ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Context-Aware Mobile Crowd Sensing using Mobile Hybrid Application Frameworks

Masterarbeit an der Universität Ulm

Vorgelegt von:
Julian Winterfeldt
julian.winterfeldt@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Betreuer:
Marc Schickler

2017

Fassung 8. Mai 2017

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Mobile Endgeräte sind heutzutage allgegenwärtig und werden in vielen Bereichen des alltäglichen Lebens eingesetzt. Als mobile Begleiter ermöglichen Smartphones den Einsatz mobiler Applikationen zu jeder Tageszeit und in einer Vielzahl von Situationen. Die Nutzung einer mobilen Applikation erfolgt heutzutage meist durch eine explizite Interaktion des Nutzers. Im Gegensatz dazu können sich kontextsensitive Anwendungen adaptiv verhalten und ermöglichen eine implizite Interaktion. Auch im Bereich des Mobile-Crowdsourcings gewinnen kontextbewusste Applikationen an Bedeutung. Datensätze können mit Kontextinformationen angereichert werden und die Datenerfassung kann kontextsensitiv erfolgen. Die Entwicklung solcher Applikationen kann auf unterschiedliche Art und Weise durchgeführt werden. Im Vergleich zur nativen Applikationsentwicklung kann die Entwicklung mobiler Webanwendungen oder hybrider Applikationen plattformunabhängig erfolgen. Cross-Platform-Ansätze vereinen die Vorteile von nativer und hybrider Applikationsentwicklung. Dabei ermöglichen Frameworks wie NativeScript den direkten Zugriff auf native Schnittstellen. Für diese Ansätze stehen jedoch keine allgemeinen Werkzeuge zur Verfügung, um Anwendungen kontextsensitiv zu implementieren.

Um die Cross-Platform-Entwicklung kontextsensitiver Applikationen zu vereinfachen, wurde ein Framework für NativeScript-Applikationen entwickelt. Das Framework stellt Komponenten zur Verfügung, welche die plattformunabhängige Entwicklung kontextsensitiver Applikationen unterstützen. Auf Basis einer logikbasierten Kontextrepräsentation und eines regelbasierten Schlussfolgerungsmechanismus können Applikationsentwickler Kontextereignisse definieren. Diese werden automatisch evaluiert und dabei ausgelöst, falls die entsprechenden Kontextbedingungen erfüllt sind. Die Datenerfassung und Bereitstellung von Kontextinformationen erfolgt dabei modularisiert. Es wurden vier beispielhafte Module entwickelt, um den Ort, die Herzfrequenz des Nutzers, ausgeübte Aktivitäten und die Umgebungslautstärke als Kontextinformationen bereitzustellen. Zudem wurde ein NativeScript-Modul entwickelt, das verschiedene Arten der Hintergrundausführung plattformunabhängig abstrahiert.

Danksagung

Zunächst möchte ich mich bei Prof. Dr. Manfred Reichert und Dr. Rüdiger Pryss für die Begutachtung der vorliegenden Arbeit bedanken. Ein besonderer Dank gilt meinem Betreuer Marc Schickler, der mir diese Arbeit ermöglichte. Vielen Dank für dein Vertrauen in meine Ideen und deine schnelle Hilfe bei Problemen. Zudem möchte ich mich ganz herzlich bei meiner Familie bedanken, die mich immer unterstützt und mir das Studium überhaupt ermöglicht hat. Vielen Dank auch an all meine Freunde, die mich in schwierigen Zeiten motiviert haben. Ganz besonders möchte ich mich bei meiner Freundin für die Unterstützung, die Motivation und das Korrekturlesen bedanken.

Inhaltsverzeichnis

1

Einleitung

Mobile Endgeräte sind heutzutage allgegenwärtig und werden in vielen Bereichen des alltäglichen Lebens eingesetzt. Allein in Deutschland besitzen knapp 50 Millionen Menschen bereits ein Smartphone [1]. Als mobile Begleiter ermöglichen Smartphones den Einsatz mobiler Anwendungen zu jeder Tageszeit und in einer Vielzahl von Situationen. Sie können in vielen Aufgabenbereichen unterstützend eingesetzt werden und ermöglichen eine Verschmelzung von digitaler und realer Welt. Smartphones sind typischerweise mit einer Vielzahl von Sensoren ausgestattet. Hierzu zählen Beschleunigungssensoren, Lichtsensoren, Mikrophone oder Sensoren zur Ortung über das Global-Positioning System (GPS). Die Nutzung einer mobilen Applikation erfolgt heutzutage meist durch eine explizite Interaktion des Nutzers. Dies erfordert mentale Aufmerksamkeit und kann in vielen Situationen unangebracht oder störend sein. Im Gegensatz dazu können kontextsensitive Anwendungen selbständig auf Situationen reagieren und ermöglichen eine implizite Interaktion. Hierfür werden über die Sensoren der mobilen Endgeräte Daten erfasst, die zur Bestimmung des Kontexts einer Entität genutzt werden können. Auf Basis dieses Wissens können kontextbewusste Applikationen ihr Verhalten anpassen oder proaktiv dem Nutzer Informationen bereitstellen. Solche Anwendungen können die explizite Interaktion durch den Nutzer reduzieren oder nur in angemessenen Situationen die Aufmerksamkeit des Nutzers beanspruchen.

Auch im Bereich des Mobile-Crowdsourcings gewinnen kontextsensitive Applikationen an Bedeutung. In diversen Anwendungsgebieten werden Kontextinformationen verwendet, um gesammelte Daten anzureichern oder das Verhalten einer mobilen Applikation anzupassen. In der mobilen Navigation werden beispielsweise Kontextinformationen gesammelt, um die Routenplanung zu optimieren oder um Verkehrsaufkommen anzuzei-

gen [2]. Im Rahmen der Gesundheitsforschung können kontextbewusste Applikationen eingesetzt werden, um Situationsbedingungen während der Datenerfassung zu bestimmen [3]. Kontextsensitive Mobile-Crowdsourcing-Applikationen können auch die Kommunikation zwischen Bürgern und der öffentlichen Verwaltung verbessern [4]. Die Verwendung kontextsensitiver Mobile-Crowdsourcing-Applikationen ist in zahlreichen Anwendungsgebieten denkbar.

1.1 Problemstellung

Die Entwicklung kontextsensitiver Mobile-Crowdsourcing-Applikationen kann auf unterschiedliche Art und Weise erfolgen. Dabei müssen verschiedene Plattformen und Geräteunterschiede berücksichtigt werden. Dies gilt besonders für kontextsensitive Applikationen, die zur Datenerfassung auf Sensoren und Schnittstellen der jeweiligen Plattform angewiesen sind. Das kann im Rahmen der nativen Anwendungsentwicklung zu großen Herausforderungen führen, da jede Plattform mit einer eigenen Applikation bedient werden muss. Diese Probleme können durch die Verwendung mobiler Webanwendungen vermieden werden. Durch den Einsatz von Webtechnologien kann mit einer einzelnen Codebasis für mehrere Plattformen entwickelt werden. Solche Anwendungen können als hybride mobile Applikationen auf Smartphones eingesetzt werden. Diese können jedoch meist nicht mit der Performance nativer Applikationen konkurrieren. Zudem erfordert der Zugriff auf Sensoren und Schnittstellen der mobilen Geräte oft Kenntnisse der nativen Programmiersprachen.

Cross-Platform-Ansätze verwenden im Gegensatz zu hybriden Ansätzen keine mobile Webanwendung. Cross-Platform-Applikationen bestehen aus nativen Benutzeroberflächen. Dadurch können solche Anwendungen die Performance nativer Applikationen erreichen. Es existieren diverse Cross-Platform-Frameworks, welche die plattformübergreifende Entwicklung mit Webtechnologien ermöglichen. Dies erleichtert den Einstieg für viele Entwickler und erfordert keine Kenntnisse der nativen Programmiersprachen. Das Cross-Platform-Framework NativeScript ermöglicht zudem den direkten Zugriff auf native Schnittstellen über JavaScript. Mit diesem Framework können plattformüber-

greifende native Applikationen unter Verwendung von XML, CSS und JavaScript für Android und iOS entwickelt werden. Sensoren oder andere Geräteschnittstellen können dabei verwendet werden, ohne dass die Syntax nativer Programmiersprachen bekannt sein muss. Aufgrund dieser Eigenschaften eignet sich das NativeScript-Framework besonders zur plattformübergreifenden Entwicklung kontextsensitiver Applikationen.

Für solche Cross-Platform-Ansätze stehen jedoch keine allgemeinen Werkzeuge zur Verfügung, um mobile Applikationen kontextsensitiv zu implementieren.

1.2 Zielsetzung

Um die plattformübergreifende Entwicklung kontextsensitiver Anwendungen zu vereinfachen, soll ein Framework für NativeScript-Applikationen entworfen und entwickelt werden. Das Framework soll Werkzeuge anbieten, die eine plattformübergreifende Entwicklung kontextsensitiver Applikationen unterstützen. Dabei soll es eine logikbasierte Kontextrepräsentation und einen regelbasierten Schlussfolgerungsmechanismus zur Verfügung stellen. Die Bereitstellung von Kontextinformationen soll modularisiert erfolgen. Zudem soll das Framework möglichst plattformunabhängig implementiert werden und nur zur Datenerfassung auf plattformspezifische Schnittstellen zurückgreifen. Dadurch können NativeScript-Entwickler unter Verwendung bekannter Technologien das Framework mit eigenen Modulen erweitern. Zudem soll das Framework auf allen Plattformen im Vordergrund sowie im Hintergrund einer Applikation betrieben werden können.

Des Weiteren sollen Module für das Framework implementiert werden, um den Ort, die Herzfrequenz des Nutzers, die Aktivität eines Nutzers und die Umgebungslautstärke als Kontextinformationen bereitzustellen. Zu Test- und Demonstrationszwecken soll eine kontextsensitive Beispielapplikation implementiert werden, die das Framework sowohl im Vordergrund als auch im Hintergrund einer Applikation verwendet.

1.3 Aufbau der Arbeit

In Kapitel 2 werden zunächst verwandte Arbeiten vorgestellt, die sich mit kontextsensitivem Mobile-Crowdsourcing und der Cross-Platform-Entwicklung kontextsensitiver Applikationen beschäftigen. Anschließend werden in Kapitel 3 verschiedene Ansätze zur plattformübergreifenden Entwicklung mobiler Applikationen verglichen. Kapitel 4 behandelt Aspekte der Anwendungsentwicklung mit NativeScript, die für das weitere Verständnis der Arbeit relevant sind. In Kapitel 5 wird auf die Grundlagen kontextsensitiver Systeme eingegangen. Anschließend erfolgt in Kapitel 6 eine Anforderungsanalyse für die Entwicklung des im Rahmen dieser Arbeit vorgestellten Frameworks. Zudem wird die Konzeption und Architektur des Frameworks beschrieben. In Kapitel 7 werden ausgewählte Aspekte der Implementierung vorgestellt und die Umsetzung mit den in Kapitel 6 aufgestellten Anforderungen abgeglichen. Abschließend erfolgt in Kapitel 8 eine zusammenfassende Betrachtung der Arbeit und ein Ausblick auf mögliche Weiterentwicklungen und Aspekte, die den Einsatz des Frameworks betreffen.

2

Verwandte Arbeiten

Die Berücksichtigung des Kontexts spielt eine immer bedeutendere Rolle im Bereich Mensch-Maschine-Interaktion. Der Begriff *Kontext* kann hierauf bezogen auf unterschiedliche Arten definiert werden. Die am häufigsten verwendete Definition stammt von Dey et al.: „Kontext ist jede Information, die genutzt werden kann, um die Situation einer Entität zu beschreiben. Entitäten sind Personen, Orte oder Objekte, welche für die Interaktion zwischen einem Nutzer und einer Applikation als relevant betrachtet werden, inklusive dem Nutzer und der Applikation selbst“ [5]. Im weiteren Verlauf dieser Arbeit wird der Begriff *Kontext* nach dieser Definition verwendet. In der Literatur existieren diverse Arbeiten, die Werkzeuge zur Bereitstellung von Kontextinformationen vorstellen. Dabei beschäftigen sich einige Arbeiten auch mit plattformübergreifenden Ansätzen zur Bereitstellung von Kontextinformationen auf mobilen Endgeräten. Weitere Arbeiten behandeln die Verwendung von Kontextinformationen in Mobile-Crowdsourcing Applikationen. Im Folgenden werden ausgewählte Arbeiten aus der Literatur zu diesen Themen präsentiert.

Der Begriff *Crowdsourcing* wird für eine große Anzahl verschiedenster Aktivitäten verwendet. Laut Howe [6] beschreibt der Begriff allgemein, wie die Macht der Menge genutzt werden könne, um Ziele zu erreichen, für die bisher eine kleine Gruppe von Spezialisten verantwortlich war. Es existieren jedoch aufgrund der vielen verschiedenen Anwendungsgebiete ebenso viele unterschiedliche Definitionen. Im Folgenden beschränkt sich diese Arbeit auf die Verwendung der Begriffe *Mobile-Crowdsourcing* bzw. *Mobile-Crowdsensing*, die sich auf die Möglichkeit beziehen, Wissen zu generieren, indem gewöhnliche Bürger generierte oder erfasste Daten über ihre mobilen Endgeräte bereitstellen [7]. Mit diesem Ansatz können sowohl personenbezogene als auch gruppen-

basierte Daten über die Masse der individuellen Datensätze aggregiert werden. Nach Ma et al. [8] wird die Datenerfassung in zwei Arten unterschieden. Beim sogenannten Participatory-Sensing entscheidet sich der Teilnehmer bewusst, dem Anliegen einer Applikation nachzukommen. Er entscheidet, wann, wo, welche Daten wie erfasst werden. Im Gegensatz dazu werden beim Opportunistic-Sensing Daten über eine Applikation erfasst, ohne dass sich der Nutzer dessen bewusst ist. In diesem Fall sammelt die Applikation beispielsweise opportunistisch Kontextdaten innerhalb eines Hintergrunddienstes. Mobile-Crowdsourcing wird in vielen unterschiedlichen Anwendungsgebieten eingesetzt. Im Folgenden werden beispielhaft Arbeiten aus verschiedenen Gebieten vorgestellt.

Im Bereich der Routenplanung wird Mobile-Crowdsourcing bereits von einer großen Masse tagtäglich, bewusst oder unbewusst, eingesetzt. Routenplaner wie Google-Maps sammeln anonym Kontextdaten ihrer Nutzer, um das Verkehrsaufkommen in Straßennetzen einzuschätzen [2]. Dabei werden der Ort eines Nutzers sowie dessen Geschwindigkeit über die GPS-Position seines Smartphones nach dem Prinzip des Opportunistic-Sensing erfasst und an Google gesendet. Dadurch können dem Nutzer ortsabhängige Verkehrsaufkommen in der Google-Maps-Applikation dargestellt werden. Des Weiteren nutzt Google die gesammelten Verkehrsdaten, um die Routenplanung an das Verkehrsaufkommen anzupassen und zu optimieren.

Mobile-Crowdsourcing kann auch in der öffentlichen Verwaltung zum Einsatz kommen. Hierfür stellen Tamilin et al. [4] ein Konzept vor, mit dem Behörden in der öffentlichen Verwaltung kontextabhängige Aufgaben an Bürger mit Smartphones übermitteln können. Dieses Konzept wird zum Beispiel eingesetzt, um die Kommunikation zwischen Bürgern und öffentlicher Verwaltung zu verbessern. Damit können Bürger in Entscheidungsprozesse, wie zum Beispiel die Stadtplanung, eingebunden werden. Im Rahmen dieses Konzeptes entwickelten Tamilin et al. einen Prototypen namens sensoRcivico, der aus einer serverseitigen Anwendung und einer Applikation für Android besteht. Die Android-Applikation wurde dabei mit dem Cross-Platform-Framework Appcelerator-Titanium [9] entwickelt. Über die serverseitige Anwendung können ortsabhängige Aufgaben erstellt, verwaltet und deren Ausführung beobachtet werden. Wenn ein Nutzer der Android-Applikation die Umgebung eines bestimmten Ortes betritt, wird ihm vom System eine entsprechende Aufgabe zugestellt. Bei den Aufgaben handelt es sich zum Beispiel um

das Aufnehmen eines Fotos oder eines Videos an diesem Ort. Der Android-Prototyp verhält sich bei der Ortsbestimmung adaptiv. Er wechselt von netzbasierter Positionierung zu GPS, wenn die netzbasierte Positionierung zu ungenau ist, um zwischen zwei räumlich naheliegenden Aufgaben zu unterscheiden.

Auch im Gesundheitswesen und in der Gesundheitsforschung kann Mobile-Crowdsourcing eingesetzt werden, um Erkenntnisse zu gewinnen. Es existieren diverse Arbeiten, welche die Datenerfassung über mobile Applikationen im Rahmen von klinischen oder psychologischen Studien behandeln [10, 11, 12]. Die Verwendung von Sensordaten erfährt in diesem Bereich allgemein eine immer größere Beachtung [13, 14, 15, 16]. Verschiedene Arbeiten zeigen, wie beispielsweise die Herzfrequenz eines Nutzers bei der Datenerfassung berücksichtigt werden kann [17, 18, 19, 20]. Mobile-Crowdsourcing findet zum Beispiel auch in der Tinnitusforschung Verwendung. Die TrackYourTinnitus-Plattform [3, 21, 22] wurde im Rahmen einer Zusammenarbeit des Instituts für Datenbanken und Informationssysteme der Universität Ulm und der Tinntius Research Initiative entwickelt. Die Plattform setzt mobile Applikationen für iOS und Android ein, mit denen Daten von Tinnitus-Erkrankten gesammelt werden, um neue medizinische Erkenntnisse hinsichtlich der Ursachen und der Behandlung von Tinnitus zu gewinnen. Über die mobilen Applikationen werden Fragebögen an die Erkrankten übermittelt, mit denen Schwankungen in der individuellen Tinnituswahrnehmung erfasst werden. Die Applikationen messen während des Ausfüllens eines Fragebogens die Umgebungslautstärke über das Mikrophon der mobilen Endgeräten. Zudem können mit dem System auch Smartwatches eingesetzt werden, welche die Erfassung der Herzfrequenz ermöglichen [17]. Dadurch wird die erfasste Datenmenge mit zusätzlichen Kontextinformationen angereichert. Die dabei gewonnenen Datensätze werden anonymisiert an einen zentralen Server übertragen, auf dem sie für Forschungszwecke aggregiert werden. Den Nutzern wird dabei ein Mehrwert geboten, indem die mobilen Applikationen Statistiken bereitstellen, mit denen Korrelationen zwischen Tagesablauf, Tätigkeiten und Schwankungen des Tinnitus beobachten werden können. In der Forschung wird die Plattform zum Beispiel genutzt, um Zusammenhänge zwischen emotionaler Verfassung und Tinnituswahrnehmung zu untersuchen [23].

Mit Here-n-Now stellen Jayaraman et al. [24] ein Framework für kontextbewusstes Mobile-Crowdsourcing vor. Das Ziel des vorgestellten Frameworks ist es, Nutzern Informationen über Orte bereitzustellen, an denen andere Nutzer Daten in Echtzeit sammeln. Das Framework berücksichtigt ausgeübte Aktivitäten und Lärm an einem Ort sowie die Belebtheit des Ortes. Jayaraman et al. behandeln im Rahmen einer Android-Applikation beispielhaft zwei Fragestellungen, die mit dem System beantwortet werden können. Zum einen kann die Anwendung beantworten, ob ein Ort belebt ist, was beispielsweise für das Joggen im Park aus Sicherheitsgründen relevant sein kann. Zum anderen schätzt die Applikation ein, wie laut es an einem Ort ist. Dies kann zum Beispiel von Arbeitskollegen genutzt werden, um ein Restaurant auszusuchen, an dem Arbeitsthemen in ruhiger Umgebung besprochen werden können. Das Here-n-Now-Framework besteht aus mehreren Modulen, um Fragestellungen wie diese beantworten zu können. Das Framework verwendet die sogenannte Context-Aware-Real-Time-Open-Mobile-Miner-Engine (CAROMM) [25] zum Sammeln von Daten in Echtzeit. Die CAROMM-Engine kommt innerhalb eines Moduls zur Datensammlung und Datenanalyse zum Einsatz. Über dieses Modul werden Kontextdaten zur weiteren Verarbeitung zur Verfügung gestellt. Zu diesen Kotextinformationen gehören sensorische Daten sowie die Aktivität des Nutzers, die über eine Aktivitätenerkennung bestimmt wird. Diese basiert in der vorgestellten Anwendung auf einem neuronalen Netz und kann mit beliebigen Modellen zur Aktivitätenerkennung ausgetauscht werden. Das cloudbasierte Datenverarbeitungsmodul schlussfolgert anhand der Daten, die von den mobilen Applikationen bereitgestellt werden, auf die vorherrschende Situation an einem Ort. Nach Jayaramn et al. kann dabei jede beliebige Inferenzmethode eingesetzt werden. Konkret wurde im Rahmen der vorgestellten Arbeit eine Kombination aus Fuzzylogik und den sogenannten Context Spaces [26, 27] eingesetzt, um auf die Situation an einem bestimmten Ort zu schließen.

In der Literatur existieren diverse Arbeiten, die sich mit Systemen zur Erfassung und Bereitstellung von Kontextinformationen beschäftigen, ohne die Thematik Mobile-Crowdsourcing direkt zu tangieren. Die Arbeiten von Baldauf et al. [28], Li et al. [29] sowie Strang et al. [30] vergleichen verschiedene kontextsensitive Systeme bezüglich Eigenschaften wie Architektur, Kontexterfassung, Kontextmodellierung, Kontextverarbeitung, Schlussfolgerungsmechanismen, Skalierbarkeit oder Datenschutz. Diese Themen wer-

den in Kapitel 5 dieser Arbeit näher betrachtet. Im Folgenden werden ausgewählte Arbeiten vorgestellt, die die Kontexterfassung auf mobilen Endgeräten behandeln und dafür einen plattformübergreifenden Ansatz verwenden.

Oliveira et al. [31] präsentieren eine Lösung, wie Kontextinformationen für mobile Webanwendungen bereitgestellt werden können. Das sogenannte Mobile-Contextual-Framework verwendet einen lokalen mobilen Webserver, der Zugriff auf native Schnittstellen des mobilen Betriebssystems hat. Die Framework-Architektur besteht dabei aus einem Mobile-Context-Server und einem Remote-Context-Server. Der Mobile-Context-Server läuft auf dem mobilen Endgerät in Form einer Anwendung eines mobilen Webservers. Dabei muss ein solcher mobiler Webserver, wie zum Beispiel der CocoaHTTPServer für iOS [32], auf dem Endgerät vorinstalliert sein. Er agiert als eine Schnittstelle zwischen Browser und mobilem Betriebssystem. Diese Schnittstelle kann HTTP-Anfragen beantworten, die der Browser absetzt, wenn eine mobile Webanwendung eine entsprechende Anfrage beinhaltet. Der Mobile-Context-Server verarbeitet eine HTTP-Anfrage, indem er auf die nativen Schnittstellen des mobilen Endgeräts zugreift und die erlangten Informationen an den Browser zurücksendet. Eine mobile Webanwendung hat dadurch die Möglichkeit, über HTTP-Anfragen Geräteschnittstellen anzusprechen und Kontextinformationen erfassen zu können. Gleichzeitig können über ein Gateway auch Remote-HTTP-Anfragen an einen Remote-Context-Server abgesetzt werden. Letzterer kann zusätzliche Kontextinformationen bereitstellen oder regelbasierte Benachrichtigungen proaktiv an die mobilen Endgeräte schicken. Die Funktionsweise des Mobile-Contextual-Frameworks demonstrieren Oliveira et al. in einer prototypischen Applikation zur Buchung von Kinotickets. Hierbei kommt der Remote-Context-Server zum Einsatz, um eingescannte Codes zu verarbeiten oder den Nutzer an gebuchte Vorstellungen zu erinnern. Der Mobile-Context-Server wird eingesetzt, um Kalendereinträge zu berücksichtigen, andere Personen über die Kontakte einzuladen, ortsabhängig das nächste Kino zu bestimmen oder um eine Benachrichtigung an den Nutzer zu senden, falls dieser kurz vor einer gebuchten Vorstellung noch zu weit vom Kino entfernt ist.

Ein geräte- und plattformunabhängiges Framework für Kontextsensitivität wurde von Ntanos et al. [33] für die Open-Source-Plattform webinos [34] entwickelt. Die webinos-Plattform bietet die Möglichkeit, Webapplikationen unabhängig von Geräten oder Be-

triebssystemen betreiben zu können. Dabei soll die Plattform für mobile Endgeräte, PCs, Fernseher, Autos oder Geräte aus dem Bereich *Internet der Dinge* (englisch: Internet of Things, IoT) einsetzbar sein. Die Plattform agiert personenzentriert, indem Webapplikationen oder Dienste über alle verbundenen Geräte eines Nutzers verteilt betrieben werden können. Bei webino wird eine JavaScript-Schnittstelle für Webapplikationen bereitgestellt, die plattformübergreifend Funktionalitäten der Geräte abstrahiert. Das von Ntanos et al. vorgestellte Framework verwendet vorhandene Funktionalitäten der webinos-Plattform, um Kontextdaten zu sammeln. Es werden beispielsweise Aufrufe innerhalb der Plattform unterbrochen, um Ereignisse abzufangen und in Kontextinformationen umzuwandeln. Hierfür werden webino-Schnittstellen genutzt, mit denen zum Beispiel Informationen über abgespielte Medien, den Status von Geräten, den Ort von Geräten, den Fahrzeugstatus, Benachrichtigungsereignisse oder Navigationsereignisse gesammelt werden können. Diese Informationen werden innerhalb sogenannter Context-Objects gespeichert. Über eine Schnittstelle ist es dabei möglich, Context-Objects zu definieren oder Kontextinformationen mit Hilfe sogenannter Context-Queries abzufragen. Unter anderem kann man über diese Schnittstelle auch Polling-Regeln für webino-Schnittstellen definieren oder regelbasierte Context-Event-Listener registrieren, die ausgelöst werden, wenn dafür festgelegte Kontextbedingungen erfüllt sind. Ein besonderes Merkmal dieses Konzeptes ist, dass Kontextinformationen über einen Cloud-Speicher verteilt und damit auf allen Geräten eines Nutzers abgerufen werden können.

Wang et al. [35] entwickelten ein Framework für kontextsensitive mobile Applikationen, das als Middleware konzeptioniert ist. Die vorgestellte Middleware wird zwischen Betriebssystem und mobilen Applikationen eingesetzt und baut auf der BAE-Engine auf [36]. Die BAE-Engine abstrahiert die plattformspezifischen APIs von Android, iOS und Windows Phone und stellt der Middleware eine einheitliche Nutzung von Gerätesensoren zur Verfügung. Kontextsensitivität wird über drei Hauptmodule innerhalb der Middleware realisiert. Der sogenannte Sensor-Manager sammelt Sensordaten über die Schnittstellen der BAE-Engine und stellt diese dem Speichermodul und Schlussfolgerungsmodul zur Verfügung. Dabei können über die BAE-Engine sowohl physikalische Sensoren als auch virtuelle Sensoren, wie zum Beispiel der Kalender oder Kontakte,

genutzt werden. Das Speichermodul basiert auf einer Datenbank, in der nach dem Key-Value-Modell Kontextdaten und Anfragen von Applikationen gespeichert werden. Das Schlussfolgerungsmodul ist zwischen dem Sensor-Manager und den mobilen Applikationen angesiedelt und ist dafür verantwortlich, Kontextinformationen zu interpretieren oder High-Level-Kontextdaten aus Low-Level-Kontextdaten abzuleiten. Die Begriffe Low-Level und High-Level beziehen sich dabei auf die Aussagekraft einer Kontextinformation. Low-Level-Kontextinformationen sind zum Beispiel Sensordaten die ohne weitere Verarbeitung vom System bezogen werden und teilweise eine recht geringe Aussagekraft haben. Dahingegen können High-Level-Kontextinformationen eine komplexere Situation beschreiben und werden aus der Verarbeitung und Kombination mehrerer Low-Level-Kontextinformationen abgeleitet [37]. Wang et al. verwenden bei ihrem Framework einen regelbasierten Schlussfolgerungsmechanismus zur Ableitung von High-Level-Kontextinformationen. Im Gegensatz dazu könnten auch Ansätze aus dem statistischen Lernen, wie zum Beispiel Hidden-Markov-Modelle, verwendet werden. Diese Ansätze können jedoch einen hohen Energie- und Speicherverbrauch verursachen. Beim regelbasierten Ansatz von Wang et al. wird mit Fachwissen (englisch: domain knowledge) ein sogenannter Regelbaum für eine spezifische Situation definiert. Ein solcher Regelbaum besteht aus Kontextmerkmalen, die mit booleschen Operatoren verknüpft werden. Einzelne Kontextmerkmale beziehen sich auf Sensoren und können durch mathematische Operatoren (wie z.B. $\leq$, $\geq$ oder $=$) eingeschränkt werden. Situationen, die über einen Regelbaum definiert sind, werden beim sogenannten Knowledge-Manager registriert. Das Situation-Register evaluiert die registrierten Regelbäume periodisch und löst eine entsprechende Aktion aus, wenn sich ein Regelbaum bewahrheitet und damit eine Situation erkannt wird. Wie in Unterabschnitt 6.2.3 beschrieben, wird in dieser Arbeit ein sehr ähnlicher Ansatz verwendet, um über Regeln eine Situation zu definieren.

3

Plattformübergreifende Entwicklung mobiler Applikationen

Dieses Kapitel behandelt Grundlagen zur plattformübergreifenden Entwicklung von mobilen Applikationen. In Abschnitt 3.1 wird zunächst die plattformübergreifende Anwendungsentwicklung der nativen Anwendungsentwicklung gegenübergestellt und zwei weitverbreitete Ansätze zur plattformübergreifenden mobilen Anwendungsentwicklung eingeführt. Abschnitt 3.2 behandelt die hybride mobile Anwendungsentwicklung auf Basis einer sogenannten WebView und Abschnitt 3.3 die sogenannte Cross-Platform-Anwendungsentwicklung unter Einsatz von Webtechnologien. Für beide Ansätze werden beispielhafte Frameworks und deren Eigenschaften vorgestellt.

3.1 Grundlagen

Die Entwicklung mobiler Applikationen für mehrere Plattformen kann auf unterschiedliche Art und Weise durchgeführt werden. Die native Anwendungsentwicklung stellt die klassische Methode dar, um Anwendungen für eine spezielle Plattform zu entwickeln. Native Applikationen sind solche, die in einer plattformspezifischen Programmiersprache geschrieben werden. Die Plattform bietet für diese Sprache Programmierschnittstellen (englisch: application programming interfaces, APIs) an, um System- und Gerätefunktionen auf niedrigeren Ebenen anzusprechen. Anwendungen für Android werden beispielsweise in Java entwickelt, während iOS Anwendungen in Objective-C oder Swift programmiert werden. Bei der nativen mobilen Anwendungsentwicklung kann über Geräte-APIs oder native Bibliotheken die komplette Funktionalität einer Plattform ge-

nutzt werden. Zudem können Anwendungen speziell für die jeweilige Plattform optimiert werden. Dies ist besonders für Anwendungskonzepte von Bedeutung, die stark auf Gerätesensoren angewiesen sind (vgl. [38], [39], [40], [41], [42]).

Softwareentwickler verfolgen oft das Ziel, ein möglichst großes Publikum mit einer mobilen Applikation zu erreichen. Die Reichweite einer mobilen Anwendung hängt unter anderem davon ab, auf welchen Plattformen die Anwendung veröffentlicht wird. Der weltweite Markt der mobilen Betriebssysteme am Absatz für Smartphones wird von Google mit Android und Apple mit iOS dominiert [43]. Andere Betriebssysteme wie Windows Phone von Microsoft oder BlackBerry OS haben mit Marktanteilen von unter 1% eine vergleichsweise geringe Nutzerbasis. Aufgrund dieser Marktsituation ist es sinnvoll, dass mobile Applikationen zumindest für die mobilen Betriebssysteme Android und iOS entwickelt werden, um ein möglichst großes Publikum zu erreichen. Die Verwendung eines nativen Entwicklungsansatzes hat zur Folge, dass mehrere Applikationen entwickelt, gewartet und betrieben werden müssen. Des Weiteren muss die Heterogenität der mobilen Endgeräte behandelt werden, um ein konsistentes Nutzungserlebnis auf allen Endgeräten zu ermöglichen. Dies kann die Entwicklungszeit und -kosten einer mobilen Anwendung stark beeinflussen.

Die plattformübergreifende Anwendungsentwicklung versucht, diese Probleme zu vermeiden. Plattformübergreifende Anwendungen nutzen eine gemeinsame Codebasis für mehrere Plattformen. Dadurch ist es möglich, die Produktionszeit sowie die Entwicklungs-, Betriebs- und Wartungskosten zu senken. Die mobile plattformübergreifende Anwendungsentwicklung kann über verschiedene Ansätze realisiert werden. Webtechnologien wie JavaScript oder CSS erfreuen sich heute großer Beliebtheit in den Online-Communities [44] und werden auch für die Entwicklung mobiler Anwendungen eingesetzt. Mit Webtechnologien kann für Entwickler aus der Web-Branche oder für Entwickler mit grundlegenden Kenntnissen der Technologien der Einstieg in die mobile Applikationsentwicklung vereinfacht werden. Dadurch kann mit Webtechnologien eine deutlich größere Entwicklerbasis angesprochen werden, als mit plattformspezifischen Programmiersprachen. Aus unternehmerischer Sicht bedeutet dies auch, dass nicht für jede einzelne mobile Plattform spezialisierte Entwickler eingestellt werden müssen. Des Weiteren ist es mit Webtechnologien möglich, Code sowohl für Webseiten als auch

für mobile Applikation zu verwenden, wodurch die Wiederverwendbarkeit von Code gesteigert werden kann. Aus diesen Gründen behandelt diese Arbeit daher ausschließlich die plattformübergreifende Entwicklung mobiler Applikationen unter Einsatz von Webtechnologien.

In der mobilen Anwendungsentwicklung mit Webtechnologien wird hauptsächlich zwischen drei Ansätzen unterschieden [45]. Eine Möglichkeit sind mobile Webanwendungen. Diese werden mit Webtechnologien wie HTML, CSS oder JavaScript entwickelt und sind speziell darauf ausgerichtet, in Browsern von mobilen Endgeräten genutzt zu werden. Mobile Webanwendungen werden, wie andere Webseiten auch, im Browser angezeigt und haben dadurch keine Möglichkeit, Gerätefunktionalitäten direkt anzusprechen und zu verwenden [46]. Aufgrund dieser Einschränkung ist es mit mobilen Webanwendungen nicht möglich, ein plattformübergreifendes Framework für Kontextsensitivität zu entwickeln, weshalb diese im Folgenden nicht weiter behandelt werden. Des Weiteren können mobile Webanwendungen nicht über die plattformspezifischen Applikationsmärkte, wie z.B. über Google-Play oder den App-Store von Apple, vertrieben werden.

Eine andere Möglichkeit beruht auf sogenannten hybriden Ansätzen. Der Begriff *hybrid* kann in diesem Zusammenhang als eine Kreuzung aus einer Webanwendung und einer nativen Applikation verstanden werden. Hybride Applikationen sind mobile Webanwendungen, die als native Applikation verpackt und eingesetzt werden [45]. Die Laufzeitumgebung einer solchen Anwendung besteht aus einem Browser, der in eine native Anwendung eingebettet wird. Dieser Browser wird als WebView bezeichnet. Dabei können über spezielle Frameworks native Gerätefunktionalitäten verwendet werden [47].

Im Gegensatz hierzu generieren Cross-Platform-Ansätze native Applikationen [45] mit nativen Benutzeroberflächen und es kommt kein integrierter Browser zum Einsatz. Dabei können unter Verwendung einer einzelnen Codebasis native Anwendungen für mehrere Plattformen erzeugt werden. Diese Arbeit behandelt Cross-Platform-Ansätze, bei denen die Anwendungslogik und Benutzeroberfläche ebenfalls mit Webtechnologien oder angelehnten Technologien implementiert wird. Diese Ansätze werden teilweise ebenfalls als hybride Ansätze bezeichnet [48, 46]. Zur deutlichen Differenzierung werden diese jedoch im weiteren Verlauf der Arbeit als Cross-Platform-Ansätze bezeichnet.

Im Folgenden werden sowohl der hybride WebView-basierte Ansatz als auch der Cross-Platform-Ansatz in Kombination mit Webtechnologien näher betrachtet, da beide über den Zugriff auf Gerätefunktionalitäten und -sensoren die Möglichkeit bieten, ein plattformübergreifendes Framework für Kontextsensitivität zu entwickeln. Es existieren auch andere Cross-Platform-Ansätze, wie zum Beispiel der von Microsofts Xamarin [49], welche aber in dieser Arbeit nicht näher betrachtet werden. Deren Herangehensweisen unterscheiden sich von den hier behandelten Ansätzen dadurch, dass bei der Entwicklung der Anwendungslogik sowie der Benutzeroberfläche keine oder nur in geringem Maße Webtechnologien als Basis zum Einsatz kommen, was den Einstieg in solche Entwicklungsumgebungen erschweren kann.

3.2 Hybride WebView-basierte mobile Applikationen

Hybride mobile Applikationen vereint die Eigenschaft, dass eine mobile Webanwendung innerhalb eines Browsers läuft, der in eine native Anwendung integriert ist [45]. Daher werden hybride mobile Anwendungen mit Webtechnologien wie HTML, CSS und JavaScript erstellt. Der integrierte Browser, auch WebView genannt, nutzt die Browser-Engine des Betriebssystems, um die mobile Webanwendung im Vollbildmodus innerhalb einer nativen Container-Applikation zu rendern. Ziel dabei ist es, die Benutzeroberfläche (englisch: user interface, UI) und entsprechende Bedienungseigenschaften einer nativen Applikation im Rahmen einer Webanwendung nachzustellen. Die Container-Applikation ermöglicht dabei den Zugriff auf native Funktionalitäten, wie zum Beispiel auf Sensoren, Ressourcen und spezifische Dienste des Betriebssystems. Die Entwicklung hybrider Applikationen wird durch Frameworks vereinfacht, die eine Container-Applikation für Webanwendungen generieren und dem Entwickler eine Brücke zum Zugriff auf Gerätefunktionalitäten zur Verfügung stellen [47]. Diese Brücke stellt der Webanwendung dabei native Dienste und Funktionen über die Programmiersprache JavaScript bereit, wie Abbildung 3.1 vereinfacht illustriert.

Im folgenden Abschnitt wird das PhoneGap bzw. Apache Cordova Framework vorgestellt. Dieses gilt als das bekannteste Framework [50], das zur Entwicklung mobiler

hybrider Applikationen eingesetzt wird. Anschließend werden beispielhaft Frameworks betrachtet, die dieses als Grundlage verwenden und auf die Erstellung von mobilen Webanwendungen und deren Benutzeroberflächen spezialisiert sind.

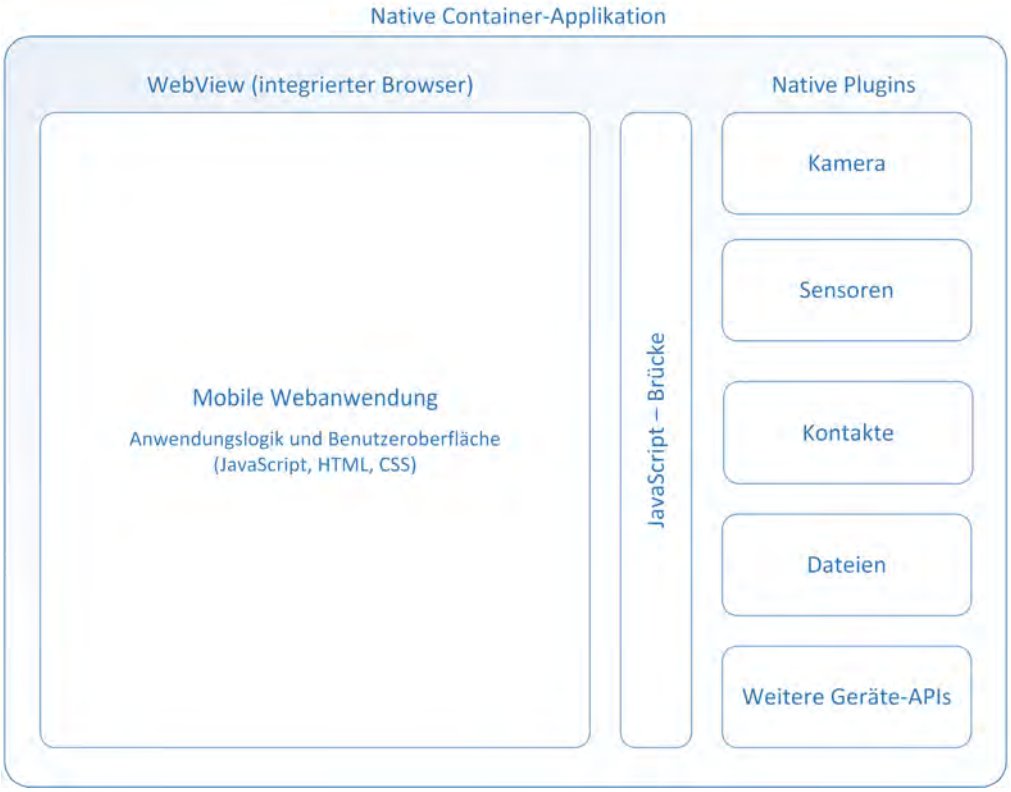

Abbildung 3.1: Vereinfachte Darstellung der Architektur WebView-basierter hybrider Applikationen (vgl. Riippi [51])

3.2.1 PhoneGap / Apache Cordova

Adobes PhoneGap [52] bzw. Apache Cordova [53] gehören zu den bekanntesten kostenfreien Frameworks [50], die auf dem hybriden Ansatz aufbauen und Webanwendungen innerhalb einer nativen Container-Applikation unter Einsatz einer WebView verwenden. Ursprünglich wurde PhoneGap von dem Unternehmen Nitobi Software entwickelt, das im Oktober 2011 von Adobe Systems gekauft wurde [54]. Heute findet die Entwicklung als Open-Source-Projekt im Rahmen der Apache Foundation unter dem Namen Apache Cordova statt [53]. PhoneGap ist mittlerweile zu einer freien Apache Cordova Distribution

geworden, wobei Apache Cordova die Basistechnologie für PhoneGap bildet [55]. Daher wird im weiteren Verlauf dieser Arbeit von Apache Cordova gesprochen.

Apache Cordova hat das Ziel, Webentwickler anzusprechen, die mobile Webanwendungen für verschiedene Plattformen entwickeln und auf den entsprechenen Märkten vertreiben wollen, ohne dabei für jede Plattform eine eigene Implementierung mit plattformspezifischen Programmiersprachen und Werkzeugen erstellen zu müssen [56]. Hierfür integriert Apache Cordova HTML5-Code in eine WebView und stellt eine Fremdfunktionsschnittstelle als Brücke zu nativen Funktionalitäten bereit [47]. Unter einer Fremdfunktionsschnittstelle (englisch: foreign function interface, FFI) „versteht man einen Mechanismus, mit dem ein Programm Routinen oder Dienstleistungen aufrufen oder nutzen kann, die in einer anderen Programmiersprache geschrieben wurden“ [57]. Mit dieser Schnittstelle können über JavaScript Gerätefunktionalitäten angesprochen werden. Hierfür werden Plugins verwendet, die mittels plattformspezifischen Programmiersprachen auf die jeweilige Gerätefunktionalität zugreifen und den Zugriff darauf über JavaScript Methoden abstrahieren [56]. So existieren zum Beispiel Plugins, die den plattformunabhängigen Zugriff auf Kamera, Beschleunigungssensor, GPS-Sensor, Kommunikationsnetzwerke, Speicher oder Kontakte ermöglichen. Eine große Entwicklergemeinschaft stellt zusätzliche Funktionalitäten über eigene Plugins bereit. Das Zusammenspiel der Komponenten einer Apache Cordova Anwendung ist in Abbildung 3.2 dargestellt. Apache Cordova stellt keinerlei Bausteine zur Erstellung einer Benutzeroberfläche zur Verfügung. Dies kann von Frameworks übernommen werden, die auf Apache Cordova aufbauen und in Unterabschnitt 3.2.2 betrachtet werden.

Die Entwicklung mit Apache Cordova hat gegenüber der nativen Anwendungsentwicklung diverse Vorteile. Die Möglichkeit, eine mobile Applikation als Webanwendung zu entwickeln, erlaubt den Einsatz von Webtechnologien, während die Anwendung dennoch über die Märkte der Plattformen vertrieben werden kann. Die Benutzung von Webanwendungen bedeutet auch, dass nur eine einzelne Codebasis für mehrere Plattformen genutzt werden kann. Im Vergleich zur nativen Anwendungsentwicklung kann dies zu geringeren Entwicklungskosten sowie einer besseren Wartbarkeit führen.

