Ortungsmodul*, *Map Service Client*, *Tour* und *POI*, um aktuelle Position zu bestimmen, Karten anzuzeigen, Pfade zu berechnen und sie ggf. zu korrigieren.

Map Service Client: Stellt einen Adapter für ein Onlinekartendienst dar. Diese Komponente bietet Methoden, mit deren Hilfe das Kartenmaterial abgerufen und dargestellt, Routen berechnet oder auch andere geografische Informationen verwendet werden können. Dieser Adapter kann eventuell erweitert oder durch einen anderen Ersetzt werden, um z. B. andere Dienste anschließen zu können.
Sie benutzt die Komponente *Map Service (external service)*.

Web Service Client: Die Komponente ist für den Zugriff auf den Webdienst der Community-Plattform zuständig mit dessen Hilfe das Teilen bzw. Abrufen von Touren und POIs realisiert werden kann.
Sie benutzt die Komponente *Webanwendung*, um auf entsprechende Inhalte zugreifen zu können.

Persistenzmodul: Implementiert verschiedene Methoden, mit deren Hilfe sich Daten und Objekte persistent speichern oder lesen lassen. Diese Komponente stellt eine Fassade für den persistenten Speicher der Anwendung dar.

Map Service (external service)

Map Service: Diese Komponente ist ein Onlinekartendienst von einem Drittanbieter. Sie stellt den Umgebungsplan, POIs, sowie andere Umgebungs- bzw. POI-bezogene Informationen zur Verfügung.

Directions Service: Implementiert einen Routenplaner im Internet. Diese Komponente stellt Methoden für die Pfadberechnung zur Verfügung und ist ein Teil des Kartendienstes.

Diese beiden Unterkomponenten dienen ausschließlich der Verständlichkeit bzw. Veranschaulichung, die Hauptkomponente *Map Service (external service)* ist ein von Drittanbietern zur Verfügung gestellter Dienst und ist somit als Black-Box zu betrachten.

Klassendiagramme

Im Folgenden sind zwei Klassendiagramme vorgestellt, die den Zusammenhang der Klassen und Objekte der Anwendung zeigen. Jedes dieser Diagramme wird mit grundsätzlichen Besonderheiten bzw. Entwurfsentscheidungen kurz vorgestellt, anschließend werden alle Klassen und deren Methoden im Detail erläutert.

Das Teildiagramm in der Abbildung 4.3 zeigt zentrale Objekte der Anwendung und deren Schnittstellen zu externen Diensten sowie zur Persistenzschicht.

Die Klasse *Persistence* stellt dabei eine Fassade zur Persistenzschicht der Anwendung dar. Sie bietet Methoden für alle möglichen Operationen auf der Persistenzschicht.

Als Schnittstelle zu externen Diensten ist *MapServiceClient* zu erwähnen. Es ist ein Interface, das die Schnittstelle des Adapters für den Zugriff auf den Onlinekartendienst, den Routenplaner und externe POIs definiert. Auf diese Art ist es möglich auch andere Kartendienste anzuschließen. Je nach Realisierung des Anbieters kann dieser Adapter entweder eine vom Anbieter gelieferte Client-API benutzen oder sie selbst implementieren. Dieses Interface wird exemplarisch durch einen Adapter für den Onlinekartendienst *Google Maps* realisiert.

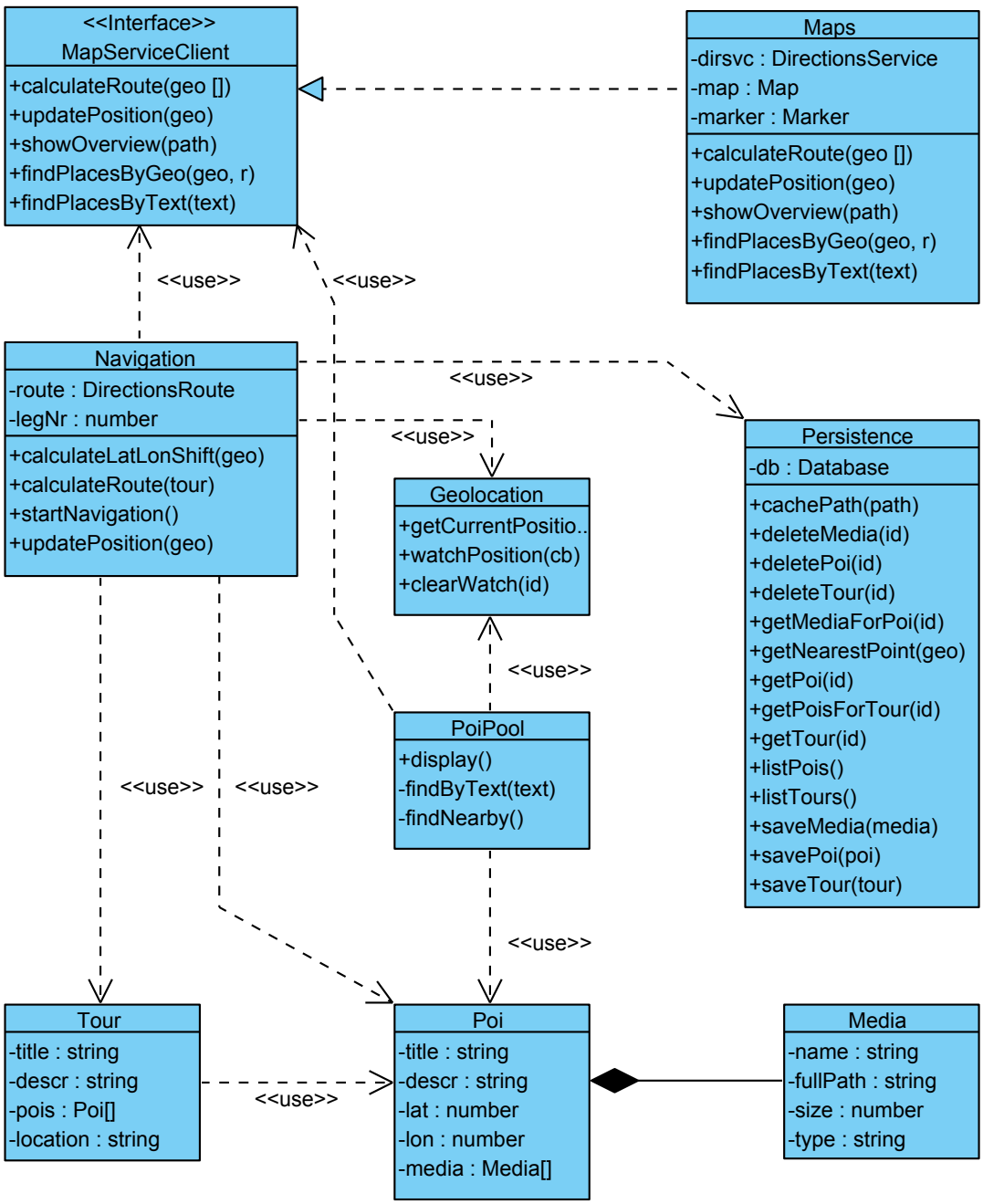


Abbildung 4.3: Klassendiagramm - Zentrale Objekte und Schnittstellen

Das Teildiagramm in der Abbildung 4.4 zeigt die Klassen, die das Datenmodell der Anwendung darstellen. Es ist an das MVC-Konzept mit ActiveRecord-Pattern angelehnt, welches das

Motto verfolgt *model does its own persistence* (Google, 2013). Das heißt, dass das Modell sich selbst um seine Persistenz kümmert.

In der Literatur werden verschiedene Varianten des MVC-Patterns verwendet, die sich hinsichtlich der Kommunikation zwischen Modell, View und Controller etwas unterscheiden. (Google, 2013; Osmani, 2014) Bei einigen wird sowohl das Modell als auch die View nur vom Controller modifiziert bzw. aktualisiert, bei den anderen informiert das Modell den Controller über seine Änderungen. Im Rahmen dieser Arbeit wurde entschieden die Variante mit dem Observer-Pattern zu verwenden, bei der die View ihr Modell überwacht und dementsprechend über dessen Änderungen informiert wird. Im Falle von Listen wurde jedoch eine Ausnahme gemacht. Da deren Modell nicht einem Geschäftsobjekt entspricht, sondern einer Sicht auf Teileigenschaften mehrerer Elemente, werden deren Views direkt vom Controller aktualisiert.

Die Klassen *Tour*, *Poi* und *Media* stellen das Modell im MVC-Konzept dar, die Views und Controller werden durch die entsprechenden Buchstaben V und C am Ende der Klassennamen gekennzeichnet.

Im Folgenden wird exemplarisch das Zusammenspiel eines Modells *Tour* mit seinen Views und Controller erläutert.

1. In der Initialisierungsphase wird eine Methode des Tour-Controllers *TourC.listTours()* aufgerufen. Diese soll eine Liste aller Touren anzeigen.

2. Als Erstes wechselt sie ggf. zur Ansicht, auf der die Tourenliste angezeigt werden soll und ruft eine statische Methode des Modells *Tour.getList()* auf, um die Liste der Titel und IDs aller Touren zu erhalten.

3. Im nächsten Schritt erstellt der Controller eine View *TourListV* und gibt ihr diese Liste und einen Verweis auf sich selbst mit.

4. Beim Aufruf der Methode *TourListV.display()* werden die Titel der Touren angezeigt und einige Methoden des Controllers an die entsprechenden UI-Elemente gebunden, sodass sie bei Benutzeraktionen mit der id des entsprechenden Elementes aufgerufen werden. Z. B. *TourC.editTour(id)* zum Bearbeiten der gewählten Tour.

5. Klickt man auf "Bearbeiten", so ruft der Controller die statische Methode *Tour.getInstance(id)* auf, die ein Tour-Objekt erzeugt. Danach wird ggf. zur entsprechenden Ansicht gewechselt, eine View *TourV* erstellt, welcher das Tour-Objekt und der Verweis auf den Controller mitgegeben werden, und anschließend *TourV.displayEditor()* aufgerufen, um die Tour in einem Formular darzustellen.

