ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

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

Konzeption und Realisierung einer mobilen Crowd Sensing Anwendung für iOS

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Sabrina Friedl
sabrina.friedl@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2016

Fassung 18. Januar 2016

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Durch die große Akzeptanz von Smartphones kommen neue mobile Anwendungsprogramme auf den Markt. Mithilfe dieser können Daten zu jeder Zeit erfasst und gesammelt werden. In Form von Formularen kann der Benutzer bewusst seine Daten in eine mobile Anwendung eintragen. Die Formulare können dabei für verschiedene Anwendungsbereiche konzeptioniert sein. Mögliche Anwendungsszenarien kommen aus der Medizin oder der Verkehrsüberwachung. Um nicht für jeden Bereich eine neue mobile Anwendung zu implementieren, verwaltet ein zentraler Server die Formulare aus allen Bereichen.

Die in der Arbeit beschriebene mobile Anwendung hat die Aufgabe diese generischen Formulare dem Benutzer zu präsentieren. Dabei wird mit dem Server mittels einer REST-API kommuniziert. Bei der Präsentation der Formulare wird auf eine komfortable Benutzerführung mithilfe von Guidelines geachtet. Die Anwendung prüft bei der Validierung der Formulare die Vollständigkeit und Richtigkeit. Das Ziel der Anwendung ist eine möglichst hohe Datenrate des Benutzers zu erhalten. Die Motivation des Benutzers Formulare auszufüllen wird zusätzlich durch ein Bonusprogramm und einer Statistik gesteigert.

Inhaltsverzeichnis

1 Einleitung

Smartphones sind aus dem derzeitigen Alltag nicht mehr wegzudenken. Im Sommer 2015 besitzt jeder zweite Deutsche ein Smartphone, das sein ständiger Begleiter ist [1]. Smartphones sind mehr als ein Kommunikationsgerät. Sie unterstützen den Benutzer in seinen alltäglichen Aufgaben, wie in der Verwaltung von Terminen oder als GPS-Navigationsgerät. Durch eine mobile Internetverbindung können schnell viele Informationen über das Smartphone ermittelt werden. Das Internet ermöglicht zudem eine Vernetzung über soziale Netzwerke. Smartphones dienen auch als transportables Unterhaltungsgerät, da Medien abgespielt und Spiele installiert werden können. Smartphones haben ebenfalls eine Digital- und Videokamera und unterschiedliche Sensoren, wie GPS-Empfänger, Barometer, Beschleunigungs- und Annäherungssensor integriert [2]. Durch diese Funktionalitäten wird das Smartphone zu einem optimalen Gerät der Datensammlung. Die Daten können vom Benutzer selbst in eine Smartphone-Anwendung eingetragen oder über die Sensoren gesammelt werden. Durch die große Akzeptanz

der Smartphones können große Mengen an Daten zu jeder Zeit erfasst werden. Das Sammeln von Daten kann dem Benutzer Vorteile z.B. bei der Verkehrsüberwachung bringen. Durch die Aufzeichnung der Fahrgeschwindigkeit kann ein potenzieller Stau ermittelt werden. Der Stau wird dem Verkehrsteilnehmer durch seine Navigationsanwendung mitgeteilt. Die Benutzer profitieren von einer Vielzahl gesammelter Daten. Je mehr Daten erfasst werden, desto größere Erkenntnisse sind über die aktuelle Verkehrslage bekannt. Für jede Art von Datenerfassung gibt es eine Anwendung, wie in diesem Beispiel $Google\ Maps$. Die Anwendung nutzt den GPS Sensor des Smartphones, um die Standortinformation zu erhalten [3]. Jede Anwendung spezialisiert sich auf einen Anwendungsbereich und nutzt ausgewählte Sensoren zur Datenerfassung. Die Umsetzung einer generischen Anwendung, die alle eingebauten Sensoren und Anwendungsbereiche unterstützt, ist aufwendig [4, 5].
Eine generische Anwendung mit Hilfe von Formularen bietet dagegen eine praktikable Lösung. Anstelle der eingebauten Sensoren werden Daten mithilfe von Formularen erhoben. Formulare haben eine einheitliche Struktur, inhaltlich können sie aber an verschiedene Anwendungsbereiche angepasst werden. Diesen Vorteil nutzt die im Rahmen dieser Arbeit entwickelte mobile Anwendung.

1.1 Ziel der Arbeit

Das Ziel der Arbeit ist die Konzeption und Realisierung einer mobilen Crowd Sensing Anwendung für das Betriebssystem iOS, wobei die Daten primär mittels Formularen erhoben werden. Bei der Umsetzung ist auf eine komfortable Benutzerführung zu achten. Ebenso sind die Rahmenbedingungen eines Smartphones zu berücksichtigen, wie z.B auch große Formulare auf kleinen Bildschirmen dargestellt werden müssen.

Als Grundgerüst für die Datenerhebung dienen Formulare, die auf einem Server zentral verwaltet werden. Die mobile Anwendung konzentriert sich auf die Anzeige der Formulare, indem sie die Formulare vom Server anfordert. Jedes Formular besitzt eine feste Struktur, damit die Anwendung ein Formular unabhängig vom Inhalt dynamisch

darstellen kann. Die Anwendung muss diese Struktur so interpretieren können, dass der Benutzer das Formular auf dem Smartphone ausfüllen kann.

1.2 Anwendungsszenarien

Formulare sind im Alltag häufig anzutreffen. Die im Rahmen dieser Arbeit entwickelte mobilen Anwendung ist so konzeptioniert, dass mehrere Anwendungsszenarien aus verschiedenen Bereichen möglich sind. Dieser Abschnitt beschränkt sich auf zwei mögliche Bereiche, die hier vorgestellt werden.

Medizin

Im Gesundheitssektor werden eine Vielzahl von Daten mittels gesundheitsbezogener Anwendung gesammelt und verarbeitet [6, 7]. Diese Anwendungen können in die Kategorien $Information$, $Coaching$, $Therapie$ und $Dokumentation$ unterteilt werden [8]. Beispielsweise gehören mobile Anwendungen, die Ernährungstipps bereitstellen in die Kategorie Information. Es werden nur informative Inhalte vermittelt. Coaching-Anwendungen können individuell auf den Benutzer eingehen. Z.B. kann eine Coaching-Anwendung aus übermittelten Daten ein Ernährungsprogramm zur Verfügungen stellen, welches an die Bedürfnisse des Benutzers angepasst ist. In der Kategorie Therapie wird das Smartphone als Therapiegerät verwendet. Dieses kann beispielsweise durch die GPS-Funktion Wanderrouten aufzeichnen [8].

Die in diesem Rahmen entwickelte Anwendung kann in den Bereich der $Dokumentation$ fallen (vgl. [8]). Wie der Name schon sagt, kann die Anwendung jegliche Art von gesundheitlichen Daten dokumentieren. So ist es durch regelmäßiges Eintragen der vitalen Werte Blutdruck, Puls und Gewicht möglich, ein Tagebuch zu führen. Werden diese Werte regelmäßig eingetragen, kann der Benutzer seine Werte in einem Diagramm ablesen. Der Benutzer hat einen guten Überblick über seine Gesundheit. Das regelmäßige Messen und Eintragen der Werte in der Anwendung kann der Patient von zuhause oder unterwegs erledigen, dadurch sind weniger Arztbesuche notwendig. Außerdem kann

das selbständige Messen dazu führen, dass der Patient seiner Gesundheit bewusst wird und für diese Verantwortung entwickeln kann [9].

Auch für den Arzt können diese regelmäßigen Messungen ein besseres Krankheitsbild des Patienten liefern, ohne sich auf "die verbale Darstellung" [8] des Patienten vertrauen zu müssen.

Gastronomie

Aber auch in der Gastronomie können verschiedene Formulare zum Einsatz kommen, wie beispielsweise für das Bestellen von Speisen und Getränken. Dem Benutzer werden verschiedene Formulare angeboten, in denen er seine Speisen und Getränke auswählen kann. Wahlweise kann dem Benutzer eine Sequenz von Formularen vorgegeben werden, die angibt, in welcher Reihenfolge er Formulare auszufüllen hat.

Bei Formularen in diesem Bereich bieten sich vor allem Listen an, bei der mehrfache Elemente gleichzeitig selektiert werden können. So können z.B. alle Suppen in einer Liste zusammengefasst werden. Der Benutzer hat dann die Wahl eine oder mehrere auszuwählen. Ein zusätzliches Textfeld für Sonderwünsche sollten dem Benutzer dennoch immer zu Verfügung stehen. Da sich die Speisekarte laufend ändert, werden diese Änderungen zentral vom Server angepasst. Diese Änderungen werden dann von den mobilen Geräten automatisch übernommen. Eine Änderung der Anwendung auf Implementierungsebene ist nicht erforderlich.

1.3 Aufbau der Arbeit

Eine Übersicht der einzelnen Kapitel dieser Arbeit wird in der Abbildung 1.1 dargelegt. Die Arbeit ist in sechs Kapitel untergliedert.
Für ein gutes Verständnis der gesamten Arbeit werden im Kapitel 2 die Grundlagen behandelt. Dazu zählen wesentliche Begriffe wie $Crowd\ Sensing$ und $REST - API$. Zu den Grundlagen zählen auch der Aufbau des Servers und das Format JSON, in

Abbildung 1.1: Kapitelübersicht

dem die Formulare beschrieben werden. Als Ergänzung wird die verwendete Programmiersprache $Swift$ vorgestellt. Im 3. Kapitel werden die Anforderungen erläutert, die an die mobile Anwendung gestellt werden. Wie die Anforderungen umgesetzt werden, wird in Kapitel 4 anschaulich dargestellt. Dabei wird erst durch die Dialogstruktur und Dialoggestaltung einen Überblick über die Anwendung gegeben. Ein wesentlicher Punkt dieser Arbeit sind die Formulare, dessen Aufbau anschließend erklärt wird. Danach folgt die Generierung und Validierung von Formularen. Anschließend werden wichtige Abläufe im System und die Architektur der mobilen Anwendung erläutert. Nach dem Implementierungsteil folgt im Kapitel 5 der Abgleich der funktionalen und nicht funktionalen Anforderungen. Der abschließende Teil dieser Arbeit in Kapitel 6 bietet eine Zusammenfassung mit einem Ausblick.

2

Grundlagen

In diesem Kapitel werden die Grundlagen, die für das weitere Verständnis der Arbeit wichtig sind, behandelt.

2.1 Crowd Sensing

Der Begriff $Crowd\ Sensing$ beschreibt ein effizientes Sammeln von Daten durch menschliche Beteiligung [10]. Für eine flächendeckende Datenerfassung in einem größeren Stadtgebiet sind die Kosten für Sensoren und deren Wartung hoch. Werden nun die Sensoren (z.B. GPS Sensor) in den mobilen Endgeräten genutzt, kann dieser Kostenaufwand gespart werden. Die Kosten für die Smartphones und somit für die Sensoren liegen beim Verbraucher. Dazu bietet die hohe Mobilität einen größeren Bereich der Datenerfassung.

Durch die heutzutage weite Verbreitung von Smartphones können entweder $unbewusst$ oder $bewusst$ Daten erfasst werden. Der Benutzer sammelt Daten unbewusst, wenn eine entsprechende Anwendung im Hintergrund läuft und Daten erfasst (vgl. [10]). Eine mögliche Anwendung ist die Überwachung des Verkehrs durch $Google\ Maps$. Die Anwendung überträgt automatisch die Standortinformationen des Smartphone an einen Server. Der Benutzer interagiert nicht mit dem Gerät.
Dagegen muss der Benutzer bei anderen Anwendungen bewusst Daten eintragen, die die Anwendung weiterleitet. Diese Art von Datenerfassung kann zu Problemen bei der Zuverlässigkeit der Datenerfassung führen, wenn zu wenige Daten eingetragen werden. Nur wenn der Benutzer sich regelmäßig und aktiv entscheidet Daten einzutragen, ist eine hohe Zuverlässigkeit gewährleistet.

In dieser Arbeit steht das bewusste Sammeln von Daten im Vordergrund. Der Benutzer wird aufgefordert seine Daten in die Anwendung einzutragen, damit diese serverseitig gesammelt werden können. Die Daten werden mithilfe von Formularen erfasst. Dabei sind die Formulare die Sensoren, durch die Daten abgetastet werden. Formulare haben den Vorteil einer einheitlichen Struktur mit einem variablen Inhalt. Dadurch können Formulare in verschiedenen Bereichen eingesetzt, um Daten zu erfassen. Das entspricht einer generischen Form der Datenerfassung.

2.2 Aufbau des Servers

Der Server ist unter anderem für die zentrale Verwaltung der Formulare verantwortlich. Er steht im Rahmen dieser Arbeit zur Verfügung. In Abb. 2.1 ist die Struktur des Server veranschaulicht, wobei nachfolgend nur auf die hervorgehobenen Elemente eingegangen wird.
Für diese Arbeit sind der $Webservice\ Manager$ und der $Module\ Manager$ die wichtigsten Komponenten des Servers. Alle Komponenten des Servers kommunizieren über einen $Enterprise\ Service\ Bus\ (ESB)$. Der Webservice Manager wird für die Kommunikation mit den $Clients$ benötigt. Wird beispielsweise ein Formular vom Benutzer gefordert, sendet der Webservice Manager eine Nachricht über den ESB an den Module

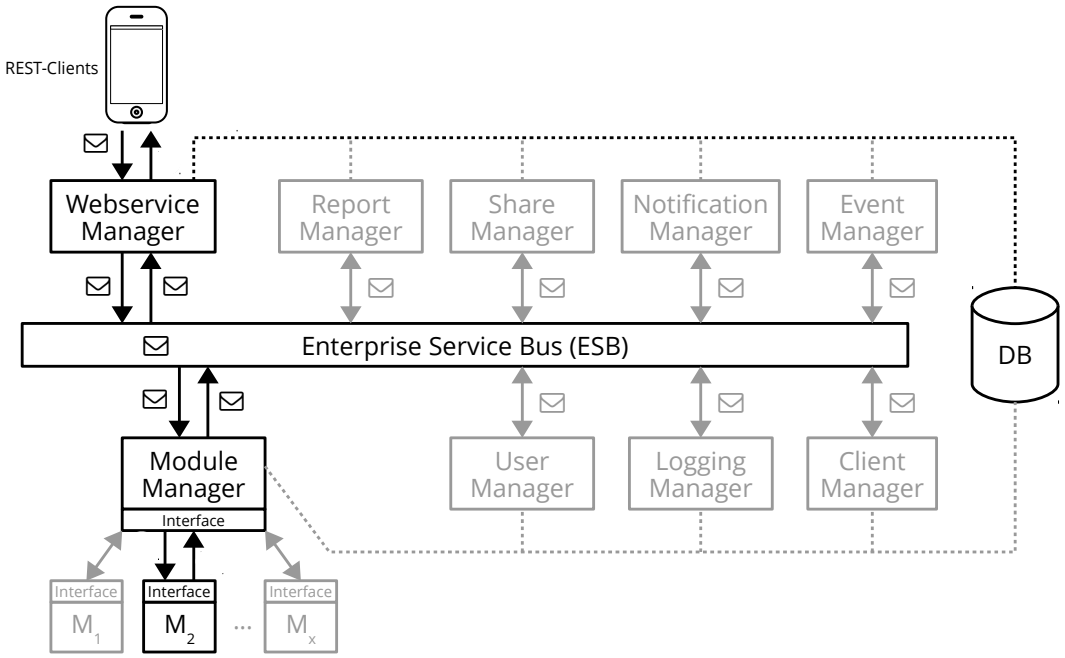

Abbildung 2.1: Struktur des Servers [11]

Manager. Dieser leitet die Nachricht an das entsprechende Modul weiter. Dieses Modul kann dann das gewünschte Formular erstellen und zurück an den Module Manager schicken. Der Webservice Manager erwartet das Formular, damit dieser es in Form einer HTTP-Response an den Client schicken kann.

