ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

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

Evaluation der Polar Fitness-Tracker im Kontext von Mobile Medical Apps

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Lukas Hennig
lukas.hennig@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2018

Fassung 19. Juni 2018

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Der menschliche Körper sendet permanent Signale aus. Auf diese Signale zu achten und sie richtig zu interpretieren, dient im Alltag der Förderung eines gesünderen Lebensstils, ist im Sport Grundlage für eine stetige Leistungsoptimierung und unterstützt in der Medizin eine erfolgreiche Rehabilitation. Die Visualisierung und Interpretation dieser Daten erlaubt Rückschlüsse auf Fehlverhalten im Alltag oder im Training. Fitnessarmbänder sind in der Lage, diese Signale aufzunehmen und zu speichern. Sie zählen die täglich zurückgelegten Schritte, berechnen den Kalorienverbrauch und messen die Herzfrequenz des Nutzers im Tagesverlauf. Der Überblick über die Tagesaktivität soll den Nutzer zu einem aktiveren und gesünderen Lebensstil anregen.

Smartphone-Apps können helfen, die von Fitness-Trackern gesammelten Daten darzustellen, zu analysieren und zu interpretieren. Diese digitale Selbstvermessung jedes Nutzers eines Fitness-Trackers kann zu einer individualisierten Diagnostik und somit zu einer individualisierten Therapie in der Medizin führen. *Polar*, als Erfinder der kabellosen Herzfrequenz-Messgeräte, ist als Marktführer im Bereich Herzfrequenz-Messung [1] prädestiniert für die Entwicklung einer solchen *Mobile Medical App*.

Danksagung

Mein Dank geht an meine Familie für die unterstützenden Worte, die zur Beendigung dieser Arbeit ihren Teil beigetragen haben, meine Freundin Maja für ihre Anteilnahme und ihren Rat bei Problemen und Anna Maria und Tobias, die ich bei Fragen zu Rate ziehen konnte.

Zusätzlich danke ich Marc Schickler für die Unterstützung während der Arbeit und Herrn Prof. Dr. Manfred Reichert für die Begutachtung danach.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Problemstellung 2

1.2 Zielsetzung 2

1.3 Struktur der Arbeit 2

2 Grundlagen **5**

2.1 mHealth 5

2.1.1 Fitness-Tracker 6

2.1.2 Herzfrequenz-Messung 8

2.2 Polar 9

2.2.1 Polar Flow 9

2.2.2 AccessLink 13

2.3 Vergleichbare Fitness-Tracker Hersteller 13

2.3.1 Garmin 13

2.3.2 Fitbit 14

3 Technische Grundlagen **15**

3.1 Ionic 3 15

3.1.1 Einordnung in Mobile Apps 15

3.1.2 Cordova 17

3.1.3 Architektur 18

3.2 REST API 23

3.3 OAuth2-Authentifizierung 25

3.4 GPS- und trainingsspezifische Dateiformate 27

4 Anforderungsanalyse **33**

4.1 Funktionale Anforderungen 33

4.2 Nichtfunktionale Anforderungen 35

5 Konzeption und Entwurf **37**
5.1 Vorstellung der AccessLink-API . . . 37
5.1.1 Ablauf der Kommunikation . . . 37
5.1.2 Registrierung als Polar-Client . . . 38
5.1.3 Authentifizierung . . . 38
5.1.4 Transaktionen . . . 38
5.2 Mock-ups . . . 41
5.3 Datenbankentwurf . . . 48
5.3.1 Datenanalyse . . . 48
5.3.2 Entity-Relationship-Modell des Datenbankentwurfes . . . 50

6 Realisierung **51**
6.1 Verwendete Frameworks . . . 51
6.2 Implementierung . . . 54
6.2.1 Authentifizierungsvorgang . . . 55
6.2.2 Nutzerbezogene Anfragen . . . 56
6.2.3 Realisierung der Kommunikation . . . 56
6.2.4 Datenbank . . . 59
6.2.5 HttpInterceptor . . . 60
6.2.6 Realisierung des User-Interfaces . . . 61

7 Anforderungsabgleich **67**
7.1 Funktionale Anforderungen . . . 67
7.2 Nichtfunktionale Anforderungen . . . 69

8 Auswertung und Ausblick **71**
8.1 Versuch zur Erhebung von Synchronisationszeiten . . . 71
8.2 Fazit . . . 73
8.3 Ausblick . . . 73

A Quelltexte **77**

1

Einleitung

Heutzutage gibt es dank des Smartphones viele Möglichkeiten, Informationen unterwegs abzurufen und im ständigen Kontakt mit der Außenwelt zu stehen. *Wearables* in Form von Fitness-Trackern können dabei Informationen über das Befinden des Nutzers sammeln und ihm oder Anderen zur Verfügung stellen. Zur Interpretation oder Verwaltung der gesammelten Daten werden anschließend mobile Anwendungen verwendet. Inwiefern diese Daten anschließend verarbeitet oder geteilt werden, hängt von der jeweiligen Anwendung ab, mit der die Daten synchronisiert wurden. Verfolgt die Anwendung einen medizinischen Ansatz, können so Informationen über die Gesundheit oder sportlichen Aktivitäten des Nutzers direkt dargestellt werden, wodurch eine neue Form der Diagnostik entsteht.

Projekte solcher mobilen Anwendungen werden bereits umgesetzt. *Track your Tinitus* (TYT) ist eine Anwendung zur Dokumentation der Entwicklung der Hörleistung von Hörgeschädigten unter Berücksichtigung der täglichen Schwankungen. Bisher war das Sammeln der relevanten Daten sehr aufwendig, da ein direkter Kontakt von Patient und Arzt zur richtigen Zeit von Nöten war. Durch die Verwendung der mobilen Anwendung TYT kann dieser Kontakt auf den Austausch von ausgefüllten, mobilen Fragebögen reduziert werden. Dadurch können viele Daten zum richtigen Zeitpunkt erhoben werden [2].

Basierend auf Herzfrequenzmessungen, Kalorienverbrauch und verschiedenen weiteren Sensoren eines Fitness-Trackers kann dieses Prinzip mit der Unterstützung von Fitness-Trackern übernommen werden.

1.1 Problemstellung

Der Markt an Smartphones ist groß mit einer Vielfalt an Herstellern und Betriebssystemen. *Android*, *IOS* und *Windows Phone* gehören zu den verbreitetest Plattformen mit jeweils eigener Struktur und Programmiersprache. Dadurch ist die Entwicklung einer Anwendung für jede dieser Plattformen aufwendig, da sie dreimal implementiert werden muss. Ionic 3, als hybrides Framework bietet eine Alternative. Ionic verwendet zur Entwicklung von Anwendungen Webtechnologien, also Typescript, CSS und HTML und kann auf jeder der Plattformen verwendet werden, was den Aufwand der Entwicklung vermindert. Inwiefern Ionic mit Polar, einem Hersteller von Fitness-Tracker kompatibel ist, soll in dieser Arbeit untersucht werden.

1.2 Zielsetzung

In dieser Arbeit wird die Entwicklung einer Ionic-Testanwendung auf dem Betriebssystem *Android* dokumentiert. Alle Daten, die der Polar Fitness-Tracker sammelt, sollen auf dem Gerät verfügbar und für den Nutzer sichtbar sein. Die Synchronisationszeit dieser Daten soll dabei möglichst gering und im Bestfall in Echtzeit erfolgen. Dabei sollen Fitness-Tracker in einem medizinischen Zusammenhang betrachtet und eventuelle Anwendungsbereiche herausgearbeitet werden. Zusätzlich soll die Arbeitsumgebung, sowohl von Polar, als auch von Ionic vorgestellt werden.

1.3 Struktur der Arbeit

Zu Beginn der Arbeit wird in Kapitel 2 Polar mit den Fitness-Trackern vorgestellt und mit der Konkurrenz verglichen. Zusätzlich werden Fitness-Tracker in den Kontext von medizinischen Verwendungsmöglichkeiten gebracht. Im folgenden Kapitel 3 werden die technischen Hintergründe der Testanwendung erläutert und auf grundlegende Funktionsweisen von Webservern eingegangen. Im Anschluss werden die Anforderungen

an die Anwendung formuliert und im nächsten Kapitel 5 entworfen. Die anschließende Realisierung in Kapitel 6 wird mit den Anforderungen aus Kapitel 4 in Kapitel 7 abgeglichen.

2

Grundlagen

2.1 mHealth

eHealth (electronic Health) ist ein Überbegriff für alle elektronische Systeme im Gesundheitswesen. Dazu gehören alle Hilfsmittel und Gerätschaften, die zur Diagnostik und Therapie der Patienten beitragen. Ein Teilgebiet von eHealth ist *mHealth*.

Zu mHealth (mobile Health) gehören Systeme zur Erreichung von Gesundheitszielen mit drahtloser und mobiler Technik, wie beispielsweise Smartphones und Tablets [3]. Das Potential von mHealth-Geräten steigt mit der Verbreitung von Smartphones und Fitness-Trackern. Aus diesem Grund können Zielgruppen, wie chronisch Kranke, Gesundheits- und Fitnessinteressierte und Ärzte direkten Kontakt pflegen ohne sich am selben Ort zu befinden. Dadurch entsteht die Chance die Effizienz im Gesundheitswesen durch Prävention und frühere Feststellung von Erkrankungen zu verbessern.
Probleme, die mit mHealth auftreten sind zum einen die Kosten, die Datensicherheit und die Unkenntnis über die Bedienung des Geräts und zum anderen können, im Falle der Diagnostik, Fehldiagnosen fatale Folgen haben und Erkrankungen unbehandelt lassen, die bei direktem Patientenkontakt erkannt worden wären.

Nichtsdestotrotz birgt mHealth das Potential, neue Therapie- und Behandlungsmöglichkeiten zu entwickeln, die die Zusammenarbeit von Arzt und Patient unterstützen [3]. Mit der Unterstützung von Fitness-Trackern kann dieser Dialog gefördert werden.

2.1.1 Fitness-Tracker

Fitness-Tracker sind Geräte, die am Körper getragen werden und mit verschiedenen Sensoren ausgestattet sind. Diese Sensoren zeichnen Schritte, Kalorienverbrauch, Beschleunigung, Höhe, Distanz, GPS-Strecke und Herzfrequenz auf. Die aufgezeichneten Daten werden auf den Fitness-Tracker gespeichert und können anschließend von Programmen analysiert, interpretiert und visualisiert werden. Dadurch können Krankheiten frühzeitig erkannt, im Training der Fortschritt überwacht und im Alltag Fehlverhalten verdeutlicht werden.

Für ein besseres Verständnis der Fitness-Tracker, wird die in der Arbeit verwendete *Polar M400* vorgestellt und anschließend mit einem höherklassigem Fitness-Tracker *Polar M600* verglichen. Beide Fitness-Tracker werden in Abbildung 2.1 dargestellt.

Polar M400

Die Polar M400 ist preislich eingeordnet ein eher günstiger Fitness-Tracker (€ 125, Stand: 13.04.2018), die die im Folgenden aufgeführten Funktionen mit sich bringt [4]. Einen ersten visuellen Eindruck verschafft die Abbildung 2.1a.

Integriertes GPS

Mit dem integrierten GPS werden Geschwindigkeit, Distanz, Höhe und Route aufgezeichnet.

Bluetooth

Durch Bluetooth können die aufgezeichneten Daten drahtlos mit Smartphones, Tablets und Computern synchronisiert werden.

Aktivitätsberichte

Mit Hilfe des integrierten *Beschleunigungssensors* können alle Bewegungen verfolgt werden. Dadurch wird dem Nutzer ein Tagesverlauf angezeigt und er wird aktiv zur Bewegung aufgefordert.

Wasserdicht

Die Polar M400 ist bis zu einer Tiefe von 30 m wasserdicht und kann somit auch im Wasser verwendet werden.

Herzfrequenzmessung

Die Polar M400 besitzt keine verarbeitete Herzfrequenzmessung. Sie kann jedoch mit einem Brustgurt gekoppelt werden, damit die Herzfrequenz gemessen werden kann. Der Brustgurt wird in Kapitel 2.1.2 genauer vorgestellt.

Polar M600

Eine Polar M600 ist ein vergleichsweise teures Modell (€250, Stand: 18.04.2018), das die Funktionen der Polar M400 um weitere ergänzt [5].

Pulsmesser am Handgelenk

Die Polar M600 besitzt einen eingebauten Pulsmesser, was eine Messung ohne zusätzlichen Brustgurt ermöglicht.

Schwimm-Metriken

Sie erkennt verschiedene Schwimmstile, sowie Tempo und Distanz.

Touchscreen

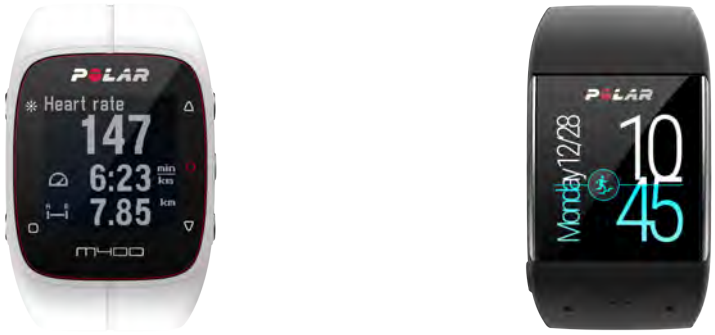
Für die Navigation durch das Menü besitzt die Polar M600 ein Touchscreen.

Android Wear (Wear OS)

Auf der Polar M600 läuft ein von Android abgeleitetes Betriebssystem, das den Zugriff auf verschiedene Apps im *Google PlayStore* ermöglicht. Die Polar M600 ist mit Mobilgeräten kompatibel, die über Android™4.3+ oder iOS™8.2+ verfügen.

Musik

Durch 4 GB Speicher und die Verbindung zum *Google Play Music* kann Musik gespeichert und abgespielt werden.



(a) Polar M400 (b) Polar M600

Abbildung 2.1: Fitness-Tracker

2.1.2 Herzfrequenz-Messung

Um die Messung der Herzfrequenz bei nicht-statischen Übungen durchzuführen, gibt es zwei verschiedene Herangehensweisen.

Zum einen kann die Messung über den *Brustgurt* erfolgen. Der Gurt wird unterhalb der Brust getragen und erfasst die Frequenz des Herzschlags mittels integrierten Hautelektroden. Die Empfänger der Herzfrequenz-Messungen sind meist Armbanduhren, die in der Lage sind die Signale des Brustgurts anzuzeigen, zu speichern und gegebenenfalls auszuwerten.

Zum anderen kann die Herzfrequenzmessung am *Handgelenk* vorgenommen werden. Anstatt das Signal vom Brustgurt zu übermitteln, strahlen die *Activity Trackern* selbst grünes Licht auf das Gewebe am Handgelenk. Da das grüne Licht stark vom Blut absorbiert wird, kann die Herzfrequenz durch pulsierende Blutgefäße ermittelt werden.

Fitness-Tracker messen bei Ruhe am besten. Je höher jedoch die Belastung wird, desto stärker weichen die Messwerte von den Belastungs-EKG-Werten ab. Der Brustgurt liefert im Gegensatz zu den Fitness-Trackern deutlich genauere Pulsmesswerte, welche mit einer medizinischen EKG-Messung mithalten können. Jedoch behindert ein Brustgurt den Sportler in seiner Bewegungsfreiheit, was wiederum ein beengendes Gefühl auslösen kann [6].

2.2 Polar

Dem Polar-Gründer Sari Säynäjäkangas kam 1975 während einer Skitour die Idee ein tragbares, kabelloses Herzfrequenz-Messgerät zu entwickeln. Bis zu diesem Zeitpunkt konnte die Herzfrequenz nur am Finger gemessen werden, was eine Messung der Herzfrequenz während nicht-statischen Übungen unmöglich machte. 1977 gründete er daher *Polar Electro* und brachte 1982 den weltweit ersten tragbare Herzfrequenz-Messer auf den Markt. Seit diesem Zeitpunkt hat Polar Electro ihr Sortiment stetig vergrößert und verbessert [7].

2.2.1 Polar Flow

Polar Flow ist die zugehörige App zu einem Polar Fitness-Tracker. Sie hat die Funktion die Sport-, Fitness- und Aktivitätsdaten der Polar Fitness-Tracker zu synchronisieren und zu analysieren. Im Folgenden werden der Synchronisationsvorgang und die Möglichkeiten der Datenanalyse vorgestellt.

Synchronisation

Nachdem die Herkunft der Fitness-Tracker und ihre Funktionsweise erläutert wurde, betrachten wir im Folgenden die Synchronisation der Daten zwischen Fitness-Tracker und PC, bzw. Smartphone. Dabei gibt es zwei Möglichkeiten, die gesammelten Daten in die Polar-Datenbank einzuspeisen:

Smartphone-Synchronisation Zum Synchronisieren mit dem Smartphone muss die *Polar Flow* App installiert sein. Nachdem sich der Nutzer ein Polar Benutzerkonto erstellt hat und sich mit diesem in der App angemeldet hat, kann er den Fitness-Tracker mit der Polar-App koppeln.
Dazu kann er sich auf seinem Wearable unter *Eingaben > Allgemeine Einstellungen > Koppeln und Synchronisieren > Mobiles Gerät koppeln und synchronisieren* einen PIN anzeigen lassen, den der Nutzer in der App bestätigen kann [8].

Nach der ersten Kopplung muss der Nutzer bei den folgenden Synchronisationen die ZURÜCK-Taste gedrückt halte, damit sich der Fitness-Tracker zuerst mit dem Smartphone und anschließend mit der App verbindet, die die Synchronisation durchführt. Sobald die Synchronisation abgeschlossen ist, wird eine Bestätigungsnachricht angezeigt.