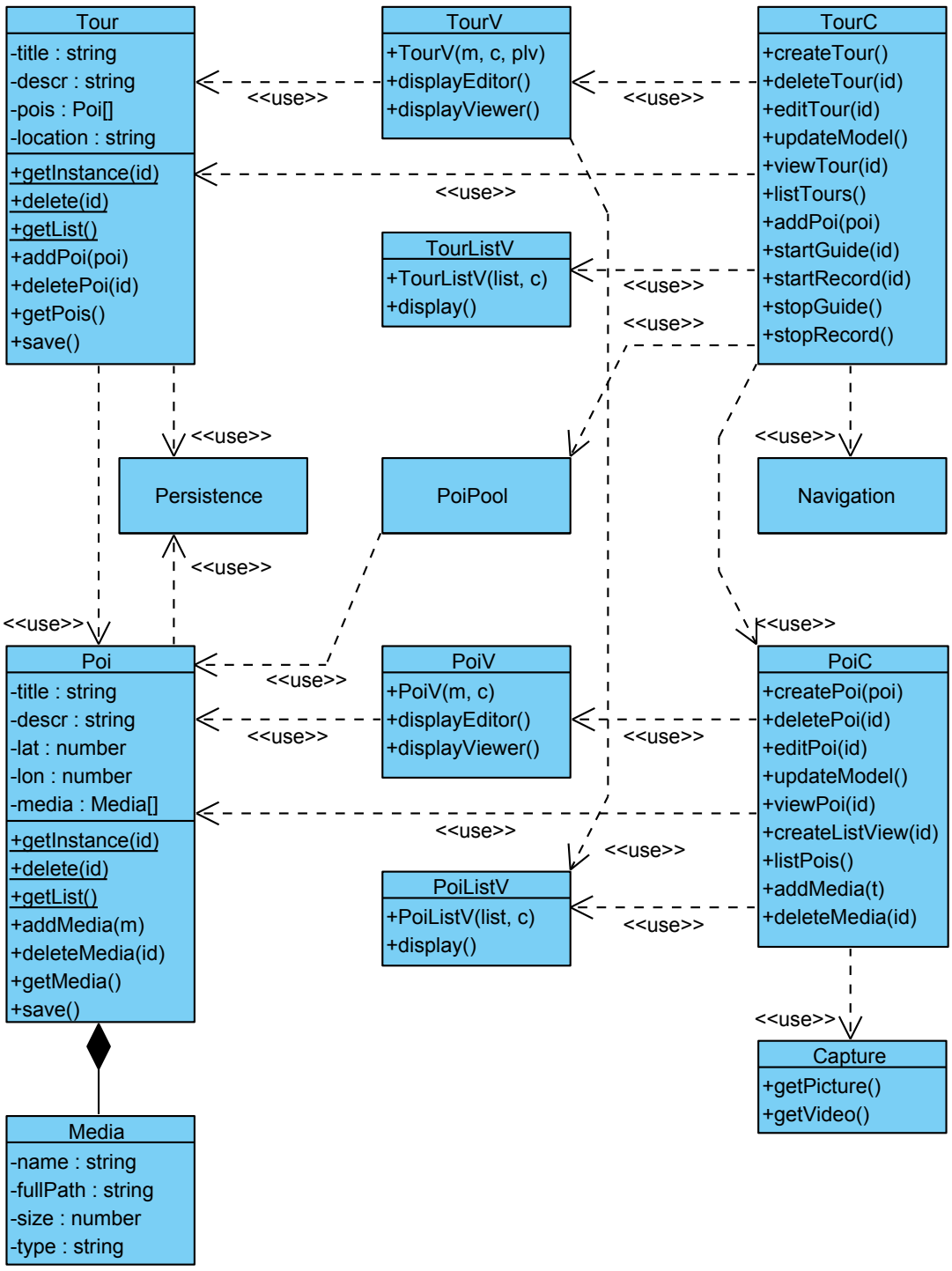


Abbildung 4.4: Klassendiagramm - MVC

Erläuterung

Die Klassendiagramme in den Abbildungen 4.3 und 4.4 zeigen wesentliche Klassen mit einigen wichtigen Attributen und Methoden. Im Folgenden werden die Attribute und Methoden der Klassen im Detail erläutert.

Navigation Diese Klasse ist für den gesamten Navigationsprozess verantwortlich. Sie bietet Methoden mit deren Hilfe der Navigationsablauf gesteuert wird. Außerdem benutzt sie die Klassen *Tour*, *Poi*, *Persistence* und *MapServiceClient* (Maps), um z. B. aktuelle Position auf einer Karte anzuzeigen oder mithilfe eines Onlinedienstes Pfade über POIs zu berechnen.

Im Folgenden werden die Attribute und Methoden der Klassen im Detail erläutert

Attribute

route : DirectionsRoute: Hält die vom *Map Service Client* gelieferte Route mit Wegbeschreibung in Form von Wegpunkten und Anweisungen.

legNr : number: Hält die Nummer aktueller Teilstrecke zwischen zwei benachbarten POIs.

Methoden

calculateLatLonShift(geo): Berechnet annähernd Distanzen in Metern welche ein Latitude- bzw. Longitude-Grad auf der Erdoberfläche nahe einer durch *geo* bestimmten Position beschreibt. Es wird ein trigonometrisches Verfahren namens Haversine angewendet, da hier, unter Annahme, dass die Erde eine richtige Kugel ist, mit einem niedrigen Aufwand gute Ergebnisse erzielt werden können. Die Genauigkeit dieses Verfahrens sinkt jedoch außerhalb der üblichen Breitengrade (Veness, 2014). Diese Distanzen dienen der Berechnung von Entfernungen zwischen zwei durch Geokoordinaten beschriebenen Punkte, z. B. für die Bestimmung der am nächsten liegenden Wegpunkte der Route.

calculateRoute(tour): Holt sich POIs von der gegebenen Tour und lässt den *Map Service Client* über diese eine Route berechnen.

startNavigation(): Startet die Navigation zum nächsten POI entlang der zuvor berechneten Route.

updatePosition(geo): Callback-Funktion. Sie nimmt die aktuelle Position vom Ortungsmodul entgegen und stößt andere für jede neuen Position wiederkehrende Prozesse an. Dazu gehören z. B. Aktualisieren der aktuellen Position auf der Karte, ggf. Neuberechnung

der Wege oder Entfernungen zu Objekten sowie andere von der Position abhängige Ereignisse. Diese Methode wird vom Ortungsmodul jedes Mal aufgerufen, wenn sich die aktuelle Position oder Ausrichtung ändert.

Maps Diese Klasse ist eine konkrete Realisierung der Schnittstelle *MapServiceClient*. Sie benutzt einen der zur Verfügung stehenden Onlinekartendienste im Web und ist für die Darstellung der Karte und Berechnung der Pfade zuständig.

Die wesentlichen Attribute und Methoden dieser Klasse werden im Folgenden erläutert.

Attribute

dirsvc : DirectionsService: Hält ein vom Kartendienstanbieter zur Verfügung gestelltes Objekt, welches die Schnittstelle zum Routenplaner darstellt und Routen im Internet berechnen lässt.

map : Map Hält ein Objekt, das eine vom Kartendienstanbieter zur Verfügung gestellte Implementierung der Karte darstellt. Diese kapselt unter Anderem Zugriffe auf die sogenannten Tiles (Kacheln) des Kartenmaterials und ist komplett für die Darstellung der Karte zuständig.

marker : Marker: Markiert die aktuelle Position auf der Karte und zeigt die Ausrichtung nach dem Kompass oder die Bewegungsrichtung.

Methoden

calculateRoute(geo[]): Lässt den Routenplanerdienst im Internet eine Route über die gegebene Liste der POI-Koordinatenpunkte berechnen.

updatePosition(geo): Aktualisiert die Position auf der Karte sowie die des Markers, berechnet ggf. auch seine Ausrichtung.

showOverview(path): Zeigt die durch *path* beschriebene Übersicht einer Route auf der Karte.

findPlacesByGeo(geo): Lässt den Onlinedienst nach POIs in der Umgebung eines durch *geo* bestimmten Ortes suchen.

findPlacesByText(text): Lässt den Onlinedienst nach POIs suchen, welche mit dem durch *text* bestimmten Suchkriterium übereinstimmen.

Persistence Diese Klasse ist für das Speichern und Lesen der Daten zuständig. Sie stellt eine Fassade für den persistenten Speicher der Anwendung dar und wird mithilfe einer Datenbank realisiert.

Im Folgenden wird auf die einzelnen Attribute und Methoden im Detail eingegangen.

Attribute

db : Database: Hält ein Objekt, welches die vom System bereitgestellte Datenbank repräsentiert und Methoden zum Absetzen von SQL-Anfragen bietet.

Methoden

cachePath(path): Diese Methode speichert den vom *MapServiceClient* berechneten Pfad in der Datenbank, um auf die einzelnen Pfadpunkte sehr schnell zugreifen zu können. Dies ist erforderlich um z. B. im Sekundentakt wiederkehrend den am nächsten liegenden Wegpunkt sowie die Entfernung zu ihm effizient zu bestimmen.

deleteMedia(id): Löscht einen durch gegebene *id* bestimmten Medieneintrag.

deletePoi(id): Löscht einen durch gegebene *id* bestimmten POI.

deleteTour(id): Löscht eine durch gegebene *id* bestimmte Tour.

getMediaForPoi(id): Gibt Medienobjekte für einen durch gegebene *id* bestimmten POI zurück.

getNearestPoint(geo): Sucht anhand der durch *geo* gegebenen Position nach dem am nächsten liegenden Wegpunkt im zuvor mit *cachePath* gespeicherten Pfad, berechnet die Entfernung zu ihm und gibt beides zurück.

getPoi(id): Gibt einen durch gegebene *id* Bestimmten POI zurück.

getPoisForTour(id): Gibt POIs für eine durch gegebene *id* Bestimmte Tour zurück.

getTour(id): Gibt eine durch gegebene *id* bestimmte Tour zurück.

listPois(): Gibt eine Liste aller POIs zurück.

listTours(): Gibt eine Liste aller Touren zurück.

saveMedia(media): Speichert Infos über durch *media* beschriebene Mediendatei. Dazu gehören z. B. Typ, Dateipfad, Größen und andere.

savePoi(poi): Speichert einen neuen bzw. aktualisiert einen bestehenden POI, wenn *poi* eine *id* enthält.

saveTour(tour): Speichert eine neue bzw. aktualisiert eine bestehende Tour, wenn *tour* eine *id* enthält.

Geolocation Diese Klasse verwaltet das Ortungsmodul des Smartphones. Sie nimmt Positionsanfragen entgegen.

Methoden

getCurrentPosition(): Gibt einmalig die aktuelle oder zuletzt bekannte Position als Geokoordinaten zurück.

watchPosition(cb): Registriert eine durch *cb* gegebene Callback-Funktion, welche regelmäßig ausgeführt wird und die aktuelle Position entgegen nimmt. Sie gibt außerdem eine id zurück, mit welcher diese Callback-Funktion anschließend abgemeldet werden muss.

clearWatch(id): Meldet die mit der gegebenen *id* verknüpfte Callback-Funktion ab und fährt das Ortungsmodul herunter, wenn dieses von keinen weiteren Anwendungen mehr benutzt wird.

PoiPool Diese Klasse ist eine Sicht auf POIs aus mehreren Quellen. Sie sucht und zeigt die lokal verwendeten und die vom Onlinekartendienst bereitgestellten POIs. Das Suchergebnis kann ggf. von der aktuellen Position oder einem Suchkriterium abhängig gemacht werden, z. B. wenn nur POIs in der Nähe bzw. mit einem bestimmten Namen angezeigt werden sollen.
Sie benutzt die Klassen *MapServiceClient* (Maps), um POIs vom Onlinekartendienst zu beziehen oder sie ggf. auf einer Karte anzuzeigen und *Poi*, um auf Listen der lokalen POIs zugreifen zu können.

Methoden

display(): Öffnet eine Ansicht, die eine Liste in der Nähe der aktuellen Position liegender POIs mit einer Kartenübersicht anzeigt. Hier kann man einen POI entweder aus der Liste wählen, oder zusätzlich eine Volltextsuche tätigen.

findByText(text): Sucht nach POIs mit einem durch *text* bestimmten Begriff.

findNearby(): Sucht nach allen POIs in der nächsten Nähe der aktuellen Position.

Tour Diese Klasse repräsentiert das Datenmodell der Tour. Sie stellt Funktionalität sowohl für die Objekte der Touren, als auch einige statischen Methoden zur Verfügung, die etwas allgemeinere Aufgaben erledigen und nicht zu den Objekten selbst gehören.

Ein Tour-Objekt enthält seine POIs nicht als Objekte, sondern als eine Liste der Titel und IDs, mit deren Hilfe die Objekte bei Bedarf instanziiert werden. Daher wirken sich die Methoden zum Hinzufügen bzw. Entfernen der POIs direkt auf der Persistenzschicht aus.
Sie benutzt die Klasse *Poi* und *Persistence*.

Attribute

title : string: Der eigentliche Titel bzw. Name der Tour.

descr : string: Etwas längere Beschreibung.

pois : Poi[]: Eine Liste der POIs, welche nur Titel mit den entsprechenden IDs enthält.

location : string: Der Ort, an dem die Tour verläuft, z. B. Hamburg.