Das Zurückschicken eines ausgefüllten Formulars funktioniert in ähnlicher Weise. Wieder wird eine Anfrage an den Webservice Manager geschickt. Diese wird an den Module Manager weiter geleitet. Der Module Manager ist auch für die serverseitige Eingabevalidierung zuständig. Ist diese nicht erfolgreich, wird die Fehlermeldung an den Client zurück geschickt. Wurden dagegen keine Fehler gefunden, kann der Modul Manager das Modul in der Datenbank speichern. Das Modul selbst erstellt eine Liste mit Formularen, die der Client als nächstes ausfüllen muss. Der Webservice liefert diese Liste wieder zurück an den Client [11].

2.3 REST-API

Rest-API ($Representational\ State\ Transfer\ Application\ Programming\ Interfaces$) ermöglicht dem Client Informationen mit einem Server auszutauschen. Um Ressourcen abzufragen, verwendet die Anwendungsschnittstelle $Uniform\ Resource\ Identifiers$ (URIs). Eine beispielhafte URI wird im Listing 2.1 gezeigt. URIs weisen jeder Ressource

```
1 beispiel.de/api/modules/Masterarbeit/FancyModule/start
```

Listing 2.1: Beispiel URI

eine eindeutige Adresse zu. So ist sichergestellt, dass mehrere Anfragen an die gleiche URI immer den gleichen Inhalt zurückliefern.
Diese Übertragung von Daten basiert auf dem $Hypertext\ Transfer\ Protocol$ (HTTP), eine auf Nachrichten basierte Sprache. GET, $DELETE$, $POST$ und PUT sind die bekanntesten Methoden, um Daten zu verwalten. GET fragt Ressourcen vom Server ab und $DELETE$ löscht dagegen eine Ressource auf dem Server. $POST$ und PUT übertragen beide eine Ressource zum Server mit dem Unterschied, dass bei $POST$ eine Ressource serverseitig erstellt wird. PUT hingegen aktualisiert eine neue Ressource, beziehungsweise erstellt eine neue Ressource, falls diese noch nicht existiert [12].

Ein möglicher GET-Request und der dazugehöre Response sind in Listing 2.2 und 2.3 dargestellt.

```
1 GET /index.html HTTP/1.1
2 HOST www.apache.org
```

Listing 2.2: Beispiel HTTP-GET-Request

```
1 HTTP/1.1 200 OK
2 Server Apache/2.0.48-dev (Unix)
3
4 [Ressource]
```

Listing 2.3: Beispiel HTTP-Response

2.4 JSON

JSON bedeutet $JavaScript\ Object\ Notation$ und ist ein Datenaustauschformat. Es ist sowohl für den Menschen als auch für Maschinen einfach zu lesen. Da es sich bei JSON um ein reines Textformat handelt, ist es von jeder Programmiersprache unabhängig. JSON baut auf zwei Arten von Strukturen auf, welche in den Abbildungen 2.2 und 2.3 skizziert sind. Zum einen besteht JSON aus einem Objekt und zum anderen aus einer Liste. Ein Objekt ist durch eine öffnende und eine schließende geschweifte Klammer gekennzeichnet, sowie einen String und einem Wert, die durch einen Doppelpunkt getrennt sind. Mehrere Paare in einem Objekt werden durch Kommata getrennt [13].

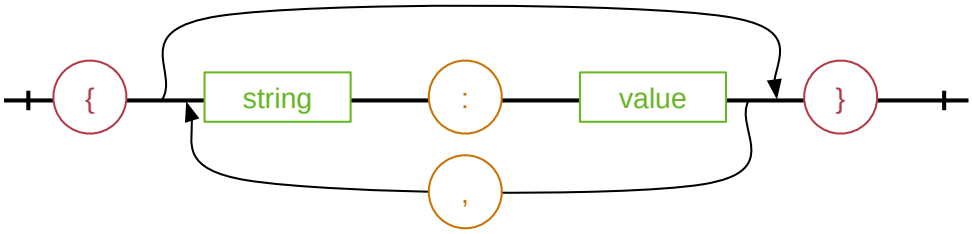

Abbildung 2.2: JSON Objekt [13]

Eine Liste (Abb. 2.3) wird durch eckige Klammern beschrieben. In ihr befinden sich Werte, die ebenfalls durch ein Komma voneinander getrennt werden (vgl. [13]). Wobei ein Wert sowohl ein Objekt, als auch eine Liste, einen String, eine Zahl, true oder false annehmen kann. Ebenfalls erlaubt ist auch ein Null-Wert.

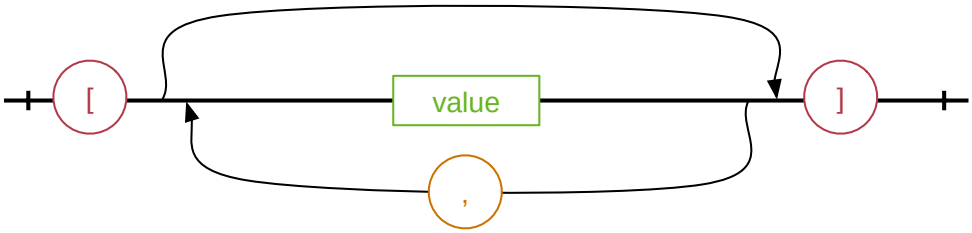

Abbildung 2.3: JSON Liste [13]

2.5 Swift

Swift ist die aktuelle Programmiersprache von Apple, die 2014 vorgestellt wurde. Sie dient dazu, Anwendungen für die mobilen Endgeräte von Apple oder Desktopprogramme zu entwickeln. Swift ist mit der zuvor schon entwickelten Programmiersprache $Objective-C$ kompatibel. Anwendungen, die bereits in Objective-C geschrieben sind, lassen sich durch Swift erweitern und verbessern [14].
Swift hat Merkmale von C als auch von Objective-C. Es enthält primitive Datentypen und Operatoren. Zudem ist es auch objektorientiert und bietet Klassen, Protokolle und Generics. Auch ein Teil der funktionalen Programmierung steckt in Swift. So gibt es die klassischen map oder $filter$ Methoden, die auf Listen angewendet werden. Außerdem werden Tupels und mehrere Rückgabewerte unterstützt, was man auch in der funktionalen Programmiersprache $Haskell$ wiederfindet.
Eine Besonderheit der Sprache, die es in C nicht gibt, ist die Möglichkeit von optionalen Objekten. Grundsätzlich können Objekte in Swift nicht $null$ sein. Ausnahmen, bei denen es sinnvoll ist, dass ein Objekt null sein kann, werden dementsprechend mit einem Fragezeichen $(?)$ gekennzeichnet.
Ein besonderer Fokus wurde auf die Einfachheit und gute Lesbarkeit gelegt. Dafür wird zum Beispiel im Vergleich zu Java keine $main$-Methode benötigt. Aber auch die Semikolons, am Ende jeder Zeile, sind keine Pflicht mehr [15].

Im Vergleich zu seinem Vorgänger bietet Swift die Möglichkeit von $Playgrounds$ an. Diese ermöglichen dem Entwickler schnelles und einfaches Testen von Code, ohne dass in das bestehende Programm eingegriffen wird. Bei jeder Eingabe wird der Code sofort compiliert und die Ausgabe angezeigt. Die Ausgabe kann nicht nur aus Zahlen oder Text bestehen, der Playground kann auch Grafiken zeichnen [15].

3

Anforderungsanalyse

Die Anforderungsanalyse beschreibt einen Entwicklungsprozess, um System- oder Software-Anforderungen zu definieren [16]. Zu Beginn des Prozesses werden die Anforderungen ermittelt und beschrieben. Beendet wird der Prozess durch den Abgleich der Anforderungen im Kapitel 5. Die Anforderungen beschreiben die Eigenschaft oder eine Bedingung des Systems. Unterschieden wird in funktionale und nichtfunktionale Anforderungen, die anschließend in diesem Kapitel vorgestellt werden.

3.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben, welche Aufgaben an die Anwendung gestellt werden.

1. **Native Anwendung:** Die Anwendung soll nativ gestaltet sein, das heißt die Anwendung wird im Rahmen dieser Arbeit speziell für das Betriebssystem iOS implementiert. So sollen auch die vom Betriebssystem bereitgestellten Oberflächenelemente genutzt werden [17].
2. **Dynamische Formulare:** Die Formulare sollen dynamisch, anhand einer vordefinierten Struktur gehalten werden. Die Struktur ist vom Server vorgegeben.
3. **Auswahl von Formulare:** Der Benutzer hat die Möglichkeit aus einer Reihe von Formularen eines auszuwählen und zu bearbeiten.
4. **Serverkommunikation:**
   - Das System bietet eine Serverkommunikation, die gewährleistet, dass die Formulare auf dem Server der Anwendung zur Verfügung stehen.
   - Ausgefüllte Formulare müssen wieder an den Server übertragen werden. Die Struktur des ausgefüllten Formulars ist vom Server vorgegeben und muss übernommen werden.
5. **Validierung von Formularen:** Ausgefüllte Formulare sollen auf ihre Richtigkeit geprüft werden. Dafür muss die Spezifikation eines Formularfeldes überprüft werden.
6. **Mitteilungen:** Die Anwendung soll auf Mitteilungen vom Server reagieren können und diese an den Benutzer weiterleiten.
7. **Bonusprogramm:** Die Anwendung soll ein Punktesystem oder Bonusprogramm enthalten. Das Bonusprogramm soll den Benutzer motivieren, Formulare auszufüllen.
8. **Interaktive Diagramme:** Gemessene Werte sollen in Diagrammen interaktiv veranschaulicht werden. Der Benutzer soll x- und y-Achsenabstände variabel verstellen können und innerhalb des Diagramm navigieren können.

3.2 Nichtfunktionale Anforderungen

Im Vergleich zu den funktionalen Anforderungen, ziehen die nichtfunktionalen Anforderungen das System als Ganzes in Betracht.

1. **Aufgabenangemessenheit:** Die Anwendung soll dem Benutzer beim Ausfüllen von Formularen unterstützen, damit Daten effektiv und effizient gesammelt werden können. Dafür muss die Anwendung alle funktionalen und nichtfunktionalen Anforderungen erfüllt haben.
2. **Verfügbarkeit:**
   - Die Anwendung soll für alle iPhones mit der iOS Version 8.0 oder höher geeignet sein.
   - Die Bedienung der Anwendung soll flüssig sein, d.h. kurze Antwortzeiten. Insbesondere die Ladezeit eines Formulars soll möglichst kurz sein.
   - Die Serverkommunikation soll transparent und verlustfrei sein. Die Daten sollen konsistent und ohne Verlust gespeichert werden können.
3. **Selbstbeschreibungsfähigkeit:** Die Anwendung soll intuitiv sein. Der Benutzer soll sich ohne Hilfe im System zurechtfinden.
4. **Fehlertoleranz:** Die Anwendung soll fehlertolerant sein. Falsche Eingaben soll das System erkennen und behandeln. Der Benutzer muss die Möglichkeit haben, fehlerhafte Eingaben zu korrigieren.
5. **Erwartungskonformität:** Das Erscheinungsbild der Anwendung soll einheitlich sein und den allgemein verwendeten Gestaltungsregeln oder Styleguides entsprechen.

4

Implementierung

Dieses Kapitel befasst sich mit der Umsetzung der Anforderungen, die an die Anwendung gestellt worden sind. Einen groben Überblick über die gesamte Anwendung geben die Dialogstruktur und die Dialoggestaltung am Anfang des Kapitels. Dabei wird besonders auf die Umsetzung der Formularseite eingegangen. Im Weiteren wird zunächst der Aufbau eines Formulars vorgestellt, um anschließend die dynamische Generierung und Validierung der Formulare zu erklären. Als nächstes wird die Architektur der mobilen Anwendung veranschaulicht und abschließend verwendete Frameworks vorgestellt.

4.1 Dialogstruktur

Die Dialogstruktur (siehe Abb. 4.1) zeigt den Aufbau der Anwendung. Jeder Dialog repräsentiert einen ViewController, der eine View besitzt, mit der eine Interaktion möglich ist.

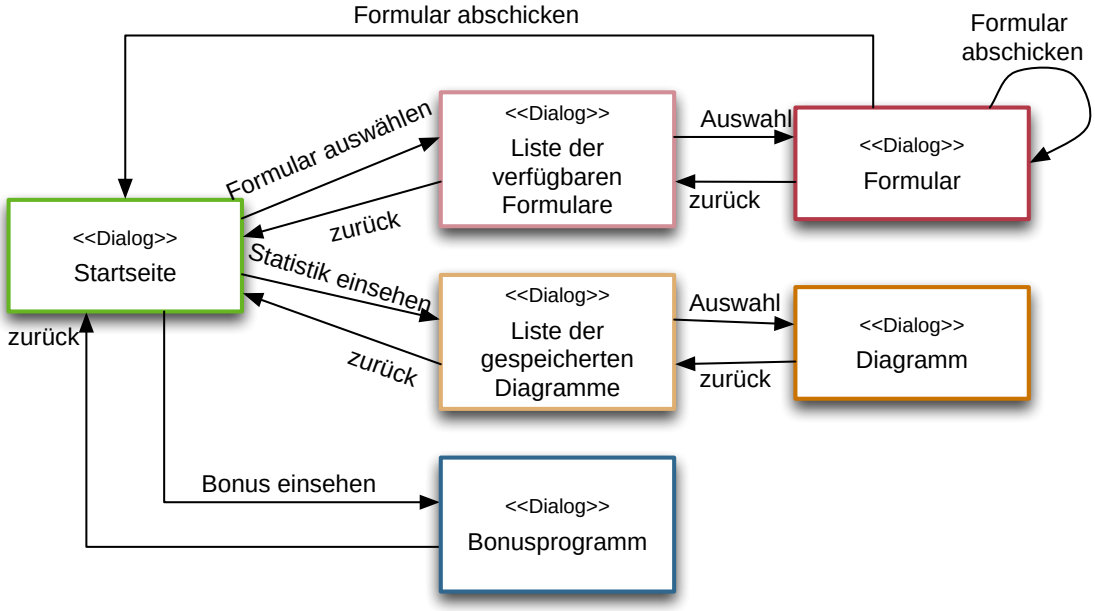

Abbildung 4.1: Dialogstruktur

Beim Start der Anwendung öffnet sich die $Startseite$ automatisch. Hier hat der Benutzer die Möglichkeit Formulare auszuwählen, seine gespeicherten Werte anzusehen oder seinen Bonus zu überprüfen. Dieser Dialog wird durch die Schaltfläche auf der Startseite $Bonus\ einsehen$ erreicht, siehe dazu Kapitel 4.2.5.