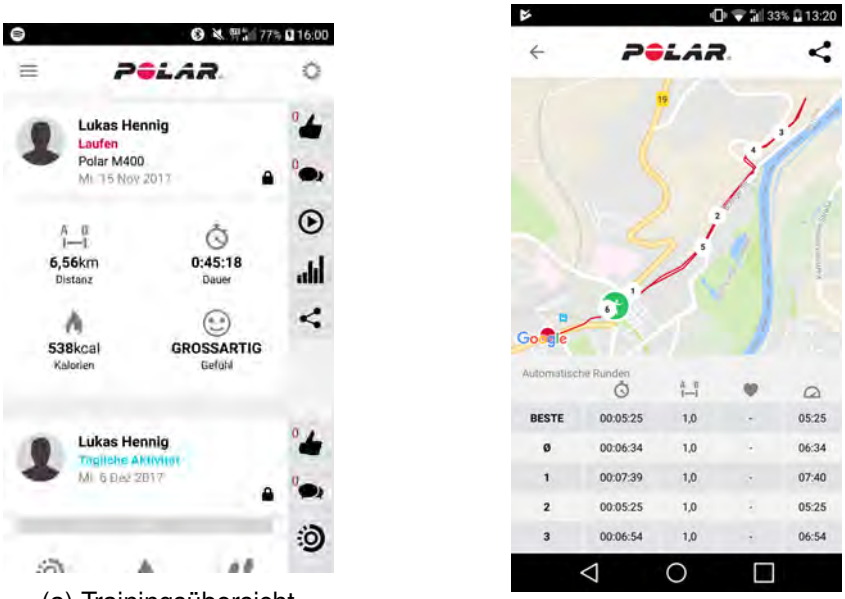
PC-Synchronisation Die Synchronisation mit dem PC läuft ähnlich ab wie mit dem Smartphone. Dabei muss die *Polar FlowSync* Software heruntergeladen werden. Nach der Installation der Software muss der Fitness-Tracker mit dem PC via USB-Kabel verbunden werden. Man wird anschließend direkt zum *Polar Webservice* weitergeleitet. Dieser fordert den Nutzer zu einer Registrierung oder, bei einem vorhandenen Account, zur Anmeldung mit dem Benutzerkonto auf.
Anschließend wird bei jeder Verbindung von PC und Fitness-Tracker die *Polar FlowSync* Software geöffnet und die Synchronisierung gestartet.

In Kapitel 8.1 wird eine Versuch vorgestellt, der den Synchronisationsablauf näher untersuchen soll.

Analyse

Trainings-Analyse Polar Flow lässt den Nutzer nach der Synchronisation jedes Detail der jeweiligen Trainingseinheit einsehen. Ein Überblick dieser Trainingseinheit wird in dem Untermenü *Feed*, wie in Abbildung 2.2a, angezeigt. Dort werden die wichtigsten Daten des Trainings, wie Distanz, Dauer und der Kalorienverbrauch, aber auch Informationen über den Trainierenden, wie Name, Trainingsdatum und verwendeter Fitness-Tracker, dargestellt.

Für detaillierte Informationen, sowie eine grafische Darstellung des Trainingsverlaufs, mit Rundenzeiten und Trainingstrecken, wählt man das jeweilige Training aus und gelangt auf die Trainingsdetailansicht. In dieser Übersicht wird der Zustand der Trainierenden zu der jeweiligen Zeit angezeigt.
Neben Daten wie Aufstieg, Abstieg, Schrittfrequenz und Kalorienverbrauch erzeugt der Fitness-Tracker, wie in Abbildung 2.2b, jeden Kilometer eine automatische Runde. Dort wird die Rundenzeit, die Herzfrequenz und das Tempo tabellarisch dargestellt. Durch



(a) Trainingsübersicht

(b) Trainingsdetailansicht

Abbildung 2.2: Polar Flow - Training

Drücken auf die Karte können diese Daten zu jedem Zeitpunkt der Strecke eingesehen werden.

Aktivitäts-Analyse Eine Übersicht der täglichen Aktivitäten kann neben den Trainingseinheiten (vgl. Abbildung 2.2a) in der *Feed* Ansicht eingesehen werden.
In der Übersicht werden neben dem prozentualen Status des täglichen Ziels, die Länge der Aktivitätszeit, die im Laufe des Tages verbrannten Kalorien und die Anzahl der gemachten Schritte angezeigt. Durch Klicken auf die Übersicht wird der detaillierte Tagesverlauf geöffnet. Wie in Abbildung 2.3a dargestellt, wird dort die Aktivitätsintensität zu einer bestimmten Tageszeit oder die Zeitspanne, in der sich der Nutzer in dem jeweiligen Intensitätsbereich aufgehalten hat, aufgezeigt. Polar unterteilt die Aktivitätsintensität in 6 verschiedene Stärken:

(a) Tagesaktivitätsverlauf

(b) Daten, die über den Tag gesammelt wurden.

Abbildung 2.3: Polar Flow - Tägliche Aktivitäten

Nicht an - Keine Aktivitätsdaten erfasst

Ruhe - Schlafen oder Ruhen im Liegen

Sitzen - Sitzen oder anderes passives Verhalten

Niedrig - Arbeit im Stehen, leichte Hausarbeit

Mittel - Spazierengehen und andere moderate Aktivitäten

Hoch - Joggen, Laufen und andere intensive Aktivitäten

Alle, von dem Fitness-Tracker gesammelten Daten, werden unterhalb des Intensitäts-Diagramms dargestellt (vgl. Abbildung 2.3b). Zusätzlich detektiert Polar anhand der Bewegung der Handgelenke, sobald der Nutzer schläft. Polar unterscheidet zwischen erholsamem und unruhigem Schlaf, wodurch das Schlafverhalten des Nutzers ermittelt werden kann.

2.2.2 AccessLink

AccessLink, als Polar-API (vgl. Kapitel 3.2), ist der Zugangspunkt zu den Trainings- und Tagesaktivitätsdaten, die von den Polar Fitness-Trackern aufgezeichnet wurden. AccessLink unterstützt JSON und XML als Austauschformat und verwendet als Authentifizierungsprotokoll das in Kapitel 3.3 vorgestellte OAuth2-Protokoll. Auf den Aufbau und die grundlegende Vorgehensweise der Kommunikation mit AccessLink wird im Kapitel 5.1 näher eingegangen.

2.3 Vergleichbare Fitness-Tracker Hersteller

Um Polar besser kennen zu lernen, wird in diesem Kapitel die Konkurrenz vorgestellt. Zu den bekanntesten Herstellern von Fitness-Trackern gehören neben Polar *Garmin* und *Fitbit*. In diesem Kapitel werden die Hersteller kurz vorgestellt und anschließend ihre Schnittstelle für Drittanbieter mit der von Polar verglichen. Die Funktionen der Fitness-Tracker der verschiedenen Hersteller sind ähnlich und wurden bereits in Kapitel 2.1.1 aufgezeigt.

2.3.1 Garmin

Garmin, mit Sitz in der Schweiz, besitzt verschiedene Anwendungsbereiche. Neben Navigationsgeräten für Automobile, Marine und Luftfahrt bietet Garmin ebenfalls Kameras, sowie Fitness- und Outdoorgeräte an. Garmin spezialisierte sich jedoch von Beginn an auf Navigation. Mit der Einführung von Fitness-Tracker hat Garmin 2004 ihr Repertoire um ein weiteres Navigationsgerät erweitert [9].

Die Garmin-API unterscheidet sich von der Polar hauptsächlich in der Hinsicht, dass Garmin für die täglichen Aktivitäten und die Trainingseinheiten des Nutzers zwei verschiedene Schnittstellen verwendet:

- *Garmin Connect* liefert die Trainingseinheiten der neueren Fitness-Trackern in FIT-Format. Ältere Modelle liefern GPX- und TCX-Daten (vgl. Kapitel 3.4).

- *Health API* liefert die täglichen Aktivitäten den Nutzers mit Daten wie Schritte, Schlafrhythmus, Kalorienverbrauch und der durchschnittlichen Herzfrequenz.

2.3.2 Fitbit

Fitbit ist ein relativ junges Unternehmen und wurde 2007 von James Park und Eric Friedman in Amerika gegründet. Sie spezialisierten sich von Anfang an auf die Herstellung von Fitness-Trackern.

Fitbit bietet dem Nutzer im Gegensatz zu Polar und Garmin die Möglichkeit eigene Anwendungen für den Fitness-Tracker zu kreieren. Dafür stellt Fitbit 3 APIs zur Verfügung:

- Die *Device-API* ist in der Lage mit dem Fitness-Tracker zu kommunizieren. Auf die API kann nur mit Anwendungen zugegriffen werden, die direkt auf dem Fitbit Gerät laufen. Sie stellt die Daten des GPS, des Displays und der Sensoren zur Verfügung und hat die Möglichkeit, diese an das Smartphone (Companion) zu schicken.
- Die *Companion-API* läuft auf dem mobilen Gerät und kann mit der Device-API kommunizieren. Durch die Companion-API ist die Anwendung in der Lage über das Smartphone auf das Internet zugreifen zu können.
- Durch die *Setting-API* ist der Entwickler in der Lage seine Anwendung für den User konfigurierbar zu machen. Die Anwendungseinstellungen werden in *JSX* geschrieben.

Zusätzlich besitzt Fitbit wie Polar und Garmin eine *Web-API*, die es dem Client ermöglicht auf die Daten des Nutzer zugreifen zu können. Dadurch kann auf die täglichen Aktivitäten, Trainingseinheiten und physischen Informationen zugegriffen werden. Fitbit liefert dabei weder GPX- noch FIT-Dateien, sondern eine TCX-Datei.

3

Technische Grundlagen

3.1 Ionic 3

Dieses Kapitel beschäftigt sich damit, *Ionic 3* in eine Sparte der *Mobile Apps* einzusortieren und den Aufbau, sowie die Funktionsweise zu erläutern.

3.1.1 Einordnung in Mobile Apps

Smartphones sind heutzutage mit verschiedensten Sensoren ausgestattet. Ein Beschleunigungssensor, GPS, eine Kamera und das Gyroskop sind nur einige davon. Mit diesen Sensoren lassen sich eine Vielfalt von unterschiedlichen Mobile Apps entwickeln. Ein großes Problem bei der Entwicklung solcher Mobile Apps ist es, auf die Schnittstellen dieser Sensoren zugreifen zu können. Dabei gibt es verschiedene Herangehensweisen mobile Apps zu entwickeln und die Sensorik zu nutzen. Im Folgendem werden die wichtigsten Mobile Apps erläutert: *Native*, *Web*-, *Cross-Plattform*- und *Hybride* Apps.

Native Apps

Smartphones von Apple, Microsoft oder einem Android-Smartphone-Hersteller besitzen eine Programmierschnittstelle, die von nativen Apps direkt genutzt werden. Durch die direkte Nutzung der Smartphone-Hardware und -Software laufen native Apps effizienter und schneller auf der jeweiligen Plattformen. Das bedeutet jedoch auch, dass, wenn eine App auf verschiedenen Plattformen (z.B. IOS, Android) funktionieren soll, für jede Plattform eine eigene App geschrieben werden muss [10].

Um dieses Problem zu umgehen, können plattformunabhängige Apps verwendet werden. Wie der Name schon sagt, sollen Apps möglichst unabhängig von ihrer Plattform entwickelt und dadurch auf verschiedenen Geräten verwendbar gemacht werden. Doch auch da gibt es verschiedene Herangehensweisen.

Web-Apps

Web-Apps (webbasierte Apps) sind keine auf dem Smartphone über einen App-Store installierte Apps, sondern über den Geräte-Browser aufgerufene Webseiten, die eine für Smartphones optimierte Benutzeroberfläche besitzen. Das heißt, dass diese Apps mit Javascript/Typescript, HTML5 und CSS3 geschrieben werden können, wodurch plattform-spezifische Anpassungen leicht realisierbar sind [10]. Dadurch sind diese Apps nicht direkt von der Programmierschnittstelle der Hersteller abhängig, sind leichter wartbar und können auf allen Plattformen verwendet werden. Dies bringt jedoch den Nachteil, dass sie nur auf die Hardware-Komponenten des Geräts zugreifen können, für die der Browser eine Schnittstelle bereitstellt. Zusätzlich sind sie weniger zuverlässig und sicher, da sie im Cache des Browsers gespeichert werden und für zusätzliche Funktionalitäten eine Internetverbindung benötigen.

Cross-Plattform-Apps

Eine *Cross-Plattform-App* ist, wie eine Web-App, plattformunabhängig. Jedoch werden Cross-Plattform-Apps auf dem Gerät installiert und nicht über den Web-Browser aufgerufen. Sie werden dabei von Frameworks, wie Xamarin oder React Native, in C# oder anderen plattformunabhängigen Programmiersprachen entwickelt. Dabei wird die Benutzeroberfläche mit Platzhaltern versehen, die während der Laufzeit plattformspezifisch verwandelt werden. Diese Umwandlung muss für jede Plattform seperat geschrieben werden. Dadurch besitzen Cross-Plattform Apps ca. 75% gemeinsamen Quellcode [11].

Hybrid-Apps

Ähnlich wie Cross-Plattform-Apps werden Hybrid-Apps auf dem Gerät installiert und verhalten sich somit wie eine native Anwendung. Der Unterschied ist jedoch, dass Hybrid-Apps mit webbasierten Programmiersprachen (z.B. Javascript/Typescript, HTML, CSS) entwickelt werden. Sie werden auf einem *WebView* dargestellt, welcher ermöglicht das Hybrid-Apps auf jeder Plattform und sogar auf dem PC laufen können. Dadurch müssen sie nur einmal entwickelt werden und sind danach auf jedem Endgerät einsatzbereit. Da nicht die nativen UI-Elemente der jeweiligen Plattform verwendet werden, büßen sie jedoch das plattformspezifische *Look and Feel* ein und haben zusätzlich eine schlechte Rechenleistung, was die Hybrid-Apps vergleichsweise langsam starten und agieren lässt [10]

Im Gegensatz zu den Web-Apps, kann eine Hybrid-App mit der Programmierschnittstelle des Betriebssystems kommunizieren. Dadurch können sie auf die Sensoren und Services der vorhandenen Plattform zugreifen.
Für die Verbindung von Ionic 3 als Hybrid-App und der Programmierschnittstelle ist *Cordova* zuständig.

3.1.2 Cordova

Cordova ist, wie in Abschnitt 3.1.1 bereits erwähnt, für die Kommunikation zwischen WebView und der Programmierschnittstelle der jeweiligen Plattform zuständig. Dies bewerkstelligt Cordova, indem es Plugins zur Verfügung stellt, die auf Sensoren oder Services der Programmierschnittstelle zugreifen können. Dadurch können native Funktionen über Javascript bzw. Typescript angefragt und verwendet werden.

In Abbildung 3.1 wird die Architektur einer Cordova Applikation dargestellt. Sie besteht aus *Cordova Plugins*, einer *Web App* und einem *WebView*. Die Web App baut sich aus HTML, Typescript und CSS zusammen und bildet mit den Ressourcen die Applikation, die dem Nutzer präsentiert wird. Diese Web App läuft nun auf einer WebView, welche eine Schnittstelle zu den Cordova Plugins besitzt.
Dabei besitzt Cordova zum einen die *Core Plugins*, die es ermöglichen auf Geräte-

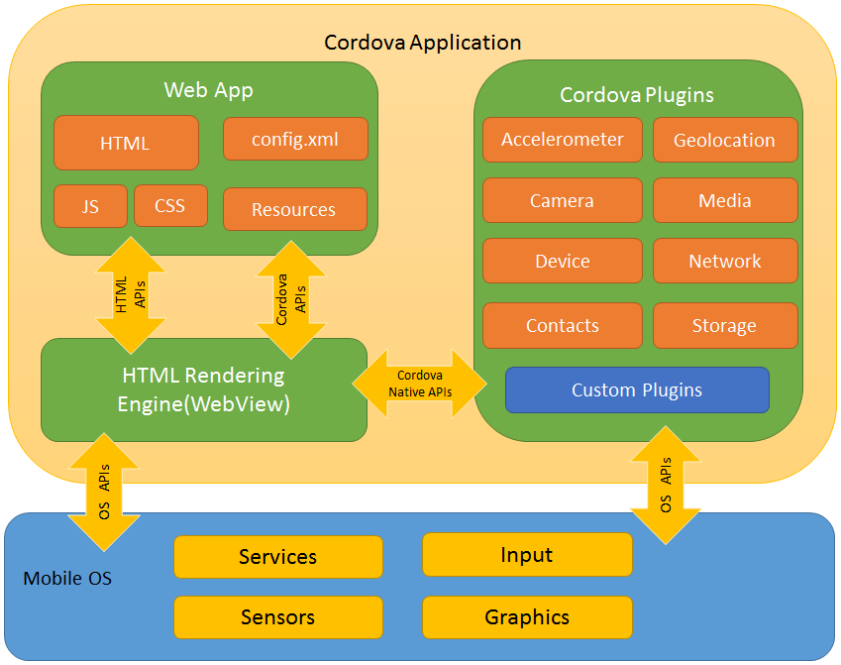


Abbildung 3.1: Umfassende Übersicht der Cordova-Architektur

Ressourcen wie Kamera oder Batterie zugreifen zu können und zum anderen stellt Cordova dem Nutzer die Möglichkeit bereit eigene Plugins zu entwerfen oder Plugins von Drittanbietern zu importieren.