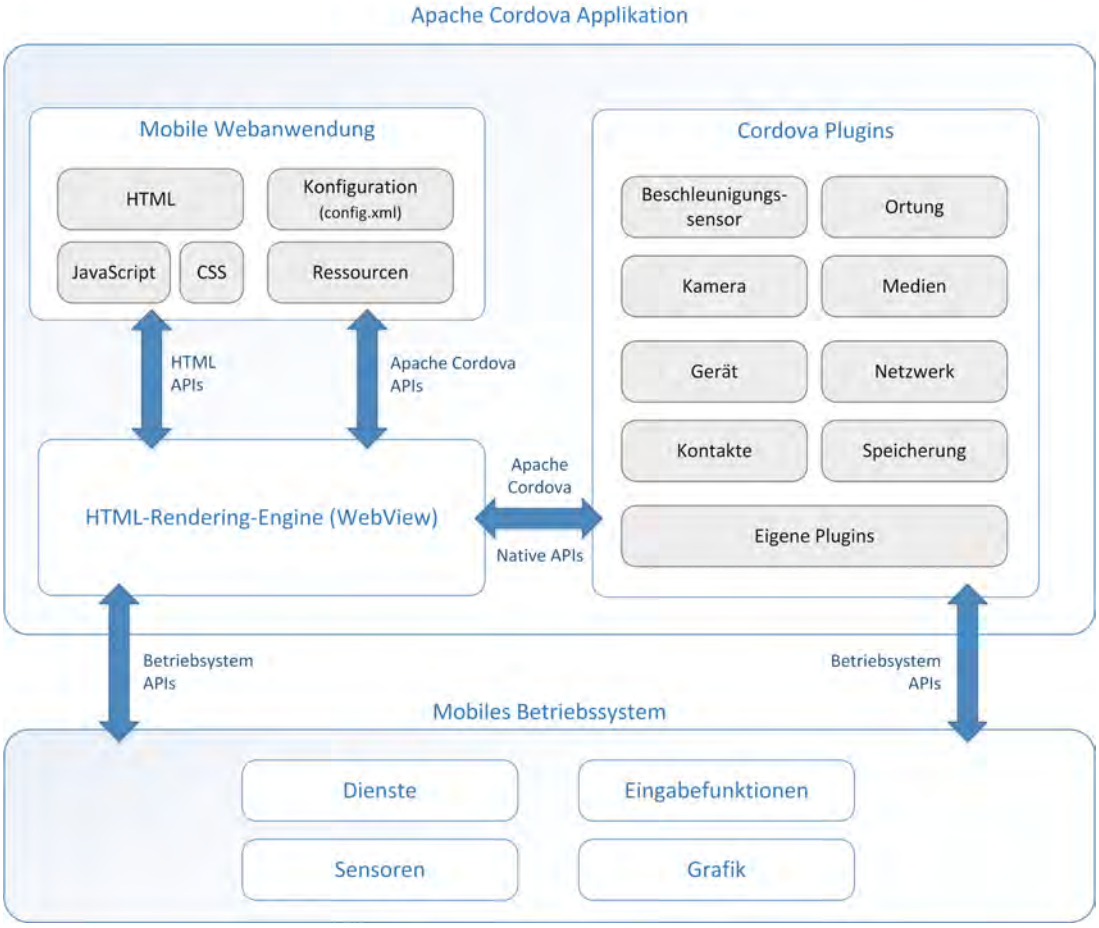

Abbildung 3.2: Das Zusammenspiel der Komponenten einer Apache Cordova Applikation (vgl. [56])

Die Verwendung einer mobilen Anwendung innerhalb einer Apache Cordova Applikation bringt auch Nachteile mit sich. Sobald ein Zugriff auf native Funktionen, Hardware oder Geräteeigenschaften benötigt wird, erfordert die Plugin-Entwicklung zur Bereitstellung nicht vorhandener oder spezieller Funktionalitäten Kenntnisse der nativen Programmiersprachen einzelner Plattformen. Dabei müssen die Funktionalitäten für jede Plattform separat implementiert werden. Des Weiteren kann unter gewissen Gesichtspunkten die Verwendung einer WebView problematisch sein [51]. Mobile Webanwendungen werden innerhalb der WebView mit JavaScript betrieben. Der JavaScript-Code muss von der Browser-Engine interpretiert werden, bevor die Benutzeroberfläche gerendert werden und der Nutzer damit interagieren kann. Dies kann auf schwächeren mobilen

Endgeräten zu Problemen hinsichtlich der Performance führen. Ein weiterer Aspekt ist die Benutzerfreundlichkeit. Eine hybride Applikation, deren Benutzeroberfläche über eine Webanwendung realisiert ist, hat es schwer, das Look-and-Feel einer nativen Benutzeroberfläche nachzuahmen, an das die Nutzer der Plattform gewöhnt sind. Frameworks, wie in Unterabschnitt 3.2.2 vorgestellt, versuchen diesem Problem mit plattformspezifischen CSS-Stilen zu begegnen. Hinzu kommt das Problem der Fragmentierung mobiler Browser-Engines. Mobile Betriebssystem und mobile Endgeräte verwenden unterschiedliche Browser-Engines und unterstützen dadurch verschiedene JavaScript-Funktionalitäten, was zu inkonsistentem Verhalten auf unterschiedlichen Endgeräten führen kann. Dieses Problem wird zum Beispiel vom Crosswalk Project [58] behandelt. Dieses stellt eine Laufzeitumgebung für mobile Webanwendungen zur Verfügung, die in Apache Cordova Applikationen integriert werden kann. Darüber kann für ein einheitliches Verhalten der WebView auf allen Geräten gesorgt werden und es müssen die Eigenheiten plattformspezifischer Browser-Engines nicht mehr berücksichtigt werden.

Im folgenden Abschnitt werden vier Frameworks vorgestellt, mit denen Benutzeroberflächen für mobile Webanwendungen erstellt werden können, die sich für eine Apache Cordova Applikation eignen.

3.2.2 User Interface Frameworks für Apache Cordova / PhoneGap

Dieser Abschnitt stellt vier ausgewählte, kostenfreie Frameworks zur Erstellung von mobilen Webanwendungen vor, die über Apache Cordova bzw. PhoneGap Applikationen eingesetzt werden können. Die Frameworks bieten dabei alle Elemente zur Gestaltung der Benutzeroberfläche und teilweise auch Ansätze zur Strukturierung der Anwendungslogik an.

Framework7

Framework7 [59] ist ein kostenfreies Open-Source-HTML-Framework zur Entwicklung von hybriden mobilen Applikationen. Das Framework wird unter der MIT-Lizenz [60]

zur Verfügung gestellt. Neben HTML kommen die Webtechnologien CSS und JavaScript zum Einsatz. Die Anwendungslogik wird dabei über CSS-Tags und JavaScript im Stile von jQuery [61] geschrieben. Framework7 war zunächst iOS-spezifisch und bot nur das Look-and-Feel von Apples mobilem Betriebssystem an. Mittlerweile wird auch ein Stil angeboten, der die Designspezifikationen von Googles sogenanntem Material-Design unterstützt. Das Framework ist unabhängig von Drittbibliotheken und verwendet eine eigene Bibliothek für das Document-Object-Model (DOM) zur Manipulation der Benutzeroberfläche. Dem Entwickler stehen mit Framework7 diverse Komponenten zur Gestaltung der Benutzeroberfläche zur Verfügung. Hierzu zählen unter anderem Navigationselemente, Toolbars, Overlays, Buttons, verschiedene Listen, Formularelemente oder speziellere Komponenten, wie zum Beispiel Chatansichten. Jedoch sind dabei nicht alle Komponenten sowohl für iOS als auch für Android verfügbar.

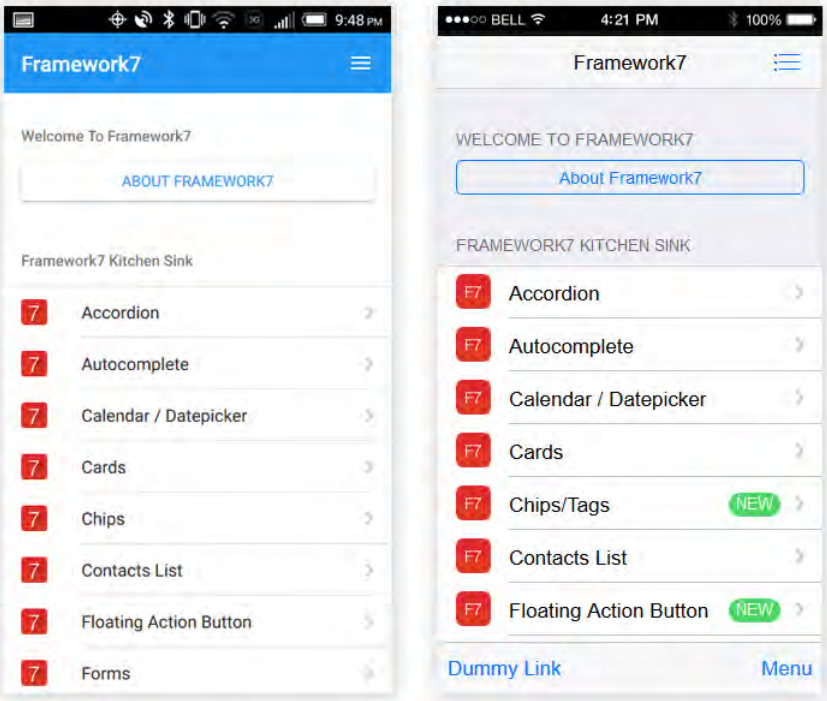

Abbildung 3.3: Beispielhafte Framework7 UI-Komponenten im Vergleich (links Android-Material-Stil, rechts iOS-Stil) [59].

Des Weiteren wird der Entwickler mit einer vorgegebenen Anwendungsstruktur unterstützt. Die Benutzeroberfläche ist dabei über Seiten strukturiert, zwischen denen mit einer JavaScript-Router-API gewechselt werden kann. Dabei kombiniert Framework7 verschiedene Technologien, wie die Zwischenspeicherung oder das Preloading von Seiten, um eine möglichst flüssige Navigation bereitzustellen. Animationen innerhalb der Benutzeroberfläche, wie zum Beispiel Seitenübergänge, werden über hardwarebeschleunigte CSS-Animationen realisiert. Die plattformspezifischen Designs werden zwar größtenteils in voneinander getrennten CSS-Dateien beschrieben. Da auf den verschiedenen Plattformen unterschiedliche UI-Komponenten eingesetzt werden, muss jedoch zur Verwendung eines plattformübergreifenden HTML-Codes teilweise manuell zwischen den Plattformen im Code differenziert werden. Alternativ kann auf eine gemeinsame Codebasis verzichtet und für jede Plattform ein eigener HTML-Code erstellt werden. Abbildung 3.3 zeigt beispielhaft wie sich die plattformspezifischen Stile von Framework7 hinsichtlich Listen, Buttons und Navigationselemente zwischen den Plattformen optisch unterscheiden.

Onsen UI 2

Onsen UI 2 [62] ist ebenfalls ein kostenloses Open-Source-Framework zur Erstellung von Benutzeroberflächen mittels HTML, JavaScript und CSS für hybride mobile Applikationen. Das Framework steht unter der Apache License Version 2.0 [63] und kommt wie Framework7 ohne Drittsoftware aus. Im Gegensatz zu Framework7 wird keine Anwendungstruktur vorgegeben. Vielmehr handelt es sich bei Onsen UI 2 um eine Bibliothek, die plattformspezifische UI-Komponenten bereitstellt. Jedoch werden vom Entwickler Asial Corporation Dokumentationen für den Einsatz mit verschiedenen JavaScript-Applikationsframeworks angeboten. Dabei handelt es sich um AngularJS [64], Angular 2 [65], React [66], Vue.js [67] sowie Meteor [68]. Für diese Frameworks sind auch die APIs standardmäßig optimiert. Theoretisch kann jedoch jedes beliebige JavaScript-Framework unter geringem zusätzlichen Aufwand zusammen mit Onsen UI 2 Komponenten eingesetzt werden. Die Dokumentation geht auf diesen Anpassungsvorgang ein. Onsen UI 2 bietet zahlreiche Benutzeroberflächenkomponenten im iOS-Stil und Android-Material-Stil

an. Alle Onsen UI-Komponenten werden im Gegensatz zu Framework7-Komponenten optisch automatisch an die jeweilige mobile Plattform angepasst. Hierfür werden keine manuellen Differenzierungen im Code benötigt. Des Weiteren ist Onsen UI 2 komplett in das Monaca Localkit [69] der Asial Corporation integriert, das unter anderem ein Kommandozeilen-Tool und einen Debugger für die lokale Entwicklungsumgebung bereitstellt. Dabei wird auch eine Live Reload Funktion angeboten, die eine laufende Applikation aktualisiert, sobald Änderungen an Projektdateien vorgenommen wurden [70].

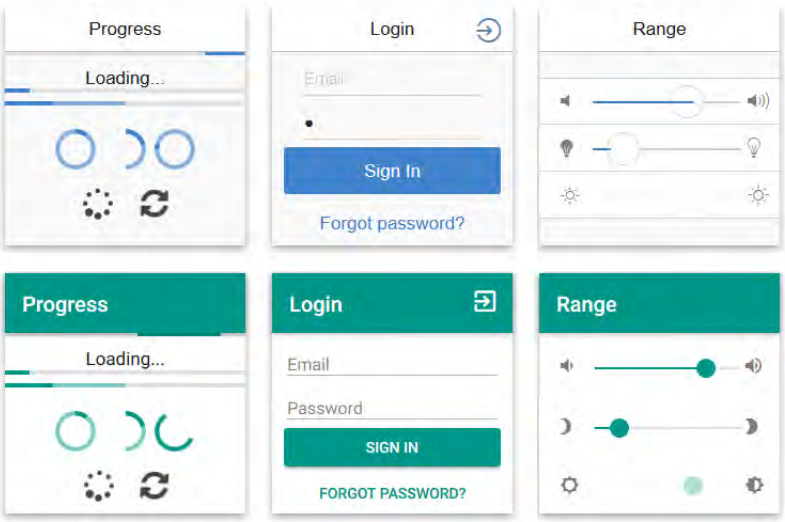

Abbildung 3.4: Nativer Look-and-Feel von beispielhaften Onsen UI-Komponenten (oben iOS-Stil, unten Android Material-Stil) [71].

Intel XDK

Das Intel® XDK [72] wurde im September 2013 [73] veröffentlicht und besteht im Vergleich zu Framework7 oder Onsen UI 2 aus einer kompletten Entwicklungsumgebung zur Erstellung mobiler hybrider Applikationen mit Apache Cordova bzw. PhoneGap. Die Entwicklungsumgebung enthält verschiedene Werkzeuge, um den Entwicklungsprozess hinsichtlich Design, Simulation, Tests, Debugging und Building zu unterstützen [74]. Dadurch soll die Entwicklungszeit einer hybriden mobilen Applikation beschleunigt werden. Die Entwicklungsumgebung kann auf Windows, Mac OS X und Linux einge-

setzt werden. Neben einem Code-Editor [75] wird dem Entwickler auch die Möglichkeit angeboten, die Apache Cordova Applikation zu konfigurieren oder Plugins zu verwalten. Mit dem enthaltenen App-Designer [76] kann die mobile Applikation über eine grafische Benutzeroberfläche per Drag-and-Drop gestaltet werden (vgl. Abbildung 3.5). Im Hintergrund kommt dabei das sogenannte-App Framework [77] zum Einsatz, ein Open-Source-HTML5-Framework, das unter MIT-Lizenz [60] gestellt ist.

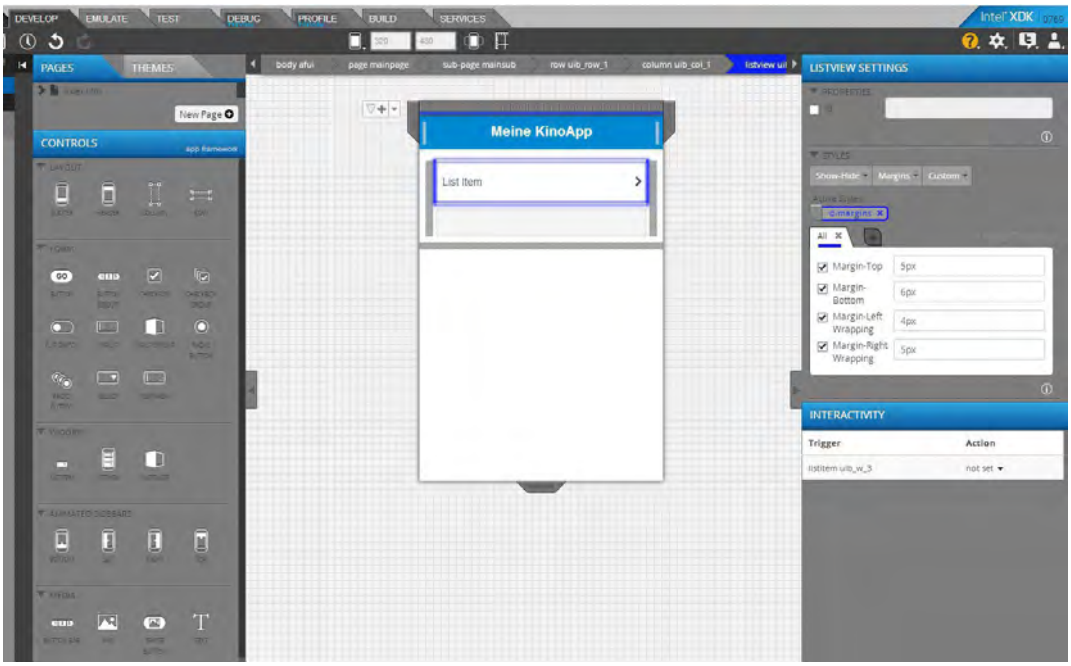

Abbildung 3.5: Der Intel® XDK App Designer zur Gestaltung von Benutzeroberflächen per Drag-and-Drop [78].

Mit dem sogenannten App-Framework-Style-Builder können UI-Elemente angepasst werden [79]. Damit lassen sich auch vorgefertigte Stile für die jeweiligen Zielplattformen auswählen. Das Framework bietet dabei Stile für iOS, Android, Blackberry und Windows 8 an. Mit einer Vorschauapplikation und der Live-Layout-Editing-Funktion [74] kann auf mehreren mobilen Endgeräten gleichzeitig am Layout der Applikation gearbeitet werden. Des Weiteren bietet das Intel® XDK die Möglichkeit, Gerätesensoren zu emulieren. Dabei kann z.B. der Beschleunigungssensor eines mobilen Endgeräts über eine grafische Oberfläche per Mausbewegung simuliert werden. Insgesamt stellt das Intel® XDK eine vollständige Entwicklungsumgebung für mobile hybride Applikationen auf Basis von

Apache Cordova bereit. Über die grafischen Benutzeroberflächen wird insbesondere der Einstieg in die Gestaltung mobiler Applikationen erleichtert.

Ionic

Das Ionic-Framework [80] ist ein kostenfreies Open-Source-Framework, das ebenfalls unter der MIT-Lizenz [60] zur Verfügung gestellt wird. Ionic hat sich zum Ziel gesetzt, eine einzelne Codebasis für alle Plattformen einer mobilen hybriden Applikation zu nutzen. Ionic-Anwendungen basieren auf Angular 2 und werden daher mit JavaScript bzw. TypeScript[1], HTML und CSS bzw. der Erweiterung Sass [82] geschrieben. Das Framework stellt, ähnlich zu den oben vorgestellten, zahlreiche UI-Komponenten [83] zur Verfügung, die das Look-and-Feel nativer Applikationen nachstellen (vgl. Abbildung 3.6). Diese werden automatisch an die optischen Eigenschaften der Zielplattformen angepasst. Unterstützt werden die Stile der mobilen Plattformen iOS, Android und Windows. Zur Kommunikation mit UI-Komponenten wird eine TypeScript-basierte API von Ionic bereitgestellt. Zusätzlich bietet Ionic eine API an, um Key/Value-Paare und JSON-Objekte zu speichern. Diese abstrahiert das darunter liegende Speichersystem plattformspezifisch [84]. Mit der Ionic CLI [85] wird, wie beim Onsen UI 2 Framework (vgl. Abschnitt 3.2.2), ein Kommandozeilen-Tool angeboten, welches zum Testen im lokalen Browser oder zum Erstellen, Emulieren oder Ausführen der Apache-Cordova-Applikation auf mobilen Endgeräten verwendet werden kann. Das Framework untestützt auch eine Funktionalität namens LiveReload [86], mit der Ionic Applikationen automatisch während der Laufzeit aktualisiert werden, wenn Änderungen in Dateien registriert werden. Ionic ist eines der meistgenutzten Frameworks für hybride mobile Applikationen, hinter dem auch eine große Entwickler-Community steht [87]. Das Ionic Ökosystem umfasst mittlerweile mehrere Produkte. Hierzu gehört eine grafische Benutzeroberfläche zur Gestaltung von mobilen hybriden Applikationen per Drag-and-Drop [88] sowie ein cloudbasierter Dienst, der von Entwicklerteams zum Nutzermanagement und zur Entwicklung und Verwaltung

[1] TypeScript [81] ist eine Open-Source Programmiersprache, die von Microsoft entwickelt wurde. Diese ist eine statisch typisierte Obermenge von JavaScript, die auf dem zukünftigen ECMAScript-6-Standard basiert und in reines JavaScript transpiliert wird. Mit TypeScript werden für JavaScript Sprachkonstrukte wie z.B. Typisierung (inklusive Vererbung von Typen und Typüberprüfung zur Kompilierzeit), generische Klassen und Methoden oder Interfaces bereitgestellt.

einer Anwendung genutzt werden kann [89]. Zusätzlich wird ein Marktplatz betrieben [90], auf dem Drittentwickler Plugins oder Applikationsdesigns anbieten und verkaufen können.

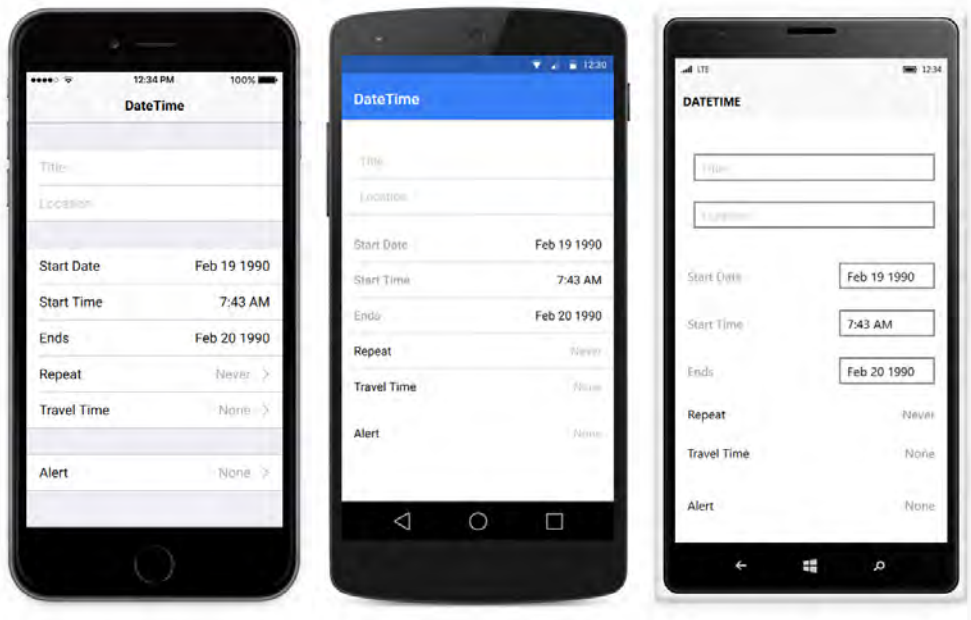

Abbildung 3.6: Nativer Look-and-Feel von beispielhaften Ionic UI-Komponenten (links iOS, mittig Android, rechts Windows) [83].

3.3 Cross-Platform-Ansätze auf Basis von Webtechnologien

Im Gegensatz zu hybriden WebView-basierten Ansätzen generieren Cross-Platform-Ansätze eine native Applikation anstatt einer Container-Applikation. Dabei kommen native UI-Elemente der mobilen Betriebssysteme zum Einsatz. Mit dem Wegfall der WebView befindet sich der Applikationscode auf einer niedrigeren Systemebene. So kann eine deutlich bessere Performance als mit WebView-basierten Applikationen erreicht werden. Gleichzeitig wird das Look-and-Feel von nativen Applikationen updatesicher eingehalten, da die Benutzeroberfläche direkt aus nativen UI-Komponenten aufgebaut wird. Cross-Platform-Ansätze auf Basis von Webtechnologien setzen dabei Technologien wie HTML, JavaScript, CSS oder daran angelehnte Technologien zur Entwicklung mobiler

Applikationen ein. Solche Cross-Platform-Ansätze können ebenfalls als hybrid bezeichnet werden, da webbasierte Technologien mit nativen Funktionalitäten verknüpft werden [48, 46]. Nach Riippi [51] wurde die Entwicklung von Cross-Platform-Frameworks unter anderem aufgrund neuartiger Webtechnologien möglich, die die direkte Manipulation des Document-Object-Models abstrahieren und stattdessen eine deklarative Syntax zur Beschreibung und Konstruktion der Benutzeroberfläche verwenden. Während diese deklarative Beschreibung der Benutzeroberfläche in der Webentwicklung für den Browser wieder in HTML-Elemente übersetzt wird, kann sie zum Beispiel auch in native UI-Elemente mobiler Plattformen überführt werden. Die native Ebene der Betriebssysteme kann bei Cross-Platform-Ansätzen mittels einer zusätzlichen Schicht in der Systemarchitektur angesprochen werden. Diese Schicht bildet eine Brücke zwischen der Anwendungslogik in JavaScript und plattformspezifischen Funktionen in der nativen Programmiersprache. Diese Funktionalität wird dabei unterschiedlich realisiert und in den folgenden Abschnitten für die einzelnen Frameworks separat behandelt. Vereinfacht kann die Architektur von Cross-Platform-Ansätzen, die Webtechnologien verwenden, wie in Abbildung 3.7 dargestellt werden.

Im Folgenden werden vier ausgewählte kostenfreie Open-Source-Frameworks vorgestellt, die den Cross-Platform-Ansatz verfolgen und mit Webtechnologien bzw. daran angelehnten Technologien verwendet werden können.

3.3.1 Fuse

Fuse zählt zu den Cross-Platform-Ansätzen und besteht aus diversen kostenfreien Tools zur Entwicklung nativer mobiler Applikationen für iOS und Android [91]. Dabei versucht Fuse, nicht nur Entwickler, sondern auch Designer mobiler Applikationen anzusprechen [92]. Die Geschäftslogik von Fuse-Applikationen wird in JavaScript geschrieben. Native plattformspezifische UI-Komponenten werden über eine Extensible-Markup-Language (XML), das sogenannte UX-Markup, abstrahiert und gebündelt [91]. UX-Markup wird dabei für die Gestaltung der Benutzeroberfläche und der Interaktion mit dieser eingesetzt. Neben der Verwendung nativer UI-Komponenten, ermöglicht die deklarative Markup-Sprache auch die Erstellung von benutzerdefinierten UI-Komponenten auf Basis

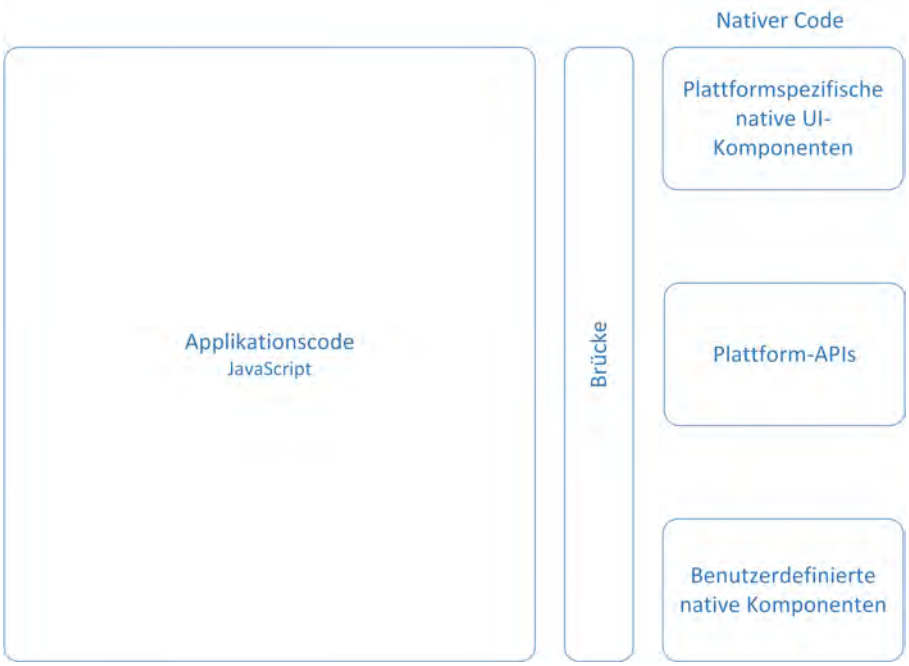

Abbildung 3.7: Vereinfachte Darstellung der Architektur von Cross-Platform-Ansätzen, die Webtechnologien einsetzen (vgl. Riippi [51])

von OpenGL. Native Plattform-APIs sind bei Fuse über JavaScript-APIs abstrahiert und können darüber innerhalb der Anwendungslogik angesprochen werden. Fuse verwendet JavaScript jedoch ausschließlich für die Geschäftslogik [92]. Der JavaScript-Code läuft innerhalb einer virtuellen Maschine (VM) in einem eigenen Thread. Dahingegen werden die Benutzeroberfläche und Framework-Komponenten zunächst in C++ und im Anschluss in nativen Code der Plattformen kompiliert, welcher ebenfalls in einem eigenen Thread läuft. Die Kommunikation zwischen der Geschäftslogik und der Benutzeroberfläche bzw. dem Framework wird dabei über ein reaktives Data Binding realisiert. Durch diesen Ansatz soll eine klare Trennung der Aufgabenbereiche nach dem Separation-of-Concerns-Prinzip [93] erreicht werden. Zusätzlich versucht dieser Ansatz, den JavaScript-Code zu minimieren, um eine bestmögliche Performance über die nativen Programmiersprachen zu erzielen. Zur Entwicklung mit Fuse werden ein Kommandozeilen-Tool, eine Vorschauanwendung, ein Import-Tool für Sketch [94] sowie Plugins für den Sublime Text 3 [95] und Atom Editor [96] bereitgestellt. Dabei wird auch eine Preview-Funktion angeboten [97]. Mit dieser werden Applikationen während

der Laufzeit aktualisiert, wenn eine JavaScript-Datei oder UX-Markup geändert wurde. Fuse befindet sich momentan noch in einer Beta-Version, weshalb einige wenige UI-Komponenten der mobilen Plattformen, wie zum Beispiel die native iOS-Navigation, noch nicht mit Fuse verwendet werden können [98].

3.3.2 Appcelerator Titanium

Die Appcelerator-Plattform besteht aus verschiedenen Werkzeugen, die es ermöglichen, plattformübergreifende native Applikationen für Android, iOS und Windows Phone zu entwickeln [99]. Mit dem Titanium-Software-Development-Kit [100] können native Applikationen in JavaScript geschrieben werden. Das Software-Development-Kit (SDK) ist eine Open-Source-Software und unter der Apache License Version 2.0 [63] lizenziert. Die sogenannte Titanium API ist Teil des SDK und abstrahiert native APIs der mobilen Betriebssysteme, welche dadurch in JavaScript angesprochen werden können. Der JavaScript-Code läuft in einer plattformspezifischen virtuellen Maschine für JavaScript. Das Titanium-SDK kann um fehlende Funktionen erweitert werden, indem Module in den plattformspezifischen Programmiersprachen geschrieben werden [101]. Seit Ende 2016 wird mit Hyperloop [102] im Rahmen der kostenpflichtigen Programme von Appcelerator eine Erweiterung angeboten, mit dem über JavaScript alle nativen APIs von iOS und Android direkt angesprochen werden können. Damit müssen native Plattformfunktionalitäten nicht mehr über die genannten Module in plattformspezifischen Programmiersprachen angesprochen werden, sondern können direkt mit JavaScript verwendet werden [103]. Über das Open-Source-Framework Alloy [104] wird für das Titanium-SDK eine Model-View-Controller-Architektur (MVC) bereitgestellt. Ähnlich wie bei Fuse kann die Benutzeroberfläche über eine XML-basierte Markup-Sprache mit nativen UI-Elementen konstruiert werden. Die Gestaltung der UI-Elemente erfolgt über sogenannte Titanium-Style-Sheets (TSS) [105]. TSS-Dateien nutzen dabei eine an die JavaScript-Object-Notation (JSON) angelehnte Syntax. Des Weiteren wird ein Kommandozeilen-Tool [106] zur Emulation, zum Testen und zum Erstellen von Applikationen bereitgestellt. Im Rahmen der kostenpflichtigen Programme werden zur Entwicklung mobiler Appcelerator-Applikationen unter anderem auch eine integrierte Entwicklungs-

umgebung [107] sowie ein Mobile-Backend-as-a-Service (MBaaS) Programm [108] angeboten.

3.3.3 React Native

React Native [109] wurde von Facebook entwickelt und ermöglicht die Entwicklung nativer mobiler Applikationen für iOS und Android nach dem Cross-Platform-Ansatz mit JavaScript. Die Entwicklung für die Universal-Windows-Platform ist seit 2016 über ein von Microsoft bereitgestelltes Plugin möglich [110]. React Native ist ein kostenfreies Open-Source-Framework, lizenziert unter der BSD-Lizenz [111], und basiert auf Facebooks React-JavaScript-Bibliothek [66]. Das komponentenbasierte React setzt die von Riippi [51] angesprochene Abstraktion des DOM um, indem es Komponenten verwendet, die deklarativ beschrieben werden. React setzt auf eine XML-basierte Syntax namens JSX. Die UI-Komponenten werden bei React in einem sogenannten virtuellen DOM [112, 113], einer leichtgewichtigen Abstraktion des realen DOM, verwaltet. Dadurch können Aktualisierungen der Benutzeroberfläche im Vergleich zur Aktualisierung echter DOM-Elemente des Browsers effektiver durchgeführt werden. Änderungen des virtuellen DOM werden dabei von React auf das tatsächliche DOM des Browsers übertragen, indem nur die spezifischen Teile des DOMs aktualisiert werden, bei denen auch eine Änderung stattgefunden hat. Bei React Native kommt ebenfalls ein virtuelles DOM zum Einsatz, womit jedoch nicht das DOM einer Webanwendung abgebildet wird. Stattdessen wird dieses virtuelle DOM verwendet, um die nativen UI-Komponenten der mobilen Applikation zu rendern. Der JavaScript-Interpreter bzw. die JavaScript-Laufzeitumgebung kann über eine asynchrone Brücke mit den plattformspezifischen UI-Komponenten und nativen APIs kommunizieren. Um eine performante Benutzeroberfläche bereitstellen zu können, setzt React Native dabei auf einen Multithreading-Ansatz, der Geschäftslogik, Benutzeroberfläche und andere Aufgaben parallelisiert [113]. Die Gestaltung von nativen UI-Komponenten wird über JavaScript-StyleSheet-Objekte [114] realisiert. React Native bietet zur Verwendung dieser eine CSS-ähnliche Syntax an. Zur Bereitstellung nicht vorhandener Funktionalitäten können native Module in den plattformspezifischen Programmiersprachen entwickelt werden. Diese Module werden beim Framework registriert,

wodurch sie innerhalb der Anwendungslogik über JavaScript angesprochen werden können. Ähnlich dazu können auch benutzerdefinierte UI-Komponenten erstellt werden. React Native stellt für die lokale Entwicklungsumgebung ein Kommandozeilen-Tool zum Testen, Emulieren und Erstellen mobiler Applikationen bereit. Dabei stehen die sogenannte Live-Reload- und Hot-Reloading-Funktion zur Verfügung [115], über welche die Applikation während der Laufzeit automatisch aktualisiert wird, sobald Dateien verändert werden.

3.3.4 Native Script

Bei NativeScript [116] handelt es sich wie bei den vorherigen drei Ansätzen um ein kostenloses Cross-Platform-Open-Source-Framework zur Entwicklung nativer mobiler Applikationen. Dabei werden iOS und Android unterstützt. Ab 2017 soll es auch möglich sein, Windows-Universal-Applikationen mit NativeScript zu entwickeln [117]. NativeScript wurde von Telerik, einer Tochterfirma der Progress Software Corporation, entwickelt und steht unter der Apache License Version 2.0 [63]. Mit NativeScript können native Applikationen mit JavaScript oder TypeScript und Angular 2 entwickelt werden. Zur Beschreibung der Benutzeroberfläche verwendet NativeScript XML [118], das in native UI-Komponenten der jeweiligen Plattformen überführt wird. Die Gestaltung der Benutzeroberfläche erfolgt dabei mit CSS, wobei aber nur eine Teilmenge der CSS-Sprache unterstützt wird [119]. NativeScript bietet zwei Ansätze zur Entwicklung mobiler Applikationen an. Mit NativeScript Core können mobile Applikationen mit JavaScript bzw. TypeScript, ähnlich zum Ansatz von Appcelerator (vgl. Unterabschnitt 3.3.2), entwickelt werden. Seit 2016 bietet Telerik in Zusammenarbeit mit Google auch eine Angular-2-Integration für NativeScript an [120, 121]. Code von Angular-basierten Webanwendungen kann dadurch auch in NativeScript-Applikationen wiederverwendet werden und umgekehrt. Durch den Einsatz von Angular und CSS stellt NativeScript unter den vorgestellten Cross-Platform-Ansätzen, das Framework dar, das den klassischen Webtechnologien am nächsten steht. Um eine gemeinsame Codebasis für alle mobilen Plattformen zu ermöglichen, beinhaltet das NativeScript-Framework Module, die native UI-Komponenten und native APIs plattformunabhängig abstrahieren [117]. Zusätzlich ist bei NativeS-

cript über eine Brücke der direkte Zugriff auf alle plattformspezifischen nativen APIs aus JavaScript bzw. TypeScript möglich. Hierbei vefolgt NativeScript einen ähnlichen Ansatz wie Appcelerator mit der proprietären Hyperloop-Erweiterung. Dies ermöglicht Entwicklern, native APIs zu verwenden, um zum Beispiel eigene Erweiterungen zu implementieren, ohne die nativen Programmiersprachen der mobilen Plattformen beherrschen zu müssen. So können beispielsweise in JavaScript eigene UI-Komponenten auf Basis von nativen Komponenten erstellt werden. Des Weiteren ist es möglich, spezielle native Funktionalitäten zu verwenden, die nicht durch die JavaScript-APIs von NativeScript abgedeckt werden. Im Gegensatz zu React Native (vgl. Unterabschnitt 3.3.3) setzt NativeScript auf ein Single-Threading-Modell, da dies einen performanteren Zugriff auf die native Ebene ermöglicht [122]. Damit die Benutzeroberfläche nicht von rechenintensiven Aufgaben blockiert wird, kann man sogenannte Worker einsetzen, die JavaScript-Code in eigenen Threads ausführen [123]. Wie auch bei den vorhergehenden Ansätzen wird ein Kommandozeilen-Tool für die lokale Entwicklungsumgebung bereitgestellt. Zudem wird eine sogenannte Livesync-Funktion [124] angeboten, die mobile Applikationen während der Laufzeit aktualisiert, sobald eine Datei verändert und gespeichert wurde.

Der Cross-Platform-Ansatz von NativeScript vereint wie kein anderes der vorgestellten Frameworks die Verwendung von klassischen Webtechnologien mit der Möglichkeit, native APIs und Komponenten direkt aus JavaScript bzw. TypeScript anzusprechen. Der direkte Zugriff auf die native Ebene erleichtert die Entwicklung von Funktionalitäten, da die syntaktischen Eigenschaften der plattformspezifischen Programmiersprachen nicht beherrscht werden müssen. Durch die Integration von Angular 2 können mobile Applikationen in NativeScript strukturiert werden. Gleichzeitig ist es möglich, Code mit Angular-Webanwendungen zu teilen. Aus diesen Gründen wurde NativeScript, im Rahmen dieser Arbeit, als Basis für die plattformübergreifende Implementierung eines Frameworks für Kontextsensitivität ausgewählt.

Das folgende Kapitel geht auf Details der Architektur von NativeScript ein und stellt die Applikationsentwicklung mit NativeScript und Angular vor.

4

NativeScript

Das in Unterabschnitt 3.3.4 vorgestellte Open-Source-Framework NativeScript ermöglicht die Entwicklung nativer mobiler Applikationen für iOS und Android mit einer einzigen Codebasis. Hierfür kommen XML, JavaScript bzw. TypeScript, CSS und Angular zum Einsatz. NativeScript bietet gegenüber anderen kostenfreien Frameworks den Vorteil, dass im JavaScript-basierten Applikationscode native Klassen und Methoden der mobilen Betriebssysteme verwendet werden können. Daher kann NativeScript um eigene Funktionalitäten, die auf nativen APIs basieren, erweitert werden, ohne dass dafür die Syntax der plattformspezifischen Programmiersprachen bekannt sein muss. Diese Eigenschaften von NativeScript führten zu der Entscheidung, das hier vorgestellte plattformübergreifende Framework für Kontextsensitivität auf Basis von NativeScript zu entwickeln.

In diesem Kapitel wird auf grundlegende Aspekte der Anwendungsentwicklung mit NativeScript eingegangen. Dabei werden das Konzept der Komponenten, die zugrundeliegende Architektur, die Abstraktion von nativen Funktionen und die Plugin-Entwicklung behandelt. Abschließend werden diverse Einschränkungen des Frameworks betrachtet.

Die Entwicklung mobiler Applikationen mit NativeScript kann auf zwei unterschiedliche Arten realisiert werden. Zum einen können mit NativeScript-Core mobile Anwendungen mit reinem JavaScript, XML und CSS entwickelt werden. Zum anderen kann die Entwicklung mit Angular 2 und TypeScript erfolgen. Dieses Kapitel geht insbesondere auf die zweite Möglichkeit ein, welche im Rahmen dieser Arbeit verwendet wurde. Bei diesem Ansatz wird der Entwicklungsprozess durch Angular mit einer vorgegebenen Strukturierung der Applikation und zusätzlichen Funktionalitäten unterstützt. Mit Type-

Script werden zusätzliche Sprachkonstrukte für JavaScript bereitgestellt, wodurch die Programmierung vereinfacht wird.

4.1 Anwendungsentwicklung mit Angular 2

Seit 2016 ist es möglich, plattformübergreifende mobile Applikationen mit NativeScript und Angular 2 zu entwickeln [120]. Damit können Entwickler, die Erfahrung mit Angular haben, vorhandenes Wissen zur Entwicklung mobiler Applikationen mit NativeScript einsetzen. Die Integration von Angular in NativeScript wurde mit der Veröffentlichung von Angular 2 und den damit verbundenen Änderungen in der Angular-Architektur möglich. Während Angular 1 noch sehr eng mit dem DOM verknüpft und damit auf Browser-basierte Umgebungen eingeschränkt war, wurde Angular 2 von dem DOM entkoppelt. Dadurch wurde es möglich, Angular auch außerhalb von Browsern einzusetzen. Angular 2 ist in TypeScript geschrieben und basiert auf Komponenten und Direktiven [125]. Der größte Unterschied zwischen der herkömmlichen Verwendung von Angular und der Verwendung in NativeScript besteht darin, dass keine Browser-basierten HTML-Elemente, sondern die nativen UI-Komponenten von NativeScript verwendet werden. Im Folgenden wird, angelehnt an die offizielle NativeScript-Dokumentation [126], die grundlegende Anwendungsentwicklung mit NativeScript und Angular anhand von einfachen Beispielen vorgestellt.

Typische NativeScript-Applikationen, die Angular verwenden, bestehen hauptsächlich aus TypeScript-Dateien für die Anwendungslogik, XML-Dateien zur Beschreibung der Benutzeroberfläche und CSS-Dateien zur Gestaltung dieser. Die Benutzeroberfläche einer mobilen Applikation wird dabei über Angular-Komponenten beschrieben. Angular-Applikationen und damit auch NativeScript-Applikationen bestehen aus einem Baum solcher Komponenten. Eine beispielhafte Definition einer Komponente ist in Listing 1 dargestellt.

Innerhalb der Klasse einer Komponente, hier im Block `class HelloWorldComponent`, kann die Logik für die Komponente implementiert werden. Mit dem `@Component`-Decorator wird die Klasse als Angular-Komponente markiert. Über diesen können ver-

```
import { Component } from "@angular/core";

@Component({
  selector: "hello-world",
  template: `
    <Label [text]="message" (tap)="onTap()"></Label>
  `
})

export class HelloWorldComponent {
  public message: string = "Hello World!";

  public onTap() {
    this.message = "Hello again!";
  }
}
```

Listing 1: Beispielhafte Definition einer Komponente mit NativeScript und Angular

schiedene Metadaten, wie zum Beispiel ein Template, bestimmt werden. Die `selector`-Eigenschaft definiert dabei einen Namen für die Angular-Komponente, damit diese innerhalb anderer Templates wiederverwendet werden kann.

Templates definieren die Ansicht (englisch: view) einer Komponente und werden in NativeScript mit XML geschrieben. Dabei kommen native UI-Elemente zum Einsatz, die von NativeScript plattformunabhängig definiert sind. Innerhalb der Templates kann die komplette Template-Syntax von Angular [127] verwendet werden. Dadurch kann Angulars Data-Binding zur Kommunikation zwischen Klasse und Template eingesetzt werden. Im dargestellten Beispiel sind bereits zwei Formen des Data-Binding-Mechanismus zu erkennen. Data-Binding wird im Rahmen der Template-Syntax über Klammerungen von Elementattributen definiert. Es wird zwischen drei Arten des Data-Bindings unterschieden [128]:

- **One-Way-Data-Binding:** Beim One-Way-Data-Binding können Elemente der Benutzeroberfläche, also des Templates, an das Modell gebunden werden. Im dargestellten Beispiel wird über `[text]="message"` das `text`-Attribut an das Klassenattribut `message` des Modells gebunden. Immer, wenn das `message`-Attribut

aktualisiert wird, wird auch der Text des `<Label>`-Elements aktualisiert. Die Information wird dabei vom Modell auf das Template übertragen.

- **Event-Binding:** Das Event-Bining ermöglicht die Kommunikation vom Template zur Klasse. Im dargestellten Beispiel wird dies über das Attribut `(tap)="onTap()"` realisiert. Die `onTap()`-Methode der Klasse wird aufgerufen, sobald der `tap`-Event für das `<Label>`-Element registriert wird. In diesem Fall fließt die Information von der Benutzeroberfläche zum Modell.
- **Two-Way-Data-Binding:** Bei dieser Art des Data-Bindings werden die vorherigen beiden Arten kombiniert. Dadurch ist ein Informationsaustausch in beide Richtungen möglich. Ein typisches Beispiel hierfür ist die Verwendung eines `<TextField>`-Elements, bei dem der Nutzer einen Text eingeben kann, der dann an das Modell übermittelt wird. Gleichzeitig kann das Modell das `<TextField>`-Element auch mit einem Text füllen. Das Two-Way-Data-Binding kann dabei über eine doppelte Klammerung, wie z.B. in `[(ngModel)]="model.message"`, realisiert werden.

Mit diesem Konzept der Komponenten können bei NativeScript sowohl komplette Ansichten als auch einzelne Elemente solcher Ansichten sowie die dazugehörige Logik implementiert werden. Zur übersichtlicheren Gestaltung des Codes und zur deutlicheren Aufteilung nach dem MVC-Muster können über den `@Component`-Decorator Templates in HTML-Dateien und die Gestaltung dieser in CSS-Dateien ausgelagert werden. Dabei ist es möglich, plattformspezifische CSS-Dateien bereitzustellen, deren Name automatisch aufgelöst wird und die nur bei der jeweiligen Plattform angewendet werden (vgl. Listing 2).

```
1  @Component({
2    selector: "hello-world",
3    templateUrl: 'hello-world.html',
4    styleUrls: ["hello-world-common.css", "hello-world.android.css",
5                "hello-world.ios.css"]
6  })
```

Listing 2: Auslagerung von Template und CSS bei NativeScript-Komponenten

In NativeScript-Applikationen werden noch weitere Angular-Konzepte verwendet. Neben der `@Component`-Direktive zur Definition von Komponenten können auch UI-Elemente oder Elementattribute mit Angular-Direktiven gesteuert werden. Des Weiteren kann die Dependency-Injection von Angular genutzt werden, um zum Beispiel Abhängigkeiten einer Komponente von Klassen oder Diensten zur Laufzeit zu bestimmen. Die Navigation innerhalb einer NativeScript-Angular-Applikation wird über den Component-Router von Angular ermöglicht. Hierbei müssen Komponenten, die Applikationsansichten auf höchster Ebene repräsentieren, als sogenannte Seiten (englisch: pages) über eine NativeScript-API registriert werden. Diese Seitennavigation wird von NativeScript automatisch in die nativen Navigationselemente der jeweiligen Plattform integriert.