Möchte der Benutzer ein Formular ausfüllen, gelangt er zunächst zu einer Auswahl an bereitgestellten Formularen. Nach der Auswahl erscheint der Dialog $Formular$. Ein Formular kann weitere Formulare implizieren, die alle korrekt ausgefüllt und abgeschickt werden müssen. Sind alle Formulare ausgefüllt, gelangt der Benutzer zur Startseite zurück.

Durch das Ausfüllen von Formularen werden numerische Werte, die in ein Formular eingetragen werden, automatisch in der Datenbank des Clients gespeichert. Diese Werte können unter $Statistik\ eingesehen$ werden. Auch hier muss der Benutzer erst in einer Liste eine Auswahl treffen, um zu dem dazugehörigen $Diagramm$ zu gelangen.

Der Benutzer hat zu jedem Zeitpunkt die Möglichkeit, eine Seite zurück zu navigieren. Ausnahme bildet der Dialog $Formular$. Sobald ein Formular abgeschickt wurde und das nachfolgende Formular erscheint, kann der Benutzer nicht mehr zum vorherigen Formular zurückkehren. Der Benutzer muss solange die gewünschten Formulare ausfüllen, bis er automatisch wieder zu der Startseite gelangt. Die vorgelegte Reihenfolge der auszufüllenden Formulare wird eingehalten, denn ein mehrmaliges Abschicken eines Formulars ist nicht erwünscht.

4.2 Dialoggestaltung

Nicht nur in der Architektur, sondern auch beim Entwerfen einer iOS Anwendung trifft das Prinzip "Weniger ist mehr" [18], wie es Mies van der Rohe (ein deutsch-amerikanerischer Architekt) sagte, zu. Auch in den $iOS\ Human\ Interface\ Guidelines$ wird von Klarheit gesprochen. Farben und Formen sollen zur Funktion der Anwendung passen und nicht nur rein dekorativ sein [19]. Das heißt Texte sind gut lesbar, die Formen sind einfach geometrisch gehalten und es herrscht ein klar definiertes Farbschema [19]. Alle Elemente, die in der Anwendung verwendet werden, sind von Apple definierte $UI\ Elemente$. Auch die standard-Schrift $Schrift\ Helvetica\ NeueInterface - Regular$ wurde beibehalten. Wichtig ist ebenfalls die Größe der einzelnen Interaktionselemente (Buttons oder Textfelder). Diese sollten für eine optimale Bedienung eine Mindestgröße von 44x44 Pixel besitzen (siehe [20]).

Die gesamte Anwendung ist in einem Farbschema gehalten, dadurch wirkt die Anwendung harmonisch. Die Grundfarbe ist türkis (siehe Abb. 4.2), eine Mischung der Farben blau und grün.

Die Farbe $petrol$ in der Abb. 4.2 ist als Hintergrundfarbe dominierend. Sie wird auch bei dem Formularfeld Ckeckbox verwendet. Für eine bessere Unterscheidung dieser

Abbildung 4.2: Farbschema der Anwendung

Formularfeldern sind radioButtons in der Farbe $cyan$. In den iOS Human Interface Guidelines wird empfohlen, Interaktivitäten in einer $Key - Farbe$ anzuzeigen [19]. Ist ein Element selektierbar, so wechselt die Schrift oder das Symbol auf $darkpetrol$. Die Schriftfarbe der einzelnen Formularfelder ist schwarz. Durch den großen Kontrast zur Hintergrundfarbe ist der Text gut lesbar. Die Farben für den Hintergrund der Diagramme bestehen aus einem Verlauf, der aus $darkpetrol$ und $lightpetrol$ besteht, wobei die Transparenz bei $lightpetrol$ auf 0,4 gesetzt ist.

Für besondere Hervorhebungen in der Anwendung werden ein Pinkton und ein Gelbton eingesetzt (vgl. Farbton $pink$ und $gelb$ in der Abb. 4.2), da sich rot und gelb als Warnfarbe durchgesetzt haben. Diese Farben müssen dennoch in das Farbschema integriert sein, damit sie nicht von der Aufgabe der Anwendung ablenken, sondern lediglich den Benutzer in der Anwendung unterstützen [19]. Der Text wird bei einer Falscheingabe pink eingefärbt, falls der Text nicht mit dem vorgegebenen Pattern übereinstimmt. Gelb (mit einem Alpha-Kanal von 0,5) werden Pflichtfelder markiert, die nicht ausgefüllt wurden. Durch die zwei unterschiedlichen Farben kann der Fehler schneller erkannt und behoben werden.

4.2.1 Startseite

Die Startseite ist übersichtlich gehalten (siehe Abb. 4.3). Sie besteht nur aus drei Buttons. Um Dynamik in die Gestaltung der Anwendung hineinzubringen, besteht der Hintergrund aus mehreren Dreiecken verschiedener Größe. Die Farben passen zu dem anwendungsübergreifendem Farbschema. Der gleiche Hintergrund ist auf allen weiteren Dialogen zu finden. Die Buttons selbst bestehen aus Rechtecken mit abgerundeten Ecken. Auch Buttons außerhalb der Startseite haben dieses Aussehen. Die Farbe ist weiß mit einem Alphakanal von 0,7. Durch die geringe Transparenz verschmelzen die Buttons leicht mit dem Hintergrund und passen besser zueinander.

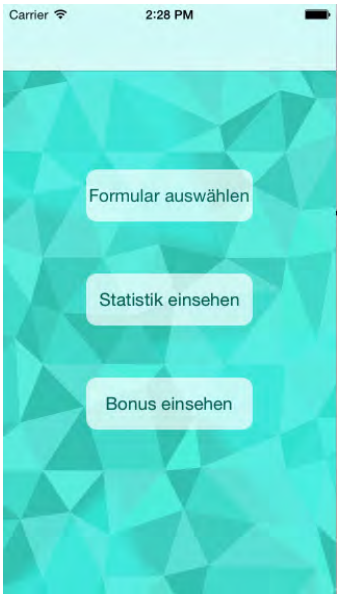

Abbildung 4.3: Ansicht der Startseite

4.2.2 Listendialog

Der Hintergrund der Startseite zieht sich wie ein roter Faden durch die gesamte Applikation, so auch bei den Listendialogen (vgl. Abb. 4.4). Die einzelnen Einträge in der Liste sind leicht transparent, sodass der Hintergrund dahinter sichtbar ist.

Die Liste wird dynamisch erzeugt und kann beliebig viele Elemente haben. In der gesamten Anwendung gibt es zwei Listendialoge. Bei der Auswahl von Formularen werden

die Namen der Formulare aufgelistet. Unter dem Menüpunkt $Statistik\ einsehen$ gibt es mehrere Diagramme zur Auswahl, die durch eine Listenauswahl gekennzeichnet sind. Diagramme können aus dieser Liste mittels einem $Löschen$-Button entfernt werden, der durch Wischen von Rechts auf dem Listeneintrag angezeigt wird (siehe Abb. 4.4). In dem Beispiel in der Abb. 4.4 werden alle bisher gespeicherten Werte des Diagramms $Blutdruck$ in der Datenbank gelöscht. Sobald allerdings der Benutzer neue Blutdruckwerte in die Anwendung einträgt, werden diese wieder in der Datenbank gespeichert und das Diagramm $Blutdruck$ erscheint zur Ansicht in der Liste.

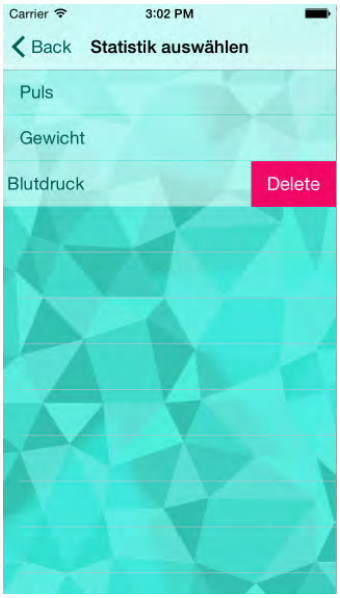

Abbildung 4.4: Ansicht der editierbaren Liste der Statistik

4.2.3 Formularseite

Das Design der Formularseite orientiert sich grundsätzlich an bekannten Formularlayouts. So stehen beispielsweise Labels möglichst nahe über dem Eingabefeld. Formularfelder sollen einspaltig aufgebaut sein, das heißt alle Fragen stehen untereinander und nicht nebeneinander [21].

Jedes Formularfeld besitzt ein Label, welches das Feld beschreibt. Damit diese Zusammengehörigkeit deutlich wird, sind diese zwei Elemente mit einem weißen Hintergrund

hinterlegt. Der Abstand zwischen den zusammengehörigen Elemente ist kleiner, als zu anderen Formularelementen.

Einheiten, die aus mehrere Formularfeldern und dem passenden Label bestehen, heißen CompositeFelder. Gekennzeichnet wird diese größere Einheit ebenfalls mit einem weißen Hintergrund (vgl. Abb. 4.5).

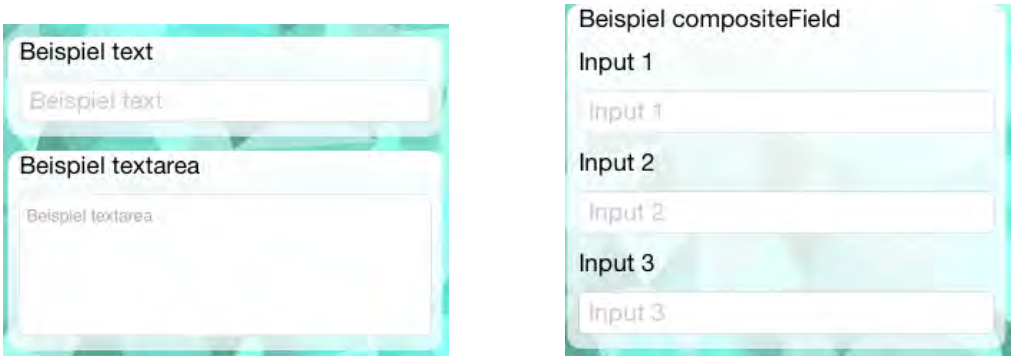

Abbildung 4.5: Vergleich von zwei einfachen Formularfeldern und einem CompositeField

Anders als im HTML Standard unterscheiden sich Radiobuttons und Checkboxen nicht in der Form, sondern lediglich in der Farbe. Denn in iOS werden die sogenannten $An/Ausschalt - Elemente$ zusammengefasst (siehe Abb. 4.6). Allerdings müssen die gleichen Bedingungen bezüglich der Auswahl der Elemente gelten. In einer Gruppe von Radiobuttons muss genau ein Element ausgewählt sein. Bei Checkboxen kann kein, ein oder mehrere Elemente ausgewählt sein.

Auch sind Auswahlen in einer Listenform bei iOS optisch anders als eine Liste, die auf dem Desktop in HTML dargestellt ist (siehe Abb. 4.7). Im Gegensatz zu einer Drop-Down Liste in HTML, bietet iOS für eine einfache Listenauswahl die $UIPickerView$ an. In einer Drop-Down Liste ist mit einem Klick auf die Schaltfläche die komplette Liste auf einmal sichtbar. Dagegen beschränkt die $UIPickerView$ die sichtbaren Listenelemente auf eine gewissen Anzahl. Hat die Liste zu viele Einträge, können sie nicht auf einmal betrachtet werden.

Für eine mehrfache Listenauswahl wird auf die Listenform zurückgegriffen, die im Listendialog (siehe Kapitel 4.2.2) verwendet wird.

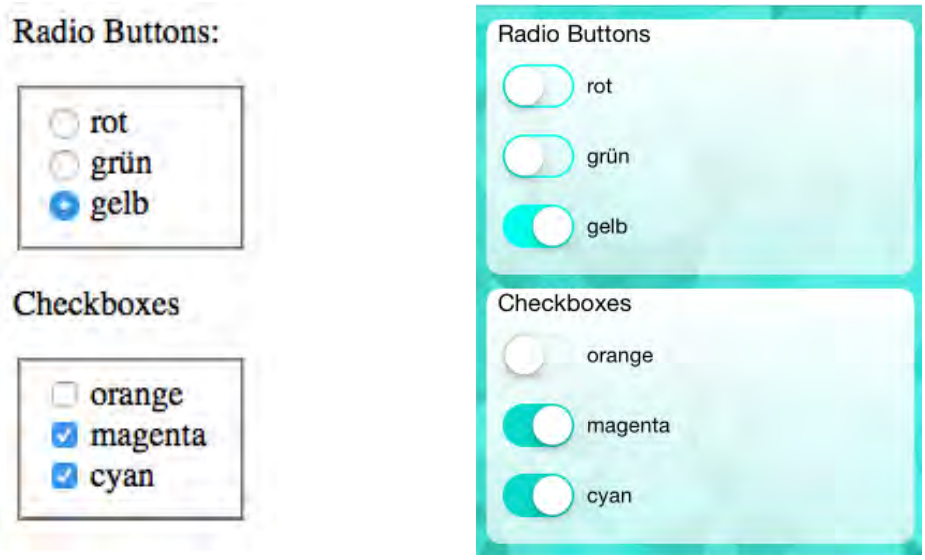

Abbildung 4.6: Vergleich HTML und iOS Layout von Radiobuttons und Checkboxen

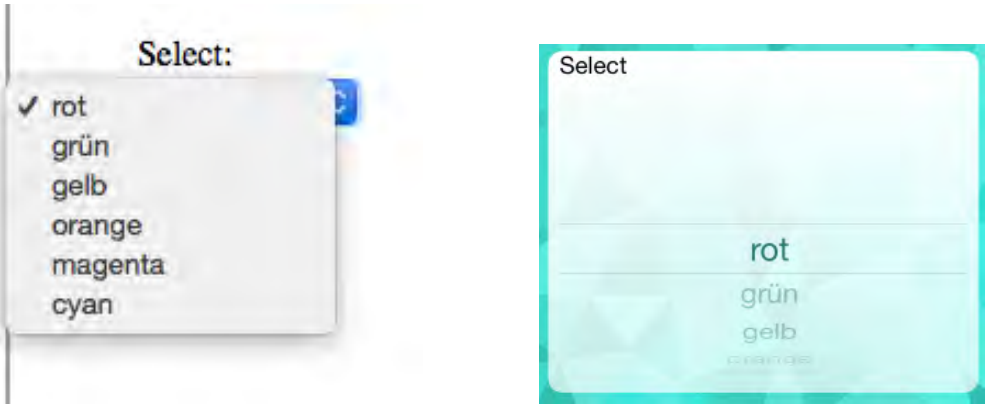

Abbildung 4.7: Vergleich HTML-Desktop und iOS Layout von einer einfachen Listenauswahl

Für ein Formular stehen auch Datum und Uhrzeit zur Verfügung. Die Eingabe ist prinzipiell eine $UIPickerView$ mit mehreren Spalten. IOS bietet die Liste für Datum und Uhrzeit standardmäßig an, der sogenannte $Date\ Picker$. Der $Date\ Picker$ verfügt über die Elemente $Time$, $Date$, $Time\ and\ Date$ und $Count\ Down\ Timer$. Jedoch ist man dann an die vordefinierte Anzeige gebunden. Die Anzeige der Uhrzeit erfolgt wie im Amerikanischen. Es gibt 12 Stunden, sowie am für Vormittag und pm für Nachmittag.