3.1.3 Architektur

Neben Cordova basiert Ionic 3 auf *Angular*. Während des Releases 2013 baute Ionic auf *AngularJS* auf, doch mit Angular 2 bzw. 3 änderte sich das zu *Angular 4*. Dadurch besitzt eine Ionic 3 App die gleiche Architektur wie eine Angular App. Diese Kapitel beschreibt diese Architektur und ihre Vorzüge.
Angular ist ein Framework zum Erstellen von Client-Anwendungen in HTML und TypeScript. Es implementiert eine Reihe an TypeScript Bibliotheken, die in die Anwendung importiert werden können. Dabei setzt sich Angular aus Modules, Components, Services und Directives zusammen. Im Folgenden wird näher darauf eingegangen:

Modules

Angular Anwendungen sind modular und besitzen als grundlegendes Strukturelement *NgModules*. NgModule beinhalten Components, Services, Provider und andere Dateien, die einer Anwendung zugehörig sind. Jede Angular Anwendung besitzt dabei mindestens ein Modul, das *root module* (nach Konvention *AppModule* genannt). Durch *bootstrapping* des AppModules wird die Anwendung gestartet. Ein NgModule wird definiert durch einen `@NgModule`-*Decorator*, dessen Metadaten-Objekt `declarations`, `exports`, `imports`, `providers` und das `bootstrapping` des Components beinhaltet (vgl. Codebeispiel A.1). Im Folgenden werden die Eigenschaften der verschiedenen Metadaten vorgestellt:

declarations
: Beinhaltet die zu dem NgModule zugehörigen Components.

exports
: Nennt eine Teilmenge der Klassen, die für andere NgModule sichtbar und wiederverwendbar sind.

imports
: Exportierte Klassen anderer Module können hier importiert werden.

providers
: Hier werden *Services (Providers)* gelistet, die in allen Bereichen der App zugänglich sein sollen.

bootstrap
: Das *root component*, das alle *views* der Anwendung verwaltet, wird hier an das Modul gebunden.

Components

Components definieren die Benutzeroberfläche (*views*) und deren Logik. Dabei besitzt jede Angular Anwendung mindestens ein Component, das *root component*. Ein Component wird durch einen `@Component`-*Decorator* definiert und enthält spezifische

Metadaten. In den Metadaten wird die Benutzeroberfläche als *HTML-Template* eingebunden. Dabei kann der HTML-Code direkt in das Metadaten-Objekt geschrieben werden oder auf das HTML-Dokument verweisen, das die Benutzeroberfläche bildet. Das Codebeispiel A.2 zeigt ein solches Component. Ionic 3 liefert mit den *Ionic Components* vorgefertigte Components, die verwendet werden können. Dazu gehören häufig benötigte Components wie Menüs, Cards und Floating Action Buttons. Im Folgendem werden die Einstellmöglichkeiten der selbst erstellten Component-Metadaten vorgestellt:

selector

Durch den *selector* wird dem HTML-Template die Möglichkeit gegeben, eine Instanz der Benutzeroberfläche des Components mit dem zugehörigem HTML-Tag zu erstellen.

templateUrl

Hier kann die relative Adresse des entsprechenden HTML-Templates angegeben werden.

template

Als Alternative zu *templateUrl* kann hier der HTML-Code direkt in das Metadaten-Objekt kodiert werden.

providers

Benötigt das Component die Funktionalität eines Services, können diese hier angegeben werden und dadurch von dem Component verwendet werden.

Services

Ein Component nutzt *Services* (auch Provider genannt) um spezifische Funktionalitäten einzubinden. Dabei sind Services nicht direkt an die Components gebunden, sondern können auch von anderen Components und auch Services verwendet werden. Zu den Funktionalitäten eines Services gehören die Kommunikation mit einem Server oder mit einer Datenbank.
Der `Injectable`-*Decorator* kennzeichnet den Service als einen solchen. Der Code

A.3 zeigt ein kurzes Beispiel eines Services und A.1 wie ein Service in ein Component importiert wird.

Directives

Directives sind im Template dafür zuständig, durch Hinzufügen, Entfernen oder Manipulieren der Elemente das Layout zu strukturieren. Die Directives werden dabei an ein HTML-Element angefügt und wirken sich so auf diese und deren Nachkommen aus. Angeführt von einem Asterisk (*) sind die am häufigsten verwendete Directives `*ngIf`, `*ngFor` und `[ngSwitch]` mit den zugehörigen`*ngSwitchCase`. Wie die Namen schon suggerieren handelt es sich dabei um If-Abfragen, For-Schleifen und Switch-Case-Anweisungen. Wie die Directives in der Anwendung aussehen zeigt das Code-Beispiel A.4.

Data binding

Ohne die Verwendung eines Frameworks ist data binding, also die Datenverbindung zwischen Component und Template, sehr aufwendig. Angular nimmt dem Nutzer einige Arbeit ab. Die Abbildung 3.2 zeigt die Möglichkeiten, wie das Component mit dem Template, aber auch das Template mit dem Component kommunizieren kann.

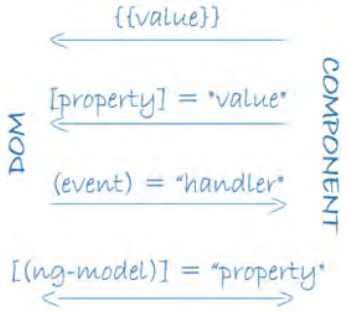


Abbildung 3.2: Databinding

{{value}}

Durch die Verwendung von geschweiften Doppelklammern können Component-

Objekte (hier `value`) in das Template importiert und verwendet werden (Interpolation).

[property] = "value"

Soll ein Attribut eines HTML-Tags verändert werden, kann dies mit Hilfe von *Property binding* bewerkstelligt werden.

(event) = "handler"

Durch das Klicken auf ein Template-Element, das eine solchen Event-Handler besitzt, wird die Methode, die unter `handler` angegeben ist, aufgerufen. Dabei kann mit der Verwendung von *Directives* Objekte direkt an die Methode übergeben werden (siehe Beispiel A.4).

[(ng-modul)] = "property"

Angular ist in der Lage *Two-way data binding* durchzuführen. Dabei wird das Component-Objekt `property` sowohl im Template, als auch im Component verändert.

Speicher

Ionic bietet verschiedene Möglichkeiten Daten lokal zu speichern. Im Kapitel 5.3 wird der Datenbankentwurf vorgestellt. Dieser orientiert sich sehr an der Struktur der Daten, die die Polar-API liefert. Daher wird ein Key-Value-Speicher verwendet, um die Daten lokal zu speichern. Im Folgenden werden zwei verschiedene Herangehensweisen der Datenspeicherung in Ionic 3 mit einer Key-Value-Struktur vorgestellt.

Ionic Storage

Ionic liefert ein eigenes Speichermodul (`IonicStorageModule`) um JSON-Objekte zu speichern. Das Modul verwendet eine Vielzahl an Speichermethoden. Dabei wird die SQLite-Storage priorisiert, da es sich dabei um die stabilste und am weitesten verbreitete Form der dateibasierten Datenbank handelt [12]. Bei Verwendung im Web oder als Web App greift Ionic Storage auf IndexedDB, WebSQL und LocalStorage zurück.

Bei Verwendung des Ionic Storages laufen alle Speicherzugriffe asynchron ab.

Daher liefern die Zugriffe die Daten in `Promises` zurück. Das Codebeispiel A.5 zeigt Beispiele von Speicherzugriffen.

LocalStorage

Der LocalStorage ist ein *HTML5 Web Storage* und daher nicht von Ionic. Es wird jedoch, wie oben bereits erwähnt, von Ionic verwendet. Mit Hilfe des Web Storages können Daten lokal gespeichert werden. Dies kann mit zwei verschiedenen Storages verwirklicht werden. Der `SessionStorage` speichert, wie der Name schon sagt nur Daten einer Sitzung und verwirft diese nach Verlassen der Sitzung. Um Daten persistent zu speichern wird der `LocalStorage` verwendet. Dabei liefert der LocalStorage bei Anfragen die Daten initial und nicht in einem Callback eines Promises. LocalStorage nimmt wiederum keine JSON-Objekte als Parameter, doch kann dies mit den JSON.stringify()- und JSON.parse()-Methoden umgangen werden (vgl. Codebeispiel A.7).

3.2 REST API

REST steht für **Re**presentational **S**tate **T**ransfer und wurde 2000 von Roy Thomas Fielding definiert [13]. Dabei bezeichnet REST einen Architekturstil eines Client-Servers-Web-Services. Clients nutzen zur Kommunikation mit einem Webservice eine Programmierschnittstelle (englisch: application programming interface - API). Diese API kann von Clients angesprochen werden, um Ressourcen anzufordern [14]. Wie in Abbildung 3.3 dargestellt, ist die API der Einstiegspunkt, um mit dem Web-Service zu kommunizieren.

Abbildung 3.3: Web-API

Die Kommunikation mit der API läuft über HTTP-Requests und HTTP-Responses ab. Um eine Ressource vom Web-Service zu erhalten, muss der Client einen HTTP-Request

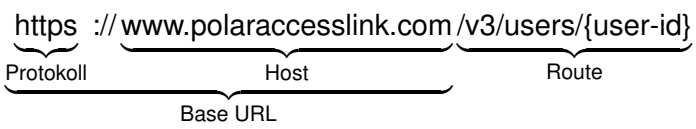


Abbildung 3.4: Beispiel URL

an die API schicken. Jede Ressource des Web-Services ist eindeutige unter ihrer URI (Uniform Resource Identifier) zu finden bzw. unter ihrer URL (Uniform Resource Locator), welche die Ressource eindeutig adressiert, daher muss der Request diese URL enthalten. Abbildung 3.4 zeigt, wie eine solche URL aufgebaut sein kann.

Die API stellt dem Nutzer eine *Base URL* zur Verfügung an die eine *Route* angefügt werden kann. Diese Base URL mitsamt der Route verweist auf eine eindeutige Ressource, auf die die API im Backend zugreifen und dem Client zurückschicken kann.

HTTP kann dabei auf verschiedene HTTP-Methoden zurückgreifen, die dem Server Informationen darüber vermitteln, was mit dem Request anzufangen ist:

GET

Mit der GET-Methode können Ressourcen vom Server angefragt werden. Dabei werden die Parameter nach einem Fragezeichen an die URL angefügt.

POST

Will der Client eine Ressource auf dem Server modifizieren oder updaten, verwendet er die POST-Methode. Die Parameter werden an den Header angehängt und nicht an die URL angefügt.

DELETE

Löscht die angegebene Ressource auf dem Server.

PUT

Ähnlich wie POST kann mit PUT eine Ressource erstellt und modifiziert werden. Im Gegensatz zu POST ist PUT idempotent und erstellt daher bei mehrmaligen Anfragen die Ressource nicht jedesmal neu, sondern aktualisiert die bereits vorhandene.

Neben oben genannten Methoden gibt es weitere wie HEAD und OPTIONS. Diese werden jedoch nicht von der Polar-API verwendet und daher nicht weiter erläutert.

Ressourcen können mit den HTTP-Methoden angefragt, verändert oder gelöscht werden. Wie die Ressourcen vor unberechtigten Zugriffen geschützt werden können, wird in Kapitel 3.3 beschrieben.

3.3 OAuth2-Authentifizierung

Um auf eine geschützte Ressource auf einem Server zugreifen zu können, verwendet der *Client* in einer traditionellen Client-Server Authentifizierung seine eigenen *Credentials*. Mit Hilfe dieser Credentials kann der Client auf seine Daten zugreifen. Sollen nun Dritte die Möglichkeit erhalten auf diese Daten zugreifen zu können, muss der Client seine eigenen Ressource-Credentials an diesen weitergeben. Das bringt jedoch einige Risiken mit sich:

- Der Drittanwender muss die Credentials lokal für zukünftige Serverzugriffe speichern.
- Server müssen Passwort-Authentifizierungen unterstützen, wodurch eine Sicherheitslücke entsteht.
- Der Client kann keine zeitliche Beschränkung der Zugriffsdauer einrichten.
- Will der Client einem Drittanweder den Zugriff verweigern, dann muss er es allen Drittanwendern verweigern.

OAuth2 versucht dem entgegenzuwirken, indem die Rolle des Clients von dem des Besitzers (*Ressource-Owner*) gelöst wird. Der Client fragt beim Ressource-Owner mit seinen *Client-Credentials* an, um auf eine geschützte Ressource (*Protected Resource*) zugreifen zu können. Gewährt dieser den Zugriff, kann der Client mit dem erhaltenen *Authorization Code* beim *Authorization Server* einen *Access Token* anfordern, mit dem der Client auf die geschützte Ressource zugreifen kann [15]. Die Abbildung 3.5 zeigt diesen Vorgang als Sequenzdiagramm.

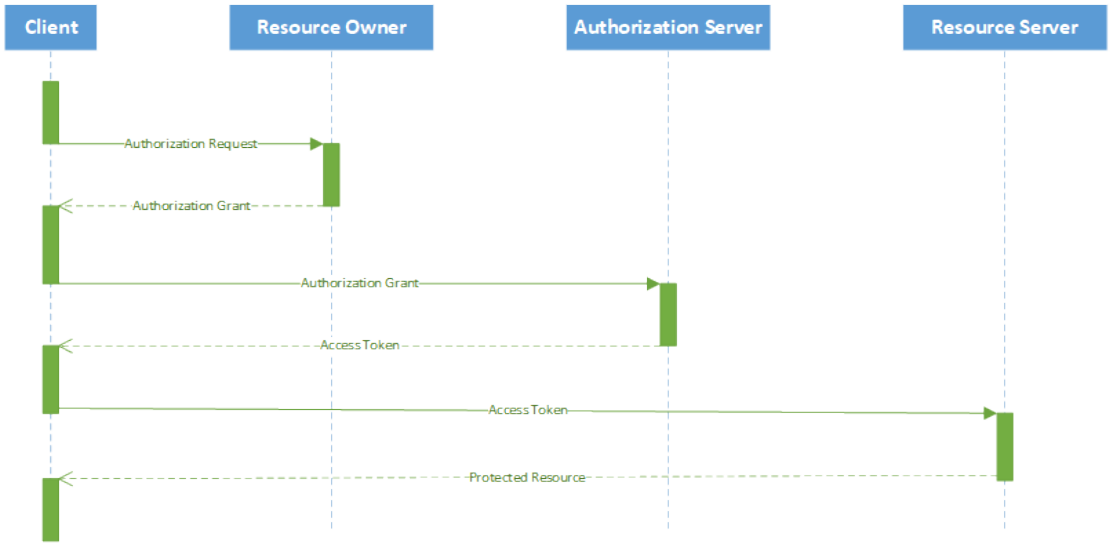


Abbildung 3.5: Ablauf der OAuth2-Authentifizierung

Dabei kann der Client verschiedene Berechtigungen anfragen. Diese Berechtigungen werden, neben einer zeitlichen Begrenzung der Berechtigungen, in einem Access Token gespeichert. Dieser Access Token enthält alle Session-Informationen, die den Client auszeichnen. Die Verwendung eines Access Tokens bringt den Vorteil, dass der Nutzer nicht für jede Anfrage alle Credentials schicken muss, sondern nur den Access Token, der als Hash leicht zu validieren ist. Im Codebeispiel 3.1 wird ein solcher Access Token dargestellt.

```
1 {
2     access_token : "2e0f70c937505dedbaa73e3a800a045a"
3     expires_in : 473039999
4     token_type : "bearer"
5     x_user_id : 40003147
6 }
```

Listing 3.1: Beispiel Access Token

3.4 GPS- und trainingsspezifische Dateiformate

Um nach einer Trainingseinheit die Daten zu speichern, werden von Polar verschiedene Dateiformate verwendet. In diesem Kapitel werden die Datenformate GPX, TCX, GeoJSON und FIT vorgestellt und erläutert, wofür sie verwendet werden können.

GPX

GPX (Global Positioning System Exchange Format) wurde 2002 von *Topografix* veröffentlicht und ist ein, auf XML basiertes Datenformat zum Speichern und Austauschen von GPS-Daten. GPX definiert hierzu einen Satz von Datentypen, die zur Formulierung von GPS-Daten und komplexen geografische Daten verwendet werden. Es wird bereits von vielen Software Anwendungen (z.B. Google Earth, Polar, Garmin) als Standard-Austauschformat für GPS-Daten verwendet [16].

Eine GPX-Datei hat als Root Element den `gpx`-Typ, gefolgt von den Metadaten (`metadata`), welche Informationen wie Namen (`name`), Beschreibung (`desc`) und Author (`author`) enthalten. Die wichtigsten, von GPX definierten Datentypen sind Wegpunkte (`wpt`), Routen (`rte`) und Tracks (`trk`), die die jeweiligen Koordinatendaten halten. In den folgenden Abbildungen wird der Aufbau der jeweiligen Datentypen vorgestellt. Die Struktur einer GPX-Datei wird im Codebeispiel A.8 dargestellt:

Waypoint (`<wpt>`)
Der `wpt`-Type repräsentiert einen Wegpunkt - einen interessanten Ort.

```
1 <wpt lat="latitude" lon="longitude">
2     <!-- Attribute des Wegpunktes -->
3 </wpt>
```

Listing 3.2: Wegpunkte einer GPX-Datei

Route (`<rte>`)

Der `rte`-Typ repräsentiert eine Route - eine geordnete Liste von Wegpunkten, die Wendepunkte markieren.

```
<rte lat="latitude" lon="longitude">
    <!-- Attribute der Route -->
    <rtept lat="latitude" lon="longitude">
        <!-- Attribute des Routenpunkts -->
    </trept>
</rte>
```

Listing 3.3: Route einer GPX-Datei

Track (`<trk>`)

Der `trk`-Typ repräsentiert einen Track - eine geordnete Liste, die einen Pfad beschreibt. Der `trkseg`-Typ besitzt eine Liste aus Trackpunkten (`trkpt`) mit den jeweiligen Attributen.

```
<trk lat="latitude" lon="longitude">
    <!-- Attribute des Tracks -->
    <trkseg>
        <trkpt lat="latitude" lon="longitude">
            <!-- Attribute des Trackpunktes -->
        </trkpt>
    </trkseg>
</trk>
```

Listing 3.4: Track einer GPX-Datei

Die Datentypen können verschiedene Attribute besitzen, die den Zustand des jeweiligen Standortes besser beschreiben. In der folgenden Auflistung werden Beispiele solcher Attribute beschrieben:

Elevation `<ele>`	Höhe des Punktes (in Meter)
Name `<name>`	Name des Wegpunktes
Description `<desc>`	Beschreibung des Wegpunktes
Time `<time>`	Datum und Uhrzeit in ISO8601-Format
Declination `<magvar>`	Abweichung der Magnetnadel von der Nordrichtung

TCX

Garmin hat TCX (Training Center XML) im Jahr 2007 vorgestellt. Ähnlich wie GPX speichert TCX GPS-Daten. Im Gegensatz zu GPX wird jedoch TCX als Aktivität behandelt und ist weniger eine Ansammlung von GPS-Punkten, als vielmehr eine Trainingszusammenfassung mit Daten wie Herzfrequenz, Kalorien, Schrittfrequenz oder Dauer. Dadurch ist TCX optimal zur Überwachung des Trainingsverlaufs mit detaillierten Übersichtsdaten zu jedem Zeitpunkt der Strecke.