4.2 Laufzeitumgebung und native Schnittstellen

NativeScript stellt nicht nur eine Ansammlung von Bibliotheken dar, über die native APIs der mobilen Plattformen aufgerufen werden können. Vielmehr stellt NativeScript eine eigene Laufzeitumgebung zur Verfügung [117]. Diese beinhaltet eine virtuelle JavaScript-Maschine, in die eine Brücke integriert ist, über die mit den nativen APIs von Android bzw. iOS kommuniziert werden kann. Die virtuellen JavaScript-Maschinen bestehen aus Googles V8 für Android und aus Apples WebKit-Implementierung JavaScriptCore für iOS. Über diese Brücke, verspricht Telerik [126], sollen 100% der nativen APIs verwendet werden können. Der Zugriff auf die nativen APIs kann ohne zusätzliche Bibliotheken direkt in JavaScript bzw. TypeScript erfolgen. Dies wird durch das folgende JavaScript-Beispiel in Listing 3 mit der Java-File-API für Android demonstriert.

```
1  var path = "some/path/file.txt";
2  var file = new java.io.File(path);
```

Listing 3: Zugriff auf native APIs in NativeScript

Beim Aufruf der nativen File-API werden intern implizit folgende Schritte von NativeScript durchgeführt [117, 129]:

1. Der in Listing 3 dargestellte JavaScript-Code wird von der virtuellen JavaScript Maschine interpretiert und ausgeführt.
2. Eine in die virtuelle Maschine integrierte Brücke bestimmt die native Methode, die aufgerufen werden muss.
3. Ein Marshalling-Dienst wandelt den JavaScript-String `path` in ein `java.lang.String`-Objekt um.
4. Die Laufzeitumgebung führt die native Methode aus, speichert für die zurückgegebene native Objektinstanz eine Referenz zwischen und erstellt ein stellvertretendes JavaScript-Objekt. Dieses wird als Proxy für das native `java.io.File`-Objekt bezeichnet. Wird eine Methode des Proxy-Objekts im weiteren Verlauf des Programms aufgerufen, wird dieser Aufruf an die zugrundeliegende native Objektinstanz weitergeleitet.
5. Das Proxy-Objekt wird als lokale Variable `file` abgespeichert und weiterer JavaScript-Code kann ausgeführt werden.

NativeScript verwendet eine virtuelle JavaScript-Maschine, die den Applikationscode interpretiert und ausführt. In diese virtuelle Maschine wird die angesprochene Brücke zum Zugriff auf native APIs integriert [129]. NativeScript injiziert hierfür über eine API ein Objekt in den globalen Sichtbarkeitsbereich der virtuellen Maschine. Das injizierte Objekt enthält dabei Callbacks für Methoden und Objekte, die plattformspezifische native APIs repräsentieren. Wenn aus dem JavaScript-Code heraus nun eine native API angefordert wird, ist bei der virtuellen Maschine ein Callback dafür registriert. Dieser Callback unterbricht die Anfrage und führt einen C++-Code aus, der die tatsächliche native API der Plattform aufruft. Weil mit C++ bei Android nicht auf die Java-APIs der Plattform zugegriffen werden kann, wird Androids Java-Native-Interface (JNI) [130] eingesetzt, welches eine Brücke zwischen C++ und Java bereitstellt. Bei iOS wird dies nicht benötigt, da C++-Code direkt die Objective-C-APIs des Betriebssystems aufrufen kann.

Zur automatisierten Abbildung der nativen APIs in JavaScript muss NativeScript zunächst die sogenannten Metadaten, eine Liste der nativen APIs, erzeugen. Dafür wird Reflexion verwendet. Da die Generierung der Metadaten aus Performancegründen nicht zur Laufzeit geschehen kann, werden diese von NativeScript im Voraus generiert und in das Framework eingebettet.

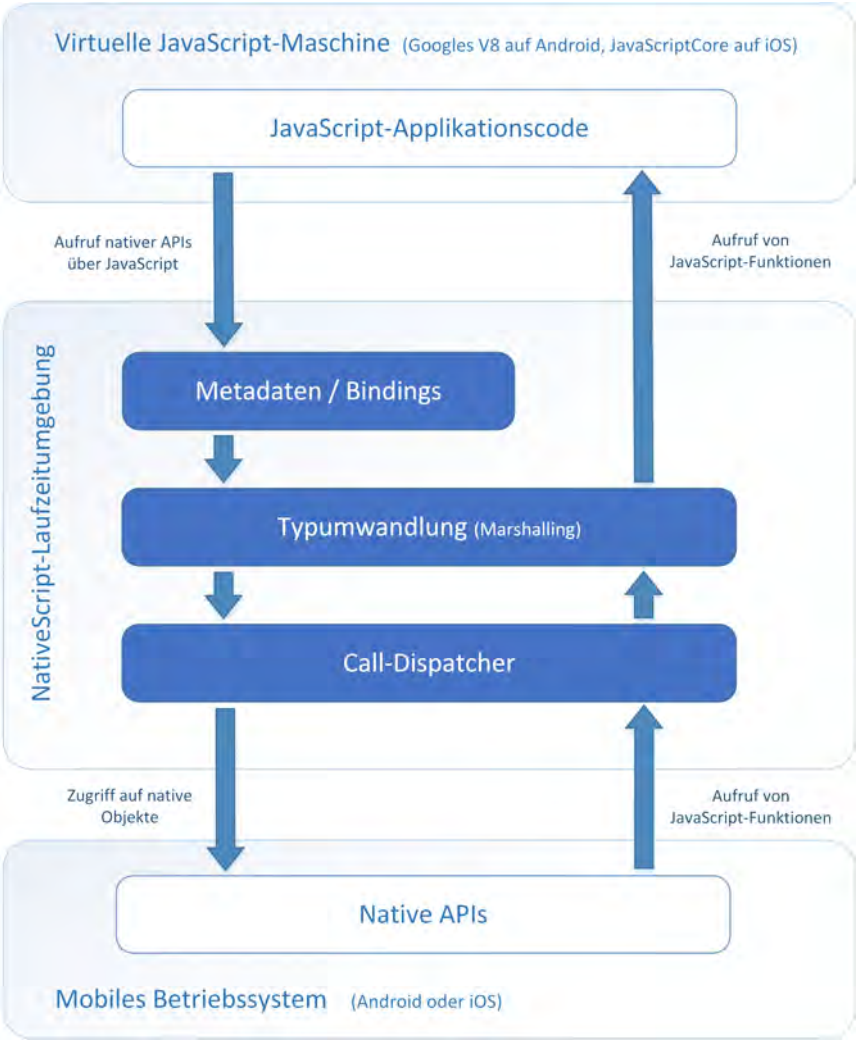

Abbildung 4.1: Ablauf des Zugriffs auf native APIs mit NativeScript (vgl. [131])

Abbildung 4.1 bietet einen abstrahierten Überblick über den Ausführungsablauf beim Zugriff auf eine native API. Bei einem solchen Aufruf innerhalb des JavaScript-Applikationscodes wird über die Metadaten die entsprechende native API bestimmt. An-

schließend werden JavaScript-Parameter im Rahmen der Typumwandlung zu nativen Objekten konvertiert. Der sogenannte Call-Dispatcher ist dann dafür verantwortlich, den tatsächlichen Aufruf der nativen API durchzuführen und die Ergebnisse wieder zurück an den Applikationscode zu transportieren. Die nativen Ergebnisse durchlaufen dabei die Typumwandlung und werden in JavaScript-Objekte oder Methodenaufrufe umgewandelt.

Aufgrund dieser Architektur basiert NativeScript im Gegensatz zu anderen Ansätzen, wie zum Beispiel React Native (vgl. Unterabschnitt 3.3.3), auf einem Single-Threading-Modell [122]. Dabei werden alle Ebenen der NativeScript-Architektur im UI-Thread der mobilen Applikation ausgeführt. Da keine Inter-Thread-Kommunikation benötigt wird und Typumwandlungsprozesse minimiert werden, kann der Zugriff auf native APIs direkter und performanter umgesetzt werden als bei einem Multi-Threading-Modell. Des Weiteren muss keine zusätzliche Abstraktionsschicht wie bei ReactNative oder Fuse eingesetzt werden. Bei Verwendung eines Single-Threading-Modells kann es vorkommen, dass die Benutzeroberfläche durch rechenintensive Aufgaben blockiert wird. Daher wurden rechenintensive oder blockierende Aufgaben, wie zum Beispiel Dateioperationen, in native Module ausgelagert. Die Entwicklung solcher Module erfordert jedoch Kenntnisse der plattformspezifischen Programmiersprachen, was nicht zur Philosophie von NativeScript passt. Seit 2016 wird von NativeScript eine Lösung für dieses Problem angeboten. Über sogenannte Worker kann JavaScript-Code in Hintergrund-Threads ausgeführt werden [123]. Worker sind Skripte, die in einem eigenen Thread und damit in einem isolierten Kontext laufen. Die NativeScript-Architektur ermöglicht auch im Kontext solcher Skripte einen Zugriff auf die nativen APIs. Zur Steuerung von Worker-Skripten wird eine Schnittstelle bereitgestellt [132], die auch eine simple JSON-basierte Kommunikation erlaubt.

Mit NativeScript können nicht nur native APIs angesprochen werden. Es ist auch möglich mit JavaScript bzw. TypeScript, native Klassen nach dem Vererbungskonzept zu erweitern sowie Java-Interfaces für Android und Objective-C-Protokolle für iOS zu implementieren. Zur Erweiterung nach dem Vererbungskonzept stellt NativeScript eine `extend()`-Methode für die Abbildungen nativer Klassen in JavaScript zur Verfügung. Diese `extend()`-Methode wird auch zur Implementierung von iOS-Protokollen verwendet [133]. Dahingegen werden Java-Interfaces für Android mit einer Syntax implementiert,

die der von anonymen Java-Klassen entspricht [134]. Zur nativen Übersetzung solcher Klassen wird für Android ein sogenannter Binding-Generator eingesetzt, der zur Kompilierzeit native Klassen anhand der JavaScript-Implementierung erzeugt [135]. Für iOS werden die JavaScript-Implementierungen nativer Klassen oder Protokolle bei der Objective-C-Laufzeitumgebung registriert [133]. Die einfachen Beispiele in Listing 4 und 5 zeigen die Erweiterung nativer Android- und iOS-Klassen sowie das Überschreiben von Methoden dieser Klassen in JavaScript.

```
var myButton = android.widget.Button.extend("MyButtonClassName", {
    // constructor
    init: function() {
      console.log("constructor called");
    },

    // method overriding
    setEnabled: function(enabled) {
      this.super.setEnable(enabled);
    }
  });
```

Listing 4: Erweiterung nativer Android-Klassen in NativeScript

```
var myViewController = UIViewController.extend({
    // method overriding
    viewDidLoad: function () {
      this.super.viewDidLoad();

      console.log("view loaded");
    },
  }, {
    name: "MyViewControllerClassName"
  });
```

Listing 5: Erweiterung nativer iOS-Klassen in NativeScript

4.3 UI-Komponenten und Module

Wie im vorherigen Abschnitt dargestellt, ermöglicht NativeScript den Zugriff auf die nativen APIs der mobilen Betriebssysteme. Aufgrund der unterschiedlichen nativen APIs muss eine spezielle Aufgabe für alle Zielplattformen einzeln implementiert werden. In der Applikationslogik kann zwischen den Plattformen differenziert werden. So wird sichergestellt, dass native API-Aufrufe auch nur für die entsprechende Plattform durchgeführt werden. Dies ist in Listing 6 beispielhaft für das Erstellen einer Datei über die jeweiligen APIs von Android und iOS dargestellt.

```
1  import app = require("application");
2
3  var path = "some/path/file.txt";
4
5  if (app.android) {
6    var file = new java.io.File(path);
7  } else if (app.ios) {
8    var fileManager = NSFileManager.defaultManager();
9    fileManager.createFileAtPathContentsAttributes(path);
10 }
```

Listing 6: Differenzierung zwischen den mobilen Plattformen in NativeScript

Diese Herangehensweise entspricht jedoch nicht der Cross-Platform-Philosophie von NativeScript. Telerik [126] versucht mit NativeScript, eine möglichst plattformunabhängige Entwicklungsumgebung bereitzustellen. Dafür soll eine einzelne Codebasis für alle Plattformen eingesetzt werden können. So muss nicht zwischen den Plattformen, wie in Listing 6, unterschieden werden. Des Weiteren soll die Entwicklung ohne Kenntnis der nativen APIs möglich sein. Dies erreicht NativeScript, indem native UI-Komponenten und plattformspezifische APIs innerhalb von Modulen abstrahiert und gebündelt sind [117, 136]. Diese Module sind in das NativeScript-Framework integriert und Bestandteil jeder NativeScript-Applikation. Zum Beispiel werden häufig verwendete Gerätefunktionalitäten, Plattformeigenschaften, Logging, Netzwerkeigenschaften, Netzwerkkommunikation, Zugriff auf Bilddateien, zeitbasierte Codeausführung oder spezielle Datenty-

pen über Module bereitgestellt. Hinzu kommen Module, die native UI-Komponenten, wie zum Beispiel verschiedene Layouts, Buttons, Listenansichten, Dialogansichten oder Formularfelder, abstrahieren. Das Codebeispiel in Listing 6 zeigt bereits, wie das `application`-Modul über `require` eingebunden werden kann. In Abbildung 4.2 wird grafisch dargestellt, wie sich die Abstraktionsschicht der Module nach Atanasov [117] in die NativeScript-Architektur integriert.

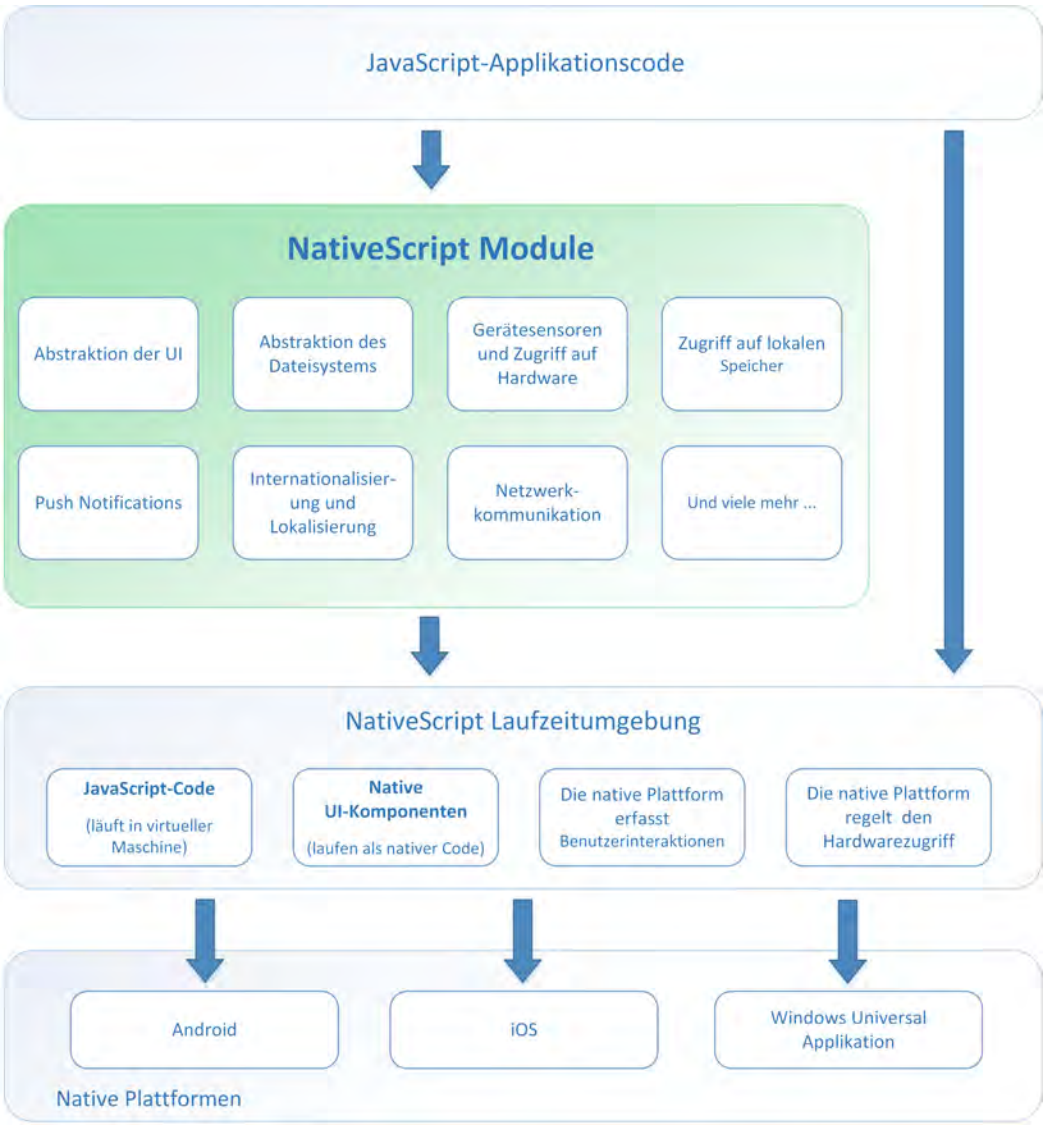

Abbildung 4.2: Die Rolle von Modulen in der NativeScript-Architektur

Wie solche Module aufgebaut sind und wie damit das NativeScript-Framework um zusätzliche Funktionalitäten mit eigenen Plugins erweitert werden kann, wird im folgenden Abschnitt näher betrachtet.

4.4 Plugin-Entwicklung

Plugins für NativeScript entsprechen NativeScript-Modulen und werden, wie im vorherigen Abschnitt vorgestellt, in die Applikationsarchitektur integriert. Die Implementierung folgt dabei den Spezifikationen von CommonJS [137]. Dadurch ist es möglich, dass nur JavaScript-Code geladen wird, der von der Applikation in ihrem aktuellen Zustand auch benötigt wird [117]. Innerhalb einer NativeScript-Applikation werden Module und Plugins als npm[1]-Pakete organisiert, die über das Kommandozeilen-Tool von NativeScript (NativeScript-CLI) verwaltet werden können. Ein NativeScript-Plugin besteht dabei aus folgenden Elementen [140]:

- eine `package.json`-Datei, die Metadaten, wie zum Beispiel den Namen, die Version, unterstützte Laufzeitumgebungen oder Abhängigkeiten von anderen npm-Paketen, enthält. Diese muss den npm-Spezifikationen genügen und es muss in dieser ein `nativescript`-Block definiert sein.
- ein oder mehrere CommonJS-Module, die eine API bereitstellen, die aus dem Applikationscode angesprochen werden kann. Wenn native APIs verwendet werden, wird über das CommonJS-Modul eine einheitliche JavaScript-API bereitgestellt, welche die nativen APIs abstrahiert.
- In einem `platforms`-Ordner können optional eine `AndroidManifest.xml` für Android und eine `Info.plist` für iOS definiert werden. Diese definieren benötigte Berechtigungen, Funktionen oder andere plattformspezifische Einstellungen für das Plugin. Für diese Dateien müssen `android`- bzw. `ios`-Unterordner für die jeweilige Plattform angelegt werden. Die für Plugins definierten Konfigurationen

[1] Der Node-Package-Manager (npm) [138] stellt einen Paketmanager für die JavaScript-Laufzeitumgebung Node.js [139] bereit.

werden von NativeScript beim Erstellungsprozess in einer einzelnen Konfigurationsdatei für die jeweilige Plattform zusammengeführt.

- Über eine optionale `include.gradle`-Konfigurationsdatei können im `android`-Ordner native Abhängigkeiten für Android definiert werden. Die entsprechenden nativen Bibliotheken können ebenfalls in diesem Ordner eingebunden werden. Analog dazu können optional über eine `build.xcconfig`-Konfigurationsdatei und einer `Podfile` im `ios`-Ordner native Abhängigkeiten für iOS definiert und Bibliotheken eingebunden werden.

Die beschriebenen Elemente können bei einem NativeScript-Plugin, das aus einem einzelnen CommonJS-Modul besteht, innerhalb folgender beispielhaften Ordnerstruktur organisiert werden:

```
custom-plugin/
|-- index.android.js
|-- index.ios.js
|-- package.json
|-- platforms/
    |-- android/
    |   |-- libs/
    |   |-- res/
    |   |-- AndroidManifest.xml
    |   |-- include.gradle
    |-- ios/
        |-- libs/
        |-- build.xcconfig
        |-- Info.plist
        |-- Podfile
```

Listing 7: Beispielhafte Ordnerstruktur eines NativeScript-Plugins

Die plattformspezifischen `index.*.js`-Dateien stellen das CommonJS-Modul und damit den Einstiegspunkt in das Plugin dar. Das Kommandozeilen-Tool von NativeScript kopiert beim Erstellungsprozess nur die für eine Plattform benötigten Dateien in die resultierende Applikation. Dieses Beispiel zeigt auch, dass Ressourcen für Android, wie

zum Beispiel Bilder oder XML-Dateien, über einen optionalen `res`-Ordner eingebunden werden können.

Über das Kommandozeilen-Tool können Plugins für eine NativeScript-Applikation installiert werden. Dabei können Plugins über veröffentlichte npm-Pakete, lokale Pfade, URLs oder über die Git-Versionsverwaltung [141] verwendet werden. Der entsprechende Befehl lautet `tns plugin add`.

Im Entwicklungsprozess eines Plugins müssen verschiedene Aspekte berücksichtigt werden. So muss ein Plugin, das in TypeScript geschrieben wurde, noch vor der Installation in JavaScript übersetzt werden, da dies von NativeScript während des Erstellungsprozesses einer Applikation nicht übernommen wird. Des Weiteren kann es notwendig sein, die von NativeScript generierten Plattformdateien zu aktualisieren, wenn ein Plugin installiert oder aktualisiert wurde. Eine Neugenerierung der plattformspezifischen Dateien einer Applikation kann über die Befehle `tns platform remove` und `tns platform add` erreicht werden.

4.5 Einschränkungen

Bei der Applikationsentwicklung mit NativeScript können Einschränkungen auftreten, die im Folgenden kurz angeschnitten werden. Dabei handelt es sich zum einen um Einschränkungen, die durch das Design und die Architektur von NativeScript bedingt und auch von Telerik dokumentiert sind. Zum anderen werden auch Einschränkungen vorgestellt, die im Rahmen dieser Arbeit beobachtet wurden und keine Erwähnung in der Dokumentation von NativeScript finden. Hierbei handelt es sich hauptsächlich um Einschränkungen, die sich auf die Verwendung nativer APIs beziehen.

Das von NativeScript verwendete Single-Threading-Modell ermöglicht einen direkten und schnellen Zugriff auf die nativen APIs der mobilen Plattformen. Die komplette NativeScript-Architektur läuft dabei im UI-Thread der mobilen Applikation. Das kann bei einer sehr komplexen Benutzeroberfläche mit vielen Animationen und dazu parallel laufenden Aufgaben zu Performanceproblemen oder einer blockierenden Benutzeroberfläche führen. Wie in Abschnitt 4.2 beschrieben, versucht NativeScript dieses Problem

mit sogenannten Worker-Skripten [132] anzugehen, mit denen rechenintensive oder blockierende Aufgaben in eigene Threads ausgelagert werden können. Dabei wird eine sehr einfache Schnittstelle zur Inter-Thread-Kommunikation angeboten, über die nur JSON-serialisierbare Objekte versendet werden können. Des Weiteren besteht keine Möglichkeit, gemeinsamen Speicher zwischen den Threads zu nutzen. Innerhalb eines Worker-Threads können auch keine weiteren Worker gestartet werden. Hierbei ist anzumerken, dass sich das Parallelisierungskonzept von NativeScript während dieser Arbeit noch in einem recht frühen Entwicklungsstadium befand.

Plattformspezifische Parallelisierungsansätze sind ebenfalls eingeschränkt. Auf Android werden beispielsweise `java.lang.Thread`- oder `android.os.AsyncTask`-Instanzen, die in JavaScript geschrieben wurden, immer im UI-Thread statt in eigenen Threads ausgeführt. Hierfür fehlt eine entsprechende Erwähnung in der NativeScript-Dokumentation. Des Weiteren werden von NativeScript keine Abstrahierungen zur Ausführung von Code im Hintergrund einer Applikation, wie z.B. in einem nativen Hintergrunddienst, angeboten. Diese Thematik wird in Abschnitt 7.4 im Rahmen der Entwicklung eines eigenen Plugins behandelt.

Telerik verspricht, dass 100% der nativen APIs mit NativeScript angesprochen werden können [126]. Jedoch ist die Verwendung der nativen APIs an diversen Stellen limitiert. Neben den bereits angesprochenen Einschränkungen hinsichtlich der Parallelisierung werden bei iOS diverse Datentypen, wie zum Beispiel Unions oder Vektoren nicht unterstützt [142]. Eine weitere Limitierung trat im Rahmen dieser Arbeit bei der Erweiterung nativer Klassen nach dem Vererbungskonzept auf Android auf. Es besteht keine Möglichkeit, einen eigenen Konstruktor für native Java-Klassen zu definieren. Diese Einschränkung zeigt dann Auswirkungen, wenn eine Klasse nicht aus dem Applikationscode, sondern vom Betriebssystem instanziiert wird. Das ist zum Beispiel der Fall, wenn Android eine `android.app.IntentService`-Instanz über einen Standardkonstruktor erstellen will. Da eine JavaScript-Implementierung einer solchen Klasse keinen Standardkonstruktor definieren kann, führt das zum Absturz einer Applikation. In Unterabschnitt 7.2.3 wird diese Problematik genauer betrachtet und ein Lösungsansatz vorgestellt.

5

Kontextsensitivität

Seit Mark Weiser [143] das Konzept des Ubiquitous-Computing in den 90er-Jahren des letzten Jahrhunderts eingeführt hat, erfahren Kontextsensitivität und kontextsensitive Systeme in der Literatur eine hohe Aufmerksamkeit. Dieses Kapitel gibt einen Einblick in das Thema Kontextsensitivität und betrachtet Grundlagen und Ansätze aus der Literatur. Es wird zunächst die Terminologie erläutert und anschließend auf wichtige Aspekte kontextsensitiver Systeme eingegangen. Hierzu zählen die Kontexterfassung, die Modellierung von Kontext, Kontextverarbeitung, Schlussfolgerungsmechanismen und die Architektur kontextsensitiver Systeme.

5.1 Kontext

Ubiquitous-Computing beschreibt ein Paradigma, bei der die physikalische Welt auf unsichtbare Art und Weise mit Software- und Hardwarekomponenten verflochten ist und eingebettete Systeme sich nahtlos und vernetzt in das alltägliche Leben integrieren bis sie nicht mehr von diesem unterschieden werden können [143, 144]. Systeme, die dem Ubiquitous-Computing-Paradigma folgen, zeichnen sich dadurch aus, dass sie ihr Verhalten an den Kontext anpassen. Nach Dey et al. kann Kontext dabei folgendermaßen definiert werden: "Kontext ist jede Information, die genutzt werden kann, um die Situation einer Entität zu beschreiben. Entitäten sind Personen, Orte oder Objekte, welche für die Interaktion zwischen einem Nutzer und einer Applikation als relevant betrachtet werden, inklusive dem Nutzer und der Applikation selbst" [5]. Den folgenden Ausführungen dient diese Definition als Grundlage. Heutzutage sind viele Geräte mit einer Vielzahl von Sensoren ausgestattet und besitzen Möglichkeiten zur Netzwerkkommunikation. Hierzu

zählen besonders mobile Endgeräte wie Smartphones, Tablets oder Wearables, Geräte aus dem Bereich Internet der Dinge, Fahrzeuge oder Smart-Home-Systeme. Solche Geräte können von kontextsensitiven Systemen genutzt werden, um Kontextinformationen zu erfassen. Damit kann zum Beispiel der Kontext eines Nutzers oder eines Gerätes bestimmt werden und das Verhalten des Systems dementsprechend angepasst werden. Im Folgenden werden grundlegende Mechanismen solcher kontextsensitiver Systeme genauer betrachtet.

5.2 Kontextsensitive Systeme

Werkzeuge oder Frameworks, welche die Entwicklung kontextsensitiver Applikationen ermöglichen, müssen verschiedene Aufgaben erfüllen. Dazu gehören nach Knappmeyer et al. [37] die Erfassung von relevanten Daten, die Schlussfolgerung auf High-Level-Kontextinformationen, die Speicherung von Kontextinformationen sowie die Verwaltung und Koordination dieser zwischen verschiedenen Systemkomponenten. Systeme, die diese Aufgaben erfüllen, sind dabei oft als geschichtete Middleware konzeptioniert. Diese Schichtenarchitektur wird in Abbildung 5.1 illustriert. Vorteil einer solchen Architektur ist die Abkapselung der verschiedenen Aufgabenbereiche, die unterschiedlich komplex ausfallen können. Die einzelnen Schichten müssen dabei verschiedenste Teilaufgaben übernehmen. Knappmeyer et al. [37] identifizieren hierbei folgende Aufgabenbereiche für kontextsensitive Systeme:

- **Wahrnehmung** / **Erfassung:** Die Wahrnehmung bzw. Erfassung von relevanten Daten ist die Grundlage, um auf den Kontext zu schließen.
- **Schlussfolgerung** / **Inferenz:** Dieser Aufgabenbereich behandelt die Ableitung von bedeutsameren Information aus rohen Daten oder Low-Level-Kontextinformationen.
- **Lernen:** Hierzu zählt das Lernen anhand von vergangenen, gespeicherten Kontextinformationen, Aktionen oder Trainingsdaten.

- **Kontextrepräsentation:** Die Repräsentation und Modellierung von Kontextinformationen in einer maschinenlesbaren Art und Weise wird als Kontextrepräsentation bezeichnet.
- **Management und Verbreitung:** Hierzu gehören die Verwaltung von Kontextinformationen und die Bereitstellung dieser für andere Systemkomponenten. Die Beschaffung von Kontextinformationen von anderen Komponenten ist hier ebenfalls von Bedeutung.
- **Aktivierung:** Auf Basis der verfügbaren Kontextinformationen wird die Ausführung eines Dienstes der Applikation ausgelöst oder angepasst.

Abbildung 5.1: Schichtenarchitektur kontextsensitiver Middlewares nach Knappmeyer et al. [37]

Nach Dey [145] kann bei der kontextbasierten Anpassung des Verhaltens einer Applikation zwischen drei Arten unterschieden werden. Die erste Art beschreibt, dass Informationen und Dienste einem Nutzer präsentiert werden. Beispielsweise werden Informationen anhand des erkannten Kontexts gefiltert oder ausgewählte Dienste vorgeschlagen. Die automatische Ausführung oder Anpassung eines Dienstes beschreibt die

zweite Art. Dabei werden zum Beispiel die Logik oder das Aktivierungsverhalten von Diensten in Abhängigkeit vom Kontext eines Nutzers angepasst. Bei der dritten Art wird ein Kontext mit einer Information oder einem Inhalt verknüpft, um später Informationen zu einer Situation bereitstellen zu können. Knappmeyer et al. [37] erweitern diese Definition, indem die Komplexität des Kontexts und der Grad der Kontextbewusstheit einbezogen werden. Es wird zwischen kontextbasierter, kontextsensitiver und situationsbewusster Verhaltensanpassung differenziert. Die kontextbasierte Anpassung bezieht sich auf Applikationen, die Kontextinformationen bei Bedarf synchron abrufen und anhand dieser ihr Verhalten anpassen. Kontextsensitive Applikationen benötigen eine ereignisbasierte und asynchrone Kontextbereitstellung und reagieren auf gewisse Kontextereignisse. Situationsbewusste Applikationen passen ihr Verhalten aufgrund eines komplexen Kontexts an. Hierfür müssen primitive Kontextdaten zu High-Level-Kontextinformationen aggregiert werden.

Der Zyklus eines kontextsensitiven Systems kann nach Knappmeyer et al. wie in Abbildung 5.2 dargestellt werden.

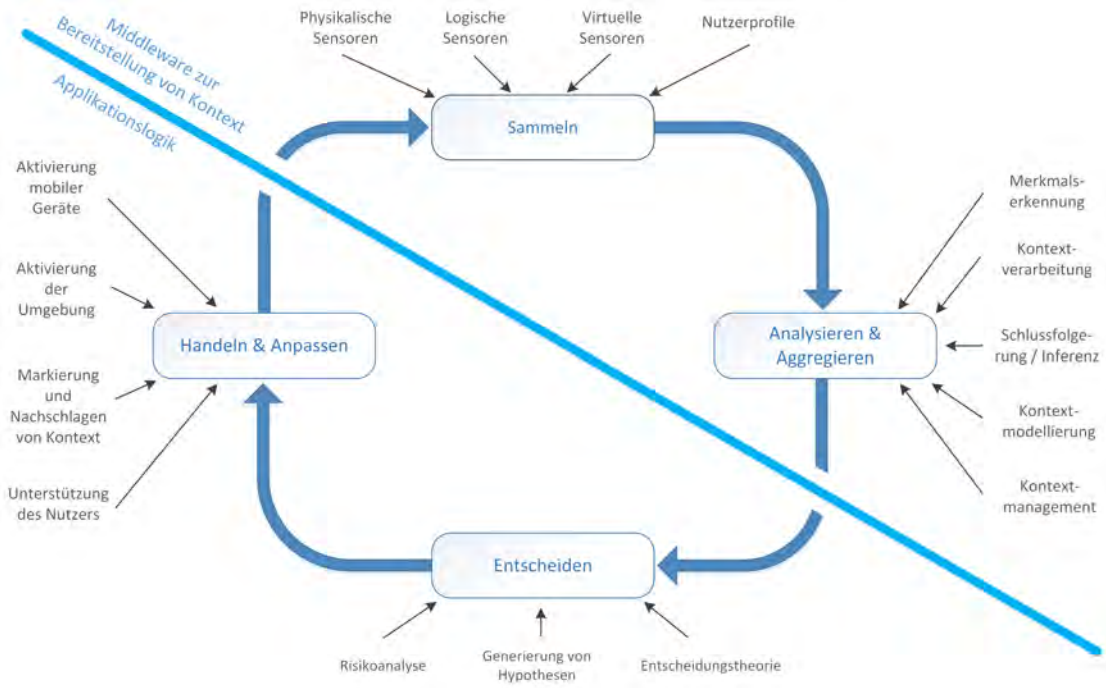

Abbildung 5.2: Zyklus kontextsensitiver Systeme nach Knappmeyer et al. [37]

5.3 Kontexterfassung und Kontextinformationen

Die Kontexterfassung beschreibt den Vorgang, bei dem Daten erfasst werden, mit denen der Kontext einer Entität beschrieben werden kann. Nach Knappmeyer et al. [37] können solche Kontextinformationen auf verschiedene Arten gesammelt werden. Es kann zwischen physikalischen Sensoren, virtuellen Sensoren, logischen Sensoren und Nutzerprofilen unterschieden werden. Physikalische Sensoren sind meistens, aber nicht zwangsweise, Hardwaresensoren die Werte der physikalischen Welt in entsprechenden physikalischen Skalen zur Verfügung stellen. Hierzu gehören unter anderem Beschleunigungssensoren, Luftdrucksensoren oder Lichtsensoren. Virtuelle Sensoren stellen Kontextinformationen aus der digitalen Welt zur Verfügung, beispielsweise Kalenderdaten, Kontakte oder Daten von Webdiensten. Logische Sensoren sind abstrakte Sensoren, die High-Level-Kontextinformationen bereitstellen, indem Daten von mehreren Sensoren und Quellen aggregiert werden. Solche Sensoren können zum Beispiel Kontextinformationen über die Stimmung eines Nutzers bereitstellen. Nutzerprofile beinhalten verschiedene Informationen über den Nutzer, wie zum Beispiel Einstellungen, Interessen oder Gewohnheiten. Solche Profile können als eine spezielle Art virtueller Sensoren verstanden werden.

Kontextinformationen können nach Knappmeyer et al. [37] in verschiedene Kategorien eingeteilt werden, indem unterschieden wird, welche Art von Kontext diese beschreiben. Des Weiteren differenzieren Oliveira et al. [31] zwischen statischen Kontextinformationen und dynamischen Kontextinformationen, die sich während der Laufzeit eines Systems verändern. Im Folgenden werden diese Kategorisierungen kombiniert dargestellt und beispielhafte Kontextinformationen aufgeführt:

- **Räumlicher Kontext (dynamisch):** Der räumliche Kontext stellt Informationen zum Aufenthaltsort einer Entität bereit. Dies kann zum Beispiel über geographische Koordinaten, die relative räumliche Nähe zu Orten sowie zu anderen Entitäten oder über die Art des Aufenthaltsorts geschehen.
- **Temporaler Kontext (dynamisch):** Der temporale Kontext besteht aus Informationen, mit denen die absolute Zeit, relative Zeit oder Tageszeit beschrieben wird. Des

Weiteren können diese Informationen symbolisch, wie zum Beispiel als Arbeitszeit oder Urlaub, beschrieben sein.

- **Gerätekontext (statisch/dynamisch):** Der Gerätekontext beinhaltet Informationen über Interaktionsgeräte des Nutzers. Hierzu zählen zum einen statische Eigenschaften wie verfügbare Eingabegeräte, Anzeigemöglichkeiten oder unterstütze Videoformate. Zum anderen umfasst der Gerätekontext auch dynamische Informationen wie die verbleibende Akkulaufzeit [146] oder den verfügbaren Arbeitsspeicher.
- **Netzwerk- und Kommunikationskontext (dynamisch):** Diese Art des Kontexts beschreibt verfügbare Kommunikationsmöglichkeiten über das Netzwerk. Dies beinhaltet Informationen, wie zum Beispiel verfügbare Wi-Fi-Access-Points in der Nähe, verfügbare Bandbreite, Verzögerung oder Übertragungskosten.
- **Umgebungskontext (dynamisch):** Der Umgebungskontext beschreibt die physikalische Umgebung einer Entität. Hierzu gehören beispielsweise die Umgebungslautstärke, der Luftdruck, die Lichtintensität oder die Luftverschmutzung.
- **Individualisierung und Nutzerprofile (statisch):** Diese Kategorie beinhaltet individuelle Einstellungen sowie Interessen oder Gewohnheiten des Nutzers. Mit diesen Informationen können Nutzer zum Beispiel eindeutig innerhalb des Systems identifiziert werden.
- **Aktivitäten (dynamisch):** Über diese Art des Kontexts kann beschrieben werden, innerhalb welcher Aktivitäten oder Aufgaben eine Entität sich gerade befindet. Hierzu zählt auch die Intention einer Entität, die geplante Aktivitäten beschreibt.
- **Mentaler Kontext (dynamisch):** Über den mentalen Kontext können Geisteszustände des Nutzers beschrieben werden. Dabei handelt es sich um Emotionen oder Stimmungen des Nutzers, die zum Beispiel über ein Stress-Level bestimmt werden können.
- **Interaktionskontext (dynamisch):** Diese Art der Kontextinformation beschreibt sowohl die soziale Interaktion zwischen Nutzern als auch die Interaktion zwischen Nutzern und Teilen des Systems.

Die beschriebenen Arten von Kontextinformationen hängen teilweise direkt oder indirekt voneinander ab [37]. Beispielsweise hängt der temporale Kontext vom räumlichen Kontext über die Zeitzone ab, in der sich eine Entität befindet. Ein weiteres Beispiel dafür ist die Abhängigkeit des Netzwerk- und Kommunikationskontexts vom räumlichen Kontext. Aufgrund dieser Abhängigkeiten ist es teilweise möglich, über die Bestimmung einer Kontextart auf eine andere Kontextart zu schließen. So kann man beispielsweise mit der Annahme arbeiten, dass ein Nutzer, der mit einem Wireless-Access-Point bei sich zu Hause kommuniziert, mit großer Wahrscheinlichkeit gerade nicht Auto fährt. Dabei wird vom Netzwerk- und Kommunikationskontext auf die Aktivität einer Entität geschlossen.

5.4 Kontextmodellierung

In kontextsensitiven Applikationen müssen Kontextinformationen in einer maschinenlesbaren Art und Weise dargestellt werden, damit das System auf den Kontext zugreifen und sein Verhalten dementsprechend anpassen kann. Hierfür können diverse Kontextmodelle zum Einsatz kommen. Diese müssen in Abhängigkeit vom Design und der Architektur eines kontextsensitiven Systems gestaltet werden. Hierbei spielen zum Beispiel die Art des verwendeten Schlussfolgerungsmechanismus oder die Speicherung von Kontextinformationen eine Rolle. Allgemein können nach Perttunen et al. [147] folgende Anforderungen an Kontextrepräsentationen gestellt werden.

- **Eindeutige Identifikation:** Verschiedene Kontexte bzw. Kontextinformationen und dazugehörige Entitäten müssen innerhalb eines Systems eindeutig identifizierbar sein. So können Informationen ohne Konflikte wiederverwendet werden. Dies gilt insbesondere für verteilte kontextsensitive Systeme.
- **Validierung:** Kontextinformationen sollten hinsichtlich ihrer Struktur und Instanziierung überprüfbar sein [30]. So kann sichergestellt werden, dass kontextsensitive Systeme zur weiteren Verarbeitung nur valide und konsistente Informationen geliefert bekommen. Dies spielt insbesondere in komplexen Systemen eine wichtige Rolle, in denen Kontextinformationen in wechselseitiger Beziehungen zueinander stehen.

- **Aussagekraft:** Eine aussagekräftige Kontextrepräsentation ermöglicht die Modellierung komplexer Entitäten und Beziehungen. Dies steht nach Brachman et al. [148] jedoch in gegenseitigem Konflikt mit der Korrektheit, Vollständigkeit und Effizienz von Schlussfolgerungsmechanismen.

- **Unsicherheit und unvollständige Informationen:** Die Kontextrepräsentation sollte es ermöglichen, die Qualität von Kontextinformationen zu kodieren. Dies ist notwendig, da Kontextinformationen oft über Messungen mit unpräzisen Sensoren gewonnen werden. Des Weiteren sollte ein kontextsensitives System mit unvollständigen Kontextinformationen umgehen können.

- **Einfachheit, Wiederverwendbarkeit und Erweiterbarkeit:** Im Gegensatz zur Anforderung der Aussagekraft, soll eine Kontextrepräsentation gleichzeitig auch nur so aussagekräftig wie nötig sein. Mit einer einfachen, nicht zu komplexen Kontextmodellierung steigt die Wiederverwendbarkeit und Erweiterbarkeit der Repräsentation.

- **Allgemeingültigkeit:** Eine Kontextrepräsentation sollte jegliche Art von Kontextinformation darstellen können [149]. Mit einer allgemeingültigen Kontextrepräsentation können auch alle möglichen Arten von kontextsensitiven Applikationen realisiert werden.

Kontextinformationen können mit verschiedenen Kontextmodellen bzw. Datenstrukturen repräsentiert werden. Strang et al. [30] unterscheiden zwischen sechs unterschiedlichen Modellen:

- **Key-Value-Modelle:** Bei Key-Value-Modellen werden Kontextinformationen einfach über einen Namen und einen Wert dargestellt. Eine hierarchische Strukturierung ist dabei nicht möglich.

- **Markup-Schema-Modelle:** Mit Markup-Schema-Modellen kann Kontext hierarchisch über Elemente und Attribute in einer Markup-Sprache wie zum Beispiel XML definiert werden. Bei der Verwendung von Markup-Sprachen ist eine Validierung über ein festgelegtes Schema möglich.

- **Graphische Modelle:** Kontextinformationen können auch mit graphischen Modellen, wie zum Beispiel über die Unified-Modeling-Language (UML), repräsentiert werden. In diesem Fall werden Kontextinformationen in einem Graph mit Knoten und Kanten definiert.
- **Objektorientierte Modelle:** In objektorientierten Modellen werden Kontextinformationen über Klassen und Objekte modelliert. Hierbei besteht die Möglichkeit, Konzepte aus der objektorientierten Programmierung, wie zum Beispiel Kapselung, Vererbung, Wiederverwendbarkeit oder Überschreiben, zu verwenden.
- **Logikbasierte Modelle:** Bei logikbasierten Modellen kann Kontext über Fakten, Ausdrücke und Regeln definiert werden. Dabei kann ein kontextsensitives System mit Ereignissen, Bedingungen und Aktionen arbeiten. Des Weiteren ist es möglich, neue Fakten anhand vorhandener Fakten abzuleiten oder unnötige bzw. ungültige Fakten und Regeln zu entfernen.
- **Ontologiebasierte Modelle:** Der Kontext wird hier als eine Ontologie modelliert, indem Begriffe, Taxonomien und logische Relationen den Kontext beschreiben. Wenn zum Beispiel Sprachen wie die Web-Ontology-Language (OWL) genutzt werden, können Schlussfolgerungsmechanismen aus den Beschreibungslogiken eingesetzt werden.

Im Rahmen dieser Arbeit wurde ein logikbasiertes Modell zur Bereitstellung und Modellierung von Kontext verwendet. Eine logikbasierte Modellierung ermöglicht eine hierarchische Strukturierung von Kontext und erfordert nicht den Einsatz zusätzlicher Technologien, wie das teilweise bei anderen Modellen der Fall ist. Hierauf wird in Kapitel 6 und Kapitel 7 näher eingegangen.

5.5 Kontextverarbeitung und Schlussfolgerungsmechanismen

Gemessene oder gesammelte Daten müssen aufbereitet und als Kontextinformationen bereitgestellt werden, damit ein kontextsensitives System sein Verhalten anpassen oder

kontextrelevante Informationen anzeigen kann. Zur Repräsentation von Kontextinformationen können die im vorherigen Abschnitt beschriebenen Modelle verwendet werden. Um aussagekräftige Kontextinformationen bereitstellen zu können, müssen gemessene Daten häufig noch weiter verarbeitet bzw. aufbereitet werden. Das Ziel ist, aus einer Menge von Low-Level-Kontextdaten auf Kontextinformationen einer höheren Semantik zu schließen. Diese Kontextverarbeitung kann nach Knappmeyer et al. [37] allgemein in mehrere Schritte unterteilt werden. Wie Abbildung 5.3 illustriert, können dabei aus rohen Sensordaten High-Level-Kontextinformationen gewonnen werden.