Methoden

static getInstance(id): Holt sich von der Persistenzschicht eine durch gegebene *id* bestimmte Tour und erstellt daraus ein Objekt.

static delete(id): Entfernt eine durch gegebene *id* bestimmte Tour in der Persistenzschicht.

static getList(): Holt von der Persistenzschicht eine Liste aller Touren, bestehend aus IDs und Titeln, und gibt sie zurück.

addPoi(poi): Fügt einen neuen POI hinzu. Dabei wird der POI direkt in der Persistenzschicht gespeichert und beim Tour-Objekt lediglich die Liste aktualisiert.

deletePoi(id): Entfernt einen durch gegebene *id* bestimmten POI aus der Tour. Dabei wird der POI direkt in der Persistenzschicht entfernt und beim Tour-Objekt lediglich die Liste aktualisiert.

getPois(): Holt von der Persistenzschicht eine Liste seiner POIs, welche lediglich aus den Titeln, IDs und Geokoordinaten besteht und gibt sie zurück.

save(): Speichert den aktuellen Zustand in der Persistenzschicht. Objekte ohne id werden beim Speichern neu angelegt.

TourV Diese Klasse ist eine der Tour-Views. Sie ist dafür zuständig die Touren visuell darzustellen und das dahinter liegende Modell zu überwachen, um die Darstellung bei Änderungen ggf. zu aktualisieren. Mithilfe dieser View kann die Tour zum Anschauen oder Bearbeiten dargestellt werden.

Methoden

TourV(m, c, plv): Diese Methode ist ein Konstruktor. Sie nimmt die durch *m* und *c* bestimmten Objekte Modell (*Tour*) und Controller (*TourC*) entgegen, richtet einen Observer ein, um das Modell überwachen zu können, und bindet Methoden des Controllers an die entsprechenden UI-Elemente. Mit dem Parameter *plv* wird auch die Listen-View zugehöriger POIs mitgegeben.

displayEditor(): Lässt die View das Modell in einem Formular zum Bearbeiten darstellen.

displayViewer(): Lässt die View das Modell in üblicher Form zur Ansicht darstellen.

TourListV Diese Klasse ist eine weitere Tour-View. Sie ist dafür zuständig Touren als Listen darzustellen. Da das Datenmodell einer Liste nicht einem Geschäftsobjekt entspricht, sondern einer Sicht auf Teileigenschaften mehrerer Objekte, werden die Listen-Views mit einigen Ausnahmen behandelt. Sie bestehen darin, dass die Aktualisierung der Darstellung nicht durch das Überwachen des Modells, sondern vom entsprechenden Controller direkt nach der Änderung am Modell erfolgt.

Methoden

TourListV(list, c): Diese Methode ist ein Konstruktor. Sie nimmt die durch *list* und *c* bestimmten Objekte Liste und Controller (*TourC*) entgegen und bindet Methoden des Controllers an die entsprechenden UI-Elemente. Dabei ist *list* eine einfache Liste von Titeln und IDs der Touren.

display(): Lässt die View die Liste entsprechend darstellen.

TourC Diese Klasse ist ein Tour-Controller. Sie ist dafür zuständig hinsichtlich der Touren auf Benutzereingaben zu reagieren, Abläufe zu steuern und Entscheidungen zu treffen.

Methoden

createTour(): Erstellt ein leeres Tour-Objekt mit einer entsprechenden View und lässt es von dieser View zum Bearbeiten darstellen. Beim Speichern in der Persistenzschicht werden Objekte ohne id als neue abgelegt.

deleteTour(id): Lässt das Tour-Modell eine durch gegebene *id* bestimmte Tour in der Persistenzschicht löschen. Dabei sorgt der Controller selbst für die Aktualisierung der Listen-View.

editTour(id): Lässt das Tour-Modell eine durch gegebene *id* bestimmte Tour instanziieren, erstellt eine View dafür und lässt das Tour-Objekt von dieser View zum Bearbeiten darstellen.

updateModel(): Übernimmt die Änderungen aus der View in das Datenmodell.

viewTour(id): Lässt das Tour-Modell eine durch gegebene *id* bestimmte Tour instanziieren, erstellt eine View dafür und lässt das Tour-Objekt von dieser View zur Ansicht darstellen.

listTours(): Lässt das Tour-Modell eine Liste aller Touren von der Persistenzschicht holen, erstellt eine Listen-View dafür und lässt diese View die Liste darstellen.

addPoi(): Lässt die Klasse *PoiPool* dem Benutzer einige POIs zur Auswahl vorschlagen und erstellt ggf. eine Kopie des gewählten POIs. Lässt anschließend den POI-Controller *PoiC* mit diesem POI ein neues, oder wenn nichts ausgewählt ein leeres POI-Objekt erstellen, es zur Bearbeitung darstellen und beim Speichern zurückgeben, sodass er der aktuellen Tour hinzugefügt werden kann.

startGuide(id): Lässt das Tour-Modell eine durch gegebene *id* bestimmte Tour instanziieren und *Navigation* eine Tourführung über diese Tour starten.

startRecord(id): Lässt das Tour-Modell eine durch gegebene *id* bestimmte Tour instanziieren und öffnet eine Ansicht, welche diese Tour und eine Karte mit den umliegenden POIs mithilfe der Klasse *PoiPool* anzeigt, die den Benutzer möglicherweise interessieren könnten. Definiert auch eine Callback-Methode, mit welcher die ausgewählten POIs der Tour hinzugefügt werden.

stopGuide(): Beendet die Tourführung.

stopRecord(): Beendet die Aufzeichnung.

Poi Diese Klasse repräsentiert Das Datenmodell des POIs. Sie stellt Funktionalität sowohl für die Objekte der POIs, als auch einige statischen Methoden zur Verfügung, die etwas allgemeinere Aufgaben erledigen und nicht zu den Objekten selbst gehören.

Ein POI-Objekt enthält seine Medien nicht als Objekte, sondern als eine Liste der IDs, sodass die entsprechenden Objekte bei Bedarf geladen werden können. Daher wirken sich die Methoden zum Hinzufügen bzw. Entfernen der Medien direkt auf der Persistenzschicht aus.

Die Medien sind ein Bestandteil der POIs und haben kein eigenes Datenmodell.
Diese Klasse benutzt die Klassen *Persistence* und *PoiPool*.

Attribute

title : string: Der eigentliche Titel bzw. Name der Tour.

descr : string: Etwas längere Beschreibung.

lat : number: Position des POIs in Breitengrad (Latitude).

lon : number: Position des POIs in Längengrad (Longitude).

media : Media[]: Eine Liste der Medien, welche nur die entsprechenden IDs enthält.

Methoden

static getInstance(id): Holt sich von der Persistenzschicht einen durch gegebene *id* bestimmten POI und erstellt daraus ein Objekt.

static delete(id): Entfernt einen durch gegebene *id* bestimmten POI in der Persistenzschicht.

static getList(): Holt von der Persistenzschicht eine Liste aller POIs, bestehend aus IDs und Titeln, und gibt sie zurück.

addMedia(m): Fügt ein neues Medien-Objekt hinzu. Dabei wird das Objekt direkt in der Persistenzschicht gespeichert und beim POI-Objekt lediglich die Liste aktualisiert.

deleteMedia(id): Entfernt ein durch gegebene *id* bestimmtes Medien-Objekt aus derm POI. Dabei wird das Medien-Objekt direkt in der Persistenzschicht entfernt und beim POI-Objekt lediglich die Liste aktualisiert.

save(): Speichert den aktuellen Zustand in der Persistenzschicht. Objekte ohne id werden beim Speichern neu angelegt.

PoiV Diese Klasse ist eine der POI-Views. Sie ist dafür zuständig die POIs visuell darzustellen und das dahinter liegende Modell zu überwachen, um die Darstellung bei Änderungen ggf. zu aktualisieren. Mithilfe dieser View kann der POI zum Anschauen oder Bearbeiten dargestellt werden.

Methoden

PoiV(m, c): Diese Methode ist ein Konstruktor, sie nimmt die durch *m* und *c* bestimmten Modell *Poi* und Controller *PoiC* entgegen, richtet Observer ein, um das Modell überwachen zu können, und bindet Methoden des Controllers an die entsprechenden UI-Elemente.

displayEditor(): Lässt die View das Modell in einem Formular zum Bearbeiten darstellen.

displayViewer(): Lässt die View Das Modell in üblicher Form zur Ansicht darstellen.

PoiListV Diese Klasse ist eine weitere POI-View. Sie ist dafür zuständig POIs als Listen darzustellen. Da das Datenmodell einer Liste nicht einem Geschäftsobjekt entspricht, sondern einer Sicht auf Teileigenschaften mehrerer Objekte, werden die Listen-Views mit einigen Ausnahmen behandelt. Sie bestehen darin, dass die Aktualisierung der Darstellung nicht durch das Überwachen des Modells, sondern vom entsprechenden Controller direkt nach der Änderung am Modell erfolgt

Methoden

PoiListV(list, c): Diese Methode ist ein Konstruktor, sie nimmt die durch *list* und *c* bestimmten Liste und Controller *PoiC* entgegen und bindet Methoden des Controllers an die entsprechenden UI-Elemente. Dabei ist *list* eine einfache Listen von Titeln und IDs der POIs.

display(): Lässt die View die Liste entsprechend darstellen.

PoiC Diese Klasse ist ein POI-Controller. Sie ist dafür zuständig hinsichtlich der POIs auf Benutzereingaben zu reagieren, Abläufe zu steuern und Entscheidungen zu treffen.

Methoden

createPoi(poi): Erstellt ein leeres oder mit Daten aus *poi* initialisiertes POI-Objekt, eine entsprechende View dafür und lässt es von dieser View zum Bearbeiten darstellen. Beim Speichern in der Persistenzschicht werden Objekte ohne id als neue abgelegt.

deletePoi(id): Lässt das POI-Modell einen durch gegebene *id* bestimmten POI in der Persistenzschicht löschen. Dabei sorgt der Controller selbst für die Aktualisierung der Listen-View.

editPoi(id): Lässt das POI-Modell einen durch gegebene *id* bestimmten POI instanziieren, erstellt eine View dafür und lässt das POI-Objekt von dieser View zum bearbeiten darstellen.

updateModel(): Übernimmt die Änderungen aus der View in das Datenmodell.

viewPoi(id): Lässt das POI-Modell einen durch gegebene *id* bestimmten POI instanziieren, erstellt eine View dafür und lässt das POI-Objekt von dieser View zur Ansicht darstellen.

listPois(): Lässt das POI-Modell eine Liste aller POIs von der Persistenzschicht holen, erstellt eine Listen-View dafür und lässt diese View die Liste darstellen.

addMedia(t): Lässt die Klasse *Capture* die Kamera des Smartphones öffnen, je nach Typ *t* entsprechende aufnahmen tätigen und das Ergebnis zurück geben, welches dem POI hinzugefügt wird. Dabei werden die Daten der Mediendatei direkt in der Persistenzschicht gespeichert und beim POI-Modell lediglich die Liste aktualisiert.

deleteMedia(id):

Media Diese Klasse ist nur ein Datenobjekt. Sie enthält nur Metadaten über die Mediendateien und hat keine Funktionalität. Sie ist ein Bestandteil des POI-Datenmodells.

Attribute

name : string: Eindeutiger Dateiname, welcher üblicherweise eine laufende Nummer oder Zeitstempel enthält.

fullPath : string: Absoluter Pfad auf Dateisystemebene.

size : number: Dateigröße in Bytes.

type : string: MIME-Type der Datei, z. B. "image/jpeg".

Capture Diese Klasse greift auf die entsprechenden Schnittstellen der Plattform zu, um Medien in Form von Bild und Video aufzunehmen und auf Dateisystemebene zu speichern.

Methoden

getPicture(): Öffnet die Kamera zur Bildaufnahme, speichert und gibt das Ergebnis zurück.

getVideo(): Öffnet die Kamera zur Videoaufnahme, speichert und gibt das Ergebnis zurück.