Im Codebeispiel A.9 wird der Aufbau einer TCX-Datei beschrieben. Eine `Activity` beschreibt eine Trainingseinheit und wird in `Laps` gegliedert. Jede `Lap` fasst ihren Daten wie Gesamtzeit (`TotalTimeSeconds`) und durchschnittlichen Herzfrequenz (`AverageHeartRateBpm`) zusammen. Die jeweiligen `Tracks` beschreiben anschließend mit ihren `Trackpoints` die Wegpunkte und ihre eigenen Herzfrequenz- und Geschwindigkeitsdaten. Eine Ausschnitt der `Tracks` wird im Codebeispiel A.10 gezeigt.

FIT

Neu bei Polar ist das *Flexible and Interoperable Data Transfer* (FIT)-Protokoll. Im Gegensatz zu TCX und GPX besitzt FIT kein XML-Format, sondern ist ein interoperables, binäres Datentransfer-Protokoll, wodurch es im Gegensatz zu den XML-Formaten weniger Speicherplatz benötigt. Wie auch TCX ist es dazu entworfen, um Daten von Sport-, Fitness oder Gesundheitstrackern zu speichern und zu teilen.

Vorteile dieses Protokolls sind:

- Kann auf verschiedenen Plattformen verwendet werden.

- Ist von kleinen eingebetteten Systeme auf große Datenbanken skalierbar.
- Kann erweitert werden und in Funktionalität wachsen.

Das FIT-Protokoll besteht aus einer vorgegebener Datenstruktur, einer globalen Liste aus *FIT messages* mit den jeweilig definierten Datentypen und einem Software Development Kit (SDK) zur Konfiguration von Produkten, um die erforderliche Programmierumgebung zu erschaffen.

Das Protokoll definiert den Prozess des Datentransfers von Gerät zu Gerät. Die gemessenen Daten werden zusammen mit dem *Produkt Profil* in einer FIT-Datei kodiert. Diese FIT-Datei kann von der Programmierumgebung des Empfänger dekodiert und strukturiert werden [17]. Die Abbildung 3.6 zeigt den Aufbau einer solchen FIT-Datei.

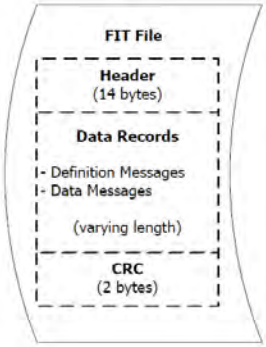


Abbildung 3.6: Aufbau einer FIT-Datei

Eine FIT-Datei besteht aus dem *Header*, den *Data Records* und der CRC (Zyklische Redundanzprüfung, englisch *cyclic redundancy check*).

Header

Im Header werden in den ersten 14 Bytes Informationen über die FIT-Datei gespeichert. Darunter sind Daten wie die Versionsnummer des Protokolls und die Länge des Data Records.

CRC

Die letzten 2 Bytes der FIT-Datei sind für die *Zyklische Redundanzprüfung* (CRC) reserviert, die dafür sorgt, dass Fehler während der Übertragung erkannt und gegebenenfalls korrigiert werden können.

Data Records

Die Data Records sind der Hauptinhalt einer FIT-Datei. Es existieren zwei Arten von Data Records: **Definition Message** und **Data Message**. Dabei definieren *Definition Messages* den Datentyp der *Data Messages*, die die Streckeninformationen des jeweiligen Typen speichern.

GeoJSON

GeoJSON ist, wie der Name impliziert, ein offenes Format um grafische Daten in JSON-Format zu speichern.

Wie im Codebeispiel A.11 verdeutlicht, besteht ein GeoJSON-Objekt aus einer `Geometry`, einem `Feature` oder einer Sammlung von Features (`FeatureCollection`) [18].

Geometry Object

`Geometry` verwendet verschiedene geometrische Typen wie `Point`, `Multipoint`, `LineString`, `MultiLineString`, `Polygon` und `MultiPolygon` um geographische Positionen und Routen darzustellen.

Feature Object

`Feature` besitzt eine `"type"`-Variable mit dem Wert "Feature", sowie ein `Geometry`-Objekt und eine `"properties"`-Variable, welche ein beliebiges JSON-Objekt enthalten kann.

FeatureCollection Object

Unter der Variable `features` der `FeatureCollection` wird eine JSON-Array gespeichert. Jeder Eintrag ist ein `Feature`.

Vergleich von GPX, TCX, FIT und GeoJSON

Da GPX hauptsächlich standortorientierte Daten speichert, ist es prädestiniert für die Visualisierung von Streckendaten. Will man jedoch zu einem bestimmten Zeitpunkt der Trainingsstrecke die Herzfrequenz, Distanz oder die bisher verbrannten Kalorien einsehen, reichen die von GPX gespeicherten Daten nicht mehr aus. Für detaillierte

Streckeninformationen sollte daher TCX oder FIT verwendet werden. Da jedoch für Polar, die FIT Anbindung noch in der Beta-Phase ist, sollte man vorerst noch davon absehen und stattdessen TCX verwenden.

	FIT	TCX	GPX
GPS Position	✓	✓	✓
Zeitstempel	✓	✓	✓
Herzfrequenz	✓	✓	
Kadenz	✓	✓	
Geschwindigkeit	✓	✓	✓
Distanz	✓	✓	✓
Kalorien	✓	✓	
Temperatur	✓		
Herzfrequenzvariabilität (HRV)	✓	✓	
Trainingszusammenfassungen	✓	✓	
Geräteinformationen	✓	✓	
Wetter Kondition	✓		
Laps	✓	✓	

Tabelle 3.1: Tabellarischer Vergleich von GPX, TCX und FIT

Da GeoJSON in der `properties`-Variablen ein beliebiges JSON-Objekt speichern kann, können auch beliebige Daten zu den jeweiligen Koordinaten gespeichert werden. Zusätzlich werden einige Javscript-Plugins zur Verfügung gestellt die GPX und TCX in einen GeoJSON parsen können (vgl. Kapitel 6.1). Die Testanwendung verwendet daher nur das GeoJSON-Format um Streckeninformationen darzustellen.

4

Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die Testanwendung definiert. Dabei legen die funktionalen Anforderungen fest, wozu die Anwendung imstande sein soll. Die nicht-funktionalen Anforderungen beschreiben, wie gut diese Anforderungen umgesetzt werden sollen.

4.1 Funktionale Anforderungen

In Tabelle 4.1 werden die funktionalen Anforderungen an die Testanwendung und die jeweilige Priorisierung mit Werten von 1 (unwichtig) bis 5 (unabdingbar) formuliert.

Abkürzung	Anforderung	Priorität
FA01	Anmeldung und Abmeldung des Polar-Nutzers	5
FA02	Lokale Datensicherung	5
FA03	Übersicht der täglichen Aktivitäten	4
FA04	Anzeigen der täglichen Aktivitäten	5
FA05	Löschen der täglichen Aktivitäten	2
FA06	Übersicht der Trainingseinheiten	4
FA07	Anzeigen der Trainingseinheiten	5
FA08	Löschen der Trainingseinheiten	1
FA09	Anzeigen des Nutzer-Fitnesszustands	4
FA10	Anzeigen des Nutzer-Profils	3

Tabelle 4.1: Funktionale Anforderungen

FA01 Anmeldung und Abmeldung des Polar-Nutzers
Der Nutzer soll sich bei der Anwendung mit seinem Polar-Benutzerkonto anmelden können. Dabei bleibt der Nutzer so lange angemeldet, bis er sich abmeldet.

FA02 Lokale Datensicherung
Alle Nutzerdaten sollen lokal auf dem Gerät gespeichert werden. Selbst das Wechseln der Benutzerkonten soll dabei berücksichtigt werden. Die Testanwendung soll alle Daten, die von der Polar-API zur Verfügung gestellt werden, logisch darstellen, sodass der Nutzer eine strukturierte Übersicht dieser Daten hat.

FA03 Übersicht der täglichen Aktivitäten
Die vom Nutzer synchronisierten, täglichen Aktivitäten sollen auf dem Gerät übersichtlich dargestellt werden.

FA04 Anzeigen der täglichen Aktivitäten
Jede tägliche Aktivität kann in einer detaillierten und separaten Seite eingesehen werden. Dabei sollen alle von der API gelieferten Daten dargestellt werden.

FA05 Löschen der täglichen Aktivitäten
Eine tägliche Aktivität kann mit allen Daten gelöscht werden.

FA06 Übersicht der Trainingseinheiten
Jede Trainingseinheit des Nutzers wird in einer Übersicht dargestellt. Die Übersicht enthält die individuellen Daten dieser Trainingseinheit und ist somit für den Nutzer eindeutig.

FA07 Anzeigen der Trainingseinheiten
Dem Nutzer steht eine ausführliche Zusammenfassung der Trainingseinheit zur Verfügung. Dabei werden alle vom Fitness-Tracker erhobenen Daten, mitsamt der Laufstrecke dargestellt.

FA08 Löschen der Trainingseinheiten
Einzelne Trainingseinheiten können gelöscht werden ohne andere Trainingseinheiten zu beeinflussen.

FA09 Anzeigen des Nutzer-Fitnesszustands
Die physischen Daten des Nutzers werden im Verhältnis zueinander dargestellt, wodurch die Trainings-, bzw. Nutzerentwicklung deutlich wird.

FA10 Anzeigen des Nutzer-Profils
Alle von der API zur Verfügung stehenden, nicht-physischen Nutzerdaten können vom Nutzer eingesehen werden.

4.2 Nichtfunktionale Anforderungen

Die in Kapitel 4.1 beschriebenen funktionale Anforderungen bestimmen die Funktionen der Anwendung. In der Tabelle 4.2 werden nun die nichtfunktionalen Anforderungen dargelegt, die die Qualität der Umsetzung festlegen.

Abkürzung	Anforderung	Priorität
NFA01	Zuverlässigkeit	5
NFA02	Korrektheit	3
NFA03	Benutzbarkeit	4
NFA04	Funktionalität	5
NFA05	Effizienz	3
NFA06	Wartbarkeit	4

Tabelle 4.2: Nichtfunktionale Anforderungen

NFA01 Zuverlässigkeit
Die Testanwendung kann schnellstmöglich auf die Daten des Polar-Nutzers zugreifen und diese, sobald diese zur Verfügung stehen, fehlerfrei herunterladen, speichern und darstellen.

NFA02 Korrektheit
Alle Daten, die von der Testanwendung dargestellt und verarbeitet werden, werden unverfälscht wiedergegeben und ändern ihren Zustand während der aktiven Nutzung der Testanwendung nicht.

NFA03 Benutzbarkeit

Es ist ohne Vorwissen erkennbar, was in der Testanwendung dargestellt wird. Dabei die Bedienung logisch und es ist einfach ersichtlich, wo benötigte Informationen innerhalb der Testanwendung zu finden sind.

NFA04 Funktionalität

Alle, in Kapitel 4.1 formulierten Anforderungen, können abhängig von ihrer Priorisierung fehlerfrei ausgeführt werden.

NFA05 Effizienz

Die Kommunikation mit den Polar-Servern und die Speicherung der geladenen Daten verschwendet keine unnötigen Ressourcen und ist vom Zeitaufwand überschaubar. Zusätzlich wird durch Wiederverwendung von Code Redundanz vermieden.

NFA06 Wartbarkeit

Da die Testanwendung sehr von der Polar-API abhängig ist, kann schnell auf Änderungen der solchen eingegangen werden. Dies unterstützt die Struktur der Anwendung.

5

Konzeption und Entwurf

In diesem Kapitel werden die grundlegenden Ideen für den Umgang mit der vorhandenen Polar-API formuliert und entworfen. In Kapitel 5.2 wird auf die Struktur der Daten eingegangen, die Polar zu Verfügung stellt. Zum besseren Verständnis werden dafür Mock-Up's vorgestellt, die die Grundstruktur der dargestellten Daten mit Hilfe einer Testanwendung visualisieren sollen.

5.1 Vorstellung der AccessLink-API

Die Polar-AccessLink-API hat einen vorgeschrieben Ablauf der Kommunikation. Dieses Kapitel befasst sich mit der Kommunikation der Daten, die von Polar erreichbar sind und mit möglichen Darstellungsformen dieser Daten.

5.1.1 Ablauf der Kommunikation

Die Polar-API hat einen vorgeschriebenen Ablauf, wie die Kommunikation mit dem User vonstatten gehen soll. Dieser Kommunikationsablauf wird von *Polar AccessLink* definiert. AccessLink bietet Zugang zu den Daten, die von Polar-Geräten aufgezeichnet werden. Die Daten stehen den Clients nach einer OAuth2-Authentifizierung (vgl. Kapitel 3.3) in Form von JSON- und XML-Format zur Verfügung. AccessLink bietet dem Client dabei die Möglichkeit Beispielanfragen in verschiedenen Programmiersprachen und Protokollen einzusehen, darunter HTTP, NodeJS, Ruby, Python und Java.

5.1.2 Registrierung als Polar-Client

Um eine Kommunikation mit der API aufbauen zu können, muss sich der Nutzer mit seinem Benutzerkonto bei Polar als Client unter *Polar Admin* registrieren. Hier werden für den Client wichtige Details der Anwendung gespeichert, darunter der Anwendungsname, sowie eine Beschreibung und die Callback-Domain, die bei der Authentifizierung benötigt wird. Der Client kann dort die verschiedenen Datenpakete (Trainingsdaten, tägliche Aktivitäten, physikalische Informationen) abonnieren und diese später abrufen. Anschließend kann er sich mit Hilfe seiner erstellten *Client-ID* und *-Secret* an der Polar-API authentifizieren.

5.1.3 Authentifizierung

AccessLink benutzt zwei verschiedene Formen zur Authentifizierung. Zum einen die *User-Token-Authentifizierung* und zum anderen die *Client-Credentials-Authentifizierung*.

User-Token-Authentifizierung Alle Requests nach `/v3/users` und Unterressourcen werden mit der User-Token-Authentifizierung angefragt. Dafür wird der Token zur OAuth2-Authentifizierung verwendet (vgl. Kapitel 3.3) und an den Authorization-Header des Requests mit `Bearer <Token>` angefügt.

Client-Credentials-Authentifizierung Alle Anfragen, die nicht an `/v3/users` gerichtet sind (z.B. `/v3/notifications`), werden mit der Client-Credentials-Authentifizierung vollzogen. Dafür wird die Client-ID und das Client-Secret Base64 codiert und im Authorization-Header als `Basic <Base64encoded>` angefügt.

Mit einem korrektem Authorization-Header können nun API-Anfragen gestellt werden. Wie diese Anfragen ablaufen wird im nächsten Kapitel erläutert.

5.1.4 Transaktionen

Die Kommmunikation basiert auf Transaktionen (*Transactions*). Die Transaktionen bestehen aus Request und Response Paaren. Die grundlegende Logik des Kommunikationsablaufs wird in Abbildung 5.1 grob dargestellt und in Kapitel 6.2 genauer erläutert.

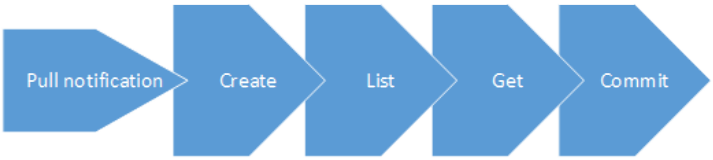


Abbildung 5.1: Kommunikation mit der Polar-API

Pull notifications - GET

Der erste Schritt zum Datenerhalt von AccessLink ist die Nachfrage nach der Verfügbarkeit neuer Daten (z.B. alle 10 Minuten oder auf Anfrage des Users). *Pull notifcations* liefern eine Liste von Stichwörtern bzw. URLs zum Anfragen der jeweiligen Daten. Mit diesen Stichwörter/URLs können Transaktionsketten erstellt werden.

Request `/v3 /notifications`

Create - POST

Die Transaktionskette beginnt mit dem Erstellen einer solchen. *Create* startet den Prozess zum Erhalt der Polar-Daten, indem die von *Pull notifications* gelieferten URLs aufgerufen werden.

Request `/v3 /users/{user-id}`
`/exercise-transactions`

Response `/v3 /users/{user-id}`
`/exercise-transactions/{transaction-id}`

List - GET

Nach dem erfolgreichen Initialisieren einer Transaktion können die darin enthaltenen Trainingseinheiten mit der bereitgestellten Transaktions-ID abgerufen werden.

Request `/v3 /users/{user-id}`
`/exercise-transactions/{transaction-id}`

Response `/v3 /users/{user-id}`
`/exercise-transactions/{transaction-id}`
`/exercise/{exercise-id}`

Get - GET

Mit *Get* können die Trainingsübersichtsdaten angefragt werden. Zusätzlich können weitere trainingsbezogene Daten wie GPX (/gpx), TCX (/tcx), FIT (/fit), Herzfrequenzbereiche (/heart-rate-zone) oder die jeweiligen Samples (/samples, /step-samples, /zone-samples) angefragt werden. Samples werden in Kapitel 5.2 vorgestellt und dementsprechend sieht die jeweilige Antwort aus.