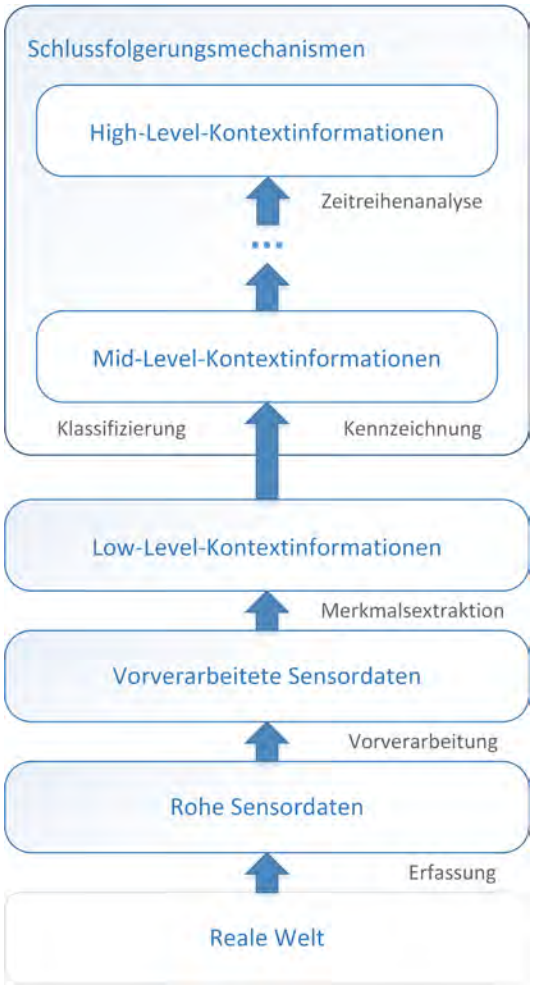

Abbildung 5.3: Schritte der Kontextverarbeitung (vgl. Knappmeyer et al. [37])

Die reale Welt kann aus dieser Sicht über Sensoren gemessen bzw. erfasst werden. Dabei werden analoge elektrische Signale gemessen, die über den Prozess der Abtastung in diskrete Werte überführt werden. Die erfassten rohen Sensordaten können in einem Vorverarbeitungsschritt aufbereitet werden. Hierfür werden zum Beispiel Filtertechniken eingesetzt. Aus den vorverarbeiteten Sensordaten können Low-Level-Kontextinformationen generiert werden. Hier werden zum Beispiel einfache mathematische Operatoren, Fuzzylogik oder statistische Verfahren angewandt, um Merkmale aus den Daten zu extrahieren. Über solche Merkmale können dann einfache Kontextinformationen beschrieben werden. Es ist auch möglich, dass solche einfachen Kontextinformationen direkt von virtuellen oder logischen Sensoren bezogen werden. Im nächsten Schritt werden Low-Level-Kontextinformationen weiter verarbeitet, um eine höhere semantische Abstraktion zu erreichen. Beispielsweise kann mit Hilfe von Klassifikationsverfahren oder Mustererkennung auf Aktivitäten oder Situationen geschlossen werden, was in dieser Darstellung zu den Mid-Level-Kontextinformationen führt. Der Einsatz von Schlussfolgerungsmechanismen ermöglicht auch die Generierung von High-Level-Kontextinformationen, mit denen zum Beispiel die Intention eines Nutzers eingeschätzt werden kann.

Diese Illustration der Kontextverarbeitung stellt insbesondere dar, wie Kontextinformationen einer niedrigeren Ebene über Schlussfolgerungsmechanismen oder Mustererkennungsverfahren in aussagekräftigere Informationen einer höheren Ebene transformiert werden können. Für diese Abstraktion können Verfahren des maschinellen Lernens, wie zum Beispiel Clustering-Algorithmen, Klassifikationsverfahren, Regressionsanalysen oder andere stochastische Modelle, eingesetzt werden. Solche Verfahren kommen besonders dann zum Einsatz, wenn notwendiges a-priori-Wissen fehlt, um einen Kontext zu beschreiben. Dies ist zum Beispiel der Fall, wenn komplexe Bewegungen über die Daten eines Beschleunigungssensors bestimmt werden sollen.

Häufig können Sitationen oder Kontextinformationen jedoch einfach mit a-priori-Wissen bestimmt werden. Zum Beispiel kann mit dem Wissen, dass die Werte eines Lichtsensors niedrig sind, wenn es dunkel ist, sehr einfach über vordefinierte Bedingungen und aktuelle Sensordaten auf eine dunkle Umgebung geschlossen werden. Dieses Prinzip kann für viele Low-Level- und Mid-Level-Kontextinformationen eingesetzt werden, bei denen eine

Kontextbestimmung per Design im Voraus möglich ist. Eine ähnliche Vorgehensweise kann auch bei regelbasierten Schlussfolgerungsmechanismen eingesetzt werden. Bei diesen werden im Voraus definierte Regeln verwendet, um höhere Kontextinformationen abzuleiten. Vorhandene Kontextinformationen werden bei diesem Verfahren einer Wissensbasis zugewiesen, während eine Regelbasis vordefinierte Regeln enthält [37]. Wenn eine Regel unter Berücksichtigung der Informationen der Wissensbasis zutrifft, wird ein Kontextereignis ausgelöst, auf das eine Applikation reagieren kann. Für solche Systeme eignet sich besonders eine logikbasierte Modellierung der Kontextinformationen, um Ungenauigkeiten zu vermeiden.

Neben der logikbasierten Bereitstellung von Kontextinformationen wird im Rahmen dieser Arbeit ein regelbasierter Inferenzmechanismus verwendet. Dieser wird in Kapitel 6 und Kapitel 7 vorgestellt und näher betrachtet.

6

Konzeption eines Cross-Platform-Frameworks für Kontextsensitivität

Frameworks für kontextsensitive Systeme auf mobilen Endgeräten sind häufig nur für eine spezielle Plattform entwickelt oder sind für den Einsatz mit mobilen Webapplikationen konzipiert worden (vgl. Kapitel 2). Mobile Webanwendungen sind im Vergleich zu nativen Applikationen hinsichtlich der Performance und dem Zugriff auf native Funktionalitäten eingeschränkt (vgl. Kapitel 3). Cross-Platform-Ansätze ermöglichen im Gegensatz dazu, die Verwendung einer nativen Benutzeroberfläche und den Zugriff auf native Schnittstellen des mobilen Betriebssystems. Dabei können die plattformübergreifenden mobilen Applikationen ähnlich zu mobilen Webanwendungen mit Webtechnologien und mit daran angelehnten Technologien entwickelt werden. Diese Eigenschaften werden besonders vom NativeScript-Framework vereint, mit dem über eine einzelne Codebasis für mehrere mobile Plattformen entwickelt werden kann (vgl. Kapitel 4).

Aus diesen Gründen wurde im Rahmen dieser Arbeit ein Framework konzeptioniert, dass Kontextsensitivität für plattformübergreifende NativeScript-Applikationen bereitstellt. Das hier vorgestellte Framework verfolgt einen gerätezentrierten Ansatz, bei dem die komplette Kontextverarbeitung auf dem mobilen Endgerät innerhalb der Applikation stattfindet. Dadurch sind Applikationsentwickler nicht auf Netzwerkkonnektivität angewiesen, wie das zum Beispiel bei cloudbasierter Kontextverarbeitung der Fall ist. Solange die Kontextdaten nicht über das Netzwerk versendet werden, müssen Entwickler auch keine Aspekte der Netzwerksicherheit oder des Datenschutzes berücksichtigen. Des Weiteren

ist das Framework im Rahmen des NativeScript-Ökosystems als eine Menge von Plugins entworfen worden. Dies hat gegenüber einer Middleware den Vorteil, dass Entwickler das Framework sehr leicht einsetzen und auch erweitern können, da die verwendeten Technologien aus der NativeScript-Entwicklung bekannt sind.

Dieses Kapitel beschreibt die Anforderungen, die an das Framework gestellt wurden. Anschließend wird die Konzeption und Architektur des Frameworks vorgestellt. Hierfür werden die einzelnen Komponenten des Frameworks und deren Zusammenspiel betrachtet.

6.1 Anforderungsanalyse

Dieser Abschnitt behandelt Anforderungen, die an das Framework für Kontextsensitivität gestellt wurden. Dabei werden zunächst die funktionalen Anforderungen und anschließend die nicht-funktionalen Anforderungen angegeben.

6.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben, welche Funktionalitäten das Framework unterstützen soll und welche Funktionen für Applikationsentwickler bereitgestellt werden sollen. Zudem wird in die funktionalen Anforderungen eine Beispielapplikation eingeschlossen, die zum Testen des Frameworks und für Demonstrationszwecke gedacht ist.

FA-01 Logikbasierte Bereitstellung von Kontextinformationen: Das Framework soll Applikationsentwicklern eine einfache Möglichkeit bieten, Kontextinformationen über logische Ausdrücke abfragen zu können. Dafür sollen Schnittstellen symbolische Methoden bereitstellen und Details der Kontexterfassung und Kontextvorverarbeitung abstrahieren.

FA-02 Definition logikbasierter Kontextbedingungen: Applikationsentwickler sollen die Möglichkeit haben, Kontextbedingungen bzw. Situationen über logische Ausdrücke beschreiben zu können. Kontextmerkmale bzw. Kontextinformationen, die

als solche logischen Ausdrücke zur Verfügung gestellt werden (vgl. **FA-01**), sollen mit booleschen Operatoren beliebig verknüpft werden können. Das Framework soll Methoden anbieten, mit denen die logischen Operatoren UND, ODER und NICHT bereitgestellt werden.

FA-03 Definition von Kontextereignissen: Unter Verwendung von logikbasierten Kontextbedingungen (vgl. **FA-01** und **FA-02**) sollen Kontextereignisse definiert werden können. Solche Kontextereignisse bestehen aus einem logischen Ausdruck zur Definition der Kontextbedingungen und aus einer Aktion. Eine Aktion beinhaltet beliebigen JavaScript-Code, der ausgeführt wird, sobald die dafür definierten Kontextbedingungen erfüllt sind. Solche Ereignisse sollen beim Framework registriert werden können.

FA-04 Modularisierte Kontexterfassung, Kontextvorverarbeitung und Bereitstellung von Kontextinformationen: Die logikbasierte Bereitstellung von Kontextinformationen (vgl. **FA-01**) soll modularisiert erfolgen. Dadurch kann die Erfassung, Vorverarbeitung und Bereitstellung von speziellen Kontextinformationen in einzelne Module gekapselt werden. Hierfür müssen Konzepte, Schnittstellen und Strukturen angeboten werden, mit denen solche Module implementiert werden können. Diese Anforderung zielt darauf ab, eine klare Trennung der Aufgabenbereiche zu ermöglichen. Solche Module werden, angelehnt an Ranganathan et al. [150], im weiteren Verlauf dieser Arbeit als Kontext-Provider bezeichnet.

FA-05 Zentrale Evaluation und Auslösung von Kontextereignissen: Das Framework soll eine zentrale Komponente bereitstellen. Diese soll automatisiert Kontextbedingungen von Kontextereignissen evaluieren und die entsprechenden Aktionen ausführen, falls die Kontextbedingungen für ein Ereignis erfüllt sind. Dabei soll die Evaluation nicht periodisch, sondern immer dann erfolgen, wenn neue Kontextinformationen zur Verfügung stehen. Die zentrale Komponente muss Schnittstellen anbieten, mit denen Kontext-Provider bei dieser angemeldet werden können. Zudem muss eine bidirektionale Kommunikation zwischen der zentralen Komponente und Kontext-Providern möglich sein. Auf diese Weise kann die zentrale Komponente die Kontexterfassung- und -verarbeitung registrierter Provider steuern. Diese

können ihrerseits proaktiv die zentrale Komponente benachrichtigen, wenn neue Kontextinformationen zur Verfügung stehen. Des Weiteren soll über die zentrale Komponente die Kontextverarbeitung allgemein gestartet und gestoppt werden können.

FA-06 Zeitliche Validität von Kontextinformationen: Es soll eine generische Möglichkeit angeboten werden, beliebige Kontextinformationen mit einer Gültigkeitsdauer zu versehen. Dies soll unabhängig von der verwendeten Datenstruktur möglich sein. Applikationsentwickler sollen die zeitliche Validität von Kontextinformationen definieren und über einfache boolesche Methoden abfragen können.

FA-07 Persistenz von Kontextinformationen: Das Framework soll für die Implementierung von Kontext-Providern eine generische Möglichkeit zur Verfügung stellen, beliebige Kontextinformationen zu speichern. Dadurch soll ermöglicht werden, dass vergangene Informationen zur Bestimmung des aktuellen Kontexts berücksichtigt werden können. Diese sollen dabei mit einem Zeitstempel versehen und in einer SQLite-Datenbank auf den mobilen Endgeräten gespeichert werden. Dieser sogenannte Kontextverlauf (englisch: context history) soll mit beliebigen Datenstrukturen umgehen können. Des Weiteren soll eine Gültigkeitsdauer für Kontextinformationen innerhalb eines Kontextverlaufs definierbar sein. Dieser soll dann abgelaufene Kontextinformationen selbständig aus der Datenbank entfernen.

FA-08 Kommunikation zwischen Kontext-Providern: Die Implementierungen einzelner Kontext-Provider sollen die Möglichkeit haben, die selben Schnittstellen zu nutzen, die auch Applikationsentwicklern zur Verfügung gestellt werden. Dadurch können Kontext-Provider auf Kontextinformationen anderer Kontext-Provider zugreifen und somit auch selbst auf Kontextereignisse reagieren.

FA-09 Kontext-Provider - Ort: Es soll ein Kontext-Provider implementiert werden, der Kontextinformationen über den Ort eines mobilen Gerätes bereitstellt. Hierfür sollen die GPS-Sensoren der mobilen Endgeräte verwendet werden. Insbesondere sollen Kontextinformationen hinsichtlich der Nähe zu einem bestimmten Ort abgefragt werden können. Über einen Kontextverlauf soll es möglich sein, zu bestimmen, ob sich ein mobiles Gerät für eine gewisse Zeitspanne in der Nähe eines

Ortes befunden hat. Zudem soll der Kontext-Provider über die GPS-Positionen Kontextinformationen zur Geschwindigkeit bereitstellen. Die Bereitstellung dieser Informationen soll plattformunabhängig und abstrahiert über boolesche Ausdrücke erfolgen.

FA-10 Kontext-Provider - Aktivitäten: Die Android- und iOS-Plattformen bieten native APIs an, um die Aktivität eines Nutzers zu bestimmen. Diese APIs sollen plattformunabhängig abstrahiert in einem Kontext-Provider eingesetzt werden. Mit diesem Kontext-Provider sollen über boolesche Ausdrücke Kontextinformationen bezüglich der aktuellen Aktivität eines Nutzers abgefragt werden können. Mit einem Kontextverlauf soll zudem bestimmt werden können, wie lange der Nutzer des mobilen Endgeräts eine Aktivität bereits ausübt. Des Weiteren soll das Modul mit Genauigkeitswerten umgehen können. Diese werden von den nativen APIs für erkannte Aktivitäten bereitgestellt.

FA-11 Kontext-Provider - Herzfrequenz: Es soll ein Kontext-Provider implementiert werden, der die Herzfrequenz des Nutzers als Kontextinformation bereitstellt. Die Bestimmung der Herzfrequenz soll über standardisierte Bluetooh-Low-Energy-Messgeräte erfolgen. Es sollen boolesche Ausdrücke über symbolische Methoden abgefragt werden können, die zum Beispiel aussagen, ob die Herzfrequenz des Nutzers größer oder kleiner gleich einem definierten Schwellenwert ist. Mit einem Kontextverlauf soll die durchschnittliche Herzfrequenz über einen konfigurierbaren Zeitraum bestimmt werden können. Hierfür sollen ebenfalls entsprechende boolesche Ausdrücke zur Verfügung stehen. Das Modul soll zudem Möglichkeiten zur Verwaltung von Herzfrequenzmessgeräten bereitstellen. Es sollen Schnittstellen angeboten werden, über die Bluetooth-Low-Energy-Geräte gesucht, gespeichert und verwaltet werden können. Dabei kann ein Herzfrequenzmesser als Standardgerät festgelegt werden. Zu diesem Standardgerät soll der Kontext-Provider periodisch versuchen, automatisch eine Verbindung herzustellen. Dadurch soll wiederholte Benutzerinteraktion zur Verbindungsherstellung vermieden werden.

FA-12 Kontext-Provider - Termine: Mit Hilfe von nativen APIs soll auf die Kalendereinträge eines Nutzers zugegriffen werden, um Kontextinformationen hinsichtlich

aktueller oder bevorstehender Termine bereitzustellen. Der Zugriff auf die nativen APIs soll dabei plattformunabhängig abstrahiert werden. Über boolesche Ausdrücke soll abgefragt werden können, ob gerade oder in naher Zukunft ein wichtiger Termin stattfindet.

FA-13 Kontext-Provider - Umgebungslautstärke: Über einen weiteren Kontext-Provider soll die Umgebungslautstärke als Kontextinformation bereitgestellt werden. Diese kann über das Mikrophon von mobilen Endgeräten bestimmt werden. Um die Lautstärke relativ zu einer ruhigen Umgebung bestimmen zu können, muss der Kontext-Provider eine Schnittstelle zur Kalibrierung anbieten. Die Umgebungslautstärke kann dann sowohl relativ zur Messskala des Mikrophons als auch relativ zu einer ruhigen Umgebung angegeben werden. Mit Hilfe eines Kontextverlaufs soll die durchschnittliche Umgebungslautstärke über eine konfigurierbare Zeitspanne bestimmt werden können. Die Bereitstellung der Informationen soll wie bei den vorangegangen Kontext-Providern erfolgen. Hierfür werden logische Ausdrücke, die über symbolische Methoden angesprochen werden können, zur Verfügung gestellt.

FA-14 Bereitstellung von rohen Kontextinformationen: Alle angesprochenen Kontext-Provider sollen eine asynchrone Schnittstelle anbieten, um rohe Kontextinformationen proaktiv anderen Systemkomponenten bereitzustellen. Dies kann Applikationsentwicklern helfen, die Funktionsweise des Frameworks im Rahmen einer Anwendung besser zu testen. Sobald neue Kontextinformationen bereitstehen, sollen Kontext-Provider diese Informationen registrierten Beobachtern mitteilen. Mit diesem Mechanismus können die Daten zum Beispiel vom Applikationsentwickler protokolliert werden.

FA-15 Hintergrundausführung: Das Framework soll im Hintergrund ausgeführt werden können, selbst wenn die dazugehörige Applikation minimiert ist. NativeScript bietet keine plattformunabhängige Möglichkeit an, Code im Hintergrund auszuführen. Die nativen Plattformen bieten jedoch Konzepte und Schnittstellen an, um dies zu realisieren. Auf Android ist es möglich, verschiedene Arten von Hintergrunddiensten zu implementieren, die unabhängig von der Applikation ausgeführt

werden können. Bei iOS besteht diese Möglichkeit nicht. Auf iOS können jedoch unter gewissen Bedingungen Aufgaben im Hintergrund ausgeführt werden, wenn eine Applikation minimiert wird. Des Weiteren gewährt das iOS-System einer Applikation kurz Rechenzeit, wenn das mobile Endgerät einen signifikanten Ortswechsel erfährt. Diese Möglichkeiten sollen im Rahmen eines Plugins für NativeScript abstrahiert werden, so dass plattformunabhängig Code in den beschriebenen Hintergrundmodi der mobilen Betriebssysteme ausgeführt werden kann. Dieses Plugin soll dabei unabhängig vom Framework für Kontextsensitivität entwickelt werden.

FA-16 Beispielapplikation: Es soll eine Beispielapplikation implementiert werden, die das Framework ausführt und auf beispielhafte Kontextereignisse reagiert. Die Applikation soll sich dabei sowohl im Vordergrund als auch im Hintergrund kontextsensitiv verhalten. Für jeden der beschriebenen Kontext-Provider (vgl. **FA-09** bis **FA-13**) soll ein beispielhaftes Kontextereignis über eine Benutzeroberfläche definiert werden können. Wenn ein Kontextereignis eintritt, soll die Applikation eine Benachrichtigung auf dem mobilen Endgerät anzeigen. Des Weiteren soll die Applikation rohe Kontextinformationen (vgl. **FA-14**) von Kontext-Providern innerhalb einer SQLite-Datenbank protokollieren und dem Nutzer anzeigen können. Die Anwendung soll zudem eine Benutzeroberfläche zur Verwaltung von Herzfrequenzmessgeräten (vgl. **FA-11**) und zur Mikrophonkalibrierung (vgl. **FA-13**) bereitstellen. Vom System erkannte Aktivitäten (vgl. **FA-10**) sollen innerhalb einer eigenen Ansicht aufgeschlüsselt werden. Dies soll das Testen der Aktivitätenerkennung unterstützen. Innerhalb einer Übersicht sollen alle vom Framework bereitgestellten Kontextinformationen symbolisch dargestellt werden. Des Weiteren soll dem Nutzer eine Möglichkeit angeboten werden, die automatische Kontextverarbeitung des Frameworks zu starten und zu stoppen.

6.1.2 Nicht-funktionale Anforderungen

Die folgenden nicht-funktionalen Anforderungen beschreiben Qualitätsmerkmale, die für das Framework allgemein oder für einzelne Funktionalitäten angestrebt wurden.

NFA-01 Wartbarkeit, Änderbarkeit und Erweiterbarkeit: Das Framework soll möglichst modular aufgebaut sein, so dass Aufgabenbereiche klar voneinander getrennt sind. Dabei sollen Funktionalitäten größtenteils in eigene, unabhängige NativeScript-Plugins ausgelagert werden. Dies soll dann geschehen, wenn die Auslagerung von Funktionalitäten sinnvoll erscheint oder einzelne Module eine ausreichende Funktionalität bereitstellen, um eigenständig genutzt werden zu können. Die Implementierung als NativeScript-Plugins ermöglicht, dass über den npm-Paketmanager einzelne Funktionalitäten aktualisiert oder neue Funktionalitäten hinzugefügt werden können. Diese Herangehensweise spiegelt sich auch in der modularisierten Kontexterfassung, Kontextvorverarbeitung und Bereitstellung von Kontextinformationen wider (vgl. **FA-04**). Der Modularisierungsansatz ermöglicht Entwicklern, eigene Kontext-Provider für das Framework zu entwickeln. Dabei können Kontext-Provider als eigene NativeScript-Plugins implementiert werden, wodurch auch diese über den npm-Paketmanager eingebunden und verwaltet werden können. Diese Trennung von einzelnen Aufgabengebieten bzw. Verantwortlichkeiten über Komponenten und Plugins soll zu einer verbesserten Wartbarkeit und Änderbarkeit führen. Über abstrakte Implementierungen und generische Schnittstellen soll das Framework beliebige Kontext-Provider-Implementierungen verwenden können. Zusätzliche Werkzeuge wie der Kontextverlauf oder die zeitliche Kontextvalidität sollen beliebige Datenstrukturen unterstützen. So wird sichergestellt, dass das Framework leicht mit neuen Kontext-Providern erweitert werden kann.

NFA-02 Plattformunabhängigkeit: Das Framework und dessen Komponenten sollen plattformunabhängig verwendet werden können. Dies schließt auch die Hintergrundverarbeitung ein, die unabhängig vom Framework als NativeScript-Plugin bereitgestellt wird (vgl. **FA-15**). Damit soll gewährleistet werden, dass eine einzelne Codebasis für alle Plattformen verwendet werden kann und Applikationsentwickler sich nicht mit Plattformspezifika beschäftigen müssen. Da bei der Kontexterfassung oft plattformspezifische native APIs verwendet werden müssen, soll der Zugriff auf diese über eigene Plugins abstrahiert werden. Dadurch muss nur auf der untersten Ebene der Frameworkarchitektur zwischen den Plattformen differenziert werden.

Alle weiteren Komponenten sollen so weit wie möglich plattformunabhängig implementiert werden.

NFA-03 Konfigurierbarkeit Die im Framework verwendeten Komponenten sowie das Plugin zur Hintergrundverarbeitung sollen vom Applikationsentwickler möglichst detailliert eingestellt werden können. Dafür soll sogar Zugriff auf Konfigurationsparameter von Modulen gewährleistet werden, die in der Architektur sehr tief angesiedelt sind und auf die nicht direkt zugegriffen werden kann. Beispielsweise sollen Zugriffsintervalle für native APIs vom Applikationsentwickler konfiguriert werden können, obwohl diese Zugriffe von dazwischenliegenden Modulen abstrahiert werden. Die Konfiguration der einzelnen Komponenten soll dabei möglichst einfach gestaltet sein. Durch einfache aber detaillierte Konfigurationsoptionen können Applikationsentwickler das Framework und dessen Funktionsweise individuell für ihre Applikation anpassen. So können Entwickler zum Beispiel den Grad der Kontextsensitivität mit dem Energieverbrauch ins Gleichgewicht bringen.

NFA-04 Leistung und Effizienz: Es muss berücksichtigt werden, dass NativeScript-Applikationen komplett im UI-Thread ausgeführt werden (siehe Abschnitt 4.2). So muss darauf geachtet werden, dass rechenintensive oder blockierende Aufgaben des Frameworks nicht die Benutzeroberfläche oder andere Aufgaben einer Applikation beeinträchtigen oder gar blockieren. Viele asynchrone native APIs der mobilen Betriebssysteme werden in eigenen Threads ausgeführt. Es existieren jedoch auch synchrone bzw. blockierende APIs. Aus diesen Gründen sollen rechenintensive oder blockierende Aufgaben über NativeScript-Worker in eigene Threads ausgelagert werden. Auf Android soll das Plugin zur Hintergrundverarbeitung (vgl. **FA-15**) die Möglichkeit nutzen, innerhalb eines eigenen Prozesses zu operieren.

6.2 Konzeption und Architektur

Dieser Abschnitt stellt die Architektur und das Konzept des entwickelten Frameworks für Kontextsensitivität vor. Zunächst wird die Gesamtarchitektur des Frameworks präsentiert und anschließend werden einzelne Komponenten und deren Funktionsweise genauer betrachtet. Dabei wird auf die Kontexterfassung, die logikbasierte Bereitstellung von Kontextinformationen und auf die zentrale Evaluation und Auslösung von Kontextereignissen eingegangen. Abschließend werden plattformspezifische Möglichkeiten zur Hintergrundausführung vorgestellt, die innerhalb des konzeptionierten NativeScript-Plugins eingesetzt werden sollen.

6.2.1 Architektur

Die Architektur des vorgestellten Frameworks besteht aus drei Hauptbestandteilen. Das Herz des Frameworks ist ein sogenannter Kontext-Prozessor. Über diese Komponente werden beliebig viele Kontext-Provider und Kontextereignisse verwaltet. Zudem wird im weiteren Verlauf dieser Arbeit davon ausgegangen, dass das Framework über ein Plugin im Hintergrund einer Applikation ausgeführt wird. Das Framework kann zwar auch ohne Einschränkungen im Vordergrund einer Applikation betrieben werden. Jedoch sollten kontextsensitive Applikationen auch auf Kontextereignisse reagieren können, wenn der Nutzer nicht mit dem mobilen Endgerät interagiert.

Abbildung 6.1 gibt einen Gesamtüberblick über die Architektur des Frameworks und darüber, wie dieses in eine NativeScript-Applikation eingebettet ist. Dabei kann eine beliebige Anzahl an Kontext-Providern beim Kontext-Prozessor registriert werden. Kontext-Provider werden als NativeScript-Plugins implementiert und sind für die Kontexterfassung, Kontextvorverarbeitung und die Bereitstellung von Kontextinformationen verantwortlich. Zur Datensammlung und Kontexterfassung verwenden Kontext-Provider weitere NativeScript-Plugins, die den Zugriff auf native APIs plattformübergreifend abstrahieren. Damit soll gewährleistet werden, dass die Implementierung von Framework-Komponenten plattformunabhängig geschehen kann. Des Weiteren können beim Kontext-Prozessor beliebig viele Kontextereignisse registriert werden. Solche Kon-

textereignisse bestehen aus Aktionen und Kontextbedingungen. Letztere sind in einem logischen Ausdruck gekapselt. Die elementaren Bausteine dieser logischen Ausdrücke werden über sogenannte Kontext-Queries evaluiert. Kontext-Queries sind Methoden, die von Kontext-Providern bereitgestellt werden und den Zugriff auf Kontextinformationen abstrahieren. Zusätzlich wurde ein Event-Bus-Plugin entwickelt, welches eine plattformunabhängige Kommunikation zwischen verschiedenen Programmteilen erlaubt. Dies wird zum Beispiel eingesetzt, um Kontextinformationen in der Benutzeroberfläche der Beispielapplikation anzuzeigen.

Der zyklische Programmablauf des Frameworks ähnelt dem allgemeinen Konzept kontextsensitiver Systeme nach Knappmeyer et al. (vgl. Abbildung 5.2). Der Ablauf kann, wie in Abbildung 6.1 dargestellt, in folgende Schritte unterteilt werden:

1. Der Kontext-Prozessor startet die Kontext-Provider und damit die Kontexterfassung und -verarbeitung.
2. Die Kontext-Provider stellen in dieser Architektur den Motor dar, der die Kontextüberprüfung antreibt. Immer dann, wenn bei einem Kontext-Provider eine neue Kontextinformation zur Verfügung steht, wird der Kontext-Prozessor benachrichtigt.
3. Daraufhin iteriert der Kontext-Prozessor über alle registrierten Kontextereignisse und überprüft deren Kontextbedingungen. Die Überprüfung verschachtelter Kontextbedingungen erfolgt dabei rekursiv und wird in Unterabschnitt 6.2.3 genauer betrachtet. Kontextereignisse werden im Rahmen dieser Architektur vom Applikationsentwickler definiert und beim Kontext-Prozessor registriert. Wenn alle Kontextbedingungen für ein Ereignis erfüllt sind, wird die registrierte Aktion ausgeführt. Eine Aktion besteht dabei aus beliebiger Programmlogik.
4. Zur Evaluation der Kontextbedingungen werden Kontext-Queries eingesetzt, um einzelne Kontextmerkmale bei den Providern abzufragen und zu evaluieren.

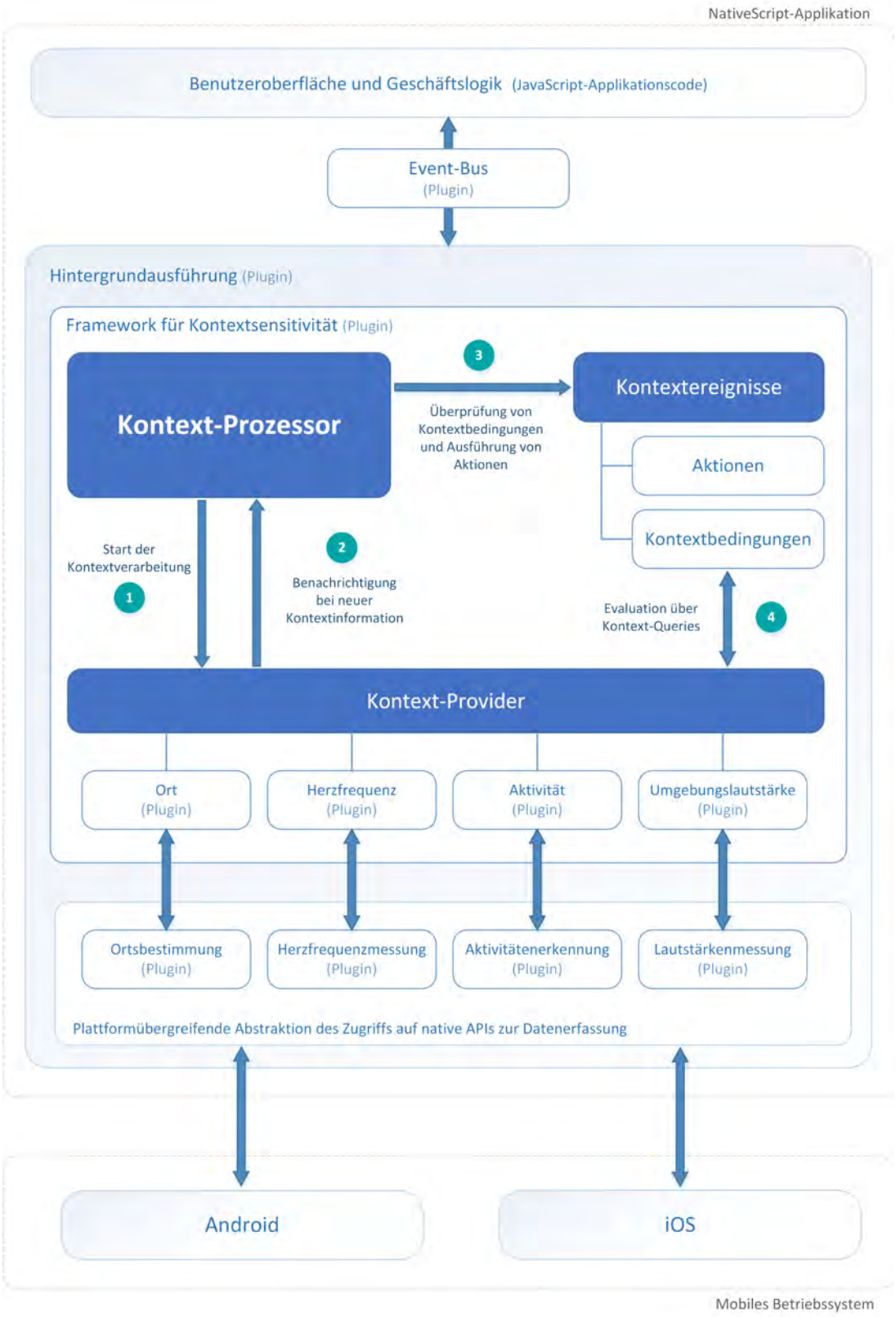

Abbildung 6.1: Abstrahierte Systemarchitektur des Frameworks

Alle Komponenten, die innerhalb der Abbildung 6.1 als Plugin gekennzeichnet sind, entsprechen NativeScript-Plugins. Hinsichtlich der Datenerfassung standen zum plattformübergreifenden Zugriff auf native APIs bereits teilweise Plugins zur Verfügung. Dazu zählt ein Plugin zur Messung der Herzfrequenz über Bluetooth-Low-Energy-Geräte sowie ein Plugin zur Bestimmung des Ortes über GPS. Die restlichen Plugins zum Zugriff auf native APIs wurden im Verlauf dieser Arbeit entwickelt. Zudem behandelt diese Arbeit die Entwicklung eines Plugins zur Ausführung von Logik im Hintergrund einer Applikation. Hierauf wird in Kapitel 7 näher eingegangen.

6.2.2 Kontexterfassung, Kontextvorverarbeitung und Bereitstellung von Kontextinformationen

Die Kontexterfassung, Kontextvorverarbeitung und Bereitstellung von Kontextinformationen erfolgt im vorgestellten Framework über sogenannte Kontext-Provider. In dieser Arbeit wurden Kontext-Provider für den örtlichen Kontext, die Herzfrequenz eines Nutzers, die Umgebungslautstärke und zur Bestimmung von Aktivitäten entwickelt.

Bei der Kontexterfassung werden Daten erfasst, die dazu verwendet werden können, den Kontext einer Entität zu beschreiben. Bei den hier implementierten Kontext-Providern kommen physikalische und logische Sensoren zum Einsatz. Der plattformübergreifende Zugriff auf Sensoren über native APIs wird in dieser Architektur mit zusätzlichen Plugins abstrahiert, damit Kontext-Provider plattformunabhängig implementiert werden können. Mit dem Konzept der Kontext-Provider können aber auch virtuelle Sensoren (vgl. Abschnitt 5.3) verwendet werden, die nicht zwangsweise native APIs benötigen. Ein Beispiel hierfür ist ein Kontext-Provider, der Informationen aus einem Benutzerprofil zur Verfügung stellt. Wie im vorherigen Abschnitt erwähnt, fungieren die Kontext-Provider als Motor des Frameworks, indem neue Kontextinformationen die Überprüfung von Kontextereignissen auslösen. Neue Informationen stehen bei der Verwendung von physikalischen Sensoren typischerweise dann zur Verfügung, wenn über die nativen Schnittstellen neue Daten geliefert werden. Darüber verändern sich auch die bereitgestellten Kontextinformationen mit der Zeit. Kontext-Provider, die solche Sensoren verwenden, werden als dynamisch bezeichnet. Im Gegensatz dazu kann eine automatische Bereitstellung von neuen Daten

bei Kontext-Providern, die virtuelle Sensoren verwenden, nicht zur Verfügung stehen. In diesem Fall wird von statischen Kontext-Providern gesprochen. Bei der Verwendung von statischen Kontext-Providern muss das Framework bzw. der Kontext-Prozessor anderweitig angetrieben werden. Sollten neben statischen auch dynamische Kontext-Provider eingesetzt werden, stellt dies kein Problem dar. In diesem Fall lösen die dynamischen Kontext-Provider eine Kontextüberprüfung aus. Sollten jedoch ausschließlich statische Kontext-Provider eingesetzt werden, kommt es zu keiner automatischen Kontextüberprüfung. Dies muss bei der Applikationsentwicklung berücksichtigt werden, indem zum Beispiel Kontextereignisse manuell überprüft werden. Im Rahmen dieses Frameworks erscheint der ausschließliche Einsatz von statischen Kontext-Providern jedoch als wenig sinnvoll.

Die Kontextvorverarbeitung beschreibt den Vorgang, bei dem erfasste Daten aufbereitet werden, um Kontextinformationen aus diesen abzuleiten. Hierfür können zum Beispiel Filtertechniken oder andere mathematische Verfahren eingesetzt werden. In der vorgestellten Architektur ist die Kontextvorverarbeitung alleinige Aufgabe der jeweiligen Kontext-Provider. Die Kontextvorverarbeitung spielt für andere Komponenten des Systems keine Rolle und kann innerhalb der Kontext-Provider nach Belieben gestaltet sein. Um die Implementierung von Kontext-Providern zu unterstützen, bietet das Framework generische Komponenten zur zeitbasierten Validierung und zur Speicherung von Kontextinformationen an. Diese Komponenten können mit beliebigen Datenstrukturen arbeiten, die im JSON-Format serialisiert werden können. Dadurch wird die Entwicklung von Kontext-Providern hinsichtlich der Kontextmodellierung nur minimal eingeschränkt.

Die Bereitstellung von Kontextinformationen muss im vorgestellten System über sogenannte Kontext-Queries erfolgen. Kontext-Queries sind Methoden, die ein Kontextmerkmal oder eine Kontextinformation symbolisch beschreiben und einen booleschen Ausdruck zurückgeben. Boolesche Ausdrücke sind in der hier vorgestellten Architektur Objekte, die das Interface `BooleanExpression` und damit eine `evaluate()`-Methode implementieren. Diese `evaluate()`-Methode hat als Rückgabewert den primitiven Datentyp `boolean`. Der Rückgabewert beschreibt, ob ein Kontextmerkmal zutrifft oder nicht. Eine beispielhafte Kontext-Query ist die Methode `isRunning() : AtomicExpression`, welche über die zurückgegebene `AtomicExpression`-Instanz

aussagt, ob der Nutzer gerade rennt oder nicht. `AtomicExpression`-Objekte implementieren das `BooleanExpression`-Interface und sind die elementarsten Bausteine, der hier verwendeten logischen Ausdrücke.

Abbildung 6.2 zeigt den Informationsfluss von der Datenerfassung bis zur Bereitstellung von Kontextinformationen über Kontext-Queries. In dieser Illustration wird deutlich, dass auch rohe Sensordaten teilweise als Kontextinformationen bereitgestellt werden können. Dies kann zum Beispiel bei logischen oder virtuellen Sensoren Sinn machen, wenn die vorhandenen Daten bereits aussagekräftig genug sind.

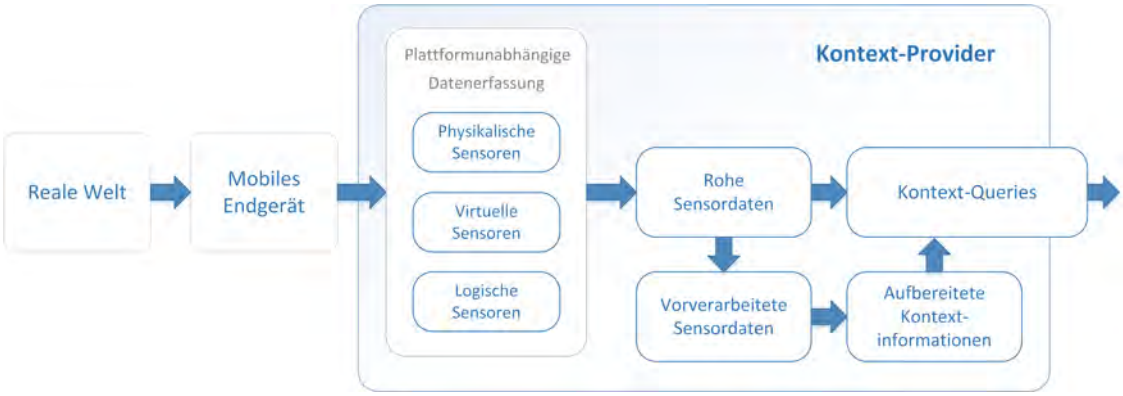

Abbildung 6.2: Informationsfluss bei Kontext-Providern

6.2.3 Regelbasierte Kontextereignisse

Wie in Unterabschnitt 6.2.1 erwähnt, können Applikationsentwickler sogenannte Kontextereignisse definieren und beim Kontext-Prozessor registrieren. Kontextereignisse bestehen dabei aus Aktionen und Kontextbedingungen. Aktionen sind beliebiger JavaScript- bzw. TypeScript-Code, der ausgeführt wird, wenn die Kontextbedingungen eines Ereignisses erfüllt sind. Dieser Code wird bei der Instanziierung eines Kontextereignisses per Callback übergeben.

Kontextbedingungen werden als logische Ausdrücke formuliert. Dafür wurde eine objektorientierte Hierarchie entworfen. Logische Ausdrücke bestehen aus einzelnen oder verschachtelten Objekten, die die Schnittstelle `BooleanExpression` realisieren. Solche Realisierungen implementieren eine `evaluate()`-Methode, die den primitiven Datentyp `boolean` zurückgibt. Über diese Methode können logische Ausdrücke evaluiert werden. Zur Bereitstellung solcher Ausdrücke wird für Kontext-Provider eine `AtomicExpression`-Klasse bereitgestellt, die ebenfalls diese Schnittstelle implementiert. Instanzen dieser Klasse werden von den Kontext-Queries (vgl. Unterabschnitt 6.2.2) der Kontext-Provider zurückgegeben. Diese Instanzen enthalten einen booleschen Callback, der innerhalb des Sichtbarkeitsbereichs eines Kontext-Providers ausgeführt und evaluiert wird. Die `AtomicExpression`-Klasse ermöglicht keine weitere Verschachtelung und beschreibt als atomarer logischer Ausdruck Kontextinformationen bzw. Kontextmerkmale. Zur Verknüpfung und Verschachtelung dieser elementaren Bausteine werden Klassen angeboten, welche die booleschen Operatoren UND, ODER und NICHT repräsentieren. Diese Klassen implementieren selbst die `BooleanExpression`-Schnittstelle und ermöglichen die Verknüpfung oder Negation von `BooleanExpression`-Instanzen. Die Beziehungen der angesprochenen Klassen werden in Abbildung A.2 im Anhang illustriert.

Kontextbedingungen können verwendet werden, um Situationen zu beschreiben, bei denen eine Aktion ausgeführt oder das Verhalten einer Applikation angepasst werden soll. In der vorgestellten Architektur können unter Einsatz von booleschen Operatoren verschachtelte, logische Ausdrücke zur Konstruktion von Kontextbedingungen verwendet werden. Die booleschen Operatoren sind unär oder binär und beinhalten dabei ein oder

zwei Kindelemente. Bei der Verwendung solcher Operatoren entstehen Regelbäume, die im Aufbau denen von Wang et al. [35] ähneln. Diese werden rekursiv evaluiert, sobald dem System neue Kontextinformationen zur Verfügung stehen. Die Evaluation von logischen Verknüpfungen erfolgt über die booleschen Operatoren von JavaScript. Ein beispielhafter Regelbaum zur Definition einer abstrakten Situation ist in Abbildung 6.3 dargestellt. Die Blätter des Baumes stellen Kontextmerkmale dar und verweisen auf Kontext-Queries, um evaluiert zu werden. Kontextmermale entsprechen immer einer `AtomicExpression`-Instanz, die von den Kontext-Queries zurückgegeben wird.