Da solch eine Schreibweise im Deutschen weniger gebräuchlich ist, wurde ein eigenes Format für die Anzeige der Uhrzeit definiert (siehe Abb. 4.8). Der Tag hat hier 24 Stunden. Für eine übersichtliche und eindeutige Schreibweise, werden die Stunden von den Minuten mit einem Doppelpunkt getrennt und danach die Bezeichnung Uhr hinten angestellt. Dafür wird ein UIPickerView erstellt, der vier Listen besitzt. Jeweils eine Liste von 0 bis 24 für die Stunden und 0 bis 59 für die Minuten. Die anderen zwei Listen beinhalten nur einen Doppelpunkt (:) oder das Element Uhr.

Abbildung 4.8: Ansicht der Eingabemöglichkeit für Datum und Uhrzeit

Verschiedene Tastaturen

Je nach Anwendungsfall wird dem Benutzer eine spezifische virtuelle Tastatur angeboten. Bei der Standardtastatur fehlt z.B das @ Zeichen, das für die Eingabe einer Email-Adresse aber wichtig ist. Somit wird eine Tastatur mit den fehlenden Zeichen (siehe Abb. 4.9) angeboten. So muss der Benutzer nicht die Ansicht zu den Sonderzeichen wechseln. Das erleichtert die Eingabe.

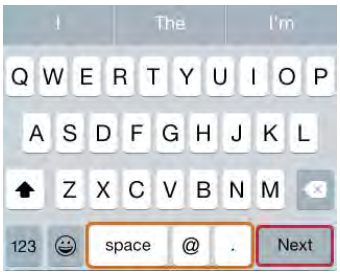

Abbildung 4.9: E-Mail Tastatur

Für die Eingabe von URLs wird anstatt einer Leertaste beispielsweise das länderspezifische $.com$ in der Hauptansicht angeboten. Ähnlich wie bei der E-Mail Tastatur, werden diese Tasten (Abb. 4.10) häufig für die Eingabe von URLs benutzt.

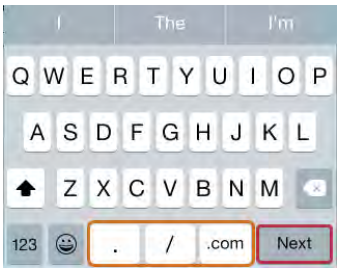

Abbildung 4.10: URL Tastatur

Für die Eingabe von Zahlen, bietet iOS ein Ziffernfeld (Abb. 4.11). Das Ziffernfeld wird für die Eingabe negativer Zahlen durch weitere Buttons ergänzt (in Abb. 4.11 orange markiert). Für eine Eingabe einer Telefonnummern ist die Taste mit dem Punkt nicht vorhanden (in Abb. 4.11 blau markiert), da sie an dieser Stelle nicht benötigt wird. Das erleichtert dem Benutzer die Eingabe, da er nicht in die Ansicht der Zahlen und Zeichen wechseln muss. Durch das Ziffernfeld können Fehler in der Eingabe vermieden werden, da nur Zahlen benutzt werden können. Eine Validierung des Textfeldes ist trotz der angepassten Tastatur notwendig.

Abbildung 4.11: Zahlen Tastatur

Der $next$-Button (in den Abbildungen 4.9 und 4.10 rot markiert) führt den Benutzer in das nächste Textfeld. Wenn kein anschließendes Textfeld folgt, schließt sich die Tastatur.

Bei der Tastatur für Zahlen (Abb. 4.11) fehlt der $next$-Button. Da die Möglichkeit bestehen soll, durch eine Taste zum nächsten Formularfeld zu gelangen, wurde die Tastatur durch einen $Weiter$-Button (grün markiert) ergänzt.

Die Tastaturen passen sich zudem der Sprache an. Aus dem englischen $next$ wird im Deutschen $weiter$ und aus einem $.com$ wird, wie es in deutschen URLs häufig vorkommt, ein $.de$.

4.2.4 Statistik

Die Statistik wird eingesetzt, um dem Benutzer einen Überblick über den Verlauf seiner Eingaben mittels Diagrammen zu verschaffen. Angezeigt werden alle numerischen Daten, die in den Formularen eingegeben wurden.

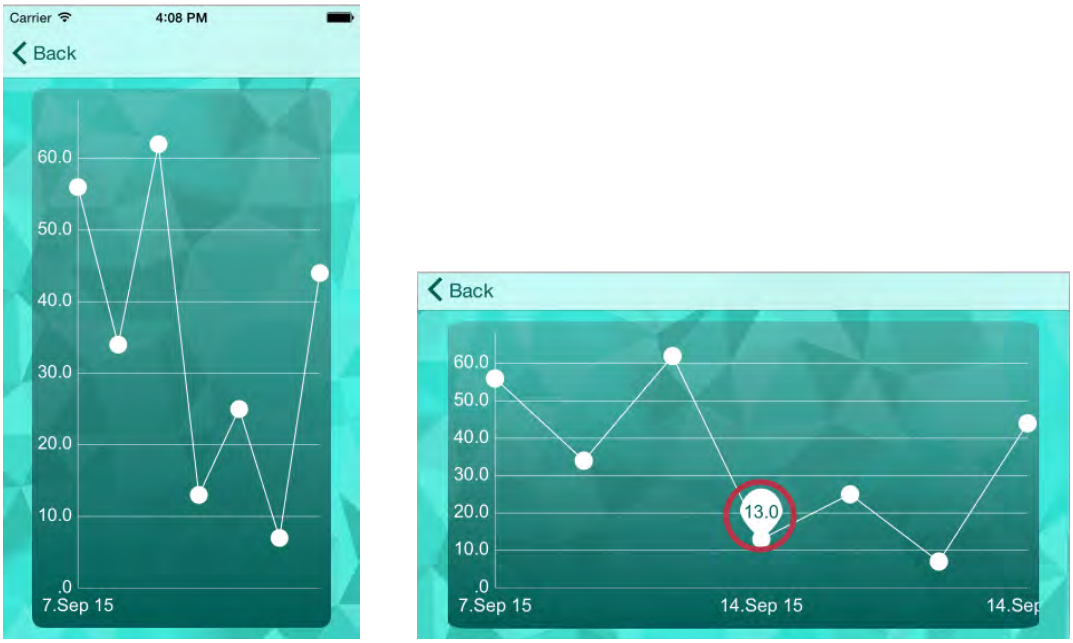

Abbildung 4.12: Statistik dargestellt in einem Diagramm

Das Diagramm ist ein Liniendiagramm. Auf der x-Achse befindet sich das Datum der Eingabe. Auf der y-Achse werden die Eingaben eingetragen. Die Werte selbst können durch Interaktion exakt angezeigt werden (vgl. runder roter Marker in Abb. 4.12). Es gibt auch die Möglichkeit im Diagramm zu zoomen. Indem man zwei Finger auseinander oder zusammen bewegt, ändern sich die Achsenabstände. So wird die Ansicht vor allem bei vielen Einträgen übersichtlicher, aber nicht alle Eingaben sind sichtbar. Wenn man die zwei Diagramme in Abb. 4.12 miteinander vergleicht, fällt auf, dass der y-Achsenabstand

beim rechten Bild größer ist. Sobald das Smartphone sich im Querformat befindet, können mehrere Einträge ohne Zoom angezeigt werden. Wenn mehr als zehn Werte im Diagramm angezeigt sind, wird automatisch gezoomt. So bleibt das Diagramm überschaubar, wenn die Einträge in einem Abstand zueinander stehen und sich nicht überlagern. Die Achsenabstände können nach Belieben wieder verkleinert werden. Auf einem Blick können mehr Werte dargestellt werden und miteinander verglichen werden.

4.2.5 Bonus

Abbildung 4.13: Medaillen

Damit das Crowd Sensing möglichst erfolgreich ist, muss der Benutzer regelmäßig seine Daten in die Anwendung eintragen. Optimal wäre ein tägliches Ausfüllen eines Formulars. Um eine höhere Benutzerakzeptanz zu erreichen, verwendet die Anwendung das Prinzip der $Gamification$. Definiert wird $Gamification$, indem in der Anwendung Spielelemente in Nicht-Spiel Kontexten verwendet werden. Dabei können Spielelemente solche Elemente sein, die in einem Großteil aller Spiele vorhanden sind oder mit Spielen assoziiert werden [22].
Ein Beispiel ist das Einführen von verschiedenen $Status$. Diese können durch eine Vergabe von Titel unterschieden werden. Jeder Status kann dabei ein höheres Level repräsentieren. Diese Art von Spielelement fördert das Wettbewerbsverhalten. Bietet die Anwendung zusätzlich noch eine Rangliste, können sich Benutzer unter einander vergleichen [23].
Eine andere Möglichkeit ist der Einsatz von Belohnungen oder Erfahrungspunkten. Das

Handeln des Benutzers hat direkten Einfluss auf mögliche Preise oder Auszeichnungen. Das Ziel von $Gamification$ ist die Verbesserung der Benutzererfahrung [22]. Dabei soll der Benutzer motiviert werden, häufiger mit der Anwendung zu arbeiten [23]. Der Spaßfaktor spielt hier eine wesentliche Rolle [22].

In der im Rahmen dieser Arbeit entwickelten mobilen Anwendung soll ein Bonusprogramm den Spaß fördern. Beim Bonusprogramm kann der Benutzer drei verschiedene Arten von Medaillen erwerben (siehe Abb. 4.13). Die Art von Belohnung soll ein Ansporn für den Benutzer sein. Das Einsammeln von Medaillen bewirkt ein häufiges Ausfüllen der Formulare. Somit kann eine große Menge von Daten erfasst werden. Für den Erwerb einer Medaille werden die Tage gezählt, an dem mindestens ein Formular richtig ausgefüllt und abgeschickt wurde. Es zählen allerdings nur die direkt aufeinander folgende Tage, d.h. es darf kein Tag zwischen dem letzten Eintrag und dem heutigen liegen, sonst wird wieder bei Null angefangen zu zählen. Sofern der Benutzer an sieben aufeinander folgenden Tagen ein Formular ausgefüllt hat, erhält er die Bronze-Medaille. Für 30 Tage die Silber-Medaille und für 365 Tage die goldene Medaille. Hat ein Benutzer an 37 Tage ein Formular ausgefüllt, erhält er 5 Bronze- und eine Silber-Medaille.

4.3 Aufbau eines Formulars

Die Struktur des Formulars ist vom Server vorgegeben. Beschrieben ist dieses in JSON. Formulare werden in $fields$ und $compositeFields$ strukturiert. Zusätzlich gibt es die Möglichkeit, dem Formular eine Versionsnummer zuzuweisen oder zusätzliche Information in einen Kopfteil ($header$) mitzugeben.

Fields: Prinzipiell besteht ein Formular aus einem oder mehreren Formularfeldern, diese werden als Liste in $fields$ (vgl. Listing 4.1) angehängt. Ein Formularfeld besteht aus Attributen, die in der Tabelle 4.1 beschrieben werden.

Attribut	**Beschreibung**
name	Ein Formularfeld besitzt immer einen eindeutigen Namen, der zur Identifizierung dient.

label	Das Label beschriftet ein Formularfeld und ist für den Benutzer sichtbar.
type	Ein Formularfeld kann verschiedene Funktionen annehmen, die verpflichtend durch einen Typ definiert werden. Die möglichen Werte eines Typs werden im Folgenden aufgelistet.
	$text$, $password$, $email$, url, $number$, tel (Telefonnummer) werden als ein einzeiliges Textfeld dargestellt. Abhängig vom genauen Wert kann das eine Auswirkung auf die Eingabemöglichkeit und Validierung haben (siehe dazu Kapitel 4.4).
	$textarea$ ist ein mehrzeiliges Textfeld, das in iOS als $Textview$ bezeichnet wird.
	$time$ oder $date$ machen die Eingabe für Uhrzeit und Datum möglich.
	$compositeField$ bildet eine Einheit aus mehreren Formularfeldern, namens $CompositeField$.
	$select$ und $radio$ repräsentieren eine einfache Selektion, d.h. es muss pro Liste genau ein Element ausgewählt werden. $radio$ wird in Form eines An/Ausschalt-Bedienfeld dargestellt, während $select$ eine Liste ist.
	$multiselect$ und $checkbox$ erlauben eine mehrfache Auswahl. Der Benutzer kann kein, ein Element oder mehrere Elemente auswählen.
pattern	Das Pattern-Attribut beinhaltet einen regulären Ausdruck, nach $JavaScriptPatternSyntax[ECMA262]$. Dadurch wird die Eingabe eines Textfeldes auf ihre Richtigkeit hin überprüft. Beispielsweise sieht ein regulärer Ausdruck für $number$ folgendermaßen aus: $\wedge[-+]?[0-9]*\backslash.9?[0-9]+([eE][-+]?[0-9]+)?$
required	Das Attribut gibt durch $true$ oder $false$ an, ob das Formular ausgefüllt werden muss oder ob es ein optionales Feld ist. Sofern das Attribut nicht vorhanden ist, wird der Wert $true$ angenommen.

compositeField	Beinhalten einen Verweis auf das $CompositeField$, welches in der JSON-Datei in einem extra Abschnitt geschrieben ist.