Request `/v3 /users/{user-id}`
`/exercise-transactions/{transaction-id}`
`/exercise/{exercise-id}`
`/gpx`

Commit - PUT

Nachdem die Daten innerhalb einer Transaktion erfolgreich übertragen wurden, muss der Client die Beendigung des Vorgangs mit einem *Commit* abschließen. Falls dies nicht innerhalb von zehn Minuten nach Erstellen einer Transaktion geschieht, sind die Daten wie zuvor abrufbar und können erneut wie neue Daten geladen werden.

Request `/v3 /users/{user-id}`
`/exercise-transactions/{transaction-id}`

Mit der Registrierung, den Transaktionen und der Authentifizierung ist es dem Client möglich auf die Daten der API zugreifen zu können. Im Folgenden Kapitel werden diese Daten und ihre Verwendungsmöglichkeiten vorgestellt.

5.2 Mock-ups

Um die Datenmodelle der Polar-API zu visualisieren, werden in diesem Kapitel Mock-ups vorgestellt, die Darstellungsmöglichkeiten dieser Daten präsentieren.

Login

Der Loginvorgang des Users ist der Einstiegspunkt der Testanwendung. Dort muss der Nutzer dem Client die Berechtigungen erteilen auf seine Daten zuzugreifen, damit sie dargestellt werden können (vgl. Kapitel 3.3). Daher wird der Nutzer auf eine Webseite Polars weitergeleitet, wo ein vorgegebener Dialog ihm die Möglichkeit bietet, sich mit dem Polar-Benutzerkonto einzuloggen und dem Client anschließend die angefragten Berechtigungen zu erteilen. Die Abbildung 5.2 zeigt diesen Vorgang zum einen bei der Erstanwendung und zum anderen wenn der Nutzer bereits eingeloggt ist und den Account wechseln möchte.

Abbildung 5.2: Login Prozess

Nach einer erfolgreichen Authentifizierung kann der Client auf die Nutzerdaten zugreifen und sie dem Nutzer darstellen. Der gesamte Login-Vorgang muss nur beim ersten Start der Testanwendung und nach aktivem Ausloggen des Nutzers vollzogen werden.

Startansicht

Die drei verschiedenen, für den Nutzer wichtigsten Daten sind die tägliche Aktivitäten, die Trainingsdaten und die physischen Informationen des Nutzers. Daher werden diese nach einem erfolgreichen Login als Reiter am unteren Bildschirmrand angezeigt.
In der Toolbar am oberen Bildschirmrand wird ein *Polar*-Icon und einen Menü-Button mit verstecktem seitlichen Menü zur Navigation angezeigt.
Für das Abfragen neuer Daten befindet sich ein Refresh-Button in der Toolbar, der es dem Nutzer ermöglicht, dies manuell zu tun. Beim Start der Testanwendung soll diese Abfrage automatisch erfolgen. Sind Daten vorhanden, wird der Nutzer mittels eines Nachrichten-Fensters gefragt, ob er die Daten herunterladen will. Ansonsten wird dieses Fenster bei einem erneuten Start oder durch Drücken des Refresh-Buttons erneut angezeigt. Es werden immer alle Daten geladen und nicht zwischen Aktivitäten, Training oder physischen Informationen unterschieden, wodurch die Testanwendung nach dem Start in allen Bereichen auf dem aktuellen Stand ist.

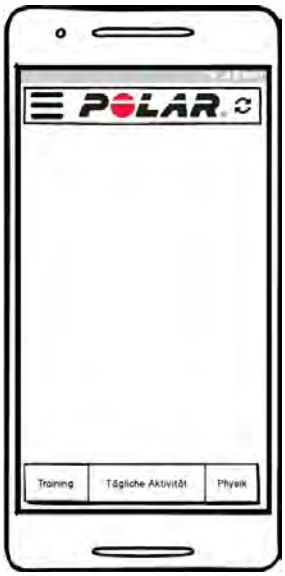


Abbildung 5.3: Startansicht ohne Daten

Durch Klicken der drei Reiter werden die unterschiedlichen Übersichten angezeigt. Die Toolbar bleibt auf jeder der drei Seiten vorhanden.

Tägliche Aktivitäten - Übersicht

Die Ansicht der täglichen Aktivitäten ist das Erste das der Nutzer nach einem erfolgreichem Login zu Gesicht bekommt, da es diejenige Ressource ist, welche am häufigsten vom Server abgerufen wird.

Damit der Nutzer einen geordneten Überblick über seine täglichen Aktivitäten erhält, wird ihm zuerst eine zusammenfassende, nach Datum sortierte Übersicht der jeweiligen Tage angezeigt (vgl. Abbildung 5.4a). Vorlage für die in der Übersicht dargestellten Werte sind die Daten, die von der Polar-API bereitgestellt werden. Zu den Daten gehören **Datum**, **Dauer**, **Schritte**, **Kalorienbedarf des Tages**, sowie **Verbrannte Kalorien des Tages**. Aus dem Bedarf und den verbrannten Kalorien berechnet sich die in Fortschrittsbalken visualisierte Prozentzahl und somit die getätigte Aktivität.
Klickt der Nutzer auf eine bestimmte Tagesübersicht, wird ihm eine detaillierte Seite angezeigt, die alle von Polar zu Verfügung gestellten Daten darstellt.

Tägliche Aktivitäten - Details

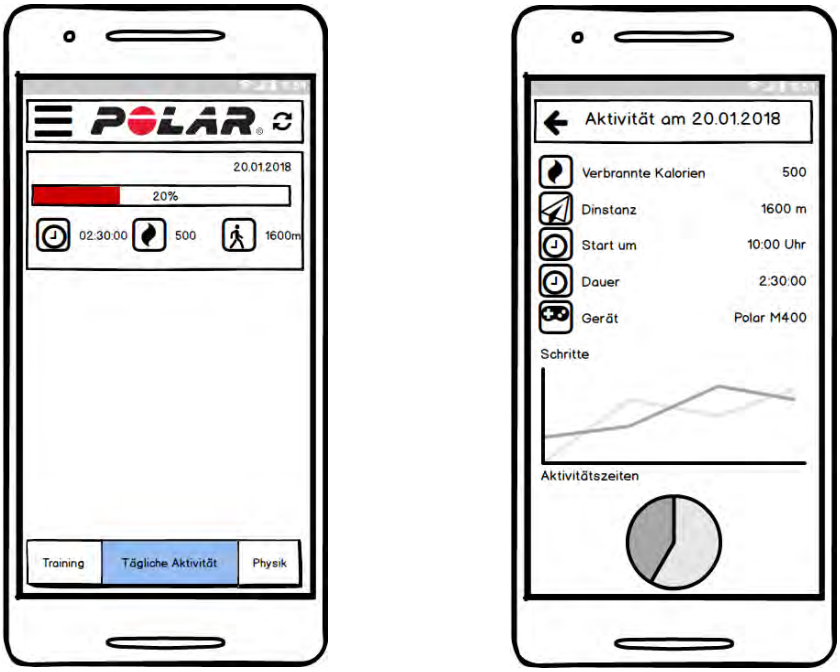
Wie auf Abbildung 5.4b dargestellt werden im oberen Bildschirmbereich die selben Daten wie in der Übersicht dargestellt, was unnötiges Navigieren verhindern soll. Zu den in der Übersicht vorgestellten Daten kommen nun die **Step-Samples** und **Zone-Samples** hinzu. Da beide Sample-Daten ohne Visualisierung durch Diagramme schwer verständlich sind, sollen die Step-Samples als Linien- und die Zone-Samples als Kuchendiagramm dargestellt werden.

Step-Samples

Step-Samples beschreiben die getätigten Schritte in bestimmten Intervallen. Ein Intervall setzt sich aus dem Startzeitpunkt und den getätigten Schritten zusammen. Durch die Verwendung eines Liniendiagrammes können diese Intervalle am besten visualisiert werden.

Zone-Samples

Wie in Kapitel 2.2.1 beschrieben existieren sechs verschiedene Zonen. Der Tag



(a) Überblick (b) Detailansicht

Abbildung 5.4: Ansicht der täglichen Aktivitäten

wird dabei in Zeitintervalle unterteilt, die jeweils die Dauer in den einzelnen Zonen beinhalten. Zum Darstellen dieser Zonen wird die Zeit in den jeweiligen Zone aller Intervalle zusammengerechnet und in einem Kuchendiagramm im Verhältnis zueinander gestellt.

Training - Überblick

Die Übersicht der Trainingseinheiten können durch Klicken auf den linken Reiter erreicht werden. Dort findet man, ähnlich wie bei den täglichen Aktivitäten, eine sortierte Sammlung von Trainingsübersichten. Die dargestellten Daten werden von der Polar-API geliefert. **Geräteinformationen**, **Startzeit**, **Dauer**, **verbrannte Kalorien**, **Distanz**, **durchschnittliche und maximale Herzfrequenzrate** und die **Sportart** gehören zu den Trainingsdaten der Übersicht.

Durch Klicken auf die Übersichtskarte werden alle Daten der Trainingseinheit, die von Polar geliefert werden im Detail dargestellt.

Training - Details

In der Trainingsdetailansichten (Abb. 5.5b und Abb. 5.5c) werden alle Daten einer Trainingseinheit dargestellt. Zu den Daten aus Abschnitt 5.2 kommen die aus Kapitel 3.4 vorgestellten Datentypen **GPX**, **TCX** und **FIT** dazu. Zusätzlich werden die **Herzfrequenzzonen** und 12 verschiedene Samples geliefert. Zu den Samples gehören unter anderem **Geschwindigkeit**, **Kadenz** (Trittfrequenz), **Höhe**, **Luftdruck**, **Temperatur** und **Leistung** des Trainierenden. Je nach Uhr und Einstellungen der Uhr können nur einige der Samples aufgezeichnet werden.

In Abbildung 5.5a werden alle Übersichtsdaten, sowie die jeweiligen Herzfrequenzzonen als Balkendiagramm angezeigt. Die verschiedenen Sample-Daten werden in einem Liniendiagramm angezeigt (vgl. Abbildung 5.5b) und sollen für eine bessere Übersicht an- und abgewählt werden können. Die Laufstrecke wird als Karte unten in der Übersicht angezeigt.

(a) Trainingsübersicht

(b) oben: Überblick
unten: Herzfrequenzzonen

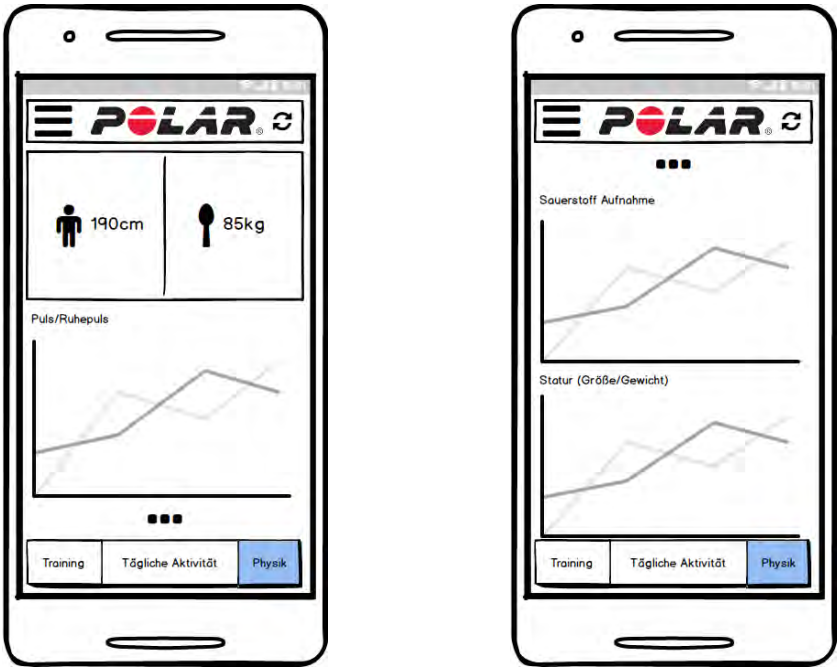
(c) oben: Verlauf
unten: Strecke

Abbildung 5.5: Training

Physische Informationen

Der rechte Reiter zeigt die physischen Daten des Nutzers. Sobald der Nutzer auf der Polar-Webseite oder in der Polar-App neue physische Daten einträgt werden diese an den Clienten gesendet. Durch Abspeichern dieser Daten wird ein Verlauf der physischen Informationen erzeugt, der den physischen Fortschritt visualisieren und somit den Nutzer motivieren soll.

Neben **Gewicht** und **Größe** werden Daten wie **Maximal-** und **Ruhepuls**, sowie die **maximale Sauerstoffaufnahme** und die **aerobe** und **anaerobe Schwelle** dargestellt.



(a) oben: Größe & Gewicht
unten: Puls & Ruhepuls

(b) oben: Sauerstoffaufnahme
unten: Gewichts- & Größenverlauf

Abbildung 5.6: Physische Informationen

Abbildung 5.6a zeigt den Ist-Zustand der Größe und des Gewichts mit dem Pulsverlauf als Liniendiagramm darunter. Die Sauerstoffaufnahme, die aerobe und anaerobe Schwelle, sowie der Gewichts- und Größenverlauf werden in Abbildung 5.6b ebenfalls als Liniendiagramm dargestellt.

User Informationen

Neben den Aktivitäts- und Trainingsinformationen soll der Nutzer seine persönlichen Daten einsehen können. Da diese jedoch statisch und für den Nutzer weniger relevant im Hinsicht auf Trainingsfortschritt sind, werden diese Informationen nicht auf der Startseite angezeigt, sondern sind über das Seitenmenü erreichbar.

Die Abbildung 5.7 zeigt die Daten, die Polar dem Clienten über den Nutzer liefert. Dazu gehören die **Polar-ID**, das **Registrierungsdatum**, der **Nutzername**, das **Geschlecht**, das **Geburtsdatum**, das **Gewicht** und die **Größe**. Polar betont dabei, dass die Größe und das Gewicht nicht über die Nutzerinformationen, sondern über die physischen Informationen (vgl. Kapitel 5.2) abgefragt werden sollen.

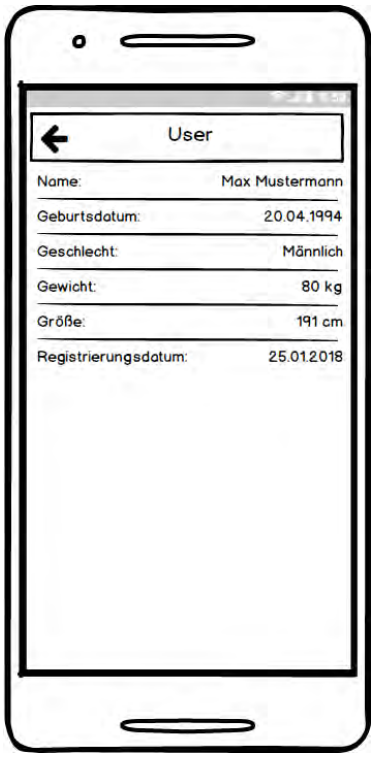


Abbildung 5.7: Übersicht der Nutzerdaten

5.3 Datenbankentwurf

Dieses Kapitel beschäftigt sich damit, wie die von Polar gelieferten und in Kapitel 5.2 vorgestellten Daten in einer Datenbank gespeichert werden können, damit diese dynamisch erweiterbar und konsistent mehreren Nutzern gegenüber ist. Die Aufzählung des Kapitels 5.3.1 zeigt die Daten, die zu speichern sind. Im Folgekapitel 5.3.2 wird dazu eine *Entity-Relationship-Modell* vorgestellt, das diesen Datenbankentwurf visualisieren soll.

5.3.1 Datenanalyse

Da die Kommunikation mit Polar auf Transaktionen basiert und diese von Polar verwaltet werden, nutzen wir die gelieferten Transaktions-IDs und jeweiligen Training- oder Akitvität-IDs um ein *Key-Value-Datenbanksystem* zu erstellen. Gespeichert werden, die von der Polar-API gelieferten JSON-Objekte oder zu GeoJSON (vgl. Kapitel 3.4) konvertierte Datentypen.

Token

Der Token ist der Einstiegspunkt der Datenbankhierarchie. Er wird unter dem *Key* „Token“ als *Value* in die Datenbank eingetragen. Ist ein Token eingetragen, ist ein Nutzer angemeldet und kann direkt zur Übersicht weitergeleitet werden. Das Codebeispiel A.12 zeigt einen solchen Token mit der *User-ID*.

User-ID

Die User-ID enthält alle Transaktions-IDs und Nutzerinformationen wie im Quelltext A.13 verdeutlicht.

Transactions

Diese Transaktions-IDs verweisen wie im Quellcode A.14 auf Listen mit den Trainings-, Aktivitäts- oder physischen Informations-IDs der jeweiligen Transaktion.

User

Die Nutzerinformationen werden, sowie sie vom Server kommen, unter der jeweiligen User-ID gespeichert.

Activity & Exercise & Physical information

Die jeweiligen IDs verweisen auf das JSON-Objekt, das die gesamten Daten der Einheit speichert. Dabei sollen alle Daten bereits so konvertiert sein, dass sie direkt verwendet werden können. Übersichten der JSON-Objekte werden in den Quellcodebeispielen A.15 bis A.17 dargestellt.

Um die Struktur der Datenbank zu verdeutlichen, wird im Kapitel 5.3.2 ein Entity-Relationship-Modell vorgestellt.

5.3.2 Entity-Relationship-Modell des Datenbankentwurfes

Zur Verdeutlichung des Datenbanksentwurf wird in diesem Kapitel ein Entity-Relationship-Modell (ER-Modell) vorgestellt, der das Verständnis der Datenstruktur visualisieren soll. Die Abbildung 5.8 zeigt das ER-Modell.