Technologieauswahl und deren Auswirkungen Es wurde eine Entscheidung getroffen die Anwendung mithilfe eines Frameworks *Cordova / PhoneGap* zu entwickeln. Dieses Framework bietet eine Möglichkeit Anwendungen mithilfe von HTML5, CSS und JavaScript zu entwickeln und diese als Hybride Applikationen auf verschiedenen Plattformen laufen zu lassen.

Aufgrund dieser Entscheidung und unter Berücksichtigung der in diesem Gebiet etablierten Entwurfsmuster (Trice, 2013) wurde eine weitere Entscheidung getroffen die Anwendung als Single Page App zu entwerfen. Dabei wird die darunterliegende Webseite beim Start der Anwendung ein einziges Mal geladen und weiterhin nur mithilfe von JavaScript und CSS modifiziert.

Die Entscheidung über die Auswahl des Frameworks wirkt sich außerdem noch sehr auf den Programmierstil aus, da das Framework asynchron konzipiert wurde und aus dem Grund fast überall Callback-Funktionen benötigt werden, um das Ergebnis zurückgeben zu können.

Im folgenden Kapitel wird das dynamische Verhalten der Anwendung anhand mehrerer Sequenzdiagramme erläutert.

Sequenzdiagramme

Im Folgenden sind mehrere Sequenzdiagramme vorgestellt, mit deren Hilfe das dynamische Verhalten der Anwendung exemplarisch anhand der wesentlichen Anwendungsfälle etwas detaillierter erläutert wird.

Um die Anzahl der beteiligten Klassen bzw. Objekte in einem Diagramm und somit dessen Größe in Grenzen zu halten, wurden die entsprechenden Abläufe in mehrere Aktionen unterteilt, sodass jede auf einem Teildiagramm einzeln abgebildet werden kann.

Die nachfolgenden drei Teildiagramme 4.5, 4.6 und 4.7 zeigen eine Sequenz, die beim Aufzeichnen einer Tour ausgeführt wird. Anschließend wird jeder Aufruf nochmal im Detail erläutert.

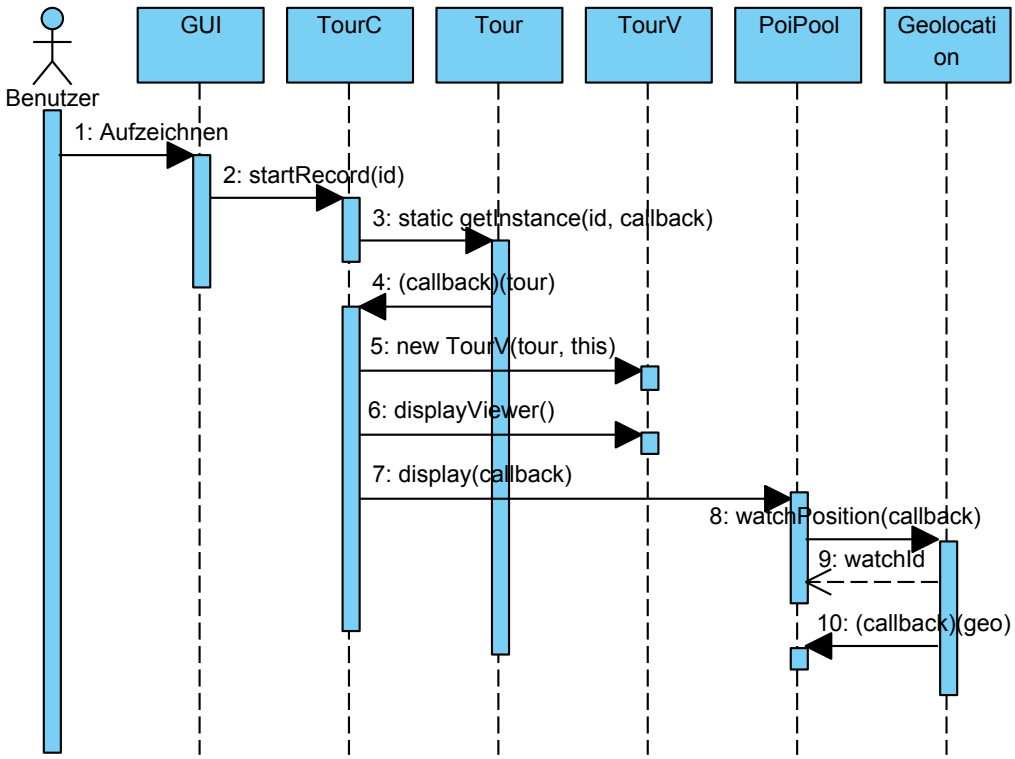


Abbildung 4.5: Sequenzdiagramm - Aufzeichnung starten

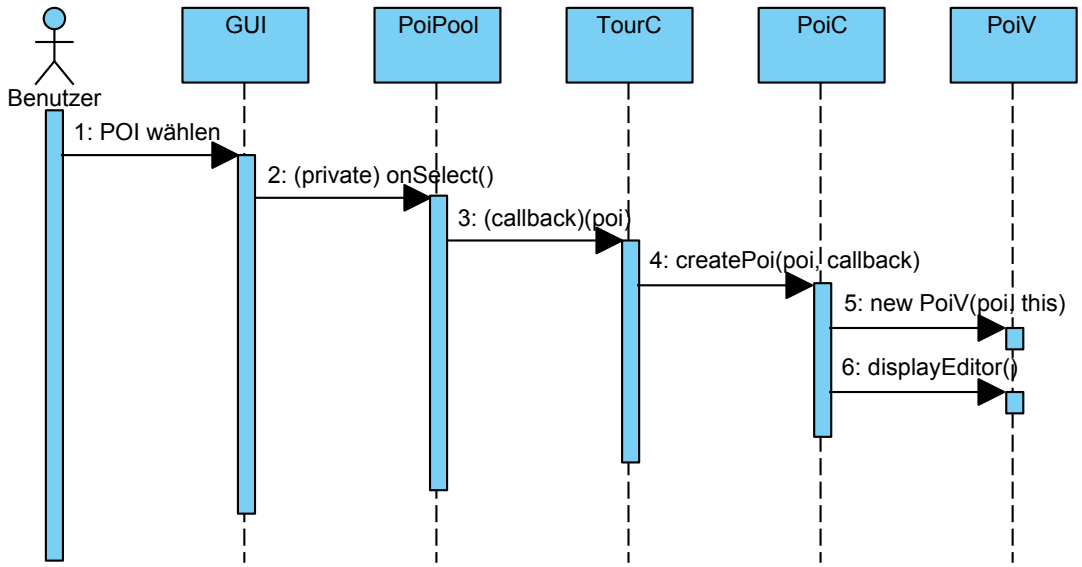


Abbildung 4.6: Sequenzdiagramm - Aufzeichnung: POI auswählen

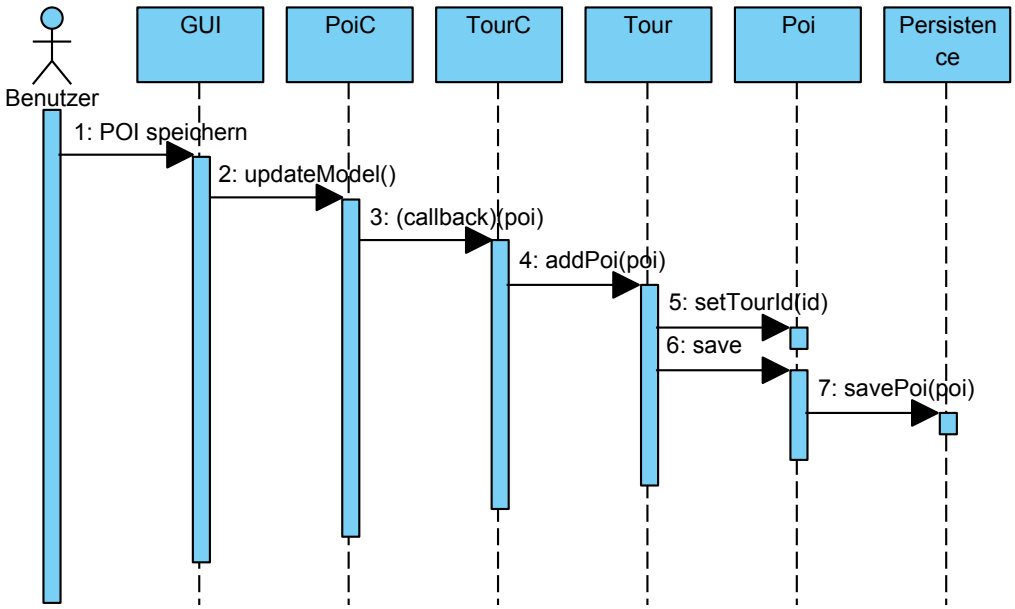


Abbildung 4.7: Sequenzdiagramm - Aufzeichnung: POI speichern

In den Teildiagrammen 4.5, 4.6 und 4.7 sind folgende Schritte abgebildet:

Teildiagramm 4.5: Aufzeichnung starten

1. Aufzeichnen: Der Benutzer startet die Aufzeichnung für eine Tour.

2. startRecord(id): Der vom Benutzer angeklickte Button ruft eine Methode *startRecord* des Tour-Controllers *TourC* auf und übergibt ihr die ID der gewählten Tour. Diese Methode wechselt unter Anderem auch zur entsprechenden Ansicht.

3. getInstance(id, callback): Der Tour-Controller *TourC* ruft eine statische Methode des Tour-Modells *Tour* auf, die den entsprechenden Eintrag in der Persistenzschicht sucht, ein Tour-Objekt instanziiert und mithilfe der gegebenen Callback-Funktion zurückgibt. Dabei wird die Methode *getTour(id)* der Klasse *Persistence* mit einer ähnlichen Callback-Funktion aufgerufen. Dieser Schritt wurde aus Platzgründen ausgelassen.

4. (callback)(tour): Das im Schritt 3 instanziierte Tour-Objekt wird mithilfe der gegebenen Callback-Funktion zurückgegeben.

5. new TourV(tour, this): In diesem Schritt wird eine Tour-View erstellt. Dies ist ein Aufruf der Konstruktor-Methode von *TourV*. Dieser Konstruktor kriegt das Modell-Objekt *tour* und einen Verweis auf den Controller *this* mit. Er richtet einen Observer für das Modell ein und bindet einige Methoden des Controllers an die entsprechenden UI-Elemente.

6. displayViewer(): In diesem Schritt wird die View *TourV* angewiesen die entsprechende Tour auf der Benutzeroberfläche zur Ansicht darzustellen.

7. display(callback): Hier wird der *PoiPool* angewiesen eine Kartenansicht mit einer Liste der in der Nähe liegenden POIs anzuzeigen. Diese Methode kriegt eine Callback-Funktion mit, die bei der Auswahl eines POIs dafür sorgen wird, dass seine Daten an den Tour-Controller *TourC* zurückgegeben werden.

8. watchPosition(callback): Dieser Aufruf registriert eine Callback-Funktion bei *Geolocation*, sodass dem *PoiPool* immer die aktuelle Position mitgeteilt wird.

9. watchId: Gibt eine ID zurück, mit welcher die Callback-Funktion registriert wurde und später abgemeldet werden muss.

10. (callback)(geo): Diese Callback-Funktion wird periodisch aufgerufen und auf diesem Wege die aktuelle Position dem *PoiPool* mitgeteilt.

Teildiagramm 4.6: POI auswählen

1. **POI wählen:** Der Benutzer wählt aus der Karte bzw. der Liste einen POI aus, welcher der Tour hinzugefügt werden soll.

2. **(private) onSelect():** Dies ist eine vom *PoiPool* an jeden angezeigten POI gebundene private Methode, die bei der Auswahl eines POIs ausgeführt wird. Sie besorgt z. B. den Titel, die Beschreibung und die Geokoordinaten des POIs und übergibt diese Daten mithilfe der im Schritt 7 des vorherigen Teildiagramms definierten Callback-Funktion an den Tour-Controller *TourC* zurück.