Tabelle 4.1: Fields

```
1  "fields":
2  [ { "name":"mySimpleTextfield",
3      "label": "einfaches Textfeld",
4      "type":"text" },
5
6    { "name":"myCompositeField",
7      "type":"compositeField",
8      "compositeField":"adress",
9      "required":true }
10 ],
```

Listing 4.1: $fields$-Abschnitt

CompositeFields: In der Tabelle 4.2 wird eine Einheit aus mehreren Formularfeldern beschrieben, die ebenfalls Attribute besitzt.

Attribut	**Beschreibung**
name	Der Name (Listing 4.2, Zeile 2) referenziert das $CompositeField$, das in $fields$ erzeugt worden ist. Für eine Zuordnung muss der Name mit dem Wert des $compositeField$ (Listing 4.1, Zeile 8) übereinstimmen.
label	Das Label ist optional die Überschrift der Einheit von Formularfeldern.
fields	Beinhaltet die Formularfelder der Einheit, die wie einfache Formularfelder (siehe Abschnitt 4.3) aufgebaut sind.

Tabelle 4.2: CompositeField

```
"compositeFields":
  [ {"name": "adress",
     "fields":[{
          "name": "street",
          "label": "Straße",
          "type": "text",
          "required":true },

        {"name": "number",
          "label": "Hausnummer",
          "type": "number",
          "required":false }]
  ] }
```

Listing 4.2: $CompositeField$-Abschnitt

4.4 Dynamische Generierung eines Formulars

Das Anzeigen eines Formulars ist ein wichtiger Teil dieser Arbeit. Die einzelnen Elemente eines Formulars in iOS wurden im Kapitel 4.2.3 bereits vorgestellt. In diesem Abschnitt wird darauf eingegangen, wie eine Formularseite dynamisch generiert wird. Der Inhalt eines Formulars ist zur Compile-Zeit nicht bekannt, deshalb muss der Aufbau zu jedem Formular passen. Die Größe des Bildschirms auf einem iPhone, vor allem bei den älteren Versionen, ist begrenzt. So werden alle Elemente des Formulars untereinander auf einer $UIScrollView$ angezeigt. Die UIScrollView dient dazu, viele Elemente auf einer kleinen Ansicht durch Scrollen sichtbar zu machen.

Damit sich Elemente nicht überlappen, bietet sich das $Auto\ Layout$ an. Dadurch kann jedes Element auf dem Bildschirm passend positioniert werden, unabhängig von den unterschiedlichen iPhone Größen. So werden auch bei kleinen Bildschirmen alle Elemente angezeigt und sind nicht außerhalb des sichtbaren Bereichs. Für das $AutoLayout$ werden $Constraints$ (Einschränkungen) definiert, die das Verhalten unter den einzelnen Elementen bestimmen und Abhängigkeiten festlegen. Dafür werden in einem $Constraint$ zwei Elemente benötigt, die in einem Verhältnis zu einander stehen. Die Elemente haben Attribute, die die Beziehung zu den Rändern, die Höhe oder die Breite beschreiben.

Dazu ist wichtig in welcher Relation (z.B gleich oder größer gleich) sich die Attribute befinden sollen. Eine Variable und eine Konstante schränken die Regel zusätzlich ein.

Die vertikalen Abstände zwischen den darzustellenden Elemente werden mithilfe des $Auto\ Layouts$ automatisch generiert. Das ist vor allem bei mehrzeiligen Labels sehr nützlich. Durch die automatische Generierung muss der Abstand nicht abgeschätzt werden, weil die Höhe des Labels zur Compile-Zeit unbekannt ist.

Jedes Element hat ein $Constraint$, das für die Zentrierung des Elements auf der x-Achse zuständig ist. So sind alle Elemente unabhängig von der Größe des Gerätes mittig angeordnet.

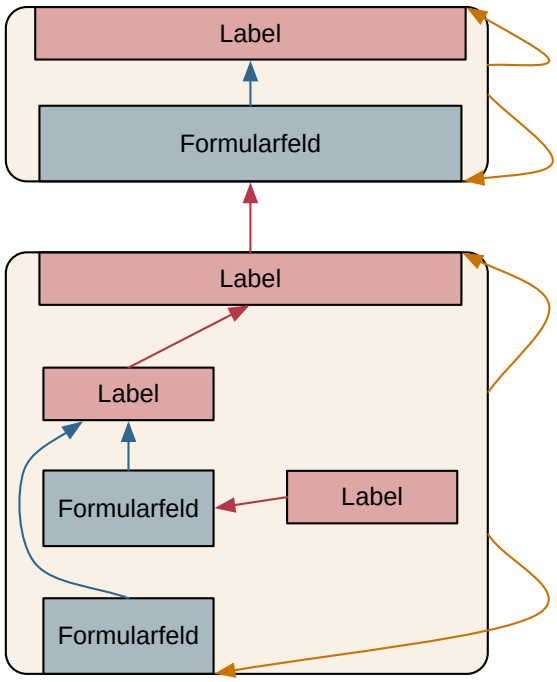

Abbildung 4.14: schematische Darstellung einer View mit ihren Abhängigkeiten

Wie die einzelnen Elemente von einander abhängig sind, zeigt die Abbildung 4.14. Schematisch werden Labels und Formularfelder auf dem Rechteck, das eine Einheit verdeutlicht, gezeigt. Prinzipiell orientiert sich ein Element am nächst oberen Element, da es zeitlich früher erzeugt worden ist und deshalb bekannt ist. In diesem Fall wird

eine Regel festgelegt, die den oberen Rahmen des unteren Element an den unteren Rahmen des oberen Elements legt. Wie groß der Abstand zwischen den Elementen ist, wird durch eine Konstante im $Constraint$ beschrieben.

Wenn Formularfelder direkt untereinander liegen, wie es beispielsweise bei $Checkboxes$ der Fall ist, orientieren sich alle Formularfelder an dem darüber liegenden Label. Diese Formularfelder besitzen zusätzlich noch ein Label, welches rechts von diesen angeordnet ist. Um diese Position festzulegen, werden zwei Regeln benötigt. Für die y-Position orientiert sich der obere Rahmen des Labels am oberen Rahmen des Formularfeldes. Der Abstand zwischen den Elementen ist ein konstanter Wert, der vom linken Rahmen des Labels gemessen wird.

Die Höhe des Rechtecks im Hintergrund (in der Abb. 4.14 gelb dargestellt) wird durch das oberste Element (immer ein Label) und durch das unterste Element bestimmt. Das Rechteck schließt sowohl oben als auch unten mit den Elementen ab.

Für die Breite der Elemente, wichtig vor allem bei Textfeldern und Textviews, werden ebenfalls $Constraints$ verwendet. Wird das Gerät vom Hochformat in den Querformat bewegt, soll zum Beispiel die Breite eines Textfeldes die komplette Breite der Ansicht ausfüllen. Dafür wird eine Regel zwischen dem Textfeld und der $UIScrollView$ aufgestellt. Das Attribut beider Elemente ist die Breite. Das Textfeld soll 90% der Breite der $UIScrollView$ haben. So entsteht ein kleiner Rand an der linken und rechten Seite.

4.5 Validierung der Formulare

Die Daten werden vor dem Abschicken auf ihre Richtigkeit überprüft, um Fehler frühzeitig zu erkennen und das Netzwerk und den Server zu entlasten. Trotz allem validiert der Server die Daten nochmals, damit das Formular serverseitig weiter verarbeitet werden kann.

Jede Art von Formularfeld muss individuell behandelt werden. Haben die Textfelder vom Typ $number$, $email$, tel kein Pattern vorgegeben, wird ihnen ein spezifisches zugewiesen. So wird sichergestellt, dass die Eingabe korrekt ist. Für den Typ $number$ kann man

zusätzlich die Minimal - und Maximalwerte definieren, die ebenfalls überprüft werden. Alle anderen Typen von Textfeldern werden definierte Pattern mitgegeben, die die Richtigkeit überprüfen. So kann man zum Beispiel bestimmte Buchstaben ausschließen.

Für die Formularfelder $Date$, $Time$, $Radiobuttons$ und einfache Listenauswahl gibt es keine Validierung. Bei diesen Elementen ist immer ein Wert in einem bestimmten Schema ausgewählt. Somit ist die Auswahl immer valide und muss nicht extra überprüft werden.

Bei der Validierung eines Formulars wird zusätzlich geprüft, ob alle Pflichtfelder ausgefüllt sind. Ist dies nicht der Fall wird dem Benutzer eine Fehlermeldung angezeigt. Pflichtfelder werden bei $compositeFields$ besonders behandelt. Hierbei müssen zwei Fälle betrachtet werden.

1. $required = true$ im $compositeField$: Innerhalb des $compositeField$ wird jedes einzelne Feld für sich behandelt.
2. $required = false$ im $compositeField$: Sobald ein Feld ausgefüllt ist, müssen alle Felder ausgefüllt sein. Dabei wird das $required$-Attribut der einzelnen Feldern nicht beachtet.

Der erste Fall ist im Listing 4.1 im Kapitel 4.3 dargestellt. Nur das Feld $street$ (Listing 4.2, Zeile 7) innerhalb des $compositeField$ muss ausgefüllt werden. Das Feld $number$ (Zeile 12) bleibt optional.

Für den zweiten Fall wird für das obige Beispiel (Listing 4.1, Zeile 9) $required = false$ angenommen. Wegen dem Pflichtfeld $street$ muss auch $number$ ausgefüllt werden. Dabei ist nicht relevant, dass das Feld $number$ optional ist.

Werden Fehler beim Validieren des Formulars entdeckt, müssen diese an den Benutzer weitergegeben werden. So können die Fehler korrigiert werden und das Formular kann abgeschickt werden. Es gibt zwei Arten von Fehlermeldungen. Wenn der Benutzer ein Pflichtfeld vergessen hat auszufüllen, wird das textuell beschrieben und die entsprechenden Felder orange markiert. Ebenso orange markiert werden die Felder innerhalb des $compositeFields$, die noch ausgefüllt werden müssen, sofern das $compositeField$ das erfordert.

Wird ein Feld ausgefüllt, aber der Inhalt stimmt nicht mit der Vorgabe überein, wird der fehlerhafte Text pink markiert. Ebenfalls wird das auch textuell erklärt.

Auch der Server validiert das Formular noch einmal. Ist dort noch ein Fehler aufgedeckt worden, muss dieser ebenso vom Benutzer korrigiert werden. Damit das möglich ist, schickt der Server direkt eine Fehlermeldung im JSON-Format zurück.

In diesem Beispiel hat der Server drei Fehler gefunden (siehe Listing 4.3). Die Namen der Formularfeldern werden unter $inputValidation$ aufgelistet. Bei CompositeFields wird sowohl der Name des CompositeFields (Zeile 6) selbst und die Namen der betroffenen Felder (Zeile 7) innerhalb aufgelistet.

```
1  "errors":
2     {
3       "inputValidation":[
4         { "name":"fancyInput" },
5
6         { "name":"fancyBanking",
7           "fields": [
8             {"name":"iban"},
9             {"name":"swift"}]
10        }]
11    }
```

Listing 4.3: serverseitige Fehlermeldung

4.6 Abläufe

Die wichtigsten Abläufe zwischen Benutzer und System werden nachfolgend vorgestellt. Der Benutzer stellt grundsätzlich eine Anfrage an das System. Dieses leitet die Anfrage gegebenenfalls weiter an den Server. Der Server liefert das gewünschte Ergebnis zurück an das System, welches das Ergebnis dem Benutzer darstellt.

Anfordern der Liste aller vorhandenen Formulare

Der Benutzer hat die Möglichkeit sich alle Formulare, die der Server zur Verfügung stellt, anzeigen zu lassen (siehe Abb. 4.15). Dafür stellt er an das System den Auftrag

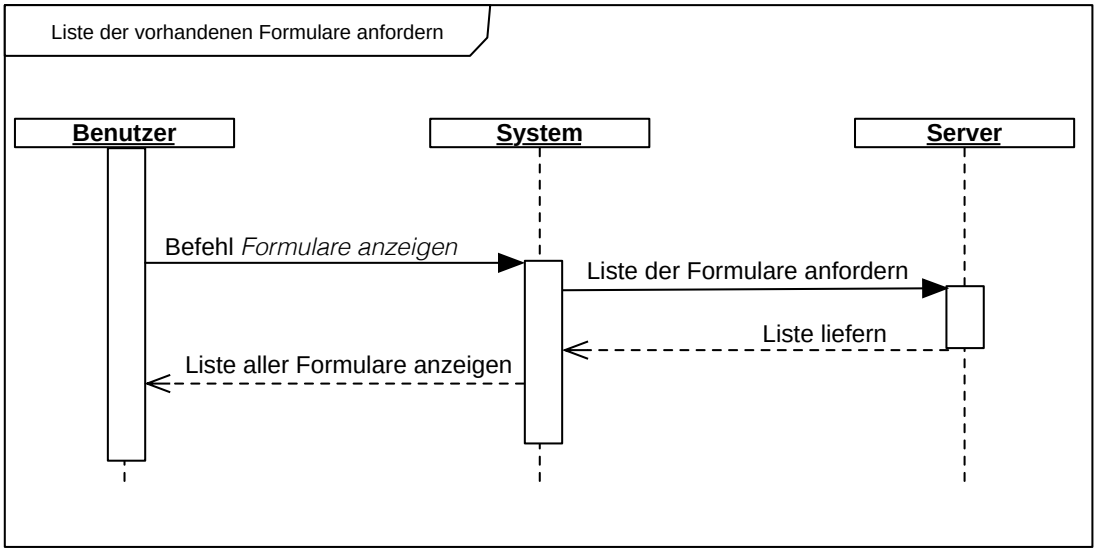

Abbildung 4.15: Liste der vorhandenen Formulare anfordern

$Formulare\ anzeigen$. Das System stellt eine Verbindung zum Server her und fordert die Formulare vom Server an. Daraufhin schickt der Server die Formulare an das System. Das System zeigt dem Nutzer alle Formulare an.

Technisch wird das mittels einem HTTP-GET-Request (Listing 4.4) umgesetzt.

```
1
2 GET /api/modules/Masterarbeit/FancyModule/start HTTP/1.1
3 Host:            crowdsensr.ma.arnim-schindler.de
4 Authorization:   Basic c2FicmluYS
```

Listing 4.4: HTTP-GET-Request

Die Datei, die vom Server zurückgeschickt wird, ist in Listing 4.5 dargestellt.

Ein Objekt $next$ signalisiert, dass es sich um kein Formularfeld handelt. Die Objekte innerhalb von $next$ zeigen die Formulare an, die verfügbar sind und mittels welchen

```
1 { "next": [
2         { "command": "pizza",
3           "method": "GET",
4           "info": "Pizza auswählen" },
5         { "command": "drink",
6           "method": "GET",
7           "info": "Getränk auswählen" }]
8 }
```

Listing 4.5: Liste aller möglichen Formularen

Parametern diese vom Server abgefragt werden (vgl. 4.5, Zeile 2 und 3). Der $info$-Wert dient zur Anzeige. Hier gibt es zwei Formulare auszuwählen. Der Benutzer kann eine Pizza oder ein Getränk auswählen. Beide Formulare können mittels eines HTTP-GET-Request und dem Befehl $pizza$ oder $drink$ vom Server angefragt werden. Der Benutzer liest in der Liste den $info$-Wert: $Pizza\ auswählen$ oder $Getränk\ auswählen$.

Anzeige eines bestimmen Formulars

Dem Benutzer wird eine Liste von Formularen angezeigt, aus der er ein bestimmtes Formular auswählen kann.

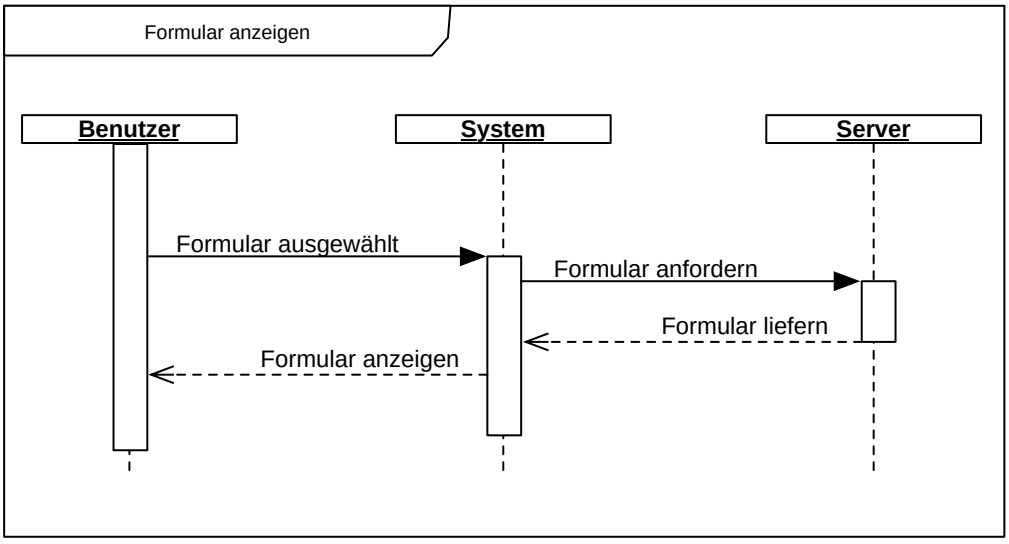

Abbildung 4.16: Formular anzeigen

Das System fordert das Formular an. Der Server schickt es an das System, welches dann das ausgewählte Formular anzeigt (siehe Abb. 4.16).

In dem Code-Beispiel 4.6 ist ein mögliches Formular dargestellt. $Form$ (Zeile 1) leitet

```
1  { "form": {
2          "version": "0.0.2",
3          "fields": [
4                              { "name":"name",
5                                "label":"Name",
6                                "type":"text",
7                                "required":true},
8                              { "name": "drink",
9                                "label": "Getränk",
10                               "type": "multiselect",
11                               "options": [
12                                     { "value": "coke",
13                                       "label": "Cola" },
14                                     { "value": "fanta",
15                                       "label": "Fanta" }]
16 }]}}
```

Listing 4.6: Struktur eines Formulars

ein, dass das Folgende ein Formular darstellt. Jedes Formular hat eine Versionsnummer (Zeile 2) und eine Liste von Formularfeldern. Dieses Beispiel enthält nur zwei Formularfelder.

Das erste zeigt ein einfaches Textfeld (Zeile 4) mit einem eindeutigen Namen. Das $label$ (Zeile 5) ist die Beschreibung des Feldes und dient als Hilfe für den Benutzer. $Required$ gibt an, dass das Feld ausgefüllt sein muss (Zeile 7).

Die Struktur eines Feldes ist immer gleich, lediglich der Typ und damit weitere abhängige Paramter ändern sich. Das Formularfeld $drink$ ist eine einfache Auswahl (Zeile 10) aus einer Liste, die zwei Einträge, $Cola$ und $Fanta$, besitzt. Die Einträge der Liste werden in $options$ (Zeile 11) geschrieben. Jeder Eintrag besitzt ein Label und ein $value$ (Zeile 12 und 13), der für eine eindeutige Zuweisung serverseitig bestimmt ist.

Abschicken eines ausgefüllten Formulars

Wenn der Benutzer ein Formular ausgefüllt hat, sendet er an das System den Befehl $Formular\ abschicken$. Danach überprüft das System die Eingabe des Formulars auf ihre Richtigkeit (siehe Abb. 4.17).

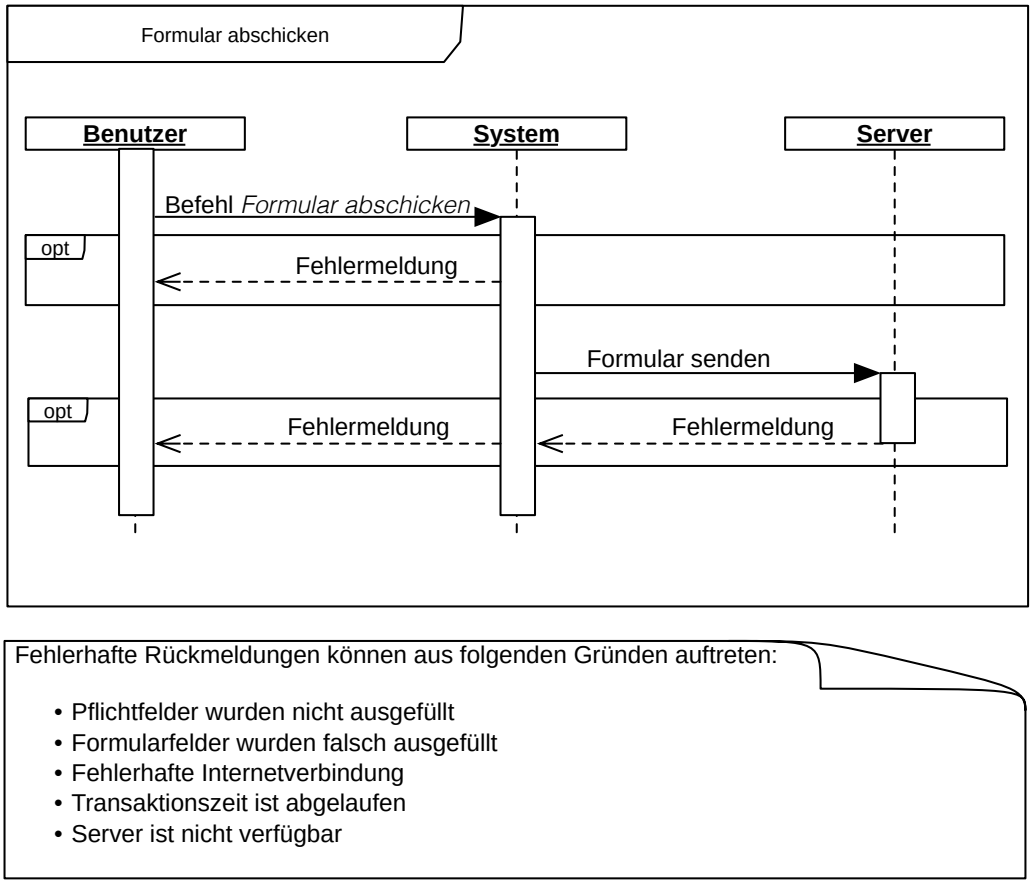

Abbildung 4.17: Formular abschicken

Falls das Formular fehlerhaft ausgefüllt wurde, bekommt der Benutzer eine entsprechende Fehlermeldung. Ist das Formular korrekt ausgefüllt, werden die Werte des Formulars an den Server gesendet. Der Benutzer bekommt die Rückmeldung, falls das Formular nicht korrekt an den Server übermittelt werden konnte. Ein Grund kann eine mangelnde Verbindung zum Server sein oder ein Fehler im Formular, der erst serverseitig entdeckt worden ist. So muss der Benutzer das Formular korrigieren und einen erneuten Versuch

starten, das Formular abzuschicken. Die eingegebenen Werte des Formulars werden ebenfalls via JSON an den Server übermittelt.
Grundsätzlich wird ein einfaches Feld durch seinen Namen eindeutig identifiziert, somit kann ihm dann der Wert zugewiesen werden. Mehrfache Auswahllisten geben eine Liste mit den ausgewählten Optionen-Elementen zurück, dabei kann die Liste leer sein, ein Element oder mehrere Elementen beinhalten. CompositeFields geben eine Liste von Elementen zurück, die wie ein einfaches Feld aufgebaut sind.

Das Beispiel-Formular aus dem vorherigen Listing 4.6 wurde ausgefüllt und an den Server geschickt, der die folgende JSON Datei erhalten hat. Im Gegensatz zu $form$

```
1  {"formreply":{
2          { "version":"0.0.2",
3            "fields": [
4                  { "name": "name",
5                    "value": "Sabrina" },
6                  { "name": "drink",
7                    "value": [ "coke","fanta" ]},
8                  { "name": "adress",
9                    "value": [
10                    { "name": "street",
11                      "value": "Blumenweg" },
12                    { "name": "streetNumber",
13                      "value": "4" }]
14             }] } }
```

Listing 4.7: Struktur eines ausgefüllten Formulars

kennzeichnet $formreply$ (Listing 4.7) ein ausgefülltes Formular, das an den Server zurückgeschickt wird.
Parameter, die dem Benutzer nur als Anzeige dienen, wie zum Beispiel das $label$, fallen beim Zurückschicken der Daten weg. Das erste Objekt ist ein einfaches Textfeld, deshalb wird auch nur der Name und der Wert benötigt (Zeile 4 und 5). Beim zweiten Objekt sind die zwei Listeneinträge $Cola$ und $Fanta$ ausgewählt. Diese werden als eine Liste mit den Optionen-Werten (Zeile 7) geschrieben. Das dritte JSON-Objekt ist ein CompositeField namens $adress$ (Zeile 8 bis 13). Dieses besteht aus zwei einfachen Textfeldern, die wie das Feld $name$ aufgebaut sind.

Anzeige der Statistik

Der Benutzer hat die Möglichkeit, gespeicherte Daten grafisch darstellen zu lassen. Die Daten werden den ausgefüllten Formularen entnommen, die einen numerischen Wert haben.

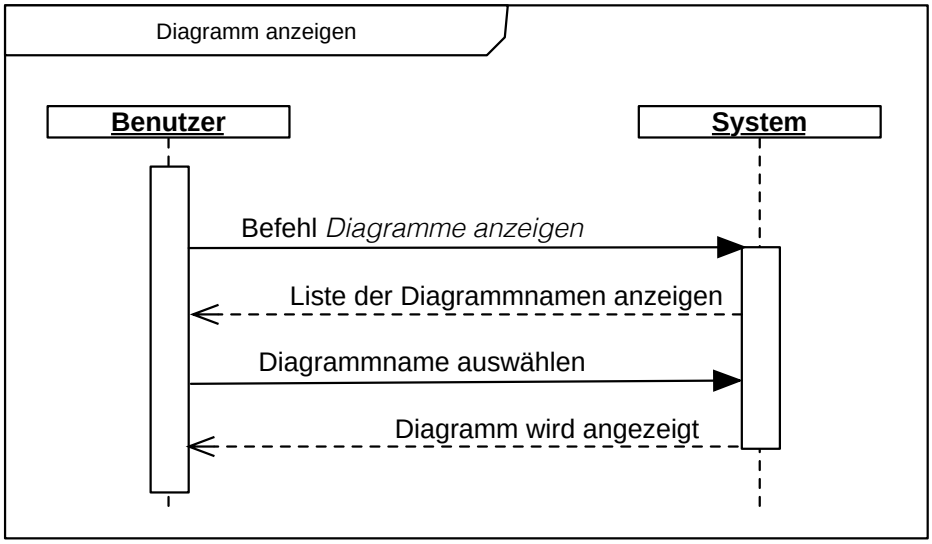

Abbildung 4.18: Statistik anzeigen

Nachdem der Benutzer den Auftrag $Diagramme\ anzeigen$ an das System gestellt hat, kann der Benutzer sein gewünschtes Formularfeld aus einer Liste auswählen (siehe Abb. 4.18). Die Werte des Feldes sind darauf in einem Diagramm wiederzufinden.

4.7 Systemmodell

Die Grafik (siehe Abb. 4.19) zeigt eine Übersicht der Datenobjekte. Jeder Dialog aus der Dialogstruktur im Kapitel 4 besitzt einen eigenen ViewController, d. h. es existieren die **ViewController**: $StartViewController$, $FormViewController$, $ChartViewController$, $FormTableViewController$, $ChartTableViewController$ und $BonusViewController$. In der Abbildung 4.19 werden die sechs ViewController zusammengefasst. Die ViewController bilden die Schnittstelle zwischen dem Benutzer und dem System.