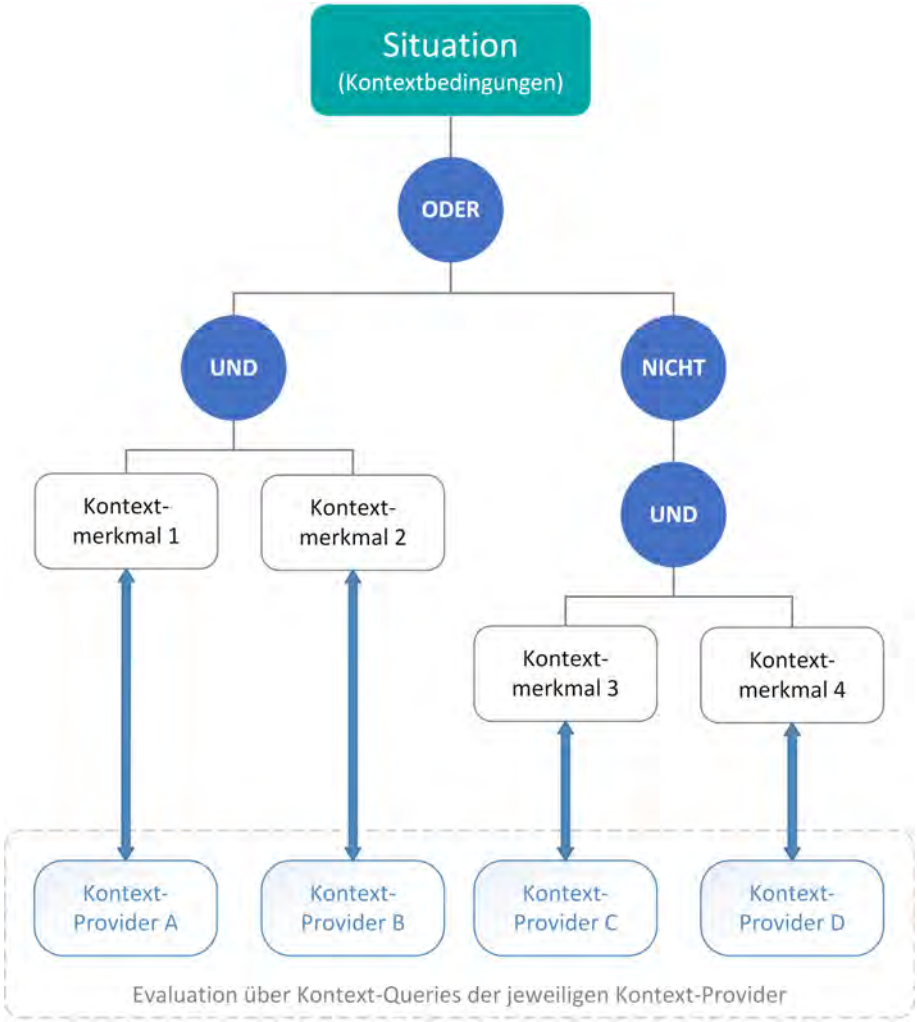

Abbildung 6.3: Regelbäume zur Definition von Situationen

6.2.4 Zentrale regelbasierte Evaluation und Auslösung von Kontextereignissen

Die zentrale Komponente des Frameworks ist der Kontext-Prozessor. Dieser hat unter anderem die Aufgabe, Kontext-Provider zu verwalten. Nach der Instanziierung des Kontext-Prozessors können Applikationsentwickler Kontext-Provider bei diesem registrieren. Eine weitere Aufgabe ist die Steuerung der Kontextverarbeitung. Kontext-Provider werden standardmäßig vom Kontext-Prozessor gestartet und gestoppt. Dies kann bei Bedarf für einzelne Kontext-Provider über einen Parameter deaktiviert werden. Der Kontext-Prozessor informiert auch registrierte Kontext-Provider über Änderungen im Programmablauf, die auf iOS auftreten, wenn beispielsweise die Applikation in den Hintergrundmodus übergeht. Über diesen Mechanismus können Kontext-Provider auf solche Änderungen reagieren. Zudem ermöglicht der Kontext-Prozessor anderen Systemkomponenten den Zugriff auf Kontext-Provider. Dies erlaubt beispielsweise auch eine Kommunikation zwischen Providern. Neben den Kontext-Providern verwaltet der Kontext-Prozessor auch Kontextereignisse, die vom Applikationsentwickler registriert werden. Sobald ein Provider dem Prozessor meldet, dass neue Kontextinformationen zur Verfügung stehen, werden alle registrierten Kontextereignisse überprüft. Sollten die Kontextbedingungen eines Ereignisses zutreffen, wird die dafür entsprechende Aktion ausgeführt (vgl. Unterabschnitt 6.2.3).

Dieses Konzept entspricht einem regelbasierten Schlussfolgerungsmechanismus wie ihn Knappmeyer et al. beschreiben [37]. Die Kontextbedingungen, die in der vorgestellten Architektur an Kontextereignisse gebunden sind, beschreiben die Regeln. Über solche logikbasierten Regeln können komplexere Situationen und damit High-Level-Kontextinformationen abgeleitet werden. Die Menge der Kontextereignisse kann daher als Regelbasis interpretiert werden. Alle Kontextinformationen bzw. Kontextmerkmale werden von Kontext-Providern erfasst und bereitgestellt. Die Bereitstellung erfolgt logikbasiert über Kontext-Queries (vgl. Unterabschnitt 6.2.2). Daher kann die Menge aller Kontext-Provider als Wissensbasis verstanden werden. Abbildung 6.4 zeigt eine abstrahierte Darstellung der Framework-Architektur und illustriert die zentrale Rolle des Kontext-Prozessors unter Verwendung einer Regel- und Wissensbasis.

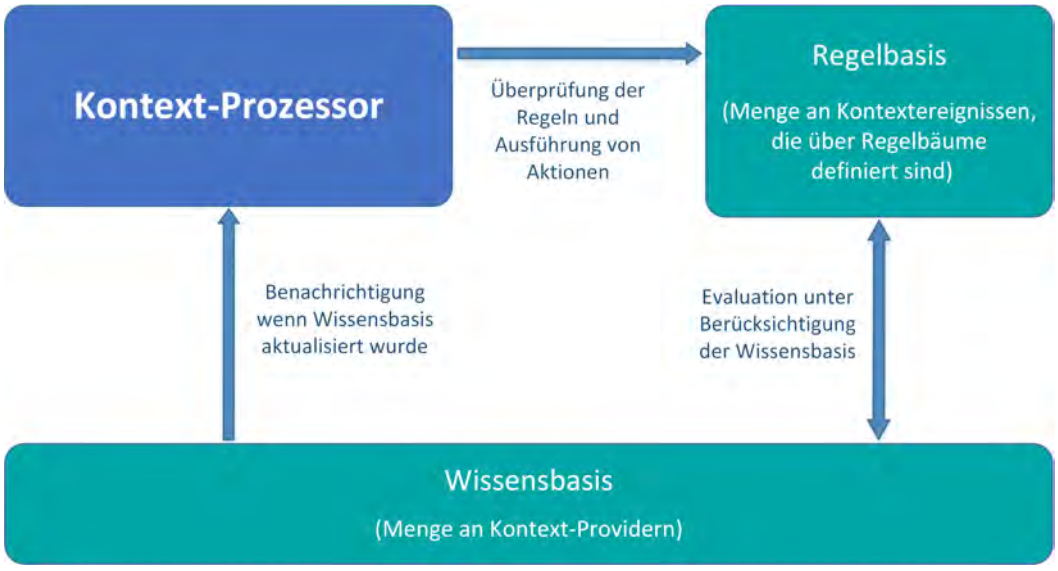

Abbildung 6.4: Regel- und Wissensbasis in der Framework-Architektur

6.2.5 Hintergrundausführung

Mobile Applikationen sollten jederzeit kontextsensitiv agieren können, um sich nach dem Ubiquitous-Computing-Paradigma (vgl. Kapitel 5) in den Alltag eines Nutzers zu integrieren. Hierfür ist ein Konzept zur Hintergrundausführung des Frameworks erforderlich, damit das System nicht auf eine Interaktion des Nutzers angewiesen ist. Durch die Ausführung im Hintergrund besteht für mobile Applikationen die Möglichkeit, proaktiv zu handeln oder Informationen bereitzustellen, wenn ein gewisser Kontext erkannt wurde.

NativeScript bietet keine Möglichkeit, plattformunabhängig Programmlogik im Hintergrund einer Applikation auszuführen. Getrennt vom Framework für Kontextsensitivität wurde im Rahmen dieser Arbeit ein Plugin entworfen, das beliebigen JavaScript- bzw. TypeScript-Code im Hintergrund ausführen kann. Über das Plugin wird die Hintergrundausführung so abstrahiert, dass Applikationsentwickler sich nicht mit plattformspezifischen Implementierungsdetails beschäftigen müssen. Über einfache Konfigurationsmöglichkeiten soll der Entwickler die Arten der Hintergrundausführung bestimmen können. So kann das Laufzeitverhalten individuell an die Anforderungen einer Applikation angepasst werden. Aus diesem Grund sollen auch möglichst viele unterschiedliche Arten

der Hintergrundausführung angeboten werden. Im Folgenden werden verschiedene Arten der Hintergrundausführung auf Android und iOS betrachtet, die im Rahmen des entwickelten Plugins berücksichtigt wurden.

Für Android können zur Hintergrundausführung sogenannte Services [151] implementiert werden. Solche können Aufgaben im Hintergrund einer Applikation oder auch vollkommen unabhängig von einer Applikation ausführen. Services haben keine Benutzeroberfläche und können von einer Applikation, von Systemereignissen oder nach dem Hochfahren eines mobilen Endgeräts automatisch gestartet werden. Zur Implementierung werden von Android zwei Arten von Services bereitgestellt. Mit einem IntentService [152] kann man dabei endliche Aufgaben im Hintergrund ausführen. Sobald die Programmlogik innerhalb eines IntentServices verarbeitet wurde, wird dieser terminiert. Dabei wird nicht auf das Beenden von asynchronen Aufgaben gewartet. Aus diesem Grund eignet sich diese Art der Hintergrundausführung nicht für kontextsensitive Applikationen, die unbestimmt viel Rechenzeit im Hintergrund benötigen. Im Gegensatz dazu kann ein Hintergrund-Service [153] auch asynchrone Aufgaben über einen unbestimmt langen Zeitraum ausführen. Solch ein Service wird entweder über die implementierte Logik oder durch das Betriebssystem beendet. So kann es vorkommen, dass das Betriebssystem kurzfristig einen Hintergrund-Service beendet, wenn dringend Hardwareressourcen benötigt werden. Der Service kann jedoch so implementiert werden, dass er vom Betriebssystem direkt wieder neu gestartet wird, wenn die kurzfristig benötigten Ressourcen wieder zur Verfügung stehen. Zudem kann ein Hintergrund-Service auch eingesetzt werden, um einen Intervall-Service zu implementieren. Abbildung 6.5 vergleicht das Laufzeitverhalten eines solchen mit dem eines normalen Hintergrund-Services. Ein Intervall-Service hat eine gewisse Laufzeit Δt_1, nach der er sich selbst terminiert, und ein Intervall Δt_2, das definiert, wann er periodisch ausgeführt wird.

Auf iOS stehen im Gegensatz zu Android keine Hintergrunddienste zur Verfügung, die unabhängig von einer Applikation ausgeführt werden können. Es besteht jedoch die Möglichkeit, Code auszuführen, wenn sich eine Applikation im Hintergrund befindet. Eine iOS-Applikation befindet sich in einem Hintergrundmodus, wenn sie vom Nutzer oder vom Betriebssystem minimiert, aber noch nicht beendet wurde. In diesem Fall bestehen verschiedene Möglichkeiten, Programmlogik im Hintergrund auszuführen.

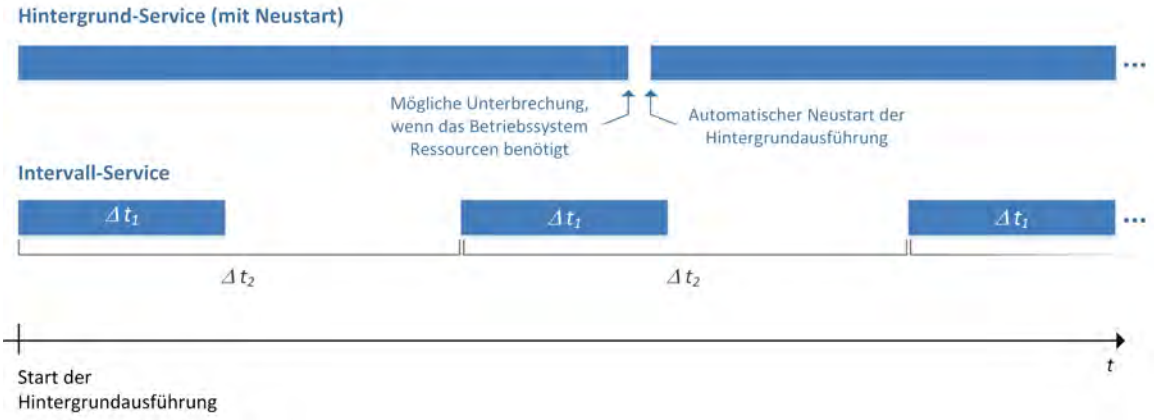

Abbildung 6.5: Möglichkeiten zur Hintergrundausführung auf Android-Geräten

Standardmäßig gewährt das iOS-Betriebssystem einer Applikation opportunistisch einen kurzen Moment Rechenzeit, wenn diese den Hintergrundmodus betritt. Dies waren zum Zeitpunkt der vorliegenden Arbeit 10 Sekunden. Unter Verwendung einer nativen Schnittstelle kann die Registrierung einer Hintergrundaufgabe erfolgen, wodurch die Zeitspanne auf ungefähr 180 Sekunden verlängert werden kann. Diese Form der einmaligen Hintergrundausführung eignet sich nicht zur Implementierung einer kontextsensitiven Applikation. Der angesprochene Hintergrundmodus kann jedoch unter gewissen Bedingungen zeitlich unbegrenzt genutzt werden. Dabei schränkt Apple eine solche langlaufende Hintergrundausführung auf gewisse Anwendungsfälle ein [154]. So wird dieser Hintergrundmodus zum Beispiel zur Verfügung gestellt, wenn die Applikation im Hintergrund zur Ortung verwendet wird, mit Bluetooth-Low-Energy-Accessoires kommuniziert oder Audiodateien abgespielt werden. Zur Aktivierung des Modus müssen für den Anwendungsfall spezifische Berechtigungen angefordert werden. Je nach dem, welche Kontext-Provider des hier vorgestellten Frameworks zum Einsatz kommen, können Applikationen diesen Modus für kontextsensitives Verhalten im Hintergrund nutzen. Dieser Modus wird nur durch einen Neustart des Gerätes, durch das manuelle Terminieren über den Taskmanager, durch das Öffnen der dazugehörigen Applikation oder vom Betriebssystem beendet, wenn dringend Ressourcen benötigt werden. Einen weiteren Hintergrundmodus stellt der sogenannte Background-Fetch dar. Dieser wird ebenfalls aktiv, wenn eine Applikation vom Nutzer oder vom Betriebssystem minimiert wird. Wie

Abbildung 6.6 illustriert, werden bei diesem Modus regelmäßig 30 Sekunden Rechenzeit zur Verfügung gestellt. Die Zeit zwischen den Ausführungen dieses Modus Δt_{opp} wird vom Betriebssystem opportunistisch bestimmt und variiert. In der Dokumentation von iOS wird beschrieben, dass dieser Hintergrundmodus ausgeführt wird, „wenn sich eine gute Gelegenheit bietet“ [154]. Im Rahmen dieser Arbeit konnte beobachtet werden, dass Δt_{opp} unter anderem von der Tageszeit abhängig ist. Während tagsüber der Modus ungefähr alle 15 Minuten ausgeführt wurde, kam es nachts teilweise über Stunden zu keiner Aktivierung.

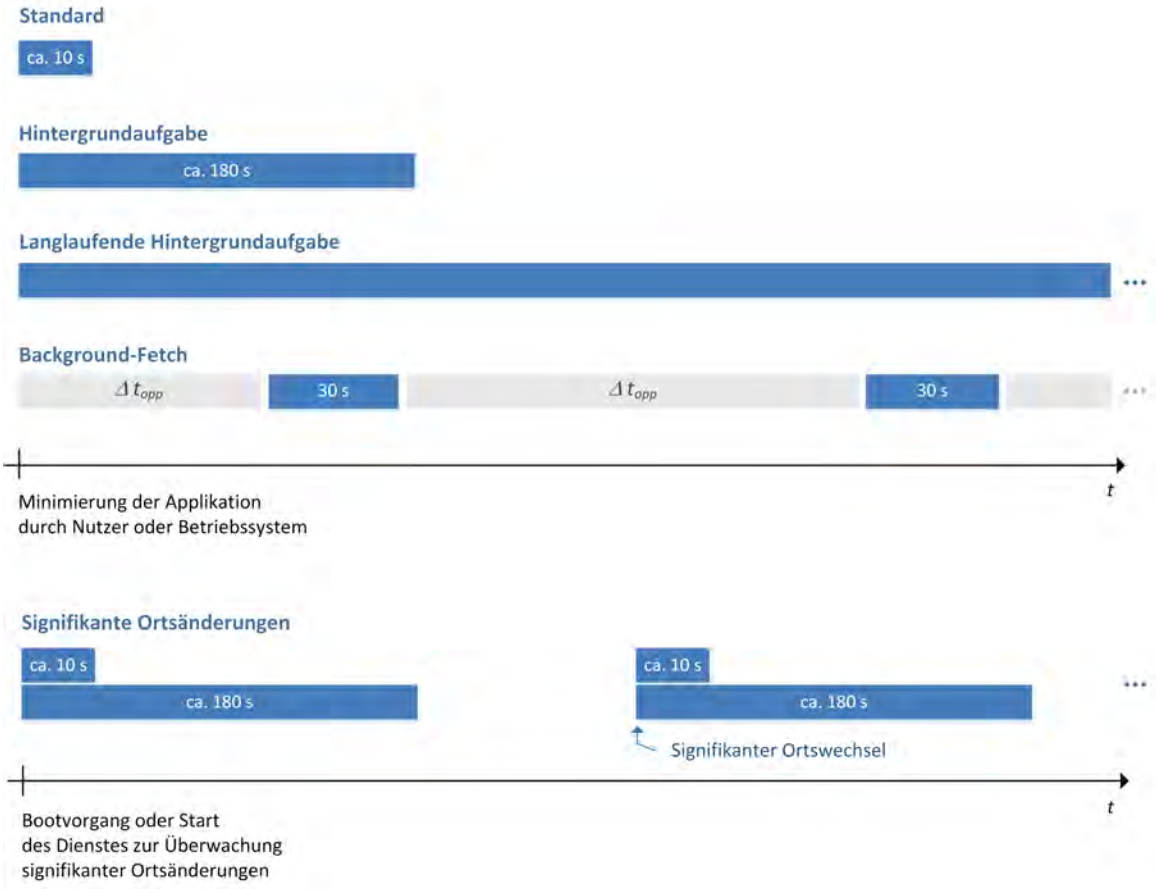

Abbildung 6.6: Möglichkeiten zur Hintergrundausführung auf iOS-Geräten

Die angesprochenen iOS-Hintergrundmodi werden erst aktiviert, wenn der Nutzer mit der Applikation explizit interagiert und diese anschließend minimiert wird. iOS bietet jedoch über eine Schnittstelle einen Ortungsdienst an, der auch komplett beendete Applikatio-

nen wieder aktiviert und der automatisch nach dem Hochfahren des Gerätes gestartet werden kann. Dieser Ortungsdienst überwacht signifikante Ortsänderungen und startet eine dafür registrierte Applikation im Hintergrund, sobald eine solche Ortsänderung registriert wurde. Dabei werden ungefähr 10 Sekunden Rechenzeit gewährleistet. Über die Registrierung einer Hintergrundaufgabe kann diese Zeitspanne auf 180 Sekunden verlängert werden. Laut der iOS-Dokumentation [155] wird dieser Hintergrundmodus mindestens alle 15 Minuten ausgelöst, auch wenn in der Zwischenzeit keine signifikanten Ortsänderungen registriert wurden. Dieses Verhalten konnte im Rahmen dieser Arbeit nicht beobachtet werden. Die Aktivierung des Modus fand nur bei tatsächlichen Ortsänderungen statt. Dennoch bietet dieser Modus eine Möglichkeit, kontextsensitives Verhalten im Hintergrund einzusetzen, selbst wenn eine Applikation komplett beendet wurde. Dies kann insbesondere für Applikationen Sinn machen, die auf Ortsänderungen reagieren wollen und nicht kontinuierlich Rechenzeit benötigen. Alternativ kann dieser Modus auch eingesetzt werden, um den Nutzer darauf hinzuweisen, dass er über den Start der Applikation das kontextsensitive Verhalten aktiveren kann. Nach dem Start der Applikation kann dann eine langlaufende Hintergrundaufgabe eingesetzt werden, um kontinuierlich Rechenzeit vom Betriebssystem zu erhalten.

7

Implementierung

In diesem Kapitel werden ausgewählte Aspekte der Implementierung des vorgestellten Frameworks anhand von Codebeispielen vorgestellt. Zunächst wird auf die Modularisierung des Frameworks eingegangen. Es wird aufgezeigt, wie mit Abhängigkeiten zwischen Plugins umgegangen wird. Danach wird die Datenerfassung über native APIs vorgestellt. Diese Daten werden von den im Rahmen der vorliegenden Arbeit entwickelten Kontext-Providern verwendet, deren Implementierung anschließend behandelt wird. Im Rahmen dieser Arbeit wurden Kontext-Provider entwickelt, um den Ort, die Herzfrequenz des Nutzers, die Aktivität eines Nutzers und die Umgebungslautstärke als Kontextinformationen bereitzustellen. Zur Hintergrundausführung des Frameworks wurde ein Plugin entwickelt, dessen Implementierung ebenfalls vorgestellt wird. Abschließend wird die Verwendung des Frameworks demonstriert und eine Beispielapplikation zum Testen des Frameworks vorgestellt.

7.1 Modularisierung und Abhängigkeiten

Die Implementierung des Frameworks sollte, wie in Unterabschnitt 6.1.2 beschrieben, möglichst modular erfolgen. Dies hat mehrere Gründe. Zum einen soll dadurch die Wartbarkeit und Änderbarkeit gewährleistet werden. Zum anderen soll NativeScript-Entwicklern eine Möglichkeit geboten werden, das Framework mit eigenen Kontext-Providern zu erweitern. Die von nativen APIs abhängige Datenerfassung wird zusätzlich über eigene Module abstrahiert, so dass die Entwicklung von Framework-Komponenten plattformunabhängig erfolgen kann. Die Modularisierung der einzelnen Komponenten ist über NativeScript-Plugins realisiert. Abbildung 7.1 zeigt die Abhängigkeitsstruktur

von Plugins bei der Verwendung des Frameworks mit einem einzelnen beispielhaften Kontext-Provider.

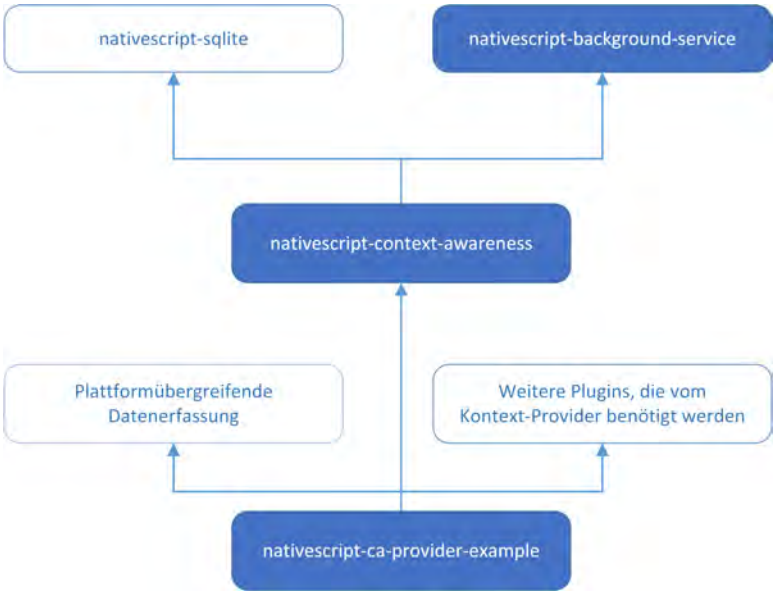

Abbildung 7.1: Plugin-Abhängigkeiten des Frameworks

Die Hauptbestandteile des Frameworks werden in einem NativeScript-Plugin namens nativescript-context-awareness gebündelt. Dieses beinhaltet den Kontext-Prozessor sowie Schnittstellen, Konzepte und abstrakte Klassen, um Kontextereignisse zu definieren und Kontext-Provider zu implementieren. Das Plugin zur Hintergrundausführung heißt nativescript-background-service und wird in Abschnitt 7.4 vorgestellt. Zu diesem Plugin besteht eine Abhängigkeit, da das Framework Änderungen im Laufzeitverhalten behandeln kann. Des Weiteren besteht eine Abhängigkeit zum nativescript-sqlite-Plugin, welches ein plattformunabhängiges Arbeiten mit SQLite-Datenbanken ermöglicht. Dieses Plugin wird verwendet, um einen generischen Kontextverlauf über das Framework bereitzustellen. Kontext-Provider werden für das Framework als eigene NativeScript-Plugins entwickelt. In Abbildung 7.1 ist das der beispielhafte Kontext-Provider nativescript-ca-provider-example. Kontext-Provider sind immer vom nativescript-context-awareness-Plugin abhängig, da dieses Schnittstellen und abstrakte Klassen zur Implementierung von Kontext-Providern zur Verfügung stellt. Weitere Abhängigkeiten von Plugins entste-

hen bei Kontext-Providern durch den Zugriff auf native APIs, welcher nach dem hier vorgestellten Konzept über eigene Plugins abstrahiert werden soll.

NativeScript nutzt zur Verwaltung von Plugins den npm-Paketmanager [138]. Bei diesem können Abhängigkeiten von anderen npm-Paketen über eine `package.json`-Datei definiert werden. Bei der Installation eines Plugins werden diese Abhängigkeiten behandelt und die entsprechenden npm-Pakete installiert. Jedoch werden diese Pakete nicht global für eine NativeScript-Applikation, sondern lokal im Sichtbarkeitsbereich des Plugins installiert. Im Falle der Kontext-Provider führt dieses Verhalten dazu, dass jeder Kontext-Provider ein eigenes nativescript-context-awareness-Paket verwendet. Aus diesem Grund können Abhängigkeiten von anderen Plugins nicht über npm-Abhängigkeiten innerhalb einer `package.json`-Datei definiert werden. Damit Applikationsentwickler sich nicht manuell um die Plugin-Abhängigkeiten der Framework-Komponenten kümmern müssen, werden automatisierte Skripte eingesetzt. Diese werden bei der Installation eines Plugins ausgeführt und installieren fehlende Plugins. Hierfür wird, wie in Listing 8 dargestellt, in der `package.json`-Datei eines Plugins ein `scripts`-Element definiert. Darüber können beliebige Befehle registriert werden. Über das Schlüsselwort `install` kann ein Befehl definiert werden, der automatisch nach der Installation des Plugins ausgeführt wird. Dies wird im Rahmen dieser Arbeit bei der Entwicklung von Plugins genutzt, um über den `node`-Befehl JavaScript-Dateien auszuführen. Diese Dateien installieren fehlende Plugins über die NativeScript-CLI, was beispielhaft für das nativescript-context-awareness-Plugin in Listing 9 dargestellt ist.

```
1  ...
2  "scripts": {
3    "build": "tsc",
4    "install": "node ./hooks/postinstall.js"
5  }
6  ...
```

Listing 8: Ausführung individueller Skripte bei Plugin-Installationen

```
1  // hooks/postinstall.js
2
3  var exec = require('child_process').exec;
4
5  return new Promise(function(resolve, reject) {
6    // install the nativescript-background-service plugin
7    exec('(cd ./../.. && tns plugin add nativescript-background-service)')
8        .stderr.pipe(process.stderr);
9    // install the nativescript-sqlite plugin
10   exec('(cd ./../.. && tns plugin add nativescript-sqlite)')
11       .stderr.pipe(process.stderr);
12   resolve();
13 });
```

Listing 9: Installation fehlender Plugins mit JavaScript

Alle im Rahmen dieser Arbeit entwickelten NativeScript-Plugins wurden mit TypeScript als npm-Pakete implementiert. Dabei müssen während der Entwicklung verwendete Module auch im lokalen Sichtbarkeitsbereich eines Plugins vorhanden sein. Dies wird vom TypeScript-Compiler erfordert und ermöglicht den Entwicklungsumgebungen, Funktionen wie eine automatische Codevervollständigung bereitzustellen. Im Gegensatz zu NativeScript-Applikationen können innerhalb der Entwicklungsumgebung npm-Abhängigkeiten problemlos verwendet werden. Diese Abhängigkeiten werden in der `package.json`-Datei eines Plugins über das `devDependencies`-Element definiert. Dadurch werden diese Abhängigkeiten bei der Installation eines Plugins nicht berücksichtigt. Solche Abhängigkeiten können über den Befehl `npm install --ignore-scripts` innerhalb der Entwicklungsumgebung installiert werden. Mit dem Parameter `--ignore-scripts` wird dabei sichergestellt, dass die oben beschriebenen Skripte nicht in der Entwicklungsumgebung ausgeführt werden. Die Kompilierung von TypeScript zu JavaScript wird typischerweise mit dem Befehl `npm run build` bei der Entwicklung von npm-Paketen ausgelöst. Dieser Befehl kann allgemein verwendet werden, um ein npm-Paket für die Veröffentlichung vorzubereiten. Wie in Listing 8 zu sehen ist, wird darüber der TypeScript-Compiler mit dem `tsc`-Befehl aufgerufen.

7.2 Datenerfassung über native APIs

Die Datenerfassung über native APIs erfolgt im hier vorgestellten Konzept plattformunabhängig. Dafür werden native APIs über eigene Plugins abstrahiert. Diese Plugins werden von den Kontext-Providern verwendet, um Daten zu erfassen und Kontextinformationen bereitzustellen. Im Rahmen dieser Arbeit werden dabei der Ort eines Gerätes, die Herzfrequenz eines Nutzers, die Aktivität eines Nutzers und die Umgebungslautstärke erfasst. Im Folgenden werden Plugins vorgestellt, mit denen diese Datenerfassung über native APIs plattformunabhängig ermöglicht wird. Für die Ortsbestimmung und das Erfassen der Herzfrequenz über Bluetooth-Low-Energy-Geräte wurden Plugins von Drittentwicklern verwendet. Zur Erkennung von Aktivitäten über die nativen APIs der mobilen Betriebssysteme und zur Messung der Umgebungslautstärke wurden eigene Plugins entwickelt.

7.2.1 Ort

Zur Bestimmung des Ortes eines mobilen Endgerätes existieren verschiedene Technologien. Heutige Smartphones sind typischerweise mit einem GPS-Empfänger ausgestattet und können so das Global-Positioning-System verwenden, um den eigenen Standort zu bestimmen. Damit kann die Ortsbestimmung außerhalb von Gebäuden mit einer Genauigkeit von ungefähr 10 Metern oder besser erfolgen [156]. Des Weiteren kann die Positionierung mobiler Endgeräte netzwerkbasiert erfolgen. Diese Positionierungsmethoden können jedoch mit der Genauigkeit von GPS-basierten Verfahren meist nicht konkurrieren [156].

Telerik bietet für NativeScript ein Plugin namens nativescript-geolocation an [157], mit dem eine plattformunabhängige Ortsbestimmung möglich ist. Das Plugin steht unter der Apache License Version 2.0 [63] zur Verfügung und wurde im Rahmen dieser Arbeit zur Erfassung des Ortes eingesetzt. Dieses verwendet den GPS-Empfänger der Endgeräte zur Positionierung. Sollte kein GPS-Signal vorhanden sein, wird auf eine netzwerkbasierte Ortsbestimmung zurückgegriffen. Intern verwendet dieses Plugin auf Android-Geräten die LocationManager-API [158] und auf iOS-Geräten das Core-Location-Framework

[159]. Mit diesem Plugin ist es möglich, Ortsänderungen zu überwachen. Immer wenn dem Plugin ein Positionsupdate zur Verfügung steht, wird ein registrierter Callback aufgerufen. Über diesen Callback wird dem Applikationsentwickler ein `Location`-Objekt zur Verfügung gestellt, welches Informationen zum Aufenthaltsort des mobilen Endgeräts bereitstellt. Mit TypeScript kann das Plugin, wie in Listing 10 beispielhaft dargestellt, verwendet werden.

```
1  import * as Geolocation from "nativescript-geolocation";
2
3  ...
4
5  // start location monitoring
6  this.locationMonitoringId : number =
7    Geolocation.watchLocation((location : Geolocation.Location) => {
8      if (location) {
9        console.log("latitude: " + location.latitude +
10                   ", longitude: " + location.longitude);
11     },
12     (e) => {
13       console.log("Error on receiving location update - " + e.message);
14     },
15     { desiredAccuracy: 3, updateDistance: 10, minimumUpdateTime: 1000 * 10}
16   );
17
18 ...
19
20 // stop location monitoring
21 Geolocation.clearWatch(this.locationMonitoringId);
```

Listing 10: Standortüberwachung mit dem Plugin nativescript-geolocation

Über das bereitgestellte `Location`-Objekt können Daten wie der Breitengrad (englisch: latitude), der Längengrad (englisch: longitude) oder die Geschwindigkeit zum Zeitpunkt der Ortsbestimmung erfasst werden. Diese Daten können von einem Kontext-Provider zur Bereitstellung von Kontextinformationen genutzt werden. Die Ortsbestimmung kann wie in Listing 10 über verschiedene Parameter konfiguriert werden. Diese Parameter werden im Rahmen des entsprechenden Kontext-Providers in Unterabschnitt 7.3.1 beschrieben.

7.2.2 Herzfrequenzerfassung über Bluetooth-Accessoires

Die Herzfrequenzerfassung wird im Rahmen dieser Arbeit über Bluetooth-Low-Energy-Geräte realisiert. Es existieren verschiedene Bauarten solcher Herzfrequenzmessgeräte. Oft sind diese als Brustgurt oder Armband im Handel erhältlich. Je nach Bauart kommen unterschiedliche Sensoren zur Messung der Herzfrequenz zum Einsatz. Zum Beispiel können bei einem Brustgurt Hautelektroden verwendet werden, während bei der Messung am Handgelenk die Herzfrequenz über das photoplethysmografische Verfahren bestimmt wird [160]. Bei Letzterem wird Licht in das Hautgewebe ausgestrahlt und die Herzfrequenz durch Messung des reflektierten Lichts berechnet [161]. Im Rahmen dieser Implementierung wurde eine Sportuhr von Mio [162] verwendet, die dieses Verfahren einsetzt. Die hier vorgestellte Herzfrequenzerfassung funktioniert jedoch mit allen Messgeräten, die das standardisierte Generic-Attribute-Profile (GATT) [163] verwenden, um Herzfrequenzwerte über Bluetooth-Low-Energy (Bluetooh-LE) bereitzustellen.

Zur Kommunikation mit Bluetooth-LE-Geräten steht für NativeScript ein kostenfreies Plugin unter der MIT-Lizenz [60] zur Verfügung. Dieses Plugin wird von Eddy Verbruggen unter dem Namen nativescript-bluetooth bereitgestellt [164] und wird hier zur Erfassung von Herzfrequenzdaten verwendet. Mit diesem Plugin wird das Suchen von Peripherie, das Verbindungsmanagement sowie das Schreiben und Lesen von Service-Charakteristiken plattformunabhängig ermöglicht. Des Weiteren werden Methoden angeboten, über die man sich bei Peripherie-Geräten, die einen Benachrichtigungsdienst anbieten, an- oder abmelden kann. Solch ein Dienst wird auch hier verwendet, um kontinuierlich über neue Herzfrequenzmessungen informiert zu werden. Nach dem standardisierten GATT-Protokoll zur Bereitstellung von Herzfrequenzmessungen wird dieser Dienst über den Wert `0x180D` identifiziert. Das entsprechende Service-Charakteristika, welches die Herzfrequenz in der Einheit min^{-1} bzw. als Schläge pro Minute (englisch: beats per minute, bpm) bereitstellt, kann über den Wert `0x2A37` angesprochen werden. Unter Verwendung des nativescript-bluetooth-Plugins kann, wie in Listing 11 beispielhaft demonstriert, ein Callback für neue Messwerte registriert werden. Dieser Callback wird aufgerufen, sobald das Bluetooth-Herzfrequenzmessgerät neue Daten zur Verfügung stellt.

```
import * as Bluetooth from "nativescript-bluetooth";

...

Bluetooth.startNotifying({
  peripheralUUID: '34234-5453-4453-54545',
  serviceUUID: '180D',
  characteristicUUID: '2A37',
  onNotify: function (result) {
    let data = new Uint8Array(result.value);
    console.log("current heart rate: " + data[1] + " bpm");
  }
});
```

Listing 11: Erfassung von Herzfrequenzdaten über Bluetooth-LE-Geräte

7.2.3 Aktivitätenerkennung

Die mobilen Betriebssysteme Android und iOS stellen native Schnittstellen bereit, die Applikationen benachrichtigen können, wenn vom Betriebssystem Aktivitäten des Nutzers erkannt wurden. Hierfür werden auf beiden Plattformen Daten unterschiedlicher Sensoren kombiniert. Die dabei verwendeten Algorithmen sind proprietär und nicht dokumentiert. Sowohl auf Android als auch auf iOS stellen die Schnittstellen Aktivitäten symbolisch über Callbacks bereit. Zu diesen symbolischen Aktivitäten gehören *Autofahren*, *Fahrradfahren*, *Laufen*, *Rennen*, *Ruhen* und nicht bekannte Tätigkeiten, die unter dem Symbol *Unbekannt* gebündelt werden. Auf Android kann zusätzlich erkannt werden, ob ein mobiles Endgerät gerade gekippt bzw. geneigt wird. Auf beiden Plattformen werden Aktivitäten mit einem Wert verknüpft, der die Zuverlässigkeit bzw. Genauigkeit der Aktivitätenerkennung für die entsprechende Messung angibt. Dadurch kann bei der Bereitstellung einer Aktivität als Kontextinformation auch die Genauigkeit berücksichtigt werden. Im Rahmen dieser Arbeit wurde ein Plugin namens nativescript-activity-recognition entwickelt, das diese nativen Schnittstellen plattformunabhängig abstrahiert und für die Entwicklung mit NativeScript bereitstellt. Im Folgenden wird auf Implementierungsdetails eingegangen und die Verwendung des Plugins demonstriert.

Auf Android findet die Aktivitätenerkennung über die Google-Play-Services [165] statt. Diese Anwendung ist Bestandteil vieler Applikationen und ist standardmäßig auf Android-Geräten vorinstalliert. Über die sogenannte ActivityRecognitionApi [166] kann die Aktivitätenerkennung gesteuert und erkannte Aktivitäten per Callback empfangen werden. Die Verwendung dieser Schnittstelle erfordert die Implementierung verschiedener nativer Klassen. Der Programmablauf zur Erfassung von erkannten Aktivitäten über das hier vorgestellte Plugin ist in Abbildung 7.2 grafisch dargestellt.

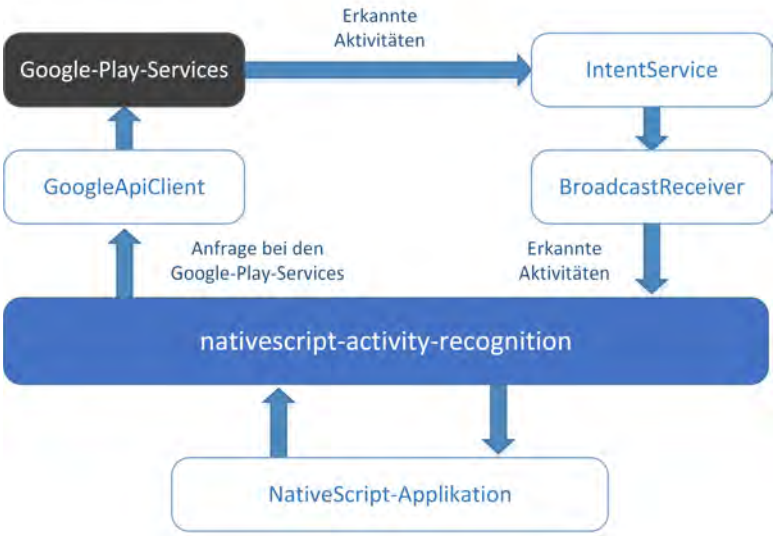

Abbildung 7.2: Nutzung der Aktivitätenerkennung auf Android

Über eine `GoogleApiClient`-Instanz kann mit den Google-Play-Services kommuniziert werden. Mit dieser Verbindung kann ein `IntentService` angemeldet werden, über den erkannte Aktivitäten bereitgestellt werden. Dieser `IntentService` muss zusätzlich über die `AndroidManifest.xml`-Datei registriert werden und wird von den Google-Play-Services aufgerufen, wenn Aktivitäten erkannt wurden. Anschließend sendet dieser einen Broadcast aus, um andere Applikationsteile über die erkannten Aktivitäten zu informieren. In diesem Fall muss ein `BroadcastReceiver` [167] implementiert werden. Die Aktivitätenerkennung der Google-Play-Services verhält sich dabei wie eine Blackbox. Zur Reduzierung des Energieverbrauchs kann bei Verwendung der ActivityRecognitionApi ein Aktualisierungsintervall definiert werden, das die Übermittlung

neu erkannter Aktivitäten zeitlich beschränkt. So kann beispielsweise festgelegt werden, dass nur alle 30 Sekunden erkannte Aktivitäten an die Applikation übermittelt werden sollen.

Bei der Entwicklung dieses Plugins wurde eine Einschränkung bei der Erweiterung nativer Android-Klassen festgestellt. Wie in Abschnitt 4.5 bereits erwähnt wurde, können mit JavaScript bzw. TypeScript keine Standardkonstruktoren für native Klassen definiert werden. Bei Verwendung der ActivityRecognitionApi ist es jedoch zwingend erforderlich, einen `IntentService` zu implementieren. Wenn das Android-Betriebssystem eine `IntentService`-Implementierung instanziieren will, wird ein Standardkonstruktor aufgerufen. Da dieser in JavaScript oder TypeScript nicht definiert werden kann, führt das zum Absturz der Applikation. Die einzige Möglichkeit, diese Limitierung zu umgehen, besteht in einer nativen Java-Implementierung. Für den hier angeführten Fall wird dabei die `IntentService`-Klasse in Java erweitert und als native Android-Bibliothek in das Plugin integriert. Da die Kommunikation mit einem `IntentService` über Intents bzw. Broadcasts abläuft, kann die Implementierung der Klasse isoliert als Java-Archiv in das Plugin eingebunden werden. Bisher wird von NativeScript keine JavaScript- oder TypeScript-basierte Lösung für dieses Problem angeboten. Daher bedeutet diese Limitierung für NativeScript-Entwickler, dass in solchen Fällen Kenntnisse der plattformspezifischen Programmiersprache nötig werden.

Die Aktivitätenerkennung von iOS steht nur auf Apple-Geräten zur Verfügung, die einen Koprozessor zur Erfassung von Bewegungsdaten verbaut haben. Dieser Koprozessor wurde erstmals im iPhone 5S eingesetzt und erfasst Sensordaten vom Beschleunigungssensor, Gyroskop und vom Kompass, um Aktivitäten zu erkennen [168]. Wie bei Android ist auch der hier verwendete Klassifizierungsalgorithmus nicht öffentlich bekannt. Die Verwendung der nativen Aktivitätenerkennung auf iOS erfolgt über das Core-Motion-Framework [169]. Beim Core-Motion-Framework kann über die Klasse `CMMotionActivityManager` ein Callback registriert werden, der aufgerufen wird, wenn Aktivitäten erkannt wurden. Die Verwendung des Core-Motion-Frameworks erfordert im Gegensatz zu den Google-Play-Services weniger Implementierungsaufwand. Die `CMMotionActivityManager`-Klasse stellt Methoden bereit, um die Übermittlung von erkannten Aktivitäten zu starten oder zu stoppen. Erkannte Aktivitäten werden

direkt per Callback in die aufrufende Klasse geliefert. Während auf Android die Genauigkeit der erkannten Aktivitäten in Prozent angegeben wird, verwendet iOS drei Stufen, um die Genauigkeit zu beschreiben. Dabei wird eine erkannte Aktivität mit einer niedrigen, mittleren oder hohen Genauigkeit markiert. Zur abstrahierten Verwendung in NativeScript-Applikationen muss eine einheitliche Beschreibung der Genauigkeit angeboten werden. Für dieses Plugin kommt dabei die detailliertere Skala von Android zum Einsatz, die mit ganzzahligen Prozentwerten arbeitet. Daher werden die symbolischen Genauigkeiten der iOS-Aktivitätenerkennung auf die Skala von Android abgebildet.

```
import { ActivityRecognition, Activity, ActivityType }
  from "nativescript-activity-recognition";

...

export class ActivityComponent implements OnInit {

  public ngOnInit() : void {
    if (ActivityRecognition.isAvailable()) {
      // request authorization on iOS (resolves always on Android)
      ActivityRecognition.requestAuthorization().then((granted) => {
        if (granted) this.start();
      });
    }
  }

  private start() : void {
    // register callback
    ActivityRecognition.onActivitiesDetected(
      (detectedActivities : Array<Activity>) => {
        for (let activity of detectedActivities) {
          if (activity.getType() == ActivityType.RUNNING) {
            console.log("running confidence: " + activity.getConfidence());
          }
        }
      });
    // request activity updates (30 seconds interval on Android)
    ActivityRecognition.requestActivityUpdates(30);
  }

  private stop() : void {
    ActivityRecognition.removeActivityUpdates();
  }
}
```

Listing 12: Verwendung des nativescript-activity-recognition-Plugins

Die Verwendung des Plugins in NativeScript-Applikationen kann wie in Listing 12 erfolgen. Über die Methode `isAvailable()` wird überprüft, ob die Aktivitätenerkennung auf dem mobilen Endgerät zur Verfügung steht. Auf Android wird abgefragt, ob die Google-Play-Services installiert sind, während bei iOS überprüft wird, ob die Hardware die Aktivitätenerkennung unterstützt. Auf iOS muss der Nutzer einmalig die Berechtigung gewähren, dass die Applikation auf die Bewegungsdaten zugreifen darf. Dies kann über die Methode `requestAuthorization()` erfolgen. Über die `onActivitiesDetected()`-Methode kann ein Callback registriert werden, der eine Liste von erkannten Aktivitäten empfängt. Mit der `requestActivityUpdates()`-Methode kann der Empfang gestartet werden. Ein optionaler Parameter definiert dabei das Aktualisierungsintervall für Android. Über die Methode `removeActivityUpdates()` kann der Empfang von erkannten Aktivitäten wieder gestoppt werden. Das Plugin stellt zudem die Klasse `Activity` bereit, die eine erkannte Aktivität repräsentiert und die Genauigkeit sowie den Typ dieser beschreibt. Mit dem Aufzählungstyp `ActivityType` werden alle verfügbaren Aktivitätstypen plattformübergreifend definiert.

7.2.4 Umgebungslautstärke

Die Messung der Umgebungslautstärke kann bei Smartphones über das integrierte Mikrophon erfolgen. Darüber lässt sich bestimmen, ob der Nutzer sich in einer lauten oder leisen Umgebung befindet. Zur Erfassung der Umgebungslautstärke wurde im Rahmen dieser Arbeit das Plugin nativescript-ambient-noise-level entwickelt, das die Messung der Lautstärke plattformunabhängig abstrahiert.

Typischerweise wird Lautstärke über den Schalldruckpegel gemessen. Schalldruck beschreibt Druckschwankungen bzw. Schwingungen, die in einem Übertragungsmedium wie Luft, bei der Ausbreitung von Schall auftreten [170]. Dabei wird der Schalldruck in der SI-Einheit Pascal (Pa) gemessen [171]. Der Schalldruckpegel (englisch: sound pressure level, SPL) wird hingegen in Dezibel (dB) angegeben und ist eine relative Größe. Zur genaueren Beschreibung wird die Einheit als dB_{SPL} bezeichnet. Der Schalldruckpegel beschreibt das logarithmische Verhältnis einer Messung zu einem Referenzwert p_0.