3. **(callback)(poi):** Übergibt die Daten des gewählten POIs mithilfe einer Callback-Funktion an den Tour-Controller zurück.

4. **createPoi(poi, callback):** Der POI-Controller *PoiC* wird angewiesen einen neuen POI zu erstellen, der ggf. mit den durch *poi* definierten Daten vorbelegt wird. Außerdem wird hier eine Callback-Funktion übergeben, die es ermöglichen Soll denn gespeicherten POI an den Tour-Controller *TourC* zurück zu geben.

5. **new PoiV(poi, this):** In diesem Schritt wird eine View erstellt, deren Konstruktor das Modell *poi* und den Verweis auf den Controller *this* mitbekommt. Sie richtet einen Observer am Modell ein und bindet einige Methoden des Controllers an die entsprechenden UI-Elemente.

6. **displayEditor():** Dieser Aufruf lässt die View *PoiV* den POI in einem Formular zum Bearbeiten darstellen.

Teildiagramm 4.7: POI speichern

1. **POI speichern:** Der Benutzer hat die Bearbeitung des neu angelegten bzw. übernommenen POIs abgeschlossen und gespeichert.

2. **updateModel():** Der vom Benutzer angeklickte Button ruft eine Methode *updateModel* des POI-Controllers *PoiC* auf, welche die Daten aus dem Formular liest, das Modell-Objekt aktualisiert und es anschließend mithilfe der im Schritt 4 des vorherigen Teildiagramms definierten Callback-Funktion an den Tour-Controller *TourC* zurückgibt.

3. **(callback)(poi):** Der gespeicherte POI wird mithilfe einer Callback-Funktion an den Tour-Controller *TourC* zurückgegeben.

4. **addPoi(poi):** Der im Schritt 3 zurückgegebene POI wird einer Tour hinzugefügt. Hier wird die POI-Liste der Tour aktualisiert.

5. **setTourId(id):** In diesem Schritt setzt das Tour-Modell seine ID am POI-Modell, um die Zugehörigkeit des POIs sicherzustellen.

6. **save():** Dieser Aufruf veranlasst den POI sich selbst persistent zu speichern.

7. **savePoi(poi):** Durch diesen Aufruf lässt der POI die Persistenzschicht seine durch *poi* gegebenen Daten speichern.

Die Nachfolgenden zwei Teildiagramme 4.8 und 4.9 zeigen eine Sequenz, die bei einer Tourführung ausgeführt wird. Anschließend wird jeder Aufruf nochmal im Detail erläutert.

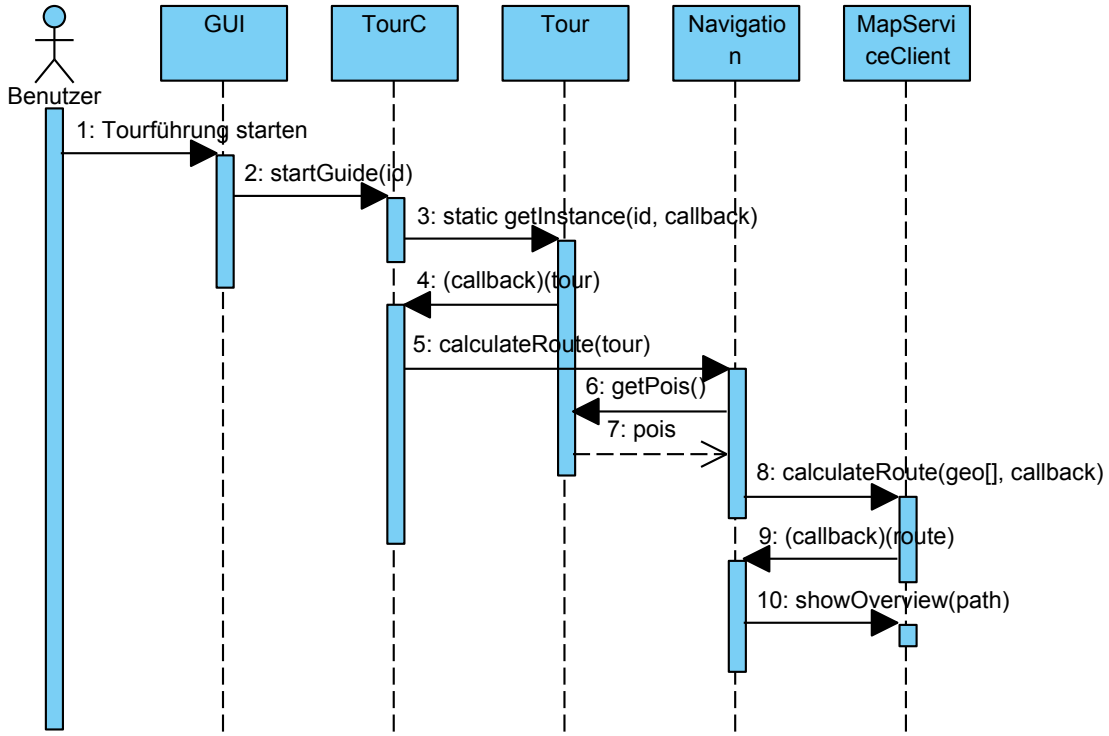


Abbildung 4.8: Sequenzdiagramm - Tourführung starten

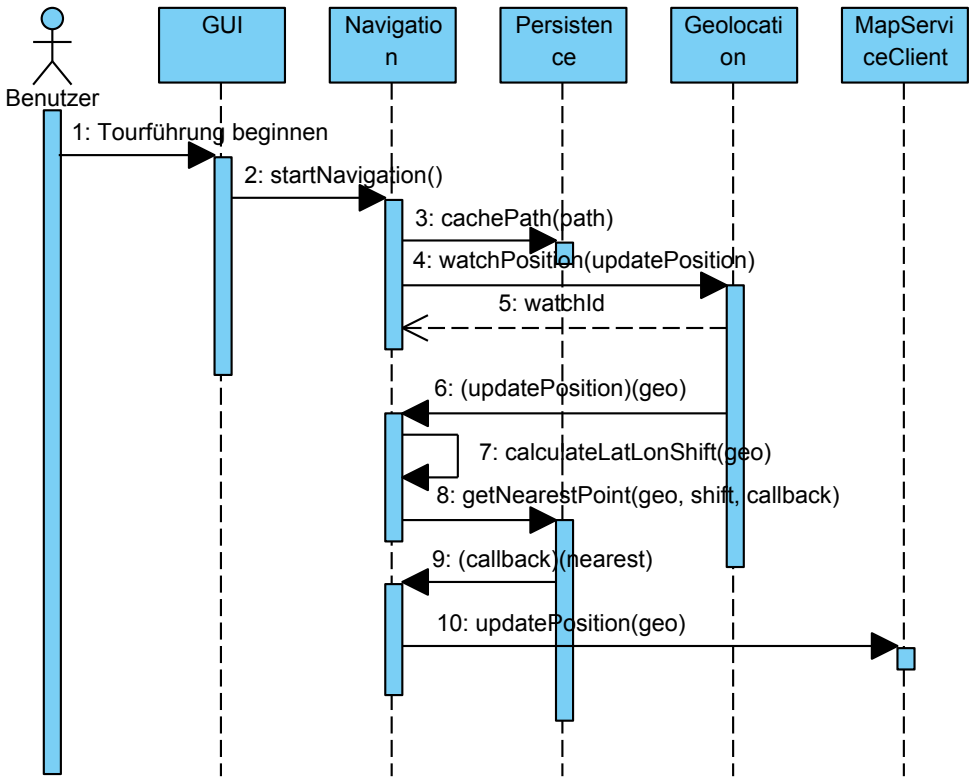


Abbildung 4.9: Sequenzdiagramm - Tourführung beginnen

In den Teildiagrammen 4.8 und 4.9 sind folgende Schritte abgebildet:

Teildiagramm 4.8: Tourführung starten

1. Tourführung starten: Der Benutzer startet die Führung mit einer Tour.

2. startGuide(id): Der vom Benutzer angeklickte Button ruft eine Methode *startGuide* des Tour-Controllers *TourC* auf und übergibt ihr die ID der gewählten Tour. Diese Methode wechselt unter Anderem auch zur entsprechenden Ansicht.

3. static getInstance(id, callback): Dieser Aufruf lässt das Tour-Modell ein Tour-Objekt instanziieren und mit der gegebenen Callback-Funktion im nächsten Schritt zurückgeben. Dabei wird die Methode *getPoi(id)* der Klasse *Persistence* mit einer ähnlichen Callback-Funktion aufgerufen, um die entsprechenden Daten von der Persistenzschicht zu holen. Dieser Schritt wurde aus Platzgründen ausgelassen.

4. **(callback)(tour):** Das im Schritt 3 instanziierte Tour-Objekt wird mithilfe der gegebenen Callback-Funktion zurückgegeben.

5. **calculateRoute(tour):** Veranlasst die Klasse *Navigation* eine Route über die gegebene Tour mithilfe von *MapServiceClietn* berechnen zu lassen.

6. **getPois():** Gibt eine Liste der POIs zurück, welche lediglich aus Titeln, IDs und Geokoordinaten besteht. Diese Daten werden benötigt, um die Route zu berechnen und die POIs auf der Karte anzuzeigen.

7. **pois:** Gibt die POI-Liste zurück.

8. **calculateRoute(geo[], callback):** Lässt den *MapServiceClient* eine Route über die gegebenen Geokoordinaten berechnen und im nächsten Schritt mithilfe der mitgegebenen Callback-Funktion zurückgeben.

9. **(callback)(route):** Gibt die berechnete Route mithilfe der Callback-Funktion zurück.

10: **showOverview(path):** Lässt den *MapServiceClient* eine Übersicht des eben berechneten Pfades auf der Karte anzeigen.

Teildiagramm 4.9: Tourführung beginnen

1. **Tourführung beginnen:** Der Benutzer hat sich die Übersicht der Tour angeschaut und beginnt nun die Tourführung.

2. **startNavigation():** Lässt die Klasse *Navigation* in den nächsten Schritten eine Tourführung vorbereiten.

3. **cachePath(path):** Dieser Aufruf zwischenspeichert den berechneten Pfad in der Persistenzschicht, um später nach den einzelnen Pfadpunkten effizient suchen zu können.

4. **watchPosition(updatePosition):** In diesem Schritt meldet *Navigation* ihre Methode *updatePosition* bei *Geolocation* als Callback-Funktion an, sodass sie jedes Mal aufgerufen wird, wenn sich die aktuelle Position oder die Ausrichtung des Gerätes ändert. Sie bekommt bei jedem Aufruf die aktuelle Position als Parameter mit.

5. **watchId:** Es wird eine ID zurückgegeben, mit welcher die eben angemeldete Methode anschließend abgemeldet werden muss.

6. **(updatePosition)(geo):** Diese Callback-Funktion wird vom System jedes Mal aufgerufen, wenn sich die geografische Position oder Ausrichtung des Gerätes ändert. Das Intervall beträgt in der Regel eine bis zwei Sekunden. Sie ist der Einstiegspunkt für alle anderen, von der Position abhängigen Berechnungen.

7. **calculateLatLonShift(geo):** In diesem Schritt werden Distanzen in Metern berechnet, welche ein Breiten- bzw. ein Längengrad auf der Erdoberfläche nahe der gegebenen Position beschreibt. Diese werden benötigt, um Entfernungen zu den POIs oder anderen Punkten des zwischengespeicherten Pfades zu berechnen.

8. **getNearestPoint(geo, shift, callback):** Mit diesem Aufruf wird nach einem zu *geo* am nächsten Liegenden POI oder anderen Punkten im zwischengespeicherten Pfad gesucht. Diese sind wichtig, um sich am Pfad orientieren zu können. Diese Funktion bekommt eine durch *geo* bestimmte Position, durch *shift* bestimmten Distanzen in Metern pro Breiten- bzw. Längengrad, und eine Callback-Funktion, mit welcher das Ergebnis zurückgegeben wird.