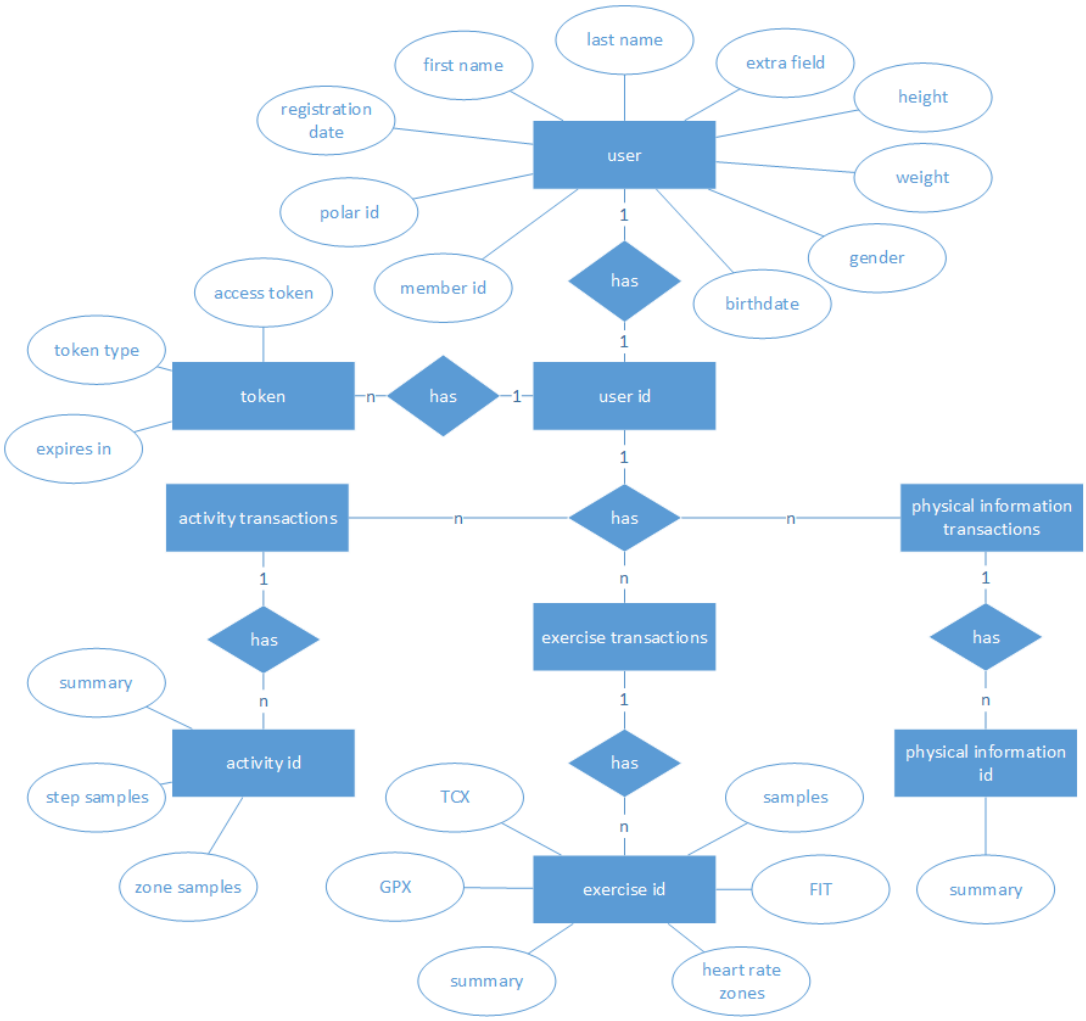


Abbildung 5.8: Entity-Relationship-Modell des Datenbankentwurfes

Mit dem *token* als Einstiegspunkt können in diesem Datenbankentwurf beliebig viele Nutzer die Testanwendung verwenden. Neben den Transaktions-IDs werden die Nutzerinformationen unter der *user-id* gespeichert. Die Datentypen und ihre Attribute werden unter der jeweiligen ID gespeichert.

6

Realisierung

Nachdem alle Anforderungen formuliert und verwendete Daten vorgestellt sind, wird in diesem Kapitel gezeigt, wie die Testanwendung realisiert wurde. Zu Beginn wird auf die verwendeten Frameworks eingegangen, um anschließend im Kapitel 6.2 ihre Verwendung in der Implementierung zu erläutern. Der im Kapitel 5.1.4 vorgestellte Ablauf der Transaktionen wird spezifiziert und die Implementierung ausführlich beschrieben. Dabei wird auf die Datenbank und die Simplifizierung des Ablaufs eingegangen.

6.1 Verwendete Frameworks

Leaflet

Leaflet ist eine Open-Source JavaScript Bibliothek für mobile, interaktive Karten (Maps). Im Jahr 2010 von Vladimir Agafonkin entwickelt, unterstützt Leaflet das Einbinden von *Map Services*, GeoJSON und Bildüberlagerungen. Mit zusätzlichen Plugins können weitere Geodaten-Formate (z.B GPX, CSV) eingebunden werden.

```
1 let map = L.map('map').fitWorld();
2
3 L.tileLayer('https://api.mapbox.com/v4/{id}/{z}/{x}/{y}.png?' +
4     'access_token={accessToken}', {
5         attribution: 'infos about the map and its contributors',
6         maxZoom: 18,
7         id: 'mapbox.run-bike-hike',
```

```
8           accessToken: apiToken
9       }).addTo(map);
10
11  let jsonLayer = L.geoJSON(this.geojson).addTo(map);
12  map.fitBounds(jsonLayer.getBounds());
```

Listing 6.1: Erstellen einer Karte mit Leaflet

Das Codebeispiel 6.1 zeigt die Implementierung einer Karte mit Hilfe eines GeoJSON. In Zeile 1 wird mit `L.map('map')` der `div` mit `id="map"` im DOM an Leaflet gebunden und bildet so den Container der Karte. Der GeoJSON wird in Zeile 11 an die Karte angefügt und anschließend werden die Grenzen der Karte an die des GeoJSON angepasst.

In Zeile 3 wird das `Tilelayer` an die Karte angefügt. Dafür wird *Mapbox* verwendet.

Mapbox

Mapbox ist eine Standortdatenplattform für Mobile- und Webanwendung. Sie bietet standortspezifische Funktionen wie Karten, Suche und Navigation. Im Codebeispiel 6.1 wird Mapbox als Tilelayer hinzugefügt und liefert so das Aussehen und die Größe der Karte. Unter der `id` können verschiedene Kartentypen angegeben werden, die in der *Mapbox Maps API* beschrieben werden. Da es sich bei der Testanwendung um die Evaluation von Fitnessdaten handelt, wurde hier das `run-bike-hike`-Aussehen verwendet.
Um Mapbox und die zugehörigen APIs oder SDKs verwenden zu können, muss ein `access token` verwendet werden, der auf der Webseite nach einer Anmeldung zugewiesen wird.

Chart.js

In der Testanwendung werden verschiedene Fitnessdatenverläufe evaluiert. Für eine anschauliche Darstellung dieser Daten wird *Chart.js* verwendet.

Chart.js kann nach der Installation acht verschiedene Diagramme, darunter Linien-, Balken- und Kuchendiagramme, erstellen. Diese Diagramme können anschließend weiter modifiziert werden, indem Farbe und Schriftart und auch die Achsenbeschriftung und Legende verändert werden können.
Ein kurzes Beispiel eines Balkendiagramms wird im Codebeispiel 6.2 dargestellt. Der `type` (hier `'bar'`) gibt an, um welches Diagramm es sich handelt. In `data` werden JSON-Arrays mit den Datensätzen und den Achsenbeschriftungen übergeben. Hier kann auch die Farbe der jeweiligen Balken verändert werden. Unter `options` können Attribute, wie Breite der Balken, Rotation des Kuchendiagramms oder Dicke des Liniendiagramms verändert werden.

```
let hrChart = new Chart(this.heartRateCanvas.nativeElement, {
        type: 'bar',
        data: {
          labels: zoneHeartRateLimit,
          datasets: [{
            label: 'Herzfrequenzzone',
            data: zoneDuration
          }],
        },
        options: options
      });
```

Listing 6.2: Ein Balkendiagram mit Chart.js

Weiterhin sind die Diagramme von Chart.js interaktiv. Eingetragene Daten können vom Nutzer durch Klicken eingesehen und ganze Datensätze ein- bzw. ausgeblendet werden. Dadurch können einzelne Datensätze genauer inspiziert und vom Nutzer evaluiert werden.

Plugins

Nicht alle Daten, die von der Polar-API geliefert werden, können direkt übernommen werden. Daher werden sie vor dem lokalen Speichern von *Plugins* geparsed, um sie für die Auswertung zur Verfügung zu stellen. Die in der Testanwendung verwendete Plugins werden im Folgenden vorgestellt.

ISO8601-durations

Alle Zeitangaben, auch die Angaben für Dauer (duration), die von der Polar-API geliefert werden, sind im *ISO 8601*-Format. Diese zu parsen ist notwendig, um im späterem Verlauf verwenden zu können. Dafür wird das *ISO8601-durations*-Plugin verwendet. Das Plugin übersetzt Dauerangaben (vgl. PnYnMnDTnHnMnS) in einen JSON, der die Zeitangaben als Variable hält. Zusätzlich können alle Zeitangaben in Sekunden ausgegeben werden.

mapbox/tcx

Mapbox stellt einen Parser zur Verfügung, der TCX-Daten in GeoJSON umwandelt. Alle Daten, die GeoJSON nicht als Geometrie-Objekt darstellen kann, werden in die `properties`-Variable angefügt (vgl. Kapitel 3.4).

mapbox/togeojson

Neben einem TCX-Parser stellt Mapbox einen GPX- bzw. KML-Parser. Dadurch können die GPX-Daten in GeoJSON übersetzt und in Leaflet verwendet werden.

6.2 Implementierung

Die Implementierung der Testanwendung ist wesentlicher Bestandteil der Evaluation der Polar-API. Dadurch kann überprüft werden, ob Ionic mit seinen Modulen mit der Polar-API kommunizieren kann. Dieses Kapitel verdeutlicht den Aufbau der Testanwendung und ihre Implementierung. Dabei wird auf den Authentifizierungsvorgang und die Anfragen an die Polar-API eingegangen. Im Anschluss wird die Datenbank, sowie die Verwendungsmöglichkeiten eines HttpInterceptors vorgestellt.

6.2.1 Authentifizierungsvorgang

Der Authentifizierungsvorgang wird mit Hilfe des *In-App-Browsers* von Ionic realisiert. Mit dem In-App-Browser kann der Nutzer Webseiten innerhalb der Anwendung einsehen ohne die Anwendung selbst zu verlassen.

Polar stellt eine URL zum Start des Authentifizierungsvorgangs. An diese URL müssen zwei Parameter angehängt werden. Zum einen muss der Umfang (`scope`) der Zugriffsberechtigung und zum anderen die Client-ID angegeben werden. Da alle Daten des Nutzers eingesehen werden sollen, werden die Parameter `scope=accesslink.read_all` und zusätzlich die `client_id={CLIENT_ID}` angefügt. Anschließend loggt sich der Nutzer nach dem Aufrufen der URL mit der E-Mail Adresse und dem Passwort ein und wird aufgefordert die Zugriffsberechtigungen an den Clienten zu übertragen. Nach Akzeptieren der Berechtigungen und dem Anerkennen der allgemeinen Geschäftsbedingungen wird der In-App-Browser an die URL, die während des Registrierungsprozesses (vgl. Kapitel 5.1.2) angegeben wurde, weitergeleitet.

Zum Abfangen dieser URL stellt der In-App-Browser *Events* zur Verfügung. So kann während des `loadstart`-Events überprüft werden, ob die aufgerufene URL mit der Callback-URL übereinstimmt. Ist dies der Fall, wird unter dem URL-Parameter `code` der Autorisierungscode bereitgestellt, mit dem im Anschluss der Access Token angefragt werden kann.

Um den Autorisierungscode als einen solchen zu identifizieren, wird im Request für den Access Token dieser Autorisierungscode mit der Kennung `grant_type = "authorization_code"` und dem Code an sich (`code={CODE}`) im Body angefügt. Ist der Request erfolgreich, wird der Access Token wie im Codebeispiel A.12 zurückgegeben. Ansonsten wird ein Fehler mit verschiedenen Kennungen, die diesen näher erläutern, zurückgegeben.

Mit dem Erhalten des Access Tokens ist der Authentifizierungsvorgang angeschlossen. Nun muss der Nutzer beim Client registriert werden bevor er auf die Daten zugreifen kann. Im Folgenden wird neben der Registrierung, auch das Löschen eines Nutzers, sowie Erhalten der Nutzerinformationen vorgestellt.

6.2.2 Nutzerbezogene Anfragen

Dieses Kapitel beschreibt alle Vorgänge, die den Nutzer betreffen. Dazu gehören die Registrierung und das Löschen eines Profils, sowie der Erhalt der Nutzerdaten.

Nutzerregistrierung

Den Nutzer zu registrieren ist essentiell, um auf die Fitness-Daten zugreifen zu können. Daher wird der Nutzer direkt nach dem Authentifizierungsvorgang registriert. Die Registrierung des Nutzers ist ein POST-Request mit einer clienteigenen Kennung, der Member-ID (`member-id`) im Body. Als Response wird der in Kapitel 5.2 vorgestellte User zurückgegeben. Nach dieser Registrierung können Fitness bezogene Anfragen an die Polar-API gestellt werden.

Löschen des Nutzerprofils

Will der Nutzer dem Clienten die Berechtigung entziehen oder der Client nicht mehr auf die Daten des Nutzers zugreifen, muss die Registrierung des Nutzers beim Clienten rückgängig gemacht werden. Dafür wir ein DELETE-Request an die Polar-API geschickt.

Nutzerinformationen

Um die grundlegenden Informationen eines Nutzers zu erhalten, wird ein GET-Request mit der Polar-`user-id` an die API gesendet. Es werden dabei die gleichen Daten, wie bei der Registrierung erhalten.

In Kapitel 5.1.4 wurde bereits auf die Abfolge der Anfragen an die Polar-API eingegangen. Im folgenden Kapitels wird die Umsetzung dieser Anfragen vorgestellt.

6.2.3 Realisierung der Kommunikation

Im Kapitel 5.1.4 wurden bereits die verschiedenen Transaktionen vorgestellt. In diesem Kapitel wird auf die Implementierung der Transaktions-Kommunikation eingegangen.

Der Einstiegspunkt des angemeldeten Nutzers ist die *Tabs*-Übersicht mit den täglichen Aktivitäten, dem Training und den physischen Informationen als Reiter. Daher startet zu diesem Zeitpunkt die nicht Login bezogene Kommunikation mit der Polar-API. Wie im Kapitel 5.1.4 beschrieben, ist der erste Schritt zum Anfragen der Daten, die Nachfrage, ob neue Daten vom Nutzer synchronisiert wurden und somit zu Verfügung stehen (`Pull notifications`). Ist dies der Fall wird eine Kaskade von Anfragen ausgelöst. Jede der verfügbaren Daten, sei es eine tägliche Aktivität, ein Training oder eine physische Information, werden unabhängig voneinander angefragt und sind somit voneinander unabhängig. Nach dem Erstellen einer Transaktion (`Create`) werden alle vorhandenen Einheiten gelistet (`List`) und nacheinander angefragt (`Get`). Nach dem Erhalt der Daten können sie lokal gespeichert werden (`Save`). Erst nach einem erfolgreichem Speichervorgang aller gelisteten Einheiten werden die Transaktionen beendet (`Commit`) und somit die Kaskade abgeschlossen. So können die Daten bei einem Fehler erneut angefragt werden.

Die Abbildung 6.1 zeigt diese Kaskade, wobei die `Get`-Methode sich bei den jeweiligen Einheiten unterscheidet.

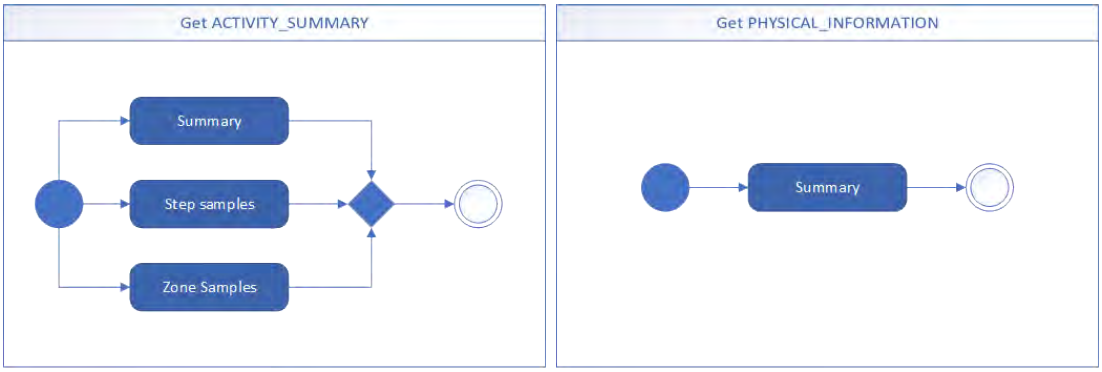
Abbildung 6.1: Überblick der Realisierung der Kommunikation mit der Polar-API

Die Anfragen an die Polar-API benötigen immer dieselben Header-Informationen `Accept: application/json`. Daher können die Methoden für `Create`, `List`, `Get` und `Commit` für alle Anfragen des Services verwendet werden. Ausgenommen sind dabei die Anfragen der TCX- und GPX-Daten. In Kapitel 6.2.3 wird genauer darauf eingegangen.

Da bei den `Get`-Anfragen alle Ergebnisse benötigt werden, wird dabei ein *Forkjoin* verwendet. Dadurch wird unnötiges Warten auf Callbacks vermieden und der Code übersichtlich gehalten.