Dieser Wert wurde auf $p_0 = 20\ \mu Pa$ festegelegt, was als untere Hörschwelle beim Menschen gilt [172]. Auf Smartphones ist die genaue Bestimmung des Schalldruckpegels nicht möglich [173], da die technischen Eigenschaften der verbauten Mikrophone nicht bekannt sind und zwischen Geräten stark variieren. Die nativen Schnittstellen von Android und iOS ermöglichen jedoch die Messung der vom Mikrophon wahrgenommenen Lautstärke in der dB_{FS}-Einheit (englisch: decibels relative to full scale). Diese logarithmische Einheit beschreibt eine Messung relativ zur vollen digitalen Skala des Mikrophons. Dabei stellt 0 dB_{FS} den maximalen Pegelwert, also die Vollaussteuerung und damit die größtmögliche messbare Lautstärke, dar. Niedrigere Lautstärken werden über negative Werte dargestellt. Messwerte dieser Skala werden vom hier vorgestellten Plugin für Entwickler bereitgestellt. Des Weiteren ermöglicht das Plugin eine Kalibrierung, über die der Nutzer eine ruhige akustische Umgebung definieren kann. Dafür wird der Pegelwert über eine konfigurierbare Zeitspanne unter Verwendung einer 16-Bit-Quantisierung gemessen und der quadratische Mittelwert dieser Messung als Referenzwert p_0 gespeichert. Zu diesem Referenzwert kann über einen gemessenen Pegelwert p ein relativer Schalldruckpegel L_p wie folgt berechnet werden [172]:

$$L_p = 20 \log_{10} \left(\frac{p}{p_0} \right) \tag{7.1}$$

Hierbei ist anzumerken, dass es sich nur um eine grobe Schätzung der Lautstärke handelt, die auf der subjektiven Kalibrierung durch den Nutzer basiert. Es werden dabei weder physiologische noch psychologische Aspekte der Akustik berücksichtigt. Zudem handelt es sich hierbei um einen gerätezentrierten Ansatz, der unterschiedliche Distanzen zu einer Schallquelle nicht beachtet. Dennoch eignen sich diese Schätzwerte, um eine grobe Aussage zur vom Nutzer subjektiv wahrgenommenen Umgebungslautstärke treffen zu können. So kann beispielsweise ausgesagt werden, ob die gemessene Lautstärke deutlich lauter als in einer ruhigen Umgebung ist [173].

Zur Messung der Pegelwerte des Mikrophons wird auf Android die native `AudioRecord`-API [174] verwendet. Mit dieser wird ein 16-Bit-quantisierter Audio-Stream aufgenommen.

Die abgetasteten Werte werden in einen Byte-Buffer geschrieben, worüber anschließend der quadratische Mittelwert der Messung berechnet wird. Dieser Mittelwert kann über den höchstmöglichen Pegelwert in die dB_{FS}-Skala transformiert werden. Der höchstmögliche Pegelwert beträgt bei 16-Bit +32767 bzw. -32768. Das Aufnehmen über die `AudioRecord`-API erfolgt synchron, was zur Blockierung der Applikation führen kann. Aus diesem Grund wurde ein NativeScript-Worker implementiert, der die Audioaufnahme in einen eigenen Thread auslagert (vgl. Abschnitt 4.2).

Bei iOS wird ebenfalls eine 16-Bit-Quantisierung zur Audioaufnahme verwendet. Hierfür kommt die native `AVAudioRecorder`-Schnittstelle [175] zum Einsatz. Im Gegensatz zu Android erfolgt die Audioaufnahme asynchron, weshalb kein NativeScript-Worker eingesetzt werden muss. Die Schnittstelle stellt den gemessenen Pegelwert vorverarbeitet als Wert der dB_{FS}-Skala zur Verfügung. Zur Bestimmung des relativen Schalldruckpegels L_p, muss dieser Wert der dB_{FS}-Skala auf iOS zunächst in den Pegelwert zurückgerechnet werden.

Das Plugin abstrahiert die beschriebenen Zugriffe auf native APIs und kann, wie in Listing 13, plattformunabhängig zur periodischen Messung der Umgebungslautstärke eingesetzt werden. Zur Überprüfung, ob ein Mikrophon zur Verfügung steht, wird die Methode `microphoneAvailable()` angeboten. Über `hasRecordAudioPermission()` und `requestRecordAudioPermission()` können die benötigten Berechtigungen ausgelesen bzw. angefragt werden.
Über die Methode `isCalibrated()` kann überprüft werden, ob bereits ein gespeicherter Kalibrierungswert vorhanden ist. Über `performCalibration()` kann der Kalibrierungsprozess gestartet werden, der einen Mittelwert der gemessenen Pegelwerte im lokalen Speicher der Applikation abspeichert. Optionale Konfigurationsparameter sowie ein Callback können über die `init()`-Methode definiert werden. Der Callback wird aufgerufen, wenn eine Messung beendet wurde, und stellt die gemessene Umgebungslautstärke zur Verfügung. Die Konfigurationsparameter werden in Tabelle B.1 im Anhang beschrieben. Über die Methoden `start()` und `stop()` kann die periodische Messung dabei gesteuert werden.

```
import { AmbientNoiseLevel, DecibelScale }
  from "nativescript-ambient-noise-level";

...

export class AmbientNoiseComponent implements OnInit {

  public ngOnInit() : void {
    // check if a microphone is available
    if (AmbientNoiseLevel.microphoneAvailable()) {
      // request permissions
      if (!AmbientNoiseLevel.hasRecordAudioPermission())
        AmbientNoiseLevel.requestRecordAudioPermission()
      }

      // initialize plugin
      AmbientNoiseLevel.init({
        decibelScale: DecibelScale.DB_SPL_ESTIMATED,
            measurementInterval: 120,
            measurementDuration: 10,
            calibrationDuration: 10,
            onNewNoiseLevelCallback:
              (decibel : number, type : DecibelScale) => {
                if (type == DecibelScale.DB_FS) {
                  console.log(decibel + " dBFS");
                } else {
                  console.log(decibel + " dB-SPL");
                }
              }
            }
      });
    }
  }

  private start() : void {
    // start measurement
    AmbientNoiseLevel.start();
  }

  private stop() : void {
    AmbientNoiseLevel.stop();
  }

  private calibrate() : void {
    if (!AmbientNoiseLevel.isCalibrated()) {
      AmbientNoiseLevel.performCalibration();
    }
  }
}
```

Listing 13: Verwendung des nativescript-ambient-noise-level-Plugins

7.3 Kontext-Provider

Die hier präsentierten Kontext-Provider verwenden die im vorherigen Kapitel vorgestellten Plugins zur Erfassung von kontextrelevanten Daten. Diese Daten werden aufbereitet und als Kontextinformation über Kontext-Queries anderen Systemkomponenten zur Verfügung gestellt. Die Implementierungsdetails solch einer Kontextvorverarbeitung können beliebig gestaltet werden und spielen für andere Komponenten keine Rolle. Kontext-Queries geben eine `AtomicExpression`-Instanz zurück, die als boolescher Ausdruck evaluiert werden kann. Solche Kontext-Queries sind öffentliche Methoden und beschreiben eine Kontextinformation symbolisch. Inwiefern ein logischer Ausdruck dabei aufgelöst wird, bleibt dem Entwickler eines Kontext-Providers überlassen. Des Weiteren können Kontext-Provider beliebige Schnittstellen zur Konfiguration oder Verwaltung bereitstellen. Der Zugriff auf diese Schnittstellen ist über den Kontext-Prozessor möglich. Dieser stellt Referenzen zu registrierten Kontext-Providern über eine Methode bereit.

Zur Implementierung von Kontext-Providern wurden diverse TypeScript-Klassen entworfen, die in Abbildung A.1 im Anhang illustriert sind. Jeder Kontext-Provider muss die abstrakte Klasse `ContextProvider` implementieren. Instanzen dieser Klasse können beim Kontext-Prozessor registriert werden. Über die Methode `onNewContextInfo()` kann ein Kontext-Provider den Kontext-Prozessor informieren, wenn neue Kontextinformationen zur Verfügung stehen. Dadurch wird die Kontextüberprüfung angestoßen. Des Weiteren kann der Kontext-Prozessor die Kontextverarbeitung über die `start()`- und `stop()`-Methoden der Provider steuern.

Für die Berücksichtigung von vergangenen Kontextdaten wird eine Klasse bereitgestellt, mit der ein generischer Kontextverlauf in Provider integriert werden kann. Dieser Kontextverlauf verwaltet Daten mit Hilfe einer SQLite-Datenbank. Diese wird über eine separate Klasse angesprochen, die wie ein Repository [176] agiert. Zusätzlich wird eine generische Klasse `ContextInformationValidity<T>` bereitgestellt, die es ermöglicht, beliebige Datenstrukturen mit einer Gültigkeitsdauer zu versehen. Diese Klasse wird mit einer beliebigen Information instanziiert und speichert dabei einen Zeitstempel zwischen. Ob eine Kontextinformation abgelaufen oder noch gültig ist, kann über die boolesche Methode `isValid(periodOfValidity : number)` abgefragt werden. Die Klassen

zur Implementierung eines Kontextverlaufs und zur Verwendung einer Gültigkeitsdauer sind beispielhaft in Abbildung A.1 im Anhang dargestellt.

Im Folgenden werden vier beispielhafte Kontext-Provider vorgestellt, die im Rahmen dieser Arbeit entwickelt wurden. Diese Kontext-Provider stellen Kontextinformationen bezüglich des Ortes, der Herzfrequenz, der ausgeübten Aktivität und der Umgebungslautstärke bereit.

7.3.1 Örtliche Nähe und Geschwindigkeit

Der Kontext-Provider nativescript-ca-provider-geolocation stellt räumliche Kontextinformationen zur Verfügung. Hierfür verwendet er das plattformunabhängige nativescript-geolocation-Plugin (vgl. Unterabschnitt 7.2.1), um regelmäßig Positionsaktualisierungen vom mobilen Betriebssystem zu erhalten. Zur Verwendung der Ortsbestimmung müssen vom Nutzer Berechtigungen erteilt werden. Der Kontext-Provider bietet hierfür entsprechende Schnittstellen an, die Methoden des nativescript-geolocation-Plugins aufrufen. Über diese Methoden können die Berechtigungen angefordert und ausgelesen werden.

Der Programmablauf des Kontext-Providers wird durch Positionsaktualisierungen der nativen APIs angetrieben. Eine Positionsaktualisierung wird in einem `Geolocation`-Objekt gekapselt und enthält Informationen über den Aufenthaltsort, die Ausrichtung oder die Geschwindigkeit eines Gerätes. Der Aufenthaltsort wird über den geographischen Längen- und Breitengrad angegeben. Sobald neue Ortsinformationen vorhanden sind, wird ein solches Objekt im Rahmen einer `ContextInformationValidity`-Instanz zwischengespeichert. Zur späteren Verwendung innerhalb der Kontext-Queries werden Positionsaktualisierungen optional auch in einem Kontextverlauf abgespeichert. Dies können Applikationsentwickler über einen Konfigurationsparameter festlegen. Danach wird der Kontext-Prozessor informiert, dass neue Kontextinformationen zur Verfügung stehen.

Zur Bereitstellung von Kontextinformationen bietet der Kontext-Provider vier Kontext-Queries an. Darüber können die räumliche Nähe zu einem festgelegten Ort sowie die gemessene Geschwindigkeit abgefragt werden. Die Distanz zwischen zwei geographi-

schen Orten wird über den Längen- und Breitengrad berechnet. Im Rahmen dieses Kontext-Providers wurde die Haversine-Formel [177] verwendet, die es ermöglicht, die Großkreisdistanz d zwischen zwei Orten unter Berücksichtigung der Erdkrümmung in Metern zu bestimmen:

$$\begin{aligned} a &= \sin^2(\Delta\varphi/2) + \cos\varphi_1 + \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2) \\ c &= 2 \cdot \arctan^2\left(\sqrt{a}, \sqrt{(1-a)}\right) \\ d &= R \cdot c \end{aligned} \tag{7.2}$$

Dabei ist φ der Breitengrad, λ der Längengrad und R der mittlere Erdradius (6371000 m). Unter Verwendung einer maximalen Distanz kann über diese Art der Distanzberechnung festgestellt werden, ob ein Nutzer sich in der Nähe eines bestimmten Ortes aufhält oder aufgehalten hat. Der Parameter der maximalen Distanz kann bei Kontext-Queries vom Applikationsentwickler angegeben werden. Bei der Wahl dieses Parameters sollte die Genaugikeit der nativen Positionierungsverfahren und die räumliche Größe eines Ortes berücksichtigt werden.

Der Kontext-Provider stellt verschiedene Konfigurationsparameter bereit, die in Tabelle B.2 im Anhang beschrieben werden. Zur Definition von Kontextereignissen werden folgende Kontext-Queries angeboten:

Aktuelle Räumliche Nähe zu einem geographischen Ort:

```
nearTo(location : Geolocation, maxDistance : number) : AtomicExpression
```

Bei dieser Kontext-Query wird die letzte Positionsaktualisierung herangezogen, um die Distanz zum angegeben Ort `location` zu bestimmen. Wenn die berechnete Distanz kleiner gleich dem Schwellenwert `maxDistance` ist, wird die zurückgegebene `AtomicExpression`-Instanz als wahr interpretiert. Wenn eine maximale Gültigkeits-

dauer über einen entsprechenden Konfigurationsparameter definiert wurde, wird diese bei der Evaluation berücksichtigt.

Aufenthalt in der Nähe eines geographischen Ortes:

```
hasBeenNearTo(location : Geolocation, maxDistance : number,
              seconds : number) : AtomicExpression
```

Hier kann über den Parameter `seconds` eine Zeitspanne definiert werden. Bei der Evaluation zum Zeitpunkt t_E werden über den Kontextverlauf alle Positionsaktualisierungen des Zeitraums $(t_E - seconds)$ bis t_E überprüft. Wenn bei allen überprüften Positionen die Distanz zum angegebenen Ort `location` kleiner gleich dem Schwellenwert `maxDistance` ist, wird die Kontext-Query als wahr interpretiert. Zur Verwendung dieser Kontext-Query muss der Applikationsentwickler über einen entsprechenden Parameter den Kontextverlauf aktivieren. Anderenfalls werden diese Kontext-Queries als falsch interpretiert.

Aktuelle Geschwindigkeit:

```
speedGreaterOrEqualThan(referenceSpeed : number) : AtomicExpression
```

```
speedLessOrEqualThan(referenceSpeed : number) : AtomicExpression
```

Über diese zwei Kontext-Queries kann die aktuelle Geschwindigkeit abgefragt werden. Hierfür wird der Geschwindigkeitswert der letzten Positionsaktualisierung herangezogen. Diese Geschwindigkeit wird mit dem Parameter `referenceSpeed` verglichen. Als Einheit wird hier Meter pro Sekunde verwendet. Die zurückgegebene `Atomic-Expression`-Instanz wird je nach Kontext-Query als wahr interpretiert, wenn

der Wert größer oder kleiner gleich dem angegebenen Referenzwert ist. Wurde eine maximale Gültigkeitsdauer über einen entsprechenden Konfigurationsparameter definiert, wird diese bei der Evaluation berücksichtigt.

7.3.2 Herzfrequenz

Die Bereitstellung der Herzfrequenz eines Nutzers erfolgt über den Kontext-Provider nativescript-ca-provider-heart-rate. Dieser verwendet das nativescript-bluetooth-Plugin (vgl. Unterabschnitt 7.2.2) zur plattformunabhängigen Erfassung der Herzfrequenz eines Nutzers über Bluetooth-LE-Herzfrequenzmessgeräte. Auf Android-Geräten ab Version 6 muss zur Verwendung dieses Plugins im Hintergrund die `ACCESS_COARSE_LOCATION`-Berechtigung vom Nutzer erteilt werden. Zum Auslesen und Anfragen dieser Berechtigung stellt der Kontext-Provider eine Schnittstelle bereit. Neben der Erfassung und Bereitstellung der Herzfrequenz, ermöglicht dieser Kontext-Provider die Verwaltung von Herzfrequenzmessgeräten und ein automatisiertes Verbindungsmanagement. Hierfür kommen die in Abbildung 7.3 dargestellten Komponenten zum Einsatz.

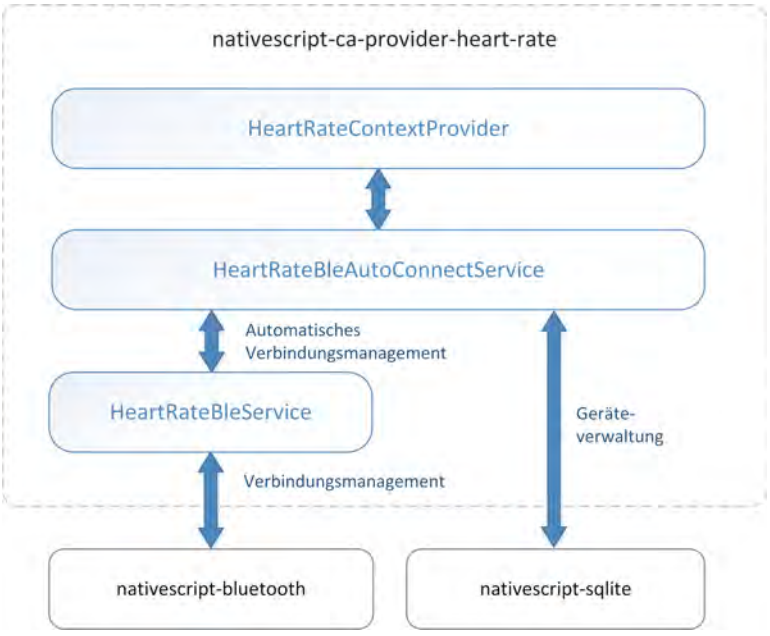

Abbildung 7.3: Komponenten des Herzfrequenz-Kontext-Providers

Die `HeartRateContextProvider`-Klasse erbt von der abstrakten Framework-Klasse `ContextProvider` und fungiert damit als Schnittstelle zum Kontext-Prozessor. Zur Verwaltung von Herzfrequenzmessgeräten und zur automatischen Verbindungsherstellung wird die `HeartRateBleAutoConnectService`-Klasse eingesetzt. Diese verwendet die Klasse `HeartRateBleService`, welche das Verbindungsmanagement über das nativescript-blueotooth-Plugin realisiert. Zum Verbindungsmanagement gehören Aufgaben wie das Suchen von Geräten, der Verbindungsauf- und abbau sowie die Erfassung der Herzfrequenz. Zur Speicherung von bekannten Messgeräten verwendet die `HeartRateBleAutoConnectService`-Klasse das nativescript-sqlite-Plugin. Damit kann das explizite Verbindungsmanagement über den Nutzer minimiert werden. Nach einer einmaligen Bestimmung eines Standard-Herzfrequenzmessgeräts kann der Kontext-Provider vollautomatisch eine Verbindung zu diesem herstellen und die Herzfrequenz erfassen. Wenn während der Kontextverarbeitung keine Verbindung zu einem Gerät besteht, wird versucht, periodisch eine Verbindung zum Standardgerät aufzubauen. Hierfür kann vom Applikationsentwickler ein Intervall definiert werden. Die vorgestellten Klassen bieten zusätzlich Methoden an, über die eine Benutzeroberfläche zur Geräteverwaltung implementiert werden kann.

Die Kontextverarbeitung wird durch eingehende Herzfrequenzwerte angetrieben. Zur Kontextvorverarbeitung kann über einen Konfigurationsparameter ein einfacher gleitender Mittelwert zur Glättung der Daten eingesetzt werden. In diesem Fall wird der Kontext-Prozessor erst benachrichtigt, wenn genügend Samples zur Berechnung des Mittelwerts zur Verfügung stehen. Zusätzlich kann ein Kontextverlauf verwendet werden, der entweder die rohen oder die geglätteten Herzfrequenzdaten speichert. Der Applikationsentwickler kann zudem eine Gültigkeitsdauer für die erfassten Daten definieren, welche im Rahmen der Kontext-Queries evaluiert wird.

In Tabelle B.3 im Anhang werden alle Konfigurationsparameter des Kontext-Providers aufgelistet. Die Bereitstellung von Kontextinformationen zur Herzfrequenz erfolgt über folgende Kontext-Queries. Dabei werden die bereitgestellte Herzfrequenz und Schwellenwertparameter in der Einheit min^{-1} bzw. als Schläge pro Minute angegeben.

Aktuelle Herzfrequenz:

```
bpmGreaterOrEqualThan(threshold : number) : AtomicExpression
```

```
bpmLessOrEqualThan(threshold : number) : AtomicExpression
```

Bei diesen zwei Kontext-Queries wird die aktuelle Herzfrequenz mit dem Schwellenwert `threshold` verglichen. Wenn über einen entsprechenden Konfigurationsparameter ein gleitender Mittelwert zur Kontextvorverarbeitung aktiviert wurde, wird hier der berechnete Mittelwert und damit eine durchschnittliche Herzfrequenz herangezogen. Wenn eine maximale Gültigkeitsdauer definiert ist, werden die Herzfrequenzwerte bei der Evaluation diesbezüglich überprüft. Die zurückgegebene `Atomic-Expression`-Instanz wird je nach Kontext-Query als wahr interpretiert, wenn der Wert größer oder kleiner gleich dem angegebenen Schwellenwert ist.

Mittlere Herzfrequenz über einen längeren Zeitraum:

```
averageBpmGreaterOrEqualFor(threshold : number,
                            seconds : number) : AtomicExpression
```

```
averageBpmLessOrEqualFor(threshold : number,
                         seconds : number) : AtomicExpression
```

Diese Kontext-Queries werden als wahr interpretiert, wenn die mittlere Herzfrequenz für einen längeren Zeitraum den angegebenen Schwellenwert `threshold` über- oder unterschreitet. Über den Parameter `seconds` wird dieser Zeitraum definiert. Zum Zeitpunkt der Evaluation t_E werden hierfür unter Verwendung des Kontextverlaufs alle Messwerte des Zeitraums $(t_E - seconds)$ bis t_E gemittelt. Die Kontext-Queries werden als falsch interpretiert, wenn der Kontextverlauf nicht aktiviert wurde.

7.3.3 Aktivität

Aktuelle und andauernde Aktivitäten werden vom Kontext-Provider nativescript-ca-provider-activity als Kontextinformationen bereitgestellt. Dieser verwendet das nativescript-activity-recognition-Plugin (vgl. Unterabschnitt 7.2.3), das plattformunabhängig über native APIs erkannte Aktivitäten zur Verfügung stellt. Der Kontext-Provider stellt Schnittstellen bereit, über die das Plugin angesprochen werden kann. Diese können vom Applikationsentwickler verwendet werden, um festzustellen, ob das mobile Endgerät die native Aktivitätenerkennung unterstützt. Zusätzlich kann auf iOS-Geräten auch die notwendige Berechtigung zum Zugriff auf Bewegungsdaten ausgelesen und angefragt werden.

Bei diesem Kontext-Provider wird die Verarbeitung über eingehende Aktivitäten angetrieben. Sobald das nativescript-activity-recognition-Plugin erkannte Aktivitäten bereitstellt, werden diese zwischengespeichert und der Kontext-Prozessor informiert. Erkannte Aktivitäten werden in einer `ContextInformationValidity`-Instanz gekapselt und optional in einem Kontextverlauf abgespeichert. Alle Aktivitäten sind dabei mit einem Genauigkeitswert versehen, der von den nativen Schnittstellen mit angegeben wird. Diese Genauigkeitswerte werden bei der Bereitstellung von Kontextinformationen berücksichtigt. Dafür kann der Applikationsentwickler über einen Konfigurationsparameter eine minimale Genauigkeit definieren, die bei einer Aktivität erreicht werden muss, damit diese als gegeben angenommen wird. Bei der Bestimmung des Parameters sollte berücksichtigt werden, dass das Verhalten der nativen Aktivitätenerkennung zwischen verschiedenen Geräten variieren kann. Zudem gibt es Unterschiede zwischen der Aktivitätenerkennung auf Android und der auf iOS. Bei Android kann über einen Konfigurationsparameter bestimmt werden, wie oft die Aktivitätenerkennung Informationen bereitstellt. Dies kann bei iOS nicht gesteuert werden. Diese Faktoren müssen von Applikationsentwicklern bei der Konfiguration und Verwendung dieses Kontext-Providers beachtet werden. Zur differenzierten Behandlung der Plattformen kann zum Beispiel die `config()`-Methode des Kontext-Providers genutzt werden.

Alle Konfigurationsparameter des Kontext-Providers sind in Tabelle B.4 im Anhang aufgelistet. Zur Spezifikation von Kontextbedingungen werden folgende Kontext-Queries bereitgestellt:

Aktuelle Aktivitäten:

```
inVehicle(minimumConfidence? : number) : AtomicExpression
```

```
isBiking(minimumConfidence? : number) : AtomicExpression
```

```
isWalking(minimumConfidence? : number) : AtomicExpression
```

```
isRunning(minimumConfidence? : number) : AtomicExpression
```

```
isStill(minimumConfidence? : number) : AtomicExpression
```

```
isTilting(minimumConfidence? : number) : AtomicExpression
```

Zur Überprüfung dieser Kontext-Queries werden die zuletzt erkannten Aktivitäten herangezogen. Der optionale Parameter `minimumConfidence` beschreibt die minimale Genauigkeit, die eine Aktivität aufweisen muss, um als gegeben interpretiert zu werden. Wenn dieser Parameter nicht angegeben ist, wird ein konfigurierbarer Standardwert verwendet. Bei der Evaluation einer Kontext-Query wird die maximale Gültigkeitsdauer für erkannte Aktivitäten berücksichtigt, insofern der entsprechende Konfigurationsparameter definiert wurde. Die zurückgegebene `Atomic-Expression`-Instanz wird als wahr interpretiert, wenn der Genauigkeitswert der entsprechenden Aktivität größer oder gleich der minimalen Genauigkeit ist. Die Kontext-Query `isTilting()` kann dabei nur auf Android wahr werden (vgl. Unterabschnitt 7.2.3).

Andauernde Aktivitäten:

```
hasBeenInVehicle(seconds : number,
                 minimumAverageConfidence? : number) : AtomicExpression
```

```
hasBeenBiking(seconds : number,
              minimumAverageConfidence? : number) : AtomicExpression
```

```
hasBeenWalking(seconds : number,
               minimumAverageConfidence? : number) : AtomicExpression
```

```
hasBeenRunning(seconds : number,
               minimumAverageConfidence? : number) : AtomicExpression
```

```
hasBeenStill(seconds : number,
             minimumAverageConfidence? : number) : AtomicExpression
```

```
hasBeenTilting(seconds : number,
               minimumAverageConfidence? : number) : AtomicExpression
```

Über diese Kontext-Queries kann festgestellt werden, ob eine Aktivität über einen gewissen Zeitraum ausgeübt wurde. Hierfür muss vom Applikationsentwickler der Kontextverlauf aktiviert werden. Anderenfalls werden die Queries als falsch interpretiert. Der betrachtete Zeitraum wird über den Parameter `seconds` definiert. Bei der Evaluation der Kontext-Queries zum Zeitpunkt t_E wird über alle erkannten Aktivitäten im Zeitraum $(t_E - seconds)$ bis t_E iteriert und die mittlere Genauigkeit der Aktivitätstypen berechnet. Wenn dieser Wert größer oder gleich dem Parameter `minimumAverageConfidence` ist, wird eine entsprechende Aktivität für diesen Zeitraum als gegeben angenommen und die dazugehörige Kontext-Query als wahr interpretiert. Wenn der Parameter `minimumAverageConfidence` nicht angeben ist, wird ein konfigurierbarer Standard-

wert verwendet. Die Kontextbedingung `hasBeenTilting()` kann dabei nur auf Android als wahr interpretiert werden (vgl. Unterabschnitt 7.2.3).

7.3.4 Umgebungslautstärke

Unter Verwendung des Plugins nativescript-ambient-noise-level (vgl. Unterabschnitt 7.2.4) wurde der Kontext-Provider nativescript-ca-provider-ambient-noise implementiert, der die Umgebungslautstärke als Kontextinformation bereitstellt. Hierfür wird die Umgebungslautstärke periodisch über das Mikrophon gemessen. Zur Überprüfung, ob auf dem mobilen Endgerät ein Mikrophon zur Verfügung steht, bietet der Kontext-Provider eine entsprechende Methode an. Zudem können über den Provider die notwendigen Berechtigungen zur Nutzung des Mikrophons ausgelesen und angefordert werden. Die Umgebungslautstärke kann in dB_{FS} oder dB_{SPL} bereitgestellt werden. Bei Verwendung der dB_{SPL}-Skala muss der Nutzer zunächst eine Kalibrierung durchführen, um eine subjektiv wahrgenommene, ruhige Umgebung zu definieren. Diese Kalibrierung kann vom Applikationsentwickler über Methoden des Kontext-Providers implementiert werden. Die zur Beschreibung der Umgebungslautstärke verwendete Einheit kann vom Entwickler über einen Konfigurationsparameter definiert werden.

Nach dem Start der Kontextverarbeitung werden periodisch Audioaufzeichnungen gestartet, um die Umgebungslautstärke zu messen. Sobald eine neue Messung zur Verfügung steht, wird diese innerhalb einer `ContextInformationValidity`-Instanz zwischengespeichert und optional in einem Kontextverlauf abgespeichert. Anschließend wird der Kontext-Prozessor benachrichtigt. Im Rahmen der Vorverarbeitung können Messungen unter Verwendung eines einfachen gleitenden Mittelwerts geglättet werden. Diese Glättung kann vom Applikationsentwickler per Konfigurationsparameter aktiviert und angepasst werden. Bei der Verwendung eines einfachen gleitenden Mittelwerts wird der Kontext-Prozessor erst benachrichtigt, wenn genügend Messungen zur Berechnung des Mittelwerts gesammelt wurden.

Bei der Verwendung dieses Kontext-Providers muss beachtet werden, dass häufige Messungen einen sehr starken Energieverbrauch verursachen können. Zudem muss berücksichtigt werden, dass die Umgebungslautstärke eine relative Kontextinformation

darstellt. So können beispielsweise Reibungsgeräusche in der Tasche eines Nutzers sehr laute Geräusche erzeugen, die vom Smartphone erfasst werden. Diese können jedoch vom Nutzer nicht wahrgenommen werden. Aufgrund dessen ist es empfehlenswert diesen Kontext-Provider nur unter gewissen Kontextbedingungen zu aktivieren. Zum Beispiel kann die Kontextbedingung `isStill()` des Kontext-Providers für Aktivitäten eingesetzt werden, um nur dann die Umgebungslautstärke zu messen, wenn sich das Gerät nicht in Bewegung befindet.

Zur Konfiguration des Kontext-Providers stehen die in Tabelle B.5 im Anhang aufgelisteten Parameter zur Verfügung. Die Bereitstellung der aktuellen Umgebungslautstärke sowie der mittleren Umgebungslautstärke für einen längeren Zeitraum erfolgt über folgende Kontext-Queries:

Aktuelle Umgebungslautstärke:

```
noiseLevelGreaterOrEqualThan(threshold : number) : AtomicExpression
```

```
noiseLevelLessOrEqualThan(threshold : number) : AtomicExpression
```

Diese zwei Kontext-Queries vergleichen die aktuelle Umgebungslautstärke mit dem Schwellenwert `threshold`. Die verwendete Einheit kann dabei vom Applikationsentwickler über einen entsprechenden Konfigurationsparameter bestimmt werden. Wenn die Verwendung eines gleitenden Mittelwerts aktiviert ist, wird dieser berechnet und als Vergleichswert herangezogen. Optional wird bei der Evaluation eine maximale Gültigkeitsdauer der Messungen berücksichtigt. Die zurückgegebene `Atomic-Expression`-Instanz wird je nach Kontext-Query als wahr interpretiert, wenn die Umgebungslautstärke größer oder kleiner gleich dem angegebenen Schwellenwert ist.

Mittlere Umgebungslautstärke über einen längeren Zeitraum:

```
averageNoiseLevelGreaterOrEqualFor(threshold : number,
                                   seconds   : number) : AtomicExpression
```

```
averageNoiseLevelLessOrEqualFor(threshold : number,
                                seconds   : number) : AtomicExpression
```

Bei diesen Kontext-Queries wird die Umgebungslautstärke über einen längeren Zeitraum betrachtet. Hierfür wird ein Kontextverlauf eingesetzt. Zum Zeitpunkt der Evaluation t_E wird der Mittelwert von allen Messwerten des Zeitraums $(t_E - seconds)$ bis t_E berechnet. Wenn dieser Mittelwert den angegebenen Schwellenwert `threshold` über- oder unterschreitet, werden die Kontext-Queries als wahr interpretiert. Die verwendete Einheit kann dabei vom Applikationsentwickler über einen entsprechenden Konfigurationsparameter bestimmt werden. Diese Kontext-Queries basieren auf der Verwendung eines Kontextverlaufs und werden immer als falsch interpretiert, wenn dieser nicht aktiviert wurde.

7.4 Hintergrundverarbeitung

Wie in Unterabschnitt 6.2.5 beschrieben wurde, sollen kontextsensitive Applikationen auch im Hintergrund operieren können. Die dort vorgestellten Arten der Hintergrundausführung wurden in einem eigenständigen NativeScript-Plugin namens nativescript-background-service umgesetzt. Dieses Plugin stellt Applikationsentwicklern nach der Installation eine TypeScript-Klasse bereit, in der beliebige Logik implementiert werden kann. Diese Logik wird über das Plugin beim Start eines Hintergrundmodus ausgeführt.

Im Folgenden wird die Verwendung des Plugins präsentiert. Anschließend wird betrachtet, wie das Plugin auf den unterschiedlichen Plattformen arbeitet. Zudem werden wichtige Aspekte zur Nutzung der iOS-Hintergrundmodi beschrieben.

7.4.1 Verwendung

Bei der Installation des Plugins wird im `app`-Verzeichnis der NativeScript-Applikation die Klasse `BackgroundService` erstellt. Dies wird über ein Skript realisiert, das nach der Installation die entsprechenden Dateioperationen durchführt (vgl. Abschnitt 7.1). Eine Auslagerung dieser Klasse ermöglicht das Aktualisieren oder Deinstallieren des Plugins, ohne dass dabei eine Implementierung des Entwicklers überschrieben oder gelöscht wird. Die Klasse erbt von einer abstrakten Klasse drei Methoden, die vom Applikationsentwickler implementiert werden können.

- `run(startMode : BackgroundMode) : void`
  Innerhalb dieser Methode kann der Entwickler beliebige Logik implementieren, die beim Start eines Hintergrundmodus ausgeführt wird.
- `onStop() : void`
  Über diese Methode können abschließende Aufgaben durchgeführt werden, wenn ein Hintergrundmodus beendet wird.
- `onModeChanged(newMode : BackgroundMode) : void`
  Diese Methode wird vom Plugin aufgerufen, wenn sich die Laufzeitumgebung auf iOS-Geräten ändert. Das ist zum Beispiel der Fall, wenn eine Applikation während eines Hintergrundmodus geöffnet und damit in den Vordergrund gebracht wird.

Die bei der Installation bereitgestellte `BackgroundService`-Klasse enthält eine beispielhafte Implementierung, die während eines Hintergrundmodus alle 10 Sekunden eine Ausgabe in der Konsole anzeigt. Diese Implementierung ist in Listing 18 im Anhang dargestellt.

Die Konfiguration des Plugins muss vor der Initialisierung einer NativeScript-Applikation innerhalb der `main.ts` erfolgen. Hierfür wird eine `init()`-Methode bereitgestellt, die wie in Listing 14 verwendet werden kann. Die verfügbaren Konfigurationsparameter werden in Tabelle B.6 im Anhang beschrieben und können zum Beispiel verwendet werden, um den automatischen Start eines Hintergrundmodus zu definieren. Zur Steuerung der Hintergrundausführung wird eine `BackgroundServiceControl`-Klasse angeboten, mit der die Ausführung manuell gestartet und gestoppt werden kann.

```
// main.ts
import { platformNativeScriptDynamic } from
  "nativescript-angular/platform";
import { AppModule } from "./app.module";
import { BackgroundService } from "nativescript-background-service";

// initialize the nativescript-background-service plugin
BackgroundService.init({
  androidUseIntentService: false,
  androidServiceInterval: 0,
  androidServiceStartSticky: true,
  androidAutoStartAfterBoot: false,
  iosAutoStartInForeground: false,
  iosRunInForeground: true,
  iosRunInBackground: true,
  iosRunInBackgroundFetch: false,
  iosBackgroundRequestMoreTime: true
});

platformNativeScriptDynamic().bootstrapModule(AppModule);
```

Listing 14: Beispielhafte Konfiguration des nativescript-background-service-Plugins

7.4.2 Implementierung - Android

Auf Android-Geräten kann das Plugin Logik innerhalb von nativen `Service`-Instanzen [151] ausführen. Solche Hintergrund-Services können unabhängig von einer Applikation ausgeführt werden. Die Implementierung eines solchen `Service`-Objektes erfolgte in TypeScript. Eine gekürzte Darstellung des Quelltextes ist im Anhang in Listing 19 zu finden. Hierbei wird innerhalb der `onStartCommand()`-Methode die benutzerdefinierte Logik ausgeführt. Über die `onDestroy()`-Methode können abschließende Aufgaben ausgeführt werden. Die `Service`-Instanz verwendet einen `PARTIAL_WAKE_LOCK` [178], um auch im Hintergrund stetig Rechenzeit vom Betriebssystem gewährt zu bekommen. Das Starten und Stoppen des Hintergrund-Services wird über Intents [179] realisiert. Ein vollautomatischer Start nach dem Hochfahren eines Android-Gerätes erfolgt über einen `BroadcastReceiver` [167]. Die `Service`-Implementierung in Listing 19 wird auch zur Umsetzung des Intervall-Services genutzt. Dafür wird der `AlarmManager` von Android [180] eingesetzt, der periodisch die `Service`-Instanz startet. Der Service beendet sich im Intervall-Modus nach einer konfigurierbaren Zeitspanne selbst.

7.4.3 Implementierung - iOS

Im Gegensatz zu Android kann auf iOS keine Hintergrundausführung unabhängig von einer Applikation stattfinden. Sowohl langlaufende Hintergrundaufgaben als auch der Background-Fetch-Modus werden nur ausgeführt, wenn die dazugehörige Applikation geöffnet und anschließend minimiert wurde (vgl. Unterabschnitt 6.2.5).

Die Implementierung der iOS-Hintergrundmodi basiert auf Ereignissen, die während der Laufzeit einer Applikation von NativeScript über das native `NSNotificationCenter` [181] bereitgestellt werden. NativeScript bietet Schnittstellen an, um Callbacks für solche Ereignisse zu registrieren. Diese wurden verwendet, um Logik auszuführen, wenn ein bestimmtes Ereignis eintritt. So kann zum Beispiel die Hintergrundausführung gestartet werden, wenn das `UIApplicationDidEnterBackgroundNotification`-Ereignis empfangen wird. In nativen iOS-Applikationen können solche Ereignisse über eine Implementierung des `UIApplicationDelegate`-Protokolls [182] behandelt werden. Dabei kann jedoch nur eine Implementierung bei der Applikation registriert werden. Der verwendete Benachrichtigungsmechanismus von NativeScript umgeht eine Implementierung dieses Protokolls. Dadurch stehen Plugins, die diese Ereignisse konsumieren wollen, nicht in gegenseitigem Konflikt zueinander. Zur Implementierung des Background-Fetch-Modus musste jedoch das `UIApplicationDelegate`-Protokoll im Rahmen dieses Plugins implementiert werden. Die Aktivierung dieses Hintergrundmodus konnte zum Zeitpunkt der Implementierung nicht über das `NSNotificationCenter` abgefangen werden, da keine Benachrichtigung für dieses Ereignis angeboten wurde. Alle anderen vorgestellten Hintergrundmodi (vgl. Unterabschnitt 6.2.5) konnten dahingegen über den Benachrichtigungsmechanismus implementiert werden. Zusätzlich kann die Überwachung von signifikanten Ortsänderungen über das Plugin gestartet oder gestoppt werden, wobei vom Entwickler beachtet werden sollte, dass dieser Modus auch nach dem Beenden einer Applikation weiterläuft. Insgesamt muss die Implementierung bei iOS nur zwischen Vordergrund, Hintergrund, Background-Fetch und signifikanter Ortsänderung unterscheiden. Die Nutzung dieser Hintergrundmodi erfordert vom Applikationsentwickler allerdings die Beachtung einiger Aspekte. Diese Faktoren werden im Folgenden für die einzelnen Hintergrundmodi vorgestellt.

Background-Fetch

Zur Verwendung dieses Hintergrundmodus muss in der `Info.plist` der NativeScript-Applikation der `UIBackgroundModes`-Wert `fetch` gesetzt werden [154]. Zudem benötigt die Applikation die Erlaubnis, die Hintergrundaktualisierung nutzen zu dürfen.

Langlaufende Hintergrundaufgabe - Bluetooth LE:

Es ist möglich, eine Hintergrundaufgabe unbegrenzt auszuführen, wenn innerhalb der `Info.plist` der NativeScript-Applikation der `UIBackgroundModes`-Schlüssel `bluetooth-central` gesetzt und im Hintergrund mit Bluetooth-Geräten kommuniziert wird [154]. Hierbei wurde jedoch beobachtet, dass das Laufzeitverhalten der Applikation nicht einer unbegrenzt langlaufenden Hintergrundaufgabe entspricht. Es war festzustellen, dass der Applikation in diesem Hintergrundmodus nur Rechenzeit zugewiesen wird, wenn auch tatsächlich eine Verbindung zu einem Bluetooth-Gerät besteht. Wenn der Modus ohne bestehende Bluetooth-Verbindung gestartet wurde oder die Verbindung zu einem Gerät abbricht, wird die Ausführung pausiert. Dies wurde unter Verwendung des Kontext-Provider nativescript-ca-provider-heart rate getestet. Um dieses Laufzeitverhalten zu behandeln, musste das automatische Verbindungsmanagement des Kontext-Providers angepasst werden. So gilt es zum Beispiel beim Wechsel in diesen Hintergrundmodus zu beachtet, ob bereits eine Bluetooth-Verbindung besteht oder nicht. In letzterem Fall muss das automatische Verbindungsmanagement neu gestartet werden, damit der Hintergrundmodus auf neue Bluetooth-Verbindungen reagiert und Rechenzeit zur Verfügung stellt. Auf zwei von drei Test-Geräten konnte eine weitere Limitierung festgestellt werden. So musste ein Bluetooth-Gerät bei jeder Verwendung der Testapplikation erstmalig im Vordergrund verbunden werden. Wenn die Applikation in den Hintergrund wechselte und während der bisherigen Laufzeit noch keine Bluetooth-Verbindung bestand, konnte im Hintergrund keine Verbindung hergestellt werden. Ein Verbindungsaufbau ist dann erst wieder möglich, wenn die Applikation in den Vordergrund wechselt. Dieses Verhalten sollte bei der Anwendungsentwicklung berücksichtigt werden.

Langlaufende Hintergrundaufgabe - Ortung

Wenn im Hintergrund Positionsaktualisierungen empfangen werden, ist es möglich, eine Hintergrundaufgabe unbegrenzt auszuführen. Dafür muss in der `Info.plist` der NativeScript-Applikation der `UIBackgroundModes`-Wert `location` gesetzt werden [154]. Im Gegensatz zur Bluetooth-basierten Hintergrundausführung wird hier tatsächlich unbegrenzt Laufzeit gewährt. Dieses Hintergrundverhalten muss vom Nutzer explizit genehmigt werden. Zum Auslesen und Abfragen der entsprechenden Berechtigung bietet das Plugin Methoden an.

Zudem wurde im Rahmen der Implementierung das nativescript-geolocation-Plugin angepasst. Dies war notwendig, da das Plugin nicht für die Ausführung in diesem Hintergrundmodus geeignet war. Damit in diesem Modus Positionsaktualisierungen von der Applikation empfangen werden können, muss der `allowsBackgroundLocationUpdates`-Parameter der `CLLocationManager`-Instanz [183] auf `true` gesetzt werden. Des Weiteren kann die Ausführung des Hintergrundmodus vom Betriebssystem aus Energiespargründen unterbrochen werden. Zur Verhinderung dieses Verhaltens kann der Parameter `pausesLocationUpdatesAutomatically` der verwendeten `CLLocationManager`-Instanz auf `false` gesetzt werden. Im Rahmen dieser Arbeit wurden diese Parameter als Konfigurationsparameter in das nativescript-geolocation-Plugin integriert. Diese werden vom Kontext-Provider nativescript-ca-provider-geolocation verwendet (vgl. Tabelle B.2).

Bei der Verwendung dieses Hintergrundmodus muss zusätzlich beachtet werden, dass die Positionsaktualisierungen im Vordergrund angefragt werden müssen. Anderenfalls gewährt das Betriebssystem nur 10 bzw. 180 Sekunden Rechenzeit, nachdem die Applikation minimiert wurde. Der langlaufende Hintergrundmodus kann dementsprechend nicht verwendet werden, wenn die Ortung erst im Hintergrund gestartet wird.