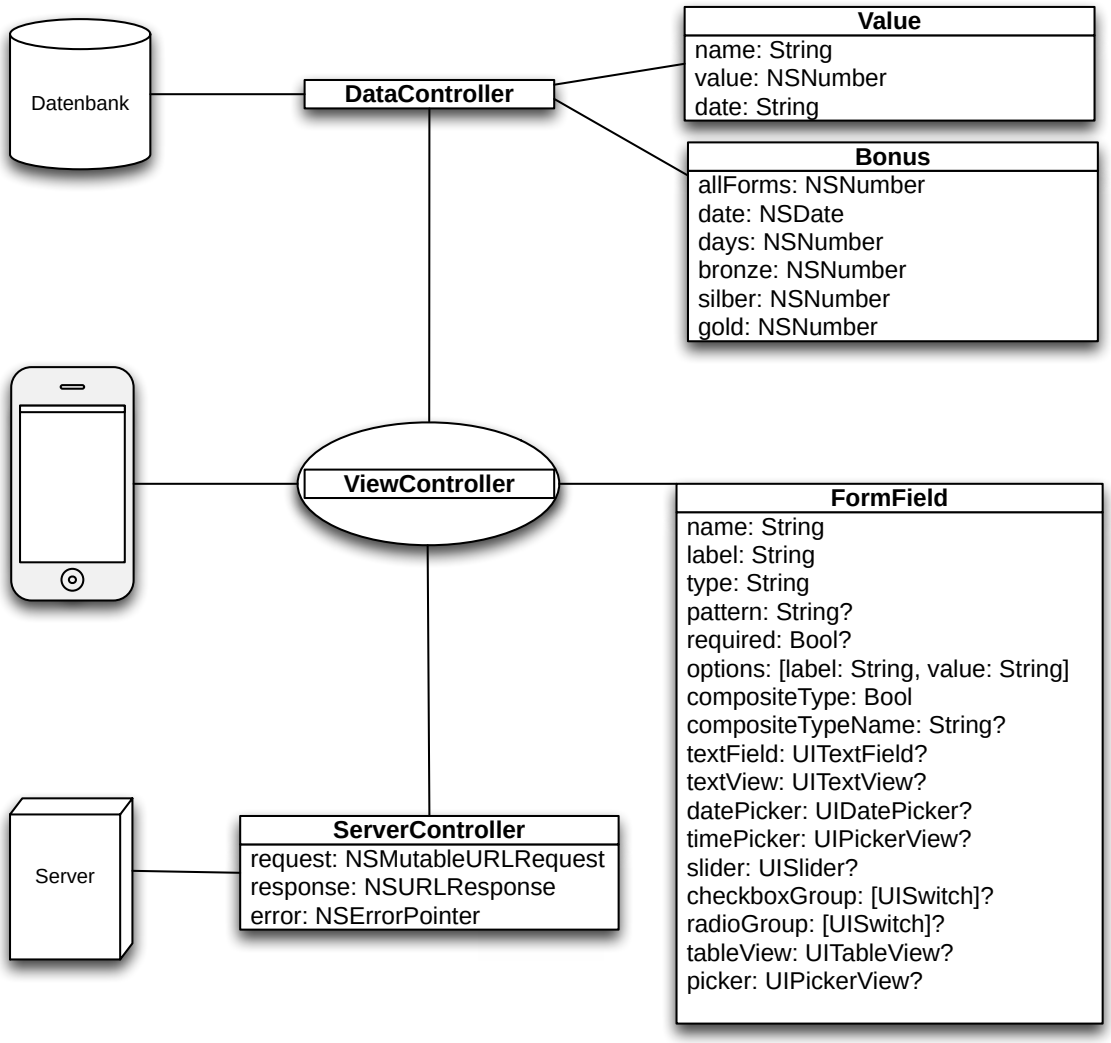

Abbildung 4.19: Systemmodell

Der $FormViewController$, der für die Ansicht eines Formulars zuständig ist, generiert für jedes einzelne Formularfeld ein Objekt der Klasse **FormField**. Ein Formularfeld besitzt immer die Attribute $name,\ label,\ type$, da sie Pflichtangaben sind. Die restlichen Attribute sind optional und sind mit einem Fragezeichen gekennzeichnet. Sie können den Wert $null$ annehmen. Abhängig von dem Typ des Formularfeldes wird genau ein UI-Element gesetzt. Beispielsweise wird für den Typ $text$ ein $UITextfield$ erzeugt. Für Gruppierungen von Elementen, z.B. bei einer mehrfachen Listenauswahl oder bei Radiobuttons, speichert $options$ diese Elemente zusätzlich in einer Liste. Zusätzliche $composite$-

Attribute eines $FormFields$ sind sowohl für die Anzeige im $FormViewController$ als auch bei der Validierung wichtig.

Der $FormViewController$ besitzt ebenfalls ein Objekt der Klassen **DataController** und **ServerController**. Auch der $FormTableViewController$ benötigt für die Kommunikation mit dem Server ein Objekt vom $ServerController$. Daten werden vom Server verlangt, eventuell verarbeitet und an $FormViewController$ und $FormTableViewController$ weitergeleitet.

$FormViewController$, $ChartTableViewController$, $ChartViewController$ und $BonusViewController$ haben eine Instanz der Klasse $DataController$. Über diese Klasse ist es möglich Daten lokal auf dem iPhone zuspeichern. Zum einen verwaltet $DataController$ die Daten, die für die Statistik benötigt werden und zum anderen die Daten für das Bonusprogramm.

Die Daten für die Statistik werden in der Klasse **Value** mittels $name, value$ und $date$ beschrieben. Der $DataController$ kann daraus eine Liste aller Namen erstellen, die der $ChartTableViewController$ dem Benutzer anzeigt. Die Werte mit dem dazugehörigen Datum werden für die grafische Darstellung im $ChartViewController$ benötigt.

Die Klasse **Bonus** besteht aus $allForms$, $date$, $days$ und je eine Variable für die bereits gesammelten Medaillen. $AllForms$ gibt die Anzahl der Formulare an, die richtig abgeschickt wurden. $Date$ gibt das Datum des zuletzt gesendeten Formulars an. $Days$ zählt die aufeinander folgende Tage, an dem ein Formular abgeschickt wurde. Der $DataController$ ist für die Aktualisierung der Tabelle zuständig (siehe dazu **updateBonus()** im nachfolgendem Kapitel 4.8). So kann der $BonusViewController$ die Anzahl der Medaillen anfragen, um die richtige Anzahl der verschiedenen Medaillenarten dem Benutzer anzuzeigen.

4.8 Funktionen

Hier werden die wichtigsten Funktionen vorgestellt, die bei der Umsetzung der Anwendung benötigt werden. Die Abbildung 4.20 beschreibt den Ablauf $Formular\ ausf\ddot{u}llen$. Die Methoden, die während dem Ablauf aufgerufen werden, sind rot markiert.

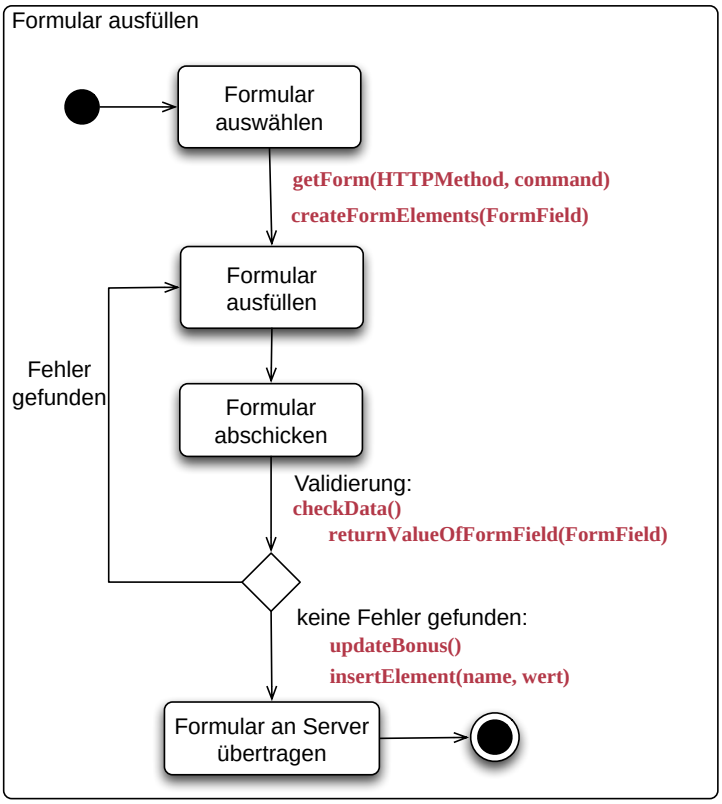

Abbildung 4.20: Aktivitätsdiagramm

Zu Anfang muss ein Formular aus einer Liste ausgewählt werden. Dann kann mit der Methode **getForm(HTTPMethod, command)** eine JSON Datei vom Server geladen werden. Damit das ausgewählte Formular geladen werden kann, muss die Methode die HTTP-Methode GET oder POST und einen zusätzlichen Befehl, welcher ein Formular beschreibt, kennen. Die Methode erzeugt dabei eine eigene JSON-Datei, aus der die einzelnen Elemente herausgelesen werden.

Anhand der einzelnen Elementen wird jeweils ein Objekt der FormField Klasse erzeugt. Mit den gesetzten Variablen in dieser Klasse stellt die Methode **createFormElements(FormField)** ein Formularfeld (siehe Kapitel 4.3) auf dem Bildschirm dar.

Der Benutzer kann nun das Formular abschicken. Danach wird das ausgefüllte Formular mit der Methode **checkData()** validiert (siehe Kapitel 4.5). Zuerst müssen die Werte, die der Benutzer eingetragen hat, mittels der Methode **returnValueOfForm-**

Field(FormField) aus den Formularfeldern gelesen werden. Da es verschiedene Typen von Formularfeldern gibt, muss die Methode jede Art einzeln behandeln. Davon ist auch abhängig, ob die Methode eine einelementige Liste von $strings$ pro Formularfeld zurück gibt, oder eine Liste mit mehreren Werten. Das ist dann der Fall, wenn beispielsweise bei $Checkboxes$ mehrere Felder ausgewählt werden.

checkData() entnimmt den Rückgabewert von $returnValueOfFormField(FormField)$ und überprüft diesen mit den Vorgaben des Formularfeldes (siehe Kapitel 4.5). Treten Fehler dabei auf, wird die Art des Fehlers zwischen gespeichert und das fehlerhafte Feld markiert. Der Benutzer muss das Formular erneut ausfüllen.

Werden keine Fehler gefunden, wird die Tabelle $Bonus$ in der Methode **updateBonus()** aktualisiert. Die Gesamtanzahl der ausgefüllten Formulare ($allForms$) wird um Eins erhöht. Wurde der letzte Eintrag am vorherigen Tag getätigt, wird das Datum und die Anzahl der fortlaufenden Tage ($days$) aktualisiert. Abhängig vom Wert $days$ werden die Variablen $bronze,\ silber\ oder\ gold$ ebenfalls aktualisiert.

Die Methode **insertElement(name, wert)** wird für numerische Formularfelder aufgerufen. Sie ist für das Speichern von Werten für die Statistik verantwortlich. Zu dem übergebenden Namen wird der Wert und das aktuelle Datum in die Tabelle $Values$ der Datenbank gespeichert.
Das Datum hat folgende Notation $d.mmm\ yy$, wobei

- d: eine Ziffer für den Tag,
- mmm: drei Buchstaben für den Monat,
- yy: zwei Ziffern für das Jahr sind.

Beispiele sind: $7.Sep\ 15$ oder $14.Sep\ 15$ (siehe Abb. 4.12 im Kapitel 4.2.4).

Am Ende der Methode **checkData()** kann das ausgefüllte Formular an den Server übertragen werden.

4.9 Verwendete Frameworks

Um die Realisierung der Anwendung zu erleichtern, werden zwei Frameworks verwendet, die öffentlich zur Verfügung stehen.
$SwiftyJSON$ unterstützt den Umgang mit JSON-Dateien.
$iOS-charts$ ermöglicht eine einheitliche dynamische Darstellung von Diagrammen innerhalb der mobilen Anwendung.

4.9.1 SwiftyJSON

Das Open Source Framework $SwiftyJSON$ erleichtert das Parsen von JSON Dateien, welches ein wichtiger Teil dieser Anwendung ist [24]. Betrachtet wird das folgende JSON Beispiel in Listing 4.8. Das besteht aus einer Liste $fields$. In dieser befindet sich ein Objekt mit Namen und Typ.

```
{ "fields": [
        { "name": "feldOne",
          "typ": "text" }]
}
```

Listing 4.8: beispielhafte JSON-Datei

Ohne ein Framework zu verwenden würden folgende Codezeilen (siehe Listing 4.9) den Typ parsen. Obwohl Swift implizite Typisierung unterstützt, der Datentyp also nicht

```
if let item = json as? NSDictionary{
    if let fields = item["fields"] as? NSArray{
        if let firstElement = fields[0] as? NSDictionary{
            if let typ = firstElement["typ"] as? NSString{
                println(typ) // = text
}}}}
```

Listing 4.9: Code-Beispiel ohne Framework

explizit angegeben werden muss, wird hier eine Typumwandlung mittels as verlangt. In diesem Beispiel muss zunächst das Array $fields$ gecastet werden. Dann wird definiert,

welches Objekt in der Liste gewünscht ist (hier das erste Element). Erst dann kann der Name des Labels gelesen werden.

Mit dem Framework $SwiftyJSON$ lässt sich der Code mit dem gleichen JSON Objekt und gleichem Ziel auf eine Zeile kürzen (siehe dazu Listing 4.10). Hier fallen vor allem

```
1 if let typ = json["fields"][0]["typ"].string {
2       println(typ) // = text
3     }
```

Listing 4.10: Code-Beispiel mit Framework

die Typumwandlungen weg, so ist das Gesuchte in einer Zeile übersichtlich dargestellt. Die einzige benötigte Typbezeichnung ist $.string$.

4.9.2 ios-charts

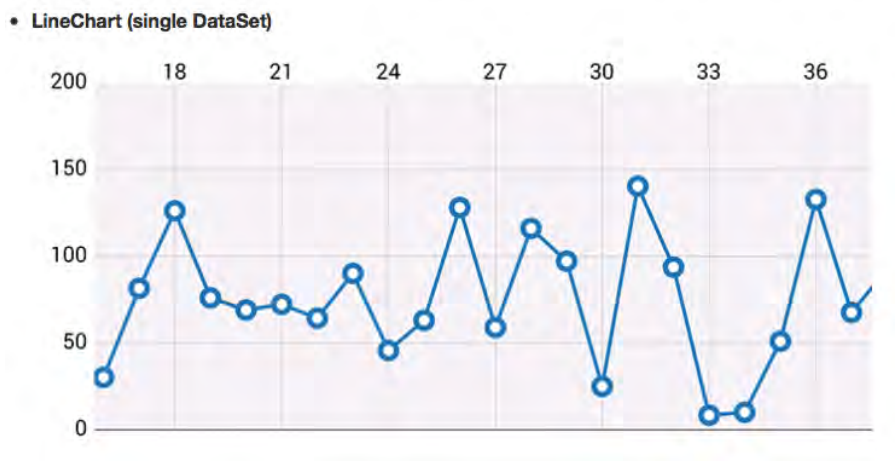

Abbildung 4.21: Ansicht des Diagramms mit Default-Werten [25]

Das zweite Framework, das in der Anwendung verwendet wird, nennt sich $ios - charts$. Das Framework von $DanielC.Gindi$ kann dynamische Diagramme erleichtert darstellen [25]. Dynamisch heißt, dass der Benutzer durch Touch-Gesten die Achsen des Diagramms skalieren kann. Insgesamt bietet das Framework acht verschiedene Arten von Diagrammen an, die man selbst gestalten kann [25]. Für diese Anwendung wird

das $LineChart$ verwendet. In der Abbildung 4.21 ist eine mögliche Darstellung eines $LineChart$ mit diesem Framework zu sehen.