9. **(callback)(nearest):** Mit dem Aufruf dieser Callback-Funktion wird der am nächsten liegende Punkt zurückgegeben.

10. **updatePosition(geo):** In diesem Schritt wird die Karte zur aktuellen Position *geo* zentriert und aktualisiert.

Folgendes Kapitel befasst sich mit einer prototypischen Realisierung und geht auf einige damit verbundene Probleme und deren Lösungen ein.

4.3 Prototypische Implementierung

Dieses Kapitel befasst sich mit einer prototypischen Realisierung der Anwendung. Hier wird über die gewählten Technologien und aufgetretene Probleme diskutiert, außerdem werden einige grundsätzliche Entscheidungen erläutert.

Zunächst wird auf den Stand der Realisierung eingegangen. In den darauffolgenden Kapiteln werden einige Lösungen im Umgang mit dem Framework allgemein, mit den Onlinekartendiensten und mit der Persistenzschicht vorgestellt. Anschließend werden einige Probleme und deren Lösungen aufgeführt.

Stand der Realisierung

Die Implementierung befindet sich aktuell auf dem Stand einiger Fallstudien. Dies ist unter Anderem darauf zurückzuführen, dass sich Cordova / PhoneGap (das Framework) selbst noch in einer Entwicklungsphase befindet und während der Bearbeitungszeit Schnittstellen geändert wurden und teilweise Fehler im Framework aus eigener Kraft behoben werden mussten. Darüber hinaus gab es einige Funktionalitäten, die auf einigen der Plattformen nur partiell oder gar nicht unterstützt wurden. Dies betrifft die Plugins, die einer beliebigen modularen Erweiterung der Kernfunktionalität des Frameworks dienen. Darunter sind z. B. *Media Capture* und *File*. Auf die einzelnen Probleme wird im Kapitel 4.3.4 etwas näher eingegangen.

Das Ziel dieser Fallstudien war die entsprechenden Technologien kennen zu lernen und somit unter Anderem folgende Fragen aus dem Weg zu räumen:

1. Wie organisiert man am besten das Benutzerinterface bzw. die Darstellung auf einem Smartphone-Bildschirm?
2. Wie benutzt man die Kamera?
3. Wie behandelt man die Dateien?
4. Wie benutzt man die Datenbankschnittstelle?
5. Wie benutzt man den Onlinekartendienst, den Routenplaner und die Karte?
6. Wie kann man ein Navigationsgerät nachbauen, um den Benutzer im Straßennetz sicher zum Ziel zu führen?

Einige dieser Fragen werden in den nachfolgenden Kapiteln behandelt und die entsprechenden Lösungen vorgestellt.

4.3.1 Cordova / PhoneGap

Wie jedes andere Framework, hat auch Cordova / PhoneGap seine eigene Besonderheiten.

Im Folgenden wird mithilfe einiger interessanten Beispiele gezeigt, wie einige Aufgaben mit diesem Framework erledigt werden.

Kamera

Um ein Video oder ein Bild aufzunehmen wird eine der beiden capture-Methoden verwendet, die vom Framework zur Verfügung gestellt werden. Diese Methoden werden asynchron aufgerufen und brauchen entsprechende Callback-Funktionen, um das Ergebnis zurück zu geben oder Fehler zu melden.

Das Listing 4.1 zeigt einen vereinfachten Beispiel, mit dem eine Serie von Bildern bzw. Videos aufgenommen und an eine Callback-Funktion übergeben wird.

```
1  function getPicture() {
2      navigator.device.capture.captureImage(onSuccess, onError);
3  }
4
5  function getVideo() {
6      navigator.device.capture.captureVideo(onSuccess, onError);
7  }
8
9  function onSuccess(files) { // callback
10     for (var i = 0; i < files.length; i++) {
11         // do something useful with them
12         var file = files[i];
13         // ...
14     }
15 }
```

Listing 4.1: Cordova / PhoneGap - Kamera

Wie das Listing 4.1 zeigt wird in der Zeile 2 die Methode *captureImage* für eine Bildaufnahme aufgerufen. Sie bekommt zwei Callback-Funktionen. Diese Methode öffnet die Kamera, sodass der Benutzer mehrere Bilder aufnehmen und sie anschließend entweder übernehmen oder verwerfen kann.

Im Erfolgsfall wird die in der Zeile 9 definierte Funktion aufgerufen und ihr ein Array mit den File-Objekten übergeben. Ein File-Objekt ist ein einfaches JavaScript-Objekt. Es enthält unter Anderem den Dateinamen, den kompletten Pfad auf Dateisystemebene, den MIME-Typ der Datei und andere Informationen. Diese Methode ruft üblicherweise eine weitere, vom Aufrufer definierte Callback-Funktion, um das Ergebnis zurück zu geben. Die Methode *captureVideo* in der Zeile 6 funktioniert identisch.

Im Fehlerfall, oder wenn der Benutzer sich entscheidet abzubrechen, wird die Callback-Funktion *onError* mit einem Error-Objekt als Parameter aufgerufen, deren Definition hier aus Platzgründen ausgelassen wurde. Mithilfe des Error-Objekts kann entschieden werden, ob der Benutzer abgebrochen hat oder ein Fehler passiert ist.

Geolocation

Obwohl die Geolocation-Schnittstelle mittlerweile von allen modernen Browsern nativ unterstützt wird, wurde bis vor Kurzem noch eine eigene Nachimplementierung vom Framework bereitgestellt. Ein Grund dafür scheint wohl die Abwärtskompatibilität zu den älteren Plattformen zu sein.

Geolocation ist wie einige anderen Schnittstellen auch Event-Basiert und benötigt ebenfalls eine Callback-Funktion, um die aktuelle Position des Gerätes zurückgeben zu können. Dabei gibt es zwei Möglichkeiten.

Bei einer einmaligen Abfrage wird die Methode *getCurrentPosition* aufgerufen, sodass beim Erfolg die Callback-Funktion mit dem Ergebnis als Parameter aufgerufen wird.

Bei der Überwachung der aktuellen Position wird die Methode *watchPosition* aufgerufen, sodass beim Erfolg die Callback-Funktion jedoch solange periodisch aufgerufen wird bis es explizit beendet wird. Üblicherweise beträgt das Intervall zwischen den einzelnen Aufrufen eine bis zwei Sekunden.

Hier gibt es einen Unterschied. Da an der Überwachung der aktuellen Position mehrere Anwendungen oder Prozesse interessiert sein können, wird hier ein Abo-ähnliches Verfahren verwendet. Dabei meldet jeder Prozess seine Callback-Funktion bei Geolocation an und erhält eine eindeutige ID, mit der diese anschließend explizit abgemeldet wird. Das Ortungsmodul des Gerätes wird heruntergefahren, sobald die letzte Callback-Funktion abgemeldet wurde.

Das nachfolgende Listing 4.2 zeigt einen vereinfachten Beispiel dieser zwei Methoden zur Abfrage der aktuellen Position.

```
function getPosition(callback) {
    navigator.geolocation.getCurrentPosition(function(pos) {
        // do something useful with them
        callback(pos.coords.latitude, pos.coords.longitude);
    }, onError, options);
}

function watchPosition(callback) {
    var watchId = navigator.geolocation.watchPosition(function(pos) {
        // do something useful with them
        callback(pos.coords.latitude, pos.coords.longitude);
    }, onError, options);
    return watchId;
}

function clearWatch(watchId) {
    navigator.geolocation.clearWatch(watchId);
}
```

Listing 4.2: Cordova / PhoneGap - Geolocation

Wie das Listing 4.2 zeigt wird in der Zeile 2 die Methode *getCurrentPosition* aufgerufen. Diese bekommt als Parameter zwei Callback-Funktionen und ein Options-Objekt. Im Erfolgsfall wird die in der selben Zeile inline definierte Callback-Funktion aufgerufen und bekommt das Position-Objekt als Parameter mit. In einem Fehlerfall wird die Callback-Funktion *onError* mit einem Error-Objekt als Parameter aufgerufen, deren Definition hier aus Platzgründen ausgelassen wurde.

Das Options-Objekt enthält einige Optionen wie Timeouts, maximal akzeptierten Alter der Position und einige anderen Optionen.

In der Zeile 9 wird die Methode *watchPosition* aufgerufen. Sie erwartet zwar die selben Argumente, deren Callback-Funktion wird im Erfolgsfall jedoch periodisch ausgeführt. Außerdem gibt sie eine *watchId* zurück, mit der diese identifiziert wird.

Die in der Zeile 9 von der Methode *watchPosition* zurückgegebene *watchId* wird in der Zeile 17 verwendet, um die Überwachung der aktuellen Position zu beenden. Dazu wird die Methode *clearWatch* mit der entsprechenden ID als Parameter aufgerufen.

Hardwarebuttons / Events

Es ist üblich, dass die Anwendungen eines Smartphones auf Hardwarebuttons oder andere Events wie z. B. eingehender Anruf reagieren müssen. Die Cordova / PhoneGap Anwendungen sind dabei auch keine Ausnahme.

Da es sich hierbei um Webanwendungen handelt, muss sichergestellt werden, dass sie auf die entsprechenden Events auch reagieren können. Dies ist mithilfe der in der Webentwicklung üblichen Events und Event-Handler realisiert. Dabei sind die Event-Handler nichts anderes, als die bereits bekannten Callback-Funktionen, die beim Auslösen der entsprechenden Events aufgerufen werden.

Das nachfolgende Listing 4.3 zeigt einige vereinfachten Beispiele dazu.

```
1  document.addEventListener('deviceready', function() {
2      // general events
3      document.addEventListener('pause', function() {
4          // save state
5      }, false);
6      document.addEventListener('resume', function() {
7          // restore state
8      }, false);
9      // other events:
10     // batterycritical, batterylow, batterystatus
11     // online, offline
12
13     // button events
14     document.addEventListener('backbutton', function() {
15         // override back button
16     }, false);
17     document.addEventListener('menubutton', function() {
18         // override menu button
19         menu.classList.toggle('open');
20     }, false);
21     // other buttons:
22     // searchbutton, startcallbutton, endcallbutton
23     // volumedownbutton, volumeupbutton
24 }, false);
```

Listing 4.3: Cordova / PhoneGap - Hardwarebuttons / Events

Wie das Listing 4.3 zeigt wird in der Zeile 1 ein Event-Handler für das *deviceready*-Event registriert. Dieses ist einem in der Webentwicklung üblichen *load*-Event ähnlich und wird ausgelöst, sobald das Framework vollständig initialisiert wurde. Dadurch wird sichergestellt, dass keine der API-Methoden aufgerufen werden kann, bevor das Framework bereit ist. Hier sollte die Initialisierung der Anwendung ablaufen.

In der Zeile 3 wird ein Event-Handler für das *pause*-Event registriert. Dieses wird ausgelöst, wenn die Anwendung in den Hintergrund wechselt. Dieser Handler sichert üblicherweise den Zustand der Anwendung, sodass der Benutzers sie bei seiner Rückkehr im selben Zustand wiederfindet.

Die Zeile 6 zeigt wie ein *resume*-Event behandelt wird. Es ist das Gegenteil zum *pause*-Event und wird ausgelöst, sobald die Anwendung wieder in den Vordergrund geholt wird. Hier wird üblicherweise deren Zustand wiederhergestellt.

Mit dem selben Prinzip werden auch einige anderen Events behandelt, auf die hier jedoch aus Platzgründen nicht weiter eingegangen wird.