Tägliche Aktivitäten und physische Informationen

Um eine Zusammenfassung eines Tages zu erhalten, werden drei verschiedene Anfragen benötigt: eine Tagesübersicht (`summary`), die Step-Samples und die Zone-Samples (vgl. Abbildung 6.2a). Bei den physischen Informationen handelt es sich um eine Anfrage (`summary`) (vgl. Abbildung 6.2b). Nach Abschluss der Anfragen werden die Daten gespeichert.



(a) Tägliche Aktivität (b) Physische Informationen

Abbildung 6.2: Ablauf der Anfragen

Training

Beim Erhalt der Trainingsinformationen verhält es sich etwas anders. Zum einen werden neben der Trainingszusammenfassung, auch die Herzfrequenzdaten angefragt und zum anderen benötigen GPX und TCX spezifische Accept-Header. Für GPX ist das `application/gpx+xml` und für TCX `application/vnd.garmin.tcx+xml`. Dadurch benötigen diese zwei Anfragen eigene Methoden im Service.
Zusätzlich werden die vorhandenen Samples in einer Liste zurückgegeben, wodurch erneut Anfragen an die Polar-API anfallen. Beispiele dieser Samples sind in Kapitel 5.2 angegeben.

Sind die Daten gespeichert und die Transaktion mit einem `Commit` abgeschlossen, werden mit Hilfe von `Events` die jeweiligen Reiter benachrichtigt, um die neuen Daten aus der Datenbank zu lesen.

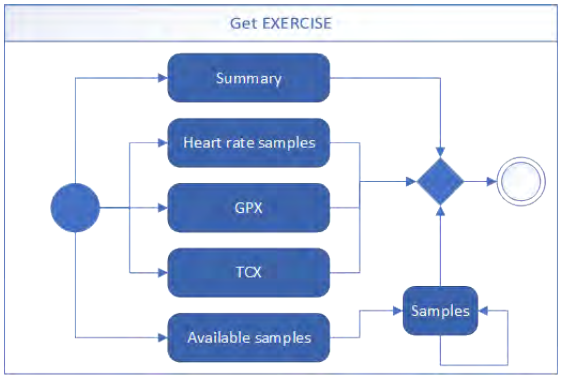


Abbildung 6.3: Ablauf der Anfragen der Trainingseinheiten

6.2.4 Datenbank

Der in Kapitel 5.3 vorgestellte Entwurf des Datenbankkonzeptes wurde mit Hilfe des LocalStorage umgesetzt. Da der Ionic.Storage Callbacks verwendet, um Daten aus dem Speicher zu lesen, wird der nutzerspezifische JSON (vgl. Abbildung 5.8 unter `user-id`) asynchron durch simultanes Lesen überschrieben.
Wird zum Beispiel eine Trainingseinheit und eine tägliche Aktivität gespeichert, geschieht dies simultan. Beide Speichervorgänge lesen den selben JSON aus dem Speicher um die jeweilige Transaktion-ID darin zu speichern. Dadurch überschreibt der zuletzt kommende Speichervorgang den zuvor hinzugefügten, wodurch diese Informationen verloren gehen. LocalStorage hingegen liefert die Einträge in der Datenbank nicht als Callback, sondern initial, dass dieses Problem verhindern wird.

Um den Speichervorgang dynamisch und den Code übersichtlich zu halten, wird ein *Datentypen-JSON* erstellt (vgl. Codebeispiel A.6). Dieser hält Informationen über die verschiedenen Datentypen (Training, tägliche Aktivität, physische Informationen), sowie die Attribute die sie beinhalten. Übergibt man den zugehörigen Datentyp aus dem JSON mitsamt der Daten, die sie beschreiben und die zu speichern sind, kann für alle Datentypen eine Speichermethode verwendet werden.

Die Daten der Polar-API werden nach dem Erhalt sofort gespeichert. Erst nach einem erfolgreichen Speichervorgang werden die Daten `committed` und können somit nicht

erneut angefragt werden. Vor dem Speichern der Daten werden diese so geparsed, wie sie im Laufe der Anwendung verwendet werden. Dazu werden die in Kapitel 6.1 vorgestellten Plugins verwendet. Das Speichern in die Datenbank läuft bei jedem Datensatz gleich ab. Zuerst wird die Validität des Tokens überprüft, mit dessen Hilfe der `User-JSON` aus dem Speicher gelesen und die Transaktion-ID an das jeweilige Attribut angefügt werden kann. Dabei wird stets überprüft, ob die Transaktions-ID bereits vorhanden ist. Anschließend werden die jeweiligen IDs unter der Transaktions-ID gespeichert. Der letzte Schritt ist das Speichern der Daten unter der jeweiligen ID.

6.2.5 HttpInterceptor

Der HttpInterceptor von Angular ist ein nützliches Interface um Http-Anfragen vor dem Absenden nochmals abzufangen und zu modifizieren. Das kann man sich zu Nutze machen, um die Authentifizierungsheader, die die Polar-API vorschreibt, an die Anfrage anzuhängen. In Kapitel 5.1.3 wurden bereits die verschiedenen Anforderungen an die Authentifizierung beschrieben. Diese Anforderungen können nun mit einem Interceptor umgesetzt werden. Dafür wird, je nach URL, ein anderer Authentifizierungsheader angefügt.

```
1  intercept(request: HttpRequest<any>, next: HttpHandler)
2                      : Observable<HttpEvent<any>> {
3      let url = new URL(request.url);
4
5      if(url.pathname.indexOf('/v3/users')){
6          // Add Bearer-Token to Authorization-Header
7      }  else { // url.pathname.indexOf('/v2/oauth2')
8          // Basic-Header with Base64 encoded Client und Secret
9      }
10     return next.handle(request)
11         .timeoutWith(30000, Observable.throw('Timeout'));
12 }
```

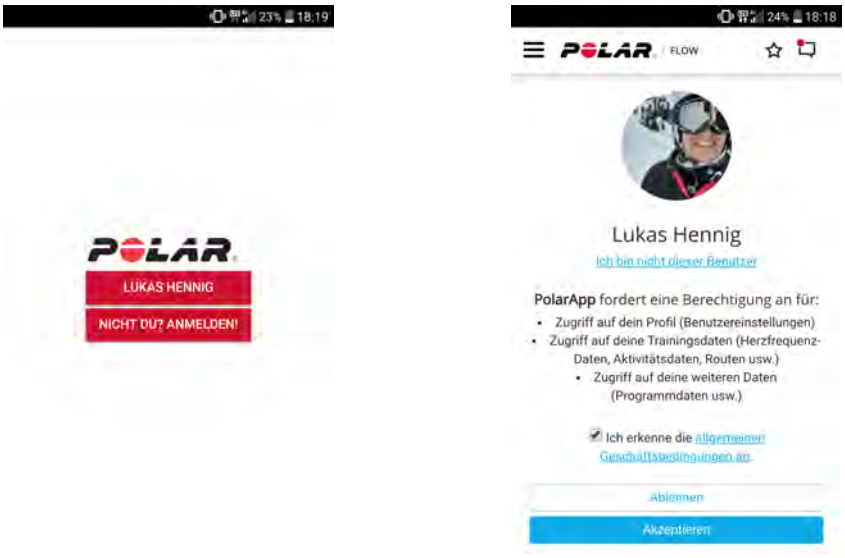
Zusätzlich wird der Request mit einem 30-sekündigen Timeout versehen, um zu lange andauernde Anfragen abzubrechen und den Nutzer anschließend zu benachrichtigen.

6.2.6 Realisierung des User-Interfaces

Mit der Implementierung der grundlegenden Logik und mit den in Kapitel 5.2 vorgestellten Mock-ups als Vorbild, kann nun das User-Interface der Anwendung realisiert werden.

Login

Der erste Start der Anwendung verlangt nach einer Anmeldung des Nutzers, daher wird ihm nur ein Button zum Beginn des Anmeldevorgangs angezeigt. Ansonsten kann er nach dem Ausloggen entscheiden, ob er den Nutzer wechseln möchte oder ob er weiterhin den aktuellen Account verwenden will. Abbildung 6.4a zeigt diesen Dialog.
Die Abbildung 6.4b zeigt die Berechtigungsfreigabe des Nutzers, die innerhalb des In-App-Browsers von statten geht und von Polar vorgegeben ist. Nach einer erfolgreichen Anmeldung werden dem Nutzer die Fitnessdaten angezeigt.



(a) Logindialog trotz angemeldeten Nutzers (b) Berechtigungsfreigabe

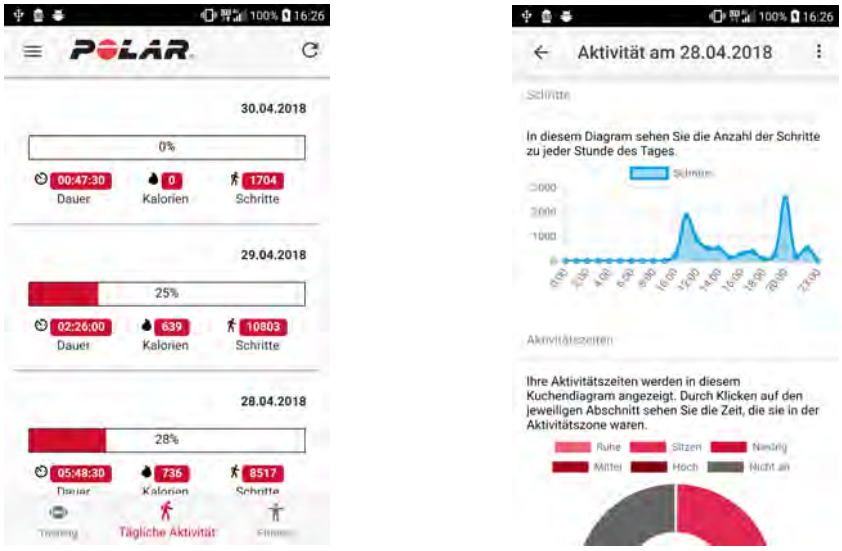
Abbildung 6.4: Loginvorgang

Anzeige der Fitnessdaten

Die Fitnessdaten werden auf einer Seite mit drei unteren Reitern zusammengefasst (vgl. Abbildung 6.5a). Dabei kann auf der grundlegenden Seite stets das seitliche Menü (oben links) mit Navigationsoptionen zur *Nutzerübersicht* und zum *Logout* aufgerufen, sowie die Verfügbarkeit neuer Daten angefragt werden (oben rechts). Mittig ist ein Polar-Logo um dem Nutzer zu verdeutlichen, dass es sich um eine Polar Anwendung handelt.

Tägliche Aktivitäten

Die täglichen Aktivitäten werden beim Start der Testanwendung angezeigt. Sie können stets unter dem Reiter „Tägliche Aktivität“ aufgerufen werden. Durch Klicken auf eine Ionic-Card mit den Übersichtsinformationen kann der Tagesverlauf im Detail betrachtet werden. Die Abbildung 6.5 zeigt die zum einen die Übersicht (6.5a) und zum anderen den Tagesverlauf im Detail (6.5b).



(a) Übersicht (b) Tagesverlauf

Abbildung 6.5: Tägliche Aktivitäten

Die jeweiligen tägliche Aktivität kann durch langes Drücken in der Übersicht oder durch das Menü in der Ansicht der täglichen Aktivität gelöscht werden.

Trainingseinheiten

Der Reiter „Training“ zeigt wie in Abbildung 6.6a eine Übersicht der Trainingseinheiten mit den wichtigsten Informationen. Durch Klicken auf die jeweilige Trainingseinheit wird die in den Abbildungen 6.6b und 6.6c dargestellte Detailsansicht dieser Trainingseinheit geöffnet. Zuerst werden die Daten aus Abbildung 6.6a wiederholt und anschließend die Trainingsinformationen mit den Herzfrequenzen, den Samples und dem Streckenverlauf dargestellt.

(a) Überblick der Trainingseinheiten

(b) oben: Überblick unten: Herzfrequenzzonen

(c) oben: Trainingssamples unten: Streckenverlauf

Abbildung 6.6: Training

Die Trainingseinheiten können wie eine tägliche Aktivität durch langes Klicken in der Übersicht oder über das Menü in der Ansicht gelöscht werden. In der Karte des Streckenverlaufs kann gezoomt und die Ansicht verschoben werden.

Physische Informationen

Die physischen Informationen werden unter dem Reiter „Fitness “ dargestellt. Wie in Abbildung 6.7 verdeutlicht, wird oben die Größe und das momentane Gewicht des Nutzers angezeigt. Darunter stellen Liniendiagramme den Pulsverlauf, den Sauerstoffverlauf und den Gewichts- bzw. Größenverlauf dar.

(a) oben: Gewicht & Größe
unten: Puls

(b) oben: Sauerstoffaufnahme
unten: Statur

Abbildung 6.7: Physische Informationen

Physische Informationen können nicht gelöscht werden. Die Silhouett in Abbildung 6.7a ist abhängig vom Geschlecht des Nutzers.

Anzeige der Nutzerdaten

Die Anzeige der Nutzerdaten ist trivial, da nur einfache Daten dargestellt werden müssen. Die Abbildung 6.8 zeigt die Darstellung der Nutzerdaten unter Verwendung von Ionic Elementen.

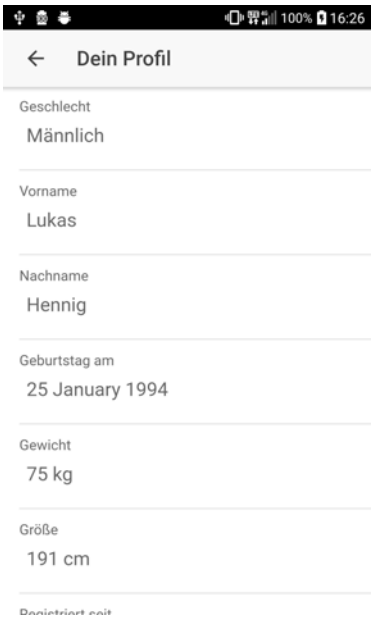


Abbildung 6.8: Anzeigen der Nutzerdaten

7

Anforderungsabgleich

Inwiefern die Testanwendung, die im Kapitel 4 vorgenommenen Anforderungen umsetzt, wird in diesem Kapitel erläutert. Dabei wird zuerst auf die funktionalen und anschließend auf die nicht-funktionalen Anforderungen eingegangen.

7.1 Funktionale Anforderungen

FA01 Anmeldung und Abmeldung des Polar-Nutzers
Mit Hilfe des In-App-Browsers kann der Nutzer den Authentifizierungsvorgang erfolgreich abschließen. Damit ist er eingeloggt und kann die Testanwendung nutzen. Der Nutzer kann sich ausloggen und die Nutzungsberechtigungen der Daten dem Clienten untersagen.

FA02 Lokale Datensicherung
Die Aufteilung der Daten in die vorgesehen Reiter *Tägliche Aktivitäten*, *Training* und *Physische Informationen* sorgt für einen strukturierten Überblick über die Daten, die lokal und flexibel gespeichert werden.

FA03 Übersicht über die täglichen Aktivitäten
Die wichtigsten Daten werden in einem *Ionic-Card-Component* zusammengefasst. Die Sortierung nach Datum verschafft dem Nutzer einen Überblick über die neuesten Aktivitäten.

FA04 Anzeigen der täglichen Aktivitäten
Durch die Verwendung von Chart.js erhält der Nutzer in Form von Diagrammen eine

genaue Ansicht über des jeweiligen Tagesverlaufs. Die Übersichtsdaten werden ebenfalls in der Ansicht vorgestellt, um alle Informationen darzustellen.

FA05 Löschen der täglichen Aktivitäten
Tägliche Aktivitäten können durch langes Drücken in der Übersicht oder durch das Menü in der Toolbar in der jeweiligen Anzeige der täglichen Aktivität gelöscht werden.

FA06 Übersicht der Trainingseinheiten
Die für eine Trainingseinheit relevantesten Daten werden in einer Übersicht dargestellt. Damit die neusten Trainingseinheiten ins Auge fallen, werden die Trainingseinheiten nach Datum sortiert.

FA07 Anzeigen der Trainingseinheiten
Durch die Verwendung von *Leaflet* und *Mapbox* kann die Laufstrecke anschaulich dargestellt werden. Die Sample-Daten der Trainingseinheit werden in einem Diagamm veranschaulicht und können durch Ein- bzw. Ausblenden der Samples genauer untersucht werden. Die Trainingsübersichts-Daten werden ebenfalls in der Ansicht aufgeführt, um alle verfügbaren Daten gesammelt darzustellen.

FA08 Löschen der Trainingseinheiten
Trainingseinheiten können in der Übersicht durch langes Drücken oder in der Ansicht im Menü der Toolbar gelöscht werden. Der Nutzer wird anschließend zur Übersicht navigiert und andere Trainingseinheiten werden dabei nicht verändert.

FA09 Anzeigen des Nutzer-Fitnesszustands
Der aktuelle Fitnesszustand, sowie der Verlauf der physischen Informationen werden dem Nutzer unter Verwendung von Liniendiagrammen übersichtlich angezeigt.

FA10 Anzeigen des Nutzer-Profils
Informationen über den Nutzer werden angezeigt und können auf einer separaten Seite eingesehen werden.

7.2 Nichtfunktionale Anforderungen

NFA01 Zuverlässigkeit

Die physischen, sowie nutzerbezogenen Daten der Polar-API, können unmittelbar nach vollenden der Synchronisation vollständig heruntergeladen, analysiert und gespeichert werden.

NFA02 Korrektheit

Dem Nutzer werden die, vom Fitness-Tracker erhobenen Daten unverfälscht angezeigt.

NFA03 Benutzbarkeit

Durch Verwendung der Reiter sind alle wichtigen Informationen auf einer Seite einzusehen. Icons beschreiben die jeweiligen Attribute und ihre Werte eindeutig.