Das Aussehen des Diagramms für die in diesem Rahmen entwickelten Anwendung orientiert sich an den Diagrammen, die in der Apple eigenen $Health$-Anwendung verwendet werden. Dafür müssen die Farben der Linien und der Beschriftung eingestellt und einen Verlauf in den Hintergrund gesetzt werden. Im Gegensatz zu dem Diagramm in der Abb. 4.21, ist die Beschriftung der x-Achse in der Anwendung unten angeordnet. Die Gitterlinien im Hintergrund werden in der Anwendung nicht benötigt. Um dennoch die Werte der Spitzen lesen zu können, wird der exakte Spitzenwert interaktiv angezeigt (vgl. Kapitel 4.2.4).)

5

Anforderungsabgleich

In diesem Abschnitt wird überprüft, ob die Anforderungen in Kapitel 3 erfüllt worden sind. Durch die Überprüfung können Fehler entdeckt und vermieden werden [26]. Sowohl bei den funktionalen als auch bei den nichtfunktionalen Anforderungen ist in Klammern ein Erfüllungsgrad angegeben. Dieser bewertet die Erfüllung einer Anforderung, indem er einen Prozentwert zwischen 0 und 100 angibt.

5.1 Funktionale Anforderungen

1. **Native Anwendung: (100%)** Diese Anwendung wurde ausschließlich für das Betriebssystem iOS entwickelt. Damit ist sichergestellt, dass die Anwendung nativ ist. Es werden auch nur die Oberflächenelemente verwendet, die das Betriebssystem bereitstellt.

2. **Dynamische Formulare: (100%)** Jedes Formular hat eine feste Struktur, die in einer JSON- Datei festgeschrieben ist. Anhand dieser Struktur kann ein Formular dynamisch generiert werden, ohne dass der Inhalt zur Compile-Zeit des Formulars bekannt ist.

3. **Auswahl von Formularen:(100%)** Je nach dem wie viele Formulare der Server zur Verfügung stellt, kann der Benutzer sich ein Formular auswählen und dieses ausfüllen.

4. **Serverkommunikation: (70%)** Eine Kommunikation via HTTP zwischen Client und Server besteht. Durch diese können Formulare vom Server geladen werden und ausgefüllte Formulare an den Server geschickt werden. Jedoch ist dafür eine Internetverbindung notwendig. Besteht keine Verbindung zum Server gehen alle ausgefüllten Formulare verloren.

5. **Validierung von Formularen: (100%)** Damit der Server bei der Validierung der Formulare entlastet ist, werden die Formulare ebenfalls beim Client auf ihre Richtigkeit geprüft. Der Benutzer kann auf Fehler sofort reagieren und sie korrigieren.

6. **Mitteilungen: (70%)** Meldet der Server einen Fehler,x kann die Anwendung auf den Fehler reagieren. Liegt der Fehler an dem Formular, wird die Fehlermeldung an den Benutzer weitergeleitet, damit er seine Eingabe korrigieren kann. Die Fehlermeldung ist allerdings ungenau, da das System nur erkennt, ob ein Feld vergessen oder falsch ausgefüllt wurde.

7. **Bonusprogramm: (100%)** Damit der Benutzer motiviert ist Formulare auszufüllen, bietet die Anwendung ein Bonusprogramm an. Eine Bewertung dieser Anforderung ist in dieser Form nicht möglich, da die Motivation des Benutzers nicht empirisch getestet worden ist.

8. **interaktive Diagramme: (90%)** Gemessene Werte werden in Diagrammen interaktiv dargestellt. Der Benutzer kann innerhalb eines Diagramm navigieren, um eine gute Übersicht seiner Werte zu erhalten. Bei vielen Einträgen kann das Diagramm unübersichtlich werden.

5.2 Nichtfunktionale Anforderungen

1. **Aufgabenangemessenheit: (80%)** Der Benutzer muss manuell seine Daten in die Anwendung eintragen, indem er Formulare ausfüllt. Der Benutzer kann selbst entscheiden, wie effektiv und effizient Daten gesammelt werden sollen.
2. **Verfügbarkeit: (90%)**
   - Die Anwendung unterstützt alle Funktionen, die für die iPhones mit der iOS Version 8.0 oder höher konzeptioniert worden ist.
   - Die Ladezeiten der Formulare sind abhängig von der Internetgeschwindigkeit. Im WLAN kann eine geringe Ladezeit eingehalten werden. So ist die Bedienung im allgemeinen flüssig und ohne lange Wartezeiten.
   - Alle Daten, die die Anwendung speichert, sind konsistent gespeichert und gehen nicht verloren.
3. **Selbstbeschreibungsfähigkeit: (100%)** Die Anwendung ist sehr übersichtlich gestaltet, so kann der Benutzer sich ohne Hilfe im System zurecht finden. Dem Benutzer wird eine Beschreibung des Bonussystem für ein besseres Verständnis angeboten.
4. **Fehlertoleranz: (100%)** Ein wichtiger Punkt ist die Fehlertoleranz. Vor allem beim Ausfüllen von Formularen können viele Fehler entstehen. Um Fehler schon im Vorhinein zu vermeiden, bietet die Anwendung für verschiedene Arten von Textfeldern eine angepasste Tastatur an. Die Anwendung kann auf falsche Eingaben reagieren und führt nicht zu einem Absturz.
5. **Erwartungskonformität: (100%)** Die Anwendung berücksichtigt die $iOS\ Human\ Interface\ Guidelines$, dadurch ist eine gute Benutzbarkeit sicher gestellt. Benutzer erkennen die verwendeten Standardelemente wieder, wodurch die Bedienbarkeit der Anwendung erleichtert wird.

5.3 Zusammenfassung

Abschließend bietet die Tabelle 5.1 eine Zusammenfassung der abgeglichen Anforderungen. Dabei werden sowohl die funktionalen, als auch die nichtfunktionalen Anforderungen mit dem Erfüllungsgrad aufgelistet.

Funktionale Anforderungen		
1	Native Anwendung	100%
2	Dynamische Formulare	100%
3	Auswahl von Formularen	100%
4	Serverkommunikation	70%
5	Validierung von Formularen	100%
6	Mitteilungen	70%
7	Bonusprogramm	100%
8	interaktive Diagramme	90%

Nichtfunktionale Anforderungen		
1	Aufgabenangemessenheit	80%
2	Verfügbarkeit	90%
3	Selbstbeschreibungsfähigkeit	100%
4	Fehlertoleranz	100%
5	Erwartungskonformität	100%

Tabelle 5.1: Anforderungsabgleich

6

Zusammenfassung

Durch den vorgegeben Server, der die Struktur der Formulare definiert, ist es möglich, generische Formulare auf mobilen Endgeräten darzustellen. Formulare auf dem Server können beliebig ausgetauscht oder erweitert werden, so dass eine zentrale Verwaltung von Formularen gegeben ist. Somit kann sich das mobile Endgerät auf das Anzeigen von Formularen unabhängig deren Inhalten konzentrieren.

Die mobile $Crowd\ Sensing$ Anwendung erhält die Daten mithilfe von Formularen. Jedes Formular ist gleich aufgebaut, da der Inhalt zur Compile-Zeit nicht bekannt ist. Mittels $Auto\ Layout$ können Regeln für die Positionierung von Formularelementen definiert werden.

Sobald der Benutzer ein Formular abgeschickt hat, werden die Einträge validiert. Jedes Formularfeld wird untersucht, ob es pflichtgemäß ausgefüllt ist. Dazu kann mithilfe eines Pattern die korrekte Eingabe sichergestellt werden.

Sollte beim Validieren Fehler in der Eingabe gefunden werden, können diese direkt vom Benutzer korrigiert werden. Selbst wenn eine Fehlermeldung vom Server geschickt wird, kann der Benutzer darauf reagieren.

Damit der Benutzer überhaupt Formulare ausfüllen kann, kommuniziert das System mit dem Server über eine REST-API. Alle Informationen, die zwischen dem System und dem Server ausgetauscht werden, sind in JSON geschrieben.

Die Anwendung hat ebenfalls ein Bonusprogramm und eine Statistik integriert. Diese sollen zu $Crowd\ Sensing$ beitragen, indem der Benutzer motiviert ist, Formulare auszufüllen. Die Statistik dient der Übersicht über eingetragene Werte in verschiedenen Formularen. Bezogen auf einen möglichen Anwendungsbereich in der Medizin können übersichtlich dargestellte Vitalwerte sowohl dem Benutzer selbst, als auch dem behandelnden Arzt hilfreich sein.

Die Gestaltung der Anwendung orientiert sich an den $iOS\ Human\ Interface\ Guidelines$. Das einheitliches Erscheinungsbild der Anwendung soll die Funktion $Crowd\ Sensing$ unterstützen.

6.1 Ausblick

Die Anwendung deckt die Basisfunktionen ab, um Daten zu sammeln. Deshalb bietet der Ausblick mögliche Optimierungen an, die die Funktion der Anwendung unterstützen können.

Automatische Eingabe: Damit der Benutzer auch in Zukunft seine Daten über die Formulare aufzeichnet, könnte die Eingabe der Werte an sich vereinfacht werden. Bisher muss der Benutzer die Formulare einzeln manuell ausfüllen. Vor allem im medizinischen Bereich könnten Vitalwerte von einem Smartphoneadapter, der z.B. den Blutzucker misst, automatisch in die Anwendung übertragen werden [27].

Schutz der Privatsphäre: Sollen die angesammelten Daten nicht nur für den Benutzer einsehbar sein, sondern auch für den behandelten Arzt, müsste die Anwendung die

Privatsphäre schützen. Das wäre durch eine Option möglich, die dem Benutzer erlaubt gewisse Werte freizugeben.

Textuelle Beschreibung von Pattern: Bei Formularfeldern mit einer spezifische Eingabe, die durch ein Pattern bestimmt ist, würde eine textuelle Beschreibung des Patterns dem Benutzer beim Ausfüllen des Formulars helfen. Wenn ein Benutzer bisher ein Formularfeld falsch eingegeben hat, wird dies lediglich als falsch gekennzeichnet und der Fehler nicht an sich beschrieben. Die textuelle Beschreibung eines Formularfeldes müsste im JSON-Schema serverseitig definiert sein.

Statistik: Für die Statistik ist es hilfreich, wenn der Server vorgibt, welche Werte für eine Statistik sinnvoll sind. So muss nicht jeder Zahlenwert, der einmal eingetragen wurde, sofort in die Statistik aufgenommen werden. Als Alternative könnte der Benutzer bei einem potentiellen Formularfeld selbst einstellen, ob dieser lokal gespeichert werden soll.

Serverkommunikation: Ein großes Problem des Systems ist der Zwang, dass Server und Client nur über Internet miteinander kommunizieren können. Für den Benutzer ist es nur dann möglich, Formulare anzufordern und ausgefüllt zurück zu schicken, wenn sein mobiles Endgerät online ist. Damit eine kurzzeitige Internetunterbrechung nicht zu einem Problem wird, könnten Formulare lokal auf dem Smartphone gespeichert werden. Erst wenn eine sichere Internetverbindung besteht, sollen die ausgefüllten Formulare an den Server übertragen werden. Serverseitig könnten aber Fehler im Formular oder bei der Übertragung auftreten, die nicht sofort behandelt werden könnten.

Literaturverzeichnis

[1] Statista: Anzahl der Smartphone-Nutzer in Deutschland. http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/ (2015) Stand: 29.9.2015.

[2] Apple, I.: iPhone 6s. http://www.apple.com/de/iphone-6s/specs/ (2015) Stand: 29.9.2015.

[3] Barth, D.: The bright side of sitting in traffic: Crowdsourcing road congestion data. https://googleblog.blogspot.ca/2009/08/bright-side-of-sitting-in-traffic.html (2009)

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

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

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

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

[8] Enste, P., Merkel, S., Romanowski, S.: Gesundheit aus der Hosentasche? Chancen und Grenzen gesundheitsbezogener Apps. https://www.econstor.eu/dspace/bitstream/10419/57223/1/690129297.pdf (2010) Stand 28.7.2015.

[9] Posth, D.: Die Interoperabilität ist noch nicht so weit entwickelt. http://www.medica.de/cipp/md_medica/custom/pub/content,oid,36935/lang,1/ticket,g_u_e_s_t/~/Die_Interoperabilität_ist_noch_nicht_so_weit_entwickelt.html (2012) Stand: 28.7.2015.

[10] Ma, H., Zhao, D., Yuan, P.: Opportunities in mobile crowd sensing. Communications Magazine, IEEE (2014)

[11] Schindler, A.: Konzeption und Entwicklung einer modularen, ereignisgesteuerten Server-Client-Architektur zur Crowd-basierten Datenerfassung mit mobilen Endgeräten. Master's thesis, Universität Ulm (2015)

[12] Masse, M.: REST API Design Rulebook. Oreilly and Associate Series. O'Reilly Media (2011)

[13] Ecma, I.: The JSON Data Interchange Format. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (2013) Stand: 17.9.2015.

[14] Apple, I.: Swift. https://developer.apple.com/swift/ (2015) Stand: 28.7.2015.

[15] Apple, I.: The Swift Programming Language. Apple Inc. (2014)

[16] Radatz, J., Geraci, A., Katki, F.: Ieee standard glossary of software engineering terminology. IEEE Std (1990)

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

[18] Blaser, W.: Mies van der Rohe: less is more. Waser (1986)

[19] Apple, I.: iOS Human Interface Guidelines. `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/` (2015)

[20] Sillmann, T.: Apps für iOS 8 professionell entwickeln: Techniken, Methoden & Strukturen von Grund auf verstehen. Carl Hanser Verlag GmbH & Company KG (2014)

[21] Bargas-Avila, J., Brenzikofer, O., Roth, S., Tuch, A., Orsini, S., Opwis, K.: Simple but Crucial User Interfaces in the World Wide Web: Introducing 20 Guidelines for Usable Web Form Design (2010)

[22] Deterding, S., Dixon, D., Khaled, R., Nacke, L.: From game design elements to gamefulness: defining gamification. In: Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, ACM (2011)

[23] Koch, M.: Gamification – Steigerung der Nutzungsmotivation durch Spielkonzepte. Universität der Bundeswehr München. (2012) Stand: 9.11.2015.

[24] Tang, P.: Swiftyjson. `https://github.com/SwiftyJSON/SwiftyJSON.git` (2014) Stand: 17.9.2015.

[25] Gindi, D.C.: ios-charts. `https://github.com/danielgindi/ios-charts.git` (2015) Stand: 17.9.2015.

[26] Ebert, C.: Systematisches Requirements Engineering: Anforderungen ermitteln, spezifizieren, analysieren und verwalten. dpunkt (2012)

[27] Schobel, J., Schickler, M., Pryss, R., Nienhaus, H., Reichert, M.: Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned.

In: 9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps. (2013) 509–518

Abbildungsverzeichnis

Name: Sabrina Friedl Matrikelnummer: 787292

Erklärung

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

Ulm, den .....

Sabrina Friedl