In der Zeile 14 wird definiert, wie die Anwendung das Betätigen der Rücktaste, des sogenannten Back-Buttons, behandeln soll. Mit dem Back-Button wird die Anwendung z. B. auf einem Android-Smartphone üblicherweise komplett geschlossen. Dieses Verhalten kann somit übersteuert werden, sodass z. B. nur ein Fenster geschlossen oder die aktuelle Operation abgebrochen wird.

Der Event-Handler in der Zeile 17 zeigt ein reelles Beispiel, wie die Anwendung auf das Betätigen des Menü-Buttons reagiert. Hier wird in der Zeile 19 die Darstellung eines HTML-Elements geändert, welches das Menü enthält und von der Variable *menu* referenziert wird. Dabei wird dem entsprechenden HTML-Element die CSS-Klasse *open* abwechselnd hinzugefügt und entfernt.

Mit dem selben Prinzip kann das Verhalten anderer Hardwarebuttons übersteuert werden, auf die hier ebenfalls aus Platzgründen nicht weiter eingegangen wird.

4.3.2 Onlinekartendienste

Onlinekartendienste werden mithilfe eines Adapters angeschlossen, der die Schnittstelle *MapServiceClient* implementiert. Da es in JavaScript keine Schnittstellen als solche gibt, muss der Entwickler selbst dafür sorgen, dass sein Adapter dieser Schnittstelle entspricht.

Im Folgenden wird eine exemplarische Implementierung des Adapters für Google Maps anhand einiger Listings vorgestellt und die wichtigsten Momente erläutert.

Google stellt eine Bibliothek zur Verfügung, mit der eine Karte erstellt, gestaltet und manipuliert, sowie anfragen an den Routenplaner gesendet werden können. Diese Bibliothek wird in Form einer externen JavaScript-Datei einfach im HTML-Code geladen und kann ohne weiteres verwendet werden.

Karte

Um eine neue Karte anzuzeigen, muss ein *Map*-Objekt erstellt werden. Der Konstruktor erwartet unter Anderem eine HTML-ID des DIV-Containers, in dem die Karte angezeigt werden soll, und ein Options-Objekt mit den Koordinaten, der gewünschten Zoom-Stufe und ggf. anderen Optionen. Um Orte zu markieren, werden Marker mit verschiedenen Icons verwendet.

Das nachfolgende Listing 4.4 zeigt einen stark vereinfachten Beispiel mit einer von Google je nach Bildschirmgröße automatisch gestalteten Karte.

```
1  var map, marker;
2
3  function initMap(divId, latitude, longitude) {
4      // create position object
5      var pos = new google.maps.LatLng(latitude, longitude);
6
7      // create map
8      map = new google.maps.Map(divId, { // options
9          mapTypeId:  google.maps.MapTypeId.ROADMAP,
10         center:     pos,
11         zoom:       zoom
12     });
13
14     // create marker
15     marker = new google.maps.Marker({ // options
16         position:   pos,
17         map:        map,
18         icon:       circle
19     });
20 }
21
22 function updatePosition(latitude, longitude) {
23     // create position object
24     var pos = new google.maps.LatLng(latitude, longitude);
25
26     // move to new position
27     marker.setPosition(pos);
28     map.panTo(pos);
29 }
```

Listing 4.4: Onlinekartendienst - Karte

Wie das Listing 4.4 zeigt wird in der Funktion *initMap* in der Zeile 5 mithilfe der gegebenen Geokoordinaten ein proprietäres Positions-Objekt erstellt. Dieses Objekt wird beim Erstellen eines neuen *Map*-Objekts in der Zeile 8 zusammen mit einigen anderen Optionen in einem inline definierten Options-Objekt verwendet. Wird dem Konstruktor dabei die ID eines HTML-Containers *divId* mitgegeben, so wird die Karte in diesem sofort angezeigt.

In der Zeile 15 wird der Karte ein Marker hinzugefügt. Mithilfe von Markern kann man beliebige Orte auf der Karte markieren. In diesem Beispiel wird der Marker die aktuelle Position des Gerätes anzeigen. Dafür wird dem Konstruktor des Markers ebenfalls ein inline definiertes Options-Objekt mit der Position, der Karte und einem Icon mitgegeben. Er lässt sich sofort auf der ihm mitgegebenen Karte darstellen.

In der Funktion *updatePosition* wird aus den gegebenen Geokoordinaten ebenfalls ein proprietäres Positions-Objekt erstellt und ab der Zeile 27 die Position des Markers und anschließend der Karte aktualisiert.

Routenplaner

Um ein Navigationsgerät nachzubauen, braucht man einen Routenplaner mit Daten. Da hier das Kartenmaterial ja sowieso über das Internet geladen wird, kann auch ein Routenplaner im Netz verwendet werden. Google bietet einen Dienst an, der auf Anfrage eine Route über mehrere Punkte berechnen kann, ggf. auch optimiert.

Um einen Routenplaner benutzen zu können, wird man etwas mehr Aufwand betreiben müssen. Als erstes müssen alle Zwischenziele in eine proprietäre Form gebracht und damit ein Request-Objekt gebaut werden. Anschließend wird ein *DirectionsService*-Objekt erstellt und eine asynchrone Abfrage mit einer Callback-Funktion ausgeführt.

Das nachfolgende Listing 4.5 zeigt einen stark vereinfachten Beispiel wie eine Route berechnet und zurückgegeben wird.

```
1  function calculateRoute(latitude, longitude, pois, callback) {
2      // create position object
3      var origin = new google.maps.LatLng(latitude, longitude);
4
5      // prepare request
6      var waypoints = [];
7      for (var i = 0; i < pois.length; i++) {
8          var poi = pois[i];
9          var pos = new google.maps.LatLng(poi.lat, poi.lon);
10         waypoints.push({ stopover: true, location: pos });
11     }
12     var request = {
13         origin: origin,
14         destination: waypoints.pop().location, // last waypoint
15         waypoints: waypoints,
16         // ...
17     };
18
19     // request route
20     var dirsvc = new google.maps.DirectionsService();
21     dirsvc.route(request, function(result, status) {
22         if (status === google.maps.DirectionsStatus.OK) {
23             // do something useful with the result
24             callback(result);
25         }
26     });
27 }
```

Listing 4.5: Onlinekartendienst - Routenplaner

Wie das Listing 4.5 zeigt wird zuerst die Anfrage vorbereitet. Dazu wird in der Zeile 3 ein proprietäres Positions-Objekt für die Ausgangsposition erstellt. Im nächsten Schritt werden ab der Zeile 6 aus Geokoordinaten der POIs die proprietären Zwischenziel-Objekte erstellt. Nachfolgend wird damit in der Zeile 12 ein Request-Objekt definiert. Dabei wird für das eigentliche Ziel die Position des letzten Zwischenziels benutzt. Das letzte Zwischenziel-Objekt

wird entfernt. Die Zwischenziel-Objekte sind nicht mit den einfachen Positions-Objekten wie Start und Ziel zu verwechseln. Sie sind zusammengesetzte Objekte und enthalten unter Anderem entweder Adressen als String oder Position-Objekte.

Im letzten Schritt wird der Routenplaner initialisiert und in der Zeile 21 die Anfrage ausgeführt. Dabei wird beim Aufruf der entsprechenden Methode zusammen mit dem Request-Objekt eine inline definierte Callback-Funktion mit übergeben. Diese soll das Ergebnis entgegen nehmen, prüfen und anschließend mithilfe einer weiteren, vom Aufrufer definierten Callback-Funktion zurückgeben.

4.3.3 Persistenzschicht

Die Persistenzschicht der Anwendung wird durch eine Datenbank realisiert, diese wird bereits vom Browser inklusive einer standardisierten API bereitgestellt und bietet einen recht performanten Zugriff auf die Daten.

Es handelt sich dabei um den *Web SQL* Standard. In den meisten Fällen wird es durch eine *SQLite 3* Datenbank realisiert. Sollte jedoch das Endgerät diesen Standard nicht unterstützen, so stellt das Framework eigene Implementierung zur Verfügung.

Obwohl das Konzept nicht vom Framework stammt, muss hier auch asynchron mit Callback-Funktionen gearbeitet werden, was in JavaScript allerdings kein Problem darstellt.

Folgendes Listing 4.6 zeigt mit einem stark vereinfachten Beispiel, wie in der Datenbank nach einer bestimmten Tour gesucht wird.

```
1  // open database
2  var db = window.openDatabase(name, ... );
3
4  // create transaction
5  db.transaction(function(tx) { // on success
6
7      var sql  = 'SELECT * FROM touren WHERE id = ?';
8      var args = [id];
9
10     // execute statement
11     tx.executeSql(sql, args, function(tx, result) { // on success
12
13         // do something useful with the result
14         var rows = result.rows;
15         // ...
16
17     }, onQueryErrorCallback);
18 }, onErrorCallback);
```

Listing 4.6: Persistenzschicht - Web SQL Schnittstelle

Wie das Listing 4.6 zeigt müssen bei einer einzigen Datenbankabfrage schon mehrmals verschachtelte Callback-Funktionen benutzt werden. In diesem Beispiel werden dafür zwei inline definierte Funktionen direkt als Parameter beim Aufruf verwendet.

Bei einer Abfrage muss als Erstes eine Transaktion erstellt werden. Eine Referenz darauf wird in Zeile 5 einer inline definierten Callback-Funktion als Parameter *tx* mitgegeben. Die Lebenszeit der Transaktion entspricht der Ausführungszeit dieser Callback-Funktion. Die Transaktionen können auch ineinander verschachtelt werden.

Das mit *tx* referenzierte Objekt bietet eine Schnittstelle zum Absetzen der SQL-Abfragen. Jeder Aufruf an diesem Objekt wird innerhalb derselben Transaktion ausgeführt.

Um eine SQL-Abfrage auszuführen, muss an diesem Objekt die Methode *tx.executeSql* aufgerufen werden. Sie benötigt als Parameter den SQL-String, ggf. ein Array mit den Werten für dessen Platzhalter und eine weitere Callback-Funktion (Zeile 11), die beim Aufruf die selbe Transaktionsreferenz und das Ergebnis *result* bekommt.

In der Zeile 14 wird mit *result.rows* auf die Liste bzw. das Array mit den einzelnen Elementen zugegriffen. Diese sind einfache JavaScript-Objekte, deren Attribute den Tabellenspalten entsprechen und auch entsprechende Werte enthalten. Hier wird üblicherweise eine weitere, vom Aufrufer definierte Callback-Funktion aufgerufen, um das Ergebnis zurück zu reichen.

Im Fehlerfall wird, je nachdem wo der Fehler aufgetreten ist, eine der beiden on error Callback-Funktionen aufgerufen und ihr ein Error-Objekt übergeben.

Aufgrund dessen, dass im Rahmen dieser Arbeit vor der Persistenzschicht der Anwendung eine eigene Fassade eingesetzt wird, könnte man auch andere Persistenzmechanismen benutzen. Dazu müsste die Fassade jedoch angepasst werden.

4.3.4 Probleme

In diesem Kapitel werden einige aufgetretene Probleme und deren Lösungen erläutert. Die meisten davon hängen damit zusammen, dass sich das Framework noch in der Entwicklungsphase befindet.

Mangelnde Schreibrechte für die Kamera

Nach einem der Updates des Frameworks funktionierte die Kamera nicht mehr, dementsprechend konnten keine Bilder bzw. Videos aufgenommen werden. Eine Fehlermeldung deutete auf fehlende Schreibrechte im Cache-Verzeichnis der Anwendung für die Kamera hin.

Das Problem ist dadurch entstanden, dass sich das vom Framework definierte Zielverzeichnis für die Kamera, in dem das aufgenommene Bild bzw. Video für die Übergabe an die Anwendung gespeichert werden soll, geändert hat. Das Framework hat nun ein privates Verzeichnis der Anwendung an die Kamera übergeben, in dem der Kameraprozess keine Schreibrechte hat.

Dieses Problem konnte nur durch eine Änderung des nativen Teils des Frameworks gelöst werden. Dabei wurde die entsprechende Stelle im nativen Code auf eine Vorgängerversion zurückgesetzt.