NFA04 Funktionalität

Alle vorgestellten funktionalen Anforderungen wurden erfolgreich umgesetzt.

NFA05 Effizienz

Es wurde stets auf einen übersichtlichen, sowie wiederverwertbaren Code geachtet. Alle Daten werden dabei so gespeichert, wie sie im späteren Gebrauch benötigt werden.

NFA06 Wartbarkeit

Durch den dynamischen Aufbau der Anwendung ist eine API-Änderung in die Testanwendung gut implementierbar ohne einen Datenverlust zu erleiden.

8

Auswertung und Ausblick

8.1 Versuch zur Erhebung von Synchronisationszeiten

Einleitung

Mit diesem Versuch soll der Zeitraum zwischen Synchronisation und Verfügbarkeit der Daten in der Polar-API bestimmt werden. Dies ermöglicht eine Beurteilung, ob Echtzeitmessung der Daten möglich ist, bzw. wie lange der Nutzer de facto warten muss, bis die Daten auf der Testanwendung verfügbar sind.

Versuchsdurchführung

Der Versuch beginnt, sobald die Synchronisation des Fitness-Trackers gestartet wird. Ab diesem Moment wird die Zeitmessung zur Ermittlung der Synchronisationszeiten gestartet. Dabei wird die Dauer vermerkt, sobald Trainingsdaten, tägliche Aktivitäten oder physische Informationen verfügbar sind. Dies wird mit Hilfe der `Pull notifications` überwacht, welche die Verfügbarkeit signalisieren. Ein Versuchsdurchlauf ist zu Ende, wenn der Fitness-Tracker die Synchronisation abgeschlossen hat. Welches Gerät zur Synchronisation verwendet wurde und ob eine Laufstrecke mit GPX- bzw. TCX-Daten vorhanden ist, wird ebenfalls vermerkt. Für alle Versuche wird die Polar M400 verwendet die bereits im Kapitel 2.1.1 vorgestellt wurde.

Ergebnisse

In Tabelle8.1 werden die Ergebnisse des Versuchs aufgeführt. Die Daten, die am längsten zur Synchronisation benötigen, sind die mit Streckeninformationen der Trainingseinheit wie die Versuche 4, 5 und 7 verdeutlichen. Dabei beträgt die durchschnittliche Synchronisationszeit 9 Minuten. Sind kaum Informationen über die Strecke vorhanden wie in Versuchen 1, 2 und 3, ist die Synchronisationszeit durchschnittlich 1 min und 40 s und somit um einiges kürzer. Versuch Nummer 6 dauerte auch ohne Streckeninformationen 15 min, allerdings wurde der Fitness-Tracker zwischen Versuch 5 und Versuch 6 sehr häufig im täglichen Gebrauch.

Nr.	Datum	Start	Verbunden	Training	Aktivität	phys. Info	Ende	Strecke
1	09.02.18	15.51	00:15	-	00:20	-	15:52	
2	10.02.18	15:03	00:17	00:55	00:45	-	15:05	
3	11.02.18	17:21	00:18	00:22	00:15	00:05	17:23	
4	18.02.18	23:26	00:12	03:26	01:45	-	23:32	✓
5	09.03.18	10:28	00:10	07:10	02:00	-	10:38	✓✓
6	13.04.18	21:15	00:10	08:00	07:15	00:15	21:30	
7	01.05.18	12:00	00:10	07:12	05:35	00:15	12:11	✓✓

Tabelle 8.1: Versuchsergebnisse

Diskussion

In den Ergebnissen wird deutlich, dass die Synchronisationszeiten von der Anzahl der Daten abhängt, die synchronisiert werden. Die physische Informationen sind bereits nach kurzer Zeit verfügbar, während die Trainingsdaten vom Vorhandensein von Streckeninformationen und die tägliche Aktivitäten vom Intervall der Synchronisation und der Häufigkeit des Gebrauchs abhängig sind.

Dabei ist zu beachten, dass die Polar M400 keine Herzfrequenzdaten ermitteln kann. Somit sind diese Informationen nicht in der Versuchsdurchführung berücksichtigt.

8.2 Fazit

In dieser Arbeit wurden die Polar Fitness-Tracker und ihre Funktionsweise im Hinblick auf Synchronisation und Entwicklungsschnittstelle betrachtet und in einem Zusammenhang mit medizinischen Anwendungsmöglichkeiten gestellt. Dabei konnten alle Daten der Schnittstelle in Ionic 3 angefragt, verarbeitet und visualisiert werden. Die Echtzeitmessung ist aufgrund der Kommunikation mit der Polar-API nicht möglich. Der Versuch in Kapitel 8.1 zeigt, dass sowohl die Synchronisationszeit als auch die Größe der Datenmenge von der Art der zu synchronisierenden Daten abhängt.

Die Verwendung von Fitness-Trackern zur Erhebung von medizinisch relevanten Daten ist mit einem Polar Fitness-Tracker unter der Verwendung einer Ionic-Anwendung möglich. Die Schnittstelle liefert nach der Synchronisation die Daten mit einer Verzögerung von wenigen Minuten und kann dadurch nahezu in Echtzeit diese Daten verwerten.

8.3 Ausblick

mHealth bietet, unter der Verwendung von Fitness-Trackern, einen ersten Ansatz der Selbstdiagnose, sowie eine vereinfachte Kommunikation zwischen Arzt und Patient. Mit einem Fitness-Tracker von Polar und Ionic 3 als Plattform können Anwendungen realisiert werden, die Herzfrequenzdaten, Kalorienverbrauch und Trainingsverhalten auswerten und dem Arzt übermitteln können. Dadurch kann das Prinzip, der in der Einleitung angesprochen *Track your Tinitus*-Anwendung, bei der die Hörleistung von Tinnituspatienten überwacht wird, auf weitere Krankheitsbilder übertragen werden und somit zur Diagnose bzw. Überwachung beitragen.

Literaturverzeichnis

[1] Electro, P.: Warum Polar wählen? https://support.polar.com/de/training_mit_polar/trainingsartikel/los_gehts/warum_polar_wahlen (2018) Accessed: 17.02.2018.

[2] 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: 2016 IEEE 29th International Symposium on Computer-Based Medical Systems (CBMS). (2016)

[3] Halber, M. In: Digitalisierung im zweiten Gesundheitsmarkt. Springer Fachmedien Wiesbaden, Wiesbaden (2017) 37–48

[4] Polar Electro Oy: Polar M400 Gebrauchsanleitung, Professorintie 5, FI-90440 KEMPELE. 1.8 de edn. (2017)

[5] Polar Electro Oy: Polar M600 Gebrauchsanleitung, Professorintie 5, FI-90440 KEMPELE. 1.0 de edn. (2016)

[6] Diehm, C.: Fitnessarmbänder messen ungenau. MMW - Fortschritte der Medizin **159** (2017) 31–31

[7] Electro, P.: Wer Wir Sind. https://www.polar.com/de/uber_polar/wer_wir_sind (2018) Accessed: 17.02.2018.

[8] Electro, P.: Wie kann ich mein Polar Gerät mit der Polar Flow App koppeln? https://support.polar.com/de/support/M400/how_do_i_pair_my_polar_m400_with_the_polar_flow_app (2018) Accessed: 17.02.2018.

[9] Garmin: Annual report. https://www8.garmin.com/aboutGarmin/invRelations/reports/2016_Annual_Report.pdf (2016)

[10] Selvarajah, K., Craven, M.P., Massey, A., Crowe, J., Vedhara, K., Raine-Fenning, N.: Native apps versus web apps: Which is best for healthcare applications? Human-Computer Interaction, Part II (2013)

[11] Xamarin: Xamarin - share code everywhere. https://www.xamarin.com/platform (2018) Accessed: 20.02.2018.

[12] Ionicframework: Storage. (https://ionicframework.com/docs/storage/)

[13] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine (2000)

[14] Masse, M.: Rest api design rulebook: Designing consistent restful web service interfaces. O'Reilly Media, Inc (2011)

[15] Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor (2012) http://www.rfc-editor.org/rfc/rfc6749.txt.

[16] TopoGrafix: Gpx: The gps exchange format. http://www.topografix.com/gpx.asp (2004) Accessed: 06.03.2018.

[17] The Dynastream Innovations Inc. ANT Products: Flexible & Interoperable Data Transfer (FIT) Protocol. ANT, thisisant.com. Rev2.3 edn. (2014)

[18] H. Butler and M. Daly and A. Doyle and S. Gillies and S. Hagen and T. Schaub: The GeoJSON Format. RFC 7946, RFC Editor (2016)

A

Quelltexte

```
1  import { NgModule }      from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { UserPage }      from '../pages/user/user';
4  import { PolarDataProvider } from "../../providers/polar/polar";
5
6  @NgModule({
7    imports:      [ BrowserModule ],
8    providers:    [ PolarDataProvider ],
9    declarations: [ AppComponent, UserPage],
10   exports:      [ AppComponent ],
11   bootstrap:    [ AppComponent ]
12 })
13 export class AppModule { }
```

Listing A.1: Aufbau eines AppModules

```
1  @Component({
2    selector: 'page-user',
3    templateUrl: 'user.html',
4    providers: [ PolarDataProvider, LocalDataProvider ]
5  })
6  export class UserPage {
7  /* . . . */
8  }
```

Listing A.2: Aufbau eines Components

```
import { Injectable } from '@angular/core';
import { HttpClient } from "@angular/common/http";

@Injectable()
export class PolarDataProvider {

    constructor(http: HttpClient) { }

    getData(url: string): Promise<any> {
        return new Promise((resolve, reject) => {
            this.http.get(url).subscribe(success => {
                resolve(success);
            }, error => {
                reject(error);
            });
        });
  }
}
```

Listing A.3: Grundlegender Aufbau eines HTTP-Services

```
<!-- Beispiel *ngIf -->
<p *ngIf="true">Das ist sichtbar</p>
<p *ngIf="false; else showThis">Das nicht</p>
<ng-template #showThis>
      <p>Aber das</p>
</ng-template>

<!-- Beispiel *ngFor -->
<ul>
```

```
10      <li *ngFor="let activity of activities"
11          (click)="showActivity(activity)">
12          Am {{activity.date | date:'dd.MM.y'}} wurden
13          {{activity['active-steps']}} Schritte gelaufen.
14      </li>
15  </ul>
16
17  <!-- Beispiel [ngSwitch] -->
18  <div [ngSwitch] = "exercise['detailed-sport-info']">
19      <p *ngSwitchCase = "RUNNING">Laufen</p>
20      <p *ngSwitchCase = "SQUASH">Squashen</p>
21      <p *ngSwitchCase = "GOLF">Golfen</p>
22      <p *ngSwitchDefault>Sportart ist nicht angegeben</p>
23  </div>
```

Listing A.4: Directives in der Anwendung

```
1   import { Storage } from '@ionic/storage';
2
3   export class MyApp {
4       constructor(private storage: Storage) { }
5
6       saveToken(token:any){
7           this.storage.set('token', token).then(() => {
8               console.log('Success');
9           });
10      }
11
12      getToken(){
13          this.storage.get('token').then((token) => {
14              resolve(token);
15          });
```

```
16      }
17  }
```

Listing A.5: Speichern und Abrufen von Daten mit dem IonicStorage

```
export const datatypes = {
  'exercise': {
    'id': 2,
    'name': 'exercise-transaction',
    'types': ['summary', 'heart-rate-zone', 'gpx', 'tcx', 'samples']
  }, 'activity': {
    'id': 1,
    'name': 'activity-transaction',
    'types': ['summary', 'steps', 'zones']
  }, 'physical': {
    'id': 0,
    'name': 'physical-information-transaction',
    'types': ['summary']
  }
};
```

Listing A.6: JSON, der Informationen über die zu speichernden Daten hält

```
export class MyApp {
    constructor(private storage: Storage) { }

    saveToken(token:any){
        localStorage.set('token', JSON.stringify(token));
    }

    getToken(){
        return JSON.parse(localStorage.get('token'));
    }
```

```
}
```

Listing A.7: Speichern und Abrufen von Daten mit dem LocalStorage

```
<gpx>
    <metadata>
        <name>GPS Track</name>
        <desc>My way to the university</desc>
        <author>
            <name>Lukas Hennig</name>
            <email id="lukas.hennig" domain="uni-ulm.de" />
            <link href="https://www.uni-ulm.de/">
                <text>Uni Ulm</text>
            </link>
        </author>
        <time>2018-09-03T14:54:06Z</time>
        <keywords>Ulm, Baden-Wuerttemberg, Germany</keywords>
        <bounds minlat="48.3997864"
                maxlat="48.4078229"
                minlon="9.949146"
                maxlon="10.0085152"/>
    </metadata>
    <wpt>...</wpt>
    <rte>...</rte>
    <trk>...</trk>
</gpx>
```

Listing A.8: Aufbau einer GPX-Datei

```
<Activities>
    <Activity Sport="Running">
        <Id></Id>
        <Lap StartTime="2018-03-09T10:00:00.0000Z">
```

```
            <TotalTimeSeconds>450.250</TotalTimeSeconds>
            <DistanceMeters>1250.0</DistanceMeters>
            <MaximumSpeed>4.2500001</MaximumSpeed>
            <Calories>520</Calories>
            <AverageHeartRateBpm>
                <Value>142</Value>
            </AverageHeartRateBpm>
            <MaximumHeartRateBpm>
                <Value>161</Value>
            </MaximumHeartRateBpm>
            <Intensity>Active</Intensity>
            <TriggerMethod>Location</TriggerMethod>
            <Track>
                ...
            </Track>
        </Lap>
    </Activity>
</Activities>
```

Listing A.9: Aufbau einer TCX-Datei

```
<Track>
    <Trackpoint>
        <Time>2018-03-09T10:00:00.0000Z</Time>
        <Position>
            <LatitudeDegrees>48.39841</LatitudeDegrees>
            <LongitudeDegrees>9.99155</LongitudeDegrees>
        </Position>
        <AltitudeMeters>478</AltitudeMeters>
        <DistanceMeters>25.0215</DistanceMeters>
        <HeartRateBpm>
            <Value>109</Value>
```

```
        </HeartRateBpm>
    </Trackpoint>
    <Trackpoint>
        ...
    </Trackpoint>
</Track>
```

Listing A.10: Aufbau eines Tracks einer TCX-Datei

```
{
    "type": "FeatureCollection",
    "features": [{
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": [102.0, 0.5]
        },
        "properties": {
            "prop0": "value0"
        }
    }, {
       "type": "Feature",
        "geometry": {
            "type": "LineString",
            "coordinates": [
                [102.0, 0.0],
                [103.0, 1.0],
                [104.0, 0.0],
                [105.0, 1.0]
            ]
        },
        "properties": {
```

```
            "prop0": "value0",
            "prop1": 0.0
        }
    }, {
        "type": "Feature",
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [
                    [100.0, 0.0],
                    [101.0, 0.0],
                    [101.0, 1.0],
                    [100.0, 1.0],
                    [100.0, 0.0]
                ]
            ]
        },
        "properties": {
            "prop0": "value0",
            "prop1": {
                "this": "that"
            }
        }
    }]
}
```

Listing A.11: Beispiel einer GeoJSON-Datei [18]

```
"token": {
    access_token: string,
    token_type : "bearer",
    expires_in: long,
```

```
    x_user_id: long
}
```

Listing A.12: Beispiel Access-Token

```
x_user_id: {
    exercise_transaction: [transactionIDs],
    activity_transaction: [transactionIDs],
    physical_information_transaction: [transactionIDs],
    user: {User}
}
```

Listing A.13: User und die Transaktion-IDs

```
transactionID: [
    exerciseIDs ||
    activityIDs ||
    physicalInformationIDs
]
```

Listing A.14: Transaction-IDs mit den jeweiligen IDs

```
exerciseID: {
    summary: {Summary},
    heart-rate-zone: {Heart-rate-zone},
    gpx: {GPX},
    tcx: {TCX},
    samples: [Samples]
}
```

Listing A.15: Struktur der Trainingsdaten lokal auf dem Handy

```
activityID: {
    summary: {Summary},
    zone_samples: [Zone_samples],
```

```
4     step_samples: [Step_samples]
5 }
```

Listing A.16: Struktur der Aktivitäts lokal auf dem Handy

```
1 physicalInformationID: {
2     summary: {Summary}
3 }
```

Listing A.17: Struktur der physischen Informationen lokal auf dem Handy

Abbildungsverzeichnis

2.1 Fitness-Tracker 8
2.2 Polar Flow - Training 11
2.3 Polar Flow - Tägliche Aktivitäten 12

3.1 Umfassende Übersicht der Cordova-Architektur 18
3.2 Databinding 21
3.3 Web-API 23
3.4 Beispiel URL 24
3.5 Ablauf der OAuth2-Authentifizierung 26
3.6 Aufbau einer FIT-Datei 30

5.1 Kommunikation mit der Polar-API 39
5.2 Login Prozess 41
5.3 Startansicht ohne Daten 42
5.4 Ansicht der täglichen Aktivitäten 44
5.5 Training 45
5.6 Physische Informationen 46
5.7 Übersicht der Nutzerdaten 47
5.8 Entity-Relationship-Modell des Datenbankentwurfes 50

6.1 Überblick der Realisierung der Kommunikation mit der Polar-API 57
6.2 Ablauf der Anfragen 58
6.3 Ablauf der Anfragen der Trainingseinheiten 59
6.4 Loginvorgang 61
6.5 Tägliche Aktivitäten 62
6.6 Training 63
6.7 Physische Informationen 64
6.8 Anzeigen der Nutzerdaten 65

Tabellenverzeichnis

3.1 Tabellarischer Vergleich von GPX, TCX und FIT 32

4.1 Funktionale Anforderungen 33
4.2 Nichtfunktionale Anforderungen 35

8.1 Versuchsergebnisse 72

Name: Lukas Hennig Matrikelnummer: 844768

Erklärung

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

Ulm, den .....

Lukas Hennig