7.4.4 Kommunikation mit Komponenten im Hintergrund

Zur plattformunabhängigen Kommunikation zwischen Komponenten einer NativeScript-Applikation wurde das Plugin nativescript-native-event-bus entwickelt. Dieses abstrahiert eine `string`-basierte Kommunikation über native Schnittstellen und kann insbesondere

für die Kommunikation zwischen Benutzeroberfläche und Hintergrundausführung eingesetzt werden. Auf Android läuft die Kommunikation über `Intent`-Objekte [179] und `BroadcastReceiver` [167] ab. Bei iOS-Geräten kommt das `NSNotificationCenter` [181] zum Einsatz. Zur Interprozesskommunikation können auf Android auch öffentliche, geräteweite `Broadcast`-Übertragungen verwendet werden. Dies ist bei der Verwendung des nativescript-background-service-Plugins notwendig, da die Hintergrundausführung von Android in einem eigenen Prozess läuft. Listing 15 zeigt vereinfacht, wie das Plugin verwendet werden kann.

```
1  import { NativeEventBus } from "nativescript-native-event-bus";
2
3  let messageId = "greeting";
4
5  // send message
6  NativeEventBus.post(messageId, "hello");
7
8  // receive message
9  NativeEventBus.subscribeForEvent(messageId,
10   (message : string) => {
11      console.log("new message: " + message);
12   });
```

Listing 15: Beispielhafte Verwendung des nativescript-native-event-bus-Plugins

7.5 Verwendung des Frameworks

Die Verwendung des Frameworks in einer NativeScript-Applikation erfolgt über die entwickelten Plugins. Im ersten Schritt müssen die Berechtigungen zum Zugriff auf die jeweiligen nativen APIs angefordert werden. Dies ermöglichen statische Methoden der Kontext-Provider, was in Listing 16 beispielhaft dargestellt wird. Dieser Schritt sollte aus Gründen der Benutzerfreundlichkeit im Vordergrund einer Applikation erfolgen.

```
1 AmbientNoiseContextProvider.requestRecordAudioPermission();
2 GeolocationContextProvider.enableLocationRequest(true);
3 ActivityContextProvider.requestAuthorization();
```

Listing 16: Verwendung des Frameworks: Berechtigungen anfordern

Die folgenden Schritte zur Verwendung des Frameworks können sowohl im Vordergrund als auch im Hintergrund einer Applikation angewendet werden und sind als beispielhafte Implementierung in Listing 17 dargestellt:

1. **Kontext-Prozessor:** Zunächst muss ein Kontext-Prozessor instanziiert werden. Bei diesem können Kontextereignisse und Kontext-Provider registriert werden.
2. **Kontext-Provider:** Anschließend können beliebig viele Kontext-Provider instanziiert, konfiguriert und beim Kontext-Prozesser registriert werden.
3. **Kontextereignisse:** Über die Kontext-Queries der Kontext-Provider werden logische Ausdrücke bereitgestellt. Mit Methoden des Kontext-Prozessors können solche zu Regelbäumen verknüpft werden. Diese Regelbäume beschreiben Kontextbedingungen und werden zur Definition von Kontextereignissen verwendet. Zudem wird für Kontextereignisse ein Callback registriert, der ausgeführt wird, wenn die Kontextbedingungen erfüllt sind.
4. **Start der Kontextverarbeitung:** Über die Methode `start()` des Kontext-Prozessors wird die Kontextverarbeitung aller Kontext-Provider gestartet.

```
import { ContextAwarenessProcessor, BooleanExpression }
  from "nativescript-context-awareness";
import { AmbientNoiseContextProvider }
  from "nativescript-ca-provider-ambient-noise";
import { GeolocationContextProvider }
  from "nativescript-ca-provider-geolocation";
import { ActivityContextProvider }
 from "nativescript-ca-provider-activity";

export class ContextProcessing {

  private contextProcessor : ContextAwarenessProcessor;
  private activityProvider : ActivityContextProvider;
  private heartRateProvider : HeartRateContextProvider;

  public start() : void {
    // 1. init context processor
    this.contextProcessor = new ContextAwarenessProcessor();

    // 2. init context provider
    this.activityProvider = new ActivityContextProvider();
    this.heartRateProvider = new HeartRateContextProvider();

    this.activityProvider.config({
      minimumConfidence : 70
    });

    this.contextProcessor.registerContextProvider(this.activityProvider);
    this.contextProcessor.registerContextProvider(this.heartRateProvider);

    // 3. define context events
    let condition : BooleanExpression = ContextAwarenessProcessor.AND(
      this.activityProvider.isRunning(),
      this.heartRateProvider.bpmGreaterOrEqualThan(180)
    );

    this.contextProcessor.addEvent(condition, () => {
      console.log("How about taking a break?");
    });

    // 4. start context processing
    this.contextProcessor.start();
  }
}
```

Listing 17: Verwendung des Frameworks

7.6 Beispielapplikation

Zum Testen des Frameworks und zu Demonstrationszwecken wurde eine beispielhafte Applikation entwickelt, die das Framework im Vordergrund und im Hintergrund einsetzt. Wie in Abbildung 7.4 zu sehen ist, kann die Kontextverarbeitung vom Nutzer über einen Schalter gestartet und gestoppt werden. Zudem werden aktuelle Kontextdaten symbolisch in einer Übersicht dargestellt.

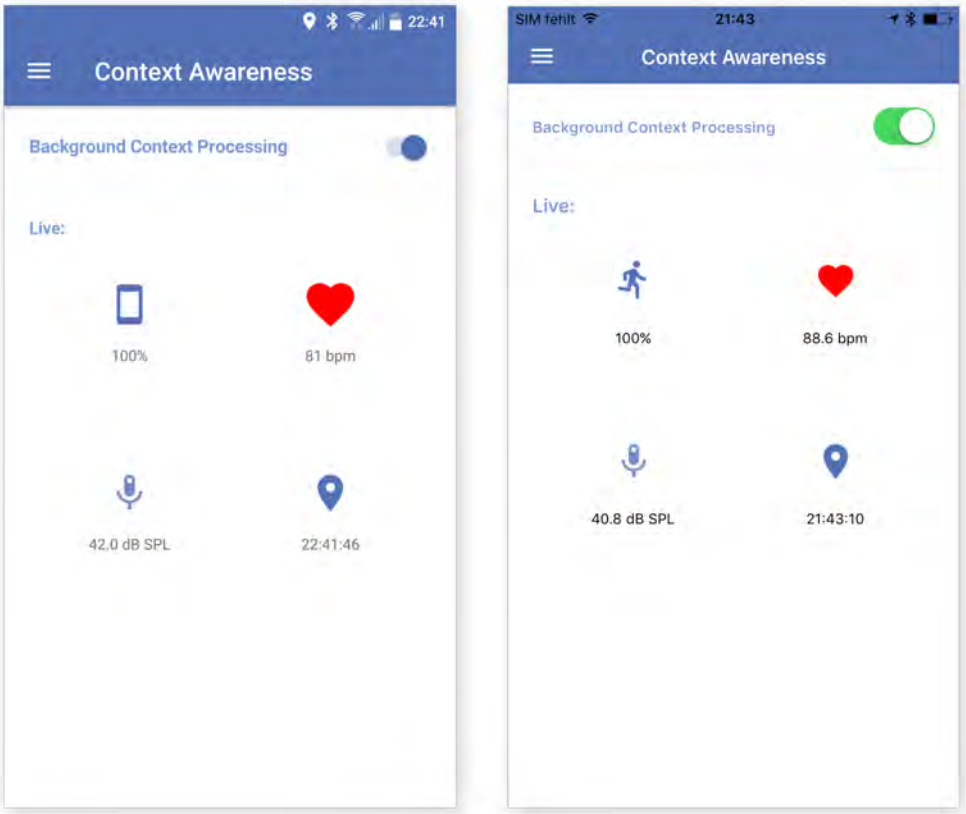

Abbildung 7.4: Beispielapplikation - Startansicht (links Android, rechts iOS)

Zur Kontextverarbeitung im Hintergrund wird auf Android-Geräten ein Hintergrund-Service und auf iOS eine langlaufende Hintergrundaufgabe verwendet. Die Applikation ermöglicht die Überwachung von Kontextdaten, die Verwaltung von Herzfrequenzmessgeräten und die Kalibrierung des Mikrophons. Zusätzlich wird eine Benutzeroberfläche

bereitgestellt, über die Kontextereignisse definiert werden können. Dabei kann für jeden im Rahmen dieser Arbeit entwickelten Kontext-Provider ein Kontextereignis angegeben werden. Wenn die definierten Kontextbedingungen für solch ein Ereignis erfüllt sind, wird eine Benachrichtigung auf dem mobilen Endgerät angezeigt. Der Zugriff auf die erwähnten Funktionalitäten ist über eine native Sidedrawer-Navigation möglich (vgl. Abbildung 7.5).

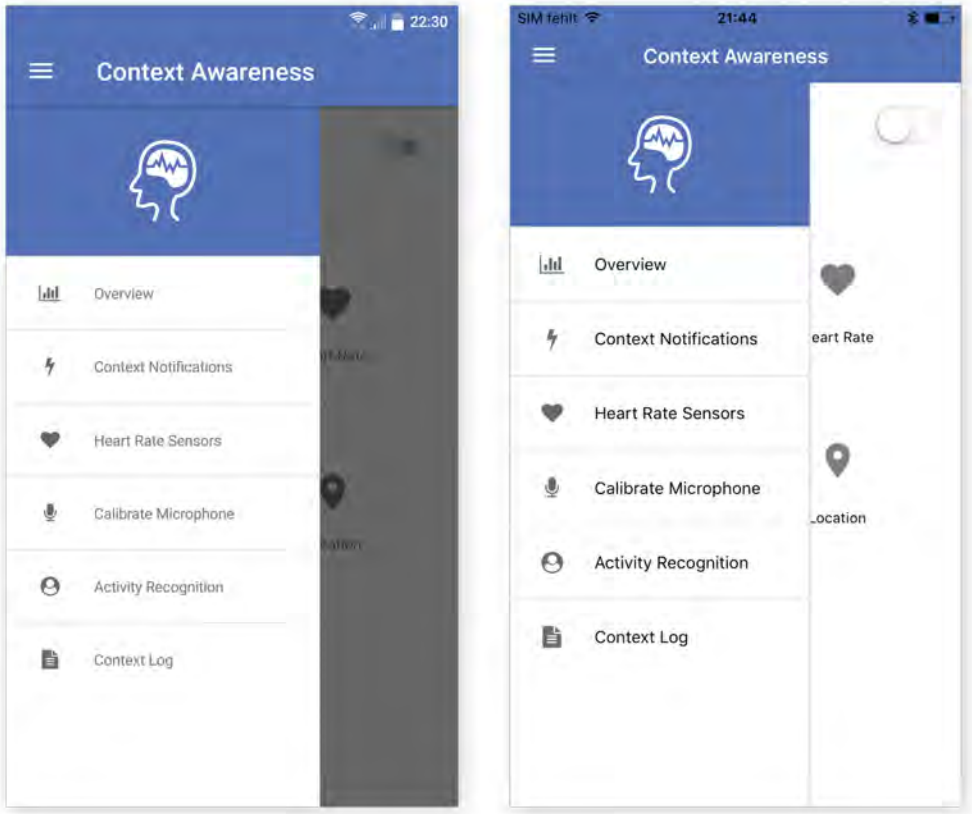

Abbildung 7.5: Beispielapplikation - Navigation (links Android, rechts iOS)

7.6.1 Darstellung von Kontextinformationen

Die Erfassung von Kontextinformationen kann über ein Ereignisprotokoll nachvollzogen werden. Dies soll Applikationsentwickler beim Testen des Frameworks unterstützen.

Sobald einem Kontext-Provider neue Kontextdaten zur Verfügung stehen, werden diese in einer SQLite-Datenbank abgespeichert. Darüber kann auch die Kontextsensitivität einer Applikation protokolliert werden, während sich diese im Hintergrund befindet. Die Log-Einträge werden über eine Benutzeroberfläche bereitgestellt (vgl. Abbildung 7.6). Zudem werden erkannte Aktivitäten des nativescript-ca-provider-activity-Moduls separat aufgeschlüsselt und über eine eigene Benutzeroberfläche dargestellt. Diese Darstellung ist in Abbildung 7.7 zu sehen.

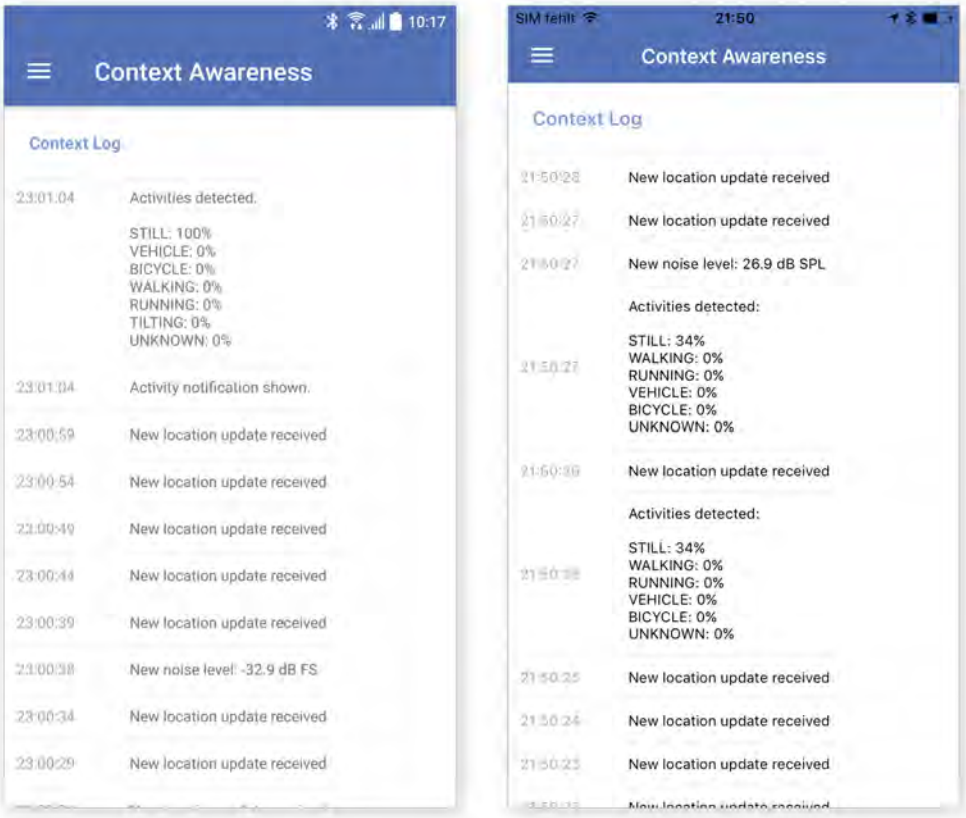

Abbildung 7.6: Beispielapplikation - Ereignisprotokollierung (links Android, rechts iOS)

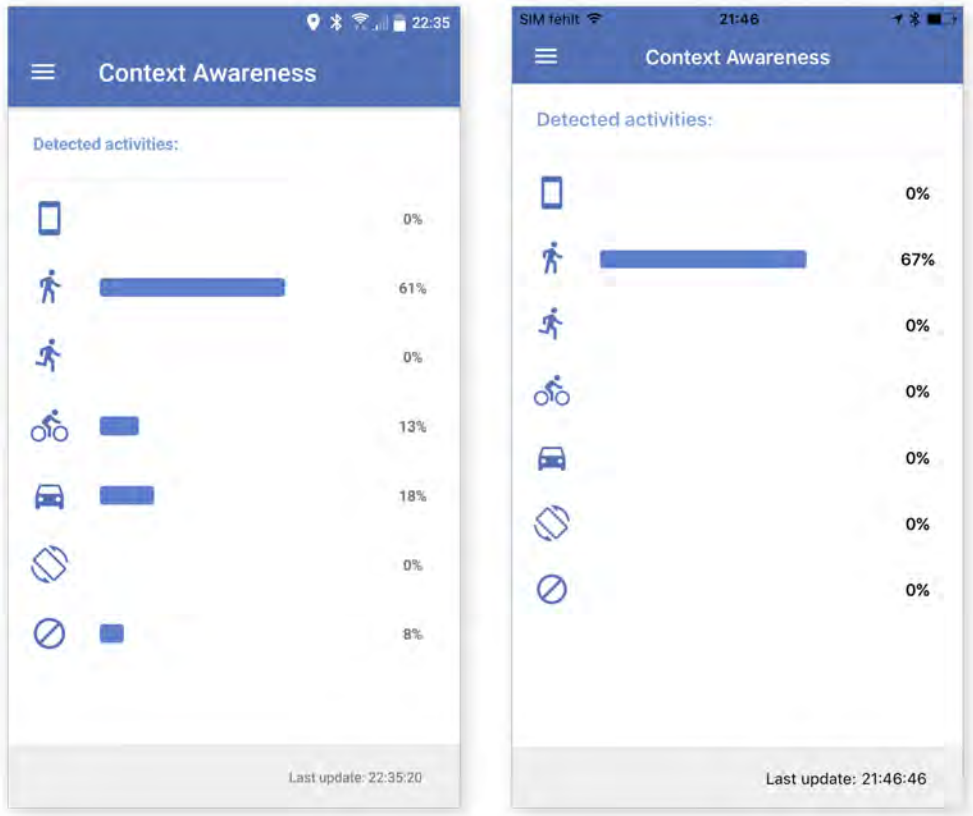

Abbildung 7.7: Beispielapplikation - Aktivitätenüberwachung (links Android, rechts iOS)

7.6.2 Verwaltung von Herzfrequenzmessgeräten

Der Kontext-Provider nativescript-ca-provider-heart-rate (vgl. Unterabschnitt 7.3.2) bietet zur Verwaltung von Bluetooth-Herzfrequenzmessgeräten diverse Schnittstellen an. In Abbildung 7.8 ist dargestellt, wie diese für die vorgestellte Beispielapplikation genutzt werden. Über die Benutzeroberfläche können Herzfrequenzmessgeräte gesucht, gespeichert und gelöscht werden. Zusätzlich kann das Standardmessgerät festgelegt werden, zu dem der Kontext-Provider automatisch eine Verbindung herstellt.

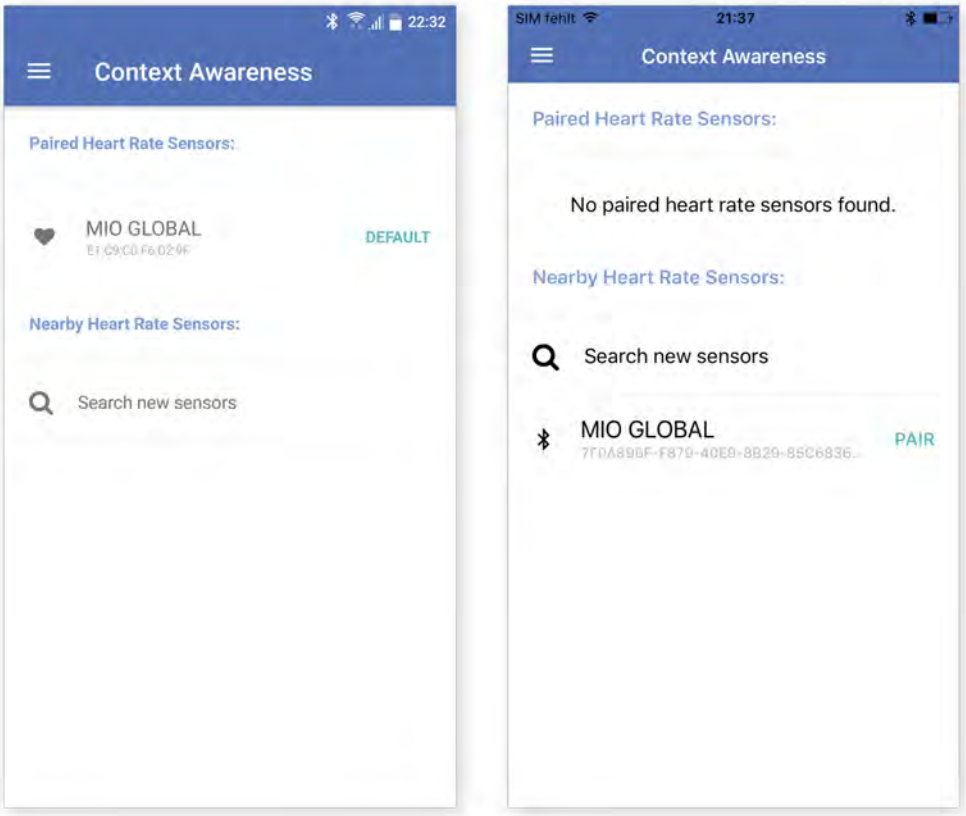

Abbildung 7.8: Beispielapplikation - Verwaltung von Herzfrequenzmessgeräten (links Android, rechts iOS)

7.6.3 Mikrophonkalibrierung

Die Verwendung des Kontext-Providers nativescript-ca-provider-ambient-noise (vgl. Unterabschnitt 7.3.4) ermöglicht, die Umgebungslautstärke relativ zu einer ruhigen Umgebung zu bestimmen. Hierfür muss der Nutzer eine Kalibrierung durchführen, bei der eine subjektiv ruhige Umgebung über das Mikrophon gemessen wird. Für diesen Zweck wurde eine Benutzeroberfläche bereitgestellt, die in Abbildung 7.9 dargestellt ist.

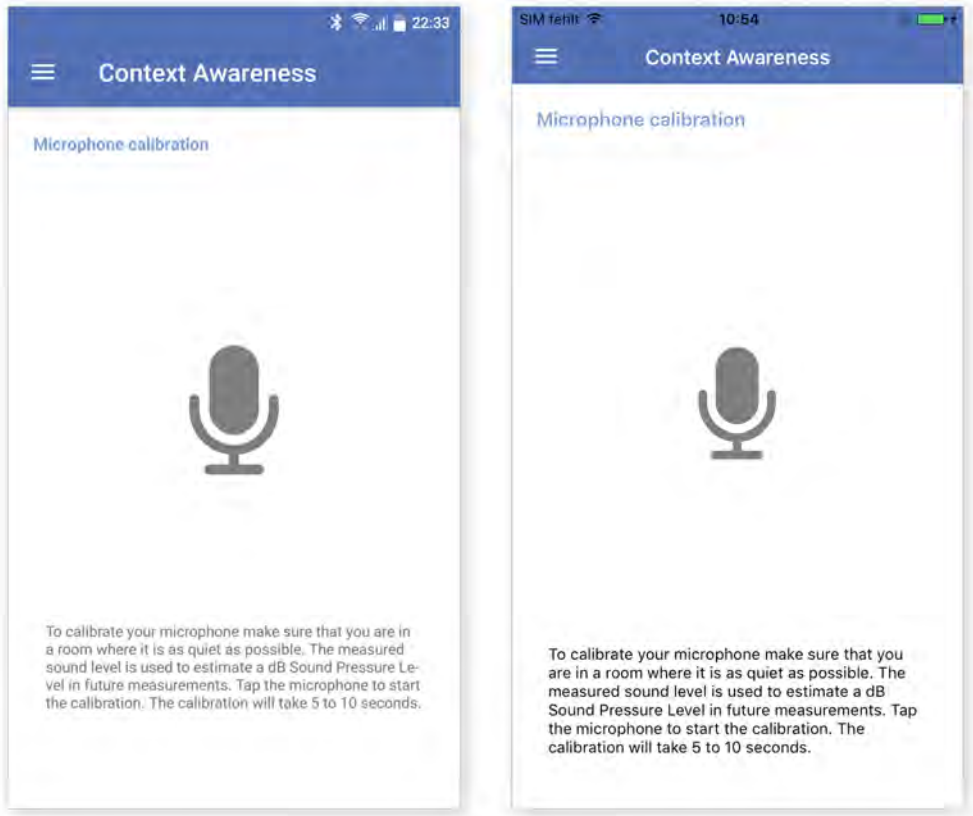

Abbildung 7.9: Beispielapplikation - Mikrophonkalibrierung (links Android, rechts iOS)

7.6.4 Kontextbenachrichtigungen

Kontextbenachrichtigungen werden auf dem Smartphone angezeigt, wenn gewisse Kontextbedingungen erfüllt sind. Es ist möglich, für die vier im Rahmen dieser Arbeit entwickelten Kontext-Provider Kontextbedingungen zu definieren. Abbildung 7.10 zeigt die Konfiguration von Benachrichtigungen, die vom Ort oder von der ausgeübten Aktivität abhängen. Des Weiteren können Kontextbenachrichtigungen definiert werden, die von der Herzfrequenz des Nutzers oder der Umgebungslautstärke abhängen. Zur Betrachtung von Kontextbedingungen über einen längeren Zeitraum kann eine Zeitspanne in Sekunden angegeben werden. Abbildung 7.11 zeigt beispielhafte Kontextbenachrichtigungen, für die Ereignisse definiert wurden, die von der gemessenen Herzfrequenz und der Umgebungslautstärke abhängen.

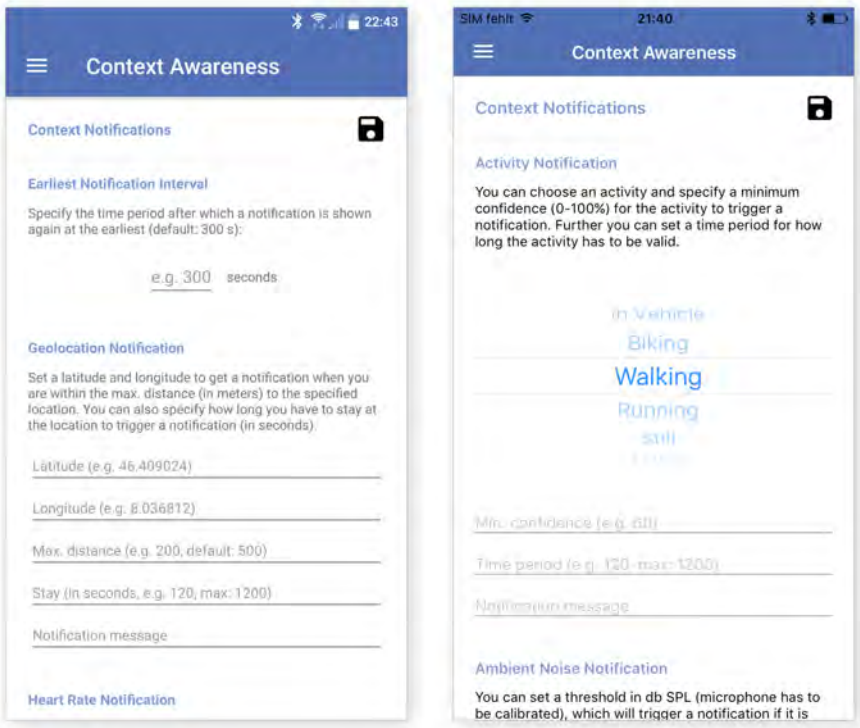

Abbildung 7.10: Beispielapplikation - Konfiguration von Kontextbenachrichtigungen (links Android, rechts iOS)

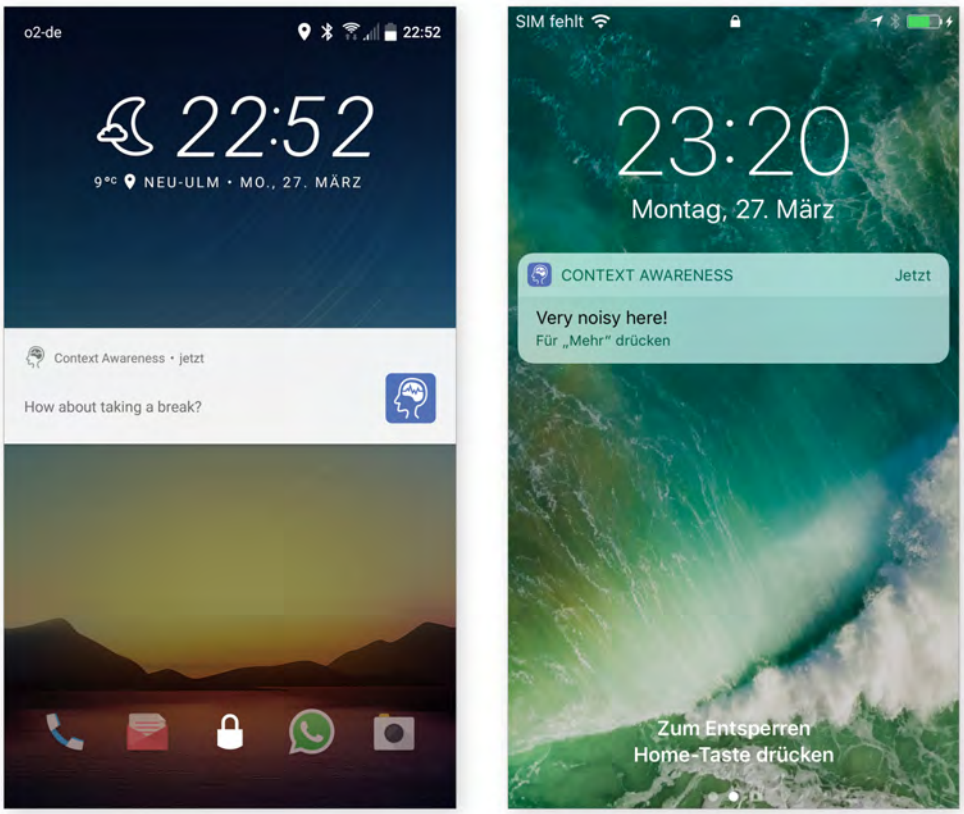

Abbildung 7.11: Beispielapplikation - Kontextbenachrichtigungen (links Android, rechts iOS)

7.7 Anforderungsabgleich

Im Folgenden werden die in Abschnitt 6.1 gestellten Anforderungen mit der Umsetzung abgeglichen.

7.7.1 Funktionale Anforderungen

Wie der Tabelle 7.1 entnommen werden kann, wurde der Großteil der funktionalen Anforderungen erfüllt. Es wurde ein Kontext-Provider konzeptioniert, der Termine aus den Kalendern der mobilen Endgeräte in Kontextinformationen übersetzt. Dabei wurde festgestellt, dass die Kalenderdaten keine Informationen enthalten, die sich direkt in Kontextinformationen übersetzen lassen. Beispielsweise konnte keine Terminpriorität plattformübergreifend ausgelesen werden. Ohne aufwändige Textanalyseverfahren ist eine sinnvolle Verwendung der Kalenderdaten im Rahmen des hier vorgestellten Frameworks nicht möglich. Aufgrund dieser Einschränkungen wurde auf die Implementierung dieses Kontext-Providers verzichtet.

ID	Bezeichnung	Status
FA-01	Logikbasierte Bereitstellung von Kontextinformationen	Erfüllt
FA-02	Definition logikbasierter Kontextbedingungen	Erfüllt
FA-03	Definition von Kontextereignissen	Erfüllt
FA-04	Modularisierte Kontexterfassung, Kontextvorverarbeitung und Bereitstellung von Kontextinformationen	Erfüllt
FA-05	Zentrale Evaluation und Auslösung von Kontextereignissen	Erfüllt
FA-06	Zeitliche Validität von Kontextinformationen	Erfüllt
FA-07	Persistenz von Kontextinformationen	Erfüllt

FA-08	Kommunikation zwischen Kontext-Providern	Erfüllt
FA-09	Kontext-Provider - Ort	Erfüllt
FA-10	Kontext-Provider - Aktivitäten	Erfüllt
FA-11	Kontext-Provider - Herzfrequenz	Erfüllt
FA-12	Kontext-Provider - Termine	Nicht erfüllt
FA-13	Kontext-Provider - Umgebungslautstärke	Erfüllt
FA-14	Bereitstellung von rohen Kontextinformationen	Erfüllt
FA-15	Hintergrundausführung	Erfüllt
FA-16	Beispielapplikation	Erfüllt

Tabelle 7.1: Anforderungsabgleich - Funktionale Anforderungen

7.7.2 Nichtfunktionale Anforderungen

In der folgenden Tabelle 7.2 werden die nichtfunktionalen Anforderungen mit der Umsetzung abgeglichen. Die nichtfunktionalen Anforderungen beschreiben Qualitätsmerkmale des Frameworks und wurden größtenteils erfüllt. Über Parallelisierungsansätze wurden bei der Implementierung Leistungsaspekte berücksichtigt. Im Rahmen weiterer Arbeiten kann die Effizienz der Evaluation von Kontextereignissen weiter optimiert werden.

ID	Bezeichnung	Status
NFA-01	Wartbarkeit, Änderbarkeit und Erweiterbarkeit	Erfüllt
NFA-02	Plattformunabhängigkeit	Erfüllt
NFA-03	Konfigurierbarkeit	Erfüllt
NFA-04	Leistung und Effizienz	Teilweise erfüllt

Tabelle 7.2: Anforderungsabgleich - Nichtfunktionale Anforderungen

8

Zusammenfassung und Ausblick

Abschließend erfolgt in diesem Kapitel eine zusammenfassende Betrachtung der vorliegenden Arbeit. Zudem wird ein Ausblick auf mögliche Weiterentwicklungen gegeben und es werden Aspekte beschrieben, die bei der Entwicklung kontextsensitiver Applikationen berücksichtigt werden sollten.

8.1 Zusammenfassung

In der vorliegenden Arbeit wurden verschiedene Ansätze der mobilen Anwendungsentwicklung verglichen. Es wurde gezeigt, wie die Entwicklung mobiler Webanwendungen sowie hybrider Applikationen im Gegensatz zur nativen Applikationsentwicklung plattformunabhängig erfolgen kann. Des Weiteren wurde dargestellt, wie Cross-Platform-Ansätze die Vorteile nativer und hybrider Applikationsentwicklung vereinen. Im Anschluss erfolgte die Betrachtung verschiedener Cross-Platform-Frameworks. Dabei wurde festgestellt, dass sich das NativeScript-Framework durch den direkten Zugriff auf native Schnittstellen besonders für die Entwicklung kontextsensitiver Applikationen eignet.

Um die Cross-Platform-Entwicklung kontextsensitiver Applikationen zu vereinfachen, wurde ein Framework für NativeScript-Applikationen entwickelt. Das Framework baut auf einer logikbasierten Kontextrepräsentation und auf einem regelbasierten Schlussfolgerungsmechanismus auf. Dadurch kann eine hierarchische Strukturierung von Kontextinformationen erfolgen. Zudem ist die Implementierung und Verwendung des Frameworks so nicht auf den Einsatz zusätzlicher Technologien angewiesen. Das Framework stellt für Applikationsentwickler verschiedene Komponenten zur Verfügung. Diese ermöglichen die Konfiguration und Steuerung der Kontexterfassung und Kontextverarbeitung.

Zudem können über diese Komponenten Kontextereignisse definiert werden. Diese Ereignisse werden vom Framework automatisch evaluiert und dabei ausgelöst, falls die entsprechenden Kontextbedingungen erfüllt sind.

Es wurde gezeigt, wie das Plugin-System von NativeScript im Rahmen des Frameworks verwendet werden kann, um die Kontexterfassung und Bereitstellung von Kontextinformationen zu modularisieren. Dadurch ist es für NativeScript-Entwickler möglich, das Framework mit eigenen Modulen unter Verwendung bekannter Technologien zu erweitern. Es wurden vier beispielhafte Module entwickelt, um den Ort, die Herzfrequenz des Nutzers, ausgeübte Aktivitäten und die Umgebungslautstärke als Kontextinformationen bereitzustellen.

Zudem wurden verschiedene native Hintergrundmodi auf Android und iOS betrachtet. Es wurde gezeigt, welche dieser Modi für kontextsensitive Applikationen sinnvoll eingesetzt werden können. Im Rahmen der Implementierung wurde ein NativeScript-Modul entwickelt, welches die Ausführung beliebiger Logik innerhalb dieser Hintergrundmodi ermöglicht. Das Modul abstrahiert die verschiedenen Arten der Hintergrundausführung und kann von Applikationsentwicklern plattformunabhängig eingesetzt werden. In Kombination mit dem vorgestellten Framework können sich Applikationen dadurch auch im Hintergrund kontextsensitiv verhalten.

8.2 Ausblick

Dieser Abschnitt stellt mögliche Erweiterungen für das Framework vor. Zudem wird auf anwendungsrelevante Aspekte eingegangen, die bei der Verwendung des Frameworks berücksichtigt werden sollten.

Die Bereitstellung weiterer Kontextinformationen kann für das vorgestellte Framework durch die Implementierung zusätzlicher Kontext-Provider erreicht werden. Dabei ist der Einsatz unterschiedlichster Datenquellen möglich. So können beispielsweise Kontext-Provider entwickelt werden, die Daten über das Internet beziehen, um Kontextinformationen über das Wetter oder die Luftverschmutzung bereitzustellen. Neben der vorgestellten Herzfrequenzerfassung, könnten über andere externe Geräte weitere Daten erfasst wer-

den, die für die Gesundheitsforschung relevant sind. Hierfür eignen sich beispielsweise tragbare Geräte, welche die Atmung oder den Blutzucker überwachen. Die Erweiterung des Frameworks über neue Kontext-Provider stellt eine interessante Möglichkeit dar, die in Zukunft weiterverfolgt werden kann.

Der regelbasierte Schlussfolgerungsmechanismus des Frameworks erfordert a-priori-Wissen zur Spezifizierung von Situationen und Kontextbedingungen. In zukünftigen Arbeiten könnte untersucht werden, inwiefern der Einsatz anderer Schlussfolgerungsmechanismen möglich ist. So könnten zum Beispiel Verfahren des maschinellen Lernens eingesetzt werden, welche auch mit unbekannten Situationen umgehen können.

Das im Rahmen dieser Arbeit vorgestellte Framework arbeitet gerätezentriert. Dabei wird mit der Annahme gearbeitet, dass die benötigten Kontextdaten über ein einzelnes Gerät erfasst werden können. Dies kann für viele Anwendungsfälle ausreichend sein. Heutzutage kommen neben Smartphones auch oft andere mobile Endgeräte, wie zum Beispiel Wearables, zum Einsatz. Die Kontextbestimmung über mehrerer Geräte ist bei dem vorgestellten Konzept nicht vorgesehen. Eine verteilte Kontexterkennung ermöglicht jedoch nicht nur eine nutzerzentrierte Kontextbestimmung, sondern kann auch nach dem Prinzip des Mobile-Crowdsourcings zur Generierung weiterer Kontextinformationen genutzt werden. Solche verteilten Systeme könnten im Rahmen zukünftiger Arbeiten in Verbindung mit dem Framework untersucht werden.

Bei der Entwicklung kontextsensitiver Applikationen sollten Aspekte des Datenschutzes beachtet werden. Dies gilt besonders bei netzwerkbasierten Anwendungen, wie zum Beispiel bei Mobile-Crowdsourcing-Applikationen. Kontextinformationen sind hochsensible Daten. Daher sollten Nutzer kontextsensitiver Applikationen informiert werden, welche Daten, wann, wie und zu welchem Zweck genutzt werden. Wenn Kontextinformationen netzwerkbasiert übertragen werden, müssen Aspekte wie Anonymisierung und Netzwerksicherheit berücksichtigt werden. Die Bereitstellung von Kontextdaten über das Netzwerk sollte für den Nutzer kontrollierbar sein. Zudem sollte das adaptive Verhalten einer kontextsensitiven Applikation für einen Nutzer verständlich sein. Unklare Verhaltensweisen einer Applikation können beim Nutzer ein Gefühl von Kontrollverlust

erzeugen. Um die beschriebene Transparenz zu gewährleisten, können entsprechende Erklärungen in einer Applikation bereitgestellt werden.

Literaturverzeichnis

[1] Charlotte Crampton: Germany: The Rise and Growth of App Usage. https://www.comscore.com/Insights/Data-Mine/Germany-The-Rise-and-Growth-of-App-Usage (2017) Online, abgerufen am 26.03.2017.

[2] Dave Barth: The bright side of sitting in traffic: Crowdsourcing road congestion data. https://googleblog.blogspot.de/2009/08/bright-side-of-sitting-in-traffic.html (2009) Online, abgerufen am 15.03.2017.

[3] Tinnitus Research Initiative: Track your Tinnitus. https://www.trackyourtinnitus.org (2017) Online, abgerufen am 15.03.2017.

[4] Tamilin, A., Carreras, I., Ssebaggala, E., Opira, A., Conci, N.: Context-aware mobile crowdsourcing. In: Proceedings of the 2012 ACM Conference on Ubiquitous Computing. UbiComp '12, New York, NY, USA, ACM (2012) 717–720

[5] Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput. **5** (2001) 4–7

[6] Howe, J.: Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business. BusinessPro collection. Three Rivers Press (2009)

[7] Guo, B., Yu, Z., Zhang, D., Zhou, X.: From participatory sensing to mobile crowd sensing. CoRR **abs/1401.3090** (2014)

[8] Ma, H., Zhao, D., Yuan, P.: Opportunities in mobile crowd sensing. IEEE Communications Magazine **52** (2014) 29–35

[9] Appcelerator Inc.: The Appcelerator Platform. http://www.appcelerator.com/ (2017) Online, abgerufen am 09.03.2017.

[10] Schobel, J., Schickler, M., Pryss, R., Reichert, M.: Process-driven data collection with smart mobile devices. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 347–362

[11] Schobel, J., Schickler, M., Pryss, R., Maier, F., Reichert, M.: Towards process-driven mobile data collection applications: Requirements, challenges, lessons learned. In: 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps. (2014) 371–382

[12] Schobel, J., Ruf-Leuschner, M., Pryss, R., Reichert, M., Schickler, M., Schauer, M., Weierstall, R., Isele, D., Nandi, C., Elbert, T.: A generic questionnaire framework supporting psychological studies with smartphone technologies. In: XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference. (2013) 69–69

[13] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: Towards flexible mobile data collection in healthcare. In: 29th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2016). (2016) 181–182

[14] Schobel, J., Pryss, R., Wipp, W., Schickler, M., Reichert, M.: A mobile service engine enabling complex data collection applications. In: 14th International Conference on Service Oriented Computing (ICSOC 2016). Number 9936 in LNCS (2016) 626–633

[15] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: A configurator component for end-user defined mobile data collection processes. In: Demo Track of the 14th International Conference on Service Oriented Computing (ICSOC 2016). (2016)

[16] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: A lightweight process engine for enabling advanced mobile applications. In: 24th International Conference on Cooperative Information Systems (CoopIS 2016). Number 10033 in LNCS, Springer (2016) 552–569

[17] Schickler, M., Pryss, R., Reichert, M., Heinzelmann, M., Schobel, J., Langguth, B., Probst, T., Schlee, W.: Using wearables in the context of chronic disorders - results of a pre-study. In: 29th IEEE Int'l Symposium on Computer-Based Medical Systems. (2016) 68–69

[18] Schobel, J., Pryss, R., Schickler, M., Ruf-Leuschner, M., Elbert, T., Reichert, M.: End-user programming of mobile services: Empowering domain experts to implement mobile data collection applications. In: 5th IEEE International Conference on Mobile Services (MS 2016), IEEE Computer Society Press (2016) 1–8

[19] Schickler, M., Schobel, J., Pryss, R., Reichert, M.: Mobile crowd sensing - a new way of collecting data from trauma samples? In: XIV Conference of European Society for Traumatic Stress Studies (ESTSS) Conference. (2015) 244

[20] Schobel, J., Schickler, M., Pryss, R., Reichert, M., Elbert, T.: A domain-specific framework for collecting data in trials with smart mobile devices. In: XIV Conference of European Society for Traumatic Stress Studies (ESTSS) Conference. (2015)

[21] Pryss, R., Reichert, M., Langguth, B., Schlee, W.: Mobile crowd sensing services for tinnitus assessment, therapy, and research. In: 2015 IEEE International Conference on Mobile Services. (2015) 352–359

[22] Schickler, M., Reichert, M., Pryss, R., Schobel, J., Schlee, W., Langguth, B.: Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health. eXamen.press. Springer Vieweg (2015)

[23] Probst, T., Pryss, R., Langguth, B., Schlee, W.: Emotional states as mediators between tinnitus loudness and tinnitus distress in daily life: Results from the TrackYourTinnitus application. Scientific Reports **6** (2016)

[24] Jayaraman, P.P., Sinha, A., Sherchan, W., Krishnaswamy, S., Zaslavsky, A., Haghighi, P.D., Loke, S., Do, M.T.: Here-n-Now: A Framework for Context-Aware Mobile Crowdsourcing. In: Proceedings of the Tenth International Conference on Pervasive Computing. (2012)

[25] Sherchan, W., Jayaraman, P.P., Krishnaswamy, S., Zaslavsky, A., Loke, S., Sinha, A.: Using on-the-move mining for mobile crowdsensing. In: 2012 IEEE 13th International Conference on Mobile Data Management. (2012) 115–124

[26] Haghighi, P.D., Zaslavsky, A., Krishnaswamy, S., Gaber, M.M.: Mobile data mining for intelligent healthcare support. In: 2009 42nd Hawaii International Conference on System Sciences. (2009) 1–10

[27] Padovitz, A., Loke, S.W., Zaslavsky, A.B.: Towards a theory of context spaces. In: PerCom Workshops, IEEE Computer Society (2004) 38–42

[28] Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. Int. J. Ad Hoc Ubiquitous Comput. **2** (2007) 263–277

[29] Li, X., Eckert, M., Martinez, J.F., Rubio, G.: Context aware middleware architectures: Survey and challenges. Sensors **15** (2015) 20570–20607

[30] Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England. (2004)

[31] Oliveira, L., Ribeiro, A.N., Campos, J.C. In: The Mobile Context Framework: Providing Context to Mobile Applications. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 144–153

[32] Hanson, R.: robbiehanson/CocoaHTTPServer: A small, lightweight, embeddable HTTP server for Mac OS X or iOS applications. `https://github.com/robbiehanson/CocoaHTTPServer` (2017) Online, abgerufen am 15.03.2017.

[33] Ntanos, C., Botsikas, C., Rovis, G., Kakavas, P., Askounis, D.: A context awareness framework for cross-platform distributed applications. Journal of Systems and Software **88** (2014) 138 – 146

[34] Fuhrhop, C., Lyle, J., Faily, S.: The webinos project. In: Proceedings of the 21st International Conference on World Wide Web, ACM (2012) 259–262

[35] Wang, P., Zhao, J., Liao, Q.: A cross-platform context-aware application developing framework for mobile terminals. In: 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems. Volume 03. (2012) 1272–1276

[36] China Mobile Research Institute: BAE/WAC Widget. http://labs.chinamobile.com/bae (2017) Online, abgerufen am 17.03.2017.

[37] Knappmeyer, M., Kiani, S.L., Reetz, E.S., Baker, N., Tonjes, R.: Survey of context provisioning middleware. IEEE Communications Surveys Tutorials **15** (2013) 1492–1519

[38] Schickler, M., Pryss, R., Reichert, M., Schobel, J., Langguth, B., Schlee, W.: Using mobile serious games in the context of chronic disorders - a mobile game concept for the treatment of tinnitus. In: 29th IEEE Int'l Symposium on Computer-Based Medical Systems (CBMS 2016). (2016) 343–348

[39] Pryss, R., Geiger, P., Schickler, M., Schobel, J., Reichert, M.: Advanced algorithms for location-based smart mobile augmented reality applications. Procedia Computer Science **94** (2016) 97–104

[40] Schickler, M., Pryss, R., Schobel, J., Reichert, M.: An engine enabling location-based mobile augmented reality applications. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 363–378

[41] Geiger, P., Schickler, M., Pryss, R., Schobel, J., Reichert, M.: Location-based mobile augmented reality applications: Challenges, examples, lessons learned. In: 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps. (2014) 383–394

[42] Geiger, P., Pryss, R., Schickler, M., Reichert, M.: Engineering an advanced location-based augmented reality engine for smart mobile devices. Technical Report UIB-2013-09, Ulm University, Ulm (2013)

[43] Gartner Inc.: Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016. http://www.gartner.com/newsroom/id/3609817 (2017) Online, abgerufen am 09.03.2017.

[44] O'Grady, S.: The RedMonk Programming Language Rankings: June 2016. http://redmonk.com/sogrady/2016/07/20/language-rankings-6-16/ (2016) Online, abgerufen am 10.03.2017.

[45] Serrano, N., Hernantes, J., Gallardo, G.: Mobile Web Apps. IEEE Software **30** (2013) 22–27

[46] Schobel, J., Schickler, M., Pryss, R., Nienhaus, H., Reichert, M.: Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. In: 9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps. (2013) 509–518

[47] Looper, J.: What is a WebView? `http://developer.telerik.com/featured/what-is-a-webview/` (2015) Online, abgerufen am 09.03.2017.

[48] Saccomani, P.: Native, Web or Hybrid Apps? What's The Difference? `https://www.mobiloud.com/blog/native-web-or-hybrid-apps/` (2017) Online, abgerufen am 10.03.2017.

[49] Xamarin Inc.: Xamarin: Mobile App Development & App Creation Software. `https://www.xamarin.com/` (2017) Online, abgerufen am 09.03.2017.

[50] Heitkötter, H., Hanschke, S., Majchrzak, T.A.: Evaluating cross-platform development approaches for mobile applications. In: International Conference on Web Information Systems and Technologies, Springer (2012) 120–138

[51] Riippi, J.: Native, hybrid and HTML5 – three years later. `https://www.vincit.fi/blog/native-hybrid-and-html5-three-years-later/` (2016) Online, abgerufen am 09.03.2017.

[52] Adobe Systems Inc.: PhoneGap. `http://phonegap.com/` (2017) Online, abgerufen am 09.03.2017.

[53] Apache Software Foundation: Apache Cordova. `http://cordova.apache.org/` (2017) Online, abgerufen am 09.03.2017.

[54] Adobe Systems Inc.: Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap. `http://news.adobe.com/press-release/adobe-creative-cloud-dps/adobe-announces-agreement-acquire-nitobi-creator-phonegap` (2011) Online, abgerufen am 09.03.2017.

[55] LeRoux, B.: PhoneGap, Cordova, and what's in a name? http://phonegap.com/blog/2012/03/19/phonegap-cordova-and-whate28099s-in-a-name/ (2017) Online, abgerufen am 09.03.2017.

[56] Apache Software Foundation: Apache Cordova - Documentation - Overview. https://cordova.apache.org/docs/en/latest/guide/overview/index.html (2017) Online, abgerufen am 09.03.2017.

[57] Steyer, R.: Cordova - Entwicklung plattformneutraler Apps. Springer Fachmedien Wiesbaden, Wiesbaden (2013)

[58] Intel Open Source Technology Center: Crosswalk - build world calss hybrid apps. https://crosswalk-project.org/ (2017) Online, abgerufen am 09.03.2017.

[59] Kharlampidi, V.: Framework7 - Full Featured HTML Framework For Building iOS & Android Apps. https://framework7.io/ (2017) Online, abgerufen am 09.03.2017.

[60] Open Source Initiative: The MIT License. https://opensource.org/licenses/MIT (2017) Online, abgerufen am 09.03.2017.

[61] jQuery Foundation: jQuery. https://jquery.com/ (2017) Online, abgerufen am 09.03.2017.

[62] Asial Corporation: Onsen UI 2: Beautiful HTML5 Hybrid Mobile App Framework and Tools. https://onsen.io/ (2017) Online, abgerufen am 09.03.2017.

[63] Apache Software Foundation: Licensing of Distributions. https://www.apache.org/licenses/ (2017) Online, abgerufen am 09.03.2017.

[64] Google Inc.: AngularJS - Superheroic JavaScript MVW Framework. https://angularjs.org/ (2017) Online, abgerufen am 09.03.2017.

[65] Google Inc.: One framework. - Angular. https://angular.io/ (2017) Online, abgerufen am 09.03.2017.

[66] Facebook Inc.: A JavaScript library for building user interfaces - React. https://facebook.github.io/react/ (2017) Online, abgerufen am 11.03.2017.

[67] You, E.: Vue.js. https://vuejs.org/ (2017) Online, abgerufen am 09.03.2017.

[68] Meteor Development Group Inc.: Build Apps with JavaScript - Meteor. https://www.meteor.com/ (2017) Online, abgerufen am 09.03.2017.

[69] Asial Corporation: Monaca Localkit - Experience Monaca's Usability and Comfort in Your Local Development Environment. https://monaca.io/localkit.html (2017) Online, abgerufen am 09.03.2017.

[70] Asial Corporation: Monaca Localkit Overview - Monaca Docs. https://docs.monaca.io/en/manual/development/monaca_localkit/overview/ (2017) Online, abgerufen am 11.03.2017.

[71] Asial Corporation: Angular 2 and AngularJS UI Components by Onsen UI Hybrid Mobile App Framework. https://onsen.io/angular2/ (2017) Online, abgerufen am 09.03.2017.

[72] Intel Corporation: Intel® XDK - One solution for IoT applications and mobile app dev! https://software.intel.com/en-us/intel-xdk (2017) Online, abgerufen am 10.03.2017.

[73] Intel Corporation: Why Intel® XDK NEW? https://software.intel.com/en-us/xdk/blog/why-intel-xdk-new (2013) Online, abgerufen am 09.03.2017.

[74] Hillar, G.: Developing Cross-Platform Mobile Apps with HTML5 and Intel XDK. http://www.drdobbs.com/mobile/developing-cross-platform-mobile-apps-wi/240169313 (2014) Online, abgerufen am 10.03.2017.

[75] Intel Corporation: Built-in Code Editor. https://software.intel.com/en-us/xdk/docs/using-the-editor-in-the-develop-tab (2017) Online, abgerufen am 10.03.2017.

[76] Intel Corporation: App Designer Overview. https://software.intel.com/en-us/xdk/docs/app-designer-overview (2017) Online, abgerufen am 10.03.2017.

[77] Intel Open Source Technology Center: GitHub - 01org/appframework - The definitive HTML5 mobile javascript framework. https://github.com/01org/appframework (2017) Online, abgerufen am 10.03.2017.

[78] Biswanger, G.: Cross-Plattform-Entwicklung mit dem Intel XDK in HTML5 und JavaScript. https://www.heise.de/developer/artikel/Cross-Plattform-Entwicklung-mit-dem-Intel-XDK-in-HTML5-und-JavaScript-2267883.html (2014) Online, abgerufen am 09.03.2017.

[79] Intel Corporation: Customizing App Framework UI skin. https://software.intel.com/en-us/xdk/article/customizing-app-framework-ui-skin (2017) Online, abgerufen am 10.03.2017.

[80] Drifty: Build Amazing Native Apps and Progressive Web Apps with Ionic Framework and Angular. https://ionicframework.com/ (2017) Online, abgerufen am 10.03.2017.

[81] Microsoft: TypeScript - JavaScript that scales. https://www.typescriptlang.org/ (2017) Online, abgerufen am 10.03.2017.

[82] Catlin, H., Weizenbaum, N., Eppstein, C.: Sass: Syntactically Awesome Style Sheets. http://sass-lang.com/ (2017) Online, abgerufen am 10.03.2017.

[83] Drifty: Ionic Component Documentation. http://ionicframework.com/docs/v2/components/ (2017) Online, abgerufen am 10.03.2017.

[84] Drifty: Storage - Ionic Framework. https://ionicframework.com/docs/v2/storage/ (2017) Online, abgerufen am 10.03.2017.

[85] Drifty: Ionic CLI Documentation. https://ionicframework.com/docs/v2/cli/ (2017) Online, abgerufen am 10.03.2017.

[86] Drifty: Serve. http://ionicframework.com/docs/v2/cli/serve/ (2017) Online, abgerufen am 11.03.2017.

[87] Drifty: GitHub - driftyco - Build amazing native and progressive web apps with Angular and open web technologies. One app running on everything. https://github.com/driftyco/ionic (2017) Online, abgerufen am 10.03.2017.

[88] Drifty: Ionic Creator - Rapid Mobile App Builder. https://ionic.io/products/creator (2017) Online, abgerufen am 10.03.2017.

[89] Drifty: Cloud - Ionic Cloud. https://ionic.io/cloud (2017) Online, abgerufen am 10.03.2017.

[90] Govier, P.: Introducing the Ionic Market: Buy and sell Ionic starters, plugins, and themes. http://blog.ionic.io/introducing-the-ionic-market-buy-and-sell-ionic-starters-plugins-and-themes/ (2015) Online, abgerufen am 10.03.2017.

[91] Fusetools AS: What is Fuse? https://www.fusetools.com/docs (2017) Online, abgerufen am 10.03.2017.

[92] Pedersen, R.: How Fuse differs from React Native and NativeScript. https://medium.com/fuseblog/how-fuse-differs-from-react-native-and-nativescript-525344f02aaf (2017) Online, abgerufen am 22.02.2017.

[93] Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., Zdun, U.: Software-Architektur: Grundlagen - Konzepte - Praxis. Spektrum Akademischer Verlag (2009)

[94] Bohemian BV: Sketch - Professional Digital Design for Mac. https://www.sketchapp.com/ (2017) Online, abgerufen am 10.03.2017.

[95] Sublime HQ Pty Ltd: Sublime Text: The text editor you'll fall in love with. https://www.sublimetext.com/3 (2017) Online, abgerufen am 10.03.2017.

[96] GitHub, Inc.: Atom. https://atom.io/ (2017) Online, abgerufen am 10.03.2017.

[97] Fusetools AS: Preview and Export - Docs - Fuse. https://www.fusetools.com/docs/basics/preview-and-export (2017) Online, abgerufen am 11.03.2017.

[98] Fusetools AS: Feature status. https://www.fusetools.com/docs/features (2017) Online, abgerufen am 10.03.2017.

[99] Appcelerator Inc.: Quick Start - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Quick_Start (2017) Online, abgerufen am 11.03.2017.

[100] Appcelerator Inc.: Titanium SDK - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Titanium_SDK (2017) Online, abgerufen am 11.03.2017.

[101] Appcelerator Inc.: Module development guide reference module for ios and android. http://www.appcelerator.com/blog/2011/10/module-development-guide-reference-module-for-ios-and-android/ (2017) Online, abgerufen am 11.03.2017.

[102] Appcelerator Inc.: Hyperloop. http://www.appcelerator.com/mobile-app-development-products/hyperloop/ (2017) Online, abgerufen am 11.03.2017.

[103] Haynie, J.: Hyperloop is Here. http://www.appcelerator.com/blog/2016/08/hyperloop-is-here/ (2016) Online, abgerufen am 11.03.2017.

[104] Appcelerator Inc.: Alloy Framework - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Alloy_Framework (2017) Online, abgerufen am 11.03.2017.

[105] Appcelerator Inc.: Titanium Style Sheets - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Alloy_Styles_and_Themes (2017) Online, abgerufen am 11.03.2017.

[106] Appcelerator Inc.: Appcelerator CLI - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Appcelerator_CLI (2017) Online, abgerufen am 11.03.2017.

[107] Appcelerator Inc.: Appcelerator Studio - Appcelerator Platform - Appcelerator Docs. http://docs.appcelerator.com/platform/latest/#!/guide/Appcelerator_Studio (2017) Online, abgerufen am 11.03.2017.

[108] Appcelerator Inc.: The next-generation MBaaS - Appcelerator Arrow. http://www.appcelerator.com/mobile-app-development-products/mbaas-arrow/ (2017) Online, abgerufen am 11.03.2017.

[109] Facebook Inc.: React Native - A framework for building native apps using React. https://facebook.github.io/react-native/ (2017) Online, abgerufen am 11.03.2017.

[110] Microsoft: React Native on the Universal Windows Platform. https://blogs.windows.com/buildingapps/2016/04/13/react-native-on-the-universal-windows-platform/#Wdg6SMKVMkBuFcA5.97 (2016) Online, abgerufen am 11.03.2017.

[111] Open Source Initiative: The 3-Clause BSD License. https://opensource.org/licenses/BSD-3-Clause (2017) Online, abgerufen am 11.03.2017.

[112] Stein, J.: Cold Dive into React Native: A Beginner's Tutorial. https://www.toptal.com/ios/cold-dive-into-react-native-a-beginners-tutorial (2017) Online, abgerufen am 11.03.2017.

[113] Li, S.: React Native: Into a new world of rapid iOS development. https://www.ibm.com/developerworks/library/mo-bluemix-react-native-ios8/ (2015) Online, abgerufen am 11.03.2017.

[114] Facebook Inc.: StyleSheet. https://facebook.github.io/react-native/docs/stylesheet.html (2017) Online, abgerufen am 11.03.2017.

[115] Bigio, M.: Introducing Hot Reloading. React Native Blog: https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html (2017) Online, abgerufen am 11.03.2017.

[116] Progress Software Corporation: Build amazing iOS and Android apps with technology you already know. https://www.nativescript.org/ (2017) Online, abgerufen am 11.03.2017.

[117] Atanasov, G.: NativeScript – a Technical Overview - Telerik Developer Network. http://developer.telerik.com/featured/nativescript-a-technical-overview/ (2014) Online, abgerufen am 11.03.2017.

[118] Progress Software Corporation: Application Architecture. https://docs.nativescript.org/angular/core-concepts/angular-application-architecture.html (2017) Online, abgerufen am 11.03.2017.

[119] Progress Software Corporation: Styling. https://docs.nativescript.org/angular/ui/styling.html (2017) Online, abgerufen am 11.03.2017.

[120] Baker, K.: Progress Announces NativeScript 2.0 for Native Mobile App Development with Angular 2. http://www.businesswire.com/news/home/20160504005336/en/Progress-Announces-NativeScript-2.0-Native-Mobile-App (2017) Online, abgerufen am 11.03.2017.

[121] Green, B.: Building Mobile Apps with Angular 2 and NativeScript. http://angularjs.blogspot.de/2015/12/building-mobile-apps-with-angular-2-and.html (2015) Online, abgerufen am 11.03.2017.

[122] Atanasov, G.: The Benefits of NativeScript's Single Threading Model. http://developer.telerik.com/featured/benefits-single-threading-model-nativescript/ (2016) Online, abgerufen am 11.03.2017.

[123] Buhov, I.: Using Workers in NativeScript to Execute JavaScript on Background Threads. https://www.nativescript.org/blog/using-workers-in-nativescript-to-execute-javascript-on-background-threads (2017) Online, abgerufen am 11.03.2017.

[124] Progress Software Corporation: Chapter 1 - Learning the NativeScript Basics. https://docs.nativescript.org/tutorial/chapter-1 (2017) Online, abgerufen am 11.03.2017.

[125] Muller, E.: Angular 2 vs. Angular 1: Key Differences. https://dzone.com/articles/typed-front-end-with-angular-2 (2017) Online, abgerufen am 11.03.2017.

[126] Progress Software Corporation: Welcome to NativeScript. http://docs.nativescript.org/ (2017) Online, abgerufen am 12.03.2017.

[127] Google Inc.: Template Syntax. https://angular.io/docs/ts/latest/guide/template-syntax.html (2017) Online, abgerufen am 11.03.2017.

[128] Progress Software Corporation: Data Binding. http://docs.nativescript.org/angular/core-concepts/angular-data-binding (2017) Online, abgerufen am 11.03.2017.

[129] Vantoll, T.: How NativeScript Works. http://developer.telerik.com/featured/nativescript-works/ (2017) Online, abgerufen am 12.03.2017.

[130] Google Inc.: Jni tips - android developers. https://developer.android.com/training/articles/perf-jni.html (2017) Online, abgerufen am 12.03.2017.

[131] Progress Software Corporation: Application Workflow. http://docs.nativescript.org/runtimes/android/advanced-topics/execution-flow (2017) Online, abgerufen am 12.03.2017.

[132] Progress Software Corporation: Multithreading Model. https://docs.nativescript.org/core-concepts/multithreading-model (2017) Online, abgerufen am 12.03.2017.

[133] Progress Software Corporation: Subclassing Objective-C Classes. http://docs.nativescript.org/angular/runtimes/ios/how-to/ObjC-Subclassing.html (2017) Online, abgerufen am 13.03.2017.

[134] Progress Software Corporation: Extending Classes And Interfaces. http://docs.nativescript.org/angular/runtimes/android/generator/extend-class-interface.html (2017) Online, abgerufen am 13.03.2017.

[135] Progress Software Corporation: Binding Generator Overview. http://docs.nativescript.org/angular/runtimes/android/generator/overview.html (2017) Online, abgerufen am 12.03.2017.

[136] Progress Software Corporation: Modules. http://docs.nativescript.org/angular/core-concepts/modules.html (2017) Online, abgerufen am 13.03.2017.

[137] Friesen, D.: CommonJS: JavaScript Standard Library. http://www.commonjs.org/ (2017) Online, abgerufen am 14.03.2017.

[138] npm, Inc.: npm. https://www.npmjs.com/ (2017) Online, abgerufen am 13.03.2017.

[139] Node.js Foundation: Node.js. https://nodejs.org/en/ (207) Online, abgerufen am 12.03.2017.

[140] Progress Software Corporation: Plugins. https://docs.nativescript.org/angular/plugins/plugins.html (2017) Online, abgerufen am 13.03.2017.

[141] Software Freedom Conservancy: Git. https://git-scm.com/ (2017) Online, abgerufen am 14.03.2017.

[142] Progress Software Corporation: Limitations. https://docs.nativescript.org/runtimes/ios/Limitations (2017) Online, abgerufen am 13.03.2017.

[143] Weiser, M.: Human-computer interaction. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995) 933–940

[144] Weiser, M., Brown, J.S.: Beyond calculation. Copernicus, New York, NY, USA (1997) 75–85

[145] Dey, A.K.: Providing Architectural Support for Building Context-aware Applications. PhD thesis, Atlanta, GA, USA (2000) AAI9994400.

[146] Pryss, R., Reichert, M., Schickler, M., Bauer, T.: Context-based assignment and execution of human-centric mobile services. In: 5th IEEE International Conference on Mobile Services (MS 2016), IEEE Computer Society Press (2016) 119–126

[147] Perttunen, M., Riekki, J., Lassila, O.: Context representation and reasoning in pervasive computing: a review. International Journal of Multimedia and Ubiquitous Engineering (2009) 1–28

[148] Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)

[149] Korpipää, P.: Blackboard-based software framework and tool for mobile device context awareness. PhD thesis, University of Oulu (2005)

[150] Ranganathan, A., Campbell, R.H.: A middleware for context-aware agents in ubiquitous computing environments. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer (2003) 143–161

[151] Google Inc.: Services - Android Developers. `https://developer.android.com/guide/components/services.html` (2017) Online, abgerufen am 22.03.2017.

[152] Google Inc.: IntentService - Android Developers. `https://developer.android.com/reference/android/app/IntentService.html` (2017) Online, abgerufen am 22.03.2017.

[153] Google Inc.: Service - Android Developers. `https://developer.android.com/reference/android/app/Service.html` (2017) Online, abgerufen am 22.03.2017.

[154] Apple Inc : Background Execution. `https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html` (2017) Online, abgerufen am 22.03.2017.

[155] Apple Inc.: Energy Efficiency Guide for iOS Apps - Reduce Location Accuracy and Duration. `https://developer.apple.com/library/content/documentation/Performance/Conceptual/EnergyGuide-iOS/LocationBestPractices.html` (2017) Online, abgerufen am 22.03.2017.

[156] Djuknic, G.M., Richton, R.E.: Geolocation and assisted gps. Computer **34** (2001) 123–125

[157] Progress Software Corporation: Location. `https://docs.nativescript.org/hardware/location` (2017) Online, abgerufen am 24.03.2017.

[158] Google Inc.: Locationmanager - android developers. https://developer.android.com/reference/android/location/LocationManager.html (2017) Online, abgerufen am 24.03.2017.

[159] Apple Inc.: Core Location - Apple Developer Documentation. https://developer.apple.com/reference/corelocation (2017) Online, abgerufen am 24.03.2017.

[160] Gregoski, M.J., Mueller, M., Vertegel, A., Shaporev, A., Jackson, B.B., Frenzel, R.M., Sprehn, S.M., Treiber, F.A.: Development and validation of a smartphone heart rate acquisition application for health promotion and wellness telehealth applications. Int. J. Telemedicine Appl. **2012** (2012)

[161] Allen, J.: Photoplethysmography and its application in clinical physiological measurement. Physiological Measurement (2007)

[162] Mio Global: Mio Global - Strapless Heart Rate Monitor Watch & Activity Fitness Tracker. https://www.mioglobal.com/ (2017) Online, abgerufen am 24.03.2014.

[163] Bluetooth SIG Inc.: Viewer - Bluetooth Technology Website. https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.service.heart_rate.xml (2017) Online, abgerufen am 24.03.2017.

[164] Eddy Verbruggen: EddyVerbruggen/nativescript-bluetooth: NativeScript Bluetooth LE plugin. https://github.com/EddyVerbruggen/nativescript-bluetooth (2017) Online, abgerufen am 24.03.2017.

[165] Google Inc.: Overview of Google Play Services - Google APIs for Android - Google Developers. https://developers.google.com/android/guides/overview (2017) Online, abgerufen am 24.03.2017.

[166] Google Inc.: ActivityRecognitionApi - Google APIs for Android - Google Developers. https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi (2017) Online, abgerufen am 24.03.2017.

[167] Google Inc.: Broadcastreceiver - android developers. https://developer.android.com/reference/android/content/BroadcastReceiver.html (2017) Online, abgerufen am 26.03.2017.

[168] Estes, A.C.: How Apple's M7 Chip Makes the iPhone 5S the Ultimate Tracking Device. http://gizmodo.com/how-apples-m7-chip-makes-the-iphone-5s-the-ultimate-tr-1286594287 (2017) Online, abgerufen am 24.03.2017.

[169] Apple Inc.: Core Motion - Apple Developer Documentation. https://developer.apple.com/reference/coremotion (2017) Online, abgerufen am 24.03.2017.

[170] Joachim Weise: Grundlagen der Schallpegelmessung. https://www.umweltmesstechnik-bayreuth.de/schall/schallpegelmessung.php (2017) Online, abgerufen am 24.03.2017.

[171] Thompson, A., Taylor, B.N.: Guide for the Use of the International System of Units (SI). NIST Special Publication 811 **2nd printing (November 2008)** (2008)

[172] Scholz, F.: Audiotechnik für Mediengestalter. De Gruyter (2015)

[173] Younas, M., Awan, I., Kryvinska, N., Strauss, C., van Thanh, D.: Mobile Web and Intelligent Information Systems: 13th International Conference, MobiWIS 2016, Vienna, Austria, August 22-24, 2016, Proceedings. Lecture Notes in Computer Science. Springer International Publishing (2016)

[174] Google Inc.: AudioRecord - Android Developers. https://developer.android.com/reference/android/media/AudioRecord.html (2017) Online, abgerufen am 25.03.2017.

[175] Apple Inc.: AVAudioRecorder - AVFoundation - Apple Developer Documentation. https://developer.apple.com/reference/avfoundation/avaudiorecorder (2017) Online, abgerufen am 25.03.2017.

[176] Microsoft: The Repository Pattern. https://msdn.microsoft.com/en-us/library/ff649690.aspx (2017) Online, abgerufen am 25.03.2017.

[177] Veness, C.: Calculate distance, bearing and more between Latitude/Longitude points. http://www.movable-type.co.uk/scripts/latlong.html (2017) Online, abgerufen am 25.03.2017.

[178] Google Inc.: PowerManager - Android Developers. https://developer.android.com/reference/android/os/PowerManager.html (2017) Online, abgerufen am 26.03.2017.

[179] Google Inc.: Intent - Android Developers. https://developer.android.com/reference/android/content/Intent.html (2017) Online, abgerufen am 26.03.2017.

[180] Google Inc.: Alarm Manager - Android Developers. https://developer.android.com/reference/android/app/AlarmManager.html (2017) Online, abgerufen am 26.03.2017.

[181] Apple Inc.: NSNotificationCenter - Foundation - Apple Developer Documentation. https://developer.apple.com/reference/foundation/nsnotificationcenter (2017) Online, abgerufen am 26.03.2017.

[182] Apple Inc.: UIApplicationDelegate - UIKit - Apple Developer Documentation. https://developer.apple.com/reference/uikit/uiapplicationdelegate (2017) Online, abgerufen 26.03.2017.

[183] Apple Inc.: CLLocationManager - Core Location - Apple Developer Documentation. https://developer.apple.com/reference/corelocation/cllocationmanager (2017) Online, abgerufen am 26.03.2017.

A

Klassendiagramme

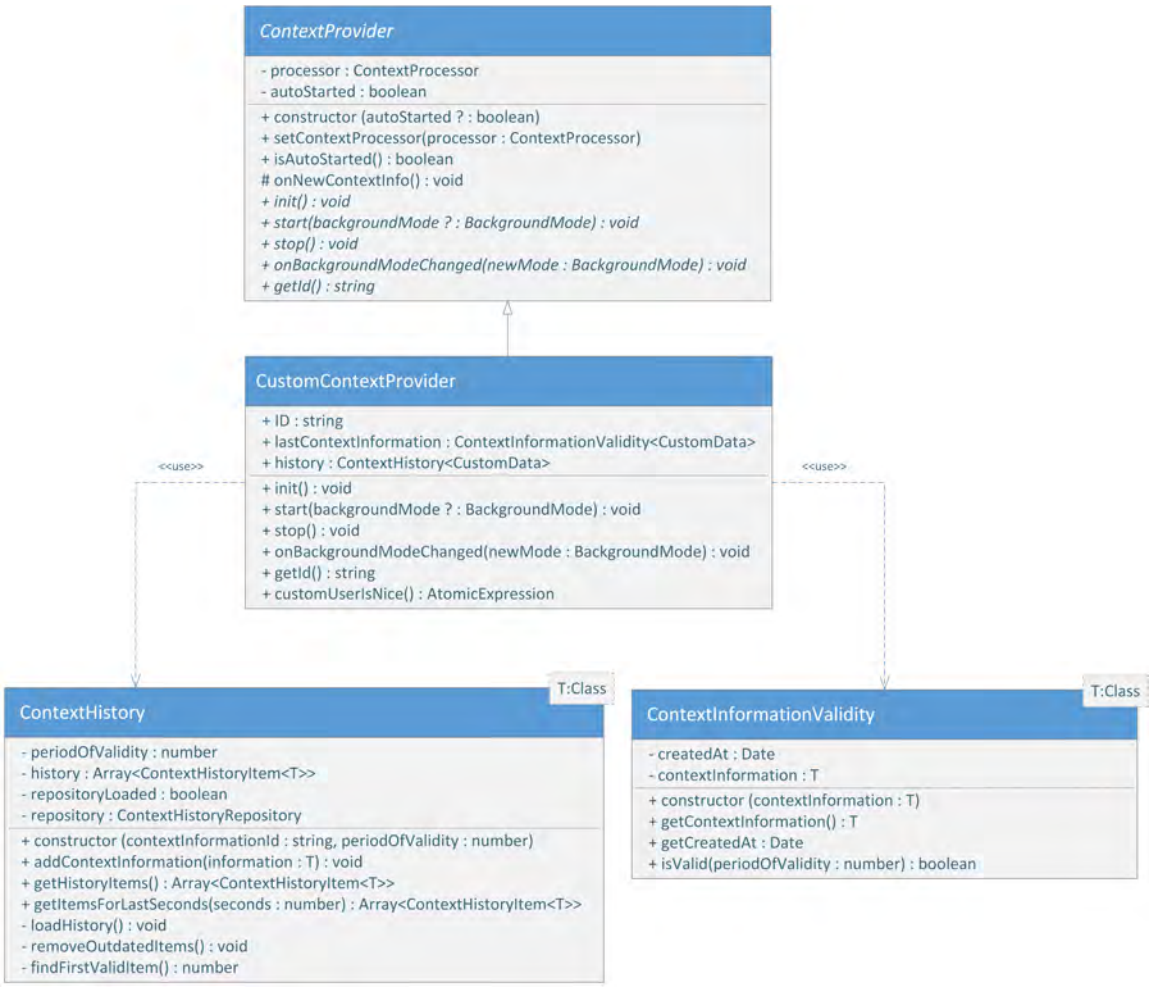

Abbildung A.1: Implementierung von Kontext-Providern

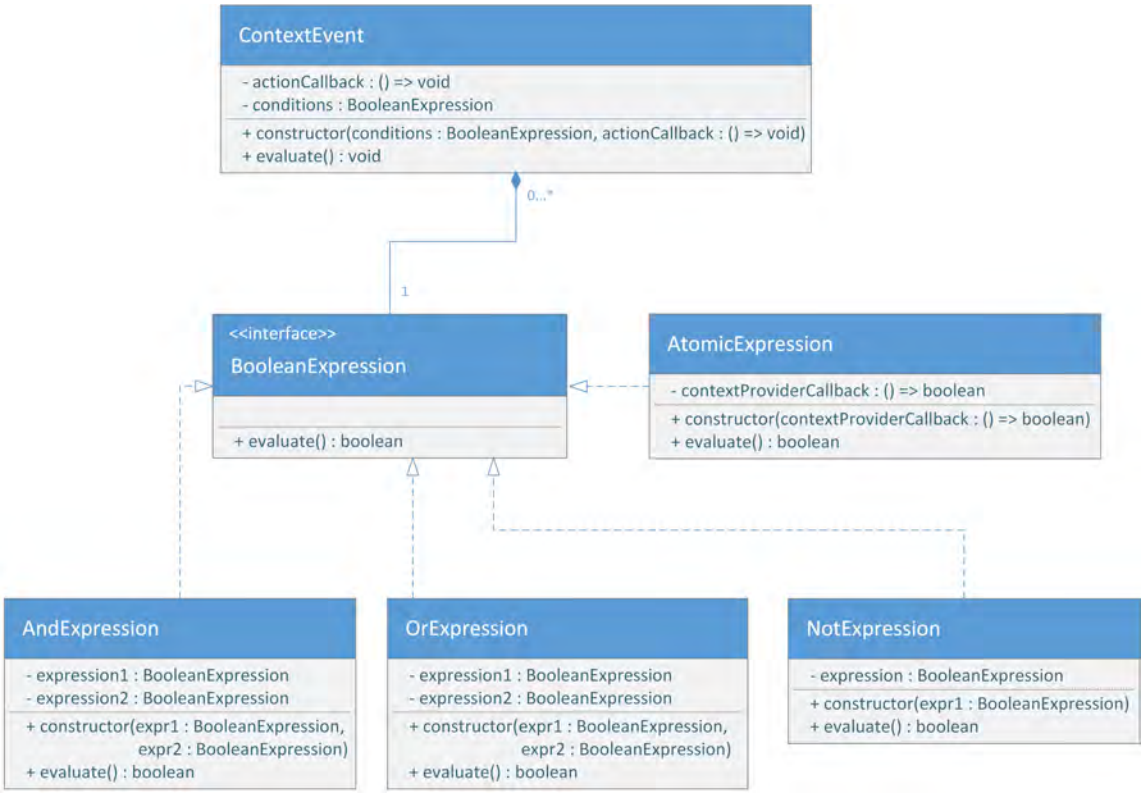

Abbildung A.2: TypeScript-Klassen zur Gestaltung von Kontextbedingungen

B

Konfigurationsparameter

B.1 nativescript-ambient-noise-level

Parameter	Datentyp	Beschreibung
`measurementInterval`	`number`	Intervall, das beschreibt, in welchen Zeitabständen die Umgebungslautstärke gemessen wird (in Sekunden, Standard: 60)
`measurementDuration`	`number`	Dauer einer Messung (in Sekunden, Standard: 5)
`calibrationDuration`	`number`	Dauer der Kalibrierungsmessung (in Sekunden, Standard: 5)
`decibelScale`	`DecibelScale`	Einheit, in der die gemessene Umgebungslautstärke zurückgegeben werden soll. Die Umgebungslautstärke wird standardmäßig in dB_{FS} bereitgestellt. Dies ist auch dann der Fall, wenn `DecibelScale.DB_SPL_ESTIMATED` spezifiziert ist, aber kein Kalibrierungswert zur Verfügung steht. (Standard: `DecibelScale.DB_FS`)

Tabelle B.1: Konfigurationsparameter des Plugins nativescript-ambient-noise-level

B.2 nativescript-ca-provider-geolocation

Parameter	Datentyp	Beschreibung
`periodOfValidity`	`number`	Eine Gültigkeitsdauer für Positionsaktualisierungen. Wenn im Rahmen der Evaluation eine betrachtete Positionsaktualisierung die spezifizierte Gültigkeitsdauer überschreitet, wird eine entsprechende Kontext-Query als falsch interpretiert. (in Sekunden, Standard: deaktiviert)
`historyPeriodOf Validity`	`number`	Wenn dieser Wert größer als 0 ist, wird ein Kontextverlauf verwendet. In diesem Fall bestimmt der Parameter die Lebensdauer der gespeicherten Positionen innerhalb des Kontextverlaufs. (in Sekunden, Standard: deaktiviert)
`desiredAccuracy`	`number`	Parameter des verwendeten nativescript-geolocation-Plugins. Definiert die gewünschte Genauigkeit der Positionsbestimmung. (1-3, Standard: 3, hohe Genauigkeit)
`updateDistance`	`number`	Parameter des verwendeten nativescript-geolocation-Plugins. Definiert einen Distanzfilter, der bestimmt, nach wie vielen zurückgelegten Metern frühestens eine Positionsaktualisierung erwünscht ist. Kann verwendet werden, um den Energieverbrauch zu reduzieren. (in Metern, Standard: 0)

`minimumUpdateTime`	`number`	Parameter des verwendeten nativescript-geolocation-Plugins. Definiert ein minimales Zeitintervall zwischen Positionsaktualisierungen für Android-Geräte. Wird auf iOS ignoriert. (in Millisekunden, Standard: 10)
`maximumAge`	`number`	Parameter des verwendeten nativescript-geolocation-Plugins. Definiert, wie alt Positionsaktualisierungen maximal sein dürfen. (in Millisekunden, Standard: deaktiviert)
`timeout`	`number`	Parameter des verwendeten nativescript-geolocation-Plugins. Definiert, wie lange maximal auf Positionsaktualisierungen gewartet wird, bevor die Positionsbestimmung deaktiviert wird. (in Millisekunden, Standard: deaktiviert)
`iosAllowsBackground LocationUpdates`	`boolean`	Dieser Parameter muss für iOS-Geräte auf `true` gesetzt werden, um Positionsaktualisierungen in (langlaufenden) Hintergrundaufgaben erhalten zu können. Wird auf Android ignoriert. (Standard: `false`)
`iosPausesLocation UpdatesAutomatically`	`boolean`	Dieser Parameter muss für iOS-Geräte auf `false` gesetzt werden, um das automatische Pausieren der Positionsaktualisierungen zu unterbinden. Dies kann für langlaufende Hintergrundaufgaben notwendig sein, die sonst aus Energiespargründen gestoppt werden könnten, wenn keine Positionsaktualisierungen empfangen werden. Wird auf Android ignoriert. (Standard: `true`)

Tabelle B.2: Konfigurationsparameter des Kontext-Providers nativescript-ca-provider-geolocation

B.3 nativescript-ca-provider-heart-rate

Parameter	Datentyp	Beschreibung
`autoConnectInterval`	`number`	Dieses Zeitintervall spezifiziert, in welchen Abständen ein automatischer Verbindungsaufbau zum Standardgerät stattfinden soll, wenn keine Verbindung vorhanden ist. Das Intervall muss größer gleich 5 Sekunden sein. (in Sekunden, Standard: 30)
`periodOfValidity`	`number`	Eine Gültigkeitsdauer für Herzfrequenzmesswerte. Wenn im Rahmen der Evaluation ein betrachteter Messwert die spezifizierte Gültigkeitsdauer überschreitet, wird eine entsprechende Kontext-Query als falsch interpretiert. (in Sekunden, Standard: deaktiviert)
`smaFrameSize`	`number`	Wenn dieser Wert größer gleich 1 ist, wird ein einfacher gleitender Mittelwert bei der Kontextvorverarbeitung verwendet. In diesem Fall spezifiziert der Parameter die Anzahl der berücksichtigen Samples und damit das verwendete Fenster. (Anzahl der Samples, Standard: deaktiviert)
`historyPeriodOf Validity`	`number`	Wenn dieser Wert größer als 0 ist, wird ein Kontextverlauf verwendet. In diesem Fall bestimmt der Parameter die Lebensdauer der gespeicherten Messwerte innerhalb des Kontextverlaufs. (in Sekunden, Standard: deaktiviert)

Tabelle B.3: Konfigurationsparameter des Kontext-Providers nativescript-ca-provider-heart-rate

B.4 nativescript-ca-provider-activity

Parameter	Datentyp	Beschreibung
`detectionInterval`	`number`	Dieses Zeitintervall spezifiziert, wie oft erkannte Aktivitäten von der nativen Android-Schnittstelle empfangen werden sollen. Kann genutzt werden, um den Energieverbrauch zu senken. Wird auf iOS ignoriert. (in Millisekunden, Standard: 0, so schnell wie möglich)
`minimumConfidence`	`number`	Standardmäßiger Schwellenwert, der die minimale Genauigkeit von erkannten Aktivitäten in Prozent spezifiziert. Wird von den Kontext-Queries verwendet, wenn kein anderer Schwellenwert angegeben wurde. (0-100, Standard: 90)
`periodOfValidity`	`number`	Eine Gültigkeitsdauer für erkannte Aktivitäten. Wenn im Rahmen der Evaluation eine betrachtete Aktivität die spezifizierte Gültigkeitsdauer überschreitet, wird eine entsprechende Kontext-Query als falsch interpretiert. (in Sekunden, Standard: deaktiviert)
`historyPeriodOf Validity`	`number`	Wenn dieser Wert größer als 0 ist, wird ein Kontextverlauf verwendet. In diesem Fall bestimmt der Parameter die Lebensdauer der gespeicherten Aktivitäten innerhalb des Kontextverlaufs. (in Sekunden, Standard: deaktiviert)

Tabelle B.4: Konfigurationsparameter des Kontext-Providers nativescript-ca-provider-activity

B.5 nativescript-ca-provider-ambient-noise

Parameter	Datentyp	Beschreibung
`measurementInterval`	`number`	Parameter des verwendeten nativescript-ambient-noise-level-Plugins. Intervall, das beschreibt, in welchen Zeitabständen die Umgebungslautstärke gemessen wird. (in Sekunden, Standard: 30)
`measurementDuration`	`number`	Parameter des verwendeten nativescript-ambient-noise-level-Plugins. Dieser Parameter spezifiziert die Dauer einer Messung. (in Sekunden, Standard: 5)
`calibrationDuration`	`number`	Parameter des verwendeten nativescript-ambient-noise-level-Plugins. Dieser Parameter spezifiziert die Dauer der Kalibrierungsmessung. (in Sekunden, Standard: 5)
`decibelScale`	`DecibelScale`	Parameter des verwendeten nativescript-ambient-noise-level-Plugins. Einheit, der gemessenen Umgebungslautstärke, die innerhalb von Kontext-Queries verwendet werden soll. Die Umgebungslautstärke wird standardmäßig in dB_{FS} verwendet. Dies ist auch dann der Fall, wenn `DecibelScale.DB_SPL_ESTIMATED` spezifiziert ist, aber kein Kalibrierungswert zur Verfügung steht. (Standard: `DecibelScale.DB_FS`)

`periodOfValidity`	`number`	Eine Gültigkeitsdauer für Messungen. Wenn im Rahmen der Evaluation eine betrachtete Messung die spezifizierte Gültigkeitsdauer überschreitet, wird eine entsprechende Kontext-Query als falsch interpretiert. (in Sekunden, Standard: deaktiviert)
`smaFrameSize`	`number`	Wenn dieser Wert größer gleich 1 ist, wird ein einfacher gleitender Mittelwert bei der Kontextvorverarbeitung verwendet. In diesem Fall spezifiziert der Parameter die Anzahl der berücksichtigten Samples und damit das verwendete Fenster. (Anzahl der Samples, Standard: deaktiviert)
`historyPeriodOf Validity`	`number`	Wenn dieser Wert größer als 0 ist, wird ein Kontextverlauf verwendet. In diesem Fall bestimmt der Parameter die Lebensdauer der gespeicherten Messwerte innerhalb des Kontextverlaufs. (in Sekunden, Standard: deaktiviert)

Tabelle B.5: Konfigurationsparameter des Kontext-Providers nativescript-ca-provider-ambient-noise

B.6 nativescript-background-service

Parameter	Datentyp	Beschreibung
`androidAutoStart AfterAppStart`	`boolean`	Wenn dieser Parameter auf `true` gesetzt ist, wird der Hintergrund-Service direkt nach dem Start der Applikation ausgeführt. (Android, Standard: `false`)
`androidAutoStart AfterBoot`	`boolean`	Wenn dieser Parameter auf `true` gesetzt ist, wird der Hintergrund-Service direkt nach dem Hochfahren des Gerätes ausgeführt. (Android, Standard: `false`)
`androidService StartSticky`	`boolean`	Dieser Wert spezifiziert, ob ein Hintergrund-Service automatisch wieder gestartet werden soll, falls dieser vom Betriebssystem beendet wurde. (Android, Standard: `true`)
`androidService Interval`	`number`	Wenn dieser Wert größer als 0 ist, wird der Hintergrund-Service im Intervall ausgeführt. In diesem Fall bestimmt der Parameter die Länge des Intervalls (mindestens 60 Sekunden). (Android, in Sekunden, Standard: deaktiviert)
`androidService IntervalTimeout`	`number`	Bestimmt die Ausführungsdauer des Hintergrundmodus bei der Verwendung eines Intervalls. Nach dieser Zeitspanne wird der Hintergrund-Service automatisch gestoppt. Dieser Wert muss kleiner gleich dem `androidServiceInterval`-Parameter sein. (Android, in Sekunden, Standard: 30)

iosRunInForeground	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die manuell aktivierte Hintergrundausführung gestartet, während die Applikation geöffnet ist. (iOS, Standard: false)
iosRunInBackground	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die manuell aktivierte Hintergrundausführung gestartet, wenn die Applikation in einen Hintergrundmodus übergeht. (iOS, Standard: false)
iosRunInBackground Fetch	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die manuell aktivierte Hintergrundausführung gestartet, wenn der Background-Fetch-Modus aktiviert wird. Hierfür muss in der Info.plist der NativeScript-Applikation der UIBackgroundModes-Wert fetch vorhanden sein. (iOS, Standard: false)
iosRunOnSignificant LocationUpdate	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die manuell aktivierte Hintergrundausführung gestartet, wenn ein signifikanter Ortswechsel registriert wird. Hierfür muss der Nutzer eine Berechtigung erteilen, dass Positionsaktualisierungen auch im Hintergrund von der Applikation empfangen werden dürfen. (iOS, Standard: false)
iosAutoStartIn Foreground	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die Hintergrundausführung automatisch gestartet, wenn die Applikation im Vordergrund geöffnet wird. (iOS, Standard: false)

iosAutoStartIn Background	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die Hintergrundausführung automatisch gestartet, wenn die Applikation in den Hintergrund übergeht. (iOS, Standard: false)
iosAutoStartIn BackgroundFetch	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die Hintergrundausführung automatisch gestartet, wenn der Background-Fetch-Modus aktiviert wird. (iOS, Standard: false)
iosAutoStartOn SignificantLocation Update	boolean	Wenn dieser Parameter auf true gesetzt ist, wird die Hintergrundausführung automatisch ausgeführt, wenn ein signifikanter Ortswechsel registriert wird. Hierfür muss der Nutzer eine Berechtigung erteilen, dass Positionsaktualisierungen auch im Hintergrund von der Applikation empfangen werden dürfen. (iOS, Standard: false)
iosBackgroundRequest MoreTime	boolean	Wenn dieser Parameter auf true gesetzt ist, wird beim Start des Hintergrundmodus eine Hintergrundaufgabe erstellt. Dadurch wird die gewährte Rechenzeit von 10 auf 180 Sekunden verlängert. (iOS, Standard: true)

`iosSignificantLocation UpdateRequestMoreTime`	`boolean`	Wenn dieser Parameter auf `true` gesetzt ist, wird beim Start des Hintergrundmodus nach einem signifikanten Ortswechsel eine Hintergrundaufgabe erstellt. Dadurch wird die gewährte Rechenzeit von 10 auf 180 Sekunden verlängert. (iOS, Standard: `true`)
`iosStartMonitoring SignificantLocation ChangesOnAppStart`	`boolean`	Wenn dieser Parameter auf `true` gesetzt ist, wird beim Start der Applikation automatisch die Überwachung signifikanter Ortsänderungen gestartet. (iOS, Standard: `false`)

Tabelle B.6: Konfigurationsparameter des Plugins nativescript-background-service

C

Quelltexte

C.1 nativescript-background-service

```
import { BackgroundMode, BackgroundServiceImplementation }
         from "nativescript-background-service";;
import { setInterval, clearInterval } from "timer";

export default class BackgroundService extends
  BackgroundServiceImplementation {

  private reconnectTimerId : number;

  public run(startMode : BackgroundMode) : void {
    console.log("BackgroundService instance started!" +
                " (background mode: " + startMode + ")");

    // log every 10 seconds
        this.reconnectTimerId = setInterval(() => {
      console.log("BackgroundService instance is running!" +
                  " (background mode: " + this.getMode() + ")");
    }, 10000);
  }

  public onStop() : void {
    // clear timer
    if (this.reconnectTimerId > 0) {
      clearInterval(this.reconnectTimerId);
    }

    console.log("BackgroundService instance: onStop()");
   }

  public onModeChanged(newMode : BackgroundMode) : void {
    console.log("BackgroundService instance - changed mode to "
                + newMode + ".");
  }
}
```

Listing 18: Beispielhafte Implementierung der BackgroundService-Klasse

```
...

@JavaProxy("de.uulm.dbis.nativescript.AndroidBackgroundService")
class AndroidBackgroundService extends android.app.Service {

  ...

  private customImplementation : BackgroundServiceImplementation;

  public onStartCommand(intent : android.content.Intent,
                        flags : number, startId : number) : number {
    ...

    // run custom logic
    if (this.customImplementation === undefined) {
      this.customImplementation = new CustomBackgroundService();
      this.customImplementation.runImplementation(BackgroundMode.
        AndroidLongRunning);
    }

    // if a interval service is used, stop the service after
    // the corresponding timeout
    if (BackgroundServiceControl.settings.androidServiceInterval > 0) {
      this.stopSelfTimerId = setTimeout(() => {
        console.log("nativescript-background-service: Interval service " +
                    "stops itself after timeout.");
        this.stopSelf();
      }, BackgroundServiceControl.settings.androidServiceIntervalTimeout
         * 1000);
    }

    if (BackgroundServiceControl.settings.androidServiceStartSticky) {
      return android.app.Service.START_STICKY;
    } else {
      return android.app.Service.START_NOT_STICKY;
    }
  }

  public onDestroy() {
    if (this.backgroundServiceImplementation != undefined) {
      this.backgroundServiceImplementation.onStop();
    }
    ...
  }
}
```

Listing 19: Implementierung der nativen Service-Klasse auf Android

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Julian Winterfeldt Matrikelnummer: 850334

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Julian Winterfeldt