Fehlerhafte Nachimplementierung von Geolocation

In den früheren Versionen hat das Framework eine eigene Nachimplementierung der Geolocation-Schnittstelle bereitgestellt. Ein Grund dafür scheint die Abwärtskompatibilität zu den älteren Plattformen zu sein. Diese Implementierung war fehlerhaft und hat die vom Browser nativ bereitgestellte Schnittstelle überschrieben.

Das Problem bestand darin, dass die mit der Methode *watchPosition* angemeldete Callback-Funktion trotz Angaben im Options-Objekt nur alle 60 Sekunden aufgerufen wurde. Eine Recherche hat ergeben, dass das Options-Objekt im nativen Teil des Frameworks gar nicht beachtet wurde. Stattdessen war ein Intervall von 60000 Millisekunden fest programmiert.

Dieses Problem konnte vorerst nur temporär durch eine Änderung des nativen Teils des Frameworks gelöst werden. Nach einiger Zeit gab es ein Update, das die Nachimplementierung komplett entfernt hat, sodass die vom Browser nativ bereitgestellte Schnittstelle bzw. Implementierung wieder zum Tragen kam.

Menübutton funktionierte nicht ordnungsgemäß

Das Verhalten des Menübuttons war nicht ordnungsgemäß. Die Anwendung reagierte nicht auf jede Berührung des Menübuttons, wie erwartet, sondern nur auf jede zweite.

Das Problem bestand darin, dass der *menubutton*-Event nur bei jeder ungeraden Berührung des Menübuttons ausgelöst wurde. Es ist möglicherweise darauf zurückzuführen, dass der native Teil des Frameworks für Android, die Klasse *CordovaActivity*, versucht hat den entsprechenden Aufruf an seine Super-Klasse weiter zu reichen und somit ein natives Menü zu öffnen bzw. zu schließen, was zu diesem Verhalten führte. Dies wurde aus Zeitgründen nicht einer

genauen Recherche unterzogen, da eine Lösung ganz schnell gefunden wurde.

Dieses Problem konnte durch eine Änderung des nativen Teils des Frameworks gelöst werden. Der entsprechende Aufruf wurde auskommentiert, sodass nicht mehr versucht wird das native Menü zu öffnen.

Öffnen von Medien mithilfe der nativen Apps

Es ist bis jetzt nicht ohne weiteres möglich die Mediendateien mithilfe der dafür vorgesehenen nativen Apps wie z. B. Galerie oder Videoplayer öffnen zu lassen.

Das Problem besteht darin, dass es keine allgemeingültige Lösung gibt, die es erlaubt die Mediendateien mithilfe der nativen Apps öffnen zu lassen. Dies gehört nicht zur Kernfunktionalität des Frameworks. Es gibt zwar einige Plugins von den Community-Mitgliedern die solche Funktionalität bieten, sie unterstützen jedoch nur die eine oder die andere Plattform.

Dieses Problem konnte bis jetzt nicht gelöst werden. Als Workaround wurde in der Anwendung für die Videos ein primitiver HTML5-Videoplayer und für die Bilder eine einfache Möglichkeit zur vergrößerten Darstellung implementiert, sodass die entsprechenden Mediendateien direkt in der Anwendung angezeigt werden können.

Pfade für private Verzeichnisse der Apps

In den früheren Versionen des Frameworks war es für Android nicht möglich den Pfad zum privaten Verzeichnis der Anwendung zu bestimmen.

Das Problem bestand darin, dass dieses Verhalten für Android-Plattformen offenbar nicht bzw. nicht richtig implementiert war. Bei einer Anfrage hat das Framework das Wurzelverzeichnis des externen Kartenspeichers zurückgegeben.

Das Problem konnte vorerst durch eine Fallunterscheidung mit einer anschließenden Korrektur gelöst werden. Nach einiger Zeit gab es ein Update, das die Schnittstelle um einige Konstanten mit den vordefinierten Pfaden erweitert hat.

4.4 Zusammenfassung

Im Kapitel 4 *Design* wurde ein Entwurf erarbeitet, nach dem die Anwendung implementiert werden soll, und anschließend eine prototypische Implementierung vorgestellt.

Im Kapitel 4.1 wurden einige Technischen Voraussetzungen hinsichtlich der Hardware und der Software zusammengetragen, welchen die Endgeräte genügen müssen, um die Anwendung ausführen zu können.

Das Kapitel 4.2 *Architektur* hat mit der *Systemarchitektur* ein Grundsystem aufgestellt und somit einen groben Überblick über alle Grundkomponenten des ganzen Systems vermittelt. Anschließend wurde mit der *Softwarearchitektur* der eigentliche Entwurf erarbeitet und mithilfe verschiedener Diagramme mit den begleitenden Erläuterungen vorgestellt. Dabei wurden auch einige grundsätzlichen Entscheidungen erläutert, die sowohl auf den Entwurf selbst als auch auf die weitere Entwicklung der Anwendung Einfluss nehmen.

Anschließend wurde im Kapitel 4.3 *Prototypische Implementierung* auf den Stand der Realisierung eingegangen, sowie auf einige Besonderheiten der Entwicklung mit Cordova / PhoneGap, mit der Benutzung der Onlinekartendienste und der Persistenzschicht. Im Anschluss wurden einige während der Entwicklung aufgetretene Probleme mit deren Lösungen vorgestellt und erläutert.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept erarbeitet, mit dem eine multiplattformfähige Reiseführeranwendung entwickelt werden kann.

Im Kapitel 1 *Einleitung* wurde die Zielsetzung definiert und die Gliederung der Arbeit aufgeführt. Das Ziel dieser Arbeit besteht darin, eine multiplattformfähige Reiseführeranwendung mit möglichst wenigem Aufwand zu entwickeln.

Das Kapitel 2 *Grundlagen und vergleichbare Arbeiten* hat sich mit den Grundlagen befasst und einige ähnliche Anwendungen vorgestellt. In diesem Kapitel wurden mehrere Ansätze zum Entwickeln mobiler Anwendungen analysiert, miteinander verglichen und anhand dessen eine Entscheidung zugunsten des Frameworks *Cordova / PhoneGap* getroffen. Dieses Framework verfolgt das Ziel es dem Entwickler zu ermöglichen hoch portierbare hybride Anwendungen mithilfe von Webtechnologien zu entwickeln.

Anschließend wurden im Kapitel 3 sowohl funktionale als auch nicht-funktionale Anforderungen an die Anwendung mithilfe der Anwendungsfälle und eines beispielhaften Anwendungsszenario definiert und zusammengefasst.

Im Kapitel 4 *Design* wurden technische Voraussetzungen definiert und ein Entwurf erarbeitet. Hier wurde mit der Systemarchitektur ein grober Überblick über das gesamte System gegeben und in der Softwarearchitektur ein detaillierter Entwurf erarbeitet. Anschließend wurde eine prototypische Implementierung vorgestellt. Hier wurden Entwicklungskonzepte gezeigt und auf einige während der Entwicklung aufgetretene Probleme und deren Lösungen eingegangen. Die Anwendung konnte nicht vollständig realisiert werden. Die Realisierung ist auf dem Stand prototypischer Fallstudien, deren Ziel war es die entsprechenden Technologien und Frameworks kennen zu lernen und deren Stärken bzw. Schwächen zu analysieren.

5.1 Ausblick

Mobile Geräte wie Smartphones oder Tabletts gewinnen immer mehr an Gewicht in unserem Leben. Es ist auch nicht zu erwarten, dass es sich in einer absehbaren Zeit ändern wird. Da man

derzeit auch zunehmend mehr reist, wird das eine oder das andere mobile Gerät immer ein Begleiter sein. Daher werden solche Anwendungen wie Reiseführer auch weiterhin benutzt. Außerdem werden die Entwickler aufgrund der Vielfalt der Geräte versuchen ihre Arbeit zu optimieren, sodass sie immer nach besseren Ansätzen zum Entwickeln multiplattformfähiger Anwendungen suchen werden.

Im Folgenden werden einige Überlegungen zur weiteren Entwicklung der im Rahmen dieser Arbeit entwickelten Anwendung vorgestellt.

- Eine ursprünglich geplante Webplattform für eine Community musste aufgegeben werden. Ein grober Konzept für die Realisierung wurde im Kapitel 4.2.2 *Softwarearchitektur* vorgestellt. Hier sollten die Community-Mitglieder ihre Touren und POIs untereinander teilen können. Eine Studie ähnlicher Anwendungen hat ergeben, dass eine Community mit der Möglichkeit die Reiseführer oder einzelne POIs zu bewerten und zu kommentieren von einem großen Vorteil ist. Daher wäre eine solche Funktionalität als Nächstes zu implementieren.
- Sehr viele Anwendungen bieten Möglichkeiten Inhalte in sozialen Netzwerken zu teilen. Eine solche Funktionalität wäre in dieser Anwendung ebenfalls vorteilhaft. Der Benutzer könnte auf seiner Reise POIs oder einzelne Medien mit seinen Freunden über soziale Netzwerke teilen.
- Derzeit sind die Smartphones oder Tabletts nicht mehr so teuer und werden dementsprechend öfter ausgetauscht. Daher wäre die Möglichkeit alle Daten und Einstellungen auf einen externen Speicher exportieren und von dort aus importieren zu können eine weitere wichtige Funktion. Dies würde die Migration auf neue Geräte wesentlich erleichtern.

Literaturverzeichnis

[Bayer 2002] Bayer, Thomas: REST Web Services. (2002). – URL http://www.oio.de/public/xml/rest-webservices.pdf. – Zugriffsdatum: 2014-07-27

[BITKOM 2012] BITKOM: Wettkampf der Smartphone-Plattformen. (2012). – URL http://www.bitkom.org/de/presse/74532_72316.aspx. – Zugriffsdatum: 2014-07-12

[Google 2013] Google: MVC Architecture. (2013). – URL https://developer.chrome.com/apps/app_frameworks#model_persistence. – Zugriffsdatum: 2014-08-11

[Heitkötter u. a. 2013] Heitkötter, Henning ; Hanschke, Sebastian ; Majchrzak, Tim A.: Evaluating Cross-Platform Development Approaches for Mobile Applications. (2013). – URL http://www3.nd.edu/~cpoellab/teaching/cse40814/crossplatform.pdf. – Zugriffsdatum: 2014-09-16

[NianticLabs@Google 2014] NianticLabs@Google: Field Trip. (2014). – URL https://play.google.com/store/apps/details?id=com.nianticproject.scout. – Zugriffsdatum: 2014-09-15

[Osmani 2014] Osmani, Addy: Learning JavaScript Design Patterns. (2014). – URL http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp. – Zugriffsdatum: 2014-08-11

[Powell 2012] Powell, Jason: What Titanium Appcelerator REALLY is and How it Works! (2012). – URL http://forumone.com/insights/what-titanium-appcelerator-really-and-how-it-works/. – Zugriffsdatum: 2014-09-13

[Trice 2013] Trice, Andrew: PhoneGap Architectural Considerations. (2013). – URL http://java.dzone.com/articles/phonegap-architectural. – Zugriffsdatum: 2014-07-12

[Tripwolf GmbH 2014] Tripwolf GmbH: tripwolf - dein Reiseführer. (2014). – URL https://play.google.com/store/apps/details?id=com.tripwolf. – Zugriffsdatum: 2014-09-15

[Veness 2014] Veness, Chris: Calculate distance, bearing and more between Latitude/Longitude points. (2014). – URL http://www.movable-type.co.uk/scripts/latlong.html#haversine. – Zugriffsdatum: 2014-07-12

[Whinnery 2014] Whinnery, Kevin: Comparing Titanium and PhoneGap. (2014). – URL http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap/. – Zugriffsdatum: 2014-09-12

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19. September 2014 Eugen